Mathematica, Maple, the Symbolic Math Toolbox of MATLAB, SageMath, and the SymPy package of Python. Although most of these have an application programming interface (API) for Python, the only one that is exclusively written in and for Python is the SymPy package, and therefore we will use this as our CAS.

The SymPy package is available in the base Anaconda environment. It can be imported in a program with the following statement:

```
import sympy as sp
```

We use the alias `sp` throughout the text.

4.1 Symbolic Expressions, Variables, and Functions

In SymPy, a **symbolic expression** is comprised of SymPy objects. Unlike numerical expressions, these are not automatically evaluated to integer or floating-point numbers. For instance, using the standard library `math` module, the expression `math.sqrt(3)/2` immediately evaulates to the floating-point approximation of about `0.866`. However, in SymPy, something else happens:[2]

```
sp.sqrt(3) / 2
```

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

This is an *exact* representation of the mathematical expression, as opposed to the approximation obtained previously.

A symbolic expression can be represented as an **expression tree**:

```
sp.srepr(sp.sqrt(3) / 2)  # Show expression tree representation
```

```
'Mul(Rational(1, 2), Pow(Integer(3), Rational(1, 2)))'
```

This can be visualized as a tree graph like that shown in figure 4.1.

2. We are pretty printing results that are mathematical expressions.

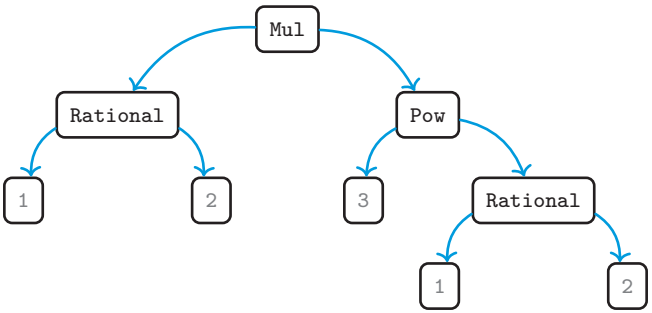

Figure 4.1. A symbolic expression tree for `sp.sqrt(3)/2`.

4.1.1 Symbolic Variables

Mathematical variables can be represented as **symbolic variables** that stand in for an unspecified number. In SymPy, symbolic variables can be created as follows:

```
x, y = sp.symbols("x, y", real=True)  # Create two real variables
```

The string passed to `sp.symbols()` can separate variables with commas and/or whitespace. The type of unspecified number being represented by the symbolic variables listed is assumed to be `complex` unless an optional argument is passed declaring otherwise. Here we have declared that `x` and `y` are real with the **predicate** `real`. Other common predicates include the following:

- Integers: `integer`, `noninteger`, `even`, and `odd`
- Real numbers: `real`, `positive`, `nonnegative`, `nonzero`, `nonpositive`, and `negative`
- Complex numbers: `complex` (default) and `imaginary`

The predicate of a symbolic variable determines the assumptions SymPy will make about it when it appears in a symbolic expression. For instance, consider the following symbolic expressions:

```
z = sp.symbols("z")  # Complex
p = sp.symbols("p", positive=True)
sp.sqrt(z**2)
sp.sqrt(x**2)  # Using real x from above
sp.sqrt(p**2)
```

$\sqrt{z^2}$
$|x|$
p

We see that the expression automatically simplifies based on the predicates provided for each variable. This will prove especially useful once we begin using the symbolic expression manipulation techniques described in the following sections.

4.1.2 Symbolic Functions

A mathematical function can be represented in SymPy by a **symbolic function**. There are a few different ways to create these, and we will consider only the simplest and most common cases here. An **undefined function** f that should be treated as monolithic and as having no special properties can be defined as follows:

```
f = sp.Function("f")  # Type: sp.core.function.UndefinedFunction
f(x) + 3 * f(x)  # Using x from above
```

↳ $4f(x)$

Predicates can be applied to functions, as well; for instance,

```
g = sp.Function("g", real=True)
f(x) + g(x, y) * g(3, -3)
```

↳ $f(x)+g(3,-3)g(x,y)$

An **applied undefined function** is an undefined function that has been given an argument. For instance,

```
h = sp.Function("h")(x)  # Types: h, sp.core.function.AppliedUndef
3 * h  ## Leave off the argument
```

↳ $3h(x)$

Undefined functions are never evaluated. At times we want to define a function that is always to be evaluated; in SymPy such a function is called a **fully evaluated function**. A fully evaluated function can be created as a regular Python function, as in the following case:

```
def F(x):
    return x**2 - 4
F(x)**2
```

↳ $\left(x^2-4\right)^2$

For **piecewise functions**, regular Python functions with `if` statements will work, but it is preferable to use the `sp.Piecewise()` function. For instance,

```
G = sp.Piecewise(
    (x**2, x <= 0),  # x^2 for x ≤ 0
    (3*x, True)  # 3x for x > 0
)
```

Many common mathematical functions are built in to SymPy, including those shown in table 4.1.

Table 4.1: Elementary mathematical functions in SymPy.

Kind	SymPy Functions (`sp.` prefix suppressed)
Complex	`Abs()`, `arg()`, `conjugate()`, `im()`, `re()`, `sign()`
Trigonmetric	`sin()`, `cos()`, `tan()`, `sec()`, `csc()`, `cot()`
Inverse Trigonometric	`asin()`, `acos()`, `atan()`, `atan2()`, `asec()`, `acsc()`, `acot()`
Hyperbolic	`sinh()`, `cosh()`, `tanh()`, `coth()`, `sech()`, `csch()`
Inverse Hyperbolic	`asinh()`, `acosh()`, `atanh()`, `acoth()`, `asech()`
Integer	`ceiling()`, `floor()`, `frac()` `get_integer_part()`
Exponential	`exp()`, `log()`
Miscellaneous	`Min()`, `Max()`, `root()`, `sqrt()`

In rare cases, we must define a **custom function**; that is, a subclass of the `sp.Function` class. Such a function needs to have its behavior thoroughly defined. Once it is completed, it should behave just as built-in functions like `sp.sin()`. For a tutorial on writing custom functions, see SymPy Development Team (2023d).

4.2 Manipulating Symbolic Expressions

In engineering symbolic analysis, the need to manipulate, often algebraically, mathematical expressions arises constantly. SymPy has several powerful tools for manipulating symbolic expressions, the most useful of which we will consider here.

4.2.1 The `simplify()` Function and Method

A built-in Sympy function and method, `sp.simplify()`, is a common SymPy tool for manipulation because simplification is often what we want. Recall that some basic simplification occurs automatically; however, in many cases this automatic simplification is insufficient. Applying `sp.simplify()` typically results in an expression as simple as or simpler than its input; however, the precise meaning of "simpler" is quite vague, which can lead to frustrating cases in which a version of an expression we consider to be simpler is not chosen by the `sp.simplify()` algorithm. In such cases, we will often use the more manual techniques considered later in this section.

The predicates (i.e., assumptions) used to define the symbolic variables and functions that appear in a symbolic expression are respected by `sp.simplify()`. Consider the following example:

```
x = sp.symbols("x", real=True)
e0 = (x**2 + 2*x + 3*x)/(x**2 + 2*x); e0  # For display
e0.simplify()  # Returns simplified expression, leaves e0 unchanged
```