calculator.py

```python
import ast
import math
import operator
from typing import Dict


class Calculator:
    """
        The Calculator (wrapper) class can evaluate a (python) mathematical expression string in conjunction with a
        variable-to-value mapping and compute its result.
    """

    def __init__(self, verbose: bool = False) -> None:
        # save params
        self.verbose = verbose
        # create ast calculator object
        self.calc = self.Calc()

    """
        Mapping between ast operation object and operator operation object.
    """
    _OP_MAP: Dict[type(ast), type(operator)] = {
        ast.Add: operator.add,
        ast.Sub: operator.sub,
        ast.Mult: operator.mul,
        ast.Div: operator.truediv,
        ast.Invert: operator.neg,
        ast.USub: operator.sub
    }

    """
        Mapping between ast comparison object and operator comparison object.
    """
    _COMP_MAP: Dict[type(ast), type(operator)] = {
        ast.Lt: operator.lt,
        ast.LtE: operator.le,
        ast.Gt: operator.gt,
        ast.GtE: operator.ge,
        ast.Eq: operator.eq
    }

    """
        Mapping between mathematical functions (string) and mathematical objects.
    """
    _CALL_MAP: Dict[str, type(math)] = {
        "sin": math.sin,
        "cos": math.cos,
        "tan": math.tan,
        "sinh": math.sinh,
    "cosh": math.cosh
}

def eval_expr(self, variable_map: Dict[str, float], computation_string: str) -> float:
    """
    Given a mapping from variable names to values and a mathematical (python) expression, it evaluates the
    expression.
    :param variable_map: a dictionary map containing all variables of the computation_string
    :param computation_string: a python-syntax-compatible input string
    :return: the result of the expression
    """
    return self.calc.evaluate(variable_map, computation_string)

"""
    Internal Calc class for the actual calculation.
"""

class Calc(ast.NodeVisitor):

    def __init__(self) -> None:
        """
        Initializes the actual expression evaluator.
        """
        # init variable map
        self.var_map: Dict[str, float] = dict()

    def visit_BinOp(self, node: ast) -> float:
        """
        Binary operation evaluator.
        :param node: ast tree node
        :return: result of binary operation (LHS op RHS)
        """
        left = self.visit(node.left)
        right = self.visit(node.right)
        return Calculator._OP_MAP[type(node.op)](left, right)

    def visit_Num(self, node: ast) -> float:
        """
        Numeral evaluator.
        :param node: ast tree node
        :return: numeral value
        """
        return node.n

    def visit_Expr(self, node: ast) -> float:
        """
        Expression evaluator.
        :param node: ast tree node
        :return: value of the expression evaluated with the given variable map
```

```python
        """
        return self.visit(node.value)

    def visit_IfExp(self,
                    node: ast) -> float:  # added for ternary operations of the (python syntax: a if expr else b)
        """
        Ternary operator evaluator.
        :param node: ast tree node
        :return: value of if clause if comparison evaluates to true, value of else clause otherwise
        """
        if self.visit(node.test):  # evaluate comparison
            return self.visit(node.body)  # use left
        else:
            return self.visit(node.orelse)  # use right

    def visit_Compare(self, node: ast) -> bool:  # added for ternary operations (python syntax: a if expr else b)
        """
        Comparison evaluator.
        :param node: ast tree node
        :return: whether the comparison evaluates to true of false
        """
        left = self.visit(node.left)
        right = self.visit(node.comparators[0])
        return Calculator._COMP_MAP[type(node.ops[0])](left, right)

    def visit_Name(self, node: ast) -> float:
        """
        Variable evaluator.
        :param node: ast tree node
        :return: variable value
        """
        return self.var_map[node.id]

    def visit_Call(self, node: ast) -> float:
        """
        Function evaluator.
        :param node: ast tree node
        :return: value of the evaluated mathematical function
        """
        return Calculator._CALL_MAP[node.func.id](self.visit(node.args[0]))

    def visit_UnaryOp(self, node: ast) -> float:
        return Calculator._OP_MAP[type(node.op)](0.0, self.visit(node.operand))

    @classmethod
    def evaluate(cls,
                 variable_map: Dict[str, float],
                 expression: str) -> float:
        """
```

```
            Entry point for calculator.
            :param variable_map: mapping from value names to values
            :param expression: mathematical expression in string format
            :return: result of the evaluated string
            """
            # remove LHS of the equality sign e.g. ’res=...’ --> ’...’
            if "=" in expression:
                expression = expression[expression.find("=") + 1:]
            # parse tree
            tree = ast.parse(expression)
            # create calculator
            calc = cls()
            # add the variable value mapping
            calc.var_map = variable_map
            # evaluate expression tree and return result
            return calc.visit(tree.body[0])


’’’
    safe (in contrast to evaluate()) python expression evaluator class
        -input:
            - map: variable name -> value
            - computation string (must be python syntax, e.g. for ternary operations)
        - output: resulting value

    credits: https://stackoverflow.com/questions/33029168/how-to-calculate-an-equation-in-a-string-python

’’’

if __name__ == "__main__":

    ’’’
        simple example for debugging purpose
    ’’’

    variables = dict()
    variables["a"] = 7
    variables["b"] = 2

    for var in variables:
        print("name: {}, value: {}".format(var, str(variables[var])))

    computation = "cos(-a + b) if (a > b) else (a + 5) * b"
    calculator = Calculator()
    result = calculator.eval_expr(variables, computation)
    print("{} = {}".format(computation, str(result)))
```