# Expressions_Sympy

May 8, 2019

# 1 Les expressions de la bibliothèque Sympy

# 2 Syntaxe, arbres syntaxiques, dérivation symbolique

Marc Lorenzi
8 mai 2019

```
In [1]: from sympy import *
        import matplotlib.pyplot as plt
        init_printing()
```

```
In [2]: plt.rcParams['figure.figsize'] = (16, 6)
```

SymPy est une bibliothèque Python qui permet de faire du calcul symbolique, c'est à dire du calcul **exact**. Dans ce notebook nous allons parler des objets sans doute les plus importants définis par cette bibliothèque : les **expressions**.

Sympy définit un grand nombre de classes et de fonctions, nous n'aborderons dans ce notebook qu'une toute petite partie. Pour tout savoir, rien ne vaut la documentation officielle qui contient entre autres un excellent tutoriel. Vous la trouverez à l'adresse https://docs.sympy.org

## 2.1 1. Notion d'expression

### 2.1.1 1.1 Symboles

Pour un mathématicien, une "expression" est quelque chose du genre $x + y$, ou $3x + 2y + z \sin^2 z$, où $x, y, z$ sont des "variables". Avant de définir précisément ce qu'est une expression, parlons des **symboles**.

```
In [3]: x = Symbol('x')
```

La ligne ci-dessus vient de définir une nouvelle **variable Python** dont le nom est $x$ (il s'agit du $x$ à gauche du signe =). Après affectation, cette variable contient un objet du type "symbole", et le nom de ce symbole est $x$ (le 'x' entre parenthèses à droite du signe =). Un petit test ?

```
In [4]: x
```

```
Out[4]:
```
$x$

Évidemment ceci n'est pas très spectaculaire. Tentons autre chose :

```
In [5]: x = Symbol('y')
        x
```

Out[5]:

$y$

Eh oui, maintenant la variable $x$ contient un symbole, mais le nom de ce symbole est $y$. Ce n'est pas très judicieux mais c'est faisable. Faisons-nous pour la suite un petit stock de symboles. La fonction `symbols` (sans majuscule) permet de définir plusieurs symboles à la fois.

```
In [6]: x, y, z, t = symbols('x y z t')
```

```
In [7]: x, y, z, t
```

Out[7]:

$(x, y, z, t)$

Histoire d'insister sur le fait que je ne parlerai que d'une petite partie des possibilités de SymPy, voici les **méthodes** que possède l'objet $x$.

```
In [8]: print(dir(x))
```

```
['_Symbol__xnew_cached_', '__abs__', '__add__', '__and__', '__class__', '__complex__', '__delatt
```

Effectivement :-) ... Par exemple :

```
In [9]: x.is_Symbol
```

Out[9]: `True`

**A retenir** : Un **symbole** est créé par la fonction `Symbol`. On le stocke dans une **variable Python**. L'usage est (la plupart du temps) de donner le même nom à la variable et au symbole. On peut aussi créer plusieurs symboles à la fois avec la fonction `symbols`.

**Remarque** : lorsqu'on crée un symbole, on peut y adjoindre des hypothèses sur ce symbole. Par exemple, c'est un entier, un réel positif, etc. Je n'en parlerai pas ici, consultez la documentation.

### 2.1.2 1.2 Premières expressions

Voici notre première expression autre qu'un symbole.

```
In [10]: expr = x ** 2 + sqrt(y)
```

Affichons `expr`. Il y a au moins trois façons de voir cette expression.

D'abord la vision "Python". On utilise la fonction `print`.

```
In [11]: print(expr)
```

```
x**2 + sqrt(y)
```

Deuxième façon, taper juste `expr`.

```
In [12]: expr
```

```
Out[12]:
```
$x^2 + \sqrt{y}$

Cette fois-ci l'affichage obtenu est plus conforme à nos habitudes mathématiques. Il existe une troisième méthode qui nous sera fort utile : la fonction `srepr` renvoie une chaîne de caractères.

```
In [13]: print(srepr(expr))
```

```
Add(Pow(Symbol('x'), Integer(2)), Pow(Symbol('y'), Rational(1, 2)))
```

Qu'est-ce que c'est que ça ?

Nous voyons là quelque chose qui se rapproche beaucoup de la structure interne des expressions SymPy. `Pow`, `Add`, `Integer`, etc, sont des fonctions définies par `sympy` (en fait des constructeurs de classe). Et vous pouvez si le coeur vous en dit les utiliser pour créér des expressions :

```
In [14]: Mul(Add(x, y), Add(z, t))
```

```
Out[14]:
```
$(t + z)(x + y)$

Cela revient au même que

```
In [15]: (x + y) * (z + t)
```

```
Out[15]:
```
$(t + z)(x + y)$

Remarquez que ce qui est affiché ne correspond pas **exactement** à ce que nous avons entré. Par défaut, `sympy` considère que l'addition et la multiplication sont commutatives. On ne peut pas être certain de la façon dont `sympy` va traiter l'expression que l'on tape. Sympy effectue également certaines simplifications "triviales" :

```
In [16]: x + x
```

```
Out[16]:
```
$2x$

```
In [17]: y * 1
```

```
Out[17]:
```
$y$

### 2.1.3 1.3 Les nombres, les constantes

Certaines expressions SymPy sont juste des nombres. Si vous voulez créer une expression constante, égale à 2 par exemple, il vous faut entrer `Integer(2)`. Pourquoi ? Tentons une expérience.

```
In [18]: 2 / 3
```

Out[18]:

0.6666666666666666

Mais enfin, SymPy est une bibliothèque de calcul symbolique, il devrait donc nous renvoyer la fraction $\frac{2}{3}$ ? Certes, mais où voyez-vous SymPy dans la cellule ci-dessus ? SymPy n'a pas pris le contrôle de notre machine, ce que nous avons entré c'est une ligne de **Python**. Et la réponse est une **réponse de Python**. En revanche :

```
In [19]: Integer(2) / 3
```

Out[19]:

$$\frac{2}{3}$$

Cette fois-ci, nous avons calculé le quotient de **l'entier Sympy** 2 par l'entier 3, et SymPy nous renvoie la valeur "espérée" : les objets de la classe Integer savent comment se diviser par 3 :-).

Le rationnel $\frac{2}{3}$ renvoyé ci-dessus est donc une expression. Quel genre d'expression, précisément ?

```
In [20]: srepr(Integer(2) / 3)
```

```
Out[20]: 'Rational(2, 3)'
```

Notez le constructeur `Rational`, vous pouvez évidemment vous en servir pour manipuler des rationnels.

```
In [21]: Rational(1, 3) + Rational(5, 7)
```

Out[21]:

$$\frac{22}{21}$$

### 2.1.4 1.4 Classes, objets, champs, méthodes

Python est un langage **orienté objets**. Dans un tel langage on définit des **classes** qui sont, sans entrer dans les détails, des "types de données". Définir une classe, c'est définir le comportement des **objets** de la classe, que l'on appelle aussi les **instances** de la classe.

- Classe = type de données
- instance de la classe = objet ayant le type en question

Un objet possède

- des **champs**, qui sont des valeurs associées à l'objet.
- des **méthodes**, qui sont des fonctions qui permettent à l'objet de se modifier, d'interagir avec d'autres objets, etc.

Par exemple, si vous entrez

```
In [22]: w = 3 + 1j * 2
```

vous fabriquez un objet de la classe `complex` et vous l'affectez à la variable $w$.

```
In [23]: w.__class__
```

```
Out[23]: complex
```

Cet objet possède deux champs :

```
In [24]: print(w.real, w.imag)
```

```
3.0 2.0
```

L'objet $z$ possède un certain nombre de méthodes. Lesquelles ? La fonction malnommée `dir` nous le dit :

```
In [25]: print(dir(w))
```

```
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__
```

Par exemple,

```
In [26]: w.__abs__()
```

```
Out[26]:
```

3.605551275463989

Lorsque vous tapez `abs(w)`, Python appelle la méthode `__abs__` de l'objet $w$.

Encore un petit exemple, celui des listes.

```
In [27]: s = [1, 2, 3]
         print(dir(s))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__
```

Vous avez sûrement reconnu ci-dessus un certain nombre des méthodes de la liste $s$, comme `append`, `reverse` ou `sort` ...

Comme ce notebook n'est pas un cours de programmation objet, je n'en dirai pas plus. Je ferai tout au plus quelques remarques par-ci par-là.

### 2.1.5 1.5 Soustraction et division

La soustraction des expressions c'est sûrement `Sub` ? Et la division c'est `Div` ? **Pas du tout**. La soustraction et la division cela n'existe pas, ouvrez votre cours de maths.

```
In [28]: srepr(x - y)
```

```
Out[28]: "Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```

Pour `sympy`, $x - y$ c'est $x + (-1) \times y$, ce en quoi il n'a pas tort.

```
In [29]: srepr(x / y)
```

```
Out[29]: "Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))"
```

Et $\frac{x}{y}$ c'est $x \times y^{-1}$.

Nous maîtrisons donc les 5 opérations de l'arithmétique :

- $+$ c'est la fonction `Add`.
- $\times$ c'est la fonction `Mul`.
- L'exponentiation c'est la fonction `Pow`.
- $-$ et $/$ sont transformées en sommes, produits et puissances.

### 2.1.6 1.6 Des expressions plus compliquées

Contrairement au dicton, quand on peut faire simple on peut faire compliqué :-). SymPy connaît les "fonctions" usuelles (remarquez les guillemets, on y revient plus loin), sinus, cosinus, exponentielle, etc. Et aussi des fonctions moins usuelles (pour un étudiant de prépa) comme la fonction $\Gamma$, la fonction $\zeta$ ou les fonctions hypergéométriques.

Voici une expression "compliquée".

```
In [30]: expr = (x + y ** 2) * sin(1 / z) + t * atan(t / 2)
         expr
```

Out[30]:

$$t \operatorname{atan}\left(\frac{t}{2}\right) + \left(x + y^2\right) \sin\left(\frac{1}{z}\right)$$

Quelle est sa représentation en tant qu'objet SymPy ? Si vous comprenez la réponse vous avez fait un grand pas.

```
In [31]: srepr(expr)
```

```
Out[31]: "Add(Mul(Symbol('t'), atan(Mul(Rational(1, 2), Symbol('t')))), Mul(Add(Symbol('x'), Pow
```

Voici une fonction qui renvoie des expressions compliquées.

```
In [32]: def compliquee(n, x):
             expr = x
             for k in range(n):
                 expr = sqrt (1 + expr)
             return expr
```

```
In [33]: compliquee(10, x)
```

Out[33]:

$$\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{x+1}}}}}}}}}}$$

```
In [34]: print(compliquee(10, x))
```

```
sqrt(sqrt(sqrt(sqrt(sqrt(sqrt(sqrt(sqrt(sqrt(sqrt(x + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1)
```

```
In [35]: print(srepr(compliquee(10, x)))
```

```
Pow(Add(Pow(Add(Pow(Add(Pow(Add(Pow(Add(Pow(Add(Pow(Add(Pow(Add(Pow(Add(Symbol('x'), Int
```

### 2.1.7 1.7 Finalement, c'est quoi une expression ?

Les programmeurs de SymPy ont défini dans cette bibliothèque une **classe** `Expr`, qui est la classe **ancêtre** de toutes les expressions. Les **objets** qui sont des **instances** de la classe `Expr` **sont** les expressions. SymPy définit en fait toute une **hiérarchie** de classes, dont celles qui nous intéressent **descendent** de la classe `Expr`. Par exemple, les classes `Mul`, `Add`, `Symbol`, `Integral`, etc. Tout objet de la classe `Add` est aussi par **héritage** un objet de la classe `Expr`, et est donc une expression de plein droit. Et de même pour toutes les classes qui héritent de la classe `Expr`.

Si nous voulons rester à un niveau un peu moins concret que l'implémentation réelle des expressions, nous pouvons dire qu'une expression est :

- un Symbole

ou

- Un Entier

ou

- Un Rationnel

ou

- Add(expression, Expression, . . . , Expression)

ou

- Mul(Expression, Expression, . . . , Expression)

ou

- Pow(Expression, Expression)

ou

- Bien d'autres choses, qu'il est hors de question d'examiner ici de façon exhaustive.

Voici les **classes filles** de la classe `Expr`, c'est à les classes qui sont ses héritières directes.

```
In [36]: print(Expr.__subclasses__())
```

```
[<class 'sympy.core.expr.AtomicExpr'>, <class 'sympy.core.expr.UnevaluatedExpr'>, <class 'sympy.
```

Si vous voulez avoir une idée de **toutes** les classes qui **héritent** de la classe `Expr`, c'est à dire les sous-classes, les sous-sous-classes, etc., voici une fonction qui permet de les obtenir. Elle effectue un **parcours de graphe** en appelant récursivement la méthode `__subclasses__`, que possède toute classe Python.

```
In [37]: def heritieres(classe):
             sous_classes = set()
             s = [classe]
             while s != []:
                 c = s.pop()
                 for enfant in c.__subclasses__():
                     if enfant not in sous_classes:
                         sous_classes.add(enfant)
                         s.append(enfant)
             return sous_classes

In [38]: print(heritieres(Expr))
```

```
{<class 'sympy.core.numbers.Float'>, <class 'sympy.matrices.expressions.hadamard.HadamardProduct
```

```
In [39]: expr = exp(x)
         print(expr.__class__)
```

```
exp
```

Et voici les champs et méthodes de `expr`.

```
In [40]: print(dir(expr))
```

```
['__abs__', '__add__', '__class__', '__complex__', '__delattr__', '__dict__', '__dir__', '__div_
```

**Exercice** : Quelles sont les classes qui héritent de la classe `Symbol` ? De la classe `Integer` ?

Au cas où vous vous poseriez la question, voici une fonction qui renvoie les **ancêtres** d'une classe. Toute classe possède un champ `__bases__` qui est la liste de ses classes parentes (Python autorisant **l'héritage multiple**, une classe peut avoir plusieurs mères :-)).

```
In [41]: def ancetres(classe):
             sur_classes = set()
             s = [classe]
             while s != []:
                 c = s.pop()
                 for parent in c.__bases__:
                     if parent not in sur_classes:
                         sur_classes.add(parent)
                         s.append(parent)
             return sur_classes

In [42]: Integer.__bases__
```

```
Out[42]: (sympy.core.numbers.Rational,)

In [43]: Rational.__bases__

Out[43]: (sympy.core.numbers.Number,)

In [44]: Number.__bases__

Out[44]: (sympy.core.expr.AtomicExpr,)
```

Et caetera :-).

```
In [45]: ancetres(Integer)

Out[45]: {object,
          sympy.core.basic.Atom,
          sympy.core.basic.Basic,
          sympy.core.evalf.EvalfMixin,
          sympy.core.expr.AtomicExpr,
          sympy.core.expr.Expr,
          sympy.core.numbers.Number,
          sympy.core.numbers.Rational}

In [46]: ancetres(exp)

Out[46]: {object,
          sympy.core.basic.Basic,
          sympy.core.evalf.EvalfMixin,
          sympy.core.expr.Expr,
          Application,
          ExpBase,
          Function}

In [47]: ancetres(Symbol)

Out[47]: {object,
          sympy.core.basic.Atom,
          sympy.core.basic.Basic,
          sympy.core.evalf.EvalfMixin,
          sympy.core.expr.AtomicExpr,
          sympy.core.expr.Expr,
          sympy.logic.boolalg.Boolean}
```

Remarquez que dans tous ces exemples, `Expr` est dans la liste des ancêtres.

Si vous avez de bons yeux, vous avez également remarqué la classe `object`, qui est est dans toutes les listes. En fait, toutes les classes Python ont la classe `object` pour ancêtre. Sauf une : laquelle à votre avis ???

```
In [48]: ancetres(list), ancetres(int), ancetres(complex), ancetres(object)

Out[48]: ({object}, {object}, {object}, set())
```

## 2.2 2. Analyse syntaxique des expressions

Chaque expression SymPy possède des **champs** qui permettent d'analyser cette expression. Voici les deux plus importants pour nous.

### 2.2.1 2.1 Les champs `func` et `args`

Chaque expression possède un champ `func` qui est le "type" de l'expression.

```
In [49]: expr = (x + y) * (z + 2)

In [50]: expr.func

Out[50]: sympy.core.mul.Mul
```

L'expression ci-dessus est en effet un produit de deux expressions plus simples. Lesquelles ? C'est là qu'intervient le champ `args`.

```
In [51]: expr.args

Out[51]:
```

$(z + 2,\ x + y)$

`expr.args` est le $n$-uplet des **sous-expressions** de `expr`. La documentation `sympy` garantit que pour **TOUTE** expression `expr` on a

$$expr = expr.fun(*expr.args)$$

Rappelons que si $f$ est une fonction Python et $s$ est, par exemple, le triplet $(x, y, z)$, alors $f(*s) = f(x, y, z)$. L'étoile est essentielle, car $f(s) = f((x, y, z))$, avec DEUX paires de parenthèses.

### 2.2.2 2.2 Expressions atomiques

Que se passe-t-il si notre expression est un symbole ou un nombre ?

```
In [52]: x.func

Out[52]: sympy.core.symbol.Symbol
```

Logique. Et les arguments de $x$ ?

```
In [53]: x.args

Out[53]:
```

()

Pas d'arguments. Normal, puisque $x$ n'est pas une fonction. Mais comment récupérer le fait que $x$ c'est 'x' ?

```
In [54]: repr(x)

Out[54]: 'x'
```

C'était facile. Ceci marche aussi sur les nombres.

```
In [55]: expr = Integer(17)
         expr.func, expr.args, srepr(expr)

Out[55]: (sympy.core.numbers.Integer, (), 'Integer(17)')
```

### 2.2.3 2.3 Hauteur d'une expression

Comment mesurer la complexité d'une expression $e$ ? Une bonne indication de cette complexité est la **hauteur** $h(e)$ de l'expression, que nous pouvons définir récursivement.

- Si $e$ est un symbole ou nombre, $h(e) = 1$.
- Sinon, soient $e_0, \ldots, e_{n-1}$ les arguments de $e$. On pose

$$h(e) = 1 + \max(h(e_0), \ldots, h(e_{n-1}))$$

```
In [56]: def hauteur(expr):
             gs = expr.args
             if len(gs) == 0: return 1
             else:
                 return 1 + max([hauteur(g) for g in gs])
```

Voici quelques exemples.

```
In [57]: hauteur(y)
```

```
Out[57]:
1
```

```
In [58]: hauteur(x ** 2)
```

```
Out[58]:
2
```

```
In [59]: hauteur(x ** 2 + y)
```

```
Out[59]:
3
```

```
In [60]: hauteur(cos(x) ** 2 + sin(x) ** 2)
```

```
Out[60]:
4
```

```
In [61]: hauteur(compliquee(10, x))
```

```
Out[61]:
21
```

Je vous laisse essayer d'autres exemples.

### 2.2.4 2.4 Représentation arborescente des expressions

Soit $e$ une expression. Nous allons définir **l'arbre syntaxique** de $e$, $T(e)$ ($T$ comme "tree") récursivement.

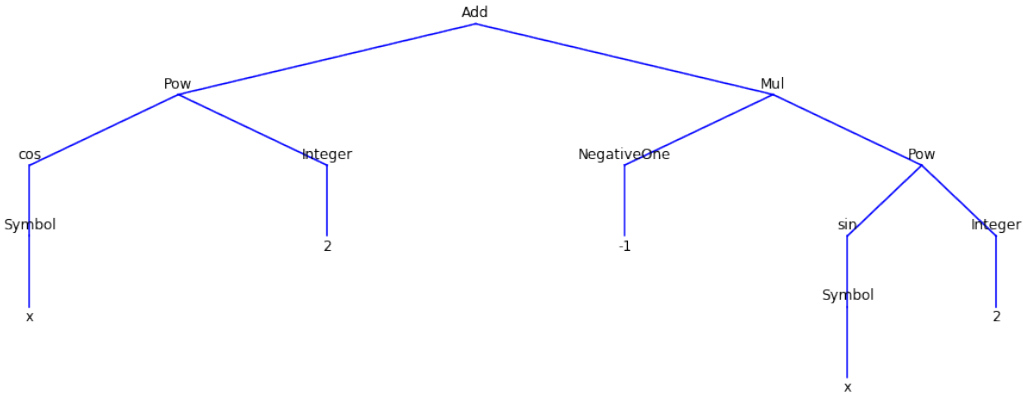
- Si $e$ est une expression atomique comme un entier ou un symbole, $T(e)$ a une racine étiquetée `Integer` ou `Symbol`. Sous cette racine se trouve une feuille, étiquetée par la valeur de l'expression, du genre 32 ou 'x'.
- Si $e = F(e_0, \ldots, e_{n-1})$ est une expression composée, $T(e)$ a une racine étiquetée par $F$ ($F$ est ce que renvoie `e.func`). Sous la racine se trouvent $n$ fils, qui sont les arbres syntaxiques $T(e_0), T(e_1), \ldots, T(e_{n-1})$.

La fonction `dessin_arbre` ci-dessous prend une expression en paramètre et dessine son arbre syntaxique. Je ne la détaillerai pas, libre à vous d'examiner son code. Je signale simplement l'utilité des deux paramètres optionnels.

- `bornes` est un quadruplet qui contient les coordonnées minimales et maximales du rectangle dans lequel on dessine l'arbre.
- $d$ est la distance entre deux niveaux de l'arbre. On l'évalue en calculant la hauteur de l'expression.

```
In [62]: def dessin_arbre(expr, bornes=(-1, 1, -1, 1), d=None):
             xmin, xmax, ymin, ymax = bornes
             if d == None:
                 d = (ymax - ymin) / hauteur(expr)
             plt.axis('off')
             f = expr.func
             gs = expr.args
             xc = (xmin + xmax) / 2
             plt.text(xc, ymax + 0.1 * d, f.__name__, fontsize=12, horizontalalignment='center')
             n = len(gs)
             if n == 0:
                 plt.plot([xc, xc], [ymax, ymax - d], 'b')
                 plt.text(xc, ymax - 1.2 * d, repr(expr), fontsize=12, horizontalalignment='cent
             for k in range(n):
                 x1 = xmin + k * (xmax - xmin) / n
                 x2 = xmin + (k + 1) * (xmax - xmin) / n
                 plt.plot([xc, (x1 + x2) / 2], [ymax, ymax - d], 'b')
                 dessin_arbre(gs[k], (x1, x2, ymin, ymax - d), d)

In [63]: dessin_arbre(cos(x) ** 2 - sin(x) ** 2)
```

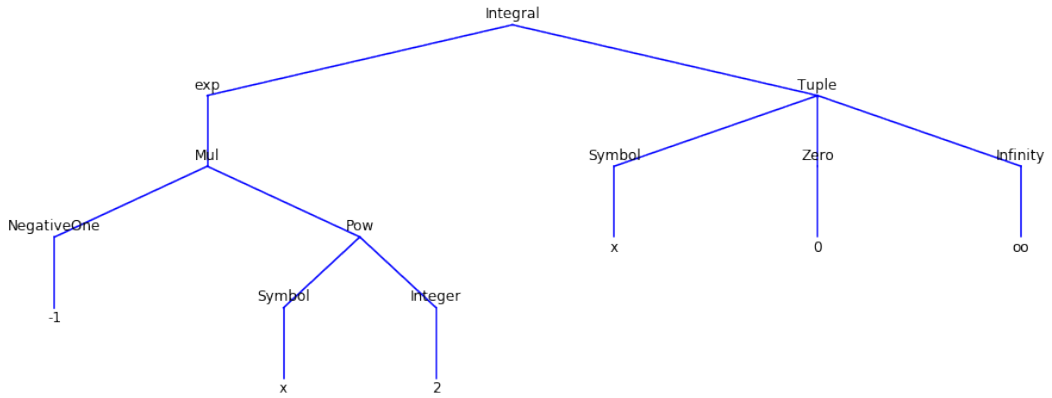



In [64]:
```
expr = Integral(exp(-x **2), (x, 0, oo))
expr
```

Out[64]:

$$\int_0^\infty e^{-x^2}\,dx$$

In [65]:
```
dessin_arbre(expr)
```

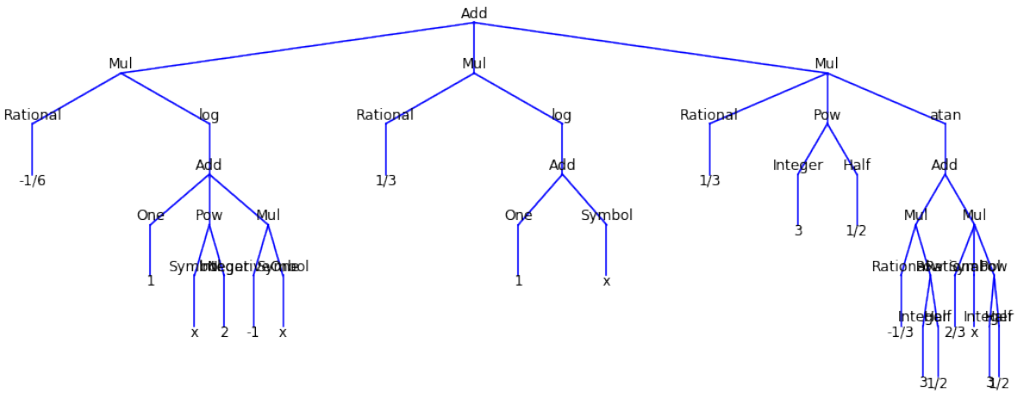



In [66]:
```
expr = integrate(1 / (x ** 3 + 1), x)
expr
```

Out[66]:

$$\frac{\log(x+1)}{3} - \frac{\log\left(x^2-x+1\right)}{6} + \frac{\sqrt{3}\,\operatorname{atan}\left(\frac{2\sqrt{3}x}{3} - \frac{\sqrt{3}}{3}\right)}{3}$$

In [67]:
```
dessin_arbre(expr)
```




Essayez avec d'autres expressions ...

## 2.3 3. Dérivation symbolique

`sympy` permet évidemment d'effectuer des dérivations symboliques grâce à la fonction `diff` :

```
In [68]: diff(sqrt(1 + x ** 2), x)
```

Out[68]:

$$\frac{x}{\sqrt{x^2+1}}$$

Nous allons dans cette section écrire notre propre fonction de dérivation. Appelons cette fonction `derivee`. Cette fonction prendra deux paramètres : une expression `f` et un symbole `x` et renverra la dérivée de $f$ par rapport à $x$.

### 2.3.1 3.1 Les dérivées usuelles

Avant toutes choses, ouvrons notre cours de maths et stockons dans un dictionnaire les dérivées des "fonctions" usuelles.

```
In [69]: derivees_usuelles = {sin: lambda t: cos(t),
                              cos: lambda t: -sin(t),
                              tan: lambda t: 1 / cos(t) ** 2,
                              exp: lambda t: exp(t),
                              log: lambda t: 1 / t,
                              sqrt: lambda t: 1/(2 * sqrt(t)),
                              asin: lambda t: 1 / sqrt(1 - t ** 2),
                              acos: lambda t: -1 / sqrt(1 - t ** 2),
                              atan: lambda t: 1 / (t ** 2 + 1),
                              sinh: lambda t: cosh(t),
                              cosh: lambda t: sinh(t),
                              tanh: lambda t: 1 / cosh(t) ** 2,
                              asinh: lambda t: 1 / sqrt(t ** 2 + 1),
                              acosh: lambda t: 1 / sqrt(t ** 2 - 1),
                              atanh: lambda t: 1 / (1 - t ** 2)}
```

Une clé du dictionnaire est un "nom de fonction" $f$. `derivees_usuelles[f]` est une fonction Python qui prend un paramètre $t$ et renvoie l'expression $f'(t)$.

### 2.3.2 3.2 La fonction de dérivation

Comment dériver l'expression $f$ par rapport au symbole $x$ ? Il suffit de considérer un certain nombre de cas :

- Si $f$ est le symbole $x$, on renvoie 1.
- Si $f$ est un nombre ou un symbole autre que $x$, on renvoie 0.
- Si $f$ est une somme, ou un produit, ou une puissance, on appelle une fonction adaptée (voir plus loin !).
- Si $f$ est une "fonction" (une composée, en fait), on appelle aussi une fonction adaptée. Nous supposerons que $f$ est une fonction d'une seule variable. Sinon les choses deviennent un peu plus compliquées. Restons raisonnables ...

Mais comment savoir dans quel cas on se trouve ? Rappelons-nous, les expressions de SymPy sont des objets qui sont des instances de classes. Ces objets contiennent des champs qui permettent de savoir de quel genre ils sont, ou ne sont pas. Le champ `is_Symbol` de l'expression $f$, par exemple, vaut `True` si $f$ est un symbole, et `False` sinon.

Le code de la fonction `derivee` est maintenant évident.

```
In [70]: def derivee(f, x):
             if f.is_Symbol and repr(f) == repr(x): return Integer(1)
             elif f.is_Number or f.is_NumberSymbol or f.is_Symbol: return Integer(0)
             elif f.is_Add: return derivee_somme(f.args, x)
             elif f.is_Mul: return derivee_produit(f.args, x)
             elif f.is_Pow: return derivee_puissance(f.args[0], f.args[1], x)
             elif f.is_Function: return derivee_fonction(f.func, f.args[0], x) # <-- f.args[0] c
             else: raise Exception('Not Implemented')
```

Si vous vous posez une question à propos des tests de la troisième ligne, voici l'explication :

```
In [71]: pi.is_NumberSymbol

Out[71]: True

In [72]: E.is_NumberSymbol

Out[72]: True
```

Pour SymPy, $\pi$ et $e$ ne sont pas des nombres mais des **symboles de nombres**.

Évidemment, pour l'instant on ne peut dériver que des constantes et des variables.

```
In [73]: derivee(Rational(3, 5), x)

Out[73]:
   0

In [74]: derivee(x, x)
```

Out[74]:
1

In [75]: derivee(x, t)

Out[75]:
0

### 2.3.3 3.3 Dérivée d'une somme

Dériver une somme, c'est facile :

$$\left(\sum_{k=0}^{n-1} g_k\right)' = \sum_{k=0}^{n-1} g_k'$$

```
In [76]: def derivee_somme(gs, x):
             s = Integer(0)
             for g in gs:
                 s = Add(s, derivee(g, x))
             return s
```

Testons.

```
In [77]: derivee(x + y + z, x)
```

Out[77]:
1

Peut-on dériver $x + x$ ? Non. Pourquoi ?

```
In [78]: x + x
```

Out[78]:
$2x$

Eh oui, SymPy transforme automatiquement $x + x$ en $2x$, et notre fonction ne sait pas encore dériver les produits. Alors apprenons-lui comment faire.

**Remarque** : Il est possible d'interdire à SymPy d'évaluer automatiquement ce genre d'expression. Je n'en parlerai pas dans ce notebook.

### 2.3.4 3.4 Dérivée d'un produit

Tout le monde le sait, $(uv)' = u'v + uv'$. Et $(uvw)'$ ? Eh bien

$$(uvw)' = u'vw + uv'w + uvw'$$

Cela se généralise facilement par récurrence sur le nombre de facteurs :

$$(g_0 g_1 \cdots g_{n-1})' = g_0' g_1 \cdots g_{n-1} + g_0 g_1' \cdots g_{n-1} + \cdots + g_0 g_1 \cdots g_{n-1}'$$

c'est à dire une somme de $n$ termes où dans chaque terme, on dérive un et un seul des facteurs.

La fonction ci-dessous fait le travail. Elle prend en paramètre une liste d'expressions et renvoie la dérivée du produit de ces expressions. Remarquons tout de même les deux boucles imbriquées, qui nous promettent une complexité en $O(n^2)$ où $n$ est le nombre de facteurs. telle quelle, cette fonction n'est pas efficace mais nous nous en contenterons.

```
In [79]: def derivee_produit(gs, x):
             s = Integer(0)
             for k in range(len(gs)):
                 g1 = derivee(gs[k], x)
                 p = Integer(1)
                 for j in range(len(gs)):
                     if j != k: p = Mul(p, gs[j])
                     else: p = p * g1
                 s = s + p
             return s
```

Maintenant, on peut dériver $x + x$ !

```
In [80]: derivee(x + x, x)
```

Out[80]:

$2$

```
In [81]: derivee(x * y * z, x)
```

Out[81]:

$yz$

Remarquons que nous ne savons pas encore dériver $x \times x$ :

```
In [82]: x * x
```

Out[82]:

$x^2$

La fonction `derivee_produit` est suffisamment compliquée pour demander à être testée plus en profondeur. À cet effet, définissons pour $x$ réel et $n$ entier naturel la $n$ième **puissance descendante** de $x$ : $x^{\underline{0}} = 1$ et, si $n \geq 1$ :

$$x^{\underline{n}} = x(x-1)(x-2)\dots(x-n+1)$$

```
In [83]: def power_dn(x, n):
             p = 1
             for k in range(n): p *= x - k
             return p
```

Voici $x^{\underline{10}}$.

```
In [84]: p = power_dn(x, 10)
         p
```

Out[84]:

$x\,(x-9)\,(x-8)\,(x-7)\,(x-6)\,(x-5)\,(x-4)\,(x-3)\,(x-2)\,(x-1)$

Dérivons.

```
In [85]: p1 = derivee(p, x)
         p1
```

Out[85]:

$x\,(x-9)\,(x-8)\,(x-7)\,(x-6)\,(x-5)\,(x-4)\,(x-3)\,(x-2)+x\,(x-9)\,(x-8)\,(x-7)\,(x-6)\,(x-5)\,(x-4$
$x\,(x-9)\,(x-8)\,(x-7)\,(x-6)\,(x-5)\,(x-4)\,(x-2)\,(x-1)+x\,(x-9)\,(x-8)\,(x-7)\,(x-6)\,(x-5)\,(x-3)\,(x$
$x\,(x-9)\,(x-8)\,(x-7)\,(x-6)\,(x-4)\,(x-3)\,(x-2)\,(x-1)+x\,(x-9)\,(x-8)\,(x-7)\,(x-5)\,(x-4)\,(x-3)\,(x$
$x\,(x-9)\,(x-8)\,(x-6)\,(x-5)\,(x-4)\,(x-3)\,(x-2)\,(x-1)+x\,(x-9)\,(x-7)\,(x-6)\,(x-5)\,(x-4)\,(x-3)\,(x$
$x\,(x-8)\,(x-7)\,(x-6)\,(x-5)\,(x-4)\,(x-3)\,(x-2)\,(x-1)+(x-9)\,(x-8)\,(x-7)\,(x-6)\,(x-5)\,(x-4)\,(x-$

Puis développons avec la fonction `expand` de SymPy.

```
In [86]: p1 = expand(derivee(p, x))
         p1
```

Out[86]:

$10x^9 - 405x^8 + 6960x^7 - 66150x^6 + 379638x^5 - 1346625x^4 + 2894720x^3 - 3518100x^2 + 2053152x - 362880$

maintenant faisons le "contraire" : développons d'abord $p$, **puis** dérivons. On devrait trouver le même résultat. Enfin, cela reste pour l'instant un **voeu pieux** :

```
In [87]: expand(p)
```

Out[87]:

$x^{10} - 45x^9 + 870x^8 - 9450x^7 + 63273x^6 - 269325x^5 + 723680x^4 - 1172700x^3 + 1026576x^2 - 362880x$

Eh oui, il y a des puissances dans ce que l'on veut dériver. Alors réglons le cas des puissances.

### 2.3.5 3.5 Dérivée d'une puissance

Soient $g$ et $h$ deux "fonctions". On a

$$g^h = e^{h \ln g}$$

et donc

$$(g^h)' = (h' \ln g + hg' \frac{1}{g}) g^h$$

D'où la fonction `derivee_puissance`.

```
In [88]: def derivee_puissance(g, h, x):
             g1 = derivee(g, x)
             h1 = derivee(h, x)
             return Mul(Add(Mul(h1, log(g)), Mul(h, g1, Pow(g, Integer(-1)))), Pow(g, h))
```

Maintenant, on peut dériver $x \times x$ :-).

```
In [89]: derivee(x * x, x)
```

Out[89]:

$2x$

Et aussi $\sqrt{x}$. En effet :

```
In [90]: srepr(sqrt(x))
```

```
Out[90]: "Pow(Symbol('x'), Rational(1, 2))"

In [91]: derivee(sqrt(x), x)

Out[91]:
```

$$\frac{1}{2\sqrt{x}}$$

Et aussi des choses beaucoup moins triviales

```
In [92]: derivee(sqrt(x ** 2 + 1), x)

Out[92]:
```

$$\frac{x}{\sqrt{x^2+1}}$$

```
In [93]: derivee(x ** (x ** 2 - x), x)

Out[93]:
```

$$x^{x^2-x}\left((2x-1)\log(x)+\frac{x^2-x}{x}\right)$$

```
In [94]: derivee((x ** 2 * (2 - x)) ** Rational(1, 3), x)

Out[94]:
```

$$\frac{\sqrt[3]{x^2(2-x)}\left(-x^2+2x(2-x)\right)}{3x^2(2-x)}$$

```
In [95]: factor(_)

Out[95]:
```

$$\frac{\sqrt[3]{-x^2(x-2)}(3x-4)}{3x(x-2)}$$

Et nous pouvons aussi finir de tester notre fonction qui dérive les produits !

```
In [96]: p1 = expand(derivee(p, x))
         p2 = derivee(expand(p), x)
         p1, p2

Out[96]:
```

$$\left(10x^9-405x^8+6960x^7-66150x^6+379638x^5-1346625x^4+2894720x^3-3518100x^2+2053152x-362880,\ 1\right.$$

```
In [97]: p1 - p2

Out[97]:
```

$$0$$

### 2.3.6 3.6 Dérivée d'une composée

Étant données deux fonctions $g$ et $h$, on a

$$(g \circ h)' = (g' \circ h) \times h'$$

D'où le code ci-dessous. On cherche la dérivée de $g$ dans le dictionnaire des dérivées usuelles.

- Si $g$ est une clé du dictionnaire, aucun problème.
- Si $g$ n'est pas une clé du dictionnaire la fonction renvoie une expression "formelle". Précisément, elle construit la dérivée "non évaluée" $\frac{d}{dt}g(t)$ (fonction `Derivative` de SymPy), puis elle remplace $t$ par $h$ (la méthode `subs` permet de faire cela). Elle multiplie ensuite par $h'$.

```
In [98]: def derivee_fonction(g, h, x):
             if not (g in derivees_usuelles):
                 return Mul(Derivative(g(t), t).subs(t, h), derivee(h, x))
             else:
                 return Mul(derivees_usuelles[g](h), derivee(h, x))
```

Maintenant, plus aucune (?) dérivée ne nous résiste :-)

```
In [99]: derivee(exp(-x ** 2), x)
```

Out[99]:

$$-2xe^{-x^2}$$

```
In [100]: derivee(x ** sin(x), x)
```

Out[100]:

$$x^{\sin(x)}\left(\log(x)\cos(x) + \frac{\sin(x)}{x}\right)$$

```
In [101]: derivee(sin(cos(tan(sin(cos(tan(x)))))) , x)
```

Out[101]:

$$\frac{\sin(\tan(x))\sin(\tan(\sin(\cos(\tan(x)))))\cos(\cos(\tan(x)))\cos(\cos(\tan(\sin(\cos(\tan(x))))))}{\cos^2(x)\cos^2(\sin(\cos(\tan(x))))}$$

```
In [102]: expr = integrate(1 / (x ** 4 + 1), x)
          expr
```

Out[102]:

$$-\frac{\sqrt{2}\log\left(x^2 - \sqrt{2}x + 1\right)}{8} + \frac{\sqrt{2}\log\left(x^2 + \sqrt{2}x + 1\right)}{8} + \frac{\sqrt{2}\operatorname{atan}\left(\sqrt{2}x - 1\right)}{4} + \frac{\sqrt{2}\operatorname{atan}\left(\sqrt{2}x + 1\right)}{4}$$

```
In [103]: derivee(expr, x)
```

Out[103]:

$$-\frac{\sqrt{2}\left(2x-\sqrt{2}\right)}{8\left(x^2-\sqrt{2}x+1\right)}+\frac{\sqrt{2}\left(2x+\sqrt{2}\right)}{8\left(x^2+\sqrt{2}x+1\right)}+\frac{1}{2\left(\left(\sqrt{2}x+1\right)^2+1\right)}+\frac{1}{2\left(\left(\sqrt{2}x-1\right)^2+1\right)}$$

```
In [104]: simplify(_)
```

Out[104]:

$$\frac{1}{x^4+1}$$

**Remarque** : La simplification des expressions est un sujet à part entière, et c'est un sujet compliqué. Nous avons ci-dessus utilisé la fonction `simplify`, qui est une sorte de **fonction magique**. Si vous voulez plus de détails sur le sujet de la simplification, consultez la documentation de SymPy.

Un dernier exemple. Créons deux fonctions "abstraites" $f$ et $g$. Notre fonction `derivee` ne sait pas les dériver. Elle renvoie tout de même des résultats cohérents ci-dessous :

```
In [105]: f = Function('f')
          g = Function('g')

In [106]: derivee(f(x ** 2 + x), x)
```

Out[106]:

$$(2x+1)\left.\frac{d}{dt}f(t)\right|_{t=x^2+x}$$

```
In [107]: derivee(g(asin(f(x))), x)
```

Out[107]:

$$\frac{\left.\frac{d}{dx}f(x)\ \frac{d}{dt}g(t)\right|_{t=\mathrm{asin}\,(f(x))}}{\sqrt{1-f^2(x)}}$$

### 2.3.7 3.7 Rajouter des dérivées usuelles

Comment faire pour booster notre fonction `derivee` ? Imaginons qu'une nouvelle fonction usuelle (ou pas) devienne pour nous très importante : il suffit de la rajouter au dictionnaire des dérivées usuelles. Prenons deux exemples.

Une fonction importante en probabilités est la primitive de $x \mapsto e^{-x^2}$ qui s'annule en 0, ou plus exactement $\frac{2}{\sqrt{\pi}}$ fois cette primitive. Appelons la $\psi$.

Tout d'abord, disons à SymPy que $\psi$ est une **fonction**.

```
In [108]: psi = Function('psi')
```

On peut maintenant considérer des expressions du genre :

```
In [109]: cos(psi(x ** 2 + 1))
```

Out[109]:

$\cos\left(\psi\left(x^2+1\right)\right)$

Ensuite, rajoutons $\psi$ dans le dictionnaire. On a

$$\psi'(t) = \frac{2}{\sqrt{\pi}} e^{-t^2}$$

```
In [110]: derivees_usuelles[psi] = lambda t: 2 / sqrt(pi) * exp(-t ** 2)
```

Et voilà. Quelques tests ?

```
In [111]: derivee(psi(x), x)
```

Out[111]:

$$\frac{2e^{-x^2}}{\sqrt{\pi}}$$

**Remarque** : Dans SymPy, cette fonction s'appelle `erf`.

```
In [112]: diff(erf(x), x)
```

Out[112]:

$$\frac{2e^{-x^2}}{\sqrt{\pi}}$$

```
In [113]: derivee(erf(x), x)
```

Out[113]:

$$\frac{d}{dx}\operatorname{erf}(x)$$

Ben oui, notre fonction ne sait pas que `erf`, c'est $\psi$.

Quelle est la dérivée de $\ln \psi$ ?

```
In [114]: derivee(log(psi(x)), x)
```

Out[114]:

$$\frac{2e^{-x^2}}{\sqrt{\pi}\psi(x)}$$

Et sa dérivée cinquième ?

```
In [115]: expr = log(psi(x))
          for k in range(5):
              expr = derivee(expr, x)
          expr
```

Out[115]:

$$\frac{32x^4e^{-x^2}}{\sqrt{\pi}\psi(x)} + \frac{480x^3e^{-2x^2}}{\pi\psi^2(x)} - \frac{96x^2e^{-x^2}}{\sqrt{\pi}\psi(x)} + \frac{1600x^2e^{-3x^2}}{\pi^{\frac{3}{2}}\psi^3(x)} - \frac{400xe^{-2x^2}}{\pi\psi^2(x)} + \frac{1920xe^{-4x^2}}{\pi^2\psi^4(x)} + \frac{24e^{-x^2}}{\sqrt{\pi}\psi(x)} - \frac{320e^{-3x^2}}{\pi^{\frac{3}{2}}\psi^3(x)} + \frac{768e^{-5x^2}}{\pi^{\frac{5}{2}}\psi^5(x)}$$

```
In [116]: factor(_)
```

Out[116]:

$$\frac{8\left(4\pi^4x^4\psi^4(x)e^{4x^2}+60\pi^{\frac{7}{2}}x^3\psi^3(x)e^{3x^2}-12\pi^4x^2\psi^4(x)e^{4x^2}+200\pi^3x^2\psi^2(x)e^{2x^2}-50\pi^{\frac{7}{2}}x\psi^3(x)e^{3x^2}+240\pi^{\frac{5}{2}}x\psi(x\right.}{\pi^{\frac{9}{2}}\psi^5(x)}$$

Prenons un autre exemple. Pour $x \neq 0$, posons

$$\phi(x) = \frac{1}{x}\int_0^x \frac{\arctan t}{t}dt$$

On vérifie facilement que pour tout $x$ non nul

$$\phi'(x) = \frac{1}{x}\left(\frac{\arctan x}{x} - \phi(x)\right)$$

In [117]:
```
phi = Function('phi')
```

La variable $\phi$ contient une expression SymPy qui est une fonction au sens mathématique du terme. En clair, $\phi(x)$ est elle-même une expression.

In [118]:
```
derivees_usuelles[phi] = lambda t: (atan(t) / t - phi(t)) / t
```

Que vaut $\phi''(x)$ ?

In [119]:
```
derivee(derivee(phi(x), x), x)
```

Out[119]:

$$\frac{-\frac{-\phi(x)+\frac{\operatorname{atan}(x)}{x}}{x}+\frac{1}{x(x^2+1)}-\frac{\operatorname{atan}(x)}{x^2}}{x}-\frac{-\phi(x)+\frac{\operatorname{atan}(x)}{x}}{x^2}$$

In [120]:
```
factor(_)
```

Out[120]:

$$\frac{2x^3\phi(x)-3x^2\operatorname{atan}(x)+2x\phi(x)+x-3\operatorname{atan}(x)}{x^3\left(x^2+1\right)}$$

Que vaut $x^2\phi'(x) + x\phi(x)$ ?

In [121]:
```
x ** 2 * derivee(phi(x), x) + x * phi(x)
```

Out[121]:

$$x\left(-\phi(x)+\frac{\operatorname{atan}(x)}{x}\right)+x\phi(x)$$

Simplifions.

In [122]:
```
simplify(_)
```

Out[122]:

$\operatorname{atan}(x)$

$\phi$ est donc solution de l'équation différentielle

$$x^2y' + xy = \arctan x$$

### 2.3.8 3.8 Et maintenant ?

Notre fonction de dérivation ne dérive évidemment pas toutes les expressions que l'on peut fabriquer avec SymPy. Pour ne prendre qu'un exemple :

```
In [123]: f = Function('f')
          g = Function('g')
          h = Function('h')
          expr = Integral(f(t, x), (t, g(x), h(x)))
          expr
```

Out[123]:

$$\int_{g(x)}^{h(x)} f(t,x)\,dt$$

```
In [124]: isinstance(expr, Integral)
```

```
Out[124]: True
```

SymPy sait dériver cela.

```
In [125]: diff(expr, x)
```

Out[125]:

$$-f(g(x),x)\frac{d}{dx}g(x) + f(h(x),x)\frac{d}{dx}h(x) + \int_{g(x)}^{h(x)} \frac{\partial}{\partial x} f(t,x)\,dt$$

Si nous voulons que notre fonction `derivee` puisse dériver ce genre d'expression, il faut reprendre son code et rajouter un cas, celui où l'expression à dériver est du genre "intégrale". C'est tout à fait possible.

**Exercice** : Faites-le, apprenez à la fonction `derivee` à dériver par rapport à $x$ des expressions du genre $\int_g^h f$ où $g, h, f$ sont des expressions. Adaptez pour cela la formule écrite juste au-dessus.

Ce qui est aussi tout à fait certain c'est qu'au fil du temps nous nous apercevrons que d'autres types d'expressions ne sont pas dérivables avec notre fonction. À chaque fois il faudra réécrire le code de la fonction `derivee` qui va s'allonger, s'allonger .... Ceci est évidemment problématique.

La philosophie du code source de SymPy est tout à fait différente. Ce code est **orienté objet** : chaque nouveau genre d'expression donne lieu à la définition d'une nouvelle **classe**. Et c'est à l'intérieur de cette classe qu'est définie la **méthode** permettant aux objets de "se dériver". Je n'en dirai pas plus. Si vous voulez en savoir plus allez sur le site de SymPy https://www.sympy.org. Vous y trouverez un lien vers le code source de SymPy. Allez par exemple dans le répertoire `sympy/core` et regardez le fichier `add.py`. Puis cherchez la ligne

`def _eval_derivative(self, s):`

Votre navigateur peut la trouver automatiquement !

**Exercice** : Faites de même avec les classes `Mul` et `Pow`. La façon dont la dérivée d'un produit est calculée par SymPy est-elle meilleure que la nôtre ?

**Exertcice** : Allez voir dans le répertoire `sympy/integrals` le fichier `integrals.py`. Comment SymPy dérive-t-il une intégrale ? Comparez avec ce que vous avez fait dans l'exercice ci-dessus, que vous avez forcément fait.

Bonne lecture :-).

```
In [ ]:
```