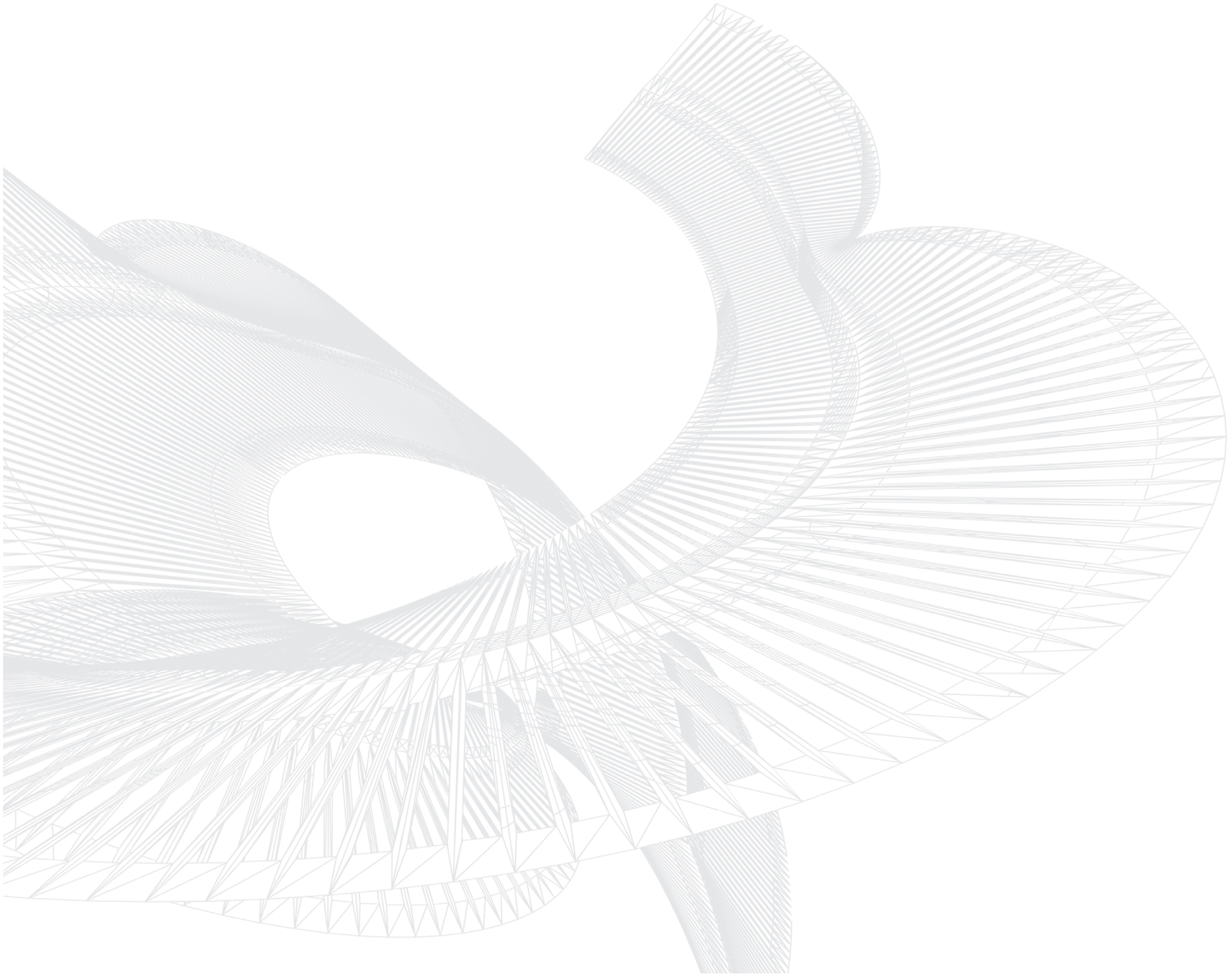
Wolfram *Mathematica*® Tutorial Collection

CORE LANGUAGE



For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Building Up Calculations

Using Previous Results

In doing calculations, you will often need to use previous results that you have got. In *Mathematica*, `%` always stands for your last result.

`%`	the last result generated
`%%`	the next-to-last result
`%%…%` (*k* times)	the kth previous result
`%`*n*	the result on output line `Out[`*n*`]` (to be used with care)

Ways to refer to your previous results.

Here is the first result.

```
In[1]:= 77^2
Out[1]= 5929
```

This adds 1 to the last result.

```
In[2]:= % + 1
Out[2]= 5930
```

This uses both the last result, and the result before that.

```
In[3]:= 3 % + %^2 + %%
Out[3]= 35 188 619
```

You will have noticed that all the input and output lines in *Mathematica* are numbered. You can use these numbers to refer to previous results.

This adds the results on lines 2 and 3 above.

```
In[4]:= %2 + %3
Out[4]= 35 194 549
```

If you use a text-based interface to *Mathematica*, then successive input and output lines will always appear in order. However, if you use a notebook interface to *Mathematica*, as discussed in "Notebook Interfaces", then successive input and output lines need not appear in order. You can for example "scroll back" and insert your next calculation wherever you want in the notebook. You should realize that `%` is always defined to be the last result that *Mathematica* generated. This may or may not be the result that appears immediately above your present position in the notebook. With a notebook interface, the only way to tell when a particular result was generated is to look at the `Out[`*n*`]` label that it has. Because you can insert and delete anywhere in a notebook, the textual ordering of results in a notebook need have no relation to the order in which the results were generated.

Defining Variables

When you do long calculations, it is often convenient to give *names* to your intermediate results. Just as in standard mathematics, or in other computer languages, you can do this by introducing named *variables*.

This sets the value of the *variable* `x` to be `5`.

```
In[1]:= x = 5
Out[1]= 5
```

Whenever `x` appears, *Mathematica* now replaces it with the value `5`.

```
In[2]:= x^2
Out[2]= 25
```

This assigns a new value to `x`.

```
In[3]:= x = 7 + 4
Out[3]= 11
```

`pi` is set to be the numerical value of π to 40-digit accuracy.

```
In[4]:= pi = N[Pi, 40]
Out[4]= 3.141592653589793238462643383279502884197
```

Here is the value you defined for `pi`.

```
In[5]:= pi
Out[5]= 3.1415926535897932384626433832795028841971
```

This gives the numerical value of π^2, to the same accuracy as `pi`.

```
In[6]:= pi^2
Out[6]= 9.869604401089358618834490999876151135313
```

x=*value*	assign a value to the variable x
x=y=*value*	assign a value to both x and y
x=. or `Clear[`x`]`	remove any value assigned to x

Assigning values to variables.

It is very important to realize that values you assign to variables are *permanent*. Once you have assigned a value to a particular variable, the value will be kept until you explicitly remove it. The value will, of course, disappear if you start a whole new *Mathematica* session.

Forgetting about definitions you made earlier is the single most common cause of mistakes when using *Mathematica*. If you set `x = 5`, *Mathematica* assumes that you *always* want `x` to have the value `5`, until or unless you explicitly tell it otherwise. To avoid mistakes, you should remove values you have defined as soon as you have finished using them.

- Remove values you assign to variables as soon as you finish using them.

A useful principle in using *Mathematica*.

The variables you define can have almost any name. There is no limit on the length of their names. One constraint, however, is that variable names can never *start* with numbers. For example, `x2` could be a variable, but `2 x` means `2 * x`.

Mathematica uses both uppercase and lowercase letters. There is a convention that built-in *Mathematica* objects always have names starting with uppercase (capital) letters. To avoid confusion, you should always choose names for your own variables that start with lowercase letters.

aaaaa	a variable name containing only lowercase letters
Aaaaa	a built-in object whose name begins with a capital letter

Naming conventions.

You can type formulas involving variables in *Mathematica* almost exactly as you would in mathematics. There are a few important points to watch, however.

- `x y` means `x` times `y`.
- `xy` with no space is the variable with name `xy`.
- `5 x` means `5` times `x`.
- `x^2 y` means `(x^2) y`, not `x^(2 y)`.

Some points to watch when using variables in *Mathematica*.

Values for Symbols

When *Mathematica* transforms an expression such as `x + x` into `2 x`, it is treating the variable `x` in a purely symbolic or formal fashion. In such cases, `x` is a symbol which can stand for any expression.

Often, however, you need to replace a symbol like `x` with a definite “value”. Sometimes this value will be a number; often it will be another expression.

To take an expression such as `1 + 2 x` and replace the symbol `x` that appears in it with a definite value, you can create a *Mathematica* transformation rule, and then apply this rule to the expression. To replace `x` with the value `3`, you would create the transformation rule `x -> 3`. You must type `->` as a pair of characters, with no space in between. You can think of `x -> 3` as being a rule in which “`x` goes to `3`”.

To apply a transformation rule to a particular *Mathematica* expression, you type *expr* `/.` *rule*. The “replacement operator” `/.` is typed as a pair of characters, with no space in between.

This uses the transformation rule `x -> 3` in the expression `1 + 2 x`.

```
In[1]:=  1 + 2 x /. x -> 3
Out[1]=  7
```

You can replace `x` with any expression. Here every occurrence of `x` is replaced by `2 - y`.

```
In[2]:=  1 + x + x^2 /. x -> 2 - y

Out[2]=  3 + (2 - y)^2 - y
```

Here is a transformation rule. *Mathematica* treats it like any other symbolic expression.

```
In[3]:=  x -> 3 + y

Out[3]=  x → 3 + y
```

This applies the transformation rule on the previous line to the expression `x^2 - 9`.

```
In[4]:=  x^2 - 9 /. %

Out[4]=  -9 + (3 + y)^2
```

expr `/.` *x*`->`*value*	replace *x* by *value* in the expression *expr*
expr `/.` `{`*x*`->`*xval*`,` *y*`->`*yval*`}`	perform several replacements

Replacing symbols by values in expressions.

You can apply rules together by putting the rules in a list.

```
In[5]:=  (x + y) (x - y)^2 /. {x -> 3, y -> 1 - a}

Out[5]=  (4 - a) (2 + a)^2
```

The replacement operator `/.` allows you to apply transformation rules to a particular expression. Sometimes, however, you will want to define transformation rules that should *always* be applied. For example, you might want to replace `x` with `3` whenever `x` occurs.

As discussed in "Defining Variables", you can do this by *assigning* the value `3` to `x` using `x = 3`. Once you have made the assignment `x = 3`, `x` will always be replaced by `3`, whenever it appears.

This assigns the value `3` to `x`.

```
In[6]:=  x = 3

Out[6]=  3
```

Now `x` will automatically be replaced by `3` wherever it appears.

```
In[7]:=  x^2 - 1

Out[7]=  8
```

This assigns the expression 1 + a to be the value of x.

```
In[8]:=  x = 1 + a
Out[8]=  1 + a
```

Now x is replaced by 1 + a.

```
In[9]:=  x^2 - 1
```
Out[9]= $-1 + (1 + a)^2$

You can define the value of a symbol to be any expression, not just a number. You should realize that once you have given such a definition, the definition will continue to be used whenever the symbol appears, until you explicitly change or remove the definition. For most people, forgetting to remove values you have assigned to symbols is the single most common source of mistakes in using *Mathematica*.

x=*value*	define a value for *x* which will always be used
x=.	remove any value defined for *x*

Assigning values to symbols.

The symbol x still has the value you assigned to it.

```
In[10]:=  x + 5 - 2 x
Out[10]=  6 + a - 2 (1 + a)
```

This removes the value you assigned to x.

```
In[11]:=  x =.
```

Now x has no value defined, so it can be used as a purely symbolic variable.

```
In[12]:=  x + 5 - 2 x
Out[12]=  5 - x
```

A symbol such as x can serve many different purposes in *Mathematica*, and in fact, much of the flexibility of *Mathematica* comes from being able to mix these purposes at will. However, you need to keep some of the different uses of x straight in order to avoid making mistakes. The most important distinction is between the use of x as a name for another expression, and as a symbolic variable that stands only for itself.

Traditional programming languages that do not support symbolic computation allow variables to be used only as names for objects, typically numbers, that have been assigned as values for them. In *Mathematica*, however, `x` can also be treated as a purely formal variable, to which various transformation rules can be applied. Of course, if you explicitly give a definition, such as `x = 3`, then `x` will always be replaced by `3`, and can no longer serve as a formal variable.

You should understand that explicit definitions such as `x = 3` have a global effect. On the other hand, a replacement such as *expr* `/. x -> 3` affects only the specific expression *expr*. It is usually much easier to keep things straight if you avoid using explicit definitions except when absolutely necessary.

You can always mix replacements with assignments. With assignments, you can give names to expressions in which you want to do replacements, or to rules that you want to use to do the replacements.

This assigns a value to the symbol `t`.

```
In[13]:= t = 1 + x^2
```

Out[13]= $1+x^2$

This finds the value of `t`, and then replaces `x` by `2` in it.

```
In[14]:= t /. x -> 2

Out[14]= 5
```

This finds the value of `t` for a different value of `x`.

```
In[15]:= t /. x -> 5 a
```

Out[15]= $1+25\,a^2$

This finds the value of `t` when `x` is replaced by `Pi`, and then evaluates the result numerically.

```
In[16]:= t /. x -> Pi // N

Out[16]= 10.8696
```

The Four Kinds of Bracketing in *Mathematica*

There are four kinds of bracketing used in *Mathematica*. Each kind of bracketing has a very different meaning. It is important that you remember all of them.

(*term*)	parentheses for grouping
f[*x*]	square brackets for functions
{*a*,*b*,*c*}	curly braces for lists
v[[*i*]]	double brackets for indexing (`Part`[*v*, *i*])

The four kinds of bracketing in *Mathematica*.

When the expressions you type in are complicated, it is often a good idea to put extra space inside each set of brackets. This makes it somewhat easier for you to see matching pairs of brackets. *v*[[{*a*, *b*}]] is, for example, easier to recognize than *v*[[{*a*, *b*}]].

Sequences of Operations

In doing a calculation with *Mathematica*, you usually go through a sequence of steps. If you want to, you can do each step on a separate line. Often, however, you will find it convenient to put several steps on the same line. You can do this simply by separating the pieces of input you want to give with semicolons.

$expr_1$;$expr_2$;$expr_3$	do several operations, and give the result of the last one
$expr_1$;$expr_2$;	do the operations, but print no output

Ways to do sequences of operations in *Mathematica*.

This does three operations on the same line. The result is the result from the last operation.

```
In[1]:=  x = 4; y = 6; z = y + 6
Out[1]= 12
```

If you end your input with a semicolon, it is as if you are giving a sequence of operations, with an "empty" one at the end. This has the effect of making *Mathematica* perform the operations you specify, but display no output.

expr`;`	do an operation, but display no output

Inhibiting output.

Putting a semicolon at the end of the line tells *Mathematica* to show no output.

```
In[2]:= x = 67 - 5;
```

You can still use `%` to get the output that would have been shown.

```
In[3]:= %
Out[3]= 62
```

Lists

Making Lists of Objects

In doing calculations, it is often convenient to collect together several objects, and treat them as a single entity. *Lists* give you a way to make collections of objects in *Mathematica*. As you will see later, lists are very important and general structures in *Mathematica*.

A list such as `{3, 5, 1}` is a collection of three objects. But in many ways, you can treat the whole list as a single object. You can, for example, do arithmetic on the whole list at once, or assign the whole list to be the value of a variable.

Here is a list of three numbers.

```
In[1]:= {3, 5, 1}
Out[1]= {3, 5, 1}
```

This squares each number in the list, and adds `1` to it.

```
In[2]:= {3, 5, 1}^2 + 1
Out[2]= {10, 26, 2}
```

This takes differences between corresponding elements in the two lists. The lists must be the same length.

```
In[3]:= {6, 7, 8} - {3.5, 4, 2.5}
Out[3]= {2.5, 3, 5.5}
```

The value of `%` is the whole list.

```
In[4]:= %
Out[4]= {2.5, 3, 5.5}
```

You can apply any of the mathematical functions in "Some Mathematical Functions" to whole lists.

```
In[5]:= Exp[%] // N
Out[5]= {12.1825, 20.0855, 244.692}
```

Just as you can set variables to be numbers, so also you can set them to be lists.

This assigns v to be a list.

```
In[6]:= v = {2, 4, 3.1}
Out[6]= {2, 4, 3.1}
```

Wherever v appears, it is replaced by the list.

```
In[7]:= v / (v - 1)
```

Out[7]= $\left\{2, \frac{4}{3}, 1.47619\right\}$

Collecting Objects Together

We first encountered lists in "Making Lists of Objects" as a way of collecting numbers together. Here, we shall see many different ways to use lists. You will find that lists are some of the most flexible and powerful objects in *Mathematica*. You will see that lists in *Mathematica* represent generalizations of several standard concepts in mathematics and computer science.

At a basic level, what a *Mathematica* list essentially does is to provide a way for you to collect together several expressions of any kind.

Here is a list of numbers.

```
In[1]:= {2, 3, 4}
Out[1]= {2, 3, 4}
```

This gives a list of symbolic expressions.

```
In[2]:= x^% - 1
```

Out[2]= $\{-1 + x^2, -1 + x^3, -1 + x^4\}$

You can differentiate these expressions.

```
In[3]:= D[%, x]
```

Out[3]= $\{2\,x, 3\,x^2, 4\,x^3\}$

And then you can find values when x is replaced with 3.

```
In[4]:= % /. x -> 3
Out[4]= {6, 27, 108}
```

The mathematical functions that are built into *Mathematica* are mostly set up to be "listable" so that they act separately on each element of a list. This is, however, not true of all functions in *Mathematica*. Unless you set it up specially, a new function `f` that you introduce will treat lists just as single objects. "Applying Functions to Parts of Expressions" and "Structural Operations" will describe how you can use `Map` and `Thread` to apply a function like this separately to each element in a list.

Making Tables of Values

You can use lists as tables of values. You can generate the tables, for example, by evaluating an expression for a sequence of different parameter values.

This gives a table of the values of i^2, with i running from 1 to 6.

```
In[1]:= Table[i^2, {i, 6}]
Out[1]= {1, 4, 9, 16, 25, 36}
```

Here is a table of $\sin(n/5)$ for n from 0 to 4.

```
In[2]:= Table[Sin[n / 5], {n, 0, 4}]
```

Out[2]= $\left\{0, \text{Sin}\left[\frac{1}{5}\right], \text{Sin}\left[\frac{2}{5}\right], \text{Sin}\left[\frac{3}{5}\right], \text{Sin}\left[\frac{4}{5}\right]\right\}$

This gives the numerical values.

```
In[3]:= N[%]
Out[3]= {0., 0.198669, 0.389418, 0.564642, 0.717356}
```

You can also make tables of formulas.

```
In[4]:= Table[x^i + 2 i, {i, 5}]
```

Out[4]= $\left\{2 + x, 4 + x^2, 6 + x^3, 8 + x^4, 10 + x^5\right\}$

`Table` uses exactly the same iterator notation as the functions `Sum` and `Product`, which are discussed in "Sums and Products".

```
In[5]:= Product[x^i + 2 i, {i, 5}]
```

Out[5]= $(2 + x)\left(4 + x^2\right)\left(6 + x^3\right)\left(8 + x^4\right)\left(10 + x^5\right)$

This makes a table with values of `x` running from `0` to `1` in steps of `0.25`.

```
In[6]:=  Table[Sqrt[x], {x, 0, 1, 0.25}]
Out[6]=  {0, 0.5, 0.707107, 0.866025, 1.}
```

You can perform other operations on the lists you get from `Table`.

```
In[7]:=  %^2 + 3
Out[7]=  {3, 3.25, 3.5, 3.75, 4.}
```

`TableForm` displays lists in a "tabular" format. Notice that both words in the name `TableForm` begin with capital letters.

```
In[8]:=  % // TableForm
                    3
                    3.25
Out[8]//TableForm=  3.5
                    3.75
                    4.
```

All the examples so far have been of tables obtained by varying a single parameter. You can also make tables that involve several parameters. These multidimensional tables are specified using the standard *Mathematica* iterator notation, discussed in "Sums and Products".

This makes a table of $x^i + y^j$ with i running from 1 to 3 and j running from 1 to 2.

```
In[9]:=  Table[x^i + y^j, {i, 3}, {j, 2}]
```

Out[9]= $\{\{x+y, x+y^2\}, \{x^2+y, x^2+y^2\}, \{x^3+y, x^3+y^2\}\}$

The table in this example is a *list of lists*. The elements of the outer list correspond to successive values of i. The elements of each inner list correspond to successive values of j, with i fixed.

Sometimes you may want to generate a table by evaluating a particular expression many times, without incrementing any variables.

This creates a list containing four copies of the symbol `x`.

```
In[10]:=  Table[x, {4}]
Out[10]=  {x, x, x, x}
```

This gives a list of four pairs of numbers sampled from `{1, 2, 3, 4}`. `Table` re-evaluates `RandomSample[{1, 2, 3, 4}, 2]` for each element in the list, so that you get four different samples.

```
In[11]:= Table[RandomSample[{1, 2, 3, 4}, 2], {4}]
Out[11]= {{3, 2}, {4, 2}, {4, 3}, {2, 1}}
```

This evaluates $\sqrt{i}$ for each of the values of *i* in the list `{1, 4, 9, 16}`.

```
In[12]:= Table[√i, {i, {1, 4, 9, 16}}]
Out[12]= {1, 2, 3, 4}
```

This creates a 3×2 table.

```
In[13]:= Table[i + 2 j, {i, 3}, {j, 2}]
Out[13]= {{3, 5}, {4, 6}, {5, 7}}
```

In this table, the length of the rows depends on the more slowly varying iterator variable, *i*.

```
In[14]:= Table[i + 2 j, {i, 3}, {j, i}]
Out[14]= {{3}, {4, 6}, {5, 7, 9}}
```

You can use `Table` to generate arrays with any number of dimensions.

This generates a three-dimensional 2×2×2 array. It is a list of lists of lists.

```
In[15]:= Table[i j^2 k^3, {i, 2}, {j, 2}, {k, 2}]
Out[15]= {{{1, 8}, {4, 32}}, {{2, 16}, {8, 64}}}
```

`Table`[f, {i_{max}}]	give a list of i_{max} values of f
`Table`[f, {i, i_{max}}]	give a list of the values of f as i runs from 1 to i_{max}
`Table`[f, {i, i_{min}, i_{max}}]	give a list of values with i running from i_{min} to i_{max}
`Table`[f, {i, i_{min}, i_{max}, di}]	use steps of di
`Table`[f, {i, i_{min}, i_{max}}, {j, j_{min}, j_{max}}, …]	generate a multidimensional table
`Table`[f, {i, {i_1, i_2, …}}]	give a list of the values of f as i successively takes the values i_1, i_2, …
`TableForm`[*list*]	display a list in tabular form

Functions for generating tables.

You can use the operations discussed in "Manipulating Elements of Lists" to extract elements of the table.

This creates a table and gives it the name `sq`.

```
In[16]:= sq = Table[j^2, {j, 7}]
Out[16]= {1, 4, 9, 16, 25, 36, 49}
```

This gives the third part of the table.

```
In[17]:= sq[[3]]
Out[17]= 9
```

This gives a list of the third through fifth parts.

```
In[18]:= sq[[3 ;; 5]]
Out[18]= {9, 16, 25}
```

This creates a 2×2 table, and gives it the name `m`.

```
In[19]:= m = Table[i - j, {i, 2}, {j, 2}]
Out[19]= {{0, -1}, {1, 0}}
```

This extracts the first sublist from the list of lists that makes up the table.

```
In[20]:= m[[1]]
Out[20]= {0, -1}
```

This extracts the second element of that sublist.

```
In[21]:= %[[2]]
Out[21]= -1
```

This does the two operations together.

```
In[22]:= m[[1, 2]]
Out[22]= -1
```

This displays `m` in a "tabular" form.

```
In[23]:= TableForm[m]
                        0  -1
Out[23]//TableForm=     1  0
```

`t[[i]]` or `Part[t,i]`	give the i^{th} sublist in t (also input as $t[\![i]\!]$)
`t[[i;;j]]` or `Part[t,i;;j]`	give a list of the parts i through j
`t[[{i1,i2,...}]]` or `Part[t,{i1,i2,...}]`	give a list of the ${i_1}^{\text{th}}$, ${i_2}^{\text{th}}$, … parts of t
`t[[i,j,...]]` or `Part[t,i,j,...]`	give the part of t corresponding to `t[[i]][[j]]` …

Ways to extract parts of tables.

As mentioned in "Manipulating Elements of Lists", you can think of lists in *Mathematica* as being analogous to "arrays". Lists of lists are then like two-dimensional arrays. When you lay them out in a tabular form, the two indices of each element are like its x and y coordinates.

Manipulating Elements of Lists

Many of the most powerful list manipulation operations in *Mathematica* treat whole lists as single objects. Sometimes, however, you need to pick out or set individual elements in a list.

You can refer to an element of a *Mathematica* list by giving its "index". The elements are numbered in order, starting at 1.

`{a,b,c}`	a list
`Part[list,i]` or `list[[i]]`	the i^{th} element of *list* (the first element is `list[[1]]`)
`Part[list,{i,j,...}]` or `list[[{i,j,...}]]`	a list of the i^{th}, j^{th}, … elements of *list*
`Part[list,i;;j]`	a list of the i^{th} through j^{th} elements of *list*

Operations on list elements.

This extracts the second element of the list.

```
In[1]:= {5, 8, 6, 9}[[2]]
Out[1]= 8
```

This extracts a list of elements.

```
In[2]:= {5, 8, 6, 9}[[{3, 1, 3, 2, 4}]]
Out[2]= {6, 5, 6, 8, 9}
```

This assigns the value of `v` to be a list.

```
In[3]:= v = {2, 4, 7}
Out[3]= {2, 4, 7}
```

You can extract elements of `v`.

```
In[4]:= v[[2]]
Out[4]= 4
```

By assigning a variable to be a list, you can use *Mathematica* lists much like "arrays" in other computer languages. Thus, for example, you can reset an element of a list by assigning a value to *v*[[*i*]].

`Part[`*v*`,`*i*`]` or *v*`[[`*i*`]]`	extract the i^{th} element of a list
`Part[`*v*`,`*i*`]=`*value* or *v*`[[`*i*`]]=`*value*	reset the i^{th} element of a list

Array-like operations on lists.

Here is a list.

```
In[5]:= v = {4, -1, 8, 7}
Out[5]= {4, -1, 8, 7}
```

This resets the third element of the list.

```
In[6]:= v[[3]] = 0
Out[6]= 0
```

Now the list assigned to `v` has been modified.

```
In[7]:= v
Out[7]= {4, -1, 0, 7}
```

Vectors and Matrices

Vectors and matrices in *Mathematica* are simply represented by lists and by lists of lists, respectively.

$\{a,b,c\}$	vector (a, b, c)
$\{\{a,b\},\{c,d\}\}$	matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$

The representation of vectors and matrices by lists.

This is a 2×2 matrix.

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

Here is the first row.

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

Here is the element m_{12}.

```
In[3]:= m[[1, 2]]
Out[3]= b
```

This is a two-component vector.

```
In[4]:= v = {x, y}
Out[4]= {x, y}
```

The objects p and q are treated as scalars.

```
In[5]:= p v + q
Out[5]= {q + p x, q + p y}
```

Vectors are added component by component.

```
In[6]:= v + {xp, yp} + {xpp, ypp}
Out[6]= {x + xp + xpp, y + yp + ypp}
```

This gives the dot (scalar) product of two vectors.

```
In[7]:= {x, y}.{xp, yp}
Out[7]= x xp + y yp
```

You can also multiply a matrix by a vector.

```
In[8]:= m.v
Out[8]= {a x + b y, c x + d y}
```

Or a matrix by a matrix.

```
In[9]:= m.m
```

Out[9]= $\{\{a^2 + b\,c,\ a\,b + b\,d\},\ \{a\,c + c\,d,\ b\,c + d^2\}\}$

Or a vector by a matrix.

```
In[10]:= v.m
Out[10]= {a x + c y, b x + d y}
```

This combination makes a scalar.

```
In[11]:= v.m.v
Out[11]= x (a x + c y) + y (b x + d y)
```

Because of the way *Mathematica* uses lists to represent vectors and matrices, you never have to distinguish between "row" and "column" vectors.

`Table`[*f*,{*i*,*n*}]	build a length-*n* vector by evaluating *f* with *i* = 1, 2, …, *n*
`Array`[*a*,*n*]	build a length-*n* vector of the form {*a*[1], *a*[2], …}
`Range`[*n*]	create the list {1, 2, 3, …, *n*}
`Range`[n_1,n_2]	create the list {n_1, $n_1 + 1$, …, n_2}
`Range`[n_1,n_2,*dn*]	create the list {n_1, $n_1 + dn$, …, n_2}
list[[*i*]] or `Part`[*list*,*i*]	give the i^{th} element in the vector *list*
`Length`[*list*]	give the number of elements in *list*
c *v*	multiply a vector by a scalar
a.*b*	dot product of two vectors
`Cross`[*a*,*b*]	cross product of two vectors (also input as $a \times b$)
`Norm`[*v*]	Euclidean norm of a vector

Functions for vectors.

`Table[`*f*`,{`*i*`,`*m*`},{`*j*`,`*n*`}]`	build an $m\times n$ matrix by evaluating f with i ranging from 1 to m and j ranging from 1 to n
`Array[`*a*`,{`*m*`,`*n*`}]`	build an $m\times n$ matrix with i, jth element $a[i, j]$
`IdentityMatrix[`*n*`]`	generate an $n\times n$ identity matrix
`DiagonalMatrix[`*list*`]`	generate a square matrix with the elements in *list* on the main diagonal
list`[[`*i*`]]` or `Part[`*list*`,`*i*`]`	give the ith row in the matrix *list*
list`[[All,`*j*`]]` or `Part[`*list*`,All,`*j*`]`	
	give the jth column in the matrix *list*
list`[[`*i*`,`*j*`]]` or `Part[`*list*`,`*i*`,`*j*`]`	give the i, jth element in the matrix *list*
`Dimensions[`*list*`]`	give the dimensions of a matrix represented by *list*

Functions for matrices.

`Column[`*list*`]`	display the elements of *list* in a column
`MatrixForm[`*list*`]`	display *list* in matrix form

Formatting constructs for vectors and matrices.

This builds a 3×3 matrix s with elements $s_{ij} = i + j$.

```
In[12]:= s = Table[i + j, {i, 3}, {j, 3}]
Out[12]= {{2, 3, 4}, {3, 4, 5}, {4, 5, 6}}
```

This displays `s` in standard two-dimensional matrix format.

```
In[13]:= MatrixForm[s]
```

Out[13]//MatrixForm=
$$\begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{pmatrix}$$

This gives a vector with symbolic elements. You can use this in deriving general formulas that are valid with any choice of vector components.

```
In[14]:= Array[a, 4]
Out[14]= {a[1], a[2], a[3], a[4]}
```

This gives a 3×2 matrix with symbolic elements. "Building Lists from Functions" discusses how you can produce other kinds of elements with `Array`.

```
In[15]:= Array[p, {3, 2}]
Out[15]= {{p[1, 1], p[1, 2]}, {p[2, 1], p[2, 2]}, {p[3, 1], p[3, 2]}}
```

Here are the dimensions of the matrix on the previous line.

```
In[16]:= Dimensions[%]
Out[16]= {3, 2}
```

This generates a 3×3 diagonal matrix.

```
In[17]:= DiagonalMatrix[{a, b, c}]
Out[17]= {{a, 0, 0}, {0, b, 0}, {0, 0, c}}
```

c *m*	multiply a matrix by a scalar
a.*b*	dot product of two matrices
`Inverse[`*m*`]`	matrix inverse
`MatrixPower[`*m*`,`*n*`]`	*n*th power of a matrix
`Det[`*m*`]`	determinant
`Tr[`*m*`]`	trace
`Transpose[`*m*`]`	transpose
`Eigenvalues[`*m*`]`	eigenvalues
`Eigenvectors[`*m*`]`	eigenvectors

Some mathematical operations on matrices.

Here is the 2×2 matrix of symbolic variables that was defined.

```
In[18]:= m
Out[18]= {{a, b}, {c, d}}
```

This gives its determinant.

```
In[19]:= Det[m]
Out[19]= -b c + a d
```

Here is the transpose of `m`.

```
In[20]:= Transpose[m]
Out[20]= {{a, c}, {b, d}}
```

This gives the inverse of `m` in symbolic form.

```
In[21]:= Inverse[m]
```

$$\text{Out[21]=}\ \left\{\left\{\frac{d}{-b\,c+a\,d},\ -\frac{b}{-b\,c+a\,d}\right\},\ \left\{-\frac{c}{-b\,c+a\,d},\ \frac{a}{-b\,c+a\,d}\right\}\right\}$$

Here is a 3×3 rational matrix.

```
In[22]:= h = Table[1 / (i + j - 1), {i, 3}, {j, 3}]
```

Out[22]= $\left\{\left\{1, \frac{1}{2}, \frac{1}{3}\right\}, \left\{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right\}, \left\{\frac{1}{3}, \frac{1}{4}, \frac{1}{5}\right\}\right\}$

This gives its inverse.

```
In[23]:= Inverse[h]
Out[23]= {{9, -36, 30}, {-36, 192, -180}, {30, -180, 180}}
```

Taking the dot product of the inverse with the original matrix gives the identity matrix.

```
In[24]:= %.h
Out[24]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

Here is a 3×3 matrix.

```
In[25]:= r = Table[i + j + 1, {i, 3}, {j, 3}]
Out[25]= {{3, 4, 5}, {4, 5, 6}, {5, 6, 7}}
```

`Eigenvalues` gives the eigenvalues of the matrix.

```
In[26]:= Eigenvalues[r]
```

Out[26]= $\left\{\frac{1}{2}\left(15+\sqrt{249}\right), \frac{1}{2}\left(15-\sqrt{249}\right), 0\right\}$

This gives a numerical approximation to the matrix.

```
In[27]:= rn = N[r]
Out[27]= {{3., 4., 5.}, {4., 5., 6.}, {5., 6., 7.}}
```

Here are numerical approximations to the eigenvalues.

```
In[28]:= Eigenvalues[rn]
```

Out[28]= $\{15.3899, -0.389867, -1.49955\times10^{-16}\}$

"Linear Algebra in *Mathematica*" discusses many other matrix operations that are built into *Mathematica*.

Getting Pieces of Lists

`First[`*list*`]`	the first element in *list*
`Last[`*list*`]`	the last element
`Part[`*list*`,`*n*`]` or *list*`[[`*n*`]]`	the n^{th} element
`Part[`*list*`,-`*n*`]` or *list*`[[-`*n*`]]`	the n^{th} element from the end
`Part[`*list*`,`*m*`;;`*n*`]`	elements m through n
`Part[`*list*`,{`n_1`,`n_2`,…}]` or *list*`[[{`n_1`,`n_2`,…}]]`	the list of elements at positions n_1, n_2, …

Picking out elements of lists.

We will use this list for the examples.

```
In[1]:= t = {a, b, c, d, e, f, g}
Out[1]= {a, b, c, d, e, f, g}
```

Here is the last element of `t`.

```
In[2]:= Last[t]
Out[2]= g
```

This gives the third element.

```
In[3]:= t[[3]]
Out[3]= c
```

This gives the list of elements 3 through 6.

```
In[4]:= t[[3 ;; 6]]
Out[4]= {c, d, e, f}
```

This gives a list of the first and fourth elements.

```
In[5]:= t[[{1, 4}]]
Out[5]= {a, d}
```

`Take[`*list*`,`*n*`]`	the first *n* elements in *list*
`Take[`*list*`,-`*n*`]`	the last *n* elements
`Take[`*list*`,{`*m*`,`*n*`}]`	elements *m* through *n* (inclusive)
`Rest[`*list*`]`	*list* with its first element dropped
`Drop[`*list*`,`*n*`]`	*list* with its first *n* elements dropped
`Most[`*list*`]`	*list* with its last element dropped
`Drop[`*list*`,-`*n*`]`	*list* with its last *n* elements dropped
`Drop[`*list*`,{`*m*`,`*n*`}]`	*list* with elements *m* through *n* dropped

Picking out sequences in lists.

This gives the first three elements of the list `t` defined above.

```
In[6]:= Take[t, 3]
Out[6]= {a, b, c}
```

This gives the last three elements.

```
In[7]:= Take[t, -3]
Out[7]= {e, f, g}
```

This gives elements 2 through 5 inclusive.

```
In[8]:= Take[t, {2, 5}]
Out[8]= {b, c, d, e}
```

This gives elements 3 through 7 in steps of 2.

```
In[9]:= Take[t, {3, 7, 2}]
Out[9]= {c, e, g}
```

This gives `t` with the first element dropped.

```
In[10]:= Rest[t]
Out[10]= {b, c, d, e, f, g}
```

This gives `t` with its first three elements dropped.

```
In[11]:= Drop[t, 3]
Out[11]= {d, e, f, g}
```

This gives `t` with only its third element dropped.

```
In[12]:= Drop[t, {3, 3}]
Out[12]= {a, b, d, e, f, g}
```

"Manipulating Expressions like Lists" shows how all the functions here can be generalized to work not only on lists, but on any *Mathematica* expressions.

The functions here allow you to pick out pieces that occur at particular positions in lists. "Finding Expressions That Match a Pattern" shows how you can use functions like `Select` and `Cases` to pick out elements of lists based not on their positions, but instead on their properties.

Testing and Searching List Elements

`Position`[*list*, *form*]	the positions at which *form* occurs in *list*
`Count`[*list*, *form*]	the number of times *form* appears as an element of *list*
`MemberQ`[*list*, *form*]	test whether *form* is an element of *list*
`FreeQ`[*list*, *form*]	test whether *form* occurs nowhere in *list*

Testing and searching for elements of lists.

"Getting Pieces of Lists" discusses how to extract pieces of lists based on their positions or indices. *Mathematica* also has functions that search and test for elements of lists, based on the values of those elements.

This gives a list of the positions at which `a` appears in the list.

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

`Count` counts the number of occurrences of `a`.

```
In[2]:= Count[{a, b, c, a, b}, a]
Out[2]= 2
```

This shows that `a` is an element of `{a, b, c}`.

```
In[3]:= MemberQ[{a, b, c}, a]
Out[3]= True
```

On the other hand, d is not.

```
In[4]:= MemberQ[{a, b, c}, d]
Out[4]= False
```

This assigns m to be the 3×3 identity matrix.

```
In[5]:= m = IdentityMatrix[3]
Out[5]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

This shows that 0 does occur *somewhere* in m.

```
In[6]:= FreeQ[m, 0]
Out[6]= False
```

This gives a list of the positions at which 0 occurs in m.

```
In[7]:= Position[m, 0]
Out[7]= {{1, 2}, {1, 3}, {2, 1}, {2, 3}, {3, 1}, {3, 2}}
```

As discussed in "Finding Expressions That Match a Pattern", the functions Count and Position, as well as MemberQ and FreeQ, can be used not only to search for *particular* list elements, but also to search for classes of elements which match specific “patterns”.

Adding, Removing and Modifying List Elements

Prepend[*list*,*element*]	add *element* at the beginning of *list*
Append[*list*,*element*]	add *element* at the end of *list*
Insert[*list*,*element*,*i*]	insert *element* at position *i* in *list*
Insert[*list*,*element*,−*i*]	insert at position *i* counting from the end of *list*
Riffle[*list*,*element*]	interleave *element* between the entries of *list*
Delete[*list*,*i*]	delete the element at position *i* in *list*
ReplacePart[*list*,*i*->*new*]	replace the element at position *i* in *list* with *new*
ReplacePart[*list*,{*i*,*j*}->*new*]	replace *list*[[*i*, *j*]] with *new*

Functions for manipulating elements in explicit lists.

This gives a list with x prepended.

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

This inserts x so that it becomes element number 2.

```
In[2]:= Insert[{a, b, c}, x, 2]
Out[2]= {a, x, b, c}
```

This interleaves x between the entries of the list.

```
In[3]:= Riffle[{a, b, c}, x]
Out[3]= {a, x, b, x, c}
```

This replaces the third element in the list with x.

```
In[4]:= ReplacePart[{a, b, c, d}, 3 -> x]
Out[4]= {a, b, x, d}
```

This replaces the 1, 2 element in a 2×2 matrix.

```
In[5]:= ReplacePart[{{a, b}, {c, d}}, {1, 2} -> x]
Out[5]= {{a, x}, {c, d}}
```

Functions like `ReplacePart` take explicit lists and give you new lists. Sometimes, however, you may want to modify a list "in place", without explicitly generating a new list.

$v=\{e_1, e_2, \ldots\}$	assign a variable to be a list
$v[[i]]=new$	assign a new value to the i^{th} element

Resetting list elements.

This defines v to be a list.

```
In[6]:= v = {a, b, c, d}
Out[6]= {a, b, c, d}
```

This sets the third element to be x.

```
In[7]:= v[[3]] = x
Out[7]= x
```

Now `v` has been changed.

```
In[8]:= v
Out[8]= {a, b, x, d}
```

m[[*i*,*j*]]=*new*	replace the $(i, j)^{\text{th}}$ element of a matrix
m[[*i*]]=*new*	replace the i^{th} row
m[[All,*i*]]=*new*	replace the i^{th} column

Resetting pieces of matrices.

This defines `m` to be a matrix.

```
In[9]:= m = {{a, b}, {c, d}}
Out[9]= {{a, b}, {c, d}}
```

This sets the first column of the matrix.

```
In[10]:= m[[All, 1]] = {x, y}; m
Out[10]= {{x, b}, {y, d}}
```

This sets every element in the first column to be `0`.

```
In[11]:= m[[All, 1]] = 0; m
Out[11]= {{0, b}, {0, d}}
```

Combining Lists

Join[$list_1$,$list_2$,…]	concatenate lists together
Union[$list_1$,$list_2$,…]	combine lists, removing repeated elements and sorting the result
Riffle[$list_1$,$list_2$]	interleave elements of $list_1$ and $list_2$

Functions for combining lists.

`Join` concatenates any number of lists together.

```
In[1]:= Join[{a, b, c}, {x, y}, {t, u}]
Out[1]= {a, b, c, x, y, t, u}
```

`Union` combines lists, keeping only distinct elements.

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

`Riffle` combines lists by interleaving their elements.

```
In[3]:= Riffle[{a, b, c}, {x, y, z}]
Out[3]= {a, x, b, y, c, z}
```

Lists as Sets

Mathematica usually keeps the elements of a list in exactly the order you originally entered them. If you want to treat a *Mathematica* list like a mathematical *set*, however, you may want to ignore the order of elements in the list.

`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*
`DeleteDuplicates` [*list*]	delete all duplicates from *list*

Set theoretical functions.

`Union` gives the elements that occur in *any* of the lists.

```
In[1]:= Union[{c, a, b}, {d, a, c}, {a, e}]
Out[1]= {a, b, c, d, e}
```

`Intersection` gives only elements that occur in *all* the lists.

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

`Complement` gives elements that occur in the first list, but not in any of the others.

```
In[3]:= Complement[{a, b, c, d}, {a, d}]
Out[3]= {b, c}
```

This gives all the subsets of the list.

```
In[4]:= Subsets[{a, b, c}]
Out[4]= {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}
```

`DeleteDuplicates` deletes all duplicate elements from the list.

```
In[5]:= DeleteDuplicates[{a, b, c, a}]
Out[5]= {a, b, c}
```

Rearranging Lists

`Sort[`*list*`]`	sort the elements of *list* into a standard order
`Union[`*list*`]`	sort elements, removing any duplicates
`Reverse[`*list*`]`	reverse the order of elements in *list*
`RotateLeft[`*list*`,`*n*`]`	rotate the elements of *list* *n* places to the left
`RotateRight[`*list*`,`*n*`]`	rotate *n* places to the right

Functions for rearranging lists.

This sorts the elements of a list into a standard order. In simple cases like this, the order is alphabetical or numerical.

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

This sorts the elements, removing any duplicates.

```
In[2]:= Union[{b, a, c, a, b}]
Out[2]= {a, b, c}
```

This rotates (“shifts”) the elements in the list two places to the left.

```
In[3]:= RotateLeft[{a, b, c, d, e}, 2]
Out[3]= {c, d, e, a, b}
```

You can rotate to the right by giving a negative displacement, or by using `RotateRight`.

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

`PadLeft[`*list*`,`*len*`,`*x*`]`	pad *list* on the left with *x* to make it length *len*
`PadRight[`*list*`,`*len*`,`*x*`]`	pad *list* on the right

Padding lists.

This pads a list with x's to make it length 10.

```
In[5]:= PadLeft[{a, b, c}, 10, x]
Out[5]= {x, x, x, x, x, x, x, a, b, c}
```

Grouping and Combining Elements of Lists

`Partition[`*list*`,`*n*`]`	partition *list* into *n*-element pieces
`Partition[`*list*`,`*n*`,`*d*`]`	use offset *d* for successive pieces
`Split[`*list*`]`	split *list* into pieces consisting of runs of identical elements

Functions for grouping together elements of lists.

Here is a list.

```
In[1]:= t = {a, b, c, d, e, f, g}
Out[1]= {a, b, c, d, e, f, g}
```

This groups the elements of the list in pairs, throwing away the single element left at the end.

```
In[2]:= Partition[t, 2]
Out[2]= {{a, b}, {c, d}, {e, f}}
```

This groups elements in triples. There is no overlap between the triples.

```
In[3]:= Partition[t, 3]
Out[3]= {{a, b, c}, {d, e, f}}
```

This makes triples of elements, with each successive triple offset by just one element.

```
In[4]:= Partition[t, 3, 1]
Out[4]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e, f}, {e, f, g}}
```

This splits up the list into runs of identical elements.

```
In[5]:= Split[{a, a, b, b, b, a, a, a, b}]
Out[5]= {{a, a}, {b, b, b}, {a, a, a}, {b}}
```

`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[6]:= Tuples[{a, b}, 2]
Out[6]= {{a, a}, {a, b}, {b, a}, {b, b}}
```

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

```
In[7]:= Tuples[{{a, b}, {1, 2, 3}}]
Out[7]= {{a, 1}, {a, 2}, {a, 3}, {b, 1}, {b, 2}, {b, 3}}
```

Ordering in Lists

`Sort[`*list*`]`	sort the elements of *list* into order
`Ordering[`*list*`]`	the positions in *list* of the elements in `Sort[`*list*`]`
`Ordering[`*list*`,`*n*`]`	the first *n* elements of `Ordering[`*list*`]`
`Ordering[`*list*`,-`*n*`]`	the last *n* elements of `Ordering[`*list*`]`
`Permutations[`*list*`]`	all possible orderings of *list*
`Min[`*list*`]`	the smallest element in *list*
`Max[`*list*`]`	the largest element in *list*

Ordering in lists.

Here is a list of numbers.

```
In[1]:= t = {17, 21, 14, 9, 18}
Out[1]= {17, 21, 14, 9, 18}
```

This gives the elements of `t` in sorted order.

```
In[2]:= Sort[t]
Out[2]= {9, 14, 17, 18, 21}
```

This gives the positions of the elements of `t`, from the position of the smallest to that of the largest.

```
In[3]:= Ordering[t]
Out[3]= {4, 3, 1, 5, 2}
```

This is the same as `Sort[t]`.

```
In[4]:= t[[%]]
Out[4]= {9, 14, 17, 18, 21}
```

This gives the smallest element in the list.

```
In[5]:= Min[t]
Out[5]= 9
```

Rearranging Nested Lists

You will encounter nested lists if you use matrices or generate multidimensional arrays and tables. *Mathematica* provides many functions for handling such lists.

`Flatten[`*list*`]`	flatten out all levels in *list*
`Flatten[`*list*`,`*n*`]`	flatten out the top n levels in *list*
`Partition[`*list*`,{`n_1`,`n_2`,…}]`	partition into blocks of size $n_1 \times n_2 \times \ldots$
`Transpose[`*list*`]`	interchange the top two levels of lists
`RotateLeft[`*list*`,{`n_1`,`n_2`,…}]`	rotate successive levels by n_i places
`PadLeft[`*list*`,{`n_1`,`n_2`,…}]`	pad successive levels to be length n_i

A few functions for rearranging nested lists.

This "flattens out" sublists. You can think of it as effectively just removing all inner braces.

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

This flattens out only one level of sublists.

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

There are many other operations you can perform on nested lists. More operations are discussed in "Manipulating Lists".

Manipulating Lists

Constructing Lists

Lists are widely used in *Mathematica*, and there are many ways to construct them.

`Range[`n`]`	the list $\{1, 2, 3, \ldots, n\}$
`Table[`$expr$`,{`i`,`n`}]`	the values of $expr$ with i from 1 to n
`Array[`f`,`n`]`	the list $\{f[1], f[2], \ldots, f[n]\}$
`NestList[`f`,`x`,`n`]`	$\{x, f[x], f[f[x]], \ldots\}$ with up to n nestings
`Normal[SparseArray[{`i_1`->`v_1`,…},`n`]]`	a length n list with element i_k being v_k
`Apply[List,`f`[`e_1`,`e_2`,…]]`	the list $\{e_1, e_2, \ldots\}$

Some explicit ways to construct lists.

This gives a table of the first five powers of two.

```
In[1]:= Table[2^i, {i, 5}]
Out[1]= {2, 4, 8, 16, 32}
```

Here is another way to get the same result.

```
In[2]:= Array[2^# &, 5]
Out[2]= {2, 4, 8, 16, 32}
```

This gives a similar list.

```
In[3]:= NestList[2 # &, 1, 5]
Out[3]= {1, 2, 4, 8, 16, 32}
```

`SparseArray` lets you specify values at particular positions.

```
In[4]:= Normal[SparseArray[{3 -> x, 4 -> y}, 5]]
Out[4]= {0, 0, x, y, 0}
```

You can also use patterns to specify values.

```
In[5]:= Normal[SparseArray[{i_ -> 2^i}, 5]]
Out[5]= {2, 4, 8, 16, 32}
```

Often you will know in advance how long a list is supposed to be, and how each of its elements should be generated. And often you may get one list from another.

`Table`[*expr*, {*i*, *list*}]	the values of *expr* with *i* taking on values from *list*
`Map`[*f*, *list*]	apply *f* to each element of *list*
`MapIndexed`[*f*, *list*]	give *f*[*elem*, {*i*}] for the *i*th 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*
`TakeWhile`[*list*, *test*]	give elements e_i from the beginning of *list* as long as *test*[e_i] is `True`
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 selects elements less than 5.

```
In[6]:= Select[{1, 3, 7, 4, 10, 2}, # < 5 &]
Out[6]= {1, 3, 4, 2}
```

This takes elements up to the first element that is not less than 5.

```
In[7]:= TakeWhile[{1, 3, 7, 4, 10, 2}, # < 5 &]
Out[7]= {1, 3}
```

This explicitly gives numbered parts.

```
In[8]:= {a, b, c, d}[[{2, 1, 4}]]
Out[8]= {b, a, d}
```

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

```
In[9]:= Pick[{a, b, c, d}, {1, 0, 1, 1}, 1]
Out[9]= {a, c, d}
```

Sometimes you may want to accumulate a list of results during the execution of a program. You can do this using `Sow` and `Reap`.

`Sow[`*val*`]`	sow the value *val* for the nearest enclosing `Reap`
`Reap[`*expr*`]`	evaluate *expr*, returning also a list of values sown by `Sow`

Using `Sow` and `Reap`.

This program iteratively squares a number.

```
In[10]:= Nest[#^2 &, 2, 6]
Out[10]= 18 446 744 073 709 551 616
```

This does the same computation, but accumulating a list of intermediate results above 1000.

```
In[11]:= Reap[Nest[(If[# > 1000, Sow[#]]; #^2) &, 2, 6]]
Out[11]= {18 446 744 073 709 551 616, {{65 536, 4 294 967 296}}}
```

An alternative but less efficient approach involves introducing a temporary variable, then starting with $t = \{\}$, and successively using `AppendTo[`*t*, *elem*`]`.

Manipulating Lists by Their Indices

`Part[`*list*`,`*spec*`]` or *list*`[[`*spec*`]]`	part or parts of a list
`Part[`*list*`,`$spec_1$`,`$spec_2$`,…]` or *list*`[[`$spec_1$`,`$spec_2$`,…]]`	part or parts of a nested list
n	the n^{th} part from the beginning
$-n$	the n^{th} part from the end
$\{i_1, i_2, \ldots\}$	a list of parts
m`;;`n	parts m through n
`All`	all parts

Getting parts of lists.

This gives a list of parts 1 and 3.

```
In[1]:= {a, b, c, d}[[{1, 3}]]
Out[1]= {a, c}
```

Here is a nested list.

```
In[2]:= m = {{a, b, c}, {d, e}, {f, g, h}};
```

This gives a list of its first and third parts.

```
In[3]:= m[[{1, 3}]]
Out[3]= {{a, b, c}, {f, g, h}}
```

This gives a list of the first part of each of these.

```
In[4]:= m[[{1, 3}, 1]]
Out[4]= {a, f}
```

And this gives a list of the first two parts.

```
In[5]:= m[[{1, 3}, {1, 2}]]
Out[5]= {{a, b}, {f, g}}
```

This gives the first two parts of `m`.

```
In[6]:= m[[1 ;; 2]]
Out[6]= {{a, b, c}, {d, e}}
```

This gives the last part of each of these.

```
In[7]:= m[[1 ;; 2, -1]]
Out[7]= {c, e}
```

This gives the second part of all sublists.

```
In[8]:= m[[All, 2]]
Out[8]= {b, e, g}
```

This gives the last two parts of all sublists.

```
In[9]:= m[[All, -2 ;; -1]]
Out[9]= {{b, c}, {d, e}, {g, h}}
```

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

This resets part 1,2 of `m`.

```
In[10]:= m[[1, 2]] = x
Out[10]= x
```

This is now the form of `m`.

```
In[11]:= m
Out[11]= {{a, x, c}, {d, e}, {f, g, h}}
```

This resets part 1 to `x` and part 3 to `y`.

```
In[12]:= m[[{1, 3}]] = {x, y}; m
Out[12]= {x, {d, e}, y}
```

This resets parts 1 and 3 both to `p`.

```
In[13]:= m[[{1, 3}]] = p; m
Out[13]= {p, {d, e}, p}
```

This restores the original form of `m`.

```
In[14]:= m = {{a, b, c}, {d, e}, {f, g, h}};
```

This now resets all parts specified by `m[[{1, 3}, {1, 2}]]`.

```
In[15]:= m[[{1, 3}, {1, 2}]] = x; m
Out[15]= {{x, x, c}, {d, e}, {x, x, h}}
```

You can use **`;;`** to indicate all indices in a given range.

```
In[16]:= m[[1 ;; 3, 2]] = y; m
Out[16]= {{x, y, c}, {d, y}, {x, y, h}}
```

It is sometimes useful to think of a nested list as being laid out in space, with each element being at a coordinate position given by its indices. There is then a direct geometrical interpretation for *list*[[$spec_1$, $spec_2$, …]]. If a given $spec_k$ is a single integer, then it represents extracting a single slice in the k^{th} dimension, while if it is a list, it represents extracting a list of parallel slices. The final result for *list*[[$spec_1$, $spec_2$, …]] is then the collection of elements obtained by slicing in each successive dimension.

Here is a nested list laid out as a two-dimensional array.

```
In[17]:= (m = {{a, b, c}, {d, e, f}, {g, h, i}}) // TableForm

                     a b c
Out[17]//TableForm=  d e f
                     g h i
```

This picks out rows 1 and 3, then columns 1 and 2.

```
In[18]:= m[[{1, 3}, {1, 2}]] // TableForm

Out[18]//TableForm=  a b
                     g h
```

`Part` is set up to make it easy to pick out structured slices of nested lists. Sometimes, however, you may want to pick out arbitrary collections of individual parts. You can do this conveniently with `Extract`.

`Part`[*list*, {i_1, i_2, …}]	the list {*list*[[i_1]], *list*[[i_2]], …}
`Extract`[*list*, {i_1, i_2, …}]	the element *list*[[i_1, i_2, …]]
`Part`[*list*, $spec_1$, $spec_2$, …]	parts specified by successive slicing
`Extract`[*list*, {{i_1, i_2, …}, {j_1, j_2, …}, …}]	the list of individual parts {*list*[[i_1, i_2, …]], *list*[[j_1, j_2, …]], …}

Getting slices versus lists of individual parts.

This extracts the individual parts 1,3 and 1,2.

```
In[19]:= Extract[m, {{1, 3}, {1, 2}}]
Out[19]= {c, b}
```

An important feature of `Extract` is that it takes lists of part positions in the same form as they are returned by functions like `Position`.

This sets up a nested list.

```
In[20]:= m = {{a[1], a[2], b[1]}, {b[2], c[1]}, {{b[3]}}};
```

This gives a list of positions in `m`.

```
In[21]:= Position[m, b[_]]
Out[21]= {{1, 3}, {2, 1}, {3, 1, 1}}
```

This extracts the elements at those positions.

```
In[22]:= Extract[m, %]
Out[22]= {b[1], b[2], b[3]}
```

`Take[`*list*`,`*spec*`]`	take the specified parts of a list
`Drop[`*list*`,`*spec*`]`	drop the specified parts of a list
`Take[`*list*`,`$spec_1$`,`$spec_2$`,…]`, `Drop[`*list*`,`$spec_1$`,`$spec_2$`,…]`	take or drop specified parts at each level in nested lists
n	the first *n* elements
−*n*	the last *n* elements
{*n*}	element *n* only
{*m*,*n*}	elements *m* through *n* (inclusive)
{*m*,*n*,*s*}	elements *m* through *n* in steps of *s*
`All`	all parts
`None`	no parts

Taking and dropping sequences of elements in lists.

This takes every second element starting at position 2.

```
In[23]:= Take[{a, b, c, d, e, f, g}, {2, -1, 2}]
Out[23]= {b, d, f}
```

This drops every second element.

```
In[24]:= Drop[{a, b, c, d, e, f, g}, {2, -1, 2}]
Out[24]= {a, c, e, g}
```

Much like `Part`, `Take` and `Drop` can be viewed as picking out sequences of slices at successive levels in a nested list. You can use `Take` and `Drop` to work with blocks of elements in arrays.

Here is a 3×3 array.

```
In[25]:= (m = {{a, b, c}, {d, e, f}, {g, h, i}}) // TableForm
                     a  b  c
Out[25]//TableForm=  d  e  f
                     g  h  i
```

Here is the first 2×2 subarray.

```
In[26]:= Take[m, 2, 2] // TableForm

Out[26]//TableForm= a b
                    d e
```

This takes all elements in the first two columns.

```
In[27]:= Take[m, All, 2] // TableForm

                    a b
Out[27]//TableForm= d e
                    g h
```

This leaves no elements from the first two columns.

```
In[28]:= Drop[m, None, 2] // TableForm

                    c
Out[28]//TableForm= f
                    i
```

`Prepend[`*list*`,`*elem*`]`	add *element* at the beginning of *list*
`Append[`*list*`,`*elem*`]`	add *element* at the end of *list*
`Insert[`*list*`,`*elem*`,`*i*`]`	insert *element* at position *i*
`Insert[`*list*`,`*elem*`,{`*i*`,`*j*`,…}]`	insert at position {*i*, *j*, …}
`Delete[`*list*`,`*i*`]`	delete the element at position *i*
`Delete[`*list*`,{`*i*`,`*j*`,…}]`	delete at position {*i*, *j*, …}

Adding and deleting elements in lists.

This makes the 2,1 element of the list be `x`.

```
In[29]:= Insert[{{a, b, c}, {d, e}}, x, {2, 1}]
Out[29]= {{a, b, c}, {x, d, e}}
```

This deletes the element again.

```
In[30]:= Delete[%, {2, 1}]
Out[30]= {{a, b, c}, {d, e}}
```

`ReplacePart[`*list*`,`*i*`->`*new*`]`	replace the element at position i in *list* with *new*
`ReplacePart[`*list*`,{`*i*`,`*j*`,…}->`*new*`]`	replace *list*`[[`*i*`, `*j*`, …]]` with *new*
`ReplacePart[`*list*`, {`i_1`->`new_1`,`i_2`->`new_2`,…}]`	replaces parts at positions i_n by new_n
`ReplacePart[`*list*`, {{`i_1`,`j_1`,…}->`new_1`,…}]`	replace parts at positions `{`i_n`, `j_n`, …}` by new_n
`ReplacePart[`*list*`, {{`i_1`,`j_1`,…},…}->`*new*`]`	replace all parts *list*`[[`i_k`, `j_k`, …]]` with *new*

Replacing parts of lists.

This replaces the third element in the list with `x`.

```
In[31]:= ReplacePart[{a, b, c, d}, 3 -> x]
Out[31]= {a, b, x, d}
```

This replaces the first and fourth parts of the list. Notice the need for double lists in specifying multiple parts to replace.

```
In[32]:= ReplacePart[{a, b, c, d}, {{1}, {4}} -> x]
Out[32]= {x, b, c, x}
```

Here is a 3×3 identity matrix.

```
In[33]:= IdentityMatrix[3]
Out[33]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

This replaces the 2,2 component of the matrix by `x`.

```
In[34]:= ReplacePart[%, {2, 2} -> x]
Out[34]= {{1, 0, 0}, {0, x, 0}, {0, 0, 1}}
```

It is important to understand that `ReplacePart` always creates a new list. It does not modify a list that has already been assigned to a symbol, the way *m*`[[…]] = `*val* does.

This assigns a list of values to `alist`.

```
In[35]:= alist = {a, b, c, d}
Out[35]= {a, b, c, d}
```

This gives a copy of the list in which the third element has been replaced with `x`.

```
In[36]:= ReplacePart[alist, 3 -> x]
Out[36]= {a, b, x, d}
```

The value of `alist` has not changed.

```
In[37]:= alist
Out[37]= {a, b, c, d}
```

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.

This generates a table corresponding to a 2×3 nested list.

```
In[1]:= Table[x^i + j, {i, 2}, {j, 3}]
Out[1]= {{1 + x, 2 + x, 3 + x}, {1 + x^2, 2 + x^2, 3 + x^2}}
```

This generates an array corresponding to the same nested list.

```
In[2]:= Array[x^#1 + #2 &, {2, 3}]
Out[2]= {{1 + x, 2 + x, 3 + x}, {1 + x^2, 2 + x^2, 3 + x^2}}
```

Elements not explicitly specified in the sparse array are taken to be `0`.

```
In[3]:= Normal[SparseArray[{{1, 3} -> 3 + x}, {2, 3}]]
Out[3]= {{0, 0, 3 + x}, {0, 0, 0}}
```

Each element in the final list contains one element from each input list.

```
In[4]:= Outer[f, {a, b}, {c, d}]
Out[4]= {{f[a, c], f[a, d]}, {f[b, c], f[b, d]}}
```

Functions like `Array`, `SparseArray` and `Outer` always generate *full arrays*, in which all sublists at a particular level are the same length.

`Dimensions[`*list*`]`	the dimensions of a full array
`ArrayQ[`*list*`]`	test whether all sublists at a given level are the same length
`ArrayDepth[`*list*`]`	the depth to which all sublists are the same length

Functions for full arrays.

Mathematica can handle arbitrary nested lists. There is no need for the lists to form a full array. You can easily generate ragged arrays using `Table`.

This generates a triangular array.

```
In[5]:= Table[x^i + j, {i, 3}, {j, i}]
```

Out[5]= $\{\{1+x\},\{1+x^2, 2+x^2\},\{1+x^3, 2+x^3, 3+x^3\}\}$

`Flatten[`*list*`]`	flatten out all levels of *list*
`Flatten[`*list*`,`*n*`]`	flatten out the top *n* levels
`ArrayFlatten[`*list*`,`*rank*`]`	create a flattened array from an array of arrays

Flattening out sublists and subarrays.

This generates a 2×3 array.

```
In[6]:= Array[a, {2, 3}]
Out[6]= {{a[1, 1], a[1, 2], a[1, 3]}, {a[2, 1], a[2, 2], a[2, 3]}}
```

`Flatten` in effect puts elements in lexicographic order of their indices.

```
In[7]:= Flatten[%]
Out[7]= {a[1, 1], a[1, 2], a[1, 3], a[2, 1], a[2, 2], a[2, 3]}
```

This creates a matrix from a block matrix.

```
In[8]:= ArrayFlatten[{{{{1}}, {{2, 3}}}, {{{4}, {7}}, {{5, 6}, {8, 9}}}}]
Out[8]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
```

`Transpose[`*list*`]`	transpose the top two levels of *list*
`Transpose[`*list*`,{`n_1`,`n_2`,…}]`	put the k^{th} level in *list* at level n_k

Transposing levels in nested lists.

This generates a 2×2×2 array.

```
In[9]:= Array[a, {2, 2, 2}]
Out[9]= {{{a[1, 1, 1], a[1, 1, 2]}, {a[1, 2, 1], a[1, 2, 2]}},
         {{a[2, 1, 1], a[2, 1, 2]}, {a[2, 2, 1], a[2, 2, 2]}}}
```

This permutes levels so that level 3 appears at level 1.

```
In[10]:= Transpose[%, {3, 1, 2}]
Out[10]= {{{a[1, 1, 1], a[2, 1, 1]}, {a[1, 1, 2], a[2, 1, 2]}},
          {{a[1, 2, 1], a[2, 2, 1]}, {a[1, 2, 2], a[2, 2, 2]}}}
```

This restores the original array.

```
In[11]:= Transpose[%, {2, 3, 1}]
Out[11]= {{{a[1, 1, 1], a[1, 1, 2]}, {a[1, 2, 1], a[1, 2, 2]}},
          {{a[2, 1, 1], a[2, 1, 2]}, {a[2, 2, 1], a[2, 2, 2]}}}
```

`Map[`*f*`,`*list*`,{`*n*`}]`	map *f* across elements at level *n*
`Apply[`*f*`,`*list*`,{`*n*`}]`	apply *f* to the elements at level *n*
`MapIndexed[`*f*`,`*list*`,{`*n*`}]`	map *f* onto parts at level *n* and their indices

Applying functions in nested lists.

Here is a nested list.

```
In[12]:= m = {{{a, b}, {c, d}}, {{e, f}, {g, h}, {i}}};
```

This maps a function `f` at level 2.

```
In[13]:= Map[f, m, {2}]
Out[13]= {{f[{a, b}], f[{c, d}]}, {f[{e, f}], f[{g, h}], f[{i}]}}
```

This applies the function at level 2.

```
In[14]:= Apply[f, m, {2}]
Out[14]= {{f[a, b], f[c, d]}, {f[e, f], f[g, h], f[i]}}
```

This applies `f` to both parts and their indices.

```
In[15]:= MapIndexed[f, m, {2}]
Out[15]= {{f[{a, b}, {1, 1}], f[{c, d}, {1, 2}]}, {f[{e, f}, {2, 1}], f[{g, h}, {2, 2}], f[{i}, {2, 3}]}}
```

`Partition`[*list*, {n_1, n_2, …}]	partition into $n_1 \times n_1 \times \ldots$ blocks
`PadLeft`[*list*, {n_1, n_2, …}]	pad on the left to make an $n_1 \times n_1 \times \ldots$ array
`PadRight`[*list*, {n_1, n_2, …}]	pad on the right to make an $n_1 \times n_1 \times \ldots$ array
`RotateLeft`[*list*, {n_1, n_2, …}]	rotate n_k places to the left at level k
`RotateRight`[*list*, {n_1, n_2, …}]	rotate n_k places to the right at level k

Operations on nested lists.

Here is a nested list.

```
In[16]:= m = {{{a, b, c}, {d, e}}, {{f, g}, {h}, {i}}};
```

This rotates different amounts at each level.

```
In[17]:= RotateLeft[m, {0, 1, -1}]
Out[17]= {{{e, d}, {c, a, b}}, {{h}, {i}, {g, f}}}
```

This pads with zeros to make a 2×3×3 array.

```
In[18]:= PadRight[%, {2, 3, 3}]
Out[18]= {{{e, d, 0}, {c, a, b}, {0, 0, 0}}, {{h, 0, 0}, {i, 0, 0}, {g, f, 0}}}
```

Partitioning and Padding Lists

`Partition`[*list*, *n*]	partition *list* into sublists of length *n*
`Partition`[*list*, *n*, *d*]	partition into sublists with offset *d*
`Split`[*list*]	split *list* into runs of identical elements
`Split`[*list*, *test*]	split into runs with adjacent elements satisfying *test*

Partitioning elements in a list.

This partitions in blocks of 3.

```
In[1]:= Partition[{a, b, c, d, e, f}, 3]
Out[1]= {{a, b, c}, {d, e, f}}
```

This partitions in blocks of 3 with offset 1.

```
In[2]:= Partition[{a, b, c, d, e, f}, 3, 1]
Out[2]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e, f}}
```

The offset can be larger than the block size.

```
In[3]:= Partition[{a, b, c, d, e, f}, 2, 3]
Out[3]= {{a, b}, {d, e}}
```

This splits into runs of identical elements.

```
In[4]:= Split[{1, 4, 1, 1, 1, 2, 2, 3, 3}]
Out[4]= {{1}, {4}, {1, 1, 1}, {2, 2}, {3, 3}}
```

This splits into runs where adjacent elements are unequal.

```
In[5]:= Split[{1, 4, 1, 1, 1, 2, 2, 3, 3}, Unequal]
Out[5]= {{1, 4, 1}, {1}, {1, 2}, {2, 3}, {3}}
```

`Partition` in effect goes through a list, grouping successive elements into sublists. By default it does not include any sublists that would "overhang" the original list.

This stops before any overhang occurs.

```
In[6]:= Partition[{a, b, c, d, e}, 2]
Out[6]= {{a, b}, {c, d}}
```

The same is true here.

```
In[7]:= Partition[{a, b, c, d, e}, 3, 1]
Out[7]= {{a, b, c}, {b, c, d}, {c, d, e}}
```

You can tell `Partition` to include sublists that overhang the ends of the original list. By default, it fills in additional elements by treating the original list as cyclic. It can also treat it as being padded with elements that you specify.

This includes additional sublists, treating the original list as cyclic.

```
In[8]:= Partition[{a, b, c, d, e}, 3, 1, {1, 1}]
Out[8]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e, a}, {e, a, b}}
```

Now the original list is treated as being padded with the element `x`.

```
In[9]:= Partition[{a, b, c, d, e}, 3, 1, {1, 1}, x]
Out[9]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e, x}, {e, x, x}}
```

This pads cyclically with elements `x` and `y`.

```
In[10]:= Partition[{a, b, c, d, e}, 3, 1, {1, 1}, {x, y}]
Out[10]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e, y}, {e, y, x}}
```

This introduces no padding, yielding sublists of differing lengths.

```
In[11]:= Partition[{a, b, c, d, e}, 3, 1, {1, 1}, {}]
Out[11]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e}, {e}}
```

You can think of `Partition` as extracting sublists by sliding a template along and picking out elements from the original list. You can tell `Partition` where to start and stop this process.

This gives all sublists that overlap the original list.

```
In[12]:= Partition[{a, b, c, d}, 3, 1, {-1, 1}, x]
Out[12]= {{x, x, a}, {x, a, b}, {a, b, c}, {b, c, d}, {c, d, x}, {d, x, x}}
```

This allows overlaps only at the beginning.

```
In[13]:= Partition[{a, b, c, d}, 3, 1, {-1, -1}, x]
Out[13]= {{x, x, a}, {x, a, b}, {a, b, c}, {b, c, d}}
```

`Partition[`*list*`,`*n*`,`*d*`]` or `Partition[`*list*`,`*n*`,`*d*`,{1,-1}]`	keep only sublists with no overhangs
`Partition[`*list*`,`*n*`,`*d*`,{1,1}]`	allow an overhang at the end
`Partition[`*list*`,`*n*`,`*d*`,{-1,-1}]`	allow an overhang at the beginning
`Partition[`*list*`,`*n*`,`*d*`,{-1,1}]`	allow overhangs at both the beginning and end
`Partition[`*list*`,`*n*`,`*d*`,{`k_L`,`k_R`}]`	specify alignments of first and last sublists
`Partition[`*list*`,`*n*`,`*d*`,`*spec*`]`	pad by cyclically repeating elements in *list*
`Partition[`*list*`,`*n*`,`*d*`,`*spec*`,`*x*`]`	pad by repeating the element *x*
`Partition[`*list*`,`*n*`,`*d*`,`*spec*`,{`x_1`,`x_2`,`…`}]`	pad by cyclically repeating the x_i
`Partition[`*list*`,`*n*`,`*d*`,`*spec*`,{}]`	use no padding

Specifying alignment and padding.

An alignment specification $\{k_L, k_R\}$ tells `Partition` to give the sequence of sublists in which the first element of the original list appears at position k_L in the first sublist, and the last element of the original list appears at position k_R in the last sublist.

This makes a appear at position 1 in the first sublist.

```
In[14]:= Partition[{a, b, c, d}, 3, 1, {1, 1}, x]
Out[14]= {{a, b, c}, {b, c, d}, {c, d, x}, {d, x, x}}
```

This makes a appear at position 2 in the first sublist.

```
In[15]:= Partition[{a, b, c, d}, 3, 1, {2, 1}, x]
Out[15]= {{x, a, b}, {a, b, c}, {b, c, d}, {c, d, x}, {d, x, x}}
```

Here a is in effect made to appear first at position 4.

```
In[16]:= Partition[{a, b, c, d}, 3, 1, {4, 1}, x]
Out[16]= {{x, x, x}, {x, x, a}, {x, a, b}, {a, b, c}, {b, c, d}, {c, d, x}, {d, x, x}}
```

This fills in padding cyclically from the list given.

```
In[17]:= Partition[{a, b, c, d}, 3, 1, {4, 1}, {x, y}]
Out[17]= {{y, x, y}, {x, y, a}, {y, a, b}, {a, b, c}, {b, c, d}, {c, d, x}, {d, x, y}}
```

Functions like `ListConvolve` use the same alignment and padding specifications as `Partition`.

In some cases it may be convenient to insert explicit padding into a list. You can do this using `PadLeft` and `PadRight`.

`PadLeft[`*list*`,`*n*`]`	pad to length *n* by inserting zeros on the left
`PadLeft[`*list*`,`*n*`,`*x*`]`	pad by repeating the element *x*
`PadLeft[`*list*`,`*n*`,{`x_1`,`x_2`,…}]`	pad by cyclically repeating the x_i
`PadLeft[`*list*`,`*n*`,`*list*`]`	pad by cyclically repeating *list*
`PadLeft[`*list*`,`*n*`,`*padding*`,`*m*`]`	leave a margin of *m* elements on the right
`PadRight[`*list*`,`*n*`]`	pad by inserting zeros on the right

Padding a list.

This pads the list to make it length 6.

```
In[18]:= PadLeft[{a, b, c}, 6]
Out[18]= {0, 0, 0, a, b, c}
```

This cyclically inserts {x, y} as the padding.

```
In[19]:= PadLeft[{a, b, c}, 6, {x, y}]
Out[19]= {x, y, x, a, b, c}
```

This also leaves a margin of 3 on the right.

```
In[20]:= PadLeft[{a, b, c}, 10, {x, y}, 3]
Out[20]= {y, x, y, x, a, b, c, x, y, x}
```

`PadLeft`, `PadRight` and `Partition` can all be used on nested lists.

This creates a 3x3 array.

```
In[21]:= PadLeft[{{a, b}, {e}, {f}}, {3, 3}, x]
Out[21]= {{x, a, b}, {x, x, e}, {x, x, f}}
```

This partitions the array into 2x2 blocks with offset 1.

```
In[22]:= Partition[%, {2, 2}, {1, 1}]
Out[22]= {{{{x, a}, {x, x}}, {{a, b}, {x, e}}}, {{{x, x}, {x, x}}, {{x, e}, {x, f}}}}
```

If you give a nested list as a padding specification, its elements are picked up cyclically at each level.

This cyclically fills in copies of the padding list.

```
In[23]:= PadLeft[{{a, b}, {e}, {f}}, {4, 4}, {{x, y}, {z, w}}]
Out[23]= {{x, y, x, y}, {z, w, a, b}, {x, y, x, e}, {z, w, z, f}}
```

Here is a list containing only padding.

```
In[24]:= PadLeft[{{}}, {4, 4}, {{x, y}, {z, w}}]
Out[24]= {{x, y, x, y}, {z, w, z, w}, {x, y, x, y}, {z, w, z, w}}
```

Sparse Arrays: Manipulating Lists

Lists are normally specified in *Mathematica* just by giving explicit lists of their elements. But particularly in working with large arrays, it is often useful instead to be able to say what the values of elements are only at certain positions, with all other elements taken to have a default value, usually zero. You can do this in *Mathematica* using `SparseArray` objects.

`{`e_1`,`e_2`,…}`, `{{`e_{11}`,`e_{12}`,…},…}`, …	ordinary lists
`SparseArray[{`pos_1`->`val_1`,`pos_2`->`val_2`,…}]`	sparse arrays

Ordinary lists and sparse arrays.

This specifies a sparse array.

```
In[1]:= SparseArray[{2 -> a, 5 -> b}]
Out[1]= SparseArray[<2>, {5}]
```

Here it is as an ordinary list.

```
In[2]:= Normal[%]
Out[2]= {0, a, 0, 0, b}
```

This specifies a two-dimensional sparse array.

```
In[3]:= SparseArray[{{1, 2} -> a, {3, 2} -> b, {3, 3} -> c}]
Out[3]= SparseArray[<3>, {3, 3}]
```

Here it is an ordinary list of lists.

```
In[4]:= Normal[%]
Out[4]= {{0, a, 0}, {0, 0, 0}, {0, b, c}}
```

`SparseArray[`*list*`]`	sparse array version of *list*
`SparseArray[{`pos_1`->`val_1`,`pos_2`->`val_2`,…}]`	sparse array with values val_i at positions pos_i
`SparseArray[{`pos_1`,`pos_2`,…}->{`val_1`,`val_2`,…}]`	the same sparse array
`SparseArray[Band[{`*i*`,`*j*`}]->`*val*`]`	banded sparse array with values *val*
`SparseArray[`*data*`,{`d_1`,`d_2`,…}]`	$d_1 \times d_2 \times \ldots$ sparse array
`SparseArray[`*data*`,`*dims*`,`*val*`]`	sparse array with default value *val*
`Normal[`*array*`]`	ordinary list version of *array*
`ArrayRules[`*array*`]`	position-value rules for *array*

Creating and converting sparse arrays.

This generates a sparse array version of a list.

```
In[5]:=  SparseArray[{a, b, c, d}]
Out[5]=  SparseArray[<4>, {4}]
```

This converts back to an ordinary list.

```
In[6]:=  Normal[%]
Out[6]=  {a, b, c, d}
```

This makes a length 7 sparse array with default value `x`.

```
In[7]:=  SparseArray[{3 -> a, 5 -> b}, 7, x]
Out[7]=  SparseArray[<2>, {7}, x]
```

Here is the corresponding ordinary list.

```
In[8]:=  Normal[%]
Out[8]=  {x, x, a, x, b, x, x}
```

This shows the rules used in the sparse array.

```
In[9]:=  ArrayRules[%%]
Out[9]=  {{3} → a, {5} → b, {_} → x}
```

This creates a banded matrix.

```
In[10]:=  SparseArray[{Band[{1, 1}] → x, Band[{2, 1}] → y}, {5, 5}] // MatrixForm
```

Out[10]//MatrixForm=
$$\begin{pmatrix} x & 0 & 0 & 0 & 0 \\ y & x & 0 & 0 & 0 \\ 0 & y & x & 0 & 0 \\ 0 & 0 & y & x & 0 \\ 0 & 0 & 0 & y & x \end{pmatrix}$$

An important feature of `SparseArray` is that the positions you specify can be patterns.

This specifies a 4×4 sparse array with 1 at every position matching `{i_, i_}`.

```
In[11]:=  SparseArray[{i_, i_} -> 1, {4, 4}]
Out[11]=  SparseArray[<4>, {4, 4}]
```

The result is a 4×4 identity matrix.

```
In[12]:=  Normal[%]
Out[12]=  {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}
```

Here is an identity matrix with an extra element.

```
In[13]:= Normal[SparseArray[{{1, 3} -> a, {i_, i_} -> 1}, {4, 4}]]
Out[13]= {{1, 0, a, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}
```

This makes the whole third column be a.

```
In[14]:= Normal[SparseArray[{{_, 3} -> a, {i_, i_} -> 1}, {4, 4}]]
Out[14]= {{1, 0, a, 0}, {0, 1, a, 0}, {0, 0, a, 0}, {0, 0, a, 1}}
```

You can think of `SparseArray[`*rules*`]` as taking all possible position specifications, then applying *rules* to determine values in each case. As usual, rules given earlier in the list will be tried first.

This generates a random diagonal matrix.

```
In[15]:= Normal[SparseArray[{{i_, i_} :> RandomReal[]}, {3, 3}]]
Out[15]= {{0.0560708, 0, 0}, {0, 0.6303, 0}, {0, 0, 0.359894}}
```

You can have rules where values depend on indices.

```
In[16]:= Normal[SparseArray[i_ -> i^2, 10]]
Out[16]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

This fills in even-numbered positions with p.

```
In[17]:= Normal[SparseArray[{_?EvenQ -> p, i_ -> i^2}, 10]]
Out[17]= {1, p, 9, p, 25, p, 49, p, 81, p}
```

You can use patterns involving alternatives.

```
In[18]:= Normal[SparseArray[{1 | 3, 2 | 4} -> a, {4, 4}]]
Out[18]= {{0, a, 0, a}, {0, 0, 0, 0}, {0, a, 0, a}, {0, 0, 0, 0}}
```

You can also give conditions on patterns.

```
In[19]:= Normal[SparseArray[i_ /; 3 < i < 7 -> p, 10]]
Out[19]= {0, 0, 0, p, p, p, 0, 0, 0, 0}
```

This makes a band-diagonal matrix.

```
In[20]:= Normal[SparseArray[{{i_, j_} /; Abs[i - j] < 2 -> i + j}, {5, 5}]]
Out[20]= {{2, 3, 0, 0, 0}, {3, 4, 5, 0, 0}, {0, 5, 6, 7, 0}, {0, 0, 7, 8, 9}, {0, 0, 0, 9, 10}}
```

Here is another way.

```
In[21]:= Normal[SparseArray[{Band[{1, 1}] → {2, 4, 6, 8, 10},
            Band[{1, 2}] → {3, 5, 7, 9}, Band[{2, 1}] → {3, 5, 7, 9}}, {5, 5}]]
Out[21]= {{2, 3, 0, 0, 0}, {3, 4, 5, 0, 0}, {0, 5, 6, 7, 0}, {0, 0, 7, 8, 9}, {0, 0, 0, 9, 10}}
```

For many purposes, *Mathematica* treats `SparseArray` objects just like the ordinary lists to which they correspond. Thus, for example, if you ask for parts of a sparse array object, *Mathematica* will operate as if you had asked for parts in the corresponding ordinary list.

This generates a sparse array object.

```
In[22]:= s = SparseArray[{2 -> a, 4 -> b, 5 -> c}, 10]
Out[22]= SparseArray[<3>, {10}]
```

Here is the corresponding ordinary list.

```
In[23]:= Normal[s]
Out[23]= {0, a, 0, b, c, 0, 0, 0, 0, 0}
```

Parts of the sparse array are just like parts of the corresponding ordinary list.

```
In[24]:= s[[2]]
Out[24]= a
```

This part has the default value 0.

```
In[25]:= s[[3]]
Out[25]= 0
```

Many operations treat `SparseArray` objects just like ordinary lists. When possible, they give sparse arrays as results.

This gives a sparse array.

```
In[26]:= 3 s + x
Out[26]= SparseArray[<3>, {10}, x]
```

Here is the corresponding ordinary list.

```
In[27]:= Normal[%]
Out[27]= {x, 3 a + x, x, 3 b + x, 3 c + x, x, x, x, x, x}
```

Dot works directly with sparse array objects.

```
In[28]:= s.s
Out[28]= a^2 + b^2 + c^2
```

You can mix sparse arrays and ordinary lists.

```
In[29]:= s.Range[10]
Out[29]= 2 a + 4 b + 5 c
```

Mathematica represents sparse arrays as expressions with head `SparseArray`. Whenever a sparse array is evaluated, it is automatically converted to an optimized standard form with structure `SparseArray[Automatic,` *dims*`,` *val*`,` …`]`.

This structure is, however, rarely evident, since even operations like `Length` are set up to give results for the corresponding ordinary list, not for the raw `SparseArray` expression structure.

This generates a sparse array.

```
In[30]:= t = SparseArray[{1 -> a, 5 -> b}, 10]
Out[30]= SparseArray[<2>, {10}]
```

Here is the underlying optimized expression structure.

```
In[31]:= InputForm[%]
Out[31]//InputForm= SparseArray[Automatic, {10}, 0,   {1, {{0, 2}, {{1}, {5}}}, {a, b}}]
```

`Length` gives the length of the corresponding ordinary list.

```
In[32]:= Length[t]
Out[32]= 10
```

`Map` also operates on individual values.

```
In[33]:= Normal[Map[f, t]]
Out[33]= {f[a], f[0], f[0], f[0], f[b], f[0], f[0], f[0], f[0], f[0]}
```

Expressions

Everything Is an Expression

Mathematica handles many different kinds of things: mathematical formulas, lists and graphics, to name a few. Although they often look very different, *Mathematica* represents all of these things in one uniform way. They are all *expressions*.

A prototypical example of a *Mathematica* expression is `f[x, y]`. You might use `f[x, y]` to represent a mathematical function $f(x, y)$. The function is named `f`, and it has two arguments, `x` and `y`.

You do not always have to write expressions in the form `f[x, y, …]`. For example, `x + y` is also an expression. When you type in `x + y`, *Mathematica* converts it to the standard form `Plus[x, y]`. Then, when it prints it out again, it gives it as `x + y`.

The same is true of other "operators", such as `^` (`Power`) and `/` (`Divide`).

In fact, everything you type into *Mathematica* is treated as an expression.

`x+y+z`	`Plus[x,y,z]`
`x y z`	`Times[x,y,z]`
`x^n`	`Power[x,n]`
`{a,b,c}`	`List[a,b,c]`
`a->b`	`Rule[a,b]`
`a=b`	`Set[a,b]`

Some examples of *Mathematica* expressions.

You can see the full form of any expression by using `FullForm[expr]`.

Here is an expression.

```
In[1]:=  x + y + z
Out[1]= x + y + z
```

This is the full form of the expression.

```
In[2]:= FullForm[%]
Out[2]//FullForm= Plus[x, y, z]
```

Here is another expression.

```
In[3]:= 1 + x^2 + (y + z)^2
```

Out[3]= $1 + x^2 + (y + z)^2$

Its full form has several nested pieces.

```
In[4]:= FullForm[%]
Out[4]//FullForm= Plus[1, Power[x, 2], Power[Plus[y, z], 2]]
```

The object f in an expression $f[x, y, \ldots]$ is known as the *head* of the expression. You can extract it using Head[*expr*]. Particularly when you write programs in *Mathematica*, you will often want to test the head of an expression to find out what kind of thing the expression is.

Head gives the "function name" f.

```
In[5]:= Head[f[x, y]]
Out[5]= f
```

Here Head gives the name of the "operator".

```
In[6]:= Head[a + b + c]
Out[6]= Plus
```

Everything has a head.

```
In[7]:= Head[{a, b, c}]
Out[7]= List
```

Numbers also have heads.

```
In[8]:= Head[23 432]
Out[8]= Integer
```

You can distinguish different kinds of numbers by their heads.

```
In[9]:= Head[345.6]
Out[9]= Real
```

`Head[`*expr*`]`	give the head of an expression: the f in $f[x, y]$
`FullForm[`*expr*`]`	display an expression in the full form used by *Mathematica*

Functions for manipulating expressions.

The Meaning of Expressions

The notion of expressions is a crucial unifying principle in *Mathematica*. It is the fact that every object in *Mathematica* has the same underlying structure that makes it possible for *Mathematica* to cover so many areas with a comparatively small number of basic operations.

Although all expressions have the same basic structure, there are many different ways that expressions can be used. Here are a few of the interpretations you can give to the parts of an expression.

meaning of f	*meaning of $x, y, \ldots$*	*examples*
Function	arguments or parameters	`Sin[x]`, `f[x,y]`
Command	arguments or parameters	`Expand[(x+1)^2]`
Operator	operands	`x+y`, `a=b`
Head	elements	`{a,b,c}`
Object type	contents	`RGBColor[r,g,b]`

Some interpretations of parts of expressions.

Expressions in *Mathematica* are often used to specify operations. So, for example, typing in `2 + 3` causes `2` and `3` to be added together, while `Factor[x^6 - 1]` performs factorization.

Perhaps an even more important use of expressions in *Mathematica*, however, is to maintain a structure, which can then be acted on by other functions. An expression like `{a, b, c}` does not specify an operation. It merely maintains a list structure, which contains a collection of three elements. Other functions, such as `Reverse` or `Dot`, can act on this structure.

The full form of the expression `{a, b, c}` is `List[a, b, c]`. The head `List` performs no operations. Instead, its purpose is to serve as a “tag” to specify the “type” of the structure.

You can use expressions in *Mathematica* to create your own structures. For example, you might want to represent points in three-dimensional space, specified by three coordinates. You could give each point as `point[x, y, z]`. The "function" `point` again performs no operation. It serves merely to collect the three coordinates together, and to label the resulting object as a `point`.

You can think of expressions like `point[x, y, z]` as being "packets of data", tagged with a particular head. Even though all expressions have the same basic structure, you can distinguish different "types" of expressions by giving them different heads. You can then set up transformation rules and programs which treat different types of expressions in different ways.

Special Ways to Input Expressions

Mathematica allows you to use special notation for many common operators. For example, although internally *Mathematica* represents a sum of two terms as `Plus[x, y]`, you can enter this expression in the much more convenient form $x + y$.

The *Mathematica* language has a definite grammar which specifies how your input should be converted to internal form. One aspect of the grammar is that it specifies how pieces of your input should be grouped. For example, if you enter an expression such as `a + b ^ c`, the *Mathemat ica* grammar specifies that this should be considered, following standard mathematical notation, as `a + (b ^ c)` rather than `(a + b) ^ c`. *Mathematica* chooses this grouping because it treats the operator `^` as having a higher *precedence* than `+`. In general, the arguments of operators with higher precedence are grouped before those of operators with lower precedence.

You should realize that absolutely every special input form in *Mathematica* is assigned a definite precedence. This includes not only the traditional mathematical operators, but also forms such as `->`, `:=` or the semicolons used to separate expressions in a *Mathematica* program.

The table in "Operator Input Forms" gives all the operators of *Mathematica* in order of decreasing precedence. The precedence is arranged, where possible, to follow standard mathematical usage, and to minimize the number of parentheses that are usually needed.

You will find, for example, that relational operators such as `<` have lower precedence than arithmetic operators such as `+`. This means that you can write expressions such as `x + y > 7` without using parentheses.

There are nevertheless many cases where you do have to use parentheses. For example, since `;` has a lower precedence than `=`, you need to use parentheses to write `x = (a; b)`. *Mathematica* interprets the expression `x = a; b` as `(x = a); b`. In general, it can never hurt to include extra parentheses, but it can cause a great deal of trouble if you leave parentheses out, and *Mathematica* interprets your input in a way you do not expect.

f [*x*,*y*]	standard form for *f* [*x*, *y*]
f@*x*	prefix form for *f* [*x*]
x//*f*	postfix form for *f* [*x*]
x~*f*~*y*	infix form for *f* [*x*, *y*]

Four ways to write expressions in *Mathematica*.

There are several common types of operators in *Mathematica*. The + in $x + y$ is an "infix" operator. The − in $-p$ is a "prefix" operator. Even when you enter an expression such as *f*[*x*, *y*, …] *Mathematica* allows you to do it in ways that mimic infix, prefix and postfix forms.

This "postfix form" is exactly equivalent to `f[x + y]`.

```
In[1]:=  x + y // f
Out[1]=  f[x + y]
```

You will often want to add functions like `N` as "afterthoughts", and give them in postfix form.

```
In[2]:=  3^(1 / 4) + 1 // N
Out[2]=  2.31607
```

It is sometimes easier to understand what a function is doing when you write it in infix form.

```
In[3]:=  {a, b, c} ~ Join ~ {d, e}
Out[3]=  {a, b, c, d, e}
```

You should notice that `//` has very low precedence. If you put `//` *f* at the end of any expression containing arithmetic or logical operators, the *f* is applied to the *whole expression*. So, for example, `x + y // f` means `f[x + y]`, not `x + f[y]`.

The prefix form `@` has a much higher precedence. `f@x + y` is equivalent to `f[x] + y`, not `f[x + y]`. You can write `f[x + y]` in prefix form as `f@(x + y)`.

Parts of Expressions

Since lists are just a particular kind of expression, it will come as no surprise that you can refer to parts of any expression much as you refer to parts of a list.

This gets the second element in the list {a, b, c}.

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

You can use the same method to get the second element in the sum x + y + z.

```
In[2]:= (x + y + z)[[2]]
Out[2]= y
```

This gives the last element in the sum.

```
In[3]:= (x + y + z)[[-1]]
Out[3]= z
```

Part 0 is the head.

```
In[4]:= (x + y + z)[[0]]
Out[4]= Plus
```

You can refer to parts of an expression such as *f*[*g*[*a*], *g*[*b*]] just as you refer to parts of nested lists.

This is part 1.

```
In[5]:= f[g[a], g[b]][[1]]
Out[5]= g[a]
```

This is part {1, 1}.

```
In[6]:= f[g[a], g[b]][[1, 1]]
Out[6]= a
```

This extracts part {2, 1} of the expression 1 + x^2.

```
In[7]:= (1 + x^2)[[2, 1]]
Out[7]= x
```

To see what part is {2, 1}, you can look at the full form of the expression.

```
In[8]:= FullForm[1 + x^2]
Out[8]//FullForm= Plus[1, Power[x, 2]]
```

You should realize that the assignment of indices to parts of expressions is done on the basis of the internal *Mathematica* forms of the expression, as shown by `FullForm`. These forms do not always correspond directly with what you see printed out. This is particularly true for algebraic expressions, where *Mathematica* uses a standard internal form, but prints the expressions in special ways.

Here is the internal form of `x / y`.

```
In[9]:= FullForm[x / y]
Out[9]//FullForm= Times[x, Power[y, -1]]
```

It is the internal form that is used in specifying parts.

```
In[10]:= (x / y)[[2]]
```

Out[10]= $\frac{1}{y}$

You can manipulate parts of expressions just as you manipulate parts of lists.

This replaces the third part of a + b + c + d by `x^2`. Note that the sum is automatically rearranged when the replacement is done.

```
In[11]:= ReplacePart[a + b + c + d, 3 -> x^2]
```

Out[11]= $a + b + d + x^2$

Here is an expression.

```
In[12]:= t = 1 + (3 + x)^2 / y
```

Out[12]= $1 + \frac{(3 + x)^2}{y}$

This is the full form of `t`.

```
In[13]:= FullForm[t]
Out[13]//FullForm= Plus[1, Times[Power[Plus[3, x], 2], Power[y, -1]]]
```

This resets a part of the expression `t`.

```
In[14]:= t[[2, 1, 1]] = x
Out[14]= x
```

Now the form of `t` has been changed.

```
In[15]:= t
```

Out[15]= $1+\frac{x^2}{y}$

`Part`[*expr*,*n*] or *expr*[[*n*]]	the n^{th} part of *expr*
`Part`[*expr*,{n_1,n_2,…}] or *expr*[[{n_1,n_2,…}]]	a combination of parts of an expression
`Part`[*expr*,n_1;;n_2]	parts n_1 through n_2 of an expression
`ReplacePart`[*expr*,*n*->*elem*]	replace the n^{th} part of *expr* by *elem*

Functions for manipulating parts of expressions.

"Manipulating Elements of Lists" discusses how you can use lists of indices to pick out several elements of a list at a time. You can use the same procedure to pick out several parts in an expression at a time.

This picks out elements 2 and 4 in the list, and gives a list of these elements.

```
In[16]:= {a, b, c, d, e}[[{2, 4}]]
Out[16]= {b, d}
```

This picks out parts 2 and 4 of the sum, and gives a *sum* of these elements.

```
In[17]:= (a + b + c + d + e)[[{2, 4}]]
Out[17]= b + d
```

Any part in an expression can be viewed as being an argument of some function. When you pick out several parts by giving a list of indices, the parts are combined using the same function as in the expression.

This picks out parts 2 through 4 of the list.

```
In[18]:= {a, b, c, d, e}[[2 ;; 4]]
Out[18]= {b, c, d}
```

Manipulating Expressions like Lists

You can use most of the list operations discussed in "Lists" on any kind of *Mathematica* expression. By using these operations, you can manipulate the structure of expressions in many ways.

Here is an expression that corresponds to a sum of terms.

```
In[1]:= t = 1 + x + x^2 + y^2
Out[1]= 1 + x + x^2 + y^2
```

`Take[t, 2]` takes the first two elements from `t`, just as if `t` were a list.

```
In[2]:= Take[t, 2]
Out[2]= 1 + x
```

`Length` gives the number of elements in `t`.

```
In[3]:= Length[t]
Out[3]= 4
```

You can use `FreeQ[`*expr*`,` *form*`]` to test whether *form* appears nowhere in *expr*.

```
In[4]:= FreeQ[t, x]
Out[4]= False
```

This gives a list of the positions at which `x` appears in `t`.

```
In[5]:= Position[t, x]
Out[5]= {{2}, {3, 1}}
```

You should remember that all functions which manipulate the structure of expressions act on the internal forms of these expressions. You can see these forms using `FullForm[`*expr*`]`. They may not be what you would expect from the printed versions of the expressions.

Here is a function with four arguments.

```
In[6]:= f[a, b, c, d]
Out[6]= f[a, b, c, d]
```

You can add an argument using `Append`.

```
In[7]:= Append[%, e]
Out[7]= f[a, b, c, d, e]
```

This reverses the arguments.

```
In[8]:= Reverse[%]
Out[8]= f[e, d, c, b, a]
```

There are a few extra functions that can be used with expressions, as discussed in "Structural Operations".

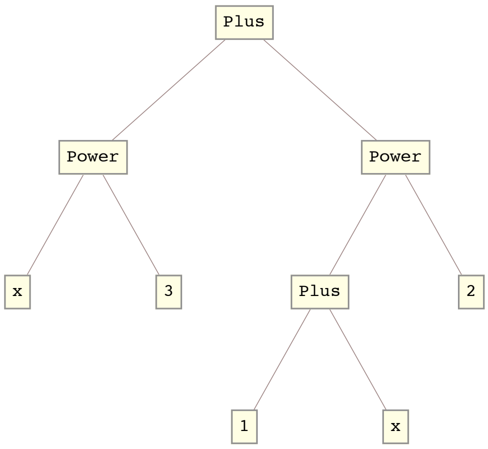
Expressions as Trees

Here is an expression in full form.

```
In[1]:= FullForm[x^3 + (1 + x)^2]
Out[1]//FullForm= Plus[Power[x, 3], Power[Plus[1, x], 2]]
```

`TreeForm` prints out expressions to show their “tree” structure.

```
In[2]:= TreeForm[x^3 + (1 + x)^2]
Out[2]//TreeForm=
```

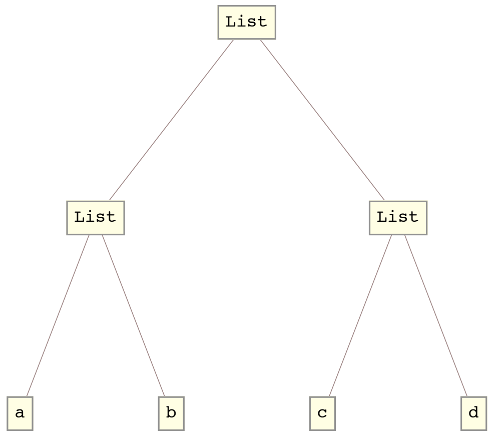


You can think of any *Mathematica* expression as a tree. In the expression above, the top node in the tree consists of a `Plus`. From this node come two “branches”, `x^3` and `(1 + x)^2`. From the `x^3` node, there are then two branches, `x` and `3`, which can be viewed as “leaves” of the tree.

This matrix is a simple tree with just two levels.

```
In[3]:= TreeForm[{{a, b}, {c, d}}]
```

Out[3]//TreeForm=



Here is a more complicated expression.

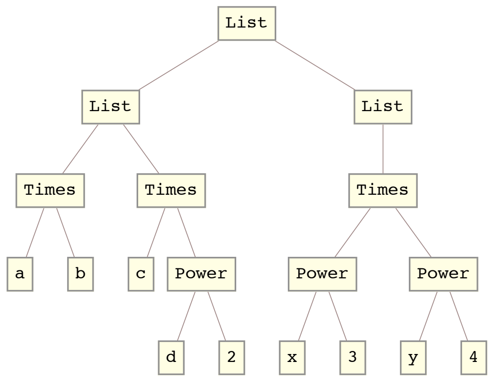
```
In[4]:= {{a b, c d^2}, {x^3 y^4}}
```

Out[4]= $\{\{a\,b, c\,d^2\}, \{x^3\,y^4\}\}$

The tree for this expression has several levels. The representation of the tree here was too long to fit on a single line, so it had to be broken onto two lines.

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

Out[5]//TreeForm=



The indices that label each part of an expression have a simple interpretation in terms of trees. Descending from the top node of the tree, each index specifies which branch to take in order to reach the part you want.

Levels in Expressions

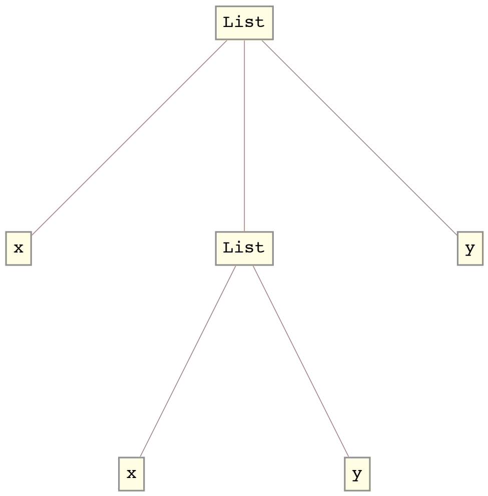
The `Part` function allows you to access specific parts of *Mathematica* expressions. But particularly when your expressions have fairly uniform structure, it is often convenient to be able to refer to a whole collection of parts at the same time.

Levels provide a general way of specifying collections of parts in *Mathematica* expressions. Many *Mathematica* functions allow you to specify the levels in an expression on which they should act.

Here is a simple expression, displayed in tree form.

```
In[1]:= (t = {x, {x, y}, y}) // TreeForm
```

Out[1]//TreeForm=



This searches for `x` in the expression `t` down to level 1. It finds only one occurrence.

```
In[2]:= Position[t, x, 1]
Out[2]= {{1}}
```

This searches down to level 2. Now it finds both occurrences of `x`.

```
In[3]:= Position[t, x, 2]
Out[3]= {{1}, {2, 1}}
```

This searches only at level 2. It finds just one occurrence of `x`.

```
In[4]:= Position[t, x, {2}]
Out[4]= {{2, 1}}
```

`Position[`*expr*`,`*form*`,`*n*`]`	give the positions at which *form* occurs in *expr* down to level *n*
`Position[`*expr*`,`*form*`,{`*n*`}]`	give the positions exactly at level *n*

Controlling `Position` using levels.

You can think of levels in expressions in terms of trees. The level of a particular part in an expression is simply the distance down the tree at which that part appears, with the top of the tree considered as level 0.

It is equivalent to say that the parts which appear at level n are those that can be specified by a sequence of exactly n indices.

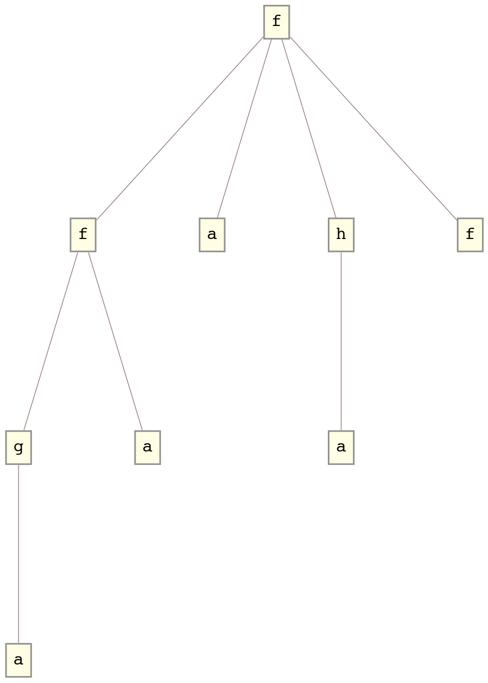
n	levels 1 through n
`Infinity`	all levels (except 0)
$\{n\}$	level n only
$\{n_1, n_2\}$	levels n_1 through n_2
`Heads->True`	include heads
`Heads->False`	exclude heads

Level specifications.

Here is an expression, displayed in tree form.

```
In[5]:= (u = f[f[g[a], a], a, h[a], f]) // TreeForm
```

Out[5]//TreeForm=



This searches for a at levels from 2 downward.

```
In[6]:= Position[u, a, {2, Infinity}]
Out[6]= {{1, 1, 1}, {1, 2}, {3, 1}}
```

This shows where f appears other than in the head of an expression.

```
In[7]:= Position[u, f, Heads -> False]
Out[7]= {{4}}
```

This includes occurrences of f in heads of expressions.

```
In[8]:= Position[u, f, Heads -> True]
Out[8]= {{0}, {1, 0}, {4}}
```

`Level[`*expr*`,`*lev*`]`	a list of the parts of *expr* at the levels specified by *lev*
`Depth[`*expr*`]`	the total number of levels in *expr*

Testing and extracting levels.

This gives a list of all parts of `u` that occur down to level 2.

```
In[9]:= Level[u, 2]
Out[9]= {g[a], a, f[g[a], a], a, a, h[a], f}
```

Here are the parts specifically at level 2.

```
In[10]:= Level[u, {2}]
Out[10]= {g[a], a, a}
```

When you have got the hang of ordinary levels, you can try thinking about *negative levels*. Negative levels label parts of expressions starting at the *bottom* of the tree. Level -1 contains all the leaves of the tree: objects like symbols and numbers.

This shows the parts of `u` at level -1.

```
In[11]:= Level[u, {-1}]
Out[11]= {a, a, a, a, f}
```

You can think of expressions as having a "depth", as shown by `TreeForm`. In general, level $-n$ in an expression is defined to consist of all subexpressions whose depth is n.

The depth of `g[a]` is 2.

```
In[12]:= Depth[g[a]]
Out[12]= 2
```

The parts of `u` at level -2 are those that have depth exactly 2.

```
In[13]:= Level[u, {-2}]
Out[13]= {g[a], h[a]}
```

Patterns

Introduction to Patterns

Patterns are used throughout *Mathematica* to represent classes of expressions. A simple example of a pattern is the expression `f[x_]`. This pattern represents the class of expressions with the form `f[`*anything*`]`.

The main power of patterns comes from the fact that many operations in *Mathematica* can be done not only with single expressions, but also with patterns that represent whole classes of expressions.

You can use patterns in transformation rules to specify how classes of expressions should be transformed.

```
In[1]:=  f[a] + f[b] /. f[x_] -> x^2
```

Out[1]= $a^2 + b^2$

You can use patterns to find the positions of all expressions in a particular class.

```
In[2]:=  Position[{f[a], g[b], f[c]}, f[x_]]
Out[2]=  {{1}, {3}}
```

The basic object that appears in almost all *Mathematica* patterns is _ (traditionally called "blank" by *Mathematica* programmers). The fundamental rule is simply that _ *stands for any expression*. On most keyboards the _ underscore character appears as the shifted version of the - dash character.

Thus, for example, the pattern `f[_]` stands for any expression of the form `f[`*anything*`]`. The pattern `f[x_]` also stands for any expression of the form `f[`*anything*`]`, but gives the name `x` to the expression *anything*, allowing you to refer to it on the right-hand side of a transformation rule.

You can put blanks anywhere in an expression. What you get is a pattern which matches all expressions that can be made by "filling in the blanks" in any way.

`f[n_]`	`f` with any argument, named `n`
`f[n_,m_]`	`f` with two arguments, named `n` and `m`
`x^n_`	`x` to any power, with the power named `n`
`x_^n_`	any expression to any power
`a_+b_`	a sum of two expressions
`{a1_,a2_}`	a list of two expressions
`f[n_,n_]`	`f` with two *identical* arguments

Some examples of patterns.

You can construct patterns for expressions with any structure.

```
In[3]:= f[{a, b}] + f[c] /. f[{x_, y_}] -> p[x + y]
Out[3]= f[c] + p[a + b]
```

One of the most common uses of patterns is for "destructuring" function arguments. If you make a definition for `f[list_]`, then you need to use functions like `Part` explicitly in order to pick out elements of the list. But if you know for example that the list will always have two elements, then it is usually much more convenient instead to give a definition instead for `f[{x_, y_}]`. Then you can refer to the elements of the list directly as `x` and `y`. In addition, *Mathematica* will not use the definition you have given unless the argument of `f` really is of the required form of a list of two expressions.

Here is one way to define a function which takes a list of two elements, and evaluates the first element raised to the power of the second element.

```
In[4]:= g[list_] := Part[list, 1]^Part[list, 2]
```

Here is a much more elegant way to make the definition, using a pattern.

```
In[5]:= h[{x_, y_}] := x^y
```

A crucial point to understand is that *Mathematica* patterns represent classes of expressions with a given *structure*. One pattern will match a particular expression if the structure of the pattern is the same as the structure of the expression, in the sense that by filling in blanks in the pattern you can get the expression. Even though two expressions may be *mathematically equal*, they cannot be represented by the same *Mathematica* pattern unless they have the same structure.

Thus, for example, the pattern `(1 + x_) ^ 2` can stand for expressions like `(1 + a) ^ 2` or `(1 + b ^ 3) ^ 2` that have the same *structure*. However, it cannot stand for the expression `1 + 2 a + a ^ 2`. Although this expression is *mathematically equal* to `(1 + a) ^ 2`, it does not have the same *structure* as the pattern `(1 + x_) ^ 2`.

The fact that patterns in *Mathematica* specify the *structure* of expressions is crucial in making it possible to set up transformation rules which change the *structure* of expressions, while leaving them mathematically equal.

It is worth realizing that in general it would be quite impossible for *Mathematica* to match patterns by mathematical, rather than structural, equivalence. In the case of expressions like `(1 + a) ^ 2` and `1 + 2 a + a ^ 2`, you can determine equivalence just by using functions like `Expand` and `Factor`. But, as discussed in "Reducing Expressions to Their Standard Form" there is no general way to find out whether an arbitrary pair of mathematical expressions are equal.

As another example, the pattern `x ^ _` will match the expression `x ^ 2`. It will not, however, match the expression `1`, even though this could be considered as `x ^ 0`. "Optional and Default Arguments" discusses how to construct a pattern for which this particular case will match. But you should understand that in all cases pattern matching in *Mathematica* is fundamentally structural.

The `x ^ n_` matches only `x ^ 2` and `x ^ 3`. `1` and `x` can mathematically be written as x^n, but do not have the same structure.

```
In[6]:=  {1, x, x^2, x^3} /. x^n_ -> r[n]
Out[6]=  {1, x, r[2], r[3]}
```

Another point to realize is that the structure *Mathematica* uses in pattern matching is the full form of expressions printed by `FullForm`. Thus, for example, an object such as `1 / x`, whose full form is `Power[x, -1]` will be matched by the pattern `x_ ^ n_`, but not by the pattern `x_ / y_`, whose full form is `Times[x_, Power[y_, -1]]`. Again, "Optional and Default Arguments" will discuss how you can construct patterns which can match all these cases.

The expressions in the list contain explicit powers of `b`, so the transformation rule can be applied.

```
In[7]:=  {a / b, 1 / b^2, 2 / b^2} /. b^n_ -> d[n]
Out[7]=  {a d[-1], d[-2], 2 d[-2]}
```

Here is the full form of the list.

```
In[8]:= FullForm[{a / b, 1 / b^2, 2 / b^2}]
Out[8]//FullForm= List[Times[a, Power[b, -1]], Power[b, -2], Times[2, Power[b, -2]]]
```

Although *Mathematica* does not use mathematical equivalences such as $x^1 = x$ when matching patterns, it does use certain structural equivalences. Thus, for example, *Mathematica* takes account of properties such as commutativity and associativity in pattern matching.

To apply this transformation rule, *Mathematica* makes use of the commutativity and associativity of addition.

```
In[9]:= f[a + b] + f[a + c] + f[b + d] /. f[a + x_] + f[c + y_] -> p[x, y]
Out[9]= f[b + d] + p[b, a]
```

The discussion considers only pattern objects such as `x_` which can stand for any single expression. Other Tutorials discuss the constructs that *Mathematica* uses to extend and restrict the classes of expressions represented by patterns.

Finding Expressions That Match a Pattern

`Cases[`*list*`,`*form*`]`	give the elements of *list* that match *form*
`Count[`*list*`,`*form*`]`	give the number of elements in *list* that match *form*
`Position[`*list*`,`*form*`,{1}]`	give the positions of elements in *list* that match *form*
`Select[`*list*`,`*test*`]`	give the elements of *list* on which *test* gives `True`
`Pick[`*list*`,`*sel*`,`*form*`]`	give the elements of *list* for which the corresponding elements of *sel* match *form*

Finding elements that match a pattern.

This gives the elements of the list which match the pattern `x^_`.

```
In[1]:= Cases[{3, 4, x, x^2, x^3}, x^_]
Out[1]= {x², x³}
```

Here is the total number of elements which match the pattern.

```
In[2]:= Count[{3, 4, x, x^2, x^3}, x^_]
Out[2]= 2
```

You can apply functions like `Cases` not only to lists, but to expressions of any kind. In addition, you can specify the level of parts at which you want to look.

`Cases`[*expr*,*lhs*->*rhs*]	find elements of *expr* that match *lhs*, and give a list of the results of applying the transformation rule to them
`Cases`[*expr*,*lhs*->*rhs*,*lev*]	test parts of *expr* at levels specified by *lev*
`Count`[*expr*,*form*,*lev*]	give the total number of parts that match *form* at levels specified by *lev*
`Position`[*expr*,*form*,*lev*]	give the positions of parts that match *form* at levels specified by *lev*

Searching for parts of expressions that match a pattern.

This returns a list of the exponents `n`.

```
In[3]:= Cases[{3, 4, x, x^2, x^3}, x^n_ -> n]
Out[3]= {2, 3}
```

The pattern `_Integer` matches any integer. This gives a list of integers appearing at any level.

```
In[4]:= Cases[{3, 4, x, x^2, x^3}, _Integer, Infinity]
Out[4]= {3, 4, 2, 3}
```

`Cases`[*expr*,*form*,*lev*,*n*]	find only the first *n* parts that match *form*
`Position`[*expr*,*form*,*lev*,*n*]	give the positions of the first *n* parts that match *form*

Limiting the number of parts to search for.

This gives the positions of the first two powers of `x` appearing at any level.

```
In[5]:= Position[{4, 4 + x^a, x^b, 6 + x^5}, x^_, Infinity, 2]
Out[5]= {{2, 2}, {3}}
```

The positions are specified in exactly the form used by functions such as `Extract` and `ReplacePart` discussed in "Lists".

```
In[6]:= ReplacePart[{4, 4 + x^a, x^b, 6 + x^5}, zzz, %]
Out[6]= {4, 4 + zzz, zzz, 6 + x^5}
```

`DeleteCases[`*expr*`,`*form*`]`	delete elements of *expr* that match *form*
`DeleteCases[`*expr*`,`*form*`,`*lev*`]`	delete parts of *expr* that match *form* at levels specified by *lev*

Deleting parts of expressions that match a pattern.

This deletes the elements which match `x^n_`.

```
In[7]:= DeleteCases[{3, 4, x, x^2, x^3}, x^n_]
Out[7]= {3, 4, x}
```

This deletes all integers appearing at any level.

```
In[8]:= DeleteCases[{3, 4, x, 2 + x, 3 + x}, _Integer, Infinity]
Out[8]= {x, x, x}
```

`ReplaceList[`*expr*`,`*lhs*`->`*rhs*`]`	find all ways that *expr* can match *lhs*

Finding arrangements of an expression that match a pattern.

This finds all ways that the sum can be written in two parts.

```
In[9]:= ReplaceList[a + b + c, x_ + y_ -> g[x, y]]
Out[9]= {g[a, b + c], g[b, a + c], g[c, a + b], g[a + b, c], g[a + c, b], g[b + c, a]}
```

This finds all pairs of identical elements. The pattern `___` stands for any sequence of elements.

```
In[10]:= ReplaceList[{a, b, b, b, c, c, a}, {___, x_, x_, ___} -> x]
Out[10]= {b, b, c}
```

Naming Pieces of Patterns

Particularly when you use transformation rules, you often need to name pieces of patterns. An object like *x*_ stands for any expression, but gives the expression the name *x*. You can then, for example, use this name on the right-hand side of a transformation rule.

An important point is that when you use *x*_, *Mathematica* requires that all occurrences of blanks with the same name *x* in a particular expression must stand for the same expression.

Thus `f[x_, x_]` can only stand for expressions in which the two arguments of `f` are exactly the same. `f[_, _]`, on the other hand, can stand for any expression of the form `f[`*x*`, `*y*`]`, where *x* and *y* need not be the same.

The transformation rule applies only to cases where the two arguments of `f` are identical.

```
In[1]:= {f[a, a], f[a, b]} /. f[x_, x_] -> p[x]
Out[1]= {p[a], f[a, b]}
```

Mathematica allows you to give names not just to single blanks, but to any piece of a pattern. The object *x* `:` *pattern* in general represents a pattern which is assigned the name *x*. In transformation rules, you can use this mechanism to name exactly those pieces of a pattern that you need to refer to on the right-hand side of the rule.

`_`	any expression
x`_`	any expression, to be named *x*
x`:`*pattern*	an expression to be named *x*, matching *pattern*

Patterns with names.

This gives a name to the complete form `_^_` so you can refer to it as a whole on the right-hand side of the transformation rule.

```
In[2]:= f[a^b] /. f[x : _^_] -> p[x]
```
Out[2]= $p[a^b]$

Here the exponent is named `n`, while the whole object is `x`.

```
In[3]:= f[a^b] /. f[x : _^n_] -> p[x, n]
```
Out[3]= $p[a^b, b]$

When you give the same name to two pieces of a pattern, you constrain the pattern to match only those expressions in which the corresponding pieces are identical.

Here the pattern matches both cases.

```
In[4]:= {f[h[4], h[4]], f[h[4], h[5]]} /. f[h[_], h[_]] -> q
Out[4]= {q, q}
```

Now both arguments of f are constrained to be the same, and only the first case matches.

```
In[5]:= {f[h[4], h[4]], f[h[4], h[5]]} /. f[x : h[_], x_] -> r[x]
Out[5]= {r[h[4]], f[h[4], h[5]]}
```

Specifying Types of Expression in Patterns

You can tell a lot about what "type" of expression something is by looking at its head. Thus, for example, an integer has head `Integer`, while a list has head `List`.

In a pattern, _*h* and *x*_*h* represent expressions that are constrained to have head *h*. Thus, for example, `_Integer` represents any integer, while `_List` represents any list.

*x*_*h*	an expression with head *h*
*x*_Integer	an integer
*x*_Real	an approximate real number
*x*_Complex	a complex number
*x*_List	a list
*x*_Symbol	a symbol

Patterns for objects with specified heads.

This replaces just those elements that are integers.

```
In[1]:= {a, 4, 5, b} /. x_Integer -> p[x]
Out[1]= {a, p[4], p[5], b}
```

You can think of making an assignment for `f[x_Integer]` as like defining a function `f` that must take an argument of "type" `Integer`.

This defines a value for the function `gamma` when its argument is an integer.

```
In[2]:= gamma[n_Integer] := (n - 1) !
```

The definition applies only when the argument of `gamma` is an integer.

```
In[3]:= gamma[4] + gamma[x]
Out[3]= 6 + gamma[x]
```

The object `4.` has head `Real`, so the definition does not apply.

```
In[4]:= gamma[4.]
Out[4]= gamma[4.]
```

This defines values for expressions with integer exponents.

```
In[5]:= d[x_^n_Integer] := n x^(n - 1)
```

The definition is used only when the exponent is an integer.

```
In[6]:= d[x^4] + d[(a + b)^3] + d[x^(1 / 2)]
```

Out[6]= $3\,(a+b)^2 + 4\,x^3 + d\left[\sqrt{x}\right]$

Putting Constraints on Patterns

Mathematica provides a general mechanism for specifying constraints on patterns. All you need do is to put `/;` *condition* at the end of a pattern to signify that it applies only when the specified condition is `True`. You can read the operator `/;` as "slash-semi", "whenever" or "provided that".

pattern`/;`*condition*	a pattern that matches only when a condition is satisfied
lhs`:>`*rhs*`/;`*condition*	a rule that applies only when a condition is satisfied
lhs`:=`*rhs*`/;`*condition*	a definition that applies only when a condition is satisfied

Putting conditions on patterns and transformation rules.

This gives a definition for `fac` that applies only when its argument `n` is positive.

```
In[1]:= fac[n_ /; n > 0] := n!
```

The definition for `fac` is used only when the argument is positive.

```
In[2]:= fac[6] + fac[-4]
Out[2]= 720 + fac[-4]
```

This gives the negative elements in the list.

```
In[3]:= Cases[{3, -4, 5, -2}, x_ /; x < 0]
Out[3]= {-4, -2}
```

You can use `/;` on whole definitions and transformation rules, as well as on individual patterns. In general, you can put `/;` *condition* at the end of any `:=` definition or `:>` rule to tell *Mathematica* that the definition or rule applies only when the specified condition holds. Note that `/;` conditions should not usually be put at the end of `=` definitions or `->` rules, since they will then be evaluated immediately, as discussed in "Immediate and Delayed Definitions".

Here is another way to give a definition which applies only when its argument `n` is positive.

```
In[4]:= fac2[n_] := n! /; n > 0
```

Once again, the factorial functions evaluate only when their arguments are positive.

```
In[5]:= fac2[6] + fac2[-4]
Out[5]= 720 + fac2[-4]
```

You can use the `/;` operator to implement arbitrary mathematical constraints on the applicability of rules. In typical cases, you give patterns which *structurally* match a wide range of expressions, but then use *mathematical* constraints to reduce the range of expressions to a much smaller set.

This rule applies only to expressions that have the structure `v[x_, 1 - x_]`.

```
In[6]:= v[x_, 1 - x_] := p[x]
```

This expression has the appropriate structure, so the rule applies.

```
In[7]:= v[a^2, 1 - a^2]
```

Out[7]= $p[a^2]$

This expression, while mathematically of the correct form, does not have the appropriate structure, so the rule does not apply.

```
In[8]:= v[4, -3]
Out[8]= v[4, -3]
```

This rule applies to any expression of the form `w[x_, y_]`, with the added restriction that `y == 1 - x`.

```
In[9]:= w[x_, y_] := p[x] /; y == 1 - x
```

The new rule does apply to this expression.

```
In[10]:= w[4, -3]
Out[10]= p[4]
```

In setting up patterns and transformation rules, there is often a choice of where to put `/;` conditions. For example, you can put a `/;` condition on the right-hand side of a rule in the form *lhs* `:>` *rhs* `/;` *condition*, or you can put it on the left-hand side in the form *lhs* `/;` *condition* `->` *rhs*. You may also be able to insert the condition inside the expression *lhs*. The only constraint is that all the names of patterns that you use in a particular condition must appear in the pattern to which the condition is attached. If this is not the case, then some of the names needed to evaluate the condition may not yet have been "bound" in the pattern-matching process. If this happens, then *Mathematica* uses the global values for the corresponding variables, rather than the values determined by pattern matching.

Thus, for example, the condition in `f[x_, y_] /; (x + y < 2)` will use values for `x` and `y` that are found by matching `f[x_, y_]`, but the condition in `f[x_ /; x + y < 2, y_]` will use the global value for `y`, rather than the one found by matching the pattern.

As long as you make sure that the appropriate names are defined, it is usually most efficient to put `/;` conditions on the smallest possible parts of patterns. The reason for this is that *Mathematica* matches pieces of patterns sequentially, and the sooner it finds a `/;` condition which fails, the sooner it can reject a match.

Putting the `/;` condition around the `x_` is slightly more efficient than putting it around the whole pattern.

```
In[11]:=  Cases[{z[1, 1], z[-1, 1], z[-2, 2]}, z[x_ /; x < 0, y_]]
Out[11]=  {z[-1, 1], z[-2, 2]}
```

You need to put parentheses around the `/;` piece in a case like this.

```
In[12]:=  {1 + a, 2 + a, -3 + a} /. (x_ /; x < 0) + a -> p[x]
Out[12]=  {1 + a, 2 + a, p[-3]}
```

It is common to use `/;` to set up patterns and transformation rules that apply only to expressions with certain properties. There is a collection of functions built into *Mathematica* for testing the properties of expressions. It is a convention that functions of this kind have names that end with the letter `Q`, indicating that they "ask a question".

`IntegerQ[`*expr*`]`	integer
`EvenQ[`*expr*`]`	even number
`OddQ[`*expr*`]`	odd number
`PrimeQ[`*expr*`]`	prime number
`NumberQ[`*expr*`]`	explicit number of any kind
`NumericQ[`*expr*`]`	numeric quantity
`PolynomialQ[`*expr*`,{`x_1`,`x_2`,…}]`	
	polynomial in x_1, x_2, …
`VectorQ[`*expr*`]`	a list representing a vector
`MatrixQ[`*expr*`]`	a list of lists representing a matrix
`VectorQ[`*expr*`,NumericQ]` , `MatrixQ[`*expr*`,NumericQ]`	
	vectors and matrices where all elements are numeric
`VectorQ[`*expr*`,`*test*`]` , `MatrixQ[`*expr*`,`*test*`]`	
	vectors and matrices for which the function *test* yields `True` on every element
`ArrayQ[`*expr*`,`*d*`]`	full array with depth matching *d*

Some functions for testing mathematical properties of expressions.

The rule applies to all elements of the list that are numbers.

```
In[13]:= {2.3, 4, 7 / 8, a, b} /. (x_ /; NumberQ[x]) -> x^2
```

```
                       49
Out[13]= {5.29, 16, ——, a, b}
                       64
```

This definition applies only to vectors of integers.

```
In[14]:= mi[list_] := list^2 /; VectorQ[list, IntegerQ]
```

The definition is now used only in the first case.

```
In[15]:= {mi[{2, 3}], mi[{2.1, 2.2}], mi[{a, b}]}
Out[15]= {{4, 9}, mi[{2.1, 2.2}], mi[{a, b}]}
```

An important feature of all the *Mathematica* property-testing functions whose names end in `Q` is that they always return `False` if they cannot determine whether the expression you give has a particular property.

4561 is an integer, so this returns `True`.

```
In[16]:= IntegerQ[4561]
Out[16]= True
```

This returns `False`, since `x` is not known to be an integer.

```
In[17]:= IntegerQ[x]
Out[17]= False
```

Functions like `IntegerQ[`*x*`]` test whether *x* is explicitly an integer. With assertions like *x* ∈ `Integers` you can use `Refine`, `Simplify` and related functions to make inferences about symbolic variables *x*.

`SameQ[`*x*`,`*y*`]` or *x*`===`*y*	*x* and *y* are identical
`UnsameQ[`*x*`,`*y*`]` or *x*`=!=`*y*	*x* and *y* are not identical
`OrderedQ[{`*a*`,`*b*`,`…`}]`	*a*, *b*, … are in standard order
`MemberQ[`*expr*`,`*form*`]`	*form* matches an element of *expr*
`FreeQ[`*expr*`,`*form*`]`	*form* matches nothing in *expr*
`MatchQ[`*expr*`,`*form*`]`	*expr* matches the pattern *form*
`ValueQ[`*expr*`]`	a value has been defined for *expr*
`AtomQ[`*expr*`]`	*expr* has no subexpressions

Some functions for testing structural properties of expressions.

With `==`, the equation remains in symbolic form; `===` yields `False` unless the expressions are manifestly equal.

```
In[18]:= {x == y, x === y}
Out[18]= {x == y, False}
```

The expression `n` is not a *member* of the list `{x, x^n}`.

```
In[19]:= MemberQ[{x, x^n}, n]
Out[19]= False
```

However, `{x, x^n}` is not completely free of `n`.

```
In[20]:= FreeQ[{x, x^n}, n]
Out[20]= False
```

You can use `FreeQ` to define a "linearity" rule for `h`.

```
In[21]:= h[a_ b_, x_] := a h[b, x] /; FreeQ[a, x]
```

Terms free of `x` are pulled out of each `h`.

```
In[22]:= h[a b x, x] + h[2 (1 + x) x^2, x]
```

Out[22]= a b h[x, x] + 2 h$\left[x^2 (1 + x), x\right]$

pattern`?`*test*	a pattern which matches an expression only if *test* yields `True` when applied to the expression

Another way to constrain patterns.

The construction *pattern* `/;` *condition* allows you to evaluate a condition involving pattern names to determine whether there is a match. The construction *pattern*`?`*test* instead applies a function *test* to the whole expression matched by *pattern* to determine whether there is a match. Using `?` instead of `/;` sometimes leads to more succinct definitions.

With this definition matches for `x_` are tested with the function `NumberQ`.

```
In[23]:= p[x_?NumberQ] := x^2
```

The definition applies only when `p` has a numerical argument.

```
In[24]:= p[4.5] + p[3 / 2] + p[u]
Out[24]= 22.5 + p[u]
```

Here is a more complicated definition. Do not forget the parentheses around the pure function.

```
In[25]:= q[{x_Integer, y_Integer}?(Function[v, v.v > 4])] := qp[x + y]
```

The definition applies only in certain cases.

```
In[26]:= {q[{3, 4}], q[{1, 1}], q[{-5, -7}]}
Out[26]= {qp[7], q[{1, 1}], qp[-12]}
```

`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[27]:= Cases[{1, 0, 2, 0, 3}, Except[0]]
Out[27]= {1, 2, 3}
```

Except can take a pattern as an argument.

```
In[28]:= Cases[{a, b, 0, 1, 2, x, y}, Except[_Integer]]
Out[28]= {a, b, x, y}
```

This picks out integers that are not 0.

```
In[29]:= Cases[{a, b, 0, 1, 2, x, y}, Except[0, _Integer]]
Out[29]= {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*.

Patterns Involving Alternatives

$patt_1 \mid patt_2 \mid \ldots$	a pattern that can have one of several forms

Specifying patterns that involve alternatives.

This defines h to give p when its argument is either a or b.

```
In[1]:= h[a | b] := p
```

The first two cases give p.

```
In[2]:= {h[a], h[b], h[c], h[d]}
Out[2]= {p, p, h[c], h[d]}
```

You can also use alternatives in transformation rules.

```
In[3]:= {a, b, c, d} /. (a | b) -> p
Out[3]= {p, p, c, d}
```

Here is another example, in which one of the alternatives is itself a pattern.

```
In[4]:= {1, x, x^2, x^3, y^2} /. (x | x^_) -> q
Out[4]= {1, q, q, q, y^2}
```

When you use alternatives in patterns, you should make sure that the same set of names appear in each alternative. When a pattern like `(a[x_] | b[x_])` matches an expression, there will always be a definite expression that corresponds to the object `x`. If you try to match a pattern like `(a[x_] | b[y_])`, then there still will be definite expressions corresponding to `x` and `y`, but the unmatched one will be `Sequence[ ]`.

Here `f` is used to name the head, which can be either `a` or `b`.

```
In[5]:= {a[2], b[3], c[4], a[5]} /. (f : (a | b))[x_] -> r[f, x]
Out[5]= {r[a, 2], r[b, 3], c[4], r[a, 5]}
```

Pattern Sequences

In some cases you may need to specify pattern sequences that are more intricate than things like *x*__ or *x* ..; for such situations you can use `PatternSequence[`p_1, p_2, …`]`.

`PatternSequence[`p_1,p_2,…`]`	a sequence of arguments matching p_1, p_2, …

Pattern sequences.

This defines a function with two or more arguments, grouping the first two.

```
In[1]:= f[x : PatternSequence[_, _], y___] := p[{x}, {y}]
```

Evaluate the function for different numbers of arguments.

```
In[2]:= {f[1], f[1, 2], f[1, 2, 3, 4, 5]}
Out[2]= {f[1], p[{1, 2}, {}], p[{1, 2}, {3, 4, 5}]}
```

This picks out the longest run of the sequence *a*, *b* in the list.

```
In[3]:= {a, b, b, a, b, a, b, a, a, b} /. {___, x : Longest[PatternSequence[a, b] ..], ___} :→ {x}
Out[3]= {a, b, a, b}
```

The empty sequence, `PatternSequence[]`, is sometimes useful to specify an optional argument.

This picks out expressions with exactly one or two arguments.

```
In[4]:= {g[], g[1], g[1, 2], g[1, 2, 3]} /. x : g[_, _ | PatternSequence[]] :> p[x]
Out[4]= {g[], p[g[1]], p[g[1, 2]], g[1, 2, 3]}
```

Flat and Orderless Functions

Although *Mathematica* matches patterns in a purely structural fashion, its notion of structural equivalence is quite sophisticated. In particular, it takes account of properties such as commutativity and associativity in functions like `Plus` and `Times`.

This means, for example, that *Mathematica* considers the expressions $x + y$ and $y + x$ equivalent for the purposes of pattern matching. As a result, a pattern like `g[x_ + y_, x_]` can match not only `g[a + b, a]`, but also `g[a + b, b]`.

This expression has exactly the same form as the pattern.

```
In[1]:= g[a + b, a] /. g[x_ + y_, x_] -> p[x, y]
Out[1]= p[a, b]
```

In this case, the expression has to be put in the form `g[b + a, b]` in order to have the same structure as the pattern.

```
In[2]:= g[a + b, b] /. g[x_ + y_, x_] -> p[x, y]
Out[2]= p[b, a]
```

Whenever *Mathematica* encounters an *orderless* or *commutative* function such as `Plus` or `Times` in a pattern, it effectively tests all the possible orders of arguments to try and find a match. Sometimes, there may be several orderings that lead to matches. In such cases, *Mathematica* just uses the first ordering it finds. For example, `h[x_ + y_, x_ + z_]` could match `h[a + b, a + b]` with `x→a`, `y→b`, `z→b` or with `x→b`, `y→a`, `z→a`. *Mathematica* tries the case `x→a`, `y→b`, `z→b` first, and so uses this match.

This can match either with `x → a` or with `x → b`. *Mathematica* tries `x → a` first, and so uses this match.

```
In[3]:= h[a + b, a + b] /. h[x_ + y_, x_ + z_] -> p[x, y, z]
Out[3]= p[a, b, b]
```

ReplaceList shows both possible matches.

```
In[4]:= ReplaceList[h[a + b, a + b], h[x_ + y_, x_ + z_] -> p[x, y, z]]
Out[4]= {p[a, b, b], p[b, a, a]}
```

As discussed in "Attributes", *Mathematica* allows you to assign certain attributes to functions, which specify how those functions should be treated in evaluation and pattern matching. Functions can for example be assigned the attribute `Orderless`, which specifies that they should be treated as commutative or symmetric, and allows their arguments to be rearranged in trying to match patterns.

`Orderless`	commutative function: *f*[*b*, *c*, *a*], etc., are equivalent to *f*[*a*, *b*, *c*]
`Flat`	associative function: *f*[*f*[*a*], *b*], etc., are equivalent to *f*[*a*, *b*]
`OneIdentity`	*f*[*f*[*a*]], etc., are equivalent to *a*
`Attributes[`*f*`]`	give the attributes assigned to *f*
`SetAttributes[`*f*`,`*attr*`]`	add *attr* to the attributes of *f*
`ClearAttributes[`*f*`,`*attr*`]`	remove *attr* from the attributes of *f*

Some attributes that can be assigned to functions.

Plus has attributes Orderless and Flat, as well as others.

```
In[5]:= Attributes[Plus]
Out[5]= {Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

This defines q to be an orderless or commutative function.

```
In[6]:= SetAttributes[q, Orderless]
```

The arguments of q are automatically sorted into order.

```
In[7]:= q[b, a, c]
Out[7]= q[a, b, c]
```

Mathematica rearranges the arguments of q functions to find a match.

```
In[8]:= f[q[a, b], q[b, c]] /. f[q[x_, y_], q[x_, z_]] -> p[x, y, z]
Out[8]= p[b, a, c]
```

In addition to being orderless, functions like `Plus` and `Times` also have the property of being *flat* or *associative*. This means that you can effectively "parenthesize" their arguments in any way, so that, for example, `x + (y + z)` is equivalent to `x + y + z`, and so on.

Mathematica takes account of flatness in matching patterns. As a result, a pattern like `g[x_ + y_]` can match `g[a + b + c]`, with `x → a` and `y → (b + c)`.

The argument of `g` is written as `a + (b + c)` so as to match the pattern.

```
In[9]:=  g[a + b + c] /. g[x_ + y_] -> p[x, y]
Out[9]=  p[a, b + c]
```

If there are no other constraints, *Mathematica* will match `x_` to the first element of the sum.

```
In[10]:=  g[a + b + c + d] /. g[x_ + y_] -> p[x, y]
Out[10]=  p[a, b + c + d]
```

This shows all the possible matches.

```
In[11]:=  ReplaceList[g[a + b + c], g[x_ + y_] -> p[x, y]]
Out[11]=  {p[a, b + c], p[b, a + c], p[c, a + b], p[a + b, c], p[a + c, b], p[b + c, a]}
```

Here `x_` is forced to match `b + d`.

```
In[12]:=  g[a + b + c + d, b + d] /. g[x_ + y_, x_] -> p[x, y]
Out[12]=  p[b + d, a + c]
```

Mathematica can usually apply a transformation rule to a function only if the pattern in the rule covers all the arguments in the function. However, if you have a flat function, it is sometimes possible to apply transformation rules even though not all the arguments are covered.

This rule applies even though it does not cover all the terms in the sum.

```
In[13]:=  a + b + c /. a + c -> p
Out[13]=  b + p
```

This combines two of the terms in the sum.

```
In[14]:=  u[a] + u[b] + v[c] + v[d] /. u[x_] + u[y_] -> u[x + y]
Out[14]=  u[a + b] + v[c] + v[d]
```

Functions like `Plus` and `Times` are both flat and orderless. There are, however, some functions, such as `Dot`, which are flat, but not orderless.

Both x_ and y_ can match any sequence of terms in the dot product.

```
In[15]:= a.b.c.d.a.b /. x_.y_.x_ -> p[x, y]
Out[15]= p[a.b, c.d]
```

This assigns the attribute Flat to the function r.

```
In[16]:= SetAttributes[r, Flat]
```

Mathematica writes the expression in the form r[r[a, b], r[a, b]] to match the pattern.

```
In[17]:= r[a, b, a, b] /. r[x_, x_] -> rp[x]
Out[17]= rp[r[a, b]]
```

Mathematica writes this expression in the form r[a, r[r[b], r[b]], c] to match the pattern.

```
In[18]:= r[a, b, b, c] /. r[x_, x_] -> rp[x]
Out[18]= r[a, rp[r[b]], c]
```

In an ordinary function that is not flat, a pattern such as x_ matches an individual argument of the function. But in a function $f[a, b, c, \ldots]$ that is flat, x_ can match objects such as $f[b, c]$ which effectively correspond to a sequence of arguments. However, in the case where x_ matches a single argument in a flat function, the question comes up as to whether the object it matches is really just the argument a itself, or $f[a]$. *Mathematica* chooses the first of these cases if the function carries the attribute OneIdentity, and chooses the second case otherwise.

This adds the attribute OneIdentity to the function r.

```
In[19]:= SetAttributes[r, OneIdentity]
```

Now x_ matches individual arguments, without r wrapped around them.

```
In[20]:= r[a, b, b, c] /. r[x_, x_] -> rp[x]
Out[20]= r[a, rp[b], c]
```

The functions Plus, Times and Dot all have the attribute OneIdentity, reflecting the fact that Plus$[x]$ is equivalent to x, and so on. However, in representing mathematical objects, it is often convenient to deal with flat functions that do not have the attribute OneIdentity.

Functions with Variable Numbers of Arguments

Unless f is a flat function, a pattern like f[x_, y_] stands only for instances of the function with exactly two arguments. Sometimes you need to set up patterns that can allow any number of arguments.

You can do this using *multiple blanks*. While a single blank such as x_ stands for a single *Mathematica* expression, a double blank such as x__ stands for a sequence of one or more expressions.

Here x__ stands for the sequence of expressions (a, b, c).

```
In[1]:= f[a, b, c] /. f[x__] -> p[x, x, x]
Out[1]= p[a, b, c, a, b, c, a, b, c]
```

Here is a more complicated definition, which picks out pairs of duplicated elements in h.

```
In[2]:= h[a___, x_, b___, x_, c___] := hh[x] h[a, b, c]
```

The definition is applied twice, picking out the two paired elements.

```
In[3]:= h[2, 3, 2, 4, 5, 3]
Out[3]= h[4, 5] hh[2] hh[3]
```

"Double blanks" __ stand for sequences of one or more expressions. "Triple blanks" ___ stand for sequences of zero or more expressions. You should be very careful whenever you use triple blank patterns. It is easy to make a mistake that can lead to an infinite loop. For example, if you define p[x_, y___] := p[x] q[y], then typing in p[a] will lead to an infinite loop, with y repeatedly matching a sequence with zero elements. Unless you are sure you want to include the case of zero elements, you should always use double blanks rather than triple blanks.

_	any single expression
*x*_	any single expression, to be named *x*
__	any sequence of one or more expressions
*x*__	sequence named *x*
*x*__*h*	sequence of expressions, all of whose heads are *h*
___	any sequence of zero or more expressions
*x*___	sequence of zero or more expressions named *x*
*x*___*h*	sequence of zero or more expressions, all of whose heads are *h*

More kinds of pattern objects.

Notice that with flat functions such as `Plus` and `Times`, *Mathematica* automatically handles variable numbers of arguments, so you do not explicitly need to use double or triple blanks, as discussed in "Flat and Orderless Functions".

When you use multiple blanks, there are often several matches that are possible for a particular expression. By default, *Mathematica* tries first those matches that assign the shortest sequences of arguments to the first multiple blanks that appear in the pattern. You can change this order by wrapping `Longest` or `Shortest` around parts of the pattern.

`Longest[`*p*`]`	match the longest sequence consistent with the pattern *p*
`Shortest[`*p*`]`	match the shortest sequence consistent with the pattern *p*

Controlling the order in which matches are tried.

This gives a list of all the matches that *Mathematica* tries.

```
In[4]:= ReplaceList[f[a, b, c, d], f[x__, y__] -> g[{x}, {y}]]
Out[4]= {g[{a}, {b, c, d}], g[{a, b}, {c, d}], g[{a, b, c}, {d}]}
```

This forces *Mathematica* to try the longest matches for *x*__ first.

```
In[5]:= ReplaceList[f[a, b, c, d], f[Longest[x__], y__] -> g[{x}, {y}]]
Out[5]= {g[{a, b, c}, {d}], g[{a, b}, {c, d}], g[{a}, {b, c, d}]}
```

Many kinds of enumeration can be done by using `ReplaceList` with various kinds of patterns.

```
In[6]:= ReplaceList[f[a, b, c, d], f[___, x__] -> g[x]]
Out[6]= {g[a, b, c, d], g[b, c, d], g[c, d], g[d]}
```

This effectively enumerates all sublists with at least one element.

```
In[7]:= ReplaceList[f[a, b, c, d], f[___, x__, ___] -> g[x]]
Out[7]= {g[a], g[a, b], g[a, b, c], g[a, b, c, d], g[b], g[b, c], g[b, c, d], g[c], g[c, d], g[d]}
```

This tries the shortest matches for *x*__ first.

```
In[8]:= ReplaceList[f[a, b, c, d], f[___, Shortest[x__], ___] -> g[x]]
Out[8]= {g[a], g[b], g[c], g[d], g[a, b], g[b, c], g[c, d], g[a, b, c], g[b, c, d], g[a, b, c, d]}
```

Optional and Default Arguments

Sometimes you may want to set up functions where certain arguments, if omitted, are given "default values". The pattern *x*_**:** *v* stands for an object that can be omitted, and if so, will be replaced by the default value *v*.

This defines a function j with a required argument x, and optional arguments y and z, with default values 1 and 2, respectively.

```
In[1]:= j[x_, y_: 1, z_: 2] := jp[x, y, z]
```

The default value of z is used here.

```
In[2]:= j[a, b]
Out[2]= jp[a, b, 2]
```

Now the default values of both y and z are used.

```
In[3]:= j[a]
Out[3]= jp[a, 1, 2]
```

*x*_**:***v*	an expression which, if omitted, is taken to have default value *v*
*x*_*h***:***v*	an expression with head *h* and default value *v*
*x*_**.**	an expression with a built-in default value

Pattern objects with default values.

Some common *Mathematica* functions have built-in default values for their arguments. In such cases, you need not explicitly give the default value in *x*_**:** *v*, but instead you can use the more convenient notation *x*_**.** in which a built-in default value is assumed.

`x_+y_.`	default for `y` is `0`
`x_ y_.`	default for `y` is `1`
`x_^y_.`	default for `y` is `1`

Some patterns with optional pieces.

Here `a` matches the pattern `x_ + y_.` with `y` taken to have the default value `0`.

```
In[4]:= {f[a], f[a + b]} /. f[x_ + y_.] -> p[x, y]
Out[4]= {p[a, 0], p[b, a]}
```

Because `Plus` is a flat function, a pattern such as `x_ + y_` can match a sum with any number of terms. This pattern cannot, however, match a single term such as `a`. However, the pattern `x_ + y_.` contains an optional piece, and can match either an explicit sum of terms in which both `x_` and `y_` appear, or a single term `x_`, with `y` taken to be `0`.

Using constructs such as *x*`_.`, you can easily construct single patterns that match expressions with several different structures. This is particularly useful when you want to match several mathematically equal forms that do not have the same structure.

The pattern matches `g[a^2]`, but not `g[a + b]`.

```
In[5]:= {g[a^2], g[a + b]} /. g[x_^n_] -> p[x, n]
Out[5]= {p[a, 2], g[a + b]}
```

By giving a pattern in which the exponent is optional, you can match both cases.

```
In[6]:= {g[a^2], g[a + b]} /. g[x_^n_.] -> p[x, n]
Out[6]= {p[a, 2], p[a + b, 1]}
```

The pattern `a_. + b_. x_` matches any linear function of `x_`.

```
In[7]:= lin[a_. + b_. x_, x_] := p[a, b]
```

In this case, `b` → `1`.

```
In[8]:= lin[1 + x, x]
Out[8]= p[1, 1]
```

Here b → 1 and a → 0.

```
In[9]:=  lin[y, y]
Out[9]=  p[0, 1]
```

Standard *Mathematica* functions such as `Plus` and `Times` have built-in default values for their arguments. You can also set up defaults for your own functions, as described in "Patterns".

Sometimes it is convenient not to assign a default value to an optional argument; such arguments can be specified with the help of `PatternSequence[]`.

p|`PatternSequence[]`	optional pattern *p* with no default value

Optional argument without a default value.

The pattern matches an optional second argument of 2, without a default value.

```
In[10]:=  {g[1], g[1, 1], g[1, 2]} /. g[x_, 2 | PatternSequence[]] :> p[x]
Out[10]=  {p[1], g[1, 1], p[1]}
```

Setting Up Functions with Optional Arguments

When you define a complicated function, you will often want to let some of the arguments of the function be “optional”. If you do not give those arguments explicitly, you want them to take on certain “default” values.

Built-in *Mathematica* functions use two basic methods for dealing with optional arguments. You can choose between the same two methods when you define your own functions in *Mathematica*.

The first method is to have the meaning of each argument determined by its position, and then to allow one to drop arguments, replacing them by default values. Almost all built-in *Mathematica* functions that use this method drop arguments from the end. For example, the built-in function `Flatten[`*list*`,` *n*`]` allows you to drop the second argument, which is taken to have a default value of `Infinity`.

You can implement this kind of “positional” argument using `_:` patterns.

f[*x*_,*k*_: *kdef*]:=*value*	a typical definition for a function whose second argument is optional, with default value *kdef*

Defining a function with positional arguments.

This defines a function with an optional second argument. When the second argument is omitted, it is taken to have the default value `Infinity`.

```
In[1]:= f[list_, n_: Infinity] := f0[list, n]
```

Here is a function with two optional arguments.

```
In[2]:= fx[list_, n1_: 1, n2_: 2] := fx0[list, n1, n2]
```

Mathematica assumes that arguments are dropped from the end. As a result `m` here gives the value of `n1`, while `n2` has its default value of `2`.

```
In[3]:= fx[k, m]
Out[3]= fx0[k, m, 2]
```

The second method that built-in *Mathematica* functions use for dealing with optional arguments is to give explicit names to the optional arguments, and then to allow their values to be given using transformation rules. This method is particularly convenient for functions like `Plot` which have a very large number of optional parameters, only a few of which usually need to be set in any particular instance.

The typical arrangement is that values for "named" optional arguments can be specified by including the appropriate transformation rules at the end of the arguments to a particular function. Thus, for example, the rule `Joined -> True`, which specifies the setting for the named optional argument `Joined`, could appear as `ListPlot[`*list*`, Joined -> True]`.

When you set up named optional arguments for a function *f*, it is conventional to store the default values of these arguments as a list of transformation rules assigned to `Options[`*f*`]`.

f[*x*_,`OptionsPattern[]`]:=*value*	a typical definition for a function with zero or more named optional arguments
`OptionValue[`*name*`]`	the value of a named optional argument in the body of the function

Named arguments.

This sets up default values for two named optional arguments `opt1` and `opt2` in the function `fn`.

```
In[4]:= Options[fn] = {opt1 -> 1, opt2 -> 2}
Out[4]= {opt1 → 1, opt2 → 2}
```

Here is the definition for a function `fn` which allows zero or more named optional arguments to be specified.

```
In[5]:= fn[x_, OptionsPattern[]] := k[x, OptionValue[opt2]]
```

With no optional arguments specified, the default rule for `opt2` is used.

```
In[6]:= fn[4]
Out[6]= k[4, 2]
```

If you explicitly give a rule for `opt2`, it will override the default rules stored in `Options[fn]`.

```
In[7]:= fn[4, opt2 -> 7]
Out[7]= k[4, 7]
```

`FilterRules[`*opts*`,Options[`*name*`]]`	the rules in *opts* used as options by the function *f*
`FilterRules[`*opts*`,Except[Options[`*name*`]]]`	the rules in *opts* not used as options by the function *f*

Filtering options.

Sometimes when you write a function you will want to pass on options to functions that it calls.

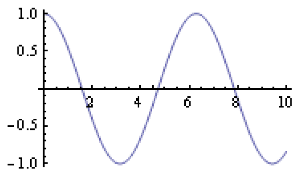
Here is a simple function that solves a differential equation numerically and plots its solution.

```
In[8]:= odeplot[de_, y_, {x_, x0_, x1_}, opts : OptionsPattern[]] :=
         Module[{sol},
          sol = NDSolve[de, y, {x, x0, x1}, FilterRules[{opts}, Options[NDSolve]]];
          If[Head[sol] === NDSolve,
           $Failed,
           Plot[Evaluate[y /. sol], {x, x0, x1},
            Evaluate[FilterRules[{opts}, Options[Plot]]]]
          ]
         ]
```

With no options given, the default options for NDSolve and Plot are used.

```
In[9]:= odeplot[{y''[x] + y[x] == 0, y[0] == 1, y'[0] == 0}, y[x], {x, 0, 10}]
```

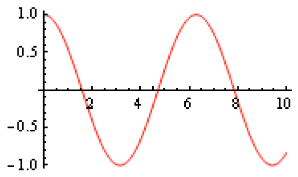
Out[9]=



This changes the method used by NDSolve and the color in the plot.

```
In[10]:= odeplot[{y''[x] + y[x] == 0, y[0] == 1, y'[0] == 0},
          y[x], {x, 0, 10}, Method → "ExplicitRungeKutta", PlotStyle → Red]
```

Out[10]=



Repeated Patterns

expr..	a pattern or other expression repeated one or more times
expr...	a pattern or other expression repeated zero or more times

Repeated patterns.

Multiple blanks such as *x*__ allow you to give patterns in which sequences of arbitrary expressions can occur. The *Mathematica* pattern repetition operators .. and ... allow you to construct patterns in which particular forms can be repeated any number of times. Thus, for example, f[a..] represents any expression of the form f[a], f[a, a], f[a, a, a], and so on.

The pattern f[a..] allows the argument a to be repeated any number of times.

```
In[1]:= Cases[{f[a], f[a, b, a], f[a, a, a]}, f[a..]]
Out[1]= {f[a], f[a, a, a]}
```

This pattern allows any number of a arguments, followed by any number of b arguments.

```
In[2]:= Cases[{f[a], f[a, a, b], f[a, b, a], f[a, b, b]}, f[a.., b..]]
Out[2]= {f[a, a, b], f[a, b, b]}
```

Here each argument can be either a or b.

```
In[3]:= Cases[{f[a], f[a, b, a], f[a, c, a]}, f[(a | b) ..]]
Out[3]= {f[a], f[a, b, a]}
```

You can use `..` and `...` to represent repetitions of any pattern. If the pattern contains named parts, then each instance of these parts must be identical.

This defines a function whose argument must consist of a list of pairs.

```
In[4]:= v[x : {{_, _} ..}] := Transpose[x]
```

The definition applies in this case.

```
In[5]:= v[{{a1, b1}, {a2, b2}, {a3, b3}}]
Out[5]= {{a1, a2, a3}, {b1, b2, b3}}
```

With this definition, the second elements of all the pairs must be the same.

```
In[6]:= vn[x : {{_, n_} ..}] := Transpose[x]
```

The definition applies in this case.

```
In[7]:= vn[{{a, 2}, {b, 2}, {c, 2}}]
Out[7]= {{a, b, c}, {2, 2, 2}}
```

The pattern *x*`..` can be extended to two arguments to control the number of repetitions more precisely.

p`..` or `Repeated[`*p*`]`	a pattern or other expression repeated one or more times
`Repeated[`*p*`,`*max*`]`	a pattern repeated up to *max* times
`Repeated[`*p*`,{`*min*`,`*max*`}]`	a pattern repeated between *min* and *max* times
`Repeated[`*p*`,{`*n*`}]`	a pattern repeated exactly *n* times

Controlling the number of repetitions.

This finds from two to three repetitions of the argument *a*.

```
In[8]:= Cases[{f[a], f[a, a], f[a, a, a], f[a, a, a, a]}, f[Repeated[a, {2, 3}]]]
Out[8]= {f[a, a], f[a, a, a]}
```

Verbatim Patterns

`Verbatim[`*expr*`]`	an expression that must be matched verbatim

Verbatim patterns.

Here the `x_` in the rule matches any expression.

```
In[1]:= {f[2], f[a], f[x_], f[y_]} /. f[x_] -> x^2
```

```
Out[1]= {4, a^2, x_^2, y_^2}
```

The `Verbatim` tells *Mathematica* that only the exact expression `x_` should be matched.

```
In[2]:= {f[2], f[a], f[x_], f[y_]} /. f[Verbatim[x_]] -> x^2
```

```
Out[2]= {f[2], f[a], x^2, f[y_]}
```

Patterns for Some Common Types of Expression

Using the objects described above, you can set up patterns for many kinds of expressions. In all cases, you must remember that the patterns must represent the structure of the expressions in *Mathematica* internal form, as shown by `FullForm`.

Especially for some common kinds of expressions, the standard output format used by *Mathematica* is not particularly close to the full internal form. But it is the internal form that you must use in setting up patterns.

`n_Integer`	an integer n
`x_Real`	an approximate real number x
`z_Complex`	a complex number z
`Complex[x_,y_]`	a complex number $x + i y$
`Complex[x_Integer,y_Integer]`	a complex number where both real and imaginary parts are integers
`(r_Rational|r_Integer)`	rational number or integer r
`Rational[n_,d_]`	a rational number $\frac{n}{d}$
`(x_/;NumberQ[x]&&Im[x]==0)`	a real number of any kind
`(x_/;NumberQ[x])`	a number of any kind

Some typical patterns for numbers.

Here are the full forms of some numbers.

```
In[1]:= {2, 2.5, 2.5 + I, 2 / 7} // FullForm
Out[1]//FullForm= List[2, 2.5`, Complex[2.5`, 1], Rational[2, 7]]
```

The rule picks out each piece of the complex numbers.

```
In[2]:= {2.5 - I, 3 + I} /. Complex[x_, y_] -> p[x, y]
Out[2]= {p[2.5, -1], p[3, 1]}
```

The fact that these expressions have different full forms means that you cannot use `x_ + I y_` to match a complex number.

```
In[3]:= {2.5 - I, x + I y} // FullForm
Out[3]//FullForm= List[Complex[2.5`, -1], Plus[x, Times[Complex[0, 1], y]]]
```

The pattern here matches both ordinary integers, and complex numbers where both the real and imaginary parts are integers.

```
In[4]:= Cases[{2.5 - I, 2, 3 + I, 2 - 0.5 I, 2 + 2 I}, _Integer | Complex[_Integer, _Integer]]
Out[4]= {2, 3 + i, 2 + 2 i}
```

As discussed in "Symbolic Computation", *Mathematica* puts all algebraic expressions into a standard form, in which they are written essentially as a sum of products of powers. In addition, ratios are converted into products of powers, with denominator terms having negative exponents, and differences are converted into sums with negated terms. To construct patterns for algebraic expressions, you must use this standard form. This form often differs from the way *Mathematica* prints out the algebraic expressions. But in all cases, you can find the full internal form using `FullForm[`*expr*`]`.

Here is a typical algebraic expression.

```
In[5]:= -1 / z^2 - z / y + 2 (x z)^2 y
```

Out[5]= $-\frac{1}{z^2} - \frac{z}{y} + 2\,x^2\,y\,z^2$

This is the full internal form of the expression.

```
In[6]:= FullForm[%]

Out[6]//FullForm= Plus[Times[-1, Power[z, -2]],
                   Times[-1, Power[y, -1], z], Times[2, Power[x, 2], y, Power[z, 2]]]
```

This is what you get by applying a transformation rule to all powers in the expression.

```
In[7]:= % /. x_^n_ -> e[x, n]

Out[7]= -z e[y, -1] - e[z, -2] + 2 y e[x, 2] e[z, 2]
```

*x*_+*y*_	a sum of two or more terms
*x*_+*y*_.	a single term or a sum of terms
*n*_Integer x_	an expression with an explicit integer multiplier
*a*_.+*b*_. *x*_	a linear expression $a + b\,x$
*x*_^*n*_	x^n with $n \neq 0$, 1
*x*_^*n*_.	x^n with $n \neq 0$
*a*_.+*b*_. *x*_+*c*_. *x*_^2	a quadratic expression with nonzero linear term

Some typical patterns for algebraic expressions.

This pattern picks out linear functions of `x`.

```
In[8]:= {1, a, x, 2 x, 1 + 2 x} /. a_. + b_. x -> p[a, b]

Out[8]= {1, a, p[0, 1], p[0, 2], p[1, 2]}
```

`x_List` or `x:{___}`	a list
`x_List/;VectorQ[x]`	a vector containing no sublists
`x_List/;VectorQ[x,NumberQ]`	a vector of numbers
`x:{___List}` or `x:{{___}...}`	a list of lists
`x_List/;MatrixQ[x]`	a matrix containing no sublists
`x_List/;MatrixQ[x,NumberQ]`	a matrix of numbers
`x:{{_,_}...}`	a list of pairs

Some typical patterns for lists.

This defines a function whose argument must be a list containing lists with either one or two elements.

```
In[9]:=  h[x : {({_} | {_, _}) ...}] := q
```

The definition applies in the second and third cases.

```
In[10]:= {h[{a, b}], h[{{a}, {b}}], h[{{a}, {b, c}}]}
Out[10]= {h[{a, b}], q, q}
```

An Example: Defining Your Own Integration Function

Now that we have introduced the basic features of patterns in *Mathematica*, we can use them to give a more or less complete example. We will show how you could define your own simple integration function in *Mathematica*.

From a mathematical point of view, the integration function is defined by a sequence of mathematical relations. By setting up transformation rules for patterns, you can implement these mathematical relations quite directly in *Mathematica*.

mathematical form	*Mathematica definition*
$\int (y+z)\,dx = \int y\,dx + \int z\,dx$	`integrate[y_+z_,x_]:=integrate[y,x]+integrate[z,x]`
$\int c y\,dx = c\int y\,dx$ (c independent of x)	`integrate[c_y_,x_]:=c integrate[y,x]/;FreeQ[c,x]`
$\int c\,dx = cx$	`integrate[c_,x_]:=cx/;FreeQ[c,x]`
$\int x^n\,dx = \frac{x^{(n+1)}}{n+1}$, $n \neq -1$	`integrate[x_^n_.,x_]:= x^(n+1)/(n+1)/;FreeQ[n,x]&&n!=-1`
$\int \frac{1}{a\,x+b}\,dx = \frac{\log(a\,x+b)}{a}$	`integrate[1/(a_.x_+b_.),x_]:= Log[ax+b]/a/;FreeQ[{a,b},x]`
$\int e^{a\,x+b}\,dx = \frac{1}{a}\,e^{a\,x+b}$	`integrate[Exp[a_.x_+b_.],x_]:= Exp[ax+b]/a/;FreeQ[{a,b},x]`

Definitions for an integration function.

This implements the linearity relation for integrals: $\int (y+z)\,dx = \int y\,dx + \int z\,dx$.

```
In[1]:= integrate[y_ + z_, x_] := integrate[y, x] + integrate[z, x]
```

The associativity of `Plus` makes the linearity relation work with any number of terms in the sum.

```
In[2]:= integrate[a x + b x^2 + 3, x]

Out[2]= integrate[3, x] + integrate[a x, x] + integrate[b x^2, x]
```

This makes `integrate` pull out factors that are independent of the integration variable `x`.

```
In[3]:= integrate[c_ y_, x_] := c integrate[y, x] /; FreeQ[c, x]
```

Mathematica tests each term in each product to see whether it satisfies the `FreeQ` condition, and so can be pulled out.

```
In[4]:= integrate[a x + b x^2 + 3, x]

Out[4]= integrate[3, x] + a integrate[x, x] + b integrate[x^2, x]
```

This gives the integral $\int c\,dx = c\,x$ of a constant.

```
In[5]:= integrate[c_, x_] := c x /; FreeQ[c, x]
```

Now the constant term in the sum can be integrated.

```
In[6]:= integrate[a x + b x^2 + 3, x]

Out[6]= 3 x + a integrate[x, x] + b integrate[x^2, x]
```

This gives the standard formula for the integral of x^n. By using the pattern `x_^n_.`, rather than `x_^n_`, we include the case of $x^1 = x$.

```
In[7]:= integrate[x_^n_., x_] := x^(n + 1) / (n + 1) /; FreeQ[n, x] && n != -1
```

Now this integral can be done completely.

```
In[8]:= integrate[a x + b x^2 + 3, x]
```

Out[8]= $3\,x + \frac{a\,x^2}{2} + \frac{b\,x^3}{3}$

Of course, the built-in integration function `Integrate` (with a capital `I`) could have done the integral anyway.

```
In[9]:= Integrate[a x + b x^2 + 3, x]
```

Out[9]= $3\,x + \frac{a\,x^2}{2} + \frac{b\,x^3}{3}$

Here is the rule for integrating the reciprocal of a linear function. The pattern `a_. x_ + b_.` stands for any linear function of `x`.

```
In[10]:= integrate[1 / (a_. x_ + b_.), x_] := Log[a x + b] / a /; FreeQ[{a, b}, x]
```

Here both `a` and `b` take on their default values.

```
In[11]:= integrate[1 / x, x]

Out[11]= Log[x]
```

Here is a more complicated case. The symbol `a` now matches `2 p`.

```
In[12]:= integrate[1 / (2 p x - 1), x]
```

Out[12]= $\frac{\text{Log}[-1 + 2\,p\,x]}{2\,p}$

You can go on and add many more rules for integration. Here is a rule for integrating exponentials.

```
In[13]:= integrate[Exp[a_. x_ + b_.], x_] := Exp[a x + b] / a /; FreeQ[{a, b}, x]
```

Transformation Rules and Definitions

Applying Transformation Rules

expr `/.` *lhs* `->` *rhs*	apply a transformation rule to *expr*
expr `/.` `{`*lhs*$_1$ `->` *rhs*$_1$`,` *lhs*$_2$ `->` *rhs*$_2$`,` …`}`	try a sequence of rules on each part of *expr*

Applying transformation rules.

The replacement operator `/.` (pronounced "slash-dot") applies rules to expressions.

```
In[1]:=  x + y /. x -> 3
Out[1]=  3 + y
```

You can give a list of rules to apply. Each rule will be tried once on each part of the expression.

```
In[2]:=  x + y /. {x -> a, y -> b}
Out[2]=  a + b
```

expr `/.` `{`*rules*$_1$`,` *rules*$_2$`,` …`}`	give a list of the results from applying each of the *rules*$_i$ to *expr*

Applying lists of transformation rules.

If you give a list of lists of rules, you get a list of results.

```
In[3]:=  x + y /. {{x -> 1, y -> 2}, {x -> 4, y -> 2}}
Out[3]=  {3, 6}
```

Functions such as `Solve` and `NSolve` return lists whose elements are lists of rules, each representing a solution.

```
In[4]:=  Solve[x^3 - 5 x^2 + 2 x + 8 == 0, x]
Out[4]=  {{x → -1}, {x → 2}, {x → 4}}
```

When you apply these rules, you get a list of results, one corresponding to each solution.

```
In[5]:=  x^2 + 6 /. %
Out[5]=  {7, 10, 22}
```

When you use *expr* /. *rules*, each rule is tried in turn on each part of *expr*. As soon as a rule applies, the appropriate transformation is made, and the resulting part is returned.

The rule for `x^3` is tried first; if it does not apply, the rule for `x^n_` is used.

```
In[6]:= {x^2, x^3, x^4} /. {x^3 -> u, x^n_ -> p[n]}
Out[6]= {p[2], u, p[4]}
```

A result is returned as soon as the rule has been applied, so the inner instance of `h` is not replaced.

```
In[7]:= h[x + h[y]] /. h[u_] -> u^2
```

Out[7]= $(x + h[y])^2$

The replacement *expr* /. *rules* tries each rule just once on each part of *expr*.

Since each rule is tried just once, this serves to swap `x` and `y`.

```
In[8]:= {x^2, y^3} /. {x -> y, y -> x}
```

Out[8]= $\{y^2, x^3\}$

You can use this notation to apply one set of rules, followed by another.

```
In[9]:= x^2 /. x -> (1 + y) /. y -> b
```

Out[9]= $(1 + b)^2$

Sometimes you may need to go on applying rules over and over again, until the expression you are working on no longer changes. You can do this using the repeated replacement operation *expr* //. *rules* (or `ReplaceRepeated[`*expr*, *rules*`]`).

expr /. *rules*	try rules once on each part of *expr*
expr //. *rules*	try rules repeatedly until the result no longer changes

Single and repeated rule application.

With the single replacement operator `/.` each rule is tried only once on each part of the expression.

```
In[10]:= x^2 + y^6 /. {x -> 2 + a, a -> 3}
```

Out[10]= $(2 + a)^2 + y^6$

With the repeated replacement operator `//.` the rules are tried repeatedly until the expression no longer changes.

```
In[11]:= x^2 + y^6 //. {x -> 2 + a, a -> 3}

Out[11]= 25 + y^6
```

Here the rule is applied only once.

```
In[12]:= log[a b c d] /. log[x_ y_] -> log[x] + log[y]

Out[12]= log[a] + log[b c d]
```

With the repeated replacement operator, the rule is applied repeatedly, until the result no longer changes.

```
In[13]:= log[a b c d] //. log[x_ y_] -> log[x] + log[y]

Out[13]= log[a] + log[b] + log[c] + log[d]
```

When you use `//.` (pronounced "slash-slash-dot"), *Mathematica* repeatedly passes through your expression, trying each of the rules given. It goes on doing this until it gets the same result on two successive passes.

If you give a set of rules that is circular, then `//.` can keep on getting different results forever. In practice, the maximum number of passes that `//.` makes on a particular expression is determined by the setting for the option `MaxIterations`. If you want to keep going for as long as possible, you can use `ReplaceRepeated[`*expr*`,` *rules*`, MaxIterations -> Infinity]`. You can always stop by explicitly interrupting *Mathematica*.

By setting the option `MaxIterations`, you can explicitly tell `ReplaceRepeated` how many times to try the rules you give.

```
In[14]:= ReplaceRepeated[x, x -> x + 1, MaxIterations -> 1000]

    ReplaceRepeated::rrlim : Exiting after x scanned 1000 times. >>

Out[14]= 1000 + x
```

The replacement operators `/.` and `//.` share the feature that they try each rule on every subpart of your expression. On the other hand, `Replace[`*expr*`,` *rules*`]` tries the rules only on the whole of *expr*, and not on any of its subparts.

You can use `Replace`, together with functions like `Map` and `MapAt`, to control exactly which parts of an expression a replacement is applied to. Remember that you can use the function `ReplacePart[`*expr*`,` *new*`,` *pos*`]` to replace part of an expression with a specific object.

The operator `/.` applies rules to all subparts of an expression.

```
In[15]:= x^2 /. x -> a
Out[15]= a^2
```

Without a level specification, `Replace` applies rules only to the whole expression.

```
In[16]:= Replace[x^2, x^2 -> b]
Out[16]= b
```

No replacement is done here.

```
In[17]:= Replace[x^2, x -> a]
Out[17]= x^2
```

This applies rules down to level 2, and so replaces `x`.

```
In[18]:= Replace[x^2, x -> a, 2]
Out[18]= a^2
```

expr `/.` *rules*	apply rules to all subparts of *expr*
`Replace[`*expr*`,`*rules*`]`	apply rules to the whole of *expr* only
`Replace[`*expr*`,`*rules*`,`*levspec*`]`	apply rules to parts of *expr* on levels specified by *levspec*

Applying rules to whole expressions.

`Replace` returns the result from using the first rule that applies.

```
In[19]:= Replace[f[u], {f[x_] -> x^2, f[x_] -> x^3}]
Out[19]= u^2
```

`ReplaceList` gives a list of the results from every rule that applies.

```
In[20]:= ReplaceList[f[u], {f[x_] -> x^2, f[x_] -> x^3}]
Out[20]= {u^2, u^3}
```

If a single rule can be applied in several ways, `ReplaceList` gives a list of all the results.

```
In[21]:= ReplaceList[a + b + c, x_ + y_ -> g[x, y]]
Out[21]= {g[a, b + c], g[b, a + c], g[c, a + b], g[a + b, c], g[a + c, b], g[b + c, a]}
```

This gives a list of ways of breaking the original list in two.

```
In[22]:= ReplaceList[{a, b, c, d}, {x__, y__} -> g[{x}, {y}]]
Out[22]= {g[{a}, {b, c, d}], g[{a, b}, {c, d}], g[{a, b, c}, {d}]}
```

This finds all sublists that are flanked by the same element.

```
In[23]:= ReplaceList[{a, b, c, a, d, b, d}, {___, x_, y__, x_, ___} -> g[x, {y}]]
Out[23]= {g[a, {b, c}], g[b, {c, a, d}], g[d, {b}]}
```

`Replace[`*expr*`,`*rules*`]`	apply *rules* in one way only
`ReplaceList[`*expr*`,`*rules*`]`	apply *rules* in all possible ways

Applying rules in one way or all possible ways.

Manipulating Sets of Transformation Rules

You can manipulate lists of transformation rules in *Mathematica* just like other symbolic expressions. It is common to assign a name to a rule or set of rules.

This assigns the "name" `sinexp` to the trigonometric expansion rule.

```
In[1]:= sinexp = Sin[2 x_] -> 2 Sin[x] Cos[x]
Out[1]= Sin[2 x_] → 2 Cos[x] Sin[x]
```

You can now request the rule "by name".

```
In[2]:= Sin[2 (1 + x) ^2] /. sinexp
Out[2]= 2 Cos[(1 + x)^2] Sin[(1 + x)^2]
```

You can use lists of rules to represent mathematical and other relations. Typically you will find it convenient to give names to the lists, so that you can easily specify the list you want in a particular case.

In most situations, it is only one rule from any given list that actually applies to a particular expression. Nevertheless, the `/.` operator tests each of the rules in the list in turn. If the list is very long, this process can take a long time.

Mathematica allows you to preprocess lists of rules so that `/.` can operate more quickly on them. You can take any list of rules and apply the function `Dispatch` to them. The result is a representation of the original list of rules, but including dispatch tables which allow `/.` to "dispatch" to potentially applicable rules immediately, rather than testing all the rules in turn.

Here is a list of rules for the first five factorials.

```
In[3]:= facs = Table[f[i] -> i!, {i, 5}]
Out[3]= {f[1] → 1, f[2] → 2, f[3] → 6, f[4] → 24, f[5] → 120}
```

This sets up dispatch tables that make the rules faster to use.

```
In[4]:= dfacs = Dispatch[facs]
Out[4]= Dispatch[{f[1] → 1, f[2] → 2, f[3] → 6, f[4] → 24, f[5] → 120}, -DispatchTables -]
```

You can apply the rules using the `/.` operator.

```
In[5]:= f[4] /. dfacs
Out[5]= 24
```

`Dispatch[`*rules*`]`	create a representation of a list of rules that includes dispatch tables
expr `/.` *drules*	apply rules that include dispatch tables

Creating and using dispatch tables.

For long lists of rules, you will find that setting up dispatch tables makes replacement operations much faster. This is particularly true when your rules are for individual symbols or other expressions that do not involve pattern objects. Once you have built dispatch tables in such cases, you will find that the `/.` operator takes a time that is more or less independent of the number of rules you have. Without dispatch tables, however, `/.` will take a time directly proportional to the total number of rules.

Making Definitions

The replacement operator `/.` allows you to apply transformation rules to a specific expression. Often, however, you want to have transformation rules automatically applied whenever possible.

You can do this by assigning explicit values to *Mathematica* expressions and patterns. Each assignment specifies a transformation rule to be applied whenever an expression of the appropriate form occurs.

expr /. *lhs*->*rhs*	apply a transformation rule to a specific expression
lhs=*rhs*	assign a value which defines a transformation rule to be used whenever possible

Manual and automatic application of transformation rules.

This applies a transformation rule for `x` to a specific expression.

```
In[1]:= (1 + x) ^6 /. x -> 3 - a
```

```
Out[1]= (4 - a)^6
```

By assigning a value to `x`, you tell *Mathematica* to apply a transformation rule for `x` whenever possible.

```
In[2]:= x = 3 - a
Out[2]= 3 - a
```

Now `x` is transformed automatically.

```
In[3]:= (1 + x) ^7
```

```
Out[3]= (4 - a)^7
```

You should realize that except inside constructs like `Module` and `Block`, all assignments you make in a *Mathematica* session are *permanent*. They continue to be used for the duration of the session, unless you explicitly clear or overwrite them.

The fact that assignments are permanent means that they must be made with care. Probably the single most common mistake in using *Mathematica* is to make an assignment for a variable like `x` at one point in your session, and then later to use `x` having forgotten about the assignment you made.

There are several ways to avoid this kind of mistake. First, you should avoid using assignments whenever possible, and instead use more controlled constructs such as the `/.` replacement operator. Second, you should explicitly use the deassignment operator `=.` or the function `Clear` to remove values you have assigned when you have finished with them.

Another important way to avoid mistakes is to think particularly carefully before assigning values to variables with common or simple names. You will often want to use a variable such as `x` as a symbolic parameter. But if you make an assignment such as `x = 3`, then `x` will be replaced by `3` whenever it occurs, and you can no longer use `x` as a symbolic parameter.

In general, you should be sure not to assign permanent values to any variables that you might want to use for more than one purpose. If at one point in your session you wanted the variable `c` to stand for the speed of light, you might assign it a value such as `3. * 10^8`. But then you cannot use `c` later in your session to stand, say, for an undetermined coefficient. One way to avoid this kind of problem is to make assignments only for variables with more explicit names, such as `SpeedOfLight`.

`x=.`	remove the value assigned to the object *x*
`Clear[x,y,…]`	clear all the values of *x*, *y*, …

Removing assignments.

This does not give what you might expect, because `x` still has the value you assigned it above.

```
In[4]:= Factor[x^2 - 1]
Out[4]= (-4 + a) (-2 + a)
```

This removes any value assigned to `x`.

```
In[5]:= Clear[x]
```

Now this gives the result you expect.

```
In[6]:= Factor[x^2 - 1]
Out[6]= (-1 + x) (1 + x)
```

Special Forms of Assignment

Particularly when you write procedural programs in *Mathematica*, you will often need to modify the value of a particular variable repeatedly. You can always do this by constructing the new value and explicitly performing an assignment such as *x* = *value*. *Mathematica*, however, provides special notations for incrementing the values of variables, and for some other common cases.

i++	increment the value of i by 1
i−−	decrement i
++i	pre-increment i
−−i	pre-decrement i
i+=di	add di to the value of i
i−=di	subtract di from i
x*=c	multiply x by c
x/=c	divide x by c

Modifying values of variables.

This assigns the value `7 x` to the variable `t`.

```
In[1]:= t = 7 x
Out[1]= 7 x
```

This increments the value of `t` by `18 x`.

```
In[2]:= t += 18 x
Out[2]= 25 x
```

The value of `t` has been modified.

```
In[3]:= t
Out[3]= 25 x
```

This sets `t` to `8`, multiplies its value by `7`, then gives the final value of `t`.

```
In[4]:= t = 8; t *= 7; t
Out[4]= 56
```

The value of `i++` is the value of `i` *before* the increment is done.

```
In[5]:= i = 5; Print[i++]; Print[i]

        5

        6
```

The value of `++i` is the value of `i` *after* the increment.

```
In[6]:= i = 5; Print[++i]; Print[i]

    6

    6
```

$x=y=value$	assign the same value to both x and y
$\{x, y\}=\{value_1, value_2\}$	assign different values to x and y
$\{x, y\}=\{y, x\}$	interchange the values of x and y

Assigning values to several variables at a time.

This assigns the value `5` to `x` and `8` to `y`.

```
In[7]:= {x, y} = {5, 8}
Out[7]= {5, 8}
```

This interchanges the values of `x` and `y`.

```
In[8]:= {x, y} = {y, x}
Out[8]= {8, 5}
```

Now `x` has value `8`.

```
In[9]:= x
Out[9]= 8
```

And `y` has value `5`.

```
In[10]:= y
Out[10]= 5
```

You can use assignments to lists to permute values of variables in any way.

```
In[11]:= {a, b, c} = {1, 2, 3}; {b, a, c} = {a, c, b}; {a, b, c}
Out[11]= {3, 1, 2}
```

When you write programs in *Mathematica*, you will sometimes find it convenient to take a list, and successively add elements to it. You can do this using the functions `PrependTo` and `AppendTo`.

`PrependTo[`*v*`,`*elem*`]`	prepend *elem* to the value of *v*
`AppendTo[`*v*`,`*elem*`]`	append *elem*
v`={`*v*`,`*elem*`}`	make a nested list containing *elem*

Assignments for modifying lists.

This assigns the value of `v` to be the list `{5, 7, 9}`.

```
In[12]:= v = {5, 7, 9}
Out[12]= {5, 7, 9}
```

This appends the element `11` to the value of `v`.

```
In[13]:= AppendTo[v, 11]
Out[13]= {5, 7, 9, 11}
```

Now the value of `v` has been modified.

```
In[14]:= v
Out[14]= {5, 7, 9, 11}
```

Although `AppendTo[`*v*`, `*elem*`]` is always equivalent to *v* = `Append[`*v*`, `*elem*`]`, it is often a convenient notation. However, you should realize that because of the way *Mathematica* stores lists, it is usually less efficient to add a sequence of elements to a particular list than to create a nested structure that consists, for example, of lists of length 2 at each level. When you have built up such a structure, you can always reduce it to a single list using `Flatten`.

This sets up a nested list structure for `w`.

```
In[15]:= w = {1}; Do[w = {w, k^2}, {k, 1, 4}]; w
Out[15]= {{{{{1}, 1}, 4}, 9}, 16}
```

You can use `Flatten` to unravel the structure.

```
In[16]:= Flatten[w]
Out[16]= {1, 1, 4, 9, 16}
```

Making Definitions for Indexed Objects

In many kinds of calculations, you need to set up "arrays" which contain sequences of expressions, each specified by a certain index. One way to implement arrays in *Mathematica* is by using lists. You can define a list, say $a = \{x, y, z, \ldots\}$, then access its elements using $a[[i]]$, or modify them using $a[[i]] = value$. This approach has a drawback, however, in that it requires you to fill in all the elements when you first create the list.

Often, it is more convenient to set up arrays in which you can fill in only those elements that you need at a particular time. You can do this by making definitions for expressions such as $a[i]$.

This defines a value for `a[1]`.

```
In[1]:= a[1] = 9
Out[1]= 9
```

This defines a value for `a[2]`.

```
In[2]:= a[2] = 7
Out[2]= 7
```

This shows all the values you have defined for expressions associated with `a` so far.

```
In[3]:= ?a

        Global`a

        a[1] = 9

        a[2] = 7
```

You can define a value for `a[5]`, even though you have not yet given values to `a[3]` and `a[4]`.

```
In[4]:= a[5] = 0
Out[4]= 0
```

This generates a list of the values of the `a[i]`.

```
In[5]:= Table[a[i], {i, 5}]
Out[5]= {9, 7, a[3], a[4], 0}
```

You can think of the expression `a[i]` as being like an "indexed" or "subscripted" variable.

`a[i]=value`	add or overwrite a value
`a[i]`	access a value
`a[i]=.`	remove a value
`?a`	show all defined values
`Clear[a]`	clear all defined values
`Table[a[i],{i,1,n}]` or `Array[a,n]`	convert to an explicit `List`

Manipulating indexed variables.

When you have an expression of the form `a[i]`, there is no requirement that the "index" *i* be a number. In fact, *Mathematica* allows the index to be any expression whatsoever. By using indices that are symbols, you can for example build up simple databases in *Mathematica*.

This defines the "object" `area` with "index" `square` to have value `1`.

```
In[6]:= area[square] = 1
Out[6]= 1
```

This adds another result to the `area` "database".

```
In[7]:= area[triangle] = 1 / 2
```

Out[7]= $\frac{1}{2}$

Here are the entries in the `area` database so far.

```
In[8]:= ? area
```

```
Global`area
```

`area[square] = 1`

`area[triangle] =` $\frac{1}{2}$

You can use these definitions wherever you want. You have not yet assigned a value for `area[pentagon]`.

```
In[9]:= 4 area[square] + area[pentagon]
Out[9]= 4 + area[pentagon]
```

Making Definitions for Functions

"Defining Functions" discusses how you can define functions in *Mathematica*. In a typical case, you would type in `f[x_] = x^2` to define a function `f`. (Actually, the definitions in "Defining Functions" use the `:=` operator, rather than the `=` one. "Immediate and Delayed Definitions" explains exactly when to use each of the `:=` and `=` operators.)

The definition `f[x_] = x^2` specifies that whenever *Mathematica* encounters an expression which matches the pattern `f[x_]`, it should replace the expression by `x^2`. Since the pattern `f[x_]` matches all expressions of the form `f[`*anything*`]`, the definition applies to functions `f` with any "argument".

Function definitions like `f[x_] = x^2` can be compared with definitions like `f[a] = b` for indexed variables discussed in "Making Definitions for Indexed Objects". The definition `f[a] = b` specifies that whenever the *particular* expression `f[a]` occurs, it is to be replaced by `b`. But the definition says nothing about expressions such as `f[y]`, where `f` appears with another "index".

To define a "function", you need to specify values for expressions of the form `f[`*x*`]`, where the argument *x* can be anything. You can do this by giving a definition for the pattern `f[x_]`, where the pattern object `x_` stands for any expression.

`f[`*x*`]=`*value*	definition for a *specific expression* *x*
`f[`*x*`_]=`*value*	definition for *any expression*, referred to as *x*

The difference between defining an indexed variable and a function.

Making definitions for `f[2]` or `f[a]` can be thought of as being like giving values to various elements of an "array" named `f`. Making a definition for `f[x_]` is like giving a value for a set of "array elements" with arbitrary "indices". In fact, you can actually think of any function as being like an array with an arbitrarily variable index.

In mathematical terms, you can think of `f` as a *mapping*. When you define values for, say, `f[1]` and `f[2]`, you specify the image of this mapping for various discrete points in its domain. Defining a value for `f[x_]` specifies the image of `f` on a continuum of points.

This defines a transformation rule for the *specific expression* f[x].

```
In[1]:= f[x] = u
Out[1]= u
```

When the specific expression f[x] appears, it is replaced by u. Other expressions of the form f[*argument*] are, however, not modified.

```
In[2]:= f[x] + f[y]
Out[2]= u + f[y]
```

This defines a value for f with *any expression* as an "argument".

```
In[3]:= f[x_] = x^2
```

Out[3]= x^2

The old definition for the *specific expression* f[x] is still used, but the new general definition for f[x_] is now used to find a value for f[y].

```
In[4]:= f[x] + f[y]
```

Out[4]= $u + y^2$

This removes all definitions for f.

```
In[5]:= Clear[f]
```

Mathematica allows you to define transformation rules for any expression or pattern. You can mix definitions for specific expressions such as f[1] or f[a] with definitions for patterns such as f[x_].

Many kinds of mathematical functions can be set up by mixing specific and general definitions in *Mathematica*. As an example, consider the factorial function. This particular function is in fact built into *Mathematica* (it is written *n*!). But you can use *Mathematica* definitions to set up the function for yourself.

The standard mathematical definition for the factorial function can be entered almost directly into *Mathematica*, in the form: f[n_] := n f[n - 1]; f[1] = 1. This definition specifies that for any *n*, f[*n*] should be replaced by *n* f[*n* - 1], except that when *n* is 1, f[1] should simply be replaced by 1.

Here is the value of the factorial function with argument 1.

```
In[6]:= f[1] = 1
Out[6]= 1
```

Here is the general recursion relation for the factorial function.

```
In[7]:= f[n_] := n f[n - 1]
```

Now you can use these definitions to find values for the factorial function.

```
In[8]:= f[10]
Out[8]= 3 628 800
```

The results are the same as you get from the built-in version of factorial.

```
In[9]:= 10 !
Out[9]= 3 628 800
```

The Ordering of Definitions

When you make a sequence of definitions in *Mathematica*, some may be more general than others. *Mathematica* follows the principle of trying to put more general definitions after more specific ones. This means that special cases of rules are typically tried before more general cases.

This behavior is crucial to the factorial function example given in "Making Definitions for Functions". Regardless of the order in which you entered them, *Mathematica* will always put the rule for the special case `f[1]` ahead of the rule for the general case `f[n_]`. This means that when *Mathematica* looks for the value of an expression of the form `f[`*n*`]`, it tries the special case `f[1]` first, and only if this does not apply, it tries the general case `f[n_]`. As a result, when you ask for `f[5]`, *Mathematica* will keep on using the general rule until the "end condition" rule for `f[1]` applies.

- *Mathematica* tries to put specific definitions before more general definitions.

Treatment of definitions in *Mathematica*.

If *Mathematica* did not follow the principle of putting special rules before more general ones, then the special rules would always be "shadowed" by more general ones. In the factorial example, if the rule for `f[n_]` was ahead of the rule for `f[1]`, then even when *Mathematica* tried to evaluate `f[1]`, it would use the general `f[n_]` rule, and it would never find the special `f[1]` rule.

Here is a general definition for `f[n_]`.

```
In[1]:= f[n_] := n f[n - 1]
```

Here is a definition for the special case `f[1]`.

```
In[2]:= f[1] = 1
Out[2]= 1
```

Mathematica puts the special case before the general one.

```
In[3]:= ? f
```

```
Global`f

f[1] = 1

f[n_] := n f[n - 1]
```

In the factorial function example used above, it is clear which rule is more general. Often, however, there is no definite ordering in generality of the rules you give. In such cases, *Mathematica* simply tries the rules in the order you give them.

These rules have no definite ordering in generality.

```
In[4]:= log[x_ y_] := log[x] + log[y]; log[x_^n_] := n log[x]
```

Mathematica stores the rules in the order you gave them.

```
In[5]:= ? log
```

```
Global`log

log[x_ y_] := log[x] + log[y]

log[x_^n_] := n log[x]
```

This rule is a special case of the rule for `log[x_ y_]`.

```
In[6]:= log[2 x_] := log[x] + log2
```

Mathematica puts the special rule before the more general one.

```
In[7]:= ? log
```

```
Global`log

log[2 x_] := log[x] + log2

log[x_ y_] := log[x] + log[y]

log[x_^n_] := n log[x]
```

Although in many practical cases, *Mathematica* can recognize when one rule is more general than another, you should realize that this is not always possible. For example, if two rules both contain complicated `/;` conditions, it may not be possible to work out which is more general, and, in fact, there may not be a definite ordering. Whenever the appropriate ordering is not clear, *Mathematica* stores rules in the order you give them.

Immediate and Delayed Definitions

You may have noticed that there are two different ways to make assignments in *Mathematica*: *lhs* = *rhs* and *lhs* := *rhs*. The basic difference between these forms is *when* the expression *rhs* is evaluated. *lhs* = *rhs* is an *immediate assignment*, in which *rhs* is evaluated at the time when the assignment is made. *lhs* := *rhs*, on the other hand, is a *delayed assignment*, in which *rhs* is not evaluated when the assignment is made, but is instead evaluated each time the value of *lhs* is requested.

lhs=*rhs* (immediate assignment)	*rhs* is evaluated when the assignment is made
lhs:=*rhs* (delayed assignment)	*rhs* is evaluated each time the value of *lhs* is requested

The two types of assignments in *Mathematica*.

This uses the `:=` operator to define the function `ex`.

```
In[1]:= ex[x_] := Expand[(1 + x)^2]
```

Because `:=` was used, the definition is maintained in an unevaluated form.

```
In[2]:= ? ex

   Global`ex

   ex[x_] := Expand[(1 + x)^2]
```

When you make an assignment with the `=` operator, the right-hand side is evaluated immediately.

```
In[3]:= iex[x_] = Expand[(1 + x)^2]
```

Out[3]= $1 + 2 x + x^2$

The definition now stored is the result of the `Expand` command.

```
In[4]:= ? iex

   Global`iex
```

$iex[x\_] = 1 + 2 x + x^2$

When you execute `ex`, the `Expand` is performed.

```
In[5]:= ex[y + 2]
```

Out[5]= $9 + 6 y + y^2$

`iex` simply substitutes its argument into the already expanded form, giving a different answer.

```
In[6]:= iex[y + 2]
```

Out[6]= $1 + 2 (2 + y) + (2 + y)^2$

As you can see from the example above, both `=` and `:=` can be useful in defining functions, but they have different meanings, and you must be careful about which one to use in a particular case.

One rule of thumb is the following. If you think of an assignment as giving the final "value" of an expression, use the `=` operator. If instead you think of the assignment as specifying a "command" for finding the value, use the `:=` operator. If in doubt, it is usually better to use the `:=` operator than the `=` one.

lhs=*rhs*	*rhs* is intended to be the "final value" of *lhs* (e.g., `f[x_] = 1 - x^2`)
lhs:=*rhs*	*rhs* gives a "command" or "program" to be executed whenever you ask for the value of *lhs* (e.g., `f[x_] := Expand[1 - x^2]`)

Interpretations of assignments with the = and := operators.

Although := is probably used more often than = in defining functions, there is one important case in which you must use = to define a function. If you do a calculation, and get an answer in terms of a symbolic parameter *x*, you often want to go on and find results for various specific values of *x*. One way to do this is to use the /. operator to apply appropriate rules for *x* in each case. It is usually more convenient, however, to use = to define a function whose argument is *x*.

Here is an expression involving x.

```
In[7]:=  D[Log[Sin[x]]^2, x]
Out[7]=  2 Cot[x] Log[Sin[x]]
```

This defines a function whose argument is the value to be taken for x.

```
In[8]:=  dlog[x_] = %
Out[8]=  2 Cot[x] Log[Sin[x]]
```

Here is the result when x is taken to be 1 + a.

```
In[9]:=  dlog[1 + a]
Out[9]=  2 Cot[1 + a] Log[Sin[1 + a]]
```

An important point to notice in the example above is that there is nothing special about the name x that appears in the x_ pattern. It is just a symbol, indistinguishable from an x that appears in any other expression.

f[*x*_]=*expr*	define a function which gives the value *expr* for any particular value of *x*

Defining functions for evaluating expressions.

You can use `=` and `:=` not only to define functions, but also to assign values to variables. If you type *x* = *value*, then *value* is immediately evaluated, and the result is assigned to *x*. On the other hand, if you type *x* := *value*, then *value* is not immediately evaluated. Instead, it is maintained in an unevaluated form, and is evaluated afresh each time *x* is used.

This evaluates `RandomReal[]` to find a pseudorandom number, then assigns this number to `r1`.

```
In[10]:= r1 = RandomReal[]
Out[10]= 0.0560708
```

Here `RandomReal[]` is maintained in an unevaluated form, to be evaluated afresh each time `r2` is used.

```
In[11]:= r2 := RandomReal[]
```

Here are values for `r1` and `r2`.

```
In[12]:= {r1, r2}
Out[12]= {0.0560708, 0.6303}
```

The value of `r1` never changes. Every time `r2` is used, however, a new pseudorandom number is generated.

```
In[13]:= {r1, r2}
Out[13]= {0.0560708, 0.359894}
```

The distinction between immediate and delayed assignments is particularly important when you set up chains of assignments.

This defines `a` to be `1`.

```
In[14]:= a = 1
Out[14]= 1
```

Here `a + 2` is evaluated to give `3`, and the result is assigned to be the value of `ri`.

```
In[15]:= ri = a + 2
Out[15]= 3
```

Here `a + 2` is maintained in an unevaluated form, to be evaluated every time the value of `rd` is requested.

```
In[16]:= rd := a + 2
```

In this case, `ri` and `rd` give the same values.

```
In[17]:= {ri, rd}
Out[17]= {3, 3}
```

Now the value of `a` is changed.

```
In[18]:= a = 2
Out[18]= 2
```

Now `rd` uses the new value for `a`, while `ri` keeps its original value.

```
In[19]:= {ri, rd}
Out[19]= {3, 4}
```

You can use delayed assignments such as *t* `:=` *rhs* to set up variables whose values you can find in a variety of different "environments". Every time you ask for *t*, the expression *rhs* is evaluated using the current values of the objects on which it depends.

The right-hand side of the delayed assignment is maintained in an unevaluated form.

```
In[20]:= t := {a, Factor[x^a - 1]}
```

This sets `a` to `4`, then finds the value of `t`.

```
In[21]:= a = 4; t
Out[21]= {4, (-1 + x) (1 + x) (1 + x^2)}
```

Here `a` is 6.

```
In[22]:= a = 6; t
Out[22]= {6, (-1 + x) (1 + x) (1 - x + x^2) (1 + x + x^2)}
```

In the example above, the symbol `a` acts as a "global variable", whose value affects the value of `t`. When you have a large number of parameters, many of which change only occasionally, you may find this kind of setup convenient. However, you should realize that implicit or hidden dependence of one variable on others can often become quite confusing. When possible, you should make all dependencies explicit, by defining functions which take all necessary parameters as arguments.

lhs->*rhs*	*rhs* is evaluated when the rule is given
lhs:>*rhs*	*rhs* is evaluated when the rule is used

Two types of transformation rules in *Mathematica*.

Just as you can make immediate and delayed assignments in *Mathematica*, so you can also set up immediate and delayed transformation rules.

The right-hand side of this rule is evaluated when you give the rule.

```
In[23]:= f[x_] -> Expand[(1 + x)^2]
Out[23]= f[x_] → 1 + 2 x + x^2
```

A rule like this is probably not particularly useful.

```
In[24]:= f[x_] -> Expand[x]
Out[24]= f[x_] → x
```

Here the right-hand side of the rule is maintained in an unevaluated form, to be evaluated every time the rule is used.

```
In[25]:= f[x_] :> Expand[x]
Out[25]= f[x_] :→ Expand[x]
```

Applying the rule causes the expansion to be done.

```
In[26]:= f[(1 + p)^2] /. f[x_] :> Expand[x]
Out[26]= 1 + 2 p + p^2
```

In analogy with assignments, you should typically use -> when you want to replace an expression with a definite value, and you should use :> when you want to give a command for finding the value.

Functions That Remember Values They Have Found

When you make a function definition using :=, the value of the function is recomputed every time you ask for it. In some kinds of calculations, you may end up asking for the same function value many times. You can save time in these cases by having *Mathematica* remember all the function values it finds. Here is an "idiom" for defining a function that does this.

`f[x_]:=f[x]=`*rhs*	define a function which remembers values that it finds

Defining a function that remembers values it finds.

This defines a function `f` which stores all values that it finds.

```
In[1]:= f[x_] := f[x] = f[x - 1] + f[x - 2]
```

Here are the end conditions for the recursive function `f`.

```
In[2]:= f[0] = f[1] = 1
Out[2]= 1
```

Here is the original definition of `f`.

```
In[3]:= ? f

Global`f

f[0] = 1

f[1] = 1

f[x_] := f[x] = f[x - 1] + f[x - 2]
```

This computes `f[5]`. The computation involves finding the sequence of values `f[5]`, `f[4]`, … `f[2]`.

```
In[4]:= f[5]
Out[4]= 8
```

All the values of `f` found so far are explicitly stored.

```
In[5]:= ? f
```

```
Global`f

f[0] = 1

f[1] = 1

f[2] = 2

f[3] = 3

f[4] = 5

f[5] = 8

f[x_] := f[x] = f[x - 1] + f[x - 2]
```

If you ask for `f[5]` again, *Mathematica* can just look up the value immediately; it does not have to recompute it.

```
In[6]:= f[5]
Out[6]= 8
```

You can see how a definition like `f[x_] := f[x] = f[x - 1] + f[x - 2]` works. The function `f[x_]` is defined to be the "program" `f[x] = f[x - 1] + f[x - 2]`. When you ask for a value of the function `f`, the "program" is executed. The program first calculates the value of `f[x - 1] + f[x - 2]`, then saves the result as `f[x]`.

It is often a good idea to use functions that remember values when you implement mathematical *recursion relations* in *Mathematica*. In a typical case, a recursion relation gives the value of a function f with an integer argument x in terms of values of the same function with arguments $x-1$, $x-2$, etc. The Fibonacci function definition $f(x) = f(x-1) + f(x-2)$ used above is an example of this kind of recursion relation. The point is that if you calculate say $f(10)$ by just applying the recursion relation over and over again, you end up having to recalculate quantities like $f(5)$ many times. In a case like this, it is therefore better just to *remember* the value of $f(5)$, and look it up when you need it, rather than having to recalculate it.

There is of course a trade-off involved in remembering values. It is faster to find a particular value, but it takes more memory space to store all of them. You should usually define functions to remember values only if the total number of different values that will be produced is comparatively small, or the expense of recomputing them is very great.

Associating Definitions with Different Symbols

When you make a definition in the form *f*[*args*] = *rhs* or *f*[*args*] := *rhs*, *Mathematica* associates your definition with the object *f*. This means, for example, that such definitions are displayed when you type ? *f*. In general, definitions for expressions in which the symbol *f* appears as the head are termed *downvalues* of *f*.

Mathematica however also supports *upvalues*, which allow definitions to be associated with symbols that do not appear directly as their head.

Consider for example a definition like `Exp[g[x_]] :=` *rhs*. One possibility is that this definition could be associated with the symbol `Exp`, and considered as a downvalue of `Exp`. This is however probably not the best thing either from the point of view of organization or efficiency.

Better is to consider `Exp[g[x_]] :=` *rhs* to be associated with `g`, and to correspond to an upvalue of `g`.

f[*args*]:=*rhs*	define a downvalue for *f*
f[*g*[*args*],…]^:=*rhs*	define an upvalue for *g*

Associating definitions with different symbols.

This is taken to define a downvalue for `f`.

```
In[1]:= f[g[x_]] := fg[x]
```

You can see the definition when you ask about `f`.

```
In[2]:= ? f
```

```
Global`f

f[g[x_]] := fg[x]
```

This defines an upvalue for `g`.

```
In[3]:= Exp[g[x_]] ^:= expg[x]
```

The definition is associated with `g`.

```
In[4]:= ? g
```

```
Global`g
```

$e^{g[x\_]}$ ^:= expg[x]

It is not associated with Exp.

```
In[5]:=  ?? Exp
```

Exp[*z*] is the exponential function. ≫

```
Attributes[Exp] = {Listable, NumericFunction, Protected, ReadProtecte
```

The definition is used to evaluate this expression.

```
In[6]:=  Exp[g[5]]
Out[6]= expg[5]
```

In simple cases, you will get the same answers to calculations whether you give a definition for *f*[*g*[*x*]] as a downvalue for *f* or an upvalue for *g*. However, one of the two choices is usually much more natural and efficient than the other.

A good rule of thumb is that a definition for *f*[*g*[*x*]] should be given as an upvalue for *g* in cases where the function *f* is more common than *g*. Thus, for example, in the case of Exp[*g*[*x*]], Exp is a built-in *Mathematica* function, while *g* is presumably a function you have added. In such a case, you will typically think of definitions for Exp[*g*[*x*]] as giving relations satisfied by *g*. As a result, it is more natural to treat the definitions as upvalues for *g* than as downvalues for Exp.

This gives the definition as an upvalue for g.

```
In[7]:=  g /: g[x_] + g[y_] := gplus[x, y]
```

Here are the definitions for g so far.

```
In[8]:=  ? g
```

```
Global`g
```

$e^{g[x\_]}$ ^:= expg[x]

```
g[x_] + g[y_] ^:= gplus[x, y]
```

The definition for a sum of g's is used whenever possible.

```
In[9]:=  g[5] + g[7]
Out[9]= gplus[5, 7]
```

Since the full form of the pattern `g[x_] + g[y_]` is `Plus[g[x_], g[y_]]`, a definition for this pattern could be given as a downvalue for `Plus`. It is almost always better, however, to give the definition as an upvalue for `g`.

In general, whenever *Mathematica* encounters a particular function, it tries all the definitions you have given for that function. If you had made the definition for `g[x_] + g[y_]` a downvalue for `Plus`, then *Mathematica* would have tried this definition whenever `Plus` occurs. The definition would thus be tested every time *Mathematica* added expressions together, making this very common operation slower in all cases.

However, by giving a definition for `g[x_] + g[y_]` as an upvalue for `g`, you associate the definition with `g`. In this case, *Mathematica* only tries the definition when it finds a `g` inside a function such as `Plus`. Since `g` presumably occurs much less frequently than `Plus`, this is a much more efficient procedure.

f[*g*]^=*value* or *f*[*g*[*args*]]^=*value*	make assignments to be associated with *g*, rather than *f*
f[*g*]^:=*value* or *f*[*g*[*args*]]^:=*value*	make delayed assignments associated with *g*
f[arg_1,arg_2,…]^=*value*	make assignments associated with the heads of *all* the arg_i

Shorter ways to define upvalues.

A typical use of upvalues is in setting up a "database" of properties of a particular object. With upvalues, you can associate each definition you make with the object that it concerns, rather than with the property you are specifying.

This defines an upvalue for `square` which gives its area.

```
In[10]:=  area[square] ^= 1
Out[10]= 1
```

This adds a definition for the perimeter.

```
In[11]:= perimeter[square] ^= 4
Out[11]= 4
```

Both definitions are now associated with the object `square`.

```
In[12]:= ? square
```

```
Global`square

area[square] ^= 1

perimeter[square] ^= 4
```

In general, you can associate definitions for an expression with any symbol that occurs at a sufficiently high level in the expression. With an expression of the form *f*[*args*], you can define an upvalue for a symbol *g* so long as either *g* itself, or an object with head *g*, occurs in *args*. If *g* occurs at a lower level in an expression, however, you cannot associate definitions with it.

`g` occurs as the head of an argument, so you can associate a definition with it.

```
In[13]:= g /: h[w[x_], g[y_]] := hwg[x, y]
```

Here `g` appears too deep in the left-hand side for you to associate a definition with it.

```
In[14]:= g /: h[w[g[x_]], y_] := hw[x, y]

         TagSetDelayed::tagpos : Tag g in h[w[g[x_]], y_] is too deep for an assigned rule to be found. ≫

Out[14]= $Failed
```

f[…]:=*rhs*	downvalue for *f*
f/:*f*[*g*[…]][…]:=*rhs*	downvalue for *f*
g/:*f*[…,*g*,…]:=*rhs*	upvalue for *g*
g/:*f*[…,*g*[…],…]:=*rhs*	upvalue for *g*

Possible positions for symbols in definitions.

As discussed in "The Meaning of Expressions", you can use *Mathematica* symbols as "tags", to indicate the "type" of an expression. For example, complex numbers in *Mathematica* are represented internally in the form `Complex`[*x*, *y*], where the symbol `Complex` serves as a tag to indicate that the object is a complex number.

Upvalues provide a convenient mechanism for specifying how operations act on objects that are tagged to have a certain type. For example, you might want to introduce a class of abstract mathematical objects of type `quat`. You can represent each object of this type by a *Mathematica* expression of the form `quat[`*data*`]`.

In a typical case, you might want `quat` objects to have special properties with respect to arithmetic operations such as addition and multiplication. You can set up such properties by defining upvalues for `quat` with respect to `Plus` and `Times`.

This defines an upvalue for `quat` with respect to `Plus`.

```
In[15]:= quat[x_] + quat[y_] ^:= quat[x + y]
```

The upvalue you have defined is used to simplify this expression.

```
In[16]:= quat[a] + quat[b] + quat[c]
Out[16]= quat[a + b + c]
```

When you define an upvalue for `quat` with respect to an operation like `Plus`, what you are effectively doing is to extend the domain of the `Plus` operation to include `quat` objects. You are telling *Mathematica* to use special rules for addition in the case where the things to be added together are `quat` objects.

In defining addition for `quat` objects, you could always have a special addition operation, say `quatPlus`, to which you assign an appropriate downvalue. It is usually much more convenient, however, to use the standard *Mathematica* `Plus` operation to represent addition, but then to "overload" this operation by specifying special behavior when `quat` objects are encountered.

You can think of upvalues as a way to implement certain aspects of object-oriented programming. A symbol like `quat` represents a particular type of object. Then the various upvalues for `quat` specify "methods" that define how `quat` objects should behave under certain operations, or on receipt of certain "messages".

Defining Numerical Values

If you make a definition such as `f[x_] :=` *value*, *Mathematica* will use the value you give for any `f` function it encounters. In some cases, however, you may want to define a value that is to be used specifically when you ask for numerical values.

expr=*value*	define a value to be used whenever possible
N[*expr*]=*value*	define a value to be used for numerical approximation

Defining ordinary and numerical values.

This defines a numerical value for the function `f`.

```
In[1]:= N[f[x_]] := Sum[x^-i / i^2, {i, 20}]
```

Defining the numerical value does not tell *Mathematica* anything about the ordinary value of `f`.

```
In[2]:= f[2] + f[5]
Out[2]= f[2] + f[5]
```

If you ask for a numerical approximation, however, *Mathematica* uses the numerical values you have defined.

```
In[3]:= N[%]
Out[3]= 0.793244
```

You can define numerical values for both functions and symbols. The numerical values are used by all numerical *Mathematica* functions, including `NIntegrate`, `FindRoot` and so on.

N[*expr*]=*value*	define a numerical value to be used when default numerical precision is requested
N[*expr*,{*n*,Infinity}]=*value*	define a numerical value to be used when *n*-digit precision and any accuracy is requested

Defining numerical values that depend on numerical precision.

This defines a numerical value for the symbol `const`, using `4 n + 5` terms in the product for `n`-digit precision.

```
In[4]:= N[const, {n_, Infinity}] := Product[1 - 2^-i, {i, 2, 4 n + 5}]
```

Here is the value of `const`, computed to 30-digit precision using the value you specified.

```
In[5]:= N[const, 30]
Out[5]= 0.577576190173204842557799443858
```

Mathematica treats numerical values essentially like upvalues. When you define a numerical value for f, *Mathematica* effectively enters your definition as an upvalue for f with respect to the numerical evaluation operation `N`.

Modifying Built-in Functions

Mathematica allows you to define transformation rules for any expression. You can define such rules not only for functions that you add to *Mathematica*, but also for intrinsic functions that are already built into *Mathematica*. As a result, you can enhance, or modify, the features of built-in *Mathematica* functions.

This capability is powerful, but potentially dangerous. *Mathematica* will always follow the rules you give it. This means that if the rules you give are incorrect, then *Mathematica* will give you incorrect answers.

To avoid the possibility of changing built-in functions by mistake, *Mathematica* "protects" all built-in functions from redefinition. If you want to give a definition for a built-in function, you have to remove the protection first. After you give the definition, you should usually restore the protection, to prevent future mistakes.

`Unprotect[`*f*`]`	remove protection
`Protect[`*f*`]`	add protection

Protection for functions.

Built-in functions are usually "protected", so you cannot redefine them.

```
In[1]:= Log[7] = 2

        Set::write : Tag Log in Log[7] is Protected. >>

Out[1]= 2
```

This removes protection for `Log`.

```
In[2]:= Unprotect[Log]
Out[2]= {Log}
```

Now you can give your own definitions for `Log`. This particular definition is not mathematically correct, but *Mathematica* will still allow you to give it.

```
In[3]:= Log[7] = 2
Out[3]= 2
```

Mathematica will use your definitions whenever it can, whether they are mathematically correct or not.

```
In[4]:= Log[7] + Log[3]
Out[4]= 2 + Log[3]
```

This removes the incorrect definition for `Log`.

```
In[5]:= Log[7] =.
```

This restores the protection for `Log`.

```
In[6]:= Protect[Log]
Out[6]= {Log}
```

Definitions you give can override built-in features of *Mathematica*. In general, *Mathematica* tries to use your definitions before it uses built-in definitions.

The rules that are built into *Mathematica* are intended to be appropriate for the broadest range of calculations. In specific cases, however, you may not like what the built-in rules do. In such cases, you can give your own rules to override the ones that are built in.

There is a built-in rule for simplifying `Exp[Log[`*expr*`]]`.

```
In[7]:= Exp[Log[y]]
Out[7]= y
```

You can give your own rule for `Exp[Log[`*expr*`]]`, overriding the built-in rule.

```
In[8]:= (Unprotect[Exp]; Exp[Log[expr_]] := explog[expr]; Protect[Exp];)
```

Now your rule is used, rather than the built-in one.

```
In[9]:= Exp[Log[y]]
Out[9]= explog[y]
```

Manipulating Value Lists

`DownValues[`*f*`]`	give the list of downvalues of *f*
`UpValues[`*f*`]`	give the list of upvalues of *f*
`DownValues[`*f*`]=`*rules*	set the downvalues of *f*
`UpValues[`*f*`]=`*rules*	set the upvalues of *f*

Finding and setting values of symbols.

Mathematica effectively stores all definitions you give as lists of transformation rules. When a particular symbol is encountered, the lists of rules associated with it are tried.

Under most circumstances, you do not need direct access to the actual transformation rules associated with definitions you have given. Instead, you can simply use *lhs* = *rhs* and *lhs* =. to add and remove rules. In some cases, however, you may find it useful to have direct access to the actual rules.

Here is a definition for `f`.

```
In[1]:= f[x_] := x^2
```

This gives the explicit rule corresponding to the definition you made for `f`.

```
In[2]:= DownValues[f]
Out[2]= {HoldPattern[f[x_]] :> x^2}
```

Notice that the rules returned by `DownValues` and `UpValues` are set up so that neither their left- nor right-hand sides get evaluated. The left-hand sides are wrapped in `HoldPattern`, and the rules are delayed, so that the right-hand sides are not immediately evaluated.

As discussed in "Making Definitions for Functions", *Mathematica* tries to order definitions so that more specific ones appear before more general ones. In general, however, there is no unique way to make this ordering, and you may want to choose a different ordering from the one that *Mathematica* chooses by default. You can do this by reordering the list of rules obtained from `DownValues` or `UpValues`.

Here are some definitions for the object `g`.

```
In[3]:= g[x_ + y_] := gp[x, y]; g[x_ y_] := gm[x, y]
```

This shows the default ordering used for the definitions.

```
In[4]:= DownValues[g]
Out[4]= {HoldPattern[g[x_ + y_]] :→ gp[x, y], HoldPattern[g[x_ y_]] :→ gm[x, y]}
```

This reverses the order of the definitions for g.

```
In[5]:= DownValues[g] = Reverse[DownValues[g]]
Out[5]= {HoldPattern[g[x_ y_]] :→ gm[x, y], HoldPattern[g[x_ + y_]] :→ gp[x, y]}
```

Functions and Programs

Defining Functions

There are many functions that are built into *Mathematica*. This tutorial discusses how you can add your own simple functions to *Mathematica*.

As a first example, consider adding a function called `f` which squares its argument. The *Mathematica* command to define this function is `f[x_] := x^2`. The _ (referred to as "blank") on the left-hand side is very important; what it means will be discussed below. For now, just remember to put a _ on the left-hand side, but not on the right-hand side, of your definition.

This defines the function `f`. Notice the _ on the left-hand side.

```
In[1]:= f[x_] := x^2
```

`f` squares its argument.

```
In[2]:= f[a + 1]
```

Out[2]= $(1+a)^2$

The argument can be a number.

```
In[3]:= f[4]
Out[3]= 16
```

Or it can be a more complicated expression.

```
In[4]:= f[3 x + x^2]
```

Out[4]= $\left(3\,x+x^2\right)^2$

You can use `f` in a calculation.

```
In[5]:= Expand[f[(x + 1 + y)]]
```

Out[5]= $1+2\,x+x^2+2\,y+2\,x\,y+y^2$

This shows the definition you made for `f`.

```
In[6]:= ? f
```

```
Global`f
```

$f[x\_] := x^2$

`f[x_]:=x^2`	define the function `f`
`?f`	show the definition of `f`
`Clear[f]`	clear all definitions for `f`

Defining a function in *Mathematica*.

The names like `f` that you use for functions in *Mathematica* are just symbols. Because of this, you should make sure to avoid using names that begin with capital letters, to prevent confusion with built-in *Mathematica* functions. You should also make sure that you have not used the names for anything else earlier in your session.

Mathematica functions can have any number of arguments.

```
In[7]:= hump[x_, xmax_] := (x - xmax)^2 / xmax
```

You can use the `hump` function just as you would any of the built-in functions.

```
In[8]:= 2 + hump[x, 3.5]
```

$$\text{Out[8]=}\ 2 + 0.285714\,(-3.5 + x)^2$$

This gives a new definition for `hump`, which overwrites the previous one.

```
In[9]:= hump[x_, xmax_] := (x - xmax)^4
```

The new definition is displayed.

```
In[10]:= ? hump
```

```
Global`hump
```

$\text{hump}[x\_, \text{xmax}\_] := (x - \text{xmax})^4$

This clears all definitions for `hump`.

```
In[11]:= Clear[hump]
```

When you have finished with a particular function, it is always a good idea to clear definitions you have made for it. If you do not do this, then you will run into trouble if you try to use the same function for a different purpose later in your *Mathematica* session. You can clear all definitions you have made for a function or symbol f by using `Clear[f]`.

Functions as Procedures

In many kinds of calculations, you may find yourself typing the same input to *Mathematica* over and over again. You can save yourself a lot of typing by defining a *function* that contains your input commands.

This constructs a product of three terms, and expands out the result.

```
In[1]:=  Expand[Product[x + i, {i, 3}]]
```

```
Out[1]=  6 + 11 x + 6 x^2 + x^3
```

This does the same thing, but with four terms.

```
In[2]:=  Expand[Product[x + i, {i, 4}]]
```

```
Out[2]=  24 + 50 x + 35 x^2 + 10 x^3 + x^4
```

This defines a function `exprod` which constructs a product of n terms, then expands it out.

```
In[3]:=  exprod[n_] := Expand[Product[x + i, {i, 1, n}]]
```

Every time you use the function, it will execute the `Product` and `Expand` operations.

```
In[4]:=  exprod[5]
```

```
Out[4]=  120 + 274 x + 225 x^2 + 85 x^3 + 15 x^4 + x^5
```

The functions you define in *Mathematica* are essentially procedures that execute the commands you give. You can have several steps in your procedures, separated by semicolons.

The result you get from the whole function is simply the last expression in the procedure. Notice that you have to put parentheses around the procedure when you define it like this.

```
In[5]:=  cex[n_, i_] := (t = exprod[n]; Coefficient[t, x^i])
```

This "runs" the procedure.

```
In[6]:=  cex[5, 3]
```

```
Out[6]=  85
```

expr$_1$; *expr*$_2$; …	a sequence of expressions to evaluate
`Module[{`*a*`,`*b*`,`…`},`*proc*`]`	a procedure with local variables *a*, *b*, …

Constructing procedures.

When you write procedures in *Mathematica*, it is usually a good idea to make variables you use inside the procedures *local*, so that they do not interfere with things outside the procedures. You can do this by setting up your procedures as *modules*, in which you give a list of variables to be treated as local.

The function `cex` defined above is not a module, so the value of `t` "escapes", and exists even after the function returns.

```
In[7]:= t
Out[7]= 120 + 274 x + 225 x^2 + 85 x^3 + 15 x^4 + x^5
```

This function is defined as a module with local variable `u`.

```
In[8]:= ncex[n_, i_] := Module[{u}, u = exprod[n]; Coefficient[u, x^i]]
```

The function gives the same result as before.

```
In[9]:= ncex[5, 3]
Out[9]= 85
```

Now, however, the value of `u` does not escape from the function.

```
In[10]:= u
Out[10]= u
```

Manipulating Options

There are a number of functions built into *Mathematica* which, like `Plot`, have various options you can set. *Mathematica* provides some general mechanisms for handling such options.

If you do not give a specific setting for an option to a function like `Plot`, then *Mathematica* will automatically use a default value for the option. The function `Options[`*function*`,` *option*`]` allows you to find out the default value for a particular option. You can reset the default using `SetOptions[`*function*`,` *option* `->` *value*`]`. Note that if you do this, the default value you have given will stay until you explicitly change it.

`Options[`*function*`]`	give a list of the current default settings for all options
`Options[`*function*`,`*option*`]`	give the default setting for a particular option
`SetOptions[`*function*`,` *option*`->`*value*`,…]`	reset defaults

Manipulating default settings for options.

Here is the default setting for the `PlotRange` option of `Plot`.

```
In[1]:= Options[Plot, PlotRange]
Out[1]= {PlotRange → {Full, Automatic}}
```

This resets the default for the `PlotRange` option. The semicolon stops *Mathematica* from printing out the rather long list of options for `Plot`.

```
In[2]:= SetOptions[Plot, PlotRange -> All];
```

Until you explicitly reset it, the default for the `PlotRange` option will now be `All`.

```
In[3]:= Options[Plot, PlotRange]
Out[3]= {PlotRange → All}
```

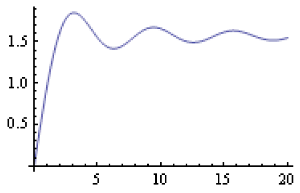
The graphics objects that you get from `Plot` or `Show` store information on the options they use. You can get this information by applying the `Options` function to these graphics objects.

`Options[`*plot*`]`	show all the options used for a particular plot
`Options[`*plot*`,`*option*`]`	show the setting for a specific option
`AbsoluteOptions[`*plot*`,`*option*`]`	show the absolute form used for a specific option, even if the setting for the option is `Automatic` or `All`

Getting information on options used in plots.

Here is a plot, with default settings for all options.

```
In[4]:= g = Plot[SinIntegral[x], {x, 0, 20}]
Out[4]=
```



The setting used for the `PlotRange` option was `All`.

```
In[5]:= Options[g, PlotRange]
Out[5]= {PlotRange → {All, All}}
```

`AbsoluteOptions` gives the *absolute* automatically chosen values used for `PlotRange`.

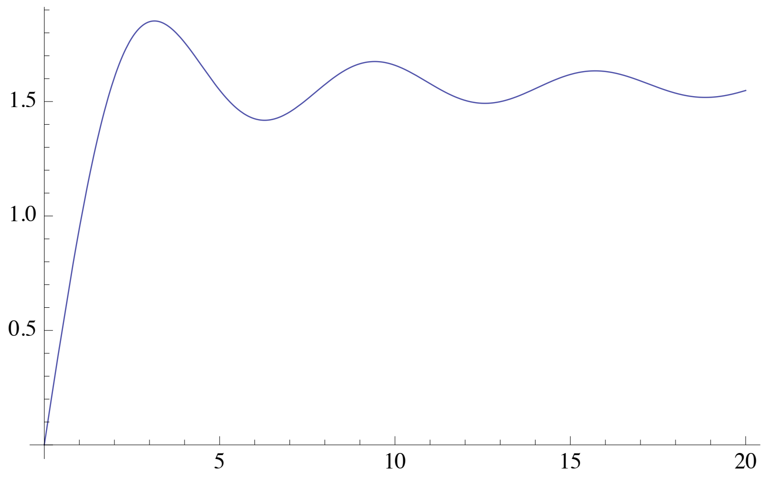
```
In[6]:= AbsoluteOptions[g, PlotRange]
Out[6]= {PlotRange → {{4.08163×10^-7, 20.}, {4.08163×10^-7, 1.85194}}}
```

While it is often convenient to use a variable to represent a graphic as in the above examples, the graphic itself can be evaluated directly. The typical ways to do this in the notebook interface are to copy and paste the graphic or to simply begin typing in the graphical output cell, at which point the output cell will be converted into a new input cell.

When a plot created with no explicit `ImageSize` is placed into an input cell, it will automatically shrink to more easily accommodate input.

The following input cell was created by copying and pasting the graphical output created in the previous example.

```
In[7]:= AbsoluteOptions[
```



```
, PlotRange]
Out[7]= {PlotRange → {{4.08163×10^-7, 20.}, {4.08163×10^-7, 1.85194}}}
```

Repetitive Operations

In using *Mathematica*, you sometimes need to repeat an operation many times. There are many ways to do this. Often the most natural is in fact to set up a structure such as a list with many elements, and then apply your operation to each of the elements.

Another approach is to use the *Mathematica* function `Do`, which works much like the iteration constructs in languages such as C and Fortran. `Do` uses the same *Mathematica* iterator notation as `Sum` and `Product`, described in "Sums and Products".

`Do`$[expr, \{i, i_{max}\}]$	evaluate $expr$ with i running from 1 to i_{max}
`Do`$[expr, \{i, i_{min}, i_{max}, di\}]$	evaluate $expr$ with i running from i_{min} to i_{max} in steps of di
`Print`$[expr]$	print $expr$
`Table`$[expr, \{i, i_{max}\}]$	make a list of the values of $expr$ with i running from 1 to i_{max}

Implementing repetitive operations.

This prints out the values of the first five factorials.

```
In[1]:= Do[Print[i!], {i, 5}]

           1

           2

           6

           24

           120
```

It is often more useful to have a list of results, which you can then manipulate further.

```
In[2]:= Table[i!, {i, 5}]
Out[2]= {1, 2, 6, 24, 120}
```

If you do not give an iteration variable, *Mathematica* simply repeats the operation you have specified, without changing anything.

```
In[3]:= r = 1; Do[r = 1 / (1 + r), {100}]; r
```

Out[3]= $\frac{573\,147\,844\,013\,817\,084\,101}{927\,372\,692\,193\,078\,999\,176}$

Transformation Rules for Functions

"Values for Symbols" discussed how you can use transformation rules of the form *x -> value* to replace symbols by values. The notion of transformation rules in *Mathematica* is, however, quite general. You can set up transformation rules not only for symbols, but for any *Mathematica* expression.

Applying the transformation rule `x -> 3` replaces `x` by `3`.

```
In[1]:=  1 + f[x] + f[y] /. x -> 3
Out[1]=  1 + f[3] + f[y]
```

You can also use a transformation rule for `f[x]`. This rule does not affect `f[y]`.

```
In[2]:=  1 + f[x] + f[y] /. f[x] -> p
Out[2]=  1 + p + f[y]
```

`f[t_]` is a *pattern* that stands for `f` with any argument.

```
In[3]:=  1 + f[x] + f[y] /. f[t_] -> t^2
```

Out[3]= $1 + x^2 + y^2$

Probably the most powerful aspect of transformation rules in *Mathematica* is that they can involve not only literal expressions, but also *patterns*. A pattern is an expression such as `f[t_]` which contains a blank (underscore). The blank can stand for any expression. Thus, a transformation rule for `f[t_]` specifies how the function `f` with *any* argument should be transformed. Notice that, in contrast, a transformation rule for `f[x]` without a blank, specifies only how the literal expression `f[x]` should be transformed, and does not, for example, say anything about the transformation of `f[y]`.

When you give a function definition such as `f[t_] := t^2`, all you are doing is telling *Mathematica* to automatically apply the transformation rule `f[t_] -> t^2` whenever possible.

You can set up transformation rules for expressions of any form.

```
In[4]:=  f[a b] + f[c d] /. f[x_ y_] -> f[x] + f[y]
Out[4]=  f[a] + f[b] + f[c] + f[d]
```

This uses a transformation rule for `x^p_`.

```
In[5]:=  1 + x^2 + x^4 /. x^p_ -> f[p]
Out[5]=  1 + f[2] + f[4]
```

"Patterns" and "Transformation Rules and Definitions" will explain in detail how to set up patterns and transformation rules for any kind of expression. Suffice it to say here that in *Mathematica* all expressions have a definite symbolic structure; transformation rules allow you to transform parts of that structure.

Functional Operations

Function Names as Expressions

In an expression like $f[x]$, the "function name" f is itself an expression, and you can treat it as you would any other expression.

You can replace names of functions using transformation rules.

```
In[1]:= f[x] + f[1 - x] /. f -> g
Out[1]= g[1 - x] + g[x]
```

Any assignments you have made are used on function names.

```
In[2]:= p1 = p2; p1[x, y]
Out[2]= p2[x, y]
```

This defines a function which takes a function name as an argument.

```
In[3]:= pf[f_, x_] := f[x] + f[1 - x]
```

This gives `Log` as the function name to use.

```
In[4]:= pf[Log, q]
Out[4]= Log[1 - q] + Log[q]
```

The ability to treat the names of functions just like other kinds of expressions is an important consequence of the symbolic nature of the *Mathematica* language. It makes possible the whole range of *functional operations*.

Ordinary *Mathematica* functions such as `Log` or `Integrate` typically operate on data such as numbers and algebraic expressions. *Mathematica* functions that represent functional operations, however, can operate not only on ordinary data, but also on functions themselves. Thus, for example, the functional operation `InverseFunction` takes a *Mathematica* function name as an argument, and represents the inverse of that function.

`InverseFunction` is a functional operation: it takes a *Mathematica* function as an argument, and returns another function which represents its inverse.

```
In[5]:= InverseFunction[ArcSin]
Out[5]= Sin
```

The result obtained from `InverseFunction` is a function which you can apply to data.

```
In[6]:= %[x]
Out[6]= Sin[x]
```

You can also use `InverseFunction` in a purely symbolic way.

```
In[7]:= InverseFunction[f][x]
```

Out[7]= $f^{(-1)}[x]$

There are many kinds of functional operations in *Mathematica*. Some represent mathematical operations; others represent various kinds of procedures and algorithms.

Unless you are familiar with advanced symbolic languages, you will probably not recognize most of the functional operations discussed. At first, the operations may seem difficult to unders-tand. But it is worth persisting. Functional operations provide one of the most conceptually and practically efficient ways to use *Mathematica*.

Applying Functions Repeatedly

Many programs you write will involve operations that need to be iterated several times. `Nest` and `NestList` are powerful constructs for doing this.

`Nest[`f`,`x`,`n`]`	apply the function f nested n times to x
`NestList[`f`,`x`,`n`]`	generate the list $\{x, f[x], f[f[x]], \ldots\}$, where f is nested up to n deep

Applying functions of one argument repeatedly.

`Nest[`f`, `x`, `n`]` takes the "name" f of a function, and applies the function n times to x.

```
In[1]:= Nest[f, x, 4]
Out[1]= f[f[f[f[x]]]]
```

This makes a list of each successive nesting.

```
In[2]:= NestList[f, x, 4]
Out[2]= {x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}
```

Here is a simple function.

```
In[3]:= recip[x_] := 1 / (1 + x)
```

You can iterate the function using `Nest`.

```
In[4]:= Nest[recip, x, 3]
```

Out[4]= $\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}$

`Nest` and `NestList` allow you to apply functions a fixed number of times. Often you may want to apply functions until the result no longer changes. You can do this using `FixedPoint` and `FixedPointList`.

`FixedPoint[`f`,`x`]`	apply the function f repeatedly until the result no longer changes
`FixedPointList[`f`,`x`]`	generate the list $\{x, f[x], f[f[x]], \ldots\}$, stopping when the elements no longer change

Applying functions until the result no longer changes.

Here is a function that takes one step in Newton's approximation to $\sqrt{3}$.

```
In[5]:= newton3[x_] := N[1 / 2 (x + 3 / x)]
```

Here are five successive iterates of the function, starting at $x = 1$.

```
In[6]:= NestList[newton3, 1.0, 5]
Out[6]= {1., 2., 1.75, 1.73214, 1.73205, 1.73205}
```

Using the function `FixedPoint`, you can automatically continue applying `newton3` until the result no longer changes.

```
In[7]:= FixedPoint[newton3, 1.0]
Out[7]= 1.73205
```

Here is the sequence of results.

```
In[8]:= FixedPointList[newton3, 1.0]
Out[8]= {1., 2., 1.75, 1.73214, 1.73205, 1.73205, 1.73205}
```

`NestWhile[`*f*`,`*x*`,`*test*`]`	apply the function *f* repeatedly until applying *test* to the result no longer yields `True`
`NestWhileList[`*f*`,`*x*`,`*test*`]`	generate the list $\{x, f[x], f[f[x]], \ldots\}$, stopping when applying *test* to the result no longer yields `True`
`NestWhile[`*f*`,`*x*`,`*test*`,`*m*`]`, `NestWhileList[`*f*`,`*x*`,`*test*`,`*m*`]`	supply the *m* most recent results as arguments for *test* at each step
`NestWhile[`*f*`,`*x*`,`*test*`,All]`, `NestWhileList[`*f*`,`*x*`,`*test*`,All]`	supply all results so far as arguments for *test*

Applying functions repeatedly until a test fails.

Here is a function which divides a number by 2.

```
In[9]:= divide2[n_] := n / 2
```

This repeatedly applies `divide2` until the result is no longer an even number.

```
In[10]:= NestWhileList[divide2, 123 456, EvenQ]
Out[10]= {123 456, 61 728, 30 864, 15 432, 7716, 3858, 1929}
```

This repeatedly applies `newton3`, stopping when two successive results are no longer considered unequal, just as in `FixedPointList`.

```
In[11]:= NestWhileList[newton3, 1.0, Unequal, 2]
Out[11]= {1., 2., 1.75, 1.73214, 1.73205, 1.73205, 1.73205}
```

This goes on until the first time a result that has been seen before reappears.

```
In[12]:= NestWhileList[Mod[5 #, 7] &, 1, Unequal, All]
Out[12]= {1, 5, 4, 6, 2, 3, 1}
```

Operations such as `Nest` take a function *f* of one argument, and apply it repeatedly. At each step, they use the result of the previous step as the new argument of *f*.

It is important to generalize this notion to functions of two arguments. You can again apply the function repeatedly, but now each result you get supplies only one of the new arguments you need. A convenient approach is to get the other argument at each step from the successive elements of a list.

`FoldList[`f`,`x`,{`a`,`b`,…}]`	create the list $\{x, f[x, a], f[f[x, a], b], \ldots\}$
`Fold[`f`,`x`,{`a`,`b`,…}]`	give the last element of the list produced by `FoldList[`f`, `x`, {`a`, `b`, …}]`

Ways to repeatedly apply functions of two arguments.

Here is an example of what `FoldList` does.

```
In[13]:= FoldList[f, x, {a, b, c}]
Out[13]= {x, f[x, a], f[f[x, a], b], f[f[f[x, a], b], c]}
```

`Fold` gives the last element of the list produced by `FoldList`.

```
In[14]:= Fold[f, x, {a, b, c}]
Out[14]= f[f[f[x, a], b], c]
```

This gives a list of cumulative sums.

```
In[15]:= FoldList[Plus, 0, {a, b, c}]
Out[15]= {0, a, a + b, a + b + c}
```

Using `Fold` and `FoldList` you can write many elegant and efficient programs in *Mathematica*. In some cases, you may find it helpful to think of `Fold` and `FoldList` as producing a simple nesting of a family of functions indexed by their second argument.

This defines a function `nextdigit`.

```
In[16]:= nextdigit[a_, b_] := 10 a + b
```

This is now like the built-in function `FromDigits`.

```
In[17]:= fromdigits[digits_] := Fold[nextdigit, 0, digits]
```

Here is an example of the function in action.

```
In[18]:= fromdigits[{1, 3, 7, 2, 9, 1}]
Out[18]= 137 291
```

Applying Functions to Lists and Other Expressions

In an expression like `f[{a, b, c}]` you are giving a list as the argument to a function. Often you need instead to apply a function directly to the elements of a list, rather than to the list as a whole. You can do this in *Mathematica* using `Apply`.

This makes each element of the list an argument of the function `f`.

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

This gives `Times[a, b, c]` which yields the product of the elements in the list.

```
In[2]:= Apply[Times, {a, b, c}]
Out[2]= a b c
```

Here is a definition of a function that works like the built-in function `GeometricMean`, written using `Apply`.

```
In[3]:= geom[list_] := Apply[Times, list]^(1/Length[list])
```

`Apply[`*f*`,{`*a*`,`*b*`,`…`}]`	apply *f* to a list, giving *f*[*a*, *b*, …]
`Apply[`*f*`,`*expr*`]` or *f*@@*expr*	apply *f* to the top level of an expression
`Apply[`*f*`,`*expr*`,{1}]` or *f*@@@*expr*	apply *f* at the first level in an expression
`Apply[`*f*`,`*expr*`,`*lev*`]`	apply *f* at the specified levels in an expression

Applying functions to lists and other expressions.

What `Apply` does in general is to replace the head of an expression with the function you specify. Here it replaces `Plus` by `List`.

```
In[4]:= Apply[List, a + b + c]
Out[4]= {a, b, c}
```

Here is a matrix.

```
In[5]:= m = {{a, b, c}, {b, c, d}}
Out[5]= {{a, b, c}, {b, c, d}}
```

Using `Apply` without an explicit level specification replaces the top-level list with `f`.

```
In[6]:= Apply[f, m]
Out[6]= f[{a, b, c}, {b, c, d}]
```

This applies `f` only to parts of `m` at level 1.

```
In[7]:= Apply[f, m, {1}]
Out[7]= {f[a, b, c], f[b, c, d]}
```

This applies `f` at levels 0 through 1.

```
In[8]:= Apply[f, m, {0, 1}]
Out[8]= f[f[a, b, c], f[b, c, d]]
```

Applying Functions to Parts of Expressions

If you have a list of elements, it is often important to be able to apply a function separately to each of the elements. You can do this in *Mathematica* using `Map`.

This applies `f` separately to each element in a list.

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

This defines a function which takes the first two elements from a list.

```
In[2]:= take2[list_] := Take[list, 2]
```

You can use `Map` to apply `take2` to each element of a list.

```
In[3]:= Map[take2, {{1, 3, 4}, {5, 6, 7}, {2, 1, 6, 6}}]
Out[3]= {{1, 3}, {5, 6}, {2, 1}}
```

`Map[`*f*`,{`*a*`,`*b*`,…}]`	apply *f* to each element in a list, giving `{`*f*`[`*a*`], `*f*`[`*b*`], …}`

Applying a function to each element in a list.

What `Map[`*f*`, `*expr*`]` effectively does is to “wrap” the function *f* around each element of the expression *expr*. You can use `Map` on any expression, not just a list.

This applies `f` to each element in the sum.

```
In[4]:= Map[f, a + b + c]
Out[4]= f[a] + f[b] + f[c]
```

This applies `Sqrt` to each argument of `g`.

```
In[5]:= Map[Sqrt, g[x^2, x^3]]
```

Out[5]= $g\left[\sqrt{x^2}, \sqrt{x^3}\right]$

`Map[`*f*`,` *expr*`]` applies *f* to the first level of parts in *expr*. You can use `MapAll[`*f*`,` *expr*`]` to apply *f* to *all* the parts of *expr*.

This defines a 2x2 matrix `m`.

```
In[6]:= m = {{a, b}, {c, d}}
Out[6]= {{a, b}, {c, d}}
```

`Map` applies `f` to the first level of `m`, in this case the rows of the matrix.

```
In[7]:= Map[f, m]
Out[7]= {f[{a, b}], f[{c, d}]}
```

`MapAll` applies `f` at *all* levels in `m`. If you look carefully at this expression, you will see an `f` wrapped around every part.

```
In[8]:= MapAll[f, m]
Out[8]= f[{f[{f[a], f[b]}], f[{f[c], f[d]}]}]
```

In general, you can use level specifications as described in "Levels in Expressions" to tell `Map` to which parts of an expression to apply your function.

This applies `f` only to the parts of `m` at level 2.

```
In[9]:= Map[f, m, {2}]
Out[9]= {{f[a], f[b]}, {f[c], f[d]}}
```

Setting the option `Heads -> True` wraps `f` around the head of each part, as well as its elements.

```
In[10]:= Map[f, m, Heads -> True]
Out[10]= f[List][f[{a, b}], f[{c, d}]]
```

`Map[`*f*`,`*expr*`]` or *f*`/@`*expr*	apply *f* to the first-level parts of *expr*
`MapAll[`*f*`,`*expr*`]` or *f*`//@`*expr*	apply *f* to all parts of *expr*
`Map[`*f*`,`*expr*`,`*lev*`]`	apply *f* to each part of *expr* at levels specified by *lev*

Ways to apply a function to different parts of expressions.

Level specifications allow you to tell `Map` to which levels of parts in an expression you want a function applied. With `MapAt`, however, you can instead give an explicit list of parts where you want a function applied. You specify each part by giving its indices, as discussed in "Parts of Expressions".

Here is a 2x3 matrix.

```
In[11]:= mm = {{a, b, c}, {b, c, d}}
Out[11]= {{a, b, c}, {b, c, d}}
```

This applies `f` to parts `{1, 2}` and `{2, 3}`.

```
In[12]:= MapAt[f, mm, {{1, 2}, {2, 3}}]
Out[12]= {{a, f[b], c}, {b, c, f[d]}}
```

This gives a list of the positions at which `b` occurs in `mm`.

```
In[13]:= Position[mm, b]
Out[13]= {{1, 2}, {2, 1}}
```

You can feed the list of positions you get from `Position` directly into `MapAt`.

```
In[14]:= MapAt[f, mm, %]
Out[14]= {{a, f[b], c}, {f[b], c, d}}
```

To avoid ambiguity, you must put each part specification in a list, even when it involves only one index.

```
In[15]:= MapAt[f, {a, b, c, d}, {{2}, {3}}]
Out[15]= {a, f[b], f[c], d}
```

`MapAt[`*f*`,`*expr*`,{`$part_1$`,`$part_2$`,`…`}]`	apply *f* to specified parts of *expr*

Applying a function to specific parts of an expression.

Here is an expression.

```
In[16]:= t = 1 + (3 + x) ^2 / x
```

$$\text{Out[16]=}\quad 1+\frac{(3+x)^2}{x}$$

This is the full form of `t`.

```
In[17]:= FullForm[t]

Out[17]//FullForm= Plus[1, Times[Power[x, -1], Power[Plus[3, x], 2]]]
```

You can use `MapAt` on any expression. Remember that parts are numbered on the basis of the full forms of expressions.

```
In[18]:= MapAt[f, t, {{2, 1, 1}, {2, 2}}]
```

$$\text{Out[18]=}\quad 1+\frac{f\left[(3+x)^2\right]}{f[x]}$$

`MapIndexed[`*f*`,`*expr*`]`	apply *f* to the elements of an expression, giving the part specification of each element as a second argument to *f*
`MapIndexed[`*f*`,`*expr*`,`*lev*`]`	apply *f* to parts at specified levels, giving the list of indices for each part as a second argument to *f*

Applying a function to parts and their indices.

This applies `f` to each element in a list, giving the index of the element as a second argument to `f`.

```
In[19]:= MapIndexed[f, {a, b, c}]

Out[19]= {f[a, {1}], f[b, {2}], f[c, {3}]}
```

This applies `f` to both levels in a matrix.

```
In[20]:= MapIndexed[f, {{a, b}, {c, d}}, 2]

Out[20]= {f[{f[a, {1, 1}], f[b, {1, 2}]}, {1}], f[{f[c, {2, 1}], f[d, {2, 2}]}, {2}]}
```

`Map` allows you to apply a function of one argument to parts of an expression. Sometimes, however, you may instead want to apply a function of several arguments to corresponding parts of several different expressions. You can do this using `MapThread`.

`MapThread[`f`,{`$expr_1$`,`$expr_2$`,`…`}]`	apply f to corresponding elements in each of the $expr_i$
`MapThread[`f`,{`$expr_1$`,`$expr_2$`,`…`},`lev`]`	apply f to parts of the $expr_i$ at the specified level

Applying a function to several expressions at once.

This applies `f` to corresponding pairs of list elements.

```
In[21]:= MapThread[f, {{a, b, c}, {ap, bp, cp}}]
Out[21]= {f[a, ap], f[b, bp], f[c, cp]}
```

`MapThread` works with any number of expressions, so long as they have the same structure.

```
In[22]:= MapThread[f, {{a, b}, {ap, bp}, {app, bpp}}]
Out[22]= {f[a, ap, app], f[b, bp, bpp]}
```

Functions like `Map` allow you to create expressions with parts modified. Sometimes you simply want to go through an expression, and apply a particular function to some parts of it, without building a new expression. A typical case is when the function you apply has certain "side effects", such as making assignments, or generating output.

`Scan[`f`,`$expr$`]`	evaluate f applied to each element of $expr$ in turn
`Scan[`f`,`$expr$`,`lev`]`	evaluate f applied to parts of $expr$ on levels specified by lev

Evaluating functions on parts of expressions.

`Map` constructs a new list in which `f` has been applied to each element of the list.

```
In[23]:= Map[f, {a, b, c}]
Out[23]= {f[a], f[b], f[c]}
```

`Scan` evaluates the result of applying a function to each element, but does not construct a new expression.

```
In[24]:= Scan[Print, {a, b, c}]

            a

            b

            c
```

Scan visits the parts of an expression in a depth-first walk, with the leaves visited first.

```
In[25]:= Scan[Print, 1 + x^2, Infinity]

         1

         x

         2

         x^2
```

Pure Functions

Function[*x*,*body*]	a pure function in which *x* is replaced by any argument you provide
Function[{x_1,x_2,…},*body*]	a pure function that takes several arguments
body&	a pure function in which arguments are specified as # or #1, #2, #3, etc.

Pure functions.

When you use functional operations such as Nest and Map, you always have to specify a function to apply. In all the examples above, we have used the "name" of a function to specify the function. Pure functions allow you to give functions which can be applied to arguments, without having to define explicit names for the functions.

This defines a function h.

```
In[1]:= h[x_] := f[x] + g[x]
```

Having defined h, you can now use its name in Map.

```
In[2]:= Map[h, {a, b, c}]
Out[2]= {f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

Here is a way to get the same result using a pure function.

```
In[3]:= Map[f[#] + g[#] &, {a, b, c}]
Out[3]= {f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

There are several equivalent ways to write pure functions in *Mathematica*. The idea in all cases is to construct an object which, when supplied with appropriate arguments, computes a particular function. Thus, for example, if *fun* is a pure function, then *fun*[*a*] evaluates the function with argument *a*.

Here is a pure function which represents the operation of squaring.

```
In[4]:=  Function[x, x^2]
Out[4]=  Function[x, x²]
```

Supplying the argument `n` to the pure function yields the square of `n`.

```
In[5]:=  %[n]
Out[5]=  n²
```

You can use a pure function wherever you would usually give the name of a function.

You can use a pure function in `Map`.

```
In[6]:=  Map[Function[x, x^2], a + b + c]
Out[6]=  a² + b² + c²
```

Or in `Nest`.

```
In[7]:=  Nest[Function[q, 1 / (1 + q)], x, 3]
```

Out[7]= $\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}$

This sets up a pure function with two arguments and then applies the function to the arguments `a` and `b`.

```
In[8]:=  Function[{x, y}, x^2 + y^3][a, b]
Out[8]=  a² + b³
```

If you are going to use a particular function repeatedly, then you can define the function using *f*[*x*_] := *body*, and refer to the function by its name *f*. On the other hand, if you only intend to use a function once, you will probably find it better to give the function in pure function form, without ever naming it.

If you are familiar with formal logic or the LISP programming language, you will recognize *Mathematica* pure functions as being like λ expressions or anonymous functions. Pure functions are also close to the pure mathematical notion of operators.

`#`	the first variable in a pure function
`#` n	the nth variable in a pure function
`##`	the sequence of all variables in a pure function
`##` n	the sequence of variables starting with the nth one

Short forms for pure functions.

Just as the name of a function is irrelevant if you do not intend to refer to the function again, so also the names of arguments in a pure function are irrelevant. *Mathematica* allows you to avoid using explicit names for the arguments of pure functions, and instead to specify the arguments by giving "slot numbers" `#`n. In a *Mathematica* pure function, `#`n stands for the nth argument you supply. `#` stands for the first argument.

`#^2 &` is a short form for a pure function that squares its argument.

```
In[9]:= Map[#^2 &, a + b + c]
```

Out[9]= $a^2 + b^2 + c^2$

This applies a function that takes the first two elements from each list. By using a pure function, you avoid having to define the function separately.

```
In[10]:= Map[Take[#, 2] &, {{2, 1, 7}, {4, 1, 5}, {3, 1, 2}}]

Out[10]= {{2, 1}, {4, 1}, {3, 1}}
```

Using short forms for pure functions, you can simplify the definition of `fromdigits` given in "Applying Functions Repeatedly".

```
In[11]:= fromdigits[digits_] := Fold[(10 #1 + #2) &, 0, digits]
```

When you use short forms for pure functions, it is very important that you do not forget the ampersand. If you leave the ampersand out, *Mathematica* will not know that the expression you give is to be used as a pure function.

When you use the ampersand notation for pure functions, you must be careful about the grouping of pieces in your input. As shown in "Operator Input Forms" the ampersand notation has fairly low precedence, which means that you can type expressions like `#1 + #2 &` without parentheses. On the other hand, if you want, for example, to set an option to be a pure function, you need to use parentheses, as in *option* `->` `(`*fun* `&)`.

Pure functions in *Mathematica* can take any number of arguments. You can use `##` to stand for all the arguments that are given, and `##`n to stand for the nth and subsequent arguments.

stands for all arguments.

```
In[12]:= f[##, ##] &[x, y]
Out[12]= f[x, y, x, y]
```

##2 stands for all arguments except the first one.

```
In[13]:= Apply[f[##2, #1] &, {{a, b, c}, {ap, bp}}, {1}]
Out[13]= {f[b, c, a], f[bp, ap]}
```

Building Lists from Functions

`Array[`f`,`n`]`	generate a length n list of the form `{`f`[1], `f`[2], …}`
`Array[`f`,{`n_1`,`n_2`,…}]`	generate an $n_1\times n_2\times\ldots$ nested list, each of whose entries consists of f applied to its indices
`NestList[`f`,`x`,`n`]`	generate a list of the form `{`x`, `f`[`x`], `f`[`f`[`x`]], …}`, where f is nested up to n deep
`FoldList[`f`,`x`,{`a`,`b`,…}]`	generate a list of the form `{`x`, `f`[`x`, `a`], `f`[`f`[`x`, `a`], `b`], …}`
`ComposeList[{`f_1`,`f_2`,…},`x`]`	generate a list of the form `{`x`, `f_1`[`x`], `f_2`[`f_1`[`x`]], …}`

Making lists from functions.

This makes a list of 5 elements, each of the form p[i].

```
In[1]:= Array[p, 5]
Out[1]= {p[1], p[2], p[3], p[4], p[5]}
```

Here is another way to produce the same list.

```
In[2]:= Table[p[i], {i, 5}]
Out[2]= {p[1], p[2], p[3], p[4], p[5]}
```

This produces a list whose elements are $i+i^2$.

```
In[3]:= Array[# + #^2 &, 5]
Out[3]= {2, 6, 12, 20, 30}
```

This generates a 2×3 matrix whose entries are m[*i*, *j*].

```
In[4]:= Array[m, {2, 3}]
Out[4]= {{m[1, 1], m[1, 2], m[1, 3]}, {m[2, 1], m[2, 2], m[2, 3]}}
```

This generates a 3×3 matrix whose elements are the squares of the sums of their indices.

```
In[5]:= Array[Plus[##]^2 &, {3, 3}]
Out[5]= {{4, 9, 16}, {9, 16, 25}, {16, 25, 36}}
```

`NestList` and `FoldList` were discussed in "Applying Functions Repeatedly". Particularly by using them with pure functions, you can construct some very elegant and efficient *Mathematica* programs.

This gives a list of results obtained by successively differentiating x^n with respect to x.

```
In[6]:= NestList[D[#, x] &, x^n, 3]
```

Out[6]= $\{x^n, n\,x^{-1+n}, (-1+n)\,n\,x^{-2+n}, (-2+n)\,(-1+n)\,n\,x^{-3+n}\}$

Selecting Parts of Expressions with Functions

"Manipulating Elements of Lists" shows how you can pick out elements of lists based on their *positions*. Often, however, you will need to select elements based not on *where* they are, but rather on *what* they are.

`Select[`*list*`,` *f*`]` selects elements of *list* using the function *f* as a criterion. `Select` applies *f* to each element of *list* in turn, and keeps only those for which the result is `True`.

This selects the elements of the list for which the pure function yields `True`, i.e., those numerically greater than 4.

```
In[1]:= Select[{2, 15, 1, a, 16, 17}, # > 4 &]
Out[1]= {15, 16, 17}
```

You can use `Select` to pick out pieces of any expression, not just elements of a list.

This gives a sum of terms involving `x`, `y` and `z`.

```
In[2]:= t = Expand[(x + y + z)^2]
```

Out[2]= $x^2 + 2\,x\,y + y^2 + 2\,x\,z + 2\,y\,z + z^2$

You can use Select to pick out only those terms in the sum that do not involve the symbol x.

```
In[3]:= Select[t, FreeQ[#, x] &]

Out[3]= y^2 + 2 y z + z^2
```

Select[*expr*, *f*]	select the elements in *expr* for which the function *f* gives True
Select[*expr*, *f*, *n*]	select the first *n* elements in *expr* for which the function *f* gives True

Selecting pieces of expressions.

"Putting Constraints on Patterns" discusses some "predicates" that are often used as criteria in Select.

This gives the first element which satisfies the criterion you specify.

```
In[4]:= Select[{-1, 3, 10, 12, 14}, # > 3 &, 1]

Out[4]= {10}
```

Expressions with Heads That Are Not Symbols

In most cases, you want the head *f* of a *Mathematica* expression like *f*[*x*] to be a single symbol. There are, however, some important applications of heads that are not symbols.

This expression has f[3] as a head. You can use heads like this to represent "indexed functions".

```
In[1]:= f[3][x, y]

Out[1]= f[3][x, y]
```

You can use any expression as a head. Remember to put in the necessary parentheses.

```
In[2]:= (a + b)[x]

Out[2]= (a + b)[x]
```

One case where we have already encountered the use of complicated expressions as heads is in working with pure functions in "Pure Functions". By giving Function[*vars*, *body*] as the head of an expression, you specify a function of the arguments to be evaluated.

With the head `Function[x, x^2]`, the value of the expression is the square of the argument.

```
In[3]:= Function[x, x^2][a + b]

Out[3]= (a + b)^2
```

There are several constructs in *Mathematica* which work much like pure functions, but which represent specific kinds of functions, typically numerical ones. In all cases, the basic mechanism involves giving a head which contains complete information about the function you want to use.

`Function[`*vars*`,`*body*`][`*args*`]`	pure function
`InterpolatingFunction[`*data*`][`*args*`]`	approximate numerical function (generated by `Interpolation` and `NDSolve`)
`CompiledFunction[`*data*`][`*args*`]`	compiled numerical function (generated by `Compile`)
`LinearSolveFunction[`*data*`][`*vec*`]`	matrix solution function (generated by `LinearSolve`)

Some expressions which have heads that are not symbols.

`NDSolve` returns a list of rules that give `y` as an `InterpolatingFunction` object.

```
In[4]:= NDSolve[{y''[x] == y[x], y[0] == y'[0] == 1}, y, {x, 0, 5}]

Out[4]= {{y → InterpolatingFunction[{{0., 5.}}, <>]}}
```

Here is the `InterpolatingFunction` object.

```
In[5]:= y /. First[%]

Out[5]= InterpolatingFunction[{{0., 5.}}, <>]
```

You can use the `InterpolatingFunction` object as a head to get numerical approximations to values of the function `y`.

```
In[6]:= %[3.8]

Out[6]= 44.7012
```

Another important use of more complicated expressions as heads is in implementing *functionals* and *functional operators* in mathematics.

As one example, consider the operation of differentiation. As discussed in "The Representation of Derivatives", an expression like `f'` represents a *derivative function*, obtained from `f` by applying a functional operator to it. In *Mathematica*, `f'` is represented as `Derivative[1][f]`: the “functional operator” `Derivative[1]` is applied to `f` to give another function, represented as `f'`.

This expression has a head which represents the application of the “functional operator” `Derivative[1]` to the “function” `f`.

```
In[7]:= f'[x] // FullForm
Out[7]//FullForm= Derivative[1][f][x]
```

You can replace the head `f'` with another head, such as `fp`. This effectively takes `fp` to be a “derivative function” obtained from `f`.

```
In[8]:= % /. f' -> fp
Out[8]= fp[x]
```

Working with Operators

You can think of an expression like $f[x]$ as being formed by applying an operator f to the expression x. You can think of an expression like $f[g[x]]$ as the result of *composing* the operators f and g, and applying the result to x.

`Composition[`f`,`g`,`$\ldots$`]`	the composition of functions f, g, $\ldots$
`InverseFunction[`f`]`	the inverse of a function f
`Identity`	the identity function

Some functional operations.

This represents the composition of the functions f, g and h.

```
In[1]:= Composition[f, g, h]
Out[1]= Composition[f, g, h]
```

You can manipulate compositions of functions symbolically.

```
In[2]:= InverseFunction[Composition[%, q]]
```

Out[2]= `Composition`$\left[q^{(-1)}, h^{(-1)}, g^{(-1)}, f^{(-1)}\right]$

The composition is evaluated explicitly when you supply a specific argument.

```
In[3]:= %[x]
Out[3]= q^(-1)[h^(-1)[g^(-1)[f^(-1)[x]]]]
```

You can get the sum of two expressions in *Mathematica* just by typing $x+y$. Sometimes it is also worthwhile to consider performing operations like addition on *operators*.

You can think of this as containing a sum of two operators f and g.

```
In[4]:= (f + g)[x]
Out[4]= (f + g)[x]
```

Using `Through`, you can convert the expression to a more explicit form.

```
In[5]:= Through[%, Plus]
Out[5]= f[x] + g[x]
```

This corresponds to the mathematical operator $1+\frac{\partial}{\partial x}$.

```
In[6]:= Identity + (D[#, x] &)
Out[6]= Identity + (∂x #1 &)
```

Mathematica does not automatically apply the separate pieces of the operator to an expression.

```
In[7]:= %[x^2]
Out[7]= (Identity + (∂x #1 &))[x^2]
```

You can use `Through` to apply the operator.

```
In[8]:= Through[%, Plus]
Out[8]= 2 x + x^2
```

`Identity[`*expr*`]`	the identity function
`Through[`p`[`f_1`,`f_2`][`x`],`q`]`	give $p[f_1[x], f_2[x]]$ if p is the same as q
`Operate[`p`,`f`[`x`]]`	give $p[f][x]$
`Operate[`p`,`f`[`x`],`n`]`	apply p at level n in f
`MapAll[`p`,`*expr*`,Heads->True]`	apply p to all parts of *expr*, including heads

Operations for working with operators.

This has a complicated expression as a head.

```
In[9]:= t = ((1 + a) (1 + b))[x]
Out[9]= ((1 + a) (1 + b))[x]
```

Functions like `Expand` do not automatically go inside heads of expressions.

```
In[10]:= Expand[%]
Out[10]= ((1 + a) (1 + b))[x]
```

With the `Heads` option set to `True`, `MapAll` goes inside heads.

```
In[11]:= MapAll[Expand, t, Heads -> True]
Out[11]= (1 + a + b + a b)[x]
```

The replacement operator `/.` does go inside heads of expressions.

```
In[12]:= t /. a -> 1
Out[12]= (2 (1 + b))[x]
```

You can use `Operate` to apply a function specifically to the head of an expression.

```
In[13]:= Operate[p, t]
Out[13]= p[(1 + a) (1 + b)][x]
```

Structural Operations

Mathematica contains some powerful primitives for making structural changes to expressions. You can use these primitives both to implement mathematical properties such as associativity and distributivity, and to provide the basis for some succinct and efficient programs.

Here we describe various operations that you can explicitly perform on expressions. "Attributes" describes how some of these operations can be performed automatically on all expressions with a particular head by assigning appropriate attributes to that head.

You can use the *Mathematica* function `Sort[`*expr*`]` to sort elements not only of lists, but of expressions with any head. In this way, you can implement the mathematical properties of commutativity or symmetry for arbitrary functions.

You can use `Sort` to put the arguments of any function into a standard order.

```
In[1]:=  Sort[f[c, a, b]]
Out[1]=  f[a, b, c]
```

`Sort[`*expr*`]`	sort the elements of a list or other expression into a standard order
`Sort[`*expr*`,`*pred*`]`	sort using the function *pred* to determine whether pairs are in order
`Ordering[`*expr*`]`	give the ordering of elements when sorted
`Ordering[`*expr*`,`*n*`]`	give the ordering of the first *n* elements when sorted
`Ordering[`*expr*`,`*n*`,`*pred*`]`	use the function *pred* to determine whether pairs are in order
`OrderedQ[`*expr*`]`	give `True` if the elements of *expr* are in standard order, and `False` otherwise
`Order[`$expr_1$`,`$expr_2$`]`	give 1 if $expr_1$ comes before $expr_2$ in standard order, and −1 if it comes after

Sorting into order.

The second argument to `Sort` is a function used to determine whether pairs are in order. This sorts numbers into descending order.

```
In[2]:=  Sort[{5, 1, 8, 2}, (#2 < #1) &]
Out[2]=  {8, 5, 2, 1}
```

This sorting criterion puts elements that do not depend on `x` before those that do.

```
In[3]:=  Sort[{x^2, y, x + y, y - 2}, FreeQ[#1, x] &]
```

Out[3]= $\{y, -2 + y, x + y, x^2\}$

`Flatten[`*expr*`]`	flatten out all nested functions with the same head as *expr*
`Flatten[`*expr*`,`*n*`]`	flatten at most *n* levels of nesting
`Flatten[`*expr*`,`*n*`,`*h*`]`	flatten functions with head *h*
`FlattenAt[`*expr*`,`*i*`]`	flatten only the i^{th} element of *expr*

Flattening out expressions.

`Flatten` removes nested occurrences of a function.

```
In[4]:= Flatten[f[a, f[b, c], f[f[d]]]]
Out[4]= f[a, b, c, d]
```

You can use `Flatten` to "splice" sequences of elements into lists or other expressions.

```
In[5]:= Flatten[{a, f[b, c], f[a, b, d]}, 1, f]
Out[5]= {a, b, c, a, b, d}
```

You can use `Flatten` to implement the mathematical property of associativity. The function `Distribute` allows you to implement properties such as distributivity and linearity.

`Distribute[`*f*`[`*a*+*b*+…,…`]]`	distribute *f* over sums to give *f*[*a*, …] + *f*[*b*, …] + …
`Distribute[`*f*`[`*args*`],`*g*`]`	distribute *f* over any arguments which have head *g*
`Distribute[`*expr*`,`*g*`,`*f*`]`	distribute only when the head is *f*
`Distribute[`*expr*`,`*g*`,`*f*`,`*gp*`,`*fp*`]`	distribute *f* over *g*, replacing them with *fp* and *gp*, respectively

Applying distributive laws.

This "distributes" `f` over `a + b`.

```
In[6]:= Distribute[f[a + b]]
Out[6]= f[a] + f[b]
```

Here is a more complicated example.

```
In[7]:= Distribute[f[a + b, c + d]]
Out[7]= f[a, c] + f[a, d] + f[b, c] + f[b, d]
```

In general, if *f* is distributive over `Plus`, then an expression like *f*[*a* + *b*] can be "expanded" to give *f*[*a*] + *f*[*b*]. The function `Expand` does this kind of expansion for standard algebraic operators such as `Times`. `Distribute` allows you to perform the same kind of expansion for arbitrary operators.

`Expand` uses the distributivity of `Times` over `Plus` to perform algebraic expansions.

```
In[8]:= Expand[(a + b) (c + d)]
Out[8]= a c + b c + a d + b d
```

This applies distributivity over lists, rather than sums. The result contains all possible pairs of arguments.

```
In[9]:= Distribute[f[{a, b}, {c, d}], List]
Out[9]= {f[a, c], f[a, d], f[b, c], f[b, d]}
```

This distributes over lists, but does so only if the head of the whole expression is `f`.

```
In[10]:= Distribute[f[{a, b}, {c, d}], List, f]
Out[10]= {f[a, c], f[a, d], f[b, c], f[b, d]}
```

This distributes over lists, making sure that the head of the whole expression is `f`. In the result, it uses `gp` in place of `List`, and `fp` in place of `f`.

```
In[11]:= Distribute[f[{a, b}, {c, d}], List, f, gp, fp]
Out[11]= gp[fp[a, c], fp[a, d], fp[b, c], fp[b, d]]
```

Related to `Distribute` is the function `Thread`. What `Thread` effectively does is to apply a function in parallel to all the elements of a list or other expression.

`Thread`$[f[\{a_1,a_2\},\{b_1,b_2\}]]$	thread f over lists to give $\{f[a_1, b_1], f[a_2, b_2]\}$
`Thread`$[f[args],g]$	thread f over objects with head g in $args$

Functions for threading expressions.

Here is a function whose arguments are lists.

```
In[12]:= f[{a1, a2}, {b1, b2}]
Out[12]= f[{a1, a2}, {b1, b2}]
```

`Thread` applies the function "in parallel" to each element of the lists.

```
In[13]:= Thread[%]
Out[13]= {f[a1, b1], f[a2, b2]}
```

Arguments that are not lists get repeated.

```
In[14]:= Thread[f[{a1, a2}, {b1, b2}, c, d]]
Out[14]= {f[a1, b1, c, d], f[a2, b2, c, d]}
```

As mentioned in "Collecting Objects Together", and discussed in more detail in "Attributes", many built-in *Mathematica* functions have the property of being “listable”, so that they are automatically threaded over any lists that appear as arguments.

Built-in mathematical functions such as `Log` are listable, so that they are automatically threaded over lists.

```
In[15]:= Log[{a, b, c}]
Out[15]= {Log[a], Log[b], Log[c]}
```

`Log` is, however, not automatically threaded over equations.

```
In[16]:= Log[x == y]
Out[16]= Log[x == y]
```

You can use `Thread` to get functions applied to both sides of an equation.

```
In[17]:= Thread[%, Equal]
Out[17]= Log[x] == Log[y]
```

`Outer[`f`,`$list_1$`,`$list_2$`]`	generalized outer product
`Inner[`f`,`$list_1$`,`$list_2$`,`g`]`	generalized inner product

Generalized outer and inner products.

`Outer[`f, $list_1$, $list_2$`]` takes all possible combinations of elements from $list_1$ and $list_2$, and combines them with f. `Outer` can be viewed as a generalization of a Cartesian product for tensors, as discussed in "Tensors".

`Outer` forms all possible combinations of elements, and applies `f` to them.

```
In[18]:= Outer[f, {a, b}, {1, 2, 3}]
Out[18]= {{f[a, 1], f[a, 2], f[a, 3]}, {f[b, 1], f[b, 2], f[b, 3]}}
```

Here `Outer` produces a lower-triangular Boolean matrix.

```
In[19]:= Outer[Greater, {1, 2, 3}, {1, 2, 3}]
Out[19]= {{False, False, False}, {True, False, False}, {True, True, False}}
```

You can use `Outer` on any sequence of expressions with the same head.

```
In[20]:= Outer[g, f[a, b], f[c, d]]
Out[20]= f[f[g[a, c], g[a, d]], f[g[b, c], g[b, d]]]
```

`Outer`, like `Distribute`, constructs all possible combinations of elements. On the other hand, `Inner`, like `Thread`, constructs only combinations of elements that have corresponding positions in the expressions it acts on.

Here is a structure built by `Inner`.

```
In[21]:= Inner[f, {a, b}, {c, d}, g]
Out[21]= g[f[a, c], f[b, d]]
```

`Inner` is a generalization of `Dot`.

```
In[22]:= Inner[Times, {a, b}, {c, d}, Plus]
Out[22]= a c + b d
```

Sequences

The function `Flatten` allows you to explicitly flatten out all sublists.

```
In[1]:= Flatten[{a, {b, c}, {d, e}}]
Out[1]= {a, b, c, d, e}
```

`FlattenAt` lets you specify at what positions you want sublists flattened.

```
In[2]:= FlattenAt[{a, {b, c}, {d, e}}, 2]
Out[2]= {a, b, c, {d, e}}
```

`Sequence` objects automatically get spliced in, and do not require any explicit flattening.

```
In[3]:= {a, Sequence[b, c], Sequence[d, e]}
Out[3]= {a, b, c, d, e}
```

`Sequence[`e_1`,`e_2`,…]`	a sequence of arguments that will automatically be spliced into any function

Representing sequences of arguments in functions.

`Sequence` works in any function.

```
In[4]:= f[Sequence[a, b], c]
Out[4]= f[a, b, c]
```

This includes functions with special input forms.

```
In[5]:= a == Sequence[b, c]
Out[5]= a == b == c
```

Here is a common way that `Sequence` is used.

```
In[6]:= {a, b, f[x, y], g[w], f[z, y]} /. f -> Sequence
Out[6]= {a, b, x, y, g[w], z, y}
```

Modularity and the Naming of Things

Modules and Local Variables

Mathematica normally assumes that all your variables are *global*. This means that every time you use a name like `x`, *Mathematica* normally assumes that you are referring to the *same* object.

Particularly when you write programs, however, you may not want all your variables to be global. You may, for example, want to use the name `x` to refer to two quite different variables in two different programs. In this case, you need the `x` in each program to be treated as a *local* variable.

You can set up local variables in *Mathematica* using *modules*. Within each module, you can give a list of variables which are to be treated as local to the module.

`Module[{`*x*`,`*y*`,`…`},`*body*`]`	a module with local variables *x*, *y*, …

Creating modules in *Mathematica*.

This defines the global variable `t` to have value 17.

```
In[1]:= t = 17
Out[1]= 17
```

The `t` inside the module is local, so it can be treated independently of the global `t`.

```
In[2]:= Module[{t}, t = 8; Print[t]]

        8
```

The global `t` still has value 17.

```
In[3]:= t
Out[3]= 17
```

The most common way that modules are used is to set up temporary or intermediate variables inside functions you define. It is important to make sure that such variables are kept local. If they are not, then you will run into trouble whenever their names happen to coincide with the names of other variables.

The intermediate variable `t` is specified to be local to the module.

```
In[4]:= f[v_] := Module[{t}, t = (1 + v)^2; t = Expand[t]]
```

This runs the function `f`.

```
In[5]:= f[a + b]
```

Out[5]= $1 + 2 a + a^2 + 2 b + 2 a b + b^2$

The global `t` still has value 17.

```
In[6]:= t

Out[6]= 17
```

You can treat local variables in modules just like other symbols. Thus, for example, you can use them as names for local functions, you can assign attributes to them, and so on.

This sets up a module which defines a local function `f`.

```
In[7]:= gfac10[k_] := Module[{f, n}, f[1] = 1; f[n_] := k + n f[n - 1]; f[10]]
```

In this case, the local function `f` is just an ordinary factorial.

```
In[8]:= gfac10[0]

Out[8]= 3 628 800
```

In this case, `f` is set up as a generalized factorial.

```
In[9]:= gfac10[2]

Out[9]= 8 841 802
```

When you set up a local variable in a module, *Mathematica* initially assigns no value to the variable. This means that you can use the variable in a purely symbolic way, even if there was a global value defined for the variable outside the module.

This uses the global value of `t` defined above, and so yields a number.

```
In[10]:= Expand[(1 + t)^3]

Out[10]= 5832
```

Here `Length` simply receives a number as its argument.

```
In[11]:= Length[Expand[(1 + t)^3]]

Out[11]= 0
```

The local variable `t` has no value, so it acts as a symbol, and `Expand` produces the anticipated algebraic result.

```
In[12]:= Module[{t}, Length[Expand[(1 + t)^3]]]
Out[12]= 4
```

`Module[{`$x=x_0$`,`$y=y_0$`,…},`*body*`]`	a module with initial values for local variables

Assigning initial values to local variables.

This specifies `t` to be a local variable, with initial value `u`.

```
In[13]:= g[u_] := Module[{t = u}, t += t / (1 + u)]
```

This uses the definition of `g`.

```
In[14]:= g[a]
```

Out[14]= $a + \frac{a}{1 + a}$

You can define initial values for any of the local variables in a module. The initial values are always evaluated before the module is executed. As a result, even if a variable `x` is defined as local to the module, the global `x` will be used if it appears in an expression for an initial value.

The initial value of `u` is taken to be the global value of `t`.

```
In[15]:= Module[{t = 6, u = t}, u^2]
Out[15]= 289
```

lhs`:=Module[`*vars*`,`*rhs*`/;`*cond*`]`	share local variables between *rhs* and *cond*

Using local variables in definitions with conditions.

When you set up `/;` conditions for definitions, you often need to introduce temporary variables. In many cases, you may want to share these temporary variables with the body of the right-hand side of the definition. *Mathematica* allows you to enclose the whole right-hand side of your definition in a module, including the condition.

This defines a function with a condition attached.

```
In[16]:= h[x_] := Module[{t}, t^2 - 1 /; (t = x - 4) > 1]
```

Mathematica shares the value of the local variable `t` between the condition and the body of the right-hand side.

```
In[17]:= h[10]
Out[17]= 35
```

Local Constants

`With[{x=x0, y=y0, ...}, body]`	define local constants x, y, …

Defining local constants.

`Module` allows you to set up local *variables*, to which you can assign values and then change them. Often, however, all you really need are local *constants*, to which you assign a value only once. The *Mathematica* `With` construct allows you to set up such local constants.

This defines a global value for `t`.

```
In[1]:= t = 17
Out[1]= 17
```

This defines a function using `t` as a local constant.

```
In[2]:= w[x_] := With[{t = x + 1}, t + t^3]
```

This uses the definition of `w`.

```
In[3]:= w[a]
Out[3]= 1 + a + (1 + a)^3
```

`t` still has its global value.

```
In[4]:= t
Out[4]= 17
```

Just as in `Module`, the initial values you define in `With` are evaluated before `With` is executed.

The expression `t + 1` which gives the value of the local constant `t` is evaluated using the global `t`.

```
In[5]:= With[{t = t + 1}, t^2]
Out[5]= 324
```

The way `With[{`$x = x_0$`, …}, `*body*`]` works is to take *body*, and replace every occurrence of x, etc. in it by x_0, etc. You can think of `With` as a generalization of the `/.` operator, suitable for application to *Mathematica* code instead of other expressions.

This replaces `x` with `a`.

```
In[6]:= With[{x = a}, x = 5]
Out[6]= 5
```

After the replacement, the body of `With` is `a = 5`, so `a` gets the global value `5`.

```
In[7]:= a
Out[7]= 5
```

This clears the value of `a`.

```
In[8]:= Clear[a]
```

In some respects, `With` is like a special case of `Module`, in which each local variable is assigned a value exactly once.

One of the main reasons for using `With` rather than `Module` is that it typically makes the *Mathematica* programs you write easier to understand. In a module, if you see a local variable x at a particular point, you potentially have to trace through all of the code in the module to work out the value of x at that point. In a `With` construct, however, you can always find out the value of a local constant simply by looking at the initial list of values, without having to trace through specific code.

If you have several `With` constructs, it is always the innermost one for a particular variable that is in effect. You can mix `Module` and `With`. The general rule is that the innermost one for a particular variable is the one that is in effect.

With nested `With` constructs, the innermost one is always the one in effect.

```
In[9]:= With[{t = 8}, With[{t = 9}, t^2]]
Out[9]= 81
```

You can mix `Module` and `With` constructs.

```
In[10]:= Module[{t = 8}, With[{t = 9}, t^2]]
Out[10]= 81
```

Local variables in inner constructs do not mask ones outside unless the names conflict.

```
In[11]:= With[{t = a}, With[{u = b}, t + u]]
Out[11]= a + b
```

Except for the question of when *x* and *body* are evaluated, `With[{`*x* = x_0`, …}, `*body*`]` works essentially like *body* `/.` *x* `->` x_0. However, `With` behaves in a special way when the expression *body* itself contains `With` or `Module` constructs. The main issue is to prevent the local constants in the various `With` constructs from conflicting with each other, or with global objects. The details of how this is done are discussed in "How Modules Work".

The `y` in the inner `With` is renamed to prevent it from conflicting with the global `y`.

```
In[12]:= With[{x = 2 + y}, Hold[With[{y = 4}, x + y]]]
Out[12]= Hold[With[{y$ = 4}, (2 + y) + y$]]
```

How Modules Work

The way modules work in *Mathematica* is basically very simple. Every time any module is used, a new symbol is created to represent each of its local variables. The new symbol is given a unique name which cannot conflict with any other names. The name is formed by taking the name you specify for the local variable, followed by `$`, with a unique "serial number" appended.

The serial number is found from the value of the global variable `$ModuleNumber`. This variable counts the total number of times any `Module` of any form has been used.

`Module` generates symbols with names of the form *x$nnn* to represent each local variable.

The basic principle of modules in *Mathematica*.

This shows the symbol generated for `t` within the module.

```
In[1]:= Module[{t}, Print[t]]

           t$1
```

The symbols are different every time any module is used.

```
In[2]:= Module[{t, u}, Print[t]; Print[u]]

           t$2

           u$2
```

For most purposes, you will never have to deal directly with the actual symbols generated inside modules. However, if for example you start up a dialog while a module is being executed, then you will see these symbols. The same is true whenever you use functions like `Trace` to watch the evaluation of modules.

You see the symbols that are generated inside modules when you use `Trace`.

```
In[3]:= Trace[Module[{t}, t = 3]]
Out[3]= {Module[{t}, t = 3], {t$3 = 3, 3}, 3}
```

This starts a dialog inside a module.

```
In[4]:= Module[{t}, t = 6; Dialog[]]
```

Inside the dialog, you see the symbols generated for local variables such as `t`.

```
In[5]:= Stack[_]
Out[5]= {Module[{t}, t = 6; Dialog[]], t$4 = 6; Dialog[], Dialog[]}
```

You can work with these symbols as you would with any other symbols.

```
In[6]:= t$4 + 1
Out[6]= 7
```

This returns from the dialog.

```
In[7]:= Return[t$4 ^ 2]
Out[7]= 36
```

Under some circumstances, it is convenient explicitly to return symbols that are generated inside modules.

You can explicitly return symbols that are generated inside modules.

```
In[8]:= Module[{t}, t]
Out[8]= t$6
```

You can treat these symbols as you would any others.

```
In[9]:= %^2 + 1
```

Out[9]= $1 + t\$6^2$

`Unique[`*x*`]`	generate a new symbol with a unique name of the form *x*$*nnn*
`Unique[{`*x*`,`*y*`,`…`}]`	generate a list of new symbols

Generating new symbols with unique names.

The function `Unique` allows you to generate new symbols in the same way as `Module` does. Each time you call `Unique`, `$ModuleNumber` is incremented, so that the names of new symbols are guaranteed to be unique.

This generates a unique new symbol whose name starts with `x`.

```
In[10]:= Unique[x]
Out[10]= x$7
```

Each time you call `Unique` you get a symbol with a larger serial number.

```
In[11]:= {Unique[x], Unique[x], Unique[x]}
Out[11]= {x$8, x$9, x$10}
```

If you call `Unique` with a list of names, you get the same serial number for each of the symbols.

```
In[12]:= Unique[{x, xa, xb}]
Out[12]= {x$11, xa$11, xb$11}
```

You can use the standard *Mathematica* `?`*name* mechanism to get information on symbols that were generated inside modules or by the function `Unique`.

Executing this module generates the symbol `q$`*nnn*.

```
In[13]:= Module[{q}, q^2 + 1]
```

Out[13]= $1 + q\$12^2$

You can see the generated symbol here.

```
In[14]:= ? q*

    q    q$12
```

Symbols generated by `Module` behave in exactly the same way as other symbols for the purposes of evaluation. However, these symbols carry the attribute `Temporary`, which specifies that they should be removed completely from the system when they are no longer used. Thus most symbols that are generated inside modules are removed when the execution of those modules is finished. The symbols survive only if they are explicitly returned.

This shows a new `q` variable generated inside a module.

```
In[15]:= Module[{q}, Print[q]]

            q$13
```

The new variable is removed when the execution of the module is finished, so it does not show up here.

```
In[16]:= ?q*

            q     q$12
```

You should realize that the use of names such as *x*$*nnn* for generated symbols is purely a convention. You can in principle give any symbol a name of this form. But if you do, the symbol may collide with one that is produced by `Module`.

An important point to note is that symbols generated by `Module` are in general unique only within a particular *Mathematica* session. The variable `$ModuleNumber` which determines the serial numbers for these symbols is always reset at the beginning of each session.

This means in particular that if you save expressions containing generated symbols in a file, and then read them into another session, there is no guarantee that conflicts will not occur.

One way to avoid such conflicts is explicitly to set `$ModuleNumber` differently at the beginning of each session. In particular, if you set `$ModuleNumber = 10^10 $SessionID`, you should avoid any conflicts. The global variable `$SessionID` should give a unique number which characterizes a particular *Mathematica* session on a particular computer. The value of this variable is determined from such quantities as the absolute date and time, the ID of your computer, and, if appropriate, the ID of the particular *Mathematica* process.

`$ModuleNumber`	the serial number for symbols generated by `Module` and `Unique`
`$SessionID`	a number that should be different for every *Mathematica* session

Variables to be used in determining serial numbers for generated symbols.

Having generated appropriate symbols to represent the local variables you have specified, `Module[`*vars*`, `*body*`]` then has to evaluate *body* using these symbols. The first step is to take the actual expression *body* as it appears inside the module, and effectively to use `With` to replace all occurrences of each local variable name with the appropriate generated symbol. After this is done, `Module` actually performs the evaluation of the resulting expression.

An important point to note is that `Module[`*vars*`, `*body*`]` inserts generated symbols only into the actual expression *body*. It does not, for example, insert such symbols into code that is called from *body*, but does not explicitly appear in *body*.

"Blocks and Local Values" discusses how you can use `Block` to set up "local values" which work in a different way.

Since `x` does not appear explicitly in the body of the module, the local value is not used.

```
In[17]:=  tmp = x^2 + 1; Module[{x = 4}, tmp]

Out[17]=  1 + x^2
```

Most of the time, you will probably set up modules by giving explicit *Mathematica* input of the form `Module[`*vars*`, `*body*`]`. Since the function `Module` has the attribute `HoldAll`, the form of *body* will usually be kept unevaluated until the module is executed.

It is, however, possible to build modules dynamically in *Mathematica*. The generation of new symbols, and their insertion into *body* are always done only when a module is actually executed, not when the module is first given as *Mathematica* input.

This evaluates the body of the module immediately, making `x` appear explicitly.

```
In[18]:=  tmp = x^2 + 1; Module[{x = 4}, Evaluate[tmp]]

Out[18]=  17
```

Variables in Pure Functions and Rules

`Module` and `With` allow you to give a specific list of symbols whose names you want to treat as local. In some situations, however, you want to automatically treat certain symbol names as local.

For example, if you use a pure function such as `Function[{x}, x + a]`, you want `x` to be treated as a "formal parameter", whose specific name is local. The same is true of the `x` that appears in a rule like `f[x_] -> x^2`, or a definition like `f[x_] := x^2`.

Mathematica uses a uniform scheme to make sure that the names of formal parameters which appear in constructs like pure functions and rules are kept local, and are never confused with global names. The basic idea is to replace formal parameters when necessary by symbols with names of the form *x*$. By convention, *x*$ is never used as a global name.

Here is a nested pure function.

```
In[1]:= Function[{x}, Function[{y}, x + y]]
Out[1]= Function[{x}, Function[{y}, x + y]]
```

Mathematica renames the formal parameter y in the inner function to avoid conflict with the global object y.

```
In[2]:= %[2 y]
Out[2]= Function[{y$}, 2 y + y$]
```

The resulting pure function behaves as it should.

```
In[3]:= %[a]
Out[3]= a + 2 y
```

In general, *Mathematica* renames the formal parameters in an object like Function[*vars*, *body*] whenever *body* is modified in any way by the action of another pure function.

The formal parameter y is renamed because the body of the inner pure function was changed.

```
In[4]:= Function[{x}, Function[{y}, x + y]][a]
Out[4]= Function[{y$}, a + y$]
```

Since the body of the inner function does not change, the formal parameter is not renamed.

```
In[5]:= Function[{x}, x + Function[{y}, y^2]][a]
Out[5]= a + Function[{y}, y²]
```

Mathematica renames formal parameters in pure functions more liberally than is strictly necessary. In principle, renaming could be avoided if the names of the formal parameters in a particular function do not actually conflict with parts of expressions substituted into the body of the pure function. For uniformity, however, *Mathematica* still renames formal parameters even in such cases.

In this case, the formal parameter `x` in the inner function shields the body of the function, so no renaming is needed.

```
In[6]:=  Function[{x}, Function[{x}, x + y]][a]
Out[6]= Function[{x}, x + y]
```

Here are three nested functions.

```
In[7]:=  Function[{x}, Function[{y}, Function[{z}, x + y + z]]]
Out[7]= Function[{x}, Function[{y}, Function[{z}, x + y + z]]]
```

Both inner functions are renamed in this case.

```
In[8]:=  %[a]
Out[8]= Function[{y$}, Function[{z$}, a + y$ + z$]]
```

As mentioned in "Pure Functions", pure functions in *Mathematica* are like λ expressions in formal logic. The renaming of formal parameters allows *Mathematica* pure functions to reproduce all the semantics of standard λ expressions faithfully.

`Function[{`*x*`,`…`},`*body*`]`	local parameters
lhs`->`*rhs* and *lhs*`:>`*rhs*	local pattern names
lhs`=`*rhs* and *lhs*`:=`*rhs*	local pattern names
`With[{`*x*`=`x_0`,`…`},`*body*`]`	local constants
`Module[{`*x*`,`…`},`*body*`]`	local variables

Scoping constructs in *Mathematica*.

Mathematica has several "scoping constructs" in which certain names are treated as local. When you mix these constructs in any way, *Mathematica* does appropriate renamings to avoid conflicts.

Mathematica renames the formal parameter of the pure function to avoid a conflict.

```
In[9]:=  With[{x = a}, Function[{a}, a + x]]
Out[9]= Function[{a$}, a$ + a]
```

Here the local constant in the inner `With` is renamed to avoid a conflict.

```
In[10]:=  With[{x = y}, Hold[With[{y = 4}, x + y]]]
Out[10]= Hold[With[{y$ = 4}, y + y$]]
```

There is no conflict between names in this case, so no renaming is done.

```
In[11]:= With[{x = y}, Hold[With[{z = x + 2}, z + 2]]]
Out[11]= Hold[With[{z = y + 2}, z + 2]]
```

The local variable y in the module is renamed to avoid a conflict.

```
In[12]:= With[{x = y}, Hold[Module[{y}, x + y]]]
Out[12]= Hold[Module[{y$}, y + y$]]
```

If you execute the module, however, the local variable is renamed again to make its name unique.

```
In[13]:= ReleaseHold[%]
Out[13]= y + y$1
```

Mathematica treats transformation rules as scoping constructs, in which the names you give to patterns are local. You can set up named patterns either using *x*_, *x*__ and so on, or using *x* : *patt*.

The x in the h goes with the x_, and is considered local to the rule.

```
In[14]:= With[{x = 5}, g[x_, x] -> h[x]]
Out[14]= g[x_, 5] → h[x]
```

In a rule like f[x_] -> x + y, the x which appears on the right-hand side goes with the name of the x_ pattern. As a result, this x is treated as a variable local to the rule, and cannot be modified by other scoping constructs.

The y, on the other hand, is not local to the rule, and *can* be modified by other scoping constructs. When this happens, *Mathematica* renames the patterns in the rule to prevent the possibility of a conflict.

Mathematica renames the x in the rule to prevent a conflict.

```
In[15]:= With[{w = x}, f[x_] -> w + x]
Out[15]= f[x$_] → x + x$
```

When you use With on a scoping construct, *Mathematica* automatically performs appropriate renamings. In some cases, however, you may want to make substitutions inside scoping constructs, without any renaming. You can do this using the /. operator.

When you substitute for `y` using `With`, the `x` in the pure function is renamed to prevent a conflict.

```
In[16]:= With[{y = x + a}, Function[{x}, x + y]]

Out[16]= Function[{x$}, x$ + (a + x)]
```

If you use `/.` rather than `With`, no such renaming is done.

```
In[17]:= Function[{x}, x + y] /. y -> a + x

Out[17]= Function[{x}, x + (a + x)]
```

When you apply a rule such as *f*[*x*_] -> *rhs*, or use a definition such as *f*[*x*_] := *rhs*, *Mathematica* implicitly has to substitute for *x* everywhere in the expression *rhs*. It effectively does this using the `/.` operator. As a result, such substitution does not respect scoping constructs. However, when the insides of a scoping construct are modified by the substitution, the other variables in the scoping construct are renamed.

This defines a function for creating pure functions.

```
In[18]:= mkfun[var_, body_] := Function[{var}, body]
```

The `x` and `x^2` are explicitly inserted into the pure function, effectively by using the `/.` operator.

```
In[19]:= mkfun[x, x^2]

Out[19]= Function[{x}, x^2]
```

This defines a function that creates a pair of nested pure functions.

```
In[20]:= mkfun2[var_, body_] := Function[{x}, Function[{var}, body + x]]
```

The `x` in the outer pure function is renamed in this case.

```
In[21]:= mkfun2[x, x^2]

Out[21]= Function[{x$}, Function[{x}, x^2 + x$]]
```

Dummy Variables in Mathematics

When you set up mathematical formulas, you often have to introduce various kinds of local objects or "dummy variables". You can treat such dummy variables using modules and other *Mathematica* scoping constructs.

Integration variables are a common example of dummy variables in mathematics. When you write down a formal integral, conventional notation requires you to introduce an integration variable with a definite name. This variable is essentially "local" to the integral, and its name, while arbitrary, must not conflict with any other names in your mathematical expression.

Here is a function for evaluating an integral.

```
In[1]:= p[n_] := Integrate[f[s] s^n, {s, 0, 1}]
```

The s here conflicts with the integration variable.

```
In[2]:= p[s + 1]
```

Out[2]= $\int_0^1 s^{1+s}\, f[s]\, ds$

Here is a definition with the integration variable specified as local to a module.

```
In[3]:= pm[n_] := Module[{s}, Integrate[f[s] s^n, {s, 0, 1}]]
```

Since you have used a module, *Mathematica* automatically renames the integration variable to avoid a conflict.

```
In[4]:= pm[s + 1]
```

Out[4]= $\int_0^1 s\$20^{1+s}\, f[s\$20]\, ds\$20$

In many cases, the most important issue is that dummy variables should be kept local, and should not interfere with other variables in your mathematical expression. In some cases, however, what is instead important is that different uses of the *same* dummy variable should not conflict.

Repeated dummy variables often appear in products of vectors and tensors. With the "summation convention", any vector or tensor index that appears exactly twice is summed over all its possible values. The actual name of the repeated index never matters, but if there are two separate repeated indices, it is essential that their names do not conflict.

This sets up the repeated index j as a dummy variable.

```
In[5]:= q[i_] := Module[{j}, a[i, j] b[j]]
```

The module gives different instances of the dummy variable different names.

```
In[6]:= q[i1] q[i2]
Out[6]= a[i1, j$29] a[i2, j$30] b[j$29] b[j$30]
```

There are many situations in mathematics where you need to have variables with unique names. One example is in representing solutions to equations. With an equation like $\cos(x) = 1$, there are an infinite number of solutions, each of the form $x = 2\pi n$, where n is a dummy variable that can be equal to any integer.

When *Mathematica* solves this equation, it creates a dummy variable.

```
In[7]:= Reduce[Cos[x] == 1, x]
Out[7]= C[1] ∈ Integers && x == 2 π C[1]
```

Here is a way to make the dummy variable unique.

```
In[8]:= Reduce[Cos[x] == 1, x, GeneratedParameters :> Unique[C]]
Out[8]= C$489[1] ∈ Integers && x == 2 π C$489[1]
```

Another place where unique objects are needed is in representing "constants of integration". When you do an integral, you are effectively solving an equation for a derivative. In general, there are many possible solutions to the equation, differing by additive "constants of integration". The standard *Mathematica* `Integrate` function always returns a solution with no constant of integration. But if you were to introduce constants of integration, you would need to use modules to make sure that they are always unique.

Blocks and Local Values

Modules in *Mathematica* allow you to treat the *names* of variables as local. Sometimes, however, you want the names to be global, but *values* to be local. You can do this in *Mathematica* using `Block`.

`Block[{`x`,`y`,…},`*body*`]`	evaluate *body* using local values for x, y, …
`Block[{`x=x_0`,`y=y_0`,…},`*body*`]`	assign initial values to x, y, …

Setting up local values.

Here is an expression involving `x`.

```
In[1]:= x^2 + 3
Out[1]= 3 + x²
```

This evaluates the previous expression, using a local value for x.

```
In[2]:= Block[{x = a + 1}, %]
```

```
Out[2]= 3 + (1 + a)^2
```

There is no global value for x.

```
In[3]:= x
```

```
Out[3]= x
```

As described in "Modules and Local Variables", the variable x in a module such as `Module[{`x`}, `*body*`]` is always set up to refer to a unique symbol, different each time the module is used, and distinct from the global symbol x. The x in a block such as `Block[{`x`}, `*body*`]` is, however, taken to be the global symbol x. What the block does is to make the *value* of x local. The value x had when you entered the block is always restored when you exit the block. And during the execution of the block, x can take on any value.

This sets the symbol t to have value 17.

```
In[4]:= t = 17
```

```
Out[4]= 17
```

Variables in modules have unique local names.

```
In[5]:= Module[{t}, Print[t]]
```

```
t$1
```

In blocks, variables retain their global names, but can have local values.

```
In[6]:= Block[{t}, Print[t]]
```

```
t
```

t is given a local value inside the block.

```
In[7]:= Block[{t}, t = 6; t^4 + 1]
```

```
Out[7]= 1297
```

When the execution of the block is over, the previous value of t is restored.

```
In[8]:= t
```

```
Out[8]= 17
```

Blocks in *Mathematica* effectively allow you to set up "environments" in which you can temporarily change the values of variables. Expressions you evaluate at any point during the execution of a block will use the values currently defined for variables in the block. This is true whether the expressions appear directly as part of the body of the block, or are produced at any point in its evaluation.

This defines a delayed value for the symbol `u`.

```
In[9]:=  u := x^2 + t^2
```

If you evaluate `u` outside a block, the global value for `t` is used.

```
In[10]:=  u
Out[10]=  289 + x^2
```

You can specify a temporary value for `t` to use inside the block.

```
In[11]:=  Block[{t = 5}, u + 7]
Out[11]=  32 + x^2
```

An important implicit use of `Block` in *Mathematica* is for iteration constructs such as `Do`, `Sum` and `Table`. *Mathematica* effectively uses `Block` to set up local values for the iteration variables in all of these constructs.

`Sum` automatically makes the value of the iterator `t` local.

```
In[12]:=  Sum[t^2, {t, 10}]
Out[12]=  385
```

The local values in iteration constructs are slightly more general than in `Block`. They handle variables such as `a[1]`, as well as pure symbols.

```
In[13]:=  Sum[a[1]^2, {a[1], 10}]
Out[13]=  385
```

When you set up functions in *Mathematica*, it is sometimes convenient to have "global variables" which can affect the functions without being given explicitly as arguments. Thus, for example, *Mathematica* itself has a global variable `$RecursionLimit` which affects the evaluation of all functions, but is never explicitly given as an argument.

Mathematica will usually keep any value you define for a global variable until you explicitly change it. Often, however, you want to set up values which last only for the duration of a particular computation, or part of a computation. You can do this by making the values local to a *Mathematica* block.

This defines a function which depends on the "global variable" `t`.

```
In[14]:= f[x_] := x^2 + t
```

In this case, the global value of `t` is used.

```
In[15]:= f[a]
Out[15]= 17 + a^2
```

Inside a block, you can set up a local value for `t`.

```
In[16]:= Block[{t = 2}, f[b]]
Out[16]= 2 + b^2
```

You can use global variables not only to set parameters in functions, but also to accumulate results from functions. By setting up such variables to be local to a block, you can arrange to accumulate results only from functions called during the execution of the block.

This function increments the global variable `t`, and returns its current value.

```
In[17]:= h[x_] := (t += x^2)
```

If you do not use a block, evaluating `h[a]` changes the global value of `t`.

```
In[18]:= h[a]
Out[18]= 17 + a^2
```

With a block, only the local value of `t` is affected.

```
In[19]:= Block[{t = 0}, h[c]]
Out[19]= c^2
```

The global value of `t` remains unchanged.

```
In[20]:= t
Out[20]= 17 + a^2
```

When you enter a block such as `Block[{`*x*`}, `*body*`]`, any value for *x* is removed. This means that you can in principle treat *x* as a "symbolic variable" inside the block. However, if you explicitly return *x* from the block, it will be replaced by its value outside the block as soon as it is evaluated.

The value of `t` is removed when you enter the block.

```
In[21]:= Block[{t}, Print[Expand[(t + 1)^2]]]
```

$1 + 2\,t + t^2$

If you return an expression involving `t`, however, it is evaluated using the global value for `t`.

```
In[22]:= Block[{t}, t^2 - 3]
```

Out[22]= $-3 + \left(17 + a^2\right)^2$

Blocks Compared with Modules

When you write a program in *Mathematica*, you should always try to set it up so that its parts are as independent as possible. In this way, the program will be easier for you to understand, maintain and add to.

One of the main ways to ensure that different parts of a program do not interfere is to give their variables only a certain "scope". *Mathematica* provides two basic mechanisms for limiting the scope of variables: modules and blocks.

In writing actual programs, modules are far more common than blocks. When scoping is needed in interactive calculations, however, blocks are often convenient.

`Module[`*vars*`,`*body*`]`	lexical scoping
`Block[`*vars*`,`*body*`]`	dynamic scoping

Mathematica variable scoping mechanisms.

Most traditional computer languages use a so-called "lexical scoping" mechanism for variables, which is analogous to the module mechanism in *Mathematica*. Some symbolic computer languages such as LISP also allow "dynamic scoping", analogous to *Mathematica* blocks.

When lexical scoping is used, variables are treated as local to a particular section of the *code* in a program. In dynamic scoping, the values of variables are local to a part of the *execution history* of the program.

In compiled languages like C and Java, there is a very clear distinction between "code" and "execution history". The symbolic nature of *Mathematica* makes this distinction slightly less clear, since "code" can in principle be built up dynamically during the execution of a program.

What `Module[`*vars*`,` *body*`]` does is to treat the form of the expression *body* at the time when the module is executed as the "code" of a *Mathematica* program. Then when any of the *vars* explicitly appears in this "code", it is considered to be local.

`Block[`*vars*`,` *body*`]` does not look at the *form* of the expression *body*. Instead, throughout the evaluation of *body*, the block uses local values for the *vars*.

This defines `m` in terms of `i`.

```
In[1]:= m = i^2
Out[1]= i²
```

The local value for `i` in the block is used throughout the evaluation of `i + m`.

```
In[2]:= Block[{i = a}, i + m]
Out[2]= a + a²
```

Here only the `i` that appears explicitly in `i + m` is treated as a local variable.

```
In[3]:= Module[{i = a}, i + m]
Out[3]= a + i²
```

Contexts

It is always a good idea to give variables and functions names that are as explicit as possible. Sometimes, however, such names may get inconveniently long.

In *Mathematica*, you can use the notion of "contexts" to organize the names of symbols. Contexts are particularly important in *Mathematica* packages which introduce symbols whose names must not conflict with those of any other symbols. If you write *Mathematica* packages, or make sophisticated use of packages that others have written, then you will need to know about contexts.

The basic idea is that the *full name* of any symbol is broken into two parts: a *context* and a *short name*. The full name is written as *context*`` ` ``*short*, where the `` ` `` is the backquote or grave accent character (ASCII decimal code 96), called a "context mark" in *Mathematica*.

Here is a symbol with short name `x`, and context `aaaa`.

```
In[1]:=  aaaa`x
Out[1]= aaaa`x
```

You can use this symbol just like any other symbol.

```
In[2]:=  %^2 - %
Out[2]= -aaaa`x + aaaa`x^2
```

You can for example define a value for the symbol.

```
In[3]:=  aaaa`x = 78
Out[3]= 78
```

Mathematica treats `a`x` and `b`x` as completely different symbols.

```
In[4]:=  a`x == b`x
Out[4]= a`x == b`x
```

It is typical to have all the symbols that relate a particular topic in a particular context. Thus, for example, symbols that represent physical units might have a context `PhysicalUnits``. Such symbols might have full names like `PhysicalUnits`Joule` or `PhysicalUnits`Mole`.

Although you can always refer to a symbol by its full name, it is often convenient to use a shorter name.

At any given point in a *Mathematica* session, there is always a *current context* `$Context`. You can refer to symbols that are in this context simply by giving their short names, unless the symbol is shadowed by the symbol with the same short name on the `$ContextPath`. If a symbol with the given short name exists on the context path, it will be used instead of the symbol in the current context.

The default context for *Mathematica* sessions is `Global``.

```
In[5]:=  $Context
Out[5]= Global`
```

Short names are sufficient for symbols that are in the current context.

```
In[6]:=  {x, Global`x}
Out[6]= {x, x}
```

Contexts in *Mathematica* work somewhat like file directories in many operating systems. You can always specify a particular file by giving its complete name, including its directory. But at any given point, there is usually a current working directory, analogous to the current *Mathematica* context. Files that are in this directory can then be specified just by giving their short names.

Like directories in many operating systems, contexts in *Mathematica* can be hierarchical. Thus, for example, the full name of a symbol can involve a sequence of context names, as in c_1`$c_2$`c_3`*name*.

context`*name* or $c_1$`c_2`…`*name*	a symbol in an explicitly specified context
`*name*	a symbol in the current context
`*context*`*name* or `$c_1$`c_2`…`*name*	a symbol in a specific context relative to the current context
name	a symbol in the current context, or found on the context search path

Specifying symbols in various contexts.

Here is a symbol in the context `` a`b` ``.

```
In[7]:= a`b`x
Out[7]= a`b`x
```

When you start a *Mathematica* session, the default current context is `` Global` ``. Symbols that you introduce will usually be in this context. However, built-in symbols such as `Pi` are in the context `` System` ``.

In order to let you easily access not only symbols in the context `` Global` ``, but also in contexts such as `` System` ``, *Mathematica* supports the notion of a *context search path*. At any point in a *Mathematica* session, there is both a current context `$Context`, and also a current context search path `$ContextPath`. The idea of the search path is to allow you to type in the short name of a symbol, then have *Mathematica* search in a sequence of contexts to find a symbol with that short name.

The context search path for symbols in *Mathematica* is analogous to the "search path" for program files provided in operating systems.

The default context path includes the contexts for system-defined symbols.

```
In[8]:= $ContextPath
Out[8]= {System`, Global`}
```

When you type in `Pi`, *Mathematica* interprets it as the symbol with full name `System`Pi`.

```
In[9]:= Context[Pi]
Out[9]= System`
```

`Context[`*s*`]`	the context of a symbol
`$Context`	the current context in a *Mathematica* session
`$ContextPath`	the current context search path
`Contexts[]`	a list of all contexts

Finding contexts and context search paths.

When you use contexts in *Mathematica*, there is no reason that two symbols which are in different contexts cannot have the same short name. Thus, for example, you can have symbols with the short name `Mole` both in the context `PhysicalUnits`` and in the context `BiologicalOrganisms``.

There is, however, then the question of which symbol you actually get when you type in only the short name `Mole`. The answer to this question is determined by which of the contexts comes first in the sequence of contexts listed in the context search path.

This introduces two symbols, both with short name `Mole`.

```
In[10]:= {PhysicalUnits`Mole, BiologicalOrganisms`Mole}
Out[10]= {PhysicalUnits`Mole, BiologicalOrganisms`Mole}
```

This adds two additional contexts to `$ContextPath`. Typically, *Mathematica* adds new contexts to the beginning of `$ContextPath`.

```
In[11]:= $ContextPath = Join[{"PhysicalUnits`", "BiologicalOrganisms`"}, $ContextPath]
Out[11]= {PhysicalUnits`, BiologicalOrganisms`, System`, Global`}
```

Now if you type in `Mole`, you get the symbol in the context `PhysicalUnits``.

```
In[12]:= Context[Mole]
Out[12]= PhysicalUnits`
```

In general, when you type in a short name for a symbol, *Mathematica* assumes that you want the symbol with that name whose context appears earliest in the context search path. As a result, symbols with the same short name whose contexts appear later in the context search path are effectively "shadowed". To refer to these symbols, you need to use their full names.

Mathematica issues a message when you introduce new symbols that "shadow" existing symbols with your current choice for `$ContextPath`. In addition, in the notebook front end *Mathematica* warns you of shadowed symbols by coloring them red.

This introduces a symbol with short name `Mole` in the context `Global``. *Mathematica* warns you that the new symbol shadows existing symbols with short name `Mole`.

```
In[13]:= Global`Mole

      Global`Mole::shdw :
        Symbol Mole appears in multiple contexts {Global`, PhysicalUnits`, BiologicalOrganisms`};
            definitions in context Global` may shadow or be shadowed by other definitions. >>

Out[13]= Global`Mole
```

Now when you type in `Mole`, you get the symbol that appears first in the context path, `PhysicalUnits``.

```
In[14]:= Context[Mole]

Out[14]= PhysicalUnits`
```

If you once introduce a symbol which shadows existing symbols, it will continue to do so until you either rearrange `$ContextPath`, or explicitly remove the symbol. You should realize that it is not sufficient to clear the *value* of the symbol; you need to actually remove the symbol completely from *Mathematica*. You can do this using the function `Remove[s]`.

`Clear[s]`	clear the values of a symbol
`Remove[s]`	remove a symbol completely from the system

Clearing and removing symbols in *Mathematica*.

This removes the symbol `PhysicalUnits`Mole`.

```
In[15]:= Remove[Mole]
```

Now if you type in `Mole`, you get the symbol `BiologicalOrganisms`Mole`.

```
In[16]:= Context[Mole]
Out[16]= BiologicalOrganisms`
```

When *Mathematica* prints out the name of a symbol, it has to choose whether to give the full name, or just the short name. What it does is to give whatever version of the name you would have to type in to get the particular symbol, given your current settings for `$Context` and `$ContextPath`.

The short name is printed for the first symbol, so this would give that symbol if you typed it in.

```
In[17]:= {BiologicalOrganisms`Mole, Global`Mole}
Out[17]= {Mole, Global`Mole}
```

If you type in a short name for which there is no symbol either in the current context, or in any context on the context search path, then *Mathematica* has to *create* a new symbol with this name. It always puts new symbols of this kind in the current context, as specified by `$Context`.

This introduces the new symbol with short name `tree`.

```
In[18]:= tree
Out[18]= tree
```

Mathematica puts `tree` in the current context `Global``.

```
In[19]:= Context[tree]
Out[19]= Global`
```

Contexts and Packages

A typical package written in *Mathematica* introduces several new symbols intended for use outside the package. These symbols may correspond for example to new functions or new objects defined in the package.

There is a general convention that all new symbols introduced in a particular package are put into a context whose name is related to the name of the package. When you read in the package, it adds this context at the beginning of your context search path `$ContextPath`.

This reads in a package for proving primality.

```
In[1]:= << PrimalityProving`
```

The package prepends its context to `$ContextPath`.

```
In[2]:= $ContextPath
Out[2]= {PrimalityProving`, System`, Global`}
```

The symbol `ProvablePrimeQ` is in the context set up by the package.

```
In[3]:= Context[ProvablePrimeQ]
Out[3]= PrimalityProving`
```

You can refer to the symbol using its short name.

```
In[4]:= ProvablePrimeQ[2143]
Out[4]= True
```

The full names of symbols defined in packages are often quite long. In most cases, however, you will only need to use their short names. The reason for this is that after you have read in a package, its context is added to `$ContextPath`, so the context is automatically searched whenever you type in a short name.

There is a complication, however, when two symbols with the same short name appear in two different packages. In such a case, *Mathematica* will warn you when you read in the second package. It will tell you which symbols will be "shadowed" by the new symbols that are being introduced.

The symbol `ProvablePrimeQ` in the context `PrimalityProving`` is shadowed by the symbol with the same short name in the new package.

```
In[5]:= << NewPrimalityProving`

     ProvablePrimeQ::shdw :
       Symbol ProvablePrimeQ appears in multiple contexts {NewPrimalityProving`, PrimalityProving`};
           definitions in context NewPrimalityProving` may shadow or be shadowed by other definitions. >>
```

You can access the shadowed symbol by giving its full name.

```
In[6]:= PrimalityProving`ProvablePrimeQ[2143]
Out[6]= True
```

Conflicts can occur not only between symbols in different packages, but also between symbols in packages and symbols that you introduce directly in your *Mathematica* session. If you define a symbol in your current context, then this symbol may become shadowed by another symbol with the same short name in packages that you read in. The reason for this is that *Mathematica* searches for symbols in contexts on the context search path before looking in the current context.

This defines a function in the current context.

```
In[7]:=  Div[f_] = 1 / f
```

$$\text{Out[7]=}\quad \frac{1}{f}$$

The `Div` function in your current context will be shadowed by the one in the package.

```
In[8]:=  << VectorAnalysis`

      Div::shdw : Symbol Div appears in multiple contexts {VectorAnalysis`, Global`}; definitions
            in context VectorAnalysis` may shadow or be shadowed by other definitions. >>
```

This sets up the coordinate system for vector analysis.

```
In[9]:=  SetCoordinates[Cartesian[x, y, z]]
Out[9]= Cartesian[x, y, z]
```

The `Div` from the package is used.

```
In[10]:=  Div[{x, y^2, x}]
Out[10]= 1 + 2 y
```

If you get into the situation where unwanted symbols are shadowing the symbols you want, the best thing to do is usually to get rid of the unwanted symbols using `Remove[`*s*`]`. An alternative that is sometimes appropriate is to rearrange the entries in `$ContextPath` and to reset the value of `$Context` so as to make the contexts that contain the symbols you want be the ones that are searched first.

`$Packages`	a list of the contexts corresponding to all packages loaded into your *Mathematica* session

Getting a list of packages.

Mathematica Packages

One of the most important features of *Mathematica* is that it is an extensible system. There is a certain amount of mathematical and other functionality that is built into *Mathematica*. But by using the *Mathematica* language, it is always possible to add more functionality.

For many kinds of calculations, what is built into the standard version of *Mathematica* will be quite sufficient. However, if you work in a particular specialized area, you may find that you often need to use certain functions that are not built into *Mathematica*.

In such cases, you may well be able to find a *Mathematica* package that contains the functions you need. *Mathematica* packages are files written in the *Mathematica* language. They consist of collections of *Mathematica* definitions which "teach" *Mathematica* about particular application areas.

<<*package*	read in a *Mathematica* package

Reading in *Mathematica* packages.

If you want to use functions from a particular package, you must first read the package into *Mathematica*. The details of how to do this are discussed in "External Programs". There are various conventions that govern the names you should use to refer to packages.

This command reads in a particular *Mathematica* package.

```
In[1]:= << PrimalityProving`
```

The `ProvablePrimeQ` function is defined in the package.

```
In[2]:= ProvablePrimeQ[1093]
Out[2]= True
```

There are a number of subtleties associated with such issues as conflicts between names of functions in different packages. These are discussed in "Contexts and Packages". One point to note, however, is that you should not refer to a function that you will read from a package before actually reading in the package. If you do this by mistake, *Mathematica* will issue a message warning about the duplicate names and use the one last defined. This means that your version of the function will not be used; it will be the one from the package. You can execute the command `Remove["`*name*`"]` to get rid of the package function.

`Remove["name"]`	remove a function that has been introduced in error

Making sure that *Mathematica* uses correct definitions from packages.

The fact that *Mathematica* can be extended using packages means that the boundary of exactly what is "part of *Mathematica*" is quite blurred. As far as usage is concerned, there is actually no difference between functions defined in packages and functions that are fundamentally built into *Mathematica*.

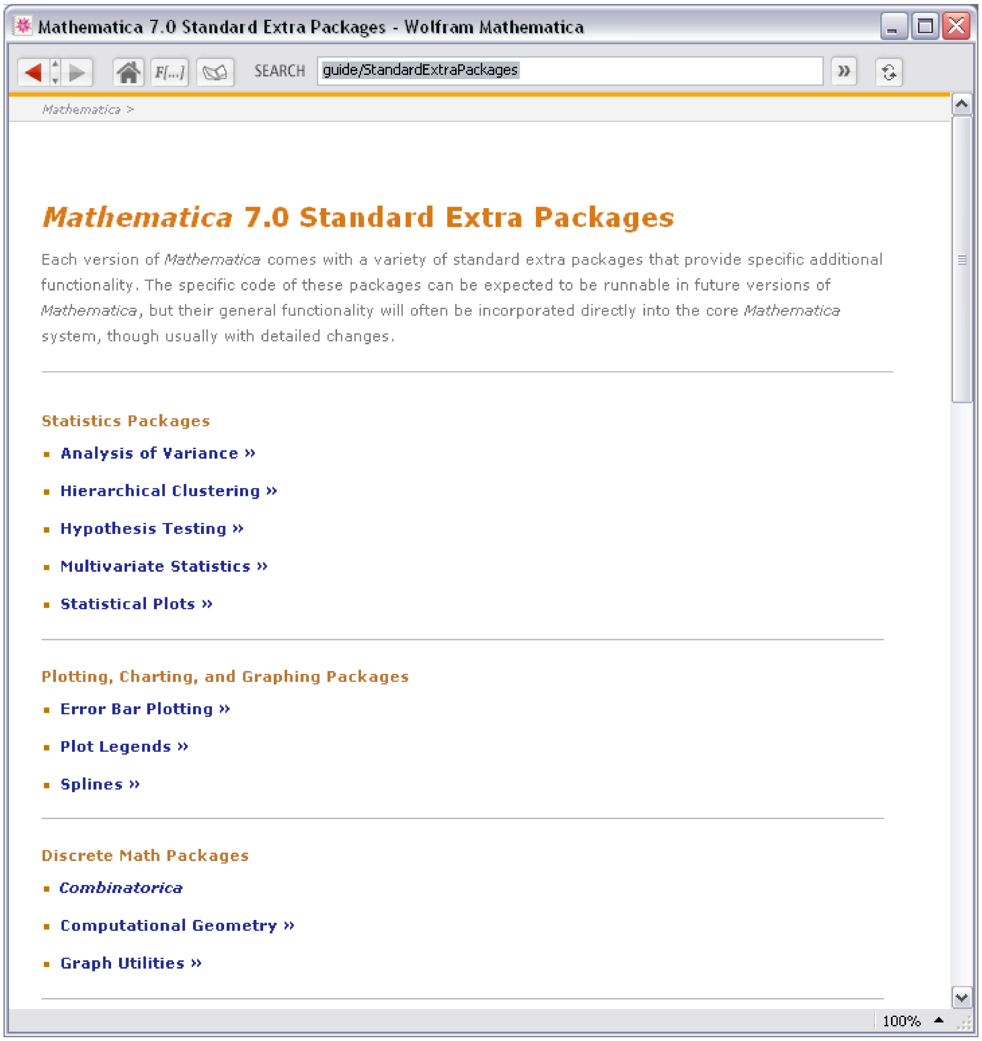
In fact, a fair number of the functions built into the core *Mathematica* system are actually implemented as *Mathematica* packages. However, on most *Mathematica* systems, the necessary packages have been preloaded, so that the functions they define are always present.

To blur the boundary of what is part of *Mathematica* even further, "Automatic Loading of Packages" describes how you can tell *Mathematica* automatically to load a particular package if you ever try to use a certain function. If you never use that function, then it will not be present. But as soon as you try to use it, its definition will be read in from a *Mathematica* package.

As a practical matter, the functions that should be considered "part of *Mathematica*" are probably those that are present in all *Mathematica* systems. It is these functions that are primarily discussed in this documentation.

Nevertheless, most versions of *Mathematica* come with a standard set of *Mathematica* packages, which contain definitions for many more functions. To use these functions, you must usually read in the necessary packages explicitly.

You can use the Documentation Center to get information on *Mathematica* 7 Standard Extra Packages.



Mathematica 7.0 Standard Extra Packages - Wolfram Mathematica

SEARCH guide/StandardExtraPackages

Mathematica >

Mathematica 7.0 Standard Extra Packages

Each version of *Mathematica* comes with a variety of standard extra packages that provide specific additional functionality. The specific code of these packages can be expected to be runnable in future versions of *Mathematica*, but their general functionality will often be incorporated directly into the core *Mathematica* system, though usually with detailed changes.

Statistics Packages

- Analysis of Variance »
- Hierarchical Clustering »
- Hypothesis Testing »
- Multivariate Statistics »
- Statistical Plots »

Plotting, Charting, and Graphing Packages

- Error Bar Plotting »
- Plot Legends »
- Splines »

Discrete Math Packages

- *Combinatorica*
- Computational Geometry »
- Graph Utilities »

100%

It is possible to set your *Mathematica* system up so that particular packages are preloaded, or are automatically loaded when needed. If you do this, then there may be many functions that appear as standard in your version of *Mathematica*, but which are not documented in the *Mathematica* system reference pages.

One point that should be mentioned is the relationship between packages and notebooks. Both are stored as files on your computer system, and both can be read into *Mathematica*. However, a notebook is intended to be displayed, typically with a notebook interface, while a package is intended only to be used as *Mathematica* input. Many notebooks in fact contain sections that can be considered as packages, and which contain sequences of definitions intended for input to *Mathematica*. There are also capabilities that allow packages set up to correspond to notebooks to be maintained automatically.

Setting Up *Mathematica* Packages

In a typical *Mathematica* package, there are generally two kinds of new symbols that are introduced. The first kind are ones that you want to "export" for use outside the package. The second kind are ones that you want to use only internally within the package. You can distinguish these two kinds of symbols by putting them in different contexts.

The usual convention is to put symbols intended for export in a context with a name *Package*` that corresponds to the name of the package. Whenever the package is read in, it adds this context to the context search path, so that the symbols in this context can be referred to by their short names.

Symbols that are not intended for export, but are instead intended only for internal use within the package, are conventionally put into a context with the name *Package*`Private`. This context is *not* added to the context search path. As a result, the symbols in this context cannot be accessed except by giving their full names.

Package`	symbols for export
Package`Private`	symbols for internal use only
System`	built-in *Mathematica* symbols
$Needed_1$` , $Needed_2$` , …	other contexts needed in the package

Contexts conventionally used in *Mathematica* packages.

There is a standard sequence of *Mathematica* commands that is typically used to set up the contexts in a package. These commands set the values of `$Context` and `$ContextPath` so that the new symbols which are introduced are created in the appropriate contexts.

`BeginPackage["`*Package*`` ` ``"]`	set *Package*`` ` `` to be the current context, and put only `` System` `` on the context search path
f`::usage="`*text*`"`, …	introduce the objects intended for export (and no others)
``Begin["`Private`"]``	set the current context to *Package*`` `Private` ``
f[*args*]=*value*, …	give the main body of definitions in the package
`End[]`	revert to the previous context (here *Package*`` ` ``)
`EndPackage[]`	end the package, prepending the *Package*`` ` `` to the context search path

The standard sequence of context control commands in a package.

```
BeginPackage["Collatz`"]

Collatz::usage =
        "Collatz[n] gives a list of the iterates in the 3n+1 problem,
        starting from n. The conjecture is that this sequence always
        terminates."

Begin["`Private`"]

Collatz[1] := {1}

Collatz[n_Integer]  := Prepend[Collatz[3 n + 1], n] /; OddQ[n] && n > 0

Collatz[n_Integer] := Prepend[Collatz[n/2], n] /; EvenQ[n] && n > 0

End[ ]

EndPackage[ ]
```

The sample package `Collatz.m`.

Defining `usage` messages at the beginning of a package is the standard way of making sure that symbols you want to export are created in the appropriate context. The way this works is that in defining these messages, the only symbols you mention are exactly the ones you want to export. These symbols are then created in the context *Package*`` ` ``, which is then current.

In the actual definitions of the functions in a package, there are typically many new symbols, introduced as parameters, temporary variables, and so on. The convention is to put all these symbols in the context *Package*`` `Private` ``, which is not put on the context search path when the package is read in.

This reads in the sample package given above.

```
In[1]:= << ExampleData/Collatz.m
```

The `EndPackage` command in the package adds the context associated with the package to the context search path.

```
In[2]:= $ContextPath
Out[2]= {Collatz`, Global`, System`}
```

The `Collatz` function was created in the context `` Collatz` ``.

```
In[3]:= Context[Collatz]
Out[3]= Collatz`
```

The parameter `n` is put in the private context `` Collatz`Private` ``.

```
In[4]:= ? Collatz`Private`*

          Collatz`Private`n
```

In the `Collatz` package, the functions that are defined depend only on built-in *Mathematica* functions. Often, however, the functions defined in one package may depend on functions defined in another package.

Two things are needed to make this work. First, the other package must be read in, so that the functions needed are defined. And second, the context search path must include the context that these functions are in.

You can explicitly tell *Mathematica* to read in a package at any point using the command `<<` *context*`` ` ``. ("Files for Packages" discusses the tricky issue of translation from system-independent context names to system-dependent file names.) Often, however, you want to set it up so that a particular package is read in only if it is needed. The command `Needs["`*context*`` `"] `` tells *Mathematica* to read in a package if the context associated with that package is not already in the list `$Packages`.

`Get["`*context*`` ` ``"]` or `<<`*context*`` ` ``	read in the package corresponding to the specified context
`Needs["`*context*`` ` ``"]`	read in the package if the specified context is not already in `$Packages`
`BeginPackage["`*Package*`` ` ``",{"`*Needed*$_1$`` ` ``", … }]`	begin a package, specifying that certain contexts in addition to ``System` `` are needed

Functions for specifying interdependence of packages.

If you use `BeginPackage["`*Package*`` ` ``"]` with a single argument, *Mathematica* puts on the context search path only the *Package*`` ` `` context and the contexts for built-in *Mathematica* symbols. If the definitions you give in your package involve functions from other packages, you must make sure that the contexts for these packages are also included in your context search path. You can do this by giving a list of the additional contexts as a second argument to `BeginPackage`. `BeginPackage` automatically calls `Needs` on these contexts, reading in the corresponding packages if necessary, and then making sure that the contexts are on the context search path.

`Begin["`*context*`` ` ``"]`	switch to a new current context
`End[]`	revert to the previous context

Context manipulation functions.

Executing a function like `Begin` which manipulates contexts changes the way that *Mathematica* interprets names you type in. However, you should realize that the change is effective only in subsequent expressions that you type in. The point is that *Mathematica* always reads in a complete input expression, and interprets the names in it, before it executes any part of the expression. As a result, by the time `Begin` is executed in a particular expression, the names in the expression have already been interpreted, and it is too late for `Begin` to have an effect.

The fact that context manipulation functions do not have an effect until the *next* complete expression is read in means that you must be sure to give those functions as separate expressions, typically on separate lines, when you write *Mathematica* packages.

The name `x` is interpreted before this expression is executed, so the `Begin` has no effect.

```
In[5]:= Begin["a`"]; Print[Context[x]]; End[]

           Global`

Out[5]= a`
```

Context manipulation functions are used primarily as part of packages intended to be read into *Mathematica*. Sometimes, however, you may find it convenient to use such functions interactively.

This can happen, for example, if you go into a dialog, say using `TraceDialog`, while executing a function defined in a package. The parameters and temporary variables in the function are typically in a private context associated with the package. Since this context is not on your context search path, *Mathematica* will print out the full names of the symbols, and will require you to type in these full names in order to refer to the symbols. You can however use `Begin["`*Package*`` `Private` ``"]` to make the private context of the package your current context. This will make *Mathematica* print out short names for the symbols, and allow you to refer to the symbols by their short names.

Files for Packages

When you create or use *Mathematica* packages, you will often want to refer to files in a system-independent way. You can use contexts to do this.

The basic idea is that on every computer system there is a convention about how files corresponding to *Mathematica* contexts should be named. Then, when you refer to a file using a context, the particular version of *Mathematica* you are using converts the context name to the file name appropriate for the computer system you are on.

`<<`*context*`` ` ``	read in the file corresponding to the specified context

Using contexts to specify files.

This reads in one of the standard packages that come with *Mathematica*.

```
In[1]:= << VectorAnalysis`
```

name`.mx`	file in `DumpSave` format
name`.mx/$SystemID/`*name*`.mx`	file in `DumpSave` format for your computer system
name`.m`	file in *Mathematica* source format
name`/init.m`	initialization file for a particular directory
dir`/…`	files in other directories specified by `$Path`

The typical sequence of files looked for by `<<` *name*`` ` ``.

Mathematica is set up so that `<< name`` will automatically try to load the appropriate version of a file. It will first try to load a *name*`.mx` file that is optimized for your particular computer system. If it finds no such file, then it will try to load a *name*`.m` file containing ordinary system-independent *Mathematica* input.

If *name* is a directory, then *Mathematica* will try to load the initialization file `init.m` in that directory. The purpose of the `init.m` file is to provide a convenient way to set up *Mathematica* packages that involve many separate files. The idea is to allow you to give just the command `<< name``, but then to load `init.m` to initialize the whole package, reading in whatever other files are necessary.

Automatic Loading of Packages

Other tutorials have discussed explicit loading of *Mathematica* packages using `<< package` and `Needs[package]`. Sometimes, however, you may want to set *Mathematica* up so that it automatically loads a particular package when the package is needed.

You can use `DeclarePackage` to give the names of symbols which are defined in a particular package. Then, when one of these symbols is actually used, *Mathematica* will automatically load the package where the symbol is defined.

`DeclarePackage["context`", {"name1", "name2", …}]`	declare that a package should automatically be loaded if a symbol with any of the names $name_i$ is used

Arranging for automatic loading of packages.

This specifies that the symbols `Div`, `Grad` and `Curl` are defined in `VectorAnalysis``.

```
In[1]:= DeclarePackage["VectorAnalysis`", {"Div", "Grad", "Curl"}]
Out[1]= VectorAnalysis`
```

When you first use `Grad`, *Mathematica* automatically loads the package that defines it.

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

When you set up a large collection of *Mathematica* packages, it is often a good idea to create an additional "names file" which contains a sequence of `DeclarePackage` commands, specifying packages to load when particular names are used. Within a particular *Mathematica* session, you then need to load explicitly only the names file. When you have done this, all the other packages will automatically be loaded if and when they are needed.

`DeclarePackage` works by immediately creating symbols with the names you specify, but giving each of these symbols the special attribute `Stub`. Whenever *Mathematica* finds a symbol with the `Stub` attribute, it automatically loads the package corresponding to the context of the symbol, in an attempt to find the definition of the symbol.

Manipulating Symbols and Contexts by Name

`Symbol["`*name*`"]`	construct a symbol with a given name
`SymbolName[`*symb*`]`	find the name of a symbol

Converting between symbols and their names.

Here is the symbol `x`.

```
In[1]:= x // InputForm
Out[1]//InputForm= x
```

Its name is a string.

```
In[2]:= SymbolName[x] // InputForm
Out[2]//InputForm= "x"
```

This gives the symbol `x` again.

```
In[3]:= Symbol["x"] // InputForm
Out[3]//InputForm= x
```

Once you have made an assignment such as `x = 2`, then whenever `x` is evaluated, it is replaced by `2`. Sometimes, however, you may want to continue to refer to `x` itself, without immediately getting the value of `x`.

You can do this by referring to `x` by name. The name of the symbol `x` is the string `"x"`, and even though `x` itself may be replaced by a value, the string `"x"` will always stay the same.

The names of the symbols x and xp are the strings "x" and "xp".

```
In[4]:= t = {SymbolName[x], SymbolName[xp]} // InputForm
Out[4]//InputForm= {"x", "xp"}
```

This assigns a value to x.

```
In[5]:= x = 2
Out[5]= 2
```

Whenever you enter x it is now replaced by 2.

```
In[6]:= {x, xp} // InputForm
Out[6]//InputForm= {2, xp}
```

The name "x" is not affected, however.

```
In[7]:= t // InputForm
Out[7]//InputForm= InputForm[{"x", "xp"}]
```

NameQ["*form*"]	test whether any symbol has a name which matches *form*
Names["*form*"]	give a list of all symbol names which match *form*
Contexts["*form*`"]	give a list of all context names which match *form*

Referring to symbols and contexts by name.

x and xp are symbols that have been created in this *Mathematica* session; xpp is not.

```
In[8]:= {NameQ["x"], NameQ["xp"], NameQ["xpp"]}
Out[8]= {True, True, False}
```

You can specify the form of symbol names using *string patterns* of the kind discussed in "String Patterns". "x*" stands, for example, for all names that start with x.

This gives a list of all symbol names in this *Mathematica* session that begin with x.

```
In[9]:= Names["x*"] // InputForm
Out[9]//InputForm= {"x", "xp"}
```

These names correspond to built-in functions in *Mathematica*.

```
In[10]:= Names["Qu*"] // InputForm
Out[10]//InputForm= {"QuadraticIrrationalQ", "Quantile", "Quartics", "QuartileDeviation", "Quartiles",
                     "QuartileSkewness", "Quiet", "Quit", "Quotient", "QuotientRemainder"}
```

This asks for names “close” to `WeierstrssP`.

```
In[11]:= Names["WeierstrssP", SpellingCorrection -> True]
Out[11]= {WeierstrassP}
```

`Clear["`*form*`"]`	clear the values of all symbols whose names match *form*
`Clear["`*context*`` `*"]``	clear the values of all symbols in the specified context
`Remove["`*form*`"]`	remove completely all symbols whose names match *form*
`Remove["`*context*`` `*"]``	remove completely all symbols in the specified context

Getting rid of symbols by name.

This clears the values of all symbols whose names start with `x`.

```
In[12]:= Clear["x*"]
```

The name `"x"` is still known, however.

```
In[13]:= Names["x*"]
Out[13]= {x, xp}
```

But the value of `x` has been cleared.

```
In[14]:= {x, xp}
Out[14]= {x, xp}
```

This removes completely all symbols whose names start with `x`.

```
In[15]:= Remove["x*"]
```

Now not even the name `"x"` is known.

```
In[16]:= Names["x*"]
Out[16]= {}
```

``Remove["Global`*"]``	remove completely all symbols in the ``Global` `` context

Removing all symbols you have introduced.

If you do not set up any additional contexts, then all the symbols that you introduce in a *Mathematica* session will be placed in the ``Global` `` context. You can remove these symbols completely using ``Remove["Global`*"]``. Built-in *Mathematica* objects are in the ``System` `` context, and are thus unaffected by this.

Intercepting the Creation of New Symbols

Mathematica creates a new symbol when you first enter a particular name. Sometimes it is useful to "intercept" the process of creating a new symbol. *Mathematica* provides several ways to do this.

`On[General::newsym]`	print a message whenever a new symbol is created
`Off[General::newsym]`	switch off the message printed when new symbols are created

Printing a message when new symbols are created.

This tells *Mathematica* to print a message whenever a new symbol is created.

```
In[1]:= On[General::newsym]
```

Mathematica now prints a message about each new symbol that it creates.

```
In[2]:= sin[k]

     General::newsym : Symbol sin is new. ≫

     General::newsym : Symbol k is new. ≫

Out[2]= sin[k]
```

This switches off the message.

```
In[3]:= Off[General::newsym]
```

Generating a message when *Mathematica* creates a new symbol is often a good way to catch typing mistakes. *Mathematica* itself cannot tell the difference between an intentionally new name, and a misspelling of a name it already knows. But by reporting all new names it encounters, *Mathematica* allows you to see whether any of them are mistakes.

`$NewSymbol`	a function to be applied to the name and context of new symbols which are created

Performing operations when new symbols are created.

When *Mathematica* creates a new symbol, you may want it not just to print a message, but instead to perform some other action. Any function you specify as the value of the global variable `$NewSymbol` will automatically be applied to strings giving the name and context of each new symbol that *Mathematica* creates.

This defines a function to be applied to each new symbol which is created.

```
In[4]:= $NewSymbol = Print["Name: ", #1, " Context: ", #2] &
Out[4]= Print[Name: , #1,  Context: , #2] &
```

The function is applied once to `v` and once to `w`.

```
In[5]:= v + w

      Name: v Context: Global`

      Name: w Context: Global`

Out[5]= v + w
```

Strings and Characters

Properties of Strings

Much of what *Mathematica* does revolves around manipulating structured expressions. But you can also use *Mathematica* as a system for handling unstructured strings of text.

"*text*"	a string containing arbitrary text

Text strings.

When you input a string of text to *Mathematica* you must always enclose it in quotes. However, when *Mathematica* outputs the string it usually does not explicitly show the quotes.

You can see the quotes by asking for the input form of the string. In addition, in a *Mathematica* notebook, quotes will typically appear automatically as soon as you start to edit a string.

When *Mathematica* outputs a string, it usually does not explicitly show the quotes.

```
In[1]:= "This is a string."
Out[1]= This is a string.
```

You can see the quotes, however, by asking for the input form of the string.

```
In[2]:= InputForm[%]
Out[2]//InputForm= "This is a string."
```

The fact that *Mathematica* does not usually show explicit quotes around strings makes it possible for you to use strings to specify quite directly the textual output you want.

The strings are printed out here without explicit quotes.

```
In[3]:= Print["The value is ", 567, "."]

        The value is 567.
```

You should understand, however, that even though the string `"x"` often appears as `x` in output, it is still a quite different object from the symbol `x`.

The string "x" is not the same as the symbol x.

```
In[4]:= "x" === x
Out[4]= False
```

You can test whether any particular expression is a string by looking at its head. The head of any string is always `String`.

All strings have head `String`.

```
In[5]:= Head["x"]
Out[5]= String
```

The pattern `_String` matches any string.

```
In[6]:= Cases[{"ab", x, "a", y}, _String]
Out[6]= {ab, a}
```

You can use strings just like other expressions as elements of patterns and transformations. Note, however, that you cannot assign values directly to strings.

This gives a definition for an expression that involves a string.

```
In[7]:= z["gold"] = 79
Out[7]= 79
```

This replaces each occurrence of the string "aa" by the symbol x.

```
In[8]:= {"aaa", "aa", "bb", "aa"} /. "aa" -> x
Out[8]= {aaa, x, bb, x}
```

Operations on Strings

Mathematica provides a variety of functions for manipulating strings. Most of these functions are based on viewing strings as a sequence of characters, and many of the functions are analogous to ones for manipulating lists.

s_1<>s_2<>… or StringJoin[{s_1,s_2,…}]	join several strings together
StringLength[s]	give the number of characters in a string
StringReverse[s]	reverse the characters in a string

Operations on complete strings.

You can join together any number of strings using <>.

```
In[1]:= "aaaaaaa" <> "bbb" <> "cccccccccc"
Out[1]= aaaaaaabbbcccccccccc
```

StringLength gives the number of characters in a string.

```
In[2]:= StringLength[%]
Out[2]= 20
```

StringReverse reverses the characters in a string.

```
In[3]:= StringReverse["A string."]
Out[3]= .gnirts A
```

StringTake[s,n]	make a string by taking the first n characters from s
StringTake[s,{n}]	take the n^{th} character from s
StringTake[s,{n_1,n_2}]	take characters n_1 through n_2
StringDrop[s,n]	make a string by dropping the first n characters in s
StringDrop[s,{n_1,n_2}]	drop characters n_1 through n_2

Taking and dropping substrings.

StringTake and StringDrop are the analogs for strings of Take and Drop for lists. Like Take and Drop, they use standard *Mathematica* sequence specifications, so that, for example, negative numbers count character positions from the end of a string. Note that the first character of a string is taken to have position 1.

Here is a sample string.

```
In[4]:= alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Out[4]= ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

This takes the first five characters from `alpha`.

```
In[5]:= StringTake[alpha, 5]
Out[5]= ABCDE
```

Here is the fifth character in `alpha`.

```
In[6]:= StringTake[alpha, {5}]
Out[6]= E
```

This drops the characters 10 through 2, counting from the end of the string.

```
In[7]:= StringDrop[alpha, {-10, -2}]
Out[7]= ABCDEFGHIJKLMNOPZ
```

`StringInsert[`*s*`,`*snew*`,`*n*`]`	insert the string *snew* at position *n* in *s*
`StringInsert[`*s*`,`*snew*`,{`n_1`,`n_2`,…}]`	insert several copies of *snew* into *s*

Inserting into a string.

`StringInsert[`*s*`, `*snew*`, `*n*`]` is set up to produce a string whose n^{th} character is the first character of *snew*.

This produces a new string whose fourth character is the first character of the string `"XX"`.

```
In[8]:= StringInsert["abcdefgh", "XX", 4]
Out[8]= abcXXdefgh
```

Negative positions are counted from the end of the string.

```
In[9]:= StringInsert["abcdefgh", "XXX", -1]
Out[9]= abcdefghXXX
```

Each copy of `"XXX"` is inserted at the specified position in the original string.

```
In[10]:= StringInsert["abcdefgh", "XXX", {2, 4, -1}]
Out[10]= aXXXbcXXXdefghXXX
```

This uses `Riffle` to add a space between the words in a list.

```
In[11]:= StringJoin[Riffle[{"cat", "in", "the", "hat"}, " "]]
Out[11]= cat in the hat
```

`StringReplacePart[`s`,`$snew$`,{`m`,`n`}]`	replace the characters at positions m through n in s by the string $snew$
`StringReplacePart[`s`,` $snew$`,{{`m_1`,`n_1`},{`m_2`,`n_2`},…}]`	replace several substrings in s by $snew$
`StringReplacePart[` s`,{`$snew_1$`,`$snew_2$`,…},` `{{`m_1`,`n_1`},{`m_2`,`n_2`},…}]`	replace substrings in s by the corresponding $snew_i$

Replacing parts of a string.

This replaces characters 2 through 6 by the string "XXX".

```
In[12]:= StringReplacePart["abcdefgh", "XXX", {2, 6}]
Out[12]= aXXXgh
```

This replaces two runs of characters by the string "XXX".

```
In[13]:= StringReplacePart["abcdefgh", "XXX", {{2, 3}, {5, -1}}]
Out[13]= aXXXdXXX
```

Now the two runs of characters are replaced by different strings.

```
In[14]:= StringReplacePart["abcdefgh", {"XXX", "YYYY"}, {{2, 3}, {5, -1}}]
Out[14]= aXXXdYYYY
```

`StringPosition[`s`,`sub`]`	give a list of the starting and ending positions at which sub appears as a substring of s
`StringPosition[`s`,`sub`,`k`]`	include only the first k occurrences of sub in s
`StringPosition[`s`,{`sub_1`,`sub_2`,…}]`	include occurrences of any of the sub_i

Finding positions of substrings.

You can use `StringPosition` to find where a particular substring appears within a given string. `StringPosition` returns a list, each of whose elements corresponds to an occurrence of the substring. The elements consist of lists giving the starting and ending character positions for the substring. These lists are in the form used as sequence specifications in `StringTake`, `StringDrop` and `StringReplacePart`.

This gives a list of the positions of the substring "abc".

```
In[15]:= StringPosition["abcdabcdaabcabcd", "abc"]
Out[15]= {{1, 3}, {5, 7}, {10, 12}, {13, 15}}
```

This gives only the first occurrence of "abc".

```
In[16]:= StringPosition["abcdabcdaabcabcd", "abc", 1]
Out[16]= {{1, 3}}
```

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

```
In[17]:= StringPosition["abcdabcdcd", {"abc", "cd"}]
Out[17]= {{1, 3}, {3, 4}, {5, 7}, {7, 8}, {9, 10}}
```

This does not include overlaps.

```
In[18]:= StringPosition["abcdabcdcd", {"abc", "cd"}, Overlaps -> False]
Out[18]= {{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[19]:= StringCount["abcdabcdcd", {"abc", "cd"}]
Out[19]= 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.

This replaces all occurrences of the character `a` by the string `XX`.

```
In[20]:= StringReplace["abcdabcdaabcabcd", "a" -> "XX"]
Out[20]= XXbcdXXbcdXXXXbcXXbcd
```

This replaces `abc` by `Y`, and `d` by `XXX`.

```
In[21]:= StringReplace["abcdabcdaabcabcd", {"abc" -> "Y", "d" -> "XXX"}]
Out[21]= YXXXYXXXaYYXXX
```

The first occurrence of `cde` is not replaced because it overlaps with `abc`.

```
In[22]:= StringReplace["abcde abacde", {"abc" -> "X", "cde" -> "Y"}]
Out[22]= Xde abaY
```

`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[23]:= StringReplaceList["aaaaa", "a" -> "X"]
Out[23]= {Xaaaa, aXaaa, aaXaa, aaaXa, aaaaX}
```

This shows the results of all possible single replacements.

```
In[24]:= StringReplaceList["abcde abacde", {"abc" -> "X", "cde" -> "Y"}]
Out[24]= {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[25]:= StringSplit["a b::c d::e f g"]
Out[25]= {a, b::c, d::e, f, g}
```

This splits at each "`::`".

```
In[26]:= StringSplit["a b::c d::e f g", "::"]
Out[26]= {a b, c d, e f g}
```

This splits at each colon or space.

```
In[27]:= StringSplit["a b::c d::e f g", {":", " "}]
Out[27]= {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[28]:= StringSplit["a b::c d::e f g", "::" -> {x, y}]
Out[28]= {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[29]:= Sort[{"cat", "fish", "catfish", "Cat"}]
Out[29]= {cat, Cat, catfish, fish}
```

`StringTrim[`*s*`]`	trims whitespace from the beginning and end of *s*
`StringTrim[`*s*`,`*patt*`]`	trims substrings matching *patt* from the beginning and end

Remove whitespace from ends of string.

```
In[30]:= StringTrim["   abcabc   "] // FullForm
Out[30]//FullForm= "abcabc"
```

`SequenceAlignment[`s_1`,`s_2`]`	finds an optimal alignment of *s1* and *s2*

Find an optimal alignment of two strings.

```
In[31]:= SequenceAlignment["abcXabcXabc", "abcYabcYabc"]
Out[31]= {abc, {X, Y}, abc, {X, Y}, abc}
```

Characters in Strings

`Characters["`*string*`"]`	convert a string to a list of characters
`StringJoin[{"`c_1`","`c_2`",…}]`	convert a list of characters to a string

Converting between strings and lists of characters.

This gives a list of the characters in the string.

```
In[1]:= Characters["A string."]
Out[1]= {A,  , s, t, r, i, n, g, .}
```

You can apply standard list manipulation operations to this list.

```
In[2]:= RotateLeft[%, 3]
Out[2]= {t, r, i, n, g, ., A,  , s}
```

`StringJoin` converts the list of characters back to a single string.

```
In[3]:= StringJoin[%]
Out[3]= tring.A s
```

`DigitQ[`*string*`]`	test whether all characters in a string are digits
`LetterQ[`*string*`]`	test whether all characters in a string are letters
`UpperCaseQ[`*string*`]`	test whether all characters in a string are uppercase letters
`LowerCaseQ[`*string*`]`	test whether all characters in a string are lowercase letters

Testing characters in a string.

All characters in the string given are letters.

```
In[4]:= LetterQ["Mixed"]
Out[4]= True
```

Not all the letters are uppercase, so the result is `False`.

```
In[5]:= UpperCaseQ["Mixed"]
Out[5]= False
```

`ToUpperCase[`*string*`]`	generate a string in which all letters are uppercase
`ToLowerCase[`*string*`]`	generate a string in which all letters are lowercase

Converting between upper and lower case.

This converts all letters to upper case.

```
In[6]:= ToUpperCase["Mixed Form"]
Out[6]= MIXED FORM
```

`CharacterRange["`c_1`","`c_2`"]`	generate a list of all characters from c_1 and c_2

Generating ranges of characters.

This generates a list of lowercase letters in alphabetical order.

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

Here is a list of uppercase letters.

```
In[8]:= CharacterRange["T", "Z"]
Out[8]= {T, U, V, W, X, Y, Z}
```

Here are some digits.

```
In[9]:= CharacterRange["0", "7"]
Out[9]= {0, 1, 2, 3, 4, 5, 6, 7}
```

`CharacterRange` will usually give meaningful results for any range of characters that have a natural ordering. The way `CharacterRange` works is by using the character codes that *Mathematica* internally assigns to every character.

This shows the ordering defined by the internal character codes used by *Mathematica*.

```
In[10]:= CharacterRange["T", "e"]
Out[10]= {T, U, V, W, X, Y, Z, [, \, ], ^, _, `, a, b, c, d, e}
```

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}
```

`"`*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["`$c_1 c_2 \ldots$`"]` 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["`$c_1 c_2 \ldots$`"]`	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.

`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[StartOfString]`, etc.	anywhere except at the particular positions `StartOfString`, etc.

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 `Shortest[`*p*`]`.

`Longest[`*p*`]`	the longest consistent match for *p* (default)
`Shortest[`*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)}
```

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

```
In[34]:= StringCases["-(a)--(bb)--(c)-", Shortest["(" ~~ __ ~~ ")"]]
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 uppercase and lowercase letters as equivalent.

`IgnoreCase->True`	treat uppercase and lowercase 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}
```

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_1 c_2 …`]`	any of the characters c_i
`[`c_1`-`c_2`]`	any character in the range c_1-c_2
`[^`c_1 c_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_1 p_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_1 c_2 …`]`	`Characters["`c_1 c_2 …`"]`
`[`c_1`-`c_2`]`	`CharacterRange["`c_1`","`c_2`"]`
`[^`c_1 c_2 …`]`	`Except[Characters["`c_1 c_2 …`"]]`
p`*`	p`...`
p`+`	p`..`
p`?`	$p \mid$ `""`
p`*?`, p`+?`, p`??`	`Shortest[`p…`]`,…
`(`p_1 p_2 …`)`	`(`p_1`~~`p_2`~~`…`)`
$p_1 \mid p_2$	$p_1 \mid p_2$

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 uppercase 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}
```

Special Characters

In addition to the ordinary characters that appear on a standard keyboard, you can include in *Mathematica* strings any of the special characters that are supported by *Mathematica*.

Here is a string containing special characters.

```
In[1]:= "α⊕β⊕…"
Out[1]= α⊕β⊕…
```

You can manipulate this string just as you would any other.

```
In[2]:= StringReplace[%, "⊕" -> " ⊙⊙ "]
Out[2]= α ⊙⊙ β ⊙⊙ …
```

Here is the list of the characters in the string.

```
In[3]:= Characters[%]
Out[3]= {α,  , ⊙, ⊙,  , β,  , ⊙, ⊙,  , …}
```

In a *Mathematica* notebook, a special character such as α can always be displayed directly. But if you use a text-based interface, then typically the only characters that can readily be displayed are the ones that appear on your keyboard.

As a result, what *Mathematica* does in such situations is to try to approximate special characters by similar-looking sequences of ordinary characters. And when this is not practical, *Mathematica* just gives the full name of the special character.

In a *Mathematica* notebook using `StandardForm`, special characters can be displayed directly.

```
In[4]:= "Lamé ⟶ αβ+"
Out[4]= Lamé ⟶ αβ+
```

In `OutputForm`, however, the special characters are approximated when possible by sequences of ordinary ones.

```
In[5]:= % // OutputForm

Out[5]//OutputForm= Lamé ⟶ αβ+
```

Mathematica always uses full names for special characters in `InputForm`. This means that when special characters are written out to files or external programs, they are by default represented purely as sequences of ordinary characters.

This uniform representation is crucial in allowing special characters in *Mathematica* to be used in a way that does not depend on the details of particular computer systems.

In `InputForm` the full names of all special characters are always written out explicitly.

```
In[6]:= "Lamé ⟶ αβ+" // InputForm

Out[6]//InputForm= "Lamé ⟶ αβ+"
```

a	a literal character
\[*Name*]	a character specified using its full name
\"	a " to be included in a string
\\	a \ to be included in a string

Ways to enter characters in a string.

You have to use \ to “escape” any " or \ characters in strings that you enter.

```
In[7]:= "Strings can contain \"quotes\" and \\ characters."

Out[7]= Strings can contain "quotes" and \ characters.
```

\\ produces a literal \ rather than forming part of the specification of α.

```
In[8]:= "\\[Alpha] is α."

Out[8]= \[Alpha] is α.
```

This breaks the string into a list of individual characters.

```
In[9]:= Characters[%]

Out[9]= {\, [, A, l, p, h, a, ],  , i, s,  , α, .}
```

This creates a list of the characters in the full name of α.

```
In[10]:= Characters[ToString[FullForm["α"]]]

Out[10]= {", \, [, A, l, p, h, a, ], "}
```

And this produces a string consisting of an actual α from its full name.

```
In[11]:= ToExpression["\"\\[" <> "Alpha" <> "]\""]

Out[11]= α
```

Newlines and Tabs in Strings

`\n`	a newline (line feed) to be included in a string
`\t`	a tab to be included in a string

Explicit representations of newlines and tabs in strings.

This prints on two lines.

```
In[1]:= "First line.\nSecond line."

Out[1]= First line.
        Second line.
```

In `InputForm` there is an explicit `\n` to represent the newline.

```
In[2]:= InputForm[%]

Out[2]//InputForm=  "First line.\nSecond line."
```

Mathematica keeps line breaks entered within a string.

```
In[3]:= "A string on
        two lines."

Out[3]= A string on
        two lines.
```

There is a newline in the string.

```
In[4]:= InputForm[%]

Out[4]//InputForm= "A string on \ntwo lines."
```

With a single backslash at the end of a line, *Mathematica* ignores the line break.

```
In[5]:=  "A string on \
         one line."
Out[5]=  A string on one line.
```

You should realize that even though it is possible to achieve some formatting of *Mathematica* output by creating strings which contain raw tabs and newlines, this is rarely a good idea. Typically a much better approach is to use the higher-level *Mathematica* formatting primitives discussed in "String-Oriented Output Formats", "Output Formats for Numbers", and "Tables and Matrices". These primitives will always yield consistent output, independent of such issues as the positions of tab settings on a particular device.

In strings with newlines, text is always aligned on the left.

```
In[6]:=  {"Here is\na string\non several lines.", "Here is\nanother"}

Out[6]=  {Here is
         a string
         on several lines., Here is
         another}
```

The front end formatting construct `Column` gives more control. Here text is aligned on the right.

```
In[7]:=  Column[{"First line", "Second", "Third"}, Right]

         First line
Out[7]=      Second
              Third
```

And here the text is centered.

```
In[8]:=  Column[{"First line", "Second", "Third"}, Center]

         First line
Out[8]=    Second
            Third
```

Character Codes

`ToCharacterCode["`*string*`"]`	give a list of the character codes for the characters in a string
`FromCharacterCode[`*n*`]`	construct a character from its character code
`FromCharacterCode[{`n_1`,`n_2`,…}]`	construct a string of characters from a list of character codes

Converting to and from character codes.

Mathematica assigns every character that can appear in a string a unique *character code*. This code is used internally as a way to represent the character.

This gives the character codes for the characters in the string.

```
In[1]:= ToCharacterCode["ABCD abcd"]
Out[1]= {65, 66, 67, 68, 32, 97, 98, 99, 100}
```

`FromCharacterCode` reconstructs the original string.

```
In[2]:= FromCharacterCode[%]
Out[2]= ABCD abcd
```

Special characters also have character codes.

```
In[3]:= ToCharacterCode["α⊕Γ⊖∅"]
Out[3]= {945, 8853, 915, 8854, 8709}
```

`CharacterRange["`c_1`","`c_2`"]`	generate a list of characters with successive character codes

Generating sequences of characters.

This gives part of the English alphabet.

```
In[4]:= CharacterRange["a", "k"]
Out[4]= {a, b, c, d, e, f, g, h, i, j, k}
```

Here is the Greek alphabet.

```
In[5]:= CharacterRange["α", "ω"]
Out[5]= {α, β, γ, δ, ε, ζ, η, θ, ι, κ, λ, μ, ν, ξ, ο, π, ρ, ς, σ, τ, υ, φ, χ, ψ, ω}
```

Mathematica assigns names such as `\[Alpha]` to a large number of special characters. This means that you can always refer to such characters just by giving their names, without ever having to know their character codes.

This generates a string of special characters from their character codes.

```
In[6]:= FromCharacterCode[{8706, 8709, 8711, 8712}]
Out[6]= ∂∅∇∈
```

You can always refer to these characters by their names, without knowing their character codes.

```
In[7]:= FullForm[%]
Out[7]//FullForm= "\[PartialD]\[EmptySet]\[Del]\[Element]"
```

Mathematica has names for all the common characters that are used in mathematical notation and in standard European languages. But for a language such as Japanese, there are more than 3,000 additional characters, and *Mathematica* does not assign an explicit name to each of them. Instead, it refers to such characters by standardized character codes.

Here is a string containing Japanese characters.

```
In[8]:= "数学"
Out[8]= 数学
```

In `FullForm`, these characters are referred to by standardized character codes. The character codes are given in hexadecimal.

```
In[9]:= FullForm[%]
Out[9]//FullForm= "\:6570\:5b66"
```

The notebook front end for *Mathematica* is typically set up so that when you enter a character in a particular font, *Mathematica* will automatically work out the character code for that character.

Sometimes, however, you may find it convenient to be able to enter characters directly using character codes.

`\.`*nn*	a character with hexadecimal code *nn*
`\:`*nnnn*	a character with hexadecimal code *nnnn*

Ways to enter characters directly in terms of character codes.

For characters with character codes below 256, you can use `\.`*nn*. For characters with character codes above 256, you must use `\:`*nnnn*. Note that in all cases you must give a fixed number of hexadecimal digits, padding with leading 0s if necessary.

This gives character codes in hexadecimal for a few characters.

```
In[10]:= BaseForm[ToCharacterCode["Aàαℵ"], 16]
```

Out[10]//BaseForm= $\{41_{16}, e0_{16}, 3b1_{16}, 2135_{16}\}$

This enters the characters using their character codes. Note the leading 0 inserted in the character code for α.

```
In[11]:= "Aàαℵ"

Out[11]= Aàαℵ
```

In assigning codes to characters, *Mathematica* follows three compatible standards: ASCII, ISO Latin-1, and Unicode. ASCII covers the characters on a normal American English keyboard. ISO Latin-1 covers characters in many European languages. Unicode is a more general standard which defines character codes for several tens of thousands of characters used in languages and notations around the world.

0 - 127 (\.00 - \.7f)	ASCII characters
1 - 31 (\.01 - \.1f)	ASCII control characters
32 - 126 (\.20 - \.7e)	printable ASCII characters
97 - 122 (\.61 - \.7a)	lower-case English letters
129 - 255 (\.81 - \.ff)	ISO Latin-1 characters
192 - 255 (\.c0 - \.ff)	letters in European languages
0 - 59391 (\:0000 - \:e7ff)	Unicode standard public characters
913 - 1009 (\:0391 - \:03f1)	Greek letters
12288 - 35839 (\:3000 - \:8bff)	Chinese, Japanese and Korean characters
8450 - 8504 (\:2102 - \:2138)	modified letters used in mathematical notation
8592 - 8677 (\:2190 - \:21e5)	arrows
8704 - 8945 (\:2200 - \:22f1)	mathematical symbols and operators
64256 - 64300 (\:fb00 - \:fb2c)	Unicode private characters defined specially by *Mathematica*

A few ranges of character codes used by *Mathematica*.

Here are all the printable ASCII characters.

```
In[12]:= FromCharacterCode[Range[32, 126]]

Out[12]=
  !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
 ~
```

Here are some ISO Latin-1 letters.

```
In[13]:= FromCharacterCode[Range[192, 255]]

Out[13]= ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ
```

Here are some special characters used in mathematical notation. The empty boxes correspond to characters not available in the current font.

```
In[14]:= FromCharacterCode[Range[8704, 8750]]

Out[14]= ∀∁∂∃∄∅∆∇∈∉∊∋∌∍∎∏∐∑−∓∔∕∖∗∘∙√∛∜∝∞∟∠∡∢∣∤∥∦∧∨∩∪∫∬∭∮
```

Here are a few Japanese characters.

```
In[15]:= FromCharacterCode[Range[30 000, 30 030]]

Out[15]= 田由甲申甴电甶男甸甹町画甼甽甾甿畀畁畂畃畄畅畆畇畈畉畊畋界畍畎
```

Raw Character Encodings

Mathematica always allows you to refer to special characters by using names such as `\[Alpha]` or explicit hexadecimal codes such as `\:03b1`. And when *Mathematica* writes out files, it by default uses these names or hexadecimal codes.

But sometimes you may find it convenient to use raw encodings for at least some special characters. What this means is that rather than representing special characters by names or explicit hexadecimal codes, you instead represent them by raw bit patterns appropriate for a particular computer system or particular font.

`$CharacterEncoding=None`	use printable ASCII names for all special characters
`$CharacterEncoding="`*name*`"`	use the raw character encoding specified by *name*
`$SystemCharacterEncoding`	the default raw character encoding for your particular computer system

Setting up raw character encodings.

When you press a key or combination of keys on your keyboard, the operating system of your computer sends a certain bit pattern to *Mathematica*. How this bit pattern is interpreted as a character within *Mathematica* will depend on the character encoding that has been set up.

The notebook front end for *Mathematica* typically takes care of setting up the appropriate character encoding automatically for whatever font you are using. But if you use *Mathematica* with a text-based interface or via files or pipes, then you may need to set `$CharacterEncoding` explicitly.

By specifying an appropriate value for `$CharacterEncoding` you will typically be able to get *Mathematica* to handle raw text generated by whatever language-specific text editor or operating system you use.

You should realize, however, that while the standard representation of special characters used in *Mathematica* is completely portable across different computer systems, any representation that involves raw character encodings will inevitably not be.

`"PrintableASCII"`	printable ASCII characters only (default)
`"ASCII"`	all ASCII including control characters
`"ISOLatin1"`	characters for common western European languages
`"ISOLatin2"`	characters for central and eastern European languages
`"ISOLatin3"`	characters for additional European languages (e.g. Catalan, Turkish)
`"ISOLatin4"`	characters for other additional European languages (e.g. Estonian, Lappish)
`"ISOLatinCyrillic"`	English and Cyrillic characters
`"AdobeStandard"`	Adobe standard PostScript font encoding
`"MacintoshRoman"`	Macintosh roman font encoding
`"WindowsANSI"`	Windows standard font encoding
`"Symbol"`	symbol font encoding
`"ZapfDingbats"`	Zapf dingbats font encoding
`"ShiftJIS"`	shift-JIS for Japanese (mixture of 8- and 16-bit)
`"EUC"`	extended Unix code for Japanese (mixture of 8- and 16-bit)
`"UTF8"`	Unicode transformation format encoding
`"Unicode"`	raw 16-bit Unicode bit patterns

Some raw character encodings supported by *Mathematica*.

Mathematica knows about various raw character encodings, appropriate for different computer systems and different languages. Copying of characters between the *Mathematica* notebook interface and user interface environment on your computer generally uses the native character encoding for that environment. *Mathematica* characters which are not included in the native encoding will be written out using standard *Mathematica* full names or hexadecimal codes.

The *Mathematica* kernel can use any character encoding you specify when it writes or reads text files. By default, `Put` and `PutAppend` produce an ASCII representation for reliable portability of *Mathematica* language files from one system to another.

This writes a string to the file `tmp`.

```
In[1]:= "a b c é α π ❦" >> tmp
```

Special characters are written out using full names or explicit hexadecimal codes.

```
In[2]:= Read["tmp", String]
Out[2]= "a b c \[EAcute] \[Alpha] \[Pi] \:2766"
```

Mathematica supports both 8- and 16-bit raw character encodings. In an encoding such as `"ISOLatin1"`, all characters are represented by bit patterns containing 8 bits. But in an encoding such as `"ShiftJIS"` some characters instead involve bit patterns containing 16 bits.

Most of the raw character encodings supported by *Mathematica* include basic ASCII as a subset. This means that even when you are using such encodings, you can still give ordinary *Mathematica* input in the usual way, and you can specify special characters using `\[` and `\:` sequences.

Some raw character encodings, however, do not include basic ASCII as a subset. An example is the `"Symbol"` encoding, in which the character codes normally used for `a` and `b` are instead used for α and β.

This gives the usual ASCII character codes for a few English letters.

```
In[3]:= ToCharacterCode["abcdefgh"]
Out[3]= {97, 98, 99, 100, 101, 102, 103, 104}
```

In the `"Symbol"` encoding, these character codes are used for Greek letters.

```
In[4]:= FromCharacterCode[%, "Symbol"]
Out[4]= αβχδεφγη
```

`ToCharacterCode["`*string*`"]`	generate codes for characters using the standard *Mathematica* encoding
`ToCharacterCode["`*string*`","`*encoding*`"]`	generate codes for characters using the specified encoding
`FromCharacterCode[{`n_1`,`n_2`,`…`}]`	generate characters from codes using the standard *Mathematica* encoding
`FromCharacterCode[{`n_1`,`n_2`,`…`},"`*encoding*`"]`	generate characters from codes using the specified encoding

Handling character codes with different encodings.

This gives the codes assigned to various characters by *Mathematica*.

```
In[5]:= ToCharacterCode["abcéπ"]
Out[5]= {97, 98, 99, 233, 960}
```

Here are the codes assigned to the same characters in the Macintosh roman encoding.

```
In[6]:= ToCharacterCode["abcéπ", "MacintoshRoman"]
Out[6]= {97, 98, 99, 142, 185}
```

Here are the codes in the Windows standard encoding. There is no code for `\[Pi]` in that encoding.

```
In[7]:= ToCharacterCode["abcéπ", "WindowsANSI"]
Out[7]= {97, 98, 99, 233, None}
```

The character codes used internally by *Mathematica* are based on Unicode. But externally *Mathematica* by default always uses plain ASCII sequences such as `\[`*Name*`]` or `\:`*nnnn* to refer to special characters. By telling it to use the raw `"Unicode"` character encoding, however, you can get *Mathematica* to read and write characters in raw 16-bit Unicode form.

Evaluation of Expressions

Principles of Evaluation

The fundamental operation that *Mathematica* performs is *evaluation*. Whenever you enter an expression, *Mathematica* evaluates the expression, then returns the result.

Evaluation in *Mathematica* works by applying a sequence of definitions. The definitions can either be ones you explicitly entered, or ones that are built into *Mathematica*.

Thus, for example, *Mathematica* evaluates the expression `6 + 7` using a built-in procedure for adding integers. Similarly, *Mathematica* evaluates the algebraic expression `x - 3 x + 1` using a built-in simplification procedure. If you had made the definition `x = 5`, then *Mathematica* would use this definition to reduce `x - 3 x + 1` to `-9`.

The two most central concepts in *Mathematica* are probably *expressions* and *evaluation*. "Expressions" discusses how all the different kinds of objects that *Mathematica* handles are represented in a uniform way using expressions. This tutorial describes how all the operations that *Mathematica* can perform can also be viewed in a uniform way as examples of evaluation.

Computation	$5+6\longrightarrow 11$
Simplification	$x-3x+1\longrightarrow 1-2x$
Execution	$x=5\longrightarrow 5$

Some interpretations of evaluation.

Mathematica is an *infinite evaluation* system. When you enter an expression, *Mathematica* will keep on using definitions it knows until it gets a result to which no definitions apply.

This defines `x1` in terms of `x2`, and then defines `x2`.

```
In[1]:= x1 = x2 + 2; x2 = 7
Out[1]= 7
```

If you ask for `x1`, *Mathematica* uses all the definitions it knows to give you a result.

```
In[2]:= x1
Out[2]= 9
```

Here is a recursive definition in which the factorial function is defined in terms of itself.

```
In[3]:= fac[1] = 1; fac[n_] := n fac[n - 1]
```

If you ask for `fac[10]`, *Mathematica* will keep on applying the definitions you have given until the result it gets no longer changes.

```
In[4]:= fac[10]
Out[4]= 3 628 800
```

When *Mathematica* has used all the definitions it knows, it gives whatever expression it has obtained as the result. Sometimes the result may be an object such as a number. But usually the result is an expression in which some objects are represented in a symbolic form.

Mathematica uses its built-in definitions for simplifying sums, but knows no definitions for `f[3]`, so leaves this in symbolic form.

```
In[5]:= f[3] + 4 f[3] + 1
Out[5]= 1 + 5 f[3]
```

Mathematica follows the principle of applying definitions until the result it gets no longer changes. This means that if you take the final result that *Mathematica* gives, and enter it as *Mathematica* input, you will get back the same result again. (There are some subtle cases discussed in "Controlling Infinite Evaluation" in which this does not occur.)

If you type in a result from *Mathematica*, you get back the same expression again.

```
In[6]:= 1 + 5 f[3]
Out[6]= 1 + 5 f[3]
```

At any given time, *Mathematica* can only use those definitions that it knows at that time. If you add more definitions later, however, *Mathematica* will be able to use these. The results you get from *Mathematica* may change in this case.

Here is a new definition for the function `f`.

```
In[7]:= f[x_] = x^2
```

Out[7]= x^2

With the new definition, the results you get can change.

```
In[8]:= 1 + 5 f[3]
Out[8]= 46
```

The simplest examples of evaluation involve using definitions such as `f[x_] = x^2` which transform one expression directly into another. But evaluation is also the process used to execute programs written in *Mathematica*. Thus, for example, if you have a procedure consisting of a sequence of *Mathematica* expressions, some perhaps representing conditionals and loops, the execution of this procedure corresponds to the evaluation of these expressions. Sometimes the evaluation process may involve evaluating a particular expression several times, as in a loop.

The expression `Print[zzzz]` is evaluated three times during the evaluation of the `Do` expression.

```
In[9]:= Do[Print[zzzz], {3}]

           zzzz

           zzzz

           zzzz
```

Reducing Expressions to Their Standard Form

The built-in functions in *Mathematica* operate in a wide variety of ways. But many of the mathematical functions share an important approach: they are set up so as to reduce classes of mathematical expressions to standard forms.

The built-in definitions for the `Plus` function, for example, are set up to write any sum of terms in a standard unparenthesized form. The associativity of addition means that expressions like `(a + b) + c`, `a + (b + c)` and `a + b + c` are all equivalent. But for many purposes it is convenient for all these forms to be reduced to the single standard form `a + b + c`. The built-in definitions for `Plus` are set up to do this.

Through the built-in definitions for `Plus`, this expression is reduced to a standard unparenthesized form.

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

Whenever *Mathematica* knows that a function is associative, it tries to remove parentheses (or nested invocations of the function) to get the function into a standard “flattened” form.

A function like addition is not only associative, but also commutative, which means that expressions like `a + c + b` and `a + b + c` with terms in different orders are equal. Once again, *Mathematica* tries to put all such expressions into a "standard" form. The standard form it chooses is the one in which all the terms are in a definite order, corresponding roughly to alphabetical order.

Mathematica sorts the terms in this sum into a standard order.

```
In[2]:= c + a + b
Out[2]= a + b + c
```

flat (associative)	$f[f[a,b],c]$ is equivalent to $f[a,b,c]$, etc.
orderless (commutative)	$f[b,a]$ is equivalent to $f[a,b]$, etc.

Two important properties that *Mathematica* uses in reducing certain functions to standard form.

There are several reasons to try to put expressions into standard forms. The most important is that if two expressions are really in standard form, it is obvious whether or not they are equal.

When the two sums are put into standard order, they are immediately seen to be equal, so that two `f`'s cancel, leaving the result `0`.

```
In[3]:= f[a + c + b] - f[c + a + b]
Out[3]= 0
```

You could imagine finding out whether `a + c + b` was equal to `c + a + b` by testing all possible orderings of each sum. It is clear that simply reducing both sums to standard form is a much more efficient procedure.

One might think that *Mathematica* should somehow automatically reduce *all* mathematical expressions to a single standard canonical form. With all but the simplest kinds of expressions, however, it is quite easy to see that you do not want the *same* standard form for all purposes.

For polynomials, for example, there are two obvious standard forms, which are good for different purposes. The first standard form for a polynomial is a simple sum of terms, as would be generated in *Mathematica* by applying the function `Expand`. This standard form is most appropriate if you need to add and subtract polynomials.

There is, however, another possible standard form that you can use for polynomials. By applying `Factor`, you can write any polynomial as a product of irreducible factors. This canonical form is useful if you want to do operations like division.

Expanded and factored forms are in a sense both equally good standard forms for polynomials. Which one you decide to use simply depends on what you want to use it for. As a result, *Mathematica* does not automatically put polynomials into one of these two forms. Instead, it gives you functions like `Expand` and `Factor` that allow you explicitly to put polynomials in whatever form you want.

Here is a list of two polynomials that are mathematically equal.

```
In[4]:= t = {x^2 - 1, (x + 1) (x - 1)}
```

Out[4]= $\{-1 + x^2, (-1 + x) (1 + x)\}$

You can write both of them in expanded form just by applying `Expand`. In this form, the equality of the polynomials is obvious.

```
In[5]:= Expand[t]
```

Out[5]= $\{-1 + x^2, -1 + x^2\}$

You can also see that the polynomials are equal by writing them both in factored form.

```
In[6]:= Factor[t]

Out[6]= {(-1 + x) (1 + x), (-1 + x) (1 + x)}
```

Although it is clear that you do not always want expressions reduced to the *same* standard form, you may wonder whether it is at least *possible* to reduce all expressions to *some* standard form.

There is a basic result in the mathematical theory of computation which shows that this is, in fact, not always possible. You cannot guarantee that any finite sequence of transformations will take any two arbitrarily chosen expressions to a standard form.

In a sense, this is not particularly surprising. If you could in fact reduce all mathematical expressions to a standard form, then it would be quite easy to tell whether any two expressions were equal. The fact that so many of the difficult problems of mathematics can be stated as questions about the equality of expressions suggests that this can in fact be difficult.

Attributes

Definitions such as `f[x_] = x^2` specify *values* for functions. Sometimes, however, you need to specify general properties of functions, without necessarily giving explicit values.

Mathematica provides a selection of *attributes* that you can use to specify various properties of functions. For example, you can use the attribute `Flat` to specify that a particular function is "flat", so that nested invocations are automatically flattened, and it behaves as if it were associative.

This assigns the attribute `Flat` to the function `f`.

```
In[1]:= SetAttributes[f, Flat]
```

Now `f` behaves as a flat, or associative, function, so that nested invocations are automatically flattened.

```
In[2]:= f[f[a, b], c]
Out[2]= f[a, b, c]
```

Attributes like `Flat` can affect not only evaluation, but also operations such as pattern matching. If you give definitions or transformation rules for a function, you must be sure to have specified the attributes of the function first.

Here is a definition for the flat function `f`.

```
In[3]:= f[x_, x_] := f[x]
```

Because `f` is flat, the definition is automatically applied to every subsequence of arguments.

```
In[4]:= f[a, a, a, b, b, b, c, c]
Out[4]= f[a, b, c]
```

`Attributes[`f`]`	give the attributes of f
`Attributes[`f`]={`$attr_1$`,`$attr_2$`,…}`	set the attributes of f
`Attributes[`f`]={}`	set f to have no attributes
`SetAttributes[`f`,`$attr$`]`	add $attr$ to the attributes of f
`ClearAttributes[`f`,`$attr$`]`	remove $attr$ from the attributes of f

Manipulating attributes of symbols.

This shows the attributes assigned to `f`.

```
In[5]:= Attributes[f]
Out[5]= {Flat}
```

This removes the attributes assigned to `f`.

```
In[6]:= Attributes[f] = {}
Out[6]= {}
```

`Orderless`	orderless, commutative function (arguments are sorted into standard order)
`Flat`	flat, associative function (arguments are "flattened out")
`OneIdentity`	$f[f[a]]$, etc. are equivalent to a for pattern matching
`Listable`	f is automatically "threaded" over lists that appear as arguments (e.g., $f[\{a, b\}]$ becomes $\{f[a], f[b]\}$)
`Constant`	all derivatives of f are zero
`NumericFunction`	f is assumed to have a numerical value when its arguments are numeric quantities
`Protected`	values of f cannot be changed
`Locked`	attributes of f cannot be changed
`ReadProtected`	values of f cannot be read
`HoldFirst`	the first argument of f is not evaluated
`HoldRest`	all but the first argument of f is not evaluated
`HoldAll`	none of the arguments of f are evaluated
`HoldAllComplete`	the arguments of f are treated as completely inert
`NHoldFirst`	the first argument of f is not affected by `N`
`NHoldRest`	all but the first argument of f is not affected by `N`
`NHoldAll`	none of the arguments of f are affected by `N`
`SequenceHold`	`Sequence` objects appearing in the arguments of f are not flattened out
`Temporary`	f is a local variable, removed when no longer used
`Stub`	`Needs` is automatically called if f is ever explicitly input

The complete list of attributes for symbols in *Mathematica*.

Here are the attributes for the built-in function `Plus`.

```
In[7]:= Attributes[Plus]
Out[7]= {Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

An important attribute assigned to built-in mathematical functions in *Mathematica* is the attribute `Listable`. This attribute specifies that a function should automatically be distributed or "threaded" over lists that appear as its arguments. This means that the function effectively gets applied separately to each element in any lists that appear as its arguments.

The built-in `Log` function is `Listable`.

```
In[8]:= Log[{5, 8, 11}]
Out[8]= {Log[5], Log[8], Log[11]}
```

This defines the function `p` to be listable.

```
In[9]:= SetAttributes[p, Listable]
```

Now `p` is automatically threaded over lists that appear as its arguments.

```
In[10]:= p[{a, b, c}, d]
Out[10]= {p[a, d], p[b, d], p[c, d]}
```

Many of the attributes you can assign to functions in *Mathematica* directly affect the evaluation of those functions. Some attributes, however, affect only other aspects of the treatment of functions. For example, the attribute `OneIdentity` affects only pattern matching, as discussed in "Flat and Orderless Functions". Similarly, the attribute `Constant` is only relevant in differentiation, and operations that rely on differentiation.

The `Protected` attribute affects assignments. *Mathematica* does not allow you to make any definition associated with a symbol that carries this attribute. The functions `Protect` and `Unprotect` discussed in "Modifying Built-in Functions" can be used as alternatives to `SetAttributes` and `ClearAttributes` to set and clear this attribute. As discussed in "Modifying Built-in Functions" most built-in *Mathematica* objects are initially protected so that you do not make definitions for them by mistake.

Here is a definition for the function `g`.

```
In[11]:= g[x_] = x + 1
Out[11]= 1 + x
```

This sets the `Protected` attribute for `g`.

```
In[12]:= Protect[g]
Out[12]= {g}
```

Now you cannot modify the definition of `g`.

```
In[13]:= g[x_] = x

    Set::write : Tag g in g[x_] is Protected. ≫

Out[13]= x
```

You can usually see the definitions you have made for a particular symbol by typing `?` f, or by using a variety of built-in *Mathematica* functions. However, if you set the attribute `ReadProtected`, *Mathematica* will not allow you to look at the definition of a particular symbol. It will nevertheless continue to use the definitions in performing evaluation.

Although you cannot modify it, you can still look at the definition of `g`.

```
In[14]:= ? g

    Global`g

    Attributes[g] = {Protected}

    g[x_] = 1 + x
```

This sets the `ReadProtected` attribute for `g`.

```
In[15]:= SetAttributes[g, ReadProtected]
```

Now you can no longer read the definition of `g`.

```
In[16]:= ? g

    Global`g

    Attributes[g] = {Protected, ReadProtected}
```

Functions like `SetAttributes` and `ClearAttributes` usually allow you to modify the attributes of a symbol in any way. However, if you once set the `Locked` attribute on a symbol, then *Mathematica* will not allow you to modify the attributes of that symbol for the remainder of your *Mathematica* session. Using the `Locked` attribute in addition to `Protected` or `ReadProtected`, you can arrange for it to be impossible for users to modify or read definitions.

`Clear[`*f*`]`	remove values for *f*, but not attributes
`ClearAll[`*f*`]`	remove both values and attributes of *f*

Clearing values and attributes.

This clears values and attributes of `p` which was given attribute `Listable` above.

```
In[17]:= ClearAll[p]
```

Now `p` is no longer listable.

```
In[18]:= p[{a, b, c}, d]
Out[18]= p[{a, b, c}, d]
```

By defining attributes for a function you specify properties that *Mathematica* should assume whenever that function appears. Often, however, you want to assume the properties only in a particular instance. In such cases, you will be better off not to use attributes, but instead to call a particular function to implement the transformation associated with the attributes.

By explicitly calling `Thread`, you can implement the transformation that would be done automatically if `p` were listable.

```
In[19]:= Thread[p[{a, b, c}, d]]
Out[19]= {p[a, d], p[b, d], p[c, d]}
```

`Orderless`	`Sort[`*f*`[`*args*`]]`
`Flat`	`Flatten[`*f*`[`*args*`]]`
`Listable`	`Thread[`*f*`[`*args*`]]`
`Constant`	`Dt[`*expr*`,Constants->`*f*`]`

Functions that perform transformations associated with some attributes.

Attributes in *Mathematica* can only be permanently defined for single symbols. However, *Mathematica* also allows you to set up pure functions which behave as if they carry attributes.

`Function[`*vars*`,`*body*`,{`$attr_1$`,…}]`	a pure function with attributes $attr_1$, …

Pure functions with attributes.

This pure function applies `p` to the whole list.

```
In[20]:= Function[{x}, p[x]][{a, b, c}]
Out[20]= p[{a, b, c}]
```

By adding the attribute `Listable`, the function gets distributed over the elements of the list before applying `p`.

```
In[21]:= Function[{x}, p[x], {Listable}][{a, b, c}]
Out[21]= {p[a], p[b], p[c]}
```

The Standard Evaluation Procedure

Here we describe the standard procedure used by *Mathematica* to evaluate expressions. This procedure is the one followed for most kinds of expression. There are however some kinds of expressions, such as those used to represent *Mathematica* programs and control structures, which are evaluated in a nonstandard way.

In the standard evaluation procedure, *Mathematica* first evaluates the head of an expression, and then evaluates each element of the expressions. These elements are in general themselves expressions, to which the same evaluation procedure is recursively applied.

The three `Print` functions are evaluated in turn, each printing its argument, then returning the value `Null`.

```
In[1]:= {Print[1], Print[2], Print[3]}

   1

   2

   3

Out[1]= {Null, Null, Null}
```

This assigns the symbol `ps` to be `Plus`.

```
In[2]:= ps = Plus
Out[2]= Plus
```

The head `ps` is evaluated first, so this expression behaves just like a sum of terms.

```
In[3]:= ps[ps[a, b], c]
Out[3]= a + b + c
```

As soon as *Mathematica* has evaluated the head of an expression, it sees whether the head is a symbol that has attributes. If the symbol has the attributes `Orderless`, `Flat` or `Listable`, then immediately after evaluating the elements of the expression *Mathematica* performs the transformations associated with these attributes.

The next step in the standard evaluation procedure is to use definitions that *Mathematica* knows for the expression it is evaluating. *Mathematica* first tries to use definitions that you have made, and if there are none that apply, it tries built-in definitions.

If *Mathematica* finds a definition that applies, it performs the corresponding transformation on the expression. The result is another expression, which must then in turn be evaluated according to the standard evaluation procedure.

- Evaluate the head of the expression.
- Evaluate each element in turn.
- Apply transformations associated with the attributes `Orderless`, `Listable` and `Flat`.
- Apply any definitions that you have given.
- Apply any built-in definitions.
- Evaluate the result.

The standard evaluation procedure.

As discussed in "Principles of Evaluation", *Mathematica* follows the principle that each expression is evaluated until no further definitions apply. This means that *Mathematica* must continue re-evaluating results until it gets an expression which remains unchanged through the evaluation procedure.

Here is an example that shows how the standard evaluation procedure works on a simple expression. We assume that `a = 7`.

`2 a x+a^2+1`	here is the original expression
`Plus[Times[2,a,x],Power[a,2],1]`	this is the internal form
`Times[2,a,x]`	this is evaluated first
`Times[2,7,x]`	a is evaluated to give 7
`Times[14,x]`	built-in definitions for `Times` give this result
`Power[a,2]`	this is evaluated next
`Power[7,2]`	here is the result after evaluating `a`
`49`	built-in definitions for `Power` give this result
`Plus[Times[14,x],49,1]`	here is the result after the arguments of `Plus` have been evaluated
`Plus[50,Times[14,x]]`	built-in definitions for `Plus` give this result
`50+14 x`	the result is printed like this

A simple example of evaluation in *Mathematica*.

Mathematica provides various ways to “trace” the evaluation process, as discussed in "Tracing Evaluation". The function `Trace[`*expr*`]` gives a nested list showing each subexpression generated during evaluation. (Note that the standard evaluation traverses the expression tree in a depth-first way, so that the smallest subparts of the expression appear first in the results of `Trace`.)

First set a to 7.

```
In[4]:= a = 7
Out[4]= 7
```

This gives a nested list of all the subexpressions generated during the evaluation of the expression.

```
In[5]:= Trace[2 a x + a^2 + 1]
Out[5]= {{{a, 7}, 2×7 x, 14 x}, {{a, 7}, 7^2, 49}, 14 x + 49 + 1, 50 + 14 x}
```

The order in which *Mathematica* applies different kinds of definitions is important. The fact that *Mathematica* applies definitions you have given before it applies built-in definitions means that you can give definitions which override the built-in ones, as discussed in "Modifying Built-in Functions".

This expression is evaluated using the built-in definition for `ArcSin`.

```
In[6]:= ArcSin[1]
```

```
         π
Out[6]=  —
         2
```

You can give your own definitions for `ArcSin`. You need to remove the protection attribute first.

```
In[7]:= Unprotect[ArcSin]; ArcSin[1] = 5 Pi / 2;
```

Your definition is used before the one that is built in.

```
In[8]:= ArcSin[1]
```

```
         5 π
Out[8]=  ———
          2
```

As discussed in "Associating Definitions with Different Symbols", you can associate definitions with symbols either as upvalues or downvalues. *Mathematica* always tries upvalue definitions before downvalue ones.

If you have an expression like $f[g[x]]$, there are in general two sets of definitions that could apply: downvalues associated with f, and upvalues associated with g. *Mathematica* tries the definitions associated with g before those associated with f.

This ordering follows the general strategy of trying specific definitions before more general ones. By applying upvalues associated with arguments before applying downvalues associated with a function, *Mathematica* allows you to make definitions for special arguments which override the general definitions for the function with any arguments.

This defines a rule for `f[g[x_]]`, to be associated with `f`.

```
In[9]:= f /: f[g[x_]] := frule[x]
```

This defines a rule for `f[g[x_]]`, to be associated with `g`.

```
In[10]:= g /: f[g[x_]] := grule[x]
```

The rule associated with `g` is tried before the rule associated with `f`.

```
In[11]:= f[g[2]]
Out[11]= grule[2]
```

If you remove rules associated with `g`, the rule associated with `f` is used.

```
In[12]:= Clear[g]; f[g[1]]
Out[12]= frule[1]
```

▪ Definitions associated with g are applied before definitions associated with f in the expression $f[g[x]]$.

The order in which definitions are applied.

Most functions such as `Plus` that are built into *Mathematica* have downvalues. There are, however, some objects in *Mathematica* which have built-in upvalues. For example, `SeriesData` objects, which represent power series, have built-in upvalues with respect to various mathematical operations.

For an expression like $f[g[x]]$, the complete sequence of definitions that are tried in the standard evaluation procedure is:

- Definitions you have given associated with g;
- Built-in definitions associated with g;
- Definitions you have given associated with f;
- Built-in definitions associated with f.

The fact that upvalues are used before downvalues is important in many situations. In a typical case, you might want to define an operation such as composition. If you give upvalues for various objects with respect to composition, these upvalues will be used whenever such objects appear. However, you can also give a general procedure for composition, to be used if no special objects are present. You can give this procedure as a downvalue for composition. Since downvalues are tried after upvalues, the general procedure will be used only if no objects with upvalues are present.

Here is a definition associated with `q` for composition of "`q` objects".

```
In[13]:= q /: comp[q[x_], q[y_]] := qcomp[x, y]
```

Here is a general rule for composition, associated with `comp`.

```
In[14]:= comp[f_[x_], f_[y_]] := gencomp[f, x, y]
```

If you compose two `q` objects, the rule associated with `q` is used.

```
In[15]:= comp[q[1], q[2]]
Out[15]= qcomp[1, 2]
```

If you compose `r` objects, the general rule associated with `comp` is used.

```
In[16]:= comp[r[1], r[2]]
Out[16]= gencomp[r, 1, 2]
```

In general, there can be several objects that have upvalues in a particular expression. *Mathematica* first looks at the head of the expression, and tries any upvalues associated with it. Then it successively looks at each element of the expression, trying any upvalues that exist. *Mathematica* performs this procedure first for upvalues that you have explicitly defined, and then for upvalues that are built-in. The procedure means that in a sequence of elements, upvalues associated with earlier elements take precedence over those associated with later elements.

This defines an upvalue for `p` with respect to `c`.

```
In[17]:= p /: c[l___, p[x_], r___] := cp[x, {l, r}]
```

This defines an upvalue for `q`.

```
In[18]:= q /: c[l___, q[x_], r___] := cq[x, {l, r}]
```

Which upvalue is used depends on which occurs first in the sequence of arguments to `c`.

```
In[19]:= {c[p[1], q[2]], c[q[1], p[2]]}
Out[19]= {cp[1, {q[2]}], cq[1, {p[2]}]}
```

Non-Standard Evaluation

While most built-in *Mathematica* functions follow the standard evaluation procedure, some important ones do not. For example, most of the *Mathematica* functions associated with the construction and execution of programs use non-standard evaluation procedures. In typical cases, the functions either never evaluate some of their arguments, or do so in a special way under their own control.

x=*y*	do not evaluate the left-hand side
`If`[*p*,*a*,*b*]	evaluate *a* if *p* is `True`, and *b* if it is `False`
`Do`[*expr*,{*n*}]	evaluate *expr* *n* times
`Plot`[*f*,{*x*,…}]	evaluate *f* with a sequence of numerical values for *x*
`Function`[{*x*},*body*]	do not evaluate until the function is applied

Some functions that use non-standard evaluation procedures.

When you give a definition such as `a = 1`, *Mathematica* does not evaluate the `a` that appears on the left-hand side. You can see that there would be trouble if the `a` was evaluated. The reason is that if you had previously set `a = 7`, then evaluating `a` in the definition `a = 1` would put the definition into the nonsensical form `7 = 1`.

In the standard evaluation procedure, each argument of a function is evaluated in turn. This is prevented by setting the attributes `HoldFirst`, `HoldRest` and `HoldAll`. These attributes make *Mathematica* "hold" particular arguments in an unevaluated form.

`HoldFirst`	do not evaluate the first argument
`HoldRest`	evaluate only the first argument
`HoldAll`	evaluate none of the arguments

Attributes for holding function arguments in unevaluated form.

With the standard evaluation procedure, all arguments to a function are evaluated.

```
In[1]:= f[1 + 1, 2 + 4]
Out[1]= f[2, 6]
```

This assigns the attribute `HoldFirst` to `h`.

```
In[2]:= SetAttributes[h, HoldFirst]
```

The first argument to `h` is now held in an unevaluated form.

```
In[3]:= h[1 + 1, 2 + 4]
Out[3]= h[1 + 1, 6]
```

When you use the first argument to `h` like this, it will get evaluated.

```
In[4]:= h[1 + 1, 2 + 4] /. h[x_, y_] -> x^y
Out[4]= 64
```

Built-in functions like `Set` carry attributes such as `HoldFirst`.

```
In[5]:=  Attributes[Set]
Out[5]=  {HoldFirst, Protected, SequenceHold}
```

Even though a function may have attributes which specify that it should hold certain arguments unevaluated, you can always explicitly tell *Mathematica* to evaluate those arguments by giving the arguments in the form `Evaluate[`*arg*`]`.

`Evaluate` effectively overrides the `HoldFirst` attribute, and causes the first argument to be evaluated.

```
In[6]:=  h[Evaluate[1 + 1], 2 + 4]
Out[6]=  h[2, 6]
```

f`[Evaluate[`*arg*`]]`	evaluate *arg* immediately, even though attributes of *f* may specify that it should be held

Forcing the evaluation of function arguments.

By holding its arguments, a function can control when those arguments are evaluated. By using `Evaluate`, you can force the arguments to be evaluated immediately, rather than being evaluated under the control of the function. This capability is useful in a number of circumstances.

The *Mathematica* `Set` function holds its first argument, so the symbol `a` is not evaluated in this case.

```
In[7]:=  a = b
Out[7]=  b
```

You can make `Set` evaluate its first argument using `Evaluate`. In this case, the result is the object which is the *value* of `a`, namely `b` is set to `6`.

```
In[8]:=  Evaluate[a] = 6
Out[8]=  6
```

`b` has now been set to `6`.

```
In[9]:=  b
Out[9]=  6
```

In most cases, you want all expressions you give to *Mathematica* to be evaluated. Sometimes, however, you may want to prevent the evaluation of certain expressions. For example, if you want to manipulate pieces of a *Mathematica* program symbolically, then you must prevent those pieces from being evaluated while you are manipulating them.

You can use the functions `Hold` and `HoldForm` to keep expressions unevaluated. These functions work simply by carrying the attribute `HoldAll`, which prevents their arguments from being evaluated. The functions provide "wrappers" inside which expressions remain unevaluated.

The difference between `Hold[`*expr*`]` and `HoldForm[`*expr*`]` is that in standard *Mathematica* output format, `Hold` is printed explicitly, while `HoldForm` is not. If you look at the full internal *Mathematica* form, you can however see both functions.

`Hold` maintains expressions in an unevaluated form.

```
In[10]:= Hold[1 + 1]
Out[10]= Hold[1 + 1]
```

`HoldForm` also keeps expressions unevaluated, but is invisible in standard *Mathematica* output format.

```
In[11]:= HoldForm[1 + 1]
Out[11]= 1 + 1
```

`HoldForm` is still present internally.

```
In[12]:= FullForm[%]
Out[12]//FullForm= HoldForm[Plus[1, 1]]
```

The function `ReleaseHold` removes `Hold` and `HoldForm`, so the expressions they contain get evaluated.

```
In[13]:= ReleaseHold[%]
Out[13]= 2
```

`Hold[`*expr*`]`	keep *expr* unevaluated
`HoldComplete[`*expr*`]`	keep *expr* unevaluated and prevent upvalues associated with *expr* from being used
`HoldForm[`*expr*`]`	keep *expr* unevaluated, and print without `HoldForm`
`ReleaseHold[`*expr*`]`	remove `Hold` and `HoldForm` in *expr*
`Extract[`*expr*`,`*index*`,Hold]`	get a part of *expr*, wrapping it with `Hold` to prevent evaluation

Functions for handling unevaluated expressions.

Parts of expressions are usually evaluated as soon as you extract them.

```
In[14]:= Extract[Hold[1 + 1, 2 + 3], 2]
Out[14]= 5
```

This extracts a part and immediately wraps it with `Hold`, so it does not get evaluated.

```
In[15]:= Extract[Hold[1 + 1, 2 + 3], 2, Hold]
Out[15]= Hold[2 + 3]
```

f`[`…`,Unevaluated[`*expr*`],`…`]`	give *expr* unevaluated as an argument to *f*

Temporary prevention of argument evaluation.

`1 + 1` evaluates to `2`, and `Length[2]` gives `0`.

```
In[16]:= Length[1 + 1]
Out[16]= 0
```

This gives the unevaluated form `1 + 1` as the argument of `Length`.

```
In[17]:= Length[Unevaluated[1 + 1]]
Out[17]= 2
```

`Unevaluated[`*expr*`]` effectively works by temporarily giving a function an attribute like `HoldFirst`, and then supplying *expr* as an argument to the function.

`SequenceHold`	do not flatten out `Sequence` objects that appear as arguments
`HoldAllComplete`	treat all arguments as completely inert

Attributes for preventing other aspects of evaluation.

By setting the attribute `HoldAll`, you can prevent *Mathematica* from evaluating the arguments of a function. But even with this attribute set, *Mathematica* will still do some transformations on the arguments. By setting `SequenceHold` you can prevent it from flattening out `Sequence` objects that appear in the arguments. And by setting `HoldAllComplete` you can also inhibit the stripping of `Unevaluated`, and prevent *Mathematica* from using any upvalues it finds associated with the arguments.

Evaluation in Patterns, Rules and Definitions

There are a number of important interactions in *Mathematica* between evaluation and pattern matching. The first observation is that pattern matching is usually done on expressions that have already been at least partly evaluated. As a result, it is usually appropriate that the patterns to which these expressions are matched should themselves be evaluated.

The fact that the pattern is evaluated means that it matches the expression given.

```
In[1]:= f[k^2] /. f[x_^(1+1)] -> p[x]
Out[1]= p[k]
```

The right-hand side of the `/;` condition is not evaluated until it is used during pattern matching.

```
In[2]:= f[{a, b}] /. f[list_ /; Length[list] > 1] -> list^2
```

Out[2]= $\{a^2, b^2\}$

There are some cases, however, where you may want to keep all or part of a pattern unevaluated. You can do this by wrapping the parts you do not want to evaluate with `HoldPattern`. In general, whenever `HoldPattern[`*patt*`]` appears within a pattern, this form is taken to be equivalent to *patt* for the purpose of pattern matching, but the expression *patt* is maintained unevaluated.

`HoldPattern[`*patt*`]`	equivalent to *patt* for pattern matching, with *patt* kept unevaluated

Preventing evaluation in patterns.

One application for `HoldPattern` is in specifying patterns which can apply to unevaluated expressions, or expressions held in an unevaluated form.

HoldPattern keeps the 1 + 1 from being evaluated, and allows it to match the 1 + 1 on the left-hand side of the /. operator.

```
In[3]:= Hold[u[1 + 1]] /. HoldPattern[1 + 1] -> x
Out[3]= Hold[u[x]]
```

Notice that while functions like Hold prevent evaluation of expressions, they do not affect the manipulation of parts of those expressions with /. and other operators.

This defines values for r whenever its argument is not an atomic object.

```
In[4]:= r[x_] := x^2 /; ! AtomQ[x]
```

According to the definition, expressions like r[3] are left unchanged.

```
In[5]:= r[3]
Out[5]= r[3]
```

However, the pattern r[x_] is transformed according to the definition for r.

```
In[6]:= r[x_]
Out[6]= x_^2
```

You need to wrap HoldPattern around r[x_] to prevent it from being evaluated.

```
In[7]:= {r[3], r[5]} /. HoldPattern[r[x_]] -> x
Out[7]= {3, 5}
```

As illustrated above, the left-hand sides of transformation rules such as *lhs* -> *rhs* are usually evaluated immediately, since the rules are usually applied to expressions which have already been evaluated. The right-hand side of *lhs* -> *rhs* is also evaluated immediately. With the delayed rule *lhs* :> *rhs*, however, the expression *rhs* is not evaluated.

The right-hand side is evaluated immediately in -> but not :> rules.

```
In[8]:= {{x -> 1 + 1}, {x :> 1 + 1}}
Out[8]= {{x → 2}, {x :→ 1 + 1}}
```

Here are the results of applying the rules. The right-hand side of the :> rule gets inserted inside the Hold without evaluation.

```
In[9]:= {x^2, Hold[x]} /. %
Out[9]= {{4, Hold[2]}, {4, Hold[1 + 1]}}
```

lhs->*rhs*	evaluate both *lhs* and *rhs*
lhs:>*rhs*	evaluate *lhs* but not *rhs*

Evaluation in transformation rules.

While the left-hand sides of transformation rules are usually evaluated, the left-hand sides of definitions are usually not. The reason for the difference is as follows. Transformation rules are typically applied using `/.` to expressions that have already been evaluated. Definitions, however, are used during the evaluation of expressions, and are applied to expressions that have not yet been completely evaluated. To work on such expressions, the left-hand sides of definitions must be maintained in a form that is at least partially unevaluated.

Definitions for symbols are the simplest case. As discussed in "Non-Standard Evaluation", a symbol on the left-hand side of a definition such as *x* = *value* is not evaluated. If *x* had previously been assigned a value *y*, then if the left-hand side of *x* = *value* were evaluated, it would turn into the quite unrelated definition *y* = *value*.

Here is a definition. The symbol on the left-hand side is not evaluated.

```
In[10]:= k = w[3]
Out[10]= w[3]
```

This redefines the symbol.

```
In[11]:= k = w[4]
Out[11]= w[4]
```

If you evaluate the left-hand side, then you define not the symbol `k`, but the *value* `w[4]` of the symbol `k`.

```
In[12]:= Evaluate[k] = w[5]
Out[12]= w[5]
```

Now `w[4]` has value `w[5]`.

```
In[13]:= w[4]
Out[13]= w[5]
```

Although individual symbols that appear on the left-hand sides of definitions are not evaluated, more complicated expressions are partially evaluated. In an expression such as *f*[*args*] on the left-hand side of a definition, the *args* are evaluated.

The 1 + 1 is evaluated, so that a value is defined for g[2].

```
In[14]:= g[1 + 1] = 5
Out[14]= 5
```

This shows the value defined for g.

```
In[15]:= ?g
```

```
Global`g

g[2] = 5
```

You can see why the arguments of a function that appears on the left-hand side of a definition must be evaluated by considering how the definition is used during the evaluation of an expression. As discussed in "Principles of Evaluation", when *Mathematica* evaluates a function, it first evaluates each of the arguments, then tries to find definitions for the function. As a result, by the time *Mathematica* applies any definition you have given for a function, the arguments of the function must already have been evaluated. An exception to this occurs when the function in question has attributes which specify that it should hold some of its arguments unevaluated.

symbol=*value*	*symbol* is not evaluated; *value* is evaluated
symbol:=*value*	neither *symbol* nor *value* is evaluated
f[*args*]=*value*	*args* are evaluated; left-hand side as a whole is not
f[HoldPattern[*arg*]]=*value*	*f*[*arg*] is assigned, without evaluating *arg*
Evaluate[*lhs*]=*value*	left-hand side is evaluated completely

Evaluation in definitions.

While in most cases it is appropriate for the arguments of a function that appears on the left-hand side of a definition to be evaluated, there are some situations in which you do not want this to happen. In such cases, you can wrap `HoldPattern` around the parts that you do not want to be evaluated.

Evaluation in Iteration Functions

The built-in *Mathematica* iteration functions such as `Table` and `Sum` evaluate their arguments in a slightly special way.

When evaluating an expression like `Table[`f`, {`i`, `i_{max}`}]`, the first step, as discussed in "Blocks and Local Values", is to make the value of i local. Next, the limit i_{max} in the iterator specification is evaluated. The expression f is maintained in an unevaluated form, but is repeatedly evaluated as a succession of values are assigned to i. When this is finished, the global value of i is restored.

The function `RandomReal[]` is evaluated four separate times here, so four different pseudorandom numbers are generated.

```
In[1]:= Table[RandomReal[], {4}]

Out[1]= {0.300949, 0.450179, 0.831238, 0.161379}
```

This evaluates `RandomReal[]` before feeding it to `Table`. The result is a list of four identical numbers.

```
In[2]:= Table[Evaluate[RandomReal[]], {4}]

Out[2]= {0.653098, 0.653098, 0.653098, 0.653098}
```

In most cases, it is convenient for the function f in an expression like `Table[`f`, {`i`, `i_{max}`}]` to be maintained in an unevaluated form until specific values have been assigned to i. This is true in particular if a complete symbolic form for f valid for any i cannot be found.

This defines `fac` to give the factorial when it has an integer argument, and to give `NaN` (standing for "Not a Number") otherwise.

```
In[3]:= fac[n_Integer] := n!; fac[x_] := NaN
```

In this form, `fac[i]` is not evaluated until an explicit integer value has been assigned to `i`.

```
In[4]:= Table[fac[i], {i, 5}]

Out[4]= {1, 2, 6, 24, 120}
```

Using `Evaluate` forces `fac[i]` to be evaluated with `i` left as a symbolic object.

```
In[5]:= Table[Evaluate[fac[i]], {i, 5}]

Out[5]= {NaN, NaN, NaN, NaN, NaN}
```

In cases where a complete symbolic form for f with arbitrary i in expressions such as `Table[`f`, {`i`, `i_{max}`}]` *can* be found, it is often more efficient to compute this form first, and then feed it to `Table`. You can do this using `Table[Evaluate[`f`], {`i`, `i_{max}`}]`.

The `Sum` in this case is evaluated separately for each value of `i`.

```
In[6]:= Table[Sum[i^k, {k, 4}], {i, 8}]
Out[6]= {4, 30, 120, 340, 780, 1554, 2800, 4680}
```

It is however possible to get a symbolic formula for the sum, valid for any value of `i`.

```
In[7]:= Sum[i^k, {k, 4}]
```

Out[7]= $i + i^2 + i^3 + i^4$

By inserting `Evaluate`, you tell *Mathematica* first to evaluate the sum symbolically, then to iterate over `i`.

```
In[8]:= Table[Evaluate[Sum[i^k, {k, 4}]], {i, 8}]
Out[8]= {4, 30, 120, 340, 780, 1554, 2800, 4680}
```

`Table[`f`,{`i`,`i_{max}`}]`	keep f unevaluated until specific values are assigned to i
`Table[Evaluate[`f`],{`i`,`i_{max}`}]`	evaluate f first with i left symbolic

Evaluation in iteration functions.

Conditionals

Mathematica provides various ways to set up *conditionals*, which specify that particular expressions should be evaluated only if certain conditions hold.

lhs`:=`*rhs*`/;`*test*	use the definition only if *test* evaluates to `True`
`If[`*test*`,`*then*`,`*else*`]`	evaluate *then* if *test* is `True`, and *else* if it is `False`
`Which[`$test_1$`,`$value_1$`,`$test_2$`,…]`	evaluate the $test_i$ in turn, giving the value associated with the first one that is `True`
`Switch[`*expr*`,`$form_1$`,`$value_1$`,`$form_2$`,…]`	compare *expr* with each of the $form_i$, giving the value associated with the first form it matches
`Switch[`*expr*`,`$form_1$`,` $value_1$`,`$form_2$`,…,_,`*def*`]`	use *def* as a default value
`Piecewise[{{`$value_1$`,`$test_1$`},…},`*def*`]`	give the value corresponding to the first $test_i$ which yields `True`

Conditional constructs.

The test gives `False`, so the "*else*" expression `y` is returned.

```
In[1]:= If[7 > 8, x, y]
Out[1]= y
```

Only the "*else*" expression is evaluated in this case.

```
In[2]:= If[7 > 8, Print[x], Print[y]]
        y
```

When you write programs in *Mathematica*, you will often have a choice between making a single definition whose right-hand side involves several branches controlled by `If` functions, or making several definitions, each controlled by an appropriate `/;` condition. By using several definitions, you can often produce programs that are both clearer, and easier to modify.

This defines a step function, with value 1 for $x > 0$, and -1 otherwise.

```
In[3]:= f[x_] := If[x > 0, 1, -1]
```

This defines the positive part of the step function using a `/;` condition.

```
In[4]:= g[x_] := 1 /; x > 0
```

Here is the negative part of the step function.

```
In[5]:= g[x_] := -1 /; x <= 0
```

This shows the complete definition using `/;` conditions.

```
In[6]:= ?g

        Global`g

        g[x_] := 1 /; x > 0

        g[x_] := -1 /; x ≤ 0
```

The function `If` provides a way to choose between two alternatives. Often, however, there will be more than two alternatives. One way to handle this is to use a nested set of `If` functions. Usually, however, it is instead better to use functions like `Which` and `Switch`.

This defines a function with three regions. Using `True` as the third test makes this the default case.

```
In[7]:= h[x_] := Which[x < 0, x^2, x > 5, x^3, True, 0]
```

This uses the first case in the `Which`.

```
In[8]:=  h[-5]
Out[8]=  25
```

This uses the third case.

```
In[9]:=  h[2]
Out[9]=  0
```

This defines a function that depends on the values of its argument modulo 3.

```
In[10]:=  r[x_] := Switch[Mod[x, 3], 0, a, 1, b, 2, c]
```

`Mod[7, 3]` is 1, so this uses the second case in the `Switch`.

```
In[11]:=  r[7]
Out[11]=  b
```

17 matches neither 0 nor 1, but does match _.

```
In[12]:=  Switch[17, 0, a, 1, b, _, q]
Out[12]=  q
```

An important point about symbolic systems such as *Mathematica* is that the conditions you give may yield neither `True` nor `False`. Thus, for example, the condition `x == y` does not yield `True` or `False` unless `x` and `y` have specific values, such as numerical ones.

In this case, the test gives neither `True` nor `False`, so both branches in the `If` remain unevaluated.

```
In[13]:=  If[x == y, a, b]
Out[13]=  If[x == y, a, b]
```

You can add a special fourth argument to `If`, which is used if the test does not yield `True` or `False`.

```
In[14]:=  If[x == y, a, b, c]
Out[14]=  c
```

`If[`*test*`,`*then*`,`*else*`,`*unknown*`]`	a form of `If` which includes the expression to use if *test* is neither `True` nor `False`
`TrueQ[`*expr*`]`	give `True` if *expr* is `True`, and `False` otherwise
lhs`===`*rhs* or `SameQ[`*lhs*`,`*rhs*`]`	give `True` if *lhs* and *rhs* are identical, and `False` otherwise
lhs`=!=`*rhs* or `UnsameQ[`*lhs*`,`*rhs*`]`	give `True` if *lhs* and *rhs* are not identical, and `False` otherwise
`MatchQ[`*expr*`,`*form*`]`	give `True` if the pattern *form* matches *expr*, and give `False` otherwise

Functions for dealing with symbolic conditions.

Mathematica leaves this as a symbolic equation.

```
In[15]:= x == y
Out[15]= x == y
```

Unless *expr* is manifestly `True`, `TrueQ[`*expr*`]` effectively assumes that *expr* is `False`.

```
In[16]:= TrueQ[x == y]
Out[16]= False
```

Unlike `==`, `===` tests whether two expressions are manifestly identical. In this case, they are not.

```
In[17]:= x === y
Out[17]= False
```

The main difference between *lhs* `===` *rhs* and *lhs* `==` *rhs* is that `===` always returns `True` or `False`, whereas `==` can leave its input in symbolic form, representing a symbolic equation, as discussed in "Equations". You should typically use `===` when you want to test the *structure* of an expression, and `==` if you want to test mathematical equality. The *Mathematica* pattern matcher effectively uses `===` to determine when one literal expression matches another.

You can use `===` to test the structure of expressions.

```
In[18]:= Head[a + b + c] === Times
Out[18]= False
```

The `==` operator gives a less useful result.

```
In[19]:= Head[a + b + c] == Times
Out[19]= Plus == Times
```

In setting up conditionals, you will often need to use combinations of tests, such as $test_1$ `&&` $test_2$ `&&` …. An important point is that the result from this combination of tests will be `False` if *any* of the $test_i$ yield `False`. *Mathematica* always evaluates the $test_i$ in turn, stopping if any of the $test_i$ yield `False`.

$expr_1$ `&&` $expr_2$ `&&` $expr_3$	evaluate until one of the $expr_i$ is found to be `False`
$expr_1$ `||` $expr_2$ `||` $expr_3$	evaluate until one of the $expr_i$ is found to be `True`

Evaluation of logical expressions.

This function involves a combination of two tests.

```
In[20]:= t[x_] := (x != 0 && 1 / x < 3)
```

Here both tests are evaluated.

```
In[21]:= t[2]
Out[21]= True
```

Here the first test yields `False`, so the second test is not tried. The second test would involve `1 / 0`, and would generate an error.

```
In[22]:= t[0]
Out[22]= False
```

The way that *Mathematica* evaluates logical expressions allows you to combine sequences of tests where later tests may make sense only if the earlier ones are satisfied. The behavior, which is analogous to that found in languages such as C, is convenient in constructing many kinds of *Mathematica* programs.

Loops and Control Structures

The execution of a *Mathematica* program involves the evaluation of a sequence of *Mathematica* expressions. In simple programs, the expressions to be evaluated may be separated by semi-colons, and evaluated one after another. Often, however, you need to evaluate expressions several times, in some kind of "loop".

`Do`[*expr*, {*i*, i_{max}}]	evaluate *expr* repetitively, with *i* varying from 1 to i_{max} in steps of 1
`Do`[*expr*, {*i*, i_{min}, i_{max}, *di*}]	evaluate *expr* with *i* varying from i_{min} to i_{max} in steps of *di*
`Do`[*expr*, {*i*, *list*}]	evaluate *expr* with *i* taking on values from *list*
`Do`[*expr*, {*n*}]	evaluate *expr* *n* times

Simple looping constructs.

This evaluates `Print[i^2]`, with `i` running from 1 to 4.

```
In[1]:= Do[Print[i^2], {i, 4}]

    1

    4

    9

    16
```

This executes an assignment for `t` in a loop with `k` running from 2 to 6 in steps of 2.

```
In[2]:= t = x; Do[t = 1 / (1 + k t), {k, 2, 6, 2}]; t
```

Out[2]= $$\frac{1}{1+\frac{6}{1+\frac{4}{1+2\,x}}}$$

The way iteration is specified in `Do` is exactly the same as in functions like `Table` and `Sum`. Just as in those functions, you can set up several nested loops by giving a sequence of iteration specifications to `Do`.

This loops over values of `i` from 1 to 4, and for each value of `i`, loops over `j` from 1 to `i - 1`.

```
In[3]:= Do[Print[{i, j}], {i, 4}, {j, i - 1}]

    {2, 1}

    {3, 1}

    {3, 2}

    {4, 1}

    {4, 2}

    {4, 3}
```

Sometimes you may want to repeat a particular operation a certain number of times, without changing the value of an iteration variable. You can specify this kind of repetition in `Do` just as you can in `Table` and other iteration functions.

This repeats the assignment `t = 1 / (1 + t)` three times.

```
In[4]:= t = x; Do[t = 1 / (1 + t), {3}]; t
```

Out[4]= $\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}$

You can put a procedure inside `Do`.

```
In[5]:= t = 67; Do[Print[t]; t = Floor[t / 2], {3}]

    67

    33

    16
```

`Nest[`*f*`,`*expr*`,`*n*`]`	apply *f* to *expr* *n* times
`FixedPoint[`*f*`,`*expr*`]`	start with *expr*, and apply *f* repeatedly until the result no longer changes
`NestWhile[`*f*`,`*expr*`,`*test*`]`	start with *expr*, and apply *f* repeatedly until applying *test* to the result no longer yields `True`

Applying functions repetitively.

`Do` allows you to repeat operations by evaluating a particular expression many times with different values for iteration variables. Often, however, you can make more elegant and efficient programs using the functional programming constructs discussed in "Applying Functions Repeatedly". `Nest[`*f*`, `*x*`, `*n*`]`, for example, allows you to apply a function repeatedly to an expression.

This nests `f` three times.

```
In[6]:= Nest[f, x, 3]
Out[6]= f[f[f[x]]]
```

By nesting a pure function, you can get the same result as in the example with `Do` above.

```
In[7]:= Nest[Function[t, 1 / (1 + t)], x, 3]
```

Out[7]= $$\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}$$

`Nest` allows you to apply a function a specified number of times. Sometimes, however, you may simply want to go on applying a function until the results you get no longer change. You can do this using `FixedPoint[`*f*`, `*x*`]`.

`FixedPoint` goes on applying a function until the result no longer changes.

```
In[8]:= FixedPoint[Function[t, Print[t]; Floor[t / 2]], 67]

    67

    33

    16

    8

    4

    2

    1

    0

Out[8]= 0
```

You can use `FixedPoint` to imitate the evaluation process in *Mathematica*, or the operation of functions such as *expr* `//.` *rules*. `FixedPoint` goes on until two successive results it gets are the same. `NestWhile` allows you to go on until an arbitrary function no longer yields `True`.

`Catch[`*expr*`]`	evaluate *expr* until `Throw[`*value*`]` is encountered, then return *value*
`Catch[`*expr*`,`*form*`]`	evaluate *expr* until `Throw[`*value*`, `*tag*`]` is encountered, where *form* matches *tag*
`Catch[`*expr*`,`*form*`,`*f*`]`	return *f*`[`*value*`, `*tag*`]` instead of *value*

Non local control of evaluation.

When the `Throw` is encountered, evaluation stops, and the current value of `i` is returned as the value of the enclosing `Catch`.

```
In[9]:= Catch[Do[Print[i]; If[i > 3, Throw[i]], {i, 10}]]
```

```
    1

    2

    3

    4
Out[9]= 4
```

`Throw` and `Catch` provide a flexible way to control the process of evaluation in *Mathematica*. The basic idea is that whenever a `Throw` is encountered, the evaluation that is then being done is stopped, and *Mathematica* immediately returns to the nearest appropriate enclosing `Catch`.

`Scan` applies the function `Print` to each successive element in the list, and in the end just returns `Null`.

```
In[10]:= Scan[Print, {7, 6, 5, 4}]

    7

    6

    5

    4
```

The evaluation of `Scan` stops as soon as `Throw` is encountered, and the enclosing `Catch` returns as its value the argument of `Throw`.

```
In[11]:= Catch[Scan[(Print[#]; If[# < 6, Throw[#]]) &, {7, 6, 5, 4}]]

    7

    6

    5
Out[11]= 5
```

The same result is obtained with `Map`, even though `Map` would have returned a list if its evaluation had not been stopped by encountering a `Throw`.

```
In[12]:= Catch[Map[(Print[#]; If[# < 6, Throw[#]]) &, {7, 6, 5, 4}]]

    7

    6

    5
Out[12]= 5
```

You can use `Throw` and `Catch` to divert the operation of functional programming constructs, allowing for example the evaluation of such constructs to continue only until some condition has been met. Note that if you stop evaluation using `Throw`, then the structure of the result you get may be quite different from what you would have got if you had allowed the evaluation to complete.

Here is a list generated by repeated application of a function.

```
In[13]:= NestList[1 / (# + 1) &, -2.5, 6]
Out[13]= {-2.5, -0.666667, 3., 0.25, 0.8, 0.555556, 0.642857}
```

Since there is no `Throw` encountered, the result here is just as before.

```
In[14]:= Catch[NestList[1 / (# + 1) &, -2.5, 6]]
Out[14]= {-2.5, -0.666667, 3., 0.25, 0.8, 0.555556, 0.642857}
```

Now the evaluation of the `NestList` is diverted, and the single number given as the argument of `Throw` is returned.

```
In[15]:= Catch[NestList[If[# > 1, Throw[#], 1 / (# + 1)] &, -2.5, 6]]
Out[15]= 3.
```

`Throw` and `Catch` operate in a completely global way: it does not matter how or where a `Throw` is generated—it will always stop evaluation and return to the enclosing `Catch`.

The `Throw` stops the evaluation of `f`, and causes the `Catch` to return just `a`, with no trace of `f` left.

```
In[16]:= Catch[f[Throw[a]]]
Out[16]= a
```

This defines a function which generates a `Throw` when its argument is larger than 10.

```
In[17]:= g[x_] := If[x > 10, Throw[overflow], x!]
```

No `Throw` is generated here.

```
In[18]:= Catch[g[4]]
Out[18]= 24
```

But here the `Throw` generated inside the evaluation of `g` returns to the enclosing `Catch`.

```
In[19]:= Catch[g[40]]
Out[19]= overflow
```

In small programs, it is often adequate to use `Throw[`*value*`]` and `Catch[`*expr*`]` in their simplest form. But particularly if you write larger programs that contain many separate pieces, it is usually much better to use `Throw[`*value*`,` *tag*`]` and `Catch[`*expr*`,` *form*`]`. By keeping the expressions *tag* and *form* local to a particular piece of your program, you can then ensure that your `Throw` and `Catch` will also operate only within that piece.

Here the `Throw` is caught by the inner `Catch`.

```
In[20]:= Catch[f[Catch[Throw[x, a], a]], b]
Out[20]= f[x]
```

But here it is caught only by the outer `Catch`.

```
In[21]:= Catch[f[Catch[Throw[x, b], a]], b]
Out[21]= x
```

You can use patterns in specifying the tags which a particular `Catch` should catch.

```
In[22]:= Catch[Throw[x, a], a | b]
Out[22]= x
```

This keeps the tag `a` completely local.

```
In[23]:= Module[{a}, Catch[Throw[x, a], a]]
Out[23]= x
```

You should realize that there is no need for the tag that appears in `Throw` to be a constant; in general it can be any expression.

Here the inner `Catch` catches all throws with tags less than 4, and continues the `Do`. But as soon as the tag reaches 4, the outer `Catch` is needed.

```
In[24]:= Catch[Do[Catch[Throw[i^2, i], n_ /; n < 4], {i, 10}], _]
Out[24]= 16
```

When you use `Catch[`*expr*`,` *form*`]` with `Throw[`*value*`,` *tag*`]`, the value returned by `Catch` is simply the expression *value* given in the `Throw`. If you use `Catch[`*expr*`,` *form*`,` *f*`]`, however, then the value returned by `Catch` is instead *f*`[`*value*`,` *tag*`]`.

Here `f` is applied to the value and tag in the `Throw`.

```
In[25]:= Catch[Throw[x, a], a, f]
Out[25]= f[x, a]
```

If there is no `Throw`, `f` is never used.

```
In[26]:= Catch[x, a, f]
Out[26]= x
```

`While[`*test*`,`*body*`]`	evaluate *body* repetitively, so long as *test* is `True`
`For[`*start*`,`*test*`,`*incr*`,`*body*`]`	evaluate *start*, then repetitively evaluate *body* and *incr*, until *test* fails

General loop constructs.

Functions like `Do`, `Nest` and `FixedPoint` provide structured ways to make loops in *Mathematica* programs, while `Throw` and `Catch` provide opportunities for modifying this structure. Sometimes, however, you may want to create loops that even from the outset have less structure. And in such cases, you may find it convenient to use the functions `While` and `For`, which perform operations repeatedly, stopping when a specified condition fails to be true.

The `While` loop continues until the condition fails.

```
In[27]:= n = 17; While[(n = Floor[n / 2]) != 0, Print[n]]

            8

            4

            2

            1
```

The functions `While` and `For` in *Mathematica* are similar to the control structures `while` and `for` in languages such as C. Notice, however, that there are a number of important differences. For example, the roles of comma and semicolon are reversed in *Mathematica* `For` loops relative to C language ones.

This is a very common form for a `For` loop. `i++` increments the value of `i`.

```
In[28]:= For[i = 1, i < 4, i++, Print[i]]

            1

            2

            3
```

Here is a more complicated `For` loop. Notice that the loop terminates as soon as the test `i^2 < 10` fails.

```
In[29]:= For[i = 1; t = x, i^2 < 10, i++, t = t^2 + i; Print[t]]
```

$$1 + x^2$$

$$2 + \left(1 + x^2\right)^2$$

$$3 + \left(2 + \left(1 + x^2\right)^2\right)^2$$

In *Mathematica*, both `While` and `For` always evaluate the loop test before evaluating the body of the loop. As soon as the loop test fails to be `True`, `While` and `For` terminate. The body of the loop is thus only evaluated in situations where the loop test is `True`.

The loop test fails immediately, so the body of the loop is never evaluated.

```
In[30]:= While[False, Print[x]]
```

In a `While` or `For` loop, or in general in any *Mathematica* procedure, the *Mathematica* expressions you give are evaluated in a definite sequence. You can think of this sequence as defining the "flow of control" in the execution of a *Mathematica* program.

In most cases, you should try to keep the flow of control in your *Mathematica* programs as simple as possible. The more the flow of control depends for example on specific values generated during the execution of the program, the more difficult you will typically find it to understand the structure and operation of the program.

Functional programming constructs typically involve very simple flow of control. `While` and `For` loops are always more complicated, since they are set up to make the flow of control depend on the values of the expressions given as tests. Nevertheless, even in such loops, the flow of control does not usually depend on the values of expressions given in the body of the loop.

In some cases, however, you may need to construct *Mathematica* programs in which the flow of control is affected by values generated during the execution of a procedure or of the body of a loop. One way to do this, which fits in with functional programming ideas, is to use `Throw` and `Catch`. But *Mathematica* also provides various functions for modifying the flow of control which work like in languages such as C.

`Break[]`	exit the nearest enclosing loop
`Continue[]`	go to the next step in the current loop
`Return[`*expr*`]`	return the value *expr*, exiting all procedures and loops in a function
`Goto[`*name*`]`	go to the element `Label[`*name*`]` in the current procedure
`Throw[`*value*`]`	return *value* as the value of the nearest enclosing `Catch` (non-local return)

Control flow functions.

The `Break[]` causes the loop to terminate as soon as `t` exceeds `19`.

```
In[31]:= t = 1; Do[t *= k; Print[t]; If[t > 19, Break[]], {k, 10}]

    1

    2

    6

    24
```

When `k < 3`, the `Continue[]` causes the loop to be continued, without executing `t += 2`.

```
In[32]:= t = 1; Do[t *= k; Print[t]; If[k < 3, Continue[]]; t += 2, {k, 10}]

    1

    2

    6

    32

    170

    1032

    7238

    57 920

    521 298

    5 213 000
```

`Return[`*expr*`]` allows you to exit a particular function, returning a value. You can think of `Throw` as a kind of non-local return which allows you to exit a whole sequence of nested functions. Such behavior can be convenient for handling certain error conditions.

Here is an example of the use of `Return`. This particular procedure could equally well have been written without using `Return`.

```
In[33]:= f[x_] := (If[x > 5, Return[big]]; t = x^3; Return[t - 7])
```

When the argument is greater than 5, the first `Return` in the procedure is used.

```
In[34]:= f[10]
Out[34]= big
```

This function "throws" `error` if its argument is negative.

```
In[35]:= h[x_] := If[x < 0, Throw[error], Sqrt[x]]
```

No `Throw` is generated here.

```
In[36]:= Catch[h[6] + 2]
```

Out[36]= $2 + \sqrt{6}$

But in this case a `Throw` is generated, and the whole `Catch` returns the value `error`.

```
In[37]:= Catch[h[-6] + 2]
Out[37]= error
```

Functions like `Continue[]` and `Break[]` allow you to "transfer control" to the beginning or end of a loop in a *Mathematica* program. Sometimes you may instead need to transfer control to a particular element in a *Mathematica* procedure. If you give a `Label` as an element in a procedure, you can use `Goto` to transfer control to this element.

This goes on looping until `q` exceeds `6`.

```
In[38]:= (q = 2; Label[begin]; Print[q]; q += 3; If[q < 6, Goto[begin]])

            2

            5
```

Note that you can use `Goto` in a particular *Mathematica* procedure only when the `Label` it specifies occurs as an element of the same *Mathematica* procedure. In general, use of `Goto` reduces the degree of structure that can readily be perceived in a program, and therefore makes the operation of the program more difficult to understand.

Collecting Expressions During Evaluation

In many computations one is concerned only with the final result of evaluating the expression given as input. But sometimes one also wants to collect expressions that were generated in the course of the evaluation. You can do this using `Sow` and `Reap`.

`Sow[`*val*`]`	sow the value *val* for the nearest enclosing `Reap`
`Reap[`*expr*`]`	evaluate *expr*, returning also a list of values sown by `Sow`

Using `Sow` and `Reap`.

Here the output contains only the final result.

```
In[1]:= a = 3; a += a^2 + 1; a = Sqrt[a + a^2]
```

Out[1]= $\sqrt{182}$

Here two intermediate results are also given.

```
In[2]:= Reap[Sow[a = 3]; a += Sow[a^2 + 1]; a = Sqrt[a + a^2]]
```

Out[2]= $\left\{\sqrt{182}, \{\{3, 10\}\}\right\}$

This computes a sum, collecting all terms that are even.

```
In[3]:= Reap[Sum[If[EvenQ[#], Sow[#], #] &[i^2 + 1], {i, 10}]]

Out[3]= {395, {{2, 10, 26, 50, 82}}}
```

Like `Throw` and `Catch`, `Sow` and `Reap` can be used anywhere in a computation.

This defines a function that can do a `Sow`.

```
In[4]:= f[x_] := (If[x < 1 / 2, Sow[x]]; 3.5 x (1 - x))
```

This nests the function, reaping all cases below 1/2.

```
In[5]:= Reap[Nest[f, 0.8, 10]]

Out[5]= {0.868312, {{0.415332, 0.446472, 0.408785, 0.456285}}}
```

`Sow` [*val*, *tag*]	sow *val* with a tag to indicate when to reap
`Sow` [*val*, {tag_1, tag_2, …}]	sow *val* for each of the tag_i
`Reap` [*expr*, *form*]	reap all values whose tags match *form*
`Reap` [*expr*, {$form_1$, $form_2$, …}]	make separate lists for each of the $form_i$
`Reap` [*expr*, {$form_1$, …}, *f*]	apply *f* to each distinct tag and list of values

Sowing and reaping with tags.

This reaps only values sown with tag `x`.

```
In[6]:=  Reap[Sow[1, x]; Sow[2, y]; Sow[3, x], x]
Out[6]=  {3, {{1, 3}}}
```

Here 1 is sown twice with tag `x`.

```
In[7]:=  Reap[Sow[1, {x, x}]; Sow[2, y]; Sow[3, x], x]
Out[7]=  {3, {{1, 1, 3}}}
```

Values sown with different tags always appear in different sublists.

```
In[8]:=  Reap[Sow[1, {x, x}]; Sow[2, y]; Sow[3, x]]
Out[8]=  {3, {{1, 1, 3}, {2}}}
```

The makes a sublist for each form of tag being reaped.

```
In[9]:=  Reap[Sow[1, {x, x}]; Sow[2, y]; Sow[3, x], {x, x, y}]
Out[9]=  {3, {{{1, 1, 3}}, {{1, 1, 3}}, {{2}}}}
```

This applies `f` to each distinct tag and list of values.

```
In[10]:=  Reap[Sow[1, {x, x}]; Sow[2, y]; Sow[3, x], _, f]
Out[10]=  {3, {f[x, {1, 1, 3}], f[y, {2}]}}
```

The tags can be part of the computation.

```
In[11]:=  Reap[Do[Sow[i / j, GCD[i, j]], {i, 4}, {j, i}]]
```

Out[11]= $\left\{\text{Null}, \left\{\left\{1, 2, 3, \frac{3}{2}, 4, \frac{4}{3}\right\}, \{1, 2\}, \{1\}, \{1\}\right\}\right\}$

Tracing Evaluation

The standard way in which *Mathematica* works is to take any expression you give as input, evaluate the expression completely, and then return the result. When you are trying to understand what *Mathematica* is doing, however, it is often worthwhile to look not just at the final result of evaluation, but also at intermediate steps in the evaluation process.

`Trace[`*expr*`]`	generate a list of all expressions used in the evaluation of *expr*
`Trace[`*expr*`,`*form*`]`	include only expressions which match the pattern *form*

Tracing the evaluation of expressions.

The expression 1 + 1 is evaluated immediately to 2.

```
In[1]:= Trace[1 + 1]
Out[1]= {1 + 1, 2}
```

The 2 ^ 3 is evaluated before the addition is done.

```
In[2]:= Trace[2 ^ 3 + 4]
```

$$\text{Out[2]= } \{\{2^3, 8\}, 8 + 4, 12\}$$

The evaluation of each subexpression is shown in a separate sublist.

```
In[3]:= Trace[2 ^ 3 + 4 ^ 2 + 1]
```

$$\text{Out[3]= } \{\{2^3, 8\}, \{4^2, 16\}, 8 + 16 + 1, 25\}$$

`Trace[`*expr*`]` gives a list which includes *all* the intermediate expressions involved in the evaluation of *expr*. Except in rather simple cases, however, the number of intermediate expressions generated in this way is typically very large, and the list returned by `Trace` is difficult to understand.

`Trace[`*expr*`, `*form*`]` allows you to “filter” the expressions that `Trace` records, keeping only those which match the pattern *form*.

Here is a recursive definition of a factorial function.

```
In[4]:= fac[n_] := n fac[n - 1]; fac[1] = 1
Out[4]= 1
```

This gives *all* the intermediate expressions generated in the evaluation of `fac[3]`. The result is quite complicated.

```
In[5]:= Trace[fac[3]]
Out[5]= {fac[3], 3 fac[3 - 1], {{3 - 1, 2}, fac[2], 2 fac[2 - 1], {{2 - 1, 1}, fac[1], 1}, 2 × 1, 2}, 3 × 2, 6}
```

This shows only intermediate expressions of the form `fac[n_]`.

```
In[6]:= Trace[fac[3], fac[n_]]
Out[6]= {fac[3], {fac[2], {fac[1]}}}
```

You can specify any pattern in `Trace`.

```
In[7]:= Trace[fac[10], fac[n_ /; n > 5]]
Out[7]= {fac[10], {fac[9], {fac[8], {fac[7], {fac[6]}}}}}
```

`Trace[`*expr*`,` *form*`]` effectively works by intercepting every expression that is about to be evaluated during the evaluation of *expr*, and picking out those that match the pattern *form*.

If you want to trace "calls" to a function like `fac`, you can do so simply by telling `Trace` to pick out expressions of the form `fac[n_]`. You can also use patterns like `f[n_, 2]` to pick out calls with particular argument structure.

A typical *Mathematica* program, however, consists not only of "function calls" like `fac[n]`, but also of other elements, such as assignments to variables, control structures, and so on. All of these elements are represented as expressions. As a result, you can use patterns in `Trace` to pick out any kind of *Mathematica* program element. Thus, for example, you can use a pattern like `k = _` to pick out all assignments to the symbol `k`.

This shows the sequence of assignments made for `k`.

```
In[8]:= Trace[(k = 2; For[i = 1, i < 4, i++, k = i / k]; k), k = _]
```

Out[8]= $\left\{\{k=2\},\left\{\left\{k=\frac{1}{2}\right\},\{k=4\},\left\{k=\frac{3}{4}\right\}\right\}\right\}$

`Trace[`*expr*`,` *form*`]` can pick out expressions that occur at any time in the evaluation of *expr*. The expressions need not, for example, appear directly in the form of *expr* that you give. They may instead occur, say, during the evaluation of functions that are called as part of the evaluation of *expr*.

Here is a function definition.

```
In[9]:= h[n_] := (k = n / 2; Do[k = i / k, {i, n}]; k)
```

You can look for expressions generated during the evaluation of h.

```
In[10]:= Trace[h[3], k = _]
```

$$\text{Out[10]=}\ \left\{\left\{k = \frac{3}{2}\right\}, \left\{\left\{k = \frac{2}{3}\right\}, \{k = 3\}, \{k = 1\}\right\}\right\}$$

Trace allows you to monitor intermediate steps in the evaluation not only of functions that you define, but also of some functions that are built into *Mathematica*. You should realize, however, that the specific sequence of intermediate steps followed by built-in *Mathematica* functions depends in detail on their implementation and optimization in a particular version of *Mathematica*.

Trace[*expr*,*f*[___]]	show all calls to the function *f*
Trace[*expr*,*i*=_]	show assignments to *i*
Trace[*expr*,_=_]	show all assignments
Trace[*expr*,Message[___]]	show messages generated

Some ways to use Trace.

The function Trace returns a list that represents the "history" of a *Mathematica* computation. The expressions in the list are given in the order that they were generated during the computation. In most cases, the list returned by Trace has a nested structure, which represents the "structure" of the computation.

The basic idea is that each sublist in the list returned by Trace represents the "evaluation chain" for a particular *Mathematica* expression. The elements of this chain correspond to different forms of the same expression. Usually, however, the evaluation of one expression requires the evaluation of a number of other expressions, often subexpressions. Each subsidiary evaluation is represented by a sublist in the structure returned by Trace.

Here is a sequence of assignments.

```
In[11]:= a[1] = a[2]; a[2] = a[3]; a[3] = a[4]
Out[11]= a[4]
```

This yields an evaluation chain reflecting the sequence of transformations for a[*i*] used.

```
In[12]:= Trace[a[1]]
Out[12]= {a[1], a[2], a[3], a[4]}
```

The successive forms generated in the simplification of `y + x + y` show up as successive elements in its evaluation chain.

```
In[13]:= Trace[y + x + y]

Out[13]= {y + x + y, x + y + y, x + 2 y}
```

Each argument of the function `f` has a separate evaluation chain, given in a sublist.

```
In[14]:= Trace[f[1 + 1, 2 + 3, 4 + 5]]

Out[14]= {{1 + 1, 2}, {2 + 3, 5}, {4 + 5, 9}, f[2, 5, 9]}
```

The evaluation chain for each subexpression is given in a separate sublist.

```
In[15]:= Trace[x x + y y]
```

Out[15]= $\{\{x\,x, x^2\}, \{y\,y, y^2\}, x^2 + y^2\}$

Tracing the evaluation of a nested expression yields a nested list.

```
In[16]:= Trace[f[f[f[1 + 1]]]]

Out[16]= {{{{1 + 1, 2}, f[2]}, f[f[2]]}, f[f[f[2]]]}
```

There are two basic ways that subsidiary evaluations can be required during the evaluation of a *Mathematica* expression. The first way is that the expression may contain subexpressions, each of which has to be evaluated. The second way is that there may be rules for the evaluation of the expression that involve other expressions which themselves must be evaluated. Both kinds of subsidiary evaluations are represented by sublists in the structure returned by `Trace`.

The subsidiary evaluations here come from evaluation of the arguments of `f` and `g`.

```
In[17]:= Trace[f[g[1 + 1], 2 + 3]]

Out[17]= {{{1 + 1, 2}, g[2]}, {2 + 3, 5}, f[g[2], 5]}
```

Here is a function with a condition attached.

```
In[18]:= fe[n_] := n + 1 /; EvenQ[n]
```

The evaluation of `fe[6]` involves a subsidiary evaluation associated with the condition.

```
In[19]:= Trace[fe[6]]

Out[19]= {fe[6], {{EvenQ[6], True}, RuleCondition[$ConditionHold[$ConditionHold[6 + 1]], True],
           $ConditionHold[$ConditionHold[6 + 1]]}, 6 + 1, 7}
```

You often get nested lists when you trace the evaluation of functions that are defined “recursively” in terms of other instances of themselves. The reason is typically that each new instance of the function appears as a subexpression in the expressions obtained by evaluating previous instances of the function.

Thus, for example, with the definition `fac[n_] := n fac[n - 1]`, the evaluation of `fac[6]` yields the expression `6 fac[5]`, which contains `fac[5]` as a subexpression.

The successive instances of `fac` generated appear in successively nested sublists.

```
In[20]:= Trace[fac[6], fac[_]]
Out[20]= {fac[6], {fac[5], {fac[4], {fac[3], {fac[2], {fac[1]}}}}}}
```

With this definition, `fp[`*n*` - 1]` is obtained directly as the value of `fp[`*n*`]`.

```
In[21]:= fp[n_] := fp[n - 1] /; n > 1
```

`fp[`*n*`]` never appears in a subexpression, so no sublists are generated.

```
In[22]:= Trace[fp[6], fp[_]]
Out[22]= {fp[6], fp[6 - 1], fp[5], fp[5 - 1], fp[4], fp[4 - 1], fp[3], fp[3 - 1], fp[2], fp[2 - 1], fp[1]}
```

Here is the recursive definition of the Fibonacci numbers.

```
In[23]:= fib[n_] := fib[n - 1] + fib[n - 2]
```

Here are the end conditions for the recursion.

```
In[24]:= fib[0] = fib[1] = 1
Out[24]= 1
```

This shows all the steps in the recursive evaluation of `fib[5]`.

```
In[25]:= Trace[fib[5], fib[_]]
Out[25]= {fib[5], {fib[4], {fib[3], {fib[2], {fib[1]}, {fib[0]}}, {fib[1]}},
           {fib[2], {fib[1]}, {fib[0]}}}, {fib[3], {fib[2], {fib[1]}, {fib[0]}}, {fib[1]}}}
```

Each step in the evaluation of any *Mathematica* expression can be thought of as the result of applying a particular transformation rule. As discussed in "Associating Definitions with Different Symbols", all the rules that *Mathematica* knows are associated with specific symbols or “tags”. You can use `Trace[`*expr*`, `*f*`]` to see all the steps in the evaluation of *expr* that are performed using transformation rules associated with the symbol *f*. In this case, `Trace` gives not only the expressions to which each rule is applied, but also the results of applying the rules.

In general, `Trace[`*expr*`,` *form*`]` picks out all the steps in the evaluation of *expr* where *form* matches *either* the expression about to be evaluated, *or* the tag associated with the rule used.

`Trace[`*expr*`,`*f*`]`	show all evaluations which use transformation rules associated with the symbol *f*
`Trace[`*expr*`,`*f*`|`*g*`]`	show all evaluations associated with either *f* or *g*

Tracing evaluations associated with particular tags.

This shows only intermediate expressions that match `fac[_]`.

```
In[26]:= Trace[fac[3], fac[_]]
Out[26]= {fac[3], {fac[2], {fac[1]}}}
```

This shows all evaluations that use transformation rules associated with the symbol `fac`.

```
In[27]:= Trace[fac[3], fac]
Out[27]= {fac[3], 3 fac[3 - 1], {fac[2], 2 fac[2 - 1], {fac[1], 1}}}
```

Here is a rule for the `log` function.

```
In[28]:= log[x_ y_] := log[x] + log[y]
```

This traces the evaluation of `log[a b c d]`, showing all transformations associated with `log`.

```
In[29]:= Trace[log[a b c d], log]
Out[29]= {log[a b c d], log[a] + log[b c d], {log[b c d], log[b] + log[c d], {log[c d], log[c] + log[d]}}}
```

`Trace[`*expr*`,`*form*`,TraceOn->`*oform*`]`	switch on tracing only within forms matching *oform*
`Trace[`*expr*`,`*form*`,TraceOff->`*oform*`]`	switch off tracing within any form matching *oform*

Switching off tracing inside certain forms.

`Trace[`*expr*`,` *form*`]` allows you to trace expressions matching *form* generated at any point in the evaluation of *expr*. Sometimes, you may want to trace only expressions generated during certain parts of the evaluation of *expr*.

By setting the option `TraceOn -> `*oform*, you can specify that tracing should be done only during the evaluation of forms which match *oform*. Similarly, by setting `TraceOff -> `*oform*, you can specify that tracing should be switched off during the evaluation of forms which match *oform*.

This shows all steps in the evaluation.

```
In[30]:= Trace[log[fac[2] x]]
Out[30]= {{{fac[2], 2 fac[2 - 1], {{2 - 1, 1}, fac[1], 1}, 2 × 1, 2}, 2 x}, log[2 x], log[2] + log[x]}
```

This shows only those steps that occur during the evaluation of `fac`.

```
In[31]:= Trace[log[fac[2] x], TraceOn -> fac]
Out[31]= {{{fac[2], 2 fac[2 - 1], {{2 - 1, 1}, fac[1], 1}, 2 × 1, 2}}}
```

This shows only those steps that do not occur during the evaluation of `fac`.

```
In[32]:= Trace[log[fac[2] x], TraceOff -> fac]
Out[32]= {{{fac[2], 2}, 2 x}, log[2 x], log[2] + log[x]}
```

`Trace[`*expr*`,`*lhs*`->`*rhs*`]`	find all expressions matching *lhs* that arise during the evaluation of *expr*, and replace them with *rhs*

Applying rules to expressions encountered during evaluation.

This tells `Trace` to return only the arguments of `fib` used in the evaluation of `fib[5]`.

```
In[33]:= Trace[fib[5], fib[n_] -> n]
Out[33]= {5, {4, {3, {2, {1}, {0}}, {1}}, {2, {1}, {0}}}, {3, {2, {1}, {0}}, {1}}}
```

A powerful aspect of the *Mathematica* `Trace` function is that the object it returns is basically a standard *Mathematica* expression which you can manipulate using other *Mathematica* functions. One important point to realize, however, is that `Trace` wraps all expressions that appear in the list it produces with `HoldForm` to prevent them from being evaluated. The `HoldForm` is not displayed in standard *Mathematica* output format, but it is still present in the internal structure of the expression.

This shows the expressions generated at intermediate stages in the evaluation process.

```
In[34]:= Trace[1 + 3^2]
Out[34]= {{3^2, 9}, 1 + 9, 10}
```

The expressions are wrapped with `HoldForm` to prevent them from evaluating.

```
In[35]:= Trace[1 + 3^2] // InputForm
Out[35]//InputForm= {{HoldForm[3^2], HoldForm[9]}, HoldForm[1 + 9],   HoldForm[10]}
```

In standard *Mathematica* output format, it is sometimes difficult to tell which lists are associated with the structure returned by `Trace`, and which are expressions being evaluated.

```
In[36]:= Trace[{1 + 1, 2 + 3}]

Out[36]= {{1 + 1, 2}, {2 + 3, 5}, {2, 5}}
```

Looking at the input form resolves any ambiguities.

```
In[37]:= InputForm[%]

Out[37]//InputForm= {{HoldForm[1 + 1], HoldForm[2]},   {HoldForm[2 + 3], HoldForm[5]}, HoldForm[{2, 5}]}
```

When you use a transformation rule in `Trace`, the result is evaluated before being wrapped with `HoldForm`.

```
In[38]:= Trace[fac[4], fac[n_] -> n + 1]

Out[38]= {5, {4, {3, {2}}}}
```

For sophisticated computations, the list structures returned by `Trace` can be quite complicated. When you use `Trace[`*expr*`, `*form*`]`, `Trace` will include as elements in the lists only those expressions which match the pattern *form*. But whatever pattern you give, the nesting structure of the lists remains the same.

This shows all occurrences of `fib[_]` in the evaluation of `fib[3]`.

```
In[39]:= Trace[fib[3], fib[_]]

Out[39]= {fib[3], {fib[2], {fib[1]}, {fib[0]}}, {fib[1]}}
```

This shows only occurrences of `fib[1]`, but the nesting of the lists is the same as for `fib[_]`.

```
In[40]:= Trace[fib[3], fib[1]]

Out[40]= {{{fib[1]}}, {fib[1]}}
```

You can set the option `TraceDepth -> `*n* to tell `Trace` to include only lists nested at most *n* levels deep. In this way, you can often pick out the "big steps" in a computation, without seeing the details. Note that by setting `TraceDepth` or `TraceOff` you can avoid looking at many of the steps in a computation, and thereby significantly speed up the operation of `Trace` for that computation.

This shows only steps that appear in lists nested at most two levels deep.

```
In[41]:= Trace[fib[3], fib[_], TraceDepth -> 2]

Out[41]= {fib[3], {fib[1]}}
```

`Trace[`*expr*`,`*form*`,TraceDepth->`*n*`]`	trace the evaluation of *expr*, ignoring steps that lead to lists nested more than *n* levels deep

Restricting the depth of tracing.

When you use `Trace[`*expr*`, `*form*`]`, you get a list of all the expressions which match *form* produced during the evaluation of *expr*. Sometimes it is useful to see not only these expressions, but also the results that were obtained by evaluating them. You can do this by setting the option `TraceForward -> True` in `Trace`.

This shows not only expressions which match `fac[_]`, but also the results of evaluating those expressions.

```
In[42]:= Trace[fac[4], fac[_], TraceForward -> True]

Out[42]= {fac[4], {fac[3], {fac[2], {fac[1], 1}, 2}, 6}, 24}
```

Expressions picked out using `Trace[`*expr*`, `*form*`]` typically lie in the middle of an evaluation chain. By setting `TraceForward -> True`, you tell `Trace` to include also the expression obtained at the end of the evaluation chain. If you set `TraceForward -> All`, `Trace` will include *all* the expressions that occur after the expression matching *form* on the evaluation chain.

With `TraceForward -> All`, all elements on the evaluation chain after the one that matches `fac[_]` are included.

```
In[43]:= Trace[fac[4], fac[_], TraceForward -> All]

Out[43]= {fac[4], 4 fac[4 - 1],
          {fac[3], 3 fac[3 - 1], {fac[2], 2 fac[2 - 1], {fac[1], 1}, 2 × 1, 2}, 3 × 2, 6}, 4 × 6, 24}
```

By setting the option `TraceForward`, you can effectively see what happens to a particular form of expression during an evaluation. Sometimes, however, you want to find out not what happens to a particular expression, but instead how that expression was generated. You can do this by setting the option `TraceBackward`. What `TraceBackward` does is to show you what *preceded* a particular form of expression on an evaluation chain.

This shows that the number `120` came from the evaluation of `fac[5]` during the evaluation of `fac[10]`.

```
In[44]:= Trace[fac[10], 120, TraceBackward -> True]

Out[44]= {{{{{{fac[5], 120}}}}}}
```

Here is the whole evaluation chain associated with the generation of the number 120.

```
In[45]:= Trace[fac[10], 120, TraceBackward -> All]
Out[45]= {{{{{{fac[5], 5 fac[5 - 1], 5 × 24, 120}}}}}}
```

`TraceForward` and `TraceBackward` allow you to look forward and backward in a particular evaluation chain. Sometimes, you may also want to look at the evaluation chains within which the particular evaluation chain occurs. You can do this using `TraceAbove`. If you set the option `TraceAbove -> True`, then `Trace` will include the initial and final expressions in all the relevant evaluation chains. With `TraceAbove -> All`, `Trace` includes all the expressions in all these evaluation chains.

This includes the initial and final expressions in all evaluation chains which contain the chain that contains 120.

```
In[46]:= Trace[fac[7], 120, TraceAbove -> True]
Out[46]= {fac[7], {fac[6], {fac[5], 120}, 720}, 5040}
```

This shows all the ways that fib[2] is generated during the evaluation of fib[5].

```
In[47]:= Trace[fib[5], fib[2], TraceAbove -> True]
Out[47]= {fib[5], {fib[4], {fib[3], {fib[2], 2}, 3}, {fib[2], 2}, 5}, {fib[3], {fib[2], 2}, 3}, 8}
```

`Trace[`*expr*`,`*form*`,`*opts*`]`	trace the evaluation of *expr* using the specified options
`TraceForward->True`	include the final expression in the evaluation chain containing *form*
`TraceForward->All`	include all expressions following *form* in the evaluation chain
`TraceBackward->True`	include the first expression in the evaluation chain containing *form*
`TraceBackward->All`	include all expressions preceding *form* in the evaluation chain
`TraceAbove->True`	include the first and last expressions in all evaluation chains which contain the chain containing *form*
`TraceAbove->All`	include all expressions in all evaluation chains which contain the chain containing *form*

Option settings for including extra steps in trace lists.

The basic way that `Trace[`*expr*`, …]` works is to intercept each expression encountered during the evaluation of *expr*, and then to use various criteria to determine whether this expression should be recorded. Normally, however, `Trace` intercepts expressions only *after* function arguments have been evaluated. By setting `TraceOriginal -> True`, you can get `Trace` also to look at expressions *before* function arguments have been evaluated.

This includes expressions which match `fac[_]` both before and after argument evaluation.

```
In[48]:= Trace[fac[3], fac[_], TraceOriginal -> True]

Out[48]= {fac[3], {fac[3 - 1], fac[2], {fac[2 - 1], fac[1]}}}
```

The list structure produced by `Trace` normally includes only expressions that constitute steps in non-trivial evaluation chains. Thus, for example, individual symbols that evaluate to themselves are not normally included. Nevertheless, if you set `TraceOriginal -> True`, then `Trace` looks at absolutely every expression involved in the evaluation process, including those that have trivial evaluation chains.

In this case, `Trace` includes absolutely all expressions, even those with trivial evaluation chains.

```
In[49]:= Trace[fac[1], TraceOriginal -> True]

Out[49]= {fac[1], {fac}, {1}, fac[1], 1}
```

option name	*default value*	
`TraceForward`	`False`	whether to show expressions following *form* in the evaluation chain
`TraceBackward`	`False`	whether to show expressions preceding *form* in the evaluation chain
`TraceAbove`	`False`	whether to show evaluation chains leading to the evaluation chain containing *form*
`TraceOriginal`	`False`	whether to look at expressions before their heads and arguments are evaluated

Additional options for `Trace`.

When you use `Trace` to study the execution of a program, there is an issue about how local variables in the program should be treated. As discussed in "How Modules Work", *Mathematica* scoping constructs such as `Module` create symbols with new names to represent local variables. Thus, even if you called a variable `x` in the original code for your program, the variable may effectively be renamed `x$`*nnn* when the program is executed.

`Trace[`*expr*`,` *form*`]` is set up so that by default a symbol *x* that appears in *form* will match all symbols with names of the form *x*$*nnn* that arise in the execution of *expr*. As a result, you can for example use `Trace[`*expr*`,` *x* `= _]` to trace assignment to all variables, local and global, that were named *x* in your original program.

`Trace[`*expr*`,`*form*`,MatchLocalNames->False]`	include all steps in the execution of *expr* that match *form*, with no replacements for local variable names allowed

Preventing the matching of local variables.

In some cases, you may want to trace only the global variable *x*, and not any local variables that were originally named *x*. You can do this by setting the option `MatchLocalNames -> False`.

This traces assignments to all variables with names of the form `x$`*nnn*.

```
In[50]:= Trace[Module[{x}, x = 5], x = _]
Out[50]= {{x$1 = 5}}
```

This traces assignments only to the specific global variable *x*.

```
In[51]:= Trace[Module[{x}, x = 5], x = _, MatchLocalNames -> False]
Out[51]= {}
```

The function `Trace` performs a complete computation, then returns a structure which represents the history of the computation. Particularly in very long computations, it is however sometimes useful to see traces of the computation as it proceeds. The function `TracePrint` works essentially like `Trace`, except that it prints expressions when it encounters them, rather than saving up all of the expressions to create a list structure.

This prints expressions encountered in the evaluation of `fib[3]`.

```
In[52]:= TracePrint[fib[3], fib[_]]

    fib[3]
     fib[3 - 1]
     fib[2]
      fib[2 - 1]
      fib[1]
      fib[2 - 2]
      fib[0]
     fib[3 - 2]
     fib[1]

Out[52]= 3
```

The sequence of expressions printed by `TracePrint` corresponds to the sequence of expressions given in the list structure returned by `Trace`. Indentation in the output from `TracePrint` corresponds to nesting in the list structure from `Trace`. You can use the `Trace` options `TraceOn`, `TraceOff` and `TraceForward` in `TracePrint`. However, since `TracePrint` produces output as it goes, it cannot support the option `TraceBackward`. In addition, `TracePrint` is set up so that `TraceOriginal` is effectively always set to `True`.

`Trace[`*expr*`,`…`]`	trace the evaluation of *expr*, returning a list structure containing the expressions encountered
`TracePrint[`*expr*`,`…`]`	trace the evaluation of *expr*, printing the expressions encountered
`TraceDialog[`*expr*`,`…`]`	trace the evaluation of *expr*, initiating a dialog when each specified expression is encountered
`TraceScan[`*f*`,`*expr*`,`…`]`	trace the evaluation of *expr*, applying *f* to `HoldForm` of each expression encountered

Functions for tracing evaluation.

This enters a dialog when `fac[5]` is encountered during the evaluation of `fac[10]`.

```
In[53]:= TraceDialog[fac[10], fac[5]]

   TraceDialog::dgbgn : Entering Dialog; use Return[] to exit.

Out[53]= fac[5]
```

Inside the dialog you can for example find out where you are by looking at the "stack".

```
In[54]:= Stack[]
Out[54]= {TraceDialog, Times, Times, Times, Times, Times, fac}
```

This returns from the dialog, and gives the final result from the evaluation of `fac[10]`.

```
In[55]:= Return[]

         TraceDialog::dgend : Exiting Dialog.

Out[55]= 3 628 800
```

The function `TraceDialog` effectively allows you to stop in the middle of a computation, and interact with the *Mathematica* environment that exists at that time. You can for example find values of intermediate variables in the computation, and even reset those values. There are however a number of subtleties, mostly associated with pattern and module variables.

What `TraceDialog` does is to call the function `Dialog` on a sequence of expressions. The `Dialog` function is discussed in detail in "Dialogs". When you call `Dialog`, you are effectively starting a subsidiary *Mathematica* session with its own sequence of input and output lines.

In general, you may need to apply arbitrary functions to the expressions you get while tracing an evaluation. `TraceScan[`*f*`, `*expr*`, …]` applies *f* to each expression that arises. The expression is wrapped with `HoldForm` to prevent it from evaluating.

In `TraceScan[`*f*`, `*expr*`, …]`, the function *f* is applied to expressions before they are evaluated. `TraceScan[`*f*`, `*expr*`, `*patt*`, `*fp*`]` applies *f* before evaluation, and *fp* after evaluation.

The Evaluation Stack

Throughout any computation, *Mathematica* maintains an *evaluation stack* containing the expressions it is currently evaluating. You can use the function `Stack` to look at the stack. This means, for example, that if you interrupt *Mathematica* in the middle of a computation, you can use `Stack` to find out what *Mathematica* is doing.

The expression that *Mathematica* most recently started to evaluate always appears as the last element of the evaluation stack. The previous elements of the stack are the other expressions whose evaluation is currently in progress.

Thus at the point when x is being evaluated, the stack associated with the evaluation of an expression like $f[g[x]]$ will have the form $\{f[g[x]], g[x], x\}$.

Stack[_] gives the expressions that are being evaluated at the time when it is called, in this case including the Print function.

```
In[1]:= f[g[Print[Stack[_]]]];

   {f[g[Print[Stack[_]]]];, f[g[Print[Stack[_]]]],
    g[Print[Stack[_]]], Print[Stack[_]]}
```

Stack[] gives the tags associated with the evaluations that are being done when it is called.

```
In[2]:= f[g[Print[Stack[]]]];

   {CompoundExpression, f, g, Print}
```

In general, you can think of the evaluation stack as showing what functions called what other functions to get to the point *Mathematica* is at in your computation. The sequence of expressions corresponds to the first elements in the successively nested lists returned by Trace with the option TraceAbove set to True.

Stack[]	give a list of the tags associated with evaluations that are currently being done
Stack[_]	give a list of all expressions currently being evaluated
Stack[*form*]	include only expressions which match *form*

Looking at the evaluation stack.

It is rather rare to call Stack directly in your main *Mathematica* session. More often, you will want to call Stack in the middle of a computation. Typically, you can do this from within a dialog, or subsidiary session, as discussed in "Dialogs".

Here is the standard recursive definition of the factorial function.

```
In[3]:= fac[1] = 1; fac[n_] := n fac[n - 1]
```

This evaluates fac[10], starting a dialog when it encounters fac[4].

```
In[4]:= TraceDialog[fac[10], fac[4]]

   TraceDialog::dgbgn : Entering Dialog; use Return[] to exit.

Out[4]= fac[4]
```

This shows what objects were being evaluated when the dialog was started.

```
In[5]:= Stack[]
Out[5]= {TraceDialog, Times, Times, Times, Times, Times, Times, fac}
```

This ends the dialog.

```
In[6]:= Return[]
        TraceDialog::dgend: Exiting Dialog.
Out[6]= 3 628 800
```

In the simplest cases, the *Mathematica* evaluation stack is set up to record *all* expressions currently being evaluated. Under some circumstances, however, this may be inconvenient. For example, executing `Print[Stack[]]` will always show a stack with `Print` as the last function.

The function `StackInhibit` allows you to avoid this kind of problem. `StackInhibit[`*expr*`]` evaluates *expr* without modifying the stack.

`StackInhibit` prevents `Print` from being included on the stack.

```
In[7]:= f[g[StackInhibit[Print[Stack[]]]]];
Out[7]= {CompoundExpression, f, g}
```

Functions like `TraceDialog` automatically call `StackInhibit` each time they start a dialog. This means that `Stack` does not show functions that are called within the dialog, only those outside.

`StackInhibit[`*expr*`]`	evaluate *expr* without modifying the stack
`StackBegin[`*expr*`]`	evaluate *expr* with a fresh stack
`StackComplete[`*expr*`]`	evaluate *expr* with intermediate expressions in evaluation chains included on the stack

Controlling the evaluation stack.

By using `StackInhibit` and `StackBegin`, you can control which parts of the evaluation process are recorded on the stack. `StackBegin[`*expr*`]` evaluates *expr*, starting a fresh stack. This means that during the evaluation of *expr*, the stack does not include anything outside the `StackBegin`. Functions like `TraceDialog[`*expr*`, …]` call `StackBegin` before they begin evaluating *expr*, so that the stack shows how *expr* is evaluated, but not how `TraceDialog` was called.

StackBegin[*expr*] uses a fresh stack in the evaluation of *expr*.

```
In[8]:= f[StackBegin[g[h[StackInhibit[Print[Stack[]]]]]]]

        {g, h}

Out[8]= f[g[h[Null]]]
```

Stack normally shows you only those expressions that are currently being evaluated. As a result, it includes only the latest form of each expression. Sometimes, however, you may find it useful also to see earlier forms of the expressions. You can do this using StackComplete.

What StackComplete[*expr*] effectively does is to keep on the stack the complete evaluation chain for each expression that is currently being evaluated. In this case, the stack corresponds to the sequence of expressions obtained from Trace with the option TraceBackward -> All as well as TraceAbove -> True.

Controlling Infinite Evaluation

The general principle that *Mathematica* follows in evaluating expressions is to go on applying transformation rules until the expressions no longer change. This means, for example, that if you make an assignment like x = x + 1, *Mathematica* should go into an infinite loop. In fact, *Mathematica* stops after a definite number of steps, determined by the value of the global variable $RecursionLimit. You can always stop *Mathematica* earlier by explicitly interrupting it.

This assignment could cause an infinite loop. *Mathematica* stops after a number of steps determined by $RecursionLimit.

```
In[1]:= x = x + 1

        $RecursionLimit::reclim : Recursion depth of 256 exceeded. ≫

Out[1]= 255 + Hold[1 + x]
```

When *Mathematica* stops without finishing evaluation, it returns a held result. You can continue the evaluation by explicitly calling ReleaseHold.

```
In[2]:= ReleaseHold[%]

        $RecursionLimit::reclim : Recursion depth of 256 exceeded. ≫

Out[2]= 510 + Hold[1 + x]
```

`$RecursionLimit`	maximum depth of the evaluation stack
`$IterationLimit`	maximum length of an evaluation chain

Global variables that limit infinite evaluation.

Here is a circular definition, whose evaluation is stopped by `$IterationLimit`.

```
In[3]:= {a, b} = {b, a}

    $IterationLimit::itlim : Iteration limit of 4096 exceeded. ≫

    $IterationLimit::itlim : Iteration limit of 4096 exceeded. ≫

Out[3]= {Hold[b], Hold[a]}
```

The variables `$RecursionLimit` and `$IterationLimit` control the two basic ways that an evaluation can become infinite in *Mathematica*. `$RecursionLimit` limits the maximum depth of the evaluation stack, or equivalently, the maximum nesting depth that would occur in the list structure produced by `Trace`. `$IterationLimit` limits the maximum length of any particular evaluation chain, or the maximum length of any single list in the structure produced by `Trace`.

`$RecursionLimit` and `$IterationLimit` are by default set to values that are appropriate for most computations, and most computer systems. You can, however, reset these variables to any integer (above a lower limit), or to `Infinity`. Note that on most computer systems, you should never set `$RecursionLimit = Infinity`, as discussed in "Memory Management".

This resets `$RecursionLimit` and `$IterationLimit` to `20`.

```
In[4]:= $RecursionLimit = $IterationLimit = 20

Out[4]= 20
```

Now infinite definitions like this are stopped after just 20 steps.

```
In[5]:= t = {t}

    $RecursionLimit::reclim : Recursion depth of 20 exceeded. ≫

Out[5]= {{{{{{{{{{{{{{{{{{{{Hold[{t}]}}}}}}}}}}}}}}}}}}}}
```

Without an end condition, this recursive definition leads to infinite computations.

```
In[6]:= fn[n_] := {fn[n - 1], n}
```

A fairly large structure is built up before the computation is stopped.

```
In[7]:= fn[10]

    $RecursionLimit::reclim : Recursion depth of 20 exceeded. ≫

Out[7]= {{{{{{{{{{{{{{{{{{{Hold[fn[-8 - 1]], -8}, -7}, -6}, -5}, -4}, -3}, -2}, -1}, 0}, 1}, 2},
          3}, 4}, 5}, 6}, 7}, 8}, 9}, 10}
```

Here is another recursive definition.

```
In[8]:= fm[n_] := fm[n - 1]
```

In this case, no complicated structure is built up, and the computation is stopped by `$IterationLimit`.

```
In[9]:= fm[0]

    $IterationLimit::itlim : Iteration limit of 20 exceeded. ≫

Out[9]= Hold[fm[-19 - 1]]
```

It is important to realize that infinite loops can take up not only time but also computer memory. Computations limited by `$IterationLimit` do not normally build up large intermediate structures. But those limited by `$RecursionLimit` often do. In many cases, the size of the structures produced is a linear function of the value of `$RecursionLimit`. But in some cases, the size can grow exponentially, or worse, with `$RecursionLimit`.

An assignment like `x = x + 1` is obviously circular. When you set up more complicated recursive definitions, however, it can be much more difficult to be sure that the recursion terminates, and that you will not end up in an infinite loop. The main thing to check is that the right-hand sides of your transformation rules will always be different from the left-hand sides. This ensures that evaluation will always "make progress", and *Mathematica* will not simply end up applying the same transformation rule to the same expression over and over again.

Some of the trickiest cases occur when you have rules that depend on complicated `/;` conditions (see "Putting Constraints on Patterns"). One particularly awkward case is when the condition involves a "global variable". *Mathematica* may think that the evaluation is finished because the expression did not change. However, a side effect of some other operation could change the value of the global variable, and so should lead to a new result in the evaluation. The best way to avoid this kind of difficulty is not to use global variables in `/;` conditions. If all else fails, you can type `Update[`*s*`]` to tell *Mathematica* to update all expressions involving *s*. `Update[]` tells *Mathematica* to update absolutely all expressions.

Interrupts and Aborts

"Interrupting Calculations" describes how you can interrupt a *Mathematica* computation by pressing appropriate keys on your keyboard.

In some cases, you may want to simulate such interrupts from within a *Mathematica* program. In general, executing `Interrupt[]` has the same effect as pressing interrupt keys. On a typical system, a menu of options is displayed, as discussed in "Interrupting Calculations".

`Interrupt[]`	interrupt a computation
`Abort[]`	abort a computation
`CheckAbort[`*expr*`,`*failexpr*`]`	evaluate *expr* and return the result, or *failexpr* if an abort occurs
`AbortProtect[`*expr*`]`	evaluate *expr*, masking the effect of aborts until the evaluation is complete

Interrupts and aborts.

The function `Abort[]` has the same effect as interrupting a computation, and selecting the `abort` option in the interrupt menu.

You can use `Abort[]` to implement an "emergency stop" in a program. In almost all cases, however, you should try to use functions like `Return` and `Throw`, which lead to more controlled behavior.

`Abort` terminates the computation, so only the first `Print` is executed.

```
In[1]:= Print[a]; Abort[]; Print[b]

           a

Out[1]= $Aborted
```

If you abort at any point during the evaluation of a *Mathematica* expression, *Mathematica* normally abandons the evaluation of the whole expression, and returns the value `$Aborted`.

You can, however, "catch" aborts using the function `CheckAbort`. If an abort occurs during the evaluation of *expr* in `CheckAbort[`*expr*`, `*failexpr*`]`, then `CheckAbort` returns *failexpr*, but the abort propagates no further. Functions like `Dialog` use `CheckAbort` in this way to contain the effect of aborts.

`CheckAbort` catches the abort, prints `c` and returns the value `aborted`.

```
In[2]:= CheckAbort[Print[a]; Abort[]; Print[b], Print[c]; aborted]

        a

        c

Out[2]= aborted
```

The effect of the `Abort` is contained by `CheckAbort`, so `b` is printed.

```
In[3]:= CheckAbort[Print[a]; Abort[], Print[c]; aborted]; Print[b]

        a

        c

        b
```

When you construct sophisticated programs in *Mathematica*, you may sometimes want to guarantee that a particular section of code in a program cannot be aborted, either interactively or by calling `Abort`. The function `AbortProtect` allows you to evaluate an expression, saving up any aborts until after the evaluation of the expression is complete.

The `Abort` is saved up until `AbortProtect` is finished.

```
In[4]:= AbortProtect[Abort[]; Print[a]]; Print[b]

        a

Out[4]= $Aborted
```

The `CheckAbort` sees the abort, but does not propagate it further.

```
In[5]:= AbortProtect[Abort[]; CheckAbort[Print[a], x]]; Print[b]

        b
```

Even inside `AbortProtect`, `CheckAbort` will see any aborts that occur, and will return the appropriate *failexpr*. Unless this *failexpr* itself contains `Abort[]`, the aborts will be “absorbed” by the `CheckAbort`.

Compiling *Mathematica* Expressions

If you make a definition like `f[x_] := x Sin[x]`, *Mathematica* will store the expression `x Sin[x]` in a form that can be evaluated for any `x`. Then when you give a particular value for `x`, *Mathematica* substitutes this value into `x Sin[x]`, and evaluates the result. The internal code that *Mathematica* uses to perform this evaluation is set up to work equally well whether the value you give for `x` is a number, a list, an algebraic object, or any other kind of expression.

Having to take account of all these possibilities inevitably makes the evaluation process slower. However, if *Mathematica* could *assume* that `x` will be a machine number, then it could avoid many steps, and potentially evaluate an expression like `x Sin[x]` much more quickly.

Using `Compile`, you can construct *compiled functions* in *Mathematica*, which evaluate *Mathematica* expressions assuming that all the parameters which appear are numbers (or logical variables). `Compile[{`x_1`, `x_2`, …}, `*expr*`]` takes an expression *expr* and returns a "compiled function" which evaluates this expression when given arguments $x_1, x_2, \ldots$.

In general, `Compile` creates a `CompiledFunction` object which contains a sequence of simple instructions for evaluating the compiled function. The instructions are chosen to be close to those found in the machine code of a typical computer, and can thus be executed quickly.

`Compile[{`x_1`,`x_2`,…},`*expr*`]`	create a compiled function which evaluates *expr* for numerical values of the x_i

Creating compiled functions.

This defines `f` to be a pure function which evaluates `x Sin[x]` for any `x`.

```
In[1]:=  f = Function[{x}, x Sin[x]]
Out[1]=  Function[{x}, x Sin[x]]
```

This creates a compiled function for evaluating `x Sin[x]`.

```
In[2]:=  fc = Compile[{x}, x Sin[x]]
Out[2]=  CompiledFunction[{x}, x Sin[x], -CompiledCode-]
```

`f` and `fc` yield the same results, but `fc` runs faster when the argument you give is a number.

```
In[3]:=  {f[2.5], fc[2.5]}
Out[3]=  {1.49618, 1.49618}
```

`Compile` is useful in situations where you have to evaluate a particular numerical or logical expression many times. By taking the time to call `Compile`, you can get a compiled function which can be executed more quickly than an ordinary *Mathematica* function.

For simple expressions such as `x Sin[x]`, there is usually little difference between the execution speed for ordinary and compiled functions. However, as the size of the expressions involved increases, the advantage of compilation also increases. For large expressions, compilation can speed up execution by a factor as large as 20.

Compilation makes the biggest difference for expressions containing a large number of simple, say arithmetic, functions. For more complicated functions, such as `BesselK` or `Eigenvalues`, most of the computation time is spent executing internal *Mathematica* algorithms, on which compilation has no effect.

This creates a compiled function for finding values of the tenth Legendre polynomial. The `Evaluate` tells *Mathematica* to construct the polynomial explicitly before doing compilation.

```
In[4]:= pc = Compile[{x}, Evaluate[LegendreP[10, x]]]
```

Out[4]= CompiledFunction$\left[\{x\}, \frac{1}{256}\left(-63 + 3465\, x^2 - 30\,030\, x^4 + 90\,090\, x^6 - 109\,395\, x^8 + 46\,189\, x^{10}\right), \text{-CompiledCode-}\right]$

This finds the value of the tenth Legendre polynomial with argument `0.4`.

```
In[5]:= pc[0.4]
Out[5]= 0.0968391
```

This uses built-in numerical code.

```
In[6]:= LegendreP[10, 0.4]
Out[6]= 0.0968391
```

Even though you can use compilation to speed up numerical functions that you write, you should still try to use built-in *Mathematica* functions whenever possible. Built-in functions will usually run faster than any compiled *Mathematica* programs you can create. In addition, they typically use more extensive algorithms, with more complete control over numerical precision and so on.

You should realize that built-in *Mathematica* functions quite often themselves use `Compile`. Thus, for example, `NIntegrate` by default automatically uses `Compile` on the expression you tell it to integrate. Similarly, functions like `Plot` and `Plot3D` use `Compile` on the expressions you ask them to plot. Built-in functions that use `Compile` typically have the option `Compiled`. Setting `Compiled -> False` tells the functions not to use `Compile`.

`Compile[{{`x_1`,`t_1`},{`x_2`,`t_2`},…},`*expr*`]`	compile *expr* assuming that x_i is of type t_i
`Compile[{{`x_1`,`t_1`,`n_1`},{`x_2`,`t_2`,`n_2`},…},`*expr*`]`	compile *expr* assuming that x_i is a rank n_i array of objects each of type t_i
`Compile[`*vars*`,`*expr*`,{{`p_1`,`pt_1`},…}]`	compile *expr*, assuming that subexpressions which match p_i are of type pt_i
`_Integer`	machine-size integer
`_Real`	machine-precision approximate real number
`_Complex`	machine-precision approximate complex number
`True|False`	logical variable

Specifying types for compilation.

`Compile` works by making assumptions about the types of objects that occur in evaluating the expression you give. The default assumption is that all variables in the expression are approximate real numbers.

`Compile` nevertheless also allows integers, complex numbers and logical variables (`True` or `False`), as well as arrays of numbers. You can specify the type of a particular variable by giving a pattern which matches only values that have that type. Thus, for example, you can use the pattern `_Integer` to specify the integer type. Similarly, you can use `True | False` to specify a logical variable that must be either `True` or `False`.

This compiles the expression `5 i + j` with the assumption that `i` and `j` are integers.

```
In[7]:= Compile[{{i, _Integer}, {j, _Integer}}, 5 i + j]
Out[7]= CompiledFunction[{i, j}, 5 i + j, -CompiledCode-]
```

This yields an integer result.

```
In[8]:= %[8, 7]
Out[8]= 47
```

This compiles an expression that performs an operation on a matrix of integers.

```
In[9]:= Compile[{{m, _Integer, 2}}, Apply[Plus, Flatten[m]]]
Out[9]= CompiledFunction[{m}, Plus @@ Flatten[m], -CompiledCode-]
```

The list operations are now carried out in a compiled way, and the result is an integer.

```
In[10]:= %[{{1, 2, 3}, {7, 8, 9}}]
Out[10]= 30
```

The types that `Compile` handles correspond essentially to the types that computers typically handle at a machine-code level. Thus, for example, `Compile` can handle approximate real numbers that have machine precision, but it cannot handle arbitrary-precision numbers. In addition, if you specify that a particular variable is an integer, `Compile` generates code only for the case when the integer is of "machine size", typically between $\pm 2^{31}$.

When the expression you ask to compile involves only standard arithmetic and logical operations, `Compile` can deduce the types of objects generated at every step simply from the types of the input variables. However, if you call other functions, `Compile` will typically not know what type of value they return. If you do not specify otherwise, `Compile` assumes that any other function yields an approximate real number value. You can, however, also give an explicit list of patterns, specifying what type to assume for an expression that matches a particular pattern.

This defines a function which yields an integer result when given an integer argument.

```
In[11]:= com[i_] := Binomial[2 i, i]
```

This compiles `x^com[i]` using the assumption that `com[_]` is always an integer.

```
In[12]:= Compile[{x, {i, _Integer}}, x^com[i], {{com[_], _Integer}}]
Out[12]= CompiledFunction[{x, i}, x^com[i], -CompiledCode-]
```

This evaluates the compiled function.

```
In[13]:= %[5.6, 1]
Out[13]= 31.36
```

The idea of `Compile` is to create a function which is optimized for certain types of arguments. `Compile` is nevertheless set up so that the functions it creates work with whatever types of arguments they are given. When the optimization cannot be used, a standard *Mathematica* expression is evaluated to find the value of the function.

Here is a compiled function for taking the square root of a variable.

```
In[14]:= sq = Compile[{x}, Sqrt[x]]

Out[14]= CompiledFunction[{x}, √x , -CompiledCode-]
```

If you give a real number argument, optimized code is used.

```
In[15]:= sq[4.5]

Out[15]= 2.12132
```

The compiled code cannot be used, so *Mathematica* prints a warning, then just evaluates the original symbolic expression.

```
In[16]:= sq[1 + u]

    CompiledFunction::cfsa: Argument 1+u at position 1 should be a machine-size real number. ≫

Out[16]= √(1 + u)
```

The compiled code generated by `Compile` must make assumptions not only about the types of arguments you will supply, but also about the types of all objects that arise during the execution of the code. Sometimes these types depend on the actual *values* of the arguments you specify. Thus, for example, `Sqrt[x]` yields a real number result for real x if x is not negative, but yields a complex number if x is negative.

`Compile` always makes a definite assumption about the type returned by a particular function. If this assumption turns out to be invalid in a particular case when the code generated by `Compile` is executed, then *Mathematica* simply abandons the compiled code in this case, and evaluates an ordinary *Mathematica* expression to get the result.

The compiled code does not expect a complex number, so *Mathematica* has to revert to explicitly evaluating the original symbolic expression.

```
In[17]:= sq[-4.5]

         CompiledFunction::cfn :
           Numerical error encountered at instruction 2; proceeding with uncompiled evaluation. ≫

Out[17]= 0. + 2.12132 i
```

An important feature of `Compile` is that it can handle not only mathematical expressions, but also various simple *Mathematica* programs. Thus, for example, `Compile` can handle conditionals and control flow structures.

In all cases, `Compile[`*vars*`, `*expr*`]` holds its arguments unevaluated. This means that you can explicitly give a "program" as the expression to compile.

This creates a compiled version of a *Mathematica* program which implements Newton's approximation to square roots.

```
In[18]:= newt = Compile[{x, {n, _Integer}}, Module[{t}, t = x; Do[t = (t + x / t) / 2, {n}]; t]]
```

Out[18]= CompiledFunction$\left[\{x, n\}, \text{Module}\left[\{t\}, t = x; \text{Do}\left[t = \frac{1}{2}\left(t + \frac{x}{t}\right), \{n\}\right]; t\right], \text{-CompiledCode-}\right]$

This executes the compiled code.

```
In[19]:= newt[2.4, 6]
Out[19]= 1.54919
```

Manipulating Compiled Code

If you use compiled code created by `Compile` only within *Mathematica* itself, then you should never need to know the details of its internal form. Nevertheless, the compiled code can be represented by an ordinary *Mathematica* expression, and it is sometimes useful to manipulate it.

For example, you can take compiled code generated by `Compile`, and feed it to external programs or devices. You can also create `CompiledFunction` objects yourself, then execute them in *Mathematica*.

In all of these cases, you need to know the internal form of `CompiledFunction` objects. The first element of a `CompiledFunction` object is always a list of patterns which specifies the types of arguments accepted by the object. The fifth element of a `CompiledFunction` object is a *Mathematica* pure function that is used if the compiled code instruction stream fails for any reason to give a result.

`CompiledFunction[` $\{arg_1, arg_2, \ldots\}, \{reg_1, reg_2, \ldots\}, \{n_l, n_i, n_r, n_c, n_t\}, instr, func$`]`	compiled code taking arguments of type arg_i and executing the instruction stream $instr$ using n_k registers of type k

The structure of a compiled code object.

This shows the explicit form of the compiled code generated by `Compile`.

```
In[1]:= Compile[{x}, x^2] // InputForm

Out[1]//InputForm= CompiledFunction[{_Real}, {{3, 0, 0}, {3, 0, 1}}, {0, 1, 2, 0, 0},
                    {{1, 5}, {7, 2, 0}, {94, 264, 3, 0, 0, 2, 0, 0, 3, 0, 1}, {2}},
                    Function[{x}, x^2], Evaluate]
```

The instruction stream in a `CompiledFunction` object consists of a list of instructions for a simple idealized computer. The computer is assumed to have numbered "registers", on which operations can be performed. There are five basic types of registers: logical, integer, real, complex and tensor. For each of these basic types it is then possible to have either a single scalar register or an array of registers of any rank. A list of the total number of registers of each type required to evaluate a particular `CompiledFunction` object is given as the second element of the object.

The actual instructions in the compiled code object are given as lists. The first element is an integer "opcode" which specifies what operation should be performed. Subsequent elements are either the numbers of registers of particular types, or literal constants. Typically the last element of the list is the number of a "destination register", into which the result of the operation should be put.

Appendix: Language Structure

Basic Objects

Expressions

Expressions are the main type of data in *Mathematica*.

Expressions can be written in the form h`[`e_1`,` e_2`,` …`]`. The object h is known generically as the *head* of the expression. The e_i are termed the *elements* of the expression. Both the head and the elements may themselves be expressions.

The *parts* of an expression can be referred to by numerical indices. The head has index 0; element e_i has index i. `Part[`*expr*`,` i`]` or *expr*`[[`i`]]` gives the part of *expr* with index i. Negative indices count from the end.

`Part[`*expr*`,` i_1`,` i_2`,` …`]`, *expr*`[[`i_1`,` i_2`,` …`]]`, or `Extract[`*expr*`, {`i_1`,` i_2`,` …`}]` gives the piece of *expr* found by successively extracting parts of subexpressions with indices i_1, i_2, …. If you think of expressions as trees, the indices specify which branch to take at each node as you descend from the root.

The pieces of an expression that are specified by giving a sequence of exactly n indices are defined to be at *level* n in the expression. You can use levels to determine the domain of application of functions like `Map`. Level 0 corresponds to the whole expression.

The *depth* of an expression is defined to be the maximum number of indices needed to specify any part of the expression, plus one. A negative level number $-n$ refers to all parts of an expression that have depth n.

Symbols

Symbols are the basic named objects in *Mathematica*.

The name of a symbol must be a sequence of letters, letter-like forms and digits, not starting with a digit. Uppercase and lowercase letters are always distinguished in *Mathematica*.

aaaaa	user-defined symbol
Aaaaa	system-defined symbol
$*Aaaa*	global or internal system-defined symbol
aaaa$	symbol renamed in a scoping construct
aa$*nn*	unique local symbol generated in a module

Conventions for symbol names.

Essentially all system-defined symbols have names that contain only ordinary English letters, together with numbers and `$`. The exceptions are π, ∞, e, i and j.

System-defined symbols conventionally have names that consist of one or more complete English words. The first letter of each word is capitalized, and the words are run together.

Once created, an ordinary symbol in *Mathematica* continues to exist unless it is explicitly removed using `Remove`. However, symbols created automatically in scoping constructs such as `Module` carry the attribute `Temporary` which specifies that they should automatically be removed as soon as they no longer appear in any expression.

When a new symbol is to be created, *Mathematica* first applies any value that has been assigned to `$NewSymbol` to strings giving the name of the symbol, and the context in which the symbol would be created.

If the message `General::newsym` is switched on, then *Mathematica* reports new symbols that are created. This message is switched off by default. Symbols created automatically in scoping constructs are not reported.

Contexts

The full name of any symbol in *Mathematica* consists of two parts: a *context* and a *short name*. The full name is written in the form *context*`*name*. The context *context*` can contain the same characters as the short name. It may also contain any number of context mark characters `, and must end with a context mark.

At any point in a *Mathematica* session, there is a *current context* `$Context` and a *context search path* `$ContextPath` consisting of a list of contexts. Symbols in the current context, or in contexts on the context search path, can be specified by giving only their short names, provided they are not shadowed by another symbol with the same short name.

name	search `$ContextPath`, then `$Context`; create in `$Context` if necessary
` *name*	search `$Context` only; create there if necessary
context ` *name*	search *context* only; create there if necessary
` *context* ` *name*	search `$Context`` *context* only; create there if necessary

Contexts used for various specifications of symbols.

With *Mathematica* packages, it is conventional to associate contexts whose names correspond to the names of the packages. Packages typically use `BeginPackage` and `EndPackage` to define objects in the appropriate context, and to add the context to the global `$ContextPath`. `EndPackage` prints a warning about any symbols that were created in a package but which are "shadowed" by existing symbols on the context search path.

The context is included in the printed form of a symbol only if it would be needed to specify the symbol *at the time of printing*.

Atomic Objects

All expressions in *Mathematica* are ultimately made up from a small number of basic or atomic types of objects.

These objects have heads which are symbols that can be thought of as "tagging" their types. The objects contain "raw data", which can usually be accessed only by functions specific to the particular type of object. You can extract the head of the object using `Head`, but you cannot directly extract any of its other parts.

`Symbol`	symbol (extract name using `SymbolName`)
`String`	character string "*cccc*" (extract characters using `Characters`)
`Integer`	integer (extract digits using `IntegerDigits`)
`Real`	approximate real number (extract digits using `RealDigits`)
`Rational`	rational number (extract parts using `Numerator` and `Denominator`)
`Complex`	complex number (extract parts using `Re` and `Im`)

Atomic objects.

Atomic objects in *Mathematica* are considered to have depth 0 and yield `True` when tested with `AtomQ`.

Numbers

`Integer`	integer *nnnn*
`Real`	approximate real number *nnn*.*nnn*
`Rational`	rational number *nnn* / *nnn*
`Complex`	complex number *nnn* + *nnn* `I`

Basic types of numbers.

All numbers in *Mathematica* can contain any number of digits. *Mathematica* does exact computations when possible with integers and rational numbers, and with complex numbers whose real and imaginary parts are integers or rational numbers.

There are two types of approximate real numbers in *Mathematica*: *arbitrary precision* and *machine precision*. In manipulating arbitrary-precision numbers, *Mathematica* tries to modify the precision so as to ensure that all digits actually given are correct.

With machine-precision numbers, all computations are done to the same fixed precision, so some digits given may not be correct.

Unless otherwise specified, *Mathematica* treats as machine-precision numbers all approximate real numbers that lie between `$MinMachineNumber` and `$MaxMachineNumber` and that are input with less than `$MachinePrecision` digits.

In `InputForm`, *Mathematica* prints machine-precision numbers with `$MachinePrecision` digits, except when trailing digits are zero.

In any implementation of *Mathematica*, the magnitudes of numbers (except 0) must lie between `$MinNumber` and `$MaxNumber`. Numbers with magnitudes outside this range are represented by `Underflow[]` and `Overflow[]`.

Character Strings

Character strings in *Mathematica* can contain any sequence of characters. They are input in the form `"`*ccccc*`"`.

The individual characters can be printable ASCII (with character codes between 32 and 126), or in general any 8- or 16-bit characters. *Mathematica* uses the Unicode character encoding for 16-bit characters.

In input form, 16-bit characters are represented when possible in the form `\[`*name*`]`, and otherwise as `\:`*nnnn*.

Null bytes can appear at any point within *Mathematica* strings.

Input Syntax

Entering Characters

- Enter it directly (e.g. `+`)
- Enter it by full name (e.g. `\[Alpha]`)
- Enter it by alias (e.g. Esc a Esc) (notebook front end only)
- Enter it by choosing from a palette (notebook front end only)
- Enter it by character code (e.g. `\:03b1`)

Typical ways to enter characters.

All printable ASCII characters can be entered directly. Those that are not alphanumeric are assigned explicit names in *Mathematica*, allowing them to be entered even on keyboards where they do not explicitly appear.

	`\[RawSpace]`	`;`	`\[RawSemicolon]`
`!`	`\[RawExclamation]`	`<`	`\[RawLess]`
`"`	`\[RawDoubleQuote]`	`=`	`\[RawEqual]`
`#`	`\[RawNumberSign]`	`>`	`\[RawGreater]`
`$`	`\[RawDollar]`	`?`	`\[RawQuestion]`
`%`	`\[RawPercent]`	`@`	`\[RawAt]`
`&`	`\[RawAmpersand]`	`[`	`\[RawLeftBracket]`
`'`	`\[RawQuote]`	`\`	`\[RawBackslash]`
`(`	`\[RawLeftParenthesis]`	`]`	`\[RawRightBracket]`
`)`	`\[RawRightParenthesis]`	`^`	`\[RawWedge]`
`*`	`\[RawStar]`	`_`	`\[RawUnderscore]`
`+`	`\[RawPlus]`	`` ` ``	`\[RawBackquote]`

,	`\[RawComma]`	{	`\[RawLeftBrace]`
-	`\[RawDash]`	|	`\[RawVerticalBar]`
.	`\[RawDot]`	}	`\[RawRightBrace]`
/	`\[RawSlash]`	~	`\[RawTilde]`
:	`\[RawColon]`		

Full names for non-alphanumeric printable ASCII characters.

All characters which are entered into the *Mathematica* kernel are interpreted according to the setting for the `CharacterEncoding` option for the stream from which they came.

`\[`*Name*`]`	a character with the specified full name
`\`*nnn*	a character with octal code *nnn*
`\.`*nn*	a character with hexadecimal code *nn*
`\:`*nnnn*	a character with hexadecimal code *nnnn*

Ways to enter characters.

Codes for characters can be generated using `ToCharacterCode`. The Unicode standard is followed, with various extensions.

8-bit characters have codes less than 256; 16-bit characters have codes between 256 and 65535. Approximately 900 characters are assigned explicit names in *Mathematica*. Other characters must be entered using their character codes.

`\\`	single backslash (decimal code 92)
`\ `	single space (decimal code 32)
`\"`	double-quote (decimal code 34)
`\b`	backspace or Ctrl+H (decimal code 8)
`\t`	tab or Ctrl+I (decimal code 9)
`\n`	newline or Ctrl+J (decimal code 10; full name `\[NewLine]`)
`\f`	form feed or Ctrl+L (decimal code 12)
`\r`	carriage return or Ctrl+M (decimal code 13)
`\000`	null byte (code 0)

Some special 8-bit characters.

Types of Input Syntax

The standard input syntax used by *Mathematica* is the one used by default in `InputForm` and `StandardForm`. You can modify the syntax by making definitions for `MakeExpression[`*expr*`, `*form*`]`.

Options can be set to specify what form of input should be accepted by a particular cell in a notebook or from a particular stream.

The input syntax in `TraditionalForm`, for example, is different from that in `InputForm` and `StandardForm`.

In general, what input syntax does is to determine how a particular string or collection of boxes should be interpreted as an expression. When boxes are set up, say with the notebook front end, there can be hidden `InterpretationBox` or `TagBox` objects which modify the interpretation of the boxes.

Character Strings

`"`*characters*`"`	a character string
`\"`	a literal " in a character string
`\\`	a literal \ in a character string
\ (at end of line)	ignore the following newline
`\!\(…\)`	a substring representing two-dimensional boxes

Entering character strings.

Character strings can contain any sequence of 8- or 16-bit characters. Characters entered by name or character code are stored the same as if they were entered directly.

In a notebook front end, text pasted into a string by default automatically has appropriate \ characters inserted so that the string stored in *Mathematica* reproduces the text that was pasted.

Within `\!\(…\)` any box structures represented using backslash sequences can be used.

`StringExpression` objects can be used to represent strings that contain symbolic constructs, such as pattern elements.

Symbol Names and Contexts

name	symbol name
\`*name*	symbol name in current context
context\`*name*	symbol name in specified context
context\`	context name
context1\`*context2*\`	compound context name
\`*context*\`	context relative to the current context

Symbol names and contexts.

Symbol names and contexts can contain any characters that are treated by *Mathematica* as letters or letter-like forms. They can contain digits but cannot start with them. Contexts must end in a backquote \`.

Numbers

digits	integer
digits.*digits*	approximate number
base^^*digits*	integer in specified base
base^^*digits*.*digits*	approximate number in specified base
*mantissa**^*n*	scientific notation ($mantissa \times 10^n$)
base^^*mantissa**^*n*	scientific notation in specified base ($mantissa \times base^n$)
number\`	machine-precision approximate number
number\`*s*	arbitrary-precision number with precision *s*
number\`\`*s*	arbitrary-precision number with accuracy *s*

Input forms for numbers.

Numbers can be entered with the notation *base*^^*digits* in any base from `2` to `36`. The base itself is given in decimal. For bases larger than `10`, additional digits are chosen from the letters `a`-`z` or `A`-`Z`. Upper- and lower-case letters are equivalent for these purposes. Floating-point numbers can be specified by including `.` in the *digits* sequence.

In scientific notation, *mantissa* can contain \` marks. The exponent *n* must always be an integer, specified in decimal.

The precision or accuracy *s* can be any real number; it does not need to be an integer.

In the form *base*`^^`*number*`` ` ``*s* the precision *s* is given in decimal, but it gives the effective number of digits of precision in the specified base, not in base 10.

An approximate number *x* is taken to be machine precision if the number of digits given in it is `Ceiling[$MachinePrecision] + 1` or less. If more digits are given, then *x* is taken to be an arbitrary-precision number. The accuracy of *x* is taken to be the number of digits that appear to the right of the decimal point, while its precision is taken to be `Log[10, Abs[x]] + Accuracy[x]`.

A number entered in the form ``0`` s`` is taken to have precision 0 and accuracy *s*.

Bracketed Objects

Bracketed objects use explicit left and right delimiters to indicate their extent. They can appear anywhere within *Mathematica* input, and can be nested in any way.

The delimiters in bracketed objects are *matchfix operators*. But since these delimiters explicitly enclose all operands, no precedence need be assigned to such operators.

(**any text**)	comment
(*expr*)	parenthesization: grouping of input

Bracketed objects without comma-separated elements.

Comments can be nested, and can continue for any number of lines. They can contain any 8- or 16-bit characters.

Parentheses must enclose a single complete expression; neither (*e*, *e*) nor () are allowed.

$\{e_1, e_2, \ldots\}$	`List`$[e_1, e_2, \ldots]$
$\langle e_1, e_2, \ldots\rangle$	`AngleBracket`$[e_1, e_2, \ldots]$
$\lfloor expr \rfloor$	`Floor`[*expr*]
$\lceil expr \rceil$	`Ceiling`[*expr*]
$\mid e_1, e_2, \ldots \mid$	`BracketingBar`$[e_1, e_2, \ldots]$
$\| e_1, e_2, \ldots \|$	`DoubleBracketingBar`$[e_1, e_2, \ldots]$
\\(*input*\\)	input or grouping of boxes

Bracketed objects that allow comma-separated elements.

The notation … is used to stand for any sequence of expressions.

{e_1, e_2, …} can include any number of elements, with successive elements separated by commas.

{} is `List[]`, a list with zero elements.

〈e_1, e_2, …〉 can be entered as `\[LeftAngleBracket]` e_1, e_2, … `\[RightAngleBracket]`.

The character `\[InvisibleComma]` can be used interchangeably with ordinary commas; the only difference is that `\[InvisibleComma]` will not be displayed.

When the delimiters are special characters, it is a convention that they are named `\[Left`*Name*`]` and `\[Right`*Name*`]`.

`\(…\)` is used to enter boxes using one-dimensional strings. Note that within the outermost `\(…\)` in a piece of input the syntax used is slightly different from outside, as described in "Input of Boxes".

h[e_1,e_2,…]	standard expression
e[[i_1,i_2,…]]	`Part`[e,i_1,i_2,…]
e⟦i_1,i_2,…⟧	`Part`[e,i_1,i_2,…]

Bracketed objects with heads.

Bracketed objects with heads explicitly delimit all their operands except the head. A precedence must be assigned to define the extent of the head.

The precedence of h[e] is high enough that !h[e] is interpreted as `Not`[h[e]]. However, h_s[e] is interpreted as (h_s)[e].

Two-Dimensional Input Forms

x^y	`Power[x, y]`	$\int_{x_{min}}^{x_{max}} y \, \mathbb{d}x$	`Integrate[y, {x, xmin, xmax}]`
$\frac{x}{y}$	`Divide[x, y]`	$\int_{x_{min}}^{x_{max}} \frac{y\,w}{z} \, \mathbb{d}x$	`Integrate[y w / z, {x, xmin, xmax}]`
$\sqrt{x}$	`Sqrt[x]`	$\sum_{x=x_{min}}^{x_{max}} y$	`Sum[y, {x, xmin, xmax}]`
$\sqrt[n]{x}$	`Power[x, 1 / n]`	$\prod_{x=x_{min}}^{x_{max}} y$	`Product[y, {x, xmin, xmax}]`
$\begin{matrix} a_{11} & a_{12} & \ldots \\ a_{21} & a_{22} & \ldots \end{matrix}$	`{{a11, a12, …}, {a21, a22, …}}`		
$\partial_x y$	`D[y, x]`		
$\partial_{x,\ldots} y$	`D[y, x, …]`		

Two-dimensional input forms with built-in evaluation rules.

Any array of expressions represented by a `GridBox` is interpreted as a list of lists. Even if the `GridBox` has only one row, the interpretation is still $\{\{a_1, a_2, \ldots\}\}$.

In the form $\int_{x_{min}}^{x_{max}} y\,w \frac{dx}{z}$ the limits x_{min} and x_{max} can be omitted, as can y and w.

x_y	`Subscript[x, y]`	$\overset{y}{x}$	`Overscript[x, y]`
x_+	`SubPlus[x]`	$\underset{y}{x}$	`Underscript[x, y]`
x_-	`SubMinus[x]`	$\overline{x}$	`OverBar[x]`
x_*	`SubStar[x]`	$\vec{x}$	`OverVector[x]`
x^+	`SuperPlus[x]`	$\tilde{x}$	`OverTilde[x]`
x^-	`SuperMinus[x]`	$\hat{x}$	`OverHat[x]`
x^*	`SuperStar[x]`	$\dot{x}$	`OverDot[x]`
$x^\dagger$	`SuperDagger[x]`	$\underline{x}$	`UnderBar[x]`

Two-dimensional input forms without built-in evaluation rules.

There is no issue of precedence for forms such as $\sqrt{x}$ and $\hat{x}$ in which operands are effectively spanned by the operator. For forms such as x_y and $x^\dagger$ a left precedence does need to be specified, so such forms are included in the main table of precedences above.

Input of Boxes

- Use a palette
- Use control keys

Ways to input boxes.

Control Keys

Ctrl+2 or Ctrl+@	square root
Ctrl+5 or Ctrl+%	switch to alternate position (e.g. subscript to superscript)
Ctrl+6 or Ctrl+^	superscript
Ctrl+7 or Ctrl+&	overscript
Ctrl+9 or Ctrl+(	begin a new cell within an existing cell
Ctrl+0 or Ctrl+)	end a new cell within an existing cell
Ctrl+- or Ctrl+_	subscript
Ctrl+= or Ctrl+Plus	underscript
Ctrl+Enter	create a new row in a table
Ctrl+,	create a new column in a table
Ctrl+.	expand current selection
Ctrl+/	fraction
Ctrl+Space	return from current position or state
Ctrl+←, Ctrl+→, Ctrl+↑, Ctrl+↓	move an object by minimal increments on the screen

Standard control keys.

On English-language keyboards both forms will work where alternates are given. On other keyboards the first form should work but the second may not.

Boxes Constructed from Text

When textual input that you give is used to construct boxes, as in `StandardForm` or `TraditionalForm` cells in a notebook, the input is handled slightly differently from when it is fed directly to the kernel.

The input is broken into *tokens*, and then each token is included in the box structure as a separate character string. Thus, for example, `xx + yyy` is broken into the tokens `"xx"`, `"+"`, `"yyy"`.

- symbol name (e.g. `x123`)
- number (e.g. `12.345`)
- operator (e.g. `+=`)
- spacing (e.g. ␣)
- character string (e.g. `"text"`)

Types of tokens in text used to construct boxes.

A `RowBox` is constructed to hold each operator and its operands. The nesting of `RowBox` objects is determined by the precedence of the operators in standard *Mathematica* syntax.

Note that spacing characters are not automatically discarded. Instead, each sequence of consecutive such characters is made into a separate token.

String-Based Input

`\(...\)`	input raw boxes
`\!\(...\)`	input and interpret boxes

Inputting raw and interpreted boxes.

Any textual input that you give between `\(` and `\)` is taken to specify boxes to construct. The boxes are only interpreted if you specify with `\!` that this should be done. Otherwise *x*`\^`*y* is left for example as `SuperscriptBox[`*x*`,` *y*`]`, and is not converted to `Power[`*x*`,` *y*`]`.

Within the outermost `\(...\)`, further `\(...\)` specify grouping and lead to the insertion of `RowBox` objects.

\(box_1 , box_2 , … \)	RowBox[box_1 , box_2 , …]
box_1 \^ box_2	SuperscriptBox[box_1 , box_2]
box_1 _ box_2	SubscriptBox[box_1 , box_2]
box_1 _ box_2 \% box_3	SubsuperscriptBox[box_1 , box_2 , box_3]
box_1 \& box_2	OverscriptBox[box_1 , box_2]
box_1 \+ box_2	UnderscriptBox[box_1 , box_2]
box_1 \+ box_2 \% box_3	UnderoverscriptBox[box_1 , box_2 , box_3]
box_1 \/ box_2	FractionBox[box_1 , box_2]
\@*box*	SqrtBox[*box*]
form \` *box*	FormBox[*box* , *form*]
* *input*	construct box by interpreting *input*
\␣	insert a space
\n	insert a newline
\t	indent at the beginning of a line

String-based ways of constructing raw boxes.

In string-based input between `\(` and `\)` spaces, tabs and newlines are discarded. `\␣` can be used to insert a single space. Special spacing characters such as `\[ThinSpace]`, `\[ThickSpace]` or `\[NegativeThinSpace]` are not discarded.

When you input typesetting forms into a string, the internal representation of the string uses the above forms. The front end displays the typeset form, but uses the `\(…\)` notation when saving the content to a file or when sending the string to the kernel for evaluation.

The Extent of Input Expressions

Mathematica will treat all input that you give on a single line as being part of the same expression.

Mathematica allows a single expression to continue for several lines. In general, it treats the input that you give on successive lines as belonging to the same expression whenever no complete expression would be formed without doing this.

Thus, for example, if one line ends with `=`, then *Mathematica* will assume that the expression must continue on the next line. It will do the same if for example parentheses or other matchfix operators remain open at the end of the line.

If at the end of a particular line the input you have given so far corresponds to a complete expression, then *Mathematica* will normally begin immediately to process that expression.

You can however explicitly tell *Mathematica* that a particular expression is incomplete by putting a `\` or a ⋱ (`\[Continuation]`) at the end of the line. *Mathematica* will then include the next line in the same expression, discarding any spaces or tabs that occur at the beginning of that line.

Special Input

`?`*symbol*	get information
`??`*symbol*	get more information
`?`s_1 s_2 …	get information on several objects
`!`*command*	execute an external command (text-based interface only)
`!!`*file*	display the contents of an external file (text-based interface only)

Special input lines.

In most implementations of *Mathematica*, you can give a line of special input anywhere in your input. The only constraint is that the special input must start at the beginning of a line.

Some implementations of *Mathematica* may not allow you to execute external commands using `!` *command*.

Front End Files

Notebook files as well as front end initialization files can contain a subset of standard *Mathematica* language syntax. This syntax includes:

- Any *Mathematica* expression in `FullForm`.
- Lists in `{…}` form. The operators `->`, `:>` and `&`. Function slots in `#` form.
- Various *Mathematica* operators such as `+`, `*`, `;`, etc.
- Special characters in `\[`*Name*`]`, `\:`*nnnn* or `\.`*xx* form.
- String representation of boxes involving `\(`, `\)` and other backslash operators.
- *Mathematica* comments delimited by `(*` and `*)`.

Some General Notations and Conventions

Function Names

The names of built-in functions follow some general guidelines.

- The name consists of complete English words, or standard mathematical abbreviations. American spelling is used.
- The first letter of each word is capitalized.
- Functions whose names end with `Q` usually "ask a question", and return either `True` or `False`.
- Mathematical functions that are named after people usually have names in *Mathematica* of the form *PersonSymbol*.

Function Arguments

The main expression or object on which a built-in function acts is usually given as the first argument to the function. Subsidiary parameters appear as subsequent arguments.

The following are exceptions:

- In functions like `Map` and `Apply`, the function to apply comes before the expression it is to be applied to.
- In scoping constructs such as `Module` and `Function`, local variables and parameter names come before bodies.
- In functions like `Write` and `Export`, the name of the file is given before the objects to be written to it.

For mathematical functions, arguments that are written as subscripts in standard mathematical notation are given before those that are written as superscripts.

Options

Some built-in functions can take *options*. Each option has a name, represented as a symbol, or in some cases a string. Options are set by giving rules of the form *name* -> *value* or *name* :> *value*. Such rules must appear after all the other arguments in a function. Rules for different options can be given in any order. If you do not explicitly give a rule for a particular option, a default setting for that option is used.

`Options[`*f*`]`	give the default rules for all options associated with *f*
`Options[`*expr*`]`	give the options set in a particular expression
`Options[`*expr*`,`*name*`]`	give the setting for the option *name* in an expression
`AbsoluteOptions[`*expr*`,`*name*`]`	give the absolute setting for *name*, even if its actual setting is `Automatic`
`SetOptions[`*f*`,`*name*`->`*value*`,…]`	set default rules for options associated with *f*
`CurrentValue[`*name*`]`	give the option setting for the front end option *name*; can be used on the left-hand side of an assignment to set the option

Operations on options.

Part Numbering

n	element *n* (starting at 1)
−*n*	element *n* from the end
0	head

Numbering of parts.

Sequence Specifications

`All`	all elements
`None`	no elements
n	elements 1 through n
$-n$	last n elements
$\{n\}$	element n only
$\{m, n\}$	elements m through n (inclusive)
$\{m, n, s\}$	elements m through n in steps of s

Specifications for sequences of parts.

The sequence specification $\{m, n, s\}$ corresponds to elements m, $m+s$, $m+2s$, $\ldots$, up to the largest element not greater than n.

Sequence specifications are used in the functions `Drop`, `Ordering`, `StringDrop`, `StringTake`, `Take` and `Thread`.

Level Specifications

n	levels 1 through n
`Infinity`	levels 1 through `Infinity`
$\{n\}$	level n only
$\{n_1, n_2\}$	levels n_1 through n_2
`Heads->True`	include heads of expressions
`Heads->False`	do not include heads of expressions

Level specifications.

The level in an expression corresponding to a non-negative integer n is defined to consist of parts specified by n indices. A negative level number $-n$ represents all parts of an expression that have depth n. The depth of an expression, `Depth[`*expr*`]`, is the maximum number of indices needed to specify any part, plus one. Levels *do not* include heads of expressions, except with the option setting `Heads -> True`. Level `0` is the whole expression. Level `-1` contains all symbols and other objects that have no subparts.

Ranges of levels specified by $\{n_1, n_2\}$ contain all parts that are neither above level n_1, nor below level n_2 in the tree. The n_i need not have the same sign. Thus, for example, `{2, -2}` specifies subexpressions which occur anywhere below the top level, but above the leaves, of the expression tree.

Level specifications are used by functions such as `Apply`, `Cases`, `Count`, `FreeQ`, `Level`, `Map`, `MapIndexed`, `Position`, `Replace` and `Scan`. Note, however, that the default level specifications are not the same for all of these functions.

Iterators

$\{i_{max}\}$	iterate i_{max} times
$\{i, i_{max}\}$	i goes from 1 to i_{max} in steps of 1
$\{i, i_{min}, i_{max}\}$	i goes from i_{min} to i_{max} in steps of 1
$\{i, i_{min}, i_{max}, di\}$	i goes from i_{min} to i_{max} in steps of di
$\{i, list\}$	i takes on the successive values in $list$
$\{i, i_{min}, i_{max}\}, \{j, j_{min}, j_{max}\}, \ldots$	i goes from i_{min} to i_{max}, and for each value of i, j goes from j_{min} to j_{max}, etc.

Iterator notation.

Iterators are used in such functions as `Sum`, `Table`, `Do` and `Range`.

The iteration parameters i_{min}, i_{max} and di do not need to be integers. The variable i is given a sequence of values starting at i_{min}, and increasing in steps of di, stopping when the next value of i would be greater than i_{max}. The iteration parameters can be arbitrary symbolic expressions, so long as $(i_{max} - i_{min}) / di$ is a number.

When several iteration variables are used, the limits for the later ones can depend on the values of earlier ones.

The variable i can be any symbolic expression; it need not be a single symbol. The value of i is automatically set up to be local to the iteration function. This is effectively done by wrapping a `Block` construct containing i around the iteration function.

The procedure for evaluating iteration functions is described in "Evaluation".

Scoping Constructs

`Function[{`*x*`,`…`},`*body*`]`	local parameters
lhs`->`*rhs* and *lhs*`:>`*rhs*	local pattern names
lhs`=`*rhs* and *lhs*`:=`*rhs*	local pattern names
`With[{`x`=`x_0`,`…`},`*body*`]`	local constants
`Module[{`*x*`,`…`},`*body*`]`	local variables
`Block[{`*x*`,`…`},`*body*`]`	local values of global variables
`DynamicModule[{`*x*`,`…`},`*body*`]`	local variables in a `Dynamic` interface

Scoping constructs in *Mathematica*. Functions in the first group scope variables lexically.

Scoping constructs allow the names or values of certain symbols to be local.

Some scoping contracts scope lexically, meaning that literal instances of the specified variables or patterns are replaced with appropriate values. When local variable names are required, symbols with names of the form *xxx* are generally renamed to *xxx*`$`. When nested scoping constructs are evaluated, new symbols are automatically generated in the inner scoping constructs so as to avoid name conflicts with symbols in outer scoping constructs.

When a transformation rule or definition is used, `ReplaceAll` (`/.`) is effectively used to replace the pattern names that appear on the right-hand side. Nevertheless, new symbols are generated when necessary to represent other objects that appear in scoping constructs on the right-hand side.

Each time it is evaluated, `Module` generates symbols with unique names of the form *xxx*`$`*nnn* as replacements for all local variables that appear in its body.

`Block` localizes the value of global variables. Any evaluations in the body of a block which rely on the global variable will use the locally specified value even if the variable does not explicitly appear in the body, but is only referenced through subsequent evaluation. The body of the `Block` may also make changes to the global variable, but any such changes will only persist until the `Block` has finished executing.

`DynamicModule` localizes its variables to each instance of the `DynamicModule` output in a notebook. This means each copy of a `DynamicModule` output created using copy and paste will use its own localized variables.

Ordering of Expressions

The canonical ordering of expressions used automatically with the attribute `Orderless` and in functions such as `Sort` satisfies the following rules:

- Integers, rational and approximate real numbers are ordered by their numerical values.
- Complex numbers are ordered by their real parts, and in the event of a tie, by the absolute values of their imaginary parts.
- Symbols are ordered according to their names, and in the event of a tie, by their contexts.
- Expressions are usually ordered by comparing their parts in a depth-first manner. Shorter expressions come first.
- Powers and products are treated specially, and are ordered to correspond to terms in a polynomial.
- Strings are ordered as they would be in a dictionary, with the uppercase versions of letters coming after lowercase ones.

Ordinary letters appear first, followed in order by script, Gothic, double-struck, Greek and Hebrew. Mathematical operators appear in order of decreasing precedence.

Mathematical Functions

The mathematical functions such as `Log[`*x*`]` and `BesselJ[`*n*`,` *x*`]` that are built into *Mathematica* have a number of features in common.

- They carry the attribute `Listable`, so that they are automatically "threaded" over any lists that appear as arguments.
- They carry the attribute `NumericFunction`, so that they are assumed to give numerical values when their arguments are numerical.
- They give exact results in terms of integers, rational numbers and algebraic expressions in special cases.

- Except for functions whose arguments are always integers, mathematical functions in *Mathematica* can be evaluated to any numerical precision, with any complex numbers as arguments. If a function is undefined for a particular set of arguments, it is returned in symbolic form in this case.
- Numerical evaluation leads to results of a precision no higher than can be justified on the basis of the precision of the arguments. Thus `N[Gamma[27 / 10], 100]` yields a high-precision result, but `N[Gamma[2.7], 100]` cannot.
- When possible, symbolic derivatives, integrals and series expansions of built-in mathematical functions are evaluated in terms of other built-in functions.

Mathematical Constants

Mathematical constants such as `E` and `Pi` that are built into *Mathematica* have the following properties:

- They do not have values as such.
- They have numerical values that can be found to any precision.
- They are treated as numeric quantities in `NumericQ` and elsewhere.
- They carry the attribute `Constant`, and so are treated as constants in derivatives.

Protection

Mathematica allows you to make assignments that override the standard operation and meaning of built-in *Mathematica* objects.

To make it difficult to make such assignments by mistake, most built-in *Mathematica* objects have the attribute `Protected`. If you want to make an assignment for a built-in object, you must first remove this attribute. You can do this by calling the function `Unprotect`.

There are a few fundamental *Mathematica* objects to which you absolutely cannot assign your own values. These objects carry the attribute `Locked`, as well as `Protected`. The `Locked` attribute prevents you from changing any of the attributes, and thus from removing the `Protected` attribute.

Abbreviated String Patterns

Functions such as `StringMatchQ`, `Names` and `Remove` allow you to give *abbreviated string patterns*, as well as full string patterns specified by `StringExpression`. Abbreviated string patterns can contain certain metacharacters, which can stand for sequences of ordinary characters.

`*`	zero or more characters
`@`	one or more characters excluding uppercase letters
`\\*`, etc.	literal `*`, etc.

Metacharacters used in abbreviated string patterns.

Evaluation

The Standard Evaluation Sequence

The following is the sequence of steps that *Mathematica* follows in evaluating an expression like $h[e_1, e_2 \ldots]$. Every time the expression changes, *Mathematica* effectively starts the evaluation sequence over again.

- If the expression is a raw object (e.g., `Integer`, `String`, etc.), leave it unchanged.
- Evaluate the head h of the expression.
- Evaluate each element e_i of the expression in turn. If h is a symbol with attributes `HoldFirst`, `HoldRest`, `HoldAll` or `HoldAllComplete`, then skip evaluation of certain elements.
- Unless h has attribute `HoldAllComplete`, strip the outermost of any `Unevaluated` wrappers that appear in the e_i.
- Unless h has attribute `SequenceHold`, flatten out all `Sequence` objects that appear among the e_i.
- If h has attribute `Flat`, then flatten out all nested expressions with head h.
- If h has attribute `Listable`, then thread through any e_i that are lists.
- If h has attribute `Orderless`, then sort the e_i into order.
- Unless h has attribute `HoldAllComplete`, use any applicable transformation rules associated with f that you have defined for objects of the form $h[f[e_1, \ldots], \ldots]$.

- Use any built-in transformation rules associated with f for objects of the form $h[f[e_1, \ldots], \ldots]$.
- Use any applicable transformation rules that you have defined for $h[f[e_1, e_2, \ldots], \ldots]$ or for $h[\ldots][\ldots]$.
- Use any built-in transformation rules for $h[e_1, e_2, \ldots]$ or for $h[\ldots][\ldots]$.

Nonstandard Argument Evaluation

There are a number of built-in *Mathematica* functions that evaluate their arguments in special ways. The control structure `While` is an example. The symbol `While` has the attribute `HoldAll`. As a result, the arguments of `While` are not evaluated as part of the standard evaluation process. Instead, the internal code for `While` evaluates the arguments in a special way. In the case of `While`, the code evaluates the arguments repeatedly, so as to implement a loop.

Control structures	arguments evaluated in a sequence determined by control flow (e.g., `CompoundExpression`)
Conditionals	arguments evaluated only when they correspond to branches that are taken (e.g., `If`, `Which`)
Logical operations	arguments evaluated only when they are needed in determining the logical result (e.g., `And`, `Or`)
Iteration functions	first argument evaluated for each step in the iteration (e.g., `Do`, `Sum`, `Plot`)
Tracing functions	form never evaluated (e.g., `Trace`)
Assignments	first argument only partially evaluated (e.g., `Set`, `AddTo`)
Pure functions	function body not evaluated (e.g., `Function`)
Scoping constructs	variable specifications not evaluated (e.g., `Module`, `Block`)
Holding functions	argument maintained in unevaluated form (e.g., `Hold`, `HoldPattern`)

Built-in functions that evaluate their arguments in special ways.

Logical Operations

In an expression of the form $e_1 \&\& e_2 \&\& e_3$ the e_i are evaluated in order. As soon as any e_i is found to be `False`, evaluation is stopped, and the result `False` is returned. This means that you can use the e_i to represent different "branches" in a program, with a particular branch being evaluated only if certain conditions are met.

The `Or` function works much like `And`; it returns `True` as soon as it finds any argument that is `True`. `Xor`, on the other hand, always evaluates *all* its arguments.

Iteration Functions

An iteration function such as `Do[`f`, {`i`, `i_{min}`, `i_{max}`}]` is evaluated as follows:

- The limits i_{min}, i_{max} are evaluated.
- The value of the iteration variable i is made local, effectively using `Block`.
- i_{min} and i_{max} are used to determine the sequence of values to be assigned to the iteration variable i.
- The iteration variable is successively set to each value, and f is evaluated in each case.
- The local values assigned to i are cleared.

If there are several iteration variables, the same procedure is followed for each variable in turn, for every value of all the preceding variables.

Unless otherwise specified, f is not evaluated until a specific value has been assigned to i, and is then evaluated for each value of i chosen. You can use `Evaluate[`f`]` to make f be evaluated immediately, rather than only after a specific value has been assigned to i.

Assignments

The left-hand sides of assignments are only partially evaluated.

- If the left-hand side is a symbol, no evaluation is performed.
- If the left-hand side is a function without hold attributes, the arguments of the function are evaluated, but the function itself is not evaluated.

The right-hand side is evaluated for immediate (`=`), but not for delayed (`:=`), assignments.

Any subexpression of the form `HoldPattern[`*expr*`]` that appears on the left-hand side of an assignment is not evaluated. When the subexpression is used for pattern matching, it matches as though it were *expr* without the `HoldPattern`.

Overriding Nonstandard Argument Evaluation

`f[...,Evaluate[expr],...]`	evaluates the argument *expr*, whether or not *f* has a `HoldFirst`, `HoldRest`, or `HoldAll` attribute specifying that it should be held

Overriding holding of arguments.

By using `Evaluate`, you can get any argument of a function evaluated immediately, even if the argument would usually be evaluated later under the control of the function. An exception to this is when the function has the `HoldComplete` attribute; in this case, the contents of the function are not modified by the evaluator.

Preventing Evaluation

Mathematica provides various functions which act as "wrappers" to prevent the expressions they contain from being evaluated.

`Hold[expr]`	treated as `Hold[expr]` in all cases
`HoldComplete[expr]`	treated as `HoldComplete[expr]` with upvalues disabled
`HoldForm[expr]`	treated as *expr* for printing
`HoldPattern[expr]`	treated as *expr* in rules, definitions and patterns
`Unevaluated[expr]`	treated as *expr* when arguments are passed to a function

Wrappers that prevent expressions from being evaluated.

Global Control of Evaluation

In the evaluation procedure described so far, two basic kinds of steps are involved:

- Iteration: evaluate a particular expression until it no longer changes.
- Recursion: evaluate subsidiary expressions needed to find the value of a particular expression.

Iteration leads to evaluation chains in which successive expressions are obtained by the application of various transformation rules.

`Trace` shows evaluation chains as lists, and shows subsidiary evaluations corresponding to recursion in sublists.

The expressions associated with the sequence of subsidiary evaluations which lead to an expression currently being evaluated are given in the list returned by `Stack[]`.

`$RecursionLimit`	maximum recursion depth
`$IterationLimit`	maximum number of iterations

Global variables controlling the evaluation of expressions.

Aborts

You can ask *Mathematica* to abort at any point in a computation, either by calling the function `Abort[]`, or by typing appropriate interrupt keys.

When asked to abort, *Mathematica* will terminate the computation as quickly as possible. If the answer obtained would be incorrect or incomplete, then *Mathematica* returns `$Aborted` instead of giving that answer.

Aborts can be caught using `CheckAbort`, and can be postponed using `AbortProtect`.

Patterns and Transformation Rules

Patterns

Patterns stand for classes of expressions. They contain *pattern objects* which represent sets of possible expressions.

_	any expression
*x*_	any expression, given the name *x*
x:*pattern*	a pattern, given the name *x*
pattern?*test*	a pattern that yields `True` when *test* is applied to its value
_*h*	any expression with head *h*
*x*_*h*	any expression with head *h*, given the name *x*
__	any sequence of one or more expressions
___	any sequence of zero or more expressions
*x*__ and *x*___	sequences of expressions, given the name *x*
__*h* and ___*h*	sequences of expressions, each with head *h*
*x*__*h* and *x*___*h*	sequences of expressions with head *h*, given the name *x*
`PatternSequence`[p_1, p_2, …]	a sequence of patterns
*x*_:*v*	an expression with default value *v*
*x*_*h*:*v*	an expression with head *h* and default value *v*
*x*_.	an expression with a globally defined default value
`Optional`[*x*_*h*]	an expression that must have head *h*, and has a globally defined default value
`Except`[*c*]	any expression except one that matches *c*
`Except`[*c*, *pattern*]	any expression matching *pattern*, except one that matches *c*
pattern..	a pattern repeated one or more times
pattern...	a pattern repeated zero or more times
`Repeated`[*pattern*, *spec*]	a pattern repeated according to *spec*
$pattern_1$ | $pattern_2$ | …	a pattern which matches at least one of the $pattern_i$
pattern/;*cond*	a pattern for which *cond* evaluates to `True`
`HoldPattern`[*pattern*]	a pattern not evaluated
`Verbatim`[*expr*]	an expression to be matched verbatim
`OptionsPattern`[]	a sequence of options
`Longest`[*pattern*]	the longest sequence consistent with *pattern*
`Shortest`[*pattern*]	the shortest sequence consistent with *pattern*

`Pattern` objects.

When several pattern objects with the same name occur in a single pattern, all the objects must stand for the same expression. Thus `f[x_, x_]` can stand for `f[2, 2]` but not `f[2, 3]`.

In a pattern object such as _*h*, the head *h* can be any expression, but cannot itself be a pattern.

A pattern object such as *x*__ stands for a *sequence* of expressions. So, for example, `f[x__]` can stand for `f[a, b, c]`, with `x` being `Sequence[a, b, c]`. If you use `x`, say in the result of a transformation rule, the sequence will be spliced into the function in which `x` appears. Thus `g[u, x, u]` would become `g[u, a, b, c, u]`.

When the pattern objects *x*_:*v* and *x*_. appear as arguments of functions, they represent arguments which may be omitted. When the argument corresponding to *x*_:*v* is omitted, *x* is taken to have value *v*. When the argument corresponding to *x*_. is omitted, *x* is taken to have a *default value* that is associated with the function in which it appears. You can specify this default value by making assignments for `Default[`*f*`]` and so on.

`Default[`*f*`]`	default value for *x*_. when it appears as any argument of the function *f*
`Default[`*f*`,`*n*`]`	default value for *x*_. when it appears as the *n*th argument (negative *n* count from the end)
`Default[`*f*`,`*n*`,`*tot*`]`	default value for the *n*th argument when there are a total of *tot* arguments

`Default` values.

A pattern like `f[x__, y__, z__]` can match an expression like `f[a, b, c, d, e]` with several different choices of `x`, `y` and `z`. The choices with `x` and `y` of minimum length are tried first. In general, when there are multiple __ or ___ in a single function, the case that is tried first takes all the __ and ___ to stand for sequences of minimum length, except the last one, which stands for "the rest" of the arguments.

When *x*_:*v* or *x*_. are present, the case that is tried first is the one in which none of them correspond to omitted arguments. Cases in which later arguments are dropped are tried next.

The order in which the different cases are tried can be changed using `Shortest` and `Longest`.

`Orderless`	$f[x, y]$ and $f[y, x]$ are equivalent
`Flat`	$f[f[x], y]$ and $f[x, y]$ are equivalent
`OneIdentity`	$f[x]$ and x are equivalent

Attributes used in matching patterns.

Pattern objects like *x*_ can represent any sequence of arguments in a function *f* with attribute `Flat`. The value of *x* in this case is *f* applied to the sequence of arguments. If *f* has the attribute `OneIdentity`, then *e* is used instead of *f*[*e*] when *x* corresponds to a sequence of just one argument.

Assignments

lhs=*rhs*	immediate assignment: *rhs* is evaluated at the time of assignment
lhs:=*rhs*	delayed assignment: *rhs* is evaluated when the value of *lhs* is requested

The two basic types of assignment in *Mathematica*.

Assignments in *Mathematica* specify transformation rules for expressions. Every assignment that you make must be associated with a particular *Mathematica* symbol.

f[*args*]=*rhs*	assignment is associated with *f* (downvalue)
t/:*f*[*args*]=*rhs*	assignment is associated with *t* (upvalue)
f[*g*[*args*]]^=*rhs*	assignment is associated with *g* (upvalue)

Assignments associated with different symbols.

In the case of an assignment like *f*[*args*] = *rhs*, *Mathematica* looks at *f*, then the head of *f*, then the head of that, and so on, until it finds a symbol with which to associate the assignment.

When you make an assignment like *lhs* ^= *rhs*, *Mathematica* will set up transformation rules associated with each distinct symbol that occurs either as an argument of *lhs*, or as the head of an argument of *lhs*.

The transformation rules associated with a particular symbol *s* are always stored in a definite order, and are tested in that order when they are used. Each time you make an assignment, the corresponding transformation rule is inserted at the end of the list of transformation rules associated with *s*, except in the following cases:

- The left-hand side of the transformation rule is identical to a transformation rule that has already been stored, and any `/;` conditions on the right-hand side are also identical. In this case, the new transformation rule is inserted in place of the old one.
- *Mathematica* determines that the new transformation rule is more specific than a rule already present, and would never be used if it were placed after this rule. In this case, the new rule is placed before the old one. Note that in many cases it is not possible to determine whether one rule is more specific than another; in such cases, the new rule is always inserted at the end.

Types of Values

`Attributes[`*f*`]`	attributes of *f*
`DefaultValues[`*f*`]`	default values for arguments of *f*
`DownValues[`*f*`]`	values for *f*`[…]`, *f*`[…][…]`, etc.
`FormatValues[`*f*`]`	print forms associated with *f*
`Messages[`*f*`]`	messages associated with *f*
`NValues[`*f*`]`	numerical values associated with *f*
`Options[`*f*`]`	defaults for options associated with *f*
`OwnValues[`*f*`]`	values for *f* itself
`UpValues[`*f*`]`	values for `…[…,` *f*`[…], …]`

Types of values associated with symbols.

Clearing and Removing Objects

expr`=.`	clear a value defined for *expr*
f`/:`*expr*`=.`	clear a value associated with *f* defined for *expr*
`Clear[`s_1`,`s_2`,…]`	clear all values for the symbols s_i, except for attributes, messages and defaults
`ClearAll[`s_1`,`s_2`,…]`	clear all values for the s_i, including attributes, messages and defaults
`Remove[`s_1`,`s_2`,…]`	clear all values, and then remove the names of the s_i

Ways to clear and remove objects.

In `Clear`, `ClearAll` and `Remove`, each argument can be either a symbol or the name of a symbol as a string. String arguments can contain the metacharacters * and @ to specify action on all symbols whose names match the pattern.

`Clear`, `ClearAll` and `Remove` do nothing to symbols with the attribute `Protected`.

Transformation Rules

lhs->*rhs*	immediate rule: *rhs* is evaluated when the rule is first given
lhs:>*rhs*	delayed rule: *rhs* is evaluated when the rule is used

The two basic types of transformation rules in *Mathematica*.

Replacements for pattern variables that appear in transformation rules are effectively done using `ReplaceAll` (the `/.` operator).

Files and Streams

File Names

name`.m`	*Mathematica* language source file
name`.nb`	*Mathematica* notebook file
name`.ma`	*Mathematica* notebook file from before Version 3
name`.mx`	*Mathematica* expression dump
name`.exe`	*MathLink* executable program
name`.tm`	*MathLink* template file
name`.ml`	*MathLink* stream file

Conventions for file names.

Most files used by *Mathematica* are completely system independent. .mx and .exe files are however system dependent. For these files, there is a convention that bundles of versions for different computer systems have names with forms such as *name* / `$SystemID` / *name*.

In general, when you refer to a file, *Mathematica* tries to resolve its name as follows:

- If the name starts with `!`, *Mathematica* treats the remainder of the name as an external command, and uses a pipe to this command.
- If the name contains metacharacters used by your operating system, then *Mathematica* passes the name directly to the operating system for interpretation.
- Unless the file is to be used for input, no further processing on the name is done.
- Unless the name given is an absolute file name under your operating system, *Mathematica* will search each of the directories specified in the list `$Path`.
- If what is found is a directory rather than a file, then *Mathematica* will look for a file *name* / `$SystemID` / *name*.

For names of the form *name*` the following further translations are done in `Get` and related functions:

- A file *name*`.mx` is used if it exists.
- If *name*`.mx` is a directory, then *name*`.mx` / `$SystemID` / *name*`.mx` is used if it exists.
- A file *name*`.m` is used if it exists.
- If *name* is a directory, then the file *name* / `init.m` is used if it exists.

In `Install`, *name*` is taken to refer to a file or directory named *name*`.exe`.

Streams

`InputStream["`*name*`",`*n*`]`	input from a file or pipe
`OutputStream["`*name*`",`*n*`]`	output to a file or pipe

Types of streams.

option name	*default value*	
`CharacterEncoding`	`Automatic`	encoding to use for special characters
`BinaryFormat`	`False`	whether to treat the file as being in binary format
`FormatType`	`InputForm`	default format for expressions
`PageWidth`	`78`	number of characters per line
`TotalWidth`	`Infinity`	maximum number of characters in a single expression

Options for output streams.

You can test options for streams using `Options`, and reset them using `SetOptions`.