sOFTWARE 1 PRACTICAL

## Operator Overloading

Week 9 – Practical 15

Before attempting this practical, you should complete practical 14. The aim of this practical is to improve the functionality of the class Vector we implemented earlier this week. You will do so by refactoring your code and overloading existing operators. All tests from previous practical should still work. You should try to ensure backward compatibility.

### Exercise 1:

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 to ***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 2:

Instead of using the method add(…) and scalar\_product(…) we would like to overload the operators + and \*. The vector addition operator is commutative, i.e. v1+v2 == v2+v1 and we can override the method \_\_add\_\_ to overload the + operator. 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\_\_. One other programming shortcut we could find useful is v1 += v2 for v1 = v1+v2. It can be implemented by overloading \_\_iadd\_\_. Similarly, the shortcut v1 \*= 3 for v1 = 3 \* v1 can be implemented by overloading \_\_imul\_\_. Implement all these operators overloading.

### Exercise 3:

There is a way to add the following functionality to our class Vector.

>>> vector1 = Vector(1, 2, **3**, 5, 6, 1)

>>> vector1[2] += 5

>>> print(vector1)

<1.0, 2.0, **8.0**, 5.0, 6.0, 1.0>

Search the web to find out how it can be done (I will NOT provide the answer). For method with undefined number of parameters search \*args. **Note**, the tests provided in previous practical should still be working, including the test for the \_\_init\_\_ method.

### Problem:

During practical 13, you implemented the **Set** ADT using binary search tree (BST). In addition, we used the nested list representation for a BST. Unfortunately, the internal representation of a set was visible to any users of the module. There was no real data encapsulation to makes our module resilient or usable. You should refactor this module using your newly found knowledge about classes, data encapsulation and operator overloading. You have also seen linked representation of general binary trees instead of the nested list, you should apply the same linked representation to BST.

Below is the list of functionalities in the API of the Python built-in **set** data type. Search the documentation and use of these methods and then implement each one of them (note that will take several hours of programming to do them all). In the module **treeset.py**, implement the class **TreeSet**, a binary search tree implementation of a set. You must use the **linked tree** representation.

1. **add**(...): Add an element to a set. This has no effect if the element is already present.
2. **remove**(...): Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.

At this stage you should implement the method \_\_contains\_\_(self, value) which returns True if value is in the set, False otherwise. Once implemented we can use the statements:

>>> s = {1,2,3}

>>> 1 **in** s

True

>>> 4 **in** s

False

1. **difference**(...): Return the difference of two or more sets as a new set. (i.e. all elements that are in this set but not the others.). In addition, implement the method \_\_sub\_\_ to be able to use the operator **-** instead of the method.

>>> a = {1,2,3,4,5}

>>> b = {3,6,7,4}

>>> a - b

{1, 2, 5}

1. **intersection**(...): Return the intersection of two sets as a new set. (i.e. all elements that are in both sets.). In addition, implement the method \_\_and\_\_ to be able to use the operator **&** instead of the method.

>>> a = {1,2,3,4,5}

>>> b = {3,6,7,4}

>>> a & b

{3,4}

1. **union**(...): Return the union of sets as a new set. (i.e. all elements that are in either set.). In addition, implement the method \_\_or\_\_ to be able to use the operator **|** instead of the method.

>>> a = {1,2,3,4,5}

>>> b = {3,6,7,4}

>>> a | b

{1, 2, 3, 4, 5, 6, 7}

1. **isdisjoint**(...): Return True if two sets have a null intersection.
2. **issubset**(...): Report whether another set contains this set.
3. **issuperset**(...): Report whether this set contains another set.
4. **symmetric\_difference**(...): Return the symmetric difference of two sets as a new set. (i.e. all elements that are in exactly one of the sets.). In addition implement the method \_\_xor\_\_ to be able to use the operator **^** instead of the method.

>>> a = {1,2,3,4,5}

>>> b = {3,6,7,4}

>>> a ^ b

{1, 2, 5, 6, 7}

1. **discard**(...): Remove an element from a set if it is a member. If the element is not a member, do nothing.
2. **clear**(...): Remove all elements from this set.
3. **pop**(...): Remove and return an arbitrary set element. Raises KeyError if the set is empty.