Home Assignment 4

General:

This assignment will cover the following topics

- Object oriented programing
- Inheritance
- classes

Many academic/scientific applications allow the user to calculate and analyze mathematical functions. Application such as the famous MATLAB are commonly used, allowing us the make complex calculations, drawing graphs and many other features desired by engineers, researches etc.
In this assignment, we describe a variety of mathematical expressions. A mathematical expression is a sequence of characters (numbers, variables, and operators) with mathematical meaning. In particular, a mathematical expression is an object, which can be evaluated, differentiated, and by using mathematical operators can create new expressions. A number is a mathematical entity and so is x+y, which involves an operator applied to two other expressions (x,y which are expression themselves). Mathematical expressions often contain variables; in order to evaluate this kind of expression, an assignment of values to the variables is required.

Input:

Our test will import your file and run a complete program: creating and evaluating mathematical expression, applying the methods define below.

Output:

The output should be printed to the terminal using the print() function.

Constraints:

- No modules are allowed. The TA should approve the use of any module.
- A request for using a module should be submitted vial Email to guyan@bgu.ac.il

You should implement the following interfaces:

1. *Variable* – a single variable with no value assignment.
   - `get_name()` – returns a string which is the variable name.
2. *Assignment* – a numeric value assignment to a variable.
   - `get_var()` – returns a variable object.
   - `get_value()` – returns a float object which is the value assignment to the variable.
   - `set_value(n)` – changing the value assignment to the variable to be of value n.
   - `__repr__(self)`
3. *Assignments* – a collection of Assignments. Each class that inherits this interface stores the assignments using a dictionary.
   - `Operator[v]`– will return the numeric value of the variable 'v' within the brackets. None object will be returned if 'v' was not found.
   - `Operator += ass` : will add the assignment 'ass' to the dictionary.
4. *Expression* – a mathematical expression

- `evaluate(self, assigms)` – gets an *Assignments* object and evaluate the expression value accordingly. A numeric value is returned. If some variable assignment is missing, a ValueError exception is raised
- `derivative(self,v)` – returns a new expression which is the **symbolic** derivative of the expression according to 'v'.
- `+ operator (self, other)` – return a new *Expression* which stands for self+other. 'other' should be an *Expression* object as well, if not raise an exception.
- `- operator (self, other)` - return a new *Expression* which stands for self-other. other should be an *Expression* object, if not raise an exception.
- `* operator (self, other)` - return a new *Expression* which stands for self*other. other should be an *Expression* object, if not raise an exception.
- `** operator(self, p)` - return a new *Expression* which stands for self to the power of p. p should be a float object, if not raise an exception.
- `__repr__(self)`
- `__eq__(self, other)`

<u>Next we describe the specific classes which implement the these interfaces:</u>

1. **<u>Variables and assignments classes:</u>**

- *ValueAssignment* – implements the *Assignment* interface. It represents a numerical assignment to a variable.
  - `Contructor` with two arguments – *Variable* object 'v' and a float object 'value'.
  - `== operator` : returns True if both the Variable name and the Value are equal.
  - `__repr__` : a string representing the assignment by the form of "x=5.5" where x is the variable name and 5.5 its value
- *SimpleDictionaryAssignment* – implements the *Assignments* interface using a dictionary.
  - `Constructor with no arguments`- initialize an empty assignments dictionary.
  - Notice: using the += operator will override a variable value if it is already stored in the object, otherwise it will add the new assignment as key-value pair in the dictionary.

2. **<u>Expressions:</u>**

1. *Constant:* this is the simplest expression. It implements the *Expression* interface.
   - Constructor that takes one float object argument - initialize the object with this value, this argument should have a default value of 0.0
   - *Notice:* two Constant objects will be equal if they have the same float value. Constant derivative is always a Constant object initialized with zero.
   - Notice: using __repr__() on a Constant object will create a string with its stored value only.

2. *VariableExpression*: implements *Expression* and *Variable* classes. It represents a variable. This object can be evaluated if given an assignment that corresponds to the variable it represents.
   - Constructor that takes a single string object argument, which is the variable name.
   - `__repr__` should return the variable name only
   - `evaluate(self, assigms)` – returns the evaluated value of the variable using 'assigms' which is an Assignments object. If no corresponding variable was found in the Assignments object, None object will be returned.
   - `derivative(self, v)` - will always return a Constant object. The Constant object might be initialized with 1.0 or 0.0 according to the variable it has been differentiated.
   - *Notice:* two *VariableExpression* will be equal if they have the same variable name.

3. You should now implement the following classes which implements *Expression*:

   - *Addition*
   - *Subtraction*
   - *Multiplication*

   - All of the above must implements the *Expression* interface. You may add other methods.
   - You should implement a constructor which takes two *Expressions* A and B that represent two mathematical expressions. The return value is a new Expression which represents the corresponding logical operation. For example, a new Addition object initialized with A and B will represent A+B.
   - Evaluating Addition, Subtraction and Multiplication can be done only if both parts (A and B) can be evaluated, otherwise the evaluation will raise a type error.
   - derivative() method will return a new Expression which is the symbolic derivation of the object. For example: assume X is an Addition object, calling X.derivative (v) will create a new Addition object with the two expressions it possess after using derivative method on both of them (more examples below).
   - `__repr_()` should return a new string of the form of *(A.__repr__()$B.__repr__()* where A and B are both parts of the Expression and $ is the corresponding operand (+, -, *).

4. *Power* - implements Expression interface.
   - Constructor: takes two arguments: *Expression* and a float number to be used as power for the first argument. For example:

```
x = Variable("x")
Y = Variable("Y")
add = Addition(x, y)
power = Power(add, 3.4)
print(power)
```

The output will be:
(x + y)^3.4

- ➢ Do not forget to complete all the methods as described in *Expression*.

5. Polynomial
   You should implement a *Polynomial* class, which also implements *Expression* interface.
   - ➢ Constructor: takes two arguments: *Variable* and a list of numbers to be used as coefficients to the variable. For example:
     ```
     x = Variable("x")
     poly1 = Polynomial(x, [3,-4,2])
     print(poly1)
     ```
     The output will be:
     (3x^-4x+2)
   - ➢ Do not forget to complete all the methods as described in *Expression*.
   - ➢ `NR_evaluate(assgms:Assignments, epsilon=0.0001, times=100)` - This method finds a root of the polynomial. Any root is OK. The method employs the Newton-Raphson method. You should use the symbolic_derivative, when you run NR. 'epsilon' is the accuracy (required distance from the root), and 'times' is the maximum number of iterations. In case the function did not converge to the desired accuracy in 'times' number of iterations, the function should raise an error.

❖ **Do not forget to fully document your code**

Pay Attention:

- Look carefully at the example table down below, it sums up all you need about the proper way to implement your program's output.
- The output of your program is checked using a simple output comparison. Make sure your `__repr__()` method is properly working.
- We will also check exceptions 'raised' according to the above description.
- No empty list will be given to Polynomial constructor and only non-empty float lists will be given.
- When using operators make sure the arguments are valid.
- Make sure not to change any of the method names and/or arguments, that includes not to add any new arguments that did not appear in the function definition in the interface.
- You should submit a single file <YOUR_ID>.py, replace YOUR_ID with your id number.
- Do not ZIP/RAR your assignment.
- some instructions for symbolic differentiation:

$\boldsymbol{f(x,y) = 2x^3 + y^3 + 4}$ $\boldsymbol{f'(x) = 6x^2}$
$\boldsymbol{f(x) = h\big(g(x)\big)}$ $\boldsymbol{f'(x) = h'\big(g(x)\big) \cdot g'(x)}$

$$\boxed{\begin{aligned}&\boldsymbol{f(x) = h\big(g(x)\big)}\\&\boldsymbol{f'(x) = h'\big(g(x)\big)\cdot g'(x)}\end{aligned}}$$

Example table

`#--- variables and assignments`	
`x = VariableExpression("x")`	
`y = VariableExpression("y")`	
`z = VariableExpression("z")`	
`print(x == y)`	`False`
`print(x == x)`	`True`
`ass1 = ValueAssignment(x, 10)`	
`ass2 = ValueAssignment(y, 20)`	
`sda =`	
`SimpleDictionaryAssignments()`	
`sda += ass1`	
`sda += ass2`	
`print(ass1)`	`x=10`
`print(ass2)`	`y=20`
`print(ass1)`	`x=10`
`print(ass2)`	`y=20`
`print(x)`	`x`
`print(x.evaluate(sda))`	`10`
`print(y)`	`y`
`print(y.evaluate(sda))`	`20`
`try:`	
`    print(z.evaluate(sda))`	
`except:`	
`    print("no assignment for`	`no assignment for all variables`
`all variables")`	
`#--- constants ---`	
`print("--- cons ---")`	`--- cons ---`
`con0 = Constant(0.0)`	
`con1 = Constant(1.0)`	
`con2 = Constant(2.0)`	
`print(con0)`	`0.0`
`print(con0.evaluate(sda))`	`0.0`
`print(con0.evaluate())`	`0.0`
`print(con0 == con1)`	`False`
`print(con0 == x)`	`False`
`print(con0 == con0)`	`True`
`print(con1)`	`1.0`
`print(con2)`	`2.0`
`#--- add ---`	
`print("--- add ---")`	`--- add ---`
`add = x + y`	
`print(add)`	`(x+y)`
`print(add.evaluate(sda))`	`30`
`print(add.derivative(x))`	`(1.0+0.0)`
`print(add.derivative(y))`	`(0.0+1.0)`
`print(add.derivative(z))`	`(0.0+0.0)`
`print(add.derivative(x).evaluat`	
`e(sda))`	`1.0`
`print(add == x)`	`False`
`#--- sub ---`	
`print("--- sub ---")`	`--- sub ---`
`sub = x-y`	
`sub2 = x - z + y`	
`print(sub)`	`(x-y)`
`print(sub.evaluate(sda))`	`-10`
`print(sub.derivative(x))`	`(1.0-0.0)`
`print(sub.derivative(y))`	`(0.0-1.0)`
`print(sub.derivative(z))`	`(0.0-0.0)`
`print(sub2 == sub)`	`False`
`try:`	

```
    print(sub2.evaluate(sda))
except:
    print("no assignment for all variables")
```

```
no assignment for all variables
```

```
#--- mul 1 ---
print("--- mul 1 ---")
mul1 = x * y
mul2 = z * y
print(mul1)
print(mul1.evaluate(sda))
print(mul1.derivative(x))
print(mul1.derivative(y))
print(mul1.derivative(z))
print(mul1.derivative(x).evaluate(sda))
print(mul2)
try:
    print(mul2.evaluate(sda))
except:
    print("no assignment for all variables")
```

```
--- mul 1 ---
(x*y)
200
((1.0*y)+(x*0.0))
((0.0*y)+(x*1.0))
((0.0*y)+(x*0.0))
20.0
(z*y)
no assignment for all variables
```

```
#--- comp 2 ---
print("--- comp 2 ---")
comp2 = add * sub
print(comp2)
print(comp2.evaluate(sda))
print(comp2.derivative(x))
print(comp2.derivative(x).evaluate(sda))
print(comp2.derivative(x).derivative(x))
```

```
--- comp 2 ---
((x+y)*(x-y))
-300
(((1.0+0.0)*(x-y))+((x+y)*(1.0-0.0)))
20.0
((((0.0+0.0)*(x-y))+((1.0+0.0)*(1.0-0.0)))+(((1.0+0.0)*(1.0-0.0))+((x+y)*(0.0-0.0))))
```

```
#--- power 1 ---
print("--- power 1 ---")
pow1 = add ** 3
print(pow1)
print(pow1.evaluate(sda))
print(pow1.derivative(x))
print(pow1.derivative(x).evaluate(sda))
```

```
--- power 1 ---
((x+y)^3)
27000
((3*((x+y)^2))*(1.0+0.0))
2700.0
```

```
#--- Poly 1 ---
print("--- Poly 1 ---")
print(Polynomial(x, [12,-8,-1]))
print(Polynomial(x, [0,0,-1]))
print(Polynomial(x, [-2,0,0]))
pol = Polynomial(x, [12,8,1])
print(pol)

print(pol.derivative(x))
print(pol.derivative(y))
v = pol.NR_evaluate(ValueAssignment(x,0.5), 0.01, 1000)
print(v)
print(pol.evaluate(sda))
```

```
--- Poly 1 ---
(-1x^2-8x+12)
(-1x^2)
(-2x)
(1x^2+8x+12)
(2x+8)
0.0
-1.9978149846503683
192.0
```

```
#--- Complex 1 ---
complex = ((x+y)*(y+x))**3.0
print(complex)
print(complex.derivative(x))
```

```
(((x+y)*(y+x))^3.0)
((3.0*(((x+y)*(y+x))^2.0))*(((1.0+0.0)*(y+x))+((x+y)*(0.0+1.0))))
```