Scientific Programming in Python — Exercise Sheet 3

(submission: 29/04/2017 8pm)

Task 1: Lists - do it yourself!

NOTE: Since this exercise is not graded, you do not have to put it on the github repository. If you do, please put all optional exercises somewhere into a different folder, so they do not clash with our evaluation!

You should get used to classes a bit. Create a class **SortedIntList**, which behaves like a list, but only works for integers and will always be sorted.

You can start with a definition like this:

```
class SortedIntList:
  '''A list for integers that will always be sorted.'''
  # your code here ...
```

Functionality that your class should offer:

- Create an empty list.
- Create a SortedIntList from some other iterable. Think about what should happen if some elements of the list are not ints.
- Adding a value to the list. We don't need some append() or insert() functions since you cannot really influence the position of new elements. Make sure that the list stays sorted.
- Removing a value from the list.
- Access and assign elements with square brackets `[]`. Make sure that the list stays sorted when changing values.
- Reading the lenght of the list with Python's built-in `len()`.
- Printing the list in some nice way with Python's built-in `print()`.
- Adding two SortedIntLists should return a new SortedIntList with all values from the old ones combined.

Test your class with some example use-cases. Depending on how you implement the square-bracket assigning `[]`, your list might even work out of the box in for-loops! Put the testing in an if-name-main statement in order to allow for side-effect free importing of your class!

Task 2: == Special Submission Task(s) ==

Your task this week is to write a **Matrix** class that simulates a mathematical matrix and can be used to do basic matrix operations.

Usuall when you write your own classes, you first have in mind how to use them. You think about scenarios that make sense and maybe even code-examples of what would be common operations. Then you think about how to implement that. We will give you these use-cases and you will have to come up with a class that is able to fit them.

Create a new folder in your repository, called `linear_algebra` and a file called `matrix.py` inside. Start your class with:

```
class Matrix:
  # ... your code here
```

In general the matrix should contain only numerical data (ints and floats), but you do not have to check for that specifically! We will only test your matrix with ints and floats. Also you do not need to handle any possible error-scenario other than those we mention here. E.g. you do not need to check if the input list is ill-formed in use-case 1 "creation". Ill-formed would be a mismatch between row or column lengths, e.g.:

```
[[1, 2],
[3, 4, 5]]
```

In the following we describe the use cases. You should make sure that your class fulfills them all. Write a separate script `test_matrix.py` in the `linear_algebra` folder, where you check that everything works. Therefore you will need to import your class from the `matrix.py` file.

Hints

Your will need two magic methods that we did not mention in class, they are: `__mul__()` and `__rmul__()`. Please inform yourself about the usage on the internet.

Evaluation Criteria

We will evaluate all the listed use cases here with different numbers. Please also make sure that you still import no libraries inside the `matrix.py` file, next week we will start working with libraries, then you can use whatever you want ;)

Use cases

1. **Creation:** In general there are two resonable ways of how to create a new matrix. Either you have the complete data already, or you want to use a single value to create a matrix for a given shape. Specific creation procedures like the second one are usually realized with static functions inside of the class. Thus the following code should work as expected:

```
# create a matrix with data being [[1, 2], [3, 4]]
a = Matrix([[1, 2], [3, 4]])

# create a matrix filled with zeros with 2 rows
# and 3 columns
b = Matrix.filled(rows=2, cols=3, value=0)
# this should have the values:
# [[0, 0],
# [0, 0],
# [0, 0]]
```

2. **Printing:** Make sure that your matrix can be printed with nice formatting for debugging and such with python's built-in print-function, like:

```
a = Matrix([[1, 2], [3, 4]])
print(a)
```

   The result should look like this:

```
[[1, 2],
[3, 4]]
```

3. **Data readout:** There should be a way to get all the data stored in the matrix in nested lists format. Make sure that the internal data cannot be modified in this way! This includes making sure that this only gives a copy of the internal data.

```
a = Matrix([[1, 2], [3, 4]])

# should return true
print(a.data == [[1, 2], [3, 4]])

# this must not work!
a.data = [[0, 0], [0, 0]]

# also careful with copying!
internal = a.data
internal[0][0] = 42
```

```
# should still be true
print(a.data == [[1, 2], [3, 4]])
```

4. **Access and modify values:** Normally the square brackets `[]` are used to access and modify values in containers. Your matrix should also support that. A 2D-list can be accessed with e.g. `mylist[0][0]`. For the matrix the following will be even nicer:

```
a = Matrix([[1, 2], [3, 4]])

# should print "1"
print(a[0, 0])

# should print "3"
print(a[1, 0])

# modify
a[0, 1] = 42
# should be true
print(a.data == [[1, 42], [3, 4]])
```

5. **Matrix transpose:** There should be a way to get the transpose matrix of another one. Do this with a property aswell.

```
a = Matrix([[1, 2], [3, 4]])
b = a.T

# should be true
print(type(b) is Matrix)
print(b.data == [[1, 3], [2, 4]])

# again make sure that you copy the data
a[0, 0] = 42
# should still be true
print(b.data == [[1, 3], [2, 4]])
```

6. **Matrix addition:** Adding two matrices with same shape should give the correct result:

```
a = Matrix([[1, 2], [3, 4]])
b = Matrix.filled(2, 2, 1)
c = a + b

# should be true
print(type(c) is Matrix)
```

```
print(c.data == [[2, 3], [4, 5]])
```

7. **Scalar multiplication:** Implement scalar multiplication that works with ints and floats:

```
a = Matrix([[1, 2], [3, 4]])
b = 2 * a

# should be true
print(type(b) is Matrix)
print(b.data == [[2, 4], [6, 8]])
```

8. **Matrix multiplication:** Implement matrix multiplication that works for matrices with the correct shape:

```
a = Matrix([[1, 2]])
b = Matrix([[3], [4]])

# multiply, different orders
c = a * b
d = b * a

# expected results:
print(type(c) is Matrix) # true
print(type(d) is Matrix) # true
print(c.data == [[11]]) # true
print(d.data == [[3, 6], [4, 8]]) # true
```