Algorithms & Data Structures

Week 4 -classes

# Introduction

A vector is an object that has both a magnitude (or length) and a direction. Think of it like an arrow pointing in a specific direction, with a certain length. In Physics they can represent forces. Imagine you’re pushing a box; the force you apply can be depicted as a vector showing the direction and strength of your push. In Computer Graphics every pixel’s position can be represented as a vector.

For example, the vector represents a position on a flat map and is said to have a dimension of 2, the vector could represent a position in space and his said to have a dimension of 3. The dimension of the vector can be seen as the number of values in it.

The aim of this week’s exercises is to build a Python class representing a vector.

A vector of dimension can be represented by a list of n values in Python. We would like to create a class Vector with two basic operations on vectors, scalar product and vector addition. Below is an example of these two operations on a vector of dimension 3:

Scalar product:

Addition:

For example,

Scalar product:

Addition:

# Implementing a Vector Class

### **Exercise 1:** *Class’ constructor*

First, 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 values of a dimensional vector into a python list of values. For example, the values of the dimension 3 vector is stored in a list [2.0, 4.0, 1.1], whereas the dimension 2 vector is stored in a list [5.1, 2.3]. The constructor will take only one parameter, a list of floats. The instance attribute \_vector should be assigned a **copy** of the list passed in the parameters.

def **\_\_init\_\_**(*self*, data):

“”” some doc-string “””

*Pass*

## Exercise 2:

Another very useful method to write is \_\_str\_\_. This dunder method 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 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([1.0, 2.1, 3.2])

>>> print(my\_vector)

<1.0, 2.1, 3.2>

>>> other\_vector = Vector([1.1, 2.2, 3.3, 1.0])

>>> print(other\_vector)

<1.1, 2.2, 3.3, 1.0>

>>>

# 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 values 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 values in a vector)

## Exercise 4:

Implement the following accessor and mutator:

* get(index) which returns the value at position index in the vector,
* set(index, value) which set the value at position index to the new float value. The method does not return value.

## Exercise 5:

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 6:

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 return None if it is not the case.
* You must check that both vectors have the same dimension, return None 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([1.0, 2.0, 3.0])

>>> vector2 = Vector([0.0, 1.0, 3.0])

>>> added = vector1.add(vector2)

>>> print(added)

<1.0, 3.0, 6.0>

## Exercise 7:

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

>>> vector1 = Vector([1.0, 2.0, 3.0])

>>> vector2 = Vector([1.0, 2.0, 3.0])

>>> 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. As we are dealing with float values, you may want to explore what the function isclose() form the math module does.

**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([1.0, 2.0, 3.0])

>>> vector2 = Vector([1.0, 2.0, 3.0])

>>> vector1.equals(vector2)

True

>>> vector3 = Vector([0.0, 2.0, 0.0])

>>> vector3.equals(vector1)

False

>>> vector1 == vector2

False

# Operator overloading the class Vector

### Exercise 8:

We are now able to compare two vectors, however it would be nice if we could use the operator == instead of .equals(…). Fortunately Python allows to overload operators such as +, \*, == and !=. To overload the operator ==, we must ***override*** the method \_\_eq\_\_. (for the operator !=, override the method \_\_ne\_\_).

Implement the two methods:

def \_\_eq\_\_(self, other\_vector):

<body>

def \_\_ne\_\_(self, other\_vector):

<body>

### Exercise 9:

Instead of using the method add(…) and scalar\_product(…) we would like to overload the operators + and \*.

1. The vector addition operator is commutative, i.e. v1+v2 == v2+v1 and we can override the method \_\_add\_\_ to overload the + operator.
2. When considering the multiplication, it is a little bit more complicated, 3 \* v1 is allowed, but v1 \* 3 is not. Investigate the methods \_\_mul\_\_ and \_\_rmul\_\_.
3. One other programming shortcut we could find useful is v1 += v2 for v1 = v1+v2. It can be implemented by overloading \_\_iadd\_\_.
4. Similarly, the shortcut v1 \*= 3 for v1 = 3 \* v1 can be implemented by overloading \_\_imul\_\_. Implement all these operators overloading dunder methods.