Exam for the course DAT171 Object oriented programming in Python

Time: 15th March 2016 8:30-12:30

Teacher: Mikael Öhman (mobile: 0736 837 674, office: 031 772 1301)

Permitted aids: Cay Horstmann: Python for everyone. Manuals and lecture notes are available on the computers.

Teacher will visit the rooms: Around 9:30 and 11:30

Formalities: In each file you hand in, you should write your examination code. Also enter the computer number (from the exam cover).

The documentation and lectures notes are available on `C:\__EXAM__`. Verify that these files are there immediately. Handing in the code should be done under the same folder: `C:\__EXAM__\Assignments\`, when you are finished. If you do not store the files here, they are not included in the exam. Save the files for each question in the appropriate folder, e.g. `C:\__EXAM__\Assignments\Question1\`. *Make sure you only hand in one solution for each question, otherwise you will receive* ***zero*** *points.* Complex sections of your code should have a descriptive comment of what is achieved.

When you finish the exam you should log out and fill in the (empty) exam cover page like normal and hand this in at the end of the exam.

Corrections: The results will be announced at the latest on 24th of March on the course homepage. The review will be the same day 12:20-13:00 and on the 29th of March 11:00-11:30.

Grading: There is a total of 25 points which yields grades at the standard 40%, 60%, and 80% limits:

```python
def grade(points):
    if points >= 20:
        return 'Grade 5'
    elif points >= 15:
        return 'Grade 4'
    elif points >= 10:
        return 'Grade 3'
    else:
        return 'Fail'
```

Question 1 (6p total)

You are writing a bot to strip out all the highlighted words from forum posts. This forum uses the old "BB-Code" standard, in which bold fonts are marked as: `'Paragraph of text with some [b]bold text[/b] in some places'`

Part A (3p)

Write a support function that extracts all the sections within `'[b]'` and `'[/b]'` tags, and returns a list of strings. Ignore any other highlights, like italics `'[i]'`.

```
>>> x = extract_bold('[i]Text[/i] with [b]bold text[/b] some [b]places[/b]')
>>> print(x)
[‘bold text’, ‘places’]
```

You may assume that the input text is correctly formatted, with both opening `'[b]'` and closing `'[/b]'`.

Hint: `help(str.split)` *you may also find* `maxsplit` *option useful.*

Part B (3p)

In the folder "bbcodes" available on the computer, there is a set of text files. Loop through all the files in the folder, and extract all the bold text segments. Print all the *unique* extracted text segments from all files, and also write them to one file (line by line). When obtaining the unique text segments, you should ignore case ("Foo" and "foo" should be treated as the same string). Of course, the code should be general enough to work with much larger sets of files.

Hint: Checkout the `os` *module, and the* `listdir` *function. Alternative could be the glob module and function.*

Question 2 (5p total)

Working efficiently with numerical matrices was the initial design goals of NumPy and SciPy. In many scientific problems the need to assemble data into a matrix that you later need to process occur. In this task you will examine the performance of assembling and multiplying matrix formats from NumPy and SciPy.

To simulate the assembling of a matrix we have created files with input data where each row contains: row-index column-index value

Hint: use the following to read these files:

```
import numpy as np
ij = np.loadtxt(filename, usecols=[0, 1], dtype=np.int)
values = np.loadtxt(filename, usecols=[2], dtype=np.float)
```

and the `time` module to measure time.

The file `small.txt` contain values for a small debugging matrix, while the file `large.txt` contains the actual matrix to be used in this task.

The matrix formats to be examined in this task are:

- NumPy ndarray
- SciPy CSC matrix
- SciPy CSR matrix
- SciPy DoK matrix
- SciPy LiL matrix

Part A (2.5p)

Read the data from the given file.

Create a function `assemble(matrix, ij, values)` that assembles into the existing matrix matrix from the given input `ij` and `values` (by looping over the input data and adding values into correct positions). The function must print timing information for its body.

For each of the matrix formats:

- Create an empty matrix of the correct format and with datatype float.
- Assemble into the matrix using the function assemble.

Compare the timing information for the different matrix types.

Part B (1p)

Create a function `multiply(matrix, rounds)` that calculates `matrix * matrix` repeated `rounds` times. The function must print timing information for its body.

For each of the matrix formats call the function multiply specifying 100 rounds in order to obtain a stable average time for benchmarking.

Compare the timing information for the different matrix types.

Part C (0.5p)

Add timing information to your code so that the full run including creating the matrix, assembling and multiplying is included.

Compare the timing information for the different matrix types.

Part D (1p)

As can be seen from part A - C, some formats are better for assembling and others for processing (multiplication in our case).

Choose one type for assembling and another for multiplication so that the total time (including converting from the first to the second type) is the shortest.

Compare the timing information for total times above.

Question 3 (13p total)

Do not use any NumPy or SciPy for this question. This question is about creating classes and not how to get performance in Python. It should only require libraries and components from the standard library, if any.

Matrices are very common and important data structures. In some application, we have a common need to construct special matrices, and we can add optimized functions by overloading operations.

Part A (2p)

Create a base class `Matrix` that stores the matrix dimensions (`rows`, `columns`). Create the specialized classes that inherit from `Matrix`:

- `IdentityMatrix(size)`
- `DiagonalMatrix(diagonal_values)`
- `DenseMatrix(values)`

These should of course be implemented in a smart (and obvious) way, like not storing unnecessary zeros. It should be possible to create them like such:

```
>>> m = DiagonalMatrix([2.0, 3.5, 2.2, 7.0])
>>> m = IdentityMatrix(3)
>>> m = DenseMatrix([[2.0, 0.0, 1.4],
                     [0.5, 1.0, 1.0],
                     [0.8, 3.0, 0.0]])
```

Part B (2p)

The matrices should print nicely. You may decide the details on how to print it, but it should be made clear to the user what it is that is printed (type of object, dimensions etc.). It should handle very large matrices by truncating information if necessary, so that it remains readable, e.g. `'[1, 2, 3, ... 4, 5, 6]'`.

Part C (4p)

Implement indexing for all your matrices using `[i,j]`. You must check and treat each type of matrix differently here, as the type of the matrix should not change due to setting new values (e.g. DiagonalMatrix should only allow diagonal components). Throw suitable exceptions when necessary. Hint: You need to overload `__getitem__(self, indices)` and `__setitem__(self, indices)`, where a call `my_matrix[a,b]` is automatically treated as `my_matrix[(a,b)]` where the tuple `(a,b)` is passed as the argument `indices` above.

Raise a `IndexError` exception with a *helpful* error message if the indices are invalid.

Part D (4p)

The multiplication operator should support multiplication by scalar (returns a new matrix), and dot product with vector (i.e. a `list` of numbers). If operation is not supported, a suitable error should be raised. Make sure these methods return copies. Hint: You might need to have a look at both `__rmul__` and `__mul__`.

```
>>> m = ... # as shown on previous page for any of the matrix types.
>>> v = [1.0, 3.5, 0.4]
>>> u = m * v
>>> m2 = 2 * m
>>> m3 = m * 3
```

Part E (2p)

Write a custom exception for the `DenseMatrix` constructor and raise it if the input in `values` is inconsistent with the row lengths. Store useful information in the exception so that it prints a helpful explanation of exactly what went wrong, such as matrix size vs. vector length. Test the error with a try-except statement and catch the error when doing something intentionally incorrect, e.g.:

```
m = DenseMatrix([[1,2,3],[4,5,6],[7,8],[9,10,11]])
```