class_dog_sol

December 7, 2019

1 Exercise 1

Partie A Définissez une nouvelle classe en utilisant les informations données ci-dessous.

Classe: - Dog

Attributs: - name
- tricks

Méthodes: - add_trick - bark

Instances: - Lola - Boby

States: - (name: Lola, tricks: 'rollover', 'playdead') - (name: Boby, tricks: [])

[34]:
```
# Partie A
# Votre code ici de a à z
class Dog:
    """
    This is a dog class
    """
    def __init__(self, name): # constructeur
        self.name = name
        self.tricks = [] # création nouvelle liste vide pour chaque chien

    def add_trick(self, trick): # méthode
        self.tricks.append(trick)

    def bark(self, intensity=3): # méthode avec valeur par défaut du paramètre
        if intensity==1:
            print('whouaf')
        elif intensity==2:
            print('Whouaf!')
        else:
            print('WHOUAF!')

# Création nouveaux objets
lola = Dog('Lola')
boby = Dog('Boby')
```

```
# Utilisation méthodes de la classe
lola.add_trick('roll over')
lola.add_trick('play dead')

boby.bark(2)
```

```
Whouaf!
```

Partie B Modifiez votre code afin que la `race` et l' `age` du chien deviennent des attributs fixés lors de l'instanciation de la classe. Ajoutez également un attribut `mood` qui lors de l'instanciation du chien a une valeur de 5. Ainsi, vous devriez avoir les modifications ci-dessous.

Attributs: - name (string)
- race (string) - age (integer) - mood (integer) - tricks (list)

[35]:
```
# Partie B
# Pensez à faire un copier-coller de votre partie précédente
# Votre code ici
class Dog:
    """
    This is a dog class
    """
    def __init__(self, name, race, age): # constructeur
        self.name = name
        self.race = race
        self.age = age
        self.mood = 5
        self.tricks = [] # création nouvelle liste vide pour chaque chien

    def add_trick(self, trick): # méthode
        self.tricks.append(trick)

    def bark(self, intensity=3): # méthode avec valeur par défaut du paramètre
        if intensity==1:
            print('whouaf')
        elif intensity==2:
            print('Whouaf!')
        else:
            print('WHOUAF!')
```

Partie C Créez une méthode `eat` qui incrémente l'humeur de 3 et une méthode `leash` qui diminue l'humeur de 1. L'humeur du chien doit être un entier entre 0 (grincheux) et 10 (heureux).

Définissez également une méthode d'instance `description` qui renvoie le nom et l'âge du chien.

Ex: Lola is 3 yo.

[36]:
```python
# Partie C
# Pensez à faire un copier-coller de votre partie précédente
# Votre code ici
class Dog:
    """
    This is a dog class
    """
    def __init__(self, name, race, age): # constructeur
        self.name = name
        self.race = race
        self.age = age
        self.mood = 5
        self.tricks = [] # création nouvelle liste vide pour chaque chien

    def add_trick(self, trick): # méthode
        self.tricks.append(trick)

    def bark(self, intensity=3): # méthode avec valeur par défaut du paramètre
        if intensity==1:
            print('whouaf')
        elif intensity==2:
            print('Whouaf!')
        else:
            print('WHOUAF!')

    def eat(self):
        self.mood += 3
        self.mood = min(self.mood, 10) # mood not more than 10

    def leash (self):
        self.mood -= 1
        self.mood = max(self.mood, 0) # mood not less than 0

    def description(self):
        return "{0} is {1} yo.".format(self.name,self.age)
```

Partie D Créez un nouvel objet de la classe `Dog` appelé Tobi et stocké dans une variable `tobi` en définissant les valeurs des attributs `race` et `age`. Puis, utilisez la fonction `leash` et vérifiez sa `mood`.

[37]:
```python
# Votre code ici
tobi = Dog("Tobi", "Australian shepherd", 5)
tobi.leash()
tobi.mood
```

[37]: 4

Faites aboyer `tobi` avec `bark`.

[38]:
```
# Votre code ici
tobi.bark(3)
```

```
WHOUAF!
```

Utilisez la méthode `description` pour `tobi`.

[39]:
```
# Votre code ici
tobi.description()
```

[39]:
```
'Tobi is 5 yo.'
```

Utilisez `.__dict__` pour voir tous les détails de `tobi`, `lola`, `boby`.

[40]:
```
# Votre code ici
print(tobi.__dict__, lola.__dict__, boby.__dict__)
```

```
{'name': 'Tobi', 'race': 'Australian shepherd', 'age': 5, 'mood': 4, 'tricks':
[]} {'name': 'Lola', 'tricks': ['roll over', 'play dead']} {'name': 'Boby',
'tricks': []}
```

Veuillez noter que ci-dessus les attributs `race` et `age` ne sont pas présents pour `lola` et `boby`.

Partie E En utilisant la classe `Dog`, instanciez trois nouveaux chiens, chacun avec un âge différent.

[41]:
```
# Votre code ici
bloom = Dog("Bloom","Bulldog", 8)
jazz = Dog("Jazz", "Cocker", 3)
luigi = Dog("Luigi","Chow Chow", 1)
```

Ecrivez une fonction appelée `get_oldest_age()`, qui renvoie le plus grand âge en prennant un nombre quelconque d'âges (*args).

[44]:
```
# Votre code ici
def get_oldest_age(*args):
    return max(args)
```

Utilisez vos trois instances de chiens précédemment créées pour indiquer l'âge du chien le plus âgé comme suit: The oldest dog is 8 yo.

[45]:
```
# Votre code ici
print("The oldest dog is {} yo.".format(get_oldest_age(bloom.age, jazz.age,␣
 ↪luigi.age)))
```

```
The oldest dog is 8 yo.
```

class_student_sol

December 7, 2019

1 Exercise 2

Définissez une classe `Student` qui gère les informations suivantes pour un étudiant: `name`, `year` et `grades`.

- Définissez un constructeur qui initialise un nouveau `Student` avec `name` et `year`.
- Écrivez deux méthodes disctinctes (`get_name`, `get_year`) pour obtenir le `name` et la `year` d'un `Student`.
- Écrivez une méthode `add_grade` pour ajouter une nouvelle note.
- Implémentez une méthode `final_grade` qui retourne la note moyenne.
- Ajoutez une méthode `evaluate_year` qui fait passer l'étudiant (c'est-à-dire que l'étudiant passe à l'année suivante et commence avec de nouvelles notes) si la note moyenne est supérieure à 4.
- Ajoutez une méthode `__str__`qui retourne une chaîne de caractères contenant le `name`, la `year` et la `final_grade` si possible.

[15]:
```python
# Votre code ici
class Student:
    def __init__(self, name:str, year:int): #constructeur
        self.name = name
        self.year = year
        self.grades = []

    def get_name(self):
        return self.name

    def get_year(self):
        return self.year

    def add_grade(self, grade:int):
        self.grades.append(grade)

    def final_grade(self):
        if len (self.grades)>0:
            return sum(self.grades)/len(self.grades)
        else:
```

```
            return 0

    def evaluate_year(self):
        if self.final_grade() >= 4:
            self.year += 1
            self.grades = []

    def __str__(self):
        return self.name + " is in year " + str(self.year) + ("  with average␣
↪grade " + str(self.final_grade()) if len(self.grades) > 0 else " ")
```

Tester votre code en faisant les étapes suivantes:

- Créez 3 nouvelles instances de la classe `Student`.
- Utilisez `add_grade` sur chacune d'entre elles au moins 2 fois.
- Utilisez `evaluate_year` sur chacune d'entre elles.
- Utilisez `__str__` sur chacune d'entre elles.
- Utilisez `__dict__` sur chacune d'entre elles.

[16]:
```
# Instanciation
# Votre code ici
Tanguy = Student("Tanguy", 1)
Tarik = Student("Tarik", 1)
Xavier = Student("Xavier", 1)
```

[17]:
```
# add_grade
# Votre code ici
Tanguy.add_grade(4)
Tarik.add_grade(4)
Xavier.add_grade(4)

Tanguy.add_grade(6)
Tarik.add_grade(5)
Xavier.add_grade(4)
```

[18]:
```
# evaluate_year
# Votre code ici
Tanguy.evaluate_year()
Tarik.evaluate_year()
Xavier.evaluate_year()
```

[19]:
```
# __str__
# Votre code ici
print(Tanguy.__str__(), Tarik.__str__(), Xavier.__str__())
```

```
Tanguy is in year 2  Tarik is in year 2  Xavier is in year 2
```

[23]:
```python
# __dict__
# Votre code ici
print(Tanguy.__dict__, Tarik.__dict__, Xavier.__dict__)
```

```
{'name': 'Tanguy', 'year': 2, 'grades': []} {'name': 'Tarik', 'year': 2,
'grades': []} {'name': 'Xavier', 'year': 2, 'grades': []}
```

Graph avec des classes - solution

December 7, 2019

1 Créer des graphes avec des classes

Les semaines précédentes, vous avez vu comment implémenter des graphes avec des dictionnaires et des listes. Cette fois nous allons travailler avec des classes pour représenter des graphes orientés.

1.1 Exercice 1

Pour implémenter notre graphes nous allons utiliser deux classes différentes, une pour definir les graphes et une pour definir les arêtes. Les sommets seront des chaines de caractères.

- La classe `Edge` pour definir les arêtes. Chaque instance aura comme attribut un sommet de départ `from_vertex`, un sommet d'arrivée `to_vertex` et un poids `weight`. La classe aura comme unique methode `__repr__`pour retourner l'objet sous la forme d'un dictionaire.
- La classe `Graph`pour definir un graphe. Chaque instance aura comme attribut un set `edges` contenant les arêtes (qui seront des objets) et un set `vertices` contenant les sommets. Astuce: pour initialiser un set vide utilisez `set()`

La classe `Graph` aura comme méthodes: + `add_vertex` pour créer un sommet + `edge_exist` pour vérifier si une arête existe, elle retourne `weight` si oui, `False` sinon. + `update_weight` pour mettre le poids à jour d'une connexion existante, et appeler la méthode `new_edge` sinon. + `new_edge` pour créer une instance de `Edge`et l'ajouter au set `edges` si la connexion n'existe pas déjà, si elle existe avec un autre poids mettre à jour le poids, si elle existe de façon identique alors retournez le dans la console avec `print(...)`. Astuce: Ici il peut être judicieux de créer une méthode privée `__generate_edge` et de l'appeler puisque avant de créer une connexion il faut s'assurer que les sommets existes et les créer sinon. + `del_edge` pour supprimer un `edge` + `__repr__` pour retourner l'objet sous forme de dictionnaire.

NB: Dans cette structure il est possible de definir une classe pour sommets également pour avoir plus d'information sur chaque sommet, nous verrons ça la semaine prochaine.

[1]:
```
class Edge:
    # votre code ici
    def __init__(self, from_vertex, to_vertex, weight):
        self.from_vertex = from_vertex
        self.to_vertex = to_vertex
        self.weight = weight

    def __repr__(self):
        return str({"from_vertex": self.from_vertex, "to_vertex": self.
↪to_vertex, "weight": self.weight})
```

```
class Graph:
    # votre code ici
    def __init__(self):
        self.edges = set()
        self.vertices = set()

    def add_vertex(self, name):
        self.vertices.add(name)

    def edge_exist(self, from_vertex, to_vertex):
        for edge in self.edges:
            if edge.from_vertex is from_vertex and edge.to_vertex is to_vertex:
                return edge.weight
        return False

    def __generate_edge(self, from_vertex, to_vertex, weight):
        if from_vertex in self.vertices and to_vertex in self.vertices:
            new_edge = Edge(from_vertex, to_vertex, weight)
            self.edges.add(new_edge)
            return
        else:
            if to_vertex not in self.vertices:
                self.add_vertex(to_vertex)
            if from_vertex not in self.vertices:
                self.add_vertex(from_vertex)
            new_edge = Edge(from_vertex, to_vertex, weight)
            self.edges.add(new_edge)

    def update_weight(self, from_vertex, to_vertex, new_weight):
        for edges in self.edges:
            if edges.from_vertex is from_vertex and edges.to_vertex is␣
↪to_vertex:
                edges.weight = new_weight
                print("weight has been updated")
                return
        self.new_edge(from_vertex, to_vertex, new_weight)
        print(
            "The vertex between {} and {} does not exist and will be created".
↪format(
                from_vertex, to_vertex))

    def new_edge(self, from_vertex, to_vertex, weight):
        test_existence = self.edge_exist(from_vertex, to_vertex)
        if test_existence:
            if test_existence is weight:
```

```
                print("Edge between {} and {} with the same weight already␣
↪exist".format(from_vertex, to_vertex))
            else:
                print("Edge between {} and {} with the same weight already␣
↪exist but with a"
                      "different weight and will be overwritten".
↪format(from_vertex, to_vertex))
                self.update_weight(from_vertex, to_vertex, weight)
        else:
            self.__generate_edge(from_vertex, to_vertex, weight)

    def del_edge(self, from_vertex, to_vertex):
        for edge in self.edges:
            if edge.from_vertex is from_vertex and edge.to_vertex is to_vertex:
                self.edges.remove(edge)
                print("Edge between {} and {} has been deleted".
↪format(from_vertex, to_vertex))
                return

    def __repr__(self):
        return str({"edges": self.edges, "Vertices": self.vertices})
```

[2]:
```
graph = Graph()
graph.add_vertex("Lausanne")
graph.add_vertex("Geneve")
graph.new_edge("Geneve", "Lausanne", 35)
graph.new_edge("Lausanne", "Berne", 100)
graph.new_edge("Lausanne", "Berne", 110)
graph.update_weight("Geneve", "Berne", 120)
print(graph.vertices, graph.edges)
graph.del_edge("Geneve", "Berne")

print(graph.vertices, graph.edges)
print(graph)
```

```
Edge between Lausanne and Berne with the same weight already exist but with
adifferent weight and will be overwritten
weight has been updated
The vertex between Geneve and Berne does not exist and will be created
{'Berne', 'Lausanne', 'Geneve'} {{'from_vertex': 'Lausanne', 'to_vertex':
'Berne', 'weight': 110}, {'from_vertex': 'Geneve', 'to_vertex': 'Lausanne',
'weight': 35}, {'from_vertex': 'Geneve', 'to_vertex': 'Berne', 'weight': 120}}
Edge between Geneve and Berne has been deleted
{'Berne', 'Lausanne', 'Geneve'} {{'from_vertex': 'Lausanne', 'to_vertex':
'Berne', 'weight': 110}, {'from_vertex': 'Geneve', 'to_vertex': 'Lausanne',
'weight': 35}}
```

```
{'edges': {{'from_vertex': 'Lausanne', 'to_vertex': 'Berne', 'weight': 110},
{'from_vertex': 'Geneve', 'to_vertex': 'Lausanne', 'weight': 35}}, 'Vertices':
{'Berne', 'Lausanne', 'Geneve'}}
```

[ ]: