# Overview of the application

The skeleton provided implements a simple arithmetic calculator.

The calculator is given an expression, in the infix format, and it converts it to a postfix expression.

Then it builds an expression tree, which is then evaluated.

The expression consists of numbers, the 4 arithmetic operators and left and right parenthesis.

For the convenience of parsing the expression, all its elements above are separated by white space (e.g. 1 space), for example an expression like (5+2)\*(3+1) must be entered as

( 5 + 2 ) \* ( 3 + 1 ), otherwise the program will raise a parsing error such as “ValueError: token '(5' is not valid”.

The program also supports an algebraic expression, where instead of numbers it uses letters (essentially Python identifiers), for example ( a + 2) \* ( b + 1 ). The evaluate method of the calculator takes an optional context which defines the values for these variables, e.g. a = 5 and b = 3, and thus is able to evaluate the expression tree to a numerical value.

# Code Walk-thru

**Operator** is base class for all the operators, which derive from it: **Plus**, **Minus**, **Multiply** and **Divide**.

Each of the operators’ subclasses define only 2 class attributes: symbol and priority.

The symbol is the literal(s) used for that operation in the expression.

The priority is an integer: operators with higher priority takes precedence.

The base class has some utility methods like:

* *fromsymbol(cls, symbol)* – this is the alternative constructor that we discussed in the lecture, which instantiates an **Operator** subclass from its symbol
* *get\_symbols(cls)* – another class helper method which returns a list of operators’ symbols
* *\_\_eq\_\_(self, other)* – to support the **Operator** equality. 2 operator instances are equal if they are of the type, or if not, if they have the same priority
* all the remaining relational operators *(\_\_gt\_\_, \_\_lt\_\_*, etc.) are similar and are left to be implemented by you
* the string representations for the Operator, as discussed in the module: *\_\_str\_\_* and *\_\_repr\_\_*

**Calculator** is the main implementation class and contains only class and static methods.

The public “interface” to this class is just 1 method: *calculate(infix\_expr)*.

The other methods are internal to its implementation:

* *\_infix2postfix(cls, infix*expr) – converts an infix expression to a postfix one
* *\_build\_tree(cls, postfix\_expr)* – build an expression tree from the postfix expression
* *\_buildexpr(infix\_expr)* – is just a wrapper around *\_infix2postfix* and *\_build\_tree*, returning the expression tree for the infix expression
* *postfix\_eval(cls, postfix\_expr, context=None)* – computes the value of a postfix expression by calling the *evaluate* method on its root (essentially traversing the expression tree in post-order and applying the operator)

**Context** class is just a placeholder to provide the numerical values for algebraic symbols used in an expression.

**Expression** is the base class for all other expressions. It is an abstract class defined by subclassing abc.ABC as described in the lecture. It has a couple of helper methods and 1 abstract method, *evaluate(self, context=None)*.

The **BinaryExpression** is an Expression subtype for expression with 2 operands: it has a left and a right operand defined as read-only properties.

In addition it has a factory class method, *create(cls, op, left\_expr, right\_expr)*, which creates the correct **BinaryExpression** subclass, based on the operator.

Lastly, it has a few “pythonic” aspects, as discussed in the lecture:

* support iteration (of the operands) by implementing *\_\_iter\_\_* special method
* supports equality by implementing \_\_eq\_\_ method
* support a string representation by implementing \_\_str\_\_, \_\_repr\_\_ and \_\_format\_\_ methods.  
  \_\_str\_\_ returns the user-friendly infix representation of the expression and \_\_repr\_\_ returns a combination of class name, operator and operands.  
  \_\_format\_\_ supports a custom format expression: if the format specifier ends in ‘p’, it returns the expression in postfix format, otherwise as an infix format.
* It is hashable by implementing the \_\_hash\_\_ special method

**BinaryExpression** has 4 subclasses: **AddExpression**, **SubtractExpression**, **MultiplyExpression** and **DivideExpression**. Each of these expressions have the operator as a class attribute, and implement the initializer, and the abstract *evaluate* method.

Finally, there is a **TerminalExpression** which a direct subtype of **Expression** and represents an operand which is accessed through the read-only property *token*. It also implements the “pythonic” methods *\_\_str\_\_*, *\_\_repr\_\_*, *\_\_format\_\_* and *\_\_hash\_\_*.

# Assignment

The assignment consists of completed all the TODO markups in the code:

1. Implement *get\_symbols* and *is\_operator* methods in the **Operator** class, and also the rest of the relational methods
2. Complete the **Operator** subclasses: **Minus**, **Multiply** and **Divide**
   1. Once this step is completed, you may run the unit tests and step thru the code
3. Add support for unary expression in the Calculator *\_build\_tree* method
4. Implement the *\_postfix\_eval* and *calculate* methods of the **Calculator**
5. implement the **UnaryExpression** (expression with 1 operand only) class
   1. Look at the code for **BinaryExpression**
6. Implement the **SquareRoot** class. At this point, the program should be able to evaluate an expression like this: “sqrt 9 + 7” or “sqrt ( 5 + 4 )”
7. Optional assignment: make the **Expression** class callable, either by subclassing the Callable ABC (from collections.abc) or just by simply implementing the “callable” protocol. Either way, it requires implementing the *\_\_call\_\_* method. The method will allow to evaluate an expression by simply calling it, instead of invoking its *evaluate* method:  
   expr = AddExpression(TerminalExpression(4), TerminalExpression(5)  
   val = expr()

After completing the implementation, all the tests should pass.

To run the tests, navigate to the homework root directory in a cmd (Windows) or Terminal (Mac) window and enter:

$ python3 -m unittest tests.calculator\_tests

token=(, postfix=[], stack.py=['(']

token=5, postfix=['5'], stack.py=['(']

…

.

----------------------------------------------------------------------

Ran 3 tests in 0.003s

OK

# Instructions for download

Download the homework\_mod1.zip file and expand it in a directory. The directory structure should look like this:

.

├── ReadMe.docx

├── \_\_init\_\_.py

├── calculator.py

├── tests

│   ├── \_\_init\_\_.py

│   └── calculator\_tests.py

└── utils

    ├── \_\_init\_\_.py

    └── stack.py

Open the project in your preferred IDE or a text editor, and open the calculator.py file; follow the TODO comments.

# What’s Next

The next assignment will build upon this one, to implement the GUI for this calculator.