Programmation orientée objet

Version 2022

mars 11, 2022

1 Vecteur

Créer une classe Vecteur3D. Chaque vecteur aura 3 attributs : x, y, z

- Écrire une méthode norme qui renvoie la norme.
- Écrire la méthode `__add__` pour faire la somme entre deux veteurs
- Écrire la méthode `__mul__` pour faire soit le produit par un scalaire ($2\vec{u}$) ou le produit scalaire ($\vec{u} \cdot \vec{v}$).

2 Bibliographie

Un livre est décrit par son titre, auteur et année de publication (pour faire les choses simplements). Ecrire une classe `Livre` qui enregistre ces informations. Ecrire la méthode `__repr__` et `__str__`.

Une bibliographe est une liste de livre. Écrire la classe `Bibliographie` qui enregistre une liste de livre (on stockera la liste de livre sous forme d'une liste qui sera un attribut de la bibliographie).

L'objectif final est de pouvoir faire ceci

```
livre1 = Livre("A very nice book", "F. Dupont", 2014)
livre2 = Livre("A very smart book", "A. Einstein", 1923)
livre3 = Livre("A very stupid comic", "D. Duck", 1937)

bibliographie = Bibliographie([book1, book2, book3])
```

Maintenant que tout est fait sous forme d'objet, on peut imaginer écrire plusieurs méthode :

- Écrire une méthode `save_to_json` pour sauvegarder la bibliographie et une fonction (ou une méthode de class) `load_from_json`.
- Écrire une méthode `filter_by_year` qui fait une nouvelle bibliographie ne contenant que les livres d'une année.
- Écrire une méthode `to_latex` ou `to_html` qui formate correctement la bibliographie.

En latex cela devra donner

```
\begin{thebibliography}{9}
\bibitem{Dupont2014}
F. Dupont (2014) \emph{A very nice book}

\bibitem{Einstein1923}
A. Einstein (1923) \emph{A very smart book}

\bibitem{Duck1937}
D. Duck (1937) \emph{A very stupid comic}
\end{thebibliography}
```

Et en HTML

```
<table>
    <thead>
        <tr> <th>Auteur</th><th>Titre</th><th>Année</th></tr>
    </thead>
    <tbody>
       <tr><td>F. Dupont</td><td>2014</td><td>A very nice book</td></tr>
       <tr><td>A. Einstein</td><td>1923</td><td>A very smart book</td></
↪tr>
       <tr><td>D. Duck</td><td>1937</td><td>A very stupid comic</td></tr>
    </tbody>
</table>
```

Remarque : si un objet possède une méthode `_repr_html_`, alors le jupyter notebook utilsera automatiquement la représentation en HTML.

3 Impedance complexe d'un circuit bibolaire

Objectif : faire comprendre à Python ce circuit pour pouvoir ensuite faire des calculs. Ici, on demandera de calculer l'impédance complexe à une fréquence donnée.

Il y a plusieurs objets de nature différente donc de classe différente (résistance, condensateur, circuit parallèle, ...). Mais ces objets sont tous des circuits bibolaires. Tous ces objets devront mettre en oeuvre un méthode pour calculer leur impédance à une fréquence donnée.

Code final en Python (objectif à atteindre pour que l'objet soit le plus simple à utiliser)

```
R1 = Resistor(10)
R2 = Resistor(5)
L1 = Inductor(15E-6)
C1 = Capacitor(10E-6)

circuit = R2 + (L1|R1|C1)
```

(suite sur la page suivante)

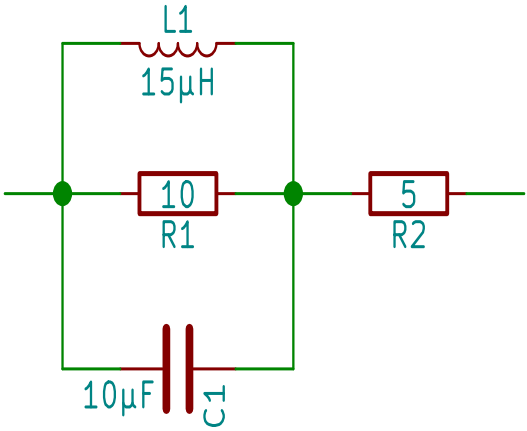

Fig. 1 – Exemple de circuit

(suite de la page précédente)

```
print(circuit.impedance(50))
```

Ici, le | représente une structure parallèle et le + une structure série.

La structure de classe sera la suivante

```
class BipolarCircuit(object):
    pass

class Combination(BipolarCircuit):
    pass

class Serial(Combination):
    pass

# idem for parallel

class Device(BipolarCircuit):
    pass

class Resistor(Device):
    pass
```

4 Système de calcul formel

Note : Cet exercice 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 à un ou plusieurs argument (sinus, somme, opposé, ...). Le nom de la classe du noeud désignera la 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)
```

4.1 Structure du programme

Voici la structure de base

```
class Expr(object):
    pass

class Node(Expr):
    pass

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 affichées 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
```

4.2 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. Ecrire 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. Faire 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. Implémenter ce principe.