TP 9 – Micro framemwork de spécification de code

Dans ce TP, nous allons faire de la génération de code Python à partir de code Python et en Python. Le but du TP sera de générer du code Python à partir de classes Pythons qui serviront de "spécification", c'est à dire que ces classes vont avoir peu d'information, mais ces informations vont nous servir à dériver du code. Le TP se découpe en deux parties, une première partie d'introspection et collecte d'information, puis une génération de texte qui sera du code Python. S'il vous reste du temps, vous pourrez effectuer une dernière partie sur la compilation en mémoire du code.

Exercice 1. Une spécification ?

Nous allons considérer directement des classes Python comme étant la spécification du code que nous voulons générer. Voilà un exemple de code que nous aimerions pouvoir écrire comme spécification :

```
@specification                          |       @specification
class Engine(object):                   |       class Car(Vehicule):
  pass                                  |         pass
                                        |
@specification                          |
class Vehicule(object):                 |
  number_of_wheels = Attribute(int)     |
  engine = Attribute(Engine)            |
```

Et de cette spécification, nous aimerions obtenir le code généré suivant :

```
class Engine(object):
  pass

class Vehicule(object):
  def __init__(self, number_of_wheels=None, engine=None):
    self.number_of_wheels = number_of_wheels
    self.engine = engine

  @property
  def number_of_wheels(self):
    return self._number_of_wheels

  @number_of_wheels.setter
  def number_of_wheels(self, value):
    assert value is None or isinstance(value, int)
    self._number_of_wheels = value

  @property
  def engine(self):
    return self._engine

  @engine.setter
  def engine(self, value):
    assert value is None or isinstance(value, Engine)
    self._engine = value

class Car(Vehicule):
  pass
```

De cette spécification, on peut observer 2 choses :

— il y a un décorateur particulier pour préciser qu'une classe est une spécification,

— il y a une classe spécial `Property` qui est utilisé pour indiquer qu'une variable de classe représente un attribut et son type.

Un décorateur sur une classe en Python s'implémente en réalisant une fonction qui prend en paramètre une classe et qui retourne la classe. Ce décorateur va faire très peu de chose, il va juste "marquer" la classe comme étant une spécification en rajoutant un attribut `specification` avec la valeur `True` sur la classe qu'il décore. Ce marqueur sera utilisé plus tard pour filtrer les classes à introspecter de celles à ne pas analyser.

1. Créer un fichier `spec.py` qui sera utilisé pour écrire l'implémentation de notre mini-framework de spécification (c'est-à-dire, le décorateur et la classe `Attribute`).

2. Écrire le décorateur `specification` (c'est-à-dire, écrire une fonction `specification` qui prend un paramètre), qui rajoute l'attribut à la classe passée en paramètre et qui retourne la classe modifiée.

3. Écrire une classe `Attribute` qui va représenter un attribut de la spécification. Cette classe ne possède qu'une référence vers le type qu'elle est censé représenter.

4. Créer un autre fichier `cars.py` qui contiendra le code de la spécification exemple donné plus haut dans ce sujet.

5. Écrire ensuite la spécification d'exemple (celui du vehicule) dans `cars.py` en utilisant votre micro-framework `spec.py`.

Exercice 2. Introspection de classes et de modules

Maintenant que nous sommes capable d'exprimer que des classes sont des spécifications et de décrire des attributs sur ces classes, nous allons proposer les fonctions d'analyse de la spécification pour produire le code associé.

1. Écrire une fonction `gen_class_header` qui prend une classe spécification en paramètre et qui retourne l'entête de la classe uniquement (pour l'instant) sous forme de chaîne de caractère. Cette fonction doit gérer l'héritage en générant correctement le code relatif à l'héritage (*Indication*, l'attribut `__bases__` permet d'obtenir la liste des classes dont hérite une classe et pour vous aider dans la génération du texte, regardez l'utilisation de `join` qui est une méthode sur les chaines de caractères).

2. Écrire une fonction `gen_class_body` qui prend une classe spécification en paramètre et qui retourne une liste où chaque élément et l'équivalent du texte de chaque fonctions (faites bien attention à l'indentation, le code de chaque méthode devra être placé ensuite dans une classe).

3. Écrire une fonction `gen_class` qui prend une classe spécification et qui retourne une grande chaine de caractère représentant la classe générée à partir de la classe de spécification.

Maintenant que nous sommes capable de générer le code pour une classe, il ne nous reste plus qu'à être capable de récupérer toutes les classes d'un module.

4. Écrire une fonction `collect_specifications` qui prend un module en paramètre et qui retourne la liste des classes qui seront analyser plus tard (*Indication*, l'attribut `__dict__` d'un module permet de récupérer un dictionnaire des éléments qu'il contient, le module `inspect` donne accés à des fonctions qui permettent de tester si un élément qu'il contient et la fonction `hasattr(elem, 'attr')` permet de vérifier que l'attribut `attr` existe dans `elem`).

Exercice 3. Génération de code

Nous avons tout ce qu'il nous faut pour pouvoir générer du code à partir d'une spécification. Nous sommes capable d'analyser des classes de spécification et de retourner le code correspondant. Nous sommes capable de récupérer toutes les classes spécification depuis un module. Il ne reste plus qu'à écrire le tout dans un fichier.

1. Écrire une fonction `gen_module` qui prend un module en paramètres, qui collecte la totalité des classes de spécification et qui écrit dans un fichier le code des classes. Le nom du fichier sera nommé en fonction du nom du module. Par exemple pour `cars.py` qui correspond au fichier de spécification, le fichier généré sera nommé : `cars_gen.py` (*Indication*, on peut accéder au nom d'un module par l'attribut `__name__`).