Calcul formel

oct. 09, 2019

1 Système de calcul formel

Note : Ce TD est a but purement pédagogique. Pour utiliser un système de calcul formel sous Python, la librairie `sympy` existe et fonctionnera bien mieux que ce que l'on va faire !

L'objectif de ce TD est de réaliser un système de calcul formel qui permettra de manipuler des expressions algébriques simples et de réaliser des opérations simples. Par exemple, on souhaite pouvoir effectuer

```
x = Symbol('x')
y = Symbol('y')

s = 2*x*y + sin(x)*y

print(s.diff(x)) # Dérivée par rapport à x
```

Chaque expression sera représentée par un arbre. Les feuilles de l'arbre seront soit les symboles soit les constantes numériques. Les noeuds seront des fonctions. Le nom de la classe du noeud désignera le type fonction. Les « enfants » du noeud seront les arguments de la fonction. Par exemple l'expression ci dessus correspondra à l'objet suivant

```
# sA : 2*x*y
sA = Prod(Prod(Number(2), Symbol('x')), Symbol('y'))
# sB : sin(x)*y
sB = Prod(Sin(Symbol('x')), Symbol('y'))

s = Sum(sA, sB)
```

2 Structure du programme

Voici la structure de base

```
class Expr(object):
    pass

class Node(Expr):
    pass
```

(suite sur la page suivante)

(suite de la page précédente)

```
class Leave(Expr):
    pass
```

Pour les feuilles

```
class Symbol(Leave):
    pass

class Number(Leave):
    pass
```

Ensuite on définit les fonctions

```
class Function(Node):
    """ Function with an arbitrary number of arguments """
    pass
```

Les opérateurs sont des fonctions comme les autres, mais elle seront simplement affiché différemment

```
class BinaryOperator(Function):
    pass

class Sum(BinaryOperator):
    pass
# Idem pour Sub, Div, Prod, Pow

class UnitaryOperator(Function):
    pass

class Neg(UnitaryOperateor):
    pass
```

Les fonction mathématiques, qui prennent un seul argument

```
class MathFunction(Function):
    pass

class Sin(MathFunction):
    pass
```

3 Questions

On va procéder étape par étape. Il sera plus facile de commencer par les feuilles avant d'écrire la structure globale.

1. Ecrire le `__init__` de la classe Symbol et Number

2. Ecire une méthode `display` sur ces classes afin de renvoyer une chaîne de caractère contenant le symbole ou le nombre
3. Ecrire le `__init__` de la class Sin ainsi que le display. Le display devra appeler le display de l'argument. Par exemple ceci devra fonctionner

```
>>> x = Symbol('x')
>>> Sin(x).display()
sin(x)
>>> Sin(Sin(x)).display()
sin(sin(x))
```

4. Généraliser le __init__ et le display de `Sin` afin de le mettre dans la class MathFunction. On rajoutera un attribut de classe à chaque sous classe de MathFunction

```
class Sin(MathFunction):
    funtion_name = 'sin'
```

5. Faite de même pour les opérateurs binaires. On pourra commencer par simplement le faire pour Sum, puis généraliser avec un attribut de classe

```
class Sum(BinaryOperator):
    operator_name = '+'
```

6. A ce stade quelque chose comme ceci devrait fonctionner

```
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> Sum(x, Sin(Prod(x, y)))
```

Rajouter les méthodes __add__, __mul__, etc à la classe Expr afin de pouvoir écrire :

```
>>> x + Sin(x*y)
```

7. Ecrire les méthodes `evaluate` afin de calculer la valeur numérique d'une expression. Cette méthode fonctionnera de la sorte :

```
>>> expr = x + Sin(x*y)
>>> expr.evaluate(x=1, y=3)
```

On aura donc le protocole suivant

```
def evaluate(self, **kwd):
    pass
```

Le dictionnaire kwd sera passé récursivement jusqu'aux feuilles et sera utilisé pour évaluer les symboles.

Les opérateurs binaires numériques sont définis dans le module `operator` et les fonctions dans le module `math`. Afin de factoriser le code, on rajoutera donc simplement un attribut de classe du type `operator_function = operator.add` pour les opérateurs binaires et `math_function = math.sin` pour les fonctions.

8. Maintenant que vous avez compris le principe, il devrait être facile d'écrire une méthode `diff` qui effectue la dérivée par rapport à une variable !
9. Reste à simplifier les expressions. Une technique consiste à créer des règles de simplifications sous forme de méthode que l'on regroupe ensuite dans une liste

```
class Sum(BinaryOperator):
    operator_name = '+'
    operator_function = operator.add

    def simplication_de_deux_nombres(self):
        if isinstance(self.arg1, Number) and
                        isinstance(self.arg2, Number):
            return Number(self.arg1.value + self.arg2.value)

    def simplication_addition_avec_zero(self):
        pass

    liste_simplication = ['simplication_de_deux_nombres',
                    'simplication_addition_avec_zero']
```

Ensuite, il faut réussir à appeler correctement et de façon recursive ces méthodes...

10. Pour l'affichage des opérateurs binaires, les règles de priorité peuvent être utilisées pour éviter de mettre trop de parenthèses. Par exemple, dans le cas `a*(b+c)`, la multiplication appelle le display de l'addition. Comme elle est prioritaire, l'addition va renvoyer le résulat avec des parenthèses. Dans le cas inverse `a + b*c`, c'est inutile. Il faut donc que le display d'un opérateur passe sa priorité à ces enfants lors de l'appel de display.