Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
NAME = ""
COLLABORATORS = ""
```

Homework 10: Expressions as Classes

About This Homework

The homework consists of 9 questions, for a total of 69 points.

The instructions for working on homework assignments are available on Canvas; as a summary:

- Write your code only where indicated via `#YOUR CODE HERE`. If you write code in other places, it will be discarded during grading.
- Do not add/remove cells.
- The tests are implemented with `assert` statements: if they fail, you will see an error (a Python exception). If you see no error, you can assume that they pass.

Once you are done working on it, you can download the .ipynb and submit it to this form. This homework is due at **11:59pm on Thursday 12 May 2022**.

We will now describe a more sophisticated representation for an expression, based on a hierarchy of classes. The class **Expr** is the generic class denoting an expression. It is an *abstract* class: only its subclasses will be instantiated. For every operator, such as `+`, there will be a subclass, such as `Plus`. Variables will correspond to a special subclass, called `V`. Numerical constants will be just represented by numbers, and not by subclasses of `Expr`.

The `Expr` class

The `Expr` class implements the various methods used to emulate numerical types, such as `__add__`, `__sub__`, and so forth. In this way, we can build an expression simply by writing

```
x = V()
x * 3 / 2
```

In the above, `V()` creates a variable (an object of class `V`), and assigns it to `x`. Then,

```
x * 3
```

will also be an expression, composed of a multiplication node, with children the variable in `x`, and `3`. The multiplication node is implemented via a subclass `Multiply` of `Expr`. To make this work, we will define the `__mul__` method of `Expr` so that it produces a `Multiply` object, with as children the two operands being multiplied.

Similarly,

```
x * 3 / 2
```

will create an object of class `Divide`, with as left child the `Multiply` node for `x * 3`, and as right child, the number 2. Thus, the `__truediv__` method of `Expr` will create a `Divide` node.

Expressions can be evaluated. You can assign a value to variables either when you create them:

```
x = V(value=2)
e = x * 3 / 2
e.eval()
```

which yields `3`. Or you can assign a value to a variable later:

```
x = V()
e = x * 3 / 2

x.assign(3)
e.eval()
```

which again yields `3`.

Benefits of class-based representation

Compared with the representation of expressions seen in the previous chapter, this class-based representation offers several advantages.

First, we can build expressions in a natural way as shown above, via the over-riding of the usual arithmetic operators.

Second, if we introduce a new operator, all we need is provide the implementation of the new operator: we do not need to modify the shared code that traverses the tree, and add one more case to a long case-analysis. In other words, the code is far more modular. This may seem a small point, but if one were to extend the representation of expressions to involve tensors (matrices) and operations on tensors, as is done in the symbolic representation of expressions used in machine-learning, the number of operators could easily grow to a hundred or more, making a modular approach the only reasonable one.

Third, it becomes possible to attach methods to the expression objects, and as we will see in the next chapter, this can be very useful to implement machine learning frameworks. But later about that.

Defining an expression class

We define an abstract class `Expr`, representing a generic expression. This generic class has as subclasses the classes that represent the various operators, such as `Plus`, `Minus`, `Multiply`, as well as the variable class `V`. Here is our basic implementation.

In [2]:
```
class Expr(object):
    """Abstract class representing expressions"""

    name = "expr" # Not used, but just to define it.

    def __init__(self, *args):
        """An object is created by passing to the constructor the children"""
        self.children = args
        self.value = None # The value of the expression
        self.child_values = None # The values of the children; useful to have

    def eval(self):
        """Evaluates the expression."""
        # First, we evaluate the children.
        self.child_values = [c.eval() if isinstance(c, Expr) else c
                             for c in self.children]
        # Then, we evaluate the expression itself.
        self.value = self.op(*self.child_values)
        return self.value

    def op(self):
        """This operator must be implemented in subclasses; it should
        compute self.value from self.values, thus implementing the
        operator at the expression node."""
        raise NotImplementedError()

    def __repr__(self):
        """Represents the expression in a somewhat readable way."""
        if len(self.children) == 1:
            # Unary operators
            return "({}{})".format(self.__class__.name, self.children[0])
        elif len(self.children) == 2:
```

```
            return "({} {} {})".format(
                self.children[0], self.__class__.name, self.children[1]
            )
        # Catch-all.
        return "{}({})".format(self.__class__.__name__,
                               ', '.join(repr(c) for c in self.children))

    # Expression constructors

    def __add__(self, other):
        return Plus(self, other)

    def __radd__(self, other):
        return Plus(self, other)

    def __sub__(self, other):
        return Minus(self, other)

    def __rsub__(self, other):
        return Minus(other, self)

    def __mul__(self, other):
        return Multiply(self, other)

    def __rmul__(self, other):
        return Multiply(other, self)

    def __truediv__(self, other):
        return Divide(self, other)

    def __rtruediv__(self, other):
        return Divide(other, self)

    def __neg__(self):
        return Negative(self)
```

Variables are created specifying a name, and an initial value. If nothing is specified, variables have random initial values. You can assign a value to a variable using the `assign` method. The `eval` method of a variable simply returns its value.

In [3]:
```
import random
import string

class V(Expr):
    """Variable."""

    def __init__(self, value=None):
        super().__init__()
        self.children = []
        self.value = random.gauss(0, 1) if value is None else value
        self.name = ''.join(
            random.choices(string.ascii_letters + string.digits, k=3))

    def eval(self):
        return self.value
```

```
    def assign(self, value):
        self.value = value

    def __repr__(self):
        return "V({}, value={})".format(self.name, self.value)
```

Here are the constructors for the other operators; for them, we just need to provide an implementation for `op`, since all the rest is inherited from `Expr`. We actually also define a name, as a class attribute, that we use in the representation method.

In [4]:
```
class Plus(Expr):
    name = "+"
    def op(self, x, y):
        return x + y

class Minus(Expr):
    name = "-"
    def op(self, x, y):
        return x - y

class Multiply(Expr):
    name = "*"
    def op(self, x, y):
        return x * y

class Divide(Expr):
    name = "/"
    def op(self, x, y):
        return x / y

class Negative(Expr):
    name = "-"
    def op(self, x):
        return -x
```

We can build and evaluate expressions quite simply.

In [5]:
```
e = V(2) + 3
print(e)
print(e.eval())
```

```
(V(rDj, value=2) + 3)
5
```

In [6]:
```
e = (V() + V(2)) * (2 + V(1))
print(e)
print(e.eval())
```

```
((V(Ei9, value=1.9639207820415083) + V(iS6, value=2)) * (V(dBa, value=1) + 2))
11.891762346124525
```

If we want to be able to assign values to variables, or refer to them in our code later, we need to assign our variable objects to Python variables:

In [7]:
```
x = V()
y = V()
e = x + y

print(e.eval()) # This uses the initial random values.

x.assign(2)
y.assign(3)
print(e.eval())
```

```
1.8226158351059087
5
```

Defining Expression Equality

If we test equality between expressions, we are in for a surprise.

In [8]:
```
x = V()
e1 = x + 4
e2 = x + 4
e1 == e2
```

Out[8]:
```
False
```

Why is the result False?

Python knows how to compare objects that belong to its own types. So you can do comparisons between strings, numbers, tuples, and more, and it all works as expected. This is why we could check equality of expressions represented as trees: those expression trees are composed entirely of standard Python types, namely, strings, numbers, and tuples.

However, `Expr`, `V`, etc, are classes we defined, and Python has no idea of what it means for objects of user-defined classes to be equal.
In this case, Python defaults to considering equal two objects if they are the *same* object. The two expressions *e1* and *e2* above are not the same object: they are two distinct objects, which just happen to represent the same expression.

If we want to have a notion of expression equality that represents our idea that "two expression objects are equal if they represent the same expression", we need to define equality ourselves. This can be easily done, by defining an _ _ eq _ _ method. This method has the form:

```
def __eq__(self, other):
    ...
    return <True/False>
```

Here, self is the object on which the method is called, and other is another object -- any other object. Our job is to define when the object self is equal to the object other. This can

be easily done; using again our way of adding methods to existing classes, we write:

In [9]:
```
def expr_eq(self, other):
    if isinstance(other, Expr):
        # The operators have to be the same
        if self.__class__ != other.__class__:
            return False
        # and their corresponding children need to be equal
        if len(self.children) != len(other.children):
            return False
        for c1, c2 in zip(self.children, other.children):
            if c1 != c2: return False
        return True
    else:
        return False

Expr.__eq__ = expr_eq
```

If we did not define equality for variables, two variables would be considered equal according to `Expr.__eq__`, since `V` is a subclass of `Expr`. This would yield non-intended consequences (all variables would be considered equal).

Thus, we define equality for variables to be the basic equality for objects: two variables are equal iff they are the same object. In the definition below, `object.__eq__` is this primitive notion of equality, defined over the class `object` of Python, which is the base class for all classes.

In [10]:
```
V.__eq__  = object.__eq__
```

In [11]:
```
x = V()
y = V()
z = x
print(x == y)
print(x == z)
```

```
False
True
```

Once expression equality is thus defined, we get the expected result when we compare expressions:

In [12]:
```
x = V()
e1 = x + 4
e2 = x + 4
e1 == e2
```

Out[12]:
```
True
```

Having to define equality "by hand" is very pedantic, but it does give us the flexibility of defining precisely what it means for two expressions to be equal.

Variable Occurrence

Now that we have expressions, let us play with them. First, as a warm-up exercise, let us write an `occurs` method for expressions, which checks if a given variable occurs in the expression. First we define it for a variable: of course, a variable occurs in itself only if the variable is the same as the one whose occurrence we are checking.

Question 1: Variable occurrence in variables

In [13]:
```
### Variable occurrence in a variable

def v_contains(self, var):
    """Returns True of var is the same as self, and False otherwise."""
    # YOUR CODE HERE
    if self == var: return True
    else: return False

V.__contains__ = v_contains
```

In [14]:
```
## Here you can also test your code.

x = V()
y = V()
print(x in x)
print(x in y)
```

```
True
False
```

In [15]:
```
### Tests for variable occurrence

x = V()
y = V()
assert x in x
assert not x in y
z = x
assert x in z
```

Question 2: Occurrence of a variable in an expression

Once we define occurrence of a variable in a variable, we can define occurrence in a variable in a general expression. Of course, a variable appears in an expression if it appears in some of its children.

In [16]:
```
### Occurrence of a variable in an expression

def expr_contains(self, var):
```

```
    # YOUR CODE HERE
    for v in self.children:
        if isinstance(v, Expr) and var in v.children: return True
        elif var in self.children: return True
    return False

Expr.__contains__ = expr_contains
```

In [17]:
```
## Here you can also test your code.

x = V()
y = V()
z = V()
e = x + (2 * y)

print(x in e)
print(y in e)
print(z in e)
```

```
True
True
False
```

In [18]:
```
## Tests for occurrence: 5 points.

x = V()
y = V()
z = V()
e = x + (2 * y)

assert x in e
assert y in e
assert z not in e
```

In [19]:
```
## Hidden tests for occurrence: 5 points.
```

Variable Substitution

Another fun thing we can do is substitute a variable with an expression. Suppose you define an expression:

```
x = V()
y = V()
e = (x + 1) * (y + 1)
```

Suppose you also have another expression:

```
z = V()
f = y + z
```

Then, you can replace all occurrences of variable `x` in `e` with expression `f`:

```
new_e = e.replace(x, f)
```

and `new_e` should be then equal to:

```
((x + z) + 1) * (y + 1)
```

Let us implement variable substitution. Let us begin by defining variable substitution for variables.

Question 3: Variable replacement for variables

In [20]:
```
### Variable replacement in variables

def v_replace(self, x, e):
    """If self is x, replaces all occurrences of x with e."""
    # YOUR CODE HERE
    if self.children == []:
        if self == x: self = e
    else:
        for i in range(len(self.children)):
            if self.children[i] == x: self.children[i] = e
    return self

V.replace = v_replace
```

In [21]:
```
## Here you can also test your code

x = V()
y = V()
z = V()
print(x == x.replace(x, x))
print(y == x.replace(x, y))
print(x == x.replace(y, z))
```

```
True
True
True
```

In [22]:
```
## Tests for variable replacement in variables. 2 points.
x = V()
y = V()
z = V()
assert x == x.replace(x, x)
assert y == x.replace(x, y)
assert x == x.replace(y, z)
assert x.replace(x, y).replace(y, z) == z
```

In [23]:
```
## Other tests for variable replacement. 2 points.

x = V()
y = V()
```

```
e = x.replace(x, y + 3 * x)

x.assign(2)
y.assign(3)
assert e.eval() == 9
```

In [24]:
```
## Hidden tests for variable replacement.  2 points.
```

We now define variable replacement for expressions. Consider a simple expression:

In [25]:
```
x = V()
y = V()
z = V()
e = x + y
```

Suppose we want to return `e.replace(x, z)`. The idea is:

- carry out the replacement for each child (via a recursive call, as usual),
- then return an expression built out of the replacements.

For instance, consider `e = x + y`. To compute

```
e.replace(x, z)
```

we first replace `x` with `z`, and then we return `Plus(z, y)`.

The problem is exactly in the last sentence. A `Plus` object, after carrying out the replacement in the children, should return a `Plus` object with the new children. Similarly, a `Minus` object should return a `Minus` object, a `Multiply` object should return a `Multiply` object, and so forth. If we implement this in the straightforward way, we need to add a `replace` method to all of these classes, so that they can return an object of the appropriate type. This is a lot of work.

Is there a better way? It turns out, yes. In an object,

```
self.__class__
```

is the class of the object. So if you want to return a new object of the same class, created say with arguments `x` and `y`, all you need to do is:

```
self.__class__(x, y)
```

In this way, if you are in a `Plus` object, `self.__class__` is `Plus`, and everything works. Using this idea, we, that is, you, can implement the replacement method directly for the `Expr` class.

Question 4: Replacement for expressions

In [26]:
```
### Replacement for expressions

def expr_replace(self, x, e):
    # YOUR CODE HERE
    if isinstance(self, int): return self
    elif not isinstance(self, int) and not isinstance(self, V):
        left = self.children[0]
        right = self.children[1]
        return self.__class__(expr_replace(left, x, e), expr_replace(right, x, e))
    else:
        if self == x: return e
    return self


Expr.replace = expr_replace
```

In [27]:
```
### Here you can debug your code.

x = V()
y = V()
z = V()

e = (1 + x) * (1 + y)
f = e.replace(x, x + z)
print(f)
```

```
(((V(VRz, value=2.158637507157798) + V(CqU, value=-0.9216474925875536)) + 1) * (V(4i
j, value=-1.1285546451470219) + 1))
```

In [28]:
```
### Tests for expression replacement. 10 points.

x = V()
y = V()
z = V()

e = (1 + x) * (1 + y)
f = e.replace(x, x + z)
assert f == (1 + (x + z)) * (1 + y)

x.assign(1)
y.assign(2)
z.assign(3)
assert e.eval() == 6
assert f.eval() == 15

e = (x + y) / (x - y)
f = e.replace(x, 2 * x).replace(y, 3 * y)
assert f.eval() == (2 + 6) / (2 - 6)
```

In [29]:
```
### Hidden tests for expression replacement. 10 points.
```

Expression Derivation

We will develop here a method `derivate` such that, for an expression `e`, the method call `e.derivate(x)` returns the derivative of the expression with respect to the variable `x`. As in the previous chapter, we use the following derivation formulas:

- For a constant c, $\partial c/\partial x = 0$.
- For a variable $y \neq x$, $\partial y/\partial x = 0$.
- $\partial x/\partial x = 1$.

For operators, we can use:

$$\frac{\partial}{\partial x}(f \pm g) = \frac{\partial f}{\partial x} \pm \frac{\partial g}{\partial x},$$
$$\frac{\partial}{\partial x}(f \cdot g) = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x},$$
$$\frac{\partial}{\partial x}\left(\frac{f}{g}\right) = \frac{\frac{\partial f}{\partial x} \cdot g - f \cdot \frac{\partial g}{\partial x}}{g^2}.$$

Let us begin with implementing derivation for variables.

Question 5. Derivation for a variable

In [30]:
```
### Derivation of variables

def v_derivate(self, x):
    # YOUR CODE HERE
    if self == x: return 1
    return 0

V.derivate = v_derivate
```

In [31]:
```
## Here you can debug your code.

x = V()
y = V()
print(x.derivate(x))
print(x.derivate(y))
```

```
1
0
```

In [32]:
```
## Tests for variable derivation

x = V()
y = V()
assert x.derivate(x) == 1
assert x.derivate(y) == 0
```

Derivation for expressions

This time, there is no clever trick to implement `derivate` as a method of `Expr`, because the derivative behaves in a different way for the different operators. Hence, we will need to implement `derivate` for each individual operator. We let you do it. There are two things to be careful about:

- Children of an operator may not be expressions; they can also be constants such as 2.3 or 4.1.
- If the *new* children of an expression are all numbers, return the numerical result of the expression rather than a symbolic expression. For example, do not return `Plus(1, 0)`; rather, just return `1`. Note that, if you put the derivatives of the two children in, say, `df` and `dg`, you can simplify automatically by returning `df + dg`, which will return an expression if one of `df` or `dg` is an expression, and a number otherwise.

We give you the solution for `Plus`, mainly because it might be difficult to believe that the solution is this simple.

In [33]:
```
def plus_derivate(self, x):
    f, g = self.children
    df = f.derivate(x) if isinstance(f, Expr) else 0
    dg = g.derivate(x) if isinstance(g, Expr) else 0
    return df + dg

Plus.derivate = plus_derivate
```

Question 6: Derivation for Minus

You can now work out the case for Minus, which is very similar.

In [34]:
```
### Derivative of Minus

def minus_derivate(self, x):
    # YOUR CODE HERE
    fx, gx = self.children
    if isinstance(fx, Expr): dfx = fx.derivate(x)
    else: dfx = 0
    if isinstance(gx, Expr): dgx = gx.derivate(x)
    else: dgx = 0
    return dfx - dgx

Minus.derivate = minus_derivate
```

In [35]:
```
### Here you can debug your code.

x = V()
y = V()
z = V()

e = x + y
e.derivate(y)
```

Out[35]: 1

In [36]:
```
### Tests for derivatives of Plus and Minus. 10 points.

x = V()
y = V()
z = V()

e = x + y
assert e.derivate(x) == 1
f = x - y
assert f.derivate(x) == 1
assert f.derivate(y) == -1

h = e + f
assert h.derivate(x) == 2

u = x + 4
assert u.derivate(x) == 1
v = 3 - y
assert v.derivate(x) == 0
assert v.derivate(y) == -1
```

Question 7: derivative of Negative

Since we are at it, let us take care of unary minus, or `Negative`.

In [37]:
```
### Derivative of Negative

def negative_derivate(self, x):
    # YOUR CODE HERE
    fx = self.children[0]
    if isinstance(self, Expr): dfx = fx.derivate(x)
    else: dfx = 0
    return -dfx

Negative.derivate = negative_derivate
```

In [38]:
```
### Here you can debug your code.

x = V()
y = V()
e = x + (-y)
e.derivate(y)
```

Out[38]: -1

In [39]:
```
### Tests for negative. 4 points.

x = V()
y = V()
e = -x
```

```
assert e.derivate(x) == -1
assert e.derivate(y) == 0
```

Now for multiplication and divisions. Be careful to use the formulas exactly as given, otherwise the result may not match. E.g. use

$$\frac{\partial}{\partial x}(f \cdot g) = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x}$$

and *not*, for instance, $\frac{\partial}{\partial x}(f \cdot g) = g \cdot \frac{\partial f}{\partial x} + f \cdot \frac{\partial g}{\partial x}$ (note the swap of the factors in the first term).

We would like you to return simplified expressions. You can again use the trick that, if you have two expressions `df` and `g` (or similar), then `df * g` will be an expression if one of `df` and `g` are expressions, and a number if both `df` and `g` are numbers.

Question 8: Derivation of multiplication

In [40]:
```
### Derivative of multiplication.

def multiply_derivate(self, x):
    # YOUR CODE HERE
    left, right = self.children
    if isinstance(left, int):
        return ((left * right.derivate(x)) + (x * 0))
    if isinstance(right, int):
        return ((left.derivate(x) * right) + (x * 0))
    else:
        return ((left.derivate(x) * right) + (left * right.derivate(x)))

Multiply.derivate = multiply_derivate
```

If you did the above right, it will be 4-5 lines wrong. If you wrote much more than that, please think at it again. The solution is quite simple; you can just trust that the operators `*` and `+` will build it.

In [41]:
```
## Here you can debug your code.

x = V()
e = x * x
de = e.derivate(x)
de
```

Out[41]:
```
((1 * V(x5S, value=1.2962715824513196)) + (V(x5S, value=1.2962715824513196) * 1))
```

In [42]:
```
## Tests for derivative of multiplication. 4 points.

x = V()
y = V(value=2)
```

```
e = x * y
# This is ugly.  It is.  We have not implemented 0, 1 simplifications.
assert e.derivate(x) == 1 * y + x * 0

# To remedy ugliness, we now test numerically.
f = x * x
x.assign(3)
assert f.derivate(x).eval() == 6, f.derivate(x).eval()
x.assign(4)
assert f.derivate(x).eval() == 8, f.derivate(x).eval()

h = 3 * x
assert h.derivate(x).eval() == 3
u = x * 3
assert u.derivate(x).eval() == 3
```

Question 9: Derivative of division

For division, the expression is:

$$\frac{\partial}{\partial x}\left(\frac{f}{g}\right) = \frac{\frac{\partial f}{\partial x} \cdot g - f \cdot \frac{\partial g}{\partial x}}{g^2}.$$

Note that you can obtain the denominator just by doing `g * g`, if `g` is the second child. Again, the correct solution is quite short.

In [45]:
```
### Derivative of division

def divide_derivate(self, x):
    # YOUR CODE HERE
    left, right = self.children
    squared = right * right
    if isinstance(left, int):
        return (((-left * right.derivate(x)) + (x * 0))/(squared))
    elif isinstance(right, int):
        return (((left.derivate(x) * right) + (x * 0))/(squared))
    else:
        return (((left.derivate(x) * right) - (left * right.derivate(x)))/(squared)

Divide.derivate = divide_derivate
```

In [46]:
```
## Here you can debug your code.

x = V()
y = V()
print("x:", x)
print("y:", y)

e = x / y

e.derivate(y)
```

```
x: V(Tiv, value=0.77662701250257)
y: V(iIP, value=0.032634097803542976)
```

Out[46]:
```
(((0 * V(iIP, value=0.032634097803542976)) - (V(Tiv, value=0.77662701250257) * 1))
/ (V(iIP, value=0.032634097803542976) * V(iIP, value=0.032634097803542976)))
```

In [47]:
```
### Tests for derivative of division. 10 points.

x = V()
y = V()
e = x / 2
f = e.derivate(x)
x.assign(3)
assert f.eval() == 1/2

g = 1 / x
assert g.derivate(x).eval() == - 1 / 9
assert g.derivate(y).eval() == 0
```

In [48]:
```
## Miscellaneous tests.  10 points.

x = V(value=2)
y = V(value=3)

f = (x + 1) * (y + 1) / (x - 1) * (y - 1)
df = f.derivate(x)

assert df.eval() == -16

x.assign(0.5)
y.assign(0.5)
assert df.eval() == 6
```

In [49]:
```
## Miscellaneous tests. 10 points.

x = V(value=0)

f = (3 * x * x + 2 * x + 4) / (5 * x * x - 7 * x + 12)

df = f.derivate(x)
assert abs(df.eval() - 0.3611) < 0.01
x.assign(1)
assert df.eval() == 0.53
x.assign(1000)
assert abs(df.eval()) < 0.001
x.assign(-0.5)
assert abs(df.eval() - 0.1) < 0.01
```

In [50]:
```
## Final hidden tests on everything. 10 points.
```