Exam for the course DAT171 Object oriented programming in Python

Time: 19th Mars 2015 14:00-18:00

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 15:00 and 16:30

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 recieve* ***zero*** *points.* Complex sections of your code should have a descriptive comment of what is achieved.

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

To run the Lectures notes in IPython notebook, you must copy the link from the start menu, and change the "Start in:" property in the shortcut to `C:\`.

Corrections: The results will be announced the lastest on 31st of March on the course homepage. The review will the same day 12:20-13:00 and on the 1st of April 12:20-13:00.

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

```
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)

Part A (2p)

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

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 you 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 PlayingCard:
    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 (6p total)

The goal here is to evaluate the numerical integration methods available in numpy and scipy. Our main interest is the performance w.r.t to accuracy. To benchmark the alternatives, we look at the problem of integrating

$$\int_0^5 f(x)dx$$

where

$$f(x) = \cos(x) - \sin(x^2).$$

For part 1 and 2, choose an evenly distributed array of x with N values. *Hint: numpy.linspace*

Part 1: 1.5p

Using a for-loop, evaluate $f(x_i)$ for every x_i. Use `numpy.trapz` to evaluate the integral by passing in the evluated f_i and x_i arrays.

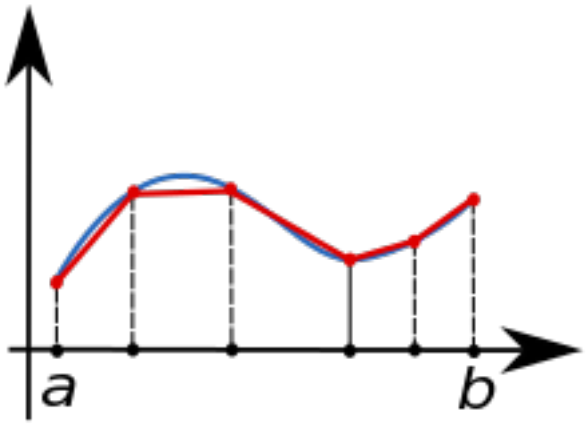

Figure 1: Illustration of the integral using the trapezodial (trapz) rule from a to b

Hint: use `help(numpy.trapz)` *to quickly check the usage instructions*

Part 2: 1.5p

Instead of looping, use NumPy functions directly on the NumPy array x to evaluate $f(x)$. Use the trapezodial rule again to evaluate the integral (which should give exact the same answer as part 1).

Part 3: 2p

Find a suitable method in SciPy to directly evaluate the integral $\int_0^5 f(x)dx$ (without first evaluating the function). *Hint: This should be by far the fastest method*

Part 4: 1p

Measure the time to compute the integral for the 3 methods. Pick an N so that the accuracy matches that of part 3.

Hint: use `time.time()` *to measure the evaluation times*

Question 3 (5p total)

Often one obtains data in an *interleaved* form $[x_1, y_1,\ x_2, y_2,\ \ldots,\ x_n, y_n]$ (two lists, stored in one list, with every other value). When looping over such a list, we want to obtain both `x` and `y` values directly. While we could do:

```
data = [0, 1, 2, 3, 4, 5] # interleaved data
for i in range(len(data)//2):
    x = data[i*2]
    y = data[i*2+1]
    print("x =", x, "and y =", y)
```

but it can get messy if we need to do this in nested loops and in many places.

Instead, write an *iterator* `Deinterleave` that grabs 2 values at a time. The iterator should work as:

```
>>> data = [0, 1, 2, 3, 4, 5]
>>> for x, y in DeInterleave(data):
>>>     print("x =", x, "and y =", y)
x = 0 and y = 1
x = 2 and y = 3
x = 4 and y = 5
```

You may assume that the data has even length.

Hint: See the lecture notes on iterators.

Question 4 (8p total)

In December 1998 NASA launched the *Mars Climate Orbiter*. This $327.6 million research project ended prematurely when the orbiter crashed into mars due to not properly handling american and SI units. These are the catastrophic errors that can be caused by using simple floating point numbers without keeping track of units.

Create an (empty) base class `DistanceUnit` and subclass this into `Meter` and `Foot` (they should inherit from `DistanceUnit`). `Meter` and `Foot` should define a variable for scaling the value to the SI unit (which would be 1.0 for `Meter`, and 0.3048 for `Foot`).

Create a class `Distance` whose constructor takes a magnitude and a `DistanceUnit` as input. The constructor in `Distance` should check that the unit is a `DistanceUnit` (this can be done with the `issubclass` function) and if not, raise a suitable error.

Overload the following in `Distance`:

- `+` The result from the `+` operation should be a new `Distance` with the same unit as the left hand operand, e.g. `meter + foot` should have it's results in `meter` and `foot + meter` sould have the result in `foot`.
- `-` Should have the same behavior as `+`.
- `*` Multiplication with a (unitless) factor. Both `__rmul__` and `__mul__` should be supported (e.g. `3.4 * x` and `x * 3.4`).
- `/` division between "Distance/Distance" should return a (unitless) factor, and division with a (unitless) factor "Distance/float" should return a `Distance`. You can use `isinstance` or `type` to check what to use. *Hint: `__truediv__` is the normal division in Python 3.*
- The `str` conversion so that it is printed nicely (add suitable functionality to the units to support this).

These units should be able to handle:

```
>>> meter = Distance(1, Meter)
>>> foot = Distance(1, Foot)
>>>
>>> x = 3 * meter
>>> y = 2.3 * foot
>>>
>>> print("x =", x)
x = 3m
>>> print("y*2 =", y*2)
y = 4.6ft
>>> print("x+y =", x + y)
x+y = 7.40208m
>>> print("(2.5m - 4ft)/5 =", (2.5 * meter - 4 * foot)/5)
(2.5m - 4ft)/5 = 0.25616m
>>> print("x/y =", x / y)
x/y = 4.27935638
```

You may not assume that Meter and Foot are the only possible units. It should be possible to add new `DistanceUnits` without modifying the `Distance` class