



A free, open-source alternative to Mathematica

The Mathics Team

October 10, 2020

Contents

I.	Manual	5
1.	Introduction	6
2.	Installation and Running	8
3.	Language tutorials	9
4.	Examples	22
5.	Django-based Web Interface	25
6.	Implementation	27
II.	Reference of built-in symbols	31
I.	Algebra	32
II.	Arithmetic functions	38
III.	Assignment	48
IV.	Attributes	58
V.	Calculus functions	63
VI.	Combinatorial	68
VII.	Compilation	71
VIII.	Comparison	72
IX.	Control statements	76
X.	Date and Time	81
XI.	Differential equation solver functions	85
XII.	Evaluation	86
XIII.	Exponential, trigonometric and hyperbolic functions	90
XIV.	Functional programming	97
XV.	Graphics	99
XVI.	Graphics (3D)	117
XVII.	Image[] and image related functions.	121

XVIII.	Input and Output	127
XIX.	Integer functions	136
XX.	lohooks	139
XXI.	Linear algebra	140
XXII.	List functions	147
XXIII.	Logic	170
XXIV.	Manipulate	173
XXV.	Quantities	174
XXVI.	Number theoretic functions	176
XXVII.	Numeric evaluation	181
XXVIII.	Options and default arguments	186
XXIX.	Patterns and rules	189
XXX.	Plotting	196
XXXI.	Physical and Chemical data	206
XXXII.	Random number generation	208
XXXIII.	Recurrence relation solvers	212
XXXIV.	Special functions	213
XXXV.	Scoping	222
XXXVI.	String functions	226
XXXVII.	Structure	239
XXXVIII.	System functions	246
XXXIX.	Tensor functions	249
XL.	XML	252
XLI.	Optimization	253
XLII.	File Operations	254
XLIII.	Importing and Exporting	269
III.	License	274
A.	GNU General Public License	275
B.	Included software and data	284

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, open-source 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* home-page, <https://mathics.org>.

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? . . .	6	Who is behind it? . . .	7
What does it offer? . . .	6	What is missing? . . .	7

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 which includes a rich set of Python tools numeric computation, <https://numpy.org/numpy>, and symbolic mathematics, <https://sympy.org>.

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 L^AT_EX (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 slow, so any serious usage in cutting-edge industry or research will fail, unfortunately. Although Cython can be used to speed up parts of *Mathics*, more is needed to speed up pattern matching. Replacing recursion with iteration may help here.

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.

In the future we intend to make better use of the graphics available in the excellent packages:

- `sympy` plotting, <https://docs.sympy.org/latest/modules/plotting.html>
- `matplotlib` `pyplot`, <https://matplotlib.org/>

[org/api/pyplot_api.html](https://mathics.org/api/pyplot_api.html), and

- `networkx`, <https://networkx.github.io/>

Who is behind it?

Mathics was created by Jan Pöschk in 2011. From 2013 to about 2017 it had been maintained mostly by Angus Griffith and Ben Jones. Since then, a number of others have been people involved in *Mathics*; the list can be found in the `AUTHORS.txt` file, <https://github.com/mathics/Mathics/blob/master/AUTHORS.txt>.

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 and Running

Mathics runs natively on a computer that has Python or PyPy 3.6 or later installed. Since *Mathics* relies on *sympy* which in turn relies on *numpy*, you will need at least those installed.

Since installation may change, see <https://github.com/mathics/Mathics/wiki/Installing-and-Running> for the most recent instructions for installing from PyPI, source, or from *docker*.

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.

However if you google for “Mathematica Tutorials” you will find easily dozens of other tutorials which are applicable. Be warned though that *Mathics* does not yet offer the full range and features and capabilities of *Mathematica*®.

Contents

Basic calculations . . .	10	Strings	11	Formatting output . . .	18
Symbols and assignments . . .	11	Lists	12	Graphics	20
Comparisons and Boolean logic . . .	11	The structure of things	13	3D Graphics	20
		Functions and patterns	15	Plotting	21
		Control statements . .	15		
		Scoping	16		

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 \wedge :

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

Integer divisions yield rational numbers:

```
>> 6 / 4
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]
0.877583
```

When entering floating point numbers in your query, *Mathics* will perform a numerical evalua-

tion 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)
43.663 + 8.28556I
>> Tan[I + 0.5]
0.195577 + 0.842966I
```

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]
3.141592653589793238462643~
~383279502884197169399375~
~105820974944592307816406~
~286208998628034825342117068
```

Division by zero is forbidden:

```
>> 1 / 0
Indeterminateexpression1/0encountered.
ComplexInfinity
```

Other expressions involving *Infinity* are evaluated:

```
>> Infinity + 2 Infinity
∞
```

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

```
>> 0 ^ 0
Indeterminateexpression0^0encountered.
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
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 delimiters:

```
>> "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
Stringexpected.
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}
```

Objectsofunequallengthcannotbecombined.

```
{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 + 4 I
```

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]
a2
```

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 ex-
    pressions.
___ or BlankNullSequence[]
    matches a sequence of zero or more ex-
    pressions.
_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 func-
tions:

```

>> 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, a, 9, 2.5, 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 re-
peatedly, 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 match-
ing, simply yielding *x*. With a delayed rule this
can be avoided:

```

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

```

While ReplaceAll and ReplaceRepeated sim-
ply take the first possible match into ac-
count, ReplaceList returns a list of all possi-
ble matches. This can be used to get all subse-
quences 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 expres-
sion:

```

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

```

In addition to defining functions as rules for cer-
tain patterns, there are *pure* functions that can be
defined using the & postfix operator, where ev-
erything before it is treated as the function 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]
2x3
```

`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]
Recursiondepthof30exceeded.
$Aborted

>> f[x_] := f[x + 1]; Block[{
$IterationLimit = 30}, f[1]]
Iterationlimitof30exceeded.
$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 ex-

pressions involving c :

```
>> c + 1 // MathMLForm
<math display="block"><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 display="block">
<mtext>&lt;not&nbsp;closed</mtext>
</math>

>> d + 1 // MathMLForm
<math display="block"><mrow>
<mn>1</mn> <mo>+</mo>
<mtext>&lt;not&nbsp;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]

>> 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]isnotavalidboxstructure.
```

```
>> squared[1, 2] // TeXForm
Power[squared[1,2],
  2]isnotavalidboxstructure.
```

= 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 delimiters are not shown in the default output form:

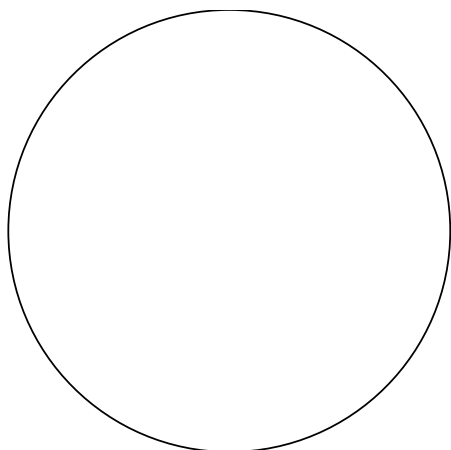
```
>> InputForm[%]
RowBox[{ "m", "+", "n" ]}
```

Graphics

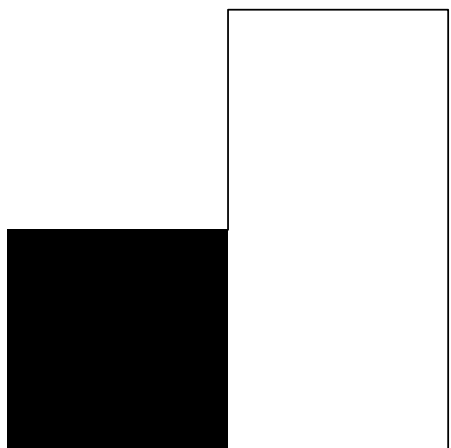
Two-dimensional graphics can be created using the function Graphics and a list of graphics primitives. For three-dimensional graphics see the following section. The following primitives are available:

```
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[]}]
```

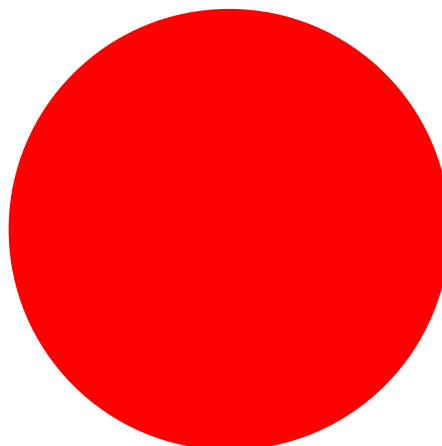
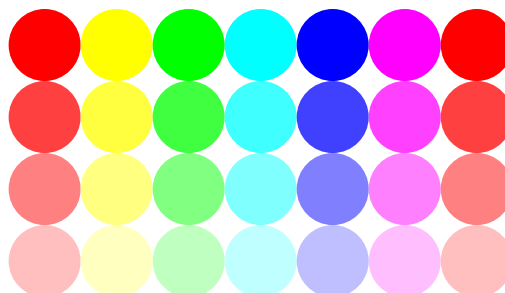


Table of hues:

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



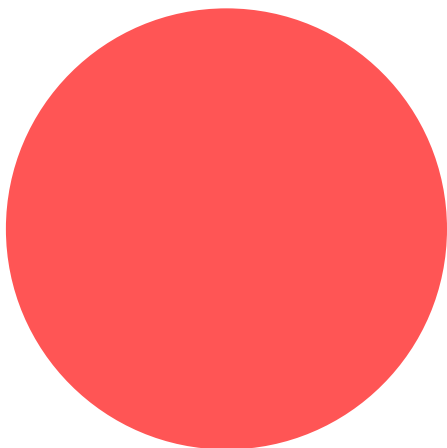
Colors can be mixed and altered using the following functions:

`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.

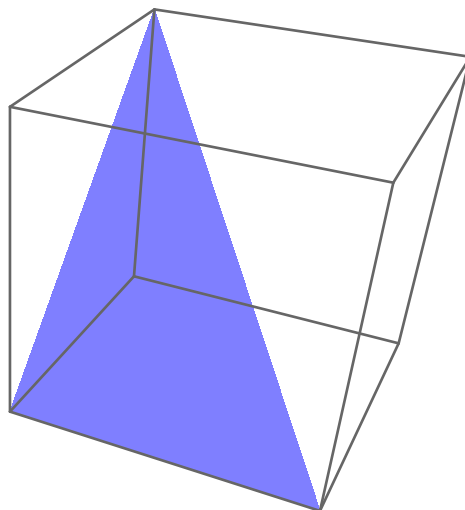
```
Line[{{x1, y1, z1}, {x2, y2, z3}, ...}]
```

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

```
Graphics3DBox[List[StyleBox[Graphics[List[EdgeForm[Graphics3DBox[Polygon[{{0,0,0}, {1,1,1}, {1,0,0}], Rule[ImageSize, 16]], Rule[ImageSizeMultipliers, List[1, 1]], Polygon3DBox[List[List[0, 0, 0], List[1, 1, 1], List[1, 0, 0]]], Rule[$OptionSyntax, Ignore], Rule[AspectRatio, Automatic], Rule[Axes, True], Rule[AxesStyle, List[]], Rule[Background, Automatic], Rule[BoxRatios, Automatic], Rule[ImageSize, Automatic], Rule[LabelStyle, List[]], Rule[Lighting, Automatic], Rule[PlotRange, Automatic], Rule[PlotRangePadding, Automatic], Rule[TicksStyle, List[]], Rule[ViewPoint, List[1.3, -2.4, 2.]]]isnotavalidboxstructure.
```

Graphics3D produces a Graphics3DBox:

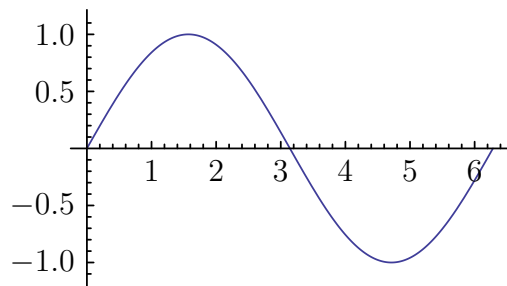
```
>> Head[ToBoxes[Graphics3D[{Polygon[]}]]]
```

Graphics3DBox

Plotting

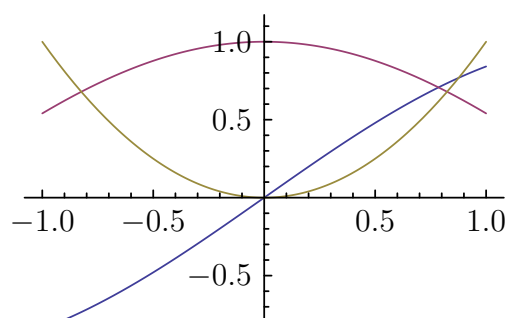
Mathics can plot functions:

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



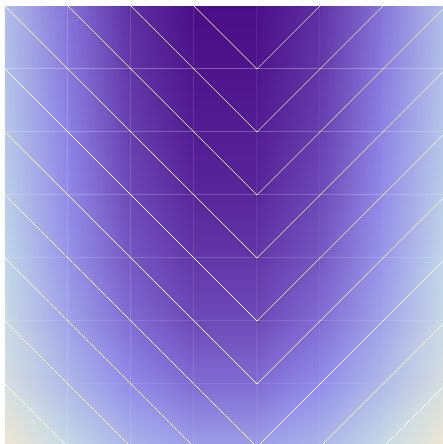
You can also plot multiple functions at once:

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



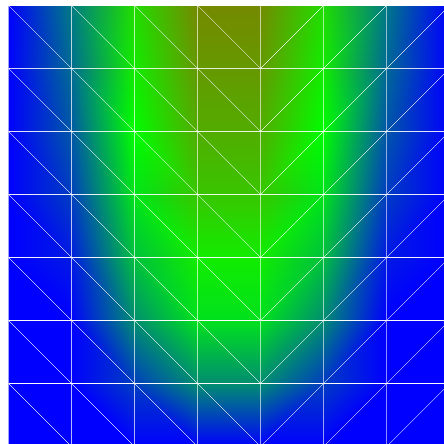
Two-dimensional functions can be plotted using `DensityPlot`:

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



You can use a custom coloring function:

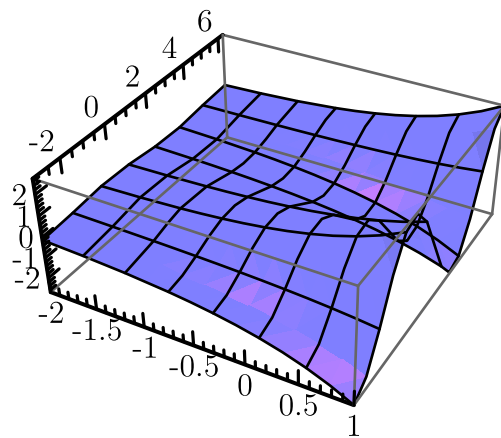
```
>> DensityPlot[x ^ 2 + 1 / y, {x, -1, 1}, {y, 1, 4}, ColorFunction -> (Blend[{Red, Green, Blue}, #]&)]
```



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	22	Linear algebra	23	Dice	24
-------------------------	----	--------------------------	----	----------------	----

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

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

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

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

$$\left\{ \left\{ x \rightarrow -\sqrt{5} \right\}, \left\{ x \rightarrow \sqrt{5} \right\} \right\}$$

And test the second derivative:

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

$$\{1.65086, -0.064079\}$$

Thus, there is a local maximum at $x = \text{Sqrt}[5]$ and a local minimum at $x = -\text{Sqrt}[5]$. Compute the inflection points numerically, chopping imaginary parts close to 0:

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

$$\left\{ \left\{ x \rightarrow -1.0852 \right\}, \left\{ x \rightarrow -3.21463 \right\}, \left\{ x \rightarrow 4.29983 \right\} \right\}$$

Insert into the third derivative:

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

$$\{-3.67683, 0.694905, 0.00671894\}$$

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]
```

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

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]
```

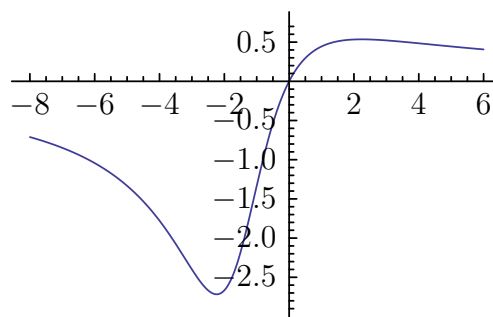
$$0$$

```
>> 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:

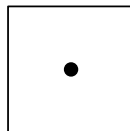
```
>> SetAttributes[Dice, Orderless]

>> 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]]

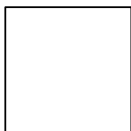
>> Dice[1]
```



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

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

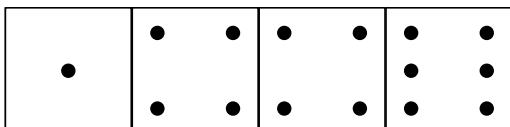
```
>> Dice[]
```



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:

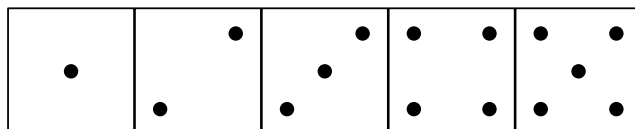
```
>> Dice[a___] + Dice[b___] ^:= Dice
  [Sequence @@ {a, b}]
```

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}]
TagPlusinDice[a___]
+ Dice[b___]isProtected.
$Failed
```

We can now combine dice:

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

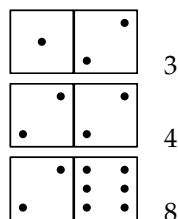


Let’s write a function that returns the sum of the rolled dice:

```
>> 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
```



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

5. Django-based Web Interface

In the future we plan on providing an interface to Jupyter as a separate package. However currently as part *Mathics*, we distribute a browser-based interface using Django 3.1. Since a Jupyter-based interface seems preferable to the home-grown interface described here, it is doubtful there will be future improvements to the this interface.

When you enter Mathics in the top after the Mathics logo and the word “Mathics” you’ll see a *menubar*.

It looks like this:



Contents

Saving and loading
worksheets 25

Persistence of Mathics
Definitions in a
Session 25

Keyboard commands 26

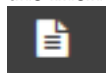
Saving and loading worksheets

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 which is the rightmost graphic of the menu bar. It looks like this:



The “Ctrl+S” key combination does the same thing.

Saved worksheets can be loaded again using the *File Open* button which is the left-most button in the menu bar. It looks like this:



The “Ctrl+O” key combination does the same thing.

Persistence of Mathics Definitions in a Session

When you use the Django-based 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!

If you are using a public terminal, to erase all your definitions and close the browser window. When you use *Mathics* in a browser, use the command Quit []

Keyboard commands

There are some keyboard commands you can use in the Django-based Web interface of *Mathics*.

Shift+Return

This evaluates the current cell (the most important one, for sure). On the right-hand side you may also see an "=" button which can be clicked to do the same thing.

Ctrl+D

This moves the cursor over to the documentation pane on the right-hand side. From here you can perform a search for a pre-defined *Mathics* function, or symbol. Clicking on the "?" symbol on the right-hand side does the same thing.

Ctrl+C

This moves the cursor back to document code pane area where you type *Mathics* expressions

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	27	Documentation markup	28	Adding built-in symbols	30
Documentation and tests	27	Classes	28		

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 doc-
    umentation.
-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> en-
    tries.
<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) tran-
    script in its own paragraph.
<con>text</con>
    an inline console transcript.
<em>text</em>
    emphasized (italic) text.
<url>url</url>
    a URL.

    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.

Classes

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

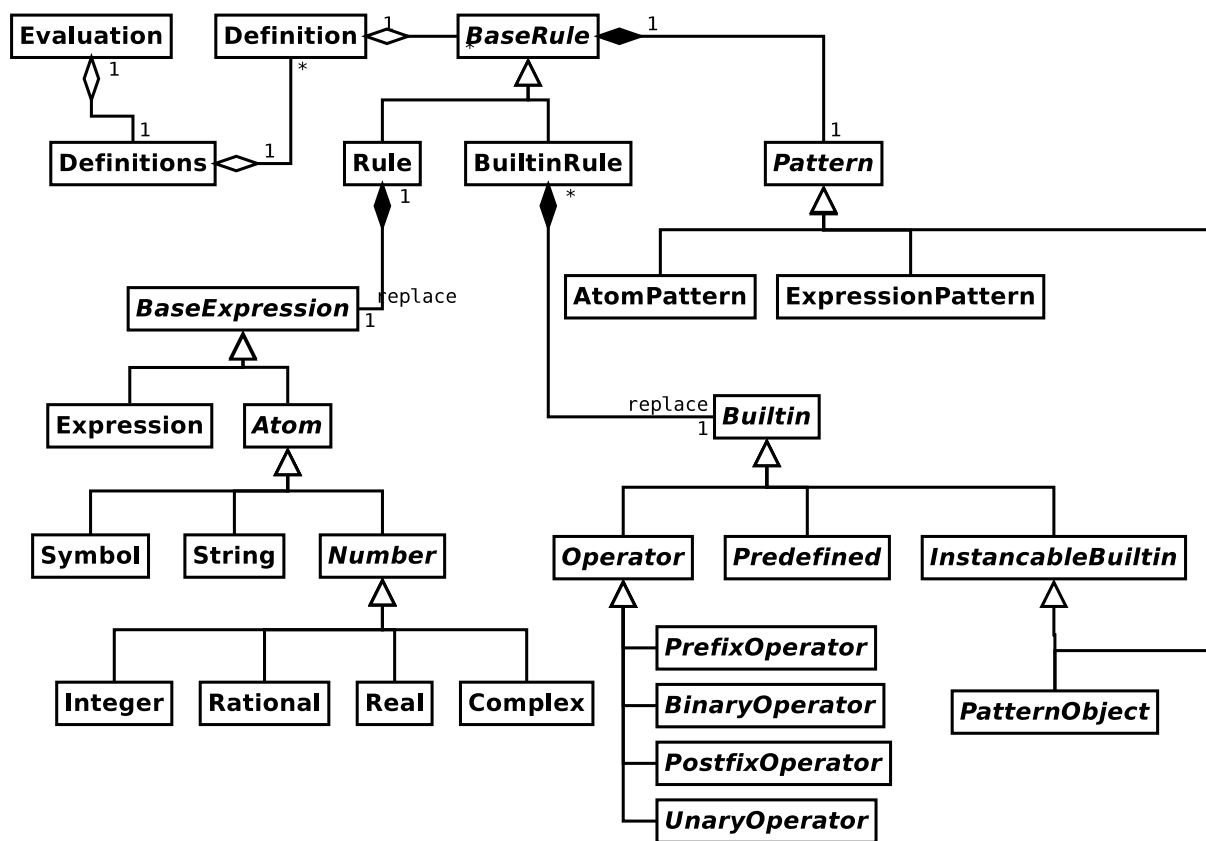


Figure 6.1.: UML class diagram

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'] explicitly if you want that.

Part II.

Reference of built-in symbols

I. Algebra

Contents

Apart	32	ExpandDenominator	34	PolynomialQ	36
Cancel	32	Exponent	35	PowerExpand	36
Coefficient	33	Factor	35	Simplify	36
CoefficientList	33	FactorTermsList	35	Together	37
Denominator	33	MinimalPolynomial	36	UpTo	37
Expand	34	Missing	36	Variables	37
ExpandAll	34	Numerator	36		

Apart

Apart[*expr*]
writes *expr* as a sum of individual fractions.
Apart[*expr*, *var*]
treats *var* as the 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]
```

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

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

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

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]
```

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

Coefficient

Coefficient[*expr*, *form*]
returns the coefficient of *form* in the polynomial *expr*.
Coefficient[*expr*, *form*, *n*]
return the coefficient of *form*^{*n*} in *expr*.

```
>> Coefficient[(x + y)^4, (x^2)* (y^2)]
```

6


```
>> Coefficient[a x^2 + b y^3 + c x
+ d y + 5, x]
c

>> Coefficient[(x + 3 y)^5, x]
405 y^4

>> Coefficient[(x + 3 y)^5, x * y
^4]
405

>> Coefficient[(x + 2)/(y - 3) + (x
+ 3)/(y - 2), x]
1
-3 + y + 1
-2 + y

>> Coefficient[x * Cos[x + 3] + 6 * y,
x]
Cos[3 + x]

>> Coefficient[(x + 1)^3, x, 2]
3

>> Coefficient[a x^2 + b y^3 + c x
+ d y + 5, y, 3]
b

>> Coefficient[(x + 2)^3 + (x + 3)
^2, x, 0]
17

>> Coefficient[(x + 2)^3 + (x + 3)
^2, y, 0]
(2 + x)^3 + (3 + x)^2

>> Coefficient[a x^2 + b y^3 + c x
+ d y + 5, x, 0]
5 + b y^3 + d y
```

CoefficientList

`CoefficientList[poly, var]`
returns a list of coefficients of powers of *var* in *poly*, starting with power 0.

`CoefficientList[poly, {var1, var2, ...}]`
returns an array of coefficients of the *vari*.

```
>> CoefficientList[(x + 3)^5, x]
{243, 405, 270, 90, 15, 1}
```

```
>> CoefficientList[(x + y)^4, x]
{y^4, 4 y^3, 6 y^2, 4 y, 1}

>> CoefficientList[a x^2 + b y^3 +
c x + d y + 5, x]
{5 + b y^3 + d y, c, a}

>> CoefficientList[(x + 2)/(y - 3) +
x/(y - 2), x]
{2
-3 + y, 1
-3 + y, 1
-2 + y}

>> CoefficientList[(x + y)^3, z]
{(x + y)^3}

>> CoefficientList[a x^2 + b y^3 +
c x + d y + 5, {x, y}]
{{5, d, 0, b}, {c, 0, 0, 0}, {a, 0, 0, 0}}

>> CoefficientList[(x - 2 y + 3 z)
^3, {x, y, z}]
{{{0, 0, 0, 27}, {0, 0, -54, 0},
{0, 36, 0, 0}, {-8, 0, 0, 0}}, {{0,
0, 27, 0}, {0, -36, 0, 0}, {12,
0, 0, 0}, {0, 0, 0, 0}}, {{0, 9, 0,
0}, {-6, 0, 0, 0}, {0, 0, 0, 0},
{0, 0, 0, 0}}, {{1, 0, 0, 0}, {0, 0,
0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}}
```

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*, as well as trigonometric identities.

```
>> 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 -> 4x + 4y}
```

Expand expands trigonometric identities

```
>> Expand[Sin[x + y], Trig -> True]
Cos[x] Sin[y] + Cos[y] Sin[x]
```

Expand does not change any other expression.

```
>> Expand[Sin[x (1 + y)]]
Sin[x (1 + y)]
```

Expand also works in Galois fields

```
>> Expand[(1 + a)^12, Modulus -> 3]
1 + a^3 + a^9 + a^12

>> Expand[(1 + a)^12, Modulus -> 4]
1 + 2a^2 + 3a^4 + 3a^8 + 2a^10 + a^12
```

ExpandAll

ExpandAll[expr]
expands out negative integer powers and products of sums in *expr*.

```
>> ExpandAll[(a + b)^2 / (c + d)^2]
a^2 / (c^2 + 2cd + d^2) + 2ab / (c^2 + 2cd + d^2)
+ b^2 / (c^2 + 2cd + d^2)
```

ExpandAll descends into sub expressions

```
>> ExpandAll[(a + Sin[x (1 + y)])^2]
2a Sin[x + xy] + a^2 + Sin[x + xy]^2
```

ExpandAll also expands heads

```
>> ExpandAll[((1 + x)(1 + y))[x]]
(1 + x + y + xy)[x]
```

ExpandAll can also work in finite fields

```
>> ExpandAll[(1 + a)^6 / (x + y)^3, Modulus -> 3]
(1 + 2a^3 + a^6) / (x^3 + y^3)
```

ExpandDenominator

ExpandDenominator[expr]
expands out negative integer powers and products of sums in *expr*.

```
>> ExpandDenominator[(a + b)^2 / ((c + d)^2 (e + f))]
(a + b)^2 / (c^2e + c^2f + 2cde + 2cdf + d^2e + d^2f)
```

Exponent

Exponent[expr, form]
returns the maximum power with which *form* appears in the expanded form of *expr*.

Exponent[expr, form, h]
applies *h* to the set of exponents with which *form* appears in *expr*.

```
>> Exponent[5 x^2 - 3 x + 7, x]
2

>> Exponent[(x^3 + 1)^2 + 1, x]
6

>> Exponent[x^(n + 1) + Sqrt[x] + 1, x]
Max  $\left[\frac{1}{2}, 1 + n\right]$ 

>> Exponent[x / y, y]
-1

>> Exponent[(x^2 + 1)^3 - 1, x, Min]
2

>> Exponent[1 - 2 x^2 + a x^3, x, List]
{0, 2, 3}

>> Exponent[0, x]
-∞

>> Exponent[1, x]
0
```

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)]

$$\frac{2 + 2x + 3x^2 + x^4}{(1 + x)^2 (1 + x^2)^2}$$

```

FactorTermsList

FactorTermsList[poly]
returns a list of 2 elements. The first element is the numerical factor in *poly*. The second one is the remaining of the polynomial with numerical factor removed

FactorTermsList[poly, {x1, x2, ...}]
returns a list of factors in *poly*. The first element is the numerical factor in *poly*. The next ones are factors that are independent of variables lists which are created by removing each variable *xi* from right to left. The last one is the remaining of polynomial after dividing *poly* to all previous factors

```
>> FactorTermsList[2 x^2 - 2]
{2, -1 + x^2}

>> FactorTermsList[x^2 - 2 x + 1]
{1, 1 - 2x + x^2}

>> f = 3 (-1 + 2 x) (-1 + y) (1 - a)
3 (-1 + 2x) (-1 + y) (1 - a)

>> FactorTermsList[f]
{-3, -1 + a - 2ax - ay + 2x + y - 2xy + 2axy}

>> FactorTermsList[f, x]
{-3, 1 - a - y + ay, -1 + 2x}

>> FactorTermsList[f, {x, y}]
{-3, -1 + a, -1 + y, -1 + 2x}
```

MinimalPolynomial

MinimalPolynomial[s, x]
gives the minimal polynomial in *x* for which the algebraic number *s* is a root.

```
>> MinimalPolynomial[7, x]
-7 + x

>> MinimalPolynomial[Sqrt[2] + Sqrt[3], x]
1 - 10x^2 + x^4
```

```
>> MinimalPolynomial[Sqrt[1 + Sqrt
[3]], x]
-2 - 2x2 + x4

>> MinimalPolynomial[Sqrt[I + Sqrt
[6]], x]
49 - 10x4 + x8
```

Missing

Numerator

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

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

>> Numerator[2 / 3]
2

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

PolynomialQ

`PolynomialQ[expr, var]`
returns True if *expr* is a polynomial in *var*,
and returns False otherwise.
`PolynomialQ[expr, {var1, ...}]`
tests whether *expr* is a polynomial in the
vari.

```
>> PolynomialQ[x3 - 2 x/y + 3xz, x]
True

>> PolynomialQ[x3 - 2 x/y + 3xz, y]
False

>> PolynomialQ[f[a] + f[a]2, f[a]]
True

>> PolynomialQ[x2 + axy2 - bSin[c], {x, y}]
True
```

```
>> PolynomialQ[x2 + axy2 - bSin[c], {a, b, c}]
False
```

PowerExpand

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

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

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

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]]
```

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

UpTo

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, Sin[y]}
```

II. Arithmetic functions

Basic arithmetic functions, including complex number arithmetic.

Contents

Abs	38	I	41	Power (^)	44
Boole	38	Im	41	Product	44
ComplexInfinity	39	Indeterminate	41	Rational	45
Complex	39	InexactNumberQ . . .	42	Re	45
Conjugate	39	Infinity	42	RealNumberQ	45
CubeRoot	39	IntegerQ	42	Real	45
DirectedInfinity	39	Integer	42	Sign	45
Divide (/)	40	MachineNumberQ . .	42	Sqrt	46
ExactNumberQ	40	Minus (-)	42	Subtract (-)	46
Factorial (!)	40	NumberQ	42	Sum	47
Gamma	41	Piecewise	43	Times (*)	47
HarmonicNumber . . .	41	Plus (+)	43		
		Pochhammer	43		

Abs

Abs[*x*]
returns the absolute value of *x*.

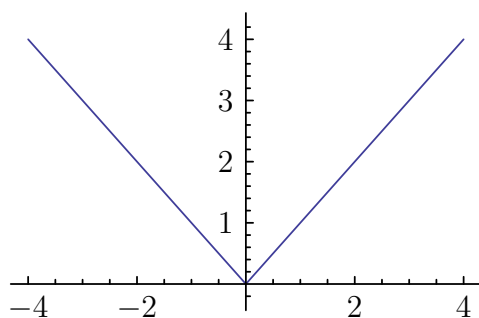
```
>> Abs[-3]
3
```

Abs returns the magnitude of complex numbers:

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

```
>> Abs[3.0 + I]
3.16228
```

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



Boole

Boole[*expr*]
returns 1 if *expr* is True and 0 if *expr* is False.

```
>> Boole[2 == 2]
1
```

```
>> Boole[7 < 5]
0
```

```
>> Boole[a == 7]
Boole[a==7]
```

ComplexInfinity

ComplexInfinity
represents an infinite complex quantity of undetermined direction.

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

```
>> 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] - IConjugate[b]}, {-I}}

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

CubeRoot

`CubeRoot[n]`
finds the real-valued cube root of the given n .

```
>> CubeRoot[16]
 $22^{\frac{1}{3}}$ 
```

DirectedInfinity

`DirectedInfinity[z]`
represents an infinite multiple of the complex number z .
`DirectedInfinity[]`
is the same as `ComplexInfinity`.

```
>> DirectedInfinity[1]
 $\infty$ 

>> 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]
Indeterminateexpression
- Infinity + Infinityencountered.
Indeterminate
```

Divide (/)

`Divide[a, b]`
 a / b
represents the division of a by b .

```
>> 30 / 5
6

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

```
>> Pi / 4
      Pi
      4
```

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

```
>> Pi / 4.0
0.785398

>> 1 / 8
      1
      8

>> N[%]
0.125
```

Nested divisions:

```
>> a / b / c
      a
      bc

>> a / (b / c)
      ac
      b

>> a / b / (c / (d / e))
      ad
      bce

>> a / (b ^ 2 * c ^ 3 / e)
      ae
      b^2 c^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]`
`n!`
computes the factorial of `n`.

```
>> 20!
2432902008176640000
```

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

```
>> 10.5!
1.18994 × 107

>> (-3.0+1.5*I)!
0.0427943 - 0.00461565I
```

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[z, x]`
is the upper incomplete gamma function.
`Gamma[z, x0, x1]`
is equivalent to `Gamma[z, x0] - Gamma[z, x1]`.

`Gamma[z]` is equivalent to $(z - 1)!$:

```
>> Simplify[Gamma[z] - (z - 1)!]
0
```

Exact arguments:

```
>> Gamma[8]
5040

>> Gamma[1/2]
√Pi

>> Gamma[1, x]
E-x
```



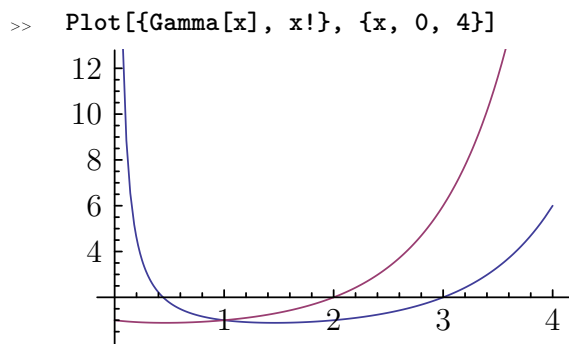
```
>> Gamma[0, x]
ExpIntegralE[1, x]
```

Numeric arguments:

```
>> Gamma[123.78]
4.21078 × 10204
```

```
>> Gamma[1. + I]
0.498016 - 0.15495I
```

Both Gamma and Factorial functions are continuous:



HarmonicNumber

HarmonicNumber[n]
returns the n th harmonic number.

```
>> Table[HarmonicNumber[n], {n, 8}]
{1, 3/2, 11/6, 25/12, 137/60, 49/20, 363/140, 761/280}
```

```
>> HarmonicNumber[3.8]
2.03806
```

I

I
represents the imaginary number $\sqrt{-1}$.

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

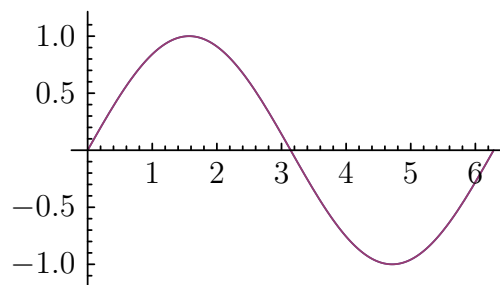
```
>> (3+I)*(3-I)
10
```

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
Indeterminateexpression00encountered.
Indeterminate
>> Tan[Indeterminate]
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}]

$$\frac{\pi^2}{6}$$

```

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
```

MachineNumberQ

`MachineNumberQ[expr]`
returns `True` if *expr* is a machine-precision real or complex number.

```
= True
>> MachineNumberQ
[3.14159265358979324]
False
>> MachineNumberQ[1.5 + 2.3 I]
True
>> MachineNumberQ
[2.71828182845904524 +
3.14159265358979324 I]
False
```

Minus (-)

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

```
>> -a //FullForm
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

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

Heaviside function

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

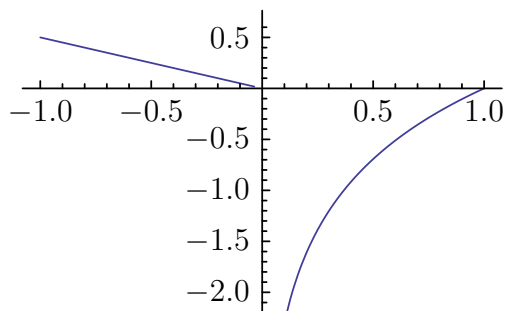
>> Integrate[Piecewise[{{1, x <= 0}, {-1, x > 0}}, x]
Piecewise[{{x, x <= 0}, {-x, True}}]

>> Integrate[Piecewise[{{1, x <= 0}, {-1, x > 0}}, {x, -1, 2}]
-1
```

Piecewise defaults to 0 if no other case is matching.

```
>> Piecewise[{{1, False}}]
0

>> Plot[Piecewise[{{Log[x], x > 0}, {x*-0.5, x < 0}}, {x, -1, 1}]
```



```
>> Piecewise[{{0 ^ 0, False}}, -1]
-1
```

Plus (+)

`Plus[a, b, ...]`
 $a + b + \dots$
represents the sum of the terms a, b, \dots

```
>> 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 + 3a + 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]
500500
```

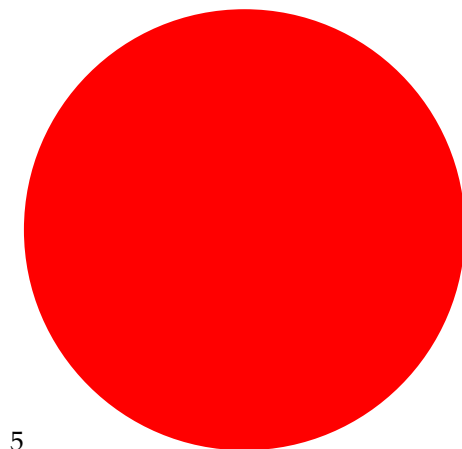
Plus has default value 0:

```
>> 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]
6652800
```

Power (^)

`Power[a, b]`
 a^b
 represents a raised to the power of b .

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

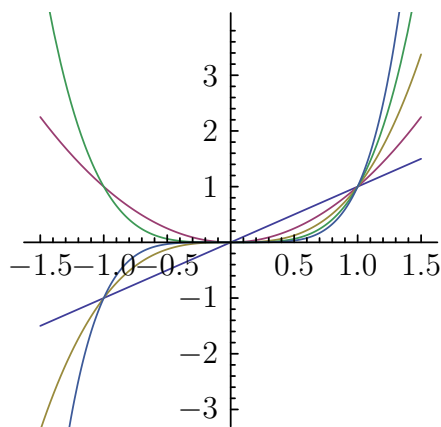
>> 4 ^ (1/3)
2^(2/3)

>> 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)
1.5874
```

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
-3.68294 + 6.95139 I

>> (1.5 + 1.0 I) ^ (3.5 + 1.5 I)
-3.19182 + 0.645659 I
```

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 $\{i, \dots\}, \{j, \dots\}, \dots$ 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^(n/2 + n^2/2)

>> Product[f[i], {i, 1, 7}]
f[1] f[2] f[3] f[4] f[5] f[6] f[7]
```

Symbolic products involving the factorial are evaluated:

```
>> Product[k, {k, 3, n}]
n! / 2
```

Evaluate the n th primorial:

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

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

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

Rational

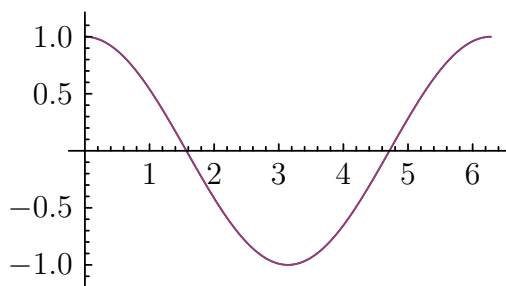
Rational
is the head of rational numbers.
Rational[*a*, *b*]
constructs the rational number a / b .

```
>> 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
```

Sign

Sign[*x*]
return -1, 0, or 1 depending on whether *x* is negative, zero, or positive.

```
>> Sign[19]
1
>> Sign[-6]
-1
>> Sign[0]
0
>> Sign[{-5, -10, 15, 20, 0}]
{-1, -1, 1, 1, 0}
>> Sign[3 - 4*I]
 $\frac{3}{5} - \frac{4I}{5}$ 
```

Sqrt

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

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

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

```
>> Sqrt[5] // N
2.23607
```

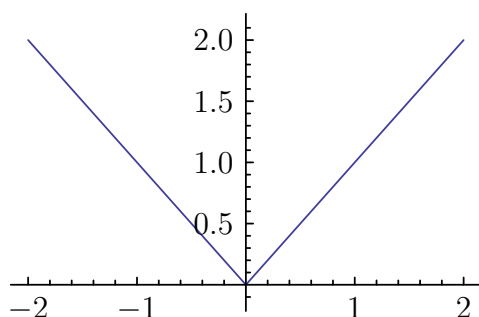
```
>> Sqrt[a]^2
a
```

Complex numbers:

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

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

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



Subtract (-)

`Subtract[a, b]`

$a - b$

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 $\{i, \dots\}, \{j, \dots\}, \dots$ being in outermost-to-innermost order.

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

Double sum:

```
>> Sum[i * j, {i, 1, 10}, {j, 1, 10}]
3025
```

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]
```

```
>> Sum[f[i], {i, 1, 7}]
f[1] + f[2] + f[3] + f[4] + f[5] + f[6] + f[7]
```

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
```

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

Infinite sums:

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

>> Sum[1 / k ^ 2, {k, 1, Infinity}]
Pi^2
6
```

Times (*)

```
Times[a, b, ...]
a * b * ...
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]]
16326
```

Times has default value 1:

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

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

III. Assignment

Contents

AddTo (+=)	48	Information (??)	52	SubValues	55
Clear	49	LoadModule	52	SubtractFrom (-=)	55
ClearAll	49	Messages	52	TagSet	55
Decrement (--).	49	NValues	53	TagSetDelayed	55
DefaultValues	49	OwnValues	53	TimesBy (*=)	56
Definition	50	PreDecrement (--).	53	Unset (=.)	56
DivideBy (/=)	51	PreIncrement (++)	53	UpSet (^=)	56
DownValues	51	Quit	54	UpSetDelayed (^:=)	57
Increment (++)	51	Set (=)	54	UpValues	57
		SetDelayed (:=)	55		

AddTo (+=)

AddTo[x, dx]
 $x += dx$
 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]
SymbolSinisProtected.
```

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]
Symbollockislocked.
```

Decrement (--)

`Decrement[x]`
`x--`
decrements x by 1, returning the original value of x .

```
>> a = 5;

>> a
5

>> a
5
```

DefaultValues

`DefaultValues[symbol]`
gives the list of default values associated with *symbol*.

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

>> DefaultValues[f]
{HoldPattern[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] ^:= 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 ~ y ~ 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]
  Attributes[r] = {Orderless}
  Default[r, 1] = 2
  Options[r] = {Opt -> 3}
```

ClearAll clears everything:

```
>> ClearAll[r]

>> Definition[r]
  Null
```

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

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

DivideBy (/=)

```
DivideBy[x, dx]
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]
{HoldPattern[f[x_]] :> x^2}
```

Mathics will sort the rules you assign to a symbol according to their specificity. 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]
{HoldPattern[f[x_Real]] :> 3,
 HoldPattern[f[x_Integer]] :> 2,
 HoldPattern[f[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 (++)

`Increment[x]`
`x++`
increments *x* by 1, returning the original value of *x*.

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

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

```
>> a
3
```

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

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

Information (??)

`Information[symbol]`
Prints information about a *symbol*

Information does not print information for

ReadProtected symbols. `Information` uses `InputForm` to format values.

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

```
>> Information[a]
a = 2
```

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

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

```
>> f::usage = "f[x] returns the
square of x";

>> Information[f]
      f[x] returns the square of x
      f[x_] = x2
      g[f]^2

>> ? Table
      'Table[expr, {i, n}]'
      evaluates expr with i ranging from
      1 to n, returning
      a list of the results.
      'Table[expr, {i, start, stop, step}]'
      evaluates expr with i ranging from
      start to stop,
      incrementing by step.
      'Table[expr, {i, {e1, e2, ..., ei}}]'
      evaluates expr with i taking on the
      values e1, e2,
      ..., ei.
```

```
>> Information[Table]
      'Table[expr, {i, n}]'
      evaluates expr with i ranging from
      1 to n, returning
      a list of the results.
      'Table[expr, {i, start, stop, step}]'
      evaluates expr with i ranging from
      start to stop,
      incrementing by step.
      'Table[expr, {i, {e1, e2, ..., ei}}]'
      evaluates expr with i taking on the
      values e1, e2,
      ..., ei.
```

```
Attributes[Table] = {HoldAll,
                     Protected}
```

LoadModule

```
LoadModule[module]
      'Load Maths definitions from the
      python module module
```

```
>> LoadModule["nomodule"]
      Pythonmodule nomodule does not exist.
      $Failed

>> LoadModule["sys"]
      Pythonmodule sys is not a pymathics module.
      $Failed
```

```
# » LoadModule["pymathics.testpymathicsmodule"]
# = pymathics.testpymathicsmodule # »
MyPyTestContext'MyPyTestFunction[a] # =
This is a PyMathics output # » MyPyTest-
Context'MyPyTestSymbol # = 1234 # » ??
MyPyTestContext'MyPyTestFunction # = # .
MyPyTestFunction[m] # . Just an example
function in pymathics module. # . # . At-
tributes[MyPyTestContext'MyPyTestFunction]
= {HoldFirst, OneIdentity, Protected} # »
Quit[] #n» MyPyTestContext'MyPyTestSymbol
# = MyPyTestContext'MyPyTestSymbol # » ??
MyPyTestContext'MyPyTestFunction # = Null
```

Messages

```
Messages[symbol]
      gives the list of messages associated with
      symbol.
```

```
>> 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[symbol]
      gives the list of numerical values associ-
      ated with symbol.
```

```
>> 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

`OwnValues[symbol]`
gives the list of ownvalues associated with *symbol*.

```
>> 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 (--)

`PreDecrement[x]`
`--x`
decrements *x* by 1, returning the new value of *x*.

`--a` is equivalent to $a = a - 1$:

```
>> a = 2;

>> --a
1

>> a
1
```

PreIncrement (++)

`PreIncrement[x]`
`++x`
increments *x* by 1, returning the new value of *x*.

`++a` is equivalent to $a = a + 1$:

```
>> 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 (=)

Set[expr, value]

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 ownvalue:

```
>> 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 (:=)

SetDelayed[expr, value]

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

`SubValues[symbol]`
gives the list of subvalues associated with *symbol*.

```
>> f[1][x_] := x
>> f[2][x_] := x ^ 2
>> SubValues[f]
{HoldPattern[f[2][x_]]:>x^2,
 HoldPattern[f[1][x_]]:>x}
>> Definition[f]
      f[2][x_] = x^2
      f[1][x_] = x
```

SubtractFrom (==)

`SubtractFrom[x, dx]`
 $x \text{ -= } dx$
is equivalent to $x = x - dx$.

```
>> a = 10;
>> a -= 2
8
>> a
8
```

TagSet

`TagSet[f, expr, value]`
 $f /: \text{expr} = \text{value}$
assigns *value* to *expr*, associating the corresponding rule with the symbol *f*.

Create an upvalue without using UpSet:

```
>> 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;
      Tagxnotfoundortoodeepforanassignedrule.
>> g /: f[g[x]] = 3;
>> f[g[x]]
3
```

TagSetDelayed

`TagSetDelayed[f, expr, value]`
 $f /: \text{expr} := \text{value}$
is the delayed version of TagSet.

TimesBy (*=)

```
TimesBy[x, dx]
x *= dx
is equivalent to x = x * dx.
```

```
>> a = 10;

>> a *= 2
20

>> a
20
```

Unset (=.)

```
Unset[x]
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 (^:=)

`UpSetDelayed[expression, value]`
expression ^:= *value*
assigns *expression* to the value of *f*[*x*]
(without evaluating *expression*), associating the value with *x*.

```
>> a[b] ^= x
>> x = 2;
>> a[b]
2
>> UpValues[b]
{HoldPattern[a[b]]:>x}
```

UpValues

`UpValues[symbol]`
gives the list of upvalues associated with *symbol*.

```
>> 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

There are several builtin-attributes which have a predefined meaning in *Mathics*. However, you can set any symbol as an attribute, in contrast to *Mathematica*®.

Contents

Attributes	58	HoldRest	59	Protect	61
ClearAttributes	59	Listable	60	Protected	61
Constant	59	Locked	60	ReadProtected	61
Flat	59	NHoldAll	60	SequenceHold	62
HoldAll	59	NHoldFirst	60	SetAttributes	62
HoldAllComplete	59	NHoldRest	60	Unprotect	62
HoldFirst	59	OneIdentity	60		
		Orderless	61		

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}
Argumenta
+ bat 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 ig-

nored:

```
>> 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]
Eisnotavalidvariable.
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, {}]
      Symbollockislocked.
>> ClearAttributes[lock, Flat]
      Symbollockislocked.
>> Attributes[lock] = {}
      Symbollockislocked.
      {}
>> 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;
SymbolAisProtected.

>> 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;
SymbolpisProtected.

>> f[p] ^= 3;
Tagpinf[p]isProtected.

>> Format[p] = "text";
SymbolpisProtected.
```

However, attributes might still be set:

```
>> SetAttributes[p, Flat]

>> Attributes[p]
{Flat, 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]
{Protected}

>> Unprotect[p]
```

If a symbol is Protected and Locked, it can never be changed again:

```
>> SetAttributes[p, {Protected, Locked}]

>> p = 2
SymbolpisProtected.
2

>> Unprotect[p]
Symbolpislocked.
```

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]
Attributes[p] = {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[Sequence[a, b]]
```

E.g., `Set` has attribute `SequenceHold` to allow assignment of sequences to variables:

```
>> s = Sequence[a, b];  
  
>> s  
Sequence[a, b]  
  
>> Plus[s]  
a + b
```

SetAttributes

`SetAttributes[symbol, attrib]`
adds *attrib* to *symbol*'s attributes.

```
>> SetAttributes[f, Flat]  
  
>> Attributes[f]  
{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 functions

Contents

Complexes	63	FindRoot	65	Reals	66
D	64	Integers	65	Root	66
Derivative (')	64	Integrate	66	Solve	67
DiscreteLimit	64	Limit	66		

Complexes

Complexes
is the set of complex numbers.

D

$D[f, x]$
gives the partial derivative of f with respect to x .
 $D[f, x, y, \dots]$
differentiates successively with respect to x, y , etc.
 $D[f, \{x, n\}]$
gives the multiple derivative of order n .
 $D[f, \{x1, x2, \dots\}]$
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]
-Cos[Cos[x]] Sin[x]

>> D[Sin[x], {x, 2}]
-Sin[x]
```

```
>> D[Cos[t], {t, 2}]
-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
Derivative[0, 1][f][x, x]
+ Derivative[1, 0][f][x, x]
```

Chain rule:

```
>> D[f[2x+1, 2y, x+y], x]
2f^(1,0,0)[1 + 2x, 2y,
x + y] + f^(0,0,1)[1 + 2x, 2y, x + y]
```

```
>> D[f[x^2, x, 2y], {x,2}, y] //
Expand
```

$$8x f^{(1,1,1)}[x^2, x, 2y] + 8x^2 f^{(2,0,1)}[x^2, x, 2y] + 2f^{(0,2,1)}[x^2, x, 2y] + 4f^{(1,0,1)}[x^2, x, 2y]$$

Compute the gradient vector of a function:

```
>> D[x ^ 3 * Cos[y], {{x, y}}
```

$$\{3x^2 \cos[y], -x^3 \sin[y]\}$$

Hesse matrix:

```
>> D[Sin[x] * Cos[y], {{x,y}, 2}]
```

$$\left\{ \left\{ -\cos[y] \sin[x], -\cos[x] \sin[y] \right\}, \left\{ -\cos[x] \sin[y], -\cos[y] \sin[x] \right\} \right\}$$

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)
```

DiscreteLimit

`DiscreteLimit[f, k->Infinity]`

gives the limit of the sequence f as k tends to infinity.

```
>> DiscreteLimit[n/(n + 1), n ->
Infinity]
```

```
1
```

```
>> DiscreteLimit[f[n], n ->
Infinity]
```

```
f[∞]
```


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 -> 1.5708}

>> FindRoot[Sin[x] + Exp[x], {x, 0}]
{x -> -0.588533}

>> FindRoot[Sin[x] + Exp[x] == Pi, {
x, 0}]
{x -> 0.866815}
```

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 -> 1.14911}

>> 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 -> -1.}
```

Find complex roots:

```
>> FindRoot[x ^ 2 + x + 1, {x, -I}]
{x -> -0.5 - 0.866025I}
```

The function has to return numerical values:

```
>> FindRoot[f[x] == 0, {x, 0}]
The function value is not a number at x = 0..
FindRoot[f[x] - 0, {x, 0}]
```

The derivative must not be 0:

```
>> FindRoot[Sin[x] == x, {x, 0}]
Encountered a singular derivative at the point x = 0..
FindRoot[Sin[x] - x, {x, 0}]
```

Integers

```
Integers
  is the set of integer numbers.
```

Limit a solution to integer numbers:

```
>> Solve[-4 - 4 x + x^4 + x^5 == 0,
x, Integers]
{{x -> -1}}

>> Solve[x^4 == 4, x, Integers]
{}
```

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]
-Cos[x] -  $\frac{\cos^3[x]}{3} + \frac{2\cos^5[x]}{5}$ 
```

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]

$$-\frac{\sqrt{3}\log\left[-2+\sqrt{3}+x\right]}{6} + \frac{\sqrt{3}\log\left[-2-\sqrt{3}+x\right]}{6}$$

```

```
>> Integrate[4 Sin[x] Cos[x], x]
2Sin[x]^2
```

Integration in TeX:

```
>> Integrate[f[x], {x, a, b}] //
TeXForm
\int_a^b f(x) dx
```

Sometimes there is a loss of precision during integration. You can check the precision of your result with the following sequence of commands.

```
>> Integrate[Abs[Sin[phi]], {phi,
0, 2Pi}] // N
4.

>> % // Precision
MachinePrecision

>> Integrate[ArcSin[x / 3], x]
x ArcSin[ $\frac{x}{3}$ ] +  $\sqrt{9 - x^2}$ 

>> 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]
 $\infty$ 

>> 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 -> 1}}
```

Root

`Root[f, i]`
represents the *i*-th complex root of the polynomial *f*

```
>> Root[#1 ^ 2 - 1 &, 1]
-1

>> Root[#1 ^ 2 - 1 &, 2]
1
```

Roots that can't be represented by radicals:

```
>> Root[#1 ^ 5 + 2 #1 + 1 &, 2]
Root[#1^5 + 2#1 + 1 &, 2]
```

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 or Integers.

```
>> Solve[x ^ 2 - 3 x == 4, x]
{{x -> -1}, {x -> 4}}

>> Solve[4 y - 8 == 0, y]
{{y -> 2}}
```

Apply the solution:

```
>> sol = Solve[2 x^2 - 10 x - 12 ==
0, x]
{{x -> -1}, {x -> 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]
{{x -> 1/2 - I/2 Sqrt[3]},
 {x -> 1/2 + I/2 Sqrt[3]}}

>> Solve[(x^2 + 3 x + 2)/(4 x - 2)
 == 0, x]
{{x -> -2}, {x -> -1}}
```

Transcendental equations:

```
>> Solve[Cos[x] == 0, x]
{{x -> Pi/2}, {x -> 3Pi/2}}
```

Solve can only solve equations with respect to symbols or functions:

```
>> Solve[f[x + y] == 3, f[x + y]]
{{f[x + y] -> 3}}

>> Solve[a + b == 2, a + b]
a + b is not a valid variable.
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.
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}] //
Simplify
```

$$\left\{ \left\{ x -> 0, y -> 0 \right\}, \left\{ x -> 1, y -> 1 \right\}, \left\{ x -> -\frac{1}{2} + \frac{I}{2}\sqrt{3}, y -> -\frac{1}{2} - \frac{I}{2}\sqrt{3} \right\}, \left\{ x -> \frac{(1 - I\sqrt{3})^2}{4}, y -> -\frac{1}{2} + \frac{I}{2}\sqrt{3} \right\} \right\}$$

```
>> 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 -> -1, z -> -I},
 {x -> -1, z -> I}, {x -> 1,
 z -> -I}, {x -> 1, z -> I}}
```

Domain specification:

```
>> Solve[x^2 == -1, x, Reals]
{}

>> Solve[x^2 == 1, x, Reals]
{{x -> -1}, {x -> 1}}

>> Solve[x^2 == -1, x, Complexes]
{{x -> -I}, {x -> I}}

>> Solve[4 - 4 * x^2 - x^4 + x^6 ==
 0, x, Integers]
{{x -> -1}, {x -> 1}}
```

VI. Combinatorial

Contents

Binomial	68	MatchingDissimilarity	69	SokalSneathDissimilarity	69
DiceDissimilarity . . .	68	Multinomial	69	Subsets	70
Fibonacci	68	RogersTanimotoDissimilarity	69	YuleDissimilarity . . .	70
JaccardDissimilarity . .	68	RussellRaoDissimilarity	69		

Binomial

`Binomial[n, k]`
gives the binomial coefficient n choose k .

```
>> Binomial[5, 3]
10
```

Binomial supports inexact numbers:

```
>> Binomial[10.5, 3.2]
165.286
```

Some special cases:

```
>> Binomial[10, -2]
0

>> Binomial[-10.5, -3.5]
0.
```

DiceDissimilarity

`DiceDissimilarity[u, v]`
returns the Dice dissimilarity between the two boolean 1-D lists u and v , which is defined as $(c_{tf} + c_{ft}) / (2 * c_{tt} + c_{ft} + c_{tf})$, where n is $\text{len}(u)$ and c_{ij} is the number of occurrences of $u[k]=i$ and $v[k]=j$ for $k < n$.

```
>> DiceDissimilarity[{1, 0, 1, 1, 0, 1, 1}, {0, 1, 1, 0, 0, 0, 1}]
1/2
```

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
```

JaccardDissimilarity

`JaccardDissimilarity[u, v]`
returns the Jaccard-Needham dissimilarity between the two boolean 1-D lists u and v , which is defined as $(c_{tt} + c_{ft} + c_{tf}) / (c_{tt} + c_{ft} + c_{tf})$, where n is $\text{len}(u)$ and c_{ij} is the number of occurrences of $u[k]=i$ and $v[k]=j$ for $k < n$.

```
>> JaccardDissimilarity[{1, 0, 1, 1, 0, 1, 1}, {0, 1, 1, 0, 0, 0, 1}]
2/3
```

MatchingDissimilarity

`MatchingDissimilarity[u, v]`
returns the Matching dissimilarity between the two boolean 1-D lists u and v , which is defined as $(c_tf + c_ft) / n$, where n is $\text{len}(u)$ and c_ij is the number of occurrences of $u[k]=i$ and $v[k]=j$ for $k < n$.

```
>> MatchingDissimilarity[{1, 0, 1, 1, 0, 1, 1}, {0, 1, 1, 0, 0, 0, 1}]  
 $\frac{4}{7}$ 
```

Multinomial

`Multinomial[n1, n2, ...]`
gives the multinomial coefficient $(n1 + n2 + \dots) / (n1! n2! \dots)$.

```
>> Multinomial[2, 3, 4, 5]  
2522520
```

```
>> Multinomial[]  
1
```

Multinomial is expressed in terms of Binomial:

```
>> Multinomial[a, b, c]  
Binomial[a, a] Binomial[a + b, b] Binomial[a + b + c, c]
```

`Multinomial[n-k, k]` is equivalent to `Binomial[n, k]`.

```
>> Multinomial[2, 3]  
10
```

RogersTanimotoDissimilarity

`RogersTanimotoDissimilarity[u, v]`
returns the Rogers-Tanimoto dissimilarity between the two boolean 1-D lists u and v , which is defined as $R / (c_tt + c_ff + R)$ where n is $\text{len}(u)$, c_ij is the number of occurrences of $u[k]=i$ and $v[k]=j$ for $k < n$, and $R = 2 * (c_tf + c_ft)$.

```
>> RogersTanimotoDissimilarity[{1, 0, 1, 1, 0, 1, 1}, {0, 1, 1, 0, 0, 0, 1}]  
 $\frac{8}{11}$ 
```

RussellRaoDissimilarity

`RussellRaoDissimilarity[u, v]`
returns the Russell-Rao dissimilarity between the two boolean 1-D lists u and v , which is defined as $(n - c_tt) / c_tt$ where n is $\text{len}(u)$ and c_ij is the number of occurrences of $u[k]=i$ and $v[k]=j$ for $k < n$.

```
>> RussellRaoDissimilarity[{1, 0, 1, 1, 0, 1, 1}, {0, 1, 1, 0, 0, 0, 1}]  
 $\frac{5}{7}$ 
```

SokalSneathDissimilarity

`SokalSneathDissimilarity[u, v]`
returns the Sokal-Sneath dissimilarity between the two boolean 1-D lists u and v , which is defined as $R / (c_tt + R)$ where n is $\text{len}(u)$, c_ij is the number of occurrences of $u[k]=i$ and $v[k]=j$ for $k < n$, and $R = 2 * (c_tf + c_ft)$.

```
>> SokalSneathDissimilarity[{1, 0, 1, 1, 0, 1, 1}, {0, 1, 1, 0, 0, 0, 1}]  
 $\frac{4}{5}$ 
```

Subsets

```
Subsets[list]
    finds a list of all possible subsets of list.
Subsets[list, n]
    finds a list of all possible subsets containing at most n elements.
Subsets[list, {n}]
    finds a list of all possible subsets containing exactly n elements.
Subsets[list, {min, max}]
    finds a list of all possible subsets containing between min and max elements.
Subsets[list, spec, n]
    finds a list of the first n possible subsets.
Subsets[list, spec, {n}]
    finds the nth possible subset.
```

All possible subsets (power set):

```
>> Subsets[{a, b, c}]
{{}, {a}, {b}, {c}, {a,
  b}, {a, c}, {b, c}, {a, b, c}}
```

All possible subsets containing up to 2 elements:

```
>> Subsets[{a, b, c, d}, 2]
{{}, {a}, {b}, {c}, {d},
 {a, b}, {a, c}, {a, d},
 {b, c}, {b, d}, {c, d}}
```

Subsets containing exactly 2 elements:

```
>> Subsets[{a, b, c, d}, {2}]
{{a, b}, {a, c}, {a, d},
 {b, c}, {b, d}, {c, d}}
```

The first 5 subsets containing 3 elements:

```
>> Subsets[{a, b, c, d, e}, {3}, 5]
{{a, b, c}, {a, b, d}, {a,
  b, e}, {a, c, d}, {a, c, e}}
```

All subsets with even length:

```
>> Subsets[{a, b, c, d, e}, {0, 5,
  2}]
{{}, {a, b}, {a, c}, {a, d}, {a, e},
 {b, c}, {b, d}, {b, e}, {c, d}, {c,
  e}, {d, e}, {a, b, c, d}, {a, b, c, e},
 {a, b, d, e}, {a, c, d, e}, {b, c, d, e}}
```

The 25th subset:

```
>> Subsets[Range[5], All, {25}]
{{2, 4, 5}}
```

The odd-numbered subsets of {a,b,c,d} in reverse order:

```
>> Subsets[{a, b, c, d}, All, {15,
  1, -2}]
{{b, c, d}, {a, b, d}, {c, d},
 {b, c}, {a, c}, {d}, {b}, {}}
```

YuleDissimilarity

```
YuleDissimilarity[u, v]
    returns the Yule dissimilarity between the two boolean 1-D lists u and v, which is defined as  $R / (c_{tt} * c_{ff} + R / 2)$  where n is len(u), c_ij is the number of occurrences of u[k]=i and v[k]=j for k<n, and  $R = 2 * c_{tf} * c_{ft}$ .
```

```
>> YuleDissimilarity[{1, 0, 1, 1,
  0, 1, 1}, {0, 1, 1, 0, 0, 0, 1}]
6
5
```

VII. Compilation

Contents

Compile	71	CompiledCodeBox . .	71	CompiledFunction . .	71
-------------------	----	---------------------	----	----------------------	----

Compile

`Compile[{x1, x2, ...}, expr_]`
Compiles *expr* assuming each *xi* is a *Real* number.
`Compile[{x1, t1} {x2, t1} ..., expr_]`
Compiles assuming each *xi* matches type *ti*.

```
>> cf = Compile[{x, y}, x + 2 y]
CompiledFunction[{x, y},
  x + 2y, -CompiledCode-]

>> cf[2.5, 4.3]
11.1

>> cf = Compile[{x, _Real}}, Sin[x]]
CompiledFunction[{x},
  Sin[x], -CompiledCode-]

>> cf[1.4]
0.98545
```

Compile supports basic flow control

```
>> cf = Compile[{x, _Real}, {y,
  _Integer}}, If[x == 0.0 && y <=
  0, 0.0, Sin[x ^ y] + 1 / Min[x,
  0.5]] + 0.5]

CompiledFunction[{x,
  y}, 0.5 + If[x == 0.0 && y <= 0,
  0., Sin[x^y] + 1 / Min[x, 0.5]],
  -CompiledCode-]
```

```
>> cf[3.5, 2]
2.18888
```

Loops and variable assignments are not yet supported

```
>> Compile[{a, _Integer}, {b,
  _Integer}}, While[b != 0, {a, b}
  = {b, Mod[a, b]}; a] (* GCD of
  a, b *)
```

ExpressionWhile[b!=0, {a,b} = {b, Mod[a,b]}; a could not be compiled.

```
Compile[{a, _Integer},
  {b, _Integer}}, While[b!=0,
  {a, b} = {b, Mod[a, b]}; a]
```

CompiledCodeBox

CompiledFunction

VIII. Comparison

Contents

BooleanQ	72	LessEqual (<=)	73	SameQ (===)	75
Equal (==)	72	Max	73	SympyComparison	75
Greater (>)	73	Min	74	TrueQ	75
GreaterEqual (>=)	73	Negative	74	Unequal (!=)	75
Inequality	73	NonNegative	74	UnsameQ (!=)	75
Less (<)	73	NonPositive	74	ValueQ	75
		Positive	74		

BooleanQ

`BooleanQ[expr]`
returns True if *expr* is either True or False.

```
>> BooleanQ[True]
True

>> BooleanQ[False]
True

>> BooleanQ[a]
False

>> BooleanQ[1 < 2]
True
```

Equal (==)

`Equal[x, y]`
`x == y`
yields True if *x* and *y* are known to be equal, or False if *x* and *y* are known to be unequal.
`lhs == rhs`
represents the equation *lhs* = *rhs*.

```
>> 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
```

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 (>)

```
Greater[x, y]
x > y
  yields True if  $x$  is known to be greater
  than  $y$ .
lhs > rhs
  represents the inequality  $lhs > rhs$ .
```

```
>> a > b > c //FullForm
Greater[a, b, c]

>> Greater[3, 2, 1]
True
```

GreaterEqual (>=)

```
GreaterEqual[x, y]
x >= y
  yields True if  $x$  is known to be greater
  than or equal to  $y$ .
lhs >= rhs
  represents the inequality  $lhs \geq rhs$ .
```

Inequality

```
Inequality
  is the head of expressions involving dif-
  ferent inequality operators (at least tem-
  porarily). 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 (<)

```
Less[x, y]
x < y
  yields True if  $x$  is known to be less than  $y$ .
lhs < rhs
  represents the inequality  $lhs < rhs$ .
```

LessEqual (<=)

```
LessEqual[x, y]
x <= y
  yields True if  $x$  is known to be less than
  or equal to  $y$ .
lhs <= rhs
  represents the inequality  $lhs \leq rhs$ .
```

Max

```
Max[e1, e2, ..., ei]
  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 sym-
bolic form:

```
>> Max[x, y]
Max[x, y]

>> Max[5, x, -3, y, 40]
Max[40, x, y]
```

With no arguments, Max gives -Infinity:

```
>> Max[]
-∞
```

Min

```
Min[e1, e2, ..., ei]  
returns the expression with the lowest  
value among the ei.
```

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}}]  
-∞
```

Min with symbolic arguments remains in symbolic form:

```
>> Min[x, y]  
Min[x, y]
```

```
>> Min[5, x, -3, y, 40]  
Min[-3, x, y]
```

With no arguments, Min gives Infinity:

```
>> Min[]  
∞
```

Negative

```
Negative[x]  
returns True if x is a negative real number.
```

```
>> Negative[0]  
False
```

```
>> Negative[-3]  
True
```

```
>> Negative[10/7]  
False
```

```
>> Negative[1+2I]  
False
```

```
>> Negative[a + b]  
Negative[a + b]
```

NonNegative

```
NonNegative[x]  
returns True if x is a positive real number  
or zero.
```

```
>> {Positive[0], NonNegative[0]}  
{False, True}
```

NonPositive

```
NonPositive[x]  
returns True if x is a negative real number  
or zero.
```

```
>> {Negative[0], NonPositive[0]}  
{False, True}
```

Positive

```
Positive[x]  
returns True if x is a positive real number.
```

```
>> Positive[1]  
True
```

Positive returns False if x is zero or a complex number:

```
>> Positive[0]  
False
```

```
>> Positive[1 + 2 I]  
False
```

SameQ (===)

```
SameQ[x, y]  
x === y  
returns True if x and y are structurally  
identical.
```

Any object is the same as itself:

```
>> a===a  
True
```

Unlike Equal, SameQ only yields True if x and y have the same type:

```
>> {1==1., 1==1.}
{True, False}
```

SympyComparison

TrueQ

```
TrueQ[expr]
returns True if and only if expr is True.
```

```
>> TrueQ[True]
True

>> TrueQ[False]
False

>> TrueQ[a]
False
```

Unequal (!=)

```
Unequal[x, y]
x != y
yields False if x and y are known to be
equal, or True if x and y are known to be
unequal.
lhs == rhs
represents the inequality lhs ≠ rhs.
```

```
>> 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 (!=)

```
UnsameQ[x, y]
x != y
returns True if x and y are not structurally
identical.
```

```
>> a!=a
False

>> 1!=1.
True
```

ValueQ

```
ValueQ[expr]
returns True if and only if expr is defined.
```

```
>> ValueQ[x]
False

>> x = 1;

>> ValueQ[x]
True
```

IX. Control statements

Contents

Abort	76	FixedPoint	77	NestList	79
Break	76	FixedPointList	77	NestWhile	79
CompoundExpression (;)	76	For	78	Return	79
Continue	76	If	78	Switch	79
Do	77	Interrupt	78	Which	80
		Nest	78	While	80

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, ...]`
`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]
0.739085

>> 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., 0.540302, 0.857~
~553, 0.65429, 0.79348}
```

Observe the convergence of Newton's method for approximating square roots:

```
>> newton[n_] := FixedPointList[.5(# + n/#)&, 1.];

>> newton[9]
{1., 5., 3.4, 3.02353, 3.00009, 3., 3., 3.}
```

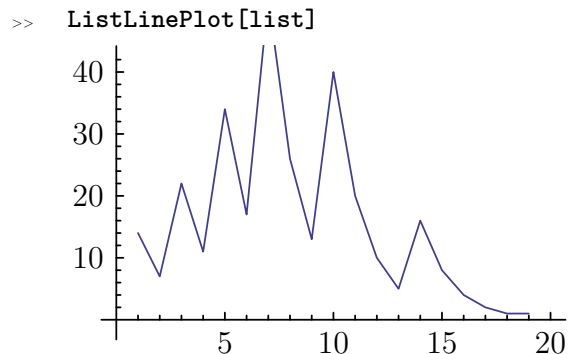
Plot the "hailstone" sequence of a number:

```
>> 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}
```



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];
```

Interrupt

`Interrupt[]`
Interrupt an evaluation and returns
\$Aborted.

```
>> Print["a"]; Interrupt[]; Print["
b"]
a
$Aborted
```

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]
(1 + (1 + x)2)2
```

NestList

`NestList[f, expr, n]`
starting with *expr*, iteratively applies *f* *n*
times and returns a list of all intermedi-
ate results.

```
>> NestList[f, x, 3]
{x, f[x], f[f[x]], f[f[f[x]]]}

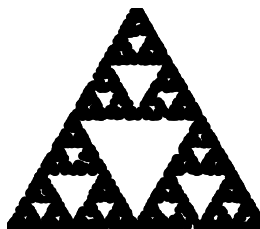
>> NestList[2 # &, 1, 8]
{1, 2, 4, 8, 16, 32, 64, 128, 256}
```

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]
625
 2
```

Return

`Return[expr]`
aborts a function call and returns *expr*.

```
>> f[x_] := (If[x < 0, Return[0]];
x)

>> f[-1]
0

>> Do[If[i > 3, Return[]]; Print[i], {i, 10}]
1
2
3
```

Return only exits from the innermost control flow construct.

```
>> g[x_] := (Do[If[x < 0, Return
[0]], {i, {2, 1, 0, -1}}]; x)
```

```
>> g[-1]
-1
```

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, True, c]
```

Which must be called with an even number of arguments:

```
>> Which[a, b, c]
Whichcalledwith3arguments.
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
```


X. Date and Time

Contents

AbsoluteTime	81	DateString	83	\$SystemTimeZone	83
AbsoluteTiming	81	\$DateStringFormat	83	TimeUsed	84
DateDifference	82	EasterSunday	83	\$TimeZone	84
DateList	82	Pause	83	Timing	84
DatePlus	82	SessionTime	83		

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[]
3.81135 × 109

>> AbsoluteTime[{2000}]
3 155 673 600

>> AbsoluteTime[{"01/02/03", {"Day",
    "Month", "YearShort"}}]
3 253 046 400

>> AbsoluteTime["6 June 1991"]
2885 155 200

>> AbsoluteTime[{"6-6-91", {"Day",
    "Month", "YearShort"}}]
2885 155 200

```

AbsoluteTiming

```

AbsoluteTiming[expr]
    measures the actual time it takes to eval-
    uate expr. It returns a list containing the
    measured time in seconds and the result
    of the evaluation.

>> AbsoluteTiming[50!]
{0.000229359, 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]
    returns the difference between date1 and
    date2 in days.
DateDifference[date1, date2, unit]
    returns the difference in the specified
    unit.
DateDifference[date1, date2, {unit1,
unit2, ...}]
    represents the difference as a list of inte-
    ger multiples of each unit, with any re-
    mainder expressed in the smallest unit.

```

```
>> DateDifference[{2042, 1, 4},
{2057, 1, 1}]
5476

>> DateDifference[{1936, 8, 14},
{2000, 12, 1}, "Year"]
{64.3425, Year}

>> DateDifference[{2010, 6, 1},
{2015, 1, 1}, "Hour"]
{40200, 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]
{1900, 1, 1, 0, 0, 0.}

>> DateList[3155673600]
{2000, 1, 1, 0, 0, 0.}

>> DateList[{2003, 5, 0.5, 0.1,
0.767}]
{2003, 4, 30, 12, 6, 46.02}

>> DateList[{2012, 1, 300., 10}]
{2012, 10, 26, 10, 0, 0.}

>> DateList["31/10/1991"]
{1991, 10, 31, 0, 0, 0.}
```

```
>> DateList["1/10/1991"]
The interpretation of 1/10/
1991 is ambiguous.
{1991, 1, 10, 0, 0, 0.}

>> DateList[{"31/10/91", {"Day", "
Month", "YearShort"}}]
{1991, 10, 31, 0, 0, 0.}

>> DateList[{"31 10/91", {"Day", "
", "Month", "/"}, "YearShort"}}]
{1991, 10, 31, 0, 0, 0.}
```

If not specified, the current year assumed

```
>> DateList[{"5/18", {"Month", "Day
"}}]
{2020, 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 1991

>> 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}
```

EasterSunday

`EasterSunday[year]`
returns the date of the Gregorian Easter Sunday as {year, month, day}.

```
>> EasterSunday[2000]
{2000, 4, 23}

>> EasterSunday[2030]
{2030, 4, 21}
```

Pause

`Pause[n]`
pauses for *n* seconds.

```
>> Pause[0.5]
```

SessionTime

`SessionTime[]`
returns the total time in seconds since this session started.

```
>> SessionTime[]
178.848
```

\$SystemTimeZone

`$SystemTimeZone`
gives the current time zone for the computer system on which Mathics is being run.

```
>> $SystemTimeZone
-5.
```

TimeUsed

`TimeUsed[]`
returns the total CPU time used for this session, in seconds.

```
>> TimeUsed[]  
179.247
```

\$TimeZone

`$TimeZone`
gives the current time zone to assume for dates and times.

```
>> $TimeZone  
-5.
```

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!]  
{0.00026875, 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}
```

XI. Differential equation solver functions

Contents

C	85	DSolve	85
-------------	----	------------------	----

C

`C[n]`

represents the n th constant in a solution to a differential equation.

```
>> DSolve[D[y[x, t], t] + 2 D[y[x,
t], x] == 0, y[x, t], {x, t}]
{{y[x, t] -> C[1] [-2t + x]}}
```

DSolve

`DSolve[eq, y[x], x]`

solves a differential equation for the function $y[x]$.

```
>> DSolve[y''[x] == 0, y[x], x]
{{y[x] -> x C[2] + C[1]}}

>> DSolve[y''[x] == y[x], y[x], x]
{{y[x] -> C[1] E^-x + C[2] E^x}}

>> DSolve[y''[x] == y[x], y, x]
{{y -> (Function[{x},
C[1] E^-x + C[2] E^x])}}
```

DSolve can also solve basic PDE

```
>> DSolve[D[f[x, y], x] / f[x, y] +
3 D[f[x, y], y] / f[x, y] == 2,
f, {x, y}]
{{f -> (Function[{x, y},
E^(x/5 + 3y/5) C[1] [3x - y])}]}
```

```
>> DSolve[D[f[x, y], x] x + D[f[x,
y], y] y == 2, f[x, y], {x, y}]
{{f[x, y] -> 2 Log[x] + C[1] [y/x]}}
```

XII. Evaluation

Contents

Evaluate	86	HoldForm	87	\$RecursionLimit	88
Exit	86	In	87	ReleaseHold	88
\$HistoryLength	86	\$IterationLimit	87	Sequence	89
Hold	87	\$Line	87	Unevaluated	89
HoldComplete	87	Out	88		

Evaluate

`Evaluate[expr]`
forces evaluation of *expr*, even if it occurs inside a held argument or a Hold form.

Create a function *f* with a held argument:

```
>> SetAttributes[f, HoldAll]
```

```
>> f[1 + 2]
f[1 + 2]
```

Evaluate forces evaluation of the argument, even though *f* has the HoldAll attribute:

```
>> 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`
specifies the maximum number of In and Out entries.

```
>> $HistoryLength
100
>> $HistoryLength = 1;
>> 42
42
>> %
42
>> %%
%3
>> $HistoryLength = 0;
>> 42
42
>> %
%7
```

Exit

`Exit[]`
terminates the Mathics session.
`Exit[n]`
terminates with exit code *n*.

Exit is an alias for Quit.

Hold

`Hold[expr]`
prevents *expr* from being evaluated.

```
>> Attributes[Hold]
{HoldAll, Protected}
```

HoldComplete

`HoldComplete[expr]`
prevents *expr* from being evaluated, and also prevents Sequence objects from being spliced into argument lists.

```
>> Attributes[HoldComplete]
{HoldAllComplete, Protected}
```

HoldForm

`HoldForm[expr]`
is equivalent to `Hold[expr]`, but prints as *expr*.

```
>> HoldForm[1 + 2 + 3]
1 + 2 + 3
```

HoldForm has attribute HoldAll:

```
>> Attributes[HoldForm]
{HoldAll, Protected}
```

In

`In[k]`
gives the *k*th line of input.

```
>> x = 1
1

>> x = x + 1
2

>> Do[In[2], {3}]

>> x
5

>> In[-1]
5

>> Definition[In]
Attributes[In] = {Protected}
In[6] = Definition[In]
In[5] = In[-1]
In[4] = x
In[3] = Do[In[2], {3}]
In[2] = x = x + 1
In[1] = x = 1
```

\$IterationLimit

`$IterationLimit`
specifies the maximum number of times a reevaluation of an expression may happen.

Calculations terminated by `$IterationLimit` return `$Aborted`:
> `$IterationLimit = 1000`

\$Line

`$Line`
holds the current input line number.

```
>> $Line
1

>> $Line
2

>> $Line = 12;

>> 2 * 5
10

>> Out[13]
10

>> $Line = -1;
Non - negative integer expected.
```

Out

`Out[k]`
%k
gives the result of the *k*th input line.
%, %%, etc.
gives the result of the previous input line, of the line before the previous input line, etc.

```
>> 42
42

>> %
42

>> 43;
```

```
>> %
43

>> 44
44

>> %1
42

>> %%
44

>> Hold[Out[-1]]
Hold[%]

>> Hold[%4]
Hold[%4]

>> Out[0]
Out[0]
```

\$RecursionLimit

\$RecursionLimit
specifies the maximum allowable recursion depth after which a calculation is terminated.

Calculations terminated by \$RecursionLimit return \$Aborted:

```
>> a = a + a
Recursiondepthof200exceeded.
$Aborted

>> $RecursionLimit
200

>> $RecursionLimit = x;
Cannotset$RecursionLimittox;valuemustbeanintegerbetween20and512;usetheMATHICS_MAX_RECURSION_DEPTH

>> $RecursionLimit = 512
512

>> a = a + a
Recursiondepthof512exceeded.
$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[x, a, b, y]

>> 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

`Unevaluated[expr]`
temporarily leaves *expr* in an unevaluated form when it appears as a function argument.

`Unevaluated` is automatically removed when function arguments are evaluated:

```
>> Sqrt[Unevaluated[x]]
       $\sqrt{x}$ 
```

```
>> 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, Unevaluated[b], Unevaluated[c]]
```

However, unevaluated sequences are kept:

```
>> g[Unevaluated[Sequence[a, b, c]]]
      g[Unevaluated[Sequence[a, b, c]]]
```

XIII. Exponential, trigonometric and hyperbolic functions

Mathics basically supports all important trigonometric and hyperbolic functions. Numerical values and derivatives can be computed; however, most special exact values and simplification rules are not implemented yet.

Contents

AnglePath	91	ArcTan	93	InverseHaversine	95
AngleVector	91	ArcTanh	93	Log	95
ArcCos	91	Cos	93	Log10	95
ArcCosh	92	Cosh	93	Log2	95
ArcCot	92	Cot	93	LogisticSigmoid	95
ArcCoth	92	Coth	93	Pi	96
ArcCsc	92	Csc	94	Sec	96
ArcCsch	92	Csch	94	Sech	96
ArcSec	92	Degree	94	Sin	96
ArcSech	92	E	94	Sinh	96
ArcSin	93	Exp	94	Tan	96
ArcSinh	93	GoldenRatio	94	Tanh	96
		Haversine	95		

AnglePath

AnglePath[*{phi1, phi2, ...}*]
 returns the points formed by a turtle starting at {0, 0} and angled at 0 degrees going through the turns given by angles *phi1, phi2, ...* and using distance 1 for each step.

AnglePath[*{{r1, phi1}, {r2, phi2}, ...}*]
 instead of using 1 as distance, use *r1, r2, ...* as distances for the respective steps.

AngleVector[*phi0, {phi1, phi2, ...}*]
 returns the points on a path formed by a turtle starting with direction *phi0* instead of 0.

AngleVector[*{x, y}, {phi1, phi2, ...}*]
 returns the points on a path formed by a turtle starting at {*x, y*} instead of {0, 0}.

AngleVector[*{{x, y}, phi0}, {phi1, phi2, ...}*]
 specifies initial position {*x, y*} and initial direction *phi0*.

AngleVector[*{{x, y}, {dx, dy}}, {phi1, phi2, ...}*]
 specifies initial position {*x, y*} and a slope {*dx, dy*} that is understood to be the initial direction of the turtle.

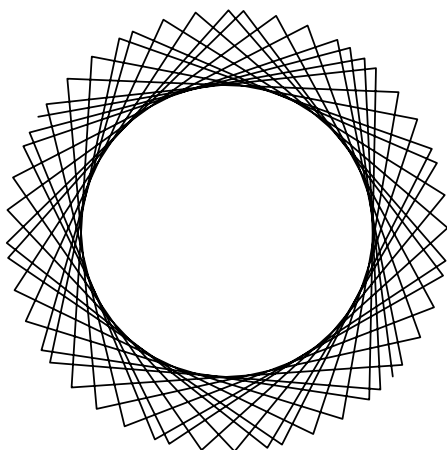
```
>> AnglePath[{90 Degree, 90 Degree,
  90 Degree, 90 Degree}]
{{0,0}, {0,1}, {-1,
  1}, {-1,0}, {0,0}}

>> AnglePath[{{1, 1}, 90 Degree},
  {{1, 90 Degree}, {2, 90 Degree},
  {1, 90 Degree}, {2, 90 Degree
  }}]
{{1,1}, {0,1}, {0,
  -1}, {1, -1}, {1,1}}

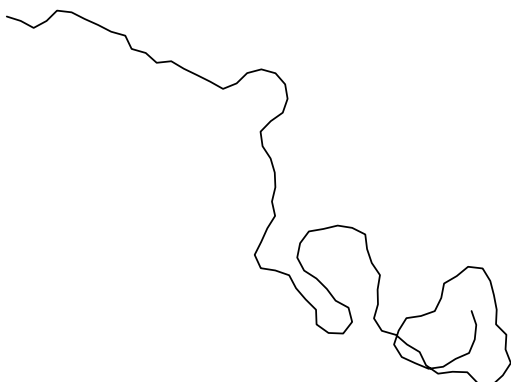
>> AnglePath[{a, b}]
{{0,0}, {Cos[a], Sin[a]}, {Cos[
  a]+Cos[a+b], Sin[a]+Sin[a+b]}}

>> Precision[Part[AnglePath[{N[1/3,
  100], N[2/3, 100]}], 2, 1]]
100.

>> Graphics[Line[AnglePath[Table
  [1.7, {50}]]]]
```



```
>> Graphics[Line[AnglePath[
  RandomReal[{-1, 1}, {100}]]]]
```



AngleVector

`AngleVector[phi]`
returns the point at angle ϕ on the unit circle.

`AngleVector[{r, phi}]`
returns the point at angle ϕ on a circle of radius r .

`AngleVector[{x, y}, phi]`
returns the point at angle ϕ on a circle of radius 1 centered at $\{x, y\}$.

`AngleVector[{x, y}, {r, phi}]`
returns point at angle ϕ on a circle of radius r centered at $\{x, y\}$.

```
>> AngleVector[90 Degree]
{0, 1}

>> AngleVector[{1, 10}, a]
{1 + Cos[a], 10 + Sin[a]}
```

ArcCos

`ArcCos[z]`
returns the inverse cosine of z .

```
>> ArcCos[1]
0

>> ArcCos[0]
Pi/2

>> Integrate[ArcCos[x], {x, -1, 1}]
Pi
```

ArcCosh

`ArcCosh[z]`
returns the inverse hyperbolic cosine of z .

```
>> ArcCosh[0]
I/2 Pi

>> ArcCosh[0.]
0. + 1.5708 I
```

ArcCot

```
>> ArcCot[0]
      Pi
      2

>> ArcCot[1]
      Pi
      4
```

ArcCoth

```
>> ArcCoth[0]
       $\frac{I}{2}\text{Pi}$ 
>> ArcCoth[1]
       $\infty$ 
>> ArcCoth[0.0]
       $0. + 1.5708I$ 
>> ArcCoth[0.5]
       $0.549306 - 1.5708I$ 
```

ArcCsc

```
>> ArcCsc[1]
      Pi
      --
      2
```

ArcCsch

```
>> ArcCsch[0]
ComplexInfinity
>> ArcCsch[1.0]
0.881374
```

ArcSec

```
>> ArcSec[1]
0
>> ArcSec[-1]
Pi
```

ArcSech

```
>> ArcSech[0]
      ∞
>> ArcSech[1]
      0
>> ArcSech[0.5]
      1.31696
```

ArcSin

```
>> ArcSin[0]
0
```

```
>> ArcSin[1]
      Pi
      2
```

ArcSinh

`ArcSinh[z]`
returns the inverse hyperbolic sine of z.

```
>> ArcSinh[0]
0

>> ArcSinh[0.]
0.

>> ArcSinh[1.0]
0.881374
```

ArcTan

`ArcTan[z]`
returns the inverse tangent of z.

```
>> ArcTan[1]
      Pi
      4

>> ArcTan[1.0]
0.785398

>> ArcTan[-1.0]
-0.785398

>> ArcTan[1, 1]
      Pi
      4
```

ArcTanh

`ArcTanh[z]`
returns the inverse hyperbolic tangent of z.

```
>> ArcTanh[0]
0

>> ArcTanh[1]
∞

>> ArcTanh[0]
0
```

```
>> ArcTanh[.5 + 2 I]
0.0964156 + 1.12656 I

>> ArcTanh[2 + I]
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.]
0.642093
```

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 *)
 $\frac{1}{\sin[1]}$ 

>> Csc[1.]
1.1884
```

Csch

Csch[z]
returns the hyperbolic cosecant of z.

```
>> Csch[0]
ComplexInfinity
```

Degree

Degree
is the number of radians in one degree.

```
>> Cos[60 Degree]
 $\frac{1}{2}$ 
```

Degree has the value of $\text{Pi} / 180$

```
>> Degree == Pi / 180
True
```

E

E
is the constant e.

```
>> N[E]
2.71828
```

```
>> N[E, 50]
2.718281828459045235360287~
~4713526624977572470937000

>> Attributes[E]
{Constant, Protected, ReadProtected}
```

Exp

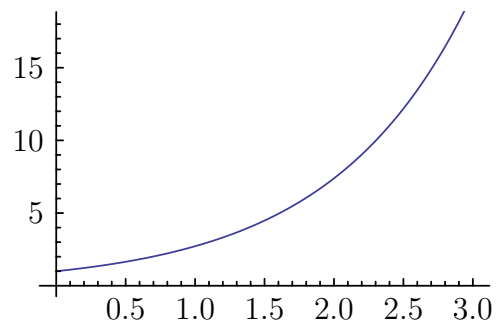
Exp[z]
returns the exponential function of z.

```
>> Exp[1]
E

>> Exp[10.0]
22026.5

>> Exp[x] //FullForm
Power[E, x]

>> Plot[Exp[x], {x, 0, 3}]
```



GoldenRatio

GoldenRatio
is the golden ratio.

```
>> N[GoldenRatio]
1.61803
```

Haversine

Haversine[z]
returns the haversine function of z.

```
>> Haversine[1.5]
0.464631
```

```
>> Haversine[0.5 + 2I]
-1.15082 + 0.869405I
```

InverseHaversine

Haversine[z]
returns the inverse haversine function of z.

```
>> InverseHaversine[0.5]
1.5708

>> InverseHaversine[1 + 2.5 I]
1.76459 + 2.33097I
```

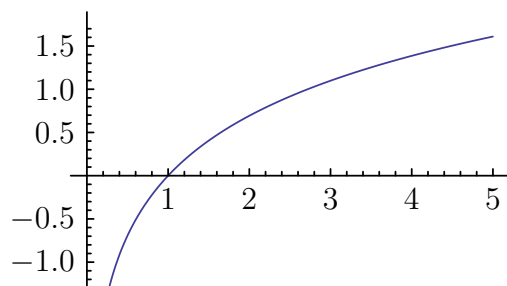
Log

Log[z]
returns the natural logarithm of z.

```
>> Log[{0, 1, E, E * E, E ^ 3, E ^ x}]
{-∞, 0, 1, 2, 3, Log[E^x]}

>> 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.}]
{0.30103, 0.69897}
```

```
>> Log10[E ^ 3]
3
Log[10]
```

Log2

Log2[z]
returns the base-2 logarithm of z.

```
>> Log2[4 ^ 8]
16

>> Log2[5.6]
2.48543

>> Log2[E ^ 2]
2
Log[2]
```

LogisticSigmoid

LogisticSigmoid[z]
returns the logistic sigmoid of z.

```
>> LogisticSigmoid[0.5]
0.622459

>> LogisticSigmoid[0.5 + 2.3 I]
1.06475 + 0.808177I

>> LogisticSigmoid[{-0.2, 0.1, 0.3}]
{0.450166, 0.524979, 0.574443}
```

Pi

Pi
is the constant π .

```
>> N[Pi]
3.14159

>> N[Pi, 50]
3.141592653589793238462643~
~3832795028841971693993751
```

```
>> Attributes[Pi]
{Constant, Protected, ReadProtected}
```

Sec

Sec[z]
returns the secant of z.

```
>> Sec[0]
1

>> Sec[1] (* Sec[1] in Mathematica *)

$$\frac{1}{\cos[1]}$$


>> Sec[1.]
1.85082
```

Sech

Sech[z]
returns the hyperbolic secant of z.

```
>> Sech[0]
1
```

Sin

Sin[z]
returns the sine of z.

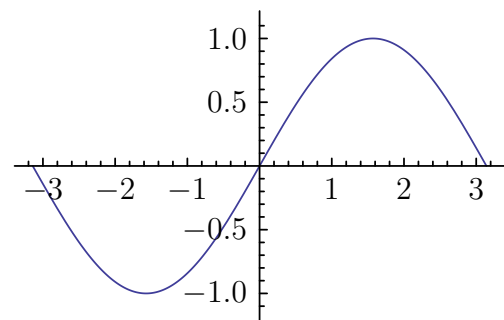
```
>> Sin[0]
0

>> Sin[0.5]
0.479426

>> Sin[3 Pi]
0

>> Sin[1.0 + I]
1.29846 + 0.634964I
```

```
>> 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
```


XIV. Functional programming

Contents

Composition	97	Identity	98	SlotSequence	98
Function (&)	97	Slot	98		

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]`
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]
x^y

>> Function[x, Function[y, x^y]][x]
x^y
```

Slots in inner functions are not affected by outer function application:

```
>> g[#] & [h[#]] & [5]
g[h[5]]
```

Identity

`Identity[x]`
is the identity function, which returns x unchanged.

```
>> Identity[x]
x
>> Identity[x, y]
Identity[x, y]
```

Slot

`# n`
represents the n th argument to a pure function.
`#`
is short-hand for `#1`.
`#0`
represents the pure function itself.

```
>>
```

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 n th argument.

```
>> Plus[##]& [1, 2, 3]
6
>> Plus[##2]& [1, 2, 3]
5
>> FullForm[##]
SlotSequence[1]
```

XV. Graphics

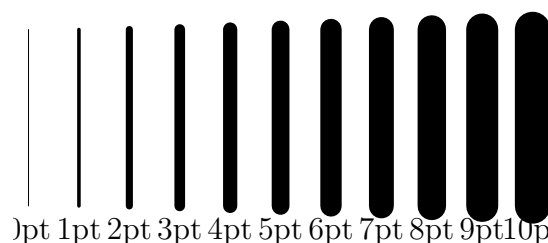
Contents

AbsoluteThickness . . .	99	FaceForm	106	Orange	111
Arrow	100	FilledCurve	106	Point	112
ArrowBox	101	FilledCurveBox	106	PointBox	112
Arrowheads	101	FontColor	106	PointSize	112
Automatic	102	Graphics	107	Polygon	112
BernsteinBasis	102	GraphicsBox	107	PolygonBox	112
BezierCurve	102	Gray	108	Purple	113
BezierCurveBox	102	GrayLevel	108	RGBColor	113
BezierFunction	102	Green	108	Rectangle	114
Black	102	Hue	108	RectangleBox	114
Blend	103	Inset	109	Red	114
Blue	103	InsetBox	109	RegularPolygon	114
CMYKColor	103	LABColor	109	RegularPolygonBox . .	115
Circle	104	LCHColor	109	Small	115
CircleBox	104	LUVColor	109	Text	115
ColorDistance	104	Large	109	Thick	115
Cyan	104	LightRed	109	Thickness	115
Darker	104	Lighter	110	Thin	115
Directive	105	Line	110	Tiny	115
Disk	105	LineBox	110	White	116
DiskBox	106	Magenta	111	XYZColor	116
EdgeForm	106	Medium	111	Yellow	116
		Offset	111		

AbsoluteThickness

`AbsoluteThickness[p]`
sets the line thickness for subsequent graphics primitives to *p* points.

```
>> Graphics[Table[{
  AbsoluteThickness[t], Line[{{20
t, 10}, {20 t, 80}}], Text[
ToString[t]<>"pt", {20 t, 0}],
{t, 0, 10}]]
```



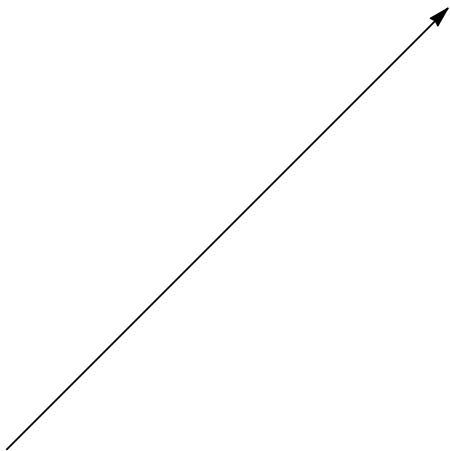
Arrow

`Arrow[{p1, p2}]`
 represents a line from $p1$ to $p2$ that ends with an arrow at $p2$.

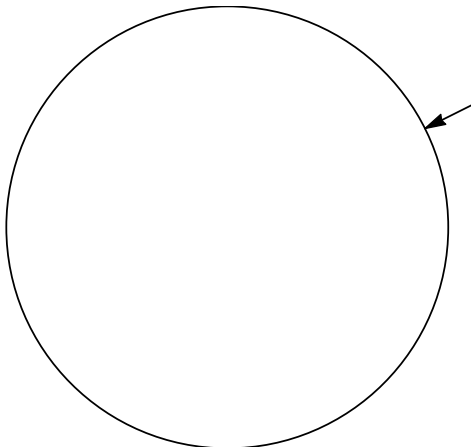
`Arrow[{p1, p2}, s]`
 represents a line with arrow that keeps a distance of s from $p1$ and $p2$.

`Arrow[{point_1, point_2}, {s1, s2}]`
 represents a line with arrow that keeps a distance of $s1$ from $p1$ and a distance of $s2$ from $p2$.

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

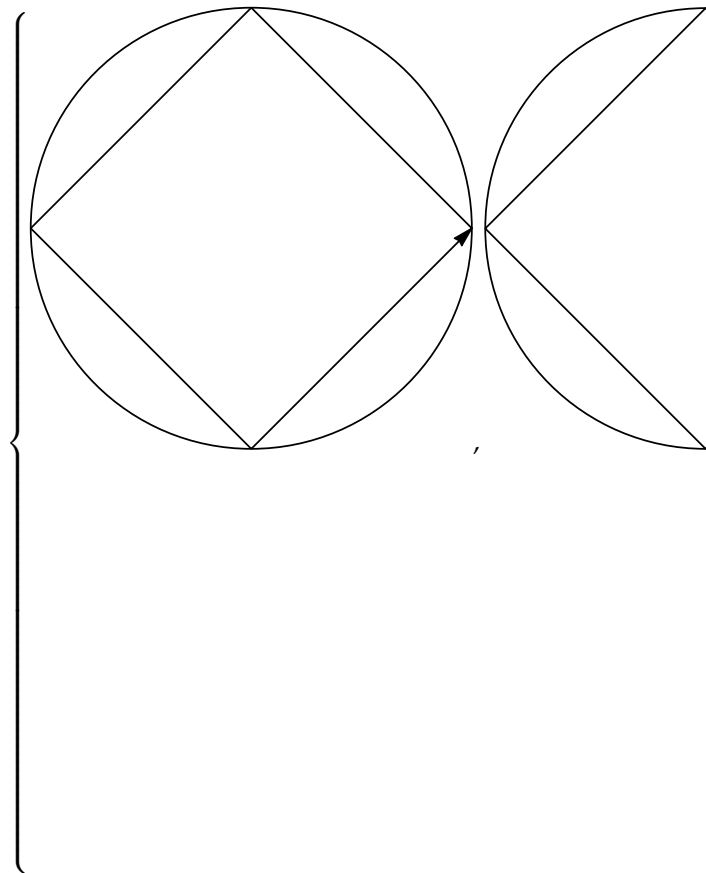


```
>> Graphics[{Circle[], Arrow[{2,1}, {0,0}, 1]}]
```



Keeping distances may happen across multiple segments:

```
>> Table[Graphics[{Circle[], Arrow[
Table[{Cos[phi], Sin[phi]}, {phi
,0,2*Pi,Pi/2}],{d, d}]}],{d
,0,2,0.5}]
```



ArrowBox

Arrowheads

`Arrowheads[s]`
specifies that `Arrow[]` draws one arrow of size s (relative to width of image, defaults to 0.04).

`Arrowheads[{spec1, spec2, ..., specn}]`
specifies that `Arrow[]` draws n arrows as defined by $spec1, spec2, \dots, specn$.

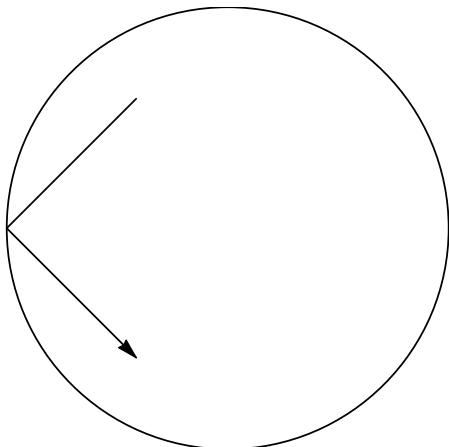
`Arrowheads[{s}]`
specifies that one arrow of size s should be drawn.

`Arrowheads[{s, pos}]`
specifies that one arrow of size s should be drawn at position pos (for the arrow to be on the line, pos has to be between 0, i.e. the start for the line, and 1, i.e. the end of the line).

`Arrowheads[{s, pos, g}]`
specifies that one arrow of size s should be drawn at position pos using Graphics g .

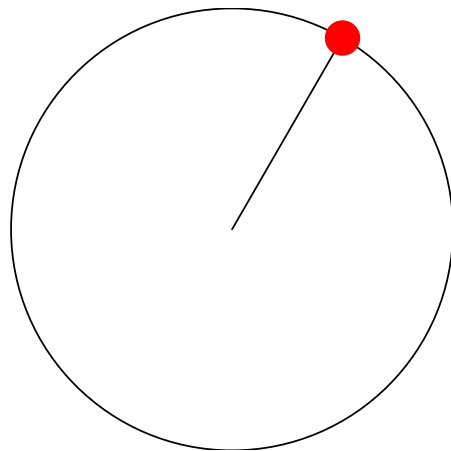
Arrows on both ends can be achieved using negative sizes:

```
>> Graphics[{Circle[], Arrowheads
[{-0.04, 0.04}], Arrow[{0, 0},
{2, 2}], {1, 1}}]
```

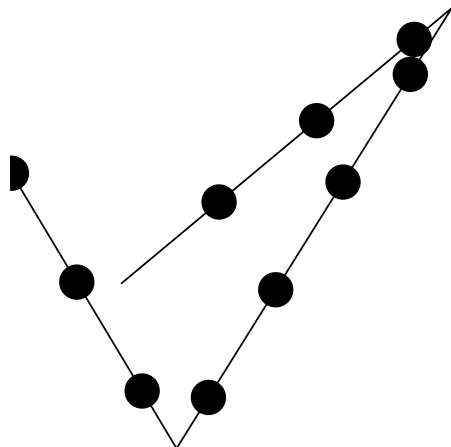


You may also specify our own arrow shapes:

```
>> Graphics[{Circle[], Arrowheads
[{{0.04, 1, Graphics[{Red, Disk
[]}]}}], Arrow[{0, 0}, {Cos[Pi
/3], Sin[Pi/3]}]}]
```



```
>> Graphics[{Arrowheads[Table
[{0.04, i/10, Graphics[Disk
[]]], {i, 1, 10}]], Arrow[{0, 0},
{6, 5}, {1, -3}, {-2, 2}]}]
```



Automatic

`Automatic`
is used to specify an automatically computed option value.

`Automatic` is the default for `PlotRange`, `ImageSize`, and other graphical options:

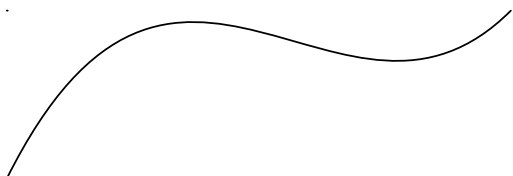
```
>> Cases[Options[Plot], HoldPattern
[_ :> Automatic]]
{Background:>Automatic,
 Exclusions:>Automatic,
 ImageSize:>Automatic,
 MaxRecursion:>Automatic,
 PlotRange:>Automatic,
 PlotRangePadding:>Automatic}
```

BernsteinBasis

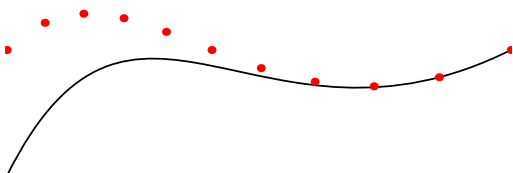
BezierCurve

`BezierCurve[{p1, p2 ...}]`
represents a bezier curve with $p1$, $p2$ as control points.

```
>> Graphics[BezierCurve[{{0, 0},{1,
1},{2, -1},{3, 0}}]]
```



```
>> Module[{p={{0, 0},{1, 1},{2,
-1},{4, 0}}}, Graphics[{
BezierCurve[p], Red, Point[Table
[BezierFunction[p][x], {x, 0, 1,
0.1}]]]}]]
```



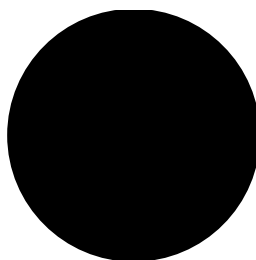
BezierCurveBox

BezierFunction

Black

`Black`
represents the color black in graphics.

```
>> Graphics[{EdgeForm[Black], Black
, Disk[]}, ImageSize->Small]
```



```
>> Black // ToBoxes
StyleBox [GraphicsBox [{EdgeForm [■], ■, RectangleBox [
-> Ignore, AspectRatio
-> Automatic, Axes
-> False, AxesStyle
-> {}, Background
-> Automatic, ImageSize
-> 16, LabelStyle-> {}, PlotRange
-> Automatic, PlotRangePadding
-> Automatic, TicksStyle
-> {}], ImageSizeMultipliers
-> {1, 1}]
```

```
>> Black
```



Blend

`Blend[{c1, c2}]`
represents the color between $c1$ and $c2$.
`Blend[{c1, c2}, x]`
represents the color formed by blending $c1$ and $c2$ with factors $1 - x$ and x respectively.
`Blend[{c1, c2, ..., cn}, x]`
blends between the colors $c1$ to cn according to the factor x .

```
>> Blend[{Red, Blue}]
```



```
>> Blend[{Red, Blue}, 0.3]
```



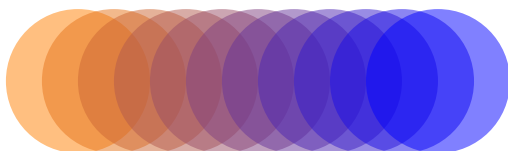
```
>> Blend[{Red, Blue, Green}, 0.75]
```



```
>> Graphics[Table[{Blend[{Red,
Green, Blue}, x], Rectangle[{10
x, 0}]], {x, 0, 1, 1/10}]]
```



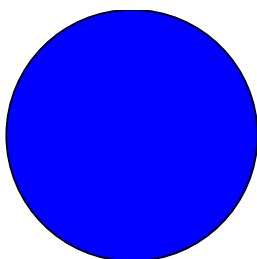
```
>> Graphics[Table[{Blend[{RGBColor
[1, 0.5, 0, 0.5], RGBColor[0, 0,
1, 0.5]}], x], Disk[{5x, 0}]], {
x, 0, 1, 1/10}]]
```



Blue

Blue
represents the color blue in graphics.

```
>> Graphics[{EdgeForm[Black], Blue,
Disk[]}, ImageSize->Small]
```



```
>> Blue // ToBoxes
StyleBox [GraphicsBox [{EdgeForm [■], ■, RectangleBox [{0,0}]}], $OptionSyntax
- > Ignore, AspectRatio
- > Automatic, Axes
- > False, AxesStyle
- > {}, Background
- > Automatic, ImageSize
- > 16, LabelStyle- > {}, PlotRange
- > Automatic, PlotRangePadding
- > Automatic, TicksStyle
- > {}, ImageSizeMultipliers
- > {1,1}]
```

```
>> Blue
■
```

CMYKColor

CMYKColor[*c*, *m*, *y*, *k*]
represents a color with the specified cyan, magenta, yellow and black components.

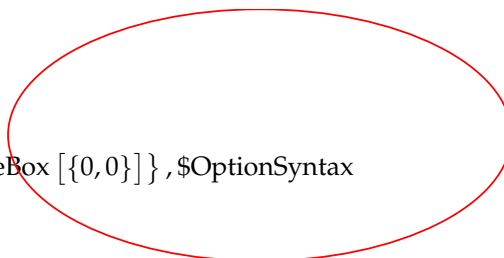
```
>> Graphics[MapIndexed[{CMYKColor
@@ #1, Disk[2*#2 ~Join~{0}]] &,
IdentityMatrix[4]], ImageSize->
Small]
```



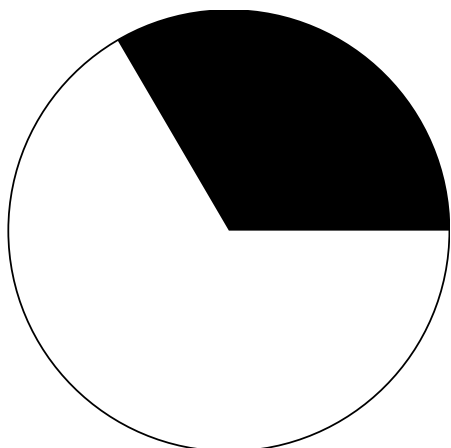
Circle

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}]]]
```



```
>> Graphics[{Circle[], Disk[{0, 0},
{1, 1}, {0, 2.1}]}]
```



CircleBox

ColorDistance

`ColorDistance[c1, c2]`
returns a measure of color distance between the colors *c1* and *c2*.

`ColorDistance[list, c2]`
returns a list of color distances between the colors in *list* and *c2*.

The option `DistanceFunction` specifies the method used to measure the color distance. Available options are:

CIE76: euclidean distance in the LABColor space
CIE94: euclidean distance in the LCH-Color space
CIE2000 or CIEDE2000: CIE94 distance with corrections
CMC: Colour Measurement Committee metric (1984)
DeltaL: difference in the L component of LCHColor
DeltaC: difference in the C component of LCHColor
DeltaH: difference in the H component of LCHColor

It is also possible to specify a custom distance

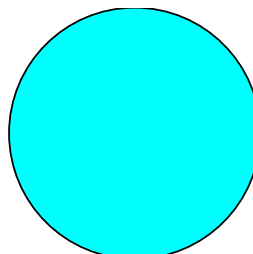
```
>> ColorDistance[Magenta, Green]
2.2507

>> ColorDistance[{Red, Blue}, {
Green, Yellow}, DistanceFunction
-> {"CMC", "Perceptibility"}]
{1.0495, 1.27455}
```

Cyan

Cyan
represents the color cyan in graphics.

```
>> Graphics[{EdgeForm[Black], Cyan,
Disk[]}, ImageSize->Small]
```



```
>> Cyan // ToBoxes
```

```
StyleBox[GraphicsBox[{EdgeForm[Black], Cyan, RectangleBox[
- > Ignore, AspectRatio
- > Automatic, Axes
- > False, AxesStyle
- > {}, Background
- > Automatic, ImageSize
- > 16, LabelStyle- > {}, PlotRange
- > Automatic, PlotRangePadding
- > Automatic, TicksStyle
- > {}], ImageSizeMultipliers
- > {1, 1}]]
```

```
>> Cyan
```



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}]]
```

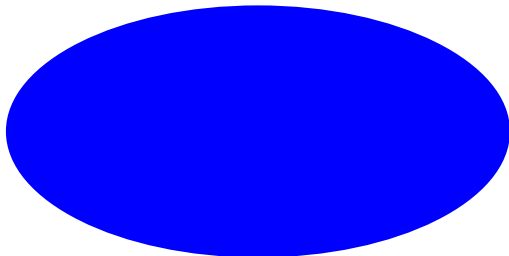


Directive

Disk

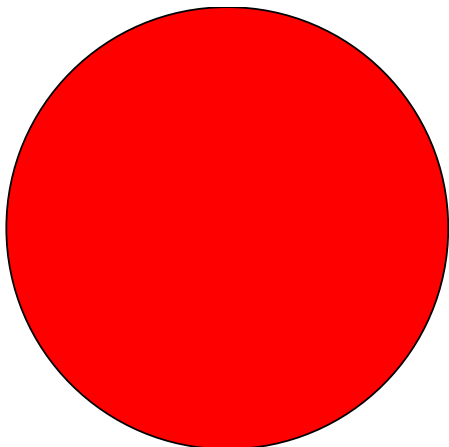
```
Disk[{cx, cy}, r]  
    fills a circle with center (cx, cy) and radius r.  
Disk[{cx, cy}, {rx, ry}]  
    fills an ellipse.  
Disk[{cx, cy}]  
    chooses radius 1.  
Disk[]  
    chooses center (0, 0) and radius 1.  
Disk[{x, y}, ..., {t1, t2}]  
    is a sector from angle t1 to t2.
```

```
>> Graphics[{Blue, Disk[{0, 0}, {2, 1}]}]
```



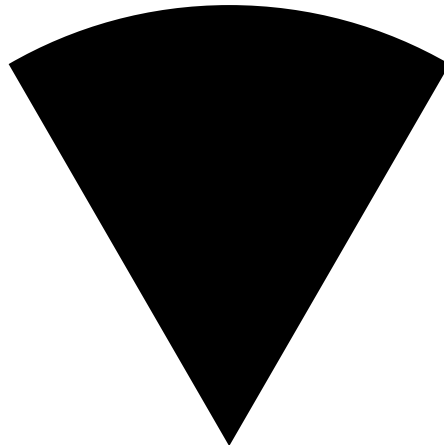
The outer border can be drawn using EdgeForm:

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

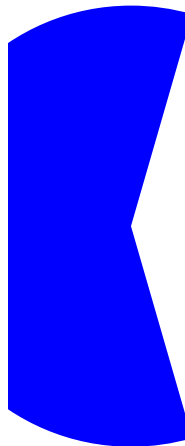


Disk can also draw sectors of circles and ellipses

```
>> Graphics[Disk[{0, 0}, 1, {Pi / 3, 2 Pi / 3}]]
```



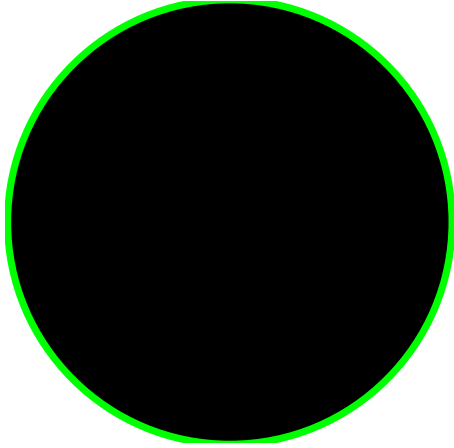
```
>> Graphics[{Blue, Disk[{0, 0}, {1, 2}, {Pi / 3, 5 Pi / 3}]]]
```



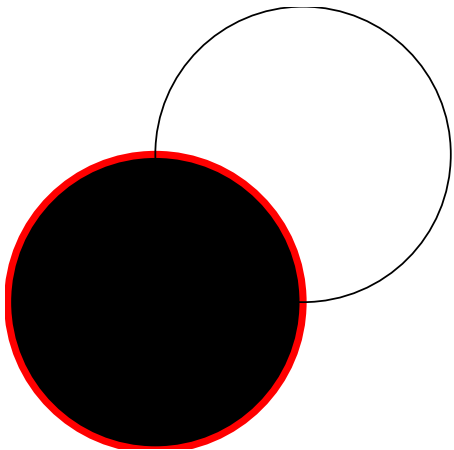
DiskBox

EdgeForm

```
>> Graphics[{EdgeForm[{Thick, Green}], Disk[]}]
```



```
>> Graphics[{Style[Disk[], EdgeForm[{Thick, Red}]], Circle[{1, 1}]}]
```



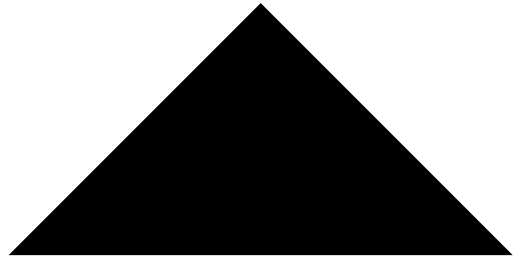
FaceForm

FilledCurve

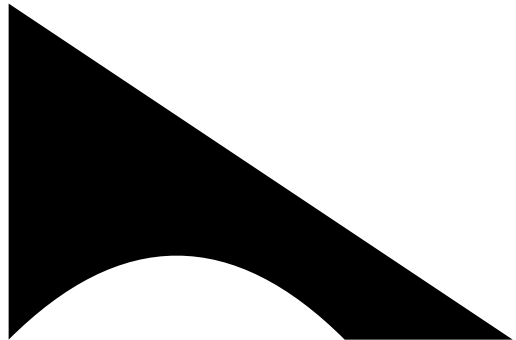
```
FilledCurve[{segment1, segment2 ...}]
```

represents a filled curve.

```
>> Graphics[FilledCurve[{Line[{0, 0}, {1, 1}, {2, 0}]}]]
```



```
>> Graphics[FilledCurve[{BezierCurve[{0, 0}, {1, 1}, {2, 0}], Line[{3, 0}, {0, 2}]}]]
```



FilledCurveBox

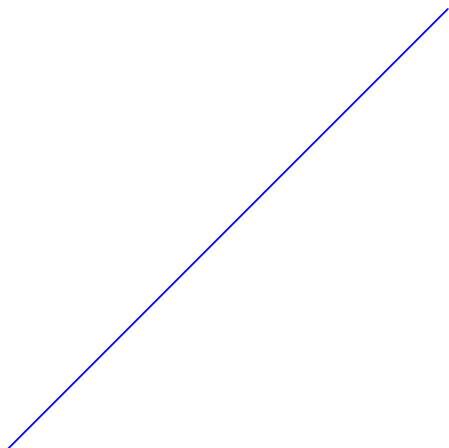
FontColor

FontColor
is an option for Style to set the font color.

Graphics

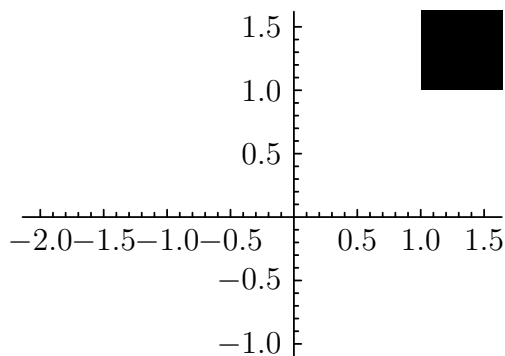
Graphics[primitives, options]
represents a graphic.

```
>> Graphics[{Blue, Line[{{0,0},
{1,1}}]}]
```



Graphics supports PlotRange:

```
>> Graphics[{Rectangle[{1, 1}]},
Axes -> True, PlotRange -> {{-2,
1.5}, {-1, 1.5}}]
```



```
>> Graphics[{Rectangle[], Red, Disk
[{1,0}], PlotRange
->{{0,1},{0,1}}]
```



Graphics produces GraphicsBox boxes:

```
>> Graphics[Rectangle[]] // ToBoxes
// Head
GraphicsBox
```

In TeXForm, Graphics produces Asymptote figures:

```
>> Graphics[Circle[]] // TeXForm

\begin{asy}
usepackage("amsmath");
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}
```

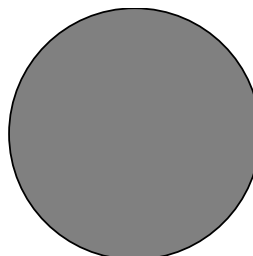
GraphicsBox

Gray

Gray

represents the color gray in graphics.

```
>> Graphics[{EdgeForm[Black], Gray,
Disk[]}, ImageSize->Small]
```



```
>> Gray // ToBoxes
```

```
StyleBox [GraphicsBox [{EdgeForm [■], ■, RectangleBox [
- > Ignore, AspectRatio
- > Automatic, Axes
- > False, AxesStyle
- > {}, Background
- > Automatic, ImageSize
- > 16, LabelStyle- > {}, PlotRange
- > Automatic, PlotRangePadding
- > Automatic, TicksStyle
- > {}], ImageSizeMultipliers
- > {1,1}]
```

```
>> Gray
```



GrayLevel

```
GrayLevel[g]
```

represents a shade of gray specified by g , ranging from 0 (black) to 1 (white).

```
GrayLevel[g, a]
```

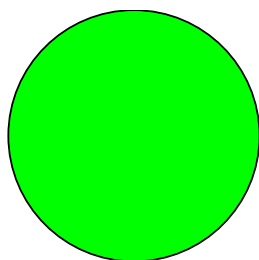
represents a shade of gray specified by g with opacity a .

Green

```
Green
```

represents the color green in graphics.

```
>> Graphics[{EdgeForm[Black], Green, Disk[]}, ImageSize->Small]
```



```
>> Green // ToBoxes
```

```
StyleBox[GraphicsBox[{EdgeForm[Black], Green, RectangleBox[{0,0}]}], $OptionSyntax]
```

```
- > Ignore, AspectRatio
- > Automatic, Axes
- > False, AxesStyle
- > {}, Background
- > Automatic, ImageSize
- > 16, LabelStyle- > {}, PlotRange
- > Automatic, PlotRangePadding
- > Automatic, TicksStyle
- > {}, ImageSizeMultipliers
- > {1,1}]
```

```
>> Green
```



Hue

```
Hue[h, s, l, a]
```

represents the color with hue h , saturation s , lightness l and opacity a .

```
Hue[h, s, l]
```

is equivalent to `Hue[h, s, l, 1]`.

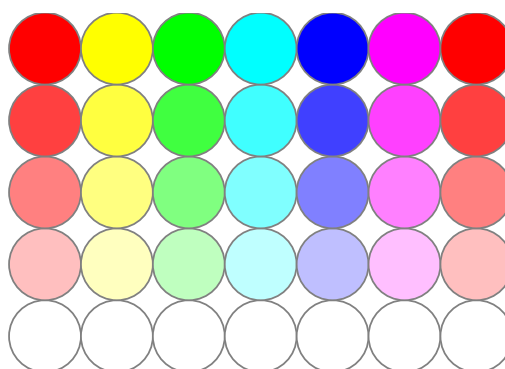
```
Hue[h, s]
```

is equivalent to `Hue[h, s, 1, 1]`.

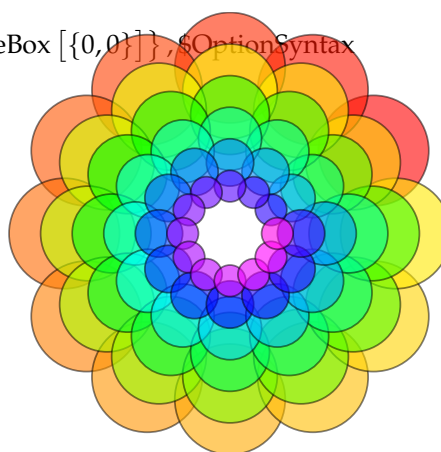
```
Hue[h]
```

is equivalent to `Hue[h, 1, 1, 1]`.

```
>> 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

LABColor

`LABColor[l, a, b]`

represents a color with the specified lightness, red/green and yellow/blue components in the CIE 1976 L*a*b* (CIELAB) color space.

LCHColor

`LCHColor[l, c, h]`

represents a color with the specified lightness, chroma and hue components in the CIE LCh CIE Lab cube color space.

LUVColor

`LCHColor[l, u, v]`

represents a color with the specified components in the CIE 1976 L*u*v* (CIELUV) color space.

Large

`ImageSize -> Large`

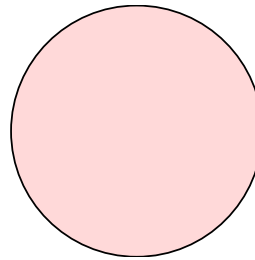
produces a large image.

LightRed

`LightRed`

represents the color light red in graphics.

```
>> Graphics[{EdgeForm[Black],  
LightRed, Disk[]}, ImageSize->  
Small]
```



```
>> LightRed // ToBoxes
```

```
StyleBox [GraphicsBox [{EdgeForm [Black], LightRed, RectangleBox [0, 0, 1, 1]},  
- > Ignore, AspectRatio  
- > Automatic, Axes  
- > False, AxesStyle  
- > {}, Background  
- > Automatic, ImageSize  
- > 16, LabelStyle - > {}, PlotRange  
- > Automatic, PlotRangePadding  
- > Automatic, TicksStyle  
- > {}], ImageSizeMultipliers  
- > {1, 1}]
```

Lighter

`Lighter[c, f]`

is equivalent to `Blend[{c, White}, f]`.

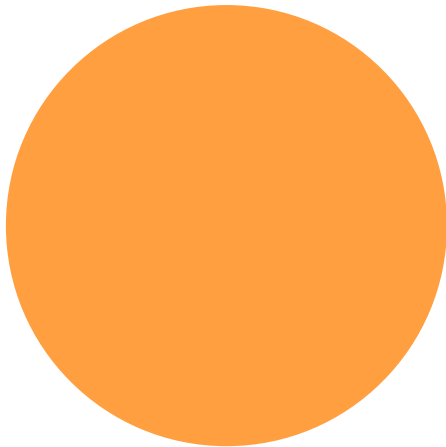
`Lighter[c]`

is equivalent to `Lighter[c, 1/3]`.

```
>> Lighter[Orange, 1/4]
```



```
>> Graphics[{Lighter[Orange, 1/4],
Disk[]}]
```



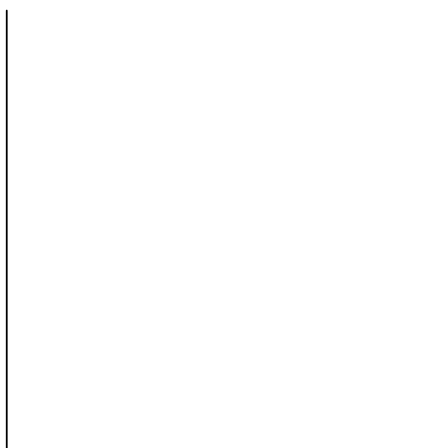
```
>> Graphics[Table[{Lighter[Orange,
x], Disk[{12x, 0}]}], {x, 0, 1,
1/6}]]
```



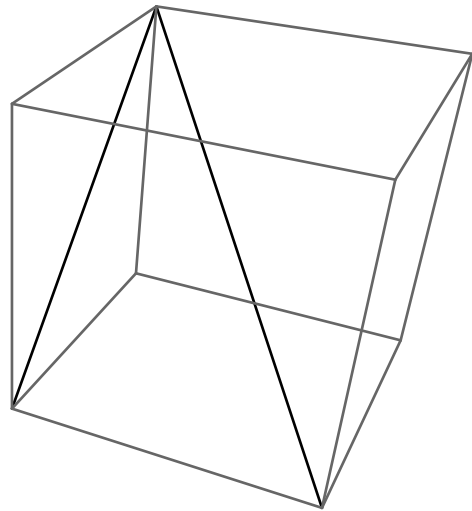
Line

`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}}]]
```

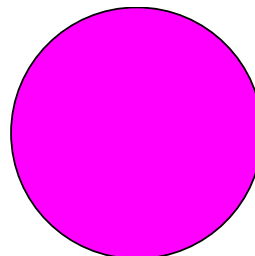


LineBox

Magenta

Magenta
represents the color magenta in graphics.

```
>> Graphics[{EdgeForm[Black],
Magenta, Disk[]}, ImageSize->
Small]
```



```
>> Magenta // ToBoxes
```

```
StyleBox [GraphicsBox [{EdgeForm [Black], RectangleBox[80,0], GraphicsBox[{EdgeForm [Magenta], RectangleBox[80,0]}], AspectRatio->Ignore, Axes->Automatic, AxesStyle->False, Background->{ }, ImageSize->Automatic, LabelStyle->16, PlotRange->{ }, PlotRangePadding->Automatic, TicksStyle->Automatic, ImageSizeMultipliers->{1,1}]
```

```
>> Magenta
```



Medium

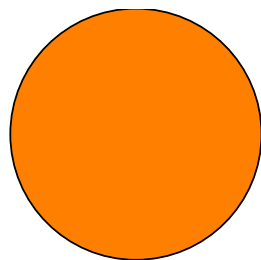
ImageSize -> Medium
produces a medium-sized image.

Offset

Orange

Orange
represents the color orange in graphics.

```
>> Graphics[{EdgeForm[Black],  
Orange, Disk[]}, ImageSize->  
Small]
```



```
>> Orange // ToBoxes
```

```
StyleBox [GraphicsBox [{EdgeForm [Black], RectangleBox[80,0], GraphicsBox[{EdgeForm [Orange], RectangleBox[80,0]}], AspectRatio->Ignore, Axes->Automatic, AxesStyle->False, Background->{ }, ImageSize->Automatic, LabelStyle->16, PlotRange->{ }, PlotRangePadding->Automatic, TicksStyle->Automatic, ImageSizeMultipliers->{1,1}]
```

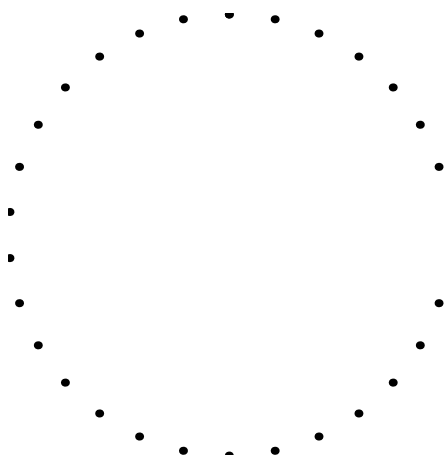
Point

Point[{*point_1*, *point_2* ...}]
represents the point primitive.
Point[{{*p_11*, *p_12*, ...}, {*p_21*, *p_22*, ...}, ...}]
represents a number of point primitives.

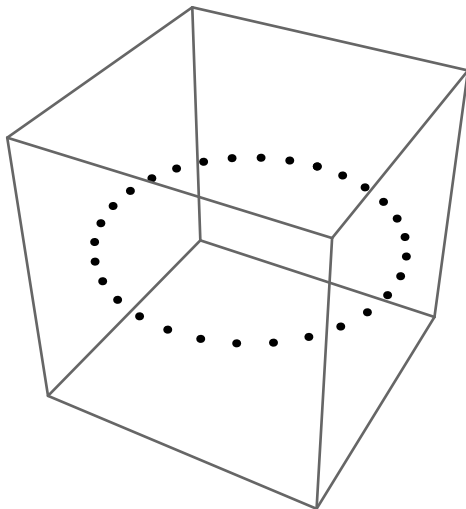
```
>> Graphics[Point[{0,0}]]
```

```
.
```

```
>> 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.}]]]
```



PointSize

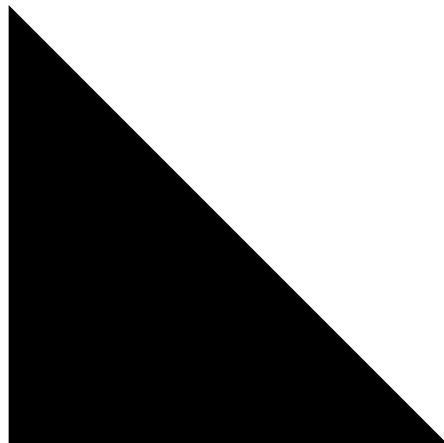
PointSize

`PointSize[t]`
sets the diameter of points to t , which is relative to the overall width.

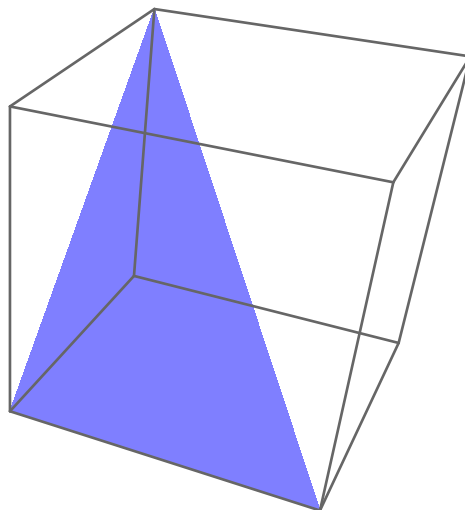
Polygon

`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}}]]
```

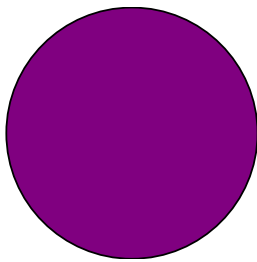


PolygonBox

Purple

`Purple`
represents the color purple in graphics.


```
>> Graphics[{EdgeForm[Black],
Purple, Disk[]}, ImageSize->
Small]
```



```
>> Purple // ToBoxes
StyleBox [GraphicsBox [{EdgeForm [■], ■, RectangleBox [
- > Ignore, AspectRatio
- > Automatic, Axes
- > False, AxesStyle
- > {}, Background
- > Automatic, ImageSize
- > 16, LabelStyle- > {}, PlotRange
- > Automatic, PlotRangePadding
- > Automatic, TicksStyle
- > {}], ImageSizeMultipliers
- > {1, 1}]]
```

RGBColor

`RGBColor[r, g, b]`
represents a color with the specified red, green and blue components.

```
>> Graphics[MapIndexed[{RGBColor @@
#1, Disk[2*#2 ~Join~{0}]} &,
IdentityMatrix[3]], ImageSize->
Small]
```



```
>> RGBColor[0, 1, 0]
■
```

```
>> RGBColor[0, 1, 0] // ToBoxes
```

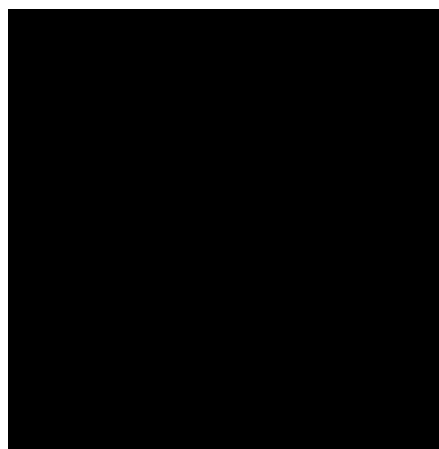
```
StyleBox [GraphicsBox [{EdgeForm [■], ■, RectangleBox [
- > Ignore, AspectRatio
- > Automatic, Axes
- > False, AxesStyle
- > {}, Background
- > Automatic, ImageSize
- > 16, LabelStyle- > {}, PlotRange
- > Automatic, PlotRangePadding
- > Automatic, TicksStyle
- > {}], ImageSizeMultipliers
- > {1, 1}]]
```

Rectangle

`Rectangle[{xmin, ymin}]`
represents a unit square with bottom-left corner at {xmin, ymin}.

`Rectangle[{xmin, ymin}, {xmax, ymax}]`
is a rectangle extending from {xmin, ymin} to {xmax, ymax}.

```
>> Graphics[Rectangle[]]
```



```
>> Graphics[{Blue, Rectangle[{0.5, 0}], Orange, Rectangle[{0, 0.5}]}]
```



```
>> Red
```



RegularPolygon

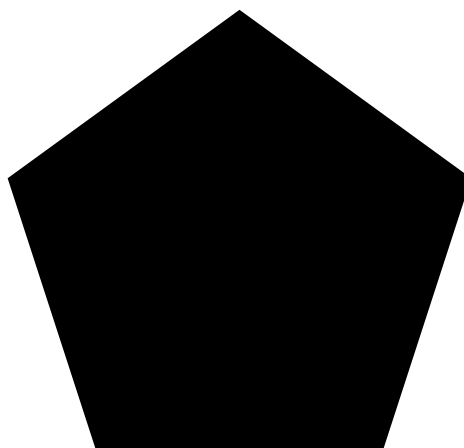
`RegularPolygon[n]`
gives the regular polygon with n edges.

`RegularPolygon[r, n]`
gives the regular polygon with n edges and radius r .

`RegularPolygon[{r, phi}, n]`
gives the regular polygon with radius r with one vertex drawn at angle phi .

`RegularPolygon[{x, y}, r, n]`
gives the regular polygon centered at the position $\{x, y\}$.

```
>> Graphics[RegularPolygon[5]]
```

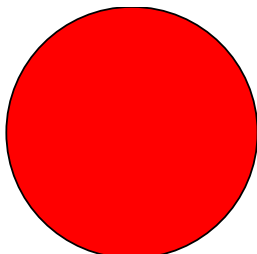


RectangleBox

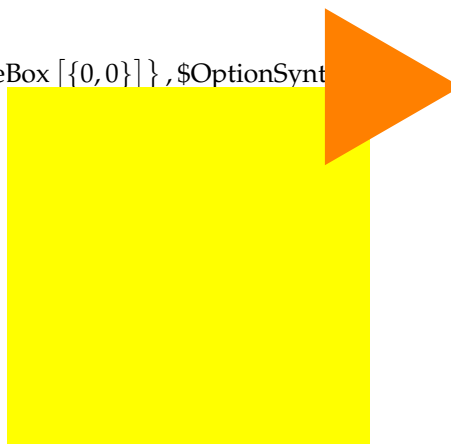
Red

`Red`
represents the color red in graphics.

```
>> Graphics[{EdgeForm[Black], Red, Disk[]}, ImageSize->Small]
```



```
>> Graphics[{Yellow, Rectangle[], Orange, RegularPolygon[{1, 1}, {0.25, 0}, 3]}]
```



```
>> Red // ToBoxes
```

```
StyleBox[GraphicsBox[{EdgeForm[Black], Red, RectangleBox[{0, 0}]}], $OptionSyntax]
- > Ignore, AspectRatio
- > Automatic, Axes
- > False, AxesStyle
- > {}, Background
- > Automatic, ImageSize
- > 16, LabelStyle- > {}, PlotRange
- > Automatic, PlotRangePadding
- > Automatic, TicksStyle
- > {}, ImageSizeMultipliers
- > {1, 1}]
```

RegularPolygonBox

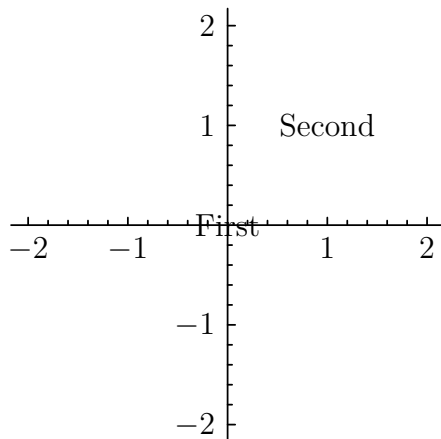
Small

`ImageSize -> Small`
produces a small image.

Text

`Text["text", {x, y}]`
draws *text* centered on position {x, y}.

```
>> Graphics[{Text["First", {0, 0}],
  Text["Second", {1, 1}], Axes->
  True, PlotRange->{{-2, 2}, {-2,
  2}}]
```



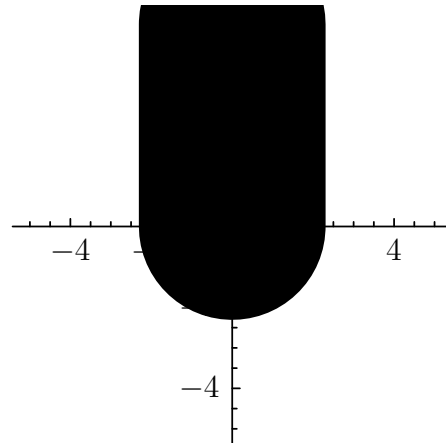
Thick

`Thick`
sets the line width for subsequent graphics primitives to 2pt.

Thickness

`Thickness[t]`
sets the line thickness for subsequent graphics primitives to *t* times the size of the plot area.

```
>> Graphics[{Thickness[0.2], Line
  [{0, 0}, {0, 5}], Axes->True,
  PlotRange->{{-5, 5}, {-5, 5}}]
```



Thin

`Thin`
sets the line width for subsequent graphics primitives to 0.5pt.

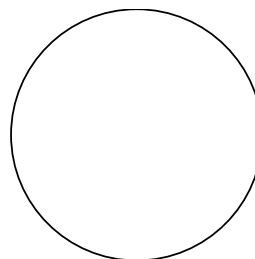
Tiny

`ImageSize -> Tiny`
produces a tiny image.

White

`White`
represents the color white in graphics.

```
>> Graphics[{EdgeForm[Black], White
  , Disk[]}, ImageSize->Small]
```



```

>> White // ToBoxes
StyleBox [GraphicsBox [{EdgeForm [Black], RectangleBox[{0,0},{1,1}], PlotRangeStyle->Ignore, AspectRatio->Automatic, Axes->False, AxesStyle->{Automatic, Automatic}, Background->{Automatic, Automatic}, ImageSize->16, LabelStyle->{Automatic, Automatic}, PlotRangePadding->Automatic, TicksStyle->{Automatic, Automatic}, ImageSizeMultipliers->{1,1}], ImageSizeMultipliers->{1,1}]

>> Yellow // ToBoxes
StyleBox [GraphicsBox [{EdgeForm [Black], RectangleBox[{0,0},{1,1}], PlotRangeStyle->Ignore, AspectRatio->Automatic, Axes->False, AxesStyle->{Automatic, Automatic}, Background->{Automatic, Automatic}, ImageSize->16, LabelStyle->{Automatic, Automatic}, PlotRangePadding->Automatic, TicksStyle->{Automatic, Automatic}, ImageSizeMultipliers->{1,1}], ImageSizeMultipliers->{1,1}]

>> White
□

>> Yellow
■

```

XYZColor

`XYZColor[x, y, z]`
 represents a color with the specified components in the CIE 1931 XYZ color space.

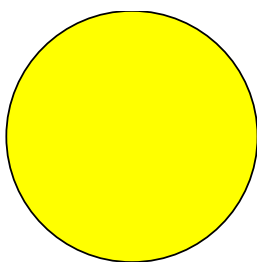
Yellow

`Yellow`
 represents the color yellow in graphics.

```

>> Graphics[{EdgeForm[Black],
Yellow, Disk[]}, ImageSize->Small]

```



XVI. Graphics (3D)

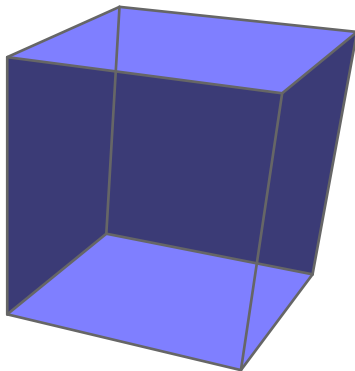
Contents

Cuboid	118	Line3DBox	119	Sphere	120
Graphics3D	119	Point3DBox	119	Sphere3DBox	120
Graphics3DBox	119	Polygon3DBox	119		

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}]]
```



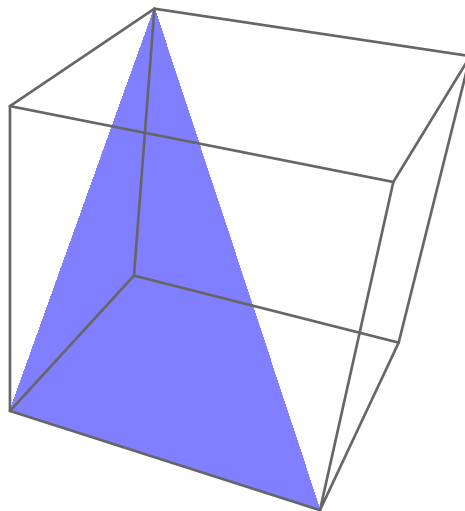
```
>> Graphics3D[{Red, Cuboid[{0, 0,
0}, {1, 1, 0.5}], Blue, Cuboid
[{0.25, 0.25, 0.5}, {0.75, 0.75,
1}]]}
```

```
Graphics3DBox[List[StyleBox[Graphics[List[Edge
RGBColor[1, 0, 0], Rectangle[List[0,
0]]], Rule[ImageSize, 16]],
Rule[ImageSizeMultipliers, List[1,
1]]], Polygon3DBox[List[List[List[0.,
0., 0.], List[0., 1., 0.], List[0., 1., 0.5]],
List[List[0., 0., 0.], List[0., 0., 0.5],
List[0., 1., 0.5]], List[List[1., 0.,
0.], List[1., 1., 0.], List[1., 1., 0.5]],
List[List[1., 0., 0.], List[1., 0., 0.5],
List[1., 1., 0.5]], List[List[0., 0.,
0.], List[1., 0., 0.], List[1., 0., 0.5]],
List[List[0., 0., 0.], List[0., 0., 0.5],
List[1., 0., 0.5]], List[List[0., 1.,
0.], List[1., 1., 0.], List[1., 1., 0.5]],
List[List[0., 1., 0.], List[0., 1., 0.5],
List[1., 1., 0.5]], List[List[0., 0.,
0.], List[0., 1., 0.], List[1., 1., 0.]],
List[List[0., 0., 0.], List[1., 0., 0.], List[1.,
1., 0.]], List[List[0., 0., 0.5], List[0., 1.,
0.5], List[1., 1., 0.5]], List[List[0., 0.,
0.5], List[1., 0., 0.5], List[1., 1., 0.5]]],
StyleBox[Graphics[List[EdgeForm[GrayLevel[0]],
RGBColor[0, 0, 1], Rectangle[List[0,
0]]], Rule[ImageSize, 16]],
Rule[ImageSizeMultipliers, List[1, 1]]],
Polygon3DBox[List[List[List[0.25, 0.25
, 0.5], List[0.25, 0.75, 0.5], List[0.25, 0.75
, 1.]], List[List[0.25, 0.25, 0.5], List[0.25,
0.25, 1.], List[0.25, 0.75, 1.]], List[List[
0.75, 0.25, 0.5], List[0.75, 0.75, 0.5], List[
0.75, 0.75, 1.]], List[List[0.75, 0.25, 0.5],
List[0.75, 0.25, 1.], List[0.75, 0.75, 1.]],
List[List[0.25, 0.25, 0.5], List[0.75, 0.25,
0.5], List[0.75, 0.25, 1.]], List[List[0.25,
0.25, 0.5], List[0.25, 0.25, 1.], List[0.75,
0.25, 1.]], List[List[0.25, 0.75, 0.5], List[
0.75, 0.75, 0.5], List[0.75, 0.75, 1.]],
List[List[0.25, 0.75, 0.5], List[0.25, 0.75,
1.], List[0.75, 0.75, 1.]], List[List[0.25
, 0.25, 0.5], List[0.25, 0.75, 0.5], List[
0.75, 0.75, 0.5]], List[List[0.25, 0.25,
0.5], List[0.75, 0.25, 0.5], List[0.75, 0.75,
0.5]], List[List[0.25, 0.25, 1.], List[0.25,
0.75, 1.], List[0.75, 0.75, 1.]], List[List[
0.25, 0.25, 1.], List[0.75, 0.25, 1.], List[
0.75, 0.75, 1.]]], Rule[$OptionSyntax,
Ignore], Rule[AspectRatio, Automatic],
Rule[Axes, False], Rule[AxesStyle,
List[]], Rule[Background, Automatic],
Rule[BoxRatios, Automatic],
Rule[ImageSize, Automatic]
```

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));
\end{asy}
```

Graphics3DBox

Line3DBox

Point3DBox

Polygon3DBox

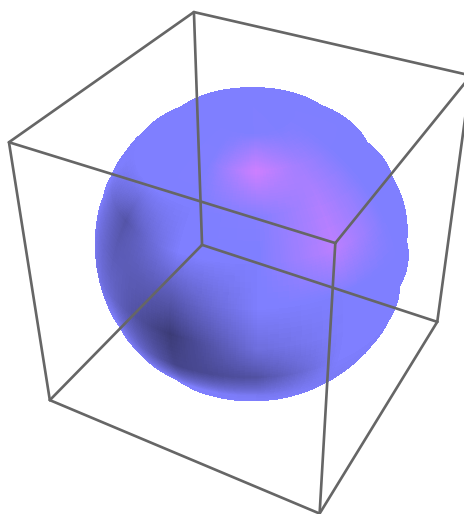
Sphere

`Sphere[{x, y, z}]`
is a sphere of radius 1 centered 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]]}

Graphics3DBox[List[StyleBox[Graphics[List[EdgeForm[GrayLevel[0]],
RGBColor[1, 1, 0], Rectangle[List[0,
0]]], Rule[ImageSize, 16]],
Rule[ImageSizeMultipliers, List[1,
1]]], Sphere3DBox[List[List[-1, 0, 0],
List[1, 0, 0], List[0, 0, 1.732050807~
~5688772]], 1]], Rule[$OptionSyntax,
Ignore], Rule[AspectRatio, Automatic],
Rule[Axes, False], Rule[AxesStyle,
List[]], Rule[Background, Automatic],
Rule[BoxRatios, Automatic],
Rule[ImageSize, Automatic],
Rule[LabelStyle, List[]], Rule[Lighting,
Automatic], Rule[PlotRange,
Automatic], Rule[PlotRangePadding,
Automatic], Rule[TicksStyle,
List[]], Rule[ViewPoint, List[1.3,
- 2.4, 2.]]]isnotavalidboxstructure.
```

Sphere3DBox

XVII. Image[] and image related functions.

Note that you (currently) need scikit-image installed in order for this module to work.

Contents

Binarize	121	ImageAdd	123	ImageSubtract	124
BinaryImageQ	121	ImageAdjust	123	ImageTake	125
Blur	121	ImageAspectRatio	123	ImageType	125
BoxMatrix	121	Image	123	MaxFilter	125
Closing	122	ImageBox	123	MedianFilter	125
ColorCombine	122	ImageChannels	123	MinFilter	125
ColorConvert	122	ImageColorSpace	123	MorphologicalCompo- nents	125
ColorNegate	122	ImageConvolve	124	Opening	125
ColorQuantize	122	ImageData	124	PillowImageFilter	125
ColorSeparate	122	ImageDimensions	124	PixelValue	125
Colorize	122	ImageExport	124	PixelValuePositions	125
DiamondMatrix	122	ImageImport	124	RandomImage	125
Dilation	122	ImageMultiply	124	Sharpen	125
DiskMatrix	122	ImagePartition	124	TextRecognize	126
DominantColors	123	ImageQ	124	Threshold	126
EdgeDetect	123	ImageReflect	124	WordCloud	126
Erosion	123	ImageResize	124		
GaussianFilter	123	ImageRotate	124		

Binarize

`Binarize[image]`
gives a binarized version of *image*, in which each pixel is either 0 or 1.

`Binarize[image, t]`
map values $x > t$ to 1, and values $x \leq t$ to 0.

`Binarize[image, {t1, t2}]`
map $t1 < x < t2$ to 1, and all other values to 0.

BinaryImageQ

`BinaryImageQ[$image]`
returns True if the pixels of *\$image* are binary bit values, and False otherwise.

Blur

`Blur[image]`
gives a blurred version of *image*.

`Blur[image, r]`
blurs *image* with a kernel of size *r*.

BoxMatrix

`BoxMatrix[$s]`
Gives a box shaped kernel of size $2s + 1$.

Closing

`Closing[image, ker]`
Gives the morphological closing of *image* with respect to structuring element *ker*.

ColorCombine

`ColorCombine[channels, colspace]`
Gives an image with *colspace* and the respective components described by the given channels.

ColorConvert

`ColorConvert[c, colspace]`
returns the representation of *c* in the color space *colspace*. *c* may be a color or an image.

Valid values for *colspace* are:

CMYK: convert to CMYKColor
Grayscale: convert to GrayLevel
HSB: convert to Hue
LAB: convert to LabColor
LCH: convert to LCHColor
LUV: convert to LUVColor
RGB: convert to RGBColor
XYZ: convert to XYZColor

ColorNegate

`ColorNegate[image]`
Gives a version of *image* with all colors negated.

ColorQuantize

`ColorQuantize[image, n]`
gives a version of *image* using only *n* colors.

ColorSeparate

`ColorSeparate[image]`
Gives each channel of *image* as a separate grayscale image.

Colorize

`Colorize[values]`
returns an image where each number in the rectangular matrix *values* is a pixel and each occurrence of the same number is displayed in the same unique color, which is different from the colors of all non-identical numbers.
`Colorize[image]`
gives a colorized version of *image*.

DiamondMatrix

`DiamondMatrix[s]`
Gives a diamond shaped kernel of size $2s + 1$.

Dilation

`Dilation[image, ker]`
Gives the morphological dilation of *image* with respect to structuring element *ker*.

DiskMatrix

`DiskMatrix[s]`
Gives a disk shaped kernel of size $2s + 1$.

DominantColors

`DominantColors[image]`
gives a list of colors which are dominant in the given image.

`DominantColors[image, n]`
returns at most n colors.

`DominantColors[image, n, prop]`
returns the given property *prop*, which may be “Color” (return RGB colors), “LABColor” (return LAB colors), “Count” (return the number of pixels a dominant color covers), “Coverage” (return the fraction of the image a dominant color covers), or “CoverageImage” (return a black and white image indicating with white the parts that are covered by a dominant color).

The option “ColorCoverage” specifies the minimum amount of coverage needed to include a dominant color in the result.

The option “MinColorDistance” specifies the distance (in LAB color space) up to which colors are merged and thus regarded as belonging to the same dominant color.

EdgeDetect

`EdgeDetect[image]`
returns an image showing the edges in *image*.

Erosion

`Erosion[image, ker]`
Gives the morphological erosion of *image* with respect to structuring element *ker*.

GaussianFilter

`GaussianFilter[image, r]`
blurs *image* using a Gaussian blur filter of radius r .

ImageAdd

`ImageAdd[image, expr_1, expr_2, ...]`
adds all *expr_i* to *image* where each *expr_i* must be an image or a real number.

ImageAdjust

`ImageAdjust[image]`
adjusts the levels in *image*.

`ImageAdjust[image, c]`
adjusts the contrast in *image* by c .

`ImageAdjust[image, {c, b}]`
adjusts the contrast c , and brightness b in *image*.

`ImageAdjust[image, {c, b, g}]`
adjusts the contrast c , brightness b , and gamma g in *image*.

ImageAspectRatio

`ImageAspectRatio[image]`
gives the aspect ratio of *image*.

Image

ImageBox

ImageChannels

`ImageChannels[image]`
gives the number of channels in *image*.

ImageColorSpace

`ImageColorSpace[image]`
gives *image*’s color space, e.g. “RGB” or “CMYK”.

ImageConvolve

```
ImageConvolve[image, kernel]
```

Computes the convolution of *image* using *kernel*.

ImageData

```
ImageData[image]
```

gives a list of all color values of *image* as a matrix.

```
ImageData[image, stype]
```

gives a list of color values in type *stype*.

ImageDimensions

```
ImageDimensions[image]
```

Returns the dimensions of *image* in pixels.

ImageExport

ImageImport

ImageMultiply

```
ImageMultiply[image, expr_1, expr_2,  
...]
```

multiplies all *expr_i* with *image* where each *expr_i* must be an image or a real number.

ImagePartition

```
ImagePartition[image, s]
```

Partitions an image into an array of *s* x *s* pixel subimages.

```
ImagePartition[image, {w, h}]
```

Partitions an image into an array of *w* x *h* pixel subimages.

ImageQ

```
ImageQ[Image[$pixels]]
```

returns True if \$pixels has dimensions from which an Image can be constructed, and False otherwise.

ImageReflect

```
ImageReflect[image]
```

Flips *image* top to bottom.

```
ImageReflect[image, side]
```

Flips *image* so that *side* is interchanged with its opposite.

```
ImageReflect[image, side_1 -> side_2]
```

Flips *image* so that *side_1* is interchanged with *side_2*.

ImageResize

```
ImageResize[image, width]
```

```
ImageResize[image, {width, height}]
```

The default sampling method is Bicubic

ImageRotate

```
ImageRotate[image]
```

Rotates *image* 90 degrees counterclockwise.

```
ImageRotate[image, theta]
```

Rotates *image* by a given angle *theta*

ImageSubtract

```
ImageSubtract[image, expr_1, expr_2,  
...]
```

subtracts all *expr_i* from *image* where each *expr_i* must be an image or a real number.

ImageTake

`ImageTake[image, n]`
gives the first n rows of *image*.
`ImageTake[image, -n]`
gives the last n rows of *image*.
`ImageTake[image, {r1, r2}]`
gives rows $r1, \dots, r2$ of *image*.
`ImageTake[image, {r1, r2}, {c1, c2}]`
gives a cropped version of *image*.

ImageType

`ImageType[image]`
gives the interval storage type of *image*,
e.g. "Real", "Bit32", or "Bit".

MaxFilter

`MaxFilter[image, r]`
gives *image* with a maximum filter of radius r applied on it. This always picks the largest value in the filter's area.

MedianFilter

`MedianFilter[image, r]`
gives *image* with a median filter of radius r applied on it. This always picks the median value in the filter's area.

MinFilter

`MinFilter[image, r]`
gives *image* with a minimum filter of radius r applied on it. This always picks the smallest value in the filter's area.

MorphologicalComponents

Opening

`Opening[image, ker]`
Gives the morphological opening of *image* with respect to structuring element *ker*.

PillowImageFilter

PixelValue

`PixelValue[image, {x, y}]`
gives the value of the pixel at position $\{x, y\}$ in *image*.

PixelValuePositions

`PixelValuePositions[image, val]`
gives the positions of all pixels in *image* that have value *val*.

RandomImage

`RandomImage[max]`
creates an image of random pixels with values 0 to *max*.
`RandomImage[{min, max}]`
creates an image of random pixels with values *min* to *max*.
`RandomImage[... , size]`
creates an image of the given *size*.

Sharpen

`Sharpen[image]`
gives a sharpened version of *image*.
`Sharpen[image, r]`
sharpens *image* with a kernel of size r .

TextRecognize

```
TextRecognize[{image}]
```

Recognizes text in *image* and returns it as string.

Threshold

```
Threshold[image]
```

gives a value suitable for binarizing *image*.

The option “Method” may be “Cluster” (use Otsu’s threshold), “Median”, or “Mean”.

WordCloud

```
WordCloud[{word1, word2, ...}]
```

Gives a word cloud with the given list of words.

```
WordCloud[{weight1 -> word1, weight2 -> word2, ...}]
```

Gives a word cloud with the words weighted using the given weights.

```
WordCloud[{weight1, weight2, ...} -> {word1, word2, ...}]
```

Also gives a word cloud with the words weighted using the given weights.

```
WordCloud[{word1, weight1}, {word2, weight2}, ...}]
```

Gives a word cloud with the words weighted using the given weights.

XVIII. Input and Output

Contents

BaseForm	127	Message	130	StandardForm	133
Center	127	MessageName (: :) . .	130	StringForm	133
Check	128	NonAssociative	130	Style	133
Format	128	NumberForm	130	Subscript	133
FullForm	128	Off	131	SubscriptBox	133
General	128	On	131	Subsuperscript	133
Grid	128	OutputForm	131	SubsuperscriptBox . .	133
GridBox	129	Postfix (/)	131	Superscript	133
Infix	129	Precedence	132	SuperscriptBox	133
InputForm	129	Prefix (@)	132	Syntax	134
Left	129	Print	132	TableForm	134
MakeBoxes	129	Quiet	132	TeXForm	134
MathMLForm	130	Right	133	ToBoxes	135
MatrixForm	130	Row	133	\$UseSansSerif	135
		RowBox	133		

BaseForm

`BaseForm[expr, n]`
prints numbers in *expr* in base *n*.

```
>> BaseForm[33, 2]
100 0012

>> BaseForm[234, 16]
ea16

>> BaseForm[12.3, 2]
1 100.010011001100110012

>> BaseForm[-42, 16]
-2a16

>> BaseForm[x, 2]
x

>> BaseForm[12, 3] // FullForm
BaseForm[12, 3]
```

Bases must be between 2 and 36:

```
>> BaseForm[12, -3]
Positivemachine
— 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.
```

Center

`Center`
is used with the `ColumnAlignments` option to `Grid` or `TableForm` to specify a centered column.

Check

```
Check[expr, failexpr]
  evaluates expr, and returns the result, unless
  messages were generated, in which case it
  evaluates and failexpr will be returned.
Check[expr, failexpr, {s1::t1,s2::t2,...}]
  checks only for the specified messages.
```

Return err when a message is generated:

```
>> Check[1/0, err]
      Infiniteexpression1/0encountered.
      err
```

Check only for specific messages:

```
>> Check[Sin[0^0], err, Sin::argx]
      Indeterminateexpression0^0encountered.
      Indeterminate
>> Check[1/0, err, Power::infy]
      Infiniteexpression1/0encountered.
      err
```

Format

```
Format[expr]
  holds values specifying how expr should be
  printed.
```

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 ~ 2 ~ 3
>> f[1]
      1
```

Raw objects cannot be formatted:

```
>> Format[3] = "three";
      Cannotassigntorawobject3.
```

Format types must be symbols:

```
>> Format[r, a + b] = "r";
      Formattypea + bisnotasymbol.
```

Formats must be attached to the head of an expression:

```
>> f /: Format[g[f]] = "my f";
      Tagfnotfoundortoodeepforanassignedrule.
```

FullForm

```
FullForm[expr]
  displays the underlying form of expr.
```

```
>> FullForm[a + b * c]
      Plus[a, 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]
      Rulecalledwith1argument;2argumentsareexpected.
```

Grid

```
Grid[{{a1, a2, ...}, {b1, b2, ...}, ...}]
  formats several expressions inside a
  GridBox.
```

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


GridBox

`GridBox[{{...}, {...}}]`
is a box construct that represents a sequence of boxes arranged in a grid.

Infix

`Infix[expr, oper, prec, assoc]`
displays *expr* with the infix operator *oper*, with precedence *prec* and associativity *assoc*.

`Infix` can be used with `Format` to display certain forms with user-defined infix notation:

```
>> 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]
a#b#c

>> g[a, b] + c
c + a#b

>> g[a, b] * c
c(a#b)

>> Infix[{a, b, c}, {"+", "-"}]
a + b - c
```

InputForm

`InputForm[expr]`
displays *expr* in an unambiguous form suitable for input.

```
>> InputForm[a + b * c]
a + b * c
```

```
>> InputForm["A string"]
"A string"

>> InputForm[f' [x]]
Derivative[1] [f] [x]

>> InputForm[Derivative[1, 0] [f] [x]]
Derivative[1, 0] [f] [x]
```

Left

`Left`
is used with operator formatting constructs to specify a left-associative operator.

MakeBoxes

`MakeBoxes[expr]`
is a low-level formatting primitive that converts *expr* to box form, without evaluating it.

```
\( ... \)
directly inputs box objects.
```

String representation of boxes

```
>> \(\mathbf{x}^2\)
SuperscriptBox[x, 2]

>> \(\mathbf{x}_2\)
SubscriptBox[x, 2]

>> \(\mathbf{a}^{\mathbf{b}}^{\mathbf{c}}\)
UnderoverscriptBox[a, b, c]

>> \(\mathbf{a}^{\mathbf{b}}^{\mathbf{c}}\)
UnderoverscriptBox[a, c, b]

>> \(\mathbf{x}^{\mathbf{y}}\)
OverscriptBox[x, y]

>> \(\mathbf{x}_y\)
UnderscriptBox[x, y]
```

MathMLForm

`MathMLForm[expr]`
displays *expr* as a MathML expression.

```
>> MathMLForm[HoldForm[Sqrt[a^3]]]
<math display="block">
  <msqrt><msup><mi>a</mi>
  <mn>3</mn></msup>
</msqrt></math>
```

MatrixForm

`MatrixForm[m]`
displays a matrix *m*, hiding the underlying list structure.

```
>> Array[a,{4,3}]/MatrixForm
( 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] )
```

Message

`Message[symbol::msg, expr1, expr2, ...]`
displays the specified message, replacing placeholders in the message text with the corresponding expressions.

```
>> 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, Mr007!
```

MessageName (::)

`MessageName[symbol, tag]`
symbol::tag
identifies a message.

`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

`NonAssociative`
is used with operator formatting constructs to specify a non-associative operator.

NumberForm

`NumberForm[expr, n]`
prints a real number *expr* with *n*-digits of precision.
`NumberForm[expr, {n, f}]`
prints with *n*-digits and *f* digits to the right of the decimal point.

```
>> NumberForm[N[Pi], 10]
3.141592654

>> NumberForm[N[Pi], {10, 5}]
3.14159
```

Off

`Off[symbol::tag]`
turns a message off so it is no longer printed.

```
>> Off[Power::infy]
```

```
>> 1 / 0
ComplexInfinity

>> Off[Power::indet, Syntax::com]

>> {0 ^ 0,}
{Indeterminate, Null}
```

On

`On[symbol::tag]`
turns a message on for printing.

```
>> Off[Power::infy]

>> 1 / 0
ComplexInfinity

>> On[Power::infy]

>> 1 / 0
Infiniteexpression1/0encountered.
ComplexInfinity
```

OutputForm

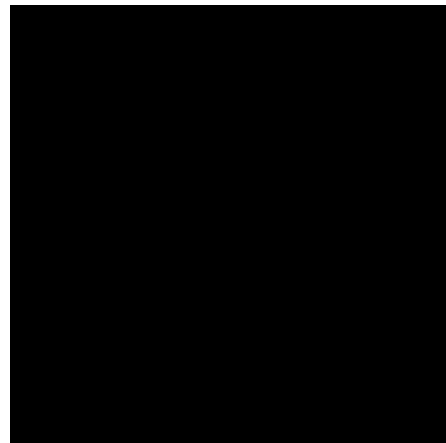
`OutputForm[expr]`
displays *expr* in a plain-text form.

```
>> 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 (//)

$x // f$
is equivalent to $f[x]$.

```
>> 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]
1000.
```

Prefix (@)

$f @ x$
is equivalent to $f[x]$.

```
>> 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]
~ (a + b)

>> q[a*b]
~ ab

>> q[a]+b
b+ ~ 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[expr, ...]`
prints each *expr* in string form.

```
>> 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]
Arguments2and3ofQuiet[expr,
All, All]shouldnotbothbeAll.
Quiet[expr, All, All]

>> Quiet[x + x, {a::b}, {a::b}]
InQuiet[x + x, {a::b},
{a::b}]themessagename(s){a::b}appearinboththelistofofmessage
Quiet[x + x, {a::b}, {a::b}]
```

Right

`Right`

is used with operator formatting constructs to specify a right-associative operator.

Row

`Row[{expr, ...}]`

formats several expressions inside a `RowBox`.

RowBox

`RowBox[{...}]`

is a box construct that represents a sequence of boxes arranged in a horizontal row.

StandardForm

`StandardForm[expr]`

displays *expr* in the default form.

```
>> 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[str, expr1, expr2, ...]`

displays the string *str*, replacing placeholders in *str* with the corresponding expressions.

```
>> StringForm["'1' bla '2' blub '"
bla '2'", a, b, c]
a bla b blub c bla b
```

Style

Subscript

`Subscript[a, i]`
displays as a_i .

```
>> Subscript[x,1,2,3] // TeXForm
x_{1,2,3}
```

SubscriptBox

Subsuperscript

`Subsuperscript[a, b, c]`
displays as a_b^c .

```
>> Subsuperscript[a, b, c] //
TeXForm
a_b^c
```

SubsuperscriptBox

Superscript

`Superscript[x, y]`
displays as x^y .

```
>> Superscript[x,3] // TeXForm
x^3
```

SuperscriptBox

Syntax

`Syntax`

is a symbol to which all syntax messages are assigned.

```
>> 1 +
>> Sin[1]
>> ^ 2
>> 1.5''
```

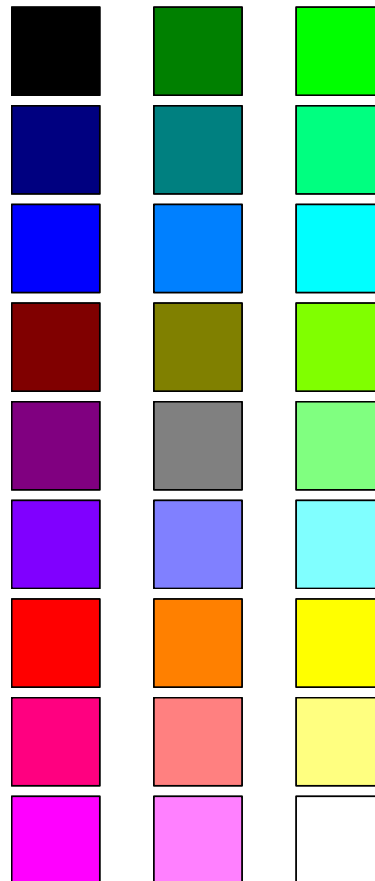
TableForm

`TableForm[expr]`
displays *expr* as a table.

```
>> 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[expr]`
displays *expr* using TeX math mode commands.

```
>> TeXForm[HoldForm[Sqrt[a^3]]]
\sqrt{a^3}
```

ToBoxes

`ToBoxes[expr]`
evaluates *expr* and converts the result to box form.

Unlike `MakeBoxes`, `ToBoxes` evaluates its argument:

```
>> ToBoxes[a + a]
      RowBox[{2, a}]

>> ToBoxes[a + b]
      RowBox[{a, +, b}]

>> ToBoxes[a ^ b] // FullForm
      SuperscriptBox["a", "b"]
```

`$UseSansSerif`

`$UseSansSerif`
specifies the font of the web interface.

When `True`, the output in `MathMLForm` uses SansSerif fonts instead of the standard ones...

XIX. Integer functions

Contents

BitLength	136	Floor	137	IntegerLength	138
Ceiling	136	FromDigits	137	IntegerReverse	138
DigitCount	136	IntegerDigits	137	IntegerString	138

BitLength

BitLength $[x]$
gives the number of bits needed to represent the integer x . x 's sign is ignored.

```
>> BitLength[1023]
10
>> BitLength[100]
7
>> BitLength[-5]
3
>> BitLength[0]
0
```

Ceiling

Ceiling $[x]$
gives the first integer greater than x .

```
>> Ceiling[1.2]
2
>> Ceiling[3/2]
2
```

For complex x , take the ceiling of real and imaginary parts.

```
>> Ceiling[1.3 + 0.7 I]
2 + I
```

DigitCount

DigitCount $[n, b, d]$
returns the number of times digit d occurs in the base b representation of n .

DigitCount $[n, b]$
returns a list indicating the number of times each digit occurs in the base b representation of n .

DigitCount $[n]$
returns a list indicating the number of times each digit occurs in the decimal representation of n .

```
>> DigitCount[1022]
{1, 2, 0, 0, 0, 0, 0, 0, 1}
>> DigitCount[Floor[Pi * 10^100]]
{8, 12, 12, 10, 8, 9, 8, 12, 14, 8}
>> DigitCount[1022, 2]
{9, 1}
>> DigitCount[1022, 2, 1]
9
```

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 and 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
```

FromDigits

FromDigits[l]
returns the integer corresponding to the decimal representation given by l . l can be a list of digits or a string.

FromDigits[l, b]
returns the integer corresponding to the base b representation given by l . l can be a list of digits or a string.

```
>> FromDigits["123"]
123
>> FromDigits[{1, 2, 3}]
123
>> FromDigits[{1, 0, 1}, 1000]
1 000 001
```

FromDigits can handle symbolic input:

```
>> FromDigits[{a, b, c}, 5]
c + 5 (5a + b)
```

Note that FromDigits does not automatically detect if you are providing a non-decimal representation:

```
>> FromDigits["a0"]
100
>> FromDigits["a0", 16]
160
```

FromDigits on empty lists or strings returns 0:

```
>> FromDigits[{}]
0
>> FromDigits[""]
0
```

IntegerDigits

IntegerDigits[n]
returns the decimal representation of integer x as list of digits. x 's sign is ignored.

IntegerDigits[n, b]
returns the base b representation of integer x as list of digits. x 's sign is ignored.

IntegerDigits[$n, b, length$]
returns a list of length $length$. If the number is too short, the list gets padded with 0 on the left. If the number is too long, the $length$ least significant digits are returned.

```
>> IntegerDigits[12345]
{1, 2, 3, 4, 5}
>> IntegerDigits[-500]
{5, 0, 0}
>> IntegerDigits[12345, 10, 8]
{0, 0, 0, 1, 2, 3, 4, 5}
>> IntegerDigits[12345, 10, 3]
{3, 4, 5}
>> IntegerDigits[11, 2]
{1, 0, 1, 1}
>> IntegerDigits[123, 8]
{1, 7, 3}
>> IntegerDigits[98765, 20]
{12, 6, 18, 5}
```

IntegerLength

`IntegerLength[x]`
gives the number of digits in the base-10 representation of x .
`IntegerLength[x, b]`
gives the number of base- b digits in x .

```
>> IntegerLength[123456]
6
>> IntegerLength[10^10000]
10001
>> IntegerLength[-10^1000]
1001
```

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]
```

0 is a special case:

```
>> IntegerLength[0]
0
```

IntegerReverse

`IntegerReverse[n]`
returns the integer that has the reverse decimal representation of x without sign.
`IntegerReverse[n, b]`
returns the integer that has the reverse base b representation of x without sign.

```
>> IntegerReverse[1234]
4321
>> IntegerReverse[1022, 2]
511
```

```
>> IntegerReverse[-123]
321
```

IntegerString

`IntegerString[n]`
returns the decimal representation of integer x as string. x 's sign is ignored.
`IntegerString[n, b]`
returns the base b representation of integer x as string. x 's sign is ignored.
`IntegerString[n, b, length]`
returns a string of length $length$. If the number is too short, the string gets padded with 0 on the left. If the number is too long, the $length$ least significant digits are returned.

For bases > 10 , alphabetic characters a, b, ... are used to represent digits 11, 12, Note that base must be an integer in the range from 2 to 36.

```
>> IntegerString[12345]
12345
>> IntegerString[-500]
500
>> IntegerString[12345, 10, 8]
00012345
>> IntegerString[12345, 10, 3]
345
>> IntegerString[11, 2]
1011
>> IntegerString[123, 8]
173
>> IntegerString[32767, 16]
7fff
>> IntegerString[98765, 20]
c6i5
```

XX. Iohooks

Contents

\$Post	139	\$PrePrint	139	\$SyntaxHandler	139
\$Pre	139	\$PreRead	139		

\$Post

\$Post
is a global variable whose value, if set, is applied to every output expression.

```
>> $Pre = .; $Post = .; $PrePrint =  
    .; $EnlapsedTime = .;  
    [Processinginput...]  
  
>> 2 + 2  
    4
```

\$Pre

\$Pre
is a global variable whose value, if set, is applied to every input expression.

Set *Timing* as the **\$Pre** function, stores the elapsed time in a variable, stores just the result in `Out[$Line]` and print a formatted version showing the elapsed time

```
>> $Pre := (Print["[Processing  
    input...]"; #1]&  
  
>> $Post := (Print["[Storing result  
    ...]"; #1]&  
    [Processinginput...]  
    [Storingresult...]  
  
>> $PrePrint := (Print["The result  
    is:"]; {TimeUsed[], #1})&  
    [Processinginput...]  
    [Storingresult...]  
  
>> 2 + 2  
    [Processinginput...]  
    [Storingresult...]  
    Theresultis :  
    {226.281,4}
```

\$PrePrint

\$PrePrint
is a global variable whose value, if set, is applied to every output expression before it is printed.

\$PreRead

\$PreRead
is a global variable whose value, if set, is applied to the text or box form of every input expression before it is fed to the parser.
(Not implemented yet)

\$SyntaxHandler

\$SyntaxHandler
is a global variable whose value, if set, is applied to any input string that is found to contain a syntax error.
(Not implemented yet)

XXI. Linear algebra

Contents

BrayCurtisDistance . . .	140	FittedModel	142	NullSpace	144
CanberraDistance . . .	140	Inverse	142	PseudoInverse	145
ChessboardDistance . .	140	LeastSquares	142	QRDecomposition . . .	145
CosineDistance	141	LinearModelFit	143	RowReduce	145
Cross	141	LinearSolve	143	SingularValueDecom- position	145
DesignMatrix	141	ManhattanDistance . .	143	SquaredEuclideanDis- tance	145
Det	141	MatrixExp	143	Tr	146
Eigensystem	141	MatrixPower	144	VectorAngle	146
Eigenvalues	141	MatrixRank	144		
Eigenvectors	142	Norm	144		
EuclideanDistance . .	142	Normalize	144		

BrayCurtisDistance

`BrayCurtisDistance[u, v]`
returns the Bray Curtis distance between u and v .

```
>> BrayCurtisDistance[-7, 5]
6

>> BrayCurtisDistance[{-1, -1},
{10, 10}]
11
9
```

CanberraDistance

`CanberraDistance[u, v]`
returns the canberra distance between u and v , which is a weighted version of the Manhattan distance.

```
>> CanberraDistance[-7, 5]
1

>> CanberraDistance[{-1, -1}, {1,
1}]
2
```

ChessboardDistance

`ChessboardDistance[u, v]`
returns the chessboard distance (also known as Chebyshev distance) between u and v , which is the number of moves a king on a chessboard needs to get from square u to square v .

```
>> ChessboardDistance[-7, 5]
12

>> ChessboardDistance[{-1, -1}, {1,
1}]
2
```

CosineDistance

`CosineDistance[u, v]`
returns the cosine distance between u and v .

```
>> N[CosineDistance[{7, 9}, {71,
89}]]
0.0000759646
```

```
>> CosineDistance[{a, b}, {c, d}]
1
+ 
$$\frac{-ac - bd}{\sqrt{\text{Abs}[a]^2 + \text{Abs}[b]^2} \sqrt{\text{Abs}[c]^2 + \text{Abs}[d]^2}}$$

```

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 less than 10.
Cross[{1, 2}, {3, 4, 5}]
```

DesignMatrix

`DesignMatrix[m, f, x]`
returns the design matrix.

```
>> DesignMatrix[{{2, 1}, {3, 4}, {5, 3}, {7, 6}}, x, x]
{{1, 2}, {1, 3}, {1, 5}, {1, 7}}
>> DesignMatrix[{{2, 1}, {3, 4}, {5, 3}, {7, 6}}, f[x], x]
{{1, f[2]}, {1, f[3]}, {1, f[5]}, {1, f[7]}}
```

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 the list `{Eigenvalues[m], Eigenvectors[m]}`.

```
>> Eigensystem[{{1, 1, 0}, {1, 0, 1}, {0, 1, 1}}]
{{2, -1, 1}, {{1, 1, 1}, {1, -2, 1}, {-1, 0, 1}}}
```

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
{1, Cos[theta] +  $\sqrt{-1 + \text{Cos}[\text{theta}]^2}$ , Cos[theta] -  $\sqrt{-1 + \text{Cos}[\text{theta}]^2}$ }
>> 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}}]
{{-0.355518, -1.15048}, {-0.62896, 0.777438}}
```

EuclideanDistance

EuclideanDistance $[u, v]$
returns the euclidean distance between u and v .

```
>> EuclideanDistance[-7, 5]
12

>> EuclideanDistance[{-1, -1}, {1, 1}]
2√2

>> EuclideanDistance[{a, b}, {c, d}]
√Abs[a - c]^2 + Abs[b - d]^2
```

FittedModel

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}}]
{{1, 0, 0}, {0, √3/2, -1/2}, {0, 1/2, √3/2}}
```

LeastSquares

LeastSquares $[m, b]$
computes the least squares solution to $m x = b$, finding an x that solves for b optimally.

```
>> LeastSquares[{{1, 2}, {2, 3}, {5, 6}}, {1, 5, 3}]
{-28/13, 31/13}

>> Simplify[LeastSquares[{{1, 2}, {2, 3}, {5, 6}}, {1, x, 3}]]
{12/13 - 8x/13, -4/13 + 7x/13}

>> LeastSquares[{{1, 1, 1}, {1, 1, 2}}, {1, 3}]
Solving for underdetermined system not implemented.
LeastSquares[{{1, 1, 1}, {1, 1, 2}}, {1, 3}]
```

LinearModelFit

LinearModelFit $[m, f, x]$
returns the design matrix.

```
>> m = LinearModelFit[{{2, 1}, {3, 4}, {5, 3}, {7, 6}}, x, x];
```

```

>> m["BasisFunctions"]
{1, x}

>> m["BestFit"]
0.186441 + 0.779661x

>> m["BestFitParameters"]
{0.186441, 0.779661}

>> m["DesignMatrix"]
{{1, 2}, {1, 3}, {1, 5}, {1, 7}}

>> m["Function"]
0.186441 + 0.779661#1&

>> m["Response"]
{1, 4, 3, 6}

>> m["FitResiduals"]
{-0.745763, 1.47458, -1.08475, 0.355932}

>> m = LinearModelFit[{{2, 2, 1}, {3, 2, 4}, {5, 6, 3}, {7, 9, 6}}, {Sin[x], Cos[y]}, {x, y}];

>> m["BasisFunctions"]
{1, Sin[x], Cos[y]}

>> m["Function"]
3.33077 - 5.65221Cos[#2] - 5.01042Sin[#1]&

>> m = LinearModelFit[{{1, 4}, {1, 5}, {1, 7}}, {1, 2, 3}];

>> m["BasisFunctions"]
{#1, #2}

>> m["FitResiduals"]
{-0.142857, 0.214286, -0.0714286}

```

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}]

```

Linearequationencounteredthathasnosolution.

```

LinearSolve[{1, 2, 3}, {4, 5, 6}, {7, 8, 9}], {1, -2, 3}]

```

ManhattanDistance

`ManhattanDistance[u, v]`
returns the Manhattan distance between *u* and *v*, which is the number of horizontal or vertical moves in the gridlike Manhattan city layout to get from *u* to *v*.

```

>> ManhattanDistance[-7, 5]
12

>> ManhattanDistance[{-1, -1}, {1, 1}]
4

```

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}}]
{{5.16266, 3.02952}, {3.02952, 8.19218}}

```

MatrixPower

`MatrixPower[m, n]`
computes the n th power of a matrix m .

```
>> MatrixPower[{{1, 2}, {1, 1}},  
10]  
{{3 363, 4 756}, {2 378, 3 363}}  
  
>> 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}]  
 $\sqrt{\text{Abs}[a]^2 + \text{Abs}[b]^2 + \text{Abs}[c]^2}$ 
```

```
>> 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}}]

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

>> PseudoInverse[{{1.0, 2.5}, {2.5, 1.0}}]

$$\left\{ \{-0.190476, 0.47619\}, \{0.47619, -0.190476\} \right\}$$

```

QRDecomposition

QRDecomposition $[m]$
computes the QR decomposition of the matrix m .

```
>> QRDecomposition[{{1, 2}, {3, 4}, {5, 6}}]

$$\left\{ \left\{ \left\{ \frac{\sqrt{35}}{35}, \frac{3\sqrt{35}}{35}, \frac{\sqrt{35}}{7} \right\}, \left\{ \frac{13\sqrt{210}}{210}, \frac{2\sqrt{210}}{105}, -\frac{\sqrt{210}}{42} \right\} \right\}, \left\{ \left\{ \sqrt{35}, \frac{44\sqrt{35}}{35} \right\}, \left\{ 0, \frac{2\sqrt{210}}{35} \right\} \right\} \right\}$$

```

RowReduce

RowReduce $[matrix]$
returns the reduced row-echelon form of $matrix$.

```
>> RowReduce[{{1, 0, a}, {1, 1, b}}]

$$\left\{ \{1, 0, a\}, \{0, 1, -a + b\} \right\}$$

```

```
>> 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 $[m]$
calculates the singular value decomposition for the matrix m .

SingularValueDecomposition returns u , s , w such that $m=usv$, $uu=1$, $vv=1$, and s is diagonal.

```
>> SingularValueDecomposition[{{1.5, 2.0}, {2.5, 3.0}}]

$$\left\{ \left\{ \{0.538954, 0.842335\}, \{0.842335, -0.538954\} \right\}, \left\{ \{4.63555, 0\}, \{0., 0.107862\} \right\}, \left\{ \{0.628678, 0.777666\}, \{-0.777666, 0.628678\} \right\} \right\}$$

```

SquaredEuclideanDistance

SquaredEuclideanDistance $[u, v]$
returns squared the euclidean distance between u and v .

```
>> SquaredEuclideanDistance[-7, 5]
144
>> SquaredEuclideanDistance[{-1, -1}, {1, 1}]
8
```

Tr

Tr $[m]$
computes the trace of the matrix m .

```
>> Tr[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]
15
```

Symbolic trace:

```
>> Tr[{a, b, c}, {d, e, f}, {g, h,
    i}]
a + e + i
```

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}]
     $\frac{\text{Pi}}{3}$ 
```

XXII. List functions

Contents

Accumulate	147	IntersectingQ	154	RankedMax	162
All	147	Intersection	154	RankedMin	162
Append	148	Join	155	Reap	163
AppendTo	148	Keys	155	ReplacePart	163
Array	148	Kurtosis	155	Rest	163
Association	149	Last	155	Reverse	164
AssociationQ	149	LeafCount	155	Riffle	164
Cases	149	Length	156	RotateLeft	164
Catenate	149	Level	156	RotateRight	164
CentralMoment	149	LevelQ	157	Select	165
ClusteringComponents	149	List	157	Skewness	165
Complement	150	ListQ	157	Sow	165
ConstantArray	150	Mean	157	Span (;;)	165
ContainsOnly	150	Median	157	Split	165
Correlation	150	MemberQ	157	SplitBy	165
Count	150	Most	158	StandardDeviation	166
Covariance	150	Nearest	158	SubsetQ	166
Delete	151	None	158	Table	166
DeleteCases	151	NotListQ	158	Take	167
DeleteDuplicates	152	PadLeft	159	TakeLargest	167
DisjointQ	152	PadRight	159	TakeLargestBy	167
Drop	152	Part	160	TakeSmallest	167
Extract	152	Partition	160	TakeSmallestBy	167
FindClusters	153	Permutations	161	Tally	168
First	153	Pick	161	Total	168
FirstPosition	153	Position	161	Tuples	168
Fold	153	Prepend	161	Union	169
FoldList	154	PrependTo	161	UnitVector	169
Gather	154	Quantile	162	Values	169
GatherBy	154	Quartiles	162	Variance	169
		Range	162		

Accumulate

`Accumulate[list]`
accumulates the values of *list*, returning a new list.

```
>> 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}}
```

AppendTo

`AppendTo[s, item]`
append *item* to value of *s* and sets *s* to the result.

```
>> s = {};
>> AppendTo[s, 1]
      {1}
>> s
      {1}
```

Append works on expressions with heads other than List:

```
>> y = f[];
>> AppendTo[y, x]
      f[x]
>> y
      f[x]
```

Array

`Array[f, n]`
returns the *n*-element list $\{f[1], \dots, f[n]\}$.
`Array[f, n, a]`
returns the *n*-element list $\{f[a], \dots, f[a + n]\}$.
`Array[f, {n, m}, {a, b}]`
returns an *n*-by-*m* matrix created by applying *f* to indices ranging from (*a*, *b*) to (*a* + *n*, *b* + *m*).
`Array[f, dims, origins, h]`
returns an expression with the specified dimensions and index origins, with head *h* (instead of List).

```
>> 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]
```

Association

`Association[key1 -> val1, key2 -> val2, ...]`
`<|key1 -> val1, key2 -> val2, ...|>`
represents an association between keys and values.

Association is the head of associations:

```
>> Head[<|a -> x, b -> y, c -> z|>]
      Association
>> <|a -> x, b -> y|>
      <|a -> x, b -> y|>
```

```
>> Association[{a -> x, b -> y}]
<|a -> x, b -> y|>
```

Associations can be nested:

```
>> <|a -> x, b -> y, <|a -> z, d ->
t|>|>
<|a -> z, b -> y, d -> t|>
```

AssociationQ

AssociationQ[*expr*]
return True if *expr* is a valid Association object, and False otherwise.

```
>> AssociationQ[<|a -> 1, b -> 2|>]
True
>> AssociationQ[<|a, b|>]
False
```

Cases

Cases[*list*, *pattern*]
returns the elements of *list* that match *pattern*.
Cases[*list*, *pattern*, *ls*]
returns the elements matching at level-spec *ls*.

```
>> Cases[{a, 1, 2.5, "string"},
_Integer|_Real]
{1, 2.5}
>> Cases[_Complex] [{1, 2I, 3, 4-I,
5}]
{2I, 4 - I}
```

Catenate

Catenate[{*l1*, *l2*, ...}]
concatenates the lists *l1*, *l2*, ...

```
>> Catenate[{ {1, 2, 3}, {4, 5} }]
{1, 2, 3, 4, 5}
```

CentralMoment

CentralMoment[*list*, *r*]
gives the the *r*th central moment (i.e. the *r*th moment about the mean) of *list*.

```
>> CentralMoment[{1.1, 1.2, 1.4,
2.1, 2.4}, 4]
0.100845
```

ClusteringComponents

ClusteringComponents[*list*]
forms clusters from *list* and returns a list of cluster indices, in which each element shows the index of the cluster in which the corresponding element in *list* ended up.

ClusteringComponents[*list*, *k*]
forms *k* clusters from *list* and returns a list of cluster indices, in which each element shows the index of the cluster in which the corresponding element in *list* ended up.

For more detailed documentation regarding options and behavior, see **FindClusters**[].

```
>> ClusteringComponents[{1, 2, 3,
1, 2, 10, 100}]
{1, 1, 1, 1, 1, 1, 2}
>> ClusteringComponents[{10, 100,
20}, Method -> "KMeans"]
{1, 0, 1}
```

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. The expression will be sorted and each element will only occur once.

```
>> 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]

>> Complement[{c, b, a}]
      {a, b, c}
```

ConstantArray

`ConstantArray[expr, n]`
returns a list of n copies of *expr*.

```
>> ConstantArray[a, 3]
      {a, a, a}

>> ConstantArray[a, {2, 3}]
      {{a, a, a}, {a, a, a}}
```

ContainsOnly

`ContainsOnly[list1, list2]`
yields True if *list1* contains only elements that appear in *list2*.

```
>> ContainsOnly[{b, a, a}, {a, b, c}]
      True
```

The first list contains elements not present in the second list:

```
>> ContainsOnly[{b, a, d}, {a, b, c}]
      False

>> ContainsOnly[{}, {a, b, c}]
      True
```

Use `Equal` as the comparison function to have numerical tolerance:

```
>> ContainsOnly[{a, 1.0}, {1, a, b}, {SameTest -> Equal}]
      True
```

Correlation

`Correlation[a, b]`
computes Pearson's correlation of two equal-sized vectors *a* and *b*.

An example from Wikipedia:

```
>> Correlation[{10, 8, 13, 9, 11,
      14, 6, 4, 12, 7, 5}, {8.04,
      6.95, 7.58, 8.81, 8.33, 9.96,
      7.24, 4.26, 10.84, 4.82, 5.68}]
      0.816421
```

Count

`Count[list, pattern]`
returns the number of times *pattern* appears in *list*.
`Count[list, pattern, ls]`
counts the elements matching at level-spec *ls*.

```
>> Count[{3, 7, 10, 7, 5, 3, 7,
      10}, 3]
      2

>> Count[{{a, a}, {a, a, a}, a}, a, {2}]
      5
```

Covariance

`Covariance[a, b]`
computes the covariance between the equal-sized vectors *a* and *b*.

```
>> Covariance[{0.2, 0.3, 0.1},
      {0.3, 0.3, -0.2}]
      0.025
```

Delete

```
Delete[expr, i]
  deletes the element at position i in expr.
  The position is counted from the end if i
  is negative.
Delete[expr, {m, n, ...}]
  deletes the element at position {m, n, ...}.
Delete[expr, {{m1, n1, ...}, {m2,
n2, ...}, ...}]
  deletes the elements at several positions.
```

Delete the element at position 3:

```
>> Delete[{a, b, c, d}, 3]
{a, b, d}
```

Delete at position 2 from the end:

```
>> Delete[{a, b, c, d}, -2]
{a, b, d}
```

Delete at positions 1 and 3:

```
>> Delete[{a, b, c, d}, {{1}, {3}}]
{b, d}
```

Delete in a 2D array:

```
>> Delete[{{a, b}, {c, d}}, {2, 1}]
{{a, b}, {d}}
```

Deleting the head of a whole expression gives a Sequence object:

```
>> Delete[{a, b, c}, 0]
Sequence[a, b, c]
```

Delete in an expression with any head:

```
>> Delete[f[a, b, c, d], 3]
f[a, b, d]
```

Delete a head to splice in its arguments:

```
>> Delete[f[a, b, u + v, c], {3, 0}]
f[a, b, u, v, c]
```

```
>> Delete[{a, b, c}, 0]
Sequence[a, b, c]
```

Delete without the position:

```
>> Delete[{a, b, c, d}]
Delete called with 1 argument; 2 arguments are expected.
Delete[{a, b, c, d}]
```

Delete with many arguments:

```
>> Delete[{a, b, c, d}, 1, 2]
Delete called with 3 arguments; 2 arguments are expected.
Delete[{a, b, c, d}, 1, 2]
```

Delete the element out of range:

```
>> Delete[{a, b, c, d}, 5]
Part{5} of {a, b, c, d} does not exist.
Delete[{a, b, c, d}, 5]
```

Delete the position not integer:

```
>> Delete[{a, b, c, d}, {1, n}]
Position specification n in {a, b, c, d} is not a machine
– sized integer or a list of machine
– sized integers.
Delete[{a, b, c, d}, {1, n}]
```

DeleteCases

```
DeleteCases[list, pattern]
  returns the elements of list that do not
  match pattern.
```

```
>> DeleteCases[{a, 1, 2.5, "string"}, _Integer | _Real]
{a, string}
>> DeleteCases[{a, b, 1, c, 2, 3}, _Symbol]
{1, 2, 3}
```

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 does
  not change the order of the remaining 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}
```

```
>> Extract[{a, b}, {c, d}], {{1},
{2, 2}}]
{{a,b},d}
```

DisjointQ

`DisjointQ[a, b]`
gives True if \$a and \$b are disjoint, or False if \$a and \$b have any common elements.

Drop

`Drop[expr, n]`
returns *expr* with the first *n* leaves removed.

```
>> 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
```

FindClusters

`FindClusters[list]`
returns a list of clusters formed from the elements of *list*. The number of cluster is determined automatically.
`FindClusters[list, k]`
returns a list of *k* clusters formed from the elements of *list*.

```
>> FindClusters[{1, 2, 20, 10, 11, 40, 19, 42}]
{{1,2,20,10,11,19}, {40,42}}
>> FindClusters[{25, 100, 17, 20}]
{{25,17,20}, {100}}
>> FindClusters[{3, 6, 1, 100, 20, 5, 25, 17, -10, 2}]
{{3,6,1,5, -10,2}, {100}, {20,25,17}}
>> FindClusters[{1, 2, 10, 11, 20, 21}]
{{1,2}, {10,11}, {20,21}}
>> FindClusters[{1, 2, 10, 11, 20, 21}, 2]
{{1,2,10,11}, {20,21}}
>> FindClusters[{1 -> a, 2 -> b, 10 -> c}]
{{a,b}, {c}}
>> FindClusters[{1, 2, 5} -> {a, b, c}]
{{a,b}, {c}}
>> FindClusters[{1, 2, 3, 1, 2, 10, 100}, Method -> "Agglomerate"]
{{1,2,3,1,2,10}, {100}}
>> FindClusters[{1, 2, 3, 10, 17, 18}, Method -> "Agglomerate"]
{{1,2,3}, {10}, {17,18}}
```



```
>> FindClusters[{{1}, {5, 6}, {7},
{2, 4}}, DistanceFunction -> (
Abs[Length[#1] - Length[#2]]&)]
{{{1}, {7}}, {{5, 6}, {2, 4}}}

>> FindClusters[{"meep", "heap", "
deep", "weep", "sheep", "leap",
"keep"}, 3]
{{meep, deep, weep, keep},
{heap, leap}, {sheep}}
```

FindClusters' automatic distance function detection supports scalars, numeric tensors, boolean vectors and strings.

The Method option must be either "Agglomerate" or "Optimize". If not specified, it defaults to "Optimize". Note that the Agglomerate and Optimize methods usually produce different clusterings.

The runtime of the Agglomerate method is quadratic in the number of clustered points n , builds the clustering from the bottom up, and is exact (no element of randomness). The Optimize method's runtime is linear in n , Optimize builds the clustering from top down, and uses random sampling.

First

First[*expr*]
returns the first element 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]
```

FirstPosition

FirstPosition[*expr*, *pattern*]
gives the position of the first element in *expr* that matches *pattern*, or Missing["NotFound"] if no such element is found.

FirstPosition[*expr*, *pattern*, *default*]
gives default if no element matching *pattern* is found.

FirstPosition[*expr*, *pattern*, *default*, *levelspec*]
finds only objects that appear on levels specified by *levelspec*.

```
>> FirstPosition[{a, b, a, a, b, c,
b}, b]
{2}

>> FirstPosition[{{a, a, b}, {b, a,
a}, {a, b, a}}, b]
{1, 3}

>> FirstPosition[{x, y, z}, b]
Missing[NotFound]
```

Find the first position at which x^2 appears:

```
>> FirstPosition[{1 + x^2, 5, x^4,
a + (1 + x^2)^2}, x^2]
{1, 2}
```

Fold

Fold[*f*, *x*, *list*]
returns the result of iteratively applying the binary operator *f* to each element of *list*, starting with *x*.

Fold[*f*, *list*]
is equivalent to **Fold**[*f*, **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[f, x, list]`
returns a list starting with x , where each element is the result of applying the binary operator f to the previous result and the next element of $list$.

`FoldList[f, list]`
is equivalent to `FoldList[f, First[list], Rest[list]]`.

```
>> 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}
```

Gather

`Gather[list, test]`
gathers leaves of $list$ into sub lists of items that are the same according to $test$.

`Gather[list]`
gathers leaves of $list$ into sub lists of items that are the same.

The order of the items inside the sub lists is the same as in the original list.

```
>> Gather[{1, 7, 3, 7, 2, 3, 9}]
{{1}, {7, 7}, {3, 3}, {2}, {9}}

>> Gather[{1/3, 2/6, 1/9}]
{{1/3, 1/3}, {1/9, 1/9}}
```

GatherBy

`GatherBy[list, f]`
gathers leaves of $list$ into sub lists of items whose image under f identical.

`GatherBy[list, {f, g, ...}]`
gathers leaves of $list$ into sub lists of items whose image under f identical. Then, gathers these sub lists again into sub sub lists, that are identical under g .

```
>> GatherBy[{{1, 3}, {2, 2}, {1, 1}}, Total]
{{{1, 3}, {2, 2}}, {{1, 1}}}]

>> GatherBy[{"xy", "abc", "ab"}, StringLength]
{{xy, ab}, {abc}}

>> GatherBy[{{2, 0}, {1, 5}, {1, 0}}, Last]
{{{2, 0}, {1, 0}}, {{1, 5}}}]

>> GatherBy[{{1, 2}, {2, 1}, {3, 5}, {5, 1}, {2, 2, 2}}, {Total, Length}]
{{{1, 2}, {2, 1}}, {{3, 5}}, {{5, 1}}, {{2, 2, 2}}}]
```

IntersectingQ

`IntersectingQ[a, b]`
gives True if there are any common elements in a and b , or False if a and b are disjoint.

Intersection

`Intersection[a, b, ...]`
gives the intersection of the sets. The resulting list will be sorted and each element will only occur once.

```
>> Intersection[{1000, 100, 10, 1}, {1, 5, 10, 15}]
{1, 10}

>> Intersection[{{a, b}, {x, y}}, {{x, x}, {x, y}, {x, z}}]
{{x, y}}

>> Intersection[{c, b, a}]
{a, b, c}

>> Intersection[{1, 2, 3}, {2, 3, 4}, SameTest->Less]
{3}
```

Join

```
Join[l1, l2]
concatenates the lists l1 and l2.
```

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]
      HeadsPlusandTimesareexpectedtobethesame.
Join[a + b, cd]
```

Keys

```
Keys[<|key1 -> val1, key2 -> val2,
...|>]
return a list of the keys keyi in an association.
Keys[{key1 -> val1, key2 -> val2, ...}]
return a list of the keyi in a list of rules.
```

```
>> Keys[<|a -> x, b -> y|>]
      {a, b}

>> Keys[{a -> x, b -> y}]
      {a, b}
```

Keys automatically threads over lists:

```
>> Keys[<|a -> x, b -> y|>, {w ->
      z, {}}]
      {{a, b}, {w, {}}}
```

Keys are listed in the order of their appearance:

```
>> Keys[{c -> z, b -> y, a -> x}]
      {c, b, a}
```

Kurtosis

```
Kurtosis[list]
gives the Pearson measure of kurtosis for
list (a measure of existing outliers).
```

```
>> Kurtosis[{1.1, 1.2, 1.4, 2.1,
      2.4}]
      1.42098
```

Last

```
Last[expr]
returns the last element in expr.
```

Last[expr] is equivalent to expr[[-1]] .

```
>> Last[{a, b, c}]
      c

>> Last[x]
      Nonatomicexpressionexpected.
Last[x]
```

LeafCount

```
LeafCount[expr]
returns the total number of indivisible
subexpressions in expr.
```

```
>> LeafCount[1 + x + y^a]
      6

>> LeafCount[f[x, y]]
      3

>> LeafCount[{1 / 3, 1 + I}]
      7

>> LeafCount[Sqrt[2]]
      5

>> LeafCount[100!]
      1
```

Length

`Length[expr]`
returns the number of leaves in *expr*.

Length of a list:

```
>> 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]
{List, List, List, {a}, {{a}}, {{{a}}}}
```

```
>> Level[x^2 + y^3, 3, Heads ->
True]
{Plus, Power, x, 2, x^2, Power, y, 3, y^3}
```

```
>> Level[a ^ 2 + 2 * b, {-1}, Heads
-> True]
{Plus, Power, a, 2, 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[e1, e2, ..., ei]`
`{e1, e2, ..., ei}`
 represents a list containing the elements $e1...ei$.

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
```

Mean

`Mean[list]`
 returns the statistical mean of *list*.

```
>> Mean[{26, 64, 36}]
42

>> Mean[{1, 1, 2, 3, 5, 8}]
 $\frac{10}{3}$ 
```

```
>> Mean[{a, b}]
 $\frac{a + b}{2}$ 
```

Median

`Median[list]`
 returns the median of *list*.

```
>> Median[{26, 64, 36}]
36
```

For lists with an even number of elements, Median returns the mean of the two middle values:

```
>> Median[{-11, 38, 501, 1183}]
 $\frac{539}{2}$ 
```

Passing a matrix returns the medians of the respective columns:

```
>> Median[{{100, 1, 10, 50}, {-1,
1, -2, 2}}]
 $\left\{\frac{99}{2}, 1, 4, 26\right\}$ 
```

MemberQ

`MemberQ[list, pattern]`
 returns True if *pattern* matches any element of *list*, or False otherwise.

```
>> MemberQ[{a, b, c}, b]
True

>> MemberQ[{a, b, c}, d]
False

>> MemberQ[{"a", b, f[x]}, _?
NumericQ]
False

>> MemberQ[_List][{{}}]
True
```

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]
Nonatomicexpressionexpected.
Most[x]
```

Nearest

`Nearest[list, x]`
returns the one item in *list* that is nearest to *x*.

`Nearest[list, x, n]`
returns the *n* nearest items.

`Nearest[list, x, {n, r}]`
returns up to *n* nearest items that are not farther from *x* than *r*.

`Nearest[{p1 -> q1, p2 -> q2, ...}, x]`
returns *q1, q2, ...* but measures the distances using *p1, p2, ...*

`Nearest[{p1, p2, ...} -> {q1, q2, ...}, x]`
returns *q1, q2, ...* but measures the distances using *p1, p2, ...*

```
>> Nearest[{5, 2.5, 10, 11, 15, 8.5, 14}, 12]
{11}
```

Return all items within a distance of 5:

```
>> Nearest[{5, 2.5, 10, 11, 15, 8.5, 14}, 12, {All, 5}]
{11, 10, 14}

>> Nearest[{Blue -> "blue", White -> "white", Red -> "red", Green -> "green"}, {Orange, Gray}]
{{red}, {white}}
```

```
>> Nearest[{{0, 1}, {1, 2}, {2, 3}} -> {a, b, c}, {1.1, 2}]
{b}
```

None

`None`
is a possible value for `Span` and `Quiet`.

NotListQ

`NotListQ[expr]`
returns true if *expr* is not a list.

PadLeft

`PadLeft[list, n]`
pads *list* to length *n* by adding 0 on the left.

`PadLeft[list, n, x]`
pads *list* to length *n* by adding *x* on the left.

`PadLeft[list, {n1, $n2, ...}, x]`
pads *list* to lengths *n1, n2* at levels 1, 2, ... respectively by adding *x* on the left.

`PadLeft[list, n, x, m]`
pads *list* to length *n* by adding *x* on the left and adding a margin of *m* on the right.

`PadLeft[list, n, x, {m1, m2, ...}]`
pads *list* to length *n* by adding *x* on the left and adding margins of *m1, m2, ...* on levels 1, 2, ... on the right.

`PadLeft[list]`
turns the ragged list *list* into a regular list by adding 0 on the left.

```
>> PadLeft[{1, 2, 3}, 5]
{0, 0, 1, 2, 3}

>> PadLeft[x[a, b, c], 5]
x[0, 0, a, b, c]

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

```
>> PadLeft[{}, {1, 2}, {1, 2, 3}]
{{0,0,0}, {0,1,2}, {1,2,3}}

>> PadLeft[{1, 2, 3}, 10, {a, b, c}, 2]
{b,c,a,b,c,1,2,3,a,b}

>> PadLeft[{{1, 2, 3}}, {5, 2}, x, 1]
{{x,x}, {x,x}, {x,x}, {3,x}, {x,x}}
```

PadRight

`PadRight[list, n]`
pads *list* to length *n* by adding 0 on the right.

`PadRight[list, n, x]`
pads *list* to length *n* by adding *x* on the right.

`PadRight[list, {n1, $n2, ...}, x]`
pads *list* to lengths *n1, n2* at levels 1, 2, ... respectively by adding *x* on the right.

`PadRight[list, n, x, m]`
pads *list* to length *n* by adding *x* on the left and adding a margin of *m* on the left.

`PadRight[list, n, x, {m1, m2, ...}]`
pads *list* to length *n* by adding *x* on the right and adding margins of *m1, m2, ...* on levels 1, 2, ... on the left.

`PadRight[list]`
turns the ragged list *list* into a regular list by adding 0 on the right.

```
>> PadRight[{1, 2, 3}, 5]
{1,2,3,0,0}

>> PadRight[x[a, b, c], 5]
x[a,b,c,0,0]

>> PadRight[{1, 2, 3}, 2]
{1,2}

>> PadRight[{{}}, {1, 2}, {1, 2, 3}]
{{0,0,0}, {1,2,0}, {1,2,3}}

>> PadRight[{1, 2, 3}, 10, {a, b, c}, 2]
{b,c,1,2,3,a,b,c,a,b}
```

```
>> PadRight[{{1, 2, 3}}, {5, 2}, x, 1]
{{x,x}, {x,1}, {x,x}, {x,x}, {x,x}}
```

Part

`Part[expr, i]`
returns part *i* of *expr*.

Extract an element from a list:

```
>> A = {a, b, c, d};

>> A[[3]]
c
```

Negative indices 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,c,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*.
`Partition[list, n, d]`
partitions *list* into sublists of length *n*
which overlap *d* indices.

```
>> 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}}
```

Permutations

`Permutations[list]`
gives all possible orderings of the items
in *list*.
`Permutations[list, n]`
gives permutations up to length *n*.
`Permutations[list, {n}]`
gives permutations of length *n*.

```
>> Permutations[{y, 1, x}]
{{y,1,x}, {y,x,1}, {1,y,x},
 {1,x,y}, {x,y,1}, {x,1,y}}
```

Elements are differentiated by their position in *list*, not their value.

```
>> Permutations[{a, b, b}]
{{a,b,b}, {a,b,b}, {b,a,b},
 {b,b,a}, {b,a,b}, {b,b,a}}
```

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

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


Pick

```
Pick[list, sel]
  returns those items in list that are True in
  sel.
Pick[list, sel, patt]
  returns those items in list that match patt
  in sel.
```

```
>> Pick[{a, b, c}, {False, True,
False}]
{b}

>> Pick[f[g[1, 2], h[3, 4]], {{True
, False}, {False, True}}]
f[g[1], h[4]]

>> Pick[{a, b, c, d, e}, {1, 2,
3.5, 4, 5.5}, _Integer]
{a, b, d}
```

Position

```
Position[expr, patt]
  returns the list of positions for which expr
  matches patt.
Position[expr, patt, ls]
  returns the positions on levels specified
  by levelspec ls.
```

```
>> Position[{1, 2, 2, 1, 2, 3, 2},
2]
{{2}, {3}, {5}, {7}}
```

Find positions upto 3 levels deep

```
>> Position[{1 + Sin[x], x, (Tan[x]
- y)^2}, x, 3]
{{1, 2, 1}, {2}}
```

Find all powers of x

```
>> Position[{1 + x^2, x y ^ 2, 4 y,
x ^ z}, x^_]
{{1, 2}, {4}}
```

Use Position as an operator

```
>> Position[_Integer][{1.5, 2,
2.5}]
{{2}}
```

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}
```

PrependTo

```
PrependTo[s, item]
  prepends item to value of s and sets s to
  the result.
```

Assign *s* to a list

```
>> s = {1, 2, 4, 9}
{1, 2, 4, 9}
```

Add a new value at the beginning of the list:

```
>> PrependTo[s, 0]
{0, 1, 2, 4, 9}
```

The value assigned to *s* has changed:

```
>> s
{0, 1, 2, 4, 9}
```

PrependTo works with a head other than List:

```
>> y = f[a, b, c];
```

```
>> PrependTo[y, x]
f[x, a, b, c]
```

```
>> y
f[x, a, b, c]
```

Quantile

`Quantile[list, q]`
returns the q th quantile of *list*.

```
>> Quantile[Range[11], 1/3]
4
>> Quantile[Range[16], 1/4]
5
```

Quartiles

`Quartiles[list]`
returns the 1/4, 1/2, and 3/4 quantiles of *list*.

```
>> Quartiles[Range[25]]
{ 27/4, 13, 77/4 }
```

Range

`Range[n]`
returns a list of integers from 1 to n .
`Range[a, b]`
returns a list of integers from a to b .

```
>> Range[5]
{1, 2, 3, 4, 5}
>> Range[-3, 2]
{-3, -2, -1, 0, 1, 2}
>> Range[0, 2, 1/3]
{0, 1/3, 2/3, 1, 4/3, 5/3, 2}
```

RankedMax

`RankedMax[list, n]`
returns the n th largest element of *list* (with $n = 1$ yielding the largest element, $n = 2$ yielding the second largest element, and so on).

```
>> RankedMax[{482, 17, 181, -12},
2]
181
```

RankedMin

`RankedMin[list, n]`
returns the n th smallest element of *list* (with $n = 1$ yielding the smallest element, $n = 2$ yielding the second smallest element, and so on).

```
>> RankedMin[{482, 17, 181, -12},
2]
17
```

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 $f[\text{tag}, \{e1, e2, \dots\}]$.

```
>> 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
{Null, {f[1, {b}]}}
```

When no value is sown, an empty list is returned:

```
>> Reap[x]
{x, {}}
```

ReplacePart

```
ReplacePart[expr, i -> new]
  replaces part i in expr with new.
ReplacePart[expr, {{i, j} -> e1, {k,
l} -> e2}]
  replaces parts i and j with e1, and parts k
and l with e2.
```

```
>> 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 re-
  moved.
```

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]
```

Reverse

```
Reverse[expr]
  reverses the order of expr's items (on the
  top level)
Reverse[expr, n]
  reverses the order of items in expr on level
  n
Reverse[expr, {n1, n2, ...}]
  reverses the order of items in expr on lev-
  els n1, n2, ...
```

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

>> Reverse[x[a, b, c]]
x[c,b,a]
```

```
>> Reverse[{{1, 2}, {3, 4}}, 1]
{{3,4}, {1,2}}

>> Reverse[{{1, 2}, {3, 4}}, 2]
{{2,1}, {4,3}}

>> Reverse[{{1, 2}, {3, 4}}, {1,
2}]
{{4,3}, {2,1}}
```

Riffle

`Riffle[list, x]`
inserts a copy of *x* between each element of *list*.

`Riffle[{a1, a2, ...}, {b1, b2, ...}]`
interleaves the elements of both lists, returning {a1, b1, a2, b2, ...}.

```
>> 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}
```

RotateLeft

`RotateLeft[expr]`
rotates the items of *expr* by one item to the left.

`RotateLeft[expr, n]`
rotates the items of *expr* by *n* items to the left.

`RotateLeft[expr, {n1, n2, ...}]`
rotates the items of *expr* by *n1* items to the left at the first level, by *n2* items to the left at the second level, and so on.

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

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

```
>> RotateLeft[x[a, b, c], 2]
x[c,a,b]

>> RotateLeft[{{a, b, c}, {d, e, f
}, {g, h, i}}, {1, 2}]
{{f,d,e}, {i,g,h}, {c,a,b}}
```

RotateRight

`RotateRight[expr]`
rotates the items of *expr* by one item to the right.

`RotateRight[expr, n]`
rotates the items of *expr* by *n* items to the right.

`RotateRight[expr, {n1, n2, ...}]`
rotates the items of *expr* by *n1* items to the right at the first level, by *n2* items to the right at the second level, and so on.

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

>> RotateRight[Range[10], 3]
{8,9,10,1,2,3,4,5,6,7}

>> RotateRight[x[a, b, c], 2]
x[b,c,a]

>> RotateRight[{{a, b, c}, {d, e, f
}, {g, h, i}}, {1, 2}]
{{h,i,g}, {b,c,a}, {e,f,d}}
```

Select

`Select[{e1, e2, ...}, f]`
returns a list of the elements *ei* for which *f*[*ei*] returns True.

Find numbers greater than zero:

```
>> Select[{-3, 0, 1, 3, a}, #>0&]
{1,3}
```

Select works on an expression with any head:

```
>> Select[f[a, 2, 3], NumberQ]
f[2,3]
```

```
>> Select[a, True]
Nonatomicexpressionexpected.
Select[a, True]
```

Skewness

Skewness[*list*]
gives Pearson's moment coefficient of skewness for *list* (a measure for estimating the symmetry of a distribution).

```
>> Skewness[{1.1, 1.2, 1.4, 2.1, 2.4}]
0.407041
```

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, 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 -> a, x -> y}, {2 -> a}, {z -> c, z -> a}}
```

SplitBy

SplitBy[*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]
{{1, 4/3}, {5/3, 2, 7/3}, {8/3, 3}}

>> SplitBy[{1, 2, 1, 1.2}, {Round, Identity}]
{{{1}}, {{2}}, {{1}}, {1.2}}}
```

StandardDeviation

`StandardDeviation[list]`

computes the standard deviation of *list*.
list may consist of numerical values or symbols. Numerical values may be real or complex.

`StandardDeviation[{a1, a2, ...}, {b1, b2, ...}, ...]` will yield `{StandardDeviation[{a1, b1, ...}, StandardDeviation[{a2, b2, ...}], ...}`.

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

>> StandardDeviation[{7, -5, 101,
100}]

$$\frac{\sqrt{13297}}{2}$$


>> StandardDeviation[{a, a}]
0

>> StandardDeviation[{{1, 10}, {-1,
20}}]

$$\{\sqrt{2}, 5\sqrt{2}\}$$

```

SubsetQ

`SubsetQ[list1, list2]`

returns True if *list2* is a subset of *list1*, and False otherwise.

```
>> SubsetQ[{1, 2, 3}, {3, 1}]
True
```

The empty list is a subset of every list:

```
>> SubsetQ[{}, {}]
True
```

```
>> SubsetQ[{1, 2, 3}, {}]
True
```

Every list is a subset of itself:

```
>> SubsetQ[{1, 2, 3}, {1, 2, 3}]
True
```

Table

`Table[expr, {i, n}]`

evaluates *expr* with *i* ranging from 1 to *n*, returning a list of the results.

`Table[expr, {i, start, stop, step}]`

evaluates *expr* with *i* ranging from *start* to *stop*, incrementing by *step*.

`Table[expr, {i, {e1, e2, ..., ei}}]`

evaluates *expr* with *i* taking on the values *e1, e2, ..., ei*.

```
>> 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}]
{Pi,  $\frac{3\text{Pi}}{2}$ , 2Pi}

>> 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[expr, n]`

returns *expr* with all but the first *n* leaves removed.

```
>> 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}}
```

TakeLargest

`TakeLargest[list, f, n]`
returns the a sorted list of the n largest items in *list*.

```
>> TakeLargest[{100, -1, 50, 10},
               2]
      {100, 50}
```

None, Null, Indeterminate and expressions with head Missing are ignored by default:

```
>> TakeLargest[{-8, 150, Missing[
abc]}, 2]
      {150, -8}
```

You may specify which items are ignored using the option `ExcludedForms`:

```
>> TakeLargest[{-8, 150, Missing[
abc]}, 2, ExcludedForms -> {}]
      {Missing[abc], 150}
```

TakeLargestBy

`TakeLargestBy[list, f, n]`
returns the a sorted list of the n largest items in *list* using *f* to retrieve the items' keys to compare them.

For details on how to use the `ExcludedForms` option, see `TakeLargest[]`.

```
>> TakeLargestBy[{{1, -1}, {10,
100}, {23, 7, 8}, {5, 1}}, Total
, 2]
      {{10, 100}, {23, 7, 8}}

>> TakeLargestBy[{"abc", "ab", "x
"}, StringLength, 1]
      {abc}
```

TakeSmallest

`TakeSmallest[list, f, n]`
returns the a sorted list of the n smallest items in *list*.

For details on how to use the `ExcludedForms` option, see `TakeLargest[]`.

```
>> TakeSmallest[{100, -1, 50, 10},
               2]
      {-1, 10}
```

TakeSmallestBy

`TakeSmallestBy[list, f, n]`
returns the a sorted list of the n smallest items in *list* using *f* to retrieve the items' keys to compare them.

For details on how to use the `ExcludedForms` option, see `TakeLargest[]`.

```
>> TakeSmallestBy[{{1, -1}, {10,
100}, {23, 7, 8}, {5, 1}}, Total
, 2]
      {{1, -1}, {5, 1}}

>> TakeSmallestBy[{"abc", "ab", "x
"}, StringLength, 1]
      {x}
```

Tally

Tally[*list*]
counts and returns the number of occurrences of objects and returns the result as a list of pairs {object, count}.

Tally[*list*, *test*]
counts the number of occurrences of objects and uses *\$test* to determine if two objects should be counted in the same bin.

```
>> Tally[{a, b, c, b, a}]
{{a,2}, {b,2}, {c,1}}
```

Tally always returns items in the order as they first appear in *list*:

```
>> Tally[{b, b, a, a, a, d, d, d, d, c}]
{{b,2}, {a,3}, {d,4}, {c,1}}
```

Total

Total[*list*]
adds all values in *list*.

Total[*list*, *n*]
adds all values up to level *n*.

Total[*list*, {*n*}]
totals only the values at level {*n*}.

Total[*list*, {*n*₁, *n*₂}]
totals at levels {*n*₁, *n*₂}.

```
>> 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[{*list*₁, *list*₂, ...}]
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}}
```

Union

Union[*a*, *b*, ...]
gives the union of the given set or sets. The resulting list will be sorted and each element will only occur once.

```
>> Union[{5, 1, 3, 7, 1, 8, 3}]
{1,3,5,7,8}
```

```
>> Union[{a, b, c}, {c, d, e}]
{a,b,c,d,e}
```

```
>> Union[{c, b, a}]
{a,b,c}
```



```
>> Union[{{a, 1}, {b, 2}}, {{c, 1},
      {d, 3}}, SameTest->(SameQ[Last
      [#1], Last[#2]]&)]
      {{b, 2}, {c, 1}, {d, 3}}
>> Union[{1, 2, 3}, {2, 3, 4},
      SameTest->Less]
      {1, 2, 2, 3, 4}
```

UnitVector

`UnitVector[n, k]`
 returns the n -dimensional unit vector
 with a 1 in position k .
`UnitVector[k]`
 is equivalent to `UnitVector[2, k]`.

```
>> UnitVector[2]
      {0, 1}
>> UnitVector[4, 3]
      {0, 0, 1, 0}
```

Values

`Values[<|key1 -> val1, key2 -> val2, ...|>]`
 return a list of the values val_i in an asso-
 ciation.
`Values[{key1 -> val1, key2 -> val2, ...}]`
 return a list of the val_i in a list of rules.

```
>> Values[<|a -> x, b -> y|>]
      {x, y}
>> Values[{a -> x, b -> y}]
      {x, y}
```

Values automatically threads over lists:

```
>> Values[{<|a -> x, b -> y|>, {c
      -> z, {}}}]
      {{x, y}, {z, {}}}
```

Values are listed in the order of their appearance:

```
>> Values[{c -> z, b -> y, a -> x}]
      {z, y, x}
```

Variance

`Variance[list]`
 computes the variance of $list$. $list$ may
 consist of numerical values or symbols.
 Numerical values may be real or com-
 plex.
`Variance[{a1, a2, ...}, {b1, b2, ...}, ...]` will yield
`{Variance[{a1, b1, ...}, Variance[{a2, b2, ...}], ...}`.

```
>> Variance[{1, 2, 3}]
      1
>> Variance[{7, -5, 101, 3}]
      7475
      3
>> Variance[{1 + 2I, 3 - 10I}]
      74
>> Variance[{a, a}]
      0
>> Variance[{{1, 3, 5}, {4, 10,
      100}}]
      { 9 49 9025 }
      { 2' 2' 2 }
```

XXIII. Logic

Contents

AllTrue	170	False	171	Or ()	171
And (&&)	170	Implies (=>)	171	True	171
AnyTrue	170	NoneTrue	171	Xor (xor)	172
Equivalent (==)	171	Not (!)	171		

AllTrue

AllTrue[{*expr1*, *expr2*, ...}, *test*]
returns True if all applications of *test* to *expr1*, *expr2*, ... evaluate to True.

AllTrue[*list*, *test*, *level*]
returns True if all applications of *test* to items of *list* at *level* evaluate to True.

AllTrue[*test*]
gives an operator that may be applied to expressions.

```
>> AllTrue[{2, 4, 6}, EvenQ]
True
>> AllTrue[{2, 4, 7}, EvenQ]
False
```

And (&&)

And[*expr1*, *expr2*, ...]
expr1 && *expr2* && ...
evaluates each expression in turn, returning False as soon as an expression evaluates to False. If all expressions evaluate to True, And returns True.

```
>> True && True && False
False
```

If an expression does not evaluate to True or False, And returns a result in symbolic form:

```
>> a && b && True && c
a&& b&& c
```

AnyTrue

AnyTrue[{*expr1*, *expr2*, ...}, *test*]
returns True if any application of *test* to *expr1*, *expr2*, ... evaluates to True.

AnyTrue[*list*, *test*, *level*]
returns True if any application of *test* to items of *list* at *level* evaluates to True.

AnyTrue[*test*]
gives an operator that may be applied to expressions.

```
>> AnyTrue[{1, 3, 5}, EvenQ]
False
>> AnyTrue[{1, 4, 5}, EvenQ]
True
```

Equivalent (==)

Equivalent[*expr1*, *expr2*, ...]
expr1 == *expr2* == ...
is equivalent to (*expr1* && *expr2* && ...) || (!*expr1* && !*expr2* && ...)

```
>> Equivalent[True, True, False]
False
```

If all expressions do not evaluate to True or False, Equivalent returns a result in symbolic form:

```
>> Equivalent[a, b, c]
abc
```

Otherwise, `Equivalent` returns a result in DNF

```
>> Equivalent[a, b, True, c]
a&&b&&c
```

False

```
False
  represents the Boolean false value.
```

Implies (=>)

```
Implies[expr1, expr2]
expr1 => expr2
  evaluates each expression in turn, returning True as soon as the first expression evaluates to False. If the first expression evaluates to True, Implies returns the second expression.
```

```
>> Implies[False, a]
True

>> Implies[True, a]
a
```

If an expression does not evaluate to True or False, `Implies` returns a result in symbolic form:

```
>> Implies[a, Implies[b, Implies[
  True, c]]]
abc
```

NoneTrue

```
NoneTrue[{expr1, expr2, ...}, test]
  returns True if no application of test to expr1, expr2, ... evaluates to True.
NoneTrue[list, test, level]
  returns True if no application of test to items of list at level evaluates to True.
NoneTrue[test]
  gives an operator that may be applied to expressions.
```

```
>> NoneTrue[{1, 3, 5}, EvenQ]
True
```

```
>> NoneTrue[{1, 4, 5}, EvenQ]
False
```

Not (!)

```
Not[expr]
!expr
  negates the logical expression expr.
```

```
>> !True
False

>> !False
True

>> !b
!b
```

Or (||)

```
Or[expr1, expr2, ...]
expr1 || expr2 || ...
  evaluates each expression in turn, returning True as soon as an expression evaluates to True. If all expressions evaluate to False, Or returns False.
```

```
>> False || True
True
```

If an expression does not evaluate to True or False, `Or` returns a result in symbolic form:

```
>> a || False || b
a||b
```

True

```
True
  represents the Boolean true value.
```

Xor (xor)

```
Xor[expr1, expr2, ...]  
expr1 xor expr2 xor...  
    evaluates each expression in turn, return-  
    ing True as soon as not all expressions  
    evaluate to the same value. If all expres-  
    sions evaluate to the same value, Xor re-  
    turns False.
```

```
>> Xor[False, True]  
True
```

```
>> Xor[True, True]  
False
```

If an expression does not evaluate to True or False, Xor returns a result in symbolic form:

```
>> Xor[a, False, b]  
ab
```

XXIV. Manipulate

Contents

Manipulate	173	System'Private'ManipulateParameter	173
----------------------	-----	------------------------------------	-----

Manipulate

`Manipulate[expr1, {u, u_min, u_max}]`
interactively compute and display an expression with different values of u .

`Manipulate[expr1, {u, u_min, u_max, du}]`
allows u to vary between u_{min} and u_{max} in steps of du .

`Manipulate[expr1, {{u, u_init}, u_min, u_max, ...}]`
starts with initial value of u_{init} .

`Manipulate[expr1, {{u, u_init, u_lbl}, ...}]`
labels the u control by u_{lbl} .

`Manipulate[expr1, {u, {u_1, u_2, ...}}]`
sets u to take discrete values u_1, u_2, \dots .

`Manipulate[expr1, {u, ...}, {v, ...}, ...]`
control each of u, v, \dots .

System'Private'ManipulateParameter

XXV. Quantities

Contents

KnownUnitQ	174	QuantityMagnitude . .	174	QuantityUnit	175
Quantity	174	QuantityQ	174	UnitConvert	175

KnownUnitQ

KnownUnitQ[*unit*]
returns True if *unit* is a canonical unit,
and False otherwise.

```
>> KnownUnitQ["Feet"]
True

>> KnownUnitQ["Foo"]
False
```

Quantity

Quantity[*magnitude*, *unit*]
represents a quantity with size *magnitude*
and unit specified by *unit*.
Quantity[*unit*]
assumes the magnitude of the specified
unit to be 1.

```
>> Quantity["Kilogram"]
1kilogram

>> Quantity[10, "Meters"]
10meter

>> Quantity[{10,20}, "Meters"]
{10meter,20meter}
```

QuantityMagnitude

QuantityMagnitude[*quantity*]
gives the amount of the specified *quantity*.
QuantityMagnitude[*quantity*, *unit*]
gives the value corresponding to *quantity*
when converted to *unit*.

```
>> QuantityMagnitude[Quantity["Kilogram"]]
1

>> QuantityMagnitude[Quantity[10, "Meters"]]
10

>> QuantityMagnitude[Quantity[{10,20}, "Meters"]]
{10,20}
```

QuantityQ

QuantityQ[*expr*]
return True if *expr* is a valid Association
object, and False otherwise.

```
>> QuantityQ[Quantity[3, "Meters"]]
True

>> QuantityQ[Quantity[3, "Maters"]]
UnabletointerpretunitspecificationMaters.
False
```

QuantityUnit

`QuantityUnit[quantity]`
returns the unit associated with the specified *quantity*.

```
>> QuantityUnit[Quantity["Kilogram
"]]
kilogram

>> QuantityUnit[Quantity[10, "
Meters"]]
meter

>> QuantityUnit[Quantity[{10,20}, "
Meters"]]
{meter,meter}
```

UnitConvert

`UnitConvert[quantity, targetunit]`
converts the specified *quantity* to the specified *targetunit*.
`UnitConvert[quantity]`
converts the specified *quantity* to its "SI Base" units.

Convert from miles to kilometers:

```
>> UnitConvert[Quantity[5.2, "miles
"], "kilometers"]
8.36859kilometer
```

Convert a Quantity object to the appropriate SI base units:

```
>> UnitConvert[Quantity[3.8, "
Pounds"]]
1.72365kilogram
```

XXVI. Number theoretic functions

Contents

CoprimeQ	176	LCM	177	PrimePi	179
Divisors	176	MantissaExponent . . .	177	PrimePowerQ	179
EvenQ	176	Mod	178	PrimeQ	179
FactorInteger	177	NextPrime	178	Quotient	179
FractionalPart	177	OddQ	178	QuotientRemainder . .	179
GCD	177	PowerMod	178	RandomPrime	180
IntegerExponent	177	Prime	178		

CoprimeQ

`CoprimeQ[x, y]`
tests whether x and y 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
```

Divisors

`Divisors[n]`
returns a list of the integers that divide n .

```
>> Divisors[96]
{1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 96}
>> Divisors[704]
{1, 2, 4, 8, 11, 16, 22, 32,
 44, 64, 88, 176, 352, 704}
>> Divisors[{87, 106, 202, 305}]
{{1, 3, 29, 87}, {1, 2, 53, 106},
 {1, 2, 101, 202}, {1, 5, 61, 305}}
```

EvenQ

`EvenQ[x]`
returns True if x is even, and False otherwise.

```
>> 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}}
```

FractionalPart

`FractionalPart[n]`
finds the fractional part of n .

```
>> FractionalPart[4.1]
0.1
```

```
>> FractionalPart[-5.25]
-0.25
```

GCD

`GCD[n1, n2, ...]`
computes the greatest common divisor of the given integers.

```
>> GCD[20, 30]
10
```

```
>> GCD[10, y]
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]
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
```

MantissaExponent

`MantissaExponent[n]`
finds a list containing the mantissa and exponent of a given number n .
`MantissaExponent[n, b]`
finds the base b mantissa and exponent of n .

```
>> MantissaExponent[2.5*10^20]
{0.25, 21}
```

```
>> MantissaExponent[125.24]
{0.12524, 3}
```

```
>> MantissaExponent[125., 2]
{0.976563, 7}
```

```
>> MantissaExponent[10, b]
MantissaExponent[10, b]
```

Mod

`Mod[x, m]`
returns x modulo m .

```
>> Mod[14, 6]
2
>> Mod[-3, 4]
1
>> Mod[-3, -4]
-3
>> Mod[5, 0]
The argument 0 should be nonzero.
Mod[5, 0]
```

NextPrime

`NextPrime[n]`
gives the next prime after n .
`NextPrime[n, k]`
gives the k th prime after n .

```
>> NextPrime[10000]
10007
>> NextPrime[100, -5]
73
>> NextPrime[10, -5]
-2
>> NextPrime[100, 5]
113
>> NextPrime[5.5, 100]
563
>> NextPrime[5, 10.5]
NextPrime[5, 10.5]
```

OddQ

`OddQ[x]`
returns True if x is odd, and False otherwise.

```
>> OddQ[-3]
True
```

```
>> OddQ[0]
False
```

PowerMod

`PowerMod[x, y, m]`
computes x^y modulo m .

```
>> 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 n th 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

`PrimePowerQ[n]`
returns True if n is a power of a prime number.

```
>> PrimePowerQ[9]
True

>> PrimePowerQ[52142]
False

>> PrimePowerQ[-8]
True

>> PrimePowerQ[371293]
True
```

PrimeQ

`PrimeQ[n]`
returns True if n is a prime number.

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}
```

Quotient

`Quotient[m, n]`
computes the integer quotient of m and n .

```
>> Quotient[23, 7]
3
```

QuotientRemainder

`QuotientRemainder[m, n]`
computes a list of the quotient and remainder from division of m by n .

```
>> QuotientRemainder[23, 7]
{3, 2}
```

RandomPrime

`RandomPrime[{imin, $imax}]`
gives a random prime between $imin$ and $imax$.
`RandomPrime[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}}
```

XXVII. Numeric evaluation

Support for numeric evaluation with arbitrary precision is just a proof-of-concept. Precision is not “guarded” through the evaluation process. Only integer precision is supported. However, things like `N[Pi, 100]` should work as expected.

Contents

<code>Chop</code>	181	<code>\$MaxPrecision</code>	182	<code>RealDigits</code>	185
<code>Hash</code>	181	<code>\$MinPrecision</code>	183	<code>Inter-</code>	
<code>IntegerDigits</code>	182	<code>N</code>	184	<code>nal'RealValuedNumberQ</code>	185
<code>\$MachineEpsilon</code>	182	<code>NumericQ</code>	184	<code>Inter-</code>	
<code>MachinePrecision</code>	182	<code>Precision</code>	184	<code>nal'RealValuedNumericQ</code>	185
<code>\$MachinePrecision</code>	182	<code>Rationalize</code>	184	<code>Round</code>	185

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. × 10-9
>> Chop[10 ^ -11 I]

$$\frac{I}{100\,000\,000\,000}$$

>> Chop[0. + 10 ^ -11 I]
0
```

Hash

`Hash[expr]`
returns an integer hash for the given *expr*.

`Hash[expr, type]`
returns an integer hash of the specified *type* for the given *expr*.
The types supported are “MD5”, “Adler32”, “CRC32”, “SHA”, “SHA224”, “SHA256”, “SHA384”, and “SHA512”.

```
> Hash["The Adventures of Huckleberry Finn"]
= 213425047836523694663619736686226550816
> Hash["The Adventures of Huckleberry Finn",
"SHA256"] = 950926495945903842880571834086092549189343518
> Hash[1/3] = 56073172797010645108327809727054836008
> Hash[{a, b, {c, {d, e, f}}}] = 135682164776235407777080772547528
> Hash[SomeHead[3.1415]] = 5804231647347187731544201546970
>> Hash[{a, b, c}, "xyzstr"]
Hash[{a, b, c}, xyzstr]
```

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}
```

\$MachineEpsilon

`$MachineEpsilon`
is the distance between 1.0 and the next nearest representable machine-precision number.

```
>> $MachineEpsilon
2.22045 × 10-16

>> x = 1.0 + {0.4, 0.5, 0.6}
$MachineEpsilon;

>> x - 1
{0., 0., 2.22045 × 10-16}
```

MachinePrecision

`MachinePrecision`
represents the precision of machine precision numbers.

```
>> N[MachinePrecision]
15.9546

>> N[MachinePrecision, 30]
15.9545897701910033463281614204
```

\$MachinePrecision

`$MachinePrecision`
is the number of decimal digits of precision for machine-precision numbers.

```
>> $MachinePrecision
15.9546
```

\$MaxPrecision

`$MaxPrecision`
represents the maximum number of digits of precision permitted in arbitrary-precision numbers.

```
>> $MaxPrecision
∞

>> $MaxPrecision = 10;

>> N[Pi, 11]
Requested precision 11 is larger than $MaxPrecision. Using current
= Infinity specifies that any precision should be allowed.
3.141592654
```

\$MinPrecision

`$MinPrecision`
represents the minimum number of digits of precision permitted in arbitrary-precision numbers.

```
>> $MinPrecision
0
```

```
>> $MinPrecision = 10;
```

```
>> N[Pi, 9]
```

Requested precision 9 is smaller than \$MinPrecision. Using current \$MinPrecision of 10 instead.

```
3.141592654
```

N

`N[expr, prec]`
evaluates *expr* numerically with a precision of *prec* digits.

```
>> N[Pi, 50]
```

```
3.141592653589793238462643~  
~3832795028841971693993751
```

```
>> N[1/7]
```

```
0.142857
```

```
>> 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]
```

```
11.000000000000000000 + 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]
```

```
>> N[c, 11]
```

```
11.000000000
```

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]] // InputForm
```

```
8.283185307179586
```

The precision of the result is no higher than the precision of the input

```
>> N[Exp[0.1], 100]
```

```
1.10517
```

```
>> % // Precision
```

```
MachinePrecision
```

```
>> N[Exp[1/10], 100]
1.105170918075647624811707~
~826490246668224547194737~
~518718792863289440967966~
~747654302989143318970748654

>> % // Precision
100.

>> N[Exp[1.0^20], 100]
2.7182818284590452354

>> % // Precision
20.
```

NumericQ

`NumericQ[expr]`
tests whether *expr* represents a numeric quantity.

```
>> NumericQ[2]
True

>> NumericQ[Sqrt[Pi]]
True

>> NumericQ[Sqrt[Pi]]
False
```

Precision

`Precision[expr]`
examines the number of significant digits of *expr*.

This is rather a proof-of-concept than a full im-

plementation. Precision of compound expression is not supported yet.

```
>> Precision[1]
∞

>> Precision[1/2]
∞

>> Precision[0.5]
MachinePrecision
```

Rationalize

`Rationalize[x]`
converts a real number *x* to a nearby rational number.
`Rationalize[x, dx]`
finds the rational number within *dx* of *x* with the smallest denominator.

```
>> Rationalize[2.2]
 $\frac{11}{5}$ 
```

Not all numbers can be well approximated.

```
>> Rationalize[N[Pi]]
3.14159
```

Find the exact rational representation of `N[Pi]`

```
>> Rationalize[N[Pi], 0]
 $\frac{245850922}{78256779}$ 
```

RealDigits

`RealDigits[n]`
returns the decimal representation of the real number *n* as list of digits, together with the number of digits that are to the left of the decimal point.
`RealDigits[n, b]`
returns a list of base-*b* representation of the real number *n*.
`RealDigits[n, b, len]`
returns a list of *len* digits.
`RealDigits[n, b, len, p]`
return *len* digits starting with the coefficient of b^p

Return the list of digits and exponent:

```
>> RealDigits[123.55555]
{{1, 2, 3, 5, 5, 5, 5, 5}, {3}}

>> RealDigits[0.000012355555]
{{1, 2, 3, 5, 5, 5, 5, 5}, {-4}}

>> RealDigits[-123.55555]
{{1, 2, 3, 5, 5, 5, 5, 5}, {3}}
```


Return 25 digits of in base 10:

```
>> RealDigits[Pi, 10, 25]
{{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9,
  7, 9, 3, 2, 3, 8, 4, 6, 2, 6, 4, 3}, 1}
```

Return an explicit recurring decimal form:

```
>> RealDigits[19 / 7]
{{2, {7, 1, 4, 2, 8, 5}}, 1}
```

20 digits starting with the coefficient of 10^{-5} :

```
>> RealDigits[Pi, 10, 20, -5]
{{9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3,
  2, 3, 8, 4, 6, 2, 6, 4, 3}, -4}
```

The 10000th digit of is an 8:

```
>> RealDigits[Pi, 10, 1, -10000]
{{8}, -9999}
```

RealDigits gives Indeterminate if more digits than the precision are requested:

```
>> RealDigits[123.45, 10, 18]
{{1, 2, 3, 4, 5, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, Indeterminate,
  Indeterminate}, 3}
```

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]
```

Internal'RealValuedNumberQ

Internal'RealValuedNumericQ

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
```

XXVIII. Options and default arguments

Contents

Default	186	NotOptionQ	186	OptionValue	187
FilterRules	186	OptionQ	187	Options	188

Default

`Default[f]`
gives the default value for an omitted parameter of *f*.
`Default[f, k]`
gives the default value for a parameter on the *k*th position.
`Default[f, k, n]`
gives the default value for the *k*th 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]
{HoldPattern[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}}
```

FilterRules

`FilterRules[rules, pattern]`
gives those *rules* that have a left side that matches *pattern*.
`FilterRules[rules, {pattern1, pattern2, ...}]`
gives those *rules* that have a left side that match at least one of *pattern1*, *pattern2*, ...

```
>> FilterRules[{x -> 100, y -> 1000}, x]
{x -> 100}
>> FilterRules[{x -> 100, y -> 1000, z -> 10000}, {a, b, x, z}]
{x -> 100, z -> 10000}
```

NotOptionQ

`NotOptionQ[expr]`
returns `True` if *expr* does not have the form of a valid option specification.

```
>> NotOptionQ[x]
True
>> NotOptionQ[2]
True
>> NotOptionQ["abc"]
True
>> NotOptionQ[a -> True]
False
```

OptionQ

`OptionQ[expr]`
returns True if *expr* has the form of a valid option specification.

Examples of option specifications:

```
>> OptionQ[a -> True]
True
>> OptionQ[a :> True]
True
>> OptionQ[{a -> True}]
True
>> OptionQ[{a :> True}]
True
```

OptionQ returns False if its argument is not a valid option specification:

```
>> 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]}
Optionnamebnotfound.
{OptionValue[b]}
```

The argument of `OptionValue` must be a symbol:

```
>> f[a->3] /. f[OptionsPattern[{}]]
-> {OptionValue[a+b]}
Argumenta
+ batposition1isexpectedtobeasymbol.
{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]
x2
>> f[x, n -> 3]
x3
```

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}isnotavalidlistofoptionrules.
{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}  
Argumenta  
+ bat position 1 is expected to be a symbol.  
{a -> b}
```

XXIX. Patterns and rules

Some examples:

```
>> a + b + c /. a + b -> t
c + t

>> a + 2 + b + c + x * y /.
n_Integer + s__Symbol + rest_ ->
{n, s, rest}
{2, a, b + c + xy}

>> f[a, b, c, d] /. f[first_,
rest___] -> {first, {rest}}
{a, {b, c, d}}
```

Tests and Conditions:

```
>> f[4] /. f[x_?(# > 0&)] -> x ^ 2
16

>> f[4] /. f[x_] /; x > 0 -> x ^ 2
16
```

Leaves in the beginning of a pattern rather

match fewer leaves:

```
>> f[a, b, c, d] /. f[start__,
end__] -> {{start}, {end}}
{{a}, {b, c, d}}
```

Optional arguments using Optional:

```
>> f[a] /. f[x_, y_:3] -> {x, y}
{a, 3}
```

Options using OptionsPattern and OptionValue:

```
>> f[y, a->3] /. f[x_,
OptionsPattern[{a->2, b->5}]] ->
{x, OptionValue[a], OptionValue
[b]}
{y, 3, 5}
```

The attributes Flat, Orderless, and OneIdentity affect pattern matching.

Contents

Alternatives ()	189	MatchQ	191	ReplaceAll (/.)	194
Blank	190	Optional (:)	192	ReplaceList	194
BlankNullSequence	190	OptionsPattern	192	ReplaceRepeated (//.)	195
BlankSequence	190	PatternTest (?)	192	RuleDelayed (:>)	195
Condition (/;)	191	Pattern	193	Rule (->)	195
Except	191	Repeated (..)	193	Shortest	195
HoldPattern	191	RepeatedNull (...)	193	Verbatim	195
Longest	191	Replace	194		

Alternatives (|)

Alternatives[*p*₁, *p*₂, ..., *p*_{*i*}]
*p*₁ | *p*₂ | ... | *p*_{*i*}
 is a pattern that matches any of the patterns '*p*₁, *p*₂, ..., *p*_{*i*}'.

```
>> a+b+c+d/.(a|b)->t
c + d + 2t
```

Alternatives can also be used for string expressions

```
>> StringReplace["0123 3210", "1" |
"2" -> "X"]
0XX3 3XX0
```

Blank

```
Blank[]
-
  represents any single expression in a pattern.
Blank[h]
_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

```
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

```
BlankSequence[]
--
  represents any non-empty sequence of
  expression leaves in a pattern.
BlankSequence[h]
__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[pattern, expr]
pattern /; expr
  places an additional constraint on pattern
  that only allows it to match if expr evaluates to True.
```

The controlling expression of a Condition can use variables from 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]
```

Except

`Except[c]`
represents a pattern object that matches any expression except those matching *c*.

`Except[c, p]`
represents a pattern object that matches *p* but not *c*.

```
>> Cases[{x, a, b, x, c}, Except[x]]
{a, b, c}

>> Cases[{a, 0, b, 1, c, 2, 3}, Except[1, _Integer]]
{0, 2, 3}
```

Except can also be used for string expressions:

```
>> StringReplace["Hello world!", Except[LetterCharacter] -> ""]
Helloworld
```

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}
```

Longest

```
>> StringCases["aabaab", Longest["a" ~~__ ~~"b"]]
{aabaab}

>> StringCases["aabaab", Longest[RegularExpression["a+b"]]]
{aab, aab}
```

MatchQ

`MatchQ[expr, form]`
tests whether *expr* matches *form*.

```
>> MatchQ[123, _Integer]
True

>> MatchQ[123, _Real]
False

>> MatchQ[_Integer][123]
True
```

Optional (:)

`Optional[patt, default]`
patt : *default*
is a pattern which matches *patt*, 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 *symp* : *patt* represents a Pattern object. However, there is no disambiguity, since *symp* 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 (?)

`PatternTest[pattern, test]`
`pattern ? test`
constrains `pattern` to match `expr` only if the evaluation of `test[expr]` yields `True`.

```
>> MatchQ[3, _Integer?(#>0&)]
True

>> MatchQ[-3, _Integer?(#>0&)]
False
```

Pattern

`Pattern[symb, patt]`
`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}
f[y]
```


This is equivalent to:

```
>> f[a] /. f[a:_:b] -> {a, b}
      {a, b}
```

FullForm:

```
>> FullForm[a:b:c:d:e]
      Optional[Pattern[a, b],
      Optional[Pattern[c, d], e]]

>> f[] /. f[a:_:b] -> {a, b}
      {b, b}
```

Repeated (..)

`Repeated[pattern]`
matches one or more occurrences of *pattern*.

```
>> 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 (...)

`RepeatedNull[pattern]`
matches zero or more occurrences of *pattern*.

```
>> a___Integer...//FullForm
      RepeatedNull[Pattern[a,
      BlankNullSequence[Integer]]]

>> f[x] /. f[x, 0...] -> t
      t
```

Replace

`Replace[expr, x -> y]`
yields the result of replacing *expr* with *y* if it matches the pattern *x*.
`Replace[expr, x -> y, levelspec]`
replaces only subexpressions at levels specified through *levelspec*.
`Replace[expr, {x -> y, ...}]`
performs replacement with multiple rules, yielding a single result expression.
`Replace[expr, {{a -> b, ...}, {c -> d, ...}, ...}]`
returns a list containing the result of performing each set of replacements.

```
>> Replace[x, {x -> 2}]
      2
```

By default, only the top level is searched for matches

```
>> Replace[1 + x, {x -> 2}]
      1 + x

>> Replace[x, {{x -> 1}, {x -> 2}}]
      {1, 2}
```

Replace stops after the first replacement

```
>> Replace[x, {x -> {}, _List -> y}]
      {}
```

Replace replaces the deepest levels first

```
>> Replace[x[1], {x[1] -> y, 1 ->
      2}, All]
      x[2]
```

By default, heads are not replaced

```
>> Replace[x[x[y]], x -> z, All]
      x[x[y]]
```

Heads can be replaced using the Heads option

```
>> Replace[x[x[y]], x -> z, All,
      Heads -> True]
      z[z[y]]
```

Note that heads are handled at the level of leaves

```
>> Replace[x[x[y]], x -> z, {1},
      Heads -> True]
      z[x[y]]
```

You can use Replace as an operator

```
>> Replace[{x_ -> x + 1}][10]
11
```

ReplaceAll (/.)

`ReplaceAll[expr, x -> y]`
`expr /. x -> y`
 yields the result of replacing all subexpressions of `expr` matching the pattern `x` with `y`.
`expr /. {x -> y, ...}`
 performs replacement with multiple rules, yielding a single result expression.
`expr /. {{a -> b, ...}, {c -> d, ...}, ...}`
 returns a list containing the result of performing each set of replacements.

```
>> 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}}
Elementsof{{a->x,b->y},a->w,b->z}areamixtureoflistsandnonlists.
{{a,b} /. {{a->x,b->y}, a->w,b->z}, {u,v}}
```

`ReplaceAll` also can be used as an operator:

```
>> ReplaceAll[{a -> 1}][{a, b}]
{1,b}
```

`ReplaceAll` replaces the shallowest levels first:

```
>> ReplaceAll[x[1], {x[1] -> y, 1 -> 2}]
y
```

ReplaceList

`ReplaceList[expr, rules]`
 returns a list of all possible results of applying `rules` to `expr`.

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-
negativeintegerorInfinityexpectedatposition3.
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 (//.)

`ReplaceRepeated[expr, x -> y]`
`expr //. x -> y`
 repeatedly applies the rule `x -> y` to `expr` until the result no longer changes.

```
>> 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

Log[a] + Log[f] + (Log[b] + Log[c]) d^e
```

ReplaceAll just performs a single replacement:

```
>> Log[a * (b * c)^ d ^ e * f] /.
logrules

Log[a] + Log[f (bc)^d^e]
```

RuleDelayed (:>)

```
RuleDelayed[x, y]
x :> y
represents a rule replacing x with y, with
y held unevaluated.
```

```
>> Attributes[RuleDelayed]
{HoldRest, Protected, SequenceHold}
```

Rule (->)

```
Rule[x, y]
x -> y
represents a rule replacing x with y.
```

```
>> a+b+c /. c->d
a + b + d

>> {x,x^2,y} /. x->3
{3,9,y}
```

Shortest

Verbatim

```
Verbatim[expr]
prevents pattern constructs in expr from
taking effect, allowing them to match
themselves.
```

Create a pattern matching Blank:

```
>> _ /. Verbatim[_]->t
t

>> x /. Verbatim[_]->t
x
```

Without Verbatim, Blank has its normal effect:

```
>> x /. _->t
t
```

XXX. Plotting

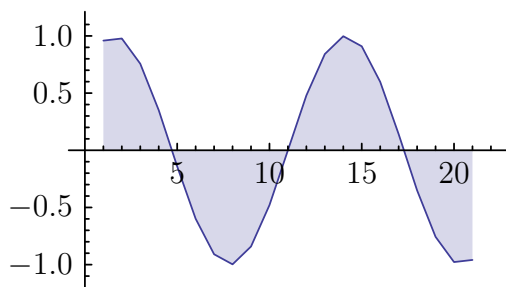
Contents

Axis	196	Full	198	PieChart	202
BarChart	197	Histogram	199	Plot	203
Bottom	197	ListLinePlot	199	Plot3D	204
ColorData	197	ListPlot	199	PolarPlot	204
ColorDataFunction . .	197	Mesh	200	Top	205
DensityPlot	198	ParametricPlot	201		

Axis

Axis
is a possible value for the Filling option.

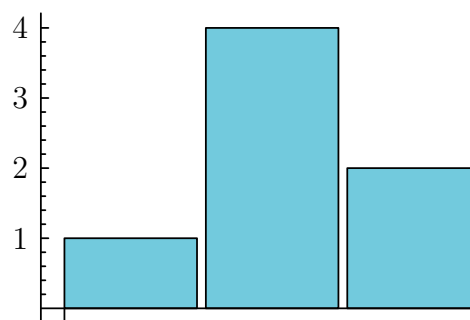
```
>> ListLinePlot[Table[Sin[x], {x, -5, 5, 0.5}], Filling->Axis]
```



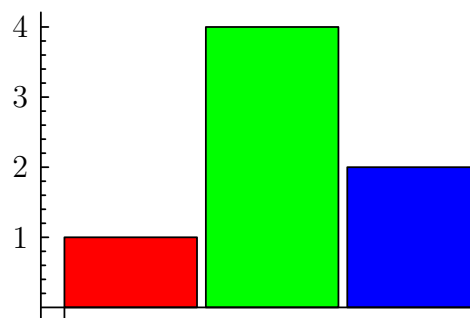
BarChart

BarChart[{p1, p2 ...}]
makes a bar chart with lengths b1, b2,

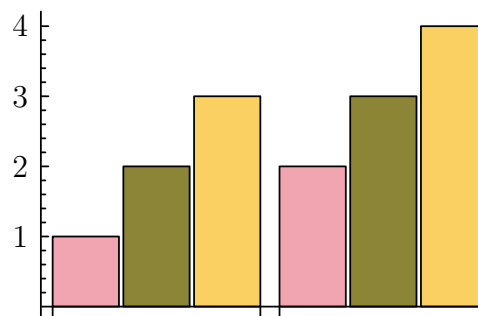
```
>> BarChart[{1, 4, 2}]
```



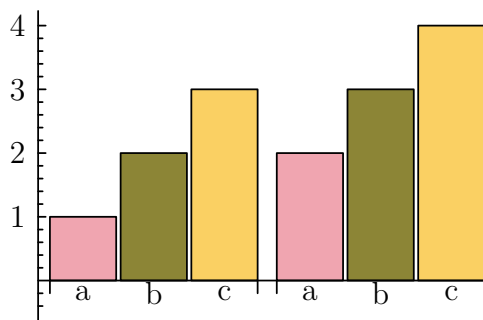
```
>> BarChart[{1, 4, 2}, ChartStyle -> {Red, Green, Blue}]
```



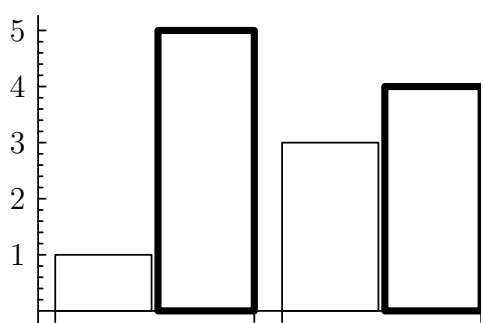
```
>> BarChart[{{1, 2, 3}, {2, 3, 4}}]
```



```
>> BarChart[{{1, 2, 3}, {2, 3, 4}},
  ChartLabels -> {"a", "b", "c"}]
```



```
>> BarChart[{{1, 5}, {3, 4}},
  ChartStyle -> {{EdgeForm[Thin],
  White}, {EdgeForm[Thick], White
  }}]
```

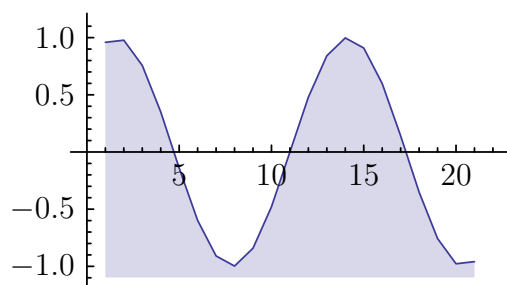


Bottom

Bottom

is a possible value for the Filling option.

```
>> ListLinePlot[Table[Sin[x], {x,
-5, 5, 0.5}], Filling->Bottom]
```



ColorData

ColorData["name"]

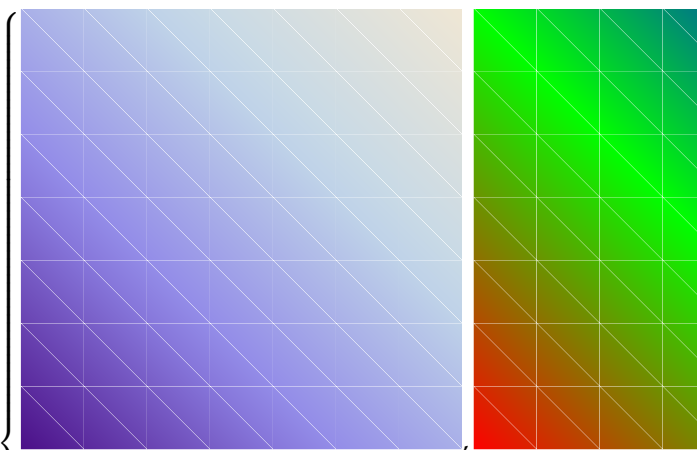
returns a color function with the given name.

Define a user-defined color function:

```
>> Unprotect[ColorData]; ColorData
["test"] := ColorDataFunction["
test", "Gradients", {0, 1},
Blend[{Red, Green, Blue}, #1
&]; Protect[ColorData]
```

Compare it to the default color function, LakeColors:

```
>> {DensityPlot[x + y, {x, -1, 1},
{y, -1, 1}], DensityPlot[x + y,
{x, -1, 1}, {y, -1, 1},
ColorFunction->"test"]}
```



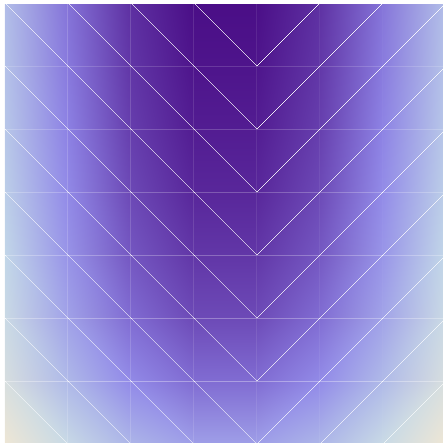
ColorDataFunction

DensityPlot

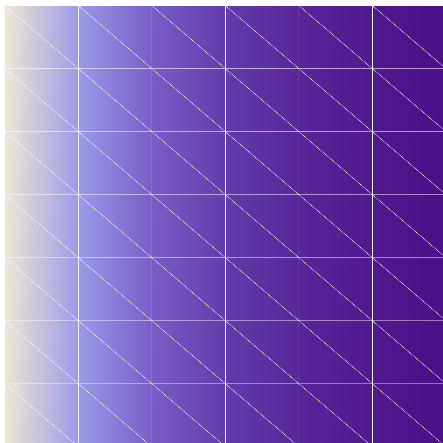
DensityPlot[f, {x, xmin, xmax}, {y, ymin, ymax}]

plots a density plot of f with x ranging from $xmin$ to $xmax$ and y ranging from $ymin$ to $ymax$.

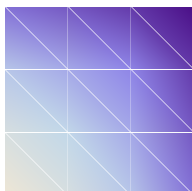
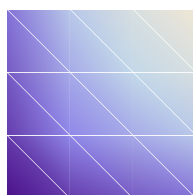
```
>> DensityPlot[x ^ 2 + 1 / y, {x,
-1, 1}, {y, 1, 4}]
```



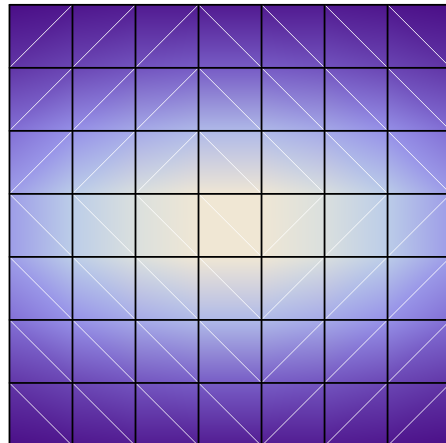
```
>> DensityPlot[1 / x, {x, 0, 1}, {y
, 0, 1}]
```



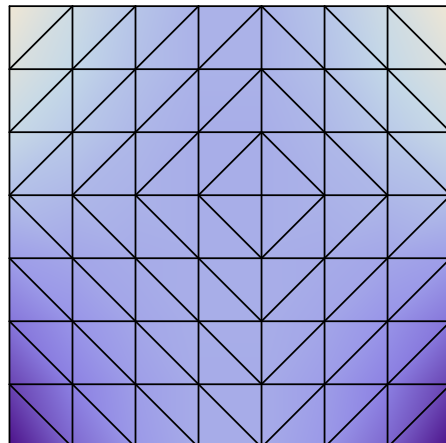
```
>> DensityPlot[Sqrt[x * y], {x, -1,
1}, {y, -1, 1}]
```



```
>> 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

Full

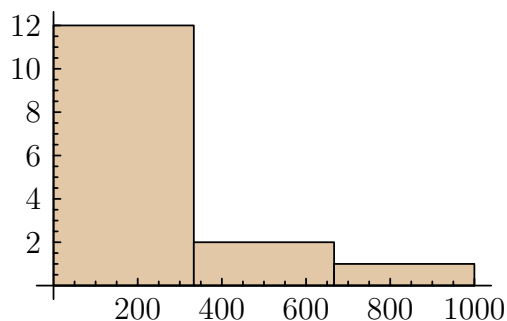
is a possible value for the Mesh and PlotRange options.

Histogram

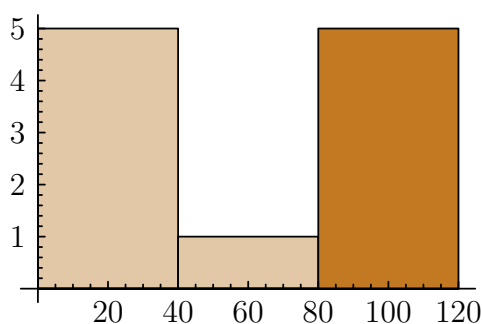
Histogram[{x1, x2 ...}]

plots a histogram using the values x1, x2,

```
>> Histogram[{3, 8, 10, 100, 1000,
500, 300, 200, 10, 20, 200, 100,
200, 300, 500}]
```



```
>> Histogram[{{1, 2, 10, 5, 50,
20}, {90, 100, 101, 120, 80}}]
```



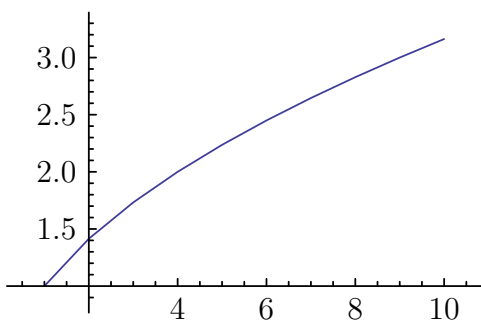
ListLinePlot

`ListLinePlot[{y1, y2, ...}]`
plots a line through a list of y -values, assuming integer x -values 1, 2, 3, ...

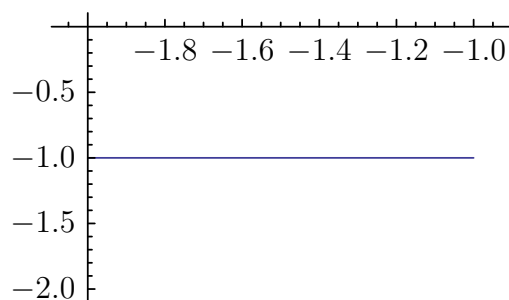
`ListLinePlot[{x1, y1}, {x2, y2}, ...]`
plots a line through a list of x, y pairs.

`ListLinePlot[{list1, list2, ...}]`
plots several lines.

```
>> ListLinePlot[Table[{n, n ^ 0.5},
{n, 10}]]
```



```
>> ListLinePlot[{{-2, -1}, {-1,
-1}}]
```



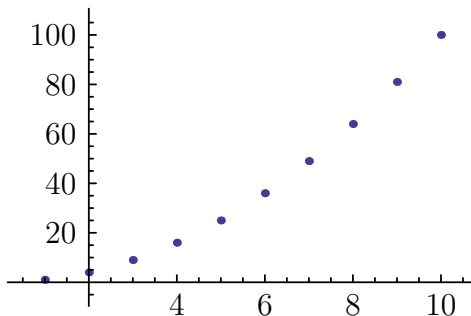
ListPlot

`ListPlot[{y1, y2, ...}]`
plots a list of y -values, assuming integer x -values 1, 2, 3, ...

`ListPlot[{x1, y1}, {x2, y2}, ...]`
plots a list of x, y pairs.

`ListPlot[{list1, list2, ...}]`
plots several lists of points.

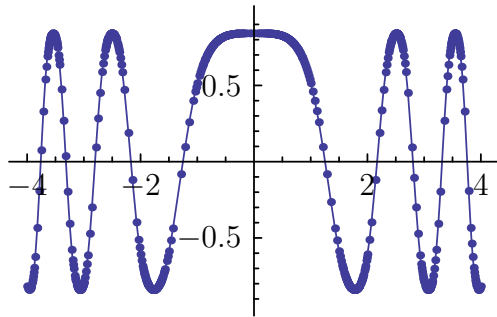
```
>> ListPlot[Table[n ^ 2, {n, 10}]]
```



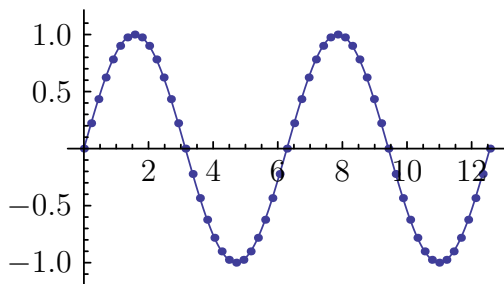
Mesh

`Mesh`
is an option for `Plot` that specifies the mesh to be drawn. The default is `Mesh->None`.

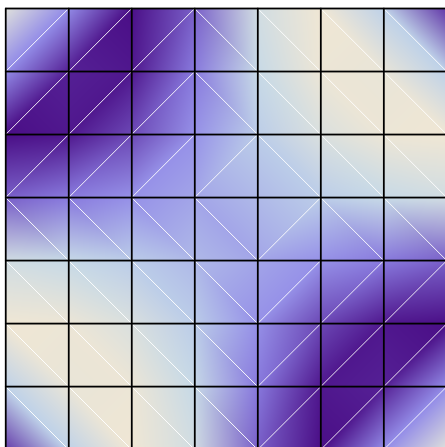
```
>> Plot[Sin[Cos[x^2]],{x,-4,4},Mesh->All]
```



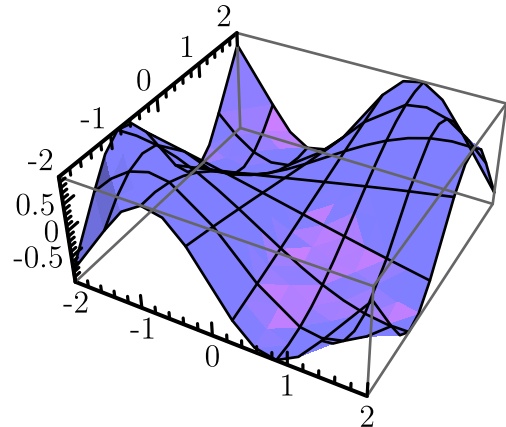
```
>> Plot[Sin[x], {x,0,4 Pi}, Mesh->Full]
```



```
>> DensityPlot[Sin[x y], {x, -2, 2}, {y, -2, 2}, Mesh->Full]
```



```
>> Plot3D[Sin[x y], {x, -2, 2}, {y, -2, 2}, Mesh->Full]
```



ParametricPlot

```
ParametricPlot[{f_x, f_y}, {u, umin, umax}]
```

plots a parametric function f with the parameter u ranging from $umin$ to $umax$.

```
ParametricPlot[{f_x, f_y}, {g_x, g_y}, ..., {u, umin, umax}]
```

plots several parametric functions f, g, \dots

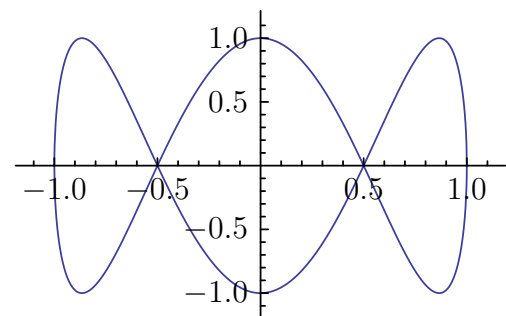
```
ParametricPlot[{f_x, f_y}, {u, umin, umax}, {v, vmin, vmax}]
```

plots a parametric area.

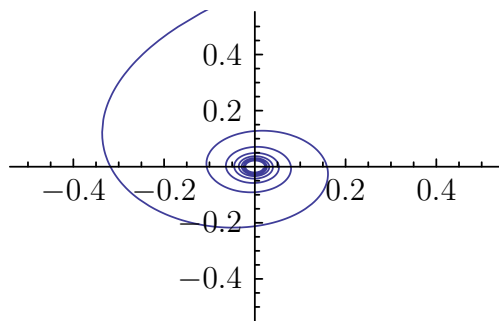
```
ParametricPlot[{f_x, f_y}, {g_x, g_y}, ..., {u, umin, umax}, {v, vmin, vmax}]
```

plots several parametric areas.

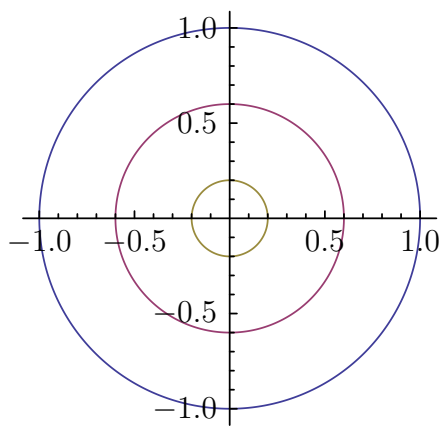
```
>> ParametricPlot[{Sin[u], Cos[3 u]}, {u, 0, 2 Pi}]
```




```
>> ParametricPlot[{Cos[u] / u, Sin[u] / u}, {u, 0, 50}, PlotRange -> 0.5]
```



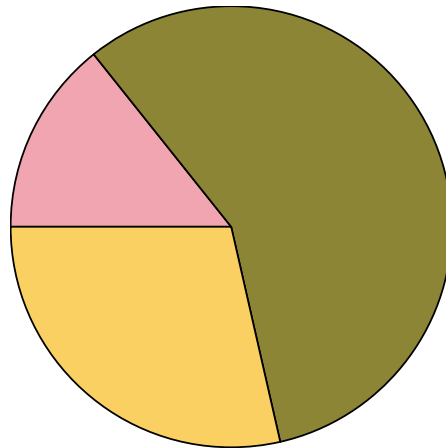
```
>> ParametricPlot[{{Sin[u], Cos[u]}, {0.6 Sin[u], 0.6 Cos[u]}, {0.2 Sin[u], 0.2 Cos[u]}}, {u, 0, 2 Pi}, PlotRange -> 1, AspectRatio -> 1]
```



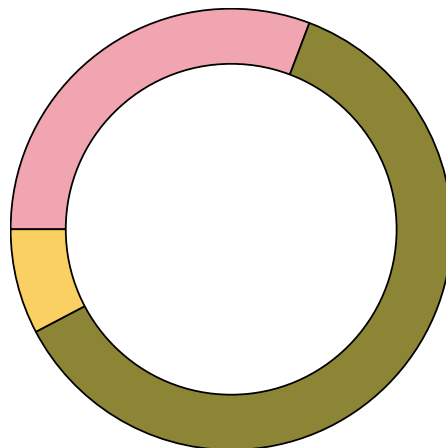
PieChart

`PieChart[{p1, p2 ...}]`
draws a pie chart.

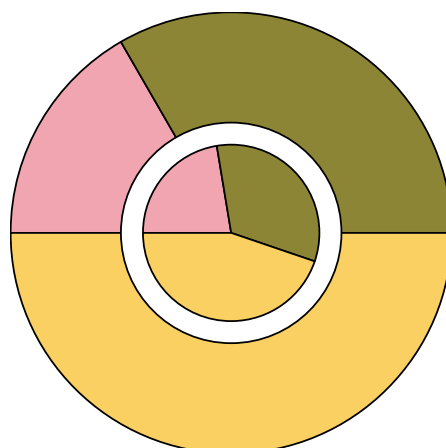
```
>> PieChart[{1, 4, 2}]
```



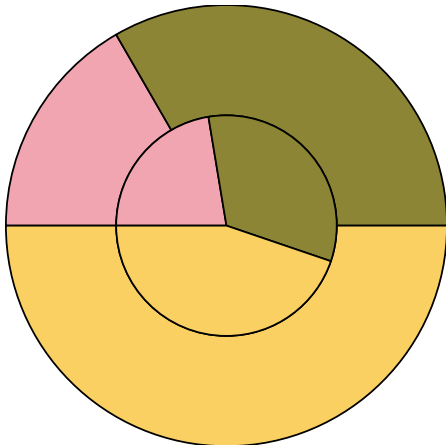
```
>> PieChart[{8, 16, 2}, SectorOrigin -> {Automatic, 1.5}]
```



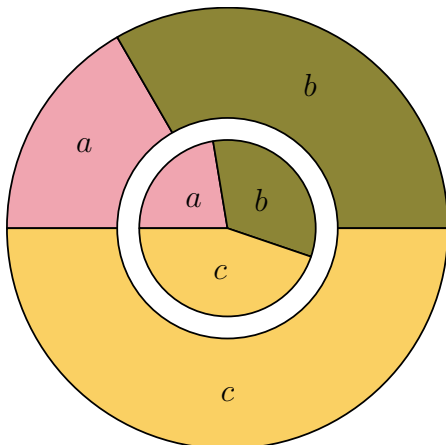
```
>> PieChart[{{10, 20, 30}, {15, 22, 30}}]
```



```
>> PieChart[{{10, 20, 30}, {15, 22, 30}}, SectorSpacing -> None]
```

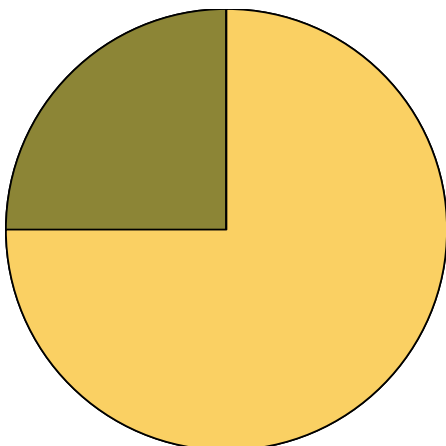


```
>> PieChart[{{10, 20, 30}, {15, 22, 30}}, ChartLabels -> {a, b, c}]
```



Negative values are clipped to 0.

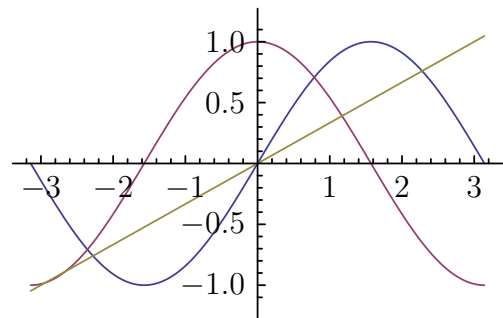
```
>> PieChart[{1, -1, 3}]
```



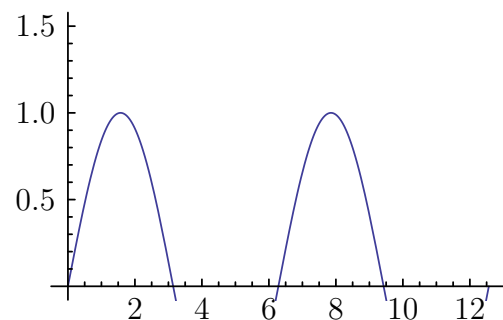
Plot

```
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, ...
```

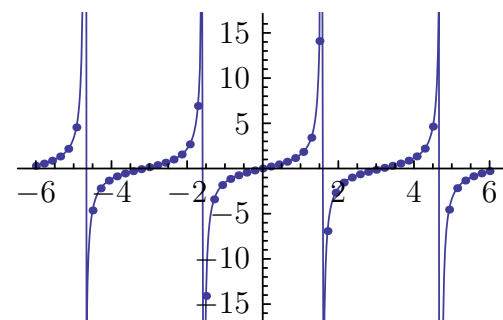
```
>> Plot[{Sin[x], Cos[x], x / 3}, {x, -Pi, Pi}]
```



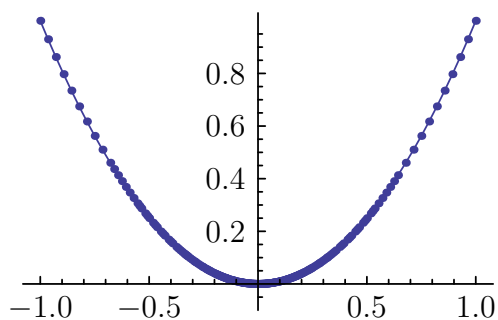
```
>> Plot[Sin[x], {x, 0, 4 Pi},
PlotRange->{{0, 4 Pi}, {0, 1.5}}]
```



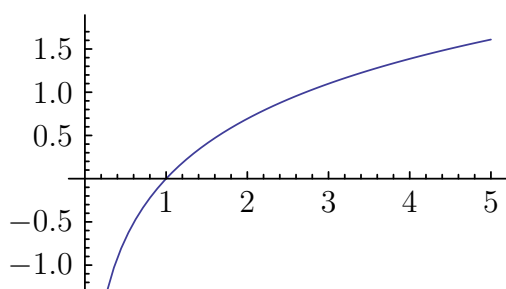
```
>> Plot[Tan[x], {x, -6, 6}, Mesh->Full]
```



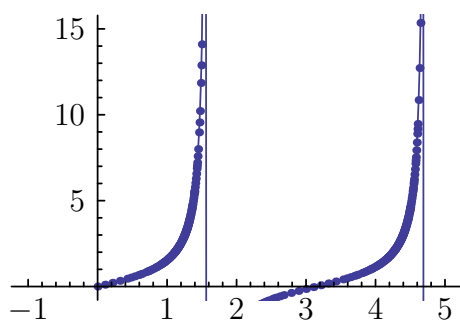
```
>> Plot[x^2, {x, -1, 1},
MaxRecursion->5, Mesh->All]
```



```
>> Plot[Log[x], {x, 0, 5},
MaxRecursion->0]
```

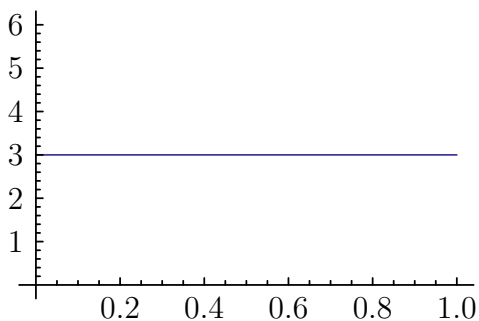


```
>> Plot[Tan[x], {x, 0, 6}, Mesh->
All, PlotRange->{{-1, 5}, {0,
15}}, MaxRecursion->10]
```



A constant function:

```
>> Plot[3, {x, 0, 1}]
```

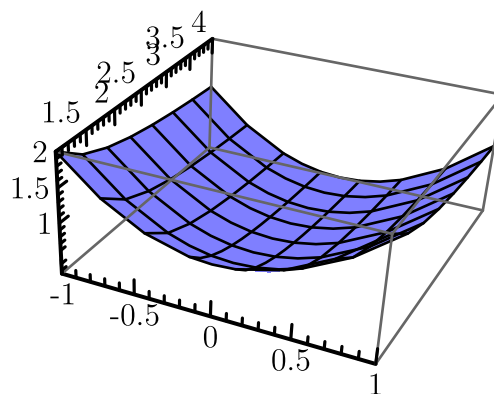


Plot3D

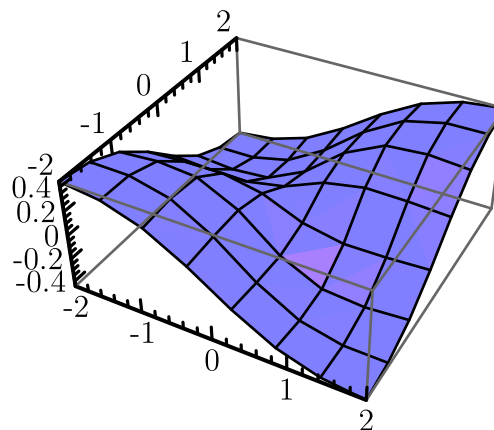
```
Plot3D[f, {x, xmin, xmax}, {y, ymin,
ymax}]
```

creates a three-dimensional plot of f with x ranging from $xmin$ to $xmax$ and y ranging from $ymin$ to $ymax$.

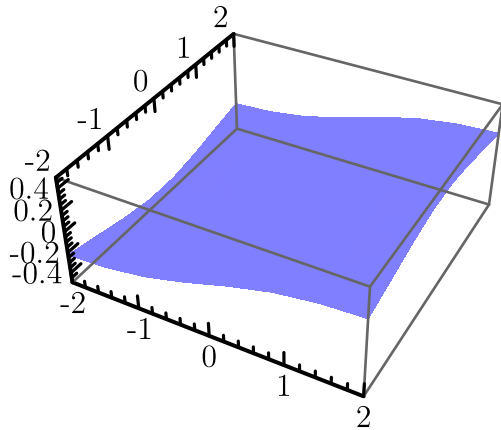
```
>> Plot3D[x ^ 2 + 1 / y, {x, -1,
1}, {y, 1, 4}]
```



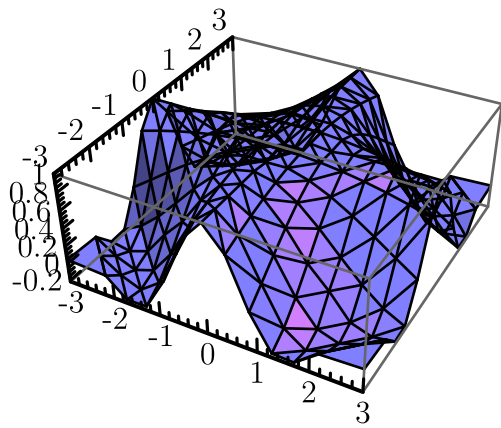
```
>> Plot3D[x y / (x ^ 2 + y ^ 2 + 1)
, {x, -2, 2}, {y, -2, 2}]
```



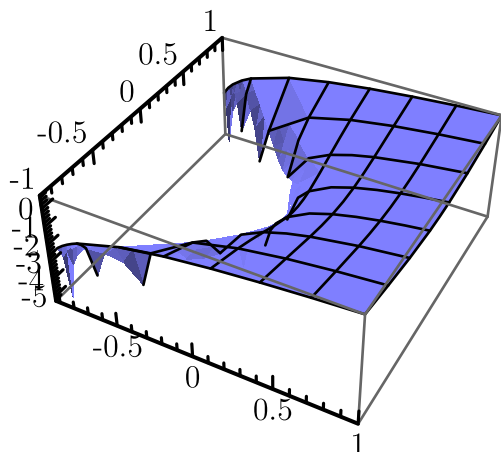
```
>> Plot3D[x / (x ^ 2 + y ^ 2 + 1),
{x, -2, 2}, {y, -2, 2}, Mesh->
None]
```



```
>> Plot3D[Sin[x y] / (x y), {x, -3,
3}, {y, -3, 3}, Mesh->All]
```



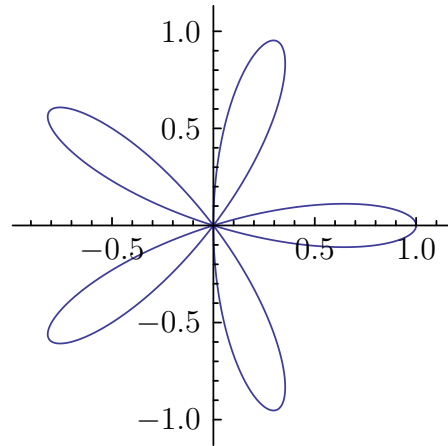
```
>> Plot3D[Log[x + y^2], {x, -1, 1},
{y, -1, 1}]
```



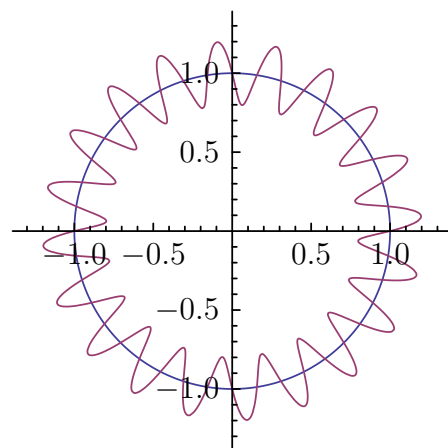
PolarPlot

`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}]
```



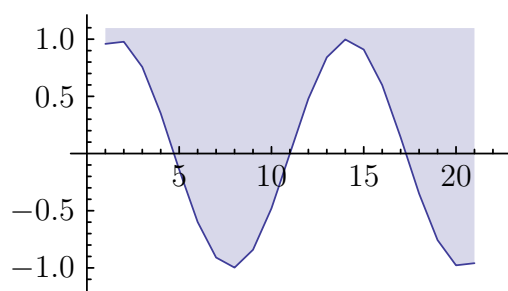
```
>> PolarPlot[{1, 1 + Sin[20 t] /
5}, {t, 0, 2 Pi}]
```



Top

`Top`
is a possible value for the `Filling` option.

```
>> ListLinePlot[Table[Sin[x], {x,  
-5, 5, 0.5}], Filling->Top]
```



XXXI. Physical and Chemical data

Contents

ElementData 207

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]
```

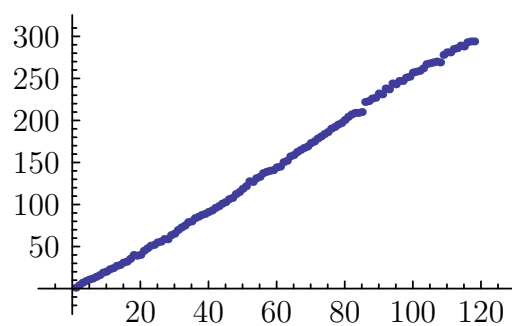
Some data is missing:

```
>> 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}]]
```



XXXII. Random number generation

Random numbers are generated using the Mersenne Twister.

Contents

Random	208	RandomInteger	209	\$RandomState	210
RandomChoice	208	RandomReal	210	SeedRandom	211
RandomComplex	209	RandomSample	210		

Random

Legacy function. Superseded by RandomReal, RandomInteger and RandomComplex.

RandomChoice

RandomChoice[*items*]
randomly picks one item from *items*.
RandomChoice[*items*, *n*]
randomly picks *n* items from *items*. Each pick in the *n* picks happens from the given set of *items*, so each item can be picked any number of times.
RandomChoice[*items*, {*n1*, *n2*, ...}]
randomly picks items from *items* and arranges the picked items in the nested list structure described by {*n1*, *n2*, ...}.
RandomChoice[*weights* -> *items*, *n*]
randomly picks *n* items from *items* and uses the corresponding numeric values in *weights* to determine how probable it is for each item in *items* to get picked (in the long run, items with higher weights will get picked more often than ones with lower weight).
RandomChoice[*weights* -> *items*]
randomly picks one items from *items* using weights *weights*.
RandomChoice[*weights* -> *items*, {*n1*, *n2*, ...}]
randomly picks a structured list of items from *items* using weights *weights*.

```
>> RandomChoice[{a, b, c}]
{c}
>> SeedRandom[42]

>> RandomChoice[{a, b, c}, 20]
{c, a, c, c, a, a, c, b, c, c,
 c, c, a, c, b, a, b, b, b}
>> SeedRandom[42]

>> RandomChoice[{"a", {1, 2}, x, {}}, 10]
{x, {}, a, x, x, {}, a, a, x, {1, 2}}
>> SeedRandom[42]

>> RandomChoice[{a, b, c}, {5, 2}]
{{c, a}, {c, c}, {a, a}, {c, b}, {c, c}}
>> SeedRandom[42]

>> RandomChoice[{1, 100, 5} -> {a, b, c}, 20]
{b, b, b, b, b, b, b, b, b, b,
 b, c, b, b, b, b, b, b, b, b}
```

```
>> SeedRandom[42]
```


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[]
0.219438 + 0.340878I

>> RandomComplex[{1+I, 5+5I}]
3.98953 + 3.43383I

>> RandomComplex[1+I, 5]
{0.978261 + 0.19637I, 0.677~
~543 + 0.83006I, 0.245785 +
0.437443I, 0.200139 + 0.554~
~727I, 0.338317 + 0.438886I}

>> RandomComplex[{1+I, 2+2I}, {2,
2}]
{{1.98229 + 1.95405I, 1.228~
~98 + 1.82254I}, {1.34541 +
1.53246I, 1.0247 + 1.59589I}}
```

RandomInteger

`RandomInteger[{min, max}]`
yields a pseudorandom integer in the range from `min` to `max` inclusive.

`RandomInteger[max]`
yields a pseudorandom integer in the range from 0 to `max` inclusive.

`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
73 34 19
61 70 54
```

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[]
0.165486

>> RandomReal[{1, 5}]
4.82871
```

RandomSample

`RandomSample[items]`
randomly picks one item from *items*.

`RandomSample[items, n]`
randomly picks *n* items from *items*. Each pick in the *n* picks happens after the previous items picked have been removed from *items*, so each item can be picked at most once.

`RandomSample[items, {n1, n2, ...}]`
randomly picks items from *items* and arranges the picked items in the nested list structure described by {*n1*, *n2*, ...}. Each item gets picked at most once.

`RandomSample[weights -> items, n]`
randomly picks *n* items from *items* and uses the corresponding numeric values in *weights* to determine how probable it is for each item in *items* to get picked (in the long run, items with higher weights will get picked more often than ones with lower weight). Each item gets picked at most once.

`RandomSample[weights -> items]`
randomly picks one items from *items* using weights *weights*. Each item gets picked at most once.

`RandomSample[weights -> items, {n1, n2, ...}]`
randomly picks a structured list of items from *items* using weights *weights*. Each item gets picked at most once.

```
>> SeedRandom[42]

>> RandomSample[{a, b, c}]
{a}

>> SeedRandom[42]

>> RandomSample[{a, b, c, d, e, f, g, h}, 7]
{b, f, a, h, c, e, d}

>> SeedRandom[42]
```

```
>> RandomSample[{"a", {1, 2}, x, {}], 3]
{{1, 2}, {}, a}

>> SeedRandom[42]

>> RandomSample[Range[100], {2, 3}]
{{84, 54, 71}, {46, 45, 40}}

>> SeedRandom[42]

>> RandomSample[Range[100] -> Range[100], 5]
{62, 98, 86, 78, 40}
```

\$RandomState

`$RandomState`
is a long number representing the internal state of the pseudorandom number generator.

```
>> Mod[$RandomState, 10^100]
558 955 023 909 916 451 941 712 ~
~673 595 824 029 350 105 428 208 ~
~610 238 588 943 308 712 498 268 ~
~629 570 823 187 040 120 626 810 926

>> IntegerLength[$RandomState]
6466
```

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 the seed.

SeedRandom can be used to get reproducible random numbers:

```
>> SeedRandom[42]

>> RandomInteger[100]
51

>> RandomInteger[100]
92

>> SeedRandom[42]

>> RandomInteger[100]
51

>> RandomInteger[100]
92
```

String seeds are supported as well:

```
>> SeedRandom["Mathics"]

>> RandomInteger[100]
27
```

Calling SeedRandom without arguments will seed the random number generator to a random state:

```
>> SeedRandom[]

>> RandomInteger[100]
18
```

XXXIII. Recurrence relation solvers

Contents

RSolve 212

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] -> C[0]}}
```

No boundary conditions gives two general parameters:

```
>> RSolve[{a[n + 2] == a[n]}, a, n]
{{a -> (Function[{n},
  C[0] + C[1] - 1^n])}}
```

One boundary condition:

```
>> RSolve[{a[n + 2] == a[n], a[0] == 1}, a, n]
{{a -> (Function[{n},
  C[0] + (1 - C[0]) - 1^n])}}
```

Two boundary conditions:

```
>> RSolve[{a[n + 2] == a[n], a[0] == 1, a[1] == 4}, a, n]
{{a -> (Function[{n},
  5/2 - (3 - 1^n)/2])}}
```

XXXIV. Special functions

Contents

AiryAi	213	ChebyshevU	216	KelvinBei	218
AiryAiPrime	214	Erf	216	KelvinBer	218
AiryAiZero	214	Erfc	216	KelvinKei	219
AiryBi	214	ExpIntegralE	216	KelvinKer	219
AiryBiPrime	214	ExpIntegralEi	217	LaguerreL	219
AiryBiZero	214	FresnelC	217	LegendreP	220
AngerJ	214	FresnelS	217	LegendreQ	220
BesselI	215	GegenbauerC	217	ProductLog	220
BesselJ	215	HankelH1	217	SphericalHarmonicY	220
BesselJZero	215	HankelH2	217	StruveH	220
BesselK	215	HermiteH	217	StruveL	221
BesselY	215	InverseErf	218	WeberE	221
BesselYZero	215	InverseErfc	218	Zeta	221
ChebyshevT	216	JacobiP	218		

AiryAi

AiryAi[x]
returns the Airy function Ai(x).

Exact values:

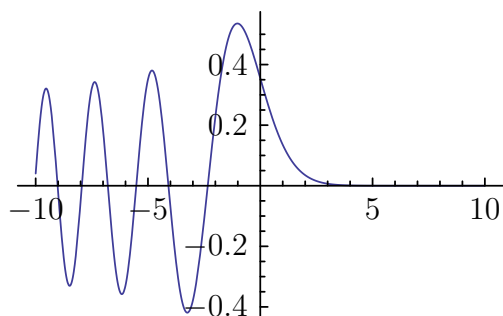
```
>> AiryAi[0]
      31/3
-----
3Gamma[2/3]
```

AiryAi can be evaluated numerically:

```
>> AiryAi[0.5]
0.231694

>> AiryAi[0.5 + I]
0.157118 - 0.24104I
```

```
>> Plot[AiryAi[x], {x, -10, 10}]
```



AiryAiPrime

AiryAiPrime[x]
returns the derivative of the Airy function AiryAi[x].

Exact values:

```
>> AiryAiPrime[0]
      32/3
-----
3Gamma[1/3]
```

Numeric evaluation:

```
>> AiryAiPrime[0.5]
-0.224911
```

AiryAiZero

`AiryAiZero[k]`
returns the k th zero of the Airy function $\text{Ai}(z)$.

```
>> N[AiryAiZero[1]]
-2.33811
```

AiryBi

`AiryBi[x]`
returns the Airy function of the second kind $\text{Bi}(x)$.

Exact values:

```
>> AiryBi[0]

$$\frac{3^{\frac{5}{6}}}{3\Gamma\left[\frac{2}{3}\right]}$$

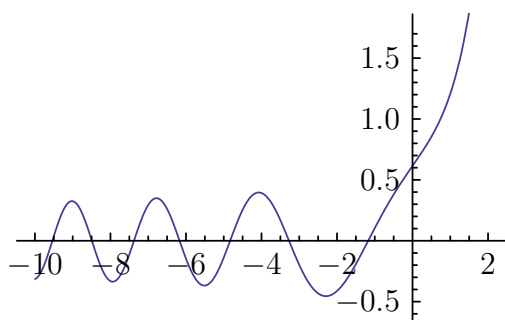
```

Numeric evaluation:

```
>> AiryBi[0.5]
0.854277

>> AiryBi[0.5 + I]
0.688145 + 0.370815I

>> Plot[AiryBi[x], {x, -10, 2}]
```



AiryBiPrime

`AiryBiPrime[x]`
returns the derivative of the Airy function of the second kind $\text{Bi}(x)$.

Exact values:

```
>> AiryBiPrime[0]

$$\frac{3^{\frac{1}{6}}}{\Gamma\left[\frac{1}{3}\right]}$$

```

Numeric evaluation:

```
>> AiryBiPrime[0.5]
0.544573
```

AiryBiZero

`AiryBiZero[k]`
returns the k th zero of the Airy function $\text{Bi}(z)$.

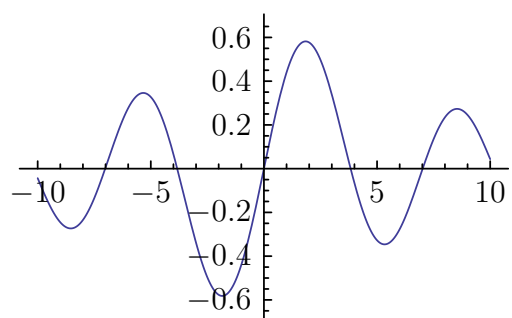
```
>> N[AiryBiZero[1]]
-1.17371
```

AngerJ

`AngerJ[n, z]`
returns the Anger function $J_n(z)$.

```
>> AngerJ[1.5, 3.5]
0.294479

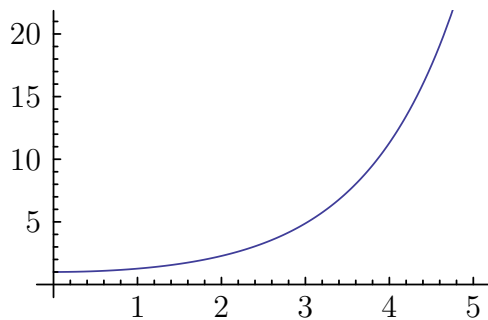
>> Plot[AngerJ[1, x], {x, -10, 10}]
```



BesselI

BesselI[n , z]
returns the modified Bessel function of the first kind $I_n(z)$.

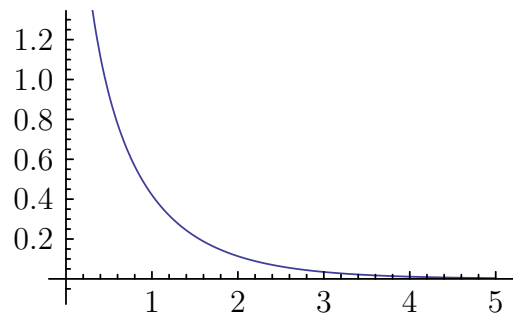
```
>> BesselI[1.5, 4]  
8.17263  
  
>> Plot[BesselI[0, x], {x, 0, 5}]
```



BesselK

BesselK[n , z]
returns the modified Bessel function of the second kind $K_n(z)$.

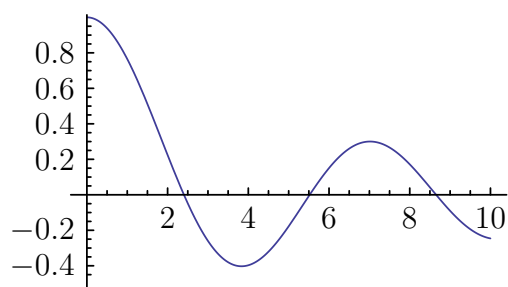
```
>> BesselK[1.5, 4]  
0.014347  
  
>> Plot[BesselK[0, x], {x, 0, 5}]
```



BesselJ

BesselJ[n , z]
returns the Bessel function of the first kind $J_n(z)$.

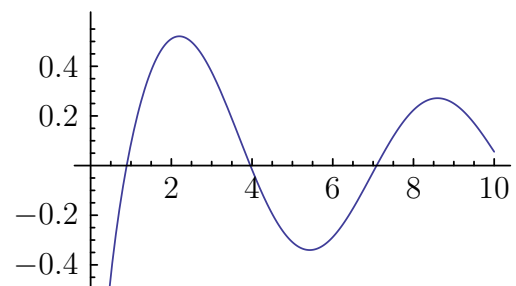
```
>> BesselJ[0, 5.2]  
-0.11029  
  
>> Plot[BesselJ[0, x], {x, 0, 10}]
```



BesselY

BesselY[n , z]
returns the Bessel function of the second kind $Y_n(z)$.

```
>> BesselY[1.5, 4]  
0.367112  
  
>> Plot[BesselY[0, x], {x, 0, 10}]
```



BesselJZero

BesselJZero[n , k]
returns the k th zero of the Bessel function of the first kind $J_n(z)$.

```
>> N[BesselJZero[0, 1]]  
2.40483
```

BesselYZero

BesselYZero[n , k]
returns the k th zero of the Bessel function of the second kind $Y_n(z)$.

```
>> N[BesselYZero[0, 1]]  
0.893577
```

ChebyshevT

ChebyshevT $[n, x]$
returns the Chebyshev polynomial of the first kind $T_n(x)$.

```
>> ChebyshevT[8, x]
1 - 32x2 + 160x4 - 256x6 + 128x8

>> ChebyshevT[1 - I, 0.5]
0.800143 + 1.08198I
```

ChebyshevU

ChebyshevU $[n, x]$
returns the Chebyshev polynomial of the second kind $U_n(x)$.

```
>> ChebyshevU[8, x]
1 - 40x2 + 240x4 - 448x6 + 256x8

>> ChebyshevU[1 - I, 0.5]
1.60029 + 0.721322I
```

Erf

Erf $[z]$
returns the error function of z .
Erf $[z0, z1]$
returns the result of $\text{Erf}[z1] - \text{Erf}[z0]$.

$\text{Erf}[x]$ is an odd function:

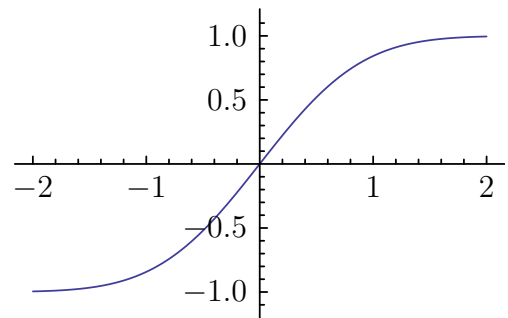
```
>> Erf[-x]
-Erf[x]

>> Erf[1.0]
0.842701

>> Erf[0]
0

>> {Erf[0, x], Erf[x, 0]}
{Erf[x], -Erf[x]}
```

```
>> Plot[Erf[x], {x, -2, 2}]
```



Erfc

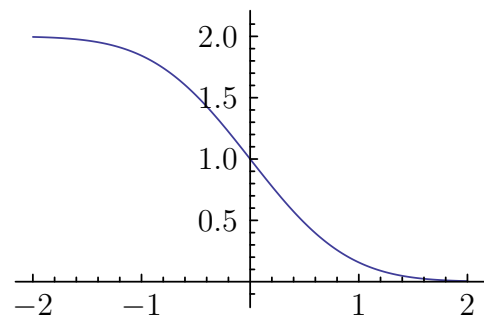
Erfc $[z]$
returns the complementary error function of z .

```
>> Erfc[-x] / 2
(2 - Erfc[x]) / 2

>> Erfc[1.0]
0.157299

>> Erfc[0]
1

>> Plot[Erfc[x], {x, -2, 2}]
```



ExpIntegralE

ExpIntegralE $[n, z]$
returns the exponential integral function $E_n(z)$.

```
>> ExpIntegralE[2.0, 2.0]
0.0375343
```


ExpIntegralEi

`ExpIntegralEi[z]`
returns the exponential integral function $\text{Ei}(z)$.

```
>> ExpIntegralEi[2.0]
4.95423
```

FresnelC

`FresnelC[z]`
is the Fresnel C integral $C(z)$.

```
>> FresnelC[{0, Infinity}]
{0, 1/2}

>> Integrate[Cos[x^2 Pi/2], {x, 0, z}]
FresnelC[z] Gamma[1/4]
-----
4 Gamma[5/4]
```

FresnelS

`FresnelS[z]`
is the Fresnel S integral $S(z)$.

```
>> FresnelS[{0, Infinity}]
{0, 1/2}

>> Integrate[Sin[x^2 Pi/2], {x, 0, z}]
3 FresnelS[z] Gamma[3/4]
-----
4 Gamma[7/4]
```

GegenbauerC

`GegenbauerC[n, m, x]`
returns the Gegenbauer polynomial $C_n^{(m)}(x)$.

```
>> GegenbauerC[6, 1, x]
-1 + 24x^2 - 80x^4 + 64x^6

>> GegenbauerC[4 - I, 1 + 2 I, 0.7]
-3.2621 - 24.9739I
```

HankelH1

`HankelH1[n, z]`
returns the Hankel function of the first kind $H_n^{(1)}(z)$.

```
>> HankelH1[1.5, 4]
0.185286 + 0.367112I
```

HankelH2

`HankelH2[n, z]`
returns the Hankel function of the second kind $H_n^{(2)}(z)$.

```
>> HankelH2[1.5, 4]
0.185286 - 0.367112I
```

HermiteH

`HermiteH[n, x]`
returns the Hermite polynomial $H_n(x)$.

```
>> HermiteH[8, x]
1680 - 13440x^2 + 13~
~440x^4 - 3584x^6 + 256x^8

>> HermiteH[3, 1 + I]
-28 + 4I

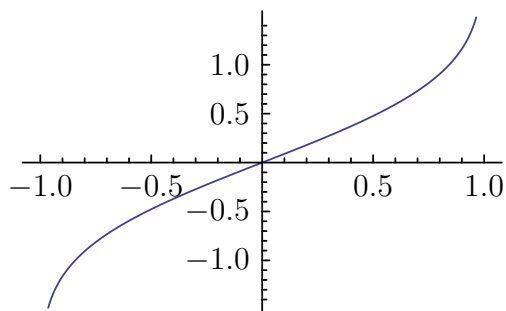
>> HermiteH[4.2, 2]
77.5291
```

InverseErf

`InverseErf[z]`
returns the inverse error function of z .

```
>> InverseErf /@ {-1, 0, 1}
{-∞, 0, ∞}
```

```
>> Plot[InverseErf[x], {x, -1, 1}]
```



InverseErf[z] only returns numeric values for $-1 \leq z \leq 1$:

```
>> InverseErf /@ {0.9, 1.0, 1.1}
{1.16309, ∞, InverseErf[1.1]}
```

InverseErfc

InverseErfc[z]
returns the inverse complementary error function of z.

```
>> InverseErfc /@ {0, 1, 2}
{∞, 0, -∞}
```

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]
1410.02 + 5797.3I
```

KelvinBei

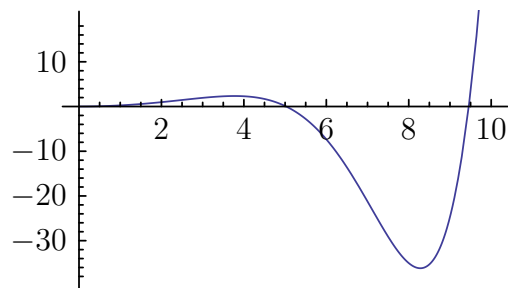
KelvinBei[z]
returns the Kelvin function bei(z).
KelvinBei[n, z]
returns the Kelvin function bei_n(z).

```
>> KelvinBei[0.5]
0.0624932
```

```
>> KelvinBei[1.5 + I]
0.326323 + 0.755606I
```

```
>> KelvinBei[0.5, 0.25]
0.370153
```

```
>> Plot[KelvinBei[x], {x, 0, 10}]
```



KelvinBer

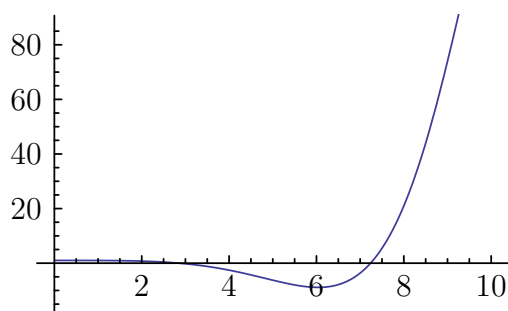
KelvinBer[z]
returns the Kelvin function ber(z).
KelvinBer[n, z]
returns the Kelvin function ber_n(z).

```
>> KelvinBer[0.5]
0.999023
```

```
>> KelvinBer[1.5 + I]
1.1162 - 0.117944I
```

```
>> KelvinBer[0.5, 0.25]
0.148824
```

```
>> Plot[KelvinBer[x], {x, 0, 10}]
```



KelvinKei

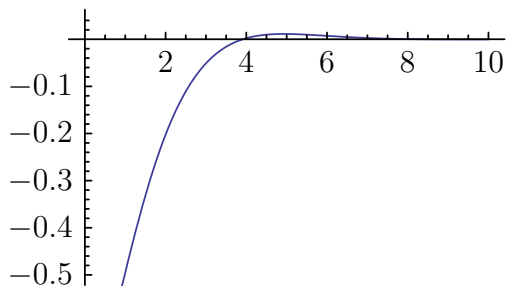
KelvinKei[z]
returns the Kelvin function kei(z).
KelvinKei[n, z]
returns the Kelvin function kei_n(z).

```
>> KelvinKei[0.5]
-0.671582

>> KelvinKei[1.5 + I]
-0.248994 + 0.303326I

>> KelvinKei[0.5, 0.25]
-2.0517

>> Plot[KelvinKei[x], {x, 0, 10}]
```



KelvinKer

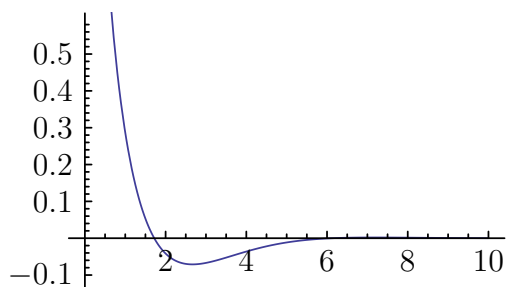
`KelvinKer[z]`
returns the Kelvin function $\ker(z)$.
`KelvinKer[n, z]`
returns the Kelvin function $\ker_n(z)$.

```
>> KelvinKer[0.5]
0.855906

>> KelvinKer[1.5 + I]
-0.167162 - 0.184404I

>> KelvinKer[0.5, 0.25]
0.450023

>> 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 $L_n^a(x)$.

```
>> LaguerreL[8, x]
1 - 8x + 14x^2 - 28x^3/3 + 35x^4/12
- 7x^5/15 + 7x^6/180 - x^7/630 + x^8/40320

>> LaguerreL[3/2, 1.7]
-0.947134

>> LaguerreL[5, 2, x]
21 - 35x + 35x^2/2 - 7x^3/2 + 7x^4/24 - x^5/120
```

LegendreP

`LegendreP[n, x]`
returns the Legendre polynomial $P_n(x)$.
`LegendreP[n, m, x]`
returns the associated Legendre polynomial $P_n^m(x)$.

```
>> LegendreP[4, x]
3/8 - 15x^2/4 + 35x^4/8

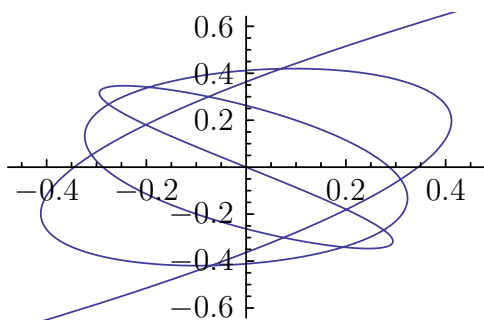
>> LegendreP[5/2, 1.5]
4.17762

>> LegendreP[1.75, 1.4, 0.53]
-1.32619

>> LegendreP[1.6, 3.1, 1.5]
-0.303998 - 1.91937I
```

`LegendreP` can be used to draw generalized Lis-sajous 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]
0.036211 - 6.56219I

>> LegendreQ[1.75, 1.4, 0.53]
2.05499

>> LegendreQ[1.6, 3.1, 1.5]
-1.71931 - 7.70273I
```

ProductLog

ProductLog[z]
returns the value of the Lambert W function at z.

The defining equation:

```
>> z == ProductLog[z] * E ^
ProductLog[z]
True
```

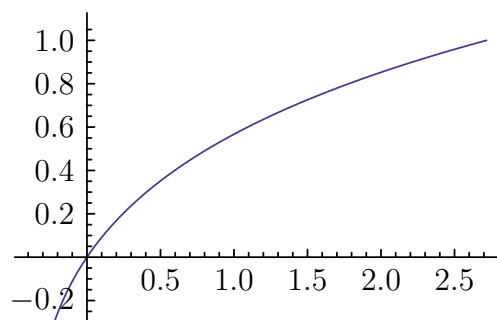
Some special values:

```
>> 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 function $Y_l^m(\theta, \phi)$.

```
>> SphericalHarmonicY[3/4, 0.5, Pi/5, Pi/3]
0.254247 + 0.14679I

>> SphericalHarmonicY[3, 1, theta, phi]

$$\frac{\sqrt{21} (1 - 5 \cos[\theta]^2) E^{I \phi} \sin[\theta]}{8 \sqrt{\pi}}$$

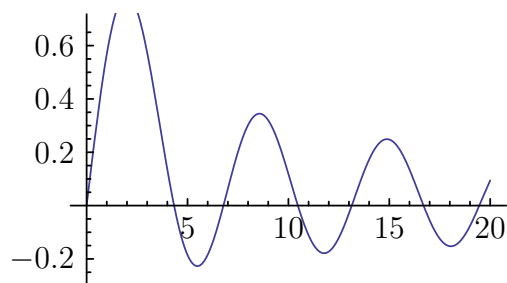
```

StruveH

StruveH[n, z]
returns the Struve function $H_n(z)$.

```
>> StruveH[1.5, 3.5]
1.13192

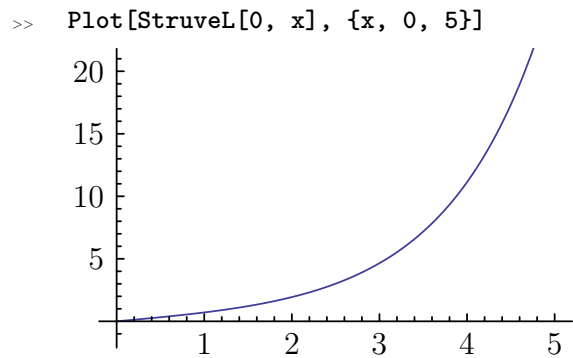
>> Plot[StruveH[0, x], {x, 0, 20}]
```



StruveL

`StruveL[n, z]`
returns the modified Struve function $L_n(z)$.

```
>> StruveL[1.5, 3.5]  
4.41126
```

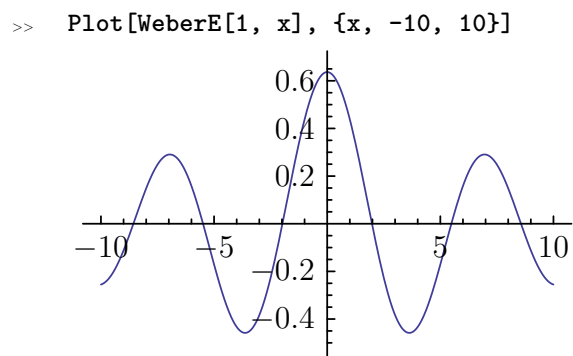


```
>> Zeta[-2.5 + I]  
0.0235936 + 0.0014078I
```

WeberE

`WeberE[n, z]`
returns the Weber function $E_n(z)$.

```
>> WeberE[1.5, 3.5]  
-0.397256
```



Zeta

`Zeta[z]`
returns the Riemann zeta function of z .

```
>> Zeta[2]  
 $\frac{\pi^2}{6}$ 
```

XXXV. Scoping

Contents

Begin	222	Sys-	End	224
BeginPackage	222	tem'Private'\$ContextPathStack 223	EndPackage	224
Block	223	Sys-	Module	224
Context	223	tem'Private'\$ContextStack 223	\$ModuleNumber . . .	224
\$ContextPath	223	\$Context	Unique	225
		Contexts		

Begin

Begin[*context*]
temporarily sets the current context to *context*.

```
>> Begin["test'"]
test'
>> {$Context, $ContextPath}
{test', {Global', System'}}
>> Context[newsymbol]
test'
>> End[]
test'
>> End[]
Nopreviouscontextdefined.
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[]
Nopreviouscontextdefined.
```

Block

Block[{*x*, *y*, ...}, *expr*]
temporarily removes the definitions of the given variables, evaluates *expr*, and restores the original definitions afterwards.

Block[{*x*=*x0*, *y*=*y0*, ...}, *expr*]
assigns temporary values to the variables during the evaluation of *expr*.

```
>> 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 variablespecificationcontainsx+y,
whichisnotasymboloranassignmenttoasymbol.
x
```

Variable names may not appear more than once:

```
>> Block[{x, x}, x]
Duplicatelocalvariablexfoundinlocalvariablespecification
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
is an internal variable tracking the values of \$ContextPath saved by Begin and BeginPackage.

System'Private'\$ContextStack

System'Private'\$ContextStack
is an internal variable tracking 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
{"CombinatoricaOld",
 "Global", "ImportExport",
 "Internal", "System",
 "System'Convert'B64Dump",
 "System'Convert'Image",
 "System'Convert'JSONDump",
 "System'Convert'TableDump",
 "System'Convert'TextDump",
 "System'Private",
 "XML", "XML'Parser"}
```

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 tem-
  porary name of the form name$number,
  where number is the current value of
  $ModuleNumber. Each time a module
  is evaluated, $ModuleNumber is incre-
  mented.
```

```
>> 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.
```

Unique

```
Unique[]
  generates a new symbol and gives a name
  of the form $number.
Unique[x]
  generates a new symbol and gives a name
  of the form x$number.
Unique[{x, y, ...}]
  generates a list of new symbols.
Unique['xxx']
  generates a new symbol and gives a name
  of the form xxxnumber.
```

Create a unique symbol with no particular name:

```
>> Unique[]
$9

>> Unique[sym]
sym$1
```

Create a unique symbol whose name begins with x:

```
>> Unique["x"]
x10
```

Each use of Unique[symbol] increments \$ModuleNumber:

```
>> {$ModuleNumber, Unique[x],
  $ModuleNumber}
{2, x$2, 3}
```

Unique[symbol] creates symbols in the same way Module does:


```
>> {Module[{x}, x], Unique[x]}
     {x$3,x$4}
```

Unique with more arguments

```
>> Unique[{x, "s"}, Flat ^ Listable
         ^ Orderless]

FlatListableObjectlessisnotknownattribute.
Unique  $\left[ \{x, s\}, \text{Flat}^{\text{Listable}^{\text{Orderless}}} \right]$ 
```

Unique call without symbol argument

```
>> Unique[x + y]

x + yisnotasymboloravalidsymbolname.
Unique  $[x + y]$ 
```

XXXVI. String functions

Contents

<code>\$CharacterEncoding</code>	226	<code>NumberString</code>	229	<code>StringReplace</code>	234
<code>\$CharacterEncodings</code>	226	<code>RegularExpression</code>	229	<code>StringRiffle</code>	234
<code>CharacterRange</code>	226	<code>RemoveDiacritics</code>	230	<code>StringSplit</code>	235
<code>Characters</code>	226	<code>StartOfLine</code>	230	<code>StringTake</code>	235
<code>DamerauLevenshteinDistance</code>	227	<code>StartOfString</code>	230	<code>StringTrim</code>	235
<code>DigitCharacter</code>	227	<code>StringCases</code>	231	<code>String</code>	235
<code>DigitQ</code>	227	<code>StringContainsQ</code>	231	<code>ToCharacterCode</code>	236
<code>EditDistance</code>	228	<code>StringDrop</code>	231	<code>ToExpression</code>	236
<code>EndOfLine</code>	228	<code>StringExpression (~~)</code>	231	<code>ToLowerCase</code>	236
<code>EndOfString</code>	228	<code>StringFreeQ</code>	232	<code>ToString</code>	236
<code>FromCharacterCode</code>	228	<code>StringInsert</code>	233	<code>ToUpperCase</code>	237
<code>HammingDistance</code>	229	<code>StringJoin (<>)</code>	233	<code>Transliterate</code>	237
<code>HexadecimalCharacter</code>	229	<code>StringLength</code>	233	<code>UpperCaseQ</code>	237
<code>LetterCharacter</code>	229	<code>StringMatchQ</code>	233	<code>Whitespace</code>	237
<code>LetterQ</code>	229	<code>StringPosition</code>	233	<code>WhitespaceCharacter</code>	237
<code>LowerCaseQ</code>	229	<code>StringQ</code>	234	<code>WordBoundary</code>	237
		<code>StringRepeat</code>	234	<code>WordCharacter</code>	238

\$CharacterEncoding

`CharacterEncoding`
specifies the default character encoding to use if no other encoding is specified.

\$CharacterEncodings

CharacterRange

`CharacterRange["a", "b"]`
returns a list of the Unicode characters from *a* to *b* inclusive.

```
>> CharacterRange["a", "e"]
{a, b, c, d, e}

>> CharacterRange["b", "a"]
{}
```

Characters

`Characters["string"]`
returns a list of the characters in *string*.

```
>> Characters["abc"]
{a, b, c}
```

DamerauLevenshteinDistance

`DamerauLevenshteinDistance[a, b]`
returns the Damerau-Levenshtein distance of *a* and *b*, which is defined as the minimum number of transpositions, insertions, deletions and substitutions needed to transform one into the other. In contrast to `EditDistance`, `DamerauLevenshteinDistance` counts transposition of adjacent items (e.g. "ab" into "ba") as one operation of change.


```
>> EditDistance[{1, E, 2, Pi}, {1,
E, Pi, 2}]
2
```

EndOfLine

EndOfString
represents the end of a line in a string.

```
>> StringReplace["aba\nbba\na\nab",
"a" ~~EndOfLine -> "c"]

abc
  bbc
    c
  ab

>> StringSplit["abc\ndef\nhij",
EndOfLine]

{abc,
 def,
 hij}
```

EndOfString

EndOfString
represents the end of a string.

Test whether strings end with “e”:

```
>> StringMatchQ[#, __ ~~"e" ~~
EndOfString] &/@ {"apple", "
banana", "artichoke"}

{True, False, True}

>> StringReplace["aab\nabb", "b" ~~
EndOfString -> "c"]

aab
  abc
```

FromCharCode

FromCharCode[*n*]
returns the character corresponding to
Unicode codepoint *n*.
FromCharCode[{*n*₁, *n*₂, ...}]
returns a string with characters corre-
sponding to *n*_{*i*}.
FromCharCode[{{*n*₁₁, *n*₁₂, ...}, {
*n*₂₁, *n*₂₂, ...}, ...}]
returns a list of strings.

```
>> FromCharCode[100]
d

>> FromCharCode[228, "ISO8859
-1"]
ä

>> FromCharCode[{100, 101,
102}]
def

>> ToCharCode[%]
{100,101,102}

>> FromCharCode[{{97, 98, 99},
{100, 101, 102}}]
{abc,def}

>> ToCharCode["abc 123"] //
FromCharCode
abc 123
```

HammingDistance

HammingDistance[*u*, *v*]
returns the Hamming distance between
u and *v*, i.e. the number of different el-
ements. *u* and *v* may be lists or strings.

```
>> HammingDistance[{1, 0, 1, 0},
{1, 0, 0, 1}]
2

>> HammingDistance["time", "dime"]
1

>> HammingDistance["TIME", "dime",
IgnoreCase -> True]
1
```

HexidecimalCharacter

HexidecimalCharacter
represents the characters 0-9, a-f and A-F.

```
>> StringMatchQ[#,  
HexidecimalCharacter] & /@ {"a",  
"1", "A", "x", "H", " ", "."}  
  
{True, True, True, False,  
False, False, False}
```

LetterCharacter

LetterCharacter
represents letters.

```
>> StringMatchQ[#, LetterCharacter]  
& /@ {"a", "1", "A", " ", "."}  
  
{True, False, True, False, False}
```

LetterCharacter also matches unicode characters.

```
>> StringMatchQ["\[Lambda]",  
LetterCharacter]  
  
True
```

LetterQ

LetterQ[*string*] yields True if all the characters in the *string* are letters, and yields False otherwise.

```
>> LetterQ["m"]  
True  
  
>> LetterQ["9"]  
False  
  
>> LetterQ["Mathics"]  
True  
  
>> LetterQ["Welcome to Mathics"]  
False
```

LowerCaseQ

LowerCaseQ[*s*]
returns True if *s* consists wholly of lower case characters.

```
>> LowerCaseQ["abc"]  
True
```

An empty string returns True.

```
>> LowerCaseQ[""]  
True
```

NumberString

NumberString
represents the characters in a number.

```
>> StringMatchQ["1234",  
NumberString]  
True  
  
>> StringMatchQ["1234.5",  
NumberString]  
True  
  
>> StringMatchQ["1.2'20",  
NumberString]  
False
```

RegularExpression

RegularExpression[*'regex'*]
represents the regex specified by the string *'regex'*.

```
>> StringSplit["1.23, 4.56 7.89",  
RegularExpression["(\\s|,)+"]]  
  
{1.23, 4.56, 7.89}
```

RemoveDiacritics

RemoveDiacritics[*s*]
returns a version of *s* with all diacritics removed.

```
>> RemoveDiacritics["en prononçant
pêcher et pêcher"]
en prononcant pecher et pecher

>> RemoveDiacritics["piñata"]
pinata
```

StartOfLine

StartOfString
represents the start of a line in a string.

```
>> StringReplace["aba\nbba\na\nab",
  StartOfLine ~~"a" -> "c"]

cba
  bba
    c
  cb

>> StringSplit["abc\ndef\nhij",
  StartOfLine]

{abc
 ,def
 ,hij}
```

StartOfString

StartOfString
represents the start of a string.

Test whether strings start with "a":

```
>> StringMatchQ[#, StartOfString ~~
"a" ~~__] &/@ {"apple", "banana", "artichoke"}

{True, False, True}

>> StringReplace["aba\nabb",
  StartOfString ~~"a" -> "c"]

cba
  abb
```

StringCases

StringCases["string", *pattern*]
gives all occurrences of *pattern* in *string*.
StringReplace["string", *pattern* -> *form*]
gives all instances of *form* that stem from
occurrences of *pattern* in *string*.
StringCases["string", {*pattern1*, *pattern2*,
...}]
gives all occurrences of *pattern1*, *pattern2*,
....
StringReplace["string", *pattern*, *n*]
gives only the first *n* occurrences.
StringReplace[{"string1", "string2",
...}, *pattern*]
gives occurrences in *string1*, *string2*, ...

```
>> StringCases["axbaxxb", "a" ~~x_
~~"b"]

{axb}

>> StringCases["axbaxxb", "a" ~~x_
~~"b"]

{axbaxxb}

>> StringCases["axbaxxb", Shortest
["a" ~~x_ ~~"b"]]

{axb, axxb}

>> StringCases["-abc- def -uvw- xyz", Shortest["-" ~~x_ ~~"-"] ->
x]

{abc, uvw}

>> StringCases["-öhi- -abc- -. -",
  "- " ~~x : WordCharacter .. ~~"- "
  -> x]

{öhi, abc}

>> StringCases["abc-abc xyz-uvw",
  Shortest[x : WordCharacter .. ~~
  "- " ~~x_] -> x]

{abc}

>> StringCases["abba", {"a" -> 10,
  "b" -> 20}, 2]

{10, 20}

>> StringCases["a#ä_123",
  WordCharacter]

{a, ä, 1, 2, 3}
```

```
>> StringCases["a#ä_123",
LetterCharacter]

{a,ä}
```

StringContainsQ

`StringContainsQ["string", patt]`
returns True if any part of *string* matches *patt*, and returns False otherwise.
`StringContainsQ[{‘s1’, “s2”, ...}, patt]’`
returns the list of results for each element of string list.
`StringContainsQ[patt]`
represents an operator form of `StringContainsQ` that can be applied to an expression.

```
>> StringContainsQ["mathics", "m" ~
~__ ~~"s"]
True

>> StringContainsQ["mathics", "a" ~
~__ ~~"m"]
False

>> StringContainsQ["Mathics", "MA"
, IgnoreCase -> True]
True

>> StringContainsQ[{"g", "a", "laxy",
, "universe", "sun"}, "u"]
{False, False, False, True, True}

>> StringContainsQ["e" ~~__ ~~"u"]
/@ {"The Sun", "Mercury", "
Venus", "Earth", "Mars", "
Jupiter", "Saturn", "Uranus", "
Neptune"}
{True, True, True, False, False,
False, False, False, True}
```

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 *n*th character dropped.
`StringDrop["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
```

StringExpression (~~)

`StringExpression[s_1, s_2, ...]`
represents a sequence of strings and symbolic string objects *s_i*.

```
>> "a" ~~ "b" // FullForm
"ab"
```

StringFreeQ

`StringFreeQ["string", patt]`
returns True if no substring in *string* matches the string expression *patt*, and returns False otherwise.

`StringFreeQ[{'s1', "s2", ...}, patt]`
returns the list of results for each element of string list.

`StringFreeQ['string', {p1, p2, ...}]`
returns True if no substring matches any of the *pi*.

`StringFreeQ[patt]`
represents an operator form of `StringFreeQ` that can be applied to an expression.

```
>> StringFreeQ["mathics", "m" ~~__
~~"s"]
False

>> StringFreeQ["mathics", "a" ~~__
~~"m"]
True

>> StringFreeQ["Mathics", "MA" ,
IgnoreCase -> True]
False

>> StringFreeQ[{"g", "a", "laxy", "
universe", "sun"}, "u"]
{True, True, True, False, False}

>> StringFreeQ["e" ~~__ ~~"u"] /@
{"The Sun", "Mercury", "Venus",
"Earth", "Mars", "Jupiter", "
Saturn", "Uranus", "Neptune"}
{False, False, False, True,
True, True, True, True, False}

>> StringFreeQ[{"A", "Galaxy", "Far
", "Far", "Away"}, {"F" ~~__ ~~"
r", "aw" ~~__}, IgnoreCase ->
True]
{True, True, False, False, False}
```

StringInsert

`StringInsert["strsource", "strnew", pos]`
returns a string with *strnew* inserted starting at position *pos* in *strsource*.

`StringInsert["strsource", "strnew", -pos]`
returns a string with *strnew* inserted at position *pos* from the end of *strsource*.

`StringInsert["strsource", "strnew", {pos_1, pos_2, ...}]`
returns a string with *strnew* inserted at each position *pos_i* in *strsource*, the *pos_i* are taken before any insertion is done.

`StringInsert[{str_1, str_2, ...}, "strnew", pos]`
inserts *strnew* to each of *s_i* at the position *pos*

```
>> StringInsert["abcdefghijklm", "X
", 4]
abcXdefghijklm

>> StringInsert["abcdefghijklm", "X
", 1]
Xabcdefghijklm

>> StringInsert["abcdefghijklm", "X
", 14]
abcdefghijklmX

(watch the empty line).
>> StringInsert["abcdefghijklm", "X
", -1]
abcdefghijklmX

>> StringInsert["abcdefghijklm", "X
", -14]
Xabcdefghijklm

(watch the empty line).
>> StringInsert["abcdefghijklm", "X
", {1, 4, 9}]
XabcXdefghXijklm

(watch the empty line).
>> StringInsert[{"abcdefghijklm", "
Mathics"}, "X", 4]
{abcXdefghijklm, MatXhics}

>> StringInsert["1234567890123456",
".", Range[-16, -4, 3]]
1.234.567.890.123.456
```


StringJoin (<>)

```
StringJoin["s1'", "s2", ...]  
returns the concatenation of the strings  
s1, s2, .
```

```
>> StringJoin["a", "b", "c"]  
abc  
  
>> "a" <> "b" <> "c" // InputForm  
"abc"  
  
StringJoin flattens lists out:  
>> StringJoin[{"a", "b"}] //  
InputForm  
"ab"  
  
>> Print[StringJoin[{"Hello", " ",  
{"world"}}, {"!"}]  
  
Helloworld!
```

StringLength

```
StringLength["string"]  
gives the length of string.
```

```
>> StringLength["abc"]  
3  
  
StringLength is listable:  
>> StringLength[{"a", "bc"}]  
{1, 2}  
  
>> StringLength[x]  
Stringexpected.  
StringLength[x]
```

StringMatchQ

```
>> StringMatchQ["abc", "abc"]  
True  
  
>> StringMatchQ["abc", "abd"]  
False  
  
>> StringMatchQ["15a94xcZ6", (  
DigitCharacter | LetterCharacter  
)..]  
True
```

Use StringMatchQ as an operator

```
>> StringMatchQ[LetterCharacter] ["a"  
"]  
True
```

StringPosition

```
StringPosition["string", patt]  
gives a list of starting and ending posi-  
tions where patt matches "string".  
StringPosition["string", patt, n]  
returns the first n matches only.  
StringPosition["string", {patt1, patt2,  
...}, n]  
matches multiple patterns.  
StringPosition[{s1, s2, ...}, patt]  
returns a list of matches for multiple  
strings.
```

```
>> StringPosition["123  
ABCxyABCzzzABCABC", "ABC"]  
{ {4, 6}, {9, 11}, {15, 17}, {18, 20} }  
  
>> StringPosition["123  
ABCxyABCzzzABCABC", "ABC", 2]  
{ {4, 6}, {9, 11} }
```

StringPosition can be useful for searching through text.

```
>> data = Import["ExampleData/  
EinsteinSzilLetter.txt"];  
  
>> StringPosition[data, "uranium"]  
{ {299, 305}, {870, 876}, {1538, 1~  
~544}, {1671, 1677}, {2300, 2306  
}, {2784, 2790}, {3093, 3099} }
```

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}
```

StringRepeat

```
StringRepeat["string", n]
  gives string repeated n times.
StringRepeat["string", n, max]
  gives string repeated n times, but not
  more than max characters.
```

```
>> StringRepeat["abc", 3]
abcabcabc

>> StringRepeat["abc", 10, 7]
abcabca
```

StringReplace

```
StringReplace["string", "a" -> "b"]
  replaces each occurrence of old with new
  in string.
StringReplace["string", {"s1" -> "sp1",
"s2" -> "sp2"}]
  performs multiple replacements of each
  si by the corresponding spi in string.
StringReplace["string", srules, n]
  only performs the first n replacements.
StringReplace[{string1, string2,
...}, srules]
  performs the replacements specified by
  srules on a list of strings.
```

StringReplace replaces all occurrences of one substring with another:

```
>> StringReplace["xyxyxyxyxyxyxy",
"xy" -> "A"]
AAxyxxAyA
```

Multiple replacements can be supplied:

```
>> StringReplace["xyzwxyzwxyzxyzw",
{"xyz" -> "A", "w" -> "BCD"}]
ABCDABCDxABCBD
```

Only replace the first 2 occurrences:

```
>> StringReplace["xyxyxyxyxyxyxy",
"xy" -> "A", 2]
AAxyxyxyxyxy
```

Also works for multiple rules:

```
>> StringReplace["abba", {"a" -> "A",
"b" -> "B"}, 2]
ABba
```

StringReplace acts on lists of strings too:

```
>> StringReplace[{"xyxyxyxy", "xyxyxyxyxyxy"},
"xy" -> "A"]
{AAxA, yAAxxAyA}
```

StringReplace also can be used as an operator:

```
>> StringReplace["y" -> "ies"]["city"]
cities
```

StringRiffle

```
StringRiffle[{s1, s2, s3, ...}]
  returns a new string by concatenating
  all the si, with spaces inserted between
  them.
StringRiffle[list, sep]
  inserts the separator sep between all ele-
  ments in list.
StringRiffle[list, {"left", "sep",
"right"}]
  use left and right as delimiters after con-
  catenation.
```

```
>> StringRiffle[{"a", "b", "c", "d",
"e"}]
a b c d e

>> StringRiffle[{"a", "b", "c", "d",
"e"}, " "]
a, b, c, d, e

>> StringRiffle[{"a", "b", "c", "d",
"e"}, {"(", " ", ")"}]
(a b c d e)
```

StringSplit

```
StringSplit["s"]
  splits the string s at whitespace, discarding
  the whitespace and returning a list of
  strings.
StringSplit["s'", "d"]
  splits s at the delimiter d.
StringSplit[s, {"d1'", "d2", ...}]
  splits s using multiple delimiters.
```

```
>> StringSplit["abc,123", ","]
{abc,123}

>> StringSplit["abc 123"]
{abc,123}

>> StringSplit["abc,123.456", {"",",",
  "."}]
{abc,123,456}

>> StringSplit["a b c",
  RegularExpression[" +"]]
{a,b,c}
```

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 nth character in string.
StringTake["string", {m, n}]
  gives characters m through n in string.
StringTake["string", {m, n, s}]
  gives characters m through n in steps of s.
```

```
>> StringTake["abcde", 2]
ab

>> StringTake["abcde", 0]

(watch the empty line).
>> StringTake["abcde", -2]
de

>> StringTake["abcde", {2}]
b
```

```
>> StringTake["abcd", {2,3}]
bc

>> StringTake["abcdefgh", {1, 5,
  2}]
ace
```

StringTake also supports standard sequence specifications

```
>> StringTake["abcdef", All]
abcdef
```

StringTrim

```
StringTrim[s]
  returns a version of s with whitespace removed
  from start and end.
```

```
>> StringJoin["a", StringTrim[" \tb
  \n "], "c"]
abc

>> StringTrim["ababaxababyaabab",
  RegularExpression["(ab)+"]]
axababya
```

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 character codes (Unicode codepoints).
`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}

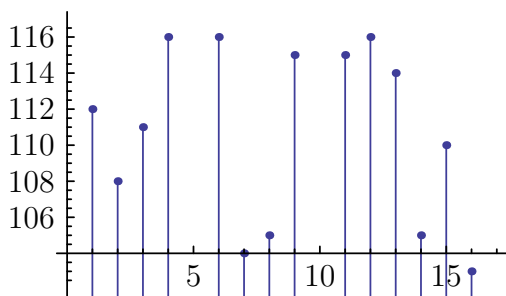
>> ToCharacterCode["ä", "UTF8"]
{195, 164}

>> ToCharacterCode["ä", "ISO8859-1"]
{228}

>> 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]`
interprets 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
```

ToLowerCase

`ToLowerCase[s]`
returns *s* in all lower case.

```
>> ToLowerCase["New York"]
new york
```

ToString

`ToString[expr]`
returns a string representation of *expr*.

```
>> ToString[2]
2

>> ToString[2] // InputForm
"2"

>> ToString[a+b]
a + b

>> "U" <> 2
String expected.
U<>2

>> "U" <> ToString[2]
U2
```

ToUpperCase

```
ToUpperCase[s]  
returns s in all upper case.
```

```
>> ToUpperCase["New York"]  
NEW YORK
```

Transliterate

```
Transliterate[s]  
transliterates a text in some script into an  
ASCII string.
```

```
# The following examples were taken from  
# https://en.wikipedia.org/wiki/Iliad, #  
https://en.wikipedia.org/wiki/Russian_language,  
and # https://en.wikipedia.org/wiki/Hiragana
```

UpperCaseQ

```
UpperCaseQ[s]  
returns True if s consists wholly of upper  
case characters.
```

```
>> UpperCaseQ["ABC"]  
True
```

An empty string returns True.

```
>> UpperCaseQ[""]  
True
```

Whitespace

```
Whitespace  
represents a sequence of whitespace char-  
acters.
```

```
>> StringMatchQ["\r\n", Whitespace  
]  
True
```

```
>> StringSplit["a\nb\r\ncd",  
Whitespace]  
{a,b,c,d}
```

```
>> StringReplace[" this has leading  
and trailing whitespace\n", (  
StartOfString ~~Whitespace)| (  
Whitespace ~~EndOfString)-> ""]  
<> " removed" // FullForm
```

```
"this has leading and trailing  
whitespace removed"
```

WhitespaceCharacter

```
WhitespaceCharacter  
represents a single whitespace character.
```

```
>> StringMatchQ["\n",  
WhitespaceCharacter]  
True  
>> StringSplit["a\nb\r\nc\r",  
WhitespaceCharacter]  
{a,b,c,d}
```

For sequences of whitespace characters use
Whitespace:

```
>> StringMatchQ["\n",  
WhitespaceCharacter]  
False  
>> StringMatchQ["\n", Whitespace]  
True
```

WordBoundary

```
WordBoundary  
represents the boundary between words.
```

```
>> StringReplace["apple banana  
orange artichoke", "e" ~~  
WordBoundary -> "E"]  
applE banana orangE artichokE
```

WordCharacter

```
WordCharacter  
represents a single letter or digit charac-  
ter.
```

```
>> StringMatchQ[#, WordCharacter]
    &/@ {"1", "a", "A", ",", " "}
    {True, True, True, False, False}
```

Test whether a string is alphanumeric:

```
>> StringMatchQ["abc123DEF",
    WordCharacter..]
    True

>> StringMatchQ["$b;123",
    WordCharacter..]
    False
```

XXXVII. Structure

Contents

Apply (@@)	239	Head	241	Scan	243
ApplyLevel (@@@)	239	Map (/@)	241	Sort	244
AtomQ	240	MapIndexed	242	SortBy	244
Combinatori- caOld'BinarySearch	240	MapThread	242	SymbolName	244
ByteCount	240	Null	242	SymbolQ	244
Depth	240	Operate	243	Symbol	244
Flatten	241	Order	243	Thread	244
FreeQ	241	OrderedQ	243	Through	245
		PatternsOrderedQ	243		

Apply (@@)

`Apply[f, expr]`
`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]`
`f @@@ expr`
is equivalent to `Apply[f, expr, {1}]`.

```
>> f @@@ {{a, b}, {c, d}}
{f[a,b],f[c,d]}
```

AtomQ

`AtomQ[x]`
is true if `x` is an atom (an object such as a number or string, which cannot be divided into subexpressions using `Part`).

```
>> AtomQ[x]
True
```

```
>> AtomQ[1.2]
True

>> AtomQ[2 + 1]
True

>> AtomQ[2 / 3]
True

>> AtomQ[x + y]
False
```

CombinatoricaOld'BinarySearch

`CombinatoricaOld'BinarySearch[l, k]`
searches the list *l*, which has to be sorted, for key *k* and returns its index in *l*. If *k* does not exist in *l*, `BinarySearch` returns $(a + b) / 2$, where *a* and *b* are the indices between which *k* would have to be inserted in order to maintain the sorting order in *l*. Please note that *k* and the elements in *l* need to be comparable under a strict total order (see https://en.wikipedia.org/wiki/Total_order).

`CombinatoricaOld'BinarySearch[l, k, f]`
the index of *k* in the elements of *l* if *f* is applied to the latter prior to comparison. Note that *f* needs to yield a sorted sequence if applied to the elements of *l*.

```
>> CombinatoricaOld'BinarySearch
    [{3, 4, 10, 100, 123}, 100]
4

>> CombinatoricaOld'BinarySearch
    [{2, 3, 9}, 7] // N
2.5

>> CombinatoricaOld'BinarySearch
    [{2, 7, 9, 10}, 3] // N
1.5

>> CombinatoricaOld'BinarySearch
    [{-10, 5, 8, 10}, -100] // N
0.5

>> CombinatoricaOld'BinarySearch
    [{-10, 5, 8, 10}, 20] // N
4.5
```

```
>> CombinatoricaOld'BinarySearch[{{
    a, 1}, {b, 7}}, 7, #[[2]]&]
2
```

ByteCount

`ByteCount[expr]`
gives the internal memory space used by *expr*, in bytes.

The results may heavily depend on the Python implementation in use.

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[*expr*, *x*]
returns True if *expr* does not contain the expression *x*.

```
>> 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

>> FreeQ[x_+y_+z_] [a+b]
True
```

Head

Head[*expr*]
returns the head of the expression or atom *expr*.

```
>> 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[Plus] [f[a], f[b], f[c]]
```

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[List, {0}] [f[a, {1}],
                    f[b, {2}], f[c, {3}]]
```

Map on levels 0 through 1 (outer expression gets index {}):

```
>> MapIndexed[f, a + b + c * d, {0,
      1}]
      f[f[a, {1}] + f[b,
          {2}] + f[cd, {3}], {}]
```

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} [ {1}, {2, 0} [ {2, 1},
          {2, 2, 0} [ {2, 2, 1}], {2, 3,
          0} [ {2, 3, 1}, {2, 3, 2}]]]
```

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 + b f[g] c^e
```

MapThread

```
MapThread[f, {{a1, a2, ...}, {b1, b2, ...}, ...}]
  returns      {f[a1, b1, ...], f[a2,
                b2, ...], ...}.
MapThread[f, {expr1, expr2, ...}, n]
  applies f at level n.
```

```
>> MapThread[f, {{a, b, c}, {1, 2,
      3}}]
      {f[a, 1], f[b, 2], f[c, 3]}

>> MapThread[f, {{{a, b}, {c, d}},
      {{e, f}, {g, h}}}, 2]
      {{f[a, e], f[b, f]},
        {f[c, g], f[d, h]}}
```

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:

```
>> ""
```

Operate

```
Operate[p, expr]
  applies p to the head of expr.
Operate[p, expr, n]
  applies p to the nth 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]]
```

Order

`Order[x, y]`
returns a number indicating the canonical ordering of x and y . 1 indicates that x is before y , -1 that y is before x . 0 indicates that there is no specific ordering. Uses the same order as `Sort`.

```
>> Order[7, 11]
1
>> Order[100, 10]
-1
>> Order[x, z]
1
>> Order[x, x]
0
```

OrderedQ

`OrderedQ[a, b]`
is True if a sorts before b according to canonical ordering.

```
>> OrderedQ[a, b]
True
>> OrderedQ[b, a]
False
```

PatternsOrderedQ

`PatternsOrderedQ[patt1, patt2]`
returns True if pattern $patt1$ would be applied before $patt2$ according to canonical pattern ordering.

```
>> PatternsOrderedQ[x_, x_]
False
>> PatternsOrderedQ[x_, x__]
True
```

```
>> PatternsOrderedQ[b, a]
True
```

Scan

`Scan[f, expr]`
applies f to each element of $expr$ and returns Null.
`'Scan[f, expr, levelspec]`
applies f to each level specified by $levelspec$ of $expr$.

```
>> Scan[Print, {1, 2, 3}]
1
2
3
```

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 + c_, 1 + b__}
```

```
>> Sort[{x_ + n*y_, x_ + y_},
PatternsOrderedQ]
{x_ + n*y_, x_ + y_}
```

SortBy

`SortBy[list, f]`
 sorts *list* (or the leaves of any other expression) according to canonical ordering of the keys that are extracted from the *list*'s elements using \$f. Chunks of leaves that appear the same under \$f are sorted according to their natural order (without applying \$f).

`SortBy[f]`
 creates an operator function that, when applied, sorts by \$f.

```
>> SortBy[{{5, 1}, {10, -1}}, Last]
{{10, -1}, {5, 1}}

>> SortBy[Total][{{5, 1}, {10, -9}}]
{{10, -9}, {5, 1}}
```

SymbolName

`SymbolName[s]`
 returns the name of the symbol *s* (without any leading context name).

```
>> SymbolName[x] // InputForm
"x"
```

SymbolQ

`SymbolQ[x]`
 is True if *x* is a symbol, or False otherwise.

```
>> 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]]
```

XXXVIII. System functions

Contents

<code>\$Aborted</code>	246	<code>\$Machine</code>	247	<code>\$ProcessorType</code>	247
<code>\$ByteOrdering</code>	246	<code>\$MachineName</code>	247	<code>\$ScriptCommandLine</code> .	248
<code>\$CommandLine</code>	246	<code>Names</code>	247	<code>\$SystemID</code>	248
<code>Environment</code>	246	<code>\$Packages</code>	247	<code>\$SystemWordLength</code> .	248
<code>\$Failed</code>	246	<code>\$ParentProcessID</code> . . .	247	<code>\$Version</code>	248
<code>GetEnvironment</code>	246	<code>\$ProcessID</code>	247		

`$Aborted`

`$Aborted`
is returned by a calculation that has been aborted.

`$ByteOrdering`

`$ByteOrdering`
returns the native ordering of bytes in binary data on your computer system.

```
>> $ByteOrdering
-1
```

`$CommandLine`

`$CommandLine`
is a list of strings passed on the command line to launch the Mathics session.

```
>> $CommandLine
{mathics/test.py,-o,-k}
```

Environment

`Environment[var]`
gives the value of an operating system environment variable.

```
>> Environment["HOME"]
/home/rocky
```

`$Failed`

`$Failed`
is returned by some functions in the event of an error.

GetEnvironment

`GetEnvironment["var$"]`
gives the setting corresponding to the variable “var” in the operating system environment.

```
>> = GetEnvironment["HOME"]
```

\$Machine

\$Machine
returns a string describing the type of computer system on which the Mathics is being run.

```
>> $Machine
linux
```

\$MachineName

\$MachineName
is a string that gives the assigned name of the computer on which Mathics is being run, if such a name is defined.

```
>> $MachineName
muffin
```

Names

Names["*pattern*"]
returns the list of names matching *pattern*.

```
>> Names["List"]
{List}
```

The wildcard * matches any character:

```
>> Names["List*"]
{List, ListLinePlot,
 ListPlot, ListQ, Listable}
```

The wildcard @ matches only lowercase characters:

```
>> Names["List@"]
{Listable}
```

```
>> x = 5;
```

```
>> Names["Global`*"]
{x}
```

The number of built-in symbols:

```
>> Length[Names["System`*"]]
1027
```

\$Packages

\$Packages
returns a list of the contexts corresponding to all packages which have been loaded into Mathics.

```
>> $Packages
{CombinatoricaOld, ImportExport,
 Internal, System, XML}
```

\$ParentProcessID

\$ParentProcessID
gives the ID assigned to the process which invokes the Mathics by the operating system under which it is run.

```
>> $ParentProcessID
25014
```

\$ProcessID

\$ProcessID
gives the ID assigned to the Mathics process by the operating system under which it is run.

```
>> $ProcessID
25015
```

\$ProcessorType

\$ProcessorType
gives a string giving the architecture of the processor on which the Mathics is being run.

```
>> $ProcessorType
x86_64
```

\$ScriptCommandLine

\$ScriptCommandLine
is a list of string arguments when running the kernel in script mode.

```
>> $ScriptCommandLine  
{}
```

\$SystemID

\$SystemID
is a short string that identifies the type of computer system on which the Mathics is being run.

```
>> $SystemID  
linux
```

\$SystemWordLength

\$SystemWordLength
gives the effective number of bits in raw machine words on the computer system where Mathics is running.

```
>> $SystemWordLength  
64
```

\$Version

\$Version
returns a string with the current Mathics version and the versions of relevant libraries.

```
>> $Version  
Mathics 1.1.dev0 on CPython  
3.6.12 (default, Aug 30  
2020, 12:38:22) using  
SymPy 1.6.2, mpmath 1.1.0
```


XXXIX. Tensor functions

Contents

ArrayDepth	249	Dot (.)	250	Outer	251
ArrayQ	249	IdentityMatrix	250	Transpose	251
DiagonalMatrix	249	Inner	250	VectorQ	251
Dimensions	250	MatrixQ	250		

ArrayDepth

`ArrayDepth[a]`
returns the depth of the non-ragged array *a*, defined as `Length[Dimensions[a]]`.

```
>> 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[expr]`
returns a list of the dimensions of the expression *expr*.

A vector of length 3:

```
>> Dimensions[{a, b, c}]
{3}
```

A 3x2 matrix:

```
>> 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 (.)

```
Dot[x, y]
x . y
    computes the vector dot product or matrix product  $x \cdot y$ .
```

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, x, y, g]
    computes a generalised inner product of  $x$  and  $y$ , using a multiplication function  $f$  and an addition function  $g$ .
```

```
>> Inner[f, {a, b}, {x, y}, g]
g[f[a, x], f[b, y]]
```

Inner can be used to compute a dot product:

```
>> Inner[Times, {a, b}, {c, d},
    Plus] == {a, b} . {c, d}
True
```

The inner product of two boolean matrices:

```
>> Inner[And, {{False, False}, {
    False, True}}, {{True, False}, {
    True, True}}, Or]
{{False, False}, {True, 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[m]
    returns True if  $m$  is a list of equal-length lists.
MatrixQ[m, f]
    only returns True if  $f[x]$  returns True for each element  $x$  of the matrix  $m$ .
```

```
>> MatrixQ[{{1, 3}, {4.0, 3/2}},
    NumberQ]
True
```

Outer

```
Outer[f, x, y]
    computes a generalised outer product of  $x$  and  $y$ , using the function  $f$  in place of multiplication.
```

```
>> 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

Transpose $[m]$

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 $[v]$

returns True if v is a list of elements which are not themselves lists.

VectorQ $[v, f]$

returns True if v is a vector and $f[x]$ returns True for each element x of v .

```
>> VectorQ[{a, b, c}]
True
```

XL. XML

Contents

XML'PlaintextImport .	252	XMLElement	252	XMLObject	252
XML'TagsImport . . .	252	XML'Parser'XMLGet .	252	XML'XMLObjectImport	252
		XML'Parser'XMLGetString	252		

XML'PlaintextImport

```
>> StringReplace[StringTake[Import
["ExampleData/InventionNo1.xml",
"Plaintext"],31],
FromCharacterCode[10]->"/"]
MuseScore 1.2/2012-09-12/5.7/40
```

XML'TagsImport

```
>> Take[Import["ExampleData/
InventionNo1.xml", "Tags"], 10]
{accidental, alter, arpeggiate,
articulations, attributes, backup,
bar-style, barline, beam, beat-type}
```

XMLElement

XML'Parser'XMLGet

XML'Parser'XMLGetString

```
>> Head[XML'Parser'XMLGetString["<a
></a>"]]
XMLObject[Document]
```

XMLObject

XML'XMLObjectImport

```
>> Part[Import["ExampleData/
InventionNo1.xml", "XMLObject"],
2, 3, 1]
XMLElement[identification,
{ }, {XMLElement[encoding,
{ }, {XMLElement[software,
{ }, {MuseScore 1.2}]}],
XMLElement[encoding-date,
{ }, {2012-09-12}]}]}]}
>> Part[Import["ExampleData/
Namespaces.xml"], 2]
XMLElement[book,
{{http://www.w3.org/2000/xmlns/,
xmlns}->urn:loc.gov:books},
{XMLElement[title, { }, {Cheaper
by the Dozen}]}, XMLElement[
{urn:ISBN:0-395-36341-6, number},
{ }, {1568491379}], XMLElement[
notes, { }, {XMLElement[p,
{{http://www.w3.org/2000/xmlns/,
xmlns}->http://www.w3.org/1999/xhtml},
{This is a, XMLElement[i,
{ }, {funny, book!}]}]}]}]}]}]
```

XLI. Optimization

Contents

Maximize	253	Minimize	253
--------------------	-----	--------------------	-----

Maximize

`Maximize[f, x]`
compute the maximum of f respect x that
change between a and b

```
>> Maximize[-2 x^2 - 3 x + 5, x]

$$\left\{ \left\{ \frac{49}{8}, \left\{ x - > -\frac{3}{4} \right\} \right\} \right\}$$

#» Maximize[1 - (x y - 3)^2, {x, y}] = {{1, {x -> 3, y -> 1}}}
#» Maximize[{x - 2 y, x^2 + y^2 <= 1}, {x, y}] =
{{Sqrt[5], {x -> Sqrt[5] / 5, y -> -2 Sqrt[5] / 5}}}
```

Minimize

`Minimize[f, x]`
compute the minimum of f respect x that
change between a and b

```
>> Minimize[2 x^2 - 3 x + 5, x]

$$\left\{ \left\{ \frac{31}{8}, \left\{ x - > \frac{3}{4} \right\} \right\} \right\}$$

#» Minimize[(x y - 3)^2 + 1, {x, y}] = {{1, {x -> 3, y -> 1}}}
#» Minimize[{x - 2 y, x^2 + y^2 <= 1}, {x, y}] =
{{-Sqrt[5], {x -> -Sqrt[5] / 5, y -> 2 Sqrt[5] / 5}}}
```

XLII. File Operations

Contents

AbsoluteFileName . . .	254	FileHash	259	\$Path	263
BinaryRead	255	FileInformation	259	\$PathnameSeparator . .	263
BinaryWrite	256	FileNameDepth	259	Put (>>)	263
Byte	256	FileNameJoin	259	PutAppend (>>>) . . .	264
Character	256	FileNameSplit	259	Read	264
Close	256	FilePrint	260	ReadList	264
Compress	256	FileType	260	Record	264
CopyDirectory	256	Find	260	RenameDirectory . . .	264
CopyFile	256	FindFile	260	RenameFile	265
CreateDirectory	257	FindList	261	ResetDirectory	265
DeleteDirectory	257	Get (<<)	261	\$RootDirectory	265
DeleteFile	257	\$HomeDirectory	261	SetDirectory	265
Directory	257	\$InitialDirectory	261	SetFileDate	265
DirectoryName	257	\$Input	261	SetStreamPosition . . .	265
DirectoryQ	257	\$InputFileName	261	Skip	266
DirectoryStack	257	InputStream	261	StreamPosition	266
EndOfFile	258	\$InstallationDirectory .	262	Streams	266
ExpandFileName	258	Needs	262	StringToStream	267
Expression	258	Number	262	\$TemporaryDirectory .	267
File	258	OpenAppend	262	ToFileName	267
FileBaseName	258	OpenRead	262	Uncompress	267
FileByteCount	258	OpenWrite	262	Word	267
FileDate	258	\$OperatingSystem . . .	262	Write	267
FileExistsQ	258	OutputStream	262	WriteString	268
FileExtension	259	ParentDirectory	263		

AbsoluteFileName

AbsoluteFileName["name"]
returns the absolute version of the given filename.

```
>> AbsoluteFileName["ExampleData/
sunflowers.jpg"]
/src/external-vcs/github/mathics/Mathics/matl
```

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/tmp654i856,180]
>> BinaryWrite[strm, {97, 98, 99}]
  OutputStream [
    /tmp/tmp654i856,180]
>> Close[strm]
  /tmp/tmp654i856
>> strm = OpenRead[%, BinaryFormat
-> True]
  InputStream [
    /tmp/tmp654i856,181]
>> 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 speci-
fied type.

BinaryWrite[channel, {x1, x2, ...}, {
type1, type2, ...}]
writes a sequence of objects using a se-
quence of specified types.

```
>> strm = OpenWrite[BinaryFormat ->
  True]
  OutputStream [
    /tmp/tmp_z0ukad,316]
```

```
>> BinaryWrite[strm, {39, 4, 122}]
  OutputStream [
    /tmp/tmp_z0ukad,316]
>> Close[strm]
  /tmp/tmp_z0ukad
>> strm = OpenRead[%, BinaryFormat
-> True]
  InputStream [
    /tmp/tmp_z0ukad,317]
>> BinaryRead[strm]
  39
>> BinaryRead[strm, "Byte"]
  4
>> BinaryRead[strm, "Character8"]
  z
>> Close[strm];
```

Write a String

```
>> strm = OpenWrite[BinaryFormat ->
  True]
  OutputStream [
    /tmp/tmp_7_debs2,318]
>> BinaryWrite[strm, "abc123"]
  OutputStream [
    /tmp/tmp_7_debs2,318]
>> Close[%]
  /tmp/tmp_7_debs2
```

Read as Bytes

```
>> strm = OpenRead[%, BinaryFormat
-> True]
  InputStream [
    /tmp/tmp_7_debs2,319]
>> BinaryRead[strm, {"Character8",
  "Character8", "Character8", "
  Character8", "Character8", "
  Character8", "Character8"}]
  {a,b,c,1,2,3,EndOfFile}
>> Close[strm]
  /tmp/tmp_7_debs2
```

Read as Characters

```
>> strm = OpenRead[%, BinaryFormat
-> True]

InputStream [
  /tmp/tmp_7_debs2, 320]

>> BinaryRead[strm, {"Byte", "Byte",
  "Byte", "Byte", "Byte", "Byte",
  "Byte"}]

{97, 98, 99, 49, 50, 51, EndOfFile}

>> Close[strm]
/tmp/tmp_7_debs2
```

Write Type

```
>> strm = OpenWrite[BinaryFormat ->
  True]

OutputStream [
  /tmp/tmpzd18g7d4, 321]

>> BinaryWrite[strm, 97, "Byte"]

OutputStream [
  /tmp/tmpzd18g7d4, 321]

>> BinaryWrite[strm, {97, 98, 99},
  {"Byte", "Byte", "Byte"}]

OutputStream [
  /tmp/tmpzd18g7d4, 321]

>> Close[%]
/tmp/tmpzd18g7d4

>> strm = OpenWrite["/dev/full",
  BinaryFormat -> True]

OutputStream [/dev/full, 322]

>> BinaryWrite[strm, {39, 4, 122}]
Nospaceleftondevice.

OutputStream [/dev/full, 322]

>> Close[strm]
Nospaceleftondevice.
/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/tmpqjg0nsk2
```

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/mahco582b
```

DeleteDirectory

```
DeleteDirectory["dir"]
  deletes a directory called dir.
```

```
>> dir = CreateDirectory[]
      /tmp/mwzoe fg0c
>> 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[]
      /src/external-vcs/github/mathics/Mathics
```

DirectoryName

```
DirectoryName["name"]
  extracts the directory name from a file-
  name.
```

```
>> 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[]
      {/src/external-vcs/github/mathics/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"]  
/src/external-vcs/github/mathics/Mathics/ExampleData/sunflowers.jpg
```

Expression

Expression
is a data type for Read.

File

FileNameBase

FileNameBase["*file*"]
gives the base name for the specified file name.

```
>> FileNameBase["file.txt"]  
file  
  
>> FileNameBase["file.tar.gz"]  
file.tar
```

FileByteCount

FileByteCount [*file*]
returns the number of bytes in *file*.

```
>> FileByteCount["ExampleData/sunflowers.jpg"]  
142286
```

FileDate

FileDate [*file*, *types*]
returns the time and date at which the file was last modified.

```
>> FileDate["ExampleData/sunflowers.jpg"]  
{2020,9,6,7,16,33.2822}
```

```
>> FileDate["ExampleData/sunflowers.jpg", "Access"]  
{2020,10,11,0,23,24.288}
```

```
>> FileDate["ExampleData/sunflowers.jpg", "Creation"]  
Missing[NotApplicable]
```

```
>> FileDate["ExampleData/sunflowers.jpg", "Change"]  
{2020,9,6,7,16,33.2822}
```

```
>> FileDate["ExampleData/sunflowers.jpg", "Modification"]  
{2020,9,6,7,16,33.2822}
```

```
>> FileDate["ExampleData/sunflowers.jpg", "Rules"]  
{Access->{2020,10,11,0,23,24.288}, Creation->Missing[  
NotApplicable], Change->{2020,9,6,7,16,33.282~  
~2}, Modification->{2020,9,6,7,16,33.2822}}
```

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*.
The types supported are “MD5”, “Adler32”, “CRC32”, “SHA”, “SHA224”, “SHA256”, “SHA384”, and “SHA512”.

```
>> 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
```

FileInformation

`FileInformation["file"]`
returns information about *file*.

This function is totally undocumented in MMA!

```
>> FileInformation["ExampleData/sunflowers.jpg"]
{File
 - > /src/external-vcs/github/mathics/Mathics/ExampleD
 FileType - > File, ByteCount - >
 142 286, Date - > 3.80837 × 109}
```

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* together 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/  
nonexistent"]  
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"]  
/src/external-vcs/github/mathics/Mathics/mathics/data/  
  
>> FindFile["VectorAnalysis"]  
/src/external-vcs/github/mathics/Mathics/mathics/pack  
  
>> FindFile["VectorAnalysis"  
VectorAnalysis"]  
/src/external-vcs/github/mathics/Mathics/mathics/pack
```

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 expres-
  sion, returning only the last one.

>> Put[x + y, "example_file"]

>> <<"example_file"
    "x"cannotbefollowedby"
    text{+}y"(line1of"./example_file").

>> Put[x + y, 2x^2 + 4z!, Cos[x] +
    I Sin[x], "example_file"]

>> <<"example_file"
    "x"cannotbefollowedby"
    text{+}y"(line1of"./example_file").

>> 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/rocky
```

\$InitialDirectory

```
$InitialDirectory
  returns the directory from which Mathics
  was started.
```

```
>> $InitialDirectory
    /src/external-vcs/github/mathics/Mathics
```

\$Input

```
$Input
  is the name of the stream from which in-
  put 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, 391]

>> Close[str]
    String
```

\$InstallationDirectory

`$InstallationDirectory`
returns the directory in which *Mathics* was installed.

```
>> $InstallationDirectory  
/src/external-vcs/github/mathics/Mathics/mathics
```

```
>> OpenRead["https://raw.  
githubusercontent.com/mathics/  
Mathics/master/README.rst"]  
  
InputStream [  
https://raw.githubusercontent.com/mathics/Mathics/mas  
412]
```

```
>> Close[%];
```

Needs

`Needs["context"]`
loads the specified context if not already in `$Packages`.

```
>> Needs["VectorAnalysis"]
```

OpenWrite

`OpenWrite['file']`
opens a file and returns an `OutputStream`.

```
>> OpenWrite[]  
  
OutputStream [  
/tmp/tmp5w2ntdaq,416]
```

Number

`Number`
is a data type for `Read`.

\$OperatingSystem

`$OperatingSystem`
gives the type of operating system running *Mathics*.

```
>> $OperatingSystem  
Unix
```

OpenAppend

`OpenAppend['file']`
opens a file and returns an `OutputStream` to which writes are appended.

```
>> OpenAppend[]  
  
OutputStream [  
/tmp/tmpel9t2jy1,405]
```

OutputStream

`OutputStream[name, n]`
represents an output stream.

OpenRead

`OpenRead['file']`
opens a file and returns an `InputStream`.

```
>> OpenRead["ExampleData/  
EinsteinSzilLetter.txt"]  
  
InputStream [  
ExampleData/EinsteinSzilLetter.txt,  
411]
```

```
>> OpenWrite[]  
  
OutputStream [  
/tmp/tmpegeni2kw,419]  
  
>> Close[%]  
/tmp/tmpegeni2kw
```

ParentDirectory

`ParentDirectory []`
returns the parent of the current working directory.
`ParentDirectory ["dir"]`
returns the parent *dir*.

```
>> ParentDirectory []  
/src/external-vcs/github/mathics
```

\$Path

`$Path`
returns the list of directories to search when looking for a file.

```
>> $Path  
{., /home/rocky,  
/src/external-vcs/github/mathics/Mathics/matl  
/src/external-vcs/github/mathics/Mathics/mathics/packages}
```

\$PathnameSeparator

`$PathnameSeparator`
returns a string for the separator 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 ~  
~608 166 064 768 844 377 641 ~  
~568 960 512 000 000 000 000  
  
>> PutAppend[10!, 20!, 30!, "  
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  
  
>> 60! >>> "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 ~
~276 341 183 223 364 380 754 ~
~172 606 361 245 952 449 277 ~
~696 409 600 000 000 000 000

>> "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 ~
~276 341 183 223 364 380 754 ~
~172 606 361 245 952 449 277 ~
~696 409 600 000 000 000 000
"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

`RenameDirectory["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[]
Directorystackisempty.
/src/external-vcs/github/mathics/Mathics
```

\$RootDirectory

```
$RootDirectory
returns the system root directory.

>> $RootDirectory
/
```

SetDirectory

```
SetDirectory[dir]
sets the current working directory to dir.

>> SetDirectory[]
/home/rocky
```

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"]
{2000,1,1,0,0,0.}
```

SetStreamPosition

```
SetStreamPosition[stream, n]
sets the current position in a stream.
```

```
>> str = StringToStream["Mathics is
cool!"]
InputStream[String,501]

>> SetStreamPosition[str, 8]
8

>> Read[str, Word]
is

>> SetStreamPosition[str, Infinity]
16
```

Skip

`Skip[stream, type]`
skips ahead in an input stream by one object of the specified *type*.
`Skip[stream, type, n]`
skips ahead in an input stream 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, 510]
>> Read[str, Word]
Mathics
>> StreamPosition[str]
7
```

Streams

`Streams[]`
returns a list of all open streams.

```
>> Streams[]
{InputStream[<stdin>, 0],
 OutputStream[<stdout>,
 1], OutputStream[<stderr>,
 2], OutputStream[
MathicsNonExampleFile,
 403], OutputStream[
/tmp/tmpwac1_a6l, 404],
 OutputStream[/tmp/tmpel9t2jy1,
 405], InputStream[
/src/external-vcs/github/mathics/Mathics/mathics/data,
 409], InputStream[
/src/external-vcs/github/mathics/Mathics/mathics/data,
 411], OutputStream[
/tmp/tmpcf5u52lh, 415],
 OutputStream[/tmp/tmp5w2ntdaq,
 416], InputStream[
String, 471], InputStream[
String, 481], InputStream[
String, 482], InputStream[
String, 483], InputStream[
String, 484], InputStream[
String, 485], InputStream[
String, 486], InputStream[
String, 487], InputStream[
String, 488], InputStream[
String, 489], InputStream[
String, 491], InputStream[
String, 492], InputStream[
String, 493], InputStream[
String, 494], InputStream[
String, 495], InputStream[
String, 499], InputStream[
String, 500], InputStream[
String, 501], InputStream[
String, 504], InputStream[
String, 505], InputStream[
String, 506], InputStream[
String, 507], InputStream[
String, 508], InputStream[
String, 509], OutputStream[
String, 510], OutputStream[
/tmp/tmpzt95vlw8, 511]}
```

StringToStream

`StringToStream[string]`
converts a *string* to an open input stream.

```
>> strm = StringToStream["abc 123"]
      InputStream [String, 514]
```

\$TemporaryDirectory

`$TemporaryDirectory`
returns the directory used for temporary files.

```
>> $TemporaryDirectory
      /tmp
```

ToFileName

`ToFileName[{"dir_1", "dir_2", ...}]`
joins the *dir_i* together into one path.

`ToFileName` has been superseded by `FileNameJoin`.

```
>> ToFileName[{"dir1", "dir2"}, "file"]
      dir1/dir2/file
>> ToFileName["dir1", "file"]
      dir1/file
>> ToFileName[{"dir1", "dir2", "dir3"}]
      dir1/dir2/dir3
```

Uncompress

`Uncompress["string"]`
recovers an expression from a string generated by `Compress`.

```
>> Compress["Mathics is cool"]
      eJxT8k0sychMLlbILFZlzs/PUQIANFwF1w==
```

```
>> Uncompress[%]
      Mathics is cool
>> a = x ^ 2 + y Sin[x] + 10 Log
      [15];
>> b = Compress[a];
>> Uncompress[b]
      x2 + ySin[x] + 10Log[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/tmp2h1o22sv, 519]
>> Write[str, 10 x + 15 y ^ 2]
>> Write[str, 3 Sin[z]]
>> Close[str]
      /tmp/tmp2h1o22sv
>> 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/tmp9o9qb kib  
  
>> FilePrint[%]  
This is a test 1 This is also a test 2  
  
>> str = OpenWrite[];  
  
>> WriteString[str, "This is a test  
1", "This is also a test 2"]  
  
>> Close[str]  
/tmp/tmpjmaaiky9  
  
>> FilePrint[%]  
This is a test 1 This is also a test 2
```

XLIII. Importing and Exporting

Contents

	\$ExportFormats	270		ImportString	271
Sys-	ExportString	270		ImportEx-	
tem‘Convert‘B64Dump‘B64Decode	FetchURL	270		port‘RegisterExport	272
Sys-	FileFormat	270		ImportEx-	
tem‘Convert‘B64Dump‘B64Encode	Import	271		port‘RegisterImport	273
Export	\$ImportFormats	271			

System‘Convert‘B64Dump‘B64Decode

System‘Convert‘B64Dump‘B64Decode[
string]
Decode *string* in Base64 coding to an expression.

```
>> System‘Convert‘B64Dump‘B64Decode
["R!="]
String"R!
="isnotavalidb64encodedstring.
$Failed
```

System‘Convert‘B64Dump‘B64Decode
[%]

Integrate[f[x], {x, 0, 2}]

```
>> System‘Convert‘B64Dump‘B64Encode
["Sf dx"]
4oirIGYg752MIHg=
```

```
>> System‘Convert‘B64Dump‘B64Decode
[%]
f dx
```

Export

System‘Convert‘B64Dump‘B64Encode

System‘Convert‘B64Dump‘B64Encode[*expr*]
Encodes *expr* in Base64 coding

```
>> System‘Convert‘B64Dump‘B64Encode
["Hello world"]
SGVsbG8gd29ybGQ=
```

```
>> System‘Convert‘B64Dump‘B64Decode
[%]
Hello world
```

```
>> System‘Convert‘B64Dump‘B64Encode
[Integrate[f[x], {x, 0, 2}]]
SW50ZWdyYXRiW2ZbeF0sIHt4LCAwLCAyfv0=
```

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
{BMP, Base64, CSV, GIF, JPEG,
  JPEG2000, PBM, PCX, PGM,
  PNG, PPM, SVG, TIFF, Text}
```

ExportString

```
ExportString[expr, form]
  exports expr to a string, in the format
  form.
Export["file", exprs, elems]
  exports exprs to a string as elements spec-
  ified by elems.
```

```
>> ExportString
[{{1,2,3,4},{3},{2},{4}}, "CSV"]
1,2,3,4
3,
2,
4,

>> ExportString[{{1,2,3,4}}, "CSV"]
1,
2,
3,
4,

>> ExportString[Integrate[f[x], {x
,0,2}], "SVG"]
<svg><mrow><msubsup>
  <mo></mo> <mn>0</mn>
  <mn>2</mn></msubsup>
  <mrow><mi>f</mi>
  <mo></mo> <mi>x</mi>
  <mo></mo></mrow> <mo>
  form="prefix" lspace="0"
  rspace="0.2em"></mo> <mrow>
  <mtext></mtext> <mi>x</mi>
  </mrow></mrow></svg>
```

FetchURL

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

>> FileFormat["ExampleData/benzene.
xyz"]
XYZ
```

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
{BMP, Base64, CSV, GIF,
 ICO, JPEG, JPEG2 000, JSON,
 PBM, PCX, PGM, PNG, PPM,
 Package, TGA, TIFF, Text, XML}
```

ImportString

`ImportString["data" , "format"]`
imports data in the specified format from a string.

`ImportString["file" , elements]`
imports the specified elements from a string.

`ImportString["data"]`
attempts to determine the format of the string from its content.

```
>> str = "Hello!\n This is a
testing text\n";

>> ImportString[str, "Elements"]
{Data, Lines, Plaintext, String, Words}

>> ImportString[str, "Lines"]
{Hello!, This is a testing text}
```

ImportExport‘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]]

>> ImportExport‘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]]

>> ImportExport`RegisterExport["
ExampleFormat2",
ExampleExporter2]

>> Export["sample.txt", "
encodethisstring", "
ExampleFormat2"];

>> FilePrint["sample.txt"]
  rapbqrguvffgevat
```

ImportExport`RegisterImport

```
RegisterImport["format", defaultFunction]
  register defaultFunction as the default
  function used when importing from a file
  of type "format".
RegisterImport["format", {"elem1" :>
conditionalFunction1, "elem2" :> conditional-
Function2, ..., defaultFunction}]
  registers multiple elements (elem1, ...)
  and their corresponding converter func-
  tions (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 func-
  tions.
```

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.

```
>> ImportExport`RegisterImport["
ExampleFormat1",
ExampleFormat1Import]
```

```
>> FilePrint["ExampleData/
ExampleData.txt"]
```

ExampleFileFormat

CreatedbyAngus

0.6294520.586355

0.7110090.687453

0.2465400.433973

0.9268710.887255

0.8251410.940900

0.8470350.127464

0.0543480.296494

0.8385450.247025

0.8386970.436220

0.3094960.833591

```
>> 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}]

>> ImportExport`RegisterImport["
  ExampleFormat2", {"Data" :>
  ExampleFormat2DataImport,
  ExampleFormat2DefaultImport}]

>> Import["ExampleData/ExampleData.
  txt", {"ExampleFormat2", "
  Elements"}]

  {Data, 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

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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. Sub-licensing 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".

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 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.

Prototype

Copyright © 2005-2010 Sam Stephenson

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 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.

MathJax

Copyright © 2009-2010 Design Science, Inc.

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Three.js

Copyright I 2010-2020 Three.js authors.

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.

pymimemagic

Copyright (c) 2009, Xiaohai Lu All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

SciPy

Copyright I 2001, 2002 Enthought, Inc. All rights reserved.

Copyright © 2003-2019 SciPy Developers. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of Enthought nor the names of the SciPy Developers may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS AS IS AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

\$Aborted, 246
\$ByteOrdering, 246
\$CharacterEncoding, 226
\$CharacterEncodings, 226
\$CommandLine, 246
\$Context, 223
\$ContextPath, 223
\$DateStringFormat, 83
\$ExportFormats, 269
\$Failed, 246
\$HistoryLength, 86
\$HomeDirectory, 261
\$ImportFormats, 271
\$InitialDirectory, 261
\$Input, 261
\$InputFileName, 261
\$InstallationDirectory, 262
\$IterationLimit, 87
\$Line, 87
\$Machine, 247
\$MachineEpsilon, 182
\$MachineName, 247
\$MachinePrecision, 182
\$MaxPrecision, 182
\$MinPrecision, 182
\$ModuleNumber, 224
\$OperatingSystem, 262
\$Packages, 247
\$ParentProcessID, 247
\$Path, 263
\$PathnameSeparator, 263
\$Post, 139
\$Pre, 139
\$PrePrint, 139
\$PreRead, 139
\$ProcessID, 247
\$ProcessorType, 247
\$RandomState, 210
\$RecursionLimit, 88
\$RootDirectory, 265
\$ScriptCommandLine, 248
\$SyntaxHandler, 139
\$SystemID, 248
\$SystemTimeZone, 83
\$SystemWordLength, 248
\$TemporaryDirectory, 267
\$TimeZone, 84
\$UseSansSerif, 135
\$Version, 248

Abort, 76
Abs, 38
AbsoluteFileName, 254
AbsoluteThickness, 99
AbsoluteTime, 81
AbsoluteTiming, 81
Accumulate, 147
AddTo, 48
AiryAi, 213
AiryAiPrime, 213
AiryAiZero, 214
AiryBi, 214
AiryBiPrime, 214
AiryBiZero, 214
All, 147
AllTrue, 170
Alternatives, 189
And, 170
AngerJ, 214
AnglePath, 90
AngleVector, 91
AnyTrue, 170
Apart, 32
Append, 148
AppendTo, 148
Apply, 239
ApplyLevel, 239
ArcCos, 91
ArcCosh, 91
ArcCot, 92
ArcCoth, 92
ArcCsc, 92
ArcCsch, 92
ArcSec, 92
ArcSech, 92
ArcSin, 92
ArcSinh, 93
ArcTan, 93
ArcTanh, 93
Array, 148
ArrayDepth, 249

ArrayQ, 249
 Arrow, 100
 ArrowBox, 101
 Arrowheads, 101
 Association, 148
 AssociationQ, 149
 AtomQ, 239
 Attributes, 58
 Automatic, 101
 Axis, 196

 BarChart, 196
 BaseForm, 127
 Begin, 222
 BeginPackage, 222
 BernsteinBasis, 102
 BesselI, 215
 BesselJ, 215
 BesselJZero, 215
 BesselK, 215
 BesselY, 215
 BesselYZero, 215
 BezierCurve, 102
 BezierCurveBox, 102
 BezierFunction, 102
 Binarize, 121
 BinaryImageQ, 121
 BinaryRead, 254
 BinaryWrite, 255
 Binomial, 68
 BitLength, 136
 Black, 102
 Blank, 190
 BlankNullSequence, 190
 BlankSequence, 190
 Blend, 102
 Block, 222
 Blue, 103
 Blur, 121
 Boole, 38
 BooleanQ, 72
 Bottom, 197
 BoxMatrix, 121
 BrayCurtisDistance, 140
 Break, 76
 Byte, 256
 ByteCount, 240

 C, 85
 CanberraDistance, 140
 Cancel, 32
 Cases, 149
 Catenate, 149
 Ceiling, 136
 Center, 127

 CentralMoment, 149
 Character, 256
 CharacterRange, 226
 Characters, 226
 ChebyshevT, 216
 ChebyshevU, 216
 Check, 128
 ChessboardDistance, 140
 Chop, 181
 Circle, 103
 CircleBox, 104
 Clear, 48
 ClearAll, 49
 ClearAttributes, 58
 Close, 256
 Closing, 122
 ClusteringComponents, 149
 CMYKColor, 103
 Coefficient, 32
 CoefficientList, 33
 ColorCombine, 122
 ColorConvert, 122
 ColorData, 197
 ColorDataFunction, 197
 ColorDistance, 104
 Colorize, 122
 ColorNegate, 122
 ColorQuantize, 122
 ColorSeparate, 122
 CombinatoricaOld'BinarySearch, 240
 Compile, 71
 CompiledCodeBox, 71
 CompiledFunction, 71
 Complement, 149
 Complex, 39
 Complexes, 63
 ComplexInfinity, 38
 Composition, 97
 CompoundExpression, 76
 Compress, 256
 Condition, 190
 Conjugate, 39
 Constant, 59
 ConstantArray, 150
 ContainsOnly, 150
 Context, 223
 Contexts, 223
 Continue, 76
 CoprimeQ, 176
 CopyDirectory, 256
 CopyFile, 256
 Correlation, 150
 Cos, 93
 Cosh, 93
 CosineDistance, 140

Cot, 93
 Coth, 93
 Count, 150
 Covariance, 150
 CreateDirectory, 257
 Cross, 141
 Csc, 94
 Csch, 94
 CubeRoot, 39
 Cuboid, 117
 Cyan, 104

 D, 63
 DamerauLevenshteinDistance, 226
 Darker, 104
 DateDifference, 81
 DateList, 82
 DatePlus, 82
 DateString, 83
 Decrement, 49
 Default, 186
 DefaultValues, 49
 Definition, 49
 Degree, 94
 Delete, 151
 DeleteCases, 151
 DeleteDirectory, 257
 DeleteDuplicates, 151
 DeleteFile, 257
 Denominator, 33
 DensityPlot, 197
 Depth, 240
 Derivative, 64
 DesignMatrix, 141
 Det, 141
 DiagonalMatrix, 249
 DiamondMatrix, 122
 DiceDissimilarity, 68
 DigitCharacter, 227
 DigitCount, 136
 DigitQ, 227
 Dilation, 122
 Dimensions, 249
 DirectedInfinity, 39
 Directive, 105
 Directory, 257
 DirectoryName, 257
 DirectoryQ, 257
 DirectoryStack, 257
 DiscreteLimit, 64
 DisjointQ, 152
 Disk, 105
 DiskBox, 106
 DiskMatrix, 122
 Divide, 39
 DivideBy, 50
 Divisors, 176
 Do, 77
 DominantColors, 123
 Dot, 250
 DownValues, 51
 Drop, 152
 DSolve, 85

 E, 94
 EasterSunday, 83
 EdgeDetect, 123
 EdgeForm, 106
 EditDistance, 227
 Eigensystem, 141
 Eigenvalues, 141
 Eigenvectors, 142
 ElementData, 206
 End, 224
 EndOfFile, 258
 EndOfLine, 228
 EndOfString, 228
 EndPackage, 224
 Environment, 246
 Equal, 72
 Equivalent, 170
 Erf, 216
 Erfc, 216
 Erosion, 123
 EuclideanDistance, 142
 Evaluate, 86
 EvenQ, 176
 ExactNumberQ, 40
 Except, 191
 Exit, 86
 Exp, 94
 Expand, 34
 ExpandAll, 34
 ExpandDenominator, 34
 ExpandFileName, 258
 ExpIntegralE, 216
 ExpIntegralEi, 217
 Exponent, 35
 Export, 269
 ExportString, 270
 Expression, 258
 Extract, 152

 FaceForm, 106
 Factor, 35
 Factorial, 40
 FactorInteger, 177
 FactorTermsList, 35
 False, 171
 FetchURL, 270

Fibonacci, 68
 File, 258
 FileBaseName, 258
 FileByteCount, 258
 FileDate, 258
 FileExistsQ, 258
 FileExtension, 259
 FileFormat, 270
 FileHash, 259
 FileInformation, 259
 FileNameDepth, 259
 FileNameJoin, 259
 FileNameSplit, 259
 FilePrint, 260
 FileType, 260
 FilledCurve, 106
 FilledCurveBox, 106
 FilterRules, 186
 Find, 260
 FindClusters, 152
 FindFile, 260
 FindList, 260
 FindRoot, 65
 First, 153
 FirstPosition, 153
 FittedModel, 142
 FixedPoint, 77
 FixedPointList, 77
 Flat, 59
 Flatten, 240
 Floor, 136
 Fold, 153
 FoldList, 154
 FontColor, 106
 For, 78
 Format, 128
 FractionalPart, 177
 FreeQ, 241
 FresnelC, 217
 FresnelS, 217
 FromCharacterCode, 228
 FromDigits, 137
 Full, 198
 FullForm, 128
 Function, 97

 Gamma, 40
 Gather, 154
 GatherBy, 154
 GaussianFilter, 123
 GCD, 177
 GegenbauerC, 217
 General, 128
 Get, 261
 GetEnvironment, 246

 GoldenRatio, 94
 Graphics, 106
 Graphics3D, 118
 Graphics3DBox, 119
 GraphicsBox, 107
 Gray, 107
 GrayLevel, 108
 Greater, 73
 GreaterEqual, 73
 Green, 108
 Grid, 128
 GridBox, 129

 HammingDistance, 228
 HankelH1, 217
 HankelH2, 217
 HarmonicNumber, 41
 Hash, 181
 Haversine, 94
 Head, 241
 HermiteH, 217
 HexadecimalCharacter, 229
 Histogram, 198
 Hold, 86
 HoldAll, 59
 HoldAllComplete, 59
 HoldComplete, 87
 HoldFirst, 59
 HoldForm, 87
 HoldPattern, 191
 HoldRest, 59
 Hue, 108

 I, 41
 Identity, 98
 IdentityMatrix, 250
 If, 78
 Im, 41
 Image, 123
 ImageAdd, 123
 ImageAdjust, 123
 ImageAspectRatio, 123
 ImageBox, 123
 ImageChannels, 123
 ImageColorSpace, 123
 ImageConvolve, 124
 ImageData, 124
 ImageDimensions, 124
 ImageExport, 124
 ImageImport, 124
 ImageMultiply, 124
 ImagePartition, 124
 ImageQ, 124
 ImageReflect, 124
 ImageResize, 124

ImageRotate, 124
 ImageSubtract, 124
 ImageTake, 125
 ImageType, 125
 Implies, 171
 Import, 270
 ImportExport'RegisterExport, 271
 ImportExport'RegisterImport, 272
 ImportString, 271
 In, 87
 Increment, 51
 Indeterminate, 41
 Inequality, 73
 InexactNumberQ, 41
 Infinity, 42
 Infix, 129
 Information, 51
 Inner, 250
 InputForm, 129
 InputStream, 261
 Inset, 109
 InsetBox, 109
 Integer, 42
 IntegerDigits, 137, 182
 IntegerExponent, 177
 IntegerLength, 138
 IntegerQ, 42
 IntegerReverse, 138
 Integers, 65
 IntegerString, 138
 Integrate, 65
 Internal'RealValuedNumberQ, 185
 Internal'RealValuedNumericQ, 185
 Interrupt, 78
 IntersectingQ, 154
 Intersection, 154
 Inverse, 142
 InverseErf, 217
 InverseErfc, 218
 InverseHaversine, 95

 JaccardDissimilarity, 68
 JacobiP, 218
 Join, 155

 KelvinBei, 218
 KelvinBer, 218
 KelvinKei, 218
 KelvinKer, 219
 Keys, 155
 KnownUnitQ, 174
 Kurtosis, 155

 LABColor, 109
 LaguerreL, 219

 Large, 109
 Last, 155
 LCHColor, 109
 LCM, 177
 LeafCount, 155
 LeastSquares, 142
 Left, 129
 LegendreP, 219
 LegendreQ, 220
 Length, 156
 Less, 73
 LessEqual, 73
 LetterCharacter, 229
 LetterQ, 229
 Level, 156
 LevelQ, 156
 Lighter, 109
 LightRed, 109
 Limit, 66
 Line, 110
 Line3DBox, 119
 LinearModelFit, 142
 LinearSolve, 143
 LineBox, 110
 List, 157
 Listable, 59
 ListLinePlot, 199
 ListPlot, 199
 ListQ, 157
 LoadModule, 52
 Locked, 60
 Log, 95
 Log10, 95
 Log2, 95
 LogisticSigmoid, 95
 Longest, 191
 LowerCaseQ, 229
 LUVColor, 109

 MachineNumberQ, 42
 MachinePrecision, 182
 Magenta, 110
 MakeBoxes, 129
 ManhattanDistance, 143
 Manipulate, 173
 MantissaExponent, 177
 Map, 241
 MapIndexed, 241
 MapThread, 242
 MatchingDissimilarity, 69
 MatchQ, 191
 MathMLForm, 130
 MatrixExp, 143
 MatrixForm, 130
 MatrixPower, 144

- MatrixQ, 250
- MatrixRank, 144
- Max, 73
- MaxFilter, 125
- Maximize, 253
- Mean, 157
- Median, 157
- MedianFilter, 125
- Medium, 111
- MemberQ, 157
- Mesh, 199
- Message, 130
- MessageName, 130
- Messages, 52
- Min, 74
- MinFilter, 125
- MinimalPolynomial, 35
- Minimize, 253
- Minus, 42
- Missing, 36
- Mod, 178
- Module, 224
- MorphologicalComponents, 125
- Most, 158
- Multinomial, 69

- N, 183
- Names, 247
- Nearest, 158
- Needs, 262
- Negative, 74
- Nest, 78
- NestList, 78
- NestWhile, 79
- NextPrime, 178
- NHoldAll, 60
- NHoldFirst, 60
- NHoldRest, 60
- NonAssociative, 130
- None, 158
- NoneTrue, 171
- NonNegative, 74
- NonPositive, 74
- Norm, 144
- Normalize, 144
- Not, 171
- NotListQ, 158
- NotOptionQ, 186
- Null, 242
- NullSpace, 144
- Number, 262
- NumberForm, 130
- NumberQ, 42
- NumberString, 229
- Numerator, 36

- NumericQ, 184
- NValues, 52

- OddQ, 178
- Off, 130
- Offset, 111
- On, 131
- OneIdentity, 60
- OpenAppend, 262
- Opening, 125
- OpenRead, 262
- OpenWrite, 262
- Operate, 242
- Optional, 191
- OptionQ, 187
- Options, 187
- OptionsPattern, 192
- OptionValue, 187
- Or, 171
- Orange, 111
- Order, 243
- OrderedQ, 243
- Orderless, 60
- Out, 87
- Outer, 250
- OutputForm, 131
- OutputStream, 262
- OwnValues, 53

- PadLeft, 158
- PadRight, 159
- ParametricPlot, 200
- ParentDirectory, 263
- Part, 159
- Partition, 160
- Pattern, 192
- PatternsOrderedQ, 243
- PatternTest, 192
- Pause, 83
- Permutations, 160
- Pi, 95
- Pick, 161
- Piecewise, 43
- PieChart, 201
- PillowImageFilter, 125
- PixelValue, 125
- PixelValuePositions, 125
- Plot, 202
- Plot3D, 203
- Plus, 43
- Pochhammer, 43
- Point, 111
- Point3DBox, 119
- PointBox, 112
- PointSize, 112

PolarPlot, 204
 Polygon, 112
 Polygon3DBox, 119
 PolygonBox, 112
 PolynomialQ, 36
 Position, 161
 Positive, 74
 Postfix, 131
 Power, 44
 PowerExpand, 36
 PowerMod, 178
 Precedence, 131
 Precision, 184
 PreDecrement, 53
 Prefix, 132
 PreIncrement, 53
 Prepend, 161
 PrependTo, 161
 Prime, 178
 PrimePi, 178
 PrimePowerQ, 179
 PrimeQ, 179
 Print, 132
 Product, 44
 ProductLog, 220
 Protect, 61
 Protected, 61
 PseudoInverse, 144
 Purple, 112
 Put, 263
 PutAppend, 263

 QRDecomposition, 145
 Quantile, 162
 Quantity, 174
 QuantityMagnitude, 174
 QuantityQ, 174
 QuantityUnit, 175
 Quartiles, 162
 Quiet, 132
 Quit, 53
 Quotient, 179
 QuotientRemainder, 179

 Random, 208
 RandomChoice, 208
 RandomComplex, 209
 RandomImage, 125
 RandomInteger, 209
 RandomPrime, 179
 RandomReal, 209
 RandomSample, 210
 Range, 162
 RankedMax, 162
 RankedMin, 162

 Rational, 45
 Rationalize, 184
 Re, 45
 Read, 264
 ReadList, 264
 ReadProtected, 61
 Real, 45
 RealDigits, 184
 RealNumberQ, 45
 Reals, 66
 Reap, 162
 Record, 264
 Rectangle, 113
 RectangleBox, 114
 Red, 114
 RegularExpression, 229
 RegularPolygon, 114
 RegularPolygonBox, 115
 ReleaseHold, 88
 RemoveDiacritics, 229
 RenameDirectory, 264
 RenameFile, 265
 Repeated, 193
 RepeatedNull, 193
 Replace, 193
 ReplaceAll, 194
 ReplaceList, 194
 ReplacePart, 163
 ReplaceRepeated, 194
 ResetDirectory, 265
 Rest, 163
 Return, 79
 Reverse, 163
 RGBColor, 113
 Riffle, 164
 Right, 133
 RogersTanimotoDissimilarity, 69
 Root, 66
 RotateLeft, 164
 RotateRight, 164
 Round, 185
 Row, 133
 RowBox, 133
 RowReduce, 145
 RSolve, 212
 Rule, 195
 RuleDelayed, 195
 RussellRaoDissimilarity, 69

 SameQ, 74
 Scan, 243
 Sec, 96
 Sech, 96
 SeedRandom, 210
 Select, 164

Sequence, 88
 SequenceHold, 62
 SessionTime, 83
 Set, 54
 SetAttributes, 62
 SetDelayed, 54
 SetDirectory, 265
 SetFileDate, 265
 SetStreamPosition, 265
 Sharpen, 125
 Shortest, 195
 Sign, 45
 Simplify, 36
 Sin, 96
 SingularValueDecomposition, 145
 Sinh, 96
 Skewness, 165
 Skip, 266
 Slot, 98
 SlotSequence, 98
 Small, 115
 SokalSneathDissimilarity, 69
 Solve, 66
 Sort, 243
 SortBy, 244
 Sow, 165
 Span, 165
 Sphere, 119
 Sphere3DBox, 120
 SphericalHarmonicY, 220
 Split, 165
 SplitBy, 165
 Sqrt, 45
 SquaredEuclideanDistance, 145
 StandardDeviation, 166
 StandardForm, 133
 StartOfLine, 230
 StartOfString, 230
 StreamPosition, 266
 Streams, 266
 String, 235
 StringCases, 230
 StringContainsQ, 231
 StringDrop, 231
 StringExpression, 231
 StringForm, 133
 StringFreeQ, 232
 StringInsert, 232
 StringJoin, 233
 StringLength, 233
 StringMatchQ, 233
 StringPosition, 233
 StringQ, 233
 StringRepeat, 234
 StringReplace, 234
 StringRiffle, 234
 StringSplit, 235
 StringTake, 235
 StringToStream, 267
 StringTrim, 235
 StruveH, 220
 StruveL, 221
 Style, 133
 Subscript, 133
 SubscriptBox, 133
 SubsetQ, 166
 Subsets, 70
 Subsuperscript, 133
 SubsuperscriptBox, 133
 Subtract, 46
 SubtractFrom, 55
 SubValues, 55
 Sum, 46
 Superscript, 133
 SuperscriptBox, 133
 Switch, 79
 Symbol, 244
 SymbolName, 244
 SymbolQ, 244
 SymPyComparison, 75
 Syntax, 133
 System'Convert'B64Dump'B64Decode, 269
 System'Convert'B64Dump'B64Encode, 269
 System'Private'\$ContextPathStack, 223
 System'Private'\$ContextStack, 223
 System'Private'ManipulateParameter, 173
 Table, 166
 TableForm, 134
 TagSet, 55
 TagSetDelayed, 55
 Take, 166
 TakeLargest, 167
 TakeLargestBy, 167
 TakeSmallest, 167
 TakeSmallestBy, 167
 Tally, 168
 Tan, 96
 Tanh, 96
 TeXForm, 134
 Text, 115
 TextRecognize, 126
 Thick, 115
 Thickness, 115
 Thin, 115
 Thread, 244
 Threshold, 126
 Through, 244
 Times, 47
 TimesBy, 56

TimeUsed, 84
 Timing, 84
 Tiny, 115
 ToBoxes, 134
 ToCharCode, 236
 ToExpression, 236
 ToFileName, 267
 Together, 36
 ToLowerCase, 236
 Top, 204
 ToString, 236
 Total, 168
 ToUpperCase, 237
 Tr, 145
 Transliterate, 237
 Transpose, 251
 True, 171
 TrueQ, 75
 Tuples, 168

 Uncompress, 267
 Unequal, 75
 Unevaluated, 89
 Union, 168
 Unique, 224
 UnitConvert, 175
 UnitVector, 169
 Unprotect, 62
 UnsameQ, 75
 Unset, 56
 UpperCaseQ, 237
 UpSet, 56
 UpSetDelayed, 57
 UpTo, 37
 UpValues, 57

 ValueQ, 75
 Values, 169
 Variables, 37
 Variance, 169
 VectorAngle, 146
 VectorQ, 251
 Verbatim, 195

 WeberE, 221
 Which, 79
 While, 80
 White, 115
 Whitespace, 237
 WhitespaceCharacter, 237
 Word, 267
 WordBoundary, 237
 WordCharacter, 237
 WordCloud, 126
 Write, 267

 WriteString, 267

 XML'Parser'XMLGet, 252
 XML'Parser'XMLGetString, 252
 XML'PlaintextImport, 252
 XML'TagsImport, 252
 XML'XMLObjectImport, 252
 XMLElement, 252
 XMLObject, 252
 Xor, 172
 XYZColor, 116

 Yellow, 116
 YuleDissimilarity, 70

 Zeta, 221