Mathics

A free, light-weight alternative to Mathematica

The Mathics Team

March 1, 2016

Contents

Part I.

Manual

1. Introduction

Mathics—to be pronounced like "Mathematics" without the "emat"—is a general-purpose computer algebra system (CAS). It is meant to be a free, light-weight alternative to *Mathematica*®. It is free both as in "free beer" and as in "freedom". There are various online mirrors running *Mathics* but it is also possible to run *Mathics* locally. A list of mirrors can be found at the *Mathics* homepage, `http://mathics.github.io`.

The programming language of *Mathics* is meant to resemble *Wolfram*'s famous *Mathematica*® as much as possible. However, *Mathics* is in no way affiliated or supported by *Wolfram*. *Mathics* will probably never have the power to compete with *Mathematica*® in industrial applications; yet, it might be an interesting alternative for educational purposes.

Contents

Why yet another CAS?

Mathematica® is great, but it has one big disadvantage: It is not free. On the one hand, people might not be able or willing to pay hundreds of dollars for it; on the other hand, they would still not be able to see what's going on "inside" the program to understand their computations better. That's what free software is for!

Mathics aims at combining the best of both worlds: the beauty of *Mathematica*® backed by a free, extensible Python core.

Of course, there are drawbacks to the *Mathematica*® language, despite all its beauty. It does not really provide object orientation and especially encapsulation, which might be crucial for big software projects. Nevertheless, *Wolfram* still managed to create their amazing *Wolfram | Alpha* entirely with *Mathematica*®, so it can't be too bad!

However, it is not even the intention of *Mathics* to be used in large-scale projects and calculations—at least not as the main framework—but rather as a tool for quick explorations and in educating people who might later switch to *Mathematica*®.

What does it offer?

Some of the most important features of *Mathics* are

- a powerful functional programming language,
- a system driven by pattern matching and rules application,
- rationals, complex numbers, and arbitrary-precision arithmetic,
- lots of list and structure manipulation routines,
- an interactive graphical user interface right in the Web browser using MathML (apart from a command line interface),

- creation of graphics (e.g. plots) and display in the browser using SVG for 2D graphics and WebGL for 3D graphics,
- export of results to $\LaTeX$ (using Asymptote for graphics),
- a very easy way of defining new functions in Python,
- an integrated documentation and testing system.

What is missing?

There are lots of ways in which *Mathics* could still be improved.

Most notably, performance is still very slow, so any serious usage in cutting-edge industry or research will fail, unfortunately. Speeding up pattern matching, maybe "outsourcing" parts of it from Python to C, would certainly improve the whole *Mathics* experience.

Apart from performance issues, new features such as more functions in various mathematical fields like calculus, number theory, or graph theory are still to be added.

Who is behind it?

Mathics was created by Jan Pöschko. Since 2013 it has been maintained by Angus Griffith. A list of all people involved in *Mathics* can be found in the `AUTHORS` file.

If you have any ideas on how to improve *Mathics* or even want to help out yourself, please contact us!

Welcome to *Mathics*, have fun!

2. Installation

Contents

Browser requirements

To use the online version of *Mathics* at `http://www.mathics.net` or a different location (in fact, anybody could run their own version), you need a decent version of a modern Web browser, such as Firefox, Chrome, or Safari. Internet Explorer, even with its relatively new version 9, lacks support for modern Web standards; while you might be able to enter queries and view results, the whole layout of *Mathics* is a mess in Internet Explorer. There might be better support in the future, but this does not have very high priority. Opera is not supported "officially" as it obviously has some problems with mathematical text inside SVG graphics, but except from that everything should work pretty fine.

Installation prerequisites

To run *Mathics*, you need Python 2.7 or higher on your computer. Since version 0.9 *Mathics* also supports Python3. On most Linux distributions and on Mac OS X, Python is already included in the system by default. For Windows, you can get it from `http://www.python.org`. Anyway, the primary target platforms for *Mathics* are Linux (especially Debian and Ubuntu) and Mac OS X. If you are on Windows and want to help by providing an installer to make setup on Windows easier, feel very welcome!

Furthermore, SQLite support is needed. Debian/Ubuntu provides the package `libsqlite3-dev`. The packages `python-dev` and `python-setuptools` are needed as well. You can install all required packages by running

```
# apt-get install python-dev
  libsqlite3-dev python-
  setuptools
```

(as super-user, i.e. either after having issued `su` or by preceding the command with `sudo`).

On Mac OS X, consider using Fink (`http://www.finkproject.org`) and install the `sqlite3-dev` package.

If you are on Windows, please figure out yourself how to install SQLite.

Get the latest version of *Mathics* from `http://www.mathics.github.io`. You will need internet access for the installation of *Mathics*.

Setup

Simply run:

```
# python setup.py install
```

In addition to installing *Mathics*, this will download the required Python packages `sympy`, `mpmath`, `django`, and `pysqlite` and install them in your

Python `site-packages` directory (usually `/usr/lib/python2.x/site-packages` on Debian or `/Library/Frameworks/Python.framework/Versions/2.x/lib/python2.x/site-packages` on Mac OS X).
Two executable files will be created in a binary directory on your `PATH` (usually `/usr/bin` on Debian or `/Library/Frameworks/Python.framework/Versions/2.x/bin` on Mac OS X): `mathics` and `mathicsserver`.

Running *Mathics*

Run

```
$ mathics
```

to start the console version of *Mathics*.
Run

```
$ mathicsserver
```

to start the local Web server of *Mathics* which serves the web GUI interface. The first time this command is run it will create the database file for saving your sessions. Issue

```
$ mathicsserver --help
```

to see a list of options.
You can set the used port by using the option `-p`, as in:

```
$ mathicsserver -p 8010
```

The default port for *Mathics* is 8000. Make sure you have the necessary privileges to start an application that listens to this port. Otherwise, you will have to run *Mathics* as super-user.
By default, the Web server is only reachable from your local machine. To be able to access it from another computer, use the option `-e`. However, the server is only intended for local use, as it is a security risk to run it openly on a public Web server! This documentation does not cover how to setup *Mathics* for being used on a public server. Maybe you want to hire a *Mathics* developer to do that for you?!

3. Language tutorials

The following sections are introductions to the basic principles of the language of *Mathics*. A few examples and functions are presented. Only their most common usages are listed; for a full description of their possible arguments, options, etc., see their entry in the Reference of built-in symbols.

Contents

Basic calculations

Mathics can be used to calculate basic stuff:

```
>> 1 + 2
   3
```

To submit a command to *Mathics*, press `Shift+Return` in the Web interface or `Return` in the console interface. The result will be printed in a new line below your query.
Mathics understands all basic arithmetic operators and applies the usual operator precedence. Use parentheses when needed:

```
>> 1 - 2 * (3 + 5)/ 4
   −3
```

The multiplication can be omitted:

```
>> 1 - 2 (3 + 5)/ 4
   −3
```

```
>> 2 4
   8
```

Powers can be entered using `^`:

```
>> 3 ^ 4
   81
```

Integer divisions yield rational numbers:

```
>> 6 / 4
```

$\frac{3}{2}$

To convert the result to a floating point number, apply the function `N`:

```
>> N[6 / 4]
   1.5
```

As you can see, functions are applied using square braces `[` and `]`, in contrast to the common notation of `(` and `)`. At first hand, this might seem strange, but this distinction between function application and precedence change is necessary to allow some general syntax structures, as you will see later.
Mathics provides many common mathematical functions and constants, e.g.:

```
>> Log[E]
   1
```

```
>> Sin[Pi]
```
0

```
>> Cos[0.5]
```
0.877582561890372716

When entering floating point numbers in your query, *Mathics* will perform a numerical evaluation and present a numerical result, pretty much like if you had applied `N`.

Of course, *Mathics* has complex numbers:

```
>> Sqrt[-4]
```
$2I$

```
>> I ^ 2
```
-1

```
>> (3 + 2 I)^ 4
```
$-119 + 120I$

```
>> (3 + 2 I)^ (2.5 - I)
```
$43.6630044263147016 + 8.28556100627573406I$

```
>> Tan[I + 0.5]
```
$0.195577310065933999 + 0.842966204845783229I$

`Abs` calculates absolute values:

```
>> Abs[-3]
```
3

```
>> Abs[3 + 4 I]
```
5

Mathics can operate with pretty huge numbers:

```
>> 100!
```
93 326 215 443 944 152 681 699 ˜
˜238 856 266 700 490 715 968 ˜
˜264 381 621 468 592 963 895 ˜
˜217 599 993 229 915 608 941 ˜
˜463 976 156 518 286 253 697 920 ˜
˜827 223 758 251 185 210 916 864 ˜
˜000 000 000 000 000 000 000 000

(`!` denotes the factorial function.) The precision of numerical evaluation can be set:

```
>> N[Pi, 100]
```
3.1415926535897932384626433˜
˜832795028841971693993751˜
˜058209749445923078164062˜
˜862089986280348253421170679

Division by zero is forbidden:

```
>> 1 / 0
```
Infinite expression (division by zero) encountered.

ComplexInfinity

Other expressions involving `Infinity` are evaluated:

```
>> Infinity + 2 Infinity
```
∞

In contrast to combinatorial belief, `0^0` is undefined:

```
>> 0 ^ 0
```
Indeterminate expression 0^0 encountered.

Indeterminate

The result of the previous query to *Mathics* can be accessed by `%`:

```
>> 3 + 4
```
7

```
>> % ^ 2
```
49

Symbols and assignments

Symbols need not be declared in *Mathics*, they can just be entered and remain variable:

```
>> x
```
x

Basic simplifications are performed:

```
>> x + 2 x
```
$3x$

Symbols can have any name that consists of characters and digits:

```
>> iAm1Symbol ^ 2
```
$\mathrm{iAm1Symbol}^2$

You can assign values to symbols:

```
>> a = 2
   2
>> a ^ 3
   8
>> a = 4
   4
>> a ^ 3
   64
```

Assigning a value returns that value. If you want to suppress the output of any result, add a ; to the end of your query:

```
>> a = 4;
```

Values can be copied from one variable to another:

```
>> b = a;
```

Now changing a does not affect b:

```
>> a = 3;

>> b
   4
```

Such a dependency can be achieved by using "delayed assignment" with the := operator (which does not return anything, as the right side is not even evaluated):

```
>> b := a ^ 2

>> b
   9
>> a = 5;

>> b
   25
```

Comparisons and Boolean logic

Values can be compared for equality using the operator ==:

```
>> 3 == 3
   True
>> 3 == 4
   False
```

The special symbols `True` and `False` are used to denote truth values. Naturally, there are inequality comparisons as well:

```
>> 3 > 4
   False
```

Inequalities can be chained:

```
>> 3 < 4 >= 2 != 1
   True
```

Truth values can be negated using `!` (logical *not*) and combined using `&&` (logical *and*) and `||` (logical *or*):

```
>> !True
   False
>> !False
   True
>> 3 < 4 && 6 > 5
   True
```

`&&` has higher precedence than `||`, i.e. it binds stronger:

```
>> True && True || False &&
   False
   True
>> True && (True || False)&&
   False
   False
```

Strings

Strings can be entered with `"` as delimeters:

```
>> "Hello world!"
   Hello world!
```

As you can see, quotation marks are not printed in the output by default. This can be changed by using `InputForm`:

```
>> InputForm["Hello world!"]
   "Hello world!"
```

Strings can be joined using `<>`:

```
>> "Hello" <> " " <> "world!"
```
Hello world!

Numbers cannot be joined to strings:

```
>> "Debian" <> 6
```
String expected.

Debian<>6

They have to be converted to strings using `ToString` first:

```
>> "Debian" <> ToString[6]
```
Debian6

Lists

Lists can be entered in *Mathics* with curly braces { and }:

```
>> mylist = {a, b, c, d}
```
$\{a, b, c, d\}$

There are various functions for constructing lists:

```
>> Range[5]
```
$\{1, 2, 3, 4, 5\}$

```
>> Array[f, 4]
```
$\{f[1], f[2], f[3], f[4]\}$

```
>> ConstantArray[x, 4]
```
$\{x, x, x, x\}$

```
>> Table[n ^ 2, {n, 2, 5}]
```
$\{4, 9, 16, 25\}$

The number of elements of a list can be determined with `Length`:

```
>> Length[mylist]
```
4

Elements can be extracted using double square braces:

```
>> mylist[[3]]
```
c

Negative indices count from the end:

```
>> mylist[[-3]]
```
b

Lists can be nested:

```
>> mymatrix = {{1, 2}, {3, 4},
   {5, 6}};
```

There are alternate forms to display lists:

```
>> TableForm[mymatrix]
```

1	2
3	4
5	6

```
>> MatrixForm[mymatrix]
```
$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

There are various ways of extracting elements from a list:

```
>> mymatrix[[2, 1]]
```
3

```
>> mymatrix[[;;, 2]]
```
$\{2, 4, 6\}$

```
>> Take[mylist, 3]
```
$\{a, b, c\}$

```
>> Take[mylist, -2]
```
$\{c, d\}$

```
>> Drop[mylist, 2]
```
$\{c, d\}$

```
>> First[mymatrix]
```
$\{1, 2\}$

```
>> Last[mylist]
```
d

```
>> Most[mylist]
```
$\{a, b, c\}$

```
>> Rest[mylist]
```
$\{b, c, d\}$

Lists can be used to assign values to multiple variables at once:

```
>> {a, b} = {1, 2};
```

```
>> a
```
1

```
>> b
```
2

Many operations, like addition and multiplication, "thread" over lists, i.e. lists are combined element-wise:

```
>> {1, 2, 3} + {4, 5, 6}
```
$\{5,7,9\}$

```
>> {1, 2, 3} * {4, 5, 6}
```
$\{4,10,18\}$

It is an error to combine lists with unequal lengths:

```
>> {1, 2} + {4, 5, 6}
```
Objects of unequal length cannot be combined.

$\{1,2\}+\{4,5,6\}$

The structure of things

Every expression in *Mathics* is built upon the same principle: it consists of a *head* and an arbitrary number of *children*, unless it is an *atom*, i.e. it can not be subdivided any further. To put it another way: everything is a function call. This can be best seen when displaying expressions in their "full form":

```
>> FullForm[a + b + c]
```
Plus $[a,b,c]$

Nested calculations are nested function calls:

```
>> FullForm[a + b * (c + d)]
```
Plus $[a,$ Times $[b,$ Plus $[c,d]]]$

Even lists are function calls of the function `List`:

```
>> FullForm[{1, 2, 3}]
```
List $[1,2,3]$

The head of an expression can be determined with `Head`:

```
>> Head[a + b + c]
```
Plus

The children of an expression can be accessed like list elements:

```
>> (a + b + c)[[2]]
```
b

The head is the 0th element:

```
>> (a + b + c)[[0]]
```
Plus

The head of an expression can be exchanged using the function `Apply`:

```
>> Apply[g, f[x, y]]
```
$g\,[x,y]$

```
>> Apply[Plus, a * b * c]
```
$a+b+c$

`Apply` can be written using the operator `@@`:

```
>> Times @@ {1, 2, 3, 4}
```
24

(This exchanges the head `List` of `{1, 2, 3, 4}` with `Times`, and then the expression `Times[1, 2, 3, 4]` is evaluated, yielding 24.) `Apply` can also be applied on a certain *level* of an expression:

```
>> Apply[f, {{1, 2}, {3, 4}},
   {1}]
```
$\{f\,[1,2]\,,f\,[3,4]\}$

Or even on a range of levels:

```
>> Apply[f, {{1, 2}, {3, 4}},
   {0, 2}]
```
$f\,[f\,[1,2]\,,f\,[3,4]]$

`Apply` is similar to `Map` (`/@`):

```
>> Map[f, {1, 2, 3, 4}]
```
$\{f\,[1]\,,f\,[2]\,,f\,[3]\,,f\,[4]\}$

```
>> f /@ {{1, 2}, {3, 4}}
```
$\{f\,[\{1,2\}]\,,f\,[\{3,4\}]\}$

The atoms of *Mathics* are numbers, symbols, and strings. `AtomQ` tests whether an expression is an atom:

```
>> AtomQ[5]
```
True

```
>> AtomQ[a + b]
   False
```

The full form of rational and complex numbers looks like they were compound expressions:

```
>> FullForm[3 / 5]
   Rational[3,5]
```

```
>> FullForm[3 + 4 I]
   Complex[3,4]
```

However, they are still atoms, thus unaffected by applying functions, for instance:

```
>> f @@ Complex[3, 4]
```
$3 + 4I$

Nevertheless, every atom has a head:

```
>> Head /@ {1, 1/2, 2.0, I, "a string", x}
```
{Integer, Rational, Real, Complex, String, Symbol}

The operator === tests whether two expressions are the same on a structural level:

```
>> 3 === 3
   True
```

```
>> 3 == 3.0
   True
```

But

```
>> 3 === 3.0
   False
```

because `3` (an `Integer`) and `3.0` (a `Real`) are structurally different.

Functions and patterns

Functions can be defined in the following way:

```
>> f[x_] := x ^ 2
```

This tells *Mathics* to replace every occurrence of `f` with one (arbitrary) parameter `x` with `x ^ 2`.

```
>> f[3]
   9
```

```
>> f[a]
```
a^2

The definition of `f` does not specify anything for two parameters, so any such call will stay unevaluated:

```
>> f[1, 2]
```
$f[1,2]$

In fact, *functions* in *Mathics* are just one aspect of *patterns*: `f[x_]` is a pattern that *matches* expressions like `f[3]` and `f[a]`. The following patterns are available:

`_` or `Blank[]`
: matches one expression.

`Pattern[`x`, `p`]`
: matches the pattern p and stores the value in x.

x`_` or `Pattern[`x`, Blank[]]`
: matches one expression and stores it in x.

`__` or `BlankSequence[]`
: matches a sequence of one or more expressions.

`___` or `BlankNullSequence[]`
: matches a sequence of zero or more expressions.

`_`h or `Blank[`h`]`
: matches one expression with head h.

x`_`h or `Pattern[`x`, Blank[`h`]]`
: matches one expression with head h and stores it in x.

p `|` q or `Alternatives[`p`, `q`]`
: matches either pattern p or q.

p `?` t or `PatternTest[`p`, `t`]`
: matches p if the test $t[p]$ yields `True`.

p `/;` c or `Condition[`p`, `c`]`
: matches p if condition c holds.

`Verbatim[`p`]`
: matches an expression that equals p, without regarding patterns inside p.

As before, patterns can be used to define functions:

```
>> g[s___] := Plus[s] ^ 2
```

```
>> g[1, 2, 3]
   36
```

`MatchQ[`e`, `p`]` tests whether e matches p:

```
>> MatchQ[a + b, x_ + y_]
   True

>> MatchQ[6, _Integer]
   True
```

`ReplaceAll` (`/.`) replaces all occurrences of a pattern in an expression using a `Rule` given by `->`:

```
>> {2, "a", 3, 2.5, "b", c} /.
   x_Integer -> x ^ 2
```

$\{4, \mathrm{a}, 9, 2.5, \mathrm{b}, c\}$

You can also specify a list of rules:

```
>> {2, "a", 3, 2.5, "b", c} /. {
   x_Integer -> x ^ 2.0,
   y_String -> 10}
```

$\{4., 10, 9., 2.5, 10, c\}$

`ReplaceRepeated` (`//.`) applies a set of rules repeatedly, until the expression doesn't change anymore:

```
>> {2, "a", 3, 2.5, "b", c} //.
   {x_Integer -> x ^ 2.0,
   y_String -> 10}
```

$\{4., 100., 9., 2.5, 100., c\}$

There is a "delayed" version of `Rule` which can be specified by `:>` (similar to the relation of `:=` to `=`):

```
>> a :> 1 + 2
```

a:>1 + 2

```
>> a -> 1 + 2
```

a->3

This is useful when the right side of a rule should not be evaluated immediately (before matching):

```
>> {1, 2} /. x_Integer -> N[x]
```

$\{1, 2\}$

Here, `N` is applied to `x` before the actual matching, simply yielding `x`. With a delayed rule this can be avoided:

```
>> {1, 2} /. x_Integer :> N[x]
```

$\{1., 2.\}$

While `ReplaceAll` and `ReplaceRepeated` simply take the first possible match into account, `ReplaceList` returns a list of all possible matches. This can be used to get all subsequences of a list, for instance:

```
>> ReplaceList[{a, b, c}, {___,
   x__, ___} -> {x}]
```

$\{\{a\}, \{a, b\}, \{a, b, c\}, \{b\}, \{b, c\}, \{c\}\}$

`ReplaceAll` would just return the first expression:

```
>> ReplaceAll[{a, b, c}, {___,
   x__, ___} -> {x}]
```

$\{a\}$

In addition to defining functions as rules for certain patterns, there are *pure* functions that can be defined using the `&` postfix operator, where everything before it is treated as the funtion body and `#` can be used as argument placeholder:

```
>> h = # ^ 2 &;

>> h[3]
   9
```

Multiple arguments can simply be indexed:

```
>> sum = #1 + #2 &;

>> sum[4, 6]
   10
```

It is also possible to name arguments using `Function`:

```
>> prod = Function[{x, y}, x * y
   ];

>> prod[4, 6]
   24
```

Pure functions are very handy when functions are used only locally, e.g., when combined with operators like `Map`:

```
>> # ^ 2 & /@ Range[5]
```
$\{1,4,9,16,25\}$

Sort according to the second part of a list:
```
>> Sort[{{x, 10}, {y, 2}, {z,
   5}}, #1[[2]] < #2[[2]] &]
```
$\{\{y,2\},\{z,5\},\{x,10\}\}$

Functions can be applied using prefix or postfix notation, in addition to using `[]`:
```
>> h @ 3
```
9
```
>> 3 // h
```
9

Control statements

Like most programming languages, *Mathics* has common control statements for conditions, loops, etc.:

`If[`*cond*`,` *pos*`,` *neg*`]`
: returns *pos* if *cond* evaluates to `True`, and *neg* if it evaluates to `False`.

`Which[`*cond1*`,` *expr1*`,` *cond2*`,` *expr2*`, ...]`
: yields *expr1* if *cond1* evaluates to `True`, *expr2* if *cond2* evaluates to `True`, etc.

`Do[`*expr*`, {`*i*`,` *max*`}]`
: evaluates *expr* *max* times, substituting *i* in *expr* with values from 1 to *max*.

`For[`*start*`,` *test*`,` *incr*`,` *body*`]`
: evaluates *start*, and then iteratively *body* and *incr* as long as *test* evaluates to `True`.

`While[`*test*`,` *body*`]`
: evaluates *body* as long as *test* evaluates to `True`.

`Nest[`*f*`,` *expr*`,` *n*`]`
: returns an expression with *f* applied *n* times to *expr*.

`NestWhile[`*f*`,` *expr*`,` *test*`]`
: applies a function *f* repeatedly on an expression *expr*, until applying *test* on the result no longer yields `True`.

`FixedPoint[`*f*`,` *expr*`]`
: starting with *expr*, repeatedly applies *f* until the result no longer changes.

```
>> If[2 < 3, a, b]
```
a
```
>> x = 3; Which[x < 2, a, x > 4,
    b, x < 5, c]
```
c

Compound statements can be entered with `;`. The result of a compound expression is its last part or `Null` if it ends with a `;`.
```
>> 1; 2; 3
```
3
```
>> 1; 2; 3;
```

Inside `For`, `While`, and `Do` loops, `Break[]` exits the loop and `Continue[]` continues to the next iteration.

```
>> For[i = 1, i <= 5, i++, If[i
   == 4, Break[]]; Print[i]]
   1
   2
   3
```

Scoping

By default, all symbols are "global" in *Mathics*, i.e. they can be read and written in any part of your program. However, sometimes "local" variables are needed in order not to disturb the global namespace. *Mathics* provides two ways to support this:

- *lexical scoping* by `Module`, and
- *dynamic scoping* by `Block`.

`Module[{`*vars*`}, `*expr*`]`
: localizes variables by giving them a temporary name of the form `name$number`, where number is the current value of `$ModuleNumber`. Each time a module is evaluated, `$ModuleNumber` is incremented.

`Block[{`*vars*`}, `*expr*`]`
: temporarily stores the definitions of certain variables, evaluates *expr* with reset values and restores the original definitions afterwards.

Both scoping constructs shield inner variables from affecting outer ones:

```
>> t = 3;

>> Module[{t}, t = 2]
   2

>> Block[{t}, t = 2]
   2

>> t
   3
```

`Module` creates new variables:

```
>> y = x ^ 3;

>> Module[{x = 2}, x * y]
```

$2x^3$

`Block` does not:

```
>> Block[{x = 2}, x * y]
   16
```

Thus, `Block` can be used to temporarily assign a value to a variable:

```
>> expr = x ^ 2 + x;

>> Block[{x = 3}, expr]
   12

>> x
```

x

`Block` can also be used to temporarily change the value of system parameters:

```
>> Block[{$RecursionLimit = 30},
    x = 2 x]
   Recursion depth of 30 exceeded.
   $Aborted
```

It is common to use scoping constructs for function definitions with local variables:

```
>> fac[n_] := Module[{k, p}, p =
    1; For[k = 1, k <= n, ++k, p
    *= k]; p]

>> fac[10]
   3 628 800

>> 10!
   3 628 800
```

Formatting output

The way results are formatted for output in *Mathics* is rather sophisticated, as compatibility to the way *Mathematica®* does things is one of the design goals. It can be summed up in the following procedure:

1. The result of the query is calculated.
2. The result is stored in `Out` (which `%` is a shortcut for).
3. Any `Format` rules for the desired output form are applied to the result. In the console version of *Mathics*, the result is formatted as `OutputForm`; `MathMLForm` for the `StandardForm` is

used in the interactive Web version; and `TeXForm` for the `StandardForm` is used to generate the LaTeX version of this documentation.

4. `MakeBoxes` is applied to the formatted result, again given either `OutputForm`, `MathMLForm`, or `TeXForm` depending on the execution context of *Mathics*. This yields a new expression consisting of "box constructs".
5. The boxes are turned into an ordinary string and displayed in the console, sent to the browser, or written to the documentation LaTeX file.

As a consequence, there are various ways to implement your own formatting strategy for custom objects.

You can specify how a symbol shall be formatted by assigning values to `Format`:

```
>> Format[x] = "y";
```

```
>> x
   y
```

This will apply to `MathMLForm`, `OutputForm`, `StandardForm`, `TeXForm`, and `TraditionalForm`.

```
>> x // InputForm
   x
```

You can specify a specific form in the assignment to `Format`:

```
>> Format[x, TeXForm] = "z";
```

```
>> x // TeXForm
   \text{z}
```

Special formats might not be very relevant for individual symbols, but rather for custom functions (objects):

```
>> Format[r[args___]] = "<an r
   object>";
```

```
>> r[1, 2, 3]
   <an r object>
```

You can use several helper functions to format expressions:

`Infix[`*expr*`, `*op*`]`
: formats the arguments of *expr* with infix operator *op*.

`Prefix[`*expr*`, `*op*`]`
: formats the argument of *expr* with prefix operator *op*.

`Postfix[`*expr*`, `*op*`]`
: formats the argument of *expr* with postfix operator *op*.

`StringForm[`*form*`, `*arg1*`, `*arg2*`, ...]`
: formats arguments using a format string.

```
>> Format[r[args___]] = Infix[{
   args}, "~"];
```

```
>> r[1, 2, 3]
   1 ~ 2 ~ 3
```

```
>> StringForm["`1` and `2`", n,
   m]
   n and m
```

There are several methods to display expressions in 2-D:

`Row[{...}]`
: displays expressions in a row.

`Grid[{{...}}]`
: displays a matrix in two-dimensional form.

`Subscript[`*expr*`, `*i1*`, `*i2*`, ...]`
: displays *expr* with subscript indices *i1*, *i2*, ...

`Superscript[`*expr*`, `*exp*`]`
: displays *expr* with superscript (exponent) *exp*.

```
>> Grid[{{a, b}, {c, d}}]
   a  b
   c  d
```

```
>> Subscript[a, 1, 2] // TeXForm
   a_{1,2}
```

If you want even more low-level control of how expressions are displayed, you can override `MakeBoxes`:

```
>> MakeBoxes[b, StandardForm] =
   "c";
```

```
>> b
   c
```

This will even apply to `TeXForm`, because `TeXForm` implies `StandardForm`:

```
>> b // TeXForm
   c
```

Except some other form is applied first:

```
>> b // OutputForm // TeXForm
   b
```

`MakeBoxes` for another form:

```
>> MakeBoxes[b, TeXForm] = "d";
```

```
>> b // TeXForm
   d
```

You can cause a much bigger mess by overriding `MakeBoxes` than by sticking to `Format`, e.g. generate invalid XML:

```
>> MakeBoxes[c, MathMLForm] = "<
   not closed";
```

```
>> c // MathMLForm
   <not closed
```

However, this will not affect formatting of expressions involving `c`:

```
>> c + 1 // MathMLForm
   <math><mrow><mn>1</mn>
    <mo>+</mo> <mi>c</mi>
    </mrow></math>
```

That's because `MathMLForm` will, when not overridden for a special case, call `StandardForm` first. `Format` will produce escaped output:

```
>> Format[d, MathMLForm] = "<not
    closed";
```

```
>> d // MathMLForm
   <math>
   <mtext>&lt;not closed</mtext>
   </math>
```

```
>> d + 1 // MathMLForm
   <math><mrow>
   <mn>1</mn> <mo>+</mo>
   <mtext>&lt;not closed</mtext>
   </mrow></math>
```

For instance, you can override `MakeBoxes` to format lists in a different way:

```
>> MakeBoxes[{items___},
   StandardForm] := RowBox[{"[",
    Sequence @@ Riffle[MakeBoxes
    /@ {items}, " "], "]"}]
```

```
>> {1, 2, 3}
   [123]
```

However, this will not be accepted as input to *Mathics* anymore:

```
>> [1 2 3]
   Invalid syntax at or near token [.
```

```
>> Clear[MakeBoxes]
```

By the way, `MakeBoxes` is the only built-in symbol that is not protected by default:

```
>> Attributes[MakeBoxes]
   {HoldAllComplete}
```

`MakeBoxes` must return a valid box construct:

```
>> MakeBoxes[squared[args___],
   StandardForm] := squared[args
   ] ^ 2
```

```
>> squared[1, 2]
   Power [squared [1, 2] , 2]
     is not a valid box structure.
```

The desired effect can be achieved in the following way:

```
>> MakeBoxes[squared[args___],
   StandardForm] :=
   SuperscriptBox[RowBox[{
   MakeBoxes[squared], "[",
   RowBox[Riffle[MakeBoxes[#]& /
   @ {args}, ","]], "]"}], 2]
```

>> `squared[1, 2]`

squared $[1,2]^2$

You can view the box structure of a formatted expression using `ToBoxes`:

>> `ToBoxes[m + n]`

RowBox [{m, +, n}]

The list elements in this `RowBox` are strings, though string delimeters are not shown in the default output form:

>> `InputForm[%]`

RowBox [{"m", "+", "n"}]

Graphics

Two-dimensional graphics can be created using the function `Graphics` and a list of graphics primitives. For three-dimensional graphics see the following section. The following primitives are available:

`Circle[{`*x*`, `*y*`}, `*r*`]`
    draws a circle.

`Disk[{`*x*`, `*y*`}, `*r*`]`
    draws a filled disk.

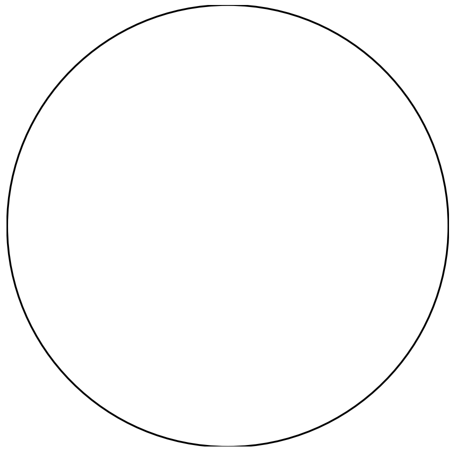
`Rectangle[{`*x1*`, `*y1*`}, {`*x2*`, `*y2*`}]`
    draws a filled rectangle.

`Polygon[{{`*x1*`, `*y1*`}, {`*x2*`, `*y2*`}, ...}]`
    draws a filled polygon.

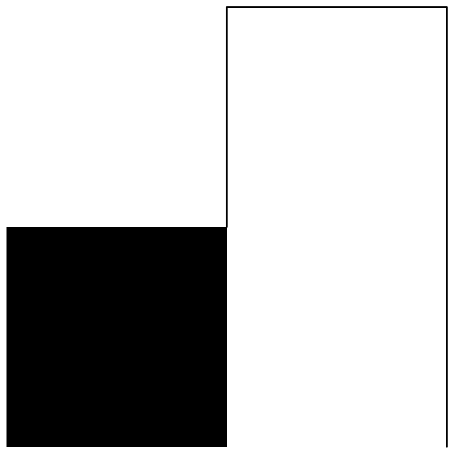
`Line[{{`*x1*`, `*y1*`}, {`*x2*`, `*y2*`}, ...}]`
    draws a line.

`Text[`*text*`, {`*x*`, `*y*`}]`
    draws text in a graphics.

>> `Graphics[{Circle[{0, 0}, 1]}]`



>> `Graphics[{Line[{{0, 0}, {0, 1}, {1, 1}, {1, -1}}], Rectangle[{0, 0}, {-1, -1}]}]`



Colors can be added in the list of graphics primitives to change the drawing color. The following ways to specify colors are supported:

`RGBColor[`*r*`, `*g*`, `*b*`]`
    specifies a color using red, green, and blue.

`CMYKColor[`*c*`, `*m*`, `*y*`, `*k*`]`
    specifies a color using cyan, magenta, yellow, and black.

`Hue[`*h*`, `*s*`, `*b*`]`
    specifies a color using hue, saturation, and brightness.

`GrayLevel[`*l*`]`
    specifies a color using a gray level.

All components range from 0 to 1. Each

color function can be supplied with an additional argument specifying the desired opacity (“alpha”) of the color. There are many predefined colors, such as `Black`, `White`, `Red`, `Green`, `Blue`, etc.

```
>> Graphics[{Red, Disk[]}]
```

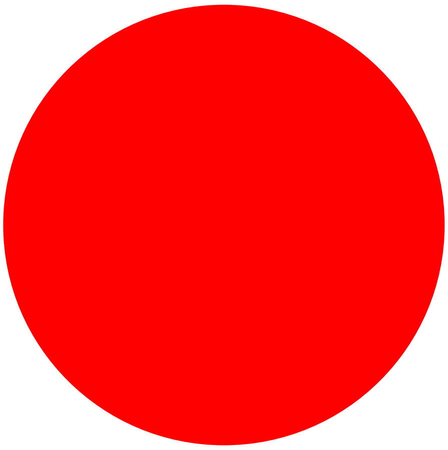
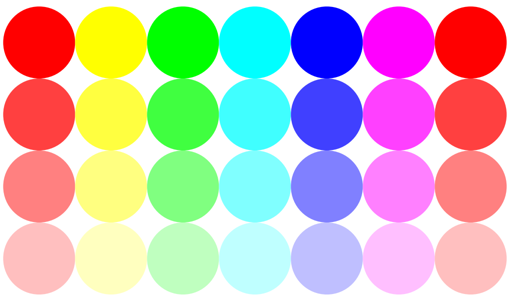


Table of hues:

```
>> Graphics[Table[{Hue[h, s],
   Disk[{12h, 8s}]}, {h, 0, 1,
   1/6}, {s, 0, 1, 1/4}]]
```



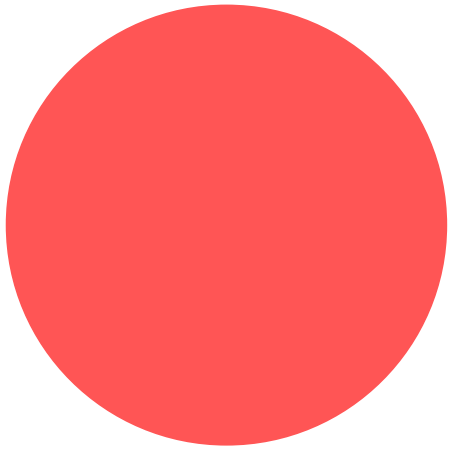
Colors can be mixed and altered using the following functions:

`Blend[{`*color1*`, `*color2*`}, `*ratio*`]`
  mixes *color1* and *color2* with *ratio*, where a ratio of 0 returns *color1* and a ratio of 1 returns *color2*.

`Lighter[`*color*`]`
  makes *color* lighter (mixes it with `White`).

`Darker[`*color*`]`
  makes *color* darker (mixes it with `Black`).

```
>> Graphics[{Lighter[Red], Disk
   []}]
```



`Graphics` produces a `GraphicsBox`:

```
>> Head[ToBoxes[Graphics[{Circle
   []}]]]
```

GraphicsBox

3D Graphics

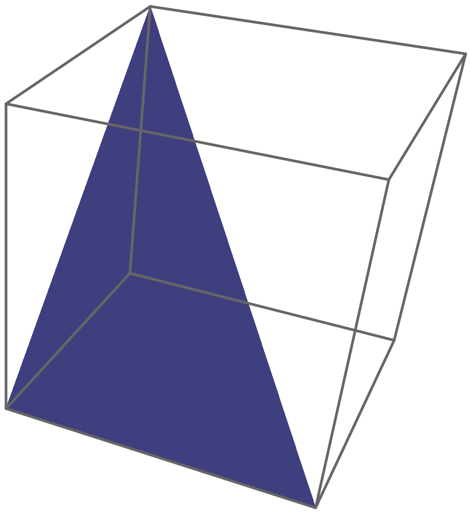
Three-dimensional graphics are created using the function `Graphics3D` and a list of 3D primitives. The following primitives are supported so far:

`Polygon[{{`*x1*`, `*y1*`, `*z1*`}, {`*x2*`, `*y2*`, `*z3*`}, ...}]`
  draws a filled polygon.

`Line[{{`*x1*`, `*y1*`, `*z1*`}, {`*x2*`, `*y2*`, `*z3*`}, ...}]`
  draws a line.

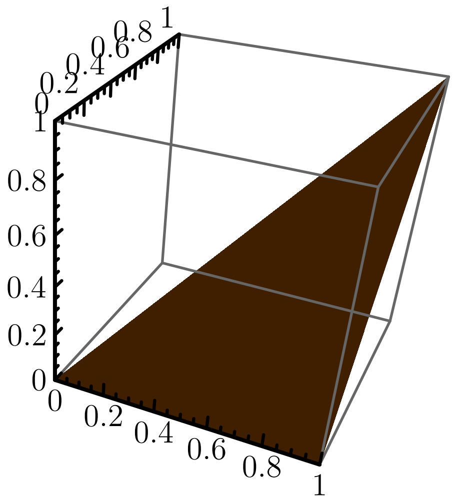
`Point[{`*x1*`, `*y1*`, `*z1*`}]`
  draws a point.

```
>> Graphics3D[Polygon[{{0,0,0},
   {0,1,1}, {1,0,0}}]]
```



Colors can also be added to three-dimensional primitives.

```
>> Graphics3D[{Orange, Polygon
   [{{0,0,0}, {1,1,1},
   {1,0,0}}]}, Axes->True]
```



`Graphics3D` produces a `Graphics3DBox`:

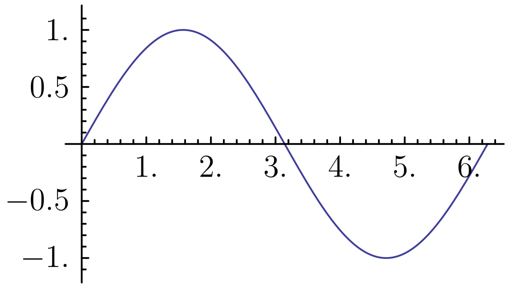
```
>> Head[ToBoxes[Graphics3D[{
   Polygon[]}]]]
```

Graphics3DBox

Plotting

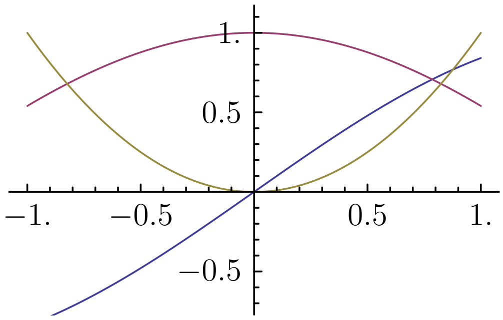
Mathics can plot functions:

```
>> Plot[Sin[x], {x, 0, 2 Pi}]
```



You can also plot multiple functions at once:

```
>> Plot[{Sin[x], Cos[x], x ^ 2},
   {x, -1, 1}]
```

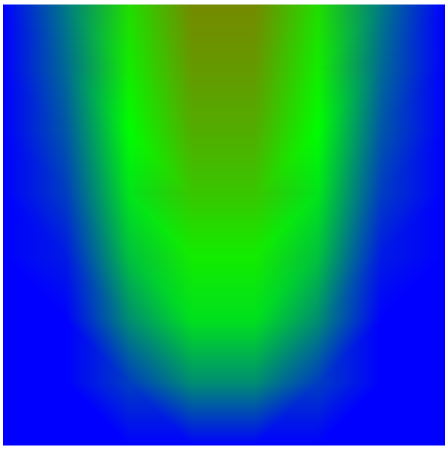


Two-dimensional functions can be plotted using `DensityPlot`:

```
>> DensityPlot[x ^ 2 + 1 / y, {x
   , -1, 1}, {y, 1, 4}]
```

You can use a custom coloring function:

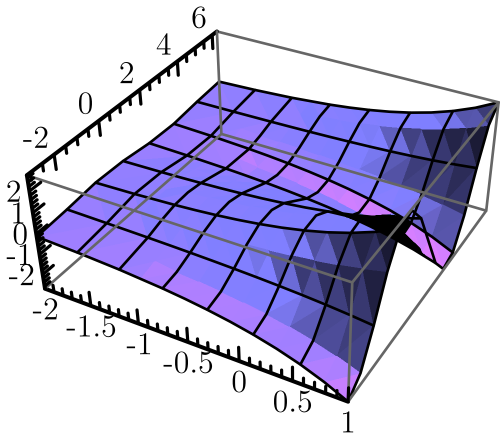
```
>> DensityPlot[x ^ 2 + 1 / y, {x
   , -1, 1}, {y, 1, 4},
   ColorFunction -> (Blend[{Red,
    Green, Blue}, #]&)]
```



One problem with `DensityPlot` is that it's still very slow, basically due to function evaluation being pretty slow in general—and `DensityPlot` has to evaluate a lot of functions.

Three-dimensional plots are supported as well:

```
>> Plot3D[Exp[x] Cos[y], {x, -2,
    1}, {y, -Pi, 2 Pi}]
```



4. Examples

Contents

Curve sketching

Let's sketch the function

```
>> f[x_] := 4 x / (x ^ 2 + 3 x + 5)
```

The derivatives are

```
>> {f'[x], f''[x], f'''[x]} // Together
```

$$\left\{-\frac{4\left(-5+x^2\right)}{\left(5+3x+x^2\right)^2}, \frac{8\left(-15-15x+x^3\right)}{\left(5+3x+x^2\right)^3}, -\frac{24\left(-20-60x-30x^2+x^4\right)}{\left(5+3x+x^2\right)^4}\right\}$$

To get the extreme values of `f`, compute the zeroes of the first derivatives:

```
>> extremes = Solve[f'[x] == 0, x]
```

$$\left\{\left\{x\text{->}-\sqrt{5}\right\}, \left\{x\text{->}\sqrt{5}\right\}\right\}$$

And test the second derivative:

```
>> f''[x] /. extremes // N
```

$\{1.65085581947099374, -0.0640789599668615036\}$

Thus, there is a local maximum at `x = Sqrt[5]` and a local minimum at `x = -Sqrt[5]`. Compute the inflection points numerically, choping imaginary parts close to 0:

```
>> inflections = Solve[f''[x] == 0, x] // N // Chop
```

$\{\{x\text{->}-1.08519961543710476\}, \{x\text{->}-3.21462740739519024\}, \{x\text{->}4.299827022832295\}\}$

Insert into the third derivative:

```
>> f'''[x] /. inflections
```

$\{-3.67683091753987803, 0.694905362720454084, 0.0067189432491760176\}$

Being different from 0, all three points are actual inflection points. `f` is not defined where its denominator is 0:

```
>> Solve[Denominator[f[x]] == 0, x]
```

$$\left\{\left\{x\text{->}-\frac{3}{2}-\frac{I}{2}\sqrt{11}\right\}, \left\{x\text{->}-\frac{3}{2}+\frac{I}{2}\sqrt{11}\right\}\right\}$$

These are non-real numbers, consequently `f` is defined on all real numbers. The behaviour of `f` at the boundaries of its definition:

```
>> Limit[f[x], x -> Infinity]
```

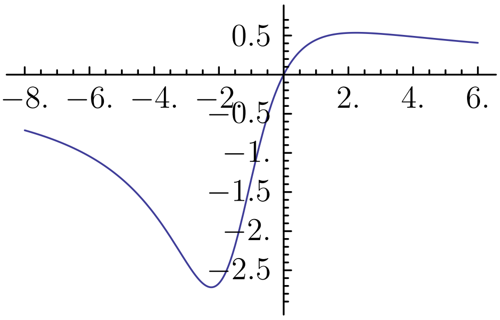
0

```
>> Limit[f[x], x -> -Infinity]
```

0

Finally, let's plot `f`:

```
>>  Plot[f[x], {x, -8, 6}]
```



Linear algebra

Let's consider the matrix

```
>>  A = {{1, 1, 0}, {1, 0, 1},
    {0, 1, 1}};
```

```
>>  MatrixForm[A]
```

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

We can compute its eigenvalues and eigenvectors:

```
>>  Eigenvalues[A]
```

$\{2, -1, 1\}$

```
>>  Eigenvectors[A]
```

$\{\{1,1,1\}, \{1,-2,1\}, \{-1,0,1\}\}$

This yields the diagonalization of `A`:

```
>>  T = Transpose[Eigenvectors[A
    ]]; MatrixForm[T]
```

$$\begin{pmatrix} 1 & 1 & -1 \\ 1 & -2 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

```
>>  Inverse[T] . A . T //
    MatrixForm
```

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
>>  % == DiagonalMatrix[
    Eigenvalues[A]]
```

True

We can solve linear systems:

```
>>  LinearSolve[A, {1, 2, 3}]
```

$\{0,1,2\}$

```
>>  A . %
```

$\{1,2,3\}$

In this case, the solution is unique:

```
>>  NullSpace[A]
```

$\{\}$

Let's consider a singular matrix:

```
>>  B = {{1, 2, 3}, {4, 5, 6},
    {7, 8, 9}};
```

```
>>  MatrixRank[B]
```

2

```
>>  s = LinearSolve[B, {1, 2, 3}]
```

$\left\{-\frac{1}{3}, \frac{2}{3}, 0\right\}$

```
>>  NullSpace[B]
```

$\{\{1,-2,1\}\}$

```
>>  B . (RandomInteger[100] *
    %[[1]] + s)
```

$\{1,2,3\}$

Dice

Let's play with dice in this example. A `Dice` object shall represent the outcome of a series of rolling a dice with six faces, e.g.:

```
>>  Dice[1, 6, 4, 4]
```

Dice $[1,6,4,4]$

Like in most games, the ordering of the individual throws does not matter. We can express this by making `Dice Orderless`:

```
>>  SetAttributes[Dice, Orderless
    ]
```

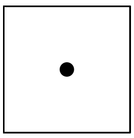
```
>>  Dice[1, 6, 4, 4]
```

Dice $[1,4,4,6]$

A dice object shall be displayed as a rectangle with the given number of points in it, positioned like on a traditional dice:

```
>> Format[Dice[n_Integer?(1 <= #
    <= 6 &)]] := Block[{p = 0.2,
    r = 0.05}, Graphics[{
   EdgeForm[Black], White,
   Rectangle[], Black, EdgeForm
   [], If[OddQ[n], Disk[{0.5,
   0.5}, r]], If[MemberQ[{2, 3,
   4, 5, 6}, n], Disk[{p, p}, r
   ]], If[MemberQ[{2, 3, 4, 5,
   6}, n], Disk[{1 - p, 1 - p},
   r]], If[MemberQ[{4, 5, 6}, n
   ], Disk[{p, 1 - p}, r]], If[
   MemberQ[{4, 5, 6}, n], Disk
   [{1 - p, p}, r]], If[n === 6,
    {Disk[{p, 0.5}, r], Disk[{1
   - p, 0.5}, r]}]}, ImageSize
   -> Tiny]]
```

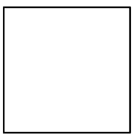
```
>> Dice[1]
```



The empty series of dice shall be displayed as an empty dice:

```
>> Format[Dice[]] := Graphics[{
   EdgeForm[Black], White,
   Rectangle[]}, ImageSize ->
   Tiny]
```

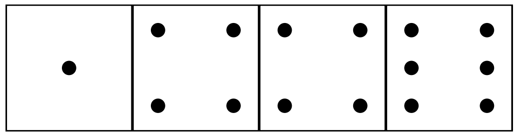
```
>> Dice[]
```



Any non-empty series of dice shall be displayed as a row of individual dice:

```
>> Format[Dice[d___Integer?(1 <=
    # <= 6 &)]] := Row[Dice /@ {
   d}]
```

```
>> Dice[1, 6, 4, 4]
```



Note that *Mathics* will automatically sort the given format rules according to their "generality", so the rule for the empty dice does not get overridden by the rule for a series of dice. We can still see the original form by using `InputForm`:

```
>> Dice[1, 6, 4, 4] // InputForm
```

Dice [1, 4, 4, 6]

We want to combine `Dice` objects using the + operator:

```
>> Dice[a___] + Dice[b___] ^:=
   Dice[Sequence @@ {a, b}]
```

The `^:=` (`UpSetDelayed`) tells *Mathics* to associate this rule with `Dice` instead of `Plus`, which is protected—we would have to unprotect it first:

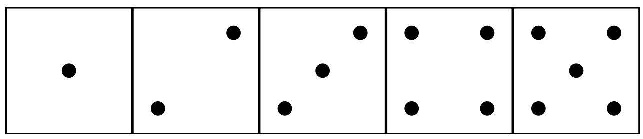
```
>> Dice[a___] + Dice[b___] :=
   Dice[Sequence @@ {a, b}]
```

Tag Plus in Dice [a___] + Dice [
  b___] is Protected.

$Failed

We can now combine dice:

```
>> Dice[1, 5] + Dice[3, 2] +
   Dice[4]
```



Let's write a function that returns the sum of the rolled dice:

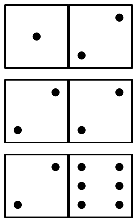
```
>> DiceSum[Dice[d___]] := Plus
   @@ {d}
```

```
>> DiceSum @ Dice[1, 2, 5]
```

8

And now let's put some dice into a table:

```
>> Table[{Dice[Sequence @@ d],
   DiceSum @ Dice[Sequence @@ d
   ]}, {d, {{1, 2}, {2, 2}, {2,
   6}}}] // TableForm
```



3

4

8

It is not very sophisticated from a mathematical point of view, but it's beautiful.

5. Web interface

Contents

Saving and loading worksheets

Worksheets exist in the browser window only and are not stored on the server, by default. To save all your queries and results, use the *Save* button in the menu bar. You have to login using your email address. If you don't have an account yet, leave the password field empty and a password will be sent to you. You will remain logged in until you press the *Logout* button in the upper right corner.
Saved worksheets can be loaded again using the *Load* button. Note that worksheet names are case-insensitive.

How definitions are stored

When you use the Web interface of *Mathics*, a browser session is created. Cookies have to be enabled to allow this. Your session holds a key which is used to access your definitions that are stored in a database on the server. As long as you don't clear the cookies in your browser, your definitions will remain even when you close and re-open the browser.
This implies that you should not store sensitive, private information in *Mathics* variables when using the online Web interface, of course. In addition to their values being stored in a database on the server, your queries might be saved for debugging purposes. However, the fact that they are transmitted over plain HTTP should make you aware that you should not transmit any sensitive information. When you want to do calculations with that kind of stuff, simply install *Mathics* locally!
When you use *Mathics* on a public terminal, use the command `Quit[]` to erase all your definitions and close the browser window.

Keyboard commands

There are some keyboard commands you can use in the web interface of *Mathics*.

`Shift+Return`
: Evaluate current cell (the most important one, for sure)

`Ctrl+D`
: Focus documentation search

`Ctrl+C`
: Back to document code

`Ctrl+S`
: Save worksheet

`Ctrl+O`
: Open worksheet

Unfortunately, keyboard commands do not work as expected in all browsers and under all operating systems. Often, they are only recognized when a textfield has focus; otherwise, the browser might do some browser-specific actions, like setting a bookmark etc.

6. Implementation

Contents

Developing

To start developing, check out the source directory. Run

```
$ python setup.py develop
```

This will temporarily overwrite the installed package in your Python library with a link to the current source directory. In addition, you might want to start the Django development server with

```
$ python manage.py runserver
```

It will restart automatically when you make changes to the source code.

Documentation and tests

One of the greatest features of *Mathics* is its integrated documentation and test system. Tests can be included right in the code as Python docstrings. All desired functionality should be covered by these tests to ensure that changes to the code don't break it. Execute

```
$ python test.py
```

to run all tests.

During a test run, the results of tests can be stored for the documentation, both in MathML and LaTeX form, by executing

```
$ python test.py -o
```

The XML version of the documentation, which can be accessed in the Web interface, is updated immediately. To produce the LaTeX documentation file, run:

```
$ python test.py -t
```

You can then create the PDF using LaTeX. All required steps can be executed by

```
$ make latex
```

in the `doc/tex` directory, which uses `latexmk` to build the LaTeX document. You just have to adjust the `Makefile` and `latexmkrc` to your environment. You need the Asymptote (version 2 at least) to generate the graphics in the documentation.

You can also run the tests for individual built-in symbols using

```
python test.py -s [name]
```

This will not re-create the corresponding documentation results, however. You have to run a complete test to do that.

Documentation markup

There is a lot of special markup syntax you can use in the documentation. It is kind of a mixture of XML, LaTeX, Python doctest, and custom markup.

The following commands can be used to specify test cases.

`>>` *query*
: a test query.

`:` *message*
: a message in the result of the test query.

`|` *print*
: a printed line in the result of the test query.

`=` *result*
: the actual result of the test query.

`.` *newline*
: a newline in the test result.

`$`*identifier*`$`
: a variable identifier in *Mathics* code or in text.

`#>` *query*
: a test query that is not shown in the documentation.

`-Graphics-`
: graphics in the test result.

`...`
: a part of the test result which is not checked in the test, e.g., for randomized or system-dependent output.

The following commands can be used to markup documentation text.

`##` *comment*
: a comment line that is not shown in the documentation.

`<dl>`*list*`</dl>`
: a definition list with `<dt>` and `<dd>` entries.

`<dt>`*title*
: the title of a description item.

`<dd>`*description*
: the description of a description item.

`<ul>`*list*`</ul>`
: an unordered list with `<li>` entries.

`<ol>`*list*`</ol>`
: an ordered list with `<li>` entries.

`<li>`*item*
: an item of an unordered or ordered list.

`'`*code*`'`
: inline *Mathics* code or other code.

`<console>`*text*`</console>`
: a console (shell/bash/Terminal) transcript in its own paragraph.

`<con>`*text*`</con>`
: an inline console transcript.

`<em>`*text*`</em>`
: emphasized (italic) text.

`<url>`*url*`</url>`
: a URL.

`<img src="`*src*`" title="`*title*`" label="`*label*`">`
: an image.

`<ref label="`*label*`">`
: a reference to an image.

`\skip`
: a vertical skip.

`\LaTeX, \Mathematica, \Mathics`
: special product and company names.

`\'`
: a single '.

To include images in the documentation, use the `img` tag, place an EPS file *src*`.eps` in `documentation/images` and run `images.sh` in the `doc` directory.

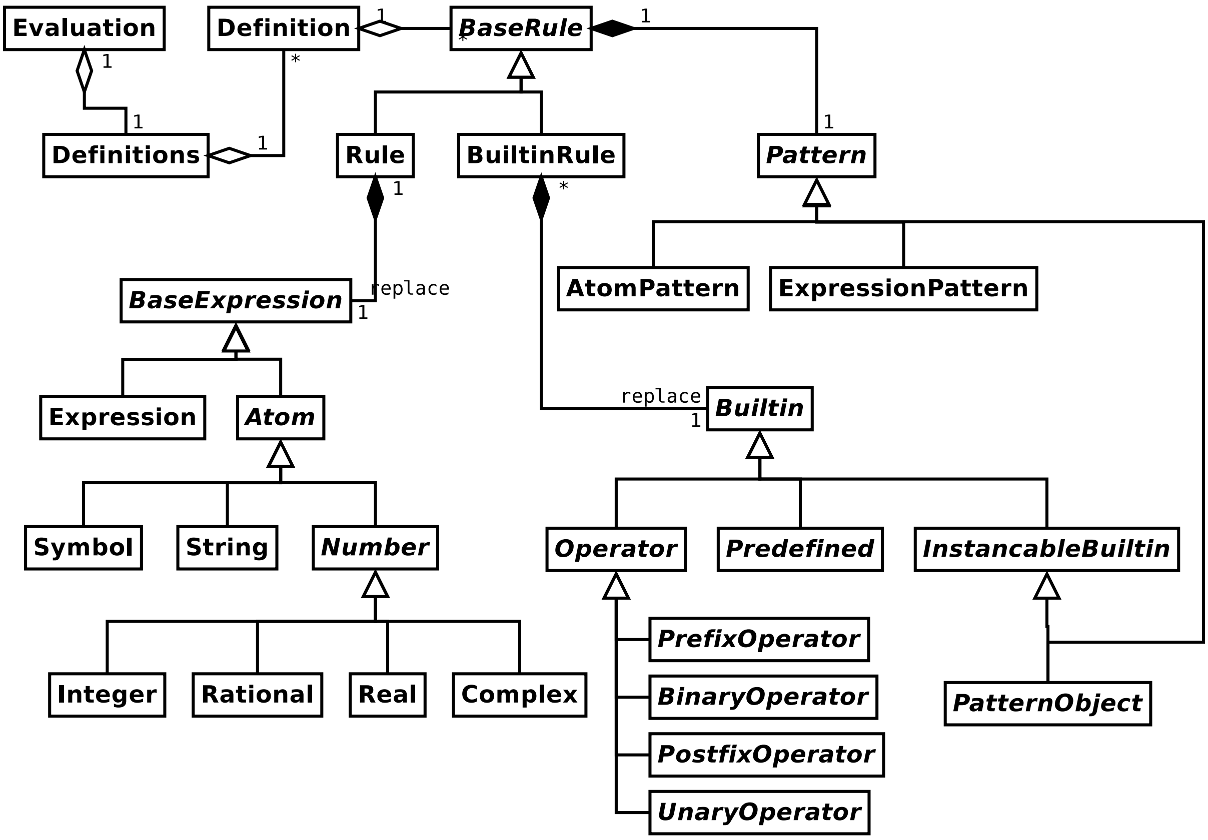


Figure 6.1.: UML class diagram

Classes

A UML diagram of the most important classes in *Mathics* can be seen in figure 6.1.

Adding built-in symbols

Adding new built-in symbols to *Mathics* is very easy. Either place a new module in the `builtin` directory and add it to the list of modules in `builtin/__init__.py` or use an existing module. Create a new class derived from `Builtin`. If you want to add an operator, you should use one of the subclasses of `Operator`. Use `SympyFunction` for symbols that have a special meaning in SymPy.
To get an idea of how a built-in class can look like, consider the following implementation of `If`:

```
class If(Builtin):
  """
  <dl>
  <dt>'If[$cond$, $pos$, $neg$]'
    <dd>returns $pos$ if $cond$ evaluates
         to 'True', and $neg$ if it
        evaluates to 'False'.
  <dt>'If[$cond$, $pos$, $neg$, $other$]'
    <dd>returns $other$ if $cond$
        evaluates to neither 'True' nor '
        False'.
  <dt>'If[$cond$, $pos$]'
    <dd>returns 'Null' if $cond$
        evaluates to 'False'.
  </dl>
  >> If[1<2, a, b]
   = a
  If the second branch is not specified,
      'Null' is taken:
  >> If[1<2, a]
   = a
  >> If[False, a] //FullForm
   = Null

  You might use comments (inside '(*' and
       '*)') to make the branches of 'If'
       more readable:
  >> If[a, (*then*) b, (*else*) c];
  """

  attributes = ['HoldRest']

  rules = {
    'If[condition_, t_]': 'If[condition,
        t, Null]',
  }

  def apply_3(self, condition, t, f,
      evaluation):
    'If[condition_, t_, f_]'

    if condition == Symbol('True'):
      return t.evaluate(evaluation)
    elif condition == Symbol('False'):
      return f.evaluate(evaluation)

  def apply_4(self, condition, t, f, u,
      evaluation):
    'If[condition_, t_, f_, u_]'

    if condition == Symbol('True'):
      return t.evaluate(evaluation)
    elif condition == Symbol('False'):
      return f.evaluate(evaluation)
    else:
      return u.evaluate(evaluation)
```

The class starts with a Python *docstring* that specifies the documentation and tests for the symbol. A list (or tuple) `attributes` can be used to assign attributes to the symbol. `Protected` is assigned by default. A dictionary `rules` can be used to add custom rules that should be applied.
Python functions starting with `apply` are converted to built-in rules. Their docstring is compiled to the corresponding *Mathics* pattern. Pattern variables used in the pattern are passed to the Python function by their same name, plus an additional `evaluation` object. This object is needed to evaluate further expressions, print messages in the Python code, etc. Unsurprisingly, the return value of the Python function is the expression which is replaced for the matched pattern. If the function does not return any value, the *Mathics* expression is left unchanged. Note that you have to return `Symbol[‘‘Null’]’` explicitely if you want that.

Part II.

Reference of built-in symbols

I. Algebra

Contents

Apart

`Apart[`*expr*`]`
: writes *expr* as sum of individual fractions.

`Apart[`*expr*`, `*var*`]`
: treats *var* as main variable.

```
>> Apart[1 / (x^2 + 5x + 6)]
```

$$\frac{1}{2+x}-\frac{1}{3+x}$$

When several variables are involved, the results can be different depending on the main variable:

```
>> Apart[1 / (x^2 - y^2), x]
```

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

```
>> Apart[1 / (x^2 - y^2), y]
```

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

`Apart` is `Listable`:

```
>> Apart[{1 / (x^2 + 5x + 6)}]
```

$$\left\{\frac{1}{2+x}-\frac{1}{3+x}\right\}$$

But it does not touch other expressions:

```
>> Sin[1 / (x ^ 2 - y ^ 2)] //
   Apart
```

$$\text{Sin}\left[\frac{1}{x^2-y^2}\right]$$

Cancel

`Cancel[`*expr*`]`
: cancels out common factors in numerators and denominators.

```
>> Cancel[x / x ^ 2]
```

$$\frac{1}{x}$$

`Cancel` threads over sums:

```
>> Cancel[x / x ^ 2 + y / y ^ 2]
```

$$\frac{1}{x}+\frac{1}{y}$$

```
>> Cancel[f[x] / x + x * f[x] /
   x ^ 2]
```

$$\frac{2f\left[x\right]}{x}$$

Denominator

`Denominator[`*expr*`]`
: gives the denominator in *expr*.

```
>> Denominator[a / b]
```
b

```
>> Denominator[2 / 3]
```
3

```
>> Denominator[a + b]
```
1

Expand

`Expand[`*expr*`]`
: expands out positive integer powers and products of sums in *expr*.

```
>> Expand[(x + y)^ 3]
```
$x^3 + 3x^2y + 3xy^2 + y^3$

```
>> Expand[(a + b)(a + c + d)]
```
$a^2 + ab + ac + ad + bc + bd$

```
>> Expand[(a + b)(a + c + d)(e +
   f)+ e a a]
```
$$2a^2e + a^2f + abe + abf + ace + acf + ade + adf + bce + bcf + bde + bdf$$

```
>> Expand[(a + b)^ 2 * (c + d)]
```
$a^2c + a^2d + 2abc + 2abd + b^2c + b^2d$

```
>> Expand[(x + y)^ 2 + x y]
```
$x^2 + 3xy + y^2$

```
>> Expand[((a + b)(c + d))^ 2 +
   b (1 + a)]
```
$$a^2c^2 + 2a^2cd + a^2d^2 + b + ab + 2abc^2 + 4abcd + 2abd^2 + b^2c^2 + 2b^2cd + b^2d^2$$

`Expand` expands items in lists and rules:

```
>> Expand[{4 (x + y), 2 (x + y)
   -> 4 (x + y)}]
```
$\{4x + 4y, 2x + 2y\text{->}4x + 4y\}$

`Expand` does not change any other expression.

```
>> Expand[Sin[x (1 + y)]]
```
$\mathrm{Sin}\left[x\left(1 + y\right)\right]$

Factor

`Factor[`*expr*`]`
: factors the polynomial expression *expr*.

```
>> Factor[x ^ 2 + 2 x + 1]
```
$(1 + x)^2$

```
>> Factor[1 / (x^2+2x+1)+ 1 / (x
   ^4+2x^2+1)]
```
$$\frac{2 + 2x + 3x^2 + x^4}{(1 + x)^2\left(1 + x^2\right)^2}$$

Numerator

`Numerator[`*expr*`]`
: gives the numerator in *expr*.

```
>> Numerator[a / b]
```
a

```
>> Numerator[2 / 3]
```
2

```
>> Numerator[a + b]
```
$a + b$

PowerExpand

`PowerExpand[`*expr*`]`
: expands out powers of the form `(x^y)^z` and `(x*y)^z` in *expr*.

>> `PowerExpand[(a ^ b)^ c]`

a^{bc}

>> `PowerExpand[(a * b)^ c]`

$a^c b^c$

`PowerExpand` is not correct without certain assumptions:

>> `PowerExpand[(x ^ 2)^ (1/2)]`

x

Simplify

`Simplify[`*expr*`]`
    simplifies *expr*.

>> `Simplify[2*Sin[x]^2 + 2*Cos[x]^2]`

2

>> `Simplify[x]`

x

>> `Simplify[f[x]]`

$f[x]$

Together

`Together[`*expr*`]`
    writes sums of fractions in *expr* together.

>> `Together[a / c + b / c]`

$$\frac{a+b}{c}$$

`Together` operates on lists:

>> `Together[{x / (y+1)+ x / (y+1)^2}]`

$$\left\{\frac{x\,(2+y)}{(1+y)^2}\right\}$$

But it does not touch other functions:

>> `Together[f[a / c + b / c]]`

$$f\left[\frac{a}{c}+\frac{b}{c}\right]$$

Variables

`Variables[`*expr*`]`
    gives a list of the variables that appear in the polynomial *expr*.

>> `Variables[a x^2 + b x + c]`

$\{a, b, c, x\}$

>> `Variables[{a + b x, c y^2 + x/2}]`

$\{a, b, c, x, y\}$

>> `Variables[x + Sin[y]]`

$\{x, \mathrm{Sin}[y]\}$

II. Arithmetic

Contents

Abs

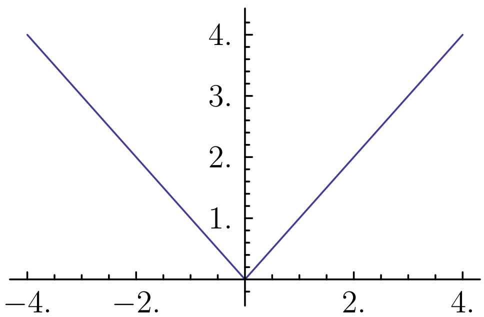
`Abs[`x`]`
: returns the absolute value of x.

```
>> Abs[-3]
   3
```

`Abs` returns the magnitude of complex numbers:

```
>> Abs[3 + I]
```
$\sqrt{10}$

```
>> Abs[3.0 + I]
   3.1622776601683793
```

```
>> Plot[Abs[x], {x, -4, 4}]
```


ComplexInfinity

`ComplexInfinity`
: represents an infinite complex quantity of undetermined direction.

```
>> 1 / ComplexInfinity
   0
```

```
>> ComplexInfinity + ComplexInfinity
```
ComplexInfinity

```
>> ComplexInfinity * Infinity
```
ComplexInfinity

```
>> FullForm[ComplexInfinity]
```
DirectedInfinity []

Complex

`Complex`
: is the head of complex numbers.

`Complex[a, b]`
: constructs the complex number `a + I b`.

```
>> Head[2 + 3*I]
```
Complex

```
>> Complex[1, 2/3]
```
$1 + \frac{2I}{3}$

```
>> Abs[Complex[3, 4]]
```
5

Conjugate

`Conjugate[z]`
: returns the complex conjugate of the complex number z.

```
>> Conjugate[3 + 4 I]
```
$3 - 4I$

```
>> Conjugate[3]
```
3

```
>> Conjugate[a + b * I]
```
Conjugate $[a]$ − IConjugate $[b]$

```
>> Conjugate[{{1, 2 + I 4, a + I b}, {I}}]
```
$\{\{1, 2 - 4I,$ Conjugate $[a] - I$Conjugate $[b]\}, \{-I\}\}$

```
>> Conjugate[1.5 + 2.5 I]
```
$1.5 - 2.5I$

DirectedInfinity

`DirectedInfinity[z]`
: represents an infinite multiple of the complex number z.

`DirectedInfinity[]`
: is the same as `ComplexInfinity`.

```
>> DirectedInfinity[1]
```
∞

```
>> DirectedInfinity[]
```
ComplexInfinity

```
>> DirectedInfinity[1 + I]
```
$\left(\frac{1}{2} + \frac{I}{2}\right)\sqrt{2}\infty$

```
>> 1 / DirectedInfinity[1 + I]
```
0

```
>> DirectedInfinity[1] + DirectedInfinity[-1]
```
Indeterminate expression $-\infty + \infty$ encountered.

Indeterminate

Divide (/)

`Divide[a, b]`</dt> <dt>`a / b`
: represents the division of a by b.

```
>> 30 / 5
```
6

```
>> 1 / 8
```
$\frac{1}{8}$

```
>> Pi / 4
```
$\frac{\text{Pi}}{4}$

Use `N` or a decimal point to force numeric evaluation:

```
>> Pi / 4.0
```
0.785398163397448310

```
>> 1 / 8
```
$\frac{1}{8}$

```
>> N[%]
```
0.125

Nested divisions:

```
>> a / b / c
```
$\frac{a}{bc}$

```
>> a / (b / c)
```
$\frac{ac}{b}$

```
>> a / b / (c / (d / e))
```
$\frac{ad}{bce}$

```
>> a / (b ^ 2 * c ^ 3 / e)
```
$\frac{ae}{b^2c^3}$

ExactNumberQ

`ExactNumberQ[`*expr*`]`
: returns `True` if *expr* is an exact number, and `False` otherwise.

```
>> ExactNumberQ[10]
```
True

```
>> ExactNumberQ[4.0]
```
False

```
>> ExactNumberQ[n]
```
False

`ExactNumberQ` can be applied to complex numbers:

```
>> ExactNumberQ[1 + I]
```
True

```
>> ExactNumberQ[1 + 1. I]
```
False

Factorial (`!`)

`Factorial[`*n*`]`</dt> <dt>*n*`!`
: computes the factorial of *n*.

```
>> 20!
```
2 432 902 008 176 640 000

`Factorial` handles numeric (real and complex) values using the gamma function:

```
>> 10.5!
```
$1.18994230839622485 \times 10^7$

```
>> (-3.0+1.5*I)!
```
$0.0427943437183768611 - 0.00461565252860394996I$

However, the value at poles is `ComplexInfinity`:

```
>> (-1.)!
```
ComplexInfinity

`Factorial` has the same operator (`!`) as `Not`, but with higher precedence:

```
>> !a! //FullForm
```
Not[Factorial[*a*]]

Gamma

`Gamma[`*z*`]`
: is the Gamma function on the complex number *z*.

>> `Gamma[8]`

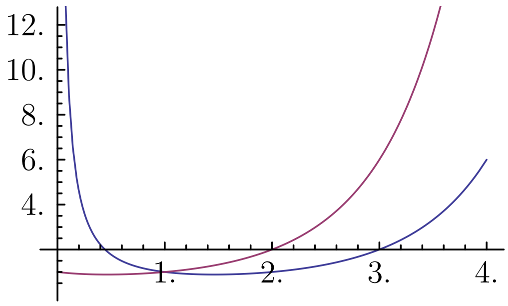
5 040

>> `Gamma[1. + I]`

0.498015668118356043 −
0.154949828301810685*I*

Both `Gamma` and `Factorial` functions are continuous:

>> `Plot[{Gamma[x], x!}, {x, 0, 4}]`



HarmonicNumber

`HarmonicNumber[n]`
: returns the nth harmonic number.

>> `Table[HarmonicNumber[n], {n, 8}]`

$$\left\{1, \frac{3}{2}, \frac{11}{6}, \frac{25}{12}, \frac{137}{60}, \frac{49}{20}, \frac{363}{140}, \frac{761}{280}\right\}$$

>> `HarmonicNumber[3.8]`

2.03806340563064920

I

`I`
: represents the imaginary number `Sqrt[-1]`.

>> `I^2`

−1

>> `(3+I)*(3-I)`

10

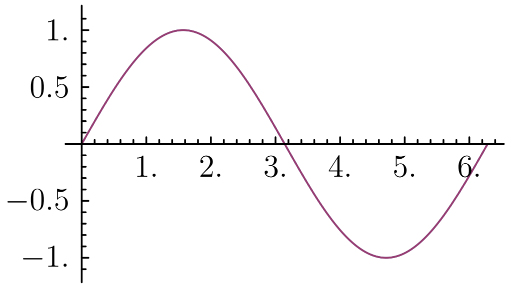
Im

`Im[z]`
: returns the imaginary component of the complex number z.

>> `Im[3+4I]`

4

>> `Plot[{Sin[a], Im[E^(I a)]}, {a, 0, 2 Pi}]`



Indeterminate

`Indeterminate`
: represents an indeterminate result.

>> `0^0`

Indeterminate expression
0^0 encountered.

Indeterminate

InexactNumberQ

`InexactNumberQ[expr]`
: returns `True` if *expr* is not an exact number, and `False` otherwise.

>> `InexactNumberQ[a]`

False

>> `InexactNumberQ[3.0]`

True

>> `InexactNumberQ[2/3]`

False

`InexactNumberQ` can be applied to complex numbers:

```
>> InexactNumberQ[4.0+I]
   True
```

Infinity

`Infinity`
: represents an infinite real quantity.

```
>> 1 / Infinity
   0
```

```
>> Infinity + 100
```
∞

Use `Infinity` in sum and limit calculations:

```
>> Sum[1/x^2, {x, 1, Infinity}]
```
$$\frac{\mathrm{Pi}^2}{6}$$

IntegerQ

`IntegerQ[`*expr*`]`
: returns `True` if *expr* is an integer, and `False` otherwise.

```
>> IntegerQ[3]
   True
```

```
>> IntegerQ[Pi]
   False
```

Integer

`Integer`
: is the head of integers.

```
>> Head[5]
   Integer
```

Minus (-)

`Minus[`*expr*`]`
: is the negation of *expr*.

```
>> -a //FullForm
```
$\mathrm{Times}[-1, a]$

`Minus` automatically distributes:

```
>> -(x - 2/3)
```
$$\frac{2}{3} - x$$

`Minus` threads over lists:

```
>> -Range[10]
```
$$\{-1, -2, -3, -4, -5, -6, -7, -8, -9, -10\}$$

NumberQ

`NumberQ[`*expr*`]`
: returns `True` if *expr* is an explicit number, and `False` otherwise.

```
>> NumberQ[3+I]
   True
```

```
>> NumberQ[5!]
   True
```

```
>> NumberQ[Pi]
   False
```

Piecewise

`Picewise[{{expr1, cond1}, ...}]`
: represents a piecewise function.

`Picewise[{{expr1, cond1}, ...}, expr]`
: represents a piecewise function with default `expr`.

Heaviside function

```
>> Piecewise[{{0, x <= 0}}, 1]
```
Piecewise $[\{\{0, x<=0\}\}, 1]$

Plus (+)

`Plus[a, b, ...]`</dt> <dt>$a + b + ...$
: represents the sum of the terms a, b, ...

```
>> 1 + 2
```
3

`Plus` performs basic simplification of terms:

```
>> a + b + a
```
$2a + b$

```
>> a + a + 3 * a
```
$5a$

```
>> a + b + 4.5 + a + b + a + 2 +
    1.5 b
```
$6.5 + 3.a + 3.5b$

Apply `Plus` on a list to sum up its elements:

```
>> Plus @@ {2, 4, 6}
```
12

The sum of the first 1000 integers:

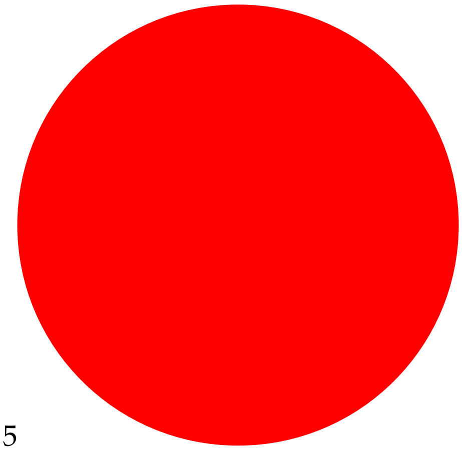
```
>> Plus @@ Range[1000]
```
500 500

`Plus` has default value 0:

```
>> DefaultValues[Plus]
```
{HoldPattern [Default [Plus]] :>0}

```
>> a /. n_. + x_ :> {n, x}
```
$\{0, a\}$

The sum of 2 red circles and 3 red circles is...

```
>> 2 Graphics[{Red,Disk[]}] + 3
   Graphics[{Red,Disk[]}]
```
5 

Pochhammer

`Pochhammer[a, n]`
: is the Pochhammer symbol (a)_n.

```
>> Pochhammer[4, 8]
```
6 652 800

Power (^)

`Power[a, b]`</dt> <dt>`a ^ b`
: represents a raised to the power of b.

```
>> 4 ^ (1/2)
```
2

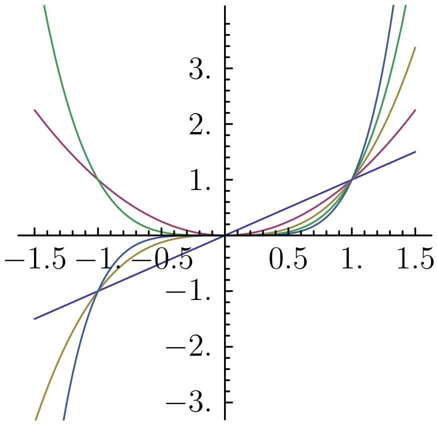
```
>> 4 ^ (1/3)
```
$2^{\frac{2}{3}}$

```
>> 3^123
```
48 519 278 097 689 642 681~
~155 855 396 759 336 072~
~749 841 943 521 979 872 827

```
>> (y ^ 2)^ (1/2)
```
$\sqrt{y^2}$

```
>> (y ^ 2)^ 3
```
y^6

```
>> Plot[Evaluate[Table[x^y, {y,
   1, 5}]], {x, -1.5, 1.5},
   AspectRatio -> 1]
```



Use a decimal point to force numeric evaluation:

```
>> 4.0 ^ (1/3)
```
1.58740105196819947

`Power` has default value 1 for its second argument:

```
>> DefaultValues[Power]
```
{HoldPattern [Default [
  Power, 2]] :>1}

```
>> a /. x_ ^ n_. :> {x, n}
```
$\{a, 1\}$

`Power` can be used with complex numbers:

```
>> (1.5 + 1.0 I)^ 3.5
```
$-3.68294005782191823 + 6.95139266402850491I$

```
>> (1.5 + 1.0 I)^ (3.5 + 1.5 I)
```
$-3.19181629045628082 + 0.645658509416156807I$

PrePlus (+)

Hack to help the parser distinguish between binary and unary Plus.

```
>> +a //FullForm
```
a

Product

`Product[`*expr*`, {`*i*`, `*imin*`, `*imax*`}]`
: evaluates the discrete product of *expr* with *i* ranging from *imin* to *imax*.

`Product[`*expr*`, {`*i*`, `*imax*`}]`
: same as `Product[`*expr*`, {`*i*`, 1, `*imax*`}]`.

`Product[`*expr*`, {`*i*`, `*imin*`, `*imax*`, `*di*`}]`
: *i* ranges from *imin* to *imax* in steps of *di*.

`Product[`*expr*`, {`*i*`, `*imin*`, `*imax*`}, {`*j*`, `*jmin*`, `*jmax*`}, ...]`
: evaluates *expr* as a multiple product, with {*i*, ...}, {*j*, ...}, ... being in outermost-to-innermost order.

```
>> Product[k, {k, 1, 10}]
```
3 628 800

```
>> 10!
```
3 628 800

```
>> Product[x^k, {k, 2, 20, 2}]
```
x^{110}

```
>> Product[2 ^ i, {i, 1, n}]
```
$2^{\frac{n}{2}+\frac{n^2}{2}}$

Symbolic products involving the factorial are evaluated:

```
>> Product[k, {k, 3, n}]
```
$\frac{n!}{2}$

Evaluate the nth primorial:

```
>> primorial[0] = 1;
```

```
>> primorial[n_Integer] :=
   Product[Prime[k], {k, 1, n}];
```

```
>> primorial[12]
```
7 420 738 134 810

Rational

`Rational`
is the head of rational numbers.

`Rational[`*a*`, `*b*`]`
constructs the rational number a / b.

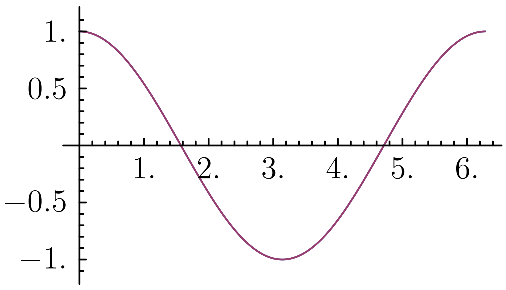
```
>> Head[1/2]
```
Rational

```
>> Rational[1, 2]
```
$\frac{1}{2}$

Re

`Re[`*z*`]`
returns the real component of the complex number *z*.

```
>> Re[3+4I]
```
3

```
>> Plot[{Cos[a], Re[E^(I a)]}, {
a, 0, 2 Pi}]
```


RealNumberQ

`RealNumberQ[`*expr*`]`
returns `True` if *expr* is an explicit number with no imaginary component.

```
>> RealNumberQ[10]
```
True

```
>> RealNumberQ[4.0]
```
True

```
>> RealNumberQ[1+I]
```
False

```
>> RealNumberQ[0 * I]
```
True

```
>> RealNumberQ[0.0 * I]
```
False

Real

`Real`
is the head of real (inexact) numbers.

```
>> x = 3. ^ -20;
```

```
>> InputForm[x]
```
2.8679719907924413*^-10

```
>> Head[x]
```
Real

Sqrt

`Sqrt[`*expr*`]`
returns the square root of *expr*.

```
>> Sqrt[4]
```
2

```
>> Sqrt[5]
```
$\sqrt{5}$

```
>> Sqrt[5] // N
```
2.23606797749979

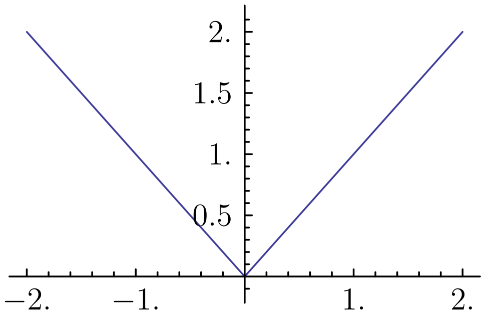
```
>> Sqrt[a]^2
```
a

Complex numbers:

```
>> Sqrt[-4]
```
$2I$

```
>> I == Sqrt[-1]
```
True

```
>> Plot[Sqrt[a^2], {a, -2, 2}]
```



Subtract (-)

`Subtract[`*a*`, `*b*`]`</dt> <dt>*a* - *b*
: represents the subtraction of *b* from *a*.

```
>> 5 - 3
```
2

```
>> a - b // FullForm
```
Plus [*a*, Times [− 1, *b*]]

```
>> a - b - c
```
$a - b - c$

```
>> a - (b - c)
```
$a - b + c$

Sum

`Sum[`*expr*`, {`*i*`, `*imin*`, `*imax*`}]`
: evaluates the discrete sum of *expr* with *i* ranging from *imin* to *imax*.

`Sum[`*expr*`, {`*i*`, `*imax*`}]`
: same as `Sum[`*expr*`, {`*i*`, 1, `*imax*`}]`.

`Sum[`*expr*`, {`*i*`, `*imin*`, `*imax*`, `*di*`}]`
: *i* ranges from *imin* to *imax* in steps of *di*.

`Sum[`*expr*`, {`*i*`, `*imin*`, `*imax*`}, {`*j*`, `*jmin*`, `*jmax*`}, ...]`
: evaluates *expr* as a multiple sum, with {*i*, ...}, {*j*, ...}, ... being in outermost-to-innermost order.

```
>> Sum[k, {k, 1, 10}]
```
55

Double sum:

```
>> Sum[i * j, {i, 1, 10}, {j, 1,
   10}]
```
3 025

Symbolic sums are evaluated:

```
>> Sum[k, {k, 1, n}]
```
$$\frac{n(1+n)}{2}$$

```
>> Sum[k, {k, n, 2 n}]
```
$$\frac{3n(1+n)}{2}$$

```
>> Sum[k, {k, I, I + 1}]
```
$1 + 2I$

```
>> Sum[1 / k ^ 2, {k, 1, n}]
```
HarmonicNumber [*n*, 2]

Verify algebraic identities:

```
>> Sum[x ^ 2, {x, 1, y}] - y * (
   y + 1)* (2 * y + 1)/ 6
```
0

```
>> (-1 + a^n)Sum[a^(k n), {k, 0,
   m-1}] // Simplify
```

$$\text{Piecewise}\left[\left\{\{m, a^n{=}{=}1\}, \left\{\frac{1-(a^n)^m}{1-a^n}, \text{True}\right\}\right\}\right](-1+a^n)$$

Infinite sums:

```
>> Sum[1 / 2 ^ i, {i, 1,
   Infinity}]
```

1

```
>> Sum[1 / k ^ 2, {k, 1,
   Infinity}]
```

$$\frac{\text{Pi}^2}{6}$$

Times (*)

`Times[`*a*`, `*b*`, ...]`</dt> <dt>*a* `*` *b* `* ...`</dt> <dt>*a* *b* `...`
: represents the product of the terms *a*, *b*, ...

```
>> 10 * 2
```

20

```
>> 10 2
```

20

```
>> a * a
```

a^2

```
>> x ^ 10 * x ^ -2
```

x^8

```
>> {1, 2, 3} * 4
```

$\{4, 8, 12\}$

```
>> Times @@ {1, 2, 3, 4}
```

24

```
>> IntegerLength[Times@@Range
   [5000]]
```

16 326

`Times` has default value 1:

```
>> DefaultValues[Times]
```

{HoldPattern [Default [Times]] :>1}

```
>> a /. n_. * x_ :> {n, x}
```

$\{1, a\}$

III. Assignment

Contents

AddTo (+=)

x `+=` *dx* is equivalent to *x* `=` *x* `+` *dx*.

```
>> a = 10;

>> a += 2
   12

>> a
   12
```

Clear

`Clear[`*symb1*`,` *symb2*`, ...]`
: clears all values of the given symbols. The arguments can also be given as strings containing symbol names.

```
>> x = 2;

>> Clear[x]

>> x
   x

>> x = 2;

>> y = 3;

>> Clear["Global`*"]

>> x
   x

>> y
   y
```

`ClearAll` may not be called for `Protected` symbols.

```
>> Clear[Sin]
   Symbol Sin is Protected.
```

The values and rules associated with built-in symbols will not get lost when applying `Clear` (after unprotecting them):

```
>> Unprotect[Sin]

>> Clear[Sin]

>> Sin[Pi]
   0
```

`Clear` does not remove attributes, messages, options, and default values associated with the symbols. Use `ClearAll` to do so.

```
>> Attributes[r] = {Flat,
   Orderless};
```

```
>> Clear["r"]
```

```
>> Attributes[r]
```
{Flat, Orderless}

ClearAll

`ClearAll[`*symb1*`, `*symb2*`, ...]`
: clears all values, attributes, messages and options associated with the given symbols. The arguments can also be given as strings containing symbol names.

```
>> x = 2;
```

```
>> ClearAll[x]
```

```
>> x
```
x

```
>> Attributes[r] = {Flat,
   Orderless};
```

```
>> ClearAll[r]
```

```
>> Attributes[r]
```
{}

`ClearAll` may not be called for `Protected` or `Locked` symbols.

```
>> Attributes[lock] = {Locked};
```

```
>> ClearAll[lock]
```
Symbol lock is locked.

Decrement (--)

```
>> a = 5;
```

```
>> a--
```
5

```
>> a
```
4

DefaultValues

```
>> Default[f, 1] = 4
```
4

```
>> DefaultValues[f]
```
$\{\text{HoldPattern}\,[\text{Default}\,[f,1]]\,{:>}4\}$

You can assign values to `DefaultValues`:

```
>> DefaultValues[g] = {Default[g
   ] -> 3};
```

```
>> Default[g, 1]
```
3

```
>> g[x_.] := {x}
```

```
>> g[a]
```
$\{a\}$

```
>> g[]
```
{3}

Definition

`Definition[`*symbol*`]`
: prints as the user-defined values and rules associated with *symbol*.

`Definition` does not print information for `ReadProtected` symbols. `Definition` uses `InputForm` to format values.

```
>> a = 2;
```

```
>> Definition[a]
```
$$a = 2$$

```
>> f[x_] := x ^ 2
```

```
>> g[f] ^:= 2
```

```
>> Definition[f]
```
$$f[x\_] = x^2$$
$$g[f] \hat{}{=}2$$

Definition of a rather evolved (though meaningless) symbol:

```
>> Attributes[r] := {Orderless}

>> Format[r[args___]] := Infix[{
   args}, "~"]

>> N[r] := 3.5

>> Default[r, 1] := 2

>> r::msg := "My message"

>> Options[r] := {Opt -> 3}

>> r[arg_., OptionsPattern[r]]
   := {arg, OptionValue[Opt]}
```

Some usage:

```
>> r[z, x, y]
```
$x \sim y \sim z$

```
>> N[r]
```
3.5

```
>> r[]
```
$\{2, 3\}$

```
>> r[5, Opt->7]
```
$\{5, 7\}$

Its definition:

```
>> Definition[r]
```
Attributes $[r]$ = {Orderless}
arg_. ∼ OptionsPattern $[r]$
= {arg, OptionValue [Opt]}
N $[r$, MachinePrecision] = 3.5
Format [args___, MathMLForm]
= Infix [{args} , "∼"]
Format [args___, OutputForm]
= Infix [{args} , "∼"]
Format [args___, StandardForm]
= Infix [{args} , "∼"]
Format [args___,
TeXForm] = Infix [{args} , "∼"]
Format [args___, TraditionalForm]
= Infix [{args} , "∼"]
Default $[r, 1]$ = 2
Options $[r]$ = {Opt->3}

For `ReadProtected` symbols, `Definition` just prints attributes, default values and options:

```
>> SetAttributes[r,
   ReadProtected]

>> Definition[r]
```
Attributes $[r]$ = {Orderless, ReadProtected}
Default $[r, 1]$ = 2
Options $[r]$ = {Opt->3}

This is the same for built-in symbols:

```
>> Definition[Plus]
```
Attributes [Plus] = {Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
Default [Plus] = 0

```
>> Definition[Level]
```
Attributes [Level] = {Protected}
Options [
Level] = {Heads->False}

`ReadProtected` can be removed, unless the symbol is locked:

```
>> ClearAttributes[r,
   ReadProtected]
```

`Clear` clears values:

```
>> Clear[r]
```

```
>> Definition[r]
```

$$\begin{aligned}\text{Attributes}[r] &= \{\text{Orderless}\}\\ \text{Default}[r,1] &= 2\\ \text{Options}[r] &= \{\text{Opt->3}\}\end{aligned}$$

`ClearAll` clears everything:

```
>> ClearAll[r]
```

```
>> Definition[r]
   Null
```

If a symbol is not defined at all, `Null` is printed:

```
>> Definition[x]
   Null
```

DivideBy (/=)

`x /= dx` is equivalent to `x = x / dx`.

```
>> a = 10;
```

```
>> a /= 2
   5
```

```
>> a
   5
```

DownValues

`DownValues[`*symbol*`]` gives the list of downvalues associated with *symbol*.

`DownValues` uses `HoldPattern` and `RuleDelayed` to protect the downvalues from being evaluated. Moreover, it has attribute `HoldAll` to get the specified symbol instead of its value.

```
>> f[x_] := x ^ 2
```

```
>> DownValues[f]
```

$\{\text{HoldPattern}[f[\text{x\_}]] :> x^2\}$

Mathics will sort the rules you assign to a symbol according to their specifity. If it cannot decide which rule is more special, the newer one will get higher precedence.

```
>> f[x_Integer] := 2
```

```
>> f[x_Real] := 3
```

```
>> DownValues[f]
```

$\{\text{HoldPattern}[f[\text{x\_Real}]] :> 3,$
$\text{HoldPattern}[f[\text{x\_Integer}]] :> 2,$
$\text{HoldPattern}[f[\text{x\_}]] :> x^2\}$

```
>> f[3]
   2
```

```
>> f[3.]
   3
```

```
>> f[a]
```

a^2

The default order of patterns can be computed using `Sort` with `PatternsOrderedQ`:

```
>> Sort[{x_, x_Integer},
   PatternsOrderedQ]
```

{x_Integer, x_}

By assigning values to `DownValues`, you can override the default ordering:

```
>> DownValues[g] := {g[x_] :> x
   ^ 2, g[x_Integer] :> x}
```

```
>> g[2]
   4
```

Fibonacci numbers:

```
>> DownValues[fib] := {fib[0] ->
    0, fib[1] -> 1, fib[n_] :>
   fib[n - 1] + fib[n - 2]}
```

```
>> fib[5]
   5
```

Increment (++)

```
>> a = 2;
```

```
>> a++
   2
>> a
   3
```

Grouping of `Increment`, `PreIncrement` and `Plus`:

```
>> ++++a+++++2//Hold//FullForm
   Hold [Plus [PreIncrement [
     PreIncrement [Increment [
     Increment [a]]]] , 2]]
```

Messages

```
>> a::b = "foo"
   foo
>> Messages[a]
   {HoldPattern [a::b] :>foo}
>> Messages[a] = {a::c :> "bar
   "};
>> a::c // InputForm
   "bar"
>> Message[a::c]
   bar
```

NValues

```
>> NValues[a]
   {}
>> N[a] = 3;
>> NValues[a]
   {HoldPattern [N [a,
     MachinePrecision]] :>3}
```

You can assign values to `NValues`:

```
>> NValues[b] := {N[b,
   MachinePrecision] :> 2}
>> N[b]
   2.
```

Be sure to use `SetDelayed`, otherwise the left-hand side of the transformation rule will be evaluated immediately, causing the head of `N` to get lost. Furthermore, you have to include the precision in the rules; `MachinePrecision` will not be inserted automatically:

```
>> NValues[c] := {N[c] :> 3}
>> N[c]
   c
```

Mathics will gracefully assign any list of rules to `NValues`; however, inappropriate rules will never be used:

```
>> NValues[d] = {foo -> bar};
>> NValues[d]
   {HoldPattern [foo] :>bar}
>> N[d]
   d
```

OwnValues

```
>> x = 3;
>> x = 2;
>> OwnValues[x]
   {HoldPattern [x] :>2}
>> x := y
>> OwnValues[x]
   {HoldPattern [x] :>y}
>> y = 5;
>> OwnValues[x]
   {HoldPattern [x] :>y}
>> Hold[x] /. OwnValues[x]
   Hold [y]
>> Hold[x] /. OwnValues[x] //
   ReleaseHold
   5
```

PreDecrement (--)

```
>> a = 2;

>> --a
   1

>> a
   1
```

PreIncrement (++)

`PreIncrement[`*x*`]` or `++`*x*
    is equivalent to *x* `=` *x* `+ 1`.

```
>> a = 2;

>> ++a
   3

>> a
   3
```

Quit

`Quit[]`
    removes all user-defined definitions.

```
>> a = 3
   3

>> Quit[]

>> a
   a
```

`Quit` even removes the definitions of protected and locked symbols:

```
>> x = 5;

>> Attributes[x] = {Locked, Protected};

>> Quit[]

>> x
   x
```

Set (=)

expr = *value*
    evaluates *value* and assigns it to *expr*.
{*s1*, *s2*, *s3*} = {*v1*, *v2*, *v3*}
    sets multiple symbols (*s1*, *s2*, ...) to the corresponding values (*v1*, *v2*, ...).

`Set` can be used to give a symbol a value:

```
>> a = 3
   3

>> a
   3
```

An assignment like this creates an own-value:

```
>> OwnValues[a]
   {HoldPattern[a]:>3}
```

You can set multiple values at once using lists:

```
>> {a, b, c} = {10, 2, 3}
   {10,2,3}

>> {a, b, {c, {d}}} = {1, 2, {{c1, c2}, {a}}}
   {1,2,{{c1,c2},{10}}}

>> d
   10
```

`Set` evaluates its right-hand side immediately and assigns it to the left-hand side:

```
>> a
   1

>> x = a
   1

>> a = 2
   2

>> x
   1
```

`Set` always returns the right-hand side, which you can again use in an assignment:

```
>> a = b = c = 2;
```

```
>>  a == b == c == 2
```
True

Set supports assignments to parts:

```
>>  A = {{1, 2}, {3, 4}};
```

```
>>  A[[1, 2]] = 5
```
5

```
>>  A
```
$\{\{1,5\},\{3,4\}\}$

```
>>  A[[;;, 2]] = {6, 7}
```
$\{6,7\}$

```
>>  A
```
$\{\{1,6\},\{3,7\}\}$

Set a submatrix:

```
>>  B = {{1, 2, 3}, {4, 5, 6},
    {7, 8, 9}};
```

```
>>  B[[1;;2, 2;;-1]] = {{t, u}, {
    y, z}};
```

```
>>  B
```
$\{\{1,t,u\},\{4,y,z\},\{7,8,9\}\}$

SetDelayed (:=)

expr := *value*
: assigns *value* to *expr*, without evaluating *value*.

`SetDelayed` is like `Set`, except it has attribute `HoldAll`, thus it does not evaluate the right-hand side immediately, but evaluates it when needed.

```
>>  Attributes[SetDelayed]
```
{HoldAll, Protected, SequenceHold}

```
>>  a = 1
```
1

```
>>  x := a
```

```
>>  x
```
1

Changing the value of a affects x:

```
>>  a = 2
```
2

```
>>  x
```
2

`Condition` (`/;`) can be used with `SetDelayed` to make an assignment that only holds if a condition is satisfied:

```
>>  f[x_] := p[x] /; x>0
```

```
>>  f[3]
```
$p[3]$

```
>>  f[-3]
```
$f[-3]$

SubValues

```
>>  f[1][x_] := x
```

```
>>  f[2][x_] := x ^ 2
```

```
>>  SubValues[f]
```
$\{\text{HoldPattern}[f[2][\text{x\_}]] :> x^2, \text{HoldPattern}[f[1][\text{x\_}]] :> x\}$

```
>>  Definition[f]
```
$$f[2][\text{x\_}] = x^2$$
$$f[1][\text{x\_}] = x$$

SubtractFrom (-=)

x `-=` dx is equivalent to x `=` x `-` dx.

```
>>  a = 10;
```

```
>>  a -= 2
```
8

```
>>  a
```
8

TagSet

`TagSet[`*f*`, `*lhs*`, `*rhs*`]` or `f /: lhs = rhs`
: sets *lhs* to be *rhs* and assigns the corresponding rule to the symbol *f*.

```
>> x /: f[x] = 2
```
2

```
>> f[x]
```
2

```
>> DownValues[f]
```
{}

```
>> UpValues[x]
```
{HoldPattern [f [x]] :>2}

The symbol *f* must appear as the ultimate head of *lhs* or as the head of a leaf in *lhs*:

```
>> x /: f[g[x]] = 3;
```
Tag *x* not found or too deep for an assigned rule.

```
>> g /: f[g[x]] = 3;
```

```
>> f[g[x]]
```
3

TagSetDelayed

`TagSetDelayed[`*f*`, `*lhs*`, `*rhs*`]` or `f /: lhs := rhs`
: is the delayed version of `TagSet`.

TimesBy (*=)

x `*=` *dx* is equivalent to *x* `=` *x* `*` *dx*.

```
>> a = 10;
```

```
>> a *= 2
```
20

```
>> a
```
20

Unset (=.)

`Unset[`*x*`]` or *x*`=.`
: removes any value belonging to *x*.

```
>> a = 2
```
2

```
>> a =.
```

```
>> a
```
a

Unsetting an already unset or never defined variable will not change anything:

```
>> a =.
```

```
>> b =.
```

`Unset` can unset particular function values. It will print a message if no corresponding rule is found.

```
>> f[x_] =.
```
Assignment on *f* for *f* [x_] not found.

$Failed

```
>> f[x_] := x ^ 2
```

```
>> f[3]
```
9

```
>> f[x_] =.
```

```
>> f[3]
```
f [3]

You can also unset `OwnValues`, `DownValues`, `SubValues`, and `UpValues` directly. This is equivalent to setting them to `{}`.

```
>> f[x_] = x; f[0] = 1;
```

```
>> DownValues[f] =.
```

```
>> f[2]
```
f [2]

`Unset` threads over lists:

```
>> a = b = 3;
```

```
>> {a, {b}} =.
```
{Null, {Null}}

UpSet (^=)

f[*x*] ^= *expression*
: evaluates *expression* and assigns it to the value of *f*[*x*], associating the value with *x*.

`UpSet` creates an upvalue:

```
>> a[b] ^= 3;
```

```
>> DownValues[a]
```
{}

```
>> UpValues[b]
```
{HoldPattern [*a* [*b*]] :>3}

```
>> a ^= 3
```
Nonatomic expression expected.

3

You can use `UpSet` to specify special values like format values. However, these values will not be saved in `UpValues`:

```
>> Format[r] ^= "custom";
```

```
>> r
```
custom

```
>> UpValues[r]
```
{}

UpSetDelayed (^:=)

```
>> a[b] ^:= x
```

```
>> x = 2;
```

```
>> a[b]
```
2

```
>> UpValues[b]
```
{HoldPattern [*a* [*b*]] :>*x*}

UpValues

```
>> a + b ^= 2
```
2

```
>> UpValues[a]
```
{HoldPattern [*a* + *b*] :>2}

```
>> UpValues[b]
```
{HoldPattern [*a* + *b*] :>2}

You can assign values to `UpValues`:

```
>> UpValues[pi] := {Sin[pi] :>
   0}
```

```
>> Sin[pi]
```
0

IV. Attributes

Contents

Attributes

`Attributes`[*symbol*]
returns the attributes of *symbol*.

`Attributes`[*symbol*] = {*attr1*, *attr2*}
sets the attributes of *symbol*, replacing any existing attributes.

```
>> Attributes[Plus]
```
{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}

`Attributes` always considers the head of an expression:

```
>> Attributes[a + b + c]
```
{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}

You can assign values to `Attributes` to set attributes:

```
>> Attributes[f] = {Flat, Orderless}
```
{Flat, Orderless}

```
>> f[b, f[a, c]]
```
$f[a, b, c]$

Attributes must be symbols:

```
>> Attributes[f] := {a + b}
```
Argument $a + b$ at position 1 is expected to be a symbol.

$Failed

Use `Symbol` to convert strings to symbols:

```
>> Attributes[f] = Symbol["Listable"]
```
Listable

```
>> Attributes[f]
```
{Listable}

ClearAttributes

`ClearAttributes`[*symbol*, *attrib*]
removes *attrib* from *symbol*'s attributes.

```
>> SetAttributes[f, Flat]
```

```
>> Attributes[f]
   {Flat}

>> ClearAttributes[f, Flat]

>> Attributes[f]
   {}
```

Attributes that are not even set are simply ignored:

```
>> ClearAttributes[{f}, {Flat}]

>> Attributes[f]
   {}
```

Constant

`Constant`
: is an attribute that indicates that a symbol is a constant.

Mathematical constants like `E` have attribute `Constant`:

```
>> Attributes[E]
   {Constant, Protected,
    ReadProtected}
```

Constant symbols cannot be used as variables in `Solve` and related functions:

```
>> Solve[x + E == 0, E]
   E is not a valid variable.
   Solve[E + x==0, E]
```

Flat

`Flat`
: is an attribute that specifies that nested occurrences of a function should be automatically flattened.

A symbol with the `Flat` attribute represents an associative mathematical operation:

```
>> SetAttributes[f, Flat]

>> f[a, f[b, c]]
   f[a,b,c]
```

`Flat` is taken into account in pattern matching:

```
>> f[a, b, c] /. f[a, b] -> d
   f[d,c]
```

HoldAll

`HoldAll`
: is an attribute specifying that all arguments of a function should be left unevaluated.

HoldAllComplete

`HoldAllComplete`
: is an attribute that includes the effects of `HoldAll` and `SequenceHold`, and also protects the function from being affected by the upvalues of any arguments.

`HoldAllComplete` even prevents upvalues from being used, and includes `SequenceHold`.

```
>> SetAttributes[f,
   HoldAllComplete]

>> f[a] ^= 3;

>> f[a]
   f[a]

>> f[Sequence[a, b]]
   f[Sequence[a,b]]
```

HoldFirst

`HoldFirst`
is an attribute specifying that the first argument of a function should be left unevaluated.

HoldRest

`HoldRest`
is an attribute specifying that all but the first argument of a function should be left unevaluated.

Listable

`Listable`
is an attribute specifying that a function should be automatically applied to each element of a list.

```
>> SetAttributes[f, Listable]
```

```
>> f[{1, 2, 3}, {4, 5, 6}]
```
$\{f[1,4], f[2,5], f[3,6]\}$

```
>> f[{1, 2, 3}, 4]
```
$\{f[1,4], f[2,4], f[3,4]\}$

```
>> {{1, 2}, {3, 4}} + {5, 6}
```
$\{\{6,7\}, \{9,10\}\}$

Locked

`Locked`
is an attribute that prevents attributes on a symbol from being modified.

The attributes of `Locked` symbols cannot be modified:

```
>> Attributes[lock] = {Flat,
   Locked};
```

```
>> SetAttributes[lock, {}]
```
Symbol lock is locked.

```
>> ClearAttributes[lock, Flat]
```
Symbol lock is locked.

```
>> Attributes[lock] = {}
```
Symbol lock is locked.

{}

```
>> Attributes[lock]
```
{Flat, Locked}

However, their values might be modified (as long as they are not `Protected` too):

```
>> lock = 3
```
3

NHoldAll

`NHoldAll`
is an attribute that protects all arguments of a function from numeric evaluation.

```
>> N[f[2, 3]]
```
$f[2., 3.]$

```
>> SetAttributes[f, NHoldAll]
```

```
>> N[f[2, 3]]
```
$f[2, 3]$

NHoldFirst

`NHoldFirst`
is an attribute that protects the first argument of a function from numeric evaluation.

NHoldRest

`NHoldRest`
: is an attribute that protects all but the first argument of a function from numeric evaluation.

OneIdentity

`OneIdentity`
: is an attribute specifying that $f[x]$ should be treated as equivalent to x in pattern matching.

`OneIdentity` affects pattern matching:

```
>> SetAttributes[f, OneIdentity]

>> a /. f[args___] -> {args}
```
$\{a\}$

It does not affect evaluation:

```
>> f[a]
```
$f[a]$

Orderless

`Orderless`
: is an attribute indicating that the leaves in an expression f[a, b, c] can be placed in any order.

The leaves of an `Orderless` function are automatically sorted:

```
>> SetAttributes[f, Orderless]

>> f[c, a, b, a + b, 3, 1.0]
```
$f[1., 3, a, b, c, a + b]$

A symbol with the `Orderless` attribute represents a commutative mathematical operation.

```
>> f[a, b] == f[b, a]
```
True

`Orderless` affects pattern matching:

```
>> SetAttributes[f, Flat]

>> f[a, b, c] /. f[a, c] -> d
```
$f[b, d]$

Protect

`Protect[`*symbol*`]`
: gives *symbol* the attribute `Protected`.

```
>> A = {1, 2, 3};

>> Protect[A]

>> A[[2]] = 4;
```
Symbol A is Protected.

```
>> A
```
$\{1, 2, 3\}$

Protected

`Protected`
: is an attribute that prevents values on a symbol from being modified.

Values of `Protected` symbols cannot be modified:

```
>> Attributes[p] = {Protected};

>> p = 2;
```
Symbol p is Protected.

```
>> f[p] ^= 3;
```
Tag p in $f[p]$ is Protected.

```
>> Format[p] = "text";
```
Symbol p is Protected.

However, attributes might still be set:

```
>> SetAttributes[p, Flat]
```

```
>> Attributes[p]
```
$\{\text{Flat}, \text{Protected}\}$

Thus, you can easily remove the attribute `Protected`:

```
>> Attributes[p] = {};
```

```
>> p = 2
```
2

You can also use `Protect` or `Unprotect`, resp.

```
>> Protect[p]
```

```
>> Attributes[p]
```
$\{\text{Protected}\}$

```
>> Unprotect[p]
```

If a symbol is `Protected` and `Locked`, it can never be changed again:

```
>> SetAttributes[p, {Protected,
   Locked}]
```

```
>> p = 2
```
Symbol p is Protected.

2

```
>> Unprotect[p]
```
Symbol p is locked.

ReadProtected

`ReadProtected`
is an attribute that prevents values on a symbol from being read.

Values associated with `ReadProtected` symbols cannot be seen in `Definition`:

```
>> ClearAll[p]
```

```
>> p = 3;
```

```
>> Definition[p]
```
$$p = 3$$

```
>> SetAttributes[p,
   ReadProtected]
```

```
>> Definition[p]
```
$\text{Attributes}\left[p\right] = \{\text{ReadProtected}\}$

SequenceHold

`SequenceHold`
is an attribute that prevents `Sequence` objects from being spliced into a function's arguments.

Normally, `Sequence` will be spliced into a function:

```
>> f[Sequence[a, b]]
```
$f[a, b]$

It does not for `SequenceHold` functions:

```
>> SetAttributes[f, SequenceHold
   ]
```

```
>> f[Sequence[a, b]]
```
$f\left[\text{Sequence}[a, b]\right]$

E.g., `Set` has attribute `SequenceHold` to allow assignment of sequences to variables:

```
>> s = Sequence[a, b];
```

```
>> s
```
$\text{Sequence}[a, b]$

```
>> Plus[s]
```
$a + b$

SetAttributes

`SetAttributes`[*symbol*, *attrib*]
adds *attrib* to *symbol*'s attributes.

```
>> SetAttributes[f, Flat]
```

```
>> Attributes[f]
```
$\{\text{Flat}\}$

Multiple attributes can be set at the same time using lists:

```
>> SetAttributes[{f, g}, {Flat,
   Orderless}]

>> Attributes[g]
```

{Flat, Orderless}

Unprotect

`Unprotect`[*symbol*]
: removes the `Protected` attribute from *symbol*.

V. Calculus

Contents

Complexes

`Complexes`
is the set of complex numbers.

D

`D[f, x]`
gives the partial derivative of f with respect to x.
`D[f, x, y, ...]`
differentiates successively with respect to x, y, etc.
`D[f, {x, n}]`
gives the multiple derivative of order n.
`D[f, {{x1, x2, ...}}]`
gives the vector derivative of f with respect to $x1$, $x2$, etc.

First-order derivative of a polynomial:

```
>> D[x^3 + x^2, x]
```
$2x + 3x^2$

Second-order derivative:

```
>> D[x^3 + x^2, {x, 2}]
```
$2 + 6x$

Trigonometric derivatives:

```
>> D[Sin[Cos[x]], x]
```
$-\mathrm{Cos}[\mathrm{Cos}[x]]\,\mathrm{Sin}[x]$

```
>> D[Sin[x], {x, 2}]
```
$-\mathrm{Sin}[x]$

```
>> D[Cos[t], {t, 2}]
```
$-\mathrm{Cos}[t]$

Unknown variables are treated as constant:

```
>> D[y, x]
```
0

```
>> D[x, x]
```
1

```
>> D[x + y, x]
```
1

Derivatives of unknown functions are represented using `Derivative`:

```
>> D[f[x], x]
```
$f'[x]$

```
>> D[f[x, x], x]
```
$f^{(0,1)}[x, x] + f^{(1,0)}[x, x]$

```
>> D[f[x, x], x] // InputForm
```
$\mathrm{Derivative}[0, 1]\left[f\right][x, x] + \mathrm{Derivative}[1, 0]\left[f\right][x, x]$

Chain rule:

```
>> D[f[2x+1, 2y, x+y], x]
```
$2f^{(1,0,0)}\left[1 + 2x, 2y, x + y\right] + f^{(0,0,1)}\left[1 + 2x, 2y, x + y\right]$

```
>> D[f[x^2, x, 2y], {x,2}, y] // Expand
```

$8xf^{(1,1,1)}[x^2, x, 2y] + 8x^2 f^{(2,0,1)}[x^2, x, 2y] + 2f^{(0,2,1)}[x^2, x, 2y] + 4f^{(1,0,1)}[x^2, x, 2y]$

Compute the gradient vector of a function:

```
>> D[x ^ 3 * Cos[y], {{x, y}}]
```

$\{3x^2\text{Cos}[y], -x^3\text{Sin}[y]\}$

Hesse matrix:

```
>> D[Sin[x] * Cos[y], {{x,y}, 2}]
```

$\{\{-\text{Cos}[y]\,\text{Sin}[x], -\text{Cos}[x]\,\text{Sin}[y]\}, \{-\text{Cos}[x]\,\text{Sin}[y], -\text{Cos}[y]\,\text{Sin}[x]\}\}$

Derivative (')

`Derivative[`*n*`][`*f*`]`
: represents the *n*th derivative of the function *f*.

`Derivative[`*n1*`, `*n2*`, ...][`*f*`]`
: represents a multivariate derivative.

```
>> Derivative[1][Sin]
```

Cos[#1]&

```
>> Derivative[3][Sin]
```

−Cos[#1]&

```
>> Derivative[2][# ^ 3&]
```

6#1&

`Derivative` can be entered using `'`:

```
>> Sin'[x]
```

Cos[x]

```
>> (# ^ 4&)''
```

$12\#1^2\&$

```
>> f'[x] // InputForm
```

Derivative[1][f][x]

```
>> Derivative[1][#2 Sin[#1]+Cos[#2]&]
```

Cos[#1] #2&

```
>> Derivative[1,2][#2^3 Sin[#1]+Cos[#2]&]
```

6Cos[#1] #2&

Deriving with respect to an unknown parameter yields 0:

```
>> Derivative[1,2,1][#2^3 Sin[#1]+Cos[#2]&]
```

0&

The 0th derivative of any expression is the expression itself:

```
>> Derivative[0,0,0][a+b+c]
```

$a + b + c$

You can calculate the derivative of custom functions:

```
>> f[x_] := x ^ 2
```

```
>> f'[x]
```

$2x$

Unknown derivatives:

```
>> Derivative[2, 1][h]
```

$h^{(2,1)}$

```
>> Derivative[2, 0, 1, 0][h[g]]
```

$h[g]^{(2,0,1,0)}$

FindRoot

`FindRoot[`*f*`, {`*x*`, `*x0*`}]`
: searches for a numerical root of *f*, starting from *x*=*x0*.

`FindRoot[`*lhs* `==` *rhs*`, {`*x*`, `*x0*`}]`
: tries to solve the equation *lhs* `==` *rhs*.

`FindRoot` uses Newton's method, so the function of interest should have a first derivative.

```
>> FindRoot[Cos[x], {x, 1}]
```

$\{x\text{->}1.57079632679489662\}$

```
>> FindRoot[Sin[x] + Exp[x],{x,
   0}]
```

$\{x\text{->}-0.588532743981861077\}$

```
>> FindRoot[Sin[x] + Exp[x] ==
   Pi,{x, 0}]
```

$\{x\text{->}0.866815239911458064\}$

`FindRoot` has attribute `HoldAll` and effectively uses `Block` to localize x. However, in the result x will eventually still be replaced by its value.

```
>> x = 3;
```

```
>> FindRoot[Tan[x] + Sin[x] ==
   Pi, {x, 1}]
```

$\{3\text{->}1.14911295431426855\}$

```
>> Clear[x]
```

`FindRoot` stops after 100 iterations:

```
>> FindRoot[x^2 + x + 1, {x, 1}]
```

The maximum number of iterations was exceeded. The result might be inaccurate.

$\{x\text{->}-1.\}$

Find complex roots:

```
>> FindRoot[x ^ 2 + x + 1, {x, -
   I}]
```

$\{x\text{->}-0.5-0.866\tilde{}\tilde{}0254037844386471I\}$

The function has to return numerical values:

```
>> FindRoot[f[x] == 0, {x, 0}]
```

The function value is not a number at $x = 0.$.

$\text{FindRoot}\left[f\left[x\right]-0, \{x,0\}\right]$

The derivative must not be 0:

```
>> FindRoot[Sin[x] == x, {x, 0}]
```

Encountered a singular derivative at the point $x = 0.$.

$\text{FindRoot}\left[\text{Sin}\left[x\right]-x, \{x,0\}\right]$

Integrate

`Integrate[`f`, `x`]`
: integrates f with respect to x. The result does not contain the additive integration constant.

`Integrate[`f`, {`x`, `a`, `b`}]`
: computes the definite integral of f with respect to x from a to b.

Integrate a polynomial:

```
>> Integrate[6 x ^ 2 + 3 x ^ 2 -
    4 x + 10, x]
```

$10x-2x^2+3x^3$

Integrate trigonometric functions:

```
>> Integrate[Sin[x] ^ 5, x]
```

$$-\text{Cos}\left[x\right]-\frac{\text{Cos}\left[x\right]^5}{5}+\frac{2\text{Cos}\left[x\right]^3}{3}$$

Definite integrals:

```
>> Integrate[x ^ 2 + x, {x, 1,
   3}]
```

$$\frac{38}{3}$$

```
>> Integrate[Sin[x], {x, 0, Pi
   /2}]
```

1

Some other integrals:

```
>> Integrate[1 / (1 - 4 x + x^2)
   , x]
```

$$-\frac{\sqrt{3}\text{Log}\left[-2+\sqrt{3}+x\right]}{6}+\frac{\sqrt{3}\text{Log}\left[-2-\sqrt{3}+x\right]}{6}$$

```
>> Integrate[4 Sin[x] Cos[x], x]
```

$2\text{Sin}\left[x\right]^2$

Integration in TeX:

```
>> Integrate[f[x], {x, a, b}] //
    TeXForm
```

\int_a^bf\left[x\right] \, dx

```
>> Integrate[ArcSin[x / 3], x]
```
$$x\text{ArcSin}\left[\frac{x}{3}\right]+\sqrt{9-x^2}$$

```
>> Integrate[f'[x], {x, a, b}]
```
$$f[b]-f[a]$$

Limit

`Limit[`*expr*`, `*x*`->`*x0*`]`
: gives the limit of *expr* as *x* approaches *x0*.

`Limit[`*expr*`, `*x*`->`*x0*`, Direction->1]`
: approaches *x0* from smaller values.

`Limit[`*expr*`, `*x*`->`*x0*`, Direction->-1]`
: approaches *x0* from larger values.

```
>> Limit[x, x->2]
```
2

```
>> Limit[Sin[x] / x, x->0]
```
1

```
>> Limit[1/x, x->0, Direction
   ->-1]
```
∞

```
>> Limit[1/x, x->0, Direction
   ->1]
```
$-\infty$

Reals

`Reals`
: is the set of real numbers.

Limit a solution to real numbers:

```
>> Solve[x^3 == 1, x, Reals]
```
$\{\{x\text{->}1\}\}$

Solve

`Solve[`*equation*`, `*vars*`]`
: attempts to solve *equation* for the variables *vars*.

`Solve[`*equation*`, `*vars*`, `*domain*`]`
: restricts variables to *domain*, which can be `Complexes` or `Reals`.

```
>> Solve[x ^ 2 - 3 x == 4, x]
```
$\{\{x\text{->}-1\},\{x\text{->}4\}\}$

```
>> Solve[4 y - 8 == 0, y]
```
$\{\{y\text{->}2\}\}$

Apply the solution:

```
>> sol = Solve[2 x^2 - 10 x - 12
    == 0, x]
```
$\{\{x\text{->}-1\},\{x\text{->}6\}\}$

```
>> x /. sol
```
$\{-1,6\}$

Contradiction:

```
>> Solve[x + 1 == x, x]
```
$\{\}$

Tautology:

```
>> Solve[x ^ 2 == x ^ 2, x]
```
$\{\{\}\}$

Rational equations:

```
>> Solve[x / (x ^ 2 + 1)== 1, x]
```
$$\left\{\left\{x\text{->}\frac{1}{2}-\frac{I}{2}\sqrt{3}\right\},\left\{x\text{->}\frac{1}{2}+\frac{I}{2}\sqrt{3}\right\}\right\}$$

```
>> Solve[(x^2 + 3 x + 2)/(4 x -
   2)== 0, x]
```
$\{\{x\text{->}-2\},\{x\text{->}-1\}\}$

Transcendental equations:

```
>> Solve[Cos[x] == 0, x]
```
$$\left\{\left\{x\text{->}\frac{\text{Pi}}{2}\right\},\left\{x\text{->}\frac{3\text{Pi}}{2}\right\}\right\}$$

Solve can only solve equations with respect to symbols or functions:

```
>> Solve[f[x + y] == 3, f[x + y
]]
```
$\{\{f[x+y]\text{->}3\}\}$

```
>> Solve[a + b == 2, a + b]
```
$a+b$ is not a valid variable.

$\text{Solve}[a+b==2, a+b]$

This happens when solving with respect to an assigned symbol:

```
>> x = 3;
```

```
>> Solve[x == 2, x]
```
3 is not a valid variable.

Solve [False, 3]

```
>> Clear[x]
```

```
>> Solve[a < b, a]
```
$a<b$ is not a well-formed equation.

$\text{Solve}[a<b, a]$

Solve a system of equations:

```
>> eqs = {3 x ^ 2 - 3 y == 0, 3
y ^ 2 - 3 x == 0};
```

```
>> sol = Solve[eqs, {x, y}]
```
$$\left\{\{x\text{->}0, y\text{->}0\}, \{x\text{->}1, y\text{->}1\}, \left\{x\text{->}-\frac{1}{2}+\frac{I}{2}\sqrt{3}, y\text{->}-\frac{1}{2}-\frac{I}{2}\sqrt{3}\right\}, \left\{x\text{->}\frac{\left(1-I\sqrt{3}\right)^2}{4}, y\text{->}-\frac{1}{2}+\frac{I}{2}\sqrt{3}\right\}\right\}$$

```
>> eqs /. sol // Simplify
```
{{True, True}, {True, True}, {True, True}, {True, True}}

An underdetermined system:

```
>> Solve[x^2 == 1 && z^2 == -1,
{x, y, z}]
```
Equations may not give solutions for all "solve" variables.

$\{\{x\text{->}-1, z\text{->}-I\}, \{x\text{->}-1, z\text{->}I\}, \{x\text{->}1, z\text{->}-I\}, \{x\text{->}1, z\text{->}I\}\}$

Domain specification:

```
>> Solve[x^2 == -1, x, Reals]
```
{}

```
>> Solve[x^2 == 1, x, Reals]
```
$\{\{x\text{->}-1\}, \{x\text{->}1\}\}$

```
>> Solve[x^2 == -1, x, Complexes
]
```
$\{\{x\text{->}-I\}, \{x\text{->}I\}\}$

VI. Combinatorial

Contents

Binomial

`Binomial[`*n*`, `*k*`]`
gives the binomial coefficient *n* choose *k*.

```
>> Binomial[5, 3]
```
10

`Binomial` supports inexact numbers:
```
>> Binomial[10.5,3.2]
```
165.286109367256421

Some special cases:
```
>> Binomial[10, -2]
```
0

```
>> Binomial[-10.5, -3.5]
```
0.

```
>> Binomial[-10, -3.5]
```
ComplexInfinity

Fibonacci

`Fibonacci[`*n*`]`
computes the *n*th Fibonacci number.

```
>> Fibonacci[0]
```
0

```
>> Fibonacci[1]
```
1

```
>> Fibonacci[10]
```
55

```
>> Fibonacci[200]
```
280 571 172 992 510 140 037~
~611 932 413 038 677 189 525

Multinomial

`Multinomial[`*n1*`, `*n2*`, ...]`
gives the multinomial coefficient `(`*n1*`+`*n2*`+...)!/(`*n1*`!`*n2*`!...)`.

```
>> Multinomial[2, 3, 4, 5]
```
2 522 520

```
>> Multinomial[]
```
1

Multinomial is expressed in terms of `Binomial`:
```
>> Multinomial[a, b, c]
```
Binomial $[a + b, b]$ Binomial $[a + b + c, c]$

`Multinomial[`*n*`-`*k*`, `*k*`]` is equivalent to `Binomial[`*n*`, `*k*`]`.
```
>> Multinomial[2, 3]
```
10

VII. Comparison

Contents

Equal (==)

```
>> a==a
```
True

```
>> a==b
```
a==b

```
>> 1==1.
```
True

Lists are compared based on their elements:

```
>> {{1}, {2}} == {{1}, {2}}
```
True

```
>> {1, 2} == {1, 2, 3}
```
False

Real values are considered equal if they only differ in their last digits:

```
>> 0.739085133215160642 ==
   0.739085133215160641
```
True

```
>> 0.73908513321516064200000000
   ==
   0.73908513321516064100000000
```
False

```
>> 0.1 ^ 10000 == 0.1 ^ 10000 +
   0.1 ^ 10016
```
False

```
>> 0.1 ^ 10000 == 0.1 ^ 10000 +
   0.1 ^ 10017
```
True

Comparisons are done using the lower precision:

```
>> N[E, 100] == N[E, 150]
```
True

Symbolic constants are compared numerically:

```
>> E > 1
```
True

```
>> Pi == 3.14
```
False

Greater (>)

```
>> a > b > c //FullForm
```
Greater $[a, b, c]$

```
>> Greater[3, 2, 1]
```
True

GreaterEqual (>=)

Inequality

`Inequality` is the head of expressions involving different inequality operators (at least temporarily). Thus, it is possible to write chains of inequalities.

```
>> a < b <= c
```
$a < b\&\&b<=c$

```
>> Inequality[a, Greater, b,
   LessEqual, c]
```
$a > b\&\&b<=c$

```
>> 1 < 2 <= 3
```
True

```
>> 1 < 2 > 0
```
True

```
>> 1 < 2 < -1
```
False

Less (<)

LessEqual (<=)

Max

`Max`[*e_1*, *e_2*, ..., *e_i*]
  returns the expression with the greatest value among the *e_i*.

Maximum of a series of numbers:

```
>> Max[4, -8, 1]
```
4

`Max` flattens lists in its arguments:

```
>> Max[{1,2},3,{-3,3.5,-Infinity
   },{{1/2}}]
```
3.5

`Max` with symbolic arguments remains in symbolic form:

```
>> Max[x, y]
```
$\text{Max}\left[x,y\right]$

```
>> Max[5, x, -3, y, 40]
```
$\text{Max}\left[40,x,y\right]$

With no arguments, `Max` gives `-Infinity`:

```
>> Max[]
```
$-\infty$

Min

`Min`[*e_1*, *e_2*, ..., *e_i*]
  returns the expression with the lowest value among the *e_i*.

Minimum of a series of numbers:

```
>> Min[4, -8, 1]
```
-8

`Min` flattens lists in its arguments:

```
>> Min[{1,2},3,{-3,3.5,-Infinity
   },{{1/2}}]
```
$-\infty$

`Min` with symbolic arguments remains in symbolic form:

```
>> Min[x, y]
```
$\text{Min}\left[x,y\right]$

```
>> Min[5, x, -3, y, 40]
```
$\text{Min}\left[-3,x,y\right]$

With no arguments, `Min` gives `Infinity`:

```
>> Min[]
```
∞

Negative

```
>> Negative[-3]
```
True

```
>> Negative[10/7]
```
False

```
>> Negative[1+2I]
```
False

```
>> Negative[a+b]
False
```

NonNegative

NonPositive

Positive

SameQ (===)

```
>> a===a
True
```

```
>> 1===1
True
```

```
>> 1===1.
False
```

Unequal (!=)

```
>> 1 != 1.
False
```

Lists are compared based on their elements:

```
>> {1} != {2}
True
```

```
>> {1, 2} != {1, 2}
False
```

```
>> {a} != {a}
False
```

```
>> "a" != "b"
True
```

```
>> "a" != "a"
False
```

UnsameQ (=!=)

```
>> a=!=a
False
```

```
>> 1=!=1.
True
```

VIII. Control

Contents

Abort

`Abort[]`
: aborts an evaluation completely and returns `$Aborted`.

```
>> Print["a"]; Abort[]; Print["b"]
```
a
$Aborted

Break

`Break[]`
: exits a `For`, `While`, or `Do` loop.

```
>> n = 0;
```

```
>> While[True, If[n>10, Break[]]; n=n+1]
```

```
>> n
```
11

CompoundExpression (;)

`CompoundExpression[`*e1*`, `*e2*`, ...]` or *e1*`; `*e2*`; ...`
: evaluates its arguments in turn, returning the last result.

```
>> a; b; c; d
```
d

If the last argument is omitted, `Null` is taken:

```
>> a;
```

Continue

`Continue[]`
: continues with the next iteration in a `For`, `While`, or `Do` loop.

```
>> For[i=1, i<=8, i=i+1, If[Mod[i,2] == 0, Continue[]]; Print[i]]
```
1
3
5
7

Do

`Do[`*expr*`, {`*max*`}]`
: evaluates *expr* *max* times.

`Do[`*expr*`, {`*i*`, `*max*`}]`
: evaluates *expr* *max* times, substituting *i* in *expr* with values from 1 to *max*.

`Do[`*expr*`, {`*i*`, `*min*`, `*max*`}]`
: starts with *i* = *max*.

`Do[`*expr*`, {`*i*`, `*min*`, `*max*`, `*step*`}]`
: uses a step size of *step*.

`Do[`*expr*`, {`*i*`, {`*i1*`, `*i2*`, ...}}]`
: uses values *i1*, *i2*, ... for *i*.

`Do[`*expr*`, {`*i*`, `*imin*`, `*imax*`}, {`*j*`, `*jmin*`, `*jmax*`}, ...]`
: evaluates *expr* for each *j* from *jmin* to *jmax*, for each *i* from *imin* to *imax*, etc.

```
>> Do[Print[i], {i, 2, 4}]
   2
   3
   4

>> Do[Print[{i, j}], {i,1,2}, {j
   ,3,5}]
   {1,3}
   {1,4}
   {1,5}
   {2,3}
   {2,4}
   {2,5}
```

You can use `Break[]` and `Continue[]` inside `Do`:

```
>> Do[If[i > 10, Break[], If[Mod
   [i, 2] == 0, Continue[]];
   Print[i]], {i, 5, 20}]
   5
   7
   9
```

FixedPoint

`FixedPoint[`*f*`, `*expr*`]`
: starting with *expr*, iteratively applies *f* until the result no longer changes.

`FixedPoint[`*f*`, `*expr*`, `*n*`]`
: performs at most *n* iterations.

```
>> FixedPoint[Cos, 1.0]
   0.739085133215160639

>> FixedPoint[#+1 &, 1, 20]
   21
```

FixedPointList

`FixedPointList[`*f*`, `*expr*`]`
: starting with *expr*, iteratively applies *f* until the result no longer changes, and returns a list of all intermediate results.

`FixedPointList[`*f*`, `*expr*`, `*n*`]`
: performs at most *n* iterations.

```
>> FixedPointList[Cos, 1.0, 4]
   {1., 0.540302305868139`
    `717, 0.857553215846393`
    `416, 0.65428979049777915
    , 0.793480358742565592}
```

Observe the convergence of Newton's method for approximating square roots:

```
>> newton[n_] := FixedPointList
   [.5(# + n/#)&, 1.];

>> newton[9]
   {1., 5., 3.4, 3.023529411764`
    `70588, 3.000091554131380`
    `18, 3.00000000139698386
    , 3.00000000000000001, 3.}
```

Plot the "hailstone" sequence of a number:

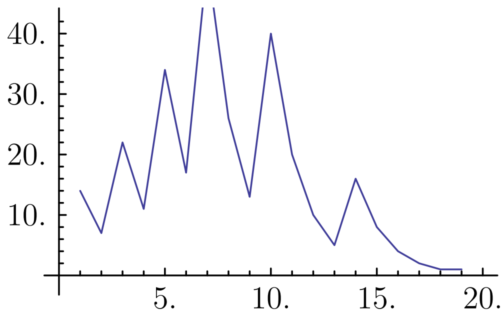
```
>> collatz[1] := 1;

>> collatz[x_ ? EvenQ] := x / 2;
```

```
>> collatz[x_] := 3 x + 1;
```

```
>> list = FixedPointList[collatz
   , 14]
```

$\{14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 1\}$

```
>> ListLinePlot[list]
```



For

`For[`*start*`,` *test*`,` *incr*`,` *body*`]`
: evaluates *start*, and then iteratively *body* and *incr* as long as *test* evaluates to `True`.

`For[`*start*`,` *test*`,` *incr*`]`
: evaluates only *incr* and no *body*.

`For[`*start*`,` *test*`]`
: runs the loop without any body.

Compute the factorial of 10 using `For`:

```
>> n := 1
```

```
>> For[i=1, i<=10, i=i+1, n = n
   * i]
```

```
>> n
```

3 628 800

```
>> n == 10!
```

True

If

`If[`*cond*`,` *pos*`,` *neg*`]`
: returns *pos* if *cond* evaluates to `True`, and *neg* if it evaluates to `False`.

`If[`*cond*`,` *pos*`,` *neg*`,` *other*`]`
: returns *other* if *cond* evaluates to neither `True` nor `False`.

`If[`*cond*`,` *pos*`]`
: returns `Null` if *cond* evaluates to `False`.

```
>> If[1<2, a, b]
```

a

If the second branch is not specified, `Null` is taken:

```
>> If[1<2, a]
```

a

```
>> If[False, a] //FullForm
```

Null

You might use comments (inside `(*` and `*)`) to make the branches of `If` more readable:

```
>> If[a, (*then*)b, (*else*)c];
```

Nest

`Nest[`*f*`,` *expr*`,` *n*`]`
: starting with *expr*, iteratively applies *f* *n* times and returns the final result.

```
>> Nest[f, x, 3]
```

$f[f[f[x]]]$

```
>> Nest[(1+#)^ 2 &, x, 2]
```

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

NestList

`NestList[`*f*`, `*expr*`, `*n*`]`
: starting with *expr*, iteratively applies *f* *n* times and returns a list of all intermediate results.

```
>> NestList[f, x, 3]
```
$\{x, f[x], f[f[x]], f[f[f[x]]]\}$

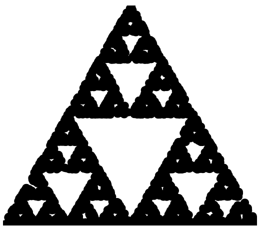
```
>> NestList[2 # &, 1, 8]
```
$\{1, 2, 4, 8, 16, 32, 64, 128, 256\}$

Chaos game rendition of the Sierpinski triangle:

```
>> vertices = {{0,0}, {1,0},
   {.5, .5 Sqrt[3]}};
```

```
>> points = NestList[.5(vertices
   [[ RandomInteger[{1,3}] ]] +
   #)&, {0.,0.}, 2000];
```

```
>> Graphics[Point[points],
   ImageSize->Small]
```



NestWhile

`NestWhile[`*f*`, `*expr*`, `*test*`]`
: applies a function *f* repeatedly on an expression *expr*, until applying *test* on the result no longer yields `True`.

`NestWhile[`*f*`, `*expr*`, `*test*`, `*m*`]`
: supplies the last *m* results to *test* (default value: 1).

`NestWhile[`*f*`, `*expr*`, `*test*`, All]`
: supplies all results gained so far to *test*.

Divide by 2 until the result is no longer an integer:

```
>> NestWhile[#/2&, 10000,
   IntegerQ]
```
$\frac{625}{2}$

Switch

`Switch[`*expr*`, `*pattern1*`, `*value1*`, `*pattern2*`, `*value2*`, ...]`
: yields the first *value* for which $expr matches the corresponding *pattern*.

```
>> Switch[2, 1, x, 2, y, 3, z]
```
y

```
>> Switch[5, 1, x, 2, y]
```
Switch $[5, 1, x, 2, y]$

```
>> Switch[5, 1, x, 2, y, _, z]
```
z

```
>> Switch[2, 1]
```
Switch called with 2 arguments. Switch must be called with an odd number of arguments.

Switch $[2, 1]$

Which

`Which[`*cond1*`, `*expr1*`, `*cond2*`, `*expr2*`, ...]`
: yields *expr1* if *cond1* evaluates to `True`, *expr2* if *cond2* evaluates to `True`, etc.

```
>> n = 5;
```

```
>> Which[n == 3, x, n == 5, y]
```
y

```
>> f[x_] := Which[x < 0, -x, x
   == 0, 0, x > 0, x]
```

```
>> f[-3]
```
3

If no test yields `True`, `Which` returns `Null`:

```
>> Which[False, a]
```

If a test does not evaluate to `True` or `False`, evaluation stops and a `Which` expression containing the remaining cases is returned:

```
>> Which[False, a, x, b, True, c
   ]
```
Which $[x, b, \text{True}, c]$

`Which` must be called with an even number of arguments:

```
>> Which[a, b, c]
```
Which called with 3 arguments.

Which $[a, b, c]$

While

`While[`*test*`, `*body*`]`
: evaluates *body* as long as *test* evaluates to `True`.

`While[`*test*`]`
: runs the loop without any body.

Compute the GCD of two numbers:

```
>> {a, b} = {27, 6};
```

```
>> While[b != 0, {a, b} = {b,
   Mod[a, b]}];
```

```
>> a
```
3

IX. Datentime

Contents

AbsoluteTime

`AbsoluteTime[]`
: gives the local time in seconds since epoch Jan 1 1900.

`AbsoluteTime[`*string*`]`
: gives the absolute time specification for a given date string.

`AbsoluteTime[{`*y*`, `*m*`, `*d*`, `*h*`, `*m*`, `*s*`}]`
: gives the absolute time specification for a given date list.

`AbsoluteTime[{‘‘string’,{’`*e1*`, `*e2*`, ...}}]`
: gives the absolute time specification for a given date list with specified elements *ei*.

```
>> AbsoluteTime[]
```
$3.66583357847 \times 10^9$

```
>> AbsoluteTime[{2000}]
```
3 155 673 600

```
>> AbsoluteTime[{"01/02/03", {"
   Day", "Month", "YearShort"}}]
```
3 253 046 400

```
>> AbsoluteTime["6 June 1991"]
```
2 885 155 200

```
>> AbsoluteTime[{"6-6-91", {"Day
   ", "Month", "YearShort"}}]
```
2 885 155 200

AbsoluteTiming

`AbsoluteTiming[`*expr*`]`
: measures the actual time it takes to evaluate *expr*. It returns a list containing the measured time in seconds and the result of the evaluation.

```
>> AbsoluteTiming[50!]
```
{0.000305891036987, 30 414˜
˜093 201 713 378 043 612 608˜
˜166 064 768 844 377 641 568˜
˜960 512 000 000 000 000}

```
>> Attributes[AbsoluteTiming]
```
{HoldAll, Protected}

DateDifference

'DateDifference[*date1*, *date2*]
: difference between dates in days.

'DateDifference[*date1*, *date2*, *unit*]
: difference between dates in specified *unit*.

'DateDifference[*date1*, *date2*, {*unit1*, *unit2*, ...}]
: difference between dates as a list in the specified units.

```
>> DateDifference[{2042, 1, 4}, {2057, 1, 1}]
```

5 476

```
>> DateDifference[{1936, 8, 14}, {2000, 12, 1}, "Year"]
```

{64.3424657534, Year}

```
>> DateDifference[{2010, 6, 1}, {2015, 1, 1}, "Hour"]
```

{40 200, Hour}

```
>> DateDifference[{2003, 8, 11}, {2003, 10, 19}, {"Week", "
Day"}]
```

{{9, Week} , {6, Day}}

DateList

`DateList[]`
: returns the current local time in the form {*year*, *month*, *day*, *hour*, *minute*, *second*}.

`DateList[`*time*`]`
: returns a formatted date for the number of seconds *time* since epoch Jan 1 1900.

`DateList[{`*y*`, `*m*`, `*d*`, `*h*`, `*m*`, `*s*`}]`
: converts an incomplete date list to the standard representation.

`DateString[`*string*`]`
: returns the formatted date list of a date string specification.

`DateString[`*string*`, {`*e1*`, `*e2*`, ...}]`
: returns the formatted date list of a *string* obtained from elements *ei*.

```
>> DateList[0]
```

{1 900, 1, 1, 0, 0, 0.}

```
>> DateList[3155673600]
```

{2 000, 1, 1, 0, 0, 0.}

```
>> DateList[{2003, 5, 0.5, 0.1, 0.767}]
```

{2 003, 4, 30, 12, 6, 46.02}

```
>> DateList[{2012, 1, 300., 10}]
```

{2 012, 10, 26, 10, 0, 0.}

```
>> DateList["31/10/1991"]
```

{1 991, 10, 31, 0, 0, 0.}

```
>> DateList[{"31/10/91", {"Day", "Month", "YearShort"}}]
```

{1 991, 10, 31, 0, 0, 0.}

```
>> DateList[{"31 10/91", {"Day", " ", "Month", "/", "
YearShort"}}]
```

{1 991, 10, 31, 0, 0, 0.}

If not specified, the current year assumed

```
>> DateList[{"5/18", {"Month", "
   Day"}}]
```

$\{2016, 5, 18, 0, 0, 0.\}$

DatePlus

`DatePlus[date, n]`
: finds the date *n* days after *date*.

`DatePlus[date, {n, ‘‘unit’}]’`
: finds the date *n* units after *date*.

`DatePlus[date, {{n1, ‘‘unit1’}, {n2, unit2}, ...}]’`
: finds the date which is *n_i* specified units after *date*.

`DatePlus[n]`
: finds the date *n* days after the current date.

`DatePlus[offset]`
: finds the date which is offset from the current date.

Add 73 days to Feb 5, 2010:

```
>> DatePlus[{2010, 2, 5}, 73]
```

$\{2010, 4, 19\}$

Add 8 weeks and 1 day to March 16, 1999:

```
>> DatePlus[{2010, 2, 5}, {{8, "
   Week"}, {1, "Day"}}]
```

$\{2010, 4, 3\}$

DateString

`DateString[]`
: returns the current local time and date as a string.

`DateString[`*elem*`]`
: returns the time formatted according to *elems*.

`DateString[{`*e1*`, `*e2*`, ...}]`
: concatinates the time formatted according to elements *ei*.

`DateString[`*time*`]`
: returns the date string of an AbsoluteTime.

`DateString[{`*y*`, `*m*`, `*d*`, `*h*`, `*m*`, `*s*`}]`
: returns the date string of a date list specification.

`DateString[`*string*`]`
: returns the formatted date string of a date string specification.

`DateString[`*spec*`, `*elems*`]`
: formats the time in turns of *elems*. Both *spec* and *elems* can take any of the above formats.

The current date and time:

```
>> DateString[];
```

```
>> DateString[{1991, 10, 31, 0,
   0}, {"Day", " ", "MonthName",
    " ", "Year"}]
```

31 October 1 991

```
>> DateString[{2007, 4, 15, 0}]
```

Sun 15 Apr 2 007 00:00:00

```
>> DateString[{1979, 3, 14}, {"
   DayName", " ", "Month", "-",
   "YearShort"}]
```

Wednesday 03-79

Non-integer values are accepted too:

```
>> DateString[{1991, 6, 6.5}]
```

Thu 6 Jun 1 991 12:00:00

$DateStringFormat

`$DateStringFormat`
: gives the format used for dates generated by DateString.

```
>> $DateStringFormat
```
{DateTimeShort}

Pause

`Pause[n]`
: pauses for *n* seconds.

```
>> Pause[0.5]
```

SessionTime

`SessionTime[]`
: returns the total time since this session started.

```
>> SessionTime[]
```
478.082336187

TimeUsed

`TimeUsed[]`
: returns the total cpu time used for this session.

```
>> TimeUsed[]
```
475.533217531

$TimeZone

`$TimeZone`
: gives the current time zone.

```
>> $TimeZone
```
1.

Timing

`Timing[expr]`
: measures the processor time taken to evaluate *expr*. It returns a list containing the measured time in seconds and the result of the evaluation.

```
>> Timing[50!]
```
{0.00027825899997, 30 414~
~093 201 713 378 043 612 608~
~166 064 768 844 377 641 568~
~960 512 000 000 000 000}

```
>> Attributes[Timing]
```
{HoldAll, Protected}

X. Diffeqns

Contents

C

`C[`*n*`]`
: represents the *n*th constant in a solution to a differential equation.

DSolve

`DSolve[`*eq*`, `*y*`[`*x*`], `*x*`]`
: solves a differential equation for the function *y*[*x*].

```
>> DSolve[y''[x] == 0, y[x], x]
```

$\{\{y[x]\text{->}xC[2]+C[1]\}\}$

```
>> DSolve[y''[x] == y[x], y[x],
   x]
```

$\{\{y[x]\text{->}C[1]E^{-x}+C[2]E^{x}\}\}$

```
>> DSolve[y''[x] == y[x], y, x]
```

$\{\{y\text{->}\left(\text{Function}\left[\{x\},C[1]\,\text{Exp}[-x]+C[2]\,\text{Exp}[x]\right]\right)\}\}$

XI. Evaluation

Contents

Evaluate

>> `SetAttributes[f, HoldAll]`

>> `f[1 + 2]`
$f[1+2]$

>> `f[Evaluate[1 + 2]]`
$f[3]$

>> `Hold[Evaluate[1 + 2]]`
Hold [3]

>> `HoldComplete[Evaluate[1 + 2]]`
HoldComplete [Evaluate [1 + 2]]

>> `Evaluate[Sequence[1, 2]]`
Sequence [1, 2]

$HistoryLength

>> `$HistoryLength`
100

>> `$HistoryLength = 1;`

>> `42`
42

>> `%`
42

>> `%%`
%3

>> `$HistoryLength = 0;`

>> `42`
42

>> `%`
%7

Hold

>> `Attributes[Hold]`
{HoldAll, Protected}

HoldComplete

>> `Attributes[HoldComplete]`
{HoldAllComplete, Protected}

HoldForm

`HoldForm[`*expr*`]` maintains *expr* in an unevaluated form, but prints as *expr*.

>> `HoldForm[1 + 2 + 3]`
$1+2+3$

`HoldForm` has attribute `HoldAll`:

```
>> Attributes[HoldForm]
```
{HoldAll, Protected}

In

```
>> x = 1
```
1

```
>> x = x + 1
```
2

```
>> Do[In[2], {3}]
```

```
>> x
```
5

```
>> In[-1]
```
5

```
>> Definition[In]
```
$$\begin{aligned} &\text{Attributes}[\text{In}] = \{\text{Protected}\} \\ &\text{In}[6] = \text{Definition}[\text{In}] \\ &\text{In}[5] = \text{In}[-1] \\ &\text{In}[4] = x \\ &\text{In}[3] = \text{Do}\left[\text{In}[2], \{3\}\right] \\ &\text{In}[2] = x = x + 1 \\ &\text{In}[1] = x = 1 \end{aligned}$$

$Line

```
>> $Line
```
1

```
>> $Line
```
2

```
>> $Line = 12;
```

```
>> 2 * 5
```
10

```
>> Out[13]
```
10

```
>> $Line = -1;
```
Non-negative integer expected.

Out

`Out[k]` or `%k`
: gives the result of the kth input line.

`%`, `%%`, etc.
: gives the result of the previous input line, of the line before the previous input line, etc.

```
>> 42
```
42

```
>> %
```
42

```
>> 43;
```

```
>> %
```

```
>> 44
```
44

```
>> %1
```
42

```
>> %%
```
44

```
>> Hold[Out[-1]]
```
Hold[%]

```
>> Hold[%4]
```
Hold[%4]

```
>> Out[0]
```
Out[0]

$RecursionLimit

```
>> a = a + a
```
Recursion depth of 200 exceeded.

$Aborted

```
>> $RecursionLimit
```
200

```
>> $RecursionLimit = x;
```
Cannot set $RecursionLimit to x; value must be an integer between 20 and 512.

```
>> $RecursionLimit = 512
```
512

```
>> a = a + a
```
Recursion depth of 512 exceeded.

$Aborted

ReleaseHold

`ReleaseHold[`*expr*`]`
: removes any `Hold`, `HoldForm`, `HoldPattern` or `HoldComplete` head from *expr*.

```
>> x = 3;
```

```
>> Hold[x]
```
Hold [x]

```
>> ReleaseHold[Hold[x]]
```
3

```
>> ReleaseHold[y]
```
y

Sequence

`Sequence[`*x1*`, `*x2*`, ...]`
: represents a sequence of arguments to a function.

`Sequence` is automatically spliced in, except when a function has attribute `SequenceHold` (like assignment functions).

```
>> f[x, Sequence[a, b], y]
```
$f\left[x,a,b,y\right]$

```
>> Attributes[Set]
```
{HoldFirst, Protected, SequenceHold}

```
>> a = Sequence[b, c];
```

```
>> a
```
Sequence [b, c]

Apply `Sequence` to a list to splice in arguments:

```
>> list = {1, 2, 3};
```

```
>> f[Sequence @@ list]
```
f [1, 2, 3]

Inside `Hold` or a function with a held argument, `Sequence` is spliced in at the first level of the argument:

```
>> Hold[a, Sequence[b, c], d]
```
Hold [a, b, c, d]

If `Sequence` appears at a deeper level, it is left unevaluated:

```
>> Hold[{a, Sequence[b, c], d}]
```
Hold [{a, Sequence [b, c], d}]

Unevaluated

```
>> Length[Unevaluated[1+2+3+4]]
```
4

`Unevaluated` has attribute `HoldAllComplete`:

```
>> Attributes[Unevaluated]
```
{HoldAllComplete, Protected}

`Unevaluated` is maintained for arguments to non-executed functions:

```
>> f[Unevaluated[x]]
```
f [Unevaluated [x]]

Likewise, its kept in flattened arguments and sequences:

```
>> Attributes[f] = {Flat};
```

```
>> f[a, Unevaluated[f[b, c]]]
```
f [a, Unevaluated [b], Unevaluated [c]]

```
>> g[a, Sequence[Unevaluated[b],
    Unevaluated[c]]]
```

$g[a, \text{Unevaluated}[b], \text{Unevaluated}[c]]$

However, unevaluated sequences are kept:

```
>> g[Unevaluated[Sequence[a, b,
   c]]]
```

$g\left[\text{Unevaluated}\left[\text{Sequence}[a, b, c]\right]\right]$

XII. Exptrig

Contents

ArcCos

`ArcCos[z]`
returns the inverse cosine of *z*.

>> `ArcCos[1]`
0

>> `ArcCos[0]`
$\frac{\text{Pi}}{2}$

>> `Integrate[ArcCos[x], {x, -1, 1}]`
Pi

ArcCosh

`ArcCosh[z]`
returns the inverse hyperbolic cosine of *z*.

>> `ArcCosh[0]`
$\frac{I}{2}\text{Pi}$

>> `ArcCosh[0.]`
0. + 1.57079632679489662*I*

>> `ArcCosh[0.00000000000000000000000000000000000000]`

0. + 1.570796326794896˜
˜61914798426245454265 88*I*

ArcCot

`ArcCot[z]`
returns the inverse cotangent of *z*.

>> `ArcCot[0]`
$\frac{\text{Pi}}{2}$

>> `ArcCot[1]`

$\frac{\text{Pi}}{4}$

ArcCoth

`ArcCoth[z]`
: returns the inverse hyperbolic cotangent of z.

>> `ArcCoth[0]`

$\frac{I}{2}\text{Pi}$

>> `ArcCoth[1]`

∞

>> `ArcCoth[0.0]`

$0. + 1.57079632679489662I$

>> `ArcCoth[0.5]`

$0.549306144334054846 - 1.57079632679489662I$

ArcCsc

`ArcCsc[z]`
: returns the inverse cosecant of z.

>> `ArcCsc[1]`

$\frac{\text{Pi}}{2}$

>> `ArcCsc[-1]`

$-\frac{\text{Pi}}{2}$

ArcCsch

`ArcCsch[z]`
: returns the inverse hyperbolic cosecant of z.

>> `ArcCsch[0]`

ComplexInfinity

>> `ArcCsch[1.0]`

0.881373587019543025

ArcSec

`ArcSec[z]`
: returns the inverse secant of z.

>> `ArcSec[1]`

0

>> `ArcSec[-1]`

Pi

ArcSech

`ArcSech[z]`
: returns the inverse hyperbolic secant of z.

>> `ArcSech[0]`

∞

>> `ArcSech[1]`

0

>> `ArcSech[0.5]`

1.31695789692481671

ArcSin

`ArcSin[z]`
: returns the inverse sine of z.

>> `ArcSin[0]`

0

>> `ArcSin[1]`

$\frac{\text{Pi}}{2}$

ArcSinh

`ArcSinh[z]`
: returns the inverse hyperbolic sine of z.

```
>> ArcSinh[0]
```
0

```
>> ArcSinh[0.]
```
0.

```
>> ArcSinh[1.0]
```
0.881373587019543025

ArcTan

`ArcTan[z]`
: returns the inverse tangent of z.

```
>> ArcTan[1]
```
$\frac{\text{Pi}}{4}$

```
>> ArcTan[1.0]
```
0.78539816339744831

```
>> ArcTan[-1.0]
```
-0.78539816339744831

```
>> ArcTan[1, 1]
```
$\frac{\text{Pi}}{4}$

ArcTanh

`ArcTanh[z]`
: returns the inverse hyperbolic tangent of z.

```
>> ArcTanh[0]
```
0

```
>> ArcTanh[1]
```
∞

```
>> ArcTanh[0]
```
0

```
>> ArcTanh[.5 + 2 I]
```
$0.0964156202029961672 + 1.12655644083482235I$

```
>> ArcTanh[2 + I]
```
$\text{ArcTanh}[2 + I]$

Cos

`Cos[z]`
: returns the cosine of z.

```
>> Cos[3 Pi]
```
-1

Cosh

`Cosh[z]`
: returns the hyperbolic cosine of z.

```
>> Cosh[0]
```
1

Cot

`Cot[z]`
: returns the cotangent of z.

```
>> Cot[0]
```
ComplexInfinity

```
>> Cot[1.]
```
0.642092615934330703

Coth

`Coth[z]`
: returns the hyperbolic cotangent of z.

```
>> Coth[0]
```
ComplexInfinity

Csc

`Csc[z]`
: returns the cosecant of z.

```
>> Csc[0]
```
ComplexInfinity

```
>> Csc[1] (* Csc[1] in
Mathematica *)
```
$$\frac{1}{\mathrm{Sin}[1]}$$

```
>> Csc[1.]
```
1.18839510577812122

Csch

`Csch[z]`
: returns the hyperbolic cosecant of z.

```
>> Csch[0]
```
ComplexInfinity

E

`E`
: is the constant e.

```
>> N[E]
```
2.71828182845904524

```
>> N[E, 50]
```
2.7182818284590452353602874713526624977572470937

```
>> Attributes[E]
```
{Constant, Protected, ReadProtected}

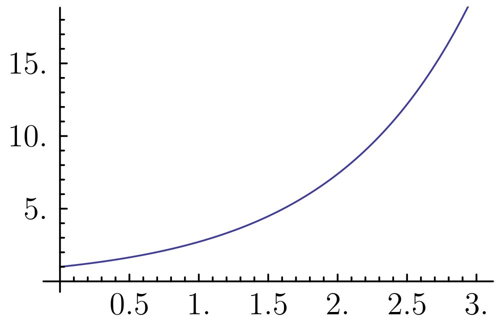
Exp

`Exp[z]`
: returns the exponential function of z.

```
>> Exp[1]
```
E

```
>> Exp[10.0]
```
22 026.4657948067169

```
>> Exp[x] //FullForm
```
Power[E, x]

```
>> Plot[Exp[x], {x, 0, 3}]
```


GoldenRatio

`GoldenRatio`
: is the golden ratio.

```
>> N[GoldenRatio]
```
1.61803398874989485

Haversine

`Haversine[z]`
: returns the haversine function of z.

```
>> Haversine[1.5]
```
0.464631399166148545

```
>> Haversine[0.5 + 2I]
```
$-1.15081866645704728 + 0.869404752237158167I$

InverseHaversine

`Haversine[z]`
: returns the inverse haversine function of z.

```
>> InverseHaversine[0.5]
```
1.57079632679489662

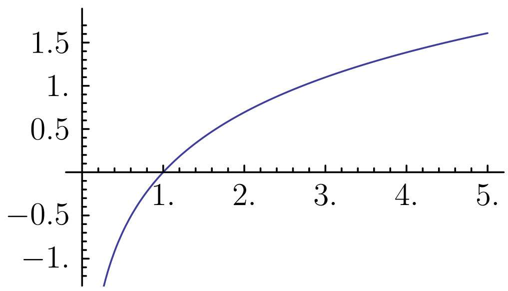
```
>> InverseHaversine[1 + 2.5 I]
```
1.76458946334982881 +
2.33097465304931242I

Log

`Log[z]`
: returns the natural logarithm of z.

```
>> Log[{0, 1, E, E * E, E ^ 3, E ^ x}]
```
$\{-\infty, 0, 1, 2, 3, \mathrm{Log}\left[E^x\right]\}$

```
>> Log[0.]
```
Indeterminate

```
>> Plot[Log[x], {x, 0, 5}]
```


Log10

`Log10[z]`
: returns the base-10 logarithm of z.

```
>> Log10[1000]
```
3

```
>> Log10[{2., 5.}]
```
{0.301029995663981195,
0.698970004336018805}

```
>> Log10[E ^ 3]
```
$\frac{3}{\mathrm{Log}[10]}$

Log2

`Log2[z]`
: returns the base-2 logarithm of z.

```
>> Log2[4 ^ 8]
```
16

```
>> Log2[5.6]
```
2.48542682717024176

```
>> Log2[E ^ 2]
```
$\frac{2}{\mathrm{Log}[2]}$

Pi

`Pi`
: is the constant π.

```
>> N[Pi]
```
3.14159265358979324

```
>> N[Pi, 50]
```
3.1415926535897932384626433˜
˜83279502884197169399375I

```
>> Attributes[Pi]
```
{Constant, Protected,
ReadProtected}

Sec

`Sec[z]`
: returns the secant of z.

```
>> Sec[0]
```
1

```
>> Sec[1] (* Sec[1] in
   Mathematica *)
```
$\frac{1}{\mathrm{Cos}[1]}$

```
>> Sec[1.]
```
1.85081571768092562

Sech

`Sech[`*z*`]`
returns the hyperbolic secant of *z*.

```
>> Sech[0]
```
1

Sin

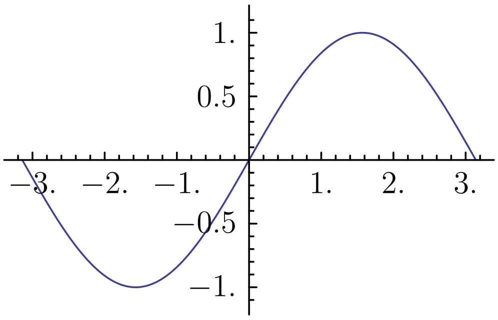
`Sin[`*z*`]`
returns the sine of *z*.

```
>> Sin[0]
```
0

```
>> Sin[0.5]
```
0.479425538604203

```
>> Sin[3 Pi]
```
0

```
>> Sin[1.0 + I]
```
1.29845758141597729 +
0.634963914784736108*I*

```
>> Plot[Sin[x], {x, -Pi, Pi}]
```


Sinh

`Sinh[`*z*`]`
returns the hyperbolic sine of *z*.

```
>> Sinh[0]
```
0

Tan

`Tan[`*z*`]`
returns the tangent of *z*.

```
>> Tan[0]
```
0

```
>> Tan[Pi / 2]
```
ComplexInfinity

Tanh

`Tanh[`*z*`]`
returns the hyperbolic tangent of *z*.

```
>> Tanh[0]
```
0

XIII. Functional

Contents

Composition

`Composition[`f`, `g`]`
: returns the composition of two functions f and g.

```
>> Composition[f, g][x]
```
$f[g[x]]$

```
>> Composition[f, g, h][x, y, z]
```
$f[g[h[x,y,z]]]$

```
>> Composition[]
```
Identity

```
>> Composition[][x]
```
x

```
>> Attributes[Composition]
```
{Flat, OneIdentity, Protected}

```
>> Composition[f, Composition[g, h]]
```
Composition $[f,g,h]$

Function (&)

`Function[`*body*`]` or *body* `&`
: represents a pure function with parameters `#1`, `#2`, etc.

`Function[{`*x1*`, `*x2*`, ...}, `*body*`]`
: represents a pure function with parameters *x1*, *x2*, etc.

```
>> f := # ^ 2 &
```

```
>> f[3]
```
9

```
>> #^3& /@ {1, 2, 3}
```
{1,8,27}

```
>> #1+#2&[4, 5]
```
9

You can use `Function` with named parameters:

```
>> Function[{x, y}, x * y][2, 3]
```
6

Parameters are renamed, when necessary, to avoid confusion:

```
>> Function[{x}, Function[{y}, f[x, y]]][y]
```
Function $[\{y\$\}, f[y, y\$]]$

```
>> Function[{y}, f[x, y]] /. x->y
```
Function $[\{y\}, f[y,y]]$

```
>> Function[y, Function[x, y^x
   ]][x][y]
```
x^y

```
>> Function[x, Function[y, x^y
   ]][x][y]
```
x^y

Slots in inner functions are not affected by outer function application:

```
>> g[#] & [h[#]] & [5]
```
$g[h[5]]$

Identity

```
>> Identity[x]
```
x

```
>> Identity[x, y]
```
Identity $[x,y]$

Slot

`#n`
: represents the nth argument to a pure function.

`#`
: is short-hand for `#1`

`#0`
: represents the pure function itself.

```
>> #
```
#1

Unused arguments are simply ignored:

```
>> {#1, #2, #3}&[1, 2, 3, 4, 5]
```
$\{1,2,3\}$

Recursive pure functions can be written using `#0`:

```
>> If[#1<=1, 1, #1 #0[#1-1]]&
   [10]
```
3 628 800

SlotSequence

`##`
: is the sequence of arguments supplied to a pure function.

`##n`
: starts with the nth argument.

```
>> Plus[##]& [1, 2, 3]
```
6

```
>> Plus[##2]& [1, 2, 3]
```
5

```
>> FullForm[##]
```
SlotSequence [1]

XIV. Graphics

Contents

AbsoluteThickness

Automatic

`Automatic`
is used to specify an automatically computed option value.

`Automatic` is the default for `PlotRange`, `ImageSize`, and other graphical options:

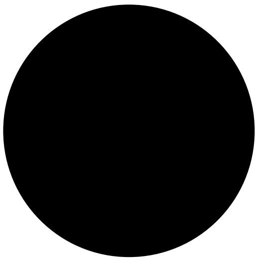
```
>> Cases[Options[Plot], _ :> Automatic]
```

{Exclusions:>Automatic, ImageSize:>Automatic, MaxRecursion:>Automatic, PlotRange:>Automatic, PlotRangePadding:>Automatic}

Black

`Black`
represents the color black in graphics.

```
>> Graphics[{Black, Disk[]}, ImageSize->Small]
```



```
>> Black
```
GrayLevel [0]

Blend

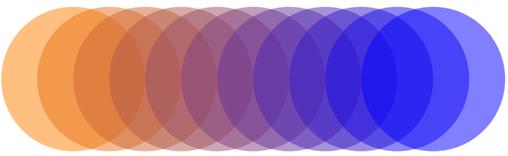
```
>> Blend[{Red, Blue}]
```
RGBColor [0.5, 0., 0.5, 1.]

```
>> Blend[{Red, Blue}, 0.3]
```
RGBColor [0.7, 0., 0.3, 1.]

```
>> Blend[{Red, Blue, Green}, 0.75]
```
RGBColor [0., 0.5, 0.5, 1.]

```
>> Graphics[Table[{Blend[{Red, Green, Blue}, x], Rectangle[{10 x, 0}]}, {x, 0, 1, 1/10}]]
```

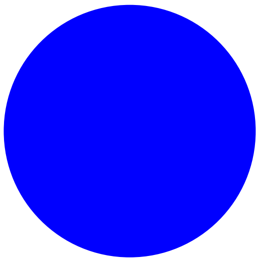
```
>> Graphics[Table[{Blend[{RGBColor[1, 0.5, 0, 0.5], RGBColor[0, 0, 1, 0.5]}, x], Disk[{5x, 0}]}, {x, 0, 1, 1/10}]]
```



Blue

`Blue`
: represents the color blue in graphics.

```
>> Graphics[{Blue, Disk[]}, ImageSize->Small]
```



```
>> Blue
```
RGBColor [0, 0, 1]

CMYKColor

Circle

`Circle[{`*cx*`, `*cy*`}, `*r*`]`
: draws a circle with center `(`*cx*`, `*cy*`)` and radius *r*.

`Circle[{`*cx*`, `*cy*`}, {`*rx*`, `*ry*`}]`
: draws an ellipse.

`Circle[{`*cx*`, `*cy*`}]`
: chooses radius 1.

`Circle[]`
: chooses center `(0, 0)` and radius 1.

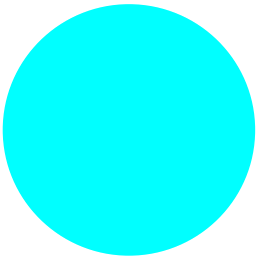
```
>> Graphics[{Red, Circle[{0, 0}, {2, 1}]}]
```

CircleBox

Cyan

`Cyan`
    represents the color cyan in graphics.

```
>> Graphics[{Cyan, Disk[]}, ImageSize->Small]
```



```
>> Cyan
```

RGBColor[0, 1, 1]

Darker

`Darker[c, f]`
    is equivalent to `Blend[{c, Black}, f]`.
`Darker[c]`
    is equivalent to `Darker[c, 1/3]`.

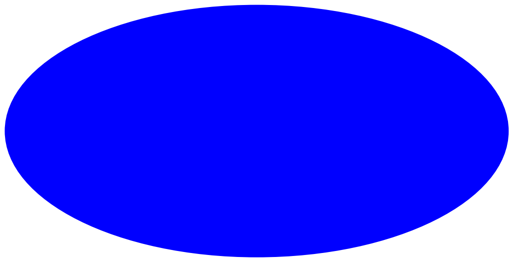
```
>> Graphics[Table[{Darker[Yellow, x], Disk[{12x, 0}]}, {x, 0, 1, 1/6}]]
```

Directive

Disk

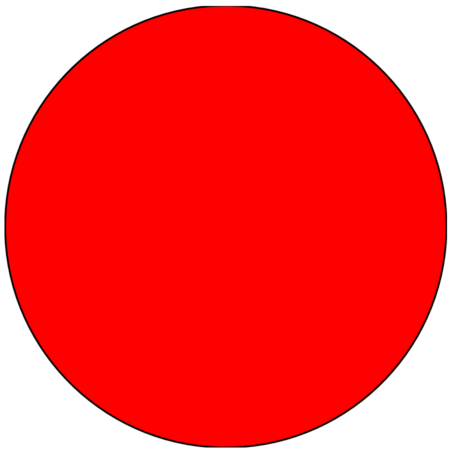
`Disk[{cx, cy}, r]`
    fills a circle with center `(cx, cy)` and radius r.
`Disk[{cx, cy}, {rx, ry}]`
    fills an ellipse.
`Disk[{cx, cy}]`
    chooses radius 1.
`Disk[]`
    chooses center `(0, 0)` and radius 1.

```
>> Graphics[{Blue, Disk[{0, 0}, {2, 1}]}]
```



The outer border can be drawn using `EdgeForm`:

```
>> Graphics[{EdgeForm[Black], Red, Disk[]}]
```



DiskBox

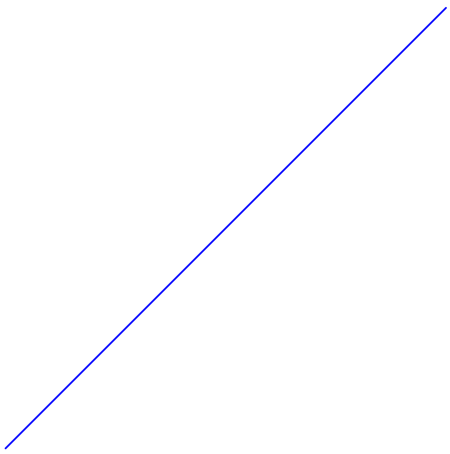
EdgeForm

FaceForm

Graphics

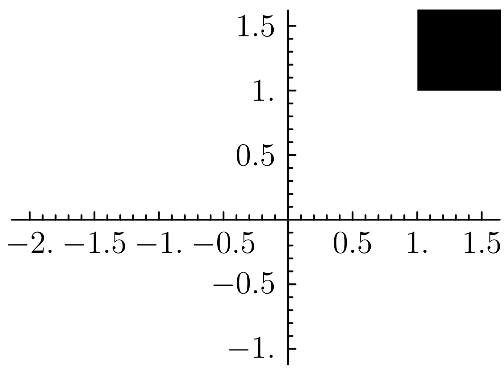
`Graphics[`*primitives*`, `*options*`]`
    represents a graphic.

```
>> Graphics[{Blue, Line[{{0,0},
   {1,1}}]}]
```



Graphics supports `PlotRange`:

```
>> Graphics[{Rectangle[{1, 1}]},
    Axes -> True, PlotRange ->
   {{-2, 1.5}, {-1, 1.5}}]
```



Graphics produces `GraphicsBox` boxes:

```
>> Graphics[Rectangle[]] //
   ToBoxes // Head
```

GraphicsBox

In `TeXForm`, `Graphics` produces Asymptote figures:

```
>> Graphics[Circle[]] // TeXForm
```

\begin{asy}
size(5.8556cm, 5.8333cm);
draw(ellipse((175,175),175,175),
rgb(0, 0, 0)+linewidth(0.66667));
clip(box((-0.33333,0.33333),
(350.33,349.67)));
\end{asy}

Invalid graphics directives yield invalid box structures:

```
>> Graphics[Circle[{a, b}]]
```

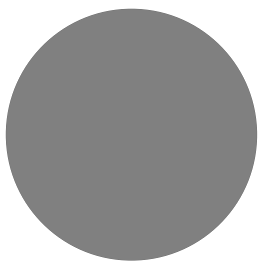
GraphicsBox [CircleBox [List [a, b]] , Rule [AspectRatio, Automatic] , Rule [Axes, False] , Rule [AxesStyle, List []] , Rule [ImageSize, Automatic] , Rule [LabelStyle, List []] , Rule [PlotRange, Automatic] , Rule [PlotRangePadding, Automatic] , Rule [TicksStyle, List []]] is not a valid box structure.

GraphicsBox

Gray

`Gray`
    represents the color gray in graphics.

```
>> Graphics[{Gray, Disk[]},
   ImageSize->Small]
```



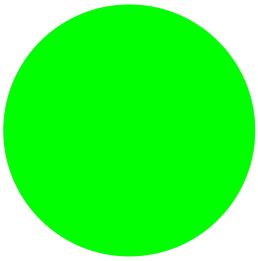
>> `Gray`
GrayLevel [0.5]

GrayLevel

Green

`Green`
represents the color green in graphics.

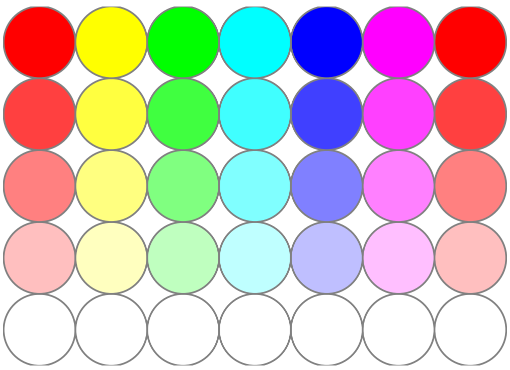
>> `Graphics[{Green, Disk[]}, ImageSize->Small]`



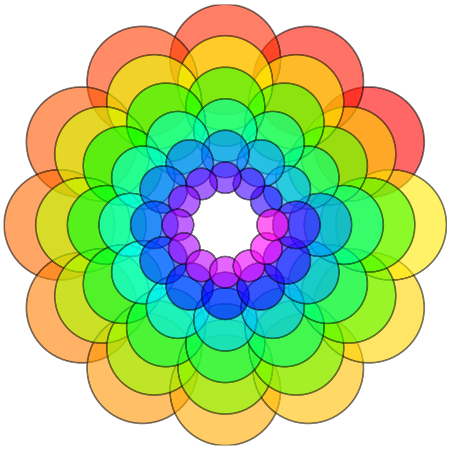
>> `Green`
RGBColor [0, 1, 0]

Hue

>> `Graphics[Table[{EdgeForm[Gray], Hue[h, s], Disk[{12h, 8s}]}, {h, 0, 1, 1/6}, {s, 0, 1, 1/4}]]`



>> `Graphics[Table[{EdgeForm[{GrayLevel[0, 0.5]}], Hue[(-11+q+10r)/72, 1, 1, 0.6], Disk[(8-r){Cos[2Pi q/12], Sin[2Pi q/12]}, (8-r)/3]}, {r, 6}, {q, 12}]]`



Inset

InsetBox

Large

`ImageSize -> Large`
produces a large image.

LightRed

`LightRed`
represents the color light red in graphics.

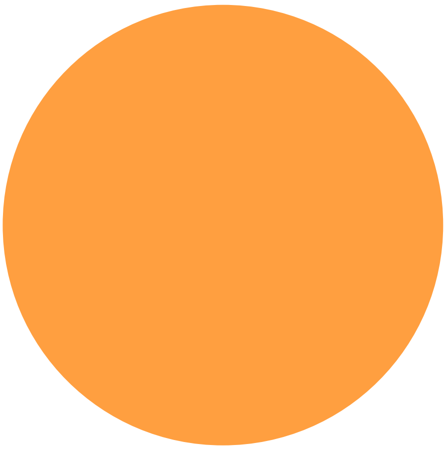
>> `Graphics[{LightRed, Disk[]}, ImageSize->Small]`

Lighter

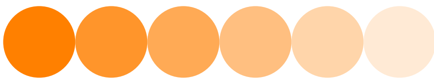
`Lighter[`*c*`, `*f*`]`
  is equivalent to `Blend[{`*c*`, White}, `*f*`]`.
`Lighter[`*c*`]`
  is equivalent to `Lighter[`*c*`, 1/3]`.

>> `Lighter[Orange, 1/4]`
RGBColor [1., 0.625, 0.25, 1.]

>> `Graphics[{Lighter[Orange, 1/4], Disk[]}]`



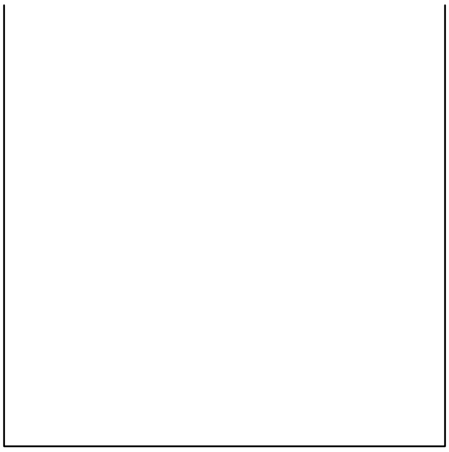
>> `Graphics[Table[{Lighter[Orange, x], Disk[{12x, 0}]}, {x, 0, 1, 1/6}]]`



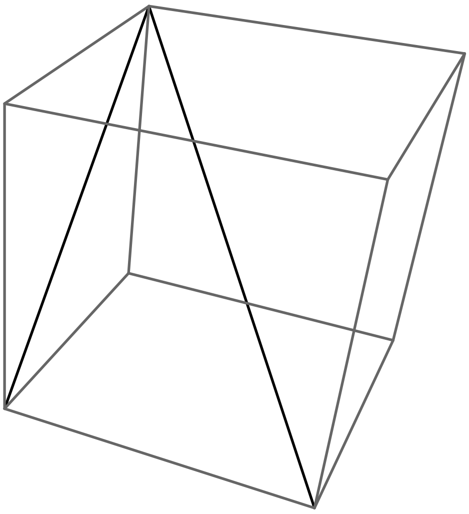
Line

`Line[{`*point_1*`, `*point_2* `...}]`
  represents the line primitive.
`Line[{{`*p_11*`, `*p_12*`, ...}, {`*p_21*`, `*p_22*`, ...}, ...}]`
  represents a number of line primitives.

>> `Graphics[Line[{{0,1},{0,0},{1,0},{1,1}}]]`



>> `Graphics3D[Line[{{0,0,0},{0,1,1},{1,0,0}}]]`

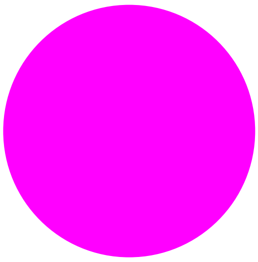


LineBox

Magenta

`Magenta`
: represents the color magenta in graphics.

>> `Graphics[{Magenta, Disk[]}, ImageSize->Small]`



>> `Magenta`

RGBColor [1, 0, 1]

Medium

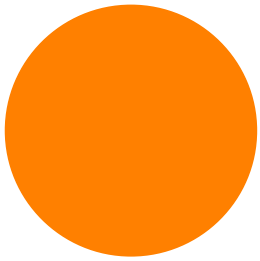
`ImageSize -> Medium`
: produces a medium-sized image.

Offset

Orange

`Orange`
: represents the color orange in graphics.

>> `Graphics[{Orange, Disk[]}, ImageSize->Small]`



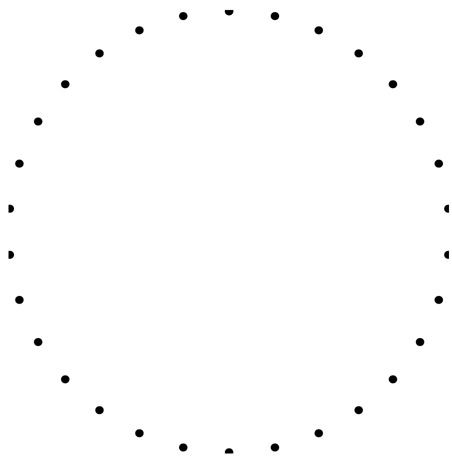
Point

`Line[{`*point_1*`,` *point_2* `...}]`
: represents the point primitive.

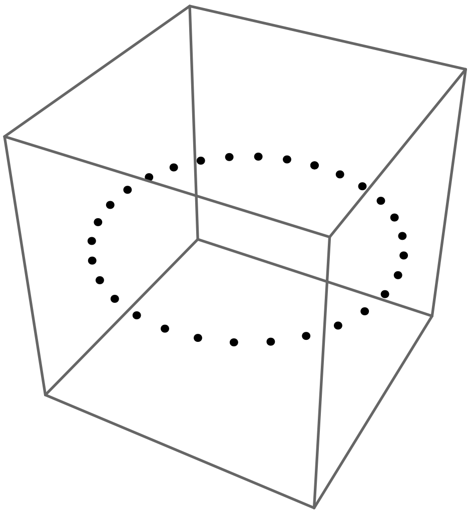
`Line[{{`*p_11*`,` *p_12*`, ...}, {`*p_21*`,` *p_22*`, ...}, ...}]`
: represents a number of point primitives.

>> `Graphics[Point[{0,0}]]`

>> `Graphics[Point[Table[{Sin[t], Cos[t]}, {t, 0, 2. Pi, Pi / 15.}]]]`



```
>> Graphics3D[Point[Table[{Sin[t
   ], Cos[t], 0}, {t, 0, 2. Pi,
   Pi / 15.}]]]
```



PointBox

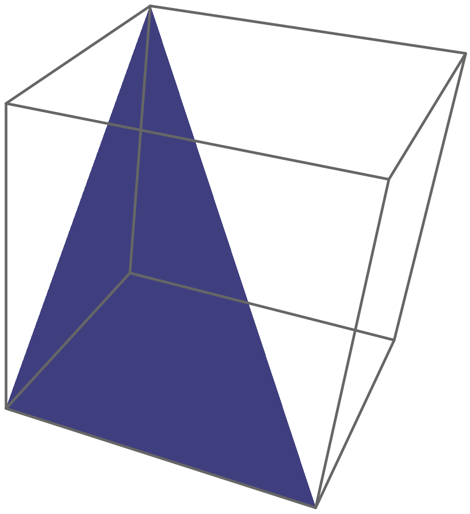
Polygon

`Polygon[{`*point_1*`, `*point_2*` ...}]`
: represents the filled polygon primitive.

`Polygon[{{`*p_11*`, `*p_12*`, ...}, {`*p_21*`, `*p_22*`, ...}, ...}]`
: represents a number of filled polygon primitives.

```
>> Graphics[Polygon
   [{{1,0},{0,0},{0,1}}]]
```

```
>> Graphics3D[Polygon
   [{{0,0,0},{0,1,1},{1,0,0}}]]
```

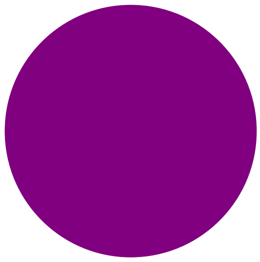


PolygonBox

Purple

`Purple`
: represents the color purple in graphics.

```
>> Graphics[{Purple, Disk[]},
   ImageSize->Small]
```



RGBColor

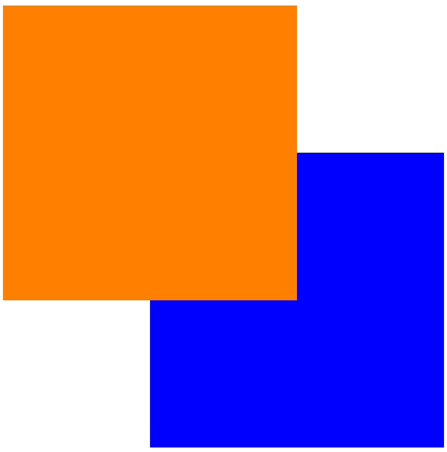
Rectangle

`Rectangle[{`*xmin*`, `*ymin*`}]`
: represents a unit square with bottom-left corner at {*xmin*, *ymin*}.

'Rectangle[{*xmin*, *ymin*}, {*xmax*, *ymax*}]
: is a rectange extending from {*xmin*, *ymin*} to {*xmax*, *ymax*}.

```
>> Graphics[Rectangle[]]
```

```
>> Graphics[{Blue, Rectangle
   [{0.5, 0}], Orange, Rectangle
   [{0, 0.5}]}]
```

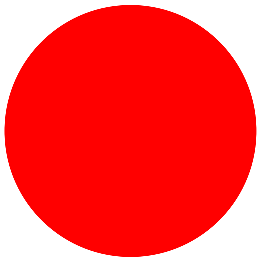


RectangleBox

Red

`Red`
: represents the color red in graphics.

```
>> Graphics[{Red, Disk[]},
   ImageSize->Small]
```



```
>> Red
```

RGBColor [1, 0, 0]

Small

`ImageSize -> Small`
: produces a small image.

Text

Thick

Thickness

Thin

Tiny

`ImageSize -> Tiny`
: produces a tiny image.

White

`White`
: represents the color white in graphics.

>> `Graphics[{White, Disk[]}, ImageSize->Small]`

>> `White`

GrayLevel [1]

Yellow

`Yellow`
: represents the color yellow in graphics.

>> `Graphics[{Yellow, Disk[]}, ImageSize->Small]`

>> `Yellow`

RGBColor [1, 1, 0]

XV. Graphics3d

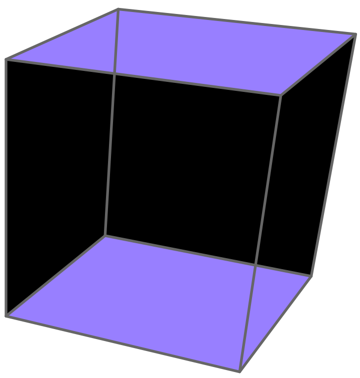
Contents

Cuboid

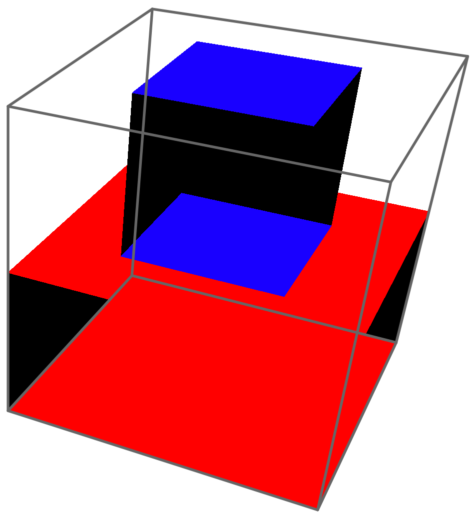
`Cuboid[{`*xmin*`, `*ymin*`, `*zmin*`}]`
is a unit cube.

`Cuboid[{`*xmin*`, `*ymin*`, `*zmin*`}, {`*xmax*`, `*ymax*`, `*zmax*`}]`
represents a cuboid extending from {*xmin, ymin, zmin*} to {*xmax, ymax, zmax*}.

```
>> Graphics3D[Cuboid[{0, 0, 1}]]
```



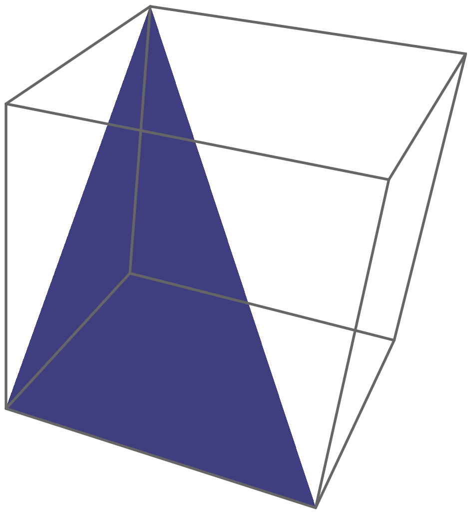
```
>> Graphics3D[{Red, Cuboid[{0,
   0, 0}, {1, 1, 0.5}], Blue,
   Cuboid[{0.25, 0.25, 0.5},
   {0.75, 0.75, 1}]}]
```



Graphics3D

`Graphics3D[`*primitives*`, `*options*`]`
represents a three-dimensional graphic.

```
>> Graphics3D[Polygon[{{0,0,0},
   {0,1,1}, {1,0,0}}]]
```



In `TeXForm`, `Graphics3D` creates Asymptote figures:

```
>> Graphics3D[Sphere[]] //
   TeXForm
```

\begin{asy}
import three;
import solids;
size(6.6667cm, 6.6667cm);
currentprojection=perspective(2.6,-4.8,4.0);
currentlight=light(rgb(0.5,0.5,1), specular=red, (2,0,2), (2,2,2), (0,2,2));
draw(surface(sphere((0, 0, 0), 1)), rgb(1,1,1));
draw(((-1,-1,-1)–(1,-1,-1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((-1,1,-1)–(1,1,-1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((-1,-1,1)–(1,-1,1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((-1,1,1)–(1,1,1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((-1,-1,-1)–(-1,1,-1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((1,-1,-1)–(1,1,-1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((-1,-1,1)–(-1,1,1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((1,-1,1)–(1,1,1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((-1,-1,-1)–(-1,-1,1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((1,-1,-1)–(1,-1,1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((-1,1,-1)–(-1,1,1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
draw(((1,1,-1)–(1,1,1)), rgb(0.4, 0.4, 0.4)+linewidth(1));
\end{asy}

Graphics3DBox

Line3DBox

Point3DBox

Polygon3DBox

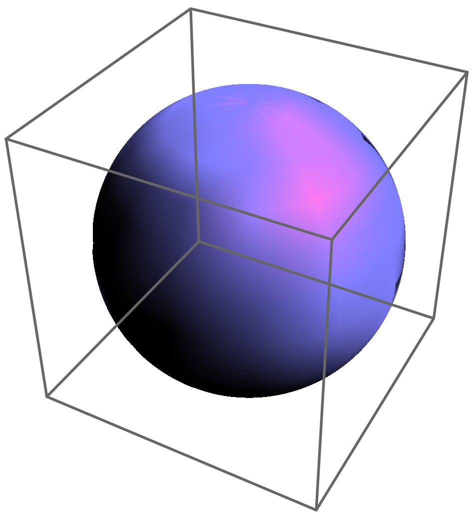
Sphere

`Sphere[{x, y, z}]`
: is a sphere of radius *1* centerd at the point {*x*, *y*, *z*}.

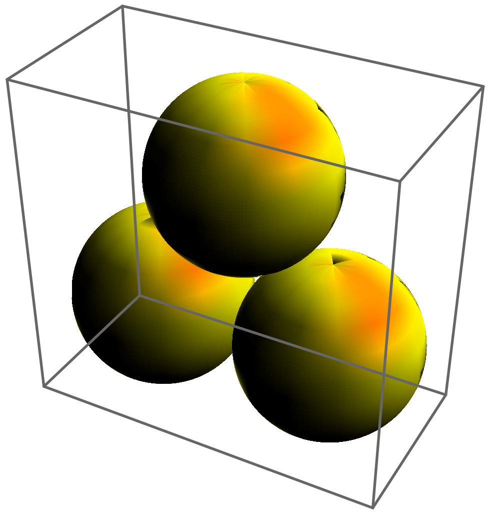
`Sphere[{x, y, z}, r]`
: is a sphere of radius *r* centered at the point *x*, *y*, *z*.

`Sphere[{{x1, y1, z1}, {x2, y2, z2}, ... }, r]`
: is a collection spheres of radius *r* centered at the points {*x1*, *y2*, *z2*}, {*x2*, *y2*, *z2*}, ...

```
>> Graphics3D[Sphere[{0, 0, 0}, 1]]
```



```
>> Graphics3D[{Yellow, Sphere[{{-1, 0, 0}, {1, 0, 0}, {0, 0, Sqrt[3.]}}, 1]}]
```



Sphere3DBox

XVI. Inout

Contents

Center

Format

Assign values to `Format` to control how particular expressions should be formatted when printed to the user.

```
>> Format[f[x___]] := Infix[{x}, "~"]
```

```
>> f[1, 2, 3]
```
$1 \sim 2 \sim 3$

```
>> f[1]
```
1

Raw objects cannot be formatted:

```
>> Format[3] = "three";
```
Cannot assign to raw object 3.

Format types must be symbols:

```
>> Format[r, a + b] = "r";
```
Format type $a + b$ is not a symbol.

Formats must be attached to the head of an expression:

```
>> f /: Format[g[f]] = "my f";
```
Tag f not found or too deep for an assigned rule.

FullForm

```
>> FullForm[a + b * c]
```
Plus $[a, \text{Times}[b, c]]$

```
>> FullForm[2/3]
```
Rational [2, 3]

```
>> FullForm["A string"]
```
"A string"

General

`General` is a symbol to which all general-purpose messages are assigned.

```
>> General::argr
```

'1' called with 1 argument;
'2' arguments are expected.

```
>> Message[Rule::argr, Rule, 2]
```

Rule called with 1 argument;
2 arguments are expected.

Grid

```
>> Grid[{{a, b}, {c, d}}]
```

a	b
c	d

GridBox

Infix

```
>> Format[g[x_, y_]] := Infix[{x
, y}, "#", 350, Left]
```

```
>> g[a, g[b, c]]
```

$a\#(b\#c)$

```
>> g[g[a, b], c]
```

$a\#b\#c$

```
>> g[a + b, c]
```

$(a+b)\#c$

```
>> g[a * b, c]
```

$ab\#c$

```
>> g[a, b] + c
```

$c+a\#b$

```
>> g[a, b] * c
```

$c(a\#b)$

```
>> Infix[{a, b, c}, {"+", "-"}]
```

$a+b-c$

InputForm

```
>> InputForm[a + b * c]
```

$a+b*c$

```
>> InputForm["A string"]
```

"A string"

```
>> InputForm[f'[x]]
```

$\text{Derivative}[1][f][x]$

```
>> InputForm[Derivative[1, 0][f
][x]]
```

$\text{Derivative}[1,0][f][x]$

Left

MakeBoxes

String representation of boxes

```
>> \(x \^ 2\)
```

SuperscriptBox[x, 2]

```
>> \(x \_ 2\)
```

SubscriptBox[x, 2]

```
>> \( a \+ b \% c\)
```

UnderoverscriptBox[a, b, c]

```
>> \( a \& b \% c\)
```

UnderoverscriptBox[a, c, b]

```
>> \(x \& y \)
```

OverscriptBox[x, y]

```
>> \(x \+ y \)
```

UnderscriptBox[x, y]

MathMLForm

```
>> MathMLForm[HoldForm[Sqrt[a
^3]]]
```

<math><msqrt><msup>
<mi>a</mi> <mn>3</mn>
</msup></msqrt></math>

MatrixForm

`>> Array[a,{4,3}]//MatrixForm`

$$\begin{pmatrix} a[1,1] & a[1,2] & a[1,3] \\ a[2,1] & a[2,2] & a[2,3] \\ a[3,1] & a[3,2] & a[3,3] \\ a[4,1] & a[4,2] & a[4,3] \end{pmatrix}$$

Message

`>> a::b = "Hello world!"`

Hello world!

`>> Message[a::b]`

Hello world!

`>> a::c := "Hello ‘1‘, Mr 00‘2‘!"`

`>> Message[a::c, "you", 3 + 4]`

Hello you, Mr 007!

MessageName (::)

`MessageName` is the head of message IDs of the form `symbol::tag`.

`>> FullForm[a::b]`

MessageName [a, "b"]

The second parameter `tag` is interpreted as a string.

`>> FullForm[a::"b"]`

MessageName [a, "b"]

NonAssociative

OutputForm

`>> OutputForm[f'[x]]`

$f'[x]$

`>> OutputForm[Derivative[1, 0][f][x]]`

Derivative $[1,0][f][x]$

`>> OutputForm["A string"]`

A string

`>> OutputForm[Graphics[Rectangle[]]]`

Postfix (//)

`>> b // a`

$a[b]$

`>> c // b // a`

$a[b[c]]$

The postfix operator `//` is parsed to an expression before evaluation:

`>> Hold[x // a // b // c // d // e // f]`

Hold $[f[e[d[c[b[a[x]]]]]]]$

Precedence

`Precedence[`*op*`]`
: returns the precedence of the built-in operator *op*.

`>> Precedence[Plus]`

310.

`>> Precedence[Plus] < Precedence[Times]`

True

Unknown symbols have precedence 670:

```
>> Precedence[f]
```
670.

Other expressions have precedence 1000:

```
>> Precedence[a + b]
```
1 000.

Prefix (@)

```
>> a @ b
```
$a[b]$

```
>> a @ b @ c
```
$a[b[c]]$

```
>> Format[p[x_]] := Prefix[{x}, "*"]
```

```
>> p[3]
```
∗3

```
>> Format[q[x_]] := Prefix[{x}, "~", 350]
```

```
>> q[a+b]
```
$\sim (a+b)$

```
>> q[a*b]
```
$\sim ab$

```
>> q[a]+b
```
$b+\sim a$

The prefix operator `@` is parsed to an expression before evaluation:

```
>> Hold[a @ b @ c @ d @ e @ f @ x]
```
Hold $[a[b[c[d[e[f[x]]]]]]]$

Print

```
>> Print["Hello world!"]
```
Hello world!

```
>> Print["The answer is ", 7 * 6, "."]
```
The answer is 42.

Quiet

`Quiet[`*expr*`, {$s1::t1$, ...}]`
: evaluates *expr*, without messages `{$s1::t1$, ...}` being displayed.

`Quiet[`*expr*`, All]`
: evaluates *expr*, without any messages being displayed.

`Quiet[`*expr*`, None]`
: evaluates *expr*, without all messages being displayed.

`Quiet[`*expr*`,` *off*`,` *on*`]`
: evaluates *expr*, with messages *off* being suppressed, but messages *on* being displayed.

```
>> a::b = "Hello";
```

```
>> Quiet[x+x, {a::b}]
```
$2x$

```
>> Quiet[Message[a::b]; x+x, {a::b}]
```
$2x$

```
>> Message[a::b]; y=Quiet[Message[a::b]; x+x, {a::b}]; Message[a::b]; y
```
Hello
Hello
$2x$

```
>> Quiet[expr, All, All]
```
Arguments 2 and 3 of Quiet [expr, All, All] should not both be All.

Quiet [expr, All, All]

```
>> Quiet[x + x, {a::b}, {a::b}]
```
In Quiet $[x+x, \{a\text{::b}\}, \{a\text{::b}\}]$ the message name(s) $\{a\text{::b}\}$ appear in both the list of messages to switch off and the list of messages to switch on.

Quiet $[x+x, \{a\text{::b}\}, \{a\text{::b}\}]$

Right

Row

RowBox

StandardForm

```
>> StandardForm[a + b * c]
```
$a + bc$

```
>> StandardForm["A string"]
```
A string

`StandardForm` is used by default:

```
>> "A string"
```
A string

```
>> f'[x]
```
$f'[x]$

StringForm

```
>> StringForm["`1` bla `2` blub
   `` bla `2`", a, b, c]
```
a bla *b* blub *c* bla *b*

Style

Subscript

```
>> Subscript[x,1,2,3] // TeXForm
```
x_{1,2,3}

SubscriptBox

Subsuperscript

```
>> Subsuperscript[a, b, c] //
   TeXForm
```
a_b^c

SubsuperscriptBox

Superscript

```
>> Superscript[x,3] // TeXForm
```
x^3

SuperscriptBox

Syntax

`Syntax` is a symbol to which all syntax messages are assigned.

```
>> 1 +
```
Incomplete expression;
more input is needed.

```
>> Sin[1)
```
Invalid syntax at or near token).

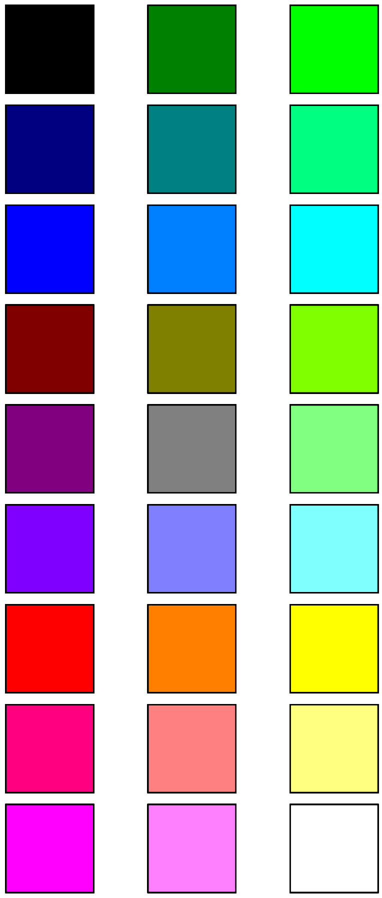
```
>> 1.5``
```
Scan error at position 4.

TableForm

```
>> TableForm[Array[a, {3,2}],
   TableDepth->1]
```
$\{a[1,1], a[1,2]\}$
$\{a[2,1], a[2,2]\}$
$\{a[3,1], a[3,2]\}$

A table of Graphics:

```
>> Table[Style[Graphics[{
   EdgeForm[{Black}], RGBColor[r
   ,g,b], Rectangle[]}],
   ImageSizeMultipliers->{0.2,
   1}], {r,0,1,1/2}, {g
   ,0,1,1/2}, {b,0,1,1/2}] //
   TableForm
```



TeXForm

```
>> TeXForm[HoldForm[Sqrt[a^3]]]
```

\sqrt{a^3}

ToBoxes

```
>> ToBoxes[a + b]
```

RowBox [{a, +, b}]

```
>> ToBoxes[a ^ b] // FullForm
```

SuperscriptBox ["a", "b"]

XVII. Integer

Contents

Ceiling

`Ceiling[x]`
Give first integer greater than x.

```
>> Ceiling[1.2]
```
2

```
>> Ceiling[3/2]
```
2

For complex x, take the ceiling of real an imaginary parts.

```
>> Ceiling[1.3 + 0.7 I]
```
$2 + I$

Floor

`Floor[x]`
gives the smallest integer less than or equal to x.

`Floor[x, a]`
gives the smallest multiple of a less than or equal to x.

```
>> Floor[10.4]
```
10

```
>> Floor[10/3]
```
3

```
>> Floor[10]
```
10

```
>> Floor[21, 2]
```
20

```
>> Floor[2.6, 0.5]
```
2.5

```
>> Floor[-10.4]
```
-11

For complex x, take the floor of real an imaginary parts.

```
>> Floor[1.5 + 2.7 I]
```
$1 + 2I$

For negative a, the smallest multiple of a greater than or equal to x is returned.

```
>> Floor[10.4, -1]
```
11

```
>> Floor[-10.4, -1]
```
-10

IntegerLength

```
>> IntegerLength[123456]
```
6

```
>> IntegerLength[10^10000]
```
10 001

```
>> IntegerLength[-10^1000]
```
1 001

`IntegerLength` with base 2:

```
>> IntegerLength[8, 2]
```
4

Check that `IntegerLength` is correct for the first 100 powers of 10:

```
>> IntegerLength /@ (10 ^ Range
   [100])== Range[2, 101]
```

True

The base must be greater than 1:

```
>> IntegerLength[3, -2]
```

Base -2 is not an integer greater than 1.

IntegerLength$[3, -2]$

XVIII. Linalg

Contents

Cross

`Cross[`a`, `b`]`
computes the vector cross product of a and b.

```
>> Cross[{x1, y1, z1}, {x2, y2, z2}]
```

{y1z2 − y2z1, − x1z2 + x2z1, x1y2 − x2y1}

```
>> Cross[{x, y}]
```

$\{-y, x\}$

```
>> Cross[{1, 2}, {3, 4, 5}]
```

The arguments are expected to be vectors of equal length, and the number of arguments is expected to be 1 less than their length.

Cross [{1,2}, {3,4,5}]

Degree

`Degree`
is number of radians in one degree.

```
>> Cos[60 Degree]
```

$\frac{1}{2}$

Det

`Det[`m`]`
computes the determinant of the matrix m.

```
>> Det[{{1, 1, 0}, {1, 0, 1}, {0, 1, 1}}]
```

−2

Symbolic determinant:

```
>> Det[{{a, b, c}, {d, e, f}, {g, h, i}}]
```

$aei - afh - bdi + bfg + cdh - ceg$

Eigensystem

`Eigensystem[`*m*`]`
: returns a list of {Eigenvalues, Eigenvectors}.

```
>> Eigenvalues[{{1, 1, 0}, {1, 0, 1}, {0, 1, 1}}] // Sort
```

$\{-1,1,2\}$

```
>> Eigenvalues[{{Cos[theta],Sin[theta],0},{-Sin[theta],Cos[theta],0},{0,0,1}}] // Sort
```

$$\left\{1,\mathrm{Cos}\left[\mathrm{theta}\right]+\sqrt{-1+\mathrm{Cos}\left[\mathrm{theta}\right]^2},\mathrm{Cos}\left[\mathrm{theta}\right]-\sqrt{-1+\mathrm{Cos}\left[\mathrm{theta}\right]^2}\right\}$$

```
>> Eigenvalues[{{7, 1}, {-4, 3}}]
```

$\{5,5\}$

Eigenvalues

`Eigenvalues[`*m*`]`
: computes the eigenvalues of the matrix *m*.

```
>> Eigenvalues[{{1, 1, 0}, {1, 0, 1}, {0, 1, 1}}] // Sort
```

$\{-1,1,2\}$

```
>> Eigenvalues[{{Cos[theta],Sin[theta],0},{-Sin[theta],Cos[theta],0},{0,0,1}}] // Sort
```

$$\left\{1,\mathrm{Cos}\left[\mathrm{theta}\right]+\sqrt{-1+\mathrm{Cos}\left[\mathrm{theta}\right]^2},\mathrm{Cos}\left[\mathrm{theta}\right]-\sqrt{-1+\mathrm{Cos}\left[\mathrm{theta}\right]^2}\right\}$$

```
>> Eigenvalues[{{7, 1}, {-4, 3}}]
```

$\{5,5\}$

Eigenvectors

`Eigenvectors[`*m*`]`
: computes the eigenvectors of the matrix *m*.

```
>> Eigenvectors[{{1, 1, 0}, {1, 0, 1}, {0, 1, 1}}]
```

$\{\{1,1,1\},\{1,-2,1\},\{-1,0,1\}\}$

```
>> Eigenvectors[{{1, 0, 0}, {0, 1, 0}, {0, 0, 0}}]
```

$\{\{0,1,0\},\{1,0,0\},\{0,0,1\}\}$

```
>> Eigenvectors[{{2, 0, 0}, {0, -1, 0}, {0, 0, 0}}]
```

$\{\{1,0,0\},\{0,1,0\},\{0,0,1\}\}$

```
>> Eigenvectors[{{0.1, 0.2}, {0.8, 0.5}}]
```

$\{\{0.309016994374947,1.\},\{-0.809016994374947,1.\}\}$

Inverse

`Inverse[`*m*`]`
: computes the inverse of the matrix *m*.

```
>> Inverse[{{1, 2, 0}, {2, 3,
0}, {3, 4, 1}}]
```

$\{\{-3,2,0\},\{2,-1,0\},\{1,-2,1\}\}$

```
>> Inverse[{{1, 0}, {0, 0}}]
```

The matrix $\{\{1,0\},\{0,0\}\}$ is singular.

Inverse$[\{\{1,0\},\{0,0\}\}]$

```
>> Inverse[{{1, 0, 0}, {0, Sqrt
[3]/2, 1/2}, {0,-1 / 2, Sqrt
[3]/2}}]
```

$$\left\{\{1,0,0\},\left\{0,\frac{\sqrt{3}}{2},-\frac{1}{2}\right\},\left\{0,\frac{1}{2},\frac{\sqrt{3}}{2}\right\}\right\}$$

LeastSquares

`LeastSquares[`*m*`, `*b*`]`
: Compute the least squares solution to $m x = b$. Finds an x that solves for b optimally.

```
>> LeastSquares[{{1, 2}, {2, 3},
{5, 6}}, {1, 5, 3}]
```

$$\left\{-\frac{28}{13},\frac{31}{13}\right\}$$

```
>> Simplify[LeastSquares[{{1,
2}, {2, 3}, {5, 6}}, {1, x,
3}]]
```

$$\left\{\frac{12}{13}-\frac{8x}{13},-\frac{4}{13}+\frac{7x}{13}\right\}$$

```
>> LeastSquares[{{1, 1, 1}, {1,
1, 2}}, {1, 3}]
```

Solving for underdetermined system not implemented.

LeastSquares$[\{\{1,1,1\},\{1,1,2\}\},\{1,3\}]$

LinearSolve

`LinearSolve[`*matrix*`, `*right*`]`
: solves the linear equation system *matrix* `. x = ` *right* and returns one corresponding solution `x`.

```
>> LinearSolve[{{1, 1, 0}, {1,
0, 1}, {0, 1, 1}}, {1, 2, 3}]
```

$\{0,1,2\}$

Test the solution:

```
>> {{1, 1, 0}, {1, 0, 1}, {0, 1,
1}} . {0, 1, 2}
```

$\{1,2,3\}$

If there are several solutions, one arbitrary solution is returned:

```
>> LinearSolve[{{1, 2, 3}, {4,
5, 6}, {7, 8, 9}}, {1, 1, 1}]
```

$\{-1,1,0\}$

Infeasible systems are reported:

```
>> LinearSolve[{{1, 2, 3}, {4,
5, 6}, {7, 8, 9}}, {1, -2,
3}]
```

Linear equation encountered that has no solution.

LinearSolve$[\{\{1,2,3\},\{4,5,6\},\{7,8,9\}\},\{1,-2,3\}]$

MatrixExp

`MatrixExp[`*m*`]`
: computes the exponential of the matrix *m*.

```
>> MatrixExp[{{0, 2}, {0, 1}}]
```

$\{\{1,-2+2E\},\{0,E\}\}$

```
>> MatrixExp[{{1.5, 0.5}, {0.5, 2.0}}]
```

$\{\{5.16266024276223, 3.029\text{\textasciitilde}$ $\text{\textasciitilde}519834622\}, \{3.029519834\text{\textasciitilde}$ $\text{\textasciitilde}622, 8.19218007738423\}\}$

MatrixPower

`MatrixPower[`*m*`, `*n*`]`
: computes the *n*th power of a matrix *m*.

```
>> MatrixPower[{{1, 2}, {1, 1}}, 10]
```

$\{\{3363, 4756\}, \{2378, 3363\}\}$

```
>> MatrixPower[{{1, 2}, {2, 5}}, -3]
```

$\{\{169, -70\}, \{-70, 29\}\}$

MatrixRank

`MatrixRank[`*matrix*`]`
: returns the rank of *matrix*.

```
>> MatrixRank[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]
```

2

```
>> MatrixRank[{{1, 1, 0}, {1, 0, 1}, {0, 1, 1}}]
```

3

```
>> MatrixRank[{{a, b}, {3 a, 3 b}}]
```

1

Norm

`Norm[`*m*`, `*l*`]`
: computes the l-norm of matrix m (currently only works for vectors!).

`Norm[`*m*`]`
: computes the 2-norm of matrix m (currently only works for vectors!).

```
>> Norm[{1, 2, 3, 4}, 2]
```

$\sqrt{30}$

```
>> Norm[{10, 100, 200}, 1]
```

310

```
>> Norm[{a, b, c}]
```

$\sqrt{\mathrm{Abs}[a]^2 + \mathrm{Abs}[b]^2 + \mathrm{Abs}[c]^2}$

```
>> Norm[{-100, 2, 3, 4}, Infinity]
```

100

```
>> Norm[1 + I]
```

$\sqrt{2}$

Normalize

`Normalize[`*v*`]`
: calculates the normalized vector *v*.

`Normalize[`*z*`]`
: calculates the normalized complex number *z*.

```
>> Normalize[{1, 1, 1, 1}]
```

$\left\{\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right\}$

```
>> Normalize[1 + I]
```

$\left(\frac{1}{2} + \frac{I}{2}\right)\sqrt{2}$

NullSpace

`NullSpace[`*matrix*`]`
: returns a list of vectors that span the nullspace of *matrix*.

```
>> NullSpace[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]
```

$\{\{1, -2, 1\}\}$

```
>> A = {{1, 1, 0}, {1, 0, 1}, {0, 1, 1}};
```

```
>> NullSpace[A]
```

$\{\}$

```
>> MatrixRank[A]
```

3

PseudoInverse

`PseudoInverse[`*m*`]`
: computes the Moore-Penrose pseudoinverse of the matrix *m*. If *m* is invertible, the pseudoinverse equals the inverse.

```
>> PseudoInverse[{{1, 2}, {2, 3}, {3, 4}}]
```

$$\left\{\left\{-\frac{11}{6}, -\frac{1}{3}, \frac{7}{6}\right\}, \left\{\frac{4}{3}, \frac{1}{3}, -\frac{2}{3}\right\}\right\}$$

```
>> PseudoInverse[{{1, 2, 0}, {2, 3, 0}, {3, 4, 1}}]
```

$\{\{-3, 2, 0\}, \{2, -1, 0\}, \{1, -2, 1\}\}$

```
>> PseudoInverse[{{1.0, 2.5}, {2.5, 1.0}}]
```

{{−0.190476190476190~
~476, 0.476190476190476191
}, {0.47619047619047619,
− 0.190476190476190476}}

RowReduce

`RowReduce[`*matrix*`]`
: returns the reduced row-echelon form of *matrix*.

```
>> RowReduce[{{1, 0, a}, {1, 1, b}}]
```

$\{\{1, 0, a\}, \{0, 1, -a+b\}\}$

```
>> RowReduce[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{pmatrix}$$

SingularValueDecomposition

`SingularValueDecomposition[`*m*`]`
: Calculate the singular value decomposition for matrix *m*. Returns u, s, w such that $m=u s v$, $uu=1$, $v v=1$, *s* is diagonal.

```
>> SingularValueDecomposition[{{1.5, 2.0}, {2.5, 3.0}}]
```

{{{0.538953533497208, 0.842~
~335496539754}, {0.842335496~
~539754, − 0.538953533497208
}}, {{4.63555452966064, 0.},
{0., 0.10786196059193}}, {{
0.628677545037648, 0.777666~
~08796156}, {−0.777666087~
~96156, 0.628677545037648}}}

VectorAngle

`VectorAngle[`*u*`, `*v*`]`
: gives the angles between vectors *u* and *v*

```
>> VectorAngle[{1, 0}, {0, 1}]
```

$$\frac{\text{Pi}}{2}$$

```
>> VectorAngle[{1, 2}, {3, 1}]
```

$$\frac{\text{Pi}}{4}$$

```
>> VectorAngle[{1, 1, 0}, {1, 0,
   1}]
```

$$\frac{\text{Pi}}{3}$$

XIX. Lists

Contents

Accumulate

`Accumulate[`*list*`]`
Accumulate values from list and return new list
</dt>

```
>> Accumulate[{1, 2, 3}]
```
$\{1,3,6\}$

All

`All`
is a possible value for `Span` and `Quiet`.

Append

`Append[`*expr*`, `*item*`]`
returns *expr* with *item* appended to its leaves.

```
>> Append[{1, 2, 3}, 4]
```
$\{1,2,3,4\}$

`Append` works on expressions with heads other than `List`:

```
>> Append[f[a, b], c]
```
$f[a,b,c]$

Unlike `Join`, `Append` does not flatten lists in *item*:

```
>> Append[{a, b}, {c, d}]
```
$\{a,b,\{c,d\}\}$

Array

```
>> Array[f, 4]
```

$\{f[1], f[2], f[3], f[4]\}$

```
>> Array[f, {2, 3}]
```

$\{\{f[1,1], f[1,2], f[1,3]\}, \{f[2,1], f[2,2], f[2,3]\}\}$

```
>> Array[f, {2, 3}, 3]
```

$\{\{f[3,3], f[3,4], f[3,5]\}, \{f[4,3], f[4,4], f[4,5]\}\}$

```
>> Array[f, {2, 3}, {4, 6}]
```

$\{\{f[4,6], f[4,7], f[4,8]\}, \{f[5,6], f[5,7], f[5,8]\}\}$

```
>> Array[f, {2, 3}, 1, Plus]
```

$f[1,1] + f[1,2] + f[1,3] + f[2,1] + f[2,2] + f[2,3]$

Cases

Complement

`Complement[`*all*`, `*e1*`, `*e2*`, ...]`
: returns an expression containing the elements in the set *all* that are not in any of *e1*, *e2*, etc.

`Complement[`*all*`, `*e1*`, `*e2*`, ..., SameTest->`*test*`]`
: applies *test* to the elements in *all* and each of the *ei* to determine equality.

The sets *all*, *e1*, etc can have any head, which must all match. The returned expression has the same head as the input expressions.

```
>> Complement[{a, b, c}, {a, c}]
```

$\{b\}$

```
>> Complement[{a, b, c}, {a, c}, {b}]
```

$\{\}$

```
>> Complement[f[z, y, x, w], f[x], f[x, z]]
```

$f[w, y]$

ConstantArray

```
>> ConstantArray[a, 3]
```

$\{a, a, a\}$

```
>> ConstantArray[a, {2, 3}]
```

$\{\{a, a, a\}, \{a, a, a\}\}$

DeleteDuplicates

`DeleteDuplicates[`*list*`]`
: deletes duplicates from *list*.

`DeleteDuplicates[`*list*`, `*test*`]`
: deletes elements from *list* based on whether the function *test* yields `True` on pairs of elements.

```
>> DeleteDuplicates[{1, 7, 8, 4, 3, 4, 1, 9, 9, 2, 1}]
```

$\{1, 7, 8, 4, 3, 9, 2\}$

```
>> DeleteDuplicates[{3,2,1,2,3,4}, Less]
```

$\{3, 2, 1\}$

Drop

```
>> Drop[{a, b, c, d}, 3]
```

$\{d\}$

```
>> Drop[{a, b, c, d}, -2]
```

$\{a, b\}$

```
>> Drop[{a, b, c, d, e}, {2, -2}]
```

$\{a, e\}$

Drop a submatrix:

```
>> A = Table[i*10 + j, {i, 4}, {
   j, 4}]
```

$\{\{11,12,13,14\},\{21,22,23,24\},\{31,32,33,34\},\{41,42,43,44\}\}$

```
>> Drop[A, {2, 3}, {2, 3}]
```

$\{\{11,14\},\{41,44\}\}$

Extract

`Extract[`*expr*`, ` *list*`]`
extracts parts of *expr* specified by *list*.
`Extract[`*expr*`, {`*list1*`, ` *list2*`, ...}]`
extracts a list of parts.

`Extract[`*expr*`, ` *i*`, ` *j*`, ...]` is equivalent to `Part[`*expr*`, {`*i*`, ` *j*`, ...}]`.

```
>> Extract[a + b + c, {2}]
```

b

```
>> Extract[{{a, b}, {c, d}},
   {{1}, {2, 2}}]
```

$\{\{a,b\},d\}$

First

`First[`*expr*`]`
returns the first elment in *expr*.

`First[`*expr*`]` is equivalent to *expr*`[[1]]`.

```
>> First[{a, b, c}]
```

a

```
>> First[a + b + c]
```

a

```
>> First[x]
```

Nonatomic expression expected.

First$[x]$

Fold

`Fold[`*expr*`, ` *x*`, ` *list*`]`
Expression on all elements of list, with initial value of x.
`Fold[`*expr*`, ` *list*`]`
The same as Fold[*expr*, First[list], Rest[list]]

```
>> Fold[Plus, 5, {1, 1, 1}]
```

8

```
>> Fold[f, 5, {1, 2, 3}]
```

$f[f[f[5,1],2],3]$

FoldList

`FoldList[`*expr*`, ` *x*`, ` *list*`]`
Apply expr successive on all elements of list, and return list, where *x* is the first element.
</dt> FoldList[*expr*, *list*]'
The same as FoldList[*expr*, First[list], Rest[list]].
</dt>

```
>> FoldList[f, x, {1, 2, 3}]
```

$\{x,f[x,1],f[f[x,1],2],f[f[f[x,1],2],3]\}$

```
>> FoldList[Times, {1, 2, 3}]
```

$\{1,2,6\}$

Join

Join concatenates lists.

```
>> Join[{a, b}, {c, d, e}]
```

$\{a,b,c,d,e\}$

```
>> Join[{{a, b}, {c, d}}, {{1,
   2}, {3, 4}}]
```

$\{\{a,b\},\{c,d\},\{1,2\},\{3,4\}\}$

The concatenated expressions may have any head.

```
>> Join[a + b, c + d, e + f]
```
$a+b+c+d+e+f$

However, it must be the same for all expressions.

```
>> Join[a + b, c * d]
```
Heads Plus and Times are expected to be the same.

Join $[a+b, cd]$

Last

`Last[`*expr*`]`
: returns the last elment in *expr*.

`Last[`*expr*`]` is equivalent to *expr*`[[-1]]`.

```
>> Last[{a, b, c}]
```
c

```
>> Last[x]
```
Nonatomic expression expected.

Last $[x]$

Length

```
>> Length[{1, 2, 3}]
```
3

`Length` operates on the `FullForm` of expressions:

```
>> Length[Exp[x]]
```
2

```
>> FullForm[Exp[x]]
```
Power $[E, x]$

The length of atoms is 0:

```
>> Length[a]
```
0

Note that rational and complex numbers are atoms, although their `FullForm` might suggest the opposite:

```
>> Length[1/3]
```
0

```
>> FullForm[1/3]
```
Rational [1, 3]

Level

`Level[`*expr*`, `*levelspec*`]`
: gives a list of all subexpressions of *expr* at the level(s) specified by *levelspec*.

Level uses standard level specifications:

n
: levels 1 through *n*

`Infinity`
: all levels from level 1

`{`*n*`}`
: level *n* only

`{`*m*`, `*n*`}`
: levels *m* through *n*

Level 0 corresponds to the whole expression.
A negative level $-n$ consists of parts with depth n.
Level -1 is the set of atoms in an expression:

```
>> Level[a + b ^ 3 * f[2 x ^ 2],
   {-1}]
```
$\{a, b, 3, 2, x, 2\}$

```
>> Level[{{{{a}}}}, 3]
```
{{a}, {{a}}, {{{a}}}}

```
>> Level[{{{{a}}}}, -4]
```
{{{{a}}}}

```
>> Level[{{{{a}}}}, -5]
```
{}

```
>> Level[h0[h1[h2[h3[a]]]], {0,
   -1}]
```
{a, h3 [a], h2 [h3 [a]], h1 [h2 [h3 [a]]], h0 [h1 [h2 [h3 [a]]]]}

Use the option `Heads -> True` to include heads:

```
>> Level[{{{{a}}}}, 3, Heads -> True]
```

$\{\mathrm{List}, \mathrm{List}, \mathrm{List}, \{a\}, \{\{a\}\}, \{\{\{a\}\}\}\}$

```
>> Level[x^2 + y^3, 3, Heads -> True]
```

$\{\mathrm{Plus}, \mathrm{Power}, x, 2, x^2, \mathrm{Power}, y, 3, y^3\}$

```
>> Level[a ^ 2 + 2 * b, {-1}, Heads -> True]
```

$\{\mathrm{Plus}, \mathrm{Power}, a, 2, \mathrm{Times}, 2, b\}$

```
>> Level[f[g[h]][x], {-1}, Heads -> True]
```

$\{f, g, h, x\}$

```
>> Level[f[g[h]][x], {-2, -1}, Heads -> True]
```

$\{f, g, h, g[h], x, f[g[h]][x]\}$

LevelQ

`LevelQ[`*expr*`]`
: tests whether *expr* is a valid level specification.

```
>> LevelQ[2]
```

True

```
>> LevelQ[{2, 4}]
```

True

```
>> LevelQ[Infinity]
```

True

```
>> LevelQ[a + b]
```

False

List

`List` is the head of lists.

```
>> Head[{1, 2, 3}]
```

List

Lists can be nested:

```
>> {{a, b, {c, d}}}
```

$\{\{a, b, \{c, d\}\}\}$

ListQ

`ListQ[`*expr*`]`
: tests whether *expr* is a `List`.

```
>> ListQ[{1, 2, 3}]
```

True

```
>> ListQ[{{1, 2}, {3, 4}}]
```

True

```
>> ListQ[x]
```

False

MemberQ

Most

`Most[`*expr*`]`
: returns *expr* with the last element removed.

`Most[`*expr*`]` is equivalent to *expr*`[[;;-2]]`.

```
>> Most[{a, b, c}]
```

$\{a, b\}$

```
>> Most[a + b + c]
```

$a + b$

```
>> Most[x]
```

Nonatomic expression expected.

$\mathrm{Most}[x]$

None

`None`
is a possible value for `Span` and `Quiet`.

NotListQ

Part

```
>> A = {a, b, c, d};
```

```
>> A[[3]]
```
c

Negative indizes count from the end:

```
>> {a, b, c}[[-2]]
```
b

`Part` can be applied on any expression, not necessarily lists.

```
>> (a + b + c)[[2]]
```
b

expr`[[0]]` gives the head of *expr*:

```
>> (a + b + c)[[0]]
```
Plus

Parts of nested lists:

```
>> M = {{a, b}, {c, d}};
```

```
>> M[[1, 2]]
```
b

You can use `Span` to specify a range of parts:

```
>> {1, 2, 3, 4}[[2;;4]]
```
$\{2,3,4\}$

```
>> {1, 2, 3, 4}[[2;;-1]]
```
$\{2,3,4\}$

A list of parts extracts elements at certain indices:

```
>> {a, b, c, d}[[{1, 3, 3}]]
```
$\{a,c,c\}$

Get a certain column of a matrix:

```
>> B = {{a, b, c}, {d, e, f}, {g
   , h, i}};
```

```
>> B[[;;, 2]]
```
$\{b,e,h\}$

Extract a submatrix of 1st and 3rd row and the two last columns:

```
>> B = {{1, 2, 3}, {4, 5, 6},
   {7, 8, 9}};
```

```
>> B[[{1, 3}, -2;;-1]]
```
$\{\{2,3\},\{8,9\}\}$

Further examples:

```
>> (a+b+c+d)[[-1;;-2]]
```
0

```
>> x[[2]]
```
Part specification is longer than depth of object.

$x[[2]]$

Assignments to parts are possible:

```
>> B[[;;, 2]] = {10, 11, 12}
```
$\{10,11,12\}$

```
>> B
```
$\{\{1,10,3\},\{4,11,6\},\{7,12,9\}\}$

```
>> B[[;;, 3]] = 13
```
13

```
>> B
```
$\{\{1,10,13\},\{4,11,13\},\{7,12,13\}\}$

```
>> B[[1;;-2]] = t;
```

```
>> B
```
$\{t,t,\{7,12,13\}\}$

```
>> F = Table[i*j*k, {i, 1, 3}, {
   j, 1, 3}, {k, 1, 3}];
```

```
>> F[[;; All, 2 ;; 3, 2]] = t;
```

```
>> F
```

$\{\{\{1,2,3\},\{2,t,6\},\{3,t,9\}\},\{\{2,4,6\},\{4,t,12\},\{6,t,18\}\},\{\{3,6,9\},\{6,t,18\},\{9,t,27\}\}\}$

```
>> F[[;; All, 1 ;; 2, 3 ;; 3]] = k;
```

```
>> F
```

$\{\{\{1,2,k\},\{2,t,k\},\{3,t,9\}\},\{\{2,4,k\},\{4,t,k\},\{6,t,18\}\},\{\{3,6,k\},\{6,t,k\},\{9,t,27\}\}\}$

Of course, part specifications have precedence over most arithmetic operations:

```
>> A[[1]] + B[[2]] + C[[3]] // Hold // FullForm
```

Hold [Plus [Part [A, 1] , Part [B, 2] , Part [C, 3]]]

Partition

`Partition[`*list*`, `*n*`]`
: partitions *list* into sublists of length *n*.

`Parition[`*list*`, `*n*`, `*d*`]`
: partitions *list* into sublists of length *n* which overlap *d* indicies.

```
>> Partition[{a, b, c, d, e, f}, 2]
```

$\{\{a,b\},\{c,d\},\{e,f\}\}$

```
>> Partition[{a, b, c, d, e, f}, 3, 1]
```

$\{\{a,b,c\},\{b,c,d\},\{c,d,e\},\{d,e,f\}\}$

Prepend

`Prepend[`*expr*`, `*item*`]`
: returns *expr* with *item* prepended to its leaves.

`Prepend` is similar to `Append`, but adds *item* to the beginning of *expr*:

```
>> Prepend[{2, 3, 4}, 1]
```

$\{1,2,3,4\}$

`Prepend` works on expressions with heads other than `List`:

```
>> Prepend[f[b, c], a]
```

$f[a,b,c]$

Unlike `Join`, `Prepend` does not flatten lists in *item*:

```
>> Prepend[{c, d}, {a, b}]
```

$\{\{a,b\},c,d\}$

Range

```
>> Range[5]
```

$\{1,2,3,4,5\}$

```
>> Range[-3, 2]
```

$\{-3,-2,-1,0,1,2\}$

```
>> Range[0, 2, 1/3]
```

$\left\{0,\frac{1}{3},\frac{2}{3},1,\frac{4}{3},\frac{5}{3},2\right\}$

Reap

`Reap[`*expr*`]`
: gives the result of evaluating *expr*, together with all values sown during this evaluation. Values sown with different tags are given in different lists.

`Reap[`*expr*`, ` *pattern*`]`
: only yields values sown with a tag matching *pattern*. `Reap[`*expr*`]` is equivalent to `Reap[`*expr*`, _]`.

`Reap[`*expr*`, {`*pattern1*`, ` *pattern2*`, ...}]`
: uses multiple patterns.

`Reap[`*expr*`, ` *pattern*`, ` *f*`]`
: applies *f* on each tag and the corresponding values sown in the form *f*`[tag, {e1, e2, ...}]`.

```
>> Reap[Sow[3]; Sow[1]]
```

$\{1, \{\{3, 1\}\}\}$

```
>> Reap[Sow[2, {x, x, x}]; Sow
   [3, x]; Sow[4, y]; Sow[4, 1],
    {_Symbol, _Integer, x}, f]
```

$\{4, \{\{f[x, \{2,2,2,3\}], f[y, \{4\}]\}, \{f[1, \{4\}]\}, \{f[x, \{2,2,2,3\}]\}\}\}$

Find the unique elements of a list, keeping their order:

```
>> Reap[Sow[Null, {a, a, b, d, c
   , a}], _, # &][[2]]
```

$\{a, b, d, c\}$

Sown values are reaped by the innermost matching `Reap`:

```
>> Reap[Reap[Sow[a, x]; Sow[b,
   1], _Symbol, Print["Inner: ",
    #1]&];, _, f]
```

Inner: x

$\{\text{Null}, \{f[1, \{b\}]\}\}$

When no value is sown, an empty list is returned:

```
>> Reap[x]
```

$\{x, \{\}\}$

ReplacePart

```
>> ReplacePart[{a, b, c}, 1 -> t
   ]
```

$\{t, b, c\}$

```
>> ReplacePart[{{a, b}, {c, d}},
    {2, 1} -> t]
```

$\{\{a, b\}, \{t, d\}\}$

```
>> ReplacePart[{{a, b}, {c, d}},
    {{2, 1} -> t, {1, 1} -> t}]
```

$\{\{t, b\}, \{t, d\}\}$

```
>> ReplacePart[{a, b, c}, {{1},
   {2}} -> t]
```

$\{t, t, c\}$

Delayed rules are evaluated once for each replacement:

```
>> n = 1;
```

```
>> ReplacePart[{a, b, c, d},
   {{1}, {3}} :> n++]
```

$\{1, b, 2, d\}$

Non-existing parts are simply ignored:

```
>> ReplacePart[{a, b, c}, 4 -> t
   ]
```

$\{a, b, c\}$

You can replace heads by replacing part 0:

```
>> ReplacePart[{a, b, c}, 0 ->
   Times]
```

abc

(This is equivalent to `Apply`.)
Negative part numbers count from the end:

```
>> ReplacePart[{a, b, c}, -1 ->
   t]
```

$\{a, b, t\}$

Rest

`Rest[`*expr*`]`
: returns *expr* with the first element removed.

`Rest[`*expr*`]` is equivalent to *expr*`[[2;;]]`.

```
>> Rest[{a, b, c}]
```
$\{b, c\}$

```
>> Rest[a + b + c]
```
$b + c$

```
>> Rest[x]
```
Nonatomic expression expected.

Rest $[x]$

Riffle

```
>> Riffle[{a, b, c}, x]
```
$\{a, x, b, x, c\}$

```
>> Riffle[{a, b, c}, {x, y, z}]
```
$\{a, x, b, y, c, z\}$

```
>> Riffle[{a, b, c, d, e, f}, {x
   , y, z}]
```
$\{a, x, b, y, c, z, d, x, e, y, f\}$

Select

```
>> Select[{-3, 0, 1, 3, a},
   #>0&]
```
$\{1, 3\}$

```
>> Select[f[a, 2, 3], NumberQ]
```
$f[2, 3]$

```
>> Select[a, True]
```
Nonatomic expression expected.

Select $[a, \text{True}]$

Sow

`Sow[`*e*`]`
: sends the value *e* to the innermost `Reap`.

`Sow[`*e*`, `*tag*`]`
: sows *e* using *tag*. `Sow[`*e*`]` is equivalent to `Sow[`*e*`, Null]`.

`Sow[`*e*`, {`*tag1*`, `*tag2*`, ...}]`
: uses multiple tags.

Span

`Span` is the head of span ranges like `1;;3`.

```
>> ;; // FullForm
```
Span $[1, \text{All}]$

```
>> 1;;4;;2 // FullForm
```
Span $[1, 4, 2]$

```
>> 2;;-2 // FullForm
```
Span $[2, -2]$

```
>> ;;3 // FullForm
```
Span $[1, 3]$

Split

`Split[`*list*`]`
: splits *list* into collections of consecutive identical elements.

`Split[`*list*`, `*test*`]`
: splits *list* based on whether the function *test* yields `True` on consecutive elements.

```
>> Split[{x, x, x, y, x, y, y, z
   }]
```
$\{\{x, x, x\}, \{y\}, \{x\}, \{y, y\}, \{z\}\}$

Split into increasing or decreasing runs of elements

```
>> Split[{1, 5, 6, 3, 6, 1, 6,
   3, 4, 5, 4}, Less]
```
$\{\{1,5,6\},\{3,6\},\{1,6\},\{3,4,5\},\{4\}\}$

```
>> Split[{1, 5, 6, 3, 6, 1, 6,
   3, 4, 5, 4}, Greater]
```
$\{\{1\},\{5\},\{6,3\},\{6,1\},\{6,3\},\{4\},\{5,4\}\}$

Split based on first element

```
>> Split[{x -> a, x -> y, 2 -> a
   , z -> c, z -> a}, First[#1]
   === First[#2] &]
```
$\{\{x\text{->}a, x\text{->}y\},\{2\text{->}a\},\{z\text{->}c, z\text{->}a\}\}$

SplitBy

`Split[`*list*`,` *f*`]`
: splits *list* into collections of consecutive elements that give the same result when *f* is applied.

```
>> SplitBy[Range[1, 3, 1/3],
   Round]
```
$$\left\{\left\{1,\frac{4}{3}\right\},\left\{\frac{5}{3},2,\frac{7}{3}\right\},\left\{\frac{8}{3},3\right\}\right\}$$

```
>> SplitBy[{1, 2, 1, 1.2}, {
   Round, Identity}]
```
$\{\{\{1\}\},\{\{2\}\},\{\{1\},\{1.2\}\}\}$

```
>> SplitBy[{1, 2, 1, 1.2}, {
   Round, Identity}]
```
$\{\{\{1\}\},\{\{2\}\},\{\{1\},\{1.2\}\}\}$

Table

```
>> Table[x, {4}]
```
$\{x,x,x,x\}$

```
>> n = 0;
```

```
>> Table[n = n + 1, {5}]
```
$\{1,2,3,4,5\}$

```
>> Table[i, {i, 4}]
```
$\{1,2,3,4\}$

```
>> Table[i, {i, 2, 5}]
```
$\{2,3,4,5\}$

```
>> Table[i, {i, 2, 6, 2}]
```
$\{2,4,6\}$

```
>> Table[i, {i, Pi, 2 Pi, Pi /
   2}]
```
$$\left\{\text{Pi},\frac{3\text{Pi}}{2},2\text{Pi}\right\}$$

```
>> Table[x^2, {x, {a, b, c}}]
```
$\{a^2,b^2,c^2\}$

`Table` supports multi-dimensional tables:

```
>> Table[{i, j}, {i, {a, b}}, {j
   , 1, 2}]
```
$\{\{\{a,1\},\{a,2\}\},\{\{b,1\},\{b,2\}\}\}$

Take

```
>> Take[{a, b, c, d}, 3]
```
$\{a,b,c\}$

```
>> Take[{a, b, c, d}, -2]
```
$\{c,d\}$

```
>> Take[{a, b, c, d, e}, {2,
   -2}]
```
$\{b,c,d\}$

Take a submatrix:

```
>> A = {{a, b, c}, {d, e, f}};
```

```
>> Take[A, 2, 2]
```
$\{\{a,b\},\{d,e\}\}$

Take a single column:

```
>> Take[A, All, {2}]
```
$\{\{b\},\{e\}\}$

Total

`Total[`*list*`]`
: Add all values up to calculate total Equivalent to Fold[Plus, *list*] or Apply[Plus, *list*]

</dt> `Total[`*list*`, ` *n*`]`
: Total all values up to level *n*

</dt> `Total[`*list*`, ${n}$]`
: Total at level ${n}$

</dt> `Total[`*list*`, ${n_1, n_2}$]`
: Total at levels ${n_1, n_2}$

</dt>

```
>> Total[{1, 2, 3}]
```
6

```
>> Total[{{1, 2, 3}, {4, 5, 6}, {7, 8 ,9}}]
```
$\{12, 15, 18\}$

Total over rows and columns

```
>> Total[{{1, 2, 3}, {4, 5, 6}, {7, 8 ,9}}, 2]
```
45

Total over rows instead of columns

```
>> Total[{{1, 2, 3}, {4, 5, 6}, {7, 8 ,9}}, {2}]
```
$\{6, 15, 24\}$

Tuples

`Tuples[`*list*`, ` *n*`]`
: returns a list of all *n*-tuples of elements in *list*.

`Tuples[{`*list1*`, ` *list2*`, ...}]`
: returns a list of tuples with elements from the given lists.

```
>> Tuples[{a, b, c}, 2]
```
$\{\{a,a\}, \{a,b\}, \{a,c\}, \{b,a\}, \{b,b\}, \{b,c\}, \{c,a\}, \{c,b\}, \{c,c\}\}$

```
>> Tuples[{}, 2]
```
$\{\}$

```
>> Tuples[{a, b, c}, 0]
```
$\{\{\}\}$

```
>> Tuples[{{a, b}, {1, 2, 3}}]
```
$\{\{a,1\}, \{a,2\}, \{a,3\}, \{b,1\}, \{b,2\}, \{b,3\}\}$

The head of *list* need not be `List`:

```
>> Tuples[f[a, b, c], 2]
```
$\{f[a,a], f[a,b], f[a,c], f[b,a], f[b,b], f[b,c], f[c,a], f[c,b], f[c,c]\}$

However, when specifying multiple expressions, `List` is always used:

```
>> Tuples[{f[a, b], g[c, d]}]
```
$\{\{a,c\}, \{a,d\}, \{b,c\}, \{b,d\}\}$

UnitVector

```
>> UnitVector[2]
```
$\{0, 1\}$

```
>> UnitVector[4, 3]
```
$\{0, 0, 1, 0\}$

XX. Logic

Contents

And (&&)

`And[`*expr1*`, `*expr2*`, ...]`
: evaluates expressions until one evaluation results in `False`, in which case `And` returns `False`. If all expressions evaluate to `True`, `And` returns `True`.

>> **`True && True && False`**
False

>> **`a && b && True && c`**
a&&*b*&&*c*

False

Not (!)

`Not` negates a logical expression.

>> **`!True`**
False

>> **`!False`**
True

>> **`!b`**
!*b*

Or (||)

`Or[`*expr1*`, `*expr2*`, ...]` evaluates expressions until one evaluation results in `True`, in which case `Or` returns `True`. If all expressions evaluate to `False`, `Or` returns `False`.

>> **`False || True`**
True

>> **`a || False || b`**
a||*b*

True

XXI. Numbertheory

Contents

CoprimeQ

Test whether two numbers are coprime by computing their greatest common divisor

```
>> CoprimeQ[7, 9]
   True

>> CoprimeQ[-4, 9]
   True

>> CoprimeQ[12, 15]
   False
```

CoprimeQ also works for complex numbers

```
>> CoprimeQ[1+2I, 1-I]
   True

>> CoprimeQ[4+2I, 6+3I]
   False

>> CoprimeQ[2, 3, 5]
   True

>> CoprimeQ[2, 4, 5]
   False
```

EvenQ

```
>> EvenQ[4]
   True

>> EvenQ[-3]
   False

>> EvenQ[n]
   False
```

FactorInteger

`FactorInteger[`*n*`]`
: returns the factorization of n as a list of factors and exponents.

```
>> factors = FactorInteger[2010]
```
$\{\{2,1\},\{3,1\},\{5,1\},\{67,1\}\}$

To get back the original number:

```
>> Times @@ Power @@@ factors
   2010
```

`FactorInteger` factors rationals using negative exponents:

```
>> FactorInteger[2010 / 2011]
```
$\{\{2,1\},\{3,1\},\{5,1\},\{67,1\},\{2011,-1\}\}$

GCD

`GCD[`*n1*`, `*n2*`, ...]`
: computes the greatest common divisor of the given integers.

>> `GCD[20, 30]`
10

>> `GCD[10, y]`
$\text{GCD}[10, y]$

`GCD` is `Listable`:

>> `GCD[4, {10, 11, 12, 13, 14}]`
$\{2, 1, 4, 1, 2\}$

`GCD` does not work for rational numbers and Gaussian integers yet.

IntegerExponent

`IntegerExponent[`*n*`, `*b*`]`
: gives the highest exponent of *b* that divides *n*.

>> `IntegerExponent[16, 2]`
4

>> `IntegerExponent[-510000]`
4

>> `IntegerExponent[10, b]`
$\text{IntegerExponent}[10, b]$

LCM

`LCM[`*n1*`, `*n2*`, ...]`
: computes the least common multiple of the given integers.

>> `LCM[15, 20]`
60

>> `LCM[20, 30, 40, 50]`
600

Mod

>> `Mod[14, 6]`
2

>> `Mod[-3, 4]`
1

>> `Mod[-3, -4]`
-3

>> `Mod[5, 0]`
The argument 0 should be nonzero.
$\text{Mod}[5, 0]$

NextPrime

`NextPrime[`*n*`]`
: gives the next prime after *n*.

`NextPrime[`*n*`,`*k*`]`
: gives the *k*th prime after *n*.

>> `NextPrime[10000]`
10 007

>> `NextPrime[100, -5]`
73

>> `NextPrime[10, -5]`
-2

>> `NextPrime[100, 5]`
113

>> `NextPrime[5.5, 100]`
563

>> `NextPrime[5, 10.5]`
$\text{NextPrime}[5, 10.5]$

OddQ

>> `OddQ[-3]`
True

>> `OddQ[0]`
False

PowerMod

```
>> PowerMod[2, 10000000, 3]
```
1

```
>> PowerMod[3, -2, 10]
```
9

```
>> PowerMod[0, -1, 2]
```
0 is not invertible modulo 2.
PowerMod [0, − 1, 2]

```
>> PowerMod[5, 2, 0]
```
The argument 0 should be nonzero.
PowerMod [5, 2, 0]

`PowerMod` does not support rational coefficients (roots) yet.

Prime

`Prime[n]`
: returns the nth prime number.

```
>> Prime[1]
```
2

```
>> Prime[167]
```
991

PrimePi

`PrimePi[x]`
: gives the number of primes less than or equal to x

```
>> PrimePi[100]
```
25

```
>> PrimePi[-1]
```
0

```
>> PrimePi[3.5]
```
2

```
>> PrimePi[E]
```
1

PrimePowerQ

Tests wheter a number is a prime power

```
>> PrimePowerQ[9]
```
True

```
>> PrimePowerQ[52142]
```
False

```
>> PrimePowerQ[-8]
```
True

```
>> PrimePowerQ[371293]
```
True

PrimeQ

For very large numbers, `PrimeQ` uses probabilistic prime testing, so it might be wrong sometimes (a number might be composite even though `PrimeQ` says it is prime). The algorithm might be changed in the future.

```
>> PrimeQ[2]
```
True

```
>> PrimeQ[-3]
```
True

```
>> PrimeQ[137]
```
True

```
>> PrimeQ[2 ^ 127 - 1]
```
True

All prime numbers between 1 and 100:

```
>> Select[Range[100], PrimeQ]
```
$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97\}$

`PrimeQ` has attribute `Listable`:

```
>> PrimeQ[Range[20]]
```
{False, True, True, False, True, False, True, False, False, False, True, False, True, False, False, False, True, False, True, False}

RandomPrime

`RandomPrime[{`*imin*`, $imax}]`
: gives a random prime between *imin* and *imax*.

'RanomPrime[*imax*]
: gives a random prime between 2 and *imax*.

`RandomPrime[`*range*`, `*n*`]`
: gives a list of *n* random primes in *range*.

```
>> RandomPrime[{14, 17}]
```
17

```
>> RandomPrime[{14, 16}, 1]
```
There are no primes in the specified interval.

RandomPrime [{14, 16} , 1]

```
>> RandomPrime[{8,12}, 3]
```
{11, 11, 11}

```
>> RandomPrime[{10,30}, {2,5}]
```
{{17, 17, 17, 17, 17} , {17, 17, 17, 17, 17}}

XXII. Numeric

Contents

BaseForm

`BaseForm[`*expr*`, `*n*`]`
prints mumbers in *expr* in base *n*.

```
>> BaseForm[33, 2]
```
$100\,001_2$

```
>> BaseForm[234, 16]
```
ea_{16}

```
>> BaseForm[12.3, 2]
```
$1\,100.010011001100110011_2$

```
>> BaseForm[-42, 16]
```
-2a_{16}

```
>> BaseForm[x, 2]
```
x

```
>> BaseForm[12, 3] // FullForm
```
BaseForm [12, 3]

Bases must be between 2 and 36:

```
>> BaseForm[12, -3]
```
Positive machine-sized integer expected at position 2 in BaseForm[12, − 3].

MakeBoxes [BaseForm [12, − 3] , StandardForm] is not a valid box structure.

```
>> BaseForm[12, 100]
```
Requested base 100 must be between 2 and 36.

MakeBoxes [BaseForm [12, 100] , StandardForm] is not a valid box structure.

Chop

`Chop[`*expr*`]`
replaces floating point numbers close to 0 by 0.

`Chop[`*expr*`, `*delta*`]`
uses a tolerance of *delta*. The default tolerance is `10^-10`.

```
>> Chop[10.0 ^ -16]
```
0

```
>> Chop[10.0 ^ -9]
```
$1. \times 10^{-9}$

```
>> Chop[10 ^ -11 I]
```
$$\frac{I}{100\,000\,000\,000}$$

```
>> Chop[0. + 10 ^ -11 I]
```
0

IntegerDigits

`IntegerDigits[`*n*`]`
: returns a list of the base-10 digits in the integer *n*.

`IntegerDigits[`*n*`, `*base*`]`
: returns a list of the base-*base* digits in *n*.

`IntegerDigits[`*n*`, `*base*`, `*length*`]`
: returns a list of length *length*, truncating or padding with zeroes on the left as necessary.

```
>> IntegerDigits[76543]
```
$\{7,6,5,4,3\}$

The sign of *n* is discarded:

```
>> IntegerDigits[-76543]
```
$\{7,6,5,4,3\}$

```
>> IntegerDigits[15, 16]
```
$\{15\}$

```
>> IntegerDigits[1234, 16]
```
$\{4,13,2\}$

```
>> IntegerDigits[1234, 10, 5]
```
$\{0,1,2,3,4\}$

MachinePrecision

`MachinePrecision`
: is a “pessimistic” (integer) estimation of the internally used standard precision.

```
>> N[MachinePrecision]
```
18.

N

`N[`*expr*`, `*prec*`]`
: evaluates *expr* numerically with a precision of *prec* digits.

```
>> N[Pi, 50]
```
3.14159265358979323846264ˆ
ˆ33832795028841971693993751

```
>> N[1/7]
```
0.142857142857142857

```
>> N[1/7, 5]
```
0.14286

You can manually assign numerical values to symbols. When you do not specify a precision, `MachinePrecision` is taken.

```
>> N[a] = 10.9
```
10.9

```
>> a
```
a

`N` automatically threads over expressions, except when a symbol has attributes `NHoldAll`, `NHoldFirst`, or `NHoldRest`.

```
>> N[a + b]
```
$10.9 + b$

```
>> N[a, 20]
```
a

```
>> N[a, 20] = 11;
```

```
>> N[a + b, 20]
```
$11. + b$

```
>> N[f[a, b]]
```
$f[10.9, b]$

```
>> SetAttributes[f, NHoldAll]
```

```
>> N[f[a, b]]
```
$f[a, b]$

The precision can be a pattern:

```
>> N[c, p_?(#>10&)] := p
```

```
>> N[c, 3]
```
c

```
>> N[c, 11]
```
11.

You can also use `UpSet` or `TagSet` to specify values for `N`:

```
>> N[d] ^= 5;
```

However, the value will not be stored in `UpValues`, but in `NValues` (as for `Set`):

```
>> UpValues[d]
```
{}

```
>> NValues[d]
```
{HoldPattern[*N*[*d*, MachinePrecision]]:>5}

```
>> e /: N[e] = 6;
```

```
>> N[e]
```
6.

Values for `N[`*expr*`]` must be associated with the head of *expr*:

```
>> f /: N[e[f]] = 7;
```
Tag *f* not found or too deep for an assigned rule.

You can use `Condition`:

```
>> N[g[x_, y_], p_] := x + y * Pi /; x + y > 3
```

```
>> SetAttributes[g, NHoldRest]
```

```
>> N[g[1, 1]]
```
g[1., 1]

```
>> N[g[2, 2]]
```
8.28318530717958648

NumericQ

`NumericQ[`*expr*`]`
: tests whether *expr* represents a numeric quantity.

```
>> NumericQ[2]
```
True

```
>> NumericQ[Sqrt[Pi]]
```
True

```
>> NumberQ[Sqrt[Pi]]
```
False

Precision

`Precision[`*expr*`]`
: examines the number of significant digits of *expr*.

This is rather a proof-of-concept than a full

implementation. Precision of compound expression is not supported yet.

```
>> Precision[1]
```
∞

```
>> Precision[1/2]
```
∞

```
>> Precision[0.5]
```
18.

Round

`Round[`*expr*`]`
: rounds *expr* to the nearest integer.

`Round[`*expr*`, `*k*`]`
: rounds *expr* to the closest multiple of *k*.

```
>> Round[10.6]
```
11

```
>> Round[0.06, 0.1]
```
0.1

```
>> Round[0.04, 0.1]
```
0

Constants can be rounded too

```
>> Round[Pi, .5]
```
3.

```
>> Round[Pi^2]
```
10

Round to exact value

```
>> Round[2.6, 1/3]
```
$\frac{8}{3}$

```
>> Round[10, Pi]
```
3Pi

Round complex numbers

```
>> Round[6/(2 + 3 I)]
```
$1 - I$

```
>> Round[1 + 2 I, 2 I]
```
$2I$

Round Negative numbers too

```
>> Round[-1.4]
```
-1

Expressions other than numbers remain unevaluated:

```
>> Round[x]
```
Round $[x]$

```
>> Round[1.5, k]
```
Round $[1.5, k]$

XXIII. Options

Contents

Default

`Default[`f`]`
: gives the default value for an omitted paramter of f.

`Default[`f`, `k`]`
: gives the default value for a parameter on the kth position.

`Default[`f`, `k`, `n`]`
: gives the default value for the kth parameter out of n.

Assign values to `Default` to specify default values.

```
>> Default[f] = 1
```
1

```
>> f[x_.] := x ^ 2
```

```
>> f[]
```
1

Default values are stored in `DefaultValues`:

```
>> DefaultValues[f]
```
$\{\text{HoldPattern}\,[\text{Default}\,[f]]\,{:>}1\}$

You can use patterns for k and n:

```
>> Default[h, k_, n_] := {k, n}
```

Note that the position of a parameter is relative to the pattern, not the matching expression:

```
>> h[] /. h[___, ___, x_., y_.,
   ___] -> {x, y}
```
$\{\{3,5\},\{4,5\}\}$

NotOptionQ

```
>> NotOptionQ[x]
```
True

```
>> NotOptionQ[2]
```
True

```
>> NotOptionQ["abc"]
```
True

```
>> NotOptionQ[a -> True]
```
False

OptionQ

```
>> OptionQ[a -> True]
```
True

```
>> OptionQ[a :> True]
```
True

```
>> OptionQ[{a -> True}]
```
True

```
>> OptionQ[{a :> True}]
```
True

```
>> OptionQ[x]
```
False

OptionValue

`OptionValue[name]`
: gives the value of the option *name* as specified in a call to a function with `OptionsPattern`.

```
>> f[a->3] /. f[OptionsPattern
   [{}]] -> {OptionValue[a]}
```
{3}

Unavailable options generate a message:

```
>> f[a->3] /. f[OptionsPattern
   [{}]] -> {OptionValue[b]}
```
Option name b not found.

{OptionValue[b]}

The argument of `OptionValue` must be a symbol:

```
>> f[a->3] /. f[OptionsPattern
   [{}]] -> {OptionValue[a+b]}
```
Argument $a + b$ at position 1 is expected to be a symbol.

{OptionValue[$a + b$]}

However, it can be evaluated dynamically:

```
>> f[a->5] /. f[OptionsPattern
   [{}]] -> {OptionValue[Symbol
   ["a"]]}
```
{5}

Options

`Options[f]`
: gives a list of optional arguments to f and their default values.

You can assign values to `Options` to specify options.

```
>> Options[f] = {n -> 2}
```
{n->2}

```
>> Options[f]
```
{n:>2}

```
>> f[x_, OptionsPattern[f]] := x
   ^ OptionValue[n]
```

```
>> f[x]
```
x^2

```
>> f[x, n -> 3]
```
x^3

Delayed option rules are evaluated just when the corresponding `OptionValue` is called:

```
>> f[a :> Print["value"]] /. f[
   OptionsPattern[{}]] :> (
   OptionValue[a]; Print["
   between"]; OptionValue[a]);
```
value
between
value

In contrast to that, normal option rules are evaluated immediately:

```
>> f[a -> Print["value"]] /. f[
   OptionsPattern[{}]] :> (
   OptionValue[a]; Print["
   between"]; OptionValue[a]);
```
value
between

Options must be rules or delayed rules:

```
>> Options[f] = {a}
```
{a} is not a valid list of option rules.

{a}

A single rule need not be given inside a list:

```
>> Options[f] = a -> b
```
a->b

```
>> Options[f]
```
{a:>b}

Options can only be assigned to symbols:

```
>> Options[a + b] = {a -> b}
```

Argument $a + b$ at position 1 is expected to be a symbol.

$\{a\text{->}b\}$

XXIV. Patterns

Contents

Alternatives (|)

```
>> a+b+c+d/.(a|b)->t
```

$c + d + 2t$

Blank

_ or Blank[]
: represents any single expression in a pattern.

_*h* or Blank[*h*]
: represents any expression with head *h*.

```
>> MatchQ[a + b, _]
```

True

Patterns of the form _*h* can be used to test the types of objects:

```
>> MatchQ[42, _Integer]
```

True

```
>> MatchQ[1.0, _Integer]
```

False

```
>> {42, 1.0, x} /. {_Integer ->
   "integer", _Real -> "real"}
   // InputForm
```

{"integer", "real", *x*}

Blank only matches a single expression:

```
>> MatchQ[f[1, 2], f[_]]
```

False

BlankNullSequence

___ or BlankNullSequence[]
: represents any sequence of expression leaves in a pattern, including an empty sequence.

BlankNullSequence is like BlankSequence, except it can match an empty sequence:

```
>> MatchQ[f[], f[___]]
```

True

BlankSequence

`__` or `BlankSequence[]`
: represents any non-empty sequence of expression leaves in a pattern.

`__`*h* or `BlankSequence[`*h*`]`
: represents any sequence of leaves, all of which have head *h*.

Use a `BlankSequence` pattern to stand for a non-empty sequence of arguments:

```
>> MatchQ[f[1, 2, 3], f[__]]
   True

>> MatchQ[f[], f[__]]
   False
```

`__`*h* will match only if all leaves have head *h*:

```
>> MatchQ[f[1, 2, 3], f[
   __Integer]]
   True

>> MatchQ[f[1, 2.0, 3], f[
   __Integer]]
   False
```

The value captured by a named `BlankSequence` pattern is a `Sequence` object:

```
>> f[1, 2, 3] /. f[x__] -> x
   Sequence[1,2,3]
```

Condition (/;)

`Condition` sets a condition on the pattern to match, using variables of the pattern.

```
>> f[3] /. f[x_] /; x>0 -> t
   t

>> f[-3] /. f[x_] /; x>0 -> t
   f[-3]
```

`Condition` can be used in an assignment:

```
>> f[x_] := p[x] /; x>0

>> f[3]
   p[3]

>> f[-3]
   f[-3]
```

HoldPattern

`HoldPattern[`*expr*`]` is equivalent to *expr* for pattern matching, but maintains it in an unevaluated form.

```
>> HoldPattern[x + x]
   HoldPattern[x + x]

>> x /. HoldPattern[x] -> t
   t
```

`HoldPattern` has attribute `HoldAll`:

```
>> Attributes[HoldPattern]
   {HoldAll, Protected}
```

MatchQ

`MatchQ[`*expr*`, `*form*`]`
: tests whether *expr* matches *form*.

```
>> MatchQ[123, _Integer]
   True

>> MatchQ[123, _Real]
   False
```

Optional (:)

`Optional[`*patt*`, `*default*`]` or *patt* `:` *default*
: is a pattern which matches *patt* and which, if omitted should be replaced by *default*.

```
>> f[x_, y_:1] := {x, y}

>> f[1, 2]
   {1,2}

>> f[a]
   {a,1}
```

Note that *symb* : *patt* represents a `Pattern` object. However, there is no disambiguity, since *symb* has to be a symbol in this case.

```
>> x:_ // FullForm
```
Pattern [x, Blank []]

```
>> _:d // FullForm
```
Optional [Blank [] , d]

```
>> x:_+y_:d // FullForm
```
Pattern [x, Plus [Blank [ ] , Optional [Pattern [y, Blank []] , d]]]

`s_.` is equivalent to `Optional[s_]` and represents an optional parameter which, if omitted, gets its value from `Default`.

```
>> FullForm[s_.]
```
Optional [Pattern [s, Blank []]]

```
>> Default[h, k_] := k
```

```
>> h[a] /. h[x_, y_.] -> {x, y}
```
$\{a, 2\}$

OptionsPattern

`OptionsPattern[`*f*`]`
: is a pattern that stands for a sequence of options given to a function, with default values taken from `Options[`*f*`]`. The options can be of the form *opt*`->`*value* or *opt*`:>`*value*, and might be in arbitrarily nested lists.

`OptionsPattern[{`*opt1*`->`*value1*`, ...}]`
: takes explicit default values from the given list. The list may also contain symbols *f*, for which `Options[`*f*`]` is taken into account; it may be arbitrarily nested. `OptionsPattern[{}]` does not use any default values.

The option values can be accessed using `OptionValue`.

```
>> f[x_, OptionsPattern[{n->2}]]
    := x ^ OptionValue[n]
```

```
>> f[x]
```
x^2

```
>> f[x, n->3]
```
x^3

Delayed rules as options:

```
>> e = f[x, n:>a]
```
x^a

```
>> a = 5;
```

```
>> e
```
x^5

Options might be given in nested lists:

```
>> f[x, {{{n->4}}}]
```
x^4

PatternTest (?)

```
>> MatchQ[3, _Integer?(#>0&)]
```
True

```
>> MatchQ[-3, _Integer?(#>0&)]
```
False

Pattern

`Pattern[`*symb*`, `*patt*`]` or *symb* : *patt*
: assigns the name *symb* to the pattern *patt*.

symb_head
: is equivalent to *symb* : *_head* (accordingly with __ and ___).

symb : *patt* : *default*
: is a pattern with name *symb* and default value *default*, equivalent to `Optional[`*patt* : *symb*`, `*default*`]`.

```
>> FullForm[a_b]
```
Pattern [a, Blank [b]]

```
>> FullForm[a:_:b]
```
Optional [Pattern [a, Blank []] , b]

`Pattern` has attribute `HoldFirst`, so it does not evaluate its name:

```
>> x = 2
```
2

```
>> x_
```
x_

Nested `Pattern` assign multiple names to the same pattern. Still, the last parameter is the default value.

```
>> f[y] /. f[a:b:_:d] -> {a, b}
```
$\{y, y\}$

This is equivalent to:

```
>> f[] /. f[a:b_:d] -> {a, b}
```
$\{d, d\}$

`FullForm`:

```
>> FullForm[a:b:c:d:e]
```
Optional [Pattern [a, b] , Optional [Pattern [c, d] , e]]

Repeated (..)

```
>> a_Integer.. // FullForm
```
Repeated [Pattern [a, Blank [Integer]]]

```
>> 0..1//FullForm
```
Repeated [0]

```
>> {{}, {a}, {a, b}, {a, a, a}, {a, a, a, a}} /. {Repeated[x : a | b, 3]} -> x
```
$\{\{\}, a, \{a, b\}, a, \{a, a, a, a\}\}$

```
>> f[x, 0, 0, 0] /. f[x, s:0..] -> s
```
Sequence [0, 0, 0]

RepeatedNull (...)

```
>> a___Integer...//FullForm
```
RepeatedNull [Pattern [a, BlankNullSequence [Integer]]]

```
>> f[x] /. f[x, 0...] -> t
```
t

ReplaceAll (/.)

```
>> a+b+c /. c->d
```
$a + b + d$

```
>> g[a+b+c,a]/.g[x_+y_,x_]->{x,y}
```
$\{a, b + c\}$

If *rules* is a list of lists, a list of all possible respective replacements is returned:

```
>> {a, b} /. {{a->x, b->y}, {a->u, b->v}}
```
$\{\{x, y\}, \{u, v\}\}$

The list can be arbitrarily nested:

```
>> {a, b} /. {{{a->x, b->y}, {a->w, b->z}}, {a->u, b->v}}
```
$\{\{\{x, y\}, \{w, z\}\}, \{u, v\}\}$

```
>> {a, b} /. {{{a->x, b->y}, a->w, b->z}, {a->u, b->v}}
```
Elements of $\{\{a->x, b->y\}, a->w, b->z\}$ are a mixture of lists and nonlists.

$\{\{a, b\} /. \{\{a->x, b->y\}, a->w, b->z\}, \{u, v\}\}$

ReplaceList

Get all subsequences of a list:

```
>> ReplaceList[{a, b, c}, {___, x__, ___} -> {x}]
```
$\{\{a\}, \{a, b\}, \{a, b, c\}, \{b\}, \{b, c\}, \{c\}\}$

You can specify the maximum number of items:

```
>> ReplaceList[{a, b, c}, {___,
   x__, ___} -> {x}, 3]
```

$\{\{a\}, \{a,b\}, \{a,b,c\}\}$

```
>> ReplaceList[{a, b, c}, {___,
   x__, ___} -> {x}, 0]
```

$\{\}$

If no rule matches, an empty list is returned:

```
>> ReplaceList[a, b->x]
```

$\{\}$

Like in `ReplaceAll`, *rules* can be a nested list:

```
>> ReplaceList[{a, b, c}, {{{___
   , x__, ___} -> {x}}, {{a, b,
   c} -> t}}, 2]
```

$\{\{\{a\}, \{a,b\}\}, \{t\}\}$

```
>> ReplaceList[expr, {}, -1]
```

Non-negative integer or Infinity expected at position 3.

ReplaceList $[$expr, $\{\}, -1]$

Possible matches for a sum:

```
>> ReplaceList[a + b + c, x_ +
   y_ -> {x, y}]
```

$\{\{a,b+c\}, \{b,a+c\}, \{c,a+b\}, \{a+b,c\}, \{a+c,b\}, \{b+c,a\}\}$

ReplaceRepeated (//.)

```
>> a+b+c //. c->d
```

$a+b+d$

Simplification of logarithms:

```
>> logrules = {Log[x_ * y_] :>
   Log[x] + Log[y], Log[x_ ^ y_]
    :> y * Log[x]};
```

```
>> Log[a * (b * c)^ d ^ e * f]
   //. logrules
```

$\mathrm{Log}[a] + \mathrm{Log}[f] + (\mathrm{Log}[b] + \mathrm{Log}[c])\, d^e$

`ReplaceAll` just performs a single replacement:

```
>> Log[a * (b * c)^ d ^ e * f]
   /. logrules
```

$\mathrm{Log}[a] + \mathrm{Log}\left[f\,(bc)^{d^e}\right]$

RuleDelayed (:>)

```
>> Attributes[RuleDelayed]
```

{HoldRest, Protected, SequenceHold}

Rule (->)

```
>> a+b+c /. c->d
```

$a+b+d$

```
>> {x,x^2,y} /. x->3
```

$\{3,9,y\}$

Verbatim

```
>> _ /. Verbatim[_]->t
```

t

```
>> x /. Verbatim[_]->t
```

x

```
>> x /. _->t
```

t

XXV. Plot

Contents

Axis

Bottom

ColorData

ColorDataFunction

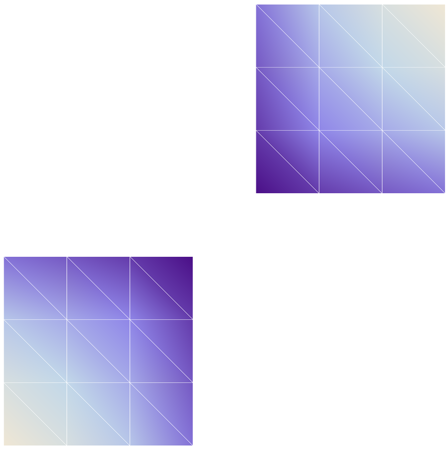
DensityPlot

`DensityPlot[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*ymin*`, `*ymax*`}]`
: plots a density plot of *f* with *x* ranging from *xmin* to *xmax* and *y* ranging from *ymin* to *ymax*.

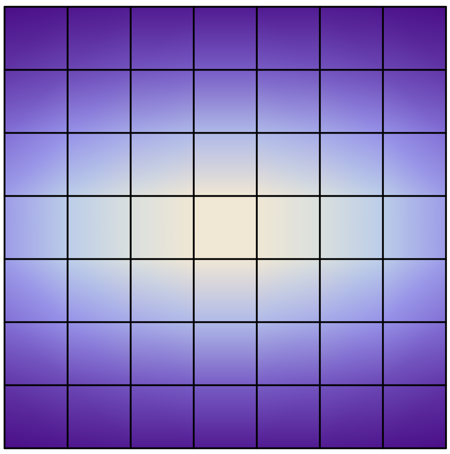
```
>> DensityPlot[x ^ 2 + 1 / y, {x
   , -1, 1}, {y, 1, 4}]
```

```
>> DensityPlot[1 / x, {x, 0, 1},
   {y, 0, 1}]
```

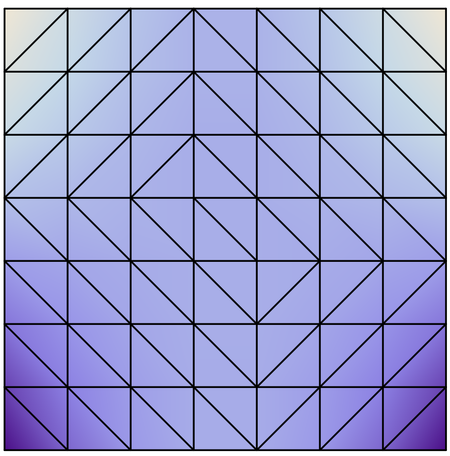
```
>> DensityPlot[Sqrt[x * y], {x, -1, 1}, {y, -1, 1}]
```



```
>> DensityPlot[1/(x^2 + y^2 + 1), {x, -1, 1}, {y, -2,2}, Mesh->Full]
```



```
>> DensityPlot[x^2 y, {x, -1, 1}, {y, -1, 1}, Mesh->All]
```



Full

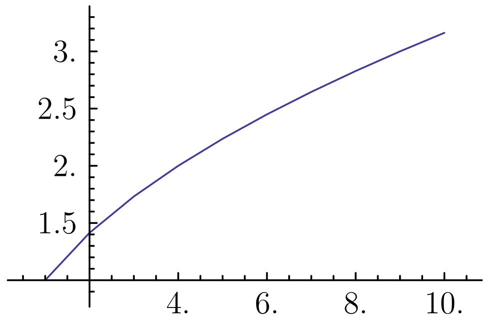
ListLinePlot

`ListLinePlot[{y_1, y_2, ...}]`
: plots a line through a list of y-values, assuming integer x-values 1, 2, 3, ...

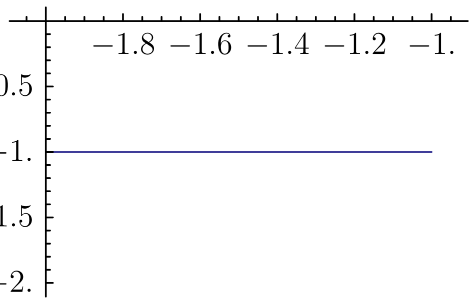
`ListLinePlot[{{x_1, y_1}, {x_2, y_2}, ...}]`
: plots a line through a list of x,y pairs.

`ListLinePlot[{list_1, list_2, ...}]`
: plots several lines.

```
>> ListLinePlot[Table[{n, n ^ 0.5}, {n, 10}]]
```




```
>> ListLinePlot[{{-2, -1}, {-1, -1}}]
```



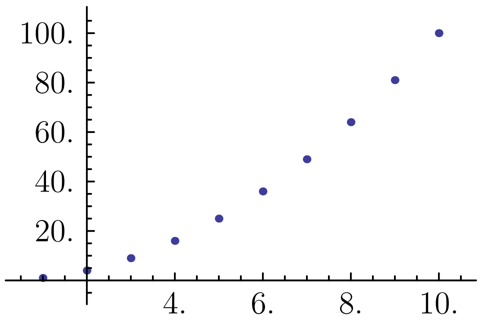
ListPlot

`ListPlot[{`*y_1*`, `*y_2*`, ...}]`
: plots a list of y-values, assuming integer x-values 1, 2, 3, ...

`ListPlot[{{`*x_1*`, `*y_1*`}, {`*x_2*`, `*y_2*`}, ...}]`
: plots a list of x,y pairs.

`ListPlot[{`*list_1*`, `*list_2*`, ...}]`
: plots a several lists of points.

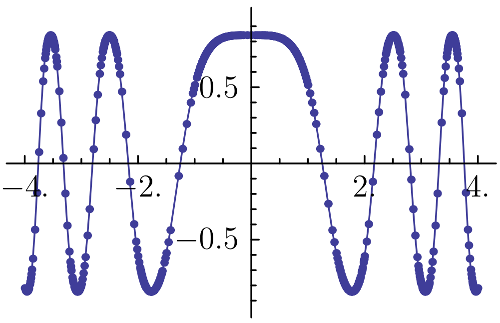
>> `ListPlot[Table[n ^ 2, {n, 10}]]`



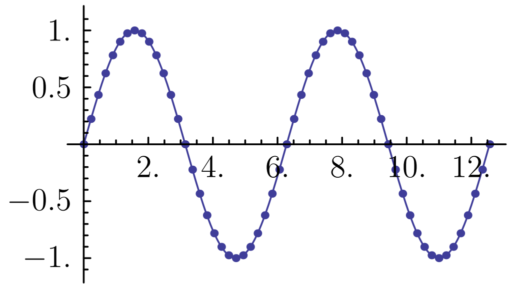
Mesh

`Mesh`
: is an option for `Plot` that specifies the mesh to be drawn. The default is `Mesh->None`.

>> `Plot[Sin[Cos[x^2]],{x,-4,4}, Mesh->All]`



>> `Plot[Sin[x], {x,0,4 Pi}, Mesh ->Full]`



»DensityPlot[Sin[x y], {x, -2, 2}, {y, -2, 2}, Mesh->Full] = -Graphics-

»Plot3D[Sin[x y], {x, -2, 2}, {y, -2, 2}, Mesh->Full] = -Graphics3D-

ParametricPlot

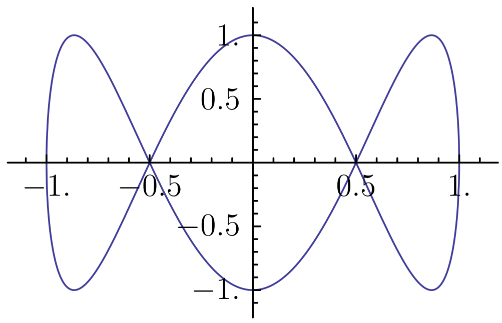
`ParametricPlot[{`*f_x*`, `*f_y*`}, {`*u*`, `*umin*`, `*umax*`}]`
: plots parametric function *f* with paramater *u* ranging from *umin* to *umax*.

`ParametricPlot[{{`*f_x*`, `*f_y*`}, {`*g_x*`, `*g_y*`}, ...}, {`*u*`, `*umin*`, `*umax*`}]`
: plots several parametric functions *f*, *g*, ...

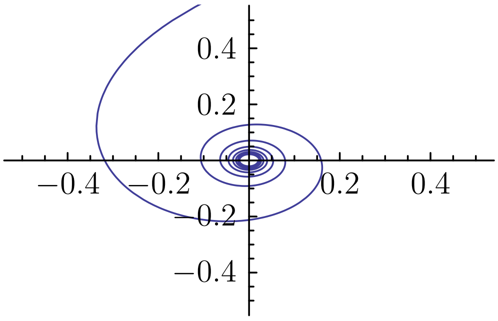
`ParametricPlot[{`*f_x*`, `*f_y*`}, {`*u*`, `*umin*`, `*umax*`}, {`*v*`, `*vmin*`, `*vmax*`}]`
: plots a parametric area.

`ParametricPlot[{{`*f_x*`, `*f_y*`}, {`*g_x*`, `*g_y*`}, ...}, {`*u*`, `*umin*`, `*umax*`}, {`*v*`, `*vmin*`, `*vmax*`}]`
: plots several parametric areas.

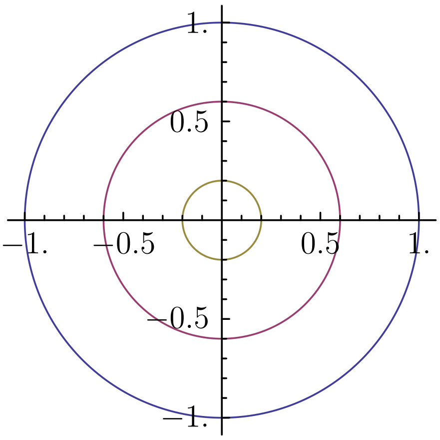
>> `ParametricPlot[{Sin[u], Cos[3 u]}, {u, 0, 2 Pi}]`



```
>> ParametricPlot[{Cos[u] / u,
   Sin[u] / u}, {u, 0, 50},
   PlotRange->0.5]
```




```
>> ParametricPlot[{{Sin[u], Cos[
   u]},{0.6 Sin[u], 0.6 Cos[u]},
    {0.2 Sin[u], 0.2 Cos[u]}}, {
   u, 0, 2 Pi}, PlotRange->1,
   AspectRatio->1]
```

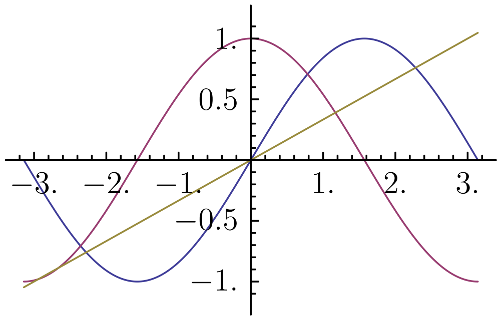


Plot

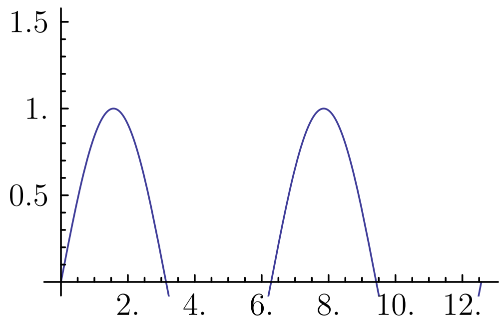
`Plot[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}]`
plots *f* with *x* ranging from *xmin* to *xmax*.

`Plot[{`*f1*`, `*f2*`, ...}, {`*x*`, `*xmin*`, `*xmax*`}]`
plots several functions *f1*, *f2*, ...

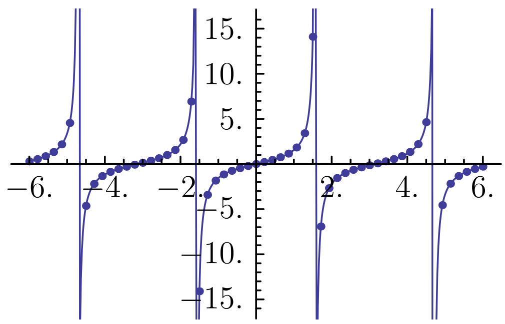
```
>> Plot[{Sin[x], Cos[x], x / 3},
    {x, -Pi, Pi}]
```



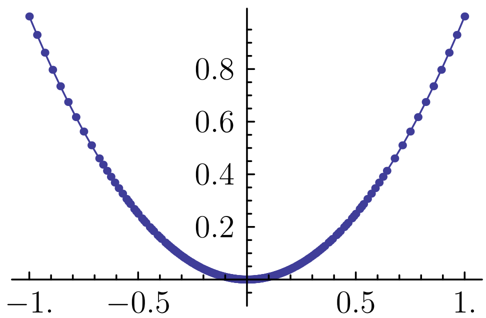

```
>> Plot[Sin[x], {x, 0, 4 Pi},
   PlotRange->{{0, 4 Pi}, {0,
   1.5}}]
```



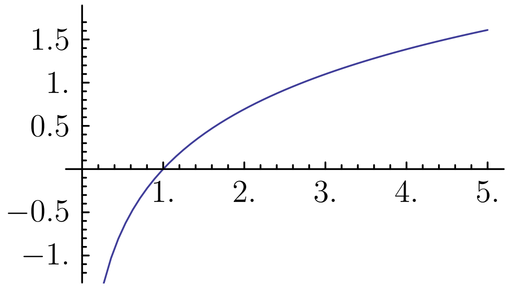

```
>> Plot[Tan[x], {x, -6, 6}, Mesh
   ->Full]
```



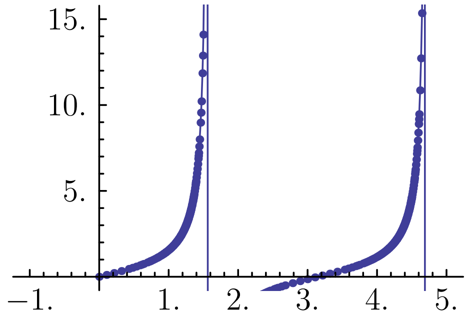

```
>> Plot[x^2, {x, -1, 1},
   MaxRecursion->5, Mesh->All]
```



```
>> Plot[Log[x], {x, 0, 5},
   MaxRecursion->0]
```

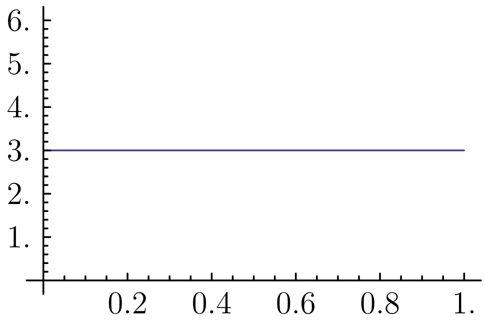



```
>> Plot[Tan[x], {x, 0, 6}, Mesh
   ->All, PlotRange->{{-1, 5},
   {0, 15}}, MaxRecursion->10]
```



A constant function:

```
>> Plot[3, {x, 0, 1}]
```

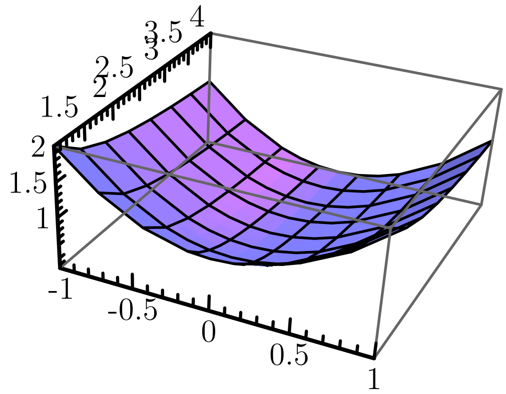


Plot3D

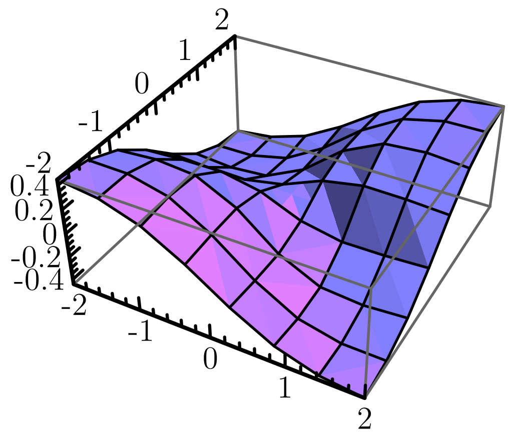
`Plot3D[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*ymin*`, `*ymax*`}]`

creates a three-dimensional plot of *f* with *x* ranging from *xmin* to *xmax* and *y* ranging from *ymin* to *ymax*.

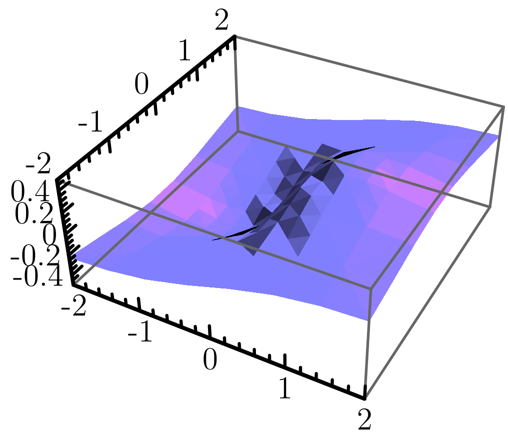
```
>> Plot3D[x ^ 2 + 1 / y, {x, -1,
   1}, {y, 1, 4}]
```



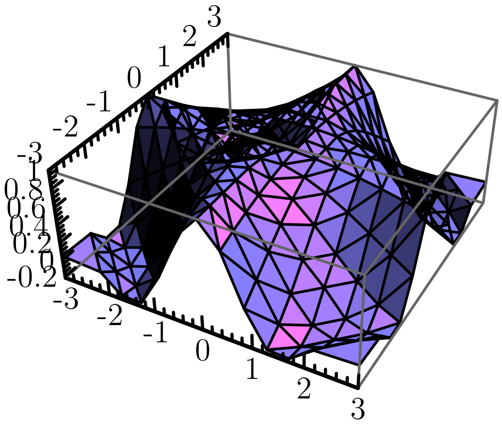

```
>> Plot3D[x y / (x ^ 2 + y ^ 2 +
   1), {x, -2, 2}, {y, -2, 2}]
```



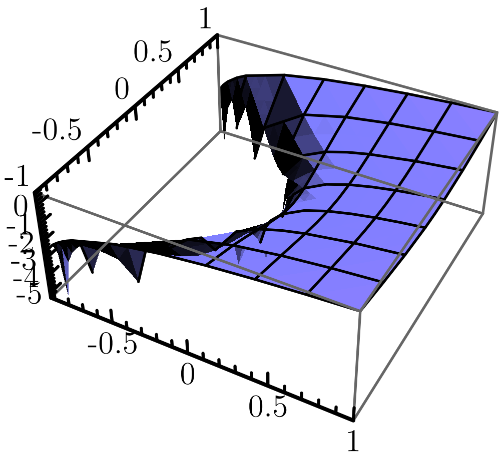

```
>> Plot3D[x / (x ^ 2 + y ^ 2 +
   1), {x, -2, 2}, {y, -2, 2},
   Mesh->None]
```



```
>> Plot3D[Sin[x y] /(x y), {x,
   -3, 3}, {y, -3, 3}, Mesh->All
   ]
```



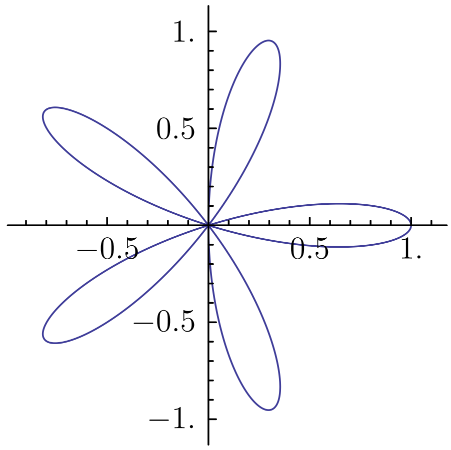
```
>> Plot3D[Log[x + y^2], {x, -1,
   1}, {y, -1, 1}]
```



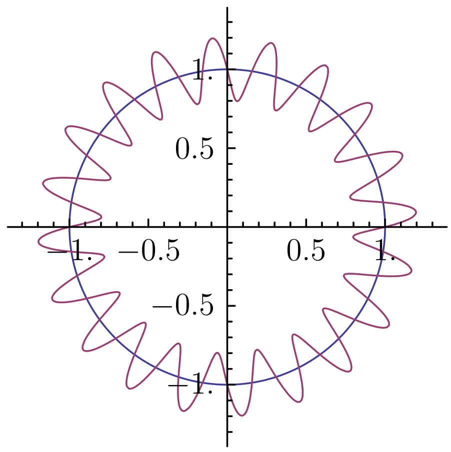
PolarPlot

`PolarPlot[`r`, {`t`, `$tmin$`, `$tmax$`}]`
: creates a polar plot of r with angle t ranging from $tmin$ to $tmax$.

```
>> PolarPlot[Cos[5t], {t, 0, Pi
   }]
```




```
>> PolarPlot[{1, 1 + Sin[20 t] /
    5}, {t, 0, 2 Pi}]
```



Top

XXVI. Physchemdata

Contents

ElementData

'ElementData["*name*", "*property*"]
: gives the value of the *property* for the chemical specified by *name*".

'ElementData[*n*, "*property*"]
: gives the value of the *property* for the *n*th chemical element".

```
>> ElementData[74]
```
Tungsten

```
>> ElementData["He", "
   AbsoluteBoilingPoint"]
```
4.22

```
>> ElementData["Carbon", "
   IonizationEnergies"]
```
{1 086.5, 2 352.6, 4 620.5
, 6 222.7, 37 831, 47 277.}

```
>> ElementData[16, "
   ElectronConfigurationString"]
```
[Ne] 3s2 3p4

```
>> ElementData[73, "
   ElectronConfiguration"]
```
{{2} , {2, 6} , {2, 6, 10} , {2,
6, 10, 14} , {2, 6, 3} , {2}}

The number of known elements:

```
>> Length[ElementData[All]]
```
118

Some properties are not appropriate for certain elements:

```
>> ElementData["He", "
   ElectroNegativity"]
```
Missing [NotApplicable]

Some data is missing:

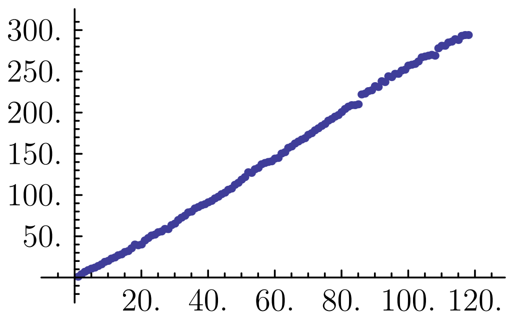
```
>> ElementData["Tc", "
   SpecificHeat"]
```
Missing [NotAvailable]

All the known properties:

```
>> ElementData["Properties"]
```

{Abbreviation,
AbsoluteBoilingPoint,
AbsoluteMeltingPoint,
AtomicNumber, AtomicRadius,
AtomicWeight, Block,
BoilingPoint, BrinellHardness,
BulkModulus, CovalentRadius,
CrustAbundance,
Density, DiscoveryYear,
ElectroNegativity,
ElectronAffinity,
ElectronConfiguration,
ElectronConfigurationString,
ElectronShellConfiguration,
FusionHeat, Group,
IonizationEnergies,
LiquidDensity, MeltingPoint,
MohsHardness, Name,
Period, PoissonRatio,
Series, ShearModulus,
SpecificHeat, StandardName,
ThermalConductivity,
VanDerWaalsRadius,
VaporizationHeat,
VickersHardness,
YoungModulus}

```
>> ListPlot[Table[ElementData[z,
    "AtomicWeight"], {z, 118}]]
```



XXVII. Randomnumbers

Contents

RandomComplex

`RandomComplex[{`*z_min*`, `*z_max*`}]`
: yields a pseudorandom complex number in the rectangle with complex corners *z_min* and *z_max*.

`RandomComplex[`*z_max*`]`
: yields a pseudorandom complex number in the rectangle with corners at the origin and at *z_max*.

`RandomComplex[]`
: yields a pseudorandom complex number with real and imaginary parts from 0 to 1.

`RandomComplex[`*range*`, `*n*`]`
: gives a list of *n* pseudorandom complex numbers.

`RandomComplex[`*range*`, {`*n1*`, `*n2*`, ...}]`
: gives a nested list of pseudorandom complex numbers.

>> **`RandomComplex[]`**

0.000118298630083 + 0.689576935039I

>> **`RandomComplex[{1+I, 5+5I}]`**

4.35934988731 + 1.10605928872I

>> **`RandomComplex[1+I, 5]`**

{0.697982552238 + 0.793745909˜˜34I, 0.850460832614 + 0.735˜˜9543912888I, 0.200147918936 + 0.182386194598I, 0.991308492˜˜785 + 0.761265796801I, 0.159˜˜292005222 + 0.200747170306I}

>> **`RandomComplex[{1+I, 2+2I}, {2, 2}]`**

{{1.96229852917 + 1.229323˜˜29535I, 1.49740922435 + 1.551˜˜059370231I}, {1.32155539407 + 1.82609753463I, 1.351500˜˜91348 + 1.40135292641I}}

RandomInteger

`RandomInteger[{`*min*`, `*max*`}]`
: yields a pseudorandom integer in the range from *min* to *max*.

`RandomInteger[`*max*`]`
: yields a pseudorandom integer in the range from 0 to *max*.

`RandomInteger[]`
: gives 0 or 1.

`RandomInteger[`*range*`, `*n*`]`
: gives a list of *n* pseudorandom integers.

`RandomInteger[`*range*`, {`*n1*`, `*n2*`, ...}]`
: gives a nested list of pseudorandom integers.

```
>> RandomInteger[{1, 5}]
   1

>> RandomInteger[100, {2, 3}] //
    TableForm
```

23	100	78
12	9	90

Calling `RandomInteger` changes `$RandomState`:

```
>> previousState = $RandomState;

>> RandomInteger[]
   1

>> $RandomState != previousState
   True
```

RandomReal

`RandomReal[{`*min*`, `*max*`}]`
: yields a pseudorandom real number in the range from *min* to *max*.

`RandomReal[`*max*`]`
: yields a pseudorandom real number in the range from 0 to *max*.

`RandomReal[]`
: yields a pseudorandom real number in the range from 0 to 1.

`RandomReal[`*range*`, `*n*`]`
: gives a list of *n* pseudorandom real numbers.

`RandomReal[`*range*`, {`*n1*`, `*n2*`, ...}]`
: gives a nested list of pseudorandom real numbers.

```
>> RandomReal[]
   0.508258480963

>> RandomReal[{1, 5}]
   1.91088246006
```

$RandomState

`$RandomState`
: is a long number representing the internal state of the pseudorandom number generator.

```
>> Mod[$RandomState, 10^100]
   8 299 456 349 730 229 433~
     ~236 924 321 927 662 092~
     ~665 751 762 344 566 074 417~
     ~112 252 317 067 146 058 075~
     ~371 996 701 917 967 419 950

>> IntegerLength[$RandomState]
   19 225
```

So far, it is not possible to assign values to `$RandomState`.

```
>> $RandomState = 42
   It is not possible to
     change the random state.
   42
```

Not even to its own value:

```
>> $RandomState = $RandomState;
   It is not possible to
     change the random state.
```

SeedRandom

`SeedRandom[`*n*`]`
: resets the pseudorandom generator with seed *n*.

`SeedRandom[]`
: uses the current date and time as seed.

`SeedRandom` can be used to get reproducible random numbers:

```
>> SeedRandom[42]

>> RandomInteger[100]
   64
```

```
>> RandomInteger[100]
   2
>> SeedRandom[42]
>> RandomInteger[100]
   64
>> RandomInteger[100]
   2
```

String seeds are supported as well:

```
>> SeedRandom["Mathics"]
>> RandomInteger[100]
   60
```

XXVIII. Recurrence

Contents

RSolve

RSolve[*eqn*, $a[n]$, *n*]
: solves a recurrence equation for the function $a[n]$.

```
>> RSolve[a[n] == a[n+1], a[n], n]
```

$\{\{a[n]\text{->}C[0]\}\}$

No boundary conditions gives two general paramaters:

```
>> RSolve[{a[n + 2] == a[n]}, a, n]
```

$\{\{a\text{->}(\text{Function}[\{n\}, C[0]+C[1]-1^n])\}\}$

One boundary condition:

```
>> RSolve[{a[n + 2] == a[n], a[0] == 1}, a, n]
```

$\{\{a\text{->}(\text{Function}[\{n\}, 1-C[1]+C[1]-1^n])\}\}$

Two boundary conditions:

```
>> RSolve[{a[n + 2] == a[n], a[0] == 1, a[1] == 4}, a, n]
```

$$\left\{\left\{a\text{->}\left(\text{Function}\left[\{n\}, \frac{5}{2}-\frac{3-1^n}{2}\right]\right)\right\}\right\}$$

XXIX. Specialfunctions

Contents

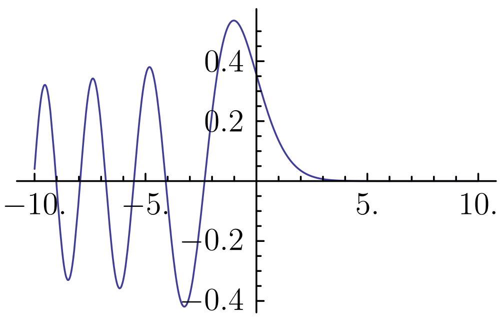
AiryAi

`AiryAi[x]`
returns the Airy function $\mathrm{Ai}(x)$.

```
>> AiryAi[0.5]
```
0.231693606480833349

```
>> AiryAi[0.5 + I]
```
0.157118446499986172 −
0.241039813840210768I

```
>> Plot[AiryAi[x], {x, -10, 10}]
```



AiryAiZero

`AiryAiZero[k]`
returns the kth zero of the Airy function $\mathrm{Ai}(z)$.

```
>> N[AiryAiZero[1]]
```
−2.33810741045976704

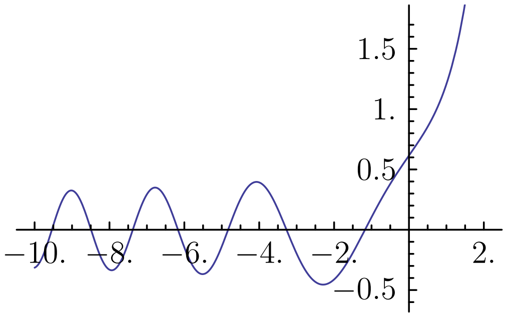
AiryBi

`AiryBi[x]`
returns the Airy function $\mathrm{Bi}(x)$.

```
>> AiryBi[0.5]
```
0.854277043103155493

```
>> AiryBi[0.5 + I]
```
0.688145273113482414 +
0.370815390737010831I

```
>> Plot[AiryBi[x], {x, -10, 2}]
```



AiryBiZero

`AiryBiZero[`*k*`]`
: returns the *k*th zero of the Airy function Bi(*z*).

```
>> N[AiryBiZero[1]]
```

−1.17371322270912792

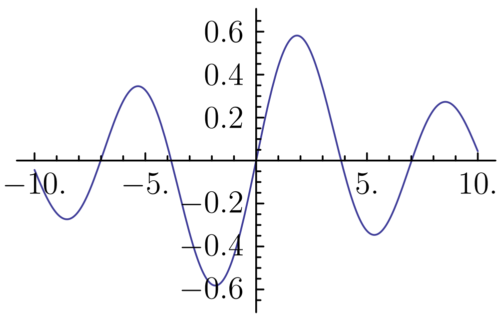
AngerJ

`AngerJ[`*n*`, `*z*`]`
: returns the Anger function J_*n*(*z*).

```
>> AngerJ[1.5, 3.5]
```

0.294478574459563408

```
>> Plot[AngerJ[1, x], {x, -10, 10}]
```



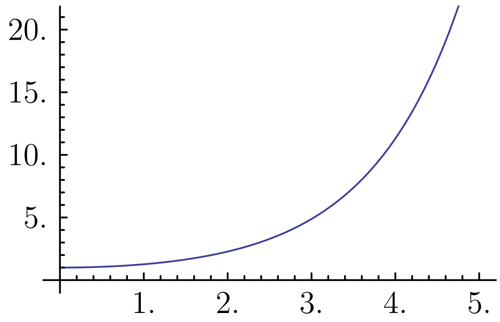
BesselI

`BesselI[`*n*`, `*z*`]`
: returns the modified Bessel function of the first kind I_*n*(*z*).

```
>> BesselI[1.5, 4]
```

8.17263323168659544

```
>> Plot[BesselI[0, x], {x, 0, 5}]
```



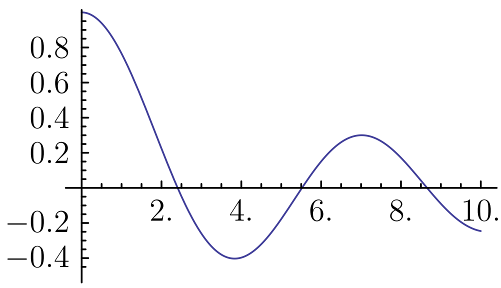
BesselJ

`BesselJ[`*n*`, `*z*`]`
: returns the Bessel function of the first kind J_*n*(*z*).

```
>> BesselJ[0, 5.2]
```

−0.11029043979098654

```
>> Plot[BesselJ[0, x], {x, 0, 10}]
```



BesselJZero

`BesselJZero[`n`, `k`]`
: returns the kth zero of the Bessel function of the first kind J_n(z).

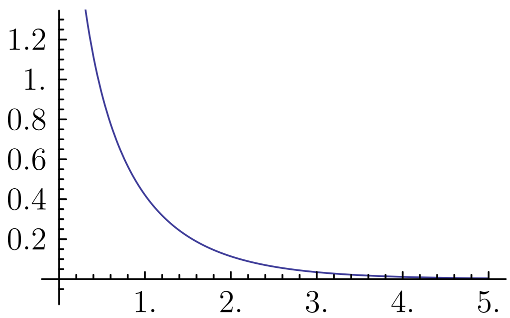
```
>> N[BesselJZero[0, 1]]
```
2.404825557695772 77

BesselK

`BesselK[`n`, `z`]`
: returns the modified Bessel function of the second kind K_n(z).

```
>> BesselK[1.5, 4]
```
0.014347030720760 0668

```
>> Plot[BesselK[0, x], {x, 0, 5}]
```

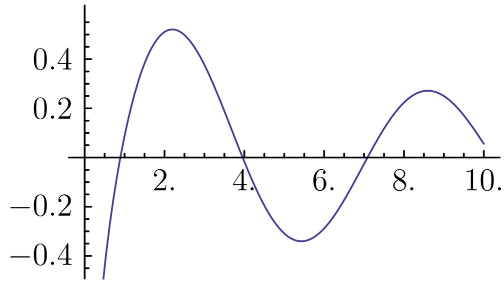


BesselY

`BesselY[`n`, `z`]`
: returns the Bessel function of the second kind Y_n(z).

```
>> BesselY[1.5, 4]
```
0.367112032460934155

```
>> Plot[BesselY[0, x], {x, 0, 10}]
```



BesselYZero

`BesselJZero[`n`, `k`]`
: returns the kth zero of the Bessel function of the second kind Y_n(z).

```
>> N[BesselYZero[0, 1]]
```
0.893576966279167522

ChebyshevT

`ChebyshevT[`n`, `x`]`
: returns the Chebyshev polynomial of the first kind T_n(x).

```
>> ChebyshevT[8, x]
```
$1 - 32x^2 + 160x^4 - 256x^6 + 128x^8$

```
>> ChebyshevT[1 - I, 0.5]
```
0.800143428851193116
+ 1.08198360440499884I

ChebyshevU

`ChebyshevU[`n`, `x`]`
: returns the Chebyshev polynomial of the second kind U_n(x).

```
>> ChebyshevU[8, x]
```
$1 - 40x^2 + 240x^4 - 448x^6 + 256x^8$

>> `ChebyshevU[1 - I, 0.5]`

1.60028685770238623 + 0.721322402936665892I

Erf

`Erf[`z`]`
returns the error function of z.

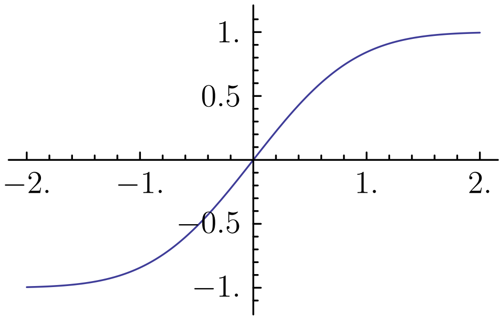
>> `Erf[1.0]`

0.842700792949714869

>> `Erf[0]`

0

>> `Plot[Erf[x], {x, -2, 2}]`



GegenbauerC

`GegenbauerC[`n`, `m`, `x`]`
returns the Generbauer polynomial C_n^(m)(x).

>> `GegenbauerC[6, 1, x]`

$-1 + 24x^2 - 80x^4 + 64x^6$

>> `GegenbauerC[4 - I, 1 + 2 I, 0.7]`

−3.26209595216525854 − 24.9739397455269944I

HankelH1

`HankelH1[`n`, `z`]`
returns the Hankel function of the first kind H_n^1 (z).

>> `HankelH1[1.5, 4]`

0.185285948354268953 + 0.367112032460934155I

HankelH2

`HankelH2[`n`, `z`]`
returns the Hankel function of the second kind H_n^2 (z).

>> `HankelH2[1.5, 4]`

0.185285948354268953 − 0.367112032460934155I

HermiteH

`ChebyshevU[`n`, `x`]`
returns the Hermite polynomial H_n(x).

>> `HermiteH[8, x]`

$1\,680 - 13\,440x^2 + 13\tilde{}\;\tilde{}440x^4 - 3\,584x^6 + 256x^8$

>> `HermiteH[3, 1 + I]`

$-28 + 4I$

>> `HermiteH[4.2, 2]`

77.5290837369752225

JacobiP

`JacobiP[`n`, `a`, `b`, `x`]`
returns the Jacobi polynomial P_n^(a,b)(x).

`>> JacobiP[1, a, b, z]`

$$\frac{a}{2} - \frac{b}{2} + z\left(1 + \frac{a}{2} + \frac{b}{2}\right)$$

`>> JacobiP[3.5 + I, 3, 2, 4 - I]`

$1\,410.02011674512937 + 5\,797.29855312717469 I$

KelvinBei

`KelvinBei[z]`
: returns the Kelvin function bei(z).

`KelvinBei[n, z]`
: returns the Kelvin function bei_n(z).

`>> KelvinBei[0.5]`

0.0624932183821994586

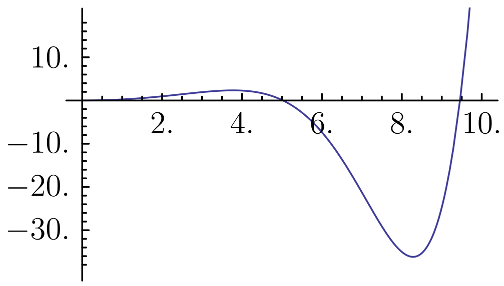
`>> KelvinBei[1.5 + I]`

$0.326323348699806294 + 0.755605578610892281 I$

`>> KelvinBei[0.5, 0.25]`

0.370152900194021013

`>> Plot[KelvinBei[x], {x, 0, 10}]`



KelvinBer

`KelvinBer[z]`
: returns the Kelvin function ber(z).

`KelvinBer[n, z]`
: returns the Kelvin function ber_n(z).

`>> KelvinBer[0.5]`

0.999023463990838256

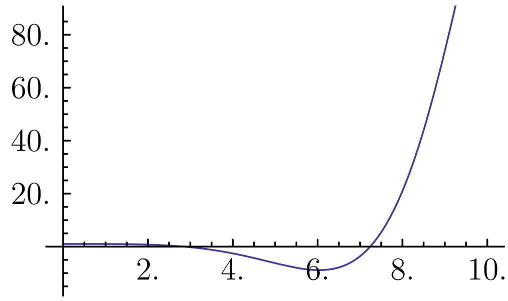
`>> KelvinBer[1.5 + I]`

$1.11620420872233787 - 0.117944469093970067 I$

`>> KelvinBer[0.5, 0.25]`

0.148824330530639942

`>> Plot[KelvinBer[x], {x, 0, 10}]`



KelvinKei

`KelvinKei[z]`
: returns the Kelvin function kei(z).

`KelvinKei[n, z]`
: returns the Kelvin function kei_n(z).

`>> KelvinKei[0.5]`

−0.671581695094367603

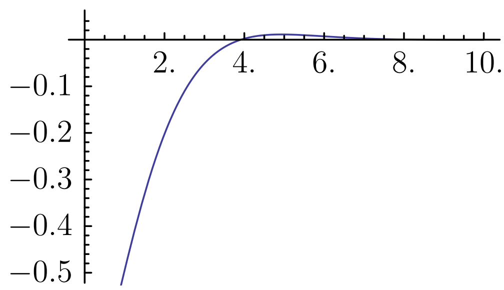
`>> KelvinKei[1.5 + I]`

$-0.248993863536003923 + 0.303326291875385478 I$

`>> KelvinKei[0.5, 0.25]`

−2.05169683896315934

`>> Plot[KelvinKei[x], {x, 0, 10}]`



KelvinKer

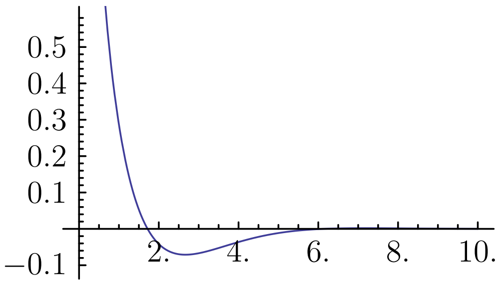
`KelvinKer[z]`
returns the Kelvin function ker(z).
`KelvinKer[n, z]`
returns the Kelvin function ker_n(z).

>> `KelvinKer[0.5]`
0.855905872118634214

>> `KelvinKer[1.5 + I]`
$-0.167162242027385125 - 0.184403720314419905I$

>> `KelvinKer[0.5, 0.25]`
0.450022838747182502

>> `Plot[KelvinKer[x], {x, 0, 10}]`



LaguerreL

`LaguerreL[n, x]`
returns the Laguerre polynomial L_n(x).
`LaguerreL[n, a, x]`
returns the generalised Laguerre polynomial L^a_n(x).

>> `LaguerreL[8, x]`
$$1-8x+14x^2-\frac{28x^3}{3}+\frac{35x^4}{12}-\frac{7x^5}{15}+\frac{7x^6}{180}-\frac{x^7}{630}+\frac{x^8}{40320}$$

>> `LaguerreL[3/2, 1.7]`
−0.94713399725341823

>> `LaguerreL[5, 2, x]`
$$21-35x+\frac{35x^2}{2}-\frac{7x^3}{2}+\frac{7x^4}{24}-\frac{x^5}{120}$$

LegendreP

`LegendreP[n, x]`
returns the Legendre polynomial P_n(x).
`LegendreP[n, m, x]`
returns the associated Legendre polynomial P^m_n(x).

>> `LegendreP[4, x]`
$$\frac{3}{8}-\frac{15x^2}{4}+\frac{35x^4}{8}$$

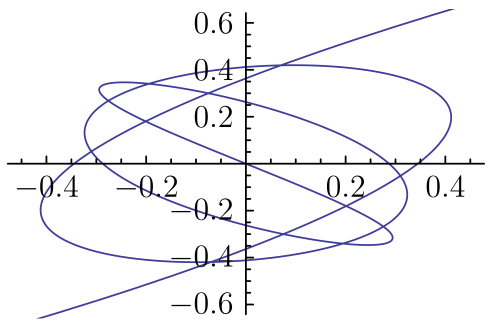
>> `LegendreP[5/2, 1.5]`
4.17761913892745532

>> `LegendreP[1.75, 1.4, 0.53]`
−1.32619280980662145

>> `LegendreP[1.6, 3.1, 1.5]`
$-0.303998161489593441 - 1.91936885256334894I$

`LegendreP` can be used to draw generalized Lissajous figures:

>> `ParametricPlot[ {LegendreP[7, x], LegendreP[5, x]}, {x, -1, 1}]`



LegendreQ

`LegendreQ[`n`, `x`]`
: returns the Legendre function of the second kind Q_n(x).

`LegendreQ[`n`, `m`, `x`]`
: returns the associated Legendre function of the second Q^m_n(x).

```
>> LegendreQ[5/2, 1.5]
```
0.0362109671796812979 − 6.56218879817530572I

```
>> LegendreQ[1.75, 1.4, 0.53]
```
2.05498907857609114

```
>> LegendreQ[1.6, 3.1, 1.5]
```
−1.71931290970694153 − 7.70273279782676974I

ProductLog

`ProductLog[`z`]`
: returns the value of the Lambert W function at z.

The defining equation:

```
>> z == ProductLog[z] * E ^
   ProductLog[z]
```
True

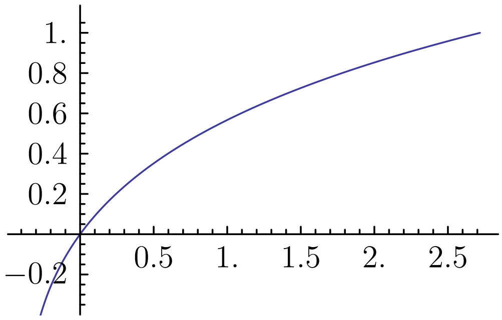
Some special values:

```
>> ProductLog[0]
```
0

```
>> ProductLog[E]
```
1

The graph of `ProductLog`:

```
>> Plot[ProductLog[x], {x, -1/E,
   E}]
```



SphericalHarmonicY

`SphericalHarmonicY[`l`, `m`, `$theta$`, `phi`]`
: returns the spherical harmonic functin Y_l^m(theta, phi).

```
>> SphericalHarmonicY[3/4, 0.5,
   Pi/5, Pi/3]
```
0.254247340352667373 + 0.146789770393358909I

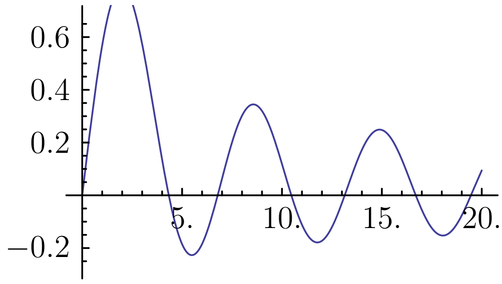
```
>> SphericalHarmonicY[3, 1,
   theta, phi]
```
$$\frac{\sqrt{21}\left(1-5\mathrm{Cos}\,[\mathrm{theta}]^{2}\right)E^{I\mathrm{phi}}\mathrm{Sin}\,[\mathrm{theta}]}{8\sqrt{\mathrm{Pi}}}$$

StruveH

`StruveH[`n`, `z`]`
: returns the Struve function H_n(z).

```
>> StruveH[1.5, 3.5]
```
1.13192125271801312

>> `Plot[StruveH[0, x], {x, 0, 20}]`



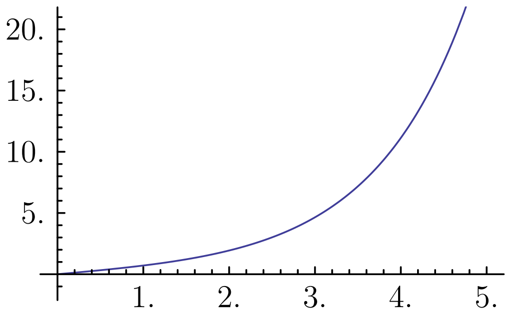
StruveL

`StruveL[n, z]`
: returns the modified Struve function L_n(z).

>> `StruveL[1.5, 3.5]`

4.41126360920433996

>> `Plot[StruveL[0, x], {x, 0, 5}]`



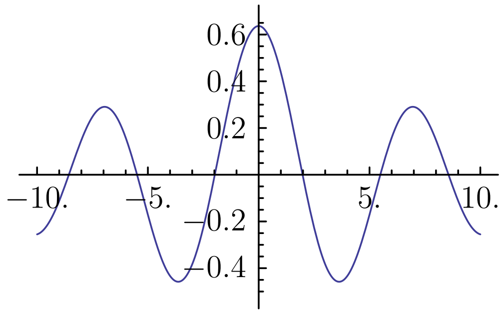
WeberE

`WeberE[n, z]`
: returns the Weber function E_n(z).

>> `WeberE[1.5, 3.5]`

−0.397256259210030809

>> `Plot[WeberE[1, x], {x, -10, 10}]`



Zeta

`Zeta[z]`
: returns the Riemann zeta function of z.

>> `Zeta[2]`

$$\frac{\mathrm{Pi}^2}{6}$$

>> `Zeta[-2.5 + I]`

0.0235936105863796486 + 0.00140779960583837704*I*

XXX. Scoping

Contents

Begin

`Begin[`*context*`]`
temporarily sets the current context to *context*.

```
>> Begin["test`"]
test`

>> {$Context, $ContextPath}
{test`, {Global`, System`}}

>> Context[newsymbol]
test`

>> End[]
test`

>> End[]
No previous context defined.
Global`
```

BeginPackage

`BeginPackage[`*context*`]`
starts the package given by *context*.

The *context* argument must be a valid context name. `BeginPackage` changes the values of `$Context` and `$ContextPath`, setting the current context to *context*.

```
>> {$Context, $ContextPath}
{Global`, {Global`, System`}}

>> BeginPackage["test`"]
test`

>> {$Context, $ContextPath}
{test`, {test`, System`}}

>> Context[newsymbol]
test`

>> EndPackage[]

>> {$Context, $ContextPath}
{Global`, {test`, Global`, System`}}

>> EndPackage[]
No previous context defined.
```

Block

`Block[{vars}, expr]`
: temporarily stores the definitions of certain variables, evaluates *expr* with reset values and restores the original definitions afterwards.

`Block[{x=x0, y=y0, ...}, expr]`
: assigns initial values to the reset variables.

```
>> n = 10
   10
>> Block[{n = 5}, n ^ 2]
   25
>> n
   10
```

Values assigned to block variables are evaluated at the beginning of the block. Keep in mind that the result of `Block` is evaluated again, so a returned block variable will get its original value.

```
>> Block[{x = n+2, n}, {x, n}]
   {12, 10}
```

If the variable specification is not of the described form, an error message is raised:

```
>> Block[{x + y}, x]
```

Local variable specification contains $x + y$, which is not a symbol or an assignment to a symbol.

x

Variable names may not appear more than once:

```
>> Block[{x, x}, x]
```

Duplicate local variable x found in local variable specification.

x

Context

`Context[symbol]`
: yields the name of the context where *symbol* is defined in.

`Context[]`
: returns the value of `$Context`.

```
>> Context[a]
   Global`
>> Context[b`c]
   b`
>> Context[Sin] // InputForm
   "System`"
>> InputForm[Context[]]
   "Global`"
```

$ContextPath

`$ContextPath`
: is the search path for contexts.

```
>> $ContextPath // InputForm
   {"Global`", "System`"}
```

System`Private`$ContextPathStack

`System`Private`$ContextPathStack`
: tracks the values of `$ContextPath` saved by `Begin` and `BeginPackage`.

System`Private`$ContextStack

`System`Private`$ContextStack`
: tracks the values of `$Context` saved by `Begin` and `BeginPackage`.

$Context

`$Context`
: is the current context.

```
>> $Context
```
Global‘

Contexts

`Contexts[]`
: yields a list of all contexts.

```
>> x = 5;
```

```
>> Contexts[] // InputForm
```
{"Global‘", "System‘", "System‘Convert‘JSONDump‘", "System‘Convert‘TableDump‘", "System‘Convert‘TextDump‘", "System‘Private‘"}

End

`End[]`
: ends a context started by `Begin`.

EndPackage

`EndPackage[]`
: marks the end of a package, undoing a previous `BeginPackage`.

After `EndPackage`, the values of `$Context` and `$ContextPath` at the time of the `BeginPackage` call are restored, with the new package's context prepended to $ContextPath.

Module

`Module[{`*vars*`}, `*expr*`]`
: localizes variables by giving them a temporary name of the form `name$number`, where number is the current value of `$ModuleNumber`. Each time a module is evaluated, `$ModuleNumber` is incremented.

```
>> x = 10;
```

```
>> Module[{x=x}, x=x+1; x]
```
11

```
>> x
```
10

```
>> t === Module[{t}, t]
```
False

Initial values are evaluated immediately:

```
>> Module[{t=x}, x = x + 1; t]
```
10

```
>> x
```
11

Variables inside other scoping constructs are not affected by the renaming of `Module`:

```
>> Module[{a}, Block[{a}, a]]
```
a

```
>> Module[{a}, Block[{}, a]]
```
a$5

$ModuleNumber

`$ModuleNumber`
: is the current "serial number" to be used for local module variables.

```
>> Unprotect[$ModuleNumber]
```

```
>> $ModuleNumber = 20;
```

```
>> Module[{x}, x]
```
x$20

```
>> $ModuleNumber = x;
```

Cannot set $ModuleNumber
to x; value must be
a positive integer.

XXXI. Strings

Contents

CharacterRange

```
>> CharacterRange["a", "e"]
```

{a, b, c, d, e}

```
>> CharacterRange["b", "a"]
```

{}

Characters

```
>> Characters["abc"]
```

{a, b, c}

FromCharacterCode

`FromCharacterCode[n]`
returns the character corresponding to character code *n*.
`FromCharacterCode[{n1, n2, ...}]`
returns a string with characters corresponding to *n_i*.
`FromCharacterCode[{{n11, n12, ...}, {n21, n22, ...}, ...}]`
returns a list of strings.

```
>> FromCharacterCode[100]
```

d

```
>> FromCharacterCode[{100, 101, 102}]
```

def

```
>> ToCharacterCode[%]
```

{100, 101, 102}

```
>> FromCharacterCode[{{97, 98, 99}, {100, 101, 102}}]
```

{abc, def}

```
>> ToCharacterCode["abc 123"] // FromCharacterCode
```

abc 123

StringDrop

StringDrop["string",n] gives "string" with the first n characters dropped. StringDrop["string",-n] gives "string" with the last n characters dropped. StringDrop["string",{n}] gives "string" with the character n dropped String-Drop["string",{m,n}] gives "string" with the characters m through n dropped.

```
>> StringDrop["abcde", 2]
```

cde

```
>> StringDrop["abcde", -2]
```

abc

```
>> StringDrop["abcde", {2}]
```
acde

```
>> StringDrop["abcde", {2,3}]
```
ade

```
>> StringDrop["abcd",{3,2}]
```
abcd

```
>> StringDrop["abcd",0]
```
abcd

StringJoin (<>)

```
>> StringJoin["a", "b", "c"]
```
abc

```
>> "a" <> "b" <> "c" //
   InputForm
```
"abc"

`StringJoin` flattens lists out:

```
>> StringJoin[{"a", "b"}] //
   InputForm
```
"ab"

```
>> Print[StringJoin[{"Hello", "
   ", {"world"}}, "!"]]
```
Hello world!

StringLength

`StringLength` gives the length of a string.

```
>> StringLength["abc"]
```
3

`StringLength` is listable:

```
>> StringLength[{"a", "bc"}]
```
$\{1,2\}$

```
>> StringLength[x]
```
String expected.

StringLength $[x]$

StringQ

`StringQ[`*expr*`]`
: returns `True` if *expr* is a `String` or `False` otherwise.

```
>> StringQ["abc"]
```
True

```
>> StringQ[1.5]
```
False

```
>> Select[{"12", 1, 3, 5, "yz",
   x, y}, StringQ]
```
$\{12, yz\}$

StringReplace

`StringReplace["`*string*`", `*s*`->`*sp*`]` or `StringReplace["`*string*`", {`*s1*`->`*sp1*`, `*s2*`->`*sp2*`}]`
: replace the string *si* by *spi* for all occurances in "*string*".

`StringReplace["`*string*`", `*srules*`, `*n*`]`
: only perform the first *n* replacements.

`StringReplace[{"`*string1*`‘ ‘,"`*string2*`", ...}, srules]`
: perform replacements on a list of strings

StringReplace replaces all occurances of one substring with another:

```
>> StringReplace["
   xyxyxyyyxxxyyxy", "xy" -> "A
   "]
```
AAAyyxxAyA

Multiple replacements can be supplied:

```
>> StringReplace["
   xyzwxyzwxxyzxyzw", {"xyz" ->
   "A", "w" -> "BCD"}]
```
ABCDABCDxAABCD

Only replace the first 2 occurances:

```
>> StringReplace["
   xyxyxyyyxxxyyxy", "xy" -> "A
   ", 2]
```
AAxyyyxxxyyxy

StringReplace acts on lists of strings too:
```
>> StringReplace[{"xyxyxxy", "
   yxyxyxxxyyxy"}, "xy" -> "A"]
```
{AAxA, yAAxxAyA}

StringSplit

```
>> StringSplit["abc,123", ","]
```
{abc, 123}

```
>> StringSplit["abc 123"]
```
{abc, 123}

```
>> StringSplit["abc,123.456",
   {",", "."}]
```
{abc, 123, 456}

StringTake

StringTake["string",n] gives the first n characters in "string" StringTake["string",-n] gives the last n characters in "string" StringTake["string",{n}] gives the n-esim character in "string" StringTake["string",{m,n}] gives characters m through n in "string"
```
>> StringTake["abcde", 2]
```
ab

```
>> StringTake["abcde", 0]
```

```
>> StringTake["abcde", -2]
```
de

```
>> StringTake["abcde", {2}]
```
b

```
>> StringTake["abcd", {2,3}]
```
bc

```
>> StringTake["abcd", {3,2}]
```

String

`String` is the head of strings.
```
>> Head["abc"]
```
String

```
>> "abc"
```
abc

Use `InputForm` to display quotes around strings:
```
>> InputForm["abc"]
```
"abc"

`FullForm` also displays quotes:
```
>> FullForm["abc" + 2]
```
Plus [2, "abc"]

ToCharacterCode

`ToCharacterCode[‘‘string’]’`
: converts the string to a list of integer character codes.

`ToCharacterCode[{‘‘string1’, "string2", ...}]’`
: converts a list of strings to character codes.

```
>> ToCharacterCode["abc"]
```
{97, 98, 99}

```
>> FromCharacterCode[%]
```
abc

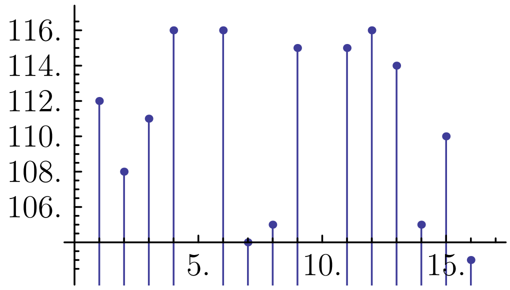
```
>> ToCharacterCode["\[Alpha]\[
   Beta]\[Gamma]"]
```
{945, 946, 947}

```
>> ToCharacterCode[{"ab", "c"}]
```
{{97, 98} , {99}}

```
>> ToCharacterCode[{"ab", x}]
```
String or list of strings expected at position 1 in ToCharacterCode [{ab, x}] .

ToCharacterCode [{ab, x}]

```
>> ListPlot[ToCharacterCode["
   plot this string"], Filling
   -> Axis]
```



ToExpression

`ToExpression[`*input*`]`
: inteprets a given string as Mathics input.

`ToExpression[`*input*`, `*form*`]`
: reads the given input in the specified form.

`ToExpression[`*input*`, `*form*`, `*h*`]`
: applies the head *h* to the expression before evaluating it.

```
>> ToExpression["1 + 2"]
```
3

```
>> ToExpression["{2, 3, 1}",
   InputForm, Max]
```
3

ToString

```
>> ToString[2]
```
2

```
>> ToString[2] // InputForm
```
"2"

```
>> ToString[a+b]
```
a + b

```
>> "U" <> 2
```
String expected.

U<>2

```
>> "U" <> ToString[2]
```
U2

XXXII. Structure

Contents

Apply (@@)

`Apply[`*f*`, `*expr*`]` or *f* `@@` *expr*
replaces the head of *expr* with *f*.
`Apply[`*f*`, `*expr*`, `*levelspec*`]`
applies *f* on the parts specified by *levelspec*.

```
>> f @@ {1, 2, 3}
```
$f[1,2,3]$

```
>> Plus @@ {1, 2, 3}
```
6

The head of *expr* need not be `List`:
```
>> f @@ (a + b + c)
```
$f[a,b,c]$

Apply on level 1:
```
>> Apply[f, {a + b, g[c, d, e * f], 3}, {1}]
```
$\{f[a,b], f[c,d,ef], 3\}$

The default level is 0:
```
>> Apply[f, {a, b, c}, {0}]
```
$f[a,b,c]$

Range of levels, including negative level (counting from bottom):
```
>> Apply[f, {{{{{a}}}}}, {2, -3}]
```
$\{\{f[f[\{a\}]]\}\}$

Convert all operations to lists:
```
>> Apply[List, a + b * c ^ e * f[g], {0, Infinity}]
```
$\{a, \{b, \{g\}, \{c,e\}\}\}$

ApplyLevel (@@@)

`ApplyLevel[`*f*`, `*expr*`]` or *f* `@@@` *expr*
is equivalent to `Apply[`*f*`, `*expr*`, {1}]`.

```
>> f @@@ {{a, b}, {c, d}}
```
$\{f[a,b], f[c,d]\}$

AtomQ

```
>> AtomQ[x]
```
True

```
>> AtomQ[1.2]
   True

>> AtomQ[2 + I]
   True

>> AtomQ[2 / 3]
   True

>> AtomQ[x + y]
   False
```

Depth

`Depth[`*expr*`]`
: gives the depth of *expr*

The depth of an expression is defined as one plus the maximum number of `Part` indices required to reach any part of *expr*, except for heads.

```
>> Depth[x]
   1

>> Depth[x + y]
   2

>> Depth[{{{{x}}}}]
   5
```

Complex numbers are atomic, and hence have depth 1:

```
>> Depth[1 + 2 I]
   1
```

`Depth` ignores heads:

```
>> Depth[f[a, b][c]]
   2
```

Flatten

`Flatten[`*expr*`]`
: flattens out nested lists in *expr*.

`Flatten[`*expr*`, `*n*`]`
: stops flattening at level *n*.

`Flatten[`*expr*`, `*n*`, `*h*`]`
: flattens expressions with head *h* instead of `List`.

```
>> Flatten[{{a, b}, {c, {d}, e}, {f, {g, h}}}]
```
$\{a,b,c,d,e,f,g,h\}$

```
>> Flatten[{{a, b}, {c, {e}, e}, {f, {g, h}}}, 1]
```
$\{a,b,c,\{e\},e,f,\{g,h\}\}$

```
>> Flatten[f[a, f[b, f[c, d]], e], Infinity, f]
```
$f[a,b,c,d,e]$

```
>> Flatten[{{a, b}, {c, d}}, {{2}, {1}}]
```
$\{\{a,c\},\{b,d\}\}$

```
>> Flatten[{{a, b}, {c, d}}, {{1, 2}}]
```
$\{a,b,c,d\}$

Flatten also works in irregularly shaped arrays

```
>> Flatten[{{1, 2, 3}, {4}, {6, 7}, {8, 9, 10}}, {{2}, {1}}]
```
$\{\{1,4,6,8\},\{2,7,9\},\{3,10\}\}$

FreeQ

```
>> FreeQ[y, x]
   True

>> FreeQ[a+b+c, a+b]
   False

>> FreeQ[{1, 2, a^(a+b)}, Plus]
   False
```

```
>> FreeQ[a+b, x_+y_+z_]
```
True

```
>> FreeQ[a+b+c, x_+y_+z_]
```
False

Head

```
>> Head[a * b]
```
Times

```
>> Head[6]
```
Integer

```
>> Head[x]
```
Symbol

Map (/@)

`Map[`*f*`, `*expr*`]` or *f* `/@` *expr*
: applies *f* to each part on the first level of *expr*.

`Map[`*f*`, `*expr*`, `*levelspec*`]`
: applies *f* to each level specified by *levelspec* of *expr*.

```
>> f /@ {1, 2, 3}
```
$\{f[1], f[2], f[3]\}$

```
>> #^2& /@ {1, 2, 3, 4}
```
$\{1, 4, 9, 16\}$

Map *f* on the second level:

```
>> Map[f, {{a, b}, {c, d, e}},
   {2}]
```
$\{\{f[a], f[b]\}, \{f[c], f[d], f[e]\}\}$

Include heads:

```
>> Map[f, a + b + c, Heads->True
   ]
```
$f[\text{Plus}]\left[f[a], f[b], f[c]\right]$

MapIndexed

`MapIndexed[`*f*`, `*expr*`]`
: applies *f* to each part on the first level of *expr*, including the part positions in the call to *f*.

`MapIndexed[`*f*`, `*expr*`, `*levelspec*`]`
: applies *f* to each level specified by *levelspec* of *expr*.

```
>> MapIndexed[f, {a, b, c}]
```
$\{f[a, \{1\}], f[b, \{2\}], f[c, \{3\}]\}$

Include heads (index 0):

```
>> MapIndexed[f, {a, b, c},
   Heads->True]
```
$f[\text{List}, \{0\}]\left[f[a, \{1\}], f[b, \{2\}], f[c, \{3\}]\right]$

Map on levels 0 through 1 (outer expression gets index {}):

```
>> MapIndexed[f, a + b + c * d,
   {0, 1}]
```
$f\left[f[a, \{1\}] + f[b, \{2\}] + f[cd, \{3\}], \{\}\right]$

Get the positions of atoms in an expression (convert operations to `List` first to disable `Listable` functions):

```
>> expr = a + b * f[g] * c ^ e;
```

```
>> listified = Apply[List, expr,
    {0, Infinity}];
```

```
>> MapIndexed[#2 &, listified,
   {-1}]
```
$\{\{1\}, \{\{2,1\}, \{\{2,2,1\}\}, \{\{2,3,1\}, \{2,3,2\}\}\}\}$

Replace the heads with their positions, too:

```
>> MapIndexed[#2 &, listified,
   {-1}, Heads -> True]
```
$\{0\}\left[\{1\}, \{2,0\}\left[\{2,1\}, \{2,2,0\}\left[\{2,2,1\}\right], \{2,3,0\}\left[\{2,3,1\}, \{2,3,2\}\right]\right]\right]$

The positions are given in the same format as used by `Extract`. Thus, mapping `Extract` on the indices given by `MapIndexed` re-constructs the original expression:

>> `MapIndexed[Extract[expr, #2] &, listified, {-1}, Heads -> True]`

$a + bf[g]c^e$

Null

`Null` is the implicit result of expressions that do not yield a result:

>> `FullForm[a:=b]`

Null

It is not displayed in StandardForm,

>> `a:=b`

in contrast to the empty string:

>> `""`

(watch the empty line).

Operate

`Operate[`*p*`, `*expr*`]`
: applies *p* to the head of *expr*.

`Operate[`*p*`, `*expr*`, `*n*`]`
: applies *p* to the *n*th head of *expr*.

>> `Operate[p, f[a, b]]`

$p[f][a, b]$

The default value of *n* is 1:

>> `Operate[p, f[a, b], 1]`

$p[f][a, b]$

With *n*=0, `Operate` acts like `Apply`:

>> `Operate[p, f[a][b][c], 0]`

$p[f[a][b][c]]$

OrderedQ

>> `OrderedQ[a, b]`

True

>> `OrderedQ[b, a]`

False

PatternsOrderedQ

>> `PatternsOrderedQ[x__, x_]`

False

>> `PatternsOrderedQ[x_, x__]`

True

>> `PatternsOrderedQ[b, a]`

True

Sort

`Sort[`*list*`]`
: sorts *list* (or the leaves of any other expression) according to canonical ordering.

`Sort[`*list*`, `*p*`]`
: sorts using *p* to determine the order of two elements.

>> `Sort[{4, 1.0, a, 3+I}]`

$\{1., 3 + I, 4, a\}$

Sort uses `OrderedQ` to determine ordering by default. You can sort patterns according to their precedence using `PatternsOrderedQ`:

>> `Sort[{items___, item_, OptionsPattern[], item_symbol, item_?test}, PatternsOrderedQ]`

{item_symbol, item_? test, item_, items___, OptionsPattern []}

When sorting patterns, values of atoms do not matter:

```
>> Sort[{a, b/;t}, PatternsOrderedQ]
```
$\{b/;t, a\}$

```
>> Sort[{2+c_, 1+b__}, PatternsOrderedQ]
```
$\{2 + \text{c\_}, 1 + \text{b\_\_}\}$

```
>> Sort[{x_ + n_*y_, x_ + y_}, PatternsOrderedQ]
```
$\{\text{x\_} + \text{n\_y\_}, \text{x\_} + \text{y\_}\}$

SymbolName

```
>> SymbolName[x] // InputForm
```
"x"

SymbolQ

```
>> SymbolQ[a]
```
True

```
>> SymbolQ[1]
```
False

```
>> SymbolQ[a + b]
```
False

Symbol

`Symbol` is the head of symbols.

```
>> Head[x]
```
Symbol

You can use `Symbol` to create symbols from strings:

```
>> Symbol["x"] + Symbol["x"]
```
$2x$

Thread

`Thread[`*f*`[`*args*`]]`
: threads *f* over any lists that appear in *args*.

`Thread[`*f*`[`*args*`], `*h*`]`
: threads over any parts with head *h*.

```
>> Thread[f[{a, b, c}]]
```
$\{f[a], f[b], f[c]\}$

```
>> Thread[f[{a, b, c}, t]]
```
$\{f[a, t], f[b, t], f[c, t]\}$

```
>> Thread[f[a + b + c], Plus]
```
$f[a] + f[b] + f[c]$

Functions with attribute `Listable` are automatically threaded over lists:

```
>> {a, b, c} + {d, e, f} + g
```
$\{a + d + g, b + e + g, c + f + g\}$

Through

`Through[`*p*`[`*f*`][`*x*`]]`
: gives $p[f[x]]$.

```
>> Through[f[g][x]]
```
$f[g[x]]$

```
>> Through[p[f, g][x]]
```
$p[f[x], g[x]]$

XXXIII. System

Contents

Names

`Names["`*pattern*`"]`
: returns the list of names matching *pattern*.

```
>> Names["List"]
```
{List}

```
>> Names["List*"]
```
{List, ListLinePlot, ListPlot, ListQ, Listable}

```
>> Names["List@"]
```
{Listable}

```
>> x = 5;
```

```
>> Names["Global`*"]
```
{x}

The number of built-in symbols:

```
>> Length[Names["System`*"]]
```
695

$Version

`$Version`
: returns a string with the current Mathics version and the versions of relevant libraries.

```
>> $Version
```
Mathics 0.9 on PyPy 2.7.10 (5f8 302b8bf9f, Nov 20 2 015, 03:42:51) using SymPy 0.7.6, mpmath 0.19

XXXIV. Tensors

Contents

ArrayDepth

```
>> ArrayDepth[{{a,b},{c,d}}]
```
2

```
>> ArrayDepth[x]
```
0

ArrayQ

`ArrayQ[`*expr*`]`
: tests whether *expr* is a full array.

`ArrayQ[`*expr*`, `*pattern*`]`
: also tests whether the array depth of *expr* matches *pattern*.

`ArrayQ[`*expr*`, `*pattern*`, `*test*`]`
: furthermore tests whether *test* yields `True` for all elements of *expr*. `ArrayQ[`*expr*`]` is equivalent to `ArrayQ[`*expr*`, _, True&]`.

```
>> ArrayQ[a]
```
False

```
>> ArrayQ[{a}]
```
True

```
>> ArrayQ[{{{a}},{{b,c}}}]
```
False

```
>> ArrayQ[{{a, b}, {c, d}}, 2,
SymbolQ]
```
True

DiagonalMatrix

`DiagonalMatrix[`*list*`]`
: gives a matrix with the values in *list* on its diagonal and zeroes elsewhere.

```
>> DiagonalMatrix[{1, 2, 3}]
```
$\{\{1,0,0\},\{0,2,0\},\{0,0,3\}\}$

```
>> MatrixForm[%]
```
$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Dimensions

```
>> Dimensions[{a, b, c}]
```
$\{3\}$

```
>> Dimensions[{{a, b}, {c, d}, {
e, f}}]
```
$\{3,2\}$

Ragged arrays are not taken into account:

```
>> Dimensions[{{a, b}, {b, c}, {
   c, d, e}}]
```
$\{3\}$

The expression can have any head:

```
>> Dimensions[f[f[a, b, c]]]
```
$\{1,3\}$

Dot (.)

Scalar product of vectors:

```
>> {a, b, c} . {x, y, z}
```
$ax + by + cz$

Product of matrices and vectors:

```
>> {{a, b}, {c, d}} . {x, y}
```
$\{ax + by, cx + dy\}$

Matrix product:

```
>> {{a, b}, {c, d}} . {{r, s}, {
   t, u}}
```
$\{\{ar + bt, as + bu\}, \{cr + dt, cs + du\}\}$

```
>> a . b
```
$a.b$

IdentityMatrix

`IdentityMatrix[n]`
: gives the identity matrix with n rows and columns.

```
>> IdentityMatrix[3]
```
$\{\{1,0,0\}, \{0,1,0\}, \{0,0,1\}\}$

Inner

```
>> Inner[f, {a, b}, {x, y}, g]
```
$g[f[a,x], f[b,y]]$

The inner product of two boolean matrices:

```
>> Inner[And, {{False, False}, {
   False, True}}, {{True, False
   }, {True, True}}, Or]
```
$\{\{\text{False},\text{False}\}, \{\text{True},\text{True}\}\}$

Inner works with tensors of any depth:

```
>> Inner[f, {{{a, b}}, {{c, d
   }}}, {{1}, {2}}, g]
```
$\{\{\{g[f[a,1], f[b,2]]\}\}, \{\{g[f[c,1], f[d,2]]\}\}\}$

MatrixQ

```
>> MatrixQ[{{1, 3}, {4.0, 3/2}},
    NumberQ]
```
True

Outer

```
>> Outer[f, {a, b}, {1, 2, 3}]
```
$\{\{f[a,1], f[a,2], f[a,3]\}, \{f[b,1], f[b,2], f[b,3]\}\}$

Outer product of two matrices:

```
>> Outer[Times, {{a, b}, {c, d
   }}, {{1, 2}, {3, 4}}]
```
$\{\{\{\{a,2a\}, \{3a,4a\}\}, \{\{b,2b\}, \{3b,4b\}\}\}, \{\{\{c,2c\}, \{3c,4c\}\}, \{\{d,2d\}, \{3d,4d\}\}\}\}$

`Outer` of multiple lists:

```
>> Outer[f, {a, b}, {x, y, z},
   {1, 2}]
```
$\{\{\{f[a,x,1], f[a,x,2]\}, \{f[a,y,1], f[a,y,2]\}, \{f[a,z,1], f[a,z,2]\}\}, \{\{f[b,x,1], f[b,x,2]\}, \{f[b,y,1], f[b,y,2]\}, \{f[b,z,1], f[b,z,2]\}\}\}$

Arrays can be ragged:

```
>> Outer[Times, {{1, 2}}, {{a, b
}, {c, d, e}}]
```

$\{\{\{\{a, b\}, \{c, d, e\}\}, \{\{2a, 2b\}, \{2c, 2d, 2e\}\}\}\}$

Word combinations:

```
>> Outer[StringJoin, {"", "re",
"un"}, {"cover", "draw", "
wind"}, {"", "ing", "s"}] //
InputForm
```

{{{"cover", "covering", "covers"}, {"draw", "drawing", "draws"}, {"wind", "winding", "winds"}}, {{"recover", "recovering", "recovers"}, {"redraw", "redrawing", "redraws"}, {"rewind", "rewinding", "rewinds"}}, {{"uncover", "uncovering", "uncovers"}, {"undraw", "undrawing", "undraws"}, {"unwind", "unwinding", "unwinds"}}}

Compositions of trigonometric functions:

```
>> trigs = Outer[Composition, {
Sin, Cos, Tan}, {ArcSin,
ArcCos, ArcTan}]
```

{{Composition [Sin, ArcSin], Composition [Sin, ArcCos], Composition [Sin, ArcTan]}, {Composition [Cos, ArcSin], Composition [Cos, ArcCos], Composition [Cos, ArcTan]}, {Composition [Tan, ArcSin], Composition [Tan, ArcCos], Composition [Tan, ArcTan]}}

Evaluate at 0:

```
>> Map[#[0] &, trigs, {2}]
```

{{0, 1, 0}, {1, 0, 1}, {0, ComplexInfinity, 0}}

Transpose

`Tranpose[`*m*`]`
: transposes rows and columns in the matrix *m*.

```
>> Transpose[{{1, 2, 3}, {4, 5,
6}}]
```

$\{\{1, 4\}, \{2, 5\}, \{3, 6\}\}$

```
>> MatrixForm[%]
```

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

VectorQ

```
>> VectorQ[{a, b, c}]
```

True

XXXV. Files

Contents

AbsoluteFileName

`AbsoluteFileName["`*name*`"]`
returns the absolute version of the given filename.

```
>> AbsoluteFileName["ExampleData
   /sunflowers.jpg"]
```

/home/angus/Mathics/mathics/data/ExampleData/sunflowers.jpg

BinaryRead

`BinaryRead[`*stream*`]`
: reads one byte from the stream as an integer from 0 to 255.

`BinaryRead[`*stream*`, `*type*`]`
: reads one object of specified type from the stream.

`BinaryRead[`*stream*`, {`*type1*`, `*type2*`, ...}]`
: reads a sequence of objects of specified types.

```
>> strm = OpenWrite[BinaryFormat -> True]
```
OutputStream [/tmp/tmpWPOETy, 298]

```
>> BinaryWrite[strm, {97, 98, 99}]
```
OutputStream [/tmp/tmpWPOETy, 298]

```
>> Close[strm]
```
/tmp/tmpWPOETy

```
>> strm = OpenRead[%, BinaryFormat -> True]
```
InputStream [/tmp/tmpWPOETy, 299]

```
>> BinaryRead[strm, {"Character8", "Character8", "Character8"}]
```
{a, b, c}

```
>> Close[strm];
```

BinaryWrite

`BinaryWrite[`*channel*`, `*b*`]`
: writes a single byte given as an integer from 0 to 255.

`BinaryWrite[`*channel*`, {b1, b2, ...}]`
: writes a sequence of byte.

`BinaryWrite[`*channel*`, ‘‘string’]’`
: writes the raw characters in a string.

`BinaryWrite[`*channel*`, `*x*`, `*type*`]`
: writes *x* as the specified type.

`BinaryWrite[`*channel*`, {`*x1*`, `*x2*`, ...}, `*type*`]`
: writes a sequence of objects as the specified type.

`BinaryWrite[`*channel*`, {`*x1*`, `*x2*`, ...}, {`*type1*`, `*type2*`, ...}]`
: writes a sequence of objects using a sequence of specified types.

```
>> strm = OpenWrite[BinaryFormat -> True]
```
OutputStream [/tmp/tmpPAJyiX, 698]

```
>> BinaryWrite[strm, {39, 4, 122}]
```
OutputStream [/tmp/tmpPAJyiX, 698]

```
>> Close[strm]
```
/tmp/tmpPAJyiX

```
>> strm = OpenRead[%, BinaryFormat -> True]
```
InputStream [/tmp/tmpPAJyiX, 699]

```
>> BinaryRead[strm]
```
39

```
>> BinaryRead[strm, "Byte"]
```
4

```
>> BinaryRead[strm, "Character8
"]
```
z

```
>> Close[strm];
```

Write a String

```
>> strm = OpenWrite[BinaryFormat
    -> True]
```
OutputStream [/tmp/tmp_Jn5Bf, 700]

```
>> BinaryWrite[strm, "abc123"]
```
OutputStream [/tmp/tmp_Jn5Bf, 700]

```
>> Close[%]
```
/tmp/tmp_Jn5Bf

Read as Bytes

```
>> strm = OpenRead[%,
   BinaryFormat -> True]
```
InputStream [/tmp/tmp_Jn5Bf, 701]

```
>> BinaryRead[strm, {"Character8
", "Character8", "Character8
", "Character8", "Character8
", "Character8", "Character8
"}]
```
{a, b, c, 1, 2, 3, EndOfFile}

```
>> Close[strm]
```
/tmp/tmp_Jn5Bf

Read as Characters

```
>> strm = OpenRead[%,
   BinaryFormat -> True]
```
InputStream [/tmp/tmp_Jn5Bf, 702]

```
>> BinaryRead[strm, {"Byte", "
Byte", "Byte", "Byte", "Byte
", "Byte", "Byte"}]
```
{97, 98, 99, 49, 50, 51, EndOfFile}

```
>> Close[strm]
```
/tmp/tmp_Jn5Bf

Write Type

```
>> strm = OpenWrite[BinaryFormat
    -> True]
```
OutputStream [/tmp/tmpTqwNQa, 703]

```
>> BinaryWrite[strm, 97, "Byte"]
```
OutputStream [/tmp/tmpTqwNQa, 703]

```
>> BinaryWrite[strm, {97, 98,
99}, {"Byte", "Byte", "Byte
"}]
```
OutputStream [/tmp/tmpTqwNQa, 703]

```
>> Close[%]
```
/tmp/tmpTqwNQa

```
>> strm = OpenWrite["/dev/full",
    BinaryFormat -> True]
```
OutputStream [/dev/full, 824]

```
>> BinaryWrite[strm, {39, 4,
122}]
```
No space left on device.

OutputStream [/dev/full, 824]

```
>> Close[strm]
```
No space left on device.

/dev/full

Byte

`Byte`
    is a data type for `Read`.

Character

`Character`
: is a data type for `Read`.

Close

`Close[`*stream*`]`
: closes an input or output stream.

```
>> Close[StringToStream["123abc
   "]]
```

String

```
>> Close[OpenWrite[]]
```

/tmp/tmpQQGwsV

Compress

`Compress[`*expr*`]`
: gives a compressed string representation of *expr*.

```
>> Compress[N[Pi, 10]]
```

eJwz1jM0MTS1NDIzNQEADRsCNw==

CopyDirectory

`CopyDirectory["`*dir1*`‘‘,"`*dir2*`"]`
: copies directory *dir1* to *dir2*.

CopyFile

`CopyFile["`*file1*`‘‘,"`*file2*`"]`
: copies *file1* to *file2*.

```
>> CopyFile["ExampleData/
   sunflowers.jpg", "
   MathicsSunflowers.jpg"]
```

MathicsSunflowers.jpg

```
>> DeleteFile["MathicsSunflowers
   .jpg"]
```

CreateDirectory

`CreateDirectory["`*dir*`"]`
: creates a directory called *dir*.

`CreateDirectory[]`
: creates a temporary directory.

```
>> dir = CreateDirectory[]
```

/tmp/mYrf1b9

DeleteDirectory

`DeleteDirectory["`*dir*`"]`
: deletes a directory called *dir*.

```
>> dir = CreateDirectory[]
```

/tmp/mZUN3d3

```
>> DeleteDirectory[dir]
```

```
>> DirectoryQ[dir]
```

False

DeleteFile

`Delete["`*file*`"]`
: deletes *file*.

`Delete[{"`*file1*`‘‘,"`*file2*`", ...}]`
: deletes a list of files.

```
>> CopyFile["ExampleData/
   sunflowers.jpg", "
   MathicsSunflowers.jpg"];
```

```
>> DeleteFile["MathicsSunflowers
   .jpg"]
```

```
>> CopyFile["ExampleData/
   sunflowers.jpg", "
   MathicsSunflowers1.jpg"];
```

```
>> CopyFile["ExampleData/
   sunflowers.jpg", "
   MathicsSunflowers2.jpg"];
```

```
>> DeleteFile[{"
   MathicsSunflowers1.jpg", "
   MathicsSunflowers2.jpg"}]
```

Directory

`Directory[]`
: returns the current working directory.

```
>> Directory[]
   /home/angus/Mathics
```

DirectoryName

`DirectoryName["`*name*`"]`
: extracts the directory name from a filename.

```
>> DirectoryName["a/b/c"]
   a/b
```

```
>> DirectoryName["a/b/c", 2]
   a
```

DirectoryQ

`DirectoryQ["`*name*`"]`
: returns `True` if the directory called *name* exists and `False` otherwise.

```
>> DirectoryQ["ExampleData/"]
   True
```

```
>> DirectoryQ["ExampleData/
   MythicalSubdir/"]
   False
```

DirectoryStack

`DirectoryStack[]`
: returns the directory stack.

```
>> DirectoryStack[]
   {/home/angus/Mathics}
```

EndOfFile

`EndOfFile`
: is returned by `Read` when the end of an input stream is reached.

ExpandFileName

`ExpandFileName["`*name*`"]`
: expands *name* to an absolute filename for your system.

```
>> ExpandFileName["ExampleData/
   sunflowers.jpg"]
   /home/angus/Mathics/ExampleData/sunflowers.jpg
```

Expression

`Expression`
: is a data type for `Read`.

FileBaseName

`FileBaseName["`*file*`"]`
: gives the base name for the specified file name.

>> `FileBaseName["file.txt"]`
file

>> `FileBaseName["file.tar.gz"]`
file.tar

FileByteCount

`FileByteCount[`*file*`]`
: returns the number of bytes in *file*.

>> `FileByteCount["ExampleData/sunflowers.jpg"]`
142 286

FileDate

`FileDate[`*file*`, `*types*`]`
: returns the time and date at which the file was last modified.

>> `FileDate["ExampleData/sunflowers.jpg"]`
{2 015, 10, 24, 14, 32, 15.3}

>> `FileDate["ExampleData/sunflowers.jpg", "Access"]`
{2 016, 3, 1, 12, 47, 13.49}

>> `FileDate["ExampleData/sunflowers.jpg", "Creation"]`
Missing [NotApplicable]

>> `FileDate["ExampleData/sunflowers.jpg", "Change"]`
{2 016, 2, 19, 15, 14, 7.19}

>> `FileDate["ExampleData/sunflowers.jpg", "Modification"]`
{2 015, 10, 24, 14, 32, 15.3}

>> `FileDate["ExampleData/sunflowers.jpg", "Rules"]`
{Access-> {2 016, 3, 1, 12, 47, 13.49} , Creation->Missing [NotApplicable] , Change-> {2 016, 2, 19, 15, 14, 7.19} , Modification-> {2~
~015, 10, 24, 14, 32, 15.3}}

FileExistsQ

`FileExistsQ["`*file*`"]`
: returns `True` if *file* exists and `False` otherwise.

>> `FileExistsQ["ExampleData/sunflowers.jpg"]`
True

>> `FileExistsQ["ExampleData/sunflowers.png"]`
False

FileExtension

`FileExtension["`*file*`"]`
: gives the extension for the specified file name.

>> `FileExtension["file.txt"]`
txt

>> `FileExtension["file.tar.gz"]`
gz

FileHash

`FileHash[`*file*`]`
: returns an integer hash for the given *file*.

`FileHash[`*file*`, `*type*`]`
: returns an integer hash of the specified *type* for the given *file*.

<dd>The types supported are "MD5", "Adler32", "CRC32", "SHA", "SHA224", "SHA256", "SHA384", and "SHA512".</dd>

>> **`FileHash["ExampleData/sunflowers.jpg"]`**

109 937 059 621 979 839 ~
~952 736 809 235 486 742 106

>> **`FileHash["ExampleData/sunflowers.jpg", "MD5"]`**

109 937 059 621 979 839 ~
~952 736 809 235 486 742 106

>> **`FileHash["ExampleData/sunflowers.jpg", "Adler32"]`**

1 607 049 478

>> **`FileHash["ExampleData/sunflowers.jpg", "SHA256"]`**

111 619 807 552 579 450 300 ~
~684 600 241 129 773 909 ~
~359 865 098 672 286 468 ~
~229 443 390 003 894 913 065

FileNameDepth

`FileNameDepth["`*name*`"]`
: gives the number of path parts in the given filename.

>> **`FileNameDepth["a/b/c"]`**

3

>> **`FileNameDepth["a/b/c/"]`**

3

FileNameJoin

`FileNameJoin[{"`*dir_1*`‘‘,"`*dir_2*`", ...}]`
: joins the *dir_i* togeather into one path.

>> **`FileNameJoin[{"dir1", "dir2", "dir3"}]`**

dir1/dir2/dir3

>> **`FileNameJoin[{"dir1", "dir2", "dir3"}, OperatingSystem -> "Unix"]`**

dir1/dir2/dir3

FileNameSplit

`FileNameSplit["`*filenams*`"]`
: splits a *filename* into a list of parts.

>> **`FileNameSplit["example/path/file.txt"]`**

{example, path, file.txt}

FilePrint

`FilePrint[`*file*`]`
: prints the raw contents of *file*.

FileType

`FileType["`*file*`"]`
: returns the type of a file, from `File`, `Directory` or `None`.

>> **`FileType["ExampleData/sunflowers.jpg"]`**

File

```
>> FileType["ExampleData"]
   Directory

>> FileType["ExampleData/
   nonexistant"]
   None
```

Find

`Find[`*stream*`, `*text*`]`
: find the first line in *stream* that contains *text*.

```
>> str = OpenRead["ExampleData/
   EinsteinSzilLetter.txt"];

>> Find[str, "uranium"]
   in manuscript, leads me
     to expect that the element
     uranium may be turned into

>> Find[str, "uranium"]
   become possible to set up
     a nuclear chain reaction in
     a large mass of uranium,

>> Close[str]
   ExampleData/EinsteinSzilLetter.txt

>> str = OpenRead["ExampleData/
   EinsteinSzilLetter.txt"];

>> Find[str, {"energy", "power"}
   ]
   a new and important source
     of energy in the immediate
     future. Certain aspects

>> Find[str, {"energy", "power"}
   ]
   by which vast amounts of
     power and large quantities
     of new radium-like

>> Close[str]
   ExampleData/EinsteinSzilLetter.txt
```

FindFile

`FindFile[`*name*`]`
: searches `$Path` for the given filename.

```
>> FindFile["ExampleData/
   sunflowers.jpg"]
   /home/angus/Mathics/mathics/data/ExampleData

>> FindFile["VectorAnalysis`"]
   /home/angus/Mathics/mathics/packages/VectorAr

>> FindFile["VectorAnalysis`
   VectorAnalysis`"]
   /home/angus/Mathics/mathics/packages/VectorAr
```

FindList

`FindList[`*file*`, `*text*`]`
: returns a list of all lines in *file* that contain *text*.

`FindList[`*file*`, {`*text1*`, `*text2*`, ...}]`
: returns a list of all lines in *file* that contain any of the specified string.

`FindList[{`*file1*`, `*file2*`, ...}, ...]`
: returns a list of all lines in any of the *filei* that contain the specified strings.

```
>> str = FindList["ExampleData/
   EinsteinSzilLetter.txt", "
   uranium"];

>> FindList["ExampleData/
   EinsteinSzilLetter.txt", "
   uranium", 1]
   {in manuscript, leads me
     to expect that the element
     uranium may be turned into}
```

Get (<<)

`<<name`
: reads a file and evaluates each expression, returning only the last one.

```
>> Put[x + y, "example_file"]

>> <<"example_file"
```
Invalid syntax at or near token }.

```
>> Put[x + y, 2x^2 + 4z!, Cos[x]
     + I Sin[x], "example_file"]

>> <<"example_file"
```
Invalid syntax at or near token }.

```
>> 40! >> "fourtyfactorial"

>> FilePrint["fourtyfactorial"]
```
815 915 283 247 897 734 345 611 269 596 115 894 272 000 000 000

```
>> <<"fourtyfactorial"
```
815 915 283 247 897 734 345 611~
~269 596 115 894 272 000 000 000

$HomeDirectory

`$HomeDirectory`
: returns the users HOME directory.

```
>> $HomeDirectory
```
/home/angus

$InitialDirectory

`$InitialDirectory`
: returns the directory from which *Mathics* was started.

```
>> $InitialDirectory
```
/home/angus/Mathics

$Input

`$Input`
: is the name of the stream from which input is currently being read.

```
>> $Input
```

$InputFileName

`$InputFileName`
: is the name of the file from which input is currently being read.

While in interactive mode, `$InputFileName` is "".

```
>> $InputFileName
```

InputStream

`InputStream[`*name*`, `*n*`]`
: represents an input stream.

```
>> str = StringToStream["Mathics
     is cool!"]
```
InputStream [String, 905]

```
>> Close[str]
```
String

$InstallationDirectory

`$InstallationDirectory`
: returns the directory in which *Mathics* was installed.

```
>> $InstallationDirectory
```
/home/angus/Mathics/mathics/

Needs

`Needs["context`"]` <dd>loads the specified context if not already in `$Packages`.

>> `Needs["VectorAnalysis`"]`

Number

`Number`
: is a data type for `Read`.

OpenAppend

`OpenAppend[``file']'`
: opens a file and returns an OutputStream to which writes are appended.

>> `OpenAppend[]`

OutputStream [/tmp/tmpW_V6Hm, 928]

OpenRead

`OpenRead[``file']'`
: opens a file and returns an InputStream.

>> `OpenRead["ExampleData/EinsteinSzilLetter.txt"]`

InputStream [ExampleData/EinsteinSzilLetter.txt, 934]

OpenWrite

`OpenWrite[``file']'`
: opens a file and returns an OutputStream.

>> `OpenWrite[]`

OutputStream [/tmp/tmp_od9dJ, 940]

$OperatingSystem

`$OperatingSystem`
: gives the type of operating system running Mathics.

>> `$OperatingSystem`

Unix

OutputStream

`OutputStream[`*name*`, `*n*`]`
: represents an output stream.

>> `OpenWrite[]`

OutputStream [/tmp/tmpgxmqww, 944]

>> `Close[%]`

/tmp/tmpgxmqww

ParentDirectory

`ParentDirectory[]`
: returns the parent of the current working directory.

`ParentDirectory["`*dir*`"]`
: returns the parent *dir*.

```
>> ParentDirectory[]
```
/home/angus

$Path

`$Path`
: returns the list of directories to search when looking for a file.

```
>> $Path
```
{/home/angus, /home/angus/Mathics/mathics/data, /home/angus/Mathics/mathics/packages}

$PathnameSeparator

`$PathnameSeparator`
: returns a string for the seperator in paths.

```
>> $PathnameSeparator
```
/

Put (>>)

expr `>>` *filename*
: write *expr* to a file.

`Put[`*expr1*`,` *expr2*`, ..., $‘‘filename’$]’`
: write a sequence of expressions to a file.

```
>> 40! >> "fourtyfactorial"
```

```
>> FilePrint["fourtyfactorial"]
```
815 915 283 247 897 734 345 611 269 596 115 894 272 000 000 000

```
>> Put[50!, "fiftyfactorial"]
```

```
>> FilePrint["fiftyfactorial"]
```
30 414 093 201 713 378 043 612 608 166 064 768 844 377 641 568 960 512 000 000 000 000

```
>> Put[10!, 20!, 30!, "
factorials"]
```

```
>> FilePrint["factorials"]
```
3 628 800
2 432 902 008 176 640 000
265 252 859 812 191 058 636 308 480 000 000

=

PutAppend (>>>)

expr `>>>` *filename*
: append *expr* to a file.

`PutAppend[`*expr1*`,` *expr2*`, ..., $‘‘filename’$]’`
: write a sequence of expressions to a file.

```
>> Put[50!, "factorials"]
```

```
>> FilePrint["factorials"]
```
30 414 093 201 713 378 043 612 608 166 064 768 844 377 64

```
>> PutAppend[10!, 20!, 30!, "
factorials"]
```

```
>> FilePrint["factorials"]
```
30 414 093 201 713 378 043 612 608 166 064 768 844 377 64
3 628 800
2 432 902 008 176 640 000
265 252 859 812 191 058 636 308 480 000 000

```
>> 60! >>> "factorials"
```

```
>> FilePrint["factorials"]
```
30 414 093 201 713 378 043 612 608 166 064 768 844 377 64
3 628 800
2 432 902 008 176 640 000
265 252 859 812 191 058 636 308 480 000 000
8 320 987 112 741 390 144 276 341 183 223 364 380 754 172

```
>> "string" >>> factorials
```

```
>> FilePrint["factorials"]
```
30 414 093 201 713 378 043 612 608 166 064 768 844 377 641 568 960 512 000 000 000 000
3 628 800
2 432 902 008 176 640 000
265 252 859 812 191 058 636 308 480 000 000
8 320 987 112 741 390 144 276 341 183 223 364 38 … 000 000 000 000
"string"

Read

`Read[stream]`
: reads the input stream and returns one expression.

`Read[stream, type]`
: reads the input stream and returns an object of the given type.

```
>> str = StringToStream["abc123
"];
```

```
>> Read[str, String]
```
abc123

```
>> str = StringToStream["abc
123"];
```

```
>> Read[str, Word]
```
abc

```
>> Read[str, Word]
```
123

```
>> str = StringToStream["123,
4"];
```

```
>> Read[str, Number]
```
123

```
>> Read[str, Number]
```
4

```
>> str = StringToStream["123 abc
"];
```

```
>> Read[str, {Number, Word}]
```
{123, abc}

ReadList

`ReadList["`*file*`"]`
: Reads all the expressions until the end of file.

`ReadList["`*file*`", `*type*`]`
: Reads objects of a specified type until the end of file.

`ReadList["`*file*`", {`*type1*`, `*type2*`, ...}]`
: Reads a sequence of specified types until the end of file.

```
>> ReadList[StringToStream["a 1
b 2"], {Word, Number}]
```
{{a, 1}, {b, 2}}

```
>> str = StringToStream["abc123
"];
```

```
>> ReadList[str]
```
{abc123}

```
>> InputForm[%]
```
{"abc123"}

Record

`Record`
: is a data type for `Read`.

RenameDirectory

`RenameyDirectory["`*dir1*`‘‘,"`*dir2*`"]`
: renames directory *dir1* to *dir2*.

RenameFile

`RenameFile["`*file1*`‘‘,"`*file2*`"]`
: renames *file1* to *file2*.

```
>> CopyFile["ExampleData/sunflowers.jpg", "MathicsSunflowers.jpg"]
```

MathicsSunflowers.jpg

```
>> RenameFile["MathicsSunflowers.jpg", "MathicsSunnyFlowers.jpg"]
```

MathicsSunnyFlowers.jpg

```
>> DeleteFile["MathicsSunnyFlowers.jpg"]
```

ResetDirectory

`ResetDirectory[]`
pops a directory from the directory stack and returns it.

```
>> ResetDirectory[]
```

Directory stack is empty.

/home/angus/Mathics

$RootDirectory

`$RootDirectory`
returns the system root directory.

```
>> $RootDirectory
```

/

SetDirectory

`SetDirectory[`*dir*`]`
sets the current working directory to *dir*.

```
>> SetDirectory[]
```

/home/angus

SetFileDate

`SetFileDate["`*file*`"]`
set the file access and modification dates of *file* to the current date.

`SetFileDate["`*file*`", `*date*`]`
set the file access and modification dates of *file* to the specified date list.

`SetFileDate["`*file*`", `*date*`, "`*type*`"]`
set the file date of *file* to the specified date list. The "*type*" can be one of "*Access*", "*Creation*", "*Modification*", or `All`.

Create a temporary file (for example purposes)

```
>> tmpfilename = $TemporaryDirectory <> "/tmp0";
```

```
>> Close[OpenWrite[tmpfilename]];
```

```
>> SetFileDate[tmpfilename, {2000, 1, 1, 0, 0, 0.}, "Access"];
```

```
>> FileDate[tmpfilename, "Access"]
```

$\{2\,000, 1, 1, 0, 0, 0.\}$

SetStreamPosition

`SetStreamPosition[`*stream*`, `*n*`]`
sets the current position in a stream.

```
>> str = StringToStream["Mathics is cool!"]
```

InputStream [String, 1 058]

```
>> SetStreamPosition[str, 8]
```

8

```
>> Read[str, Word]
```

is

```
>> SetStreamPosition[str, Infinity]
```
16

Skip

`Skip[`*stream*`, `*type*`]`
: skips ahead in an input steream by one object of the specified *type*.

`Skip[`*stream*`, `*type*`, `*n*`]`
: skips ahead in an input steream by *n* objects of the specified *type*.

```
>> str = StringToStream["a b c d"];
```

```
>> Read[str, Word]
```
a

```
>> Skip[str, Word]
```

```
>> Read[str, Word]
```
c

```
>> str = StringToStream["a b c d"];
```

```
>> Read[str, Word]
```
a

```
>> Skip[str, Word, 2]
```

```
>> Read[str, Word]
```
d

StreamPosition

`StreamPosition[`*stream*`]`
: returns the current position in a stream as an integer.

```
>> str = StringToStream["Mathics is cool!"]
```
InputStream [String, 1 067]

```
>> Read[str, Word]
```
Mathics

```
>> StreamPosition[str]
```
7

Streams

`Streams[]`
: returns a list of all open streams.

```
>> Streams[]
```
{OutputStream [MathicsNonExampleFile, 925], OutputStream [MathicsNonExampleFile, 927], OutputStream [MathicsNonExampleFile, 929], InputStream [String, 1 006], InputStream [String, 1 020], InputStream [String, 1 034], InputStream [String, 1 044], InputStream [String, 1 046], InputStream [String, 1 047], InputStream [String, 1 049], InputStream [String, 1 050], InputStream [String, 1 052], InputStream [String, 1 056], InputStream [String, 1 057], InputStream [String, 1 058], InputStream [String, 1 065], InputStream [String, 1 066], InputStream [String, 1 067], OutputStream [/tmp/tmp3RNLcZ, 1 ~ ~068], OutputStream [/tmp/tmpAmr3Lw, 1 069]}

StringToStream

`StringToStream[`*string*`]`
: converts a *string* to an open input stream.

```
>> strm = StringToStream["abc
123"]
```

InputStream [String, 1 073]

$TemporaryDirectory

`$TemporaryDirectory`
: returns the directory used for temporary files.

```
>> $TemporaryDirectory
```

/tmp

Uncompress

`Uncompress["`*string*`"]`
: recovers an expression from a string generated by `Compress`.

```
>> Compress["Mathics is cool"]
```

eJxT8k0sychMLlbILFZIzs/PUQIANFwF1w==

```
>> Uncompress[%]
```

Mathics is cool

```
>> a = x ^ 2 + y Sin[x] + 10 Log
[15];
```

```
>> b = Compress[a];
```

```
>> Uncompress[b]
```

$x^2 + y\mathrm{Sin}[x] + 10\mathrm{Log}[15]$

Word

`Word`
: is a data type for `Read`.

Write

`Write[`*channel*`, `*expr1*`, `*expr2*`, ...]`
: writes the expressions to the output channel followed by a newline.

```
>> str = OpenWrite[]
```

OutputStream [/tmp/tmpiTUG0D, 1 078]

```
>> Write[str, 10 x + 15 y ^ 2]
```

```
>> Write[str, 3 Sin[z]]
```

```
>> Close[str]
```

/tmp/tmpiTUG0D

```
>> str = OpenRead[%];
```

```
>> ReadList[str]
```

{10 x + 15 y ^ 2, 3 Sin[z]}

WriteString

`WriteString[`*stream*`, $str1, `*str2*`, ... ]`
: writes the strings to the output stream.

```
>> str = OpenWrite[];
```

```
>> WriteString[str, "This is a
test 1"]
```

```
>> WriteString[str, "This is
also a test 2"]
```

```
>> Close[str]
```

/tmp/tmp13ApMU

>> `FilePrint[%]`
This is a test 1This is also a test 2

>> `str = OpenWrite[];`

>> `WriteString[str, "This is a test 1", "This is also a test 2"]`

>> `Close[str]`
/tmp/tmpptSbbL

>> `FilePrint[%]`
This is a test 1This is also a test 2

XXXVI. Importexport

Contents

Export

`Export["`*file.ext*`", `*expr*`]`
: exports *expr* to a file, using the extension *ext* to determine the format.

`Export["`*file*`", `*expr*`, "`*format*`"]`
: exports *expr* to a file in the specified format.

`Export["`*file*`", `*exprs*`, `*elems*`]`
: exports *exprs* to a file as elements specified by *elems*.

$ExportFormats

`$ExportFormats`
: returns a list of file formats supported by Export.

```
>> $ExportFormats
```

{CSV, SVG, Text}

FileFormat

`FileFormat["`*name*`"]`
: attempts to determine what format `Import` should use to import specified file.

```
>> FileFormat["ExampleData/
   sunflowers.jpg"]
```

JPEG

```
>> FileFormat["ExampleData/
   EinsteinSzilLetter.txt"]
```

Text

```
>> FileFormat["ExampleData/lena.
   tif"]
```

TIFF

Import

`Import["`*file*`"]`
: imports data from a file.

`Import["`*file*`", `*elements*`]`
: imports the specified elements from a file.

`Import["http://`*url*`", ...]` and `Import["ftp://`*url*`", ...]`
: imports from a URL.

```
>> Import["ExampleData/
   ExampleData.txt", "Elements"]
```

{Data, Lines, Plaintext, String, Words}

```
>> Import["ExampleData/ExampleData.txt", "Lines"]
```

{Example File Format, Created by Angus, 0.629452 0.586355, 0.711009 0.687453, 0.246540 0.433973, 0.926871 0.887255, 0.825141 0.940900, 0.847035 0.127464, 0.054348 0.296494, 0.838545 0.247025, 0.838697 0.436220, 0.309496 0.833591}

```
>> Import["ExampleData/colors.json"]
```

{colorsArray-> {{colorName->black, rgbValue->(0, 0, 0), hexValue->#000 000} , {colorName->red, rgbValue->(255, 0, 0), hexValue->#FF0 000} , {colorName->green, rgbValue->(0, 255, 0), hexValue->#00FF00} , {colorName->blue, rgbValue->(0, 0, 255), hexValue->#0 000FF} , {colorName->yellow, rgbValue->(255, 255, 0), hexValue->#FFFF00} , {colorName->cyan, rgbValue->(0, 255, 255), hexValue->#00FFFF} , {colorName->magenta, rgbValue->(255, 0, 255), hexValue->#FF00FF} , {colorName->white, rgbValue->(255, 255, 255), hexValue->#FFFFFF}}}

$ImportFormats

`$ImportFormats`
returns a list of file formats supported by Import.

```
>> $ImportFormats
```

{CSV, JSON, Text}

RegisterExport

`RegisterExport["`*format*`", `*func*`]`
register *func* as the default function used when exporting from a file of type "*format*".

Simple text exporter

```
>> ExampleExporter1[filename_, data_, opts___] := Module[{strm = OpenWrite[filename], char = data}, WriteString[strm, char]; Close[strm]]
```

```
>> RegisterExport["ExampleFormat1", ExampleExporter1]
```

```
>> Export["sample.txt", "Encode this string!", "ExampleFormat1"];
```

```
>> FilePrint["sample.txt"]
```

Encode this string!

Very basic encrypted text exporter

```
>> ExampleExporter2[filename_, data_, opts___] := Module[{strm = OpenWrite[filename], char}, (* TODO: Check data *) char = FromCharacterCode[Mod[ToCharacterCode[data] - 84, 26] + 97]; WriteString[strm, char]; Close[strm]]
```

```
>> RegisterExport["ExampleFormat2", ExampleExporter2]
```

```
>> Export["sample.txt", "encodethisstring", "ExampleFormat2"];
```

```
>> FilePrint["sample.txt"]
```

rapbqrguvffgevat

RegisterImport

`RegisterImport["`*format*`", `*defaultFunction*`]`
: register *defaultFunction* as the default function used when importing from a file of type "*format*".

`RegisterImport["`*format*`", {"`*elem1*`" :> `*conditionalFunction1*`, "`*elem2*`" :> `*conditionalFunction2*`, ..., `*defaultFunction*`}]`
: registers multiple elements (*elem1*, ...) and their corresponding converter functions (*conditionalFunction1*, ...) in addition to the *defaultFunction*.

`RegisterImport["`*format*`", {"`*conditionalFunctions*`, `*defaultFunction*`, "`*elem3*`" :> `*postFunction3*`, "`*elem4*`" :> `*postFunction4*`, ...}]`
: also registers additional elements (*elem3*, ...) whose converters (*postFunction3*, ...) act on output from the low-level funcions.

First, define the default function used to import the data.

```
>> ExampleFormat1Import[
   filename_String] := Module[{
   stream, head, data}, stream =
    OpenRead[filename]; head =
   ReadList[stream, String, 2];
   data = Partition[ReadList[
   stream, Number], 2]; Close[
   stream]; {"Header" -> head, "
   Data" -> data}]
```

`RegisterImport` is then used to register the above function to a new data format.

```
>> RegisterImport["
   ExampleFormat1",
   ExampleFormat1Import]
```

```
>> FilePrint["ExampleData/
   ExampleData.txt"]
```

Example File Format
Created by Angus
0.629452 0.586355
0.711009 0.687453
0.246540 0.433973
0.926871 0.887255
0.825141 0.940900
0.847035 0.127464
0.054348 0.296494
0.838545 0.247025
0.838697 0.436220
0.309496 0.833591

```
>> Import["ExampleData/
   ExampleData.txt", {"
   ExampleFormat1", "Elements"}]
```

{Data, Header}

```
>> Import["ExampleData/
   ExampleData.txt", {"
   ExampleFormat1", "Header"}]
```

{Example File Format,
Created by Angus}

Conditional Importer:

```
>> ExampleFormat2DefaultImport[
   filename_String] := Module[{
   stream, head}, stream =
   OpenRead[filename]; head =
   ReadList[stream, String, 2];
   Close[stream]; {"Header" ->
   head}]
```

```
>> ExampleFormat2DataImport[
   filename_String] := Module[{
   stream, data}, stream =
   OpenRead[filename]; Skip[
   stream, String, 2]; data =
   Partition[ReadList[stream,
   Number], 2]; Close[stream];
   {"Data" -> data}]
```

```
>> RegisterImport["
   ExampleFormat2", {"Data" :>
   ExampleFormat2DataImport,
   ExampleFormat2DefaultImport}]
```

```
>> Import["ExampleData/
   ExampleData.txt", {"
   ExampleFormat2", "Elements"}]
```

$\{\text{Data}, \text{Header}\}$

```
>> Import["ExampleData/
   ExampleData.txt", {"
   ExampleFormat2", "Header"}]
```

{Example File Format,
Created by Angus}

```
>> Import["ExampleData/
   ExampleData.txt", {"
   ExampleFormat2", "Data"}] //
   Grid
```

0.629452	0.586355
0.711009	0.687453
0.24654	0.433973
0.926871	0.887255
0.825141	0.9409
0.847035	0.127464
0.054348	0.296494
0.838545	0.247025
0.838697	0.43622
0.309496	0.833591

Part III.

License

A. GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers `and authors` protection, the GPL clearly explains that there is no warranty for this free software. For both users `and authors` sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major

Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the works `users, your or third parties` legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.
You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model,

to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed.

Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors. All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating

where to find the applicable terms.
Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).
However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.
An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.
You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a

cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE

OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.
END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.
To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it
   does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or
        modify
    it under the terms of the GNU General Public License as
       published by
    the Free Software Foundation, either version 3 of the License
       , or
    (at your option) any later version.

    This program is distributed in the hope that it will be
       useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty
       of
```

```
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public
       License
    along with this program.  If not, see <http://www.gnu.org/
       licenses/>.
```

Also add information on how to contact you by electronic and paper mail.
If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details
       type ‘show w’.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ’show w `and ‘show c` should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.
You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.
The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

B. Included software and data

Included data

Mathics includes data from Wikipedia that is published under the Creative Commons Attribution-Sharealike 3.0 Unported License and the GNU Free Documentation License contributed by the respective authors that are listed on the websites specified in "data/elements.txt".

SPARK

The “Scanning, Parsing and Rewriting Kit” from `http://pages.cpsc.ucalgary.ca/~{}aycock/spark/`.
Copyright © 1998-2002 John Aycock
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

scriptaculous

Copyright © 2005-2008 Thomas Fuchs (`http://script.aculo.us`, `http://mir.aculo.us`)
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT

Prototype

MathJax

Three.js

Index