# ADVANCED PYTHON HOMEWORK 5

Object Oriented Mathematics

Choose one of the following tasks and solve it. If you choose exercise 1, you also should choose one of the subtasks (A), (B) or (C); it is enough to solve one variant. Every task is worth 5 points.

Lab teacher's notes: Please follow the class names exactly as the homework sheet describes, where specified. Do not forget to test your solutions with a reasonable amount of tests, and include them in the homework. If you ever make assumptions that are not specified in the exercise, make sure that they are reasonable and well documented.

Furthermore, if any statement says "for example" or "e. g." without specifying exactly how much work is needed, this may indicate that it is indeed just an example and you are expected to do at least slightly more.

Please name your solutions *ex1a*.py, ex1b*.py, ex1c*.py, ex2*.py*. The star stands for any string of your preference.

**Exercise 1.** Implement the class **Expression** with the appropriate subclasses representing different kinds of arithmetic expressions. For example, an expression:

```
(x + 2) * y
```

can be represented as:

```
        Times(Add(Variable("x"), Constant(2)), Variable("y"))
```

where **Times**, **Constant**, **Variable** are appropriate subclasses of the class **Expression**. Program for each class the following methods:

- `evaluate(self, variables)` computing the value of the expression; argument variables stores information about what values the corresponding variables have.
- `__str__` returns a nicely-formatted human-readable representation as a string.
- `__add__` and `__mul__` that for two objects $w_1$ and $w_2$ of the class `Expression` returns a new object `Expression` representing the sum and the product of $w_1$ and $w_2$ respectively.

Program your own exceptions (classes) to react to incorrect situations, e.g. division by zero or missing value assignment to a variable. It is required that constants, variables and basic arithmetic operations be defined.

The implementation of the above task can be extended in various ways:

(A) You can create a similar class hierarchy representing a simple programming language with at least these instructions: assignment statement, conditional `if` statement and a `while` loop statement. Next, add an instruction representing a sequence of instructions. It can be assumed that an arithmetic expression equal to zero is interpreted as false and true otherwise. In each of these classes you need to program a method `run(self, variables)` that executes subsequent instructions. Add a method `__str__` that returns a string with a nicely-formatted human-readable representation of the program.

(B) Having expressions in the form of such a tree, you can try to simplify these expressions, for example expressions containing only constants of the type `2 + 2 + "x"` can be simplified to `4 + "x"`, or use the property of multiplication by zero. Program at least two such simplification rules in the form of a simplifying static method of the class **Expression**.

(C) Expressions containing only one variable can be treated as functions. For expressions that allow this interpretation, implement a static method which computes and returns an expression (object of the class **Expression**) corresponding to the derivative of the **Expression** on input.

**Exercise 2.** Design and implement the class **Formula** (with appropriate subclasses) that represents a propositional formula. For example a formula

$$\neg x \vee (y \wedge true)$$

can be represented as:

```
Or(Not(Variable("x")), And(Variable("y"), Constant(True)))
```

Implement the following methods in each class:

- `evaluate(self, variables)` computing the value of the expression; the argument **variables** stores information about the values of particular variables.
- `__str__` returns a nicely-formatted human-readable representation as a string.
- `__add__` and `__mul__` that for two objects $w_1$ and $w_2$ of the class `Formula` return a new object `Formula` representing the disjunction and the conjunction of $w_1$ and $w_2$ respectively.
- `tautology` that checks whether a given formula is a tautology. In practice it is convenient to implement `tautology` as a static method of **Formula**.

Furthermore, implement a method that finds a simplification of a given formula using the equalities such as $p \wedge false \equiv false$ or $false \vee p \equiv p$. Design and implement your own exceptions (classes) handling an incorrect situations, e.g. no value assignment to a variable.