STEPHEN WOLFRAM

ADDENDUM TO THE MATHEMATICA® BOOK 5TH EDITION

A guide to the new functions and features introduced in Mathematica 5.1

www.wolfram.com

Features New in *Mathematica* Version 5.1

Numerical computation

- New highly enhanced algorithms for high-precision `LinearSolve`.
- Internal vectorization of high-precision vector operations.
- New high-performance methods for `MatrixExp`.
- Support for sparse singular value decomposition.
- Support for `HessenbergDecomposition`.
- Typeset notation for `Transpose` and `Conjugate`.
- Numerical integration of discontinuous piecewise functions.
- Numerical integration over implicitly defined regions.
- Support for event detection in `NDSolve`.
- Additional convenience functions `Boole`, `Clip` and `Rescale`.
- Package for cluster analysis and dendrograms.
- Package for interactive exploration of differential equation systems.

Symbolic computation

- General vector derivatives, including gradient, Hessian and Jacobian.
- `Piecewise` construct for representing general piecewise functions.
- Simplification with piecewise and nested piecewise functions.
- Reduction of piecewise equations and inequalities, including quantifiers.
- `Limit`, `Series` and `D` support for general piecewise functions.
- Indefinite and definite integration of general piecewise functions.
- Support for solving piecewise ordinary differential equations.
- Symbolic multiple integration over regions defined by inequalities.
- Enhanced support for solving Abel and other differential equations.
- Support for linear differential equations with non-rational coefficients.
- Nonlinear partial differential equation solutions based on complete integrals.
- Support for equations with multiple moduli in `Reduce`.
- Additional methods for solving Diophantine equations.

Language and core system

- Full support for optimized string pattern matching.
- Integrated string and expression pattern language.
- General support for complement patterns with `Except`.
- String patterns integrated into all string operations.
- Generalized `StringCases` for string analysis.
- New functions `StringSplit`, `StringCount` and `StringReplaceList`.
- `RegularExpression` construct for compact string pattern notation.
- English-language dictionary package.
- Support for generalized `Tuples` and `Subsets`.
- Expression filtering function `Pick`.
- Package for benchmarking of computer systems.

Data handling and visualization

- `ArrayPlot` for flexible large-scale array visualization.
- Package for fully automated network and tree layout in 2D and 3D.
- Highly optimized import and export of binary data.
- Import and export of XLS spreadsheet files.
- Support for HDF5, MAT (v5), DIF, and PCX.
- Export of AVI movie files.
- Import from http and ftp URLs.
- Automated encoding and decoding of .gz files.
- Integrated TeX import and parsing in notebooks.
- Symbolic names for common colors such as `Red` and `Black`.

Database access

- *DatabaseLink* for universal cross-platform database connectivity.
- Bundled drivers for most common database systems.
- Integrated language interface for database discovery, query and updating.
- Graphical interface for database connection and exploration.
- Bundled SQL engine for creating custom databases.

GUI tools

- Integrated *GUIKit* for building standalone user interfaces.
- Platform-independent *Mathematica* language GUI specification.
- Over 100 types of controls and widgets.
- Automatic layout for complex dialog boxes.
- System for creating sequential wizard interfaces.
- Large library of sample GUI applications.

Web Services

- Transparent access to web services from within *Mathematica*.
- Support for SOAP and WSDL.
- Packages for search and lookup on Wolfram Research and other sites.

Table of Contents

+■ *a section new since Version 5.0*
~■ *a section substantially modified since Version 5.0*

1.8 Lists **1**
~■ Advanced Topic: Lists as Sets ~■ Grouping and Combining Elements of Lists

1.10 Input and Output in Notebooks **3**
~■ Entering Formulas

1.11 Files and External Operations **4**
■ Importing and Exporting Data +■ Generating and Importing TeX ~■ Exchanging Material with the Web

2.3 Patterns **6**
~■ Putting Constraints on Patterns

2.4 Manipulating Lists **7**
■ Constructing Lists ■ Manipulating Lists by Their Indices ■ Nested Lists

2.8 Strings and Characters **9**
~■ Operations on Strings +■ String Patterns +■ Advanced Topic: Regular Expressions

2.12 Files and Streams **21**
+■ Advanced Topic: Binary Files

3.2 Mathematical Functions **25**
+■ Piecewise Functions

3.3 Algebraic Manipulation **26**
+■ Advanced Topic: Logical and Piecewise Functions

3.4 Manipulating Equations and Inequalities **28**
■ Equations and Inequalities over Domains

3.5 Calculus **29**
~■ Differentiation +■ Integrals over Regions

3.7 Linear Algebra **31**
~■ Basic Matrix Operations ■ Advanced Matrix Operations

A.7 *Mathematica* Sessions **33**
~■ Network License Management

A.8 *Mathematica* File Organization **35**
■ *Mathematica* Distribution Files

A.10 Listing of Major Built-in *Mathematica* Objects....................36
■ Introduction ~■ Listing

A.12 Listing of Named Characters....................60
~■ Listing

Index....................65

1.8 Lists

~■ 1.8.8 Advanced Topic: Lists as Sets

•
•
•

`Union[`*list*$_1$`,` *list*$_2$`, … ]`	give a list of the distinct elements in the *list*$_i$
`Intersection[`*list*$_1$`,` *list*$_2$`, … ]`	give a list of the elements that are common to all the *list*$_i$
`Complement[`*universal*`,` *list*$_1$`, … ]`	give a list of the elements that are in *universal,* but not in any of the *list*$_i$
+ `Subsets[`*list*`]`	give a list of all subsets of the elements in *list*

Set theoretical functions.

•
•
•

This gives all the subsets of the list.

```
In[1]:= Subsets[{a, b, c}]
Out[1]= {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}
```

~■ 1.8.10 Grouping and Combining Elements of Lists

•
•
•

+	`Tuples[`*list*`, `*n*`]`	generate all possible n-tuples of elements from *list*
+	`Tuples[{`$list_1$`, `$list_2$`, … }]`	generate all tuples whose i^{th} element is from $list_i$

Finding possible tuples of elements in lists.

This gives all possible ways of picking two elements out of the list.

```
In[1]:= Tuples[{a, b}, 2]
Out[1]= {{a, a}, {a, b}, {b, a}, {b, b}}
```

This gives all possible ways of picking one element from each list.

```
In[2]:= Tuples[{{a, b}, {1, 2, 3}}]
Out[2]= {{a, 1}, {a, 2}, {a, 3}, {b, 1}, {b, 2}, {b, 3}}
```

1.10 Input and Output in Notebooks

~■ 1.10.4 Entering Formulas

•
•
•

short form	*long form*	
⋮sum⋮	\[Sum]	summation sign $\sum$
⋮prod⋮	\[Product]	product sign $\prod$
⋮int⋮	\[Integral]	integral sign $\int$
⋮dd⋮	\[DifferentialD]	special d for use in integrals
⋮pd⋮	\[PartialD]	partial derivative operator ∂
+ ⋮co⋮	\[Conjugate]	conjugate symbol *
+ ⋮tr⋮	\[Transpose]	transpose symbol ᵀ
+ ⋮ct⋮	\[ConjugateTranspose]	conjugate transpose symbol †
⋮[[⋮, ⋮]]⋮	\[LeftDoubleBracket], \[RightDoubleBracket]	part brackets

Some special characters used in entering formulas. Section 3.10 of the complete *Mathematica Book* gives a complete list.

1.11 Files and External Operations

1.11.3 Importing and Exporting Data

•
•
•

~	table formats	"CSV", "TSV", "XLS"
~	matrix formats	"HarwellBoeing", "MAT", "MTX"
+	specialized data formats	"DIF", "FITS", "HDF5", "MPS", "SDTS", etc.

Some common formats for tabular data.

•
•
•

+■ 1.11.6 Generating and Importing TeX

•
•
•

+ `ToExpression["`*input*`", TeXForm]`	convert TeX input to *Mathematica*

Converting TeX strings to *Mathematica*.

This converts a TeX string to *Mathematica*. Note the double backslashes needed in the string.

```
In[1]:= ToExpression["\\sqrt{x y}",
                     TeXForm]
```

Out[1]= $\sqrt{x\, y}$

•
•
•

~■ 1.11.7 Exchanging Material with the Web

•
•
•

+ `Import["http://`*url*`", … ]`	import data from a website
+ `Import["ftp://`*url*`", … ]`	import data from an FTP server

Importing data from web sources.

2.3 Patterns

~■ 2.3.5 Putting Constraints on Patterns

•
•
•

+	`Except[`*c*`]`	a pattern which matches any expression except *c*
+	`Except[`*c*`, `*patt*`]`	a pattern which matches *patt* but not *c*

Patterns with exceptions.

This gives all elements except 0.

```
In[1]:= Cases[{1, 0, 2, 0, 3}, Except[0]]
Out[1]= {1, 2, 3}
```

`Except` can take a pattern as an argument.

```
In[2]:= Cases[{a, b, 0, 1, 2, x, y}, Except[_Integer]]
Out[2]= {a, b, x, y}
```

This picks out integers that are not 0.

```
In[3]:= Cases[{a, b, 0, 1, 2, x, y}, Except[0, _Integer]]
Out[3]= {1, 2}
```

`Except[`*c*`]` is in a sense a very general pattern: it matches *anything* except *c*. In many situations you instead need to use `Except[`*c*`, `*patt*`]`, which starts from expressions matching *patt*, then excludes ones that match *c*.

2.4 Manipulating Lists

■ 2.4.1 Constructing Lists

•
•
•

`Map[`*f*`, `*list*`]`	apply *f* to each element of *list*
`MapIndexed[`*f*`, `*list*`]`	give *f*`[`*elem*`, {`*i*`}]` for the ith element
`Cases[`*list*`, `*form*`]`	give elements of *list* that match *form*
`Select[`*list*`, `*test*`]`	select elements for which *test*`[`*elem*`]` is `True`
+ `Pick[`*list*`, `*sel*`, `*form*`]`	pick out elements of *list* for which the corresponding elements of *sel* match *form*
list`[[{`i_1`, `i_2`, … }]]` or `Part[`*list*`, {`i_1`, `i_2`, … }]`	give a list of the specified parts of *list*

Constructing lists from other lists.

•
•
•

This explicitly gives numbered parts.

```
In[1]:= {a, b, c, d}[[{2, 1, 4}]]
Out[1]= {b, a, d}
```

This picks out elements indicated by a 1 in the second list.

```
In[2]:= Pick[{a, b, c, d}, {1, 0, 1, 1}, 1]
Out[2]= {a, c, d}
```

•
•
•

■ 2.4.2 Manipulating Lists by Their Indices

•
•
•

You can always reset one or more pieces of a list by doing an assignment like *m*`[[ … ]] =` *value*.

•
•
•

■ 2.4.3 Nested Lists

`{`*list*$_1$`,` *list*$_2$`, … }`	list of lists
`Table[`*expr*`, {`*i*`,` *m*`}, {`*j*`,` *n*`}, … ]`	$m \times n \times \ldots$ table of values of *expr*
`Array[`*f*`, {`*m*`,` *n*`, … }]`	$m \times n \times \ldots$ array of values *f*`[`*i*`,` *j*`, … ]`
`Normal[SparseArray[{{`i_1`,` j_1`,… } ->` v_1`, … }, {`*m*`,` *n*`, … }]]`	$m \times n \times \ldots$ array with element `{`i_s`,` j_s`,… }` being v_s
`Outer[`*f*`,` *list*$_1$`,` *list*$_2$`, … ]`	generalized outer product with elements combined using *f*
+ `Tuples[`*list*`, {`*m*`,` *n*`, … }]`	all possible $m \times n \times \ldots$ arrays of elements from *list*

Ways to construct nested lists.

•
•
•

2.8 Strings and Characters

~■ 2.8.2 Operations on Strings

•
•
•

This shows where both "abc" and "cd" appear. By default, overlaps are included.

```
In[1]:= StringPosition["abcdabcdcd", {"abc", "cd"}]
Out[1]= {{1, 3}, {3, 4}, {5, 7}, {7, 8}, {9, 10}}
```

This does not include overlaps.

```
In[2]:= StringPosition["abcdabcdcd", {"abc", "cd"},
                                    Overlaps -> False]
Out[2]= {{1, 3}, {5, 7}, {9, 10}}
```

+	StringCount[*s*, *sub*]	count the occurrences of *sub* in *s*
+	StringCount[*s*, {sub_1, sub_2, … }]	count occurrences of any of the sub_i
+	StringFreeQ[*s*, *sub*]	test whether *s* is free of *sub*
+	StringFreeQ[*s*, {sub_1, sub_2, … }]	test whether *s* is free of all the sub_i

Testing for substrings.

This counts occurrences of either substring, by default not including overlaps.

```
In[3]:= StringCount["abcdabcdcd", {"abc", "cd"}]
Out[3]= 3
```

~ StringReplace[*s*, *sb* -> *sbnew*]	replace *sb* by *sbnew* wherever it appears in *s*
~ StringReplace[*s*, {sb_1 -> $sbnew_1$, sb_2 -> $sbnew_2$, … }]	replace sb_i by the corresponding $sbnew_i$
+ StringReplace[*s*, *rules*, *n*]	do at most *n* replacements
+ StringReplaceList[*s*, *rules*]	give a list of the strings obtained by making each possible single replacement
+ StringReplaceList[*s*, *rules*, *n*]	give at most *n* results

Replacing substrings according to rules.

•
•
•

StringReplace scans a string from left to right, doing all the replacements it can, and then returning the resulting string. Sometimes, however, it is useful to see what all possible single replacements would give. You can get a list of all these results using StringReplaceList.

This gives a list of the results of replacing each of the a's.

```
In[4]:= StringReplaceList["aaaaa", "a" -> "X"]
Out[4]= {Xaaaa, aXaaa, aaXaa, aaaXa, aaaaX}
```

This shows the results of all possible single replacements.

```
In[5]:= StringReplaceList["abcde abacde",
                          {"abc" -> "X", "cde" -> "Y"}]
Out[5]= {Xde abacde, abY abacde, abcde abaY}
```

+ StringSplit[*s*]	split *s* into substrings delimited by whitespace
+ StringSplit[*s*, *del*]	split at delimiter *del*
+ StringSplit[*s*, {del_1, del_2, … }]	split at any of the del_i
+ StringSplit[*s*, *del*, *n*]	split into at most *n* substrings

Splitting strings.

This splits the string at every run of spaces.

```
In[6]:= StringSplit["a b::c d::e f  g"]
Out[6]= {a, b::c, d::e, f, g}
```

This splits at each "::".

```
In[7]:= StringSplit["a b::c d::e f  g", "::"]
Out[7]= {a b, c d, e f  g}
```

This splits at each colon or space.

```
In[8]:= StringSplit["a b::c d::e f  g", {":", " "}]
Out[8]= {a, b, , c, d, , e, f, , g}
```

+ `StringSplit[`*s*`, `*del*` -> `*rhs*`]`	insert *rhs* at the position of each delimiter
+ `StringSplit[`*s*`, {`del_1` -> `rhs_1`, `del_2` -> `rhs_2`, … }]`	insert rhs_i at the position of the corresponding del_i

Splitting strings with replacements for delimiters.

This inserts {x, y} at each :: delimiter.

```
In[9]:= StringSplit["a b::c d::e f  g", "::"->{x, y}]
Out[9]= {a b, {x, y}, c d, {x, y}, e f  g}
```

`Sort[{`s_1`, `s_2`, `s_3`, … }]`	sort a list of strings

Sorting strings.

Sort sorts strings into standard dictionary order.

```
In[10]:= Sort[{"cat", "fish", "catfish", "Cat"}]
Out[10]= {cat, Cat, catfish, fish}
```

+ ■ 2.8.3 String Patterns

An important feature of string manipulation functions like StringReplace is that they handle not only literal strings but also patterns for collections of strings.

This replaces b or c by X.

```
In[1]:= StringReplace["abcd abcd", "b" | "c" -> "X"]
Out[1]= aXXd aXXd
```

This replaces any character by u.

```
In[2]:= StringReplace["abcd abcd", _ -> "u"]
Out[2]= uuuuuuuuu
```

You can specify patterns for strings by using *string expressions* that contain ordinary strings mixed with *Mathematica* symbolic pattern objects.

+ s_1 ~~ s_2 ~~ … or StringExpression[s_1, s_2, …]	a sequence of strings and pattern objects

String expressions.

Here is a string expression that represents the string ab followed by any single character.

```
In[3]:= "ab" ~~ _
Out[3]= ab ~~ _
```

This makes a replacement for each occurrence of the string pattern.

```
In[4]:= StringReplace["abc abcb abdc", "ab" ~~ _ -> "X"]
Out[4]= X Xb Xc
```

~ StringMatchQ["*s*", *patt*]	test whether "*s*" matches *patt*
+ StringFreeQ["*s*", *patt*]	test whether "*s*" is free of substrings matching *patt*
+ StringCases["*s*", *patt*]	give a list of the substrings of "*s*" that match *patt*
+ StringCases["*s*", *lhs* -> *rhs*]	replace each case of *lhs* by *rhs*
~ StringPosition["*s*", *patt*]	give a list of the positions of substrings that match *patt*
+ StringCount["*s*", *patt*]	count how many substrings match *patt*
~ StringReplace["*s*", *lhs* -> *rhs*]	replace every substring that matches *lhs*
+ StringReplaceList["*s*", *lhs* -> *rhs*]	give a list of all ways of replacing *lhs*
+ StringSplit["*s*", *patt*]	split *s* at every substring that matches *patt*
+ StringSplit["*s*", *lhs* -> *rhs*]	split at *lhs*, inserting *rhs* in its place

Functions that support string patterns.

This gives all cases of the pattern that appear in the string.

```
In[5]:= StringCases["abc abcb abdc", "ab" ~~ _]
Out[5]= {abc, abc, abd}
```

This gives each character that appears after an "ab" string.

```
In[6]:= StringCases["abc abcb abdc", "ab" ~~ x_ -> x]
Out[6]= {c, c, d}
```

This gives all pairs of identical characters in the string.

```
In[7]:= StringCases["abbcbccaabbabccaa", x_ ~~ x_]
Out[7]= {bb, cc, aa, bb, cc, aa}
```

You can use all the standard *Mathematica* pattern objects in string patterns. Single blanks (_) always stand for single characters. Double blanks (__) stand for sequences of one or more characters.

Single blank (_) stands for any single character.

```
In[8]:= StringReplace[{"ab", "abc", "abcd"}, "b" ~~ _ -> "X"]
Out[8]= {ab, aX, aXd}
```

Double blank (__) stands for any sequence of one or more characters.

```
In[9]:= StringReplace[{"ab", "abc", "abcd"}, "b" ~~ __ -> "X"]
Out[9]= {ab, aX, aX}
```

Triple blank (___) stands for any sequence of zero or more characters.

```
In[10]:= StringReplace[{"ab", "abc", "abcd"}, "b" ~~ ___ -> "X"]
Out[10]= {aX, aX, aX}
```

Object	Description
"*string*"	a literal string of characters
_	any single character
__	any sequence of one or more characters
___	any sequence of zero or more characters
*x*_, *x*__, *x*___	substrings given the name *x*
x:*pattern*	pattern given the name *x*
pattern..	pattern repeated one or more times
pattern...	pattern repeated zero or more times
{$patt_1$, $patt_2$, … } or $patt_1$ | $patt_2$ | …	a pattern matching at least one of the $patt_i$
patt /; *cond*	a pattern for which *cond* evaluates to True
pattern ? *test*	a pattern for which *test* yields True for each character
Whitespace	a sequence of whitespace characters
NumberString	the characters of a number
charobj	an object representing a character class (see below)
RegularExpression["*regexp*"]	substring matching a regular expression

Objects in string patterns.

This splits at either a colon or semicolon.

```
In[11]:= StringSplit["a:b;c:d", ":" | ";"]
Out[11]= {a, b, c, d}
```

This finds all runs containing only a or b.

```
In[12]:= StringCases["aababbcccdbaa", ("a" | "b") ..]
Out[12]= {aababb, baa}
```

Alternatives can be given in lists in string patterns.

```
In[13]:= StringCases["aababbcccdbaa", {"a", "b"} ..]
Out[13]= {aababb, baa}
```

You can use standard *Mathematica* constructs such as Characters["$c_1 c_2$ … "] and CharacterRange["c_1", "c_2"] to generate lists of alternative characters to use in string patterns.

This gives a list of characters.

```
In[14]:= Characters["aeiou"]
Out[14]= {a, e, i, o, u}
```

This replaces the vowel characters.

```
In[15]:= StringReplace["abcdefghijklm", Characters["aeiou"]->"X"]
Out[15]= XbcdXfghXjklm
```

This gives characters in the range "A" through "H".

```
In[16]:= CharacterRange["A", "H"]
Out[16]= {A, B, C, D, E, F, G, H}
```

In addition to allowing explicit lists of characters, *Mathematica* provides symbolic specifications for several common classes of possible characters in string patterns.

+	{"c_1", "c_2", … }	any of the "c_i"
+	Characters["$c_1 c_2$ … "]	any of the "c_i"
	CharacterRange["c_1", "c_2"]	any character in the range "c_1" to "c_2"
+	DigitCharacter	digit 0–9
+	LetterCharacter	letter
+	WhitespaceCharacter	space, newline, tab or other whitespace character
+	WordCharacter	letter or digit
+	Except[*p*]	any character except ones matching *p*

Specifications for classes of characters.

This picks out the digit characters in a string.

```
In[17]:= StringCases["a6;b23c456;", DigitCharacter]
Out[17]= {6, 2, 3, 4, 5, 6}
```

This picks out all characters except digits.

```
In[18]:= StringCases["a6;b23c456;", Except[DigitCharacter]]
Out[18]= {a, ;, b, c, ;}
```

This picks out all runs of one or more digits.

```
In[19]:= StringCases["a6;b23c456", DigitCharacter..]
Out[19]= {6, 23, 456}
```

The results are strings.

```
In[20]:= InputForm[%]
Out[20]//InputForm= {"6", "23", "456"}
```

This converts the strings to numbers.

```
In[21]:= ToExpression[%] + 1
Out[21]= {7, 24, 457}
```

String patterns are often used as a way to extract structure from strings of textual data. Typically this works by having different parts of a string pattern match substrings that correspond to different parts of the structure.

This picks out each = followed by a number.

```
In[22]:= StringCases["a1=6.7, b2=8.87", "=" ~~ NumberString]
Out[22]= {=6.7, =8.87}
```

This gives the numbers alone.

```
In[23]:= StringCases["a1=6.7, b2=8.87", "=" ~~ x:NumberString -> x]
Out[23]= {6.7, 8.87}
```

This extracts “variables” and “values” from the string.

```
In[24]:= StringCases["a1=6.7, b2=8.87",
           v:WordCharacter.. ~~ "=" ~~ x:NumberString -> {v, x}]
Out[24]= {{a1, 6.7}, {b2, 8.87}}
```

`ToExpression` converts them to ordinary symbols and numbers.

```
In[25]:= ToExpression[%]^2
```

Out[25]= $\{\{a1^2, 44.89\}, \{b2^2, 78.6769\}\}$

In many situations, textual data may contain sequences of spaces, newlines or tabs that should be considered “whitespace”, and perhaps ignored. In *Mathematica*, the symbol `Whitespace` stands for any such sequence.

This removes all whitespace from the string.

```
In[26]:= StringReplace["aa b   cc  d", Whitespace -> ""]
Out[26]= aabccd
```

This replaces each sequence of spaces by a single comma.

```
In[27]:= StringReplace["aa b   cc  d", Whitespace -> ","]
Out[27]= aa,b,cc,d
```

String patterns normally apply to substrings that appear at any position in a given string. Sometimes, however, it is convenient to specify that patterns can apply only to substrings at particular positions. You can do this by including symbols such as `StartOfString` in your string patterns.

Construct	Meaning
+ `StartOfString`	start of the whole string
+ `EndOfString`	end of the whole string
+ `StartOfLine`	start of a line
+ `EndOfLine`	end of a line
+ `WordBoundary`	boundary between word characters and others
+ `Except[WordBoundary]`	anywhere except a word boundary

Constructs representing special positions in a string.

This replaces "a" wherever it appears in a string.

```
In[28]:= StringReplace[{"abc", "baca"}, "a" -> "XX"]
Out[28]= {XXbc, bXXcXX}
```

This replaces "a" only when it immediately follows the start of a string.

```
In[29]:= StringReplace[{"abc", "baca"},
                    StartOfString ~~ "a" -> "XX"]
Out[29]= {XXbc, baca}
```

This replaces all occurrences of the substring "the".

```
In[30]:= StringReplace["the others", "the" -> "XX"]
Out[30]= XX oXXrs
```

This replaces only occurrences that have a word boundary on both sides.

```
In[31]:= StringReplace["the others",
              WordBoundary ~~ "the" ~~ WordBoundary -> "XX"]
Out[31]= XX others
```

String patterns allow the same kind of `/;` and other conditions as ordinary *Mathematica* patterns.

This gives cases of unequal successive characters in the string.

```
In[32]:= StringCases["aaabbcaaaabaaa", x_ ~~ y_ /; x != y]
Out[32]= {ab, bc, ab}
```

When you give an object such as *x*`__` or *e*`..` in a string pattern, *Mathematica* normally assumes that you want this to match the longest possible sequence of characters. Sometimes, however, you may instead want to match the shortest possible sequence of characters. You can specify this using `ShortestMatch[`*p*`]`.

+	`LongestMatch[`*p*`]`	the longest consistent match for *p* (default)
+	`ShortestMatch[`*p*`]`	the shortest consistent match for *p*

Objects representing longest and shortest matches.

The string pattern by default matches the longest possible sequence of characters.

```
In[33]:= StringCases["-(a)--(bb)--(c)-", "(" ~~ __ ~~ ")"]
Out[33]= {(a)--(bb)--(c)}
```

`ShortestMatch` specifies that instead the shortest possible match should be found.

```
In[34]:= StringCases["-(a)--(bb)--(c)-",
                   ShortestMatch["(" ~~ __ ~~ ")"]]
Out[34]= {(a), (bb), (c)}
```

Mathematica by default treats characters such "X" and "x" as distinct. But by setting the option `IgnoreCase -> True` in string manipulation operations, you can tell *Mathematica* to treat all such upper- and lower-case letters as equivalent.

+	`IgnoreCase -> True`	treat upper- and lower-case letters as equivalent

Specifying case-independent string operations.

This replaces all occurrences of "the", independent of case.

```
In[35]:= StringReplace["The cat in the hat.", "the" -> "a",
                                              IgnoreCase -> True]

Out[35]= a cat in a hat.
```

In some string operations, one may have to specify whether to include overlaps between substrings. By default `StringCases` and `StringCount` do not include overlaps, but `StringPosition` does.

This picks out pairs of successive characters, by default omitting overlaps.

```
In[36]:= StringCases["abcdefg", _ ~~ _]
Out[36]= {ab, cd, ef}
```

This includes the overlaps.

```
In[37]:= StringCases["abcdefg", _ ~~ _, Overlaps -> True]
Out[37]= {ab, bc, cd, de, ef, fg}
```

`StringPosition` includes overlaps by default.

```
In[38]:= StringPosition["abcdefg", _ ~~ _]
Out[38]= {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {6, 7}}
```

+ `Overlaps -> All`	include all overlaps
+ `Overlaps -> True`	include at most one overlap beginning at each position
+ `Overlaps -> False`	exclude all overlaps

Options for handling overlaps in strings.

This yields only a single match.

```
In[39]:= StringCases["abcd", __, Overlaps -> False]
Out[39]= {abcd}
```

This yields a succession of overlapping matches.

```
In[40]:= StringCases["abcd", __, Overlaps -> True]
Out[40]= {abcd, bcd, cd, d}
```

This includes all possible overlapping matches.

```
In[41]:= StringCases["abcd", __, Overlaps -> All]
Out[41]= {abcd, abc, ab, a, bcd, bc, b, cd, c, d}
```

+ ■ 2.8.4 Advanced Topic: Regular Expressions

General *Mathematica* patterns provide a powerful way to do string manipulation. But particularly if you are familiar with specialized string manipulation languages, you may sometimes find it convenient to specify string patterns using *regular expression* notation. You can do this in *Mathematica* with `RegularExpression` objects.

+ `RegularExpression["`*regex*`"]`	a regular expression specified by "*regex*"

Using regular expression notation in *Mathematica*.

This replaces all occurrences of a or b.

```
In[1]:= StringReplace["abcd acbd", RegularExpression["[ab]"] -> "XX"]
Out[1]= XXXXcd XXcXXd
```

This specifies the same operation using a general *Mathematica* string pattern.

```
In[2]:= StringReplace["abcd acbd", "a" | "b" -> "XX"]
Out[2]= XXXXcd XXcXXd
```

You can mix regular expressions with general patterns.

```
In[3]:= StringReplace["abcd acbd",
          RegularExpression["[ab]"] ~~ _ -> "YY"]
Out[3]= YYcd YYYY
```

RegularExpression in *Mathematica* supports all standard regular expression constructs.

+	c	the literal character c
+	.	any character except newline
+	[c_1c_2...]	any of the characters c_i
+	[c_1-c_2]	any character in the range c_1–c_2
+	[^c_1c_2...]	any character except the c_i
+	p*	p repeated zero or more times
+	p+	p repeated one or more times
+	p?	zero or one occurrence of p
+	p{m,n}	p repeated between m and n times
+	p*?, p+?, p??	the shortest consistent strings that match
+	(p_1p_2...)	strings matching the sequence p_1, p_2, ...
+	p_1|p_2	strings matching p_1 or p_2

Basic constructs in *Mathematica* regular expressions.

This finds substrings that match the specified regular expression.

```
In[4]:= StringCases["abcddbbbacbbaa", RegularExpression["(a|bb)+"]]
Out[4]= {a, bb, a, bbaa}
```

This does the same operation with a general *Mathematica* string pattern.

```
In[5]:= StringCases["abcddbbbacbbaa", ("a" | "bb") ..]
Out[5]= {a, bb, a, bbaa}
```

There is a close correspondence between many regular expression constructs and basic general *Mathematica* string pattern constructs.

+	.	_ (strictly Except["\\n"])
+	[$c_1c_2\ldots$]	Characters["$c_1c_2\ldots$ "]
+	[c_1-c_2]	CharacterRange["c_1", "c_2"]
+	[^$c_1c_2\ldots$]	Except[Characters["$c_1c_2\ldots$ "]]
+	p*	p...
+	p+	p..
+	p?	p|""
+	p*?, p+?, p??	ShortestMatch[p...], ...
+	($p_1p_2\ldots$)	(p_1 ~~ p_2 ~~ ...)
+	p_1|p_2	p_1 | p_2

Correspondences between regular expression and general string pattern constructs.

Just as in general *Mathematica* string patterns, there are special notations in regular expressions for various common classes of characters. Note that you need to use double backslashes (\\) to enter most of these notations in *Mathematica* regular expression strings.

+	\\d	digit 0–9 (DigitCharacter)
+	\\D	non-digit (Except[DigitCharacter])
+	\\s	space, newline, tab or other whitespace character (WhitespaceCharacter)
+	\\S	non-whitespace character (Except[WhitespaceCharacter])
+	\\w	word character (letter, digit or _) (WordCharacter)
+	\\W	non-word character (Except[WordCharacter])
+	[[:*class*:]]	characters in a named class
+	[^[:*class*:]]	characters not in a named class

Regular expression notations for classes of characters.

This gives each occurrence of a followed by digit characters.

```
In[6]:= StringCases["a10b6a77a3a#",
          RegularExpression["a\\d+"]]

Out[6]= {a10, a77, a3}
```

Here is the same thing done with a general *Mathematica* string pattern.

```
In[7]:= StringCases["a10b6a77a3a#", "a" ~~ DigitCharacter ..]
Out[7]= {a10, a77, a3}
```

Mathematica supports the standard POSIX character classes `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, `xdigit`.

This finds runs of upper-case letters.

```
In[8]:= StringCases["AaBBccDDeefG",
            RegularExpression["[[:upper:]]+"]]
Out[8]= {A, BB, DD, G}
```

This does the same thing.

```
In[9]:= StringCases["AaBBccDDeefG", CharacterRange["A", "Z"] ..]
Out[9]= {A, BB, DD, G}
```

+	`^`	the beginning of the string (`StartOfString`)
+	`$`	the end of the string (`EndOfString`)
+	`\\b`	word boundary (`WordBoundary`)
+	`\\B`	anywhere except a word boundary (`Except[WordBoundary]`)

Regular expression notations for positions in strings.

In general *Mathematica* patterns, you can use constructs like `x_` and *x*:*patt* to give arbitrary names to objects that are matched. In regular expressions, there is a way to do something somewhat like this using numbering: the nth parenthesized pattern object (*p*) in a regular expression can be referred to as *n* within the body of the pattern, and $*n* outside it.

This finds pairs of identical letters that appear together.

```
In[10]:= StringCases["aaabcccabbaacba", RegularExpression["(.)\\1"]]
Out[10]= {aa, cc, bb, aa}
```

This does the same thing using a general *Mathematica* string pattern.

```
In[11]:= StringCases["aaabcccabbaacba", x_ ~~ x_]
Out[11]= {aa, cc, bb, aa}
```

The `$1` refers to the letter matched by `(.)`.

```
In[12]:= StringCases["aaabcccabbaacba",
           RegularExpression["(.)\\1"] -> "$1"]
Out[12]= {a, c, b, a}
```

Here is the *Mathematica* pattern version.

```
In[13]:= StringCases["aaabcccabbaacba", x_ ~~ x_ -> x]
Out[13]= {a, c, b, a}
```

2.12 Files and Streams

+■ 2.12.11 Advanced Topic: Binary Files

Functions like `Read` and `Write` handle ordinary printable text. But in dealing with external data files or devices it is sometimes necessary to go to a lower level, and work directly with raw binary data. You can do this using `BinaryRead` and `BinaryWrite`.

Function	Description
+ `BinaryRead[`*stream*`]`	read one byte
+ `BinaryRead[`*stream*`, `*type*`]`	read an object of the specified type
+ `BinaryRead[`*stream*`, {`$type_1$`, `$type_2$`, … }]`	read a list of objects
+ `BinaryWrite[`*stream*`, `*b*`]`	write one byte
+ `BinaryWrite[`*stream*`, {`b_1`, `b_2`, … }]`	write a sequence of bytes
+ `BinaryWrite[`*stream*`, "`*string*`"]`	write the characters in a string
+ `BinaryWrite[`*stream*`, `*x*`, `*type*`]`	write an object of the specified type
+ `BinaryWrite[`*stream*`, {`x_1`, `x_2`, … }, `*type*`]`	write a sequence of objects
+ `BinaryWrite[`*stream*`, {`x_1`, `x_2`, … }, {`$type_1$`, `$type_2$`, … }]`	write objects of different types

Reading and writing binary data.

+	"Byte"	8-bit unsigned integer
+	"Character8"	8-bit character
+	"Character16"	16-bit character
+	"Complex64"	IEEE single-precision complex number
+	"Complex128"	IEEE double-precision complex number
+	"Complex256"	IEEE quad-precision complex number
+	"Integer8"	8-bit signed integer
+	"Integer16"	16-bit signed integer
+	"Integer32"	32-bit signed integer
+	"Integer64"	64-bit signed integer
+	"Integer128"	128-bit signed integer
+	"Real32"	IEEE single-precision real number
+	"Real64"	IEEE double-precision real number
+	"Real128"	IEEE quad-precision real number
+	"TerminatedString"	null-terminated string of 8-bit characters
+	"UnsignedInteger8"	8-bit unsigned integer
+	"UnsignedInteger16"	16-bit unsigned integer
+	"UnsignedInteger32"	32-bit unsigned integer
+	"UnsignedInteger64"	64-bit unsigned integer
+	"UnsignedInteger128"	128-bit unsigned integer

Types supported in `BinaryRead` and `BinaryWrite`.

This writes a sequence of bytes to a file.

```
In[1]:= BinaryWrite["tmp", {97,98,99,100,101}]
Out[1]= tmp
```

`BinaryWrite` automatically opens a stream for the file. This closes it.

```
In[2]:= Close["tmp"];
```

This reads the first byte from the file, returning it as an integer.

```
In[3]:= BinaryRead["tmp"]
Out[3]= 97
```

This reads the second 8 bits in the file as character.

```
In[4]:= BinaryRead["tmp", "Character8"]
Out[4]= b
```

This reads the next 32 bits as a 32-bit integer.

```
In[5]:= BinaryRead["tmp", "Integer32"]
Out[5]= EndOfFile
```

Like `Read` and `Write`, `BinaryRead` and `BinaryWrite` work with streams. But if you give a file name, they automatically open the specified file as a stream. To create a stream directly you can use `OpenRead` or `OpenWrite`. On some computer systems, the option setting `BinaryFormat->True` is required for any stream to be used with `BinaryRead` and `BinaryWrite`, in order to prevent possible corruption from such issues as newline translation.

In using *Mathematica* you are normally completely insulated from the raw representation of data inside your computer. But with `BinaryRead` and `BinaryWrite` this is no longer so. One of the subtleties that then arises is that different computers may take the bytes that make up numbers to be in different orders, as specified by their setting for `$ByteOrdering`.

This write a 32-bit integer to a file.

```
In[6]:= BinaryWrite["tmp2", 45671, "Integer32"]
Out[6]= tmp2
```

This closes the file.

```
In[7]:= Close["tmp2"];
```

This reads the integer back, but assuming an opposite byte ordering.

```
In[8]:= BinaryRead["tmp2", "Integer32",
          ByteOrdering -> -$ByteOrdering]
Out[8]= 1739718656
```

+ `BinaryReadList["`*file*`"]`	read all the bytes in a file
+ `BinaryReadList["`*file*`", `*type*`]`	read all the data, treating it as objects of a certain type
+ `BinaryReadList["`*file*`", {`*type*$_1$`, `*type*$_2$`, … }]`	treat the data as objects of a sequence of types
+ `BinaryReadList["`*file*`", `*types*`, `*n*`]`	read only the first *n* objects

Reading complete binary files.

This writes out a 128-bit real number.

```
In[9]:= BinaryWrite["tmp3", 5.67891, "Real128"]
Out[9]= tmp3
```

This reads back the bytes in the number.

```
In[10]:= BinaryReadList["tmp3", "Byte"]
Out[10]= {0, 0, 0, 0, 0, 0, 0, 224,
          89, 187, 237, 66, 115, 107, 1, 64}
```

This reads back the bytes as a sequence of 32-bit real numbers.

```
In[11]:= BinaryReadList["tmp3", "Real32"]
```

Out[11]= $\{0., -3.68935\times10^{19}, 118.866, 2.02218\}$

This treats the data as pairs containing a byte and a 32-bit real.

```
In[12]:= BinaryReadList["tmp3", {"Byte", "Real32"}]
Out[12]= {{0, 0.}, {0, -0.00332451},
          {237, 4.32454×10^-38}, {64, EndOfFile}}
```

BinaryRead and BinaryWrite allow complete flexibility in reading and writing raw binary data. But in many practical applications one instead wants to work only with particular predefined formats. You can do this using Import and Export.

In addition to many complex formats, Import and Export support files containing sequences of identical data elements, of the same types as in BinaryRead and BinaryWrite. They also support the "Bit" format, consisting of individual binary bits, represented as 0 or 1.

3.2 Mathematical Functions

3.2.3 Piecewise Functions

`Boole[`*expr*`]`	give 1 if *expr* is `True`, and 0 if it is `False`

Turning conditions into numbers.

`Boole[`*expr*`]` is a basic function that turns `True` and `False` into `1` and `0`. It is sometimes known as the **characteristic function** or **indicator function**.

This gives the area of a unit disk.

```
In[1]:= Integrate[Boole[x^2 + y^2 <= 1], {x, -1, 1}, {y, -1, 1}]
Out[1]= π
```

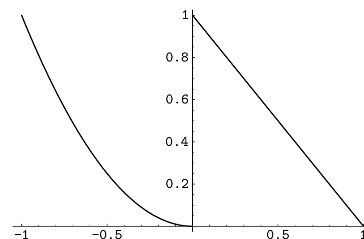
`Piecewise[{{`val_1`, `$cond_1$`}, {`val_2`, `$cond_2$`}, … }]`	give the first val_i for which $cond_i$ is `True`
`Piecewise[{{`val_1`, `$cond_1$`}, … }, `*val*`]`	give *val* if all $cond_i$ are `False`

Piecewise functions.

It is often convenient to have functions with different forms in different regions. You can do this using `Piecewise`.

This plots a piecewise function.

```
In[2]:= Plot[Piecewise[{{x^2, x < 0}, {1-x, x > 0}}], {x, -1, 1}]
```



Piecewise functions appear in systems where there is discrete switching between different domains. They are also at the core of many computational methods, including splines and finite elements. Special cases include such functions as `Abs`, `UnitStep`, `Clip`, `Sign`, `Floor` and `Max`. *Mathematica* handles piecewise functions in both symbolic and numerical situations.

3.3 Algebraic Manipulation

+■ 3.3.11 Advanced Topic: Logical and Piecewise Functions

Nested logical and piecewise functions can be expanded out much like nested arithmetic functions. You can do this using `LogicalExpand` and `PiecewiseExpand`.

+	`LogicalExpand[`*expr*`]`	expand out logical functions in *expr*
+	`PiecewiseExpand[`*expr*`]`	expand out piecewise functions in *expr*
+	`PiecewiseExpand[`*expr*`,` *assum*`]`	expand out with the specified assumptions

Expanding out logical and piecewise functions.

`LogicalExpand` puts logical expressions into a standard *disjunctive normal form* (DNF), consisting of an OR of ANDs.

By default, *Mathematica* leaves this expression unchanged.

```
In[1]:= (a || b) && c
Out[1]= (a || b) && c
```

`LogicalExpand` expands this into an OR of ANDs.

```
In[2]:= LogicalExpand[%]
Out[2]= (a && c) || (b && c)
```

`LogicalExpand` works on all logical functions, always converting them into a standard OR of ANDs form. Sometimes the results are inevitably quite large.

`Xor` can be expressed as an OR of ANDs.

```
In[3]:= LogicalExpand[Xor[a, b, c]]
Out[3]= (a && b && c) || (a && ! b && ! c) ||
         (b && ! a && ! c) || (c && ! a && ! b)
```

Any collection of nested conditionals can always in effect be flattened into a *piecewise normal form* consisting of a single `Piecewise` object. You can do this in *Mathematica* using `PiecewiseExpand`.

By default, *Mathematica* leaves this expression unchanged.

```
In[4]:= If[x > 0, If[x < 1, a, b], c]
Out[4]= If[x > 0, If[x < 1, a, b], c]
```

`PiecewiseExpand` flattens it into a single `Piecewise` object.

```
In[5]:= PiecewiseExpand[%]
```

$$Out[5]= \begin{cases} a & 0 < x < 1 \\ b & x \geq 1 \\ c & \text{True} \end{cases}$$

Functions like `Max` and `Abs`, as well as `Clip` and `UnitStep`, implicitly involve conditionals, and combinations of them can again be reduced to a single `Piecewise` object using `PiecewiseExpand`.

This gives a result as a single Piecewise object.

```
In[6]:= PiecewiseExpand[Max[Min[a, b], c]]
```

$$\text{Out[6]= } \begin{cases} a & a - c > 0 \,\&\&\, a - b \le 0 \\ b & a - b > 0 \,\&\&\, b - c > 0 \\ c & \text{True} \end{cases}$$

With x assumed real, this can also be written as a Piecewise object.

```
In[7]:= PiecewiseExpand[Abs[x], x \[Element] Reals]
```

$$\text{Out[7]= } \begin{cases} -x & x < 0 \\ x & \text{True} \end{cases}$$

Functions like Floor, Mod and FractionalPart can also be expressed in terms of Piecewise objects, though in principle they can involve an infinite number of cases.

Without a bound on x, this would yield an infinite number of cases.

```
In[8]:= PiecewiseExpand[Floor[x^2], 0 < x < 2]
```

$$\text{Out[8]= } \begin{cases} 1 & 1 \le x < \sqrt{2} \\ 2 & \sqrt{2} \le x < \sqrt{3} \\ 3 & x \ge \sqrt{3} \end{cases}$$

Mathematica by default limits the number of cases that *Mathematica* will explicitly generate in the expansion of any single piecewise function such as Floor at any stage in a computation. You can change this limit by resetting the value of $MaxPiecewiseCases.

3.4 Manipulating Equations and Inequalities

3.4.9 Equations and Inequalities over Domains

•
•
•

`Reduce` can also handle equations that involve several different moduli.

Here is an equation involving two different moduli.

```
In[1]:= Reduce[Mod[2x + 1, 5] == Mod[x, 7] && 0 < x < 50, x]

Out[1]= x == 4 || x == 7 || x == 15 ||
        x == 23 || x == 31 || x == 39 || x == 42
```

•
•
•

3.5 Calculus

~■ 3.5.1 Differentiation

•
•
•

+ D[f, {{x_1, x_2, … }}]	the gradient of a scalar function f $(\partial f/\partial x_1, \partial f/\partial x_2, \ldots)$
+ D[f, {{x_1, x_2, … }, 2}]	the Hessian matrix for f
+ D[f, {{x_1, x_2, … }, n}]	the nth order Taylor series coefficient
+ D[{f_1, f_2, … }, {{x_1, x_2, … }}]	the Jacobian for a vector function f

Vector derivatives.

This gives the gradient of the function $x^2 + y^2$.

```
In[1]:= D[x^2 + y^2, {{x, y}}]
Out[1]= {2 x, 2 y}
```

This gives the Hessian.

```
In[2]:= D[x^2 + y^2, {{x, y}, 2}]
Out[2]= {{2, 0}, {0, 2}}
```

This gives the Jacobian for a vector function.

```
In[3]:= D[{x^2 + y^2, x y}, {{x, y}}]
Out[3]= {{2 x, 2 y}, {y, x}}
```

+■ 3.5.9 Integrals over Regions

This does an integral over the unit circle.

```
In[1]:= Integrate[If[x^2 + y^2 < 1, 1, 0], {x, -1, 1}, {y, -1, 1}]
Out[1]= π
```

Here is an equivalent form.

```
In[2]:= Integrate[Boole[x^2 + y^2 < 1], {x, -1, 1}, {y, -1, 1}]
Out[2]= π
```

Even though an integral may be straightforward over a simple rectangular region, it can be significantly more complicated even over a circular region.

This gives a Bessel function.

```
In[3]:= Integrate[Exp[x] Boole[x^2 + y^2 < 1],
                  {x, -1, 1}, {y, -1, 1}]
Out[3]= 2 π BesselI[1, 1]
```

+ `Integrate[`f `Boole[`$cond$`], {`x, x_{min}, x_{max}`}, {`y, y_{min}, y_{max}`}]`	integrate f over the region where $cond$ is `True`

Integrals over regions.

Particularly if there are parameters inside the conditions that define regions, the results for integrals over regions may break into several cases.

This gives a piecewise function of a.

```
In[4]:= Integrate[Boole[a x < y], {x, 0, 1}, {y, 0, 1}]
```

$$Out[4]= \begin{cases} 1 & a \le 0 \\ \frac{2-a}{2} & 0 < a \le 1 \\ \frac{1}{2a} & \text{True} \end{cases}$$

With two parameters even this breaks into quite a few cases.

```
In[5]:= Integrate[Boole[a x < b], {x, 0, 1}]
```

$$Out[5]= \begin{cases} 1 & (a > 0 \,\&\&\, a - b \le 0) \,||\, (a \le 0 \,\&\&\, b > 0) \\ \frac{a-b}{a} & a \le 0 \,\&\&\, a - b < 0 \,\&\&\, b \le 0 \\ \frac{b}{a} & a > 0 \,\&\&\, b > 0 \,\&\&\, a - b > 0 \end{cases}$$

This involves intersecting a circle with a square.

```
In[6]:= Integrate[Boole[x^2 + y^2 < a], {x, 0, 1}, {y, 0, 1}]
```

$$Out[6]= \begin{cases} 1 & a \ge 2 \\ \frac{a\pi}{4} & 0 < a \le 1 \\ \frac{1}{2}\left(2\sqrt{-1+a} + a\,\text{ArcCot}\left[\sqrt{-1+a}\right] - a\,\text{ArcTan}\left[\sqrt{-1+a}\right]\right) & 1 < a < 2 \end{cases}$$

The region can have an infinite number of components.

```
In[7]:= Integrate[Boole[Sin[x] > 1/2] Exp[-x],
                  {x, 0, Infinity}]
```

$$Out[7]= \frac{e^{7\pi/6}}{1 + e^{2\pi/3} + e^{4\pi/3}}$$

`Integrate` effectively does Lebesgue integration.

```
In[8]:= Integrate[Boole[Element[x, Rationals]], {x, 0, 1}]
```

```
Out[8]= 0
```

3.7 Linear Algebra

~■ 3.7.7 Basic Matrix Operations

Transpose[*m*]	transpose m^{T}
+ ConjugateTranspose[*m*]	conjugate transpose $m^{\dagger}$ (Hermitian conjugate)
Inverse[*m*]	matrix inverse
Det[*m*]	determinant
Minors[*m*]	matrix of minors
Minors[*m*, *k*]	k^{th} minors
Tr[*m*]	trace
CharacteristicPolynomial[*m*, *x*]	characteristic polynomial

Some basic matrix operations.

•
•
•

■ 3.7.10 Advanced Matrix Operations

•
•
•

JordanDecomposition[*m*]	the Jordan decomposition
SchurDecomposition[*m*]	the Schur decomposition
SchurDecomposition[{*m*, *a*}]	the generalized Schur decomposition
+ HessenbergDecomposition[*m*]	the Hessenberg decomposition

Functions related to eigenvalue problems.

⋮

Numerically more stable is the *Schur decomposition*, which writes any matrix **m** in the form $\mathbf{qtq}^*$, where **q** is an orthonormal matrix, and **t** is block upper-triangular. Also related is the *Hessenberg decomposition*, which writes a matrix **m** in the form $\mathbf{php}^*$, where **p** is an orthonormal matrix, and **h** can have nonzero elements down to one element below the leading diagonal.

A.7 *Mathematica* Sessions

~■ A.7.6 Network License Management

•
•
•

- Type `./mathlm` directly on the Unix command line
- Add a line to start `mathlm` in your central system startup script

Ways to start the network license manager on Macintosh and Unix systems.

•
•
•

	`-logfile` *file*	write a log of license server actions to *file*
+	`-loglevel` *n*	how verbose to make log entries (1 to 4)
+	`-logformat` *string*	use a log format specified by *string*
+	`-language` *name*	language to use for messages (default `English`)
	`-pwfile` *file*	use the specified `mathpass` file (default `./mathpass`)
	`-timeout` *n*	suspend authorization on stopped *Mathematica* jobs after *n* hours
	`-restrict` *file*	use the specified restriction file
+	`-mathid`	print the MathID for the license server, and exit
+	`-foreground`	run `mathlm` in the foreground, logging to `stdout`
	`-install`	install `mathlm` as a Windows service (Microsoft Windows only)
	`-uninstall`	uninstall `mathlm` as a Windows service (Microsoft Windows only)

Some command-line options for `mathlm`.

For more detailed information on `mathlm`, see the *Mathematica Products System Administration Guide*.

	`monitorlm`	a program to monitor network license activity
+	`monitorlm` *name*	monitor activity for license server *name*

Monitoring network license activity.

If `monitorlm` is run in an environment where a web browser can be started, it will automatically generate HTML output in the browser. Otherwise it will generate plain text.

+	`-file` *file*	write output to a file
~	`-format` *spec*	use the specified format (`text`, `html` or `cgi`)
	`-template` *file*	use the specified file as a template for the output

Some command-line options for `monitorlm`.

A.8 *Mathematica* File Organization

■ A.8.1 *Mathematica* Distribution Files

•
•
•

```
C:\Program␣Files\Wolfram␣Research\Mathematica\5.1
```
Windows

```
/Applications/Mathematica 5.1.app
```
Macintosh

```
/usr/local/Wolfram/Mathematica/5.1
```
Unix

Default locations for the *Mathematica* installation directory.

•
•
•

A.10 Listing of Major Built-in *Mathematica* Objects

■ Introduction

⋮

+■	object or feature completely new since Version 5.0
+■	object or feature whose functionality was extensively changed since Version 5.0

New and modified objects and features in the listing.

⋮

+■ ArrayPlot

ArrayPlot[*array*] generates a plot in which the values in an array are shown in a discrete array of squares.

ArrayPlot[*array*] arranges successive rows of *array* down the page, and successive columns across, just as a table or grid would normally be formatted. ■ If *array* contains 0's and 1's, the 1's will appear as black squares and the 0's as white squares. ■ ArrayPlot by default generates grayscale output, in which zero values are shown white, and the maximum positive or negative value is shown black. ■ ArrayPlot has the same options as Graphics, with the following additions and changes:

AspectRatio	Automatic	ratio of height to width
ColorFunction	Automatic	how each cell should be colored
ColorFunctionScaling	False	whether to scale the argument to ColorFunction
ColorRules	Automatic	rules for determining colors from values
DataRange	All	the range of x and y values to assume
Frame	Automatic	whether to draw a frame around the plot
FrameTicks	None	what ticks to include on the frame
MaxPlotPoints	Infinity	the maximum number of points to include
Mesh	False	whether to draw a mesh
MeshStyle	GrayLevel[GoldenRatio-1]	the style to use for a mesh
PixelConstrained	False	whether to constrain cells to align with pixels
PlotRange	All	the range of values to plot

■ ColorRules -> {0->Red, 1->Blue, _->Black} specifies that cells with value 0 should be red, those with value 1 should be blue, and all others should be black. ■ The rules given by ColorRules are applied to the value a_{ij} of each cell. ■ If none of the rules in ColorRules applies, then ColorFunction is used to determine the color. ■ With the default setting ColorRules -> Automatic, an explicit setting ColorFunction -> *f* is used instead of ColorRules. ■ With the default setting ColorFunctionScaling -> False, each value a_{ij} is supplied as the argument to any function given for ColorFunction. ■ With ColorFunctionScaling -> True, the values are scaled to lie between 0 and 1. ■ With the setting FrameTicks -> Automatic, ticks are placed at round integers, typically multiples of 5 or 10. ■ With the setting FrameTicks -> All, ticks are also placed at the minimum and maximum i and j. ■ PlotRange -> {a_{min}, a_{max}} specifies that only those a_{ij} between a_{min} and a_{max} should be shown. ■ PlotRange -> {{i_{min}, i_{max}}, {j_{min}, j_{max}}} shows only elements with i and j in the specified ranges. The top-left element has $i = 1$, $j = 1$. i increases down the page; j increases to the right. ■ PlotRange -> {*ispec*, *jspec*, *aspec*} shows only elements in the specified ranges of i, j and value. ■ With the default setting for ColorFunction, PlotRange->{a_{min}, a_{max}} specifies that values from a_{min} to a_{max} should be shown with grayscales varying from white to black. ■ ColorFunction -> (GrayLevel[If[#==0, 1, 0]]&) generates a plot in which all nonzero values are shown as black. ■ Mesh -> True draws mesh lines between each cell in the array. ■ With the default setting Frame -> Automatic, a frame is drawn only when Mesh -> False. ■ With the default setting DataRange -> All, the array element a_{ij} will be taken to cover a unit square centered at coordinate position $x = j - 1/2$, $y = i_{max} - i + 1/2$. ■ ArrayPlot returns Graphics[Raster[*data*]]. ■ With PixelConstrained -> True, ArrayPlot generates a Raster with an absolute size that aligns cells with pixels, so that each cell is an integer number of pixels across, or each pixel is an integer number of cells across. The cells are each taken to be as large as possible given the ImageSize setting specified. ■ See also: ListDensityPlot, Raster, ListPlot3D, SparseArray, CellularAutomaton. ■ *New in Version 5.1.*

+■ BinaryRead

`BinaryRead[`*stream*`]` reads one byte of raw binary data from an input stream, and returns an integer from 0 to 255.

`BinaryRead[`*stream*`, `*type*`]` reads an object of the specified type.

`BinaryRead[`*stream*`, {`$type_1$`, `$type_2$`, … }]` reads a sequence of objects of the specified types.

Possible types to read are:

`"Byte"`	8-bit unsigned integer
`"Character8"`	8-bit character
`"Character16"`	16-bit character
`"Complex64"`	IEEE single-precision complex number
`"Complex128"`	IEEE double-precision complex number
`"Complex256"`	IEEE quad-precision complex number
`"Integer8"`	8-bit signed integer
`"Integer16"`	16-bit signed integer
`"Integer32"`	32-bit signed integer
`"Integer64"`	64-bit signed integer
`"Integer128"`	128-bit signed integer
`"Real32"`	IEEE single-precision real number
`"Real64"`	IEEE double-precision real number
`"Real128"`	IEEE quad-precision real number
`"TerminatedString"`	null-terminated string of 8-bit characters
`"UnsignedInteger8"`	8-bit unsigned integer
`"UnsignedInteger16"`	16-bit unsigned integer
`"UnsignedInteger32"`	32-bit unsigned integer
`"UnsignedInteger64"`	64-bit unsigned integer
`"UnsignedInteger128"`	128-bit unsigned integer

■ The first argument to `BinaryRead` can be `InputStream["`*name*`", `*n*`]`, or simply `"`*name*`"` if there is only one open input stream with the specified name. ■ You can open a file or pipe to get an `InputStream` object using `OpenRead`. ■ Streams for use with `BinaryRead` should be opened with `BinaryFormat->True`. ■ There is always a "current point" maintained for any stream. When you read an object from a stream, the current point is left after the input you read. Successive calls to `BinaryRead` can therefore be used to read successive objects in a stream such as a file. ■ `BinaryRead` returns `EndOfFile` if you are at the end of the file. ■ `BinaryRead` returns `Infinity` for IEEE "infinity", and `Indeterminate` for IEEE "not-a-number". ■ The following options can be given:

`ByteOrdering`	`$ByteOrdering`	what byte ordering to use
`Path`	`$Path`	the path to search for files to be opened

■ See also: `BinaryReadList`, `BinaryWrite`, `Read`, `Import`, `ImportString`. ■ *New in Version 5.1.*

+■ BinaryReadList

BinaryReadList["*file*"] reads all remaining bytes from a file, and returns them as a list of integers from 0 to 255.

BinaryReadList["*file*", *type*] reads objects of the specified type from a file, until the end of the file is reached. The list of objects read is returned.

BinaryReadList["*file*", {$type_1$, $type_2$, … }] reads objects with a sequence of types, until the end of the file is reached.

BinaryReadList["*file*", *types*, *n*] reads only the first *n* objects of the specified types.

BinaryReadList supports the same types and options as BinaryRead. ■ If *file* is not already open for reading, BinaryReadList opens it, then closes it when it is finished. If the file is already open, BinaryReadList does not close it at the end. ■ BinaryReadList["*file*", {$type_1$, … }] reads the sequence of $type_i$ in order. If the end of file is reached while part way through this sequence, EndOfFile is returned in place of elements in the sequence that have not yet been read. ■ BinaryReadList["!*command*", …] reads from a pipe. ■ BinaryReadList[*stream*] reads from an open input stream, as returned by OpenRead with BinaryFormat->True. ■ See notes for BinaryRead. ■ See also: Import, BinaryRead, Export. ■ *New in Version 5.1.*

+■ BinaryWrite

BinaryWrite[*channel*, *b*] writes a byte of data, specified as an integer from 0 to 255.

BinaryWrite[*channel*, {b_1, b_2, … }] writes a sequence of bytes.

BinaryWrite[*channel*, "*string*"] writes the raw sequence of characters in a string.

BinaryWrite[*channel*, *x*, *type*] writes an object of the specified type.

BinaryWrite[*channel*, {x_1, x_2, … }, *type*] writes a sequence of objects of the specified type.

BinaryWrite[*channel*, {x_1, x_2, … }, {$type_1$, $type_2$, … }] writes a sequence of objects with a sequence of types.

BinaryWrite supports the same types as BinaryRead. ■ The output channel used by BinaryWrite can be a single file or pipe, or list of them, each specified by a string giving their name, or by an OutputStream object that has been opened with BinaryFormat->True. ■ If any of the specified files or pipes are not already open, BinaryWrite calls OpenWrite to open them. ■ BinaryWrite does not close files and pipes after it finishes writing to them. ■ When a list of types is given, the list is effectively repeated as many times as necessary. ■ The following option can be given:

ByteOrdering	$ByteOrdering	what byte ordering to use

■ BinaryWrite[*channel*, "*string*"] uses type "Character8", so all characters in "*string*" should have character code in the range 0–255. ■ BinaryWrite returns $Failed if it encounters a data element that cannot match the type specified. ■ See also: BinaryRead, WriteString, Export. ■ *New in Version 5.1.*

+■ Black

Black represents the color black in graphics or style specifications.

Black is equivalent to GrayLevel[0]. ■ See also: GrayLevel, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ Blue

Blue represents the color blue in graphics or style specifications.

Blue is equivalent to RGBColor[0, 0, 1]. ■ See also: Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ Boole

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

Boole[*expr*] remains unchanged if *expr* is neither True nor False. ■ Boole[*expr*] is effectively equivalent to If[*expr*, 1, 0]. ■ Integrate[*f* Boole[*pred*], …] can be used to integrate *f* over the region in which *pred* is True. ■ See also: If, Piecewise, DiscreteDelta, UnitStep. ■ *New in Version 5.1.*

+■ Brown

Brown represents the color brown in graphics or style specifications.

Brown is equivalent to RGBColor[0.6, 0.4, 0.2]. ■ See also: Hue, RGBColor. ■ *New in Version 5.1.*

+■ Clip

Clip[x] gives x clipped to be between -1 and $+1$.

Clip[x, {*min*, *max*}] gives x for $min \le x \le max$, *min* for $x < min$ and *max* for $x > max$.

Clip[x, {*min*, *max*}, {v_{min}, v_{max}}] gives v_{min} for $x < min$ and v_{max} for $x > max$.

Clip[x] is effectively equivalent to Piecewise[{{-1, x < -1}, {+1, x > +1}}, x]. ■ The v_i, as well as other arguments of Clip, need not be numbers. ■ For exact numeric quantities, Clip internally uses numerical approximations to establish its result. This process can be affected by the setting of the global variable $MaxExtraPrecision. ■ See also: Sign, Piecewise, Min, Rescale, Chop. ■ *New in Version 5.1.*

~■ Conjugate

~ Conjugate[z] or z^* gives the complex conjugate of the complex number z.

Mathematical function (see Section A.3.10 of the complete *Mathematica Book*). +■ * can be entered as ESC co ESC, ESC conj ESC or \[Conjugate]. ■ See page 746 of the complete *Mathematica Book*. ■ See also: ConjugateTranspose, ComplexExpand. ■ *New in Version 1; modified in Version 5.1.*

+■ ConjugateTranspose

ConjugateTranspose[m] or $m^{\dagger}$ gives the conjugate transpose of m.

ConjugateTranspose[m] is equivalent to Conjugate[Transpose[m]]. ■ † can be entered as ESC ct ESC or \[ConjugateTranspose]. ■ ConjugateTranspose[m] can also be given as m^H, where ^{H} can be entered as ESC hc ESC or \[HermitianConjugate]. ■ See also: Conjugate, Transpose. ■ *New in Version 5.1.*

+■ Cyan

Cyan represents the color cyan in graphics or style specifications.

Cyan is equivalent to RGBColor[0, 1, 1]. ■ See also: CMYKColor, Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

~■ D

D[f, x] gives the partial derivative $\partial f/\partial x$.

D[f, {x, n}] gives the multiple derivative $\partial^n f/\partial x^n$.

D[f, x, y, …] gives $\partial/\partial x\ \partial/\partial y\ \ldots f$.

+ D[f, {{x_1, x_2, … }}] for a scalar f gives the vector derivative $(\partial f/\partial x_1, \partial f/\partial x_2, \ldots)$.

D[f, x] can be input as $\partial_x f$. The character ∂ is entered as ⁝pd⁝ or \[PartialD]. The variable x is entered as a subscript. ■ All quantities that do not explicitly depend on the variables given are taken to have zero partial derivative. ■ D[f, var_1, … , NonConstants -> {u_1, … }] specifies that every u_i implicitly depends on every var_j, so that they do not have zero partial derivative. +■ D[f, {*list*}] threads D over each element of *list*. +■ D[f, {*list*, n}] is equivalent to D[f, {*list*}, {*list*}, …] where {*list*} is repeated n times. If f is a scalar, a *list* has depth 1, then the result is a tensor of rank n, as in the nth term of the multivariate Taylor series of f. +■ D[f, {$list_1$}, {$list_2$}, …] is normally equivalent to First[Outer[D, {f}, $list_1$, $list_2$, …]]. ■ Numerical approximations to derivatives can be found using N. ■ D uses the chain rule to simplify derivatives of unknown functions. ■ D[f, x, y] can be input as $\partial_{x,y} f$. The character \[InvisibleComma], entered as ⁝,⁝, can be used instead of an ordinary comma. It does not display, but is still interpreted just like a comma. ■ See page 853 of the complete *Mathematica Book*. ■ See also: Dt, Derivative, Maximize, CoefficientArrays. ■ Related packages: Calculus\`VectorAnalysis\`, NumericalMath\`NLimit\`. ■ *New in Version 1; modified in Version 5.1.*

+■ Except

Except[c] is a pattern object which represents any expression except one that matches c.

Except[c, p] represents any expression that matches p but not c.

Except[c] is equivalent to Except[c, _]. ■ Except[c] represents the complement of the set of expressions that match c. ■ Except works in StringExpression. ■ See also: PatternTest, Condition. ■ *New in Version 5.1.*

~■ Export

`Export["`*file.ext*`", `*expr*`]` exports data to a file, converting it to a format corresponding to the file extension *ext*.

`Export["`*file*`", `*expr*`, "`*format*`"]` exports data to a file, converting it to the specified format.

`Export` can handle numerical and textual data, graphics, sounds, material from notebooks, and general expressions in various formats. ■ The following basic formats are supported for numerical and textual data:

`"CSV"`	comma-separated value tabular data (`.csv`)
`"Lines"`	list of strings to be placed on separate lines
`"List"`	list of numbers or strings to be placed on separate lines
`"Table"`	list of lists of numbers or strings to be placed in a two-dimensional array (`.dat`)
`"Text"`	single string of ordinary characters (`.txt`)
`"TSV"`	tab-separated value tabular data (`.tsv`)
`"UnicodeText"`	single string of 16-bit Unicode characters
`"Words"`	list of strings to be separated by spaces

■ In `"CSV"`, `"List"` and `"Table"` format, numbers are written in C or Fortran-like "E" notation when necessary. ■ In `"CSV"` format, columns are separated by commas, unless other settings are specified using `ConversionOptions`. ■ In `"Table"` format, columns are separated by spaces. ■ `Export["`*file*`.txt", `*expr*`]` uses `"Text"` format. ■ `Export["`*file*`.dat", `*expr*`]` uses `"Table"` format. +■ The following additional formats are also supported for numerical and textual data:

`"DIF"`	Lotus Data Interchange Format (`.dif`)
`"FITS"`	FITS astronomical data format (`.fit`, `.fits`)
`"HarwellBoeing"`	Harwell-Boeing matrix format
`"HDF"`	Hierarchical Data Format (`.hdf`)
`"HDF5"`	HDF5 format (`.h5`)
`"MAT"`	MAT matrix format (`.mat`)
`"MTX"`	Matrix Market format (`.mtx`)
`"XLS"`	Excel spreadsheet format (`.xls`)

■ All graphics formats in `Export` can handle any type of 2D or 3D *Mathematica* graphics. ■ They can also handle `Notebook` and `Cell` objects. ■ In some formats, lists of frames for animated graphics can be given. ■ The following options can be given when exporting graphics:

`ImageResolution`	`Automatic`	resolution in dpi for the image
`ImageRotated`	`False`	whether to rotate the image (landscape mode)
`ImageSize`	`Automatic`	overall image size

+■ The following graphics formats are independent of the setting for `ImageResolution`:

`"APS"`	*Mathematica* abbreviated PostScript (`.aps`)
`"EPS"`	Encapsulated PostScript (`.eps`)
`"PDF"`	Adobe Acrobat portable document format (`.pdf`)
`"PICT"`	Macintosh PICT
`"SVG"`	Scalable Vector Graphics (`.svg`)
`"WMF"`	Windows metafile format (`.wmf`)

(continued)

~■ `Export` *(continued)*

+■ The following graphics formats depend on the setting for `ImageResolution`:

`"AVI"`	Audio Video Interleave format (`.avi`)
`"BMP"`	Microsoft bitmap format (`.bmp`)
`"DICOM"`	DICOM medical imaging format (`.dcm`, `.dic`)
`"EPSI"`	Encapsulated PostScript with device-independent preview (`.epsi`)
`"EPSTIFF"`	Encapsulated PostScript with TIFF preview
`"GIF"`	GIF and animated GIF (`.gif`)
`"JPEG"`	JPEG (`.jpg`, `.jpeg`)
`"MGF"`	*Mathematica* system-independent raster graphics format (`.mgf`)
`"PBM"`	portable bitmap format (`.pbm`)
`"PCX"`	PCX format (`.pcx`)
`"PGM"`	portable graymap format (`.pgm`)
`"PNG"`	PNG format (`.png`)
`"PNM"`	portable anymap format (`.pnm`)
`"PPM"`	portable pixmap format (`.ppm`)
`"TIFF"`	TIFF (`.tif`, `.tiff`)
`"XBitmap"`	X window system bitmap (`.xbm`)

■ The following three-dimensional graphics formats are supported:

`"DXF"`	AutoCAD drawing interchange format (`.dxf`)
`"STL"`	STL stereolithography format (`.stl`)

■ The following sound formats are supported:

`"AIFF"`	AIFF format (`.aif`, `.aiff`)
`"AU"`	μ law encoding (`.au`)
`"SND"`	sound file format (`.snd`)
`"WAV"`	Microsoft wave format (`.wav`)

■ `Notebook` and `Cell` objects, as well as any box expression obtained from `ToBoxes`, can be exported in the following formats:

`"HTML"`	HTML (`.htm`, `.html`)
`"NB"`	*Mathematica* notebook format (`.nb`)
`"TeX"`	TeX (`.tex`)
`"XHTML+MathML"`	XHTML with MathML inclusions (`.xml`)

■ These formats generate markup material which maintains much of the document structure that exists within *Mathematica*. ■ The following XML formats are supported:

`"ExpressionML"`	format for *Mathematica* expressions
`"MathML"`	format for mathematical expressions (`.mml`)
`"NotebookML"`	format for notebook expressions (`.nbml`)
`"SVG"`	Scalable Vector Graphics format for graphics (`.svg`)
`"XML"`	format determined by content (`.xml`)

■ With format `"MathML"`, box expressions are exported in terms of MathML presentation elements. Other expressions are if possible exported in `TraditionalForm` format. ■ With format `"XML"`, notebook or cell expressions, and notebook objects, are exported as NotebookML. SymbolicXML expressions are exported as general XML. Other expressions are exported as ExpressionML. ■ Arbitrary *Mathematica* expressions can be exported in the following formats:

`"Dump"`	internal binary format (`.mx`)
`"Expression"`	`InputForm` textual format (`.m`)
`"ExpressionML"`	XML-based ExpressionML format

(continued)

~■ Export (continued)

+■ Lists of numbers can be exported in the following raw binary formats:

"Bit"	single bits (0 or 1)
"Byte"	8-bit unsigned integers
"Complex64"	IEEE single-precision complex numbers
"Complex128"	IEEE double-precision complex numbers
"Complex256"	IEEE quad-precision complex numbers
"Integer8"	8-bit signed integers
"Integer16"	16-bit signed integers
"Integer32"	32-bit signed integers
"Integer64"	64-bit signed integers
"Integer128"	128-bit signed integers
"Real32"	IEEE single-precision real numbers
"Real64"	IEEE double-precision real numbers
"Real128"	IEEE quad-precision real numbers
"UnsignedInteger8"	8-bit unsigned integers
"UnsignedInteger16"	16-bit unsigned integers
"UnsignedInteger32"	32-bit unsigned integers
"UnsignedInteger64"	64-bit unsigned integers
"UnsignedInteger128"	128-bit unsigned integers

+■ In each case, numbers are if possible coerced to the format specified, with high-order bits dropped when necessary. +■ Byte ordering is specified by the ByteOrdering option. +■ Raw data for real and complex numbers may differ from one type of computer to another. +■ Lists of characters or strings can be exported in the following raw binary formats:

"Character8"	8-bit characters
"Character16"	16-bit characters
"TerminatedString"	null-terminated strings of 8-bit characters

■ The following general options can be given:

ByteOrdering	$ByteOrdering	what byte order to use for binary data
CharacterEncoding	Automatic	the encoding to use for text characters
ConversionOptions	{}	private options for specific formats

■ Many details can be specified in the setting for ConversionOptions. ■ Possible formats accepted by Export are given in the list $ExportFormats. ■ Export["!*prog*", *expr*, "*format*"] exports data to a pipe. ■ See pages 207, 567 and 642 of the complete *Mathematica Book*. ■ See also: Import, ExportString, $ExportFormats, Display, Write, Put, HTMLSave, MathMLForm, DumpSave. ■ *New in Version 4; modified in Version 5.1.*

+■ Gray

Gray represents the color gray in graphics or style specifications.

Gray is equivalent to GrayLevel[0.5]. ■ See also: GrayLevel, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ Green

Green represents the color green in graphics or style specifications.

Green is equivalent to RGBColor[0, 1, 0]. ■ See also: Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ HessenbergDecomposition

HessenbergDecomposition[*m*] gives the Hessenberg decomposition of a matrix *m*.

The result is given in the form {*p*, *h*} where *p* is a unitary matrix such that *p* . *h* . Conjugate[Transpose[*p*]] == *m*. ■ See also: SchurDecomposition. ■ *New in Version 5.1.*

~■ ImageSize

ImageSize is an option for Export, Display and other graphics functions, as well as for Cell, which specifies the absolute size of an image to render.

The following settings can be given:

wspec	width specified by *wspec*
{*wspec*, *hspec*}	width and height specified

■ Specifications for both width and height can be any of the following:

Automatic (default)	determined by location or other dimension
d	*d* printer's points
72 *di*	*di* inches

■ With ImageSize->{*w*, *h*}, an object will be sized if possible so that its longer dimension just fits in a $w \times h$ region. ■ If the aspect ratio of the object is not *w*/*h*, then space will be left around it. The position of the object in the defined region is determined by the setting for the object's Alignment option. ■ ImageSize -> *w* is equivalent to ImageSize -> {*w*, Automatic}. ■ ImageSize -> {Automatic, *h*} normally determines image size from height, with width left unconstrained. ■ See page 616 of the complete *Mathematica Book*. ■ See also: ImageResolution, ImageMargins, AspectRatioFixed. ■ *New in Version 3; modified in Version 5.1.*

~■ Import

`Import["`*file.ext*`"]` imports data from a file, assuming that it is in the format indicated by the file extension *ext*, and converts it to a *Mathematica* expression.

`Import["`*file*`", "`*format*`"]` imports data in the specified format from a file.

`Import` attempts to give a *Mathematica* expression whose meaning is as close as possible to the data in the external file. ■ `Import` can handle numerical and textual data, graphics, sounds, material from notebooks, and general expressions in various formats. ■ The following basic formats are supported for textual and tabular data:

`"CSV"`	comma-separated value tabular data (`.csv`)
`"Lines"`	lines of text
`"List"`	lines consisting of numbers or strings
`"Table"`	two-dimensional array of numbers or strings
`"Text"`	string of ordinary characters
`"TSV"`	tab-separated value tabular data (`.tsv`)
`"UnicodeText"`	string of 16-bit Unicode characters
`"Words"`	words separated by spaces or newlines

■ `"Text"` and `"UnicodeText"` return single *Mathematica* strings. ■ `"Lines"` and `"Words"` return lists of *Mathematica* strings. ■ `"List"` returns a list of *Mathematica* numbers or strings. ■ `"Table"`, `"CSV"` and `"TSV"` return a list of lists of *Mathematica* numbers or strings. ■ In `"List"`, `"Table"`, `"CSV"` and `"TSV"` formats, numbers can be read in C or Fortran-like "E" notation. ■ Numbers without explicit decimal points are returned as exact integers. ■ In `"Table"` format, columns can be separated by spaces or tabs. ■ In `"Words"` format, words can be separated by any form of whitespace. ■ In `"CSV"` format, columns are taken to be separated by commas, unless other settings are specified using `ConversionOptions`. ■ `Import["`*file*`.txt"]` uses `"Text"` format. ■ `Import["`*file*`.dat"]` uses `"Table"` format. ■ `Import["`*file*`.csv"]` uses `"CSV"` format. +■ The following additional formats are also supported for numerical data:

`"DIF"`	Lotus Data Interchange Format (`.dif`)
`"FITS"`	FITS astronomical data format (`.fit`, `.fits`)
`"HarwellBoeing"`	Harwell-Boeing matrix format
`"HDF"`	Hierarchical Data Format (`.hdf`)
`"HDF5"`	HDF5 format (`.h5`)
`"MAT"`	MAT matrix format (`.mat`)
`"MTX"`	Matrix Market format (`.mtx`)
`"SDTS"`	SDTS spatial GIS data format (`.ddf`)
`"XLS"`	Excel spreadsheet format (`.xls`)

■ When appropriate, numerical data is imported as `SparseArray` objects. ■ The following format yields a list of expressions suitable for input to `NMinimize`:

`"MPS"`	MPS Mathematical Programming System format (`.mps`)

■ Two-dimensional graphics formats are imported as `Graphics` objects; sound formats are imported as `Sound` objects. ■ Animated graphics are imported as lists of `Graphics` objects. +■ The following formats yield expressions of the form `Graphics[`*data*`, `*opts*`]`:

`"APS"`	*Mathematica* abbreviated PostScript (`.aps`)
`"EPS"`	Encapsulated PostScript (`.eps`)
`"EPSI"`	Encapsulated PostScript with `image` preview (`.epsi`)
`"EPSTIFF"`	Encapsulated PostScript with TIFF preview

(continued)

~■ Import *(continued)*

+■ The following formats yield expressions of the form Graphics[Raster[*data*], *opts*]:

"BMP"	Microsoft bitmap format (.bmp)
"DICOM"	DICOM medical imaging format (.dcm, .dic)
"GIF"	GIF and animated GIF (.gif)
"JPEG"	JPEG (.jpg, .jpeg)
"MGF"	*Mathematica* system-independent raster graphics format (.mgf)
"PBM"	portable bitmap format (.pbm)
"PCX"	PCX format (.pcx)
"PGM"	portable graymap format (.pgm)
"PNG"	PNG format (.png)
"PNM"	portable anymap format (.pnm)
"PPM"	portable pixmap format (.ppm)
"TIFF"	TIFF (.tif, .tiff)
"XBitmap"	X window system bitmap (.xbm)

~■ For indexed raster formats such as "GIF", the data in Raster consists of integers, and a ColorFunction is used to specify a color map. ~■ For true color formats such as "JPEG", the data in Raster consists for example of explicit RGB triples. ■ The following formats return objects of the form Graphics3D[*data*, *opts*]:

"DXF"	AutoCAD drawing interchange format (.dxf)
"STL"	STL stereolithography format (.stl)

■ The following formats yield expressions of the form Sound[SampledSoundList[*data*, *r*]]:

"AIFF"	AIFF format (.aif, .aiff)
"AU"	μ law encoding (.au)
"SND"	sound file format (.snd)
"WAV"	Microsoft wave format (.wav)

■ The following gives a notebook expression Notebook[…] from a *Mathematica* notebook file:

"NB"	*Mathematica* notebook format (.nb)

■ The following XML formats give various types of expressions:

"ExpressionML"	arbitrary expression
"MathML"	mathematical expression or boxes (.mml)
"NotebookML"	notebook expression (.nbml)
"SymbolicXML"	SymbolicXML expression
"XML"	determined by content (.xml)

■ With format "MathML", MathML presentation elements are if possible imported as mathematical expressions using TraditionalForm interpretation rules. Otherwise, they are imported as box expressions. ■ With format "SymbolicXML", XML data of any document type is imported as a SymbolicXML expression. ■ With format "XML", Import will recognize MathML, NotebookML, and ExpressionML and interpret them accordingly. Other XML will be imported as SymbolicXML. ■ The following formats can be used for general expressions:

"Dump"	internal binary format (.mx)
"Expression"	InputForm textual format (.m)
"ExpressionML"	XML-based ExpressionML format

(continued)

~■ Import (continued)

+■ The following formats can be used for raw binary data:

"Bit"	single bits (0 or 1)
"Byte"	8-bit unsigned integers
"Character8"	8-bit characters
"Character16"	16-bit characters
"Complex64"	IEEE single-precision complex numbers
"Complex128"	IEEE double-precision complex numbers
"Complex256"	IEEE quad-precision complex numbers
"Integer8"	8-bit signed integers
"Integer16"	16-bit signed integers
"Integer32"	32-bit signed integers
"Integer64"	64-bit signed integers
"Integer128"	128-bit signed integers
"Real32"	IEEE single-precision real numbers
"Real64"	IEEE double-precision real numbers
"Real128"	IEEE quad-precision real numbers
"TerminatedString"	null-terminated strings of 8-bit characters
"UnsignedInteger8"	8-bit unsigned integers
"UnsignedInteger16"	16-bit unsigned integers
"UnsignedInteger32"	32-bit unsigned integers
"UnsignedInteger64"	64-bit unsigned integers
"UnsignedInteger128"	128-bit unsigned integers

+■ In each case, a list of individual data elements is returned. +■ Byte ordering is specified by the ByteOrdering option. ■ The following general options for all formats can be given:

ByteOrdering	$ByteOrdering	what byte order to use for binary data
CharacterEncoding	Automatic	the encoding to use for characters in text
ConversionOptions	{}	private options for specific formats
Path	$Path	the path to search for files

■ Many details can be specified in the setting for ConversionOptions. ■ Possible formats accepted by Import are given in the list $ImportFormats. ■ Import["!*prog*", "*format*"] imports data from a pipe. ■ See pages 207, 570 and 642 of the complete *Mathematica Book*. ■ See also: Export, ImportString, $ImportFormats, ReadList, BinaryReadList. ■ *New in Version 4; modified in Version 5.1.*

+■ Magenta

Magenta represents the color magenta in graphics or style specifications.

Magenta is equivalent to RGBColor[1, 0, 1]. ■ See also: CMYKColor, Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ Orange

Orange represents the color orange in graphics or style specifications.

Orange is equivalent to RGBColor[1, 0.5, 0]. ■ See also: Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ Pick

`Pick[`*list*`, `*sel*`]` picks out those elements of *list* for which the corresponding element of *sel* is `True`.

`Pick[`*list*`, `*sel*`, `*patt*`]` picks out those elements of *list* for which the corresponding element of *sel* matches *patt*.

Example: `Pick[{a, b, c}, {1, 0, 1}, 1]` ⟶ `{a, c}`. ■ *sel* can be a nested list of any depth. ■ `Pick[`*list*`, `*sel*`, `*patt*`]` picks out those *list*`[[`i_1`, `i_2`, … ]]` for which *sel*`[[`i_1`, `i_2`, … ]]` matches *patt*. ■ Depending on the arrangement of elements matching *patt* in a nested list *sel*, `Pick` may return a ragged array. ■ The heads in *list* and *sel* do not have to be `List`. ■ `Pick` works with `SparseArray` objects. ■ See also: `Cases`, `Part`, `Position`, `Boole`, `ListConvolve`. ■ *New in Version 5.1.*

+■ Piecewise

`Piecewise[{{`val_1`, `$cond_1$`}, {`val_2`, `$cond_2$`}, … }]` represents a piecewise function with values val_i in the regions defined by the conditions $cond_i$.

`Piecewise[{{`val_1`, `$cond_1$`}, … }, `*val*`]` uses default value *val* if none of the $cond_i$ apply. The default for *val* is `0`.

The $cond_i$ are typically inequalities such as $a \le x < b$. ■ The $cond_i$ are evaluated in turn, until one of them is found to yield `True`. ■ If all preceding $cond_i$ yield `False`, then the val_i corresponding to the first $cond_i$ that yields `True` is returned as the value of the piecewise function. ■ If any of the preceding $cond_i$ do not literally yield `False`, the `Piecewise` function is returned in symbolic form. ■ Only those val_i explicitly included in the returned form are evaluated. ■ Elements of the form `{`val_i`, False}` are dropped, as are all elements after the first `{`val_i`, True}`. ■ `Piecewise` can be used in such functions as `Integrate`, `Minimize`, `Reduce`, `DSolve` and `Simplify`, as well as their numeric analogs. ■ `Piecewise[{{`v_1`, `c_1`}, {`v_2`, `c_2`}, … }]` can be input in the form $\begin{cases} v_1 & c_1 \\ v_2 & c_2 \\ \ldots \end{cases}$. `{` can be entered as `:pw:` or `\[Piecewise]`. The grid of values and conditions can be constructed by first entering CTRL[,], then using CTRL[↵] (CONTROL-RETURN) and CTRL[,]. ■ In `StandardForm` and `TraditionalForm`, `Piecewise[{{`v_1`, `c_1`}, {`v_2`, `c_2`}, … }]` is normally output as $\begin{cases} v_1 & c_1 \\ v_2 & c_2 \\ \ldots \end{cases}$. ■ See also: `Which`, `PiecewiseExpand`, `Boole`, `If`, `Max`, `Clip`, `UnitStep`, `Abs`, `Sign`, `Floor`, `InterpolatingFunction`, `$MaxPiecewiseCases`. ■ *New in Version 5.1.*

+■ PiecewiseExpand

`PiecewiseExpand[`*expr*`]` expands nested piecewise functions in *expr* to give a single piecewise function.

`PiecewiseExpand[`*expr*`, `*assum*`]` expands piecewise functions using assumptions.

`PiecewiseExpand[`*expr*`, `*assum*`, `*dom*`]` does the expansion over the domain *dom*.

The result from `PiecewiseExpand[`*expr*`]` is normally `Piecewise[{`e_1`, `e_2`, … }]` where none of the e_i contain `Piecewise`. ■ `PiecewiseExpand` converts such functions as `If`, `Which`, `Abs`, `Max`, `UnitStep` and `Floor` to `Piecewise`. ■ `PiecewiseExpand[`*expr*`, `*assum*`, `*dom*`]` assumes that both the input and output of every function appearing in *expr* is in the domain *dom*. ■ Common choices for *dom* are `Reals` and `Complexes`. ■ `PiecewiseExpand` attempts to simplify combinations of conditions that appear in `Piecewise`. ■ The following options can be given:

`Assumptions`	`$Assumptions`	default assumptions to append to *assum*
`TimeConstraint`	`30`	for how many seconds to try simplifying each set of conditions

■ See also: `Piecewise`, `Expand`, `$MaxPiecewiseCases`, `Simplify`, `LogicalExpand`. ■ *New in Version 5.1.*

+■ Pink

Pink represents the color pink in graphics or style specifications.

Pink is equivalent to RGBColor[1, 0.5, 0.5]. ■ See also: Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ Purple

Purple represents the color purple in graphics or style specifications.

Purple is equivalent to RGBColor[0.5, 0, 0.5]. ■ See also: Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ Red

Red represents the color red in graphics or style specifications.

Red is equivalent to RGBColor[1, 0, 0]. ■ See also: Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+ ■ RegularExpression

`RegularExpression["`*regex*`"]` represents the generalized regular expression specified by the string "*regex*".

RegularExpression can be used to represent classes of strings, in functions like StringMatchQ, StringReplace, StringCases and StringSplit. ■ RegularExpression supports standard regular expression syntax, of the kind used in typical string manipulation languages. ■ The following basic elements can be used in regular expression strings:

c	the literal character c
.	any character except newline
[c_1c_2…]	any of the characters c_i
[c_1-c_2]	any character in the range c_1–c_2
[^c_1c_2…]	any character except the c_i
*p**	*p* repeated zero or more times
p+	*p* repeated one or more times
p?	zero or one occurrence of *p*
p{*m*,*n*}	*p* repeated between *m* and *n* times
*p**?, *p*+?, *p*??	the shortest consistent strings that match
(p_1p_2…)	strings matching the sequence p_1, p_2, …
p_1|p_2	strings matching p_1 or p_2

■ The following represent classes of characters:

\\d	digit 0–9
\\D	non-digit
\\s	space, newline, tab or other whitespace character
\\S	non-whitespace character
\\w	word character (letter, digit or _)
\\W	non-word character
[[:*class*:]]	characters in a named class
[^[:*class*:]]	characters not in a named class

■ The following named classes can be used: alnum, alpha, ascii, blank, cntrl, digit, graph, lower, print, punct, space, upper, word, xdigit. ■ The following represent positions in strings:

^	the beginning of the string (or line)
$	the end of the string (or line)
\\b	word boundary
\\B	anywhere except a word boundary

■ The following set options for all regular expression elements that follow them:

(?i)	treat upper and lower case as equivalent (ignore case)
(?m)	make ^ and $ match start and end of lines (multiline mode)
(?s)	allow . to match newline
(?-\#c)	unset options

■ \\., \\[, etc. represent literal characters ., [, etc. ■ Analogs of named *Mathematica* patterns such as *x*:*expr* can be set up in regular expression strings using (*regex*). ■ Within a regular expression string, *n* represents the substring matched by the nth parenthesized regular expression object (*regex*). ■ For the purpose of functions such as StringReplace and StringCases, any $*n* appearing in the right-hand side of a rule RegularExpression["*regex*"] -> *rhs* is taken to correspond to the substring matched by the nth parenthesized regular expression object in *regex*. ■ See also: StringExpression. ■ *New in Version 5.1.*

+■ Rescale

Rescale[x, {min, max}] gives x rescaled to run from 0 to 1 over the range min to max.

Rescale[x, {min, max}, {x_{min}, x_{max}}] gives x rescaled to run from x_{min} to x_{max} over the range min to max.

Rescale[*list*] rescales each element to run from 0 to 1 over the range Min[*list*] to Max[*list*].
■ Rescale[x, {min, max}] is effectively equivalent to (x - min)/(max - min). ■ For exact numeric quantities, Rescale internally uses numerical approximations to establish its result. This process can be affected by the setting of the global variable $MaxExtraPrecision. ■ See also: Mean, Clip. ■ *New in Version 5.1.*

+■ StringCases

StringCases["*string*", *patt*] gives a list of the substrings in "*string*" that match the string expression *patt*.

StringCases["*string*", *lhs* -> *rhs*] gives a list of the values of *rhs* corresponding to the substrings that match the string expression *lhs*.

StringCases["*string*", p, n] includes only the first n substrings that match.

StringCases["*string*", {p_1, p_2, … }] gives substrings that match any of the p_i.

StringCases[{s_1, s_2, … }, p] gives the list of results for each of the s_i.

String expressions can contain any of the objects specified in the notes for StringExpression. ■ Example: StringCases["abcadcacb", "a"~~_~~"c"] ⟶ {abc, adc}. ■ Example: StringCases["abcadcacb", "a"~~x_~~"c" -> x] ⟶ {b, d}. ■ With the default option setting Overlaps -> False, StringCases includes only substrings that do not overlap. With Overlaps -> True it includes substrings that overlap. ■ With Overlaps -> All, multiple substrings that match the same string expression are all included. With Overlaps -> True, only the first such matching substring at a given position is included. ■ Setting the option IgnoreCase -> True makes StringCases treat lower- and upper-case letters as equivalent.
■ StringCases["*string*", RegularExpression["*regex*"]] gives substrings matching the specified regular expression.
■ StringCases[s, *lhs* :> *rhs*] evaluates *rhs* only when the pattern is found. ■ See also: Cases, StringPosition, StringCount, StringReplace, StringReplaceList, Characters, StringExpression, RegularExpression. ■ *New in Version 5.1.*

+■ StringCount

StringCount["*string*", "*sub*"] gives a count of the number of times "*sub*" appears as a substring of "*string*".

StringCount["*string*", *patt*] gives the number of substrings in "*string*" that match the general string expression *patt*.

StringCount["*string*", {$patt_1$, $patt_2$, ... }] counts the number of occurrences of any of the $patt_i$.

StringCount[{s_1, s_2, ... }, *p*] gives the list of results for each of the s_i.

Example: StringCount["abbaabbaa", "bb"] ⟶ 2. ■ The string expression *patt* can contain any of the objects specified in the notes for StringExpression. ■ Example: StringCount["abcadcacb", "a"~~_~~"c"] ⟶ 2. ■ With the default option setting Overlaps -> True, StringCount counts as separate substrings that overlap. With the setting Overlaps -> False such substrings are not treated as separate. ■ With Overlaps -> All, multiple substrings that match the same string expression are all counted as separate. With Overlaps -> True, only the first such matching substring at a given position is counted as separate. ■ Setting the option IgnoreCase -> True makes StringCount treat lower- and upper-case letters as equivalent. ■ Example: StringCount["abAB", "a", IgnoreCase -> True] ⟶ 2. ■ StringCount["*string*", RegularExpression["*regex*"]] gives the number of substrings matching the specified regular expression. ■ See also: Count, StringPosition, StringCases, Characters, StringExpression, RegularExpression. ■ *New in Version 5.1.*

~■ StringDrop

StringDrop["*string*", *n*] gives "*string*" with its first *n* characters dropped.

StringDrop["*string*", -*n*] gives "*string*" with its last *n* characters dropped.

StringDrop["*string*", {*n*}] gives "*string*" with its nth character dropped.

StringDrop["*string*", {*m*, *n*}] gives "*string*" with characters *m* through *n* dropped.

+ StringDrop[{s_1, s_2, ... }, *spec*] gives the list of results for each of the s_i.

StringDrop uses the standard *sequence specification* (see page 1040 of the complete *Mathematica Book*). ■ Example: StringDrop["abcdefgh", 2] ⟶ cdefgh. ■ StringDrop["*string*", {*m*, *n*, *s*}] drops characters *m* through *n* in steps of *s*. ■ See page 407 of the complete *Mathematica Book*. ■ See also: Drop, StringTake, StringPosition, StringReplacePart. ■ *New in Version 2; modified in Version 5.1.*

+■ StringExpression

s_1 ~~ s_2 ~~ … or StringExpression[s_1, s_2, …] represents a sequence of strings and symbolic string objects s_i.

"str_1" ~~ "str_2" ~~ … yields an ordinary string obtained by concatenating the characters in the "str_i". ■ The following objects can appear in StringExpression:

"*string*"	a literal string of characters
_	any single character
__	any substring of one or more characters
___	any substring of zero or more characters
*x*_, *x*__, *x*___	substrings given the name *x*
x:*pattern*	pattern given the name *x*
pattern..	pattern repeated one or more times
pattern...	pattern repeated zero or more times
{$patt_1$, $patt_2$, … } or $patt_1$ | $patt_2$ | …	a pattern matching at least one of the $patt_i$
patt /; *cond*	a pattern for which *cond* evaluates to True
pattern ? *test*	a pattern for which *test* yields True for each character
Whitespace	a sequence of whitespace characters
NumberString	the characters of a number
charobj	an object representing a character class (see below)
RegularExpression["*regexp*"]	substring matching a regular expression
StringExpression[…]	an arbitrary string expression

■ The following represent classes of characters:

{"c_1", "c_2", … }	any of the "c_i"
Characters["c_1c_2 … "]	any of the "c_i"
CharacterRange["c_1", "c_2"]	any character in the range "c_1" to "c_2"
DigitCharacter	digit 0–9
LetterCharacter	letter
WhitespaceCharacter	space, newline, tab or other whitespace character
WordCharacter	letter or digit
Except[*p*]	any character except ones matching *p*

■ The following represent positions in strings:

StartOfString	start of the whole string
EndOfString	end of the whole string
StartOfLine	start of a line
EndOfLine	end of a line
WordBoundary	boundary between word characters and others
Except[WordBoundary]	anywhere except a word boundary

■ When constructs like __ or .. are present, there may be several different ways in which a given StringExpression can match a particular string. ■ By default, *Mathematica* will use the one that makes pattern elements that appear earlier in the StringExpression match the longest possible substrings. ■ The following determine which match will be used if there are several possibilities:

ShortestMatch[*p*]	the shortest consistent match for *p*
LongestMatch[*p*]	the longest consistent match for *p* (default)

■ In matching ordinary expressions instead of strings, the shortest instead of the longest consistent match is used. ■ StringExpression objects can be used in many string manipulation functions, including StringReplace, StringCases, StringSplit, StringMatchQ. ■ StringExpression has attributes Flat and OneIdentity. ■ See also: RegularExpression, String, StringJoin. ■ *New in Version 5.1.*

+■ StringFreeQ

StringFreeQ["*string*", *patt*] yields True if no substring in "*string*" matches the string expression *patt*, and yields False otherwise.

StringFreeQ["*string*", {$patt_1$, $patt_2$, … }] yields True if no substring matches any of the $patt_i$.

StringFreeQ[{s_1, s_2, … }, *p*] gives the list of results for each of the s_i.

The string expression *patt* can contain any of the objects specified in the notes for StringExpression. ■ See also: StringCases, StringMatchQ, FreeQ, StringExpression, RegularExpression. ■ *New in Version 5.1.*

~■ StringInsert

StringInsert["*string*", "*snew*", *n*] yields a string with "*snew*" inserted starting at position *n* in "*string*".

StringInsert["*string*", "*snew*", -*n*] inserts at position *n* from the end of "*string*".

StringInsert["*string*", "*snew*", {n_1, n_2, … }] inserts a copy of "*snew*" at each of the positions n_i.

+ StringInsert[{s_1, s_2, … }, "*snew*", *n*] gives the list of results for each of the s_i.

Example: StringInsert["abcdefg", "XYZ", 2] ⟶ aXYZbcdefg. ■ StringInsert["*string*", "*snew*", *n*] makes the first character of *snew* the nth character in the new string. ■ StringInsert["*string*", "*snew*", -*n*] makes the last character of *snew* the nth character from the end of the new string. ■ In StringInsert["*string*", "*snew*", {n_1, n_2, … }] the n_i are taken to refer to positions in "*string*" before any insertion is done. ■ See page 408 of the complete *Mathematica Book*. ■ See also: StringReplacePart, Insert, StringPosition. ■ *New in Version 2; modified in Version 5.1.*

~■ StringLength

StringLength["*string*"] gives the number of characters in a string.

Example: StringLength["tiger"] ⟶ 5. ■ StringLength counts special characters such as α as single characters, even if their full names involve many characters. +■ StringLength[{s_1, s_2, … }] gives the list of lengths of each of the s_i. ■ See page 407 of the complete *Mathematica Book*. ■ See also: Length, Characters. ■ *New in Version 1; modified in Version 5.1.*

~■ StringMatchQ

~ StringMatchQ["*string*", *patt*] tests whether *string* matches the string pattern *patt*.

+ StringMatchQ["*string*", RegularExpression["*regex*"]] tests whether *string* matches the specified regular expression.

+ StringMatchQ[{s_1, s_2, … }, *p*] gives the list of results for each of the s_i.

StringMatchQ allows abbreviated string patterns containing the metacharacters * and @ in the form specified on page 1044 of the complete *Mathematica Book*. ■ Example: StringMatchQ["apppbb", "a*b"] ⟶ True. +■ Verbatim["*p*"] specifies the verbatim string "*p*", with * and @ treated literally. ■ Setting the option IgnoreCase -> True makes StringMatchQ treat lower- and upper-case letters as equivalent. ■ Setting the option SpellingCorrection -> True makes StringMatchQ allow strings to match even if a small fraction of their characters are different. ■ See page 411 of the complete *Mathematica Book*. ■ See also: StringFreeQ, StringPosition, StringCases, Equal, Names, MatchQ, StringExpression, RegularExpression. ■ *New in Version 1; modified in Version 5.1.*

~■ StringPosition

StringPosition["*string*", "*sub*"] gives a list of the starting and ending character positions at which "*sub*" appears as a substring of "*string*".

+ StringPosition["*string*", *patt*] gives all positions at which substrings matching the general string expression *patt* appear in "*string*".

~ StringPosition["*string*", *patt*, *n*] includes only the first *n* occurrences of *patt*.

~ StringPosition["*string*", {$patt_1$, $patt_2$, … }] gives positions of all the $patt_i$.

+ StringPosition[{s_1, s_2, … }, *p*] gives the list of results for each of the s_i.

Example: StringPosition["abbaabbaa", "bb"] ⟶ {{2, 3}, {6, 7}}. +■ The string expression *patt* can contain any of the objects specified in the notes for StringExpression. ■ Example: StringPosition["abcadcacb", "a"~~_~~"c"] ⟶ {{1, 3}, {4, 6}}. ■ With the default option setting Overlaps -> True, StringPosition includes substrings that overlap. With the setting Overlaps -> False such substrings are excluded. +■ With Overlaps -> All, multiple substrings that match the same string expression are all included. With Overlaps -> True, only the first such matching substring at a given position is included. ■ Setting the option IgnoreCase -> True makes StringPosition treat lower- and upper-case letters as equivalent. ■ Example: StringPosition["abAB", "a", IgnoreCase -> True] ⟶ {{1, 1}, {3, 3}}. ■ StringPosition returns sequence specifications in the form used by StringTake, StringDrop and StringReplacePart. +■ StringPosition["*string*", RegularExpression["*regex*"]] gives positions of substrings matching the specified regular expression. ■ See page 409 of the complete *Mathematica Book*. ■ See also: Position, StringCases, StringCount, Characters, FindList, ReplaceList, StringExpression, RegularExpression. ■ *New in Version 2; modified in Version 5.1.*

~■ StringReplace

~ StringReplace["*string*", *s* -> *sp*] or StringReplace["*string*", {s_1 -> sp_1, s_2 -> sp_2, … }] replaces the string expressions s_i by sp_i whenever they appear as substrings of "*string*".

+ StringReplace["*string*", *srules*, *n*] does only the first *n* replacements.

+ StringReplace[{s_1, s_2, … }, *srules*] gives the list of results for each of the s_i.

Example: StringReplace["abbaabbaa", "ab"->"X"] ⟶ XbaXbaa. +■ The string expressions s_i can contain any of the objects specified in the notes for StringExpression. ■ StringReplace goes through a string, testing substrings that start at each successive character position. On each substring, it tries in turn each of the transformation rules you have specified. If any of the rules apply, it replaces the substring, then continues to go through the string, starting at the character position after the end of the substring. +■ If the sp_i in the replacements s_i->sp_i do not evaluate to strings, StringReplace will yield a StringExpression rather than an ordinary string. ■ In replacements of the form s_i :> sp_i, the sp_i are not evaluated until each time they are used. ■ Setting the option IgnoreCase -> True makes StringReplace treat lower- and upper-case letters as equivalent. ■ See page 410 of the complete *Mathematica Book*. ■ See also: Replace, StringReplaceList, StringReplacePart, StringPosition, StringSplit, ToLowerCase, ToUpperCase. ■ *New in Version 2; modified in Version 5.1.*

+■ StringReplaceList

StringReplaceList["*string*", *s* -> *sp*] or
StringReplaceList["*string*", {s_1 -> sp_1, s_2 -> sp_2, … }] gives a list of the strings obtained by replacing each individual occurrence of substrings in "*string*" matching the string expressions s_i.

StringReplaceList["*string*", *srules*, *n*] gives a list of the first *n* results obtained.

StringReplaceList[{s_1, s_2, … }, *srules*] gives the list of results for each of the s_i.

Example: StringReplaceList["aaa", "a" -> "X"] ⟶ {Xaa, aXa, aaX}. ■ The string expressions s_i can contain any of the objects specified in the notes for StringExpression. ■ In each of the results returned by StringReplaceList only one substring has been replaced. ■ StringReplaceList goes through a string, testing substrings that start at each successive character position. On each substring, it tries in turn each of the transformation rules you have specified, returning a result for each one that applies. ■ StringReplaceList in effect carries out a single step in the evolution of a multiway system. ■ If the sp_i in the replacements s_i->sp_i do not evaluate to strings, StringReplaceList will yield a StringExpression rather than an ordinary string. ■ In replacements of the form s_i :> sp_i, the sp_i are not evaluated until each time they are used. ■ Setting the option IgnoreCase -> True makes StringReplaceList treat lower- and upper-case letters as equivalent. ■ See also: StringReplace, ReplaceList, StringCases. ■ *New in Version 5.1.*

~■ StringReverse

StringReverse["*string*"] reverses the order of the characters in "*string*".

Example: StringReverse["abcde"] ⟶ edcba. +■ StringReverse[{s_1, s_2, … }] gives the list of results for each of the s_i. ■ See page 407 of the complete *Mathematica Book*. ■ See also: Reverse. ■ *New in Version 2; modified in Version 5.1.*

+■ StringSplit

StringSplit["*string*"] splits "*string*" into a list of substrings separated by whitespace.

StringSplit["*string*", *patt*] splits into substrings separated by delimiters matching the string expression *patt*.

StringSplit["*string*", {p_1, p_2, … }] splits at any of the p_i.

StringSplit["*string*", *patt* -> *val*] inserts *val* at the position of each delimiter.

StringSplit["*string*", {p_1 -> v_1, … }] inserts v_i at the position of each delimiter p_i.

StringSplit["*string*", *patt*, *n*] splits into at most *n* substrings.

StringSplit[{s_1, s_2, … }, *p*] gives the list of results for each of the s_i.

Example: StringSplit["a bb cc a"] ⟶ {a, bb, cc, a}. ■ StringSplit[*s*] does not return the whitespace characters that delimit the substrings it returns. ■ Whitespace includes any number of spaces, tabs and newlines. ■ The string expression *patt* can contain any of the objects specified in the notes for StringExpression. ■ StringSplit[*s*] is equivalent to StringSplit[*s*, Whitespace]. ■ If *s* contains two adjacent delimiters, StringSplit considers there to be a zero-length substring "" between them. ■ StringSplit[*s*, *patt*] by default gives the list of substrings of *s* that occur between delimiters defined by *patt*; it does not include the delimiters themselves. ■ StringSplit[*s*, *patt* -> *val*] includes *val* at the position of each delimiter. ■ StringSplit["*string*", {p_1 -> v_1, … , p_a, … }] includes v_1 at the position of delimiters matching p_1, but omits delimiters matching p_a. ■ By default, StringSplit[*s*, *patt*] drops zero-length substrings associated with delimiters that appear at the beginning or end of *s*. ■ StringSplit[*s*, *patt*, All] returns all substrings, including zero-length ones at the beginning or end. ■ Setting the option IgnoreCase -> True makes StringSplit treat lower- and upper-case letters as equivalent. ■ StringSplit["*string*", RegularExpression["*regex*"]] splits at delimiters matching the specified regular expression. ■ See also: Split, StringCases. ■ *New in Version 5.1.*

~■ StringTake

StringTake["*string*", n] gives a string containing 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[{s_1, s_2, … }, *spec*] gives the list of results for each of the s_i.

StringTake uses the standard *sequence specification* (see page 1040 of the complete *Mathematica Book*). ■ Example: StringTake["abcdefg", 3] ⟶ abc. ■ StringTake["*string*", {m, n, s}] gives characters m through n in steps of s. ■ See page 407 of the complete *Mathematica Book*. ■ See also: Take, StringDrop, StringPosition. ■ *New in Version 2; modified in Version 5.1.*

+■ Subsets

Subsets[*list*] gives a list of all possible subsets of *list*.

Subsets[*list*, n] gives all subsets containing at most n elements.

Subsets[*list*, {n}] gives all subsets containing exactly n elements.

Subsets[*list*, {n_{min}, n_{max}}] gives all subsets containing between n_{min} and n_{max} elements.

Subsets[*list*, *nspec*, s] gives the first s subsets.

Subsets[*list*, *nspec*, {s}] gives the sth subset.

Subsets[*list*] gives the power set of *list*. ■ Subsets[*list*] orders subsets with shortest first, and later elements in *list* omitted first. ■ If the elements of *list* are in the order returned by Sort, then the complete result from Subsets[*list*] will also be in this order. ■ Subsets[*list*, All] is equivalent to Subsets[*list*]. ■ Subsets[*list*, {n_{min}, n_{max}, dn}] gives subsets containing n_{min}, $n_{min} + dn$, … elements. ■ Subsets[*list*, *nspec*, *spec*] gives the same result as Take[Subsets[*list*, *nspec*], *spec*]. ■ See also: Tuples, IntegerDigits. ■ *New in Version 5.1.*

~■ Transpose

Transpose[*list*] transposes the first two levels in *list*.

Transpose[*list*, {n_1, n_2, … }] transposes *list* so that the kth level in *list* is the ${n_k}^{th}$ level in the result.

Example: Transpose[{{a,b},{c,d}}] ⟶ {{a, c}, {b, d}}. ■ Transpose gives the usual transpose of a matrix. +■ Transpose[m] can be input as m^{T}. +■ $^{\mathsf{T}}$can be entered as ESC tr ESC or \[Transpose]. ■ Acting on a tensor $T_{i_1 i_2 i_3 \ldots}$ Transpose gives the tensor $T_{i_2 i_1 i_3 \ldots}$. ■ Transpose[*list*, {n_1, n_2, … }] gives the tensor $T_{i_{n_1} i_{n_2} \ldots}$. ■ So long as the lengths of the lists at particular levels are the same, the specifications n_k do not necessarily have to be distinct. ■ Example: Transpose[Array[a, {3, 3}], {1, 1}] ⟶ {a[1, 1], a[2, 2], a[3, 3]}. ■ Transpose works on SparseArray objects. ■ See page 905 of the complete *Mathematica Book*. ■ See also: Flatten, Thread, ConjugateTranspose, Tr. ■ Related package: LinearAlgebra`MatrixManipulation`. ■ *New in Version 1; modified in Version 5.1.*

+■ Tuples

Tuples[*list*, *n*] generates a list of all possible *n*-tuples of elements from *list*.

Tuples[{$list_1$, $list_2$, … }] generates a list of all possible tuples whose i^{th} element is from $list_i$.

Example: Tuples[{a, b}, 2] ⟶ {{a, a}, {a, b}, {b, a}, {b, b}}. ■ The elements of *list* are treated as distinct, so that Tuples[*list*, *n*] for a list of length k gives output of length k^n. ■ The order of elements in Tuples[*list*, *n*] is based on the order of elements in *list*, so that Tuples[{a_1, … , a_k}, *n*] gives {{a_1, a_1, … , a_1}, {a_1, a_1, … , a_2}, … , {a_k, a_k, … , a_k}}. ■ Tuples[*list*, {n_1, n_2, … }] generates a list of all possible $n_1 \times n_2 \times \ldots$ arrays of elements in *list*. ■ The object *list* need not have head List. The head at each level in the arrays generated by Tuples will be the same as the head of *list*. ■ See also: Outer, Array, IntegerDigits, Permutations, Subsets, Distribute. ■ *New in Version 5.1.*

+■ White

White represents the color white in graphics or style specifications.

White is equivalent to GrayLevel[1]. ■ See also: GrayLevel, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ Yellow

Yellow represents the color yellow in graphics or style specifications.

Yellow is equivalent to RGBColor[1, 1, 0]. ■ See also: CMYKColor, Hue, RGBColor. ■ Related package: Graphics`Colors`. ■ *New in Version 5.1.*

+■ $MaxPiecewiseCases

$MaxPiecewiseCases gives the maximum number of cases to allow in explicit Piecewise objects generated by expanding any single piecewise function.

The default value of $MaxPiecewiseCases is 100. ■ $MaxPiecewiseCases is used not only in PiecewiseExpand, but also such functions as Integrate, Reduce and FunctionExpand. ■ See also: PiecewiseExpand. ■ *New in Version 5.1.*

A.12 Listing of Named Characters

* \[Conjugate]

Aliases: ⋮co⋮, ⋮conj⋮. ■ Superscript postfix operator with built-in evaluation rules. ■ z^* is by default interpreted as Conjugate[*z*]. ■ Not the same as keyboard * or \[Star]. ■ See also: \[ConjugateTranspose], \[HermitianConjugate].

† \[ConjugateTranspose]

Alias: ⋮ct⋮. ■ Superscript postfix operator with built-in evaluation rules. ■ $m^\dagger$ is by default interpreted as ConjugateTranspose[*m*]. ■ Not the same as \[Dagger]. ■ See also: \[HermitianConjugate], \[Conjugate], \[Transpose].

† \[Dagger]

Alias: ⋮dg⋮. ■ Letter-like form. ■ $x^\dagger$ is by default interpreted as SuperDagger[*x*]. ■ Not the same as \[ConjugateTranspose]. ■ See also: \[DoubleDagger].

; \[EntityEnd]

Letter-like form. ■ Used to indicate the end of an XML-style entity specifier. ■ See also: \[EntityStart].

& \[EntityStart]

Letter-like form. ■ Used to indicate the start of an XML-style entity specifier. ■ See also: \[EntityEnd].

ˇ \[Hacek]

Alias: ⋮hck⋮. ■ Letter-like form. ■ Used primarily in an overscript position. ■ Used as a diacritical mark in Eastern European languages. ■ Sometimes used in mathematical notation, for example in Čech cohomology. ■ See also: \[Vee], \[Breve].

ᴴ \[HermitianConjugate]

Alias: ⋮hc⋮. ■ Superscript postfix operator with built-in evaluation rules. ■ m^H is by default interpreted as ConjugateTranspose[*m*]. ■ Not the same as keyboard H. ■ See also: \[ConjugateTranspose], \[Conjugate], \[Transpose].

{ \[Piecewise]

Alias: ⋮pw⋮. ■ Prefix operator with built-in evaluation rules. ■ $\begin{cases} e_1 & c_1 \\ e_2 & c_2 \end{cases}$ is by default interpreted as Piecewise[{{e_1, c_1}, {e_2, c_2}}]. ■ Extensible character. ■ Not the same as keyboard {.

® `\[RegisteredTrademark]`

Alias: ⋮rtm⋮. ■ Letter-like form. ■ Used as a superscript to indicate a registered trademark such as *Mathematica*. ■ Typically used only on the first occurrence of a trademark in a document. ■ See also: `\[Trademark]`, `\[Copyright]`.

™ `\[Trademark]`

Alias: ⋮tm⋮. ■ Letter-like form. ■ Used to indicate a trademark that may not be registered. ■ Typically used only on the first occurrence of a trademark in a document. ■ See also: `\[RegisteredTrademark]`, `\[Copyright]`.

ᵀ `\[Transpose]`

Alias: ⋮tr⋮. ■ Superscript postfix operator with built-in evaluation rules. ■ $m^{\top}$ is by default interpreted as `Transpose[`*m*`]`. ■ Not the same as keyboard `T`. ■ See also: `\[ConjugateTranspose]`.

Index

/etc/rc.local file, 33

5.1, new features in, iii

Absent from list, Complement, 1
Alphabetic character, LetterCharacter, 14
Alphabetizing, of strings, Sort, 11
Areas, of regions, Integrate, 29
Array, 8
ArrayPlot, 37
Astronomical data, importing, Import, 4, 48
Avoiding matches, Except, 6, 41
awk, RegularExpression, 17, 51

BinaryFormat, 23
BinaryRead, 21, 38
BinaryReadList, 23, 39
BinaryWrite, 21, 39
Bits, in files, Import, 24, 48
Black, 39
Blue, 39
Boole, 25, 40
Boolean expansion, LogicalExpand, 26
Boot time, network licenses and, 33
Box function, Piecewise, 25, 49
Boxcar function, Piecewise, 25, 49
Breaking strings, StringSplit, 10, 57
Brown, 40
Byte format, 22
ByteOrdering, 23

Calculus, differential, D, 29, 41
Canonical form, JordanDecomposition, 32
Case independence, in string operations, 16
Cases, 7
Changing parts, of lists, 8
Character classes, 14, 19
Characteristic function, Boole, 25, 40
CharacteristicPolynomial, 31
CharacterRange, 14
Characters, binary formats for, 22
  replacement of, StringReplace, 10, 56
Characters, 14
Chinese remainder theorem, Reduce, 28
Clip, 40
CNF, LogicalExpand, 26
Cofactors, Minors, 31
Collating, of strings, Sort, 11
Combinations, of elements in lists, Tuples, 2, 59
Comma-separated values, exporting, Export, 4, 44
  importing, Import, 4, 48
Complement, 1
Complementary patterns, Except, 6, 41
Complex numbers, binary formats for, 22
Conditionals, denesting of, PiecewiseExpand, 26, 49
Conjugate, 40
\[Conjugate] (*), 60
ConjugateTranspose, 31, 40
\[ConjugateTranspose] (†), 60

Conjunctive normal form, LogicalExpand, 26
Count, substrings, StringCount, 10, 53
crontab file, 33
CSV format, exporting, Export, 4, 44
  importing, Import, 4, 48
Cyan, 40
Cyclic vectors, JordanDecomposition, 32

D, 29, 41
Daemon, *Mathematica*, 33
\[Dagger] (†), 60
Data input, BinaryRead, 21, 38
⋮dd⋮, \[DifferentialD] (ⅆ), 3
ddf format, importing, Import, 4, 48
Delimiters, StringSplit, 10, 57
Demon, *Mathematica*, 33
Denesting, of conditionals, PiecewiseExpand, 26, 49
Derivative tensor, D, 29, 41
Det, 31
Determinants, Det, 31
Developments, in Version 5.1, iii
Devices, BinaryWrite, 21, 39
DIF format, exporting, Export, 4, 44
  importing, Import, 4, 48
\[DifferentialD] (ⅆ), 3
Differentiation, 29
DigitCharacter, 14
Discontinuous functions, Piecewise, 25, 49
Disjointness, of sets, Intersection, 1
Disjunctive normal form, LogicalExpand, 26
Distinct elements in lists, Union, 1
Divergence, D, 29, 41
Dividing strings, StringSplit, 10, 57
DNF, LogicalExpand, 26
DOS files, BinaryFormat, 23
double binary format, 22
Duplicates, removal of in lists, Union, 1

Editing, of strings, StringReplace, 10, 56
EndOfLine, 16
EndOfString, 16
Enhancements, in current version, iii
\[EntityEnd] (⁝), 60
\[EntityStart] (&), 60
Equal, 11
Equality testing, for strings, Equal, 11
Equations, manipulation of, 28
Excel format, exporting, Export, 4, 44
  importing, Import, 4, 48
Except, 6, 14, 41
Excluded patterns, Except, 6, 41
Expansion, of conditionals, PiecewiseExpand, 26, 49
  of logic expressions, LogicalExpand, 26
  of piecewise functions, PiecewiseExpand, 26, 49
Export, 4, 44

External data, exporting, Export, 4, 44
  importing, Import, 4, 48
External interface, 4

FEM, Piecewise, 25, 49
Files, system, 35
Finite elements, Piecewise, 25, 49
Finite state machines, RegularExpression, 17, 51
FITS format, exporting, Export, 4, 44
  importing, Import, 4, 48
Flattening, of piecewise functions, PiecewiseExpand, 26, 49
Flexible Image Transport System, exporting, Export, 4, 44
  importing, Import, 4, 48
Foreign data, exporting, Export, 4, 44
  importing, Import, 4, 48
ftp, Import, 5, 48
Functions, mathematical, 25
  of matrices, JordanDecomposition, 32
Fuzzy logic, Piecewise, 25, 49

Generating tuples, Tuples, 2, 59
GIS data, importing, Import, 4, 48
Gradients, D, 29, 41
Gray, 44
Greedy string patterns, LongestMatch, 16
Green, 44
grep, RegularExpression, 17, 51

\[Hacek] (ˇ), 60
Harwell-Boeing format, Import, 4, 48
HDF, exporting, Export, 4, 44
  importing, Import, 4, 48
Hermitian conjugate, ConjugateTranspose, 31, 40
\[HermitianConjugate] (ᴴ), 60
HessenbergDecomposition, 32, 45
Hessian, D, 29, 41
Hierarchical Data Format, exporting, Export, 4, 44
  importing, Import, 4, 48
http, Import, 5, 48
Hybrid systems, Piecewise, 25, 49

IEEE numbers, 22
IgnoreCase, 16
ImageSize, 45
Import, 4, 5, 24, 48
Indicator function, Boole, 25, 40
Input, from TeX, ToExpression, 5
  full story on, 21
  in notebooks, 3
Installation, 35
$InstallationDirectory, 35
int binary format, 22
\[Integral] (∫), 3
Integrate, 29
Interfacing with *Mathematica*, 4
Intersection, 1

Interval-defined functions, `Piecewise`, 25, 49
Invariant subspaces, `JordanDecomposition`, 32
`Inverse`, 31
Iverson's convention, `Boole`, 25, 40

Jacobian, `D`, 29, 41
`JordanDecomposition`, 32

Kleene star operator, `RegularExpression`, 17, 51

`LetterCharacter`, 14
Linear algebra, 31
`List`, 1
Lists, common elements in, `Intersection`, 1
  difference between, `Complement`, 1
  distinct elements in, `Union`, 1
  intersection of, `Intersection`, 1
  nested, 8
Log file, for network licenses, 34
Logging, of network license manager, 34
`LogicalExpand`, 26
`LongestMatch`, 16
Lotus 1-2-3 format, exporting, `Export`, 4, 44
  importing, `Import`, 4, 48
Low-level data, `BinaryRead`, 21, 38

`Magenta`, 48
`Map`, 7
`MapIndexed`, 7
Masking, `Pick`, 7, 49
MAT format, exporting, `Export`, 4, 44
  importing, `Import`, 4, 48
Matrices, 31
  determinants of, `Det`, 31
  exporting, `Export`, 4, 44
  functions of, `JordanDecomposition`, 32
  Hessenberg decomposition of, `HessenbergDecomposition`, 32, 45
  importing, `Import`, 4, 48
  Jordan decomposition of, `JordanDecomposition`, 32
  minors of, `Minors`, 31
  operations on, 31
  resetting parts of, 8
  Schur decomposition of, `SchurDecomposition`, 32
  trace of, `Tr`, 31
  transpose of, `Transpose`, 31, 58
Matrix Market format, exporting, `Export`, 4, 44
  importing, `Import`, 4, 48
`$MaxPiecewiseCases`, 27, 59
Measures, of regions, `Integrate`, 29
Mensuration, `Integrate`, 29
`Minors`, 31
`monitorlm`, 34

MTX format, exporting, `Export`, 4, 44
  importing, `Import`, 4, 48
Multivariate derivative, `D`, 29, 41
Multiway systems, `StringReplaceList`, 10, 57

Nested lists, 8
New features in *Mathematica* Version 5.1, iii
Newline translation, 23
Newlines, in string patterns, 15
Non-greedy string patterns, `ShortestMatch`, 16
Normal form, Jordan, `JordanDecomposition`, 32
Notebooks, input and output in, 3
Numbers, binary formats for, 22
`NumberString`, 13

`Orange`, 48
Ordered pairs, in lists, `Tuples`, 2, 59
Ordering, of strings, `Sort`, 11
`Outer`, 8
Output, full story on, 21
  in notebooks, 3
Overlap between lists, `Intersection`, 1
`Overlaps`, 17

Pairs, in lists, `Tuples`, 2, 59
`\[PartialD]` (∂), 3
Parts, replacing, 8
Patterns, for strings, 11
⋮pd⋮, `\[PartialD]` (∂), 3
`Pick`, 7, 49
`Piecewise`, 25, 49
`\[Piecewise]` ($\{$), 60
`PiecewiseExpand`, 26, 49
`Pink`, 50
Plus function markers, 36
PNF, `PiecewiseExpand`, 26, 49
POSIX character classes, 20
Powerset, `Subsets`, 1, 58
`\[Product]` ($\prod$), 3
Production rules, in strings, `StringReplace`, 10, 56
`Purple`, 50

Quad precision binary format, 22

Ramp function, `Piecewise`, 25, 49
Raw data, `BinaryRead`, 21, 38
Reading, binary data, `BinaryRead`, 21, 38
`REAL` binary format, 22
Real numbers, binary formats for, 22
`Red`, 50
`Reduce`, 28
Regex, `RegularExpression`, 17, 51
Regions, integrals over, `Integrate`, 29
`\[RegisteredTrademark]` (®), 61
Regular expressions, `RegularExpression`, 17, 51

`RegularExpression`, 17, 51
Reluctant string patterns, `ShortestMatch`, 16
Replacements, of substrings, `StringReplace`, 10, 56
`Rescale`, 52
Revisions, in current version, iii
Rewriting, of strings, `StringReplace`, 10, 56

Saturation function, `Piecewise`, 25, 49
Sawtooth function, `Piecewise`, 25, 49
`SchurDecomposition`, 32
SDTS importing, `Import`, 4, 48
Sectionally defined functions, `Piecewise`, 25, 49
`Select`, 7
Sequential substitution systems, `StringReplace`, 10, 56
Sessions, 33
Set exponential, `Subsets`, 1, 58
Sets, complements of, `Complement`, 1
  difference between, `Complement`, 1
  intersection of, `Intersection`, 1
  union of, `Union`, 1
`ShortestMatch`, 16
Similarity transformation, `JordanDecomposition`, 32
  `SchurDecomposition`, 32
Solution of equations, symbolic, `Solve`, 28
`Solve`, 28
Solving equations, `Solve`, 28
`Sort`, 11
Sorting, of strings, `Sort`, 11
Spaces, in string patterns, 15
Splitting strings, `StringSplit`, 10, 57
Spreadsheets, exporting, `Export`, 4, 44
  importing, `Import`, 4, 48
Square wave, `Piecewise`, 25, 49
`StartOfLine`, 16
`StartOfString`, 16
Step function, `Piecewise`, 25, 49
String patterns, 11
`StringCases`, 12, 52
`StringCount`, 9, 12, 53
`StringDrop`, 53
`StringExpression`, 12, 54
`StringFreeQ`, 9, 12, 55
`StringInsert`, 55
`StringLength`, 55
`StringMatchQ`, 12, 55
`StringPosition`, 12, 16, 56
`StringReplace`, 10, 12, 16, 56
`StringReplaceList`, 10, 12, 57
`StringReverse`, 57
Strings, binary formats for, 22
`StringSplit`, 10, 12, 57
`StringTake`, 58
`Subsets`, 1, 58
Substitution, in strings, `StringReplace`, 10, 56

\[Sum] (∑), 3
Switching, Piecewise, 25, 49
syslog daemon, 34
System files, 35

Tab-delimited data, exporting, Import, 4, 48
  importing, Import, 4, 48
Tabs, in string patterns, 15
Tabular data, exporting, Export, 4, 44
  importing, Import, 4, 48
Tensor derivative, D, 29, 41
Terrain data, importing, Import, 4, 48
Test, for substrings, StringFreeQ, 9, 55
Tilde function markers, 36
Timeout, for network licenses, 34
ToExpression, 5
Tr, 31
\[Trademark] (™), 61
Translation, of strings, StringReplace, 10, 56
Transpose, 31, 58
\[Transpose] ($^{\mathsf{T}}$), 61
Triangle wave, Piecewise, 25, 49
Triples, in lists, Tuples, 2, 59
TSV format, exporting, Export, 4, 44
  importing, Import, 4, 48
Tuples, 2, 8, 59
Two-level logic, LogicalExpand, 26

Union, 1
Unique elements in lists, Union, 1
Unsigned integer binary format, 22
Updates, to *Mathematica*, iii
Upper case, ignoring in string operations, 16
URLs, Import, 5, 48

Vector derivative, D, 29, 41
Version, new features in current, iii
Volumes, of regions, Integrate, 29

Web import, Import, 5, 48
White, 59
Whitespace, StringSplit, 11, 57
Whitespace, 13
WhitespaceCharacter, 14
WordBoundary, 16
WordCharacter, 14
Writing, binary data, BinaryWrite, 21, 39

XLS format, exporting, Export, 4, 44
  importing, Import, 4, 48

Yellow, 59

$InstallationDirectory, 35
$MaxPiecewiseCases, 27, 59

Comments on this document are welcomed at comments@wolfram.com.

© 2004 Wolfram Research, Inc. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the copyright holder. Wolfram Research is the holder of the copyright to the *Mathematica* software system described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, “look and feel”, programming language, and compilation of command names. Use of the system unless pursuant to the terms of a license granted by Wolfram Research or as otherwise authorized by law is an infringement of the copyright.

Wolfram Research, Inc. makes no representations, express or implied, with respect to the product, including without limitations, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which Wolfram Research is willing to license *Mathematica* is a provision that Wolfram Research and its distribution licensees, distributors, and dealers shall in no event be liable for any indirect, incidental, or consequential damages, and that liability for direct damages shall be limited to the amount of the purchase price paid for *Mathematica*. In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. Wolfram Research shall not be responsible under any circumstances for providing information on or corrections to errors and omissions discovered at any time in this document or the software it describes, whether or not they are aware of the errors or omissions. Wolfram Research does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury, or significant loss.

Mathematica and *MathLink* are registered trademarks of Wolfram Research, Inc. *J/Link*, *.NET/Link*, *MathLM*, and *MonitorLM* are trademarks of Wolfram Research. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Macintosh and Apple are registered trademarks of Apple Computer, Inc. Motif, Unix, and the “X” device are registered trademarks and The Open Group is a trademark of The Open Group in the United States and other countries. All other product names used herein are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc. or MathTech, Inc.