SymPy

symbolic mathematics in Python (3.x)

Piotr Beling

Uniwersytet Łódzki
(University of Łódź)

2016

This presentation is based on and cites vast fragments of the official tutorial authored by the SymPy Development Team and available on http://docs.sympy.org.

Run ipython qtconsole and import SymPy

Today we will work in a python shell and we will use the `sympy` module.

Please run *ipython qtconsole*:

- Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
- Linux: `ipython3 qtconsole`

Import the whole `sympy` module into the current namespace:

```
from sympy import *
```

And initialize the best SymPy pretty-printer for your environment, to get pretty printing of SymPy expressions:

```
init_printing()
```

About SymPy and Symbolic Computation

- SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS);
- symbolic computation deals with the computation of mathematical objects symbolically;
- objects (like $\sqrt{2}$) are represented exactly, not approximately, and mathematical expressions with unevaluated variables (like x) are left in symbolic form.

Example: execute:

`import math`	imports the `math` module;
`math.sqrt(9)`	$\sqrt{9}$ calculated exactly by the `math` module;
`sqrt(9)`	similar to the above, but calculated by SymPy;
`math.sqrt(8)`	approximation of $\sqrt{8}$ calculated by the `math` module; $\sqrt{8}$ is an irrational number and cannot be represented exactly by a finite decimal;
`sqrt(8)`	SymPy gives accurate, symbolic result; symbolically simplified: $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$.

About SymPy and Symbolic Computation

- SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS);
- symbolic computation deals with the computation of mathematical objects symbolically;
- objects (like $\sqrt{2}$) are represented exactly, not approximately, and mathematical expressions with unevaluated variables (like x) are left in symbolic form.

Example: execute:

`import math`	imports the `math` module;
`math.sqrt(9)`	$\sqrt{9}$ calculated exactly by the `math` module;
`sqrt(9)`	similar to the above, but calculated by SymPy;
`math.sqrt(8)`	approximation of $\sqrt{8}$ calculated by the `math` module; $\sqrt{8}$ is an irrational number and cannot be represented exactly by a finite decimal;
`sqrt(8)`	SymPy gives accurate, symbolic result; symbolically simplified: $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$.

About SymPy and Symbolic Computation

- SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS);
- symbolic computation deals with the computation of mathematical objects symbolically;
- objects (like $\sqrt{2}$) are represented exactly, not approximately, and mathematical expressions with unevaluated variables (like x) are left in symbolic form.

Example: execute:

`import math`	imports the `math` module;
`math.sqrt(9)`	$\sqrt{9}$ calculated exactly by the `math` module;
`sqrt(9)`	similar to the above, but calculated by SymPy;
`math.sqrt(8)`	approximation of $\sqrt{8}$ calculated by the `math` module; $\sqrt{8}$ is an irrational number and cannot be represented exactly by a finite decimal;
`sqrt(8)`	SymPy gives accurate, symbolic result; symbolically simplified: $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$.

About SymPy and Symbolic Computation

- SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS);
- symbolic computation deals with the computation of mathematical objects symbolically;
- objects (like $\sqrt{2}$) are represented exactly, not approximately, and mathematical expressions with unevaluated variables (like x) are left in symbolic form.

Example: execute:

`import math`	imports the `math` module;
`math.sqrt(9)`	$\sqrt{9}$ calculated exactly by the `math` module;
`sqrt(9)`	similar to the above, but calculated by SymPy;
`math.sqrt(8)`	approximation of $\sqrt{8}$ calculated by the `math` module; $\sqrt{8}$ is an irrational number and cannot be represented exactly by a finite decimal;
`sqrt(8)`	SymPy gives accurate, symbolic result; symbolically simplified: $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$.

About SymPy and Symbolic Computation

- SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS);
- symbolic computation deals with the computation of mathematical objects symbolically;
- objects (like $\sqrt{2}$) are represented exactly, not approximately, and mathematical expressions with unevaluated variables (like x) are left in symbolic form.

Example: execute:

`import math`	imports the `math` module;
`math.sqrt(9)`	$\sqrt{9}$ calculated exactly by the `math` module;
`sqrt(9)`	similar to the above, but calculated by SymPy;
`math.sqrt(8)`	approximation of $\sqrt{8}$ calculated by the `math` module; $\sqrt{8}$ is an irrational number and cannot be represented exactly by a finite decimal;
`sqrt(8)`	SymPy gives accurate, symbolic result; symbolically simplified: $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$.

About SymPy and Symbolic Computation

- SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS);
- symbolic computation deals with the computation of mathematical objects symbolically;
- objects (like $\sqrt{2}$) are represented exactly, not approximately, and mathematical expressions with unevaluated variables (like x) are left in symbolic form.

Example: execute:

`import math`	imports the `math` module;
`math.sqrt(9)`	$\sqrt{9}$ calculated exactly by the `math` module;
`sqrt(9)`	similar to the above, but calculated by SymPy;
`math.sqrt(8)`	approximation of $\sqrt{8}$ calculated by the `math` module; $\sqrt{8}$ is an irrational number and cannot be represented exactly by a finite decimal;
`sqrt(8)`	SymPy gives accurate, symbolic result; symbolically simplified: $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$.

About SymPy and Symbolic Computation

- SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS);
- symbolic computation deals with the computation of mathematical objects symbolically;
- objects (like $\sqrt{2}$) are represented exactly, not approximately, and mathematical expressions with unevaluated variables (like x) are left in symbolic form.

Example: execute:

`import math`	imports the `math` module;
`math.sqrt(9)`	$\sqrt{9}$ calculated exactly by the `math` module;
`sqrt(9)`	similar to the above, but calculated by SymPy;
`math.sqrt(8)`	approximation of $\sqrt{8}$ calculated by the `math` module; $\sqrt{8}$ is an irrational number and cannot be represented exactly by a finite decimal;
`sqrt(8)`	SymPy gives accurate, symbolic result; symbolically simplified: $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

Variables, symbols and expressions – basics

Variables (symbols) in SymPy must be defined before use.
This can be done by the `symbols`, the `var`, or the `S` function.

Example: execute:

`x,y = symbols('x,y')`	defines 2 symbols and assigns their objects to the Python variables;
`expr = x + 2*y`	defines a symbolic expression;
`expr`	displays it;
`expr + 1`	gives $x + 2y + 1$;
`expr - x`	x and the $-x$ canceled each other,
`x*expr`	but only obvious simplifications are performed automatically,
`expand(x*expr)`	e.g. the expansion must be forced,
`factor(x**2 + 2*x*y)`	and the polynomial factorization too.

Exercise: define the additional `z` symbol and express $(x + y)(x + z)^3 - x^4$ in the expanded form.

```
z = symbols('z')
expand((x+y) * (x+z)**3 - x**4)
```

Simplification of mathematical expressions

- SymPy has dozens of functions (like `expand` or `factor`) to perform various kinds of simplification;
- there is also one general function called `simplify` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Try:

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
(sin(x)**2 + cos(x)**2).simplify()
```

Many operations in SymPy are available (with the same names) as both: stand-alone functions and methods of expressions.

- `simplify` uses heuristics to determine the simplest result;
- however "simplest" is not a well-defined term and sometimes the desired form needs to be indicated explicitly, e.g. compare `simplify(x**2+2*x+1)` with `factor(x**2+2*x+1)`;

Exercise: simplify the expression $\sin(x)\cos(y) + \cos(x)\sin(y)$

Simplification of mathematical expressions

- SymPy has dozens of functions (like `expand` or `factor`) to perform various kinds of simplification;
- there is also one general function called `simplify` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Try:

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
(sin(x)**2 + cos(x)**2).simplify()
```

Many operations in SymPy are available (with the same names) as both: stand-alone functions and methods of expressions.

- `simplify` uses heuristics to determine the simplest result;
- however "simplest" is not a well-defined term and sometimes the desired form needs to be indicated explicitly, e.g. compare `simplify(x**2+2*x+1)` with `factor(x**2+2*x+1)`;

Exercise: simplify the expression $\sin(x)\cos(y) + \cos(x)\sin(y)$

Simplification of mathematical expressions

- SymPy has dozens of functions (like `expand` or `factor`) to perform various kinds of simplification;
- there is also one general function called `simplify` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Try:

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
(sin(x)**2 + cos(x)**2).simplify()
```

Many operations in SymPy are available (with the same names) as both: stand-alone functions and methods of expressions.

- `simplify` uses heuristics to determine the simplest result;
- however "simplest" is not a well-defined term and sometimes the desired form needs to be indicated explicitly, e.g. compare `simplify(x**2+2*x+1)` with `factor(x**2+2*x+1)`;

Exercise: simplify the expression $\sin(x)\cos(y) + \cos(x)\sin(y)$

Simplification of mathematical expressions

- SymPy has dozens of functions (like `expand` or `factor`) to perform various kinds of simplification;
- there is also one general function called `simplify` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Try:

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
(sin(x)**2 + cos(x)**2).simplify()
```

Many operations in SymPy are available (with the same names) as both: stand-alone functions and methods of expressions.

- `simplify` uses heuristics to determine the simplest result;
- however "simplest" is not a well-defined term and sometimes the desired form needs to be indicated explicitly, e.g. compare `simplify(x**2+2*x+1)` with `factor(x**2+2*x+1)`;

Exercise: simplify the expression $\sin(x)\cos(y) + \cos(x)\sin(y)$

Simplification of mathematical expressions

- SymPy has dozens of functions (like `expand` or `factor`) to perform various kinds of simplification;
- there is also one general function called `simplify` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Try:

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
(sin(x)**2 + cos(x)**2).simplify()
```

Many operations in SymPy are available (with the same names) as both: stand-alone functions and methods of expressions.

- `simplify` uses heuristics to determine the simplest result;
- however "simplest" is not a well-defined term and sometimes the desired form needs to be indicated explicitly, e.g. compare `simplify(x**2+2*x+1)` with `factor(x**2+2*x+1)`;

Exercise: simplify the expression $\sin(x)\cos(y) + \cos(x)\sin(y)$

Simplification of mathematical expressions

- SymPy has dozens of functions (like `expand` or `factor`) to perform various kinds of simplification;
- there is also one general function called `simplify` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Try:

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
(sin(x)**2 + cos(x)**2).simplify()
```

Many operations in SymPy are available (with the same names) as both: stand-alone functions and methods of expressions.

- `simplify` uses heuristics to determine the simplest result;
- however "simplest" is not a well-defined term and sometimes the desired form needs to be indicated explicitly, e.g. compare `simplify(x**2+2*x+1)` with `factor(x**2+2*x+1)`;

Exercise: simplify the expression $\sin(x)\cos(y) + \cos(x)\sin(y)$

Simplification of mathematical expressions

- SymPy has dozens of functions (like `expand` or `factor`) to perform various kinds of simplification;
- there is also one general function called `simplify` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Try:

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
(sin(x)**2 + cos(x)**2).simplify()
```

Many operations in SymPy are available (with the same names) as both: stand-alone functions and methods of expressions.

- `simplify` uses heuristics to determine the simplest result;
- however "simplest" is not a well-defined term and sometimes the desired form needs to be indicated explicitly, e.g. compare `simplify(x**2+2*x+1)` with `factor(x**2+2*x+1)`;

Exercise: simplify the expression $\sin(x)\cos(y) + \cos(x)\sin(y)$

Simplification of mathematical expressions

- SymPy has dozens of functions (like `expand` or `factor`) to perform various kinds of simplification;
- there is also one general function called `simplify` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Try:

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
(sin(x)**2 + cos(x)**2).simplify()
```

Many operations in SymPy are available (with the same names) as both: stand-alone functions and methods of expressions.

- `simplify` uses heuristics to determine the simplest result;
- however "simplest" is not a well-defined term and sometimes the desired form needs to be indicated explicitly, e.g. compare `simplify(x**2+2*x+1)` with `factor(x**2+2*x+1)`;

Exercise: simplify the expression $\sin(x)\cos(y) + \cos(x)\sin(y)$

Code: `simplify(sin(x)*cos(y) + cos(x)*sin(y))`

Test symbolic expressions for equality

In SymPy, == represents exact structural equality testing, e.g:
`2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g:
`2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).

The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$

Test symbolic expressions for equality

In SymPy, `==` represents exact structural equality testing, e.g: `2*x==2*x` is `True`, as expressions are identical, but `a==b` is `False` for `a = (x + 1)**2` and `b = x**2 + 2*x + 1` (check!).
The best ways to check if $a = b$ in the mathematical sense are:

- `simplify(a-b)==0` simplifies $a - b$, and sees if it goes to 0;
- `simplify(Eq(a,b))` simplifies the symbolic equalities of `a` and `b` (`Eq(a,b)`), which may lead to a boolean value;
- `a.equals(b)` tries more techniques (e.g. it can evaluate and compare `a` with `b` numerically at random points) and returns `True` if $a = b$, `False` if $a \neq b$, or `None` if it cannot decide.

Note that all the methods above work well for most common expressions (give `True` for our `a` and `b`), but are not perfect.
It is proven that it is impossible to determine if two symbolic expressions are identically equal in general.
Exercise: check if $\cos(2x) = \cos^2(x) - \sin^2(x)$
`cos(2*x).equals(cos(x)**2 - sin(x)**2)`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses **`dict`**;
`expr.subs([(x, 2), (y, 0)])`	uses **`list`** of **`tuples`**.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses **`dict`**;
`expr.subs([(x, 2), (y, 0)])`	uses **`list`** of **`tuples`**.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses **`dict`**;
`expr.subs([(x, 2), (y, 0)])`	uses **`list`** of **`tuples`**.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a `dict`ionary, a `set`, or a sequence (e.g. a `list`) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses `dict`;
`expr.subs([(x, 2), (y, 0)])`	uses `list` of `tuples`.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a `dict`ionary, a `set`, or a sequence (e.g. a `list`) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses `dict`;
`expr.subs([(x, 2), (y, 0)])`	uses `list` of `tuples`.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a `dict`ionary, a `set`, or a sequence (e.g. a `list`) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses `dict`;
`expr.subs([(x, 2), (y, 0)])`	uses `list` of `tuples`.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a `dict`ionary, a `set`, or a sequence (e.g. a `list`) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses `dict`;
`expr.subs([(x, 2), (y, 0)])`	uses `list` of `tuples`.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a `dict`ionary, a `set`, or a sequence (e.g. a `list`) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses `dict`;
`expr.subs([(x, 2), (y, 0)])`	uses `list` of `tuples`.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses `dict`;
`expr.subs([(x, 2), (y, 0)])`	uses `list` of `tuples`.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses **`dict`**;
`expr.subs([(x, 2), (y, 0)])`	uses **`list`** of **`tuple`**s.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses **`dict`**;
`expr.subs([(x, 2), (y, 0)])`	uses **`list`** of **`tuple`**s.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses **`dict`**;
`expr.subs([(x, 2), (y, 0)])`	uses **`list`** of **`tuple`**s.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses **`dict`**;
`expr.subs([(x, 2), (y, 0)])`	uses **`list`** of **`tuple`**s.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

Substitution

Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method, e.g.

`a = cos(x)+1`	constructs the expression `a`;
`a.subs(x, y)`	gives the copy of `a` with `x` replaced by `y`;
`a`	`a` still has `x`; SymPy objects are immutable;
`a.subs(1, 2)`	$= \cos(x) + 2$; substitutes 2 for 1;
`a.subs(x, 0)`	evaluates `a` at the point $x = 0$; $\cos(0) + 1 = 2$;
`a.subs(x, 1)`	$\cos(1)$ is left unevaluated, as it is irrational.

To perform multiple substitutions at once, pass a **`dict`**ionary, a **`set`**, or a sequence (e.g. a **`list`**) of (old, new) pairs to `subs`, e.g.

`expr = x**3 + 4*x*y - z`	constructs the expression;
`expr.subs({x: 2, y: 4, z: 0})`	uses **`dict`**;
`expr.subs([(x, 2), (y, 0)])`	uses **`list`** of **`tuple`**s.

Exercise: substitute π for x^3, 1 for y, and $\sqrt{2}$ for z in `expr`

```
expr.subs({x**3: pi, y: 1, z: sqrt(2)})
```

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.

`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.

`sympify` (aka `S`) converts its argument (a `str`ing, a numeric type, a `bool`ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts `str`ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.

`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.

`sympify` (aka `S`) converts its argument (a `str`ing, a numeric type, a `bool`ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts `str`ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.

`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.

`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.
`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.
`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.
`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.
`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.
`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.
`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.
`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.
`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.
`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.
`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.
`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.
`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.
`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.
`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$

Evaluation

Expressions can be converted to floating-point approximations (numbers) using either the `evalf` method or the `N` function.
`N(expr, ...)` is equivalent to `sympify(expr).evalf(...)`.
`sympify` (aka `S`) converts its argument (a **`str`**ing, a numeric type, a **`bool`**ean, ...) to a type that can be used inside SymPy.

Examples: execute:

`cos(1).evalf()`	default accuracy is 15 decimal digits,
`pi.evalf(100)`	but it can be set; SymPy supports arbitrary-precision floating point numbers;
`N(pi, 100)`	similar to the previous one;
`N("1/3", 100)`	thanks to `sympify`, `N` accepts **`str`**ings;
`N(x+pi)`	evaluates only π; the value of x is unknown.

Exercises: evaluate the given formulas to an accuracy of 20 digits:

1. $\sqrt{3} \cdot e^{10} + 1/3$ `(sqrt(3)*exp(10) + S(1)/3).evalf(20)`
2. $5 \cdot x \cdot \pi^2 + \sqrt{8}$ `(5*x*pi**2 + sqrt(8)).evalf(20)`

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$
2. $y = \sqrt{3}$

Evaluation with substitution

To numerically evaluate an expression with a *Symbol at a point*:

- we might use `subs` followed by `evalf` (or `N`), for example:

`cos(x).subs(x, 1).evalf(50)`	$\approx \cos(1)$
`N(sqrt(x).subs(x, 3), 50)`	$\approx \sqrt{3}$

- but it is more efficient and numerically stable to pass the substitution to `evalf` (or `N`) using the `subs` flag, which takes a dictionary of *Symbol: point* pairs. For instance:

`cos(x).evalf(50, subs={x: 1})`	$\approx \cos(1)$
`N(sqrt(x), 50, subs={x: 3})`	$\approx \sqrt{3}$
`(x*y).evalf(subs={y: sqrt(2)})`	$\approx x\sqrt{2}$
`(x*y).evalf(subs={x: 2, y: sqrt(2)})`	$\approx 2\sqrt{2}$
`N("x/y", 100, subs={x: 1, y: 3})`	$\approx 1/3$

Exercises: define `expr = x + y**2` and evaluate it at points:

1. $x = \pi$, $y = \sqrt{8}$ `expr.evalf(subs={x: pi, y: sqrt(8)})`
2. $y = \sqrt{3}$ `expr.evalf(subs={y: sqrt(3)})`

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$

Plotting

The `plot` function plots functions of a single variable.

Examples: execute:

- `plot(x**2)` plots x^2 in the default range $(-10, 10)$;
- `plot(x**2, (x, -2, 5))` plots in the given range $(-2, 5)$;
- `plot(x**2, 3*x, (x, 0, 5))` plots both x^2 and $3x$;
- `plot(x**2, (x,0,5), title="square", ylabel="x*x")` a title of the plot is "square", a label for the y-axis is "x*x";
- `plot(exp(x**2), (x, 0, 5))` plots $\exp(x^2)$;
- `plot(exp(x**2), (x, 0, 5), yscale='log')` uses a logarithmic scale for the y-axis, which increases readability.

Exercise: plot $\sin(x)$ in the range $(-2\pi, 2\pi)$
Code: `plot(sin(x), (x, -2*pi, 2*pi))`

Limits

`limit(f(x), x, p)` computes $\lim_{x \to p} f(x)$ (the limit of `f(x)` at the point `p`), e.g. (execute):

`limit(sin(x)/x, x, 0)`	computes $\lim_{x \to 0} \frac{\sin(x)}{x}$;
`limit(x+y, x, 2)`	computes $\lim_{x \to 2}(x + y)$;
`limit(x**2/exp(x), x, oo)`	computes $\lim_{x \to \infty} \frac{x^2}{e^x}$.

`oo` (the lowercase letter "o" twice) denotes ∞ (infinity).

To evaluate a one-sided limit, pass additional `'+'` or `'-'` argument, e.g. `limit(1/x, x, 0, '+')` computes $\lim_{x \to 0^+} \frac{1}{x}$.

Exercises: compute:

1. $\lim_{x \to 0^-} \frac{1}{x}$
2. $\lim_{x \to 0} \frac{x+y}{x}$
3. $\lim_{x \to -\infty} \sin(x)$

Limits

`limit(f(x), x, p)` computes $\lim_{x\to p} f(x)$ (the limit of `f(x)` at the point `p`), e.g. (execute):

`limit(sin(x)/x, x, 0)`	computes $\lim_{x\to 0} \frac{\sin(x)}{x}$;
`limit(x+y, x, 2)`	computes $\lim_{x\to 2}(x+y)$;
`limit(x**2/exp(x), x, oo)`	computes $\lim_{x\to\infty} \frac{x^2}{e^x}$.

`oo` (the lowercase letter "o" twice) denotes ∞ (infinity).

To evaluate a one-sided limit, pass additional `'+'` or `'-'` argument, e.g. `limit(1/x, x, 0, '+')` computes $\lim_{x\to 0^+} \frac{1}{x}$.

Exercises: compute:

1. $\lim_{x\to 0^-} \frac{1}{x}$
2. $\lim_{x\to 0} \frac{x+y}{x}$
3. $\lim_{x\to -\infty} \sin(x)$

Limits

`limit(f(x), x, p)` computes $\lim_{x \to p} f(x)$ (the limit of `f(x)` at the point `p`), e.g. (execute):

`limit(sin(x)/x, x, 0)`	computes $\lim_{x \to 0} \frac{\sin(x)}{x}$;
`limit(x+y, x, 2)`	computes $\lim_{x \to 2}(x + y)$;
`limit(x**2/exp(x), x, oo)`	computes $\lim_{x \to \infty} \frac{x^2}{e^x}$.

`oo` (the lowercase letter "o" twice) denotes ∞ (infinity).

To evaluate a one-sided limit, pass additional `'+'` or `'-'` argument, e.g. `limit(1/x, x, 0, '+')` computes $\lim_{x \to 0^+} \frac{1}{x}$.

Exercises: compute:

1. $\lim_{x \to 0^-} \frac{1}{x}$
2. $\lim_{x \to 0} \frac{x+y}{x}$
3. $\lim_{x \to -\infty} \sin(x)$

Limits

`limit(f(x), x, p)` computes $\lim_{x \to p} f(x)$ (the limit of `f(x)` at the point `p`), e.g. (execute):

`limit(sin(x)/x, x, 0)`	computes $\lim_{x \to 0} \frac{\sin(x)}{x}$;
`limit(x+y, x, 2)`	computes $\lim_{x \to 2}(x + y)$;
`limit(x**2/exp(x), x, oo)`	computes $\lim_{x \to \infty} \frac{x^2}{e^x}$.

`oo` (the lowercase letter "o" twice) denotes ∞ (infinity).

To evaluate a one-sided limit, pass additional `'+'` or `'-'` argument, e.g. `limit(1/x, x, 0, '+')` computes $\lim_{x \to 0^+} \frac{1}{x}$.

Exercises: compute:

1. $\lim_{x \to 0^-} \frac{1}{x}$
2. $\lim_{x \to 0} \frac{x+y}{x}$
3. $\lim_{x \to -\infty} \sin(x)$

Limits

`limit(f(x), x, p)` computes $\lim_{x \to p} f(x)$ (the limit of `f(x)` at the point `p`), e.g. (execute):

`limit(sin(x)/x, x, 0)`	computes $\lim_{x \to 0} \frac{\sin(x)}{x}$;
`limit(x+y, x, 2)`	computes $\lim_{x \to 2}(x + y)$;
`limit(x**2/exp(x), x, oo)`	computes $\lim_{x \to \infty} \frac{x^2}{e^x}$.

`oo` (the lowercase letter "o" twice) denotes ∞ (infinity).

To evaluate a one-sided limit, pass additional `'+'` or `'-'` argument, e.g. `limit(1/x, x, 0, '+')` computes $\lim_{x \to 0^+} \frac{1}{x}$.

Exercises: compute:

1. $\lim_{x \to 0^-} \frac{1}{x}$
2. $\lim_{x \to 0} \frac{x+y}{x}$
3. $\lim_{x \to -\infty} \sin(x)$

Limits

`limit(f(x), x, p)` computes $\lim_{x \to p} f(x)$ (the limit of `f(x)` at the point `p`), e.g. (execute):

`limit(sin(x)/x, x, 0)`	computes $\lim_{x \to 0} \frac{\sin(x)}{x}$;
`limit(x+y, x, 2)`	computes $\lim_{x \to 2}(x + y)$;
`limit(x**2/exp(x), x, oo)`	computes $\lim_{x \to \infty} \frac{x^2}{e^x}$.

`oo` (the lowercase letter "o" twice) denotes ∞ (infinity).

To evaluate a one-sided limit, pass additional `'+'` or `'-'` argument, e.g. `limit(1/x, x, 0, '+')` computes $\lim_{x \to 0^+} \frac{1}{x}$.

Exercises: compute:

1. $\lim_{x \to 0^-} \frac{1}{x}$
2. $\lim_{x \to 0} \frac{x+y}{x}$
3. $\lim_{x \to -\infty} \sin(x)$

Limits

`limit(f(x), x, p)` computes $\lim_{x\to p} f(x)$ (the limit of `f(x)` at the point `p`), e.g. (execute):

`limit(sin(x)/x, x, 0)`	computes $\lim_{x\to 0} \frac{\sin(x)}{x}$;
`limit(x+y, x, 2)`	computes $\lim_{x\to 2}(x+y)$;
`limit(x**2/exp(x), x, oo)`	computes $\lim_{x\to\infty} \frac{x^2}{e^x}$.

`oo` (the lowercase letter "o" twice) denotes ∞ (infinity).

To evaluate a one-sided limit, pass additional `'+'` or `'-'` argument, e.g. `limit(1/x, x, 0, '+')` computes $\lim_{x\to 0^+} \frac{1}{x}$.

Exercises: compute:

1. $\lim_{x\to 0^-} \frac{1}{x}$ Code: `limit(1/x, x, 0, '-')`
2. $\lim_{x\to 0} \frac{x+y}{x}$ Code: `limit((x+y)/x, x, 0)`
3. $\lim_{x\to -\infty} \sin(x)$ Code: `limit(sin(x), x, -oo)`

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)

Sums and products

`summation(f, (i, a, b))` computes $\sum_{i=a}^{b} f(i)$ (the sum of `f` with respect to `i` from `a` to `b`);
`product(f, (i, a, b))` computes $\prod_{i=a}^{b} f(i)$ (the product of `f` with respect to `i` from `a` to `b`).

Example: execute:
`i,n = symbols('i,n', integer=True)` defines integer symbols;
`summation(1/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots$;
`summation(x/(2**i), (i, 1, oo))` $= \sum_{i=1}^{\infty} \frac{x}{2^i} = \frac{x}{2} + \frac{x}{4} + \ldots$;
`product(n**n, (n, 5, 20))` $\prod_{n=5}^{20} n^n = 5^5 \cdot 6^6 \cdot \ldots \cdot 20^{20}$;
`product(i, (i, 1, n))` $\prod_{i=1}^{n} i = n!$.

Exercise: calculate:

1. $\prod_{i=1}^{10} n^i$ `product(n**i, (i, 1, 10))`
2. $\sum_{n=1}^{\infty} \frac{6}{n^2}$ `summation(6/n**2, (n, 1, oo))`
3. $\sum_{n=1}^{\infty} \frac{x^n}{n!}$ (simplify the result expression)
   `summation(x**n/factorial(n), (n, 1, oo)).simplify()`

Solving inequalities and equations algebraically

`solveset(f, symbol, domain)` solves the inequality or equation `f` over `domain` (`S.Complexes` by default); returns a set of all values for `symbol` for which `f` is `True` or is equal to zero.

Examples: execute:

- `solveset(x**2>2, x, S.Reals)` solves $x^2 > 2$ for real x;
- `solveset(Eq(x**4, 2), x)` and `solveset(x**4-2, x)` both solve $x^4 = 2$; equations may be in the form of `Eq` instances or expressions that are assumed to be equal to 0;
- `a,b,c = symbols('a,b,c')` `simplify(solveset(a*x**2 + b*x + c, x))` finds (and simplifies) the general solution of quadratic equation.

Exercises: find all real x such that:

1. $x^4 = 2$
2. $\sin(x) = \cos(x)$
3. $x^2 > 2 \ \wedge \ x < 5$ (hint: use `&` to intersect `solveset` results)

Solving inequalities and equations algebraically

`solveset(f, symbol, domain)` solves the inequality or equation `f` over `domain` (`S.Complexes` by default); returns a set of all values for `symbol` for which `f` is `True` or is equal to zero.

Examples: execute:

- `solveset(x**2>2, x, S.Reals)` solves $x^2 > 2$ for real x;
- `solveset(Eq(x**4, 2), x)` and `solveset(x**4-2, x)` both solve $x^4 = 2$; equations may be in the form of `Eq` instances or expressions that are assumed to be equal to 0;
- `a,b,c = symbols('a,b,c')`
  `simplify(solveset(a*x**2 + b*x + c, x))` finds (and simplifies) the general solution of quadratic equation.

Exercises: find all real x such that:

1. $x^4 = 2$
2. $\sin(x) = \cos(x)$
3. $x^2 > 2 \ \wedge \ x < 5$ (hint: use `&` to intersect `solveset` results)

Solving inequalities and equations algebraically

`solveset(f, symbol, domain)` solves the inequality or equation `f` over `domain` (`S.Complexes` by default); returns a set of all values for `symbol` for which `f` is `True` or is equal to zero.

Examples: execute:

- `solveset(x**2>2, x, S.Reals)` solves $x^2 > 2$ for real x;
- `solveset(Eq(x**4, 2), x)` and `solveset(x**4-2, x)` both solve $x^4 = 2$; equations may be in the form of `Eq` instances or expressions that are assumed to be equal to 0;
- `a,b,c = symbols('a,b,c')`
  `simplify(solveset(a*x**2 + b*x + c, x))` finds (and simplifies) the general solution of quadratic equation.

Exercises: find all real x such that:

1. $x^4 = 2$
2. $\sin(x) = \cos(x)$
3. $x^2 > 2 \wedge x < 5$ (hint: use `&` to intersect `solveset` results)

Solving inequalities and equations algebraically

`solveset(f, symbol, domain)` solves the inequality or equation `f` over `domain` (`S.Complexes` by default); returns a set of all values for `symbol` for which `f` is `True` or is equal to zero.

Examples: execute:

- `solveset(x**2>2, x, S.Reals)` solves $x^2 > 2$ for real x;
- `solveset(Eq(x**4, 2), x)` and `solveset(x**4-2, x)` both solve $x^4 = 2$; equations may be in the form of `Eq` instances or expressions that are assumed to be equal to 0;
- `a,b,c = symbols('a,b,c')`
  `simplify(solveset(a*x**2 + b*x + c, x))` finds (and simplifies) the general solution of quadratic equation.

Exercises: find all real x such that:

1. $x^4 = 2$
2. $\sin(x) = \cos(x)$
3. $x^2 > 2 \wedge x < 5$ (hint: use `&` to intersect `solveset` results)

Solving inequalities and equations algebraically

`solveset(f, symbol, domain)` solves the inequality or equation `f` over `domain` (`S.Complexes` by default); returns a set of all values for `symbol` for which `f` is `True` or is equal to zero.

Examples: execute:

- `solveset(x**2>2, x, S.Reals)` solves $x^2 > 2$ for real x;
- `solveset(Eq(x**4, 2), x)` and `solveset(x**4-2, x)` both solve $x^4 = 2$; equations may be in the form of `Eq` instances or expressions that are assumed to be equal to 0;
- `a,b,c = symbols('a,b,c')`
  `simplify(solveset(a*x**2 + b*x + c, x))` finds (and simplifies) the general solution of quadratic equation.

Exercises: find all real x such that:

1. $x^4 = 2$
2. $\sin(x) = \cos(x)$
3. $x^2 > 2 \ \wedge \ x < 5$ (hint: use `&` to intersect `solveset` results)

Solving inequalities and equations algebraically

`solveset(f, symbol, domain)` solves the inequality or equation `f` over `domain` (`S.Complexes` by default); returns a set of all values for `symbol` for which `f` is `True` or is equal to zero.

Examples: execute:

- `solveset(x**2>2, x, S.Reals)` solves $x^2 > 2$ for real x;
- `solveset(Eq(x**4, 2), x)` and `solveset(x**4-2, x)` both solve $x^4 = 2$; equations may be in the form of `Eq` instances or expressions that are assumed to be equal to 0;
- `a,b,c = symbols('a,b,c')`
  `simplify(solveset(a*x**2 + b*x + c, x))` finds (and simplifies) the general solution of quadratic equation.

Exercises: find all real x such that:

1. $x^4 = 2$ `solveset(x**4 - 2, x, S.Reals)`
2. $\sin(x) = \cos(x)$ `solveset(sin(x)-cos(x), x, S.Reals)`
3. $x^2 > 2 \ \wedge \ x < 5$ (hint: use `&` to intersect `solveset` results)
   `solveset(x**2>2,x,S.Reals)&solveset(x<5,x,S.Reals)`

Derivatives

To take derivatives, use the `diff` function, e.g. `diff(cos(x), x)` differentiates $\cos(x)$ with respect to x; $\frac{\partial}{\partial x}\cos(x)$.

To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example both `diff(x**4, x, x, x)` and `diff(x**4, x, 3)` find the third derivative of x^4.

To take derivatives with respect to many variables at once, just pass each derivative in order, e.g. each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$:

```
diff(exp(x*y*z), x, y, y, z, z, z, z)
diff(exp(x*y*z), x, y, 2, z, 4)
diff(exp(x*y*z), x, y, y, z, 4)
```

Exercise: differentiates $x^3 + 2y$ with respect to:

1. x
2. y

Derivatives

To take derivatives, use the `diff` function, e.g. `diff(cos(x), x)` differentiates $\cos(x)$ with respect to x; $\frac{\partial}{\partial x}\cos(x)$.

To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example both `diff(x**4, x, x, x)` and `diff(x**4, x, 3)` find the third derivative of x^4.

To take derivatives with respect to many variables at once, just pass each derivative in order, e.g. each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$:

```
diff(exp(x*y*z), x, y, y, z, z, z, z)
diff(exp(x*y*z), x, y, 2, z, 4)
diff(exp(x*y*z), x, y, y, z, 4)
```

Exercise: differentiates $x^3 + 2y$ with respect to:

1. x
2. y

Derivatives

To take derivatives, use the `diff` function, e.g. `diff(cos(x), x)` differentiates $\cos(x)$ with respect to x; $\frac{\partial}{\partial x}\cos(x)$.

To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example both `diff(x**4, x, x, x)` and `diff(x**4, x, 3)` find the third derivative of x^4.

To take derivatives with respect to many variables at once, just pass each derivative in order, e.g. each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$:

```
diff(exp(x*y*z), x, y, y, z, z, z, z)
diff(exp(x*y*z), x, y, 2, z, 4)
diff(exp(x*y*z), x, y, y, z, 4)
```

Exercise: differentiates $x^3 + 2y$ with respect to:

1. x
2. y

Derivatives

To take derivatives, use the `diff` function, e.g. `diff(cos(x), x)` differentiates $\cos(x)$ with respect to x; $\frac{\partial}{\partial x}\cos(x)$.

To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example both `diff(x**4, x, x, x)` and `diff(x**4, x, 3)` find the third derivative of x^4.

To take derivatives with respect to many variables at once, just pass each derivative in order, e.g. each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$:

```
diff(exp(x*y*z), x, y, y, z, z, z, z)
diff(exp(x*y*z), x, y, 2, z, 4)
diff(exp(x*y*z), x, y, y, z, 4)
```

Exercise: differentiates $x^3 + 2y$ with respect to:

1. x
2. y

Derivatives

To take derivatives, use the `diff` function, e.g. `diff(cos(x), x)` differentiates $\cos(x)$ with respect to x; $\frac{\partial}{\partial x}\cos(x)$.

To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example both `diff(x**4, x, x, x)` and `diff(x**4, x, 3)` find the third derivative of x^4.

To take derivatives with respect to many variables at once, just pass each derivative in order, e.g. each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$:

```
diff(exp(x*y*z), x, y, y, z, z, z, z)
diff(exp(x*y*z), x, y, 2, z, 4)
diff(exp(x*y*z), x, y, y, z, 4)
```

Exercise: differentiates $x^3 + 2y$ with respect to:

1. x
2. y

Derivatives

To take derivatives, use the `diff` function, e.g. `diff(cos(x), x)` differentiates $\cos(x)$ with respect to x; $\frac{\partial}{\partial x}\cos(x)$.

To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example both `diff(x**4, x, x, x)` and `diff(x**4, x, 3)` find the third derivative of x^4.

To take derivatives with respect to many variables at once, just pass each derivative in order, e.g. each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$:

```
diff(exp(x*y*z), x, y, y, z, z, z, z)
diff(exp(x*y*z), x, y, 2, z, 4)
diff(exp(x*y*z), x, y, y, z, 4)
```

Exercise: differentiates $x^3 + 2y$ with respect to:

1. x
2. y

Derivatives

To take derivatives, use the `diff` function, e.g. `diff(cos(x), x)` differentiates $\cos(x)$ with respect to x; $\frac{\partial}{\partial x}\cos(x)$.

To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example both `diff(x**4, x, x, x)` and `diff(x**4, x, 3)` find the third derivative of x^4.

To take derivatives with respect to many variables at once, just pass each derivative in order, e.g. each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$:

```
diff(exp(x*y*z), x, y, y, z, z, z, z)
diff(exp(x*y*z), x, y, 2, z, 4)
diff(exp(x*y*z), x, y, y, z, 4)
```

Exercise: differentiates $x^3 + 2y$ with respect to:

1. x Code: `diff(x**3+2*y, x)`
2. y Code: `diff(x**3+2*y, y)`

Integrals

To compute an integral, use the `integrate` function.

To compute an indefinite integral, just pass the variable after the expression, e.g. `integrate(cos(x), x)` calculates $\int \cos(x)\,dx$. (Note that SymPy does not include the constant of integration.)

To compute a definite integral, pass a tuple `(integration_variable, lower_limit, upper_limit)`, e.g. `integrate(exp(-x), (x, 0, oo))` calculates $\int_0^\infty e^{-x}\,dx$.

You can pass multiple limit tuples to perform a multiple integral, e.g. `integrate(exp(-x**2-y**2),(x,-oo,oo),(y,-oo,oo))` computes $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-x^2-y^2}\,dx\,dy$.

Exercises: calculate:

1. $\int x\cos(x)\,dx$
2. $\int_0^{2\pi} \sin(x)\,dx$
3. $\int_1^\infty \frac{1}{x}\,dx$

Integrals

To compute an integral, use the `integrate` function.

To compute an indefinite integral, just pass the variable after the expression, e.g. `integrate(cos(x), x)` calculates $\int \cos(x)\,dx$. (Note that SymPy does not include the constant of integration.)

To compute a definite integral, pass a tuple `(integration_variable, lower_limit, upper_limit)`, e.g. `integrate(exp(-x), (x, 0, oo))` calculates $\int_0^\infty e^{-x}\,dx$.

You can pass multiple limit tuples to perform a multiple integral, e.g. `integrate(exp(-x**2-y**2),(x,-oo,oo),(y,-oo,oo))` computes $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-x^2-y^2}\,dx\,dy$.

Exercises: calculate:

1. $\int x\cos(x)\,dx$
2. $\int_0^{2\pi} \sin(x)\,dx$
3. $\int_1^\infty \frac{1}{x}\,dx$

Integrals

To compute an integral, use the `integrate` function.

To compute an indefinite integral, just pass the variable after the expression, e.g. `integrate(cos(x), x)` calculates $\int \cos(x)\,dx$. (Note that SymPy does not include the constant of integration.)

To compute a definite integral, pass a tuple `(integration_variable, lower_limit, upper_limit)`, e.g. `integrate(exp(-x), (x, 0, oo))` calculates $\int_0^\infty e^{-x}\,dx$.

You can pass multiple limit tuples to perform a multiple integral, e.g. `integrate(exp(-x**2-y**2),(x,-oo,oo),(y,-oo,oo))` computes $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-x^2-y^2}\,dx\,dy$.

Exercises: calculate:

1. $\int x\cos(x)\,dx$
2. $\int_0^{2\pi} \sin(x)\,dx$
3. $\int_1^\infty \frac{1}{x}\,dx$

Integrals

To compute an integral, use the `integrate` function.

To compute an indefinite integral, just pass the variable after the expression, e.g. `integrate(cos(x), x)` calculates $\int \cos(x)\,dx$. (Note that SymPy does not include the constant of integration.)

To compute a definite integral, pass a tuple `(integration_variable, lower_limit, upper_limit)`, e.g. `integrate(exp(-x), (x, 0, oo))` calculates $\int_0^\infty e^{-x}\,dx$.

You can pass multiple limit tuples to perform a multiple integral, e.g. `integrate(exp(-x**2-y**2),(x,-oo,oo),(y,-oo,oo))` computes $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-x^2-y^2}\,dx\,dy$.

Exercises: calculate:

1. $\int x\cos(x)\,dx$
2. $\int_0^{2\pi} \sin(x)\,dx$
3. $\int_1^\infty \frac{1}{x}\,dx$

Integrals

To compute an integral, use the `integrate` function.

To compute an indefinite integral, just pass the variable after the expression, e.g. `integrate(cos(x), x)` calculates $\int \cos(x)\,dx$. (Note that SymPy does not include the constant of integration.)

To compute a definite integral, pass a tuple `(integration_variable, lower_limit, upper_limit)`, e.g. `integrate(exp(-x), (x, 0, oo))` calculates $\int_0^\infty e^{-x}\,dx$.

You can pass multiple limit tuples to perform a multiple integral, e.g. `integrate(exp(-x**2-y**2),(x,-oo,oo),(y,-oo,oo))` computes $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-x^2-y^2}\,dx\,dy$.

Exercises: calculate:

1. $\int x\cos(x)\,dx$
2. $\int_0^{2\pi} \sin(x)\,dx$
3. $\int_1^\infty \frac{1}{x}\,dx$

Integrals

To compute an integral, use the `integrate` function.

To compute an indefinite integral, just pass the variable after the expression, e.g. `integrate(cos(x), x)` calculates $\int \cos(x)\,dx$. (Note that SymPy does not include the constant of integration.)

To compute a definite integral, pass a tuple `(integration_variable, lower_limit, upper_limit)`, e.g. `integrate(exp(-x), (x, 0, oo))` calculates $\int_0^\infty e^{-x}\,dx$.

You can pass multiple limit tuples to perform a multiple integral, e.g. `integrate(exp(-x**2-y**2),(x,-oo,oo),(y,-oo,oo))` computes $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-x^2-y^2}\,dx\,dy$.

Exercises: calculate:

1. $\int x\cos(x)\,dx$ Code: `integrate(x*cos(x), x)`
2. $\int_0^{2\pi} \sin(x)\,dx$ Code: `integrate(sin(x), (x, 0, 2*pi))`
3. $\int_1^\infty \frac{1}{x}\,dx$ Code: `integrate(1/x, (x, 1, oo))`

Unevaluated objects

- SymPy contains classes whose instances (so called *unevaluated objects*) represent unevaluated operations, for example:
  - `Derivative`
    (its constructor takes the same arguments as `diff`);
  - `Integral`
    (its constructor takes the same arguments as `integrate`);
  - `Sum`
    (its constructor takes the same arguments as `summation`);
- unevaluated objects are useful for delaying the evaluation, or for printing purposes, for instance (execute):

```
der = Derivative(x**3 * y**2, x, x, y)
```

`der`	displays `der`,
`latex(der)`	LaTeX representation of `der`;
`der.doit()`	solves `der`.

- unevaluated objects are also returned by `diff`, `integrate`, etc. when SymPy does not know how to compute something. For example try: `integrate(x**x, x)`

Unevaluated objects

- SymPy contains classes whose instances (so called *unevaluated objects*) represent unevaluated operations, for example:
  - `Derivative`
    (its constructor takes the same arguments as `diff`);
  - `Integral`
    (its constructor takes the same arguments as `integrate`);
  - `Sum`
    (its constructor takes the same arguments as `summation`);
- unevaluated objects are useful for delaying the evaluation, or for printing purposes, for instance (execute):

```
der = Derivative(x**3 * y**2, x, x, y)
```

`der`	displays `der`,
`latex(der)`	LaTeX representation of `der`;
`der.doit()`	solves `der`.

- unevaluated objects are also returned by `diff`, `integrate`, etc. when SymPy does not know how to compute something. For example try: `integrate(x**x, x)`

Unevaluated objects

- SymPy contains classes whose instances (so called *unevaluated objects*) represent unevaluated operations, for example:
  - `Derivative`
    (its constructor takes the same arguments as `diff`);
  - `Integral`
    (its constructor takes the same arguments as `integrate`);
  - `Sum`
    (its constructor takes the same arguments as `summation`);
- unevaluated objects are useful for delaying the evaluation, or for printing purposes, for instance (execute):

```
der = Derivative(x**3 * y**2, x, x, y)
```

`der`	displays `der`,
`latex(der)`	LaTeX representation of `der`;
`der.doit()`	solves `der`.

- unevaluated objects are also returned by `diff`, `integrate`, etc. when SymPy does not know how to compute something. For example try: `integrate(x**x, x)`

Homework

1. simplify the formula $\frac{x^2+x}{x\sin^2(y)+x\cos^2(y)}$;
2. calculate $\lim_{x\to-\infty}\left(\frac{1}{x}\ln\left(e^x+1\right)\right)$;
3. find all real x such that $x^2+2>x^3 \ \wedge \ \sqrt{x+2}>x$;
4. solve $ax^3+bx^2+cx+d=0$ for x;
5. evaluate $\frac{\partial^2}{\partial x\partial y}\left(\sin\left(x+y\right)\cos\left(xy\right)\right)$ to an accuracy of 30 decimal digits at point $x=-1$, $y=2$;
6. plot both $f(x)=3x^2$ and $\frac{\partial}{\partial x}f(x)$ together in the range $(0,5)$;
7. plot $f(x)=\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^{nx}$ in the range $(-1,1)$;
8. evaluate $\int_{-\infty}^{\infty}e^{-x^2}\,dx$ to an accuracy of 50 digits;
9. plot $\int e^{-x^2}\,dx$ in the range $(-4,4)$;
10. for volunteers: find a simple formula for the area of a triangle with vertices at points: $(0,0)$, (a,b), (c,d). Next, evaluate the formula for $a=2$, $b=2$, $c=4$, $d=0$.

Please note all the expressions you used.

Bibliography

- Official *SymPy's documentation* (especially *SymPy Tutorial*), available on http://docs.sympy.org/
- Jonathan Gross *Math and Physics with SymPy*, available on http://www.jonathangross.de/files/IPCS2016/sympy.pdf
- Paul Lutus *IPython: Math Processor*, available on http://arachnoid.com/IPython/