Homework Set #3
(Classes and user-defined types)

Imperative Progamming with Python

January 2015

The homework should be uploaded using the BlackBoard system, it should *not* be printed. Read the `guidelines.pdf` (on the website) for the submission guidelines. **Due date:** 26/01/2015.

This homework is mostly about *design choices*. The main problem is that there is an inherent ambiguity in this exercises. You will feel desperate, but don't panic. There is no such thing as *the* right design, but there are many good criteria. I have named some of them during the course: separation of concerns, do not repeat yourself, good organization of the code into functions and modules, design for change, information hiding, etc. There are many more, but it all boils down to a few things: you want to make code that is easy to understand, and that is solid enough to survive some unexpected changes. When solving this exercises, you will probably (have to) consider many different options, and you will finally settle on one. You will have to explain in the report why you made that choice, and what other things you did consider. Also, how your solution satisfies the specifications. Some things will be *under*specified, and you will have to explain why your choice is a reasonable choice. This does happen in real life, more than we would like.

Exercise 1 (5 pts.)**.** *Information hiding.*
Another way of representing colors in the RGB space is as a 3-tuple of integers (r,g,b) where $r, g, b \in \{0, \ldots, 255\}$. Take the type `Color` defined in class #6 and change its internal representation of colors to use this one. It is truly and utterly important that the interface of the class stays the same: all the methods should receive and return the same types and in the same range as before. The idea is that the change in the internal representation should go unnoticed to the outside world.

Exercise 2 (20 pts.)**.**

a) Write a user-defined type `Matrix` which represents a matrix of integers. A matrix with n rows and m columns is initialized (when constructed!) with a list of n lists of m elements. For example, a 3×2 matrix can be initialized with `[[1,2],[33,4],[5,6]]`. You should be able to sum, multiply and compare the matrices using the `+`, `*` and `==` operators respectively. You should also be able to print the contents of the matrix using `print m` where `m` is the matrix object. The above matrix, for example, should be printed as

```
1 2
33 4
5 6
```

Don't worry about the alignment of the columns.

b) Suppose that, because of our context, we know that sometimes we use *sparse matrixes*, that is, most of the elements are zero. Add/modify the necessary classes to take advantage of this feature and save some memory by not storing the elements that are 0 (probably at the expense of time). The class should be called `SparseMatrix`. Also, I have written a function `DoMagic(m)` which takes as an argument that I expect to be of type `Matrix`. I want this function to work with both matrices and sparse matrices... in an elegant way.

Exercise 3 (25 pts.)**.** *Propositional model checking.*
A formula from propositional logic is inductively defined as $\varphi ::= p \in \text{Props} \mid \neg\varphi \mid \varphi \wedge \varphi$. Mathematically speaking, a valuation is a function $v : \text{Props} \to \{\texttt{True}, \texttt{False}\}$ assigning `True` or `False` to every propositional variable. We want to write a function `satisfied(phi, v)` which returns `True` iff the formula *phi* is satisfied by the valuation v.

a) Choose an appropriate representation for the valuation.

b) Choose a representation for the formulas, and define the necessary classes. Think object-oriented: what do all formulas have in common? How are they constructed?[1]

c) Implement (the classes of) the formulas in such a way that printing them shows a nice representation of the formula, e.g: `∼((p & q) & r)`.

d) Write the `sat` function using the types of the last point.

e) Suppose now we consider the syntax $\varphi ::= p \in \text{Props} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \Rightarrow \varphi$. Modify the program of the last point to include the implication. It should be easy to do and you shouldn't need to modify any of the previous classes. If that is not the case, probably the design was not so good!

Exercise 4 (20 pts.)**.** *Bank accounts.*
Consider the following context: We will be dealing with bank accounts. They have an associated number and owner. You can withdraw and deposit money to the accounts. We have two types of bank accounts, the "Normal" account and the "Savings" account. The savings account is free to be opened whereas the normal account costs 1 euro. It is free to deposit in both account types and it is also free to withdraw from the normal account. In the case of the savings account you can withdraw for free once a day, after that, it costs 2.5 euro each time.

You should write a program that simulates the bank: a client should be able to open a new account, check the balance of their accounts, deposit and withdraw. The program should start as a bank without any account and get filled as used by (possibly) different clients. When the program is asked to finish, the information about the accounts gets lost.

Exercise 5 (15 pts.)**.** In **R**ole **P**laying **G**ames you control a character which has a name, hit points (life) and strength. Characters can be of two different classes: Fighter, Wizard. They also have a race: Human or Orc. The characters use weapons to fight, these weapons can be of different types: blunt (e.g: staff), piercing (e.g: rapier, dagger), long range (e.g: bow). When a character attacks another one, the damage is calculated using the formula $d = \text{strength} + \text{class modifier} + \text{weapon damage}$. The class modifier is +1 for Fighter and -1 for Wizard. Each weapon has its own damage points.

Read section 18.8 of the book and draw a class diagram for the RPG described above. Also include in the diagram the list of attributes and methods that you think are important for the scenario. You don't need to write any code.

Things to take into account: what if you want to add a new weapon or class or race later? would that be easy?

Exercise 6 (5 pts.)**.** The built-in function `sum(t)` calculates the sum of a list of ints. It works with some other types but not for strings. Write a function `sumall` which works for every type that implements the `+` operator.

Exercise 7 (10 pts.)**.** *Serializing.*
Modify Exercise 4 to save the accounts information to a file so that it gets reloaded when the program is ran again.

[1] Hint: consider a base class `PropositionalFormula` with a pure virtual method `sat(v)`.