Exam for the course DAT171 Object oriented programming in Python

Time: 9th June 2022 8:30-13:30

Teacher: Mikael Öhman (073-6837674)

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

Teacher will visit the rooms: Around 10:00 and 12:00

Formalities: In each file you hand in, you should write your examination code as well as the computer "number".

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 through Ladok, on or before June 29th. The review will be the same day (June 29th) 12:20-13:00 at Hörsalsvägen 7b, 3:rd floor.

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: Camel-Caser (6p)

Part A (2p)

Write the function camel_caser that converts strings from the format my_class_name to the "camel case" naming convention: MyFunctionName

It should work as

```
>>> x = camel_caser('my_class_name')
>>> print(x)
MyClassName
```

Hint: Check through help(str)

Part B (4p)

While correction computer assignments you realize to your horror that many of the students have not been following the standard naming conventions in Python! To correct this, you decide to write a script that reads a file and replaces all occurances of incorrectly named classes, and writes the output to a different file.

The script should take a file student_code.py:

```
class numbered_card:
    def __init__(self, value, suit):
        self.value = value
        self.suit = suit

class standard_deck:
    def __init__(self):
        self.cards = []
```

and produce the readable file readable_code.py:

```
class NumberedCard:
    def __init__(self, value, suit):
        self.value = value
        self.suit = suit

class StandardDeck:
    def __init__(self):
        self.cards = []
```

See the student_code.py, and readable_code.py files in the documentation folder.

Details: The script should handle the simple cases:

```
class some_class:
```

but it does not have to handle

```
class some_class(base_class):
```

It only needs to modify lines that start with class, you don't have to track the usage of the class (which would be more correct, but too difficult for an exam question).

Hint: Remember to close the files

Question 2: Pollutants (5p)

You want to analyse and visualise pollutants in Sweden and have prepared a file "pollutants.txt" with the data. To simplify the task here, you can use the stub file "question2_stub.py" to read the data and set up the headers/labels for it.

After reading and reformatting the data, it is in the format of a 3-dimensional NumPy array with the shape (26, 11, 30). There are 30 years, 26 pollutants and 11 sectors corresponding to these data columns/rows/etc. These can be seen from the headings/labels set up by the stub file.

Note! You may *not* use any loops when calculating the required data below; or for plotting individual data points. Loops may however be used when looping over the headers/labels and units.

Part A (0.5p)

Task: Create a new NumPy array with the sum of pollutants over the sectors. Remember, no loops!

Part B (2p)

Task: In the top row of a 3 column, 2 row plot, using the data from task A, plot the pollutants with unit **t**, **kg** and **g** in separate plots with the year as x-axis.

Hint: The legend should show name of pollutants for part B.

Part C (2.5p)

Task: In the bottom row, plot the sum of pollutants from each sector with unit **t**, **kg** and **g** separate plots with the year as x-axis.

Hint: The legend should show name of sectors for part C.

For full points in part B and C, the plots must include (correct) labels for x-axis, y-axis and a legend.

Question 3: Smart OuterMatrix (6p)

Outer products appear frequently in applications dealing with linear algebra. An outer product of 2 vectors, u and v, result in a matrix, with coefficients $M_{ij} = u_i v_j$

While one can construct such a matrix explicitly, such a block of numbers would take up much more memory, and many operations can be done much more efficiently:

1. Dot product, with vector x:
$$M \cdot x = \sum_j M_{ij} x_j = u_i v_j x_j = u(v \cdot x)$$
By doing the dot product first, we need only do $O(N)$ operations, as opposed to $O(N^2)$.
2. Scaling, with scalar a (you simply scale either vector):
$$M\ a = M_{ij} a = u_i (v_j a)$$
3. Matrix norm (the Frobenius norm):
$$|M| = \sqrt{\sum_{ij} M_{ij}^2} = \sqrt{\sum_i u_i^2} \sqrt{\sum_j v_j^2} = |u|\ |v|$$

By not expanding the matrix directly, we can thus perform much more efficient Matrix operations!

Part A (0.5p)

Write a class `OuterMatrix` that takes 2 NumPy arrays as inputs, and stores *copies* of them internally. This class will represent a matrix, without explicitly computing the outer product.

Part B (0.5p)

Just like other types of special matrices (like sparse matrices), implement the method `toarray()` that computes an expanded NumPy array. *Hint* ***`numpy.outer`***

Part C (0.5p)

Transposing is also common. Add the method `transpose()` which returns a transposed copy of type `OuterMatrix`.

Part D (0.5p)

Implement the efficient (Frobenius) norm as the method `norm`. *Hint* ***`numpy.linalg.norm`***

Part E (1.5p)

To make this matrix actually behave like a matrix, you must now implement indexing for *getting* values off this matrix. You should support slices and integer values (see the 4 cases in the example below)

Part F (1.5p)

Implement efficient multiplication for these cases:

```
x = np.array( ... )
m = OuterMatrix(u, v)
y = m * x  # performs dot product
m2 = m * 3.14  # computes a scaled matrix
m *= 2  # scales m
```

If the type isn't a scalar (`int`, `float`) or a 1D NumPy array; raise a *suitable* error.

Test

```
import numpy as np
a = np.array([9., 1., 2., 3., 4., 5., 6.])
b = np.array([2., 1., 4., 3., 6., 5., 7.])
c = np.array([1., 2., 1., 7., 8., 2., 4.])

m = OuterMatrix(a, b)

print("Expanded:\n{}".format(m.toarray()))
m_sub = m[1:3, 2:5]  # This should also be a OuterMatrix.
print("Slicing (expanded):\n{}".format(m_sub.toarray()))
print("Slicing:", m[1:3, 5])  # slice + int -> ndarray
print("Slicing:", m[3, 2:5])  # int + slice -> ndarray
print("Item:", m[2, 4])  # int + int -> float
print("Norm:", m.norm())
m *= 2.0
print("Norm after scaling:", m.norm())
print("Multiplication with array:", m * c)
m_sub_t = m_sub.transpose()
print("Transpose (expanded):\n{}".format(m_sub_t.toarray()))
# Exceptions
#print(m * c.tolist())
#print(m * np.stack((c ,c)))
```

Output

```
Expanded:
[[18.  9. 36. 27. 54. 45. 63.]
 [ 2.  1.  4.  3.  6.  5.  7.]
 [ 4.  2.  8.  6. 12. 10. 14.]
 [ 6.  3. 12.  9. 18. 15. 21.]
 [ 8.  4. 16. 12. 24. 20. 28.]
 [10.  5. 20. 15. 30. 25. 35.]
 [12.  6. 24. 18. 36. 30. 42.]]
Slicing (expanded):
[[ 4.  3.  6.]
 [ 8.  6. 12.]]
Slicing: [ 5. 10.]
Slicing: [12.  9. 18.]
Item: 12.0
Norm: 155.1773179301666
Norm after scaling: 310.3546358603332
Multiplication with array: [2070.  230.  460.  690.  920. 1150. 1380.]
Transpose (expanded):
[[ 4.  8.]
 [ 3.  6.]
 [ 6. 12.]]
```

Question 4: Logger (8p)

Most long running applications have a need to use different logging methods. The most common method is to simply write all messages to a file, but there are cases where the logging frequency would create files that are much to large for practical use, and we may want to only save the last N messages. In this assignment, you will implement 2 file writing classes, which will enable a common logger class to write either all data, or the last N outputs, under a given filename.

Part A.1 (1p)

Create a base class `Writer` which takes a `filename` in its `__init__` This class should have 2 abstract methods:

- `.write_line(text)` which takes a line of text to write.
- `.flush()` which "flushes" all the content to the file with the given filename. (In programming terms, flushing means to force-write all data to disk)

Part A.2 (2p)

Implement the subclass `DirectFileWriter` which opens and writes each line to a file "like normal". (This class is only a very simple wrapper for a file object.)

The file object you get from `open` has a method called `flush()` which you should call from your `flush` method.

Part A.3 (3p)

Implement the subclass `CircularWriter` which has a cicular buffer which stores the last N lines of output in memory, and writes all of it to disk when `flush` is called.

A circular buffer consist of a list of length N, starts writing at index 0, and when it reaches N it restarts at 0 (overwriting the oldest entry).

Part B.1 (0.5p)

In this question you should implement a logging class with a single method `log` which uses the file writer from Part A.

Logging codes almost always has multiple levels of verbosity, commonly called "DEBUG", "WARNING", or "ERROR". Create an `enum` which you can use to determine the logging level.

Part B.2 (1.5p)

Create a class `Logger` which takes a name of the logger, a logging level, date format (see example), and a `Writer` object (from part A).

Implement the method `.log(level, msg)` which takes a message, prepends a date, name, and log-level, formatted as:

`[date-stamp] Name::LogLevel - Some important message.`

Nothing should be printed if the level is below the log level (the levels are inclusive: `WARNING` also prints `ERROR` messages, and `DEBUG` alsos prints `WARNING`'s and `ERROR`'s.).

This is how you get a time stamp and use a custom date format:

```
from datetime import datetime
time_stamp = datetime.now()
print(time_stamp.strftime('%Y-%m-%dT%H:%M:%S'))
```

Hint: Enums have a .name variable which you can use for formatting the string

Example:

```
# Either writer should work for your logger:
use_circular = True

if use_circular:
    # We test it with a very small circular array:
    writer = CircularWriter(5, 'logfile.txt')
else:
    writer = DirectFileWriter('logfile.txt')

# We choose to store microseconds in this example since the test example is so brief:
logger = Logger('MyProgram', LogLevel.WARNING, '%Y-%m-%dT%H:%M:%S.%f', writer)

logger.log(LogLevel.DEBUG, 'This debug message should be ignored!')
for i in range(10):
    logger.log(LogLevel.WARNING, 'Testing {}'.format(i))

logger.log(LogLevel.ERROR, 'This error should be printed last in the file.')
logger.log(LogLevel.DEBUG, 'And this shouldn\'t be in the file at all.')

# Force flush the buffer before exiting to ensure everything is written to file:
writer.flush()
```