SOFTWARE 1 PRACTICAL

Classes

Week 8 – Practical 7

You may remember the exercise you have done in week 4 (additional exercise 5) regarding vectors. For your convenience I have rewritten the definition here.

A vector of dimension n can be represented by a list of n elements in Python. We would like to create a class Vector with two basic operations on vectors:

Scalar product: $\lambda \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \lambda \cdot a \\ \lambda \cdot b \\ \lambda \cdot c \end{bmatrix}$

Addition: $\begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} a+d \\ b+e \\ c+f \end{bmatrix}$

Implementing a Vector Class

Exercise 1: *Class' constructor*

First of all, create a module called vector.py, then define the class Vector. The next step is to define what will be the internal representation of a vector and then write the constructor `__init__`. The design decision is to store the element of the vector $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ in a list `[a,b,c]`. The constructor will take only one parameter, a list of float. The instance attribute `_vector`. should have a **copy** of the list passed in the parameters.

```
def __init__(self, data = None):
    """ some doc-string """
    Pass
```

Exercise 2:

Another very useful method to write is `__str__`. This will enable us to print the content of the instance using the print function. For the purpose of this exercise we have decided to represent the vector $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ with the string `'<a, b, c>'` to differentiate it from a list. Implement __str__.

```
    def __str__(self):
        pass
```

Now let see how we can instantiate (create) some vectors.

```
>>> my_vector = Vector([1, 2, 3])
>>> print(my_vector)
 <1, 2, 3>
>>> empty_vector = Vector()
>>> print(empty_vector)
 <>
```

Adding behaviours to the class Vector

We now need to think about the definition of a vector, what operation could be done? We know that we can add two vectors of same dimension, we can do the scalar product with a number (called a scalar), what else?

- Get the dimension of a vector (e.g. the number of elements in the vector)
- Get the value at a defined position in the vector
- Set a value at a defined position in the vector
- Check if they are equals, not equals
- Do the scalar product
- Do an addition between two vectors of equal size.

Exercise 3:

Implement the **method** `dim()` that returns the dimension of a vector (i.e. the number of elements in a vector)

Exercise 4:

Implement the following accessor and mutator:

- `get(index)` which returns the value of the element at position `index` in the vector
- `set(index, value)` which set the element at position `index` to the new value `value`. The method does not return any value.

Let's implement the scalar product method `scalar_product(scalar)` as an example. The method needs only one parameter, the scalar. In addition, the method should return a **new** `Vector` containing the result of the operation, but MUST NOT modify the calling instance, e.g. `my_vector.scalar_product(3)` must not modify the instance `my_vector`.

```
    def scalar_product(self, scalar):
        ''' add some doc-string'''
        pass
```

Exercise 5:

Implement the method `add(other_vector)` that emulate the vector addition operator. The method should return a new vector.

- You will have to check that `other_vector` is a Vector instance, and raise a `TypeError` if it is not the case.
- You must check that both vector have the same dimension, raise a `ValueError` if it is not the case.
- You must return a new `Vector` instance like we have done in `scalar_product(scalar)`.

Once implemented we should be able to do the following:

```
>>> vector1 = Vector([1, 2, 3])
>>> vector2 = Vector([0, 1, 3])
>>> added = vector1.add(vector2)
>>> print(added)
 <1, 3, 6>
```

Exercise 6:

In Programming, being able to compare objects is important, in particular determining if two objects are equal or not. Let's try a comparison of two vectors:

```
>>> vector1 = Vector([1, 2, 3])
>>> vector2 = Vector([1, 2, 3])
>>> vector1 == vector2
 False
>>> vector1 != vector2
 True
>>> vector3 = vector1
>>> vector3 == vector1
 True
```

As you can see, in the current state of implementation of our class Vector does not produce the expected result when comparing two vectors. In the example above the == operator return `True` if the two vectors are physically stored at the same memory address, it does not compare the content of the two vectors.

Therefore, you need to implement a method `equals(other_vector)` that returns `True` if the vectors are equals (i.e. have the same value at the same position), `False` otherwise.

Hint: to check if an object is of a certain type you can use `isinstance(var, Type)`. For example `isinstance(other_vector, Vector)`.

Once implemented we should have the following results

```
>>> vector1 = Vector([1, 2, 3])
>>> vector2 = Vector([1, 2, 3])
>>> vector1.equals(vector2)
 True
>>> vector3 = Vector([0, 2, 0])
>>> vector3.equals(vector1)
 False
>>> vector1 == vector2
 False
```