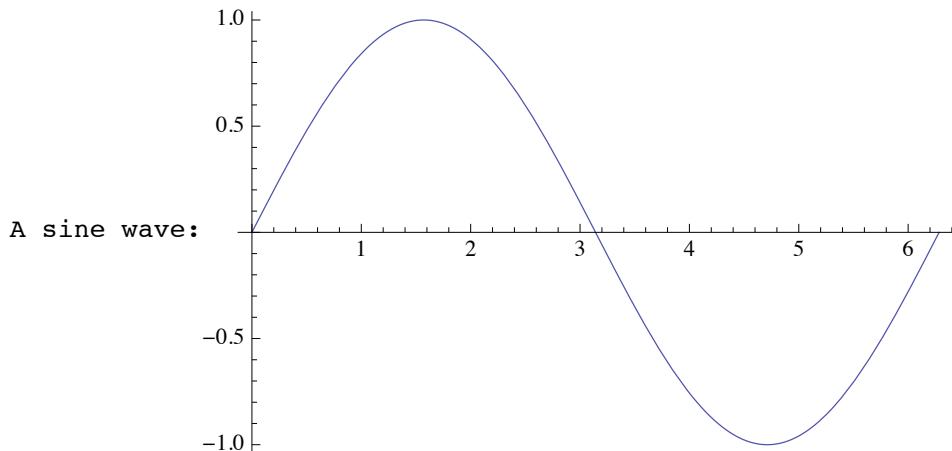


`Print` also allows mixing of text and graphics.

```
In[5]:= Print["A sine wave:", Plot[Sin[x], {x, 0, 2π}]]
```



The output generated by `Print` is usually given in the standard *Mathematica* output format. You can however explicitly specify that some other output format should be used.

This prints output in *Mathematica* input form.

```
In[6]:= Print[InputForm[a^2 + b^2]]
```

```
a^2 + b^2
```

You should realize that `Print` is only one of several mechanisms available in *Mathematica* for generating output. Another is the function `Message` described in "Messages", used for generating named messages. There are also a variety of lower-level functions described in "Streams and Low-Level Input and Output" which allow you to produce output in various formats both as part of an interactive session, and for files and external programs.

Another command which works exactly like `Print`, but only shows the printed output until the final evaluation is finished, is `PrintTemporary`.

Formatted Output

Ever since Version 3 of *Mathematica*, there has been rich support for arbitrary mathematical typesetting and layout. Underlying all that power was a so-called *box language*, which allowed notebooks themselves to be *Mathematica* expressions. This approach turned out to be very powerful, and has formed the basis of many unique features in *Mathematica*. However, despite the power of the box language, in practice it was awkward enough for users to access directly that few did.

Starting in Version 6, there is a higher-level interface to this box language which takes much of the pain out of using boxes directly, while still exposing all the same typesetting and layout power. Functions in this new layer are often referred to as *box generators*, but there is no need for you to be aware of the box language to use them effectively. In this tutorial, we will take a look at box generators that are relevant for displaying a wide variety of expressions, and we will show some ways in which they can be used to generate beautifully formatted output that goes beyond simple mathematical typesetting.

Styling Output

The *Mathematica* front end supports all the usual style mechanisms available in word processors, for example including menus for changing font characteristics. However, it used to be very difficult to access those styling mechanisms automatically in generated output. Output continued to be almost universally plain 12 pt. Courier (or Times for those people using `TraditionalForm`). To address this, the function `Style` was created. Whenever you evaluate a `Style` expression, its output will be displayed with the given style attributes active.

You can wrap `Style` around any sort of expression. Here is an example that displays prime and composite numbers using different font weights and colors via `Style`.

```
In[1]:= Table[If[PrimeQ[i], Style[i, Bold], Style[i, Gray]], {i, 1, 100}]
Out[1]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

There are hundreds of formatting options that you could apply with `Style`—see the documentation for `Style` for a more complete listing—but there are a handful that are by far the most common, listed here.

<i>Menu</i>	<i>Style[] option</i>	<i>Style[] directive</i>
Format ▶ Size ▶ 14	FontSize -> 14	14
Format ▶ Text Color ▶ Gray	FontColor -> Gray	Gray
Format ▶ Face ▶ Bold	FontWeight -> Bold	Bold
Format ▶ Face ▶ Italic	FontSlant -> Italic	Italic
Format ▶ Background Color ▶ Yellow	Background -> Yellow	
Format ▶ Font	FontFamily -> "Times"	
Format ▶ Style ▶ Subsection	"Subsection"	

Note that `style` can be arbitrarily nested, with the innermost one taking precedence if there is a conflict. Here we wrap `style` around the entire list to apply a new font to all elements of the list.

```
In[2]:= Style[%, FontFamily -> "Helvetica"]
```

```
Out[2]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

Another common thing to want is to have a portion of the output styled like text. It can look quite strange to have text appear in a font which is intended for use by code. For that purpose, we have a function `Text` which ensures that its argument will always be rendered in a text font. (Those of you familiar with *Mathematica* graphics will recognize the `Text` function as a graphics primitive, but that use does not conflict with this use outside of graphics.)

```
In[3]:= Text[%]
```

```
Out[3]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

`Style` can be used to set up a region on the screen where any option is active, not just options related to fonts. Later in this tutorial, we will see how `style` can even affect the display characteristics of other formatting constructs, like `Grid` or `Tooltip`.

Grid Layout

Using two-dimensional layout structures can be just as useful as applying style directives to those structures. In *Mathematica*, the primary function for such layout is `Grid`. `Grid` has very flexible layout features, including the ability to arbitrarily adjust things like alignment, frame elements, and spanning elements. (Other tutorials go into `Grid`'s features in greater detail, but we will cover the highlights here.)

Look again at the `Style` example which displays prime and composite numbers differently.

```
In[11]:= ptable = Table[If[PrimeQ[i], Style[i, Bold], Style[i, Gray]], {i, 1, 100}];
```

To put this into a `Grid`, we first use `Partition` to turn this 100-element list into a 10×10 array. Although you can give `Grid` a ragged array (a list whose elements are lists of different lengths), in this case we give `Grid` a regular array, and the resulting display is a nicely formatted layout.

```
Grid[Partition[ptable, 10]]
```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Notice that the columns are aligned on center, and there are no frame lines. It is an easy matter to change either of these using `Grid`'s options.

```
Grid[Partition[ptable, 10], Alignment -> Right,
      Frame -> True, Background -> LightBlue]
```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

A complete description of all `Grid`'s options and their syntax is beyond the scope of this document, but it is possible to do some remarkable things with them. See the complete `Grid` documentation for complete details.

There are a few convenience constructs related to `Grid`. One is `Column`, which takes a flat list of elements and arranges them vertically. This would be slightly awkward to do with `Grid`. Here is a simple example, viewing the options of `column` in, well, a column.

```
Column[Options[Column]]
Alignment → {Left, Baseline}
Background → None
BaselinePosition → Automatic
BaseStyle → {}
ColumnAlignments → Left
DefaultBaseStyle → Grid
DefaultElement → □
Dividers → None
Frame → None
FrameStyle → Automatic
ItemSize → Automatic
ItemStyle → None
Spacings → {0.8, 1.}
```

What about laying out a list of things horizontally? In that case, the main question you need to ask is whether you want the resulting display to line wrap like a line of math or text would, or whether you want the elements to remain on a single line. In the latter case, you would use `Grid` applied to a $1 \times n$ array.

```
In[5]:= Grid[{Range[15] !}]
Out[5]= 1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600 6227020800 87178291200 1307674368000
```

But notice in this example, that the overall grid shrinks so that it fits in the available window width. As a result, there are elements of the grid which themselves wrap onto multiple lines. This is due to the default `ItemSize` option of `Grid`. If you want to allow the elements of a grid to be as wide as they would naturally be, set `ItemSize` to `Full`.

```
In[7]:= Grid[{Range[15] !}, ItemSize → Full]
Out[7]= 1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600 6227020800 87178291200 1307674368000
```

Of course, now the whole grid is too wide to fit on one line (unless you make this window very wide), and so there are elements in the grid which you cannot see. That brings us to the other horizontal layout function: `Row`.

Given a list of elements, `Row` will allow the overall result to word wrap in the natural way, just like a line of text or math would. This type of layout will be familiar to those of you who might have used the old (and now obsolete) `SequenceForm` function.

```
Row[Range[15] !]
12624120720504040 3203628803628800399168004790016006227020800871782912001307674368000
```

As you can see, `Row` does not leave space between elements by default. But if you give a second argument, that expression is inserted between elements. Here we use a comma, but any expression can be used.

```
Row[Range[15]!, ","]
```

$$1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, \\ 479001600, 6227020800, 87178291200, 1307674368000$$

If you resize the notebook window, you will see that `Grid` with `ItemSize -> Automatic` continues to behave differently than `Row`, and each is useful in different circumstances.

Using Output as Input

This is a good time to point out that `style`, `Grid`, and all other box generators are persistent in output. If you were to take a piece of output that had some formatting created by `style` or `Grid` and reuse that as input, the literal `style` or `Grid` expressions would appear in the input expression. Those of you familiar with the old uses of `StyleBox` and even functions like `MatrixForm` will find this a change.

Consider taking the output of this `Grid` command, which has lots of embedded styles, and using it in some input expression.

```
In[17]:= Grid[Partition[Take[pTable, 16], 4], Alignment -> Right, Frame -> True, Background -> LightBlue]
```

```
Out[17]=
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

```
In[18]:= \left( \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \right)^3 + 5 // Expand
```

```
Out[18]= 125 + 75 \left( \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \right)^2 + 15 \left( \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \right) + \left( \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \right)^3
```

Notice that the grid is still a grid, it is still blue, and the elements are still bold or gray as before. Also notice that having literal `Grid` and `Style` in the expression interferes with what would have otherwise been adding a scalar to a matrix, and raising the result to a power. This

distinction is very important, since you almost always want these composite structures to resist being interpreted automatically in some way. However, if you ever do want to get rid of these wrappers and get at your data, that is easy enough to do.

```
In[19]:= % // . {Grid[a_, ___] :> a, Style[a_, ___] :> a}
Out[19]= {{216, 343, 512, 729}, {1000, 1331, 1728, 2197}, {2744, 3375, 4096, 4913}, {5832, 6859, 8000, 9261}}
```

Special Grid Entries

To allow more flexible two-dimensional layout, Grid accepts a few special symbols like `SpanFromLeft` as entries. The entry `SpanFromLeft` indicates that the grid entry immediately to the left should take up its own space and also the space of the spanning character. There are also `SpanFromAbove` and `SpanFromBoth`. See "Grids, Rows, and Columns" for detailed information.

```
Grid[{{
  {1, 2, 3, 4, 5},
  {6, 7, SpanFromLeft, SpanFromLeft, 10},
  {11, SpanFromAbove, SpanFromBoth, SpanFromBoth, 15},
  {16, 17, 18, 19, 20}}, Frame -> All]
```

1	2	3	4	5
6		7		10
11				15
16	17	18	19	20

This approach can be used to create complicated spanning setups. Typing something like the following as an input would take a long time. Luckily you can create this table interactively by using **Make Spanning** and **Split Spanning** in the **Insert ▶ Table/Matrix** submenu. If you want to see what would be involved in typing this, evaluate the cell, which will show how it should be typed as input.

```
In[18]:= // InputForm
```

1	2	3	4	5	6	7	8	9	10
11	12	13		15					
21	22			25					
31	32	33	34	35					
41									
51	52	53	54	55					
61	62	63			76	77			
71	72				86				
81	82				96				
91	92								

We have already seen how to apply things like alignment and background to a grid as a whole, or to individual columns or rows. What we have not seen though is how to override that for an individual element. Say you want your whole grid to have the same background, except for a few special elements. A convenient way to do that is to wrap each such element in `Item`, and then specify options to `Item` which override the corresponding option in `Grid`.

```
Grid[Partition[Table[If[PrimeQ[i], Item[i, Background -> LightYellow], i], {i, 1, 100}], 10], Background -> LightBlue]
```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

You could override this option with `Style` too, but the purpose of `Item` is to override it in a way that knows about the two-dimensional layout of `Grid`. Notice in the preceding output that whenever two of the yellow cells are next to each other, there is no blue space between them. That would be impossible to do with constructs other than `Item`.

The same thing goes for all `Item`'s options, not just `Background`. Consider the `Frame` option. If you want no frame elements except around certain specified elements, you might think that you have to wrap them in their own `Grid` with the `Frame -> True` setting. (We will learn a much easier way to add a frame around an arbitrary expression in the next section.)

```
Grid[Partition[Table[If[PrimeQ[i], Grid[{i}], Frame -> True], i], {i, 1, 100}], 10]
```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

But notice that adjacent framed elements do not share their boundaries. Compare that with using `Item`, below, which has enough information to not draw more frame elements than are necessary. Notice now the frames of 2 and 11 meet at a single point, and how the frames of 2 and 3 share a single-pixel line, which in turn is perfectly aligned with the left frame of 13 and 23. That is the power of `Item`.

```
Grid[Partition[Table[If[PrimeQ[i], Item[i, Frame -> True], i], {i, 1, 100}], 10]]
```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Frames and Labels

Adding a frame or a label to an expression can be done with `Grid`, but conceptually these are much simpler operations than general two-dimensional layout, and so there are correspondingly simpler ways to get them. For instance, `Framed` is a simple function for drawing a frame around an arbitrary expression. This can be useful to draw attention to parts of an expression, for instance.

```
Table[If[PrimeQ[i], Framed[i, Background -> LightYellow], i], {i, 1, 100}]
```

$$\{1, \boxed{2}, \boxed{3}, 4, \boxed{5}, 6, \boxed{7}, 8, 9, 10, \boxed{11}, 12, \boxed{13}, 14, 15, 16, \boxed{17}, 18, \boxed{19}, 20, \\ 21, 22, \boxed{23}, 24, 25, 26, 27, 28, \boxed{29}, 30, \boxed{31}, 32, 33, 34, 35, 36, \boxed{37}, 38, 39, 40, \\ \boxed{41}, 42, \boxed{43}, 44, 45, 46, \boxed{47}, 48, 49, 50, 51, 52, \boxed{53}, 54, 55, 56, 57, 58, \boxed{59}, 60, \\ \boxed{61}, 62, 63, 64, 65, 66, \boxed{67}, 68, 69, 70, \boxed{71}, 72, \boxed{73}, 74, 75, 76, 77, 78, \boxed{79}, 80, \\ 81, 82, \boxed{83}, 84, 85, 86, 87, 88, \boxed{89}, 90, 91, 92, 93, 94, 95, 96, \boxed{97}, 98, 99, 100\}$$

`Labeled` is another such function, which allows labels to be placed at arbitrary locations around a given expression. Here we add a legend to the `Grid` example from the last section. (`Spacer` is just a function that is designed to leave empty space.)

In[19]:= **Labeled**[
 Grid[Partition[pTable, 10], Alignment → Right, Frame → True],
 Text[Row[{Style["• Prime", Bold], Style["• Composite", Gray]}}, Spacer[15]]]]

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

• Prime • Composite

Panel is yet another framing construct, which uses the underlying operating system's panel frame. This is different from **Frame**, as different operating systems might use a drop shadow, rounded corners, or fancier graphic design elements for a panel frame.

In[20]:= **Panel**[%]

Out[20]=

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

• Prime • Composite

Note that **Panel** has its own concept of font family and size as well, so the contents of **Grid** change font family and size, and the **Text** changes font size. (**Text** has its own opinion about font family though, and so it remains in *Mathematica*'s text font.) We will talk about this in some detail below in the section on the **BaseStyle** option.

Finally, we should point out that **Panel** itself has an optional second argument to specify one or more labels, which are automatically positioned outside the panel, and an optional third argument to give details of that position. See the documentation for **Panel** for more detail.

In[37]:= **Panel**[pTable, "Primes and Composites"]

Primes and Composites

Out[37]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}

```
In[38]:= Panel[pTable, {"Primes and Composites"}, {{Bottom, Right}}]
```

Out[38]=

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

Primes and Composites

Other Annotations

The annotations mentioned so far have a very definite visual component. There are a number of annotations which are effectively invisible, until the user needs them. `Tooltip` for example does not change the display of its first argument, and only when you move the mouse pointer over that display is the second argument shown, as a tooltip.

```
Table[Tooltip[i, Divisors[i]], {i, 1, 100}]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

`Mouseover` is another such function, but instead of displaying the result in a tooltip, it uses the same area of the screen that had been used for the display before you moved the mouse pointer over it. If the two displays are different sizes, then the effect can be jarring, so it is a good idea to use displays which are closer to the same size, or use the `Mouseover ImageSize` option to leave space for the larger of the two displays, regardless of which is being displayed.

```
Table[Mouseover[i, Framed[Divisors[i], Background -> LightYellow]], {i, 1, 100}]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

Also similar to `Tooltip` are `StatusArea` and `PopupWindow`. `StatusArea` displays the extra information in the notebook's status area, typically in the lower-left corner, while `PopupWindow` will display extra information in a new window when clicked.

```
Table[StatusArea[i, Divisors[i]], {i, 1, 100}]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

```
Table[PopupWindow[i, Divisors[i]], {i, 1, 100}]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

Finally, you can specify an arbitrary location for an annotation by using the pair `Annotation` and `MouseAnnotation`.

```
Table[Annotation[i, Divisors[i], "Mouse"], {i, 1, 100}]
Dynamic[MouseAnnotation[]]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

```
Null
```

When using annotations that are triggered merely by moving the mouse pointer over a region of the screen, it is important to keep the user in mind. Moving the mouse is not something that should trigger a long evaluation or a lot of visual clutter. But used sparingly, annotations can be quite helpful to users.

Finally, note that all these annotations work perfectly well in graphics too. So you can provide tooltips or mouseovers to aid users in understanding a complicated graphic you have created. In fact, even visualization functions like `ListPlot` or `DensityPlot` support `Tooltip`. See the documentation for details.

```
In[2]:= Graphics[{LightBlue, EdgeForm[Gray], Tooltip[CountryData[#, "SchematicPolygon"],
Panel[CountryData[#, "Flag"], #]] & /@ CountryData[]}, ImageSize -> Full]
```



Default Styles

As we saw in the section "Frames and Labels", constructs like `Panel` actually work much like `Style`, in that they set up an environment in which a set of default styles is applied to their contents. This can be overridden by explicit `style` commands, but it can also be overridden for the `Panel` itself, through the `BaseStyle` option. `BaseStyle` can be set to a style or a list of style directives, just like you would use in `style`. And those directives then become the ambient default within the scope of that `Panel`.

As we have already seen, `Panel` by default uses the dialog font family and size. But that can be overridden by using this `BaseStyle` option.

```
Panel[Range[10]]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

In[7]:= Panel[Range[10], BaseStyle -> {"StandardForm"}]

Out[7]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Actually, almost all of these box generators have a `BaseStyle` option. For instance, here is a grid in which the default font color is blue. Notice that the elements that were gray stay gray, since the inner `Style` wrapper trumps the outer `Grid` `BaseStyle`. (This is one of the principal characteristics of *option inheritance*, which is beyond the scope of this document to discuss.)

```
Grid[Partition[ptable, 10], BaseStyle -> {FontColor -> Blue}]
```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Default Options

Say you have an expression with multiple occurrences of the same box generator, like a `Framed` or a `Panel`, and you want to change all of them to have the same set of options. It might be cumbersome to go through and add the same set of options to every occurrence of that function. Thankfully, there is an easier way.

`DefaultOptions` is an option to `Style` which, when set to a list of elements of the form `head -> {opt -> val, ...}`. sets up an environment with the given options as the ambient default for the given box-generating head. Those options will be active throughout the `Style` wrapper, but only in any instances of the associated box generator.

So if you had an expression that contained some `Framed` items, and you wanted them all to be drawn with the same background and frame style.

```
Table[If[PrimeQ[i], Framed[i], i], {i, 1, 100}]
```

Actually, that input is too short to see the advantage of this syntax. Say you had this same list, but specified manually.

```
biglist = {1, Framed[2], Framed[3], 4, Framed[5], 6, Framed[7], 8, 9, 10,
Framed[11], 12, Framed[13], 14, 15, 16, Framed[17], 18, Framed[19], 20,
21, 22, Framed[23], 24, 25, 26, 27, 28, Framed[29], 30, Framed[31], 32, 33,
34, 35, 36, Framed[37], 38, 39, 40, Framed[41], 42, Framed[43], 44, 45, 46,
Framed[47], 48, 49, 50, 51, 52, Framed[53], 54, 55, 56, 57, 58, Framed[59],
60, Framed[61], 62, 63, 64, 65, 66, Framed[67], 68, 69, 70, Framed[71], 72,
Framed[73], 74, 75, 76, 77, 78, Framed[79], 80, 81, 82, Framed[83], 84, 85,
86, 87, 88, Framed[89], 90, 91, 92, 93, 94, 95, 96, Framed[97], 98, 99, 100}

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

Now inserting `Background` and `FrameStyle` options into every `Framed` wrapper is prohibitively time consuming, although you certainly could do it (or you could write a program to do it for you). But using `DefaultOptions`, you can effectively set up an environment in which all `Framed` wrappers will use your settings for `Background` and `FrameStyle`, thus.

```
Style[biglist,
DefaultOptions → {Framed → {Background → LightYellow, FrameStyle → Blue}}]

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

This approach makes it easy to create structures that follow uniform style guidelines without having to specify those styles in more than one place, which makes for considerably cleaner code, smaller file sizes, and easier maintenance.

Mathematical Typesetting

No discussion of formatted output would be complete without at least a nod toward the formatting constructs that are unique to mathematical syntaxes.

```
{Subscript[a, b], Superscript[a, b], Underscript[a, b],
 Overscript[a, b], Subsuperscript[a, b, c], Underoverscript[a, b, c]}

{ab, ab, a,  $\overset{b}{a}$ , abc,  $\underset{b}{a}^c$ }
```

We will not discuss these at length, but we will point out that these constructs do not have any built-in mathematical meaning in the kernel. For example, `Superscript[a, b]` will not be interpreted as `Power[a, b]`, even though their displays are identical. So you can use these as structural elements in your formatted output without having to worry about their meaning affecting your display.

In[67]:= `Table[Row[{i, Row[Superscript @@@ FactorInteger[i], "x"]}], {"i", 100}]`

Out[67]=

```
{1 == 11, 2 == 21, 3 == 31, 4 == 22, 5 == 51, 6 == 21 × 31, 7 == 71, 8 == 23, 9 == 32, 10 == 21 × 51, 11 == 111,
 12 == 22 × 31, 13 == 131, 14 == 21 × 71, 15 == 31 × 51, 16 == 24, 17 == 171, 18 == 21 × 32, 19 == 191,
 20 == 22 × 51, 21 == 31 × 71, 22 == 21 × 111, 23 == 231, 24 == 23 × 31, 25 == 52, 26 == 21 × 131, 27 == 33,
 28 == 22 × 71, 29 == 291, 30 == 21 × 31 × 51, 31 == 311, 32 == 25, 33 == 31 × 111, 34 == 21 × 171, 35 == 51 × 71,
 36 == 22 × 32, 37 == 371, 38 == 21 × 191, 39 == 31 × 131, 40 == 23 × 51, 41 == 411, 42 == 21 × 31 × 71,
 43 == 431, 44 == 22 × 111, 45 == 32 × 51, 46 == 21 × 231, 47 == 471, 48 == 24 × 31, 49 == 72, 50 == 21 × 52,
 51 == 31 × 171, 52 == 22 × 131, 53 == 531, 54 == 21 × 33, 55 == 51 × 111, 56 == 23 × 71, 57 == 31 × 191,
 58 == 21 × 291, 59 == 591, 60 == 22 × 31 × 51, 61 == 611, 62 == 21 × 311, 63 == 32 × 71, 64 == 26, 65 == 51 × 131,
 66 == 21 × 31 × 111, 67 == 671, 68 == 22 × 171, 69 == 31 × 231, 70 == 21 × 51 × 71, 71 == 711, 72 == 23 × 32,
 73 == 731, 74 == 21 × 371, 75 == 31 × 52, 76 == 22 × 191, 77 == 71 × 111, 78 == 21 × 31 × 131, 79 == 791,
 80 == 24 × 51, 81 == 34, 82 == 21 × 411, 83 == 831, 84 == 22 × 31 × 71, 85 == 51 × 171, 86 == 21 × 431,
 87 == 31 × 291, 88 == 23 × 111, 89 == 891, 90 == 21 × 32 × 51, 91 == 71 × 131, 92 == 22 × 231, 93 == 31 × 311,
 94 == 21 × 471, 95 == 51 × 191, 96 == 25 × 31, 97 == 971, 98 == 21 × 72, 99 == 32 × 111, 100 == 22 × 52}
```

Using the Box Language

One final note. Those of you who are already familiar with the box language might occasionally find that these box generators get in the way of your constructing low level boxes yourselves, and inserting their display into a piece of output. That can be true for any layered technology where one abstraction layer attempts to hide the layers on which it sits. However, there is a simple loophole through which you can take boxes which you happen to know are valid, and display them directly in output: `RawBoxes`.

```
{a, b, RawBoxes[SubscriptBox["c", "d"]], e}
{a, b, cd, e}
```

As with all loopholes, `RawBoxes` gives you added flexibility, but it also allows you to shoot yourself in the foot. Use with care. And if you are not yet familiar with the box language, perhaps you should not use it at all.

Requesting Input

Mathematica usually works by taking whatever input you give, and then processing it. Sometimes, however, you may want to have a program you write explicitly request more input. You can do this using `Input` and `InputString`.

<code>Input []</code>	read an expression as input
<code>InputString []</code>	read a string as input
<code>Input ["prompt"]</code>	issue a prompt, then read an expression
<code>InputString ["prompt"]</code>	issue a prompt, then read a string

Interactive input.

Exactly how `Input` and `InputString` work depends on the computer system and *Mathematica* interface you are using. With a text-based interface, they typically just wait for standard input, terminated with a newline. With a notebook interface, however, they typically get the front end to put up a “dialog box”, in which the user can enter input.

In general, `Input` is intended for reading complete *Mathematica* expressions. `InputString`, on the other hand, is for reading arbitrary strings.

Messages

Mathematica has a general mechanism for handling messages generated during computations. Many built-in *Mathematica* functions use this mechanism to produce error and warning messages. You can also use the mechanism for messages associated with functions you write.

The basic idea is that every message has a definite name, of the form `symbol::tag`. You can use this name to refer to the message. (The object `symbol::tag` has head `MessageName`.)

<code>Quiet [expr]</code>	evaluate <i>expr</i> without printing any messages
<code>Quiet [expr, {s1::tag, s2::tag, ...}]</code>	evaluate <i>expr</i> without printing the specified messages
<code>Off [s::tag]</code>	switch off a message, so it is not printed
<code>On [s::tag]</code>	switch on a message

Controlling the printing of messages.

As discussed in "Warnings and Messages", you can use `Quiet` to control the printing of particular messages during an evaluation. Most messages associated with built-in functions are switched on by default. If you want to suppress a message permanently, you can use `Off`.

This prints a warning message. Also, the front end highlights the extra argument in red.

In[1]:= Log[a, b, c]

Log::argt: Log called with 3 arguments; 1 or 2 arguments are expected. >>

Out[1]= Log[a, b, c]

This suppresses the warning message.

In[2]:= Quiet[Log[a, b, c]]

Out[2]= Log[a, b, c]

The message reappears with the next evaluation.

In[3]:= Log[a, b, c]

Log::argt: Log called with 3 arguments; 1 or 2 arguments are expected. >>

Out[3]= Log[a, b, c]

You can use `on` and `off` to make global changes to the printing of particular messages. You can use `off` to switch off a message if you never want to see it.

You can switch off the message like this.

```
In[4]:= Off[Log::argt]
```

Now no warning message is produced.

```
In[5]:= Log[a, b, c]
```

```
Out[5]= Log[a, b, c]
```

Although most messages associated with built-in functions are switched on by default, there are some which are switched off by default, and which you will see only if you explicitly switch them on. An example is the message `General::newsym`, discussed in "Intercepting the Creation of New Symbols", which tells you every time a new symbol is created.

`s::tag`

give the text of a message

`s::tag=string`

set the text of a message

`Messages[s]`

show all messages associated with *s*

Manipulating messages.

The text of a message with the name `s::tag` is stored simply as the value of `s::tag`, associated with the symbol *s*. You can therefore see the text of a message simply by asking for `s::tag`. You can set the text by assigning a value to `s::tag`.

If you give `LinearSolve` a singular matrix, it prints a warning message.

```
In[6]:= LinearSolve[{{1, 1}, {2, 2}}, {3, 5}]
```

LinearSolve::nosol: Linear equation encountered that has no solution. >>

```
Out[6]= LinearSolve[{{1, 1}, {2, 2}}, {3, 5}]
```

Here is the text of the message.

```
In[7]:= LinearSolve::nosol
```

```
Out[7]= Linear equation encountered that has no solution.
```

This redefines the message.

```
In[8]:= LinearSolve::nosol = "Matrix encountered is not invertible."
```

```
Out[8]= Matrix encountered is not invertible.
```

Now the new form will be used.

```
In[9]:= LinearSolve[{{1, 1}, {2, 2}}, {3, 5}]
LinearSolve::nosol: Matrix encountered is not invertible. >>
Out[9]= LinearSolve[{{1, 1}, {2, 2}}, {3, 5}]
```

Messages are always stored as strings suitable for use with `StringForm`. When the message is printed, the appropriate expressions are “spliced” into it. The expressions are wrapped with `HoldForm` to prevent evaluation. In addition, any function that is assigned as the value of the global variable `$MessagePrePrint` is applied to the resulting expressions before they are given to `StringForm`. The default for `$MessagePrePrint` uses `Short` for text formatting and a combination of `Short` and `Shallow` for typesetting.

Most messages are associated directly with the functions that generate them. There are, however, some “general” messages, which can be produced by a variety of functions.

If you give the wrong number of arguments to a function *F*, *Mathematica* will warn you by printing a message such as *F::argx*. If *Mathematica* cannot find a message named *F::argx*, it will use the text of the “general” message *General::argx* instead. You can use `Off[F::argx]` to switch off the argument count message specifically for the function *F*. You can also use `Off[General::argx]` to switch off all messages that use the text of the general message.

Mathematica prints a message if you give the wrong number of arguments to a built-in function.

```
In[10]:= Sqrt[a, b]
Sqrt::argx: Sqrt called with 2 arguments; 1 argument is expected. >>
Out[10]= Sqrt[a, b]
```

This argument count message is a general one, used by many different functions.

```
In[11]:= General::argx
Out[11]= `1` called with `2` arguments; 1 argument is expected.
```

If something goes very wrong with a calculation you are doing, it is common to find that the same warning message is generated over and over again. This is usually more confusing than useful. As a result, *Mathematica* keeps track of all messages that are produced during a particular calculation, and stops printing a particular message if it comes up more than three times. Whenever this happens, *Mathematica* prints the message `General::stop` to let you know. If

if you really want to see all the messages that *Mathematica* tries to print, you can do this by switching off `General::stop`.

<code>\$MessageList</code>	a list of the messages produced during a particular computation
<code>MessageList [n]</code>	a list of the messages produced during the processing of the n^{th} input line in a <i>Mathematica</i> session

Finding out what messages were produced during a computation.

In every computation you do, *Mathematica* maintains a list `$MessageList` of all the messages that are produced. In a standard *Mathematica* session, this list is cleared after each line of output is generated. However, during a computation, you can access the list. In addition, when the n^{th} output line in a session is generated, the value of `$MessageList` is assigned to `MessageList [n]`.

This returns `$MessageList`, which gives a list of the messages produced.

```
In[12]:= Sqrt[a, b, c]; Exp[a, b]; $MessageList
          Sqrt::argx: Sqrt called with 3 arguments; 1 argument is expected. >>
          Exp::argx: Exp called with 2 arguments; 1 argument is expected. >>
Out[12]= {Sqrt::argx, Exp::argx}
```

The message names are wrapped in `HoldForm` to stop them from evaluating.

```
In[13]:= InputForm[%]
Out[13]//InputForm= {HoldForm[Sqrt::argx], HoldForm[Exp::argx]}
```

In writing programs, it is often important to be able to check automatically whether any messages were generated during a particular calculation. If messages were generated, say as a consequence of producing indeterminate numerical results, then the result of the calculation may be meaningless.

<code>Check [expr, failexpr]</code>	if no messages are generated during the evaluation of <i>expr</i> , then return <i>expr</i> , otherwise return <i>failexpr</i>
<code>Check [expr, failexpr, s₁::t₁, s₂::t₂, ...]</code>	check only for the messages $s_i::t_i$

Checking for warning messages.

Evaluating 1^0 produces no messages, so the result of the evaluation is returned.

```
In[14]:= Check[1^0, err]
Out[14]= 1
```

Evaluating 0^0 produces a message, so the second argument of `Check` is returned.

```
In[15]:= Check[0^0, err]
Power::indet: Indeterminate expression 0^0 encountered. >>
Out[15]= err
```

`Check` [*expr*, *failexpr*] tests for all messages that are actually printed out. It does not test for messages whose output has been suppressed using `Off`.

In some cases you may want to test only for a specific set of messages, say ones associated with numerical overflow. You can do this by explicitly telling `Check` the names of the messages you want to look for.

The message generated by `Sin[1, 2]` is ignored by `Check`, since it is not the one specified.

```
In[16]:= Check[Sin[1, 2], err, General::ind]
Sin::argx: Sin called with 2 arguments; 1 argument is expected. >>
Out[16]= Sin[1, 2]
```

`Message` [*s::tag*]

print a message

`Message` [*s::tag*, *expr_i*, ...]

print a message, with the *expr_i* spliced into its string form

Generating messages.

By using the function `Message`, you can mimic all aspects of the way in which built-in *Mathematica* functions generate messages. You can for example switch on and off messages using `on` and `off`, and `Message` will automatically look for `General::tag` if it does not find the specific message *s::tag*.

This defines the text of a message associated with `f`.

```
In[17]:= f::overflow = "Factorial argument `1` too large."
Out[17]= Factorial argument `1` too large.
```

Here is the function `f`.

```
In[18]:= f[x_] := If[x > 10, (Message[f::overflow, x]; Infinity), x!]
```

When the argument of `f` is greater than 10, the message is generated.

In[19]:= f[20]

`f::overflow: Factorial argument 20 too large.`

Out[19]= ∞

This switches off the message.

In[20]:= off[f::overflow]

Now the message is no longer generated.

In[21]:= f[20]

Out[21]= ∞

When you call `Message`, it first tries to find a message with the explicit name you have specified. If this fails, it tries to find a message with the appropriate tag associated with the symbol `General`. If this too fails, then *Mathematica* takes any function you have defined as the value of the global variable `$NewMessage`, and applies this function to the symbol and tag of the message you have requested.

By setting up the value of `$NewMessage` appropriately, you can, for example, get *Mathematica* to read in the text of a message from a file when that message is first needed.

International Messages

The standard set of messages for built-in *Mathematica* functions are written in American English. In some versions of *Mathematica*, messages are also available in other languages. In addition, if you set up messages yourself, you can give ones in other languages.

Languages in *Mathematica* are conventionally specified by strings. The languages are given in English, in order to avoid the possibility of needing special characters. Thus, for example, the French language is specified in *Mathematica* as "French".

<code>\$Language = "lang"</code>	set the language to use
<code>\$Language = { "lang₁", "lang₂", ... }</code>	set a sequence of languages to try

Setting the language to use for messages.

This tells *Mathematica* to use French-language versions of messages.

```
In[1]:= $Language = "French"
```

```
Out[1]= French
```

If your version of *Mathematica* has French-language messages, the message generated here will be in French.

```
In[2]:= Sqrt[a, b, c]
```

```
Sqrt::argx : Sqrt est appell  
elax\parskip\z@$$EAcute]e avec 3 arguments; il faut y avoir 1.
```

```
Out[2]= Sqrt[a, b, c]
```

<i>symbol</i> :: <i>tag</i>	the default form of a message
<i>symbol</i> :: <i>tag</i> :: <i>Language</i>	a message in a particular language

Messages in different languages.

When built-in *Mathematica* functions generate messages, they look first for messages of the form *s*::*t*::*Language*, in the language specified by *\$Language*. If they fail to find any such messages, then they use instead the form *s*::*t* without an explicit language specification.

The procedure used by built-in functions will also be followed by functions you define if you call *Message* with message names of the form *s*::*t*. If you give explicit languages in message names, however, only those languages will be used.

Documentation Constructs

When you write programs in *Mathematica*, there are various ways to document your code. As always, by far the best thing is to write clear code, and to name the objects you define as explicitly as possible.

Sometimes, however, you may want to add some "commentary text" to your code, to make it easier to understand. You can add such text at any point in your code simply by enclosing it in matching (* and *). Notice that in *Mathematica*, "comments" enclosed in (* and *) can be nested in any way.

You can use comments anywhere in the *Mathematica* code you write.

```
In[1]:= If[a > b, (*then*) p, (*else*) q]
Out[1]= If[a > b, p, q]
```

(*text*)

a comment that can be inserted anywhere in *Mathematica* code

Comments in *Mathematica*.

There is a convention in *Mathematica* that all functions intended for later use should be given a definite "usage message", which documents their basic usage. This message is defined as the value of $f::\text{usage}$, and is retrieved when you type $?f$.

```
f::usage = "text"
?f
??f
```

define the usage message for a function
get information about a function
get more information about a function

Usage messages for functions.

Here is the definition of a function f .

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

Here is a "usage message" for f .

```
In[3]:= f::usage = "f[x] gives the square of x."
Out[3]= f[x] gives the square of x.
```

This gives the usage message for f .

```
In[4]:= ?f
```

f[x] gives the square of x.

$\text{?? } f$ gives all the information *Mathematica* has about f , including the actual definition.

```
In[5]:= ??f
```

f[x] gives the square of x.

$f[x_] := x^2$

When you define a function f , you can usually display its value using `?f`. However, if you give a usage message for f , then `?f` just gives the usage message. Only when you type `??f` do you get all the details about f , including its actual definition.

If you ask for information using `?` about just one function, *Mathematica* will print out the complete usage messages for the function. If you ask for information on several functions at the same time, however, *Mathematica* will give the name of each function, if possible with a link to its usage information.

This gives all the symbols in *Mathematica* that start with "Plot".

*In[6]:= ? Plot**

▼ System`

Plot	PlotJoined	PlotRange	PlotStyle
Plot3D	PlotLabel	PlotRangeClip`. ping	
Plot3Matrix	PlotMarkers	PlotRangePad`. ding	
PlotDivision	PlotPoints	PlotRegion	

If you use *Mathematica* with a text-based interface, then messages and comments are the primary mechanisms for documenting your definitions. However, if you use *Mathematica* with a notebook interface, then you will be able to give much more extensive documentation in text cells in the notebook.

Manipulating Notebooks

Cells as *Mathematica* Expressions

Like other objects in *Mathematica*, the cells in a notebook, and in fact the whole notebook itself, are all ultimately represented as *Mathematica* expressions. With the standard notebook front end, you can use the command **Show Expression** to see the text of the *Mathematica* expression that corresponds to any particular cell.

Show Expression menu item	toggle between displayed form and underlying <i>Mathematica</i> expression
<code>Ctrl+*</code> or <code>Ctrl+8</code> (between existing cells)	put up a dialog box to allow input of a cell in <i>Mathematica</i> expression form

Handling `Cell` expressions in the notebook front end.

Here is a cell displayed in its usual way in the front end.

```
This is a text cell.
```

]

Here is the underlying *Mathematica* expression that corresponds to the cell.

```
Cell["This is a text cell.", "Text"]
```

]

<code>Cell[contents, "style"]</code>	a cell with a specific style
<code>Cell[contents, "style", options]</code>	a cell with additional options specified
<code>Cell[contents, "style1", "style2", ..., options]</code>	a cell with several styles

Mathematica expressions corresponding to cells in notebooks.

Within a given notebook, there is always a collection of *styles* that can be used to determine the appearance and behavior of cells. Typically the styles are named so as to reflect what role cells which have them will play in the notebook.

"Title"	the title of the notebook
"Section"	a section heading
"Subsection"	a subsection heading
"Text"	ordinary text
"Input"	<i>Mathematica</i> input
"Output"	<i>Mathematica</i> output

Some typical cell styles defined in notebooks.

Here are several cells in different styles.

This is a Section Style

This is a text style.

This is Input style.

Here are the expressions that correspond to these cells.

```
Cell["This is a Section Style", "Section"]
Cell["This is a text style.", "Text"]
Cell[BoxData[
  RowBox[{"This", " ", "is", " ", "Input", " ", "style", "."}]], "Input"]
```

A particular style such as "Section" or "Text" defines various settings for the options associated with a cell. You can override these settings by explicitly setting options within a specific cell.

Here is the expression for a cell in which options are set to use a gray background and to put a frame around the cell.

```
Cell["This is some text.", "Text",
  CellFrame->True,
  Background->GrayLevel[0.8]]
```

This is how the cell looks in a notebook.

This is some text.

<i>option</i>	<i>default value</i>	
CellFrame	False	whether to draw a frame around the cell
Background	Automatic	what color to draw the background for the cell
Editable	True	whether to allow the contents of the cell to be edited
TextAlignment	Left	how to align text in the cell
FontSize	12	the point size of the font for text
CellTags	{}	tags to be associated with the cell

A few of the large number of possible options for cells.

The standard notebook front end for *Mathematica* provides several ways to change the options of a cell. In simple cases, such as changing the size or color of text, there will often be a specific menu item for the purpose. But in general you can use the *Option Inspector* that is built into the front end. This is typically accessed using the **Option Inspector** menu item in the **Format** menu.

- Change settings for specific options with menus.
- Look at and modify all options with the Option Inspector.
- Edit the textual form of the expression corresponding to the cell.

- Change the settings for all cells with a particular style.

Ways to manipulate cells in the front end.

Sometimes you will want just to change the options associated with a specific cell. But often you may want to change the options associated with all cells in your notebook that have a particular style. You can do this by using the **Edit Stylesheet** command in the front end to create a custom stylesheet associated with your notebook. Then use the controls in the stylesheet to create a cell corresponding to the style you want to change and modify the options for that cell.

<code>CellPrint[Cell[...]]</code>	insert a cell into your currently selected notebook
<code>CellPrint[{Cell[...], Cell[...], ...}]</code>	insert a sequence of cells into your currently selected notebook

Inserting cells into a notebook.

This inserts a section cell into the current notebook.

```
In[1]:= CellPrint[Cell["The heading", "Section"]]
```

A screenshot of a Mathematica notebook window. Inside the window, there is a single section cell. The cell has a thin black border and contains the text 'The heading' in a bold, black, sans-serif font. To the right of the text, there are three blue square brackets arranged vertically, indicating nested structures or cell groups.

This inserts a text cell with a frame around it.

```
In[2]:= CellPrint[Cell["Some text", "Text", CellFrame -> True]]
```

A screenshot of a Mathematica notebook window. Inside the window, there is a text cell with a gray rectangular frame around it. The frame has a thin black border. Inside the frame, the text 'Some text' is displayed in a black, sans-serif font. To the right of the frame, there are three blue square brackets arranged vertically.

CellPrint allows you to take a raw Cell expression and insert it into your current notebook. The cell created by CellPrint is grouped with the input and will be overwritten if the input is reevaluated.

Notebooks as *Mathematica* Expressions

`Notebook [{cell1,cell2,...}]`

a notebook containing a sequence of cells

`Notebook [cells, options]`

a notebook with options specified

Expressions corresponding to notebooks.

Here is a simple *Mathematica* notebook.

A screenshot of a Mathematica notebook window. Inside the window, there is a section cell with the title 'Section heading'. Below it are two text cells. The first text cell contains the text 'Some text.' and the second contains 'More text.'. Both text cells have a gray rectangular frame with a thin black border. To the right of each frame, there are three blue square brackets arranged vertically.

Here is the expression that corresponds to this notebook.

```
Notebook[{
  Cell["Section heading", "Section"],
  Cell["Some text.", "Text"],
  Cell["More text.", "Text"]}]
```

Just like individual cells, notebooks in *Mathematica* can also have options. You can look at and modify these options using the Option Inspector in the standard notebook front end.

<i>option</i>	<i>default value</i>	
WindowSize	{nx,ny}	the size in pixels of the window used to display the notebook
WindowFloating	False	whether the window should float on top of others
WindowToolbars	{ }	what toolbars to include at the top of the window
ShowPageBreaks	False	whether to show where page breaks would occur if the notebook were printed
CellGrouping	Automatic	how to group cells in the notebook
Evaluator	"Local"	what kernel should be used to do evaluations in the notebook
Saveable	True	whether a notebook can be saved

A few of the large number of possible options for notebooks.

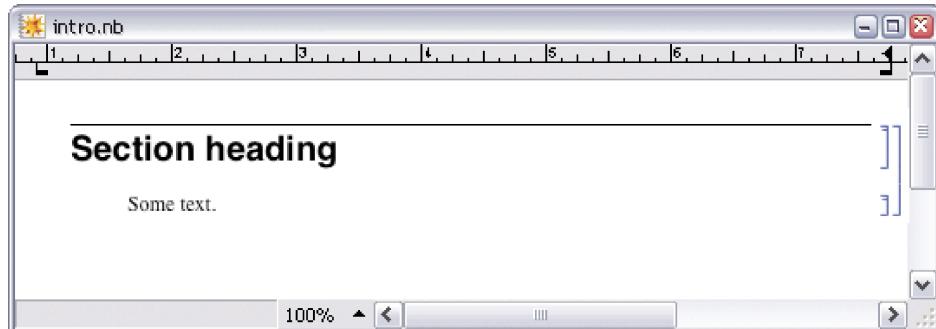
A notebook with the option setting `Saveable -> False` can always be saved using the **Save As** menu item, but does not respond to **Save** and does not prompt for saving when it is closed.

In addition to notebook options, you can also set any cell option at the notebook level. Doing this tells *Mathematica* to use that option setting as the default for all the cells in the notebook. You can override the default by explicitly setting the options within a particular cell or by using a named style which explicitly overrides the option.

Here is the expression corresponding to a notebook with a ruler displayed in the toolbar at the top of the window.

```
Notebook[{
  Cell["Section heading", "Section"],
  Cell["Some text.", "Text"],
  WindowToolbars->{"RulerBar"}]
```

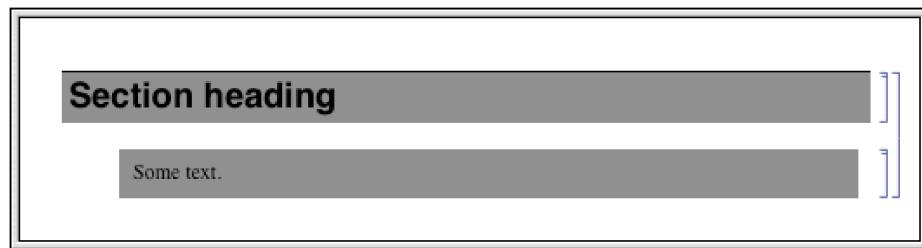
This is what the notebook looks like in the front end.



This sets the default background color for all cells in the notebook.

```
Notebook[  
  Cell["Section heading", "Section"],  
  Cell["Some text.", "Text"],  
  Background->GrayLevel[.7]]
```

Now each cell has a gray background.



If you go outside of *Mathematica* and look at the raw text of the file that corresponds to a *Mathematica* notebook, you will find that what is in the file is just the textual form of the expression that represents the notebook. One way to create a *Mathematica* notebook is therefore to construct an appropriate expression and put it in a file.

In notebook files that are written out by *Mathematica*, some additional information is typically included to make it faster for *Mathematica* to read the file in again. The information is enclosed in *Mathematica* comments indicated by `(*....*)` so that it does not affect the actual expression stored in the file.

<code>NotebookOpen ["file.nb"]</code>	open a notebook file in the front end
<code>NotebookPut [expr]</code>	create a notebook corresponding to <i>expr</i> in the front end
<code>NotebookGet [obj]</code>	get the expression corresponding to an open notebook in the front end

Setting up notebooks in the front end from the kernel.

This writes a notebook expression out to the file `sample.nb`.

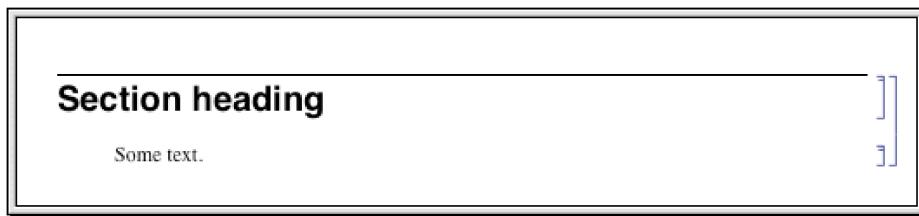
```
In[1]:= Notebook[{Cell["Section heading", "Section"], Cell["Some text.", "Text"]}] >>
"sample.nb"
```

This reads the notebook expression back from the file.

```
In[2]:= << sample.nb
Out[2]= Notebook[{Cell[Section heading, Section], Cell[Some text., Text]}]
```

This opens `sample.nb` as a notebook in the front end.

```
In[3]:= NotebookOpen ["sample.nb"];
```



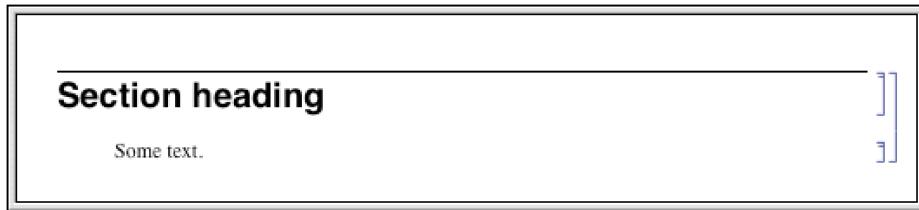
Once you have set up a notebook in the front end using `NotebookOpen`, you can then manipulate the notebook interactively just as you would any other notebook. But in order to use `NotebookOpen`, you have to explicitly have a notebook expression in a file. With `NotebookPut`, however, you can take a notebook expression that you have created in the kernel, and immediately display it as a notebook in the front end.

Here is a notebook expression in the kernel.

```
In[4]:= Notebook[{Cell["Section heading", "Section"], Cell["Some text.", "Text"]}
Out[4]= Notebook[{Cell[Section heading, Section], Cell[Some text., Text]}]
```

This uses the expression to set up a notebook in the front end.

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



You can use `NotebookGet` to get the notebook corresponding to a particular `NotebookObject` back into the kernel.

```
In[6]:= NotebookGet[%]
```

```
Out[6]= Notebook[{Cell[CellGroupData[{Cell[TextData["Section heading"], Section], Cell[TextData["Some text."], Text]}], Open}}]
```

Manipulating Notebooks from the Kernel

If you want to do simple operations on *Mathematica* notebooks, then you will usually find it convenient just to use the interactive capabilities of the standard *Mathematica* front end. But if you want to do more complicated and systematic operations, then you will often find it better to use the kernel.

<code>Notebooks[]</code>	a list of all your open notebooks
<code>Notebooks["name"]</code>	a list of all open notebooks with the specified name
<code>InputNotebook[]</code>	the notebook into which typed input will go
<code>EvaluationNotebook[]</code>	the notebook in which this function is being evaluated
<code>ButtonNotebook[]</code>	the notebook containing the button (if any) which initiated this evaluation

Functions that give the notebook objects corresponding to particular notebooks.

Within the *Mathematica* kernel, notebooks that you have open in the front end are referred to by *notebook objects* of the form `NotebookObject[fe, id]`. The first argument of `NotebookObject` specifies the `FrontEndObject` for the front end in which the notebook resides, while the second argument gives a unique serial number for the notebook.

Here is a notebook named `Example.nb`.

- 
- First Heading
 - Second Heading

This finds the corresponding notebook object in the front end.

```
In[1]:= Notebooks["Example.nb"]
Out[1]= {NotebookObject[<<Example.nb>>]}
```

This gets the expression corresponding to the notebook into the kernel.

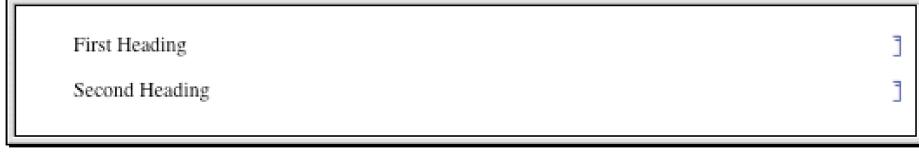
```
In[2]:= NotebookGet[First[%]]
Out[2]= Notebook[{Cell[First Heading, Section],
Cell[Second Heading, Section]}]
```

This replaces every occurrence of the string "Section" by "Text".

```
In[3]:= % /. "Section" -> "Text"
Out[3]= Notebook[{Cell[First Heading, Text],
Cell[Second Heading, Text]}]
```

This creates a new modified notebook in the front end.

```
In[4]:= NotebookPut[%]
```

- 
- First Heading
 - Second Heading

```
Out[4]= {NotebookObject[<<Untitled-1.nb>>]}
```

NotebookGet [*obj*]

get the notebook expression corresponding to the notebook object *obj*

NotebookPut [*expr*,*obj*]

replaces the notebook represented by the notebook object *obj* with one corresponding to *expr*

NotebookPut [*expr*]

creates a notebook corresponding to *expr* and makes it the currently selected notebook in the front end

Exchanging whole notebook expressions between the kernel and front end.

If you want to do extensive manipulations on a particular notebook you will usually find it convenient to use `NotebookGet` to get the whole notebook into the kernel as a single expression. But if instead you want to do a sequence of small operations on a notebook, then it is often better to leave the notebook in the front end, and then to send specific commands from the kernel to the front end to tell it what operations to do.

Mathematica is set up so that anything you can do interactively to a notebook in the front end you can also do by sending appropriate commands to the front end from the kernel.

<code>Options [obj]</code>	give a list of all options set for the notebook corresponding to notebook object <i>obj</i>
<code>Options [obj, option]</code>	give the option setting
<code>AbsoluteOptions [obj, option]</code>	give the option setting with absolute option values even when the actual setting is <code>Automatic</code>
<code>CurrentValue [obj, option]</code>	give and set the value of <i>option</i>
<code>SetOptions [obj, option->value]</code>	set the value of an option

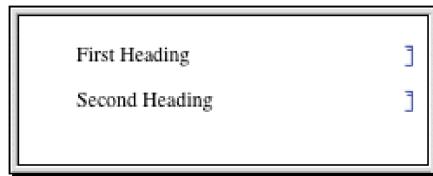
Finding and setting options for notebooks.

This gives the setting of the `WindowSize` option for your currently selected notebook.

```
In[5]:= Options[InputNotebook[], WindowSize]
Out[5]= {WindowSize -> {250., 100.}}
```

This changes the size of the currently selected notebook on the screen.

```
In[6]:= SetOptions[InputNotebook[], WindowSize -> {250, 100}]
```



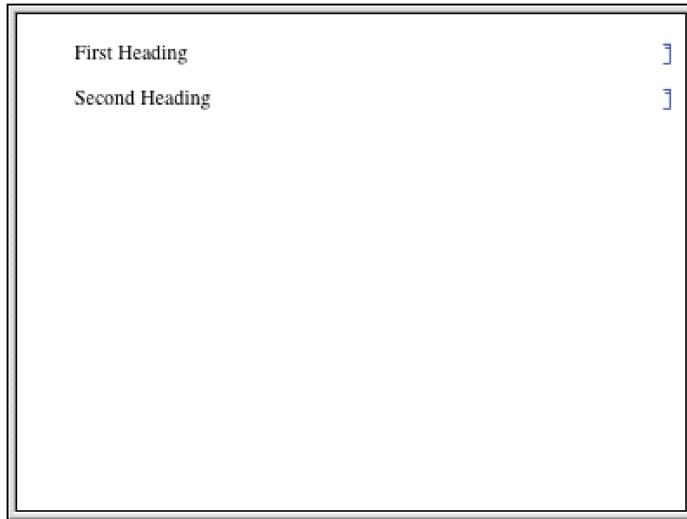
```
Out[6]= {WindowSize -> {250., 100.}}
```

Alternatively, use `CurrentValue` to directly get the value of the `WindowSize` option.

```
In[7]:= CurrentValue[InputNotebook[], WindowSize]
Out[7]= {WindowSize -> {250., 100.}}
```

This changes the option using `CurrentValue` with a simple assignment.

```
In[8]:= CurrentValue[InputNotebook[],WindowSize] = {400, 300}
```



Within any open notebook, the front end always maintains a *current selection*. The selection can consist for example of a region of text within a cell or of a complete cell. Usually the selection is indicated on the screen by some form of highlighting. The selection can also be between two characters of text, or between two cells, in which case it is usually indicated on the screen by a vertical or horizontal insertion bar.

You can modify the current selection in an open notebook by issuing commands from the kernel.

<code>SelectionMove [obj, Next, unit]</code>	move the current selection to make it be the next unit of the specified type
<code>SelectionMove [obj, Previous, unit]</code>	move to the previous unit
<code>SelectionMove [obj, After, unit]</code>	move to just after the end of the present unit of the specified type
<code>SelectionMove [obj, Before, unit]</code>	move to just before the beginning of the present unit
<code>SelectionMove [obj, All, unit]</code>	extend the current selection to cover the whole unit of the specified type

Moving the current selection in a notebook.

Character	individual character
Word	word or other token
Expression	complete subexpression
TextLine	line of text
TextParagraph	paragraph of text
GraphicsContents	the contents of the graphic
Graphics	graphic
CellContents	the contents of the cell
Cell	complete cell
CellGroup	cell group
EvaluationCell	cell associated with the current evaluation
ButtonCell	cell associated with any button that initiated the evaluation
GeneratedCell	cell generated by the current evaluation
Notebook	complete notebook

Units used in specifying selections.

Here is a simple notebook.

- Here is a first cell.
 - Here is a second cell.

This sets nb to be the notebook object corresponding to the current input notebook.

In[9]:= `nb = InputNotebook[];`

This moves the current selection within the notebook to be the next word.

In[10]:= `SelectionMove[nb, Next, Word]`

- Here is a first cell.
 - Here is a second cell.

This extends the selection to the complete first cell.

In[11]:= SelectionMove[nb, All, Cell]

This puts the selection at the end of the whole notebook.

In[12]:= SelectionMove[nb, After, Notebook]

<code>NotebookFind[obj, data]</code>	move the current selection to the next occurrence of the specified data in a notebook
<code>NotebookFind[obj, data, Previous]</code>	move to the previous occurrence
<code>NotebookFind[obj, data, All]</code>	make the current selection cover all occurrences
<code>NotebookFind[obj, data, dir, elems]</code>	search in the specified elements of each cell, going in direction <i>dir</i>
<code>NotebookFind[obj, "text", IgnoreCase -> True]</code>	do not distinguish uppercase and lowercase letters in text

Searching the contents of a notebook.

This moves the current selection to the position of the previous occurrence of the word `cell`.

In[13]:= NotebookFind[nb, "cell", Previous]

The letter α does not appear in the current notebook, so \$Failed is returned, and the selection is not moved.

In[14]:= **NotebookFind[nb, "α", Next]**

- Here is a first cell.
 - Here is a second cell.

Out[14]= \$Failed

CellContents	contents of each cell
CellStyle	the name of the style for each cell
CellLabel	the label for each cell
CellTags	tags associated with each cell
{elem₁, elem₂, ...}	several kinds of elements

Possible elements of cells to be searched by **NotebookFind**.

In setting up large notebooks, it is often convenient to insert tags which are not usually displayed, but which mark particular cells in such a way that they can be found using **NotebookFind**. You can set up tags for cells either interactively in the front end, or by explicitly setting the **CellTags** option for a cell.

NotebookLocate["tag"]	locate and select cells with the specified tag in the current notebook
NotebookLocate[{"file", "tag"}]	open another notebook if necessary

Globally locating cells in notebooks.

NotebookLocate is typically the underlying function that *Mathematica* calls when you follow a hyperlink in a notebook. The **Insert ▶ Hyperlink** menu item sets up the appropriate **NotebookLocate** as part of the script for a particular hyperlink button.

<code>NotebookWrite [obj, data]</code>	write <i>data</i> into a notebook at the current selection
<code>NotebookApply [obj, data]</code>	write <i>data</i> into a notebook, inserting the current selection in place of the first ■ that appears in <i>data</i>
<code>NotebookDelete [obj]</code>	delete whatever is currently selected in a notebook
<code>NotebookRead [obj]</code>	get the expression that corresponds to the current selection in a notebook

Writing and reading in notebooks.

`NotebookWrite [obj, data]` is similar to a **Paste** operation in the front end: it replaces the current selection in your notebook by *data*. If the current selection is a cell `NotebookWrite [obj, data]` will replace the cell with *data*. If the current selection lies between two cells, however, then `NotebookWrite [obj, data]` will create an appropriate new cell or cells.

Here is a notebook with a word of text selected.

- 
- Here is a first cell.
 - Here is a second cell.

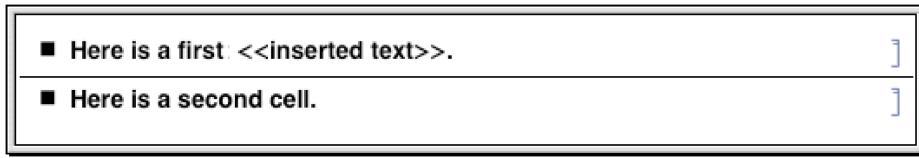
This replaces the selected word by new text.

In[15]:= NotebookWrite[nb, "<<inserted text>>"]

- 
- Here is a first <<inserted text>>.
 - Here is a second cell.

This moves the current selection to just after the first cell in the notebook.

In[16]:= SelectionMove[nb, After, Cell]

- 
- Here is a first <<inserted text>>.
 - Here is a second cell.

This now inserts a text cell after the first cell in the notebook.

In[17]:= NotebookWrite[nb, Cell["This cell contains text.", "Text"]]

```

■ Here is a first <<inserted text>>.
This cell contains text.

■ Here is a second cell.

```

This makes the current selection be the next cell in the notebook.

In[18]:= SelectionMove[nb, Next, Cell]

```

■ Here is a first <<inserted text>>.
This cell contains text.

■ Here is a second cell.

```

This reads the current selection, returning it as an expression in the kernel.

In[19]:= NotebookRead[nb]

```

■ Here is a first <<inserted text>>.
This cell contains text.

■ Here is a second cell.

```

Out[19]= Cell[Here is a second one., Section]

`NotebookWrite[obj, data]` just discards the current selection and replaces it with *data*. But particularly if you are setting up palettes, it is often convenient first to modify *data* by inserting the current selection somewhere inside it. You can do this using *selection placeholders* and `NotebookApply`. The first time the character "■", entered as `\ [SelectionPlaceholder]` or `Esc spl Esc`, appears anywhere in *data*, `NotebookApply` will replace this character by the current selection.

Here is a simple notebook with the current selection being the contents of a cell.

In[20]:= nb = InputNotebook[];

Expand $(1 + x)^4$

]

This replaces the current selection by a string that contains a copy of its previous form.

In[21]:= NotebookApply[nb, "x + 1"]

x + 1 / Expand $(1 + x)^4$

]

<code>SelectionEvaluate[obj]</code>	evaluate the current selection in place
<code>SelectionCreateCell[obj]</code>	create a new cell containing just the current selection
<code>SelectionEvaluateCreateCell[obj]</code>	evaluate the current selection and create a new cell for the result
<code>SelectionAnimate[obj]</code>	animate graphics in the current selection
<code>SelectionAnimate[obj, t]</code>	animate graphics for t seconds

Operations on the current selection.

This makes the current selection be the whole contents of the cell.

In[22]:= SelectionMove[nb, All, CellContents]

x + 1 / Expand $(1 + x)^4$

]

This evaluates the current selection in place.

In[23]:= SelectionEvaluate[nb]

x + $\frac{1}{1 + 4 x + 6 x^2 + 4 x^3 + x^4}$

]

`SelectionEvaluate` allows you to take material from a notebook and send it through the kernel for evaluation. On its own, however, `SelectionEvaluate` always overwrites the material you took. But by using functions like `SelectionCreateCell` you can maintain a record of the sequence of forms that are generated—just like in a standard *Mathematica* session.

This makes the current selection be the whole cell.

In[24]:= SelectionMove[nb, All, Cell]

This creates a new cell, and copies the current selection into it.

In[25]:= SelectionCreateCell[nb]

This wraps Factor around the contents of the current cell.

In[26]:= NotebookApply[nb, "Factor[■]"]

This evaluates the contents of the current cell, and creates a new cell to give the result.

In[27]:= SelectionEvaluateCreateCell[nb]

Functions like `NotebookWrite` and `SelectionEvaluate` by default leave the current selection just after whatever material they insert into your notebook. You can then always move the

selection by explicitly using `SelectionMove`. But functions like `NotebookWrite` and `SelectionEvaluate` can also take an additional argument which specifies where the current selection should be left after they do their work.

<code>NotebookWrite</code> [<i>obj</i> , <i>data</i> , <i>sel</i>]	write <i>data</i> into a notebook, leaving the current selection as specified by <i>sel</i>
<code>NotebookApply</code> [<i>obj</i> , <i>data</i> , <i>sel</i>]	write <i>data</i> replacing ■ by the previous current selection, then leaving the current selection as specified by <i>sel</i>
<code>SelectionEvaluate</code> [<i>obj</i> , <i>sel</i>]	evaluate the current selection, making the new current selection be as specified by <i>sel</i>
<code>SelectionCreateCell</code> [<i>obj</i> , <i>sel</i>]	create a new cell containing just the current selection, and make the new current selection be as specified by <i>sel</i>
<code>SelectionEvaluateCreateCell</code> [<i>obj</i> , <i>sel</i>]	evaluate the current selection, make a new cell for the result, and make the new current selection be as specified by <i>sel</i>

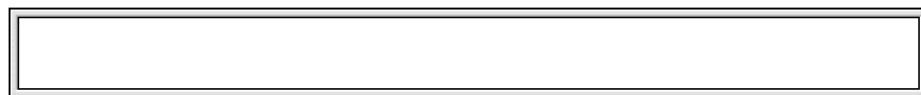
Performing operations and specifying what the new current selection should be.

<code>After</code>	immediately after whatever material is inserted (default)
<code>Before</code>	immediately before whatever material is inserted
<code>All</code>	the inserted material itself
<code>Placeholder</code>	the first ■ in the inserted material
<code>None</code>	leave the current selection unchanged

Specifications for the new current selection.

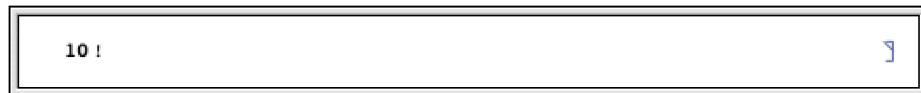
Here is a blank notebook.

In[28]:= nb = InputNotebook[];



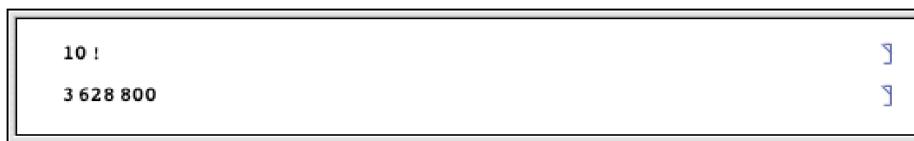
This writes 10 ! into the notebook, making the current selection be what was written.

In[29]:= NotebookWrite[nb, "10!", All]



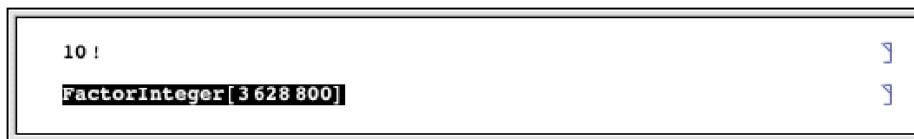
This evaluates the current selection, creating a new cell for the result, and making the current selection be the whole of the result.

*In[30]:= **SelectionEvaluateCreateCell[nb, All]***



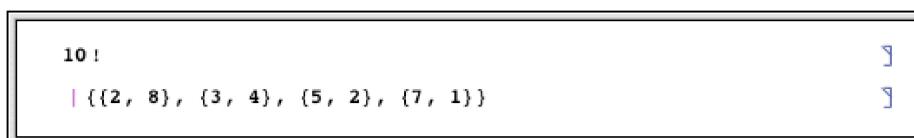
This wraps `FactorInteger` around the current selection.

*In[31]:= **NotebookApply[nb, "FactorInteger[■]", All]***



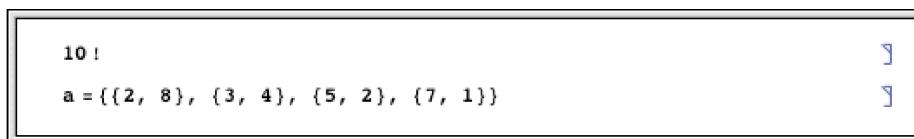
This evaluates the current selection, leaving the selection just before the result.

*In[32]:= **SelectionEvaluate[nb, Before]***



This now inserts additional text at the position of the current selection.

*In[33]:= **NotebookWrite[nb, "a = "]***



`Options [obj,option]` find the value of an option for a complete notebook

`Options [NotebookSelection[obj],option]`

find the value for the current selection

`SetOptions [obj,option->value]` set the value of an option for a complete notebook

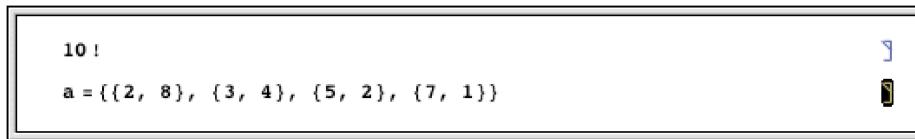
`SetOptions [NotebookSelection[obj],option->value]`

set the value for the current selection

Finding and setting options for whole notebooks and for the current selection.

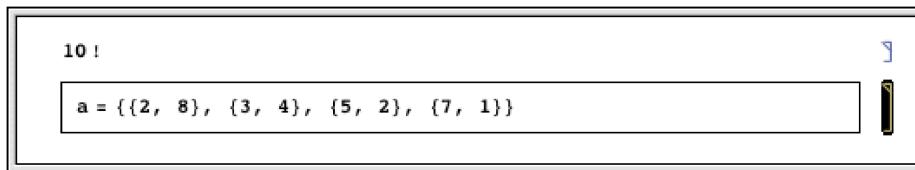
Make the current selection be a complete cell.

In[34]:= SelectionMove[nb, All, Cell]



Put a frame around the cell that is the current selection.

In[35]:= SetOptions[NotebookSelection[nb], CellFrame -> True]



<code>CreateWindow[]</code>	create a new notebook
<code>CreateWindow[options]</code>	create a notebook with specified options
<code>NotebookOpen["name"]</code>	open an existing notebook
<code>NotebookOpen["name",options]</code>	open a notebook with specified notebook options
<code>SetSelectedNotebook[obj]</code>	make the specified notebook the selected one
<code>NotebookPrint[obj]</code>	send a notebook to your printer
<code>NotebookPrint[obj,"file"]</code>	send a PostScript version of a notebook to a file
<code>NotebookPrint[obj,"!command"]</code>	send a PostScript version of a notebook to an external command
<code>NotebookSave[obj]</code>	save the current version of a notebook in a file
<code>NotebookSave[obj,"file"]</code>	save the notebook in a file with the specified name
<code>NotebookClose[obj]</code>	close a notebook

Operations on whole notebooks.

If you call `CreateWindow[]` a new empty notebook will appear on your screen.

By executing commands like `SetSelectedNotebook` and `NotebookOpen`, you tell the *Mathematica* front end to change the windows you see. Sometimes you may want to manipulate a notebook without ever having it displayed on the screen. You can do this by using the option setting `Visible -> False` in `NotebookOpen` or `CreateWindow`.

Manipulating the Front End from the Kernel

<code>\$FrontEnd</code>	the front end currently in use
<code>Options[\$FrontEnd, option]</code>	the setting for a global option in the front end
<code>AbsoluteOptions[\$FrontEnd, option]</code>	the absolute setting for an option
<code>SetOptions[\$FrontEnd, option -> value]</code>	reset an option in the front end
<code>CurrentValue[\$FrontEnd, option]</code>	return option value, and also allow setting of option when used as the left-hand side of an assignment

Manipulating global options in the front end.

Just like cells and notebooks, the complete *Mathematica* front end has various options, which you can look at and manipulate from the kernel.

This gives the object corresponding to the front end currently in use.

In[1]:= `$FrontEnd`

Out[1]= `-FrontEndObject-`

This gives the current directory used by the front end for notebook files.

In[2]:= `Options[$FrontEnd, NotebookBrowseDirectory]`

Out[2]= `{NotebookBrowseDirectory \rightarrow C:\Documents and Settings\All Users\Documents}`

<i>option</i>	<i>default value</i>	
<code>NotebookBrowseDirectory</code>	(system dependent)	the default directory for opening and saving notebook files
<code>NotebookPath</code>	(system dependent)	the path to search when trying to open notebooks
<code>Language</code>	"English"	default language for text
<code>MessageOptions</code>	(list of settings)	how to handle various help and warning messages

A few global options for the *Mathematica* front end.

By using `NotebookWrite` you can effectively input to the front end any ordinary text that you can enter on the keyboard. `FrontEndTokenExecute` allows you to send from the kernel any command that the front end can execute. These commands include both menu items and control sequences.

<code>FrontEndTokenExecute["name"]</code>	execute a named command in the front end
---	--

Executing a named command in the front end.

<code>"Indent"</code>	indent all selected lines by one tab
<code>"NotebookStatisticsDialog"</code>	display statistics about the current notebook
<code>"OpenCloseGroup"</code>	toggle a cell group between open and closed
<code>"CellSplit"</code>	split a cell in two at the current insertion point
<code>"DuplicatePreviousInput"</code>	create a new cell which is a duplicate of the nearest input cell above
<code>"FindDialog"</code>	bring up the Find dialog
<code>"ColorSelectorDialog"</code>	bring up the Color Selector dialog
<code>"GraphicsAlign"</code>	align selected graphics
<code>"CompleteSelection"</code>	complete the command name that is the current selection

A few named commands that can be given to the front end. These commands usually correspond to menu items.

Front End Tokens

Front end tokens let you perform kernel commands that would normally be done using the menus. Front end tokens are particularly convenient for writing programs to manipulate notebooks.

`FrontEndToken` is a kernel command that identifies its argument as a front end token. `FrontEndExecute` is a kernel command that sends its argument to the front end for execution. For example, the following command creates a new notebook.

```
In[10]:= FrontEndExecute[FrontEndToken["New"]]
```

`FrontEndExecute` can take a list as its argument, allowing you to execute multiple tokens in a single evaluation. When you evaluate the following command, the front end creates a new notebook and then pastes the contents of the clipboard into that notebook.

```
In[9]:= FrontEndExecute[{FrontEndToken["New"], FrontEndToken["Paste"]}]
```

Simple and Compound Front End Tokens

Front end tokens are divided into two classes: simple tokens and compound tokens that take parameters.

Simple Tokens

For simple tokens, `FrontEndToken` can have one or two arguments.

If `FrontEndToken` has one argument, the token operates on the input notebook. The following examples use the front end token "Save". The result is the same as using **File ▶ Save**.

```
In[12]:= FrontEndExecute[FrontEndToken["Save"]]
```

With two arguments, the arguments of `FrontEndToken` must be a `NotebookObject` and a front end token. For example, to save the notebook containing the current evaluation, the first argument of `FrontEndToken` is the notebook object `EvaluationNotebook`, and the second argument is the front end token "Save".

```
In[3]:= FrontEndExecute[FrontEndToken[FrontEnd`EvaluationNotebook[], "Save"]]
```

You can execute a simple, one-argument front end token with the command `FrontEndTokenExecute[token]`. This is equivalent to `FrontEndExecute[FrontEndToken[token]]`.

For example, the following command will save the input notebook.

```
In[5]:= FrontEndTokenExecute["Save"]
```

Compound Tokens

Compound tokens have a token parameter that controls some aspect of their behavior. For a compound token, the three arguments of `FrontEndToken` must be a `NotebookObject`, the front end token, and the selected token parameter.

For example, this saves the selected notebook as plain text.

```
In[6]:= FrontEndExecute[
{FrontEndToken[FrontEnd`InputNotebook[], "SaveRenameSpecial", "Text"]}]
```

Executing Notebook Commands Directly in the Front End

When you execute a command like `NotebookWrite[obj, data]` the actual operation of inserting data into your notebook is performed in the front end. Normally, however, the kernel is needed in order to evaluate the original command, and to construct the appropriate request to send to the front end. But it turns out that the front end is set up to execute a limited collection of commands directly, without ever involving the kernel.

<code>NotebookWrite[obj, data]</code>	version of <code>NotebookWrite</code> to be executed in the kernel
---------------------------------------	--

<code>FrontEnd`NotebookWrite[obj, data]</code>	
--	--

	version of <code>NotebookWrite</code> to be executed directly in the front end
--	--

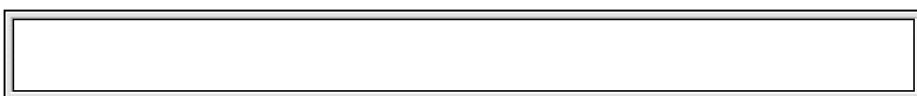
Distinguishing kernel and front end versions of commands.

The basic way that *Mathematica* distinguishes between commands to be executed in the kernel and to be executed directly in the front end is by using contexts. The kernel commands are in the usual `System`` context, but the front end commands are in the `FrontEnd`` context.

<code>FrontEndExecute[expr]</code>	send <i>expr</i> to be executed in the front end
------------------------------------	--

Sending an expression to be executed in the front end.

Here is a blank notebook.



This uses kernel commands to write data into the notebook.

In[1]:= `NotebookWrite[SelectedNotebook[], "x + y + z"]`



In the kernel, these commands do absolutely nothing.

```
In[2]:= FrontEnd`NotebookWrite[FrontEnd`SelectedNotebook[], "a + b + c + d"]
```



x + y + z

]

If they are sent to the front end, however, they cause data to be written into the notebook.

```
In[3]:= FrontEndExecute[%]
```



x + y + z

]

a + b + c + d

]

If you write sophisticated programs for manipulating notebooks, then you will have no choice but to execute these programs primarily in the kernel. But for the kinds of operations typically performed by simple buttons, you may find that it is possible to execute all the commands you need directly in the front end—without the kernel even needing to be running.

The Structure of Cells

<code>Cell[contents, "style"]</code>	a cell in a particular style
<code>Cell[contents, "style₁", "style₂", ...]</code>	a cell with multiple styles
<code>Cell[contents, "style", options]</code>	a cell with additional options set

Expressions corresponding to cells.

Here is a notebook containing a text cell and a *Mathematica* input cell.



Here is some ordinary text.

x^a / y

]

]

Here are the expressions corresponding to these cells.

```
Cell["Here is some ordinary text.", "Text"]
Cell[BoxData[
RowBox[{SuperscriptBox["x", "\[Alpha]"], "/", "y"}]], "Input"]
```

Here is a notebook containing a text cell with *Mathematica* input inside.

Text with the formula xyz^α inside.

]

This is the expression corresponding to the cell. The *Mathematica* input is in a cell embedded inside the text.

```
Cell[TextData[{
  "Text with the formula ",
  Cell[BoxData[
    SuperscriptBox["xyz", "\[Alpha]"]]],
  " inside."
}], "Text",
GeneratedCell->False]
```

"text"	plain text
TextData[{text ₁ ,text ₂ ,...}]	text potentially in different styles, or containing cells
BoxData[boxes]	formatted <i>Mathematica</i> expressions
GraphicsData["type",data]	graphics or sounds
OutputFormData["itext", "otext"]	text as generated by <code>InputForm</code> and <code>OutputForm</code>
RawData["data"]	unformatted expressions as obtained using Show Expression
CellGroupData[{cell ₁ ,cell ₂ ,...},Open]	an open group of cells
CellGroupData[{cell ₁ ,cell ₂ ,...},Closed]	a closed group of cells
StyleData["style"]	a style definition cell

Expressions representing possible forms of cell contents.

Styles and the Inheritance of Option Settings

Global	the complete front end and all open notebooks
Notebook	the current notebook
Style	the style of the current cell
Cell	the specific current cell
Selection	a selection within a cell

The hierarchy of levels at which options can be set.

Here is a notebook containing three cells.

First Section

Some text in the first section.

Some more text in the first section.

This is what happens when the setting `CellFrame -> True` is made specifically for the third cell.

First Section

Some text in the first section.

Some more text in the first section.

This is what happens when the setting `CellFrame -> True` is made globally for the whole notebook.

First Section

Some text in the first section.

Some more text in the first section.

This is what happens when the setting is made for the "Section" style.

First Section

Some text in the first section.

Some more text in the first section.

In the standard notebook front end, you can check and set options at any level by using the **Option Inspector** menu item. If you do not set an option at a particular level, then its value will always be inherited from the level above. Thus, for example, if a particular cell does not set the `CellFrame` option, then the value used will be inherited from its setting for the style of the cell or for the whole notebook that contains the cell.

As a result, if you set `CellFrame -> True` at the level of a whole notebook, then all the cells in the notebook will have frames drawn around them—unless the style of a particular cell, or the cell itself, explicitly overrides this setting.

- Choose the basic default styles for a notebook
- Choose the styles for screen and printing style environments
- Edit specific styles for the notebook

Ways to set up styles in a notebook.

Depending on what you intend to use your *Mathematica* notebook for, you may want to choose different basic default styles for the notebook. In the standard notebook front end, you can do this by selecting a different stylesheet in the **Stylesheet** menu or by using the **Edit Stylesheet** menu item.

<code>"StandardReport"</code>	styles for everyday work and for reports
<code>"NaturalColor"</code>	styles for colorful presentation of everyday work
<code>"Outline"</code>	styles for outlining ideas
<code>"Notepad"</code>	styles for working with plain text documents

Some typical choices of basic default styles.

With each choice of basic default styles, the styles that are provided will change. Thus, for example, the `Notepad` stylesheet provides a limited number of styles since it is designed to work with plain text documents.

Here is a notebook that uses `NaturalColor` default styles.

Riccati Equations

A **Riccati equation** is a first-order equation of the form

$$y'(x) = f(x) + g(x)y(x) + h(x)y(x)^2.$$

This equation was used by Count Riccati of Venice (1676-1754) to help in solving second-order ordinary differential equations.

Solving Riccati equations is considerably more difficult than solving linear ODEs.

Here is a simple Riccati equation for which the solution is available in closed form:

```
In[1]:= DSolve[y'[x] + (2/x^2) - 3 y[x]^2 == 0, y[x], x] // Simplify
Out[1]= \left\{y[x] \rightarrow -\frac{3 x^5 - 2 C[1]}{3 x^6 + 3 x C[1]}\right\}
```

option	default value	
<code>ScreenStyleEnvironment</code>	"Working"	the style environment to use for display on the screen
<code>PrintingStyleEnvironment</code>	"Printout"	the style environment to use for printed output

Options for specifying style environments.

Within a particular set of basic default styles, *Mathematica* allows for two different style environments: one for display on the screen, and another for output to a printer. The existence of separate screen and printing style environments allows you to set up styles which are separately optimized both for low-resolution display on a screen, and high-resolution printing.

"Working"	onscreen working environment
"Presentation"	onscreen environment for presentations
"Condensed"	onscreen environment for maximum display density
"Slideshow"	onscreen environment for displaying slides
"Printout"	paper printout environment

Some typical settings for style environments.

The way that *Mathematica* actually sets up the definitions for styles is by using *style definition cells*. These cells can either be given in separate *stylesheet notebooks*, or can be included in the options of a specific notebook. In either case, you can access style definitions by using the **Edit Stylesheet** menu item in the standard notebook front end.

Options for Cells

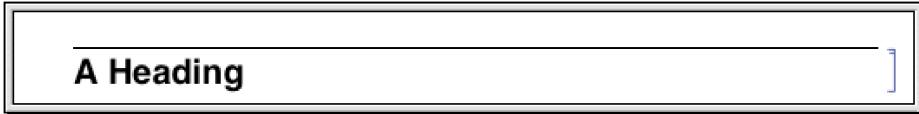
Mathematica provides a large number of options for cells. All of these options can be accessed through the **Option Inspector** menu item in the front end. They can be set either directly at the level of individual cells or at a higher level, to be inherited by individual cells.

<i>option</i>	<i>typical default value</i>	
CellDingbat	None	a dingbat to use to emphasize the cell
CellFrame	False	whether to draw a frame around the cell
Background	None	the background color for the cell
ShowCellBracket	True	whether to display the cell bracket
Magnification	1.	the magnification at which to display the cell
CellOpen	True	whether to display the contents of the cell

Some basic cell display options.

This creates a cell in "Section" style with default settings for all options.

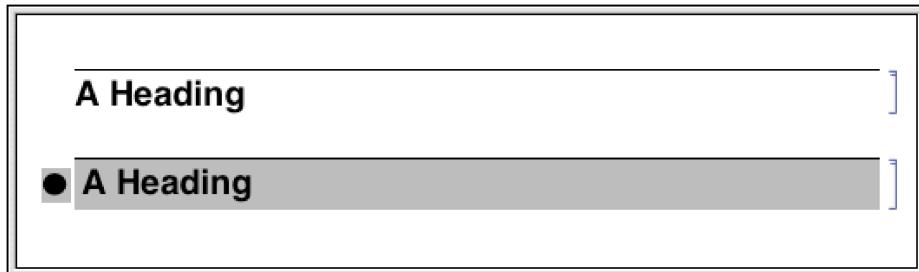
In[1]:= CellPrint[Cell["A Heading", "Section"]]



A Heading

This creates a cell with dingbat and background options modified.

```
In[2]:= CellPrint[
  Cell["A Heading", "Section", CellDingbat -> "●", Background -> GrayLevel[.7]]]
```



<i>option</i>	<i>typical default value</i>	
CellMargins	$\{\{7, 0\}, \{4, 4\}\}$	outer margins in printer's points to leave around the contents of the cell
CellFrameMargins	8	margins to leave inside the cell frame
CellElementSpacings	list of rules	details of the layout of cell elements
CellBaseline	Baseline	how to align the baseline of an inline cell with text around it

Options for cell positioning.

The option `CellMargins` allows you to specify both horizontal and vertical margins to put around a cell. You can set the horizontal margins interactively by using the margin stops in the ruler displayed when you choose the **Show Ruler** menu item in the front end.

Whenever an option can refer to all four edges of a cell, *Mathematica* follows the convention that the setting for the option takes the form $\{\{left, right\}, \{bottom, top\}\}$. By giving nonzero values for the *top* and *bottom* elements, `CellMargins` can specify gaps to leave above and below a particular cell. The values are always taken to be in printer's points.

This leaves 50 points of space on the left of the cell, and 20 points above and below.

```
In[3]:= CellPrint[Cell["First text", "Text", CellMargins -> {{50, 0}, {20, 20}}]]
```

Almost every aspect of *Mathematica* notebooks can be controlled by some option or another. More detailed aspects are typically handled by "aggregate options" such as `CellElementSpacings`. The settings for these options are lists of *Mathematica* rules, which effectively give values for a sequence of suboptions. The names of these suboptions are usually strings rather than symbols.

This shows the settings for all the suboptions associated with `CellElementSpacings`.

```
In[4]:= Options[SelectedNotebook[], CellElementSpacings]
```

```
Out[4]= {CellElementSpacings -> {CellMinHeight -> 12., ClosedCellHeight -> 19.,
ClosedGroupTopMargin -> 4., GroupIconTopMargin -> 3., GroupIconBottomMargin -> 12.}}
```

Mathematica allows you to embed cells inside pieces of text. The option `CellBaseline` determines how such "inline cells" will be aligned vertically with respect to the text around them. In direct analogy with the option `BaselinePosition` for a `Grid`, the option `CellBaseline` specifies what aspect of the cell should be considered its baseline.

Here is a cell containing an inline formula. The baseline of the formula is aligned with the baseline of the text around it.

Here is a cell in which the bottom of the formula is aligned with the baseline of the text around it.

This alignment is specified using the `CellBaseline -> Bottom` setting.

```
Cell[TextData[{  
  "the ",  
  Cell[BoxData[  
    FractionBox["x", "y"]],  
  CellBaseline->Bottom],  
  " fraction"  
}], "Subsection"]
```

<i>option</i>	<i>typical default value</i>	
<code>CellLabel</code>	<code>""</code>	a label for a cell
<code>ShowCellLabel</code>	<code>True</code>	whether to show the label for a cell
<code>CellLabelAutoDelete</code>	<code>True</code>	whether to delete the label if the cell is modified
<code>CellTags</code>	<code>{}</code>	tags for a cell
<code>ShowCellTags</code>	<code>False</code>	whether to show tags for a cell
<code>ConversionRules</code>	<code>{}</code>	rules for external conversions

Options for ancillary data associated with cells.

In addition to the actual contents of a cell, it is often useful to associate various kinds of ancillary data with cells.

In a standard *Mathematica* session, cells containing successive lines of kernel input and output are given labels of the form `In[n] :=` and `Out[n] =`. The option `ShowCellLabel` determines whether such labels should be displayed. `CellLabelAutoDelete` determines whether the label on a cell should be removed if the contents of the cell are modified. Doing this ensures that `In[n] :=` and `Out[n] =` labels are only associated with unmodified pieces of kernel input and output.

Cell tags are typically used to associate keywords or other attributes with cells, that can be searched for using functions like `NotebookFind`. Destinations for hyperlinks in *Mathematica* notebooks are usually implemented using cell tags.

The option `ConversionRules` allows you to give a list containing entries such as `"TeX" -> data` which specify how the contents of a cell should be converted to external formats. This is particularly relevant if you want to keep a copy of the original form of a cell that has been converted in *Mathematica* notebook format from some external format.

<i>option</i>	<i>typical default value</i>	
Deletable	True	whether to allow a cell to be deleted interactively with the front end
Copyable	True	whether to allow a cell to be copied
Selectable	True	whether to allow the contents of a cell to be selected
Editable	True	whether to allow the contents of a cell to be edited
Deployed	False	whether the user interface in the cell is active

Options for controlling interactive operations on cells.

The options `Deletable`, `Copyable`, `Selectable` and `Editable` allow you to control what interactive operations should be allowed on cells. By setting these options to `False` at the notebook level, you can protect all the cells in a notebook.

`Deployed` allows you to treat the contents of a cell as if they were a user interface. In a user interface, labels are typically not selectable and controls such as buttons can be used, but not modified. `Deployed` can also be set on specific elements inside a cell so that, for example, the output of `Manipulate` is always deployed even if the cell it is in has the `Deployed` option set to `False`.

<i>option</i>	<i>typical default value</i>	
Evaluator	"Local"	the name of the kernel to use for evaluations
Evaluatable	False	whether to allow the contents of a cell to be evaluated
CellAutoOverwrite	False	whether to overwrite previous output when new output is generated
GeneratedCell	False	whether this cell was generated from the kernel
InitializationCell	False	whether this cell should automatically be evaluated when the notebook is opened

Options for evaluation.

Mathematica makes it possible to specify a different evaluator for each cell in a notebook. But most often, the `Evaluator` option is set only at the notebook or global level, typically using the **Kernel Configuration Options** menu item in the front end.

The option `CellAutoOverwrite` is typically set to `True` for styles that represent *Mathematica* output. Doing this means that when you reevaluate a particular piece of input, *Mathematica* will automatically delete the output that was previously generated from that input, and will overwrite it with new output.

The option `GeneratedCell` is set whenever a cell is generated by an external request to the front end rather than by an interactive operation within the front end. Thus, for example, any cell obtained as an output or side effect from a kernel evaluation will have `GeneratedCell -> True`. Cells generated by low-level functions designed to manipulate notebooks directly, such as `NotebookWrite` and `NotebookApply`, do not have the `GeneratedCell` option set.

<i>option</i>	<i>typical default value</i>	
<code>PageBreakAbove</code>	<code>Automatic</code>	whether to put a page break just above a particular cell
<code>PageBreakWithin</code>	<code>Automatic</code>	whether to allow a page break within a particular cell
<code>PageBreakBelow</code>	<code>Automatic</code>	whether to put a page break just below a particular cell
<code>GroupPageBreakWithin</code>	<code>Automatic</code>	whether to allow a page break within a particular group of cells

Options for controlling page breaks when cells are printed.

When you display a notebook on the screen, you can scroll continuously through it. But if you print the notebook out, you have to decide where page breaks will occur. A setting of `Automatic` for a page break option tells *Mathematica* to make a page break if necessary; `True` specifies that a page break should always be made, while `False` specifies that it should never be.

Page breaks set using the `PageBreakAbove` and `PageBreakBelow` options also determine the breaks between slides in a slide show. When creating a slide show, you will typically use a cell with a special named style to determine where each slide begins. This named style will have one of the page-breaking options set on it.

Additional functionality related to this tutorial has been introduced in subsequent versions of *Mathematica*. For the latest information, see Text Styling.

Text and Font Options

<i>option</i>	<i>typical default value</i>	
PageWidth	WindowSize	how wide to assume the page to be
TextAlignment	Left	how to align successive lines of text
TextJustification	0	how much to allow lines of text to be stretched to make them fit
Hyphenation	False	whether to allow hyphenation
ParagraphIndent	0	how many printer's points to indent the first line in each paragraph

General options for text formatting.

If you have a large block of text containing no explicit newline characters, then *Mathematica* will automatically break your text into a sequence of lines. The option `PageWidth` specifies how long each line should be allowed to be.

WindowSize	the width of the window on the screen
PaperWidth	the width of the page as it would be printed
Infinity	an infinite width (no line breaking)
<i>n</i>	explicit width given in printer's points

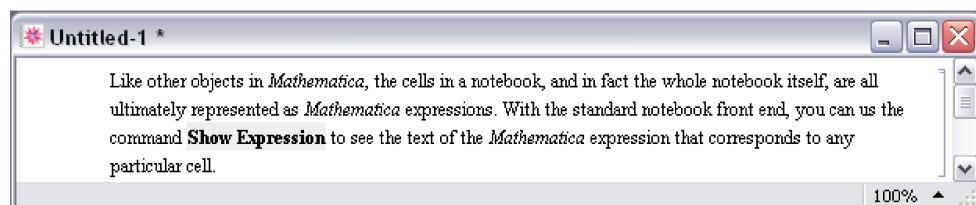
Settings for the `PageWidth` option in cells and notebooks.

The option `TextAlignment` allows you to specify how you want successive lines of text to be aligned. Since *Mathematica* normally breaks text only at space or punctuation characters, it is common to end up with lines of different lengths. Normally the variation in lengths will give your text a ragged boundary. But *Mathematica* allows you to adjust the spaces in successive lines of text so as to make the lines more nearly equal in length. The setting for `TextJustification` gives the fraction of extra space which *Mathematica* is allowed to add. `TextJustification -> 1` leads to “full justification” in which all complete lines of text are adjusted to be exactly the same length.

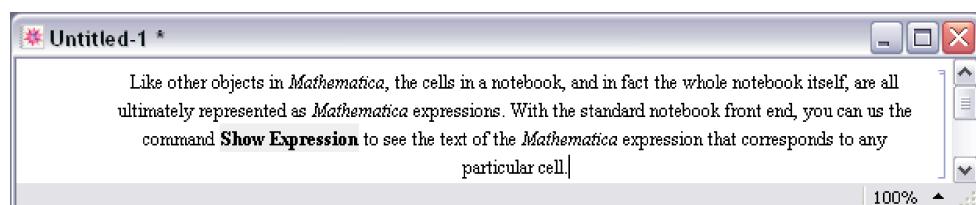
Left	aligned on the left
Right	aligned on the right
Center	centered
x	aligned at a position x running from -1 to $+1$ across the page

Settings for the `TextAlignment` option.

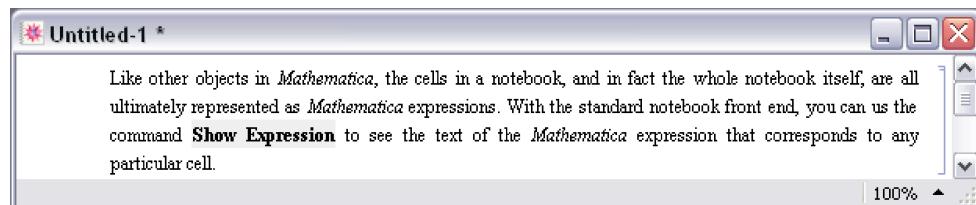
Here is text with `TextAlignment -> Left` and `TextJustification -> 0`.



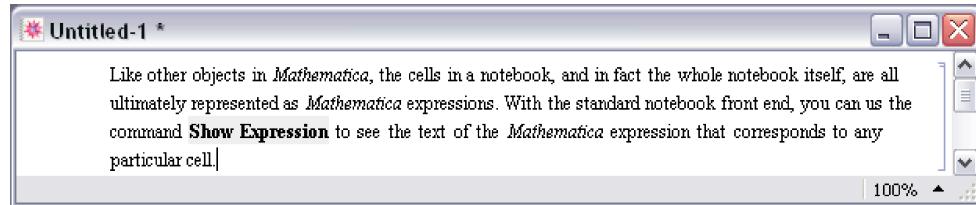
With `TextAlignment -> Center` the text is centered.



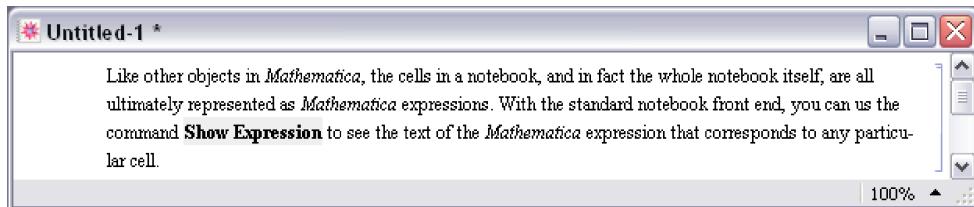
`TextJustification -> 1` adjusts word spacing so that both the left and right edges line up.



`TextJustification -> 0.5` reduces the degree of raggedness, but does not force the left and right edges to be precisely lined up.



With Hyphenation -> True the text is hyphenated.

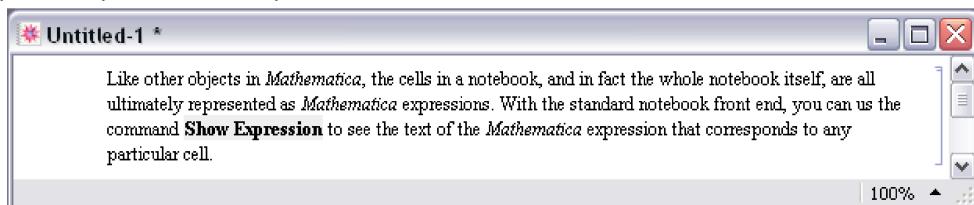


When you enter a block of text in a *Mathematica* notebook, *Mathematica* will treat any explicit newline characters that you type as paragraph breaks. The option `ParagraphIndent` allows you to specify how much you want to indent the first line in each paragraph. By giving a negative setting for `ParagraphIndent`, you can make the first line stick out to the left relative to subsequent lines.

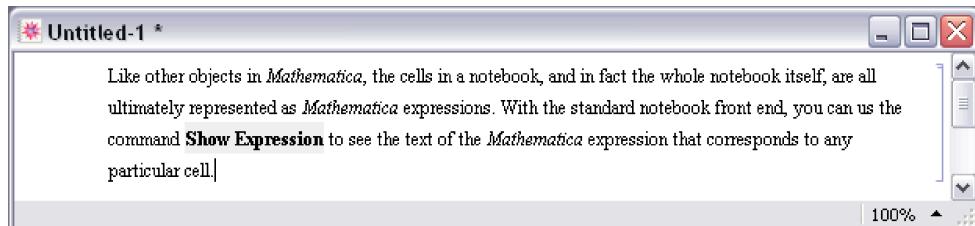
<code>LineSpacing -> {c, 0}</code>	leave space so that the total height of each line is c times the height of its contents
<code>LineSpacing -> {0, n}</code>	make the total height of each line exactly n printer's points
<code>LineSpacing -> {c, n}</code>	make the total height c times the height of the contents plus n printer's points
<hr/>	
<code>ParagraphSpacing -> {c, 0}</code>	leave an extra space of c times the height of the font before the beginning of each paragraph
<code>ParagraphSpacing -> {0, n}</code>	leave an extra space of exactly n printer's points before the beginning of each paragraph
<code>ParagraphSpacing -> {c, n}</code>	leave an extra space of c times the height of the font plus n printer's points

Options for spacing between lines of text.

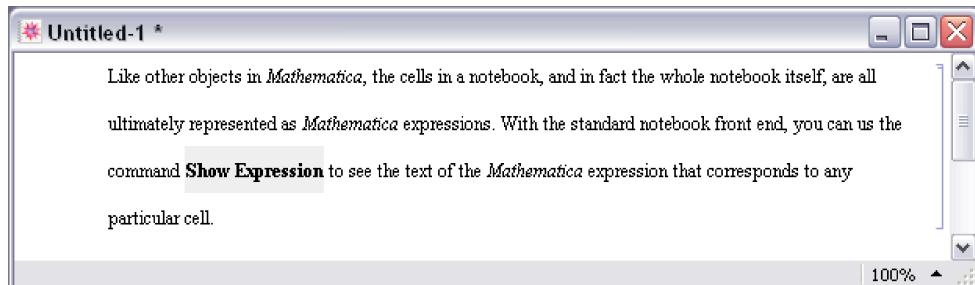
Here is some text with the default setting `LineSpacing -> {1, 1}`, which inserts just 1 printer's point of extra space between successive lines.



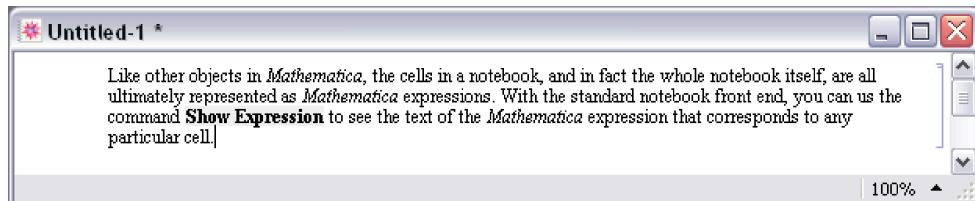
With LineSpacing -> {1, 5} the text is "looser".



LineSpacing -> {2, 0} makes the text double-spaced.



With LineSpacing -> {1, -2} the text is tight.



<i>option</i>	<i>typical default value</i>	
FontFamily	"Courier"	the family of font to use
FontSubstitutions	{ }	a list of substitutions to try for font family names
FontSize	12	the maximum height of characters in printer's points
FontWeight	"Bold"	the weight of characters to use
FontSlant	"Plain"	the slant of characters to use
FontTracking	"Plain"	the horizontal compression or expansion of characters
FontColor	GrayLevel[0]	the color of characters
Background	GrayLevel[1]	the color of the background for each character

Options for fonts.

"Courier"	text like this
"Times"	text like this
"Helvetica"	text like this

Some typical font family names.

FontWeight -> "Plain"	text like this
FontWeight -> "Bold"	text like this
FontWeight -> "ExtraBold"	text like this
FontSlant -> "Oblique"	<i>text like this</i>

Some settings of font options.

Mathematica allows you to specify the font that you want to use in considerable detail. Sometimes, however, the particular combination of font families and variations that you request may not be available on your computer system. In such cases, *Mathematica* will try to find the closest approximation it can. There are various additional options, such as `FontPostScriptName`, that you can set to help *Mathematica* find an appropriate font. In addition, you can set `FontSubstitutions` to be a list of rules that give replacements to try for font family names.

There are a great many fonts available for ordinary text. But for special technical characters, and even for Greek letters, far fewer fonts are available. The *Mathematica* system includes fonts that were built to support all of the various special characters that are used by *Mathematica*. There are three versions of these fonts: ordinary (like Times), monospaced (like Courier), and sans serif (like Helvetica).

For a given text font, *Mathematica* tries to choose the special character font that matches it best. You can help *Mathematica* to make this choice by giving rules for `"FontSerifed"` and `"FontMonospaced"` in the setting for the `FontProperties` option. You can also give rules for `"FontEncoding"` to specify explicitly from what font each character is to be taken.

Options for Expression Input and Output

<i>option</i>	<i>typical default value</i>	
AutoIndent	Automatic	whether to indent after an explicit Return character is entered
DelimiterFlashTime	0.3	the time in seconds to flash a delimiter when a matching one is entered
ShowAutoStyles	True	whether to show automatic style variations for syntactic and other constructs
ShowCursorTracker	True	whether an elliptical spot should appear momentarily to guide the eye if the cursor position jumps
ShowSpecialCharacters	True	whether to replace <code>\ [Name]</code> by a special character as soon as the <code>]</code> is entered
ShowStringCharacters	True	whether to display <code>"</code> when a string is entered
SingleLetterItalics	False	whether to put single-letter symbol names in italics
ZeroWidthTimes	False	whether to represent multiplication by a zero width character
InputAliases	{ }	additional <code>:name:</code> aliases to allow
InputAutoReplacements	{ " -> " -> " → ", ... }	strings to automatically replace on input
AutoItalicWords	{ "Mathematica", ... }	words to automatically put in italics
LanguageCategory	"NaturalLanguage"	what category of language to assume a cell contains for spell checking and hyphenation

Options associated with the interactive entering of expressions.

The option `singleLetterItalics` is typically set whenever a cell uses `TraditionalForm`.

Here is an expression entered with default options for a `StandardForm` input cell.

$$x^6 + 6 x^5 y + 15 x^4 y^2 + 20 x^3 y^3 + 15 x^2 y^4 + 6 x y^5 + y^6$$

Here is the same expression entered in a cell with `SingleLetterItalics -> True` and `ZeroWidthTimes -> True`.

$$x^6 + 6 x^5 y + 15 x^4 y^2 + 20 x^3 y^3 + 15 x^2 y^4 + 6 x y^5 + y^6$$

]

Built into *Mathematica* are a large number of aliases for common special characters. `InputAliases` allows you to add your own aliases for further special characters or for any other kind of *Mathematica* input. A rule of the form "`name`" \rightarrow `expr` specifies that `:name:` should immediately be replaced on input by `expr`.

Aliases are delimited by explicit `Esc` characters. The option `InputAutoReplacements` allows you to specify that certain kinds of input sequences should be immediately replaced even when they have no explicit delimiters. By default, for example, \rightarrow is immediately replaced by \Rightarrow . You can give a rule of the form "`seq`" \rightarrow "`rhs`" to specify that whenever `seq` appears as a token in your input, it should immediately be replaced by `rhs`.

<code>"NaturalLanguage"</code>	human natural language such as English
<code>"Mathematica"</code>	<i>Mathematica</i> input
<code>"Formula"</code>	mathematical formula
<code>None</code>	do no spell checking or hyphenation

Settings for `LanguageCategory` to control spell checking and hyphenation.

The option `LanguageCategory` allows you to tell *Mathematica* what type of contents it should assume cells have. This determines how spelling and structure should be checked, and how hyphenation should be done.

<i>option</i>	<i>typical default value</i>	
<code>StructuredSelection</code>	<code>False</code>	whether to allow only complete subexpressions to be selected
<code>DragAndDrop</code>	<code>False</code>	whether to allow drag-and-drop editing

Options associated with interactive manipulation of expressions.

Mathematica normally allows you to select any part of an expression that you see on the screen. Occasionally, however, you may find it useful to get *Mathematica* to allow only selections which correspond to complete subexpressions. You can do this by setting the option `StructuredSelection -> True`.

Here is an expression with a piece selected.

$$(-1 + x) (1 + x) \left(1 - x + x^2 - x^3 + x^4\right) \left(1 + x + x^2 + x^3 + x^4\right)$$

With `StructuredSelection -> True` only complete subexpressions can ever be selected.

$$(-1 + x) (1 + x) \left(1 - x + x^2 - x^3 + x^4\right) \left(1 + x + x^2 + x^3 + x^4\right)$$

$$(-1 + x) (1 + x) \left(1 - x + x^2 - x^3 + x^4\right) \left(1 + x + x^2 + x^3 + x^4\right)$$

$$(-1 + x) (1 + x) \left(1 - x + x^2 - x^3 + x^4\right) \left(1 + x + x^2 + x^3 + x^4\right)$$

Unlike most of the other options here, the `DragAndDrop` option can only be set for the entire front end, rather than for individual cells or cell styles.

`GridBox[data,opts]`

give options that apply to a particular grid box

`StyleBox[boxes,opts]`

give options that apply to all boxes in *boxes*

`Cell[contents,opts]`

give options that apply to all boxes in *contents*

`Cell[contents,GridBoxOptions->opts]`

give default options settings for all `GridBox` objects in *contents*

Examples of specifying options for the display of expressions.

As discussed in "Textual Input and Output", *Mathematica* provides many options for specifying how expressions should be displayed. By using `StyleBox[boxes, opts]` you can apply such options to collections of boxes. But *Mathematica* is set up so that any option that you can give to a `StyleBox` can also be given to a complete `Cell` object, or even a complete `Notebook`. Thus, for example, options like `Background` and `LineIndent` can be given to complete cells as well as to individual `StyleBox` objects.

There are some options that apply only to a particular type of box, such as `GridBox`. Usually these options are best given separately in each `GridBox` where they are needed. But sometimes you may want to specify default settings to be inherited by all `GridBox` objects that appear in a particular cell. You can do this by giving these default settings as the value of the option `GridBoxOptions` for the whole cell.

For most box types named `XXXBox`, *Mathematica* provides a cell option `XXXBoxOptions` that allows you to specify the default options settings for that type of box. Box types which take options can also have their options set in a stylesheet by defining the `xxx` style. The stylesheets which come with *Mathematica* define many such styles.

Options for Notebooks

- Use the Option Inspector menu to change options interactively.
- Use `SetOptions [obj, options]` from the kernel.
- Use `CreateWindow [options]` to create a new notebook with specified options.

Ways to change the overall options for a notebook.

This creates a notebook displayed in a 40x30 window with a thin frame.

```
In[1]:= CreateWindow[WindowFrame -> "ThinFrame",WindowSize -> {40, 30}]
```

<i>option</i>	<i>typical default value</i>	
<code>StyleDefinitions</code>	"Default.nb"	the basic stylesheet to use for the notebook
<code>ScreenStyleEnvironment</code>	"Working"	the style environment to use for screen display
<code>PrintingStyleEnvironment</code>	"Printout"	the style environment to use for printing

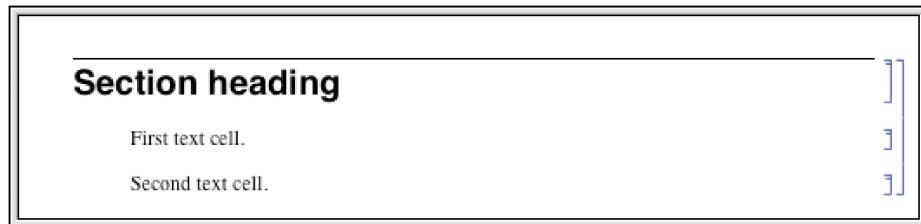
Style options for a notebook.

In giving style definitions for a particular notebook, *Mathematica* allows you either to reference another notebook, or explicitly to include the `Notebook` expression that defines the styles.

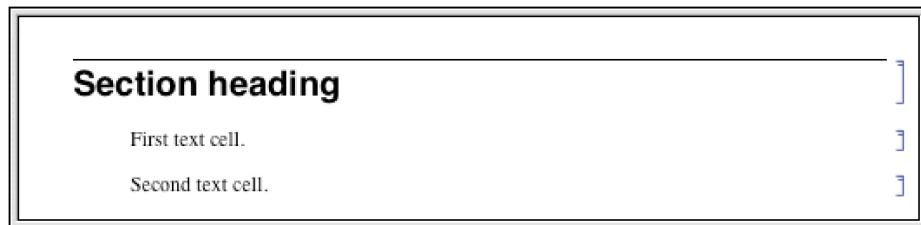
<i>option</i>	<i>typical default value</i>	
<code>CellGrouping</code>	Automatic	how to group cells in the notebook
<code>ShowPageBreaks</code>	False	whether to show where page breaks would occur if the notebook were printed
<code>NotebookAutoSave</code>	False	whether to automatically save the notebook after each piece of output

General options for notebooks.

With `CellGrouping -> Automatic`, cells are automatically grouped based on their style.



With `CellGrouping -> Manual`, you have to group cells by hand.



<i>option</i>	<i>typical default value</i>	
<code>DefaultNewCellStyle</code>	"Input"	the default style for new cells created in the notebook
<code>DefaultDuplicateCellStyle</code>	"Input"	the default style for cells created by automatic duplication of existing cells

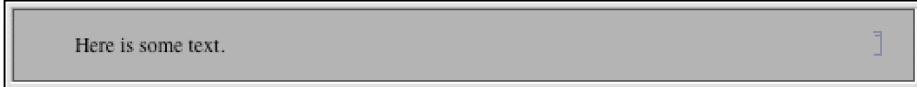
Options specifying default styles for cells created in a notebook.

Mathematica allows you to take any cell option and set it at the notebook level, thereby specifying a global default for that option throughout the notebook.

<i>option</i>	<i>typical default value</i>	
<code>Editable</code>	True	whether to allow cells in the notebook to be edited
<code>Selectable</code>	True	whether to allow cells to be selected
<code>Deletable</code>	True	whether to allow cells to be deleted
<code>ShowSelection</code>	True	whether to show the current selection highlighted
<code>Background</code>	<code>GrayLevel[1]</code>	what background color to use for the notebook
<code>Magnification</code>	1	at what magnification to display the notebook
<code>PageWidth</code>	<code>WindowWidth</code>	how wide to allow the contents of cells to be

A few cell options that are often set at the notebook level.

Here is a notebook with the `Background` option set at the notebook level.



Here is some text.

<i>option</i>	<i>typical default value</i>	
<code>Visible</code>	<code>True</code>	whether the window should be visible on the screen
<code>WindowSize</code>	<code>{Automatic, Automatic}</code>	the width and height of the window in printer's points
<code>WindowMargins</code>	<code>Automatic</code>	the margins to leave around the window when it is displayed on the screen
<code>WindowFrame</code>	<code>"Normal"</code>	the type of frame to draw around the window
<code>WindowElements</code>	<code>{"StatusArea", ...}</code>	elements to include in the window
<code>WindowTitle</code>	<code>Automatic</code>	what title should be displayed for the window
<code>WindowMovable</code>	<code>True</code>	whether to allow the window to be moved around on the screen
<code>WindowFloating</code>	<code>False</code>	whether the window should always float on top of other windows
<code>WindowClickSelect</code>	<code>True</code>	whether the window should become selected if you click in it
<code>DockedCells</code>	<code>{}</code>	a list of cells specifying the content of a docked area at the top of the window

Characteristics of the notebook window.

`WindowSize` allows you to specify how large you want a window to be; `WindowMargins` allows you to specify where you want the window to be placed on your screen. The setting `WindowMargins -> {{left, right}, {bottom, top}}` gives the margins in pixels to leave around your window on the screen. Often only two of the margins will be set explicitly; the others will be `Automatic`, indicating that these margins will be determined from the particular size of screen that you use.

`WindowClickSelect` is the principal option that determines whether a window acts like a palette. Palettes are generally windows with content that acts upon other windows, rather than windows which need to be selected for their own ends. Palettes also generally have a collection of other option settings such as `WindowFloating -> True` and `WindowFrame -> "Palette"`.

`DockedCells` allows you to specify any content that you want to stay at the top of a window and never scroll offscreen. A typical use of the `DockedCells` option is to define a custom toolbar. Many default stylesheets have the `DockedCells` option defined in certain environments to create toolbars for purposes such as presenting slideshows and editing package files.

"Normal"	an ordinary window
"Palette"	a palette window
"ModelessDialog"	a modeless dialog box window
"ModalDialog"	a modal dialog box window
"MovableModalDialog"	a modal dialog box window that can be moved around the screen
"ThinFrame"	an ordinary window with a thin frame
"Frameless"	an ordinary window with no frame at all
"Generic"	a window with a generic border

Typical possible settings for `WindowFrame`.

Mathematica allows many different types of windows. The details of how particular windows are rendered may differ slightly from one computer system to another, but their general form is always the same. `WindowFrame` specifies the type of frame to draw around the window. `WindowElements` gives a list of specific elements to include in the window.

"StatusArea"	an area used to display status messages, such as those created by <code>StatusArea</code>
"MagnificationPopUp"	a popup menu of common magnifications
"HorizontalScrollBar"	a scroll bar for horizontal motion
"VerticalScrollBar"	a scroll bar for vertical motion

Some typical possible entries in the `WindowElements` list.

Here is a window with a status area and horizontal scroll bar, but no magnification popup or vertical scroll bar.



Global Options for the Front End

In the standard notebook front end, *Mathematica* allows you to set a large number of global options. The values of all these options are by default saved in a “preferences file”, and are automatically reused when you run *Mathematica* again. These options include all the settings which can be made using the **Preferences** dialog.

style definitions	default style definitions to use for new notebooks
file locations	directories for finding notebooks and system files
data export options	how to export data in various formats
character encoding options	how to encode special characters
language options	what language to use for text
message options	how to handle messages generated by <i>Mathematica</i>
dialog settings	choices made in dialog boxes
system configuration	private options for specific computer systems

Some typical categories of global options for the front end.

You can access global front end options from the kernel by using `Options[$FrontEnd, name]`. But more often, you will want to access these options interactively using the Option Inspector in the front end.

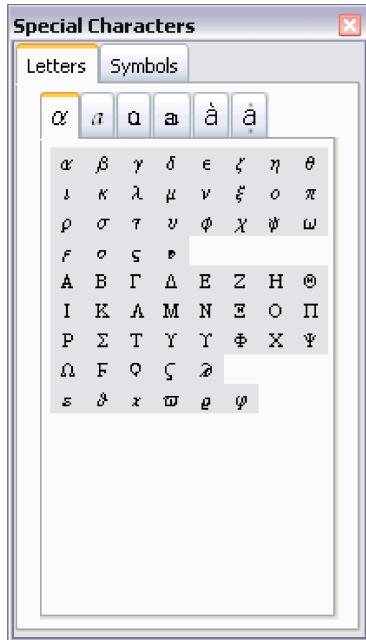
Mathematical and Other Notation

Mathematical Notation in Notebooks

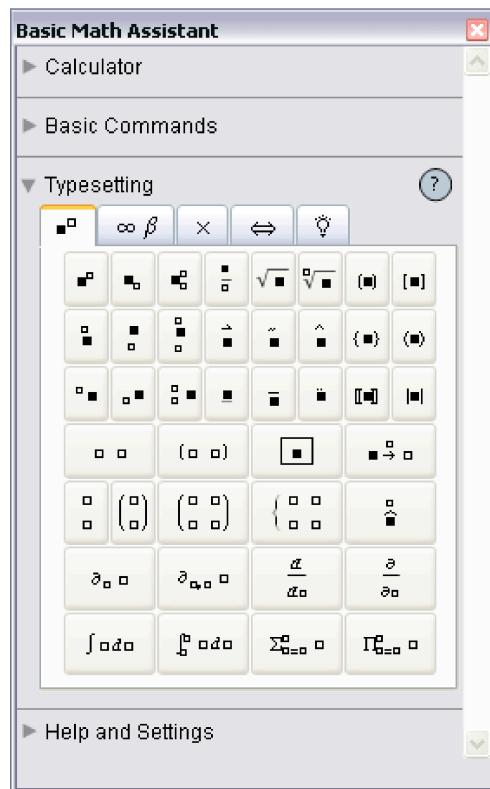
If you use a text-based interface to *Mathematica*, then the input you give must consist only of characters that you can type directly on your computer keyboard. But if you use a notebook interface then other kinds of input become possible.

There are palettes provided which operate like extensions of your keyboard, and which have buttons that you can click to enter particular forms. You can access standard palettes using the **Palettes** menu.

Clicking the π button in this palette will enter a Pi into your notebook.



Clicking the first button in this palette will create an empty structure for entering a power. You can use the mouse to fill in the structure.



You can also give input by using special keys on your keyboard. Pressing one of these keys does not lead to an ordinary character being entered, but instead typically causes some action to occur or some structure to be created.

<code>Esc p Esc</code>	the symbol π
<code>Esc inf Esc</code>	the symbol ∞
<code>Esc ee Esc</code>	the symbol e for the exponential constant (equivalent to <code>E</code>)
<code>Esc ii Esc</code>	the symbol i for $\sqrt{-1}$ (equivalent to <code>I</code>)
<code>Esc deg Esc</code>	the symbol $^\circ$ (equivalent to <code>Degree</code>)
<code>Ctrl+^</code> or <code>Ctrl+6</code>	go to the superscript for a power
<code>Ctrl+/_</code>	go to the denominator for a fraction
<code>Ctrl+@</code> or <code>Ctrl+2</code>	go into a square root
<code>Ctrl+Space</code>	return from a superscript, denominator or square root

A few ways to enter special notations on a standard English-language keyboard.

Here is a computation entered using ordinary characters on a keyboard.

In[1]:= N[Pi^2 / 6]

Out[1]= 1.64493

Here is the same computation entered using a palette or special keys.

In[2]:= N[π²/6]

Out[2]= 1.64493

Here is an actual sequence of keys that can be used to enter the input.

In[3]:= N[Esc p Esc Ctrl+^ 2 Ctrl+Space Ctrl+/ 6 Ctrl+Space]

Out[3]= 1.64493

In a traditional computer language such as C, Fortran, Java or Perl, the input you give must always consist of a string of ordinary characters that can be typed directly on a keyboard. But the *Mathematica* language also allows you to give input that contains special characters, superscripts, built-up fractions, and so on.

The language incorporates many features of traditional mathematical notation. But you should realize that the goal of the language is to provide a precise and consistent way to specify computations. And as a result, it does not follow all of the somewhat haphazard details of traditional mathematical notation.

Nevertheless, as discussed in "Forms of Input and Output", it is always possible to get *Mathematica* to produce *output* that imitates every aspect of traditional mathematical notation. And it is also possible for *Mathematica* to import text that uses such notation, and to some extent to translate it into its own more precise language.

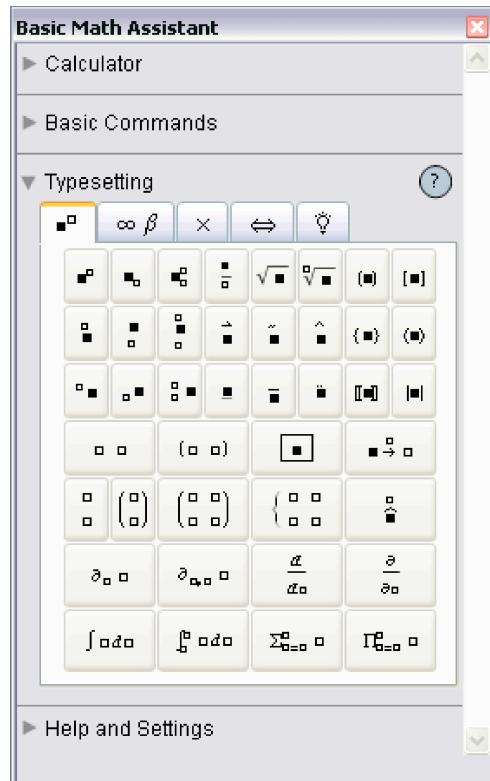
Mathematical Notation in Notebooks

If you use the notebook front end for *Mathematica*, then you can enter some of the operations discussed here in special ways.

$\sum_{i=i_{\min}}^{i_{\max}} f$	<code>Sum [f, {i, i_{min}, i_{max}}]</code>	sum
$\prod_{i=i_{\min}}^{i_{\max}} f$	<code>Product [f, {i, i_{min}, i_{max}}]</code>	product
$\int f dx$	<code>Integrate [f, x]</code>	indefinite integral
$\int_{x_{\min}}^{x_{\max}} f dx$	<code>Integrate [f, {x, x_{min}, x_{max}}]</code>	definite integral
$\partial_x f$	<code>D [f, x]</code>	partial derivative
$\partial_{x,y} f$	<code>D [f, x, y]</code>	multivariate partial derivative

Special and ordinary ways to enter mathematical operations in notebooks.

This one of the standard palettes for entering mathematical operations. When you click a button in the palette, the form shown in the button is inserted into your notebook, with the black square replaced by whatever you had selected in the notebook.



<code>Esc sum Esc</code>	summation sign Σ
<code>Esc prod Esc</code>	product sign \prod
<code>Esc int Esc</code>	integral sign \int
<code>Esc dd Esc</code>	special differential d for use in integrals
<code>Esc pd Esc</code>	partial derivative ∂
<code>Ctrl+_ or Ctrl+-</code>	move to the subscript position or lower limit of an integral
<code>Ctrl^_ or Ctrl+6</code>	move to the superscript position or upper limit of an integral
<code>Ctrl++ or Ctrl+=</code>	move to the underscript position or lower limit of a sum or product
<code>Ctrl+& or Ctrl+7</code>	move to the overscript position or upper limit of a sum or product
<code>Ctrl+%</code> or <code>Ctrl+5</code>	switch between upper and lower positions
<code>Ctrl+Space</code>	return from upper or lower positions

Ways to enter special notations on a standard English-language keyboard.

You can enter an integral like this. Be sure to use the special differential d entered as `Esc dd Esc`, not just an ordinary d .

`In[1]:=` $\int x^n dx$

`Out[1]=`
$$\frac{x^{1+n}}{1+n}$$

Here is the actual key sequence you type to get the input.

`In[2]:=` **Esc int Esc** x **Ctrl+^** n **Ctrl+Space** **Esc dd Esc** x

`Out[2]=`
$$\frac{x^{1+n}}{1+n}$$

When entering a sum, product or integral that has limits, you can create the first limit using the standard control sequences for subscripts, superscripts, underscripts, or overscripts. However, you must use `Ctrl+%` to create the second limit.

You can enter a sum like this.

`In[3]:=`
$$\sum_{x=0}^n x$$

`Out[3]=`
$$\frac{1}{2} n (1+n)$$

Here is the actual key sequence you type to get the input.

```
In[4]:= Esc sum Esc Ctrl+= x=0 Ctrl+% n Ctrl+Space x
Out[4]= 1
          - n (1 + n)
          2
```

Special Characters

Built into *Mathematica* are a large number of special characters intended for use in mathematical and other notation. "Listing of Named Characters" gives a complete listing.

Each special character is assigned a full name such as `\[Infinity]`. More common special characters are also assigned aliases, such as `Esc inf Esc`. You can set up additional aliases using the `InputAliases` notebook option discussed in "Options for Expression Input and Output".

For special characters that are supported in standard dialects of TeX, *Mathematica* also allows you to use aliases based on TeX names. Thus, for example, you can enter `\[Infinity]` using the alias `Esc \infty Esc`. *Mathematica* also supports aliases such as `Esc infinity Esc` based on names used in SGML and HTML.

Standard system software on many computer systems also supports special key combinations for entering certain special characters. On a Macintosh, for example, `Option+5` will produce ∞ in most fonts. With the notebook front end *Mathematica* automatically allows you to use special key combinations when these are available, and with a text-based interface you can get *Mathematica* to accept such key combinations if you set an appropriate value for `$CharacterEncoding`.

-
- Use a full name such as `\[Infinity]`
 - Use an alias such as `Esc inf Esc`
 - Use a TeX alias such as `Esc \infty Esc`
 - Use an SGML or HTML alias such as `Esc infinity Esc`
 - Click a button in a palette
 - Use a special key combination supported by your computer system

Ways to enter special characters.

In a *Mathematica* notebook, you can use special characters just like you use standard keyboard characters. You can include special characters both in ordinary text and in input that you intend to give to *Mathematica*.

Some special characters are set up to have an immediate meaning to *Mathematica*. Thus, for example, π is taken to be the symbol `Pi`. Similarly, \geq is taken to be the operator `>=`, while \cup is equivalent to the function `Union`.

π and \geq have immediate meanings in *Mathematica*.

In[1]:= $\pi \geq 3$

Out[1]= `True`

\cup or `\[Union]` is immediately interpreted as the `Union` function.

In[2]:= `{a, b, c} \cup {c, d, e}`

Out[2]= `{a, b, c, d, e}`

\sqcup or `\[SquareUnion]` has no immediate meaning to *Mathematica*.

In[3]:= `{a, b, c} \sqcup {c, d, e}`

Out[3]= `{a, b, c} \sqcup {c, d, e}`

Among ordinary characters such as `E` and `i`, some have an immediate meaning to *Mathematica*, but most do not. And the same is true of special characters.

Thus, for example, while π and ∞ have an immediate meaning to *Mathematica*, λ and \mathcal{L} do not.

This allows you to set up your own definitions for λ and \mathcal{L} .

λ has no immediate meaning in *Mathematica*.

In[4]:= `\lambda[2] + \lambda[3]`

Out[4]= `\lambda[2] + \lambda[3]`

This defines a meaning for λ .

In[5]:= `\lambda[x_] := \sqrt{x^2 - 1}`

Now *Mathematica* evaluates λ just as it would any other function.

In[6]:= `\lambda[2] + \lambda[3]`

Out[6]= `2 \sqrt{2} + \sqrt{3}`

Characters such as λ and \mathcal{L} are treated by *Mathematica* as letters—just like ordinary keyboard letters like a or b.

But characters such as \oplus and \sqcup are treated by *Mathematica* as *operators*. And although these particular characters are not assigned any built-in meaning by *Mathematica*, they are nevertheless required to follow a definite *syntax*.

\sqcup is an infix operator.

```
In[7]:= {a, b, c}  $\sqcup$  {c, d, e}
Out[7]= {a, b, c}  $\sqcup$  {c, d, e}
```

The definition assigns a meaning to the \sqcup operator.

```
In[8]:= x_  $\sqcup$  y_ := Join[x, y]
```

Now \sqcup can be evaluated by *Mathematica*.

```
In[9]:= {a, b, c}  $\sqcup$  {c, d, e}
Out[9]= {a, b, c, c, d, e}
```

The details of how input you give to *Mathematica* is interpreted depends on whether you are using `StandardForm` or `TraditionalForm`, and on what additional information you supply in `InterpretationBox` and similar constructs.

But unless you explicitly override its built-in rules by giving your own definitions for `MakeExpression`, *Mathematica* will always assign the same basic syntactic properties to any particular special character.

These properties not only affect the interpretation of the special characters in *Mathematica* input, but also determine the structure of expressions built with these special characters. They also affect various aspects of formatting; operators, for example, have extra space left around them, while letters do not.

Letters	a, E, π , Σ , \mathcal{L} , etc.
Letter-like forms	∞ , \emptyset , \mathcal{U} , \mathcal{L} , etc.
Operators	\oplus , ∂ , \approx , \rightleftharpoons , etc.

Types of special characters.

In using special characters, it is important to make sure that you have the correct character for a particular purpose. There are quite a few examples of characters that look similar, yet are in fact quite different.

A common issue is operators whose forms are derived from letters. An example is \sum or `\[Sum]`, which looks very similar to Σ or `\[CapitalSigma]`.

As is typical, however, the operator form \sum is slightly less elaborate and more stylized than the letter form Σ . In addition, \sum is an extensible character which grows depending on the summand, while Σ has a size determined only by the current font.

A	A	<code>\[CapitalAlpha]</code>	A	μ	μ	<code>\[Micro]</code>	<code>\[Mu]</code>
\AA	\AA	<code>\[Angstrom]</code>	\AA	\emptyset	\emptyset	<code>\[EmptySet]</code>	<code>\[CapitalOSlash]</code>
d	d	<code>\[DifferentialD]</code>	d	\prod	\prod	<code>\[Product]</code>	<code>\[CapitalPi]</code>
e	e	<code>\[ExponentialE]</code>	e	\sum	\sum	<code>\[Sum]</code>	<code>\[CapitalSigma]</code>
\epsilon	\epsilon	<code>\[Element]</code>	\epsilon	\top	\top	<code>\[Transpose]</code>	<code>\[T]</code>
i	i	<code>\[ImaginaryI]</code>	i	\cup	\cup	<code>\[Union]</code>	<code>\[U]</code>

Different characters that look similar.

In cases such as `\[CapitalAlpha]` versus A , both characters are letters. However, *Mathematica* treats these characters as different, and in some fonts, for example, they may look quite different.

The result contains four distinct characters.

```
In[10]:= Union[{A, \[CapitalAlpha], \[DifferentialD], \[Mu]}]
Out[10]= {A, \[CapitalAlpha], \[DifferentialD], \[Mu]}
```

Traditional mathematical notation occasionally uses ordinary letters as operators. An example is the d in a differential such as dx that appears in an integral.

To make *Mathematica* have a precise and consistent syntax, it is necessary at least in `StandardForm` to distinguish between an ordinary d and the d used as a differential operator.

The way *Mathematica* does this is to use a special character d or `\[DifferentialD]` as the differential operator. This special character can be entered using the alias `Esc dd Esc`.

Mathematica uses a special character for the differential operator, so there is no conflict with an ordinary d .

In[11]:= $\int x^d dx$

Out[11]= $\frac{x^{1+d}}{1+d}$

When letters and letter-like forms appear in *Mathematica* input, they are typically treated as names of symbols. But when operators appear, functions must be constructed that correspond to these operators. In almost all cases, what *Mathematica* does is to create a function whose name is the full name of the special character that appears as the operator.

Mathematica constructs a `CirclePlus` function to correspond to the operator \oplus , whose full name is `\[CirclePlus]`.

In[12]:= `a ⊕ b ⊕ c // FullForm`

Out[12]//FullForm= `CirclePlus[a, b, c]`

This constructs an `And` function, which happens to have built-in evaluation rules in *Mathematica*.

In[13]:= `a ∧ b ∧ c // FullForm`

Out[13]//FullForm= `And[a, b, c]`

Following the correspondence between operator names and function names, special characters such as \cup that represent built-in *Mathematica* functions have names that correspond to those functions. Thus, for example, \div is named `\[Divide]` to correspond to the built-in *Mathematica* function `Divide`, and \Rightarrow is named `\[Implies]` to correspond to the built-in function `Implies`.

In general, however, special characters in *Mathematica* are given names that are as generic as possible, so as not to prejudice different uses. Most often, characters are thus named mainly according to their appearance. The character \oplus is therefore named `\[CirclePlus]`, rather than, say `\[DirectSum]`, and \approx is named `\[TildeTilde]` rather than, say, `\[ApproximateEqual]`.

$\times \times$	<code>\[Times], \[Cross]</code>	$*$	$*$	<code>\[Star], *</code>
$\wedge \wedge$	<code>\[And], \[Wedge]</code>	$\backslash \backslash$	\backslash	<code>\[Backslash], \[</code>
$\vee \vee$	<code>\[Or], \[Vee]</code>	$\cdot \cdot$	\cdot	<code>\[CenterDot], .</code>
$\rightarrow \rightarrow$	<code>\[Rule], \[RightArrow]</code>	$\wedge \wedge$	\wedge	<code>\[Wedge], ^</code>
$\Rightarrow \Rightarrow$	<code>\[Implies], \[DoubleRightArrow]</code>	$ $	$ $	<code>\[VerticalBar], </code>
$= =$	<code>\[LongEqual], =</code>	$ $	$ $	<code>\[VerticalSeparator], </code>
$\{ \}$	<code>\[Piecewise], {</code>	$\langle <$	$<$	<code>\[LeftAngleBracket], <</code>

Different operator characters that look similar.

There are sometimes characters that look similar but which are used to represent different operators. An example is `\[Times]` and `\[Cross]`. `\[Times]` corresponds to the ordinary Times function for multiplication; `\[Cross]` corresponds to the Cross function for vector cross products. The \times for `\[Cross]` is drawn slightly smaller than \times for `\[Times]`, corresponding to usual careful usage in mathematical typography.

The `\[Times]` operator represents ordinary multiplication.

```
In[14]:= {5, 6, 7} × {2, 3, 1}
Out[14]= {10, 18, 7}
```

The `\[Cross]` operator represents vector cross products.

```
In[15]:= {5, 6, 7} × {2, 3, 1}
Out[15]= {-15, 9, 3}
```

The two operators display in a similar way—with `\[Times]` slightly larger than `\[Cross]`.

```
In[16]:= {a × b, a × b}
Out[16]= {a b, a × b}
```

In the example of `\[And]` and `\[Wedge]`, the `\[And]` operator—which happens to be drawn slightly larger—corresponds to the built-in *Mathematica* function `And`, while the `\[Wedge]` operator has a generic name based on the appearance of the character and has no built-in meaning.

You can mix $\backslash[\text{Wedge}]$ and $\backslash[\text{And}]$ operators. Each has a definite precedence.

```
In[17]:= a ^ b ∧ c ^ d // FullForm
Out[17]//FullForm= And[Wedge[a, b], Wedge[c, d]]
```

Some of the special characters commonly used as operators in mathematical notation look similar to ordinary keyboard characters. Thus, for example, \wedge or $\backslash[\text{Wedge}]$ looks similar to the \wedge character on a standard keyboard.

Mathematica interprets a raw \wedge as a power. But it interprets \wedge as a generic wedge function. In cases such as this where there is a special character that looks similar to an ordinary keyboard character, the convention is to use the ordinary keyboard character as the alias for the special character. Thus, for example, $\text{Esc} \wedge \text{Esc}$ is the alias for $\backslash[\text{Wedge}]$.

The raw \wedge is interpreted as a power, but the $\text{Esc} \wedge \text{Esc}$ is a generic wedge operator.

```
In[18]:= {x ^ y, x Esc ^ Esc y}
Out[18]= {x^y, x^y}
```

A related convention is that when a special character is used to represent an operator that can be typed using ordinary keyboard characters, those characters are used in the alias for the special character. Thus, for example, $\text{Esc} \rightarrow \text{Esc}$ is the alias for \rightarrow or $\backslash[\text{Rule}]$, while $\text{Esc} \& \& \text{Esc}$ is the alias for \wedge or $\backslash[\text{And}]$.

$\text{Esc} \rightarrow \text{Esc}$ is the alias for $\backslash[\text{Rule}]$, and $\text{Esc} \& \& \text{Esc}$ for $\backslash[\text{And}]$.

```
In[19]:= {x Esc->Esc y, x Esc && Esc y} // FullForm
Out[19]//FullForm= List[Rule[x, y], And[x, y]]
```

The most extreme case of characters that look alike but work differently occurs with vertical bars.

form	character name	alias	interpretation
$x y$			Alternatives[x,y]
$x y$	$\backslash[\text{VerticalSeparator}]$	Esc Esc	VerticalSeparator[x,y]
$x\backslash y$	$\backslash[\text{VerticalBar}]$	Esc_ Esc	VerticalBar[x,y]
$ x $	$\backslash[\text{LeftBracketingBar}]$	Esc1 Esc	BracketingBar[x]
	$\backslash[\text{RightBracketingBar}]$	Escr Esc	

Different types of vertical bars.

Notice that the alias for `\[VerticalBar]` is `Esc_| Esc`, while the alias for the somewhat more common `\[VerticalSeparator]` is `Esc | Esc`. *Mathematica* often gives similar-looking characters similar aliases; it is a general convention that the aliases for the less commonly used characters are distinguished by having spaces at the beginning.

<code>Esc nnn Esc</code>	built-in alias for a common character
<code>Esc _ nnn Esc</code>	built-in alias for similar but less common character
<code>Esc . nnn Esc</code>	alias globally defined in a <i>Mathematica</i> session
<code>Esc , nnn Esc</code>	alias defined in a specific notebook

Conventions for special character aliases.

The notebook front end for *Mathematica* often allows you to set up your own aliases for special characters. If you want to, you can overwrite the built-in aliases. But the convention is to use aliases that begin with a dot or comma.

Note that whatever aliases you may use to enter special characters, the full names of the characters will always be used when the characters are stored in files.

Names of Symbols and Mathematical Objects

Mathematica by default interprets any sequence of letters or letter-like forms as the name of a symbol.

All these are treated by *Mathematica* as symbols.

```
In[1]:= {ξ, Σα, R∞, ℋ, ℙ, ↳ABC, ■X, m...n}
```

```
Out[1]= {ξ, Σα, R∞, ℋ, ℙ, ↳ABC, ■X, m...n}
```

form	character name	alias	interpretation
π	<code>\[Pi]</code>	<code>Esc p Esc, Esc pi Esc</code>	equivalent to π
∞	<code>\[Infinity]</code>	<code>Esc inf Esc</code>	equivalent to ∞
e	<code>\[ExponentialE]</code>	<code>Esc ee Esc</code>	equivalent to e
i	<code>\[ImaginaryI]</code>	<code>Esc ii Esc</code>	equivalent to i
j	<code>\[ImaginaryJ]</code>	<code>Esc jj Esc</code>	equivalent to j

Symbols with built-in meanings whose names do not start with capital English letters.

Essentially all symbols with built-in meanings in *Mathematica* have names that start with capital English letters. Among the exceptions are e and i , which correspond to E and I respectively.

Forms such as e are used for both input and output in `StandardForm`.

In[2]:= { $e^{(2\pi i)}$, e^π }

Out[2]= {1, e^π }

In `OutputForm` e is output as E .

In[3]:= `OutputForm[%]`

Out[3]/OutputForm= {1, E^{Pi} }

In written material, it is standard to use very short names—often single letters—for most of the mathematical objects that one considers. But in *Mathematica*, it is usually better to use longer and more explicit names.

In written material you can always explain that a particular single-letter name means one thing in one place and another in another place. But in *Mathematica*, unless you use different contexts, a global symbol with a particular name will always be assumed to mean the same thing.

As a result, it is typically better to use longer names, which are more likely to be unique, and which describe more explicitly what they mean.

For variables to which no value will be assigned, or for local symbols, it is nevertheless convenient and appropriate to use short, often single-letter, names.

It is sensible to give the global function `LagrangianL` a long and explicit name. The local variables can be given short names.

In[4]:= `LagrangianL[$\phi_$, $\mu_$] = ($\square\phi$)2 + $\mu^2 \phi^2$`

Out[4]= $\mu^2 \phi^2 + (\square\phi)^2$

<i>form</i>	<i>input</i>	<i>interpretation</i>
x_n	$x \text{ Ctrl+}_n \text{ Ctrl+Space}$	Subscript [x, n]
x_+	$x \text{ Ctrl+}_+ \text{ Ctrl+Space}$	SubPlus [x]
x_-	$x \text{ Ctrl+}_- \text{ Ctrl+Space}$	SubMinus [x]
x_*	$x \text{ Ctrl+}_* \text{ Ctrl+Space}$	SubStar [x]
x^+	$x \text{ Ctrl+}^+ \text{ Ctrl+Space}$	SuperPlus [x]
x^-	$x \text{ Ctrl+}^- \text{ Ctrl+Space}$	SuperMinus [x]
x^*	$x \text{ Ctrl+}^* \text{ Ctrl+Space}$	SuperStar [x]
x^\dagger	$x \text{ Ctrl+}^\dagger \text{ Esc dg Esc Ctrl+Space}$	SuperDagger [x]
\bar{x}	$x \text{ Ctrl+}\& \text{ Ctrl+Space}$	OverBar [x]
\vec{x}	$x \text{ Ctrl+}\& \text{ Esc vec Esc Ctrl+Space}$	OverVector [x]
\tilde{x}	$x \text{ Ctrl+}\& \sim \text{ Ctrl+Space}$	OverTilde [x]
\hat{x}	$x \text{ Ctrl+}\& \wedge \text{ Ctrl+Space}$	OverHat [x]
\dot{x}	$x \text{ Ctrl+}\& . \text{ Ctrl+Space}$	OverDot [x]
\underline{x}	$x \text{ Ctrl+}++ \text{ Ctrl+Space}$	UnderBar [x]
\mathbf{x}	<code>Style[x, Bold]</code>	x

Creating objects with annotated names.

Note that with a notebook front end, you can change the style of text using menu items.

<i>option</i>	<i>typical default value</i>	
<code>SingleLetterItalics</code>	<code>False</code>	whether to use italics for single-letter symbol names
<code>MultiLetterItalics</code>	<code>False</code>	whether to use italics for multi-letter symbol names
<code>SingleLetterStyle</code>	<code>None</code>	the style name or directives to use for single-letter symbol names
<code>MultiLetterStyle</code>	<code>None</code>	the style name or directives to use for multi-letter symbol names

Options for cells in a notebook.

It is conventional in traditional mathematical notation that names consisting of single ordinary English letters are normally shown in italics, while other names are not. If you use `TraditionalForm`, then *Mathematica* will by default follow this convention. You can explicitly specify whether you want the convention followed by setting the `SingleLetterItalics` option for particular cells or cell styles. You can further specify styles for names using single English letters or multiple English letters by specifying values for the options `SingleLetterStyle` and `MultiLetterStyle`.

Letters and Letter-like Forms

Greek Letters

<i>form</i>	<i>full name</i>	<i>aliases</i>	<i>form</i>	<i>full name</i>	<i>aliases</i>
α	<code>\[Alpha]</code>	<code>:a;, :alpha;</code>	A	<code>\[CapitalAlpha]</code>	<code>:A;, :Alpha;</code>
β	<code>\[Beta]</code>	<code>:b;, :beta;</code>	B	<code>\[CapitalBeta]</code>	<code>:B;, :Beta;</code>
γ	<code>\[Gamma]</code>	<code>:g;, :gamma;</code>	Γ	<code>\[CapitalGamma]</code>	<code>:G;, :Gamma;</code>
δ	<code>\[Delta]</code>	<code>:d;, :delta;</code>	Δ	<code>\[CapitalDelta]</code>	<code>:D;, :Delta;</code>
ϵ	<code>\[Epsilon]</code>	<code>:e;, :epsilon;</code>	E	<code>\[CapitalEpsilon]</code>	<code>:E;, :Epsilon;</code>
ε	<code>\[CurlyEpsilon]</code>	<code>:ce;, :cepsilon;</code>	Z	<code>\[CapitalZeta]</code>	<code>:Z;, :Zeta;</code>
ζ	<code>\[Zeta]</code>	<code>:z;, :zeta;</code>	H	<code>\[CapitalEta]</code>	<code>:H;, :Et;, :Eta;</code>
η	<code>\[Eta]</code>	<code>:h;, :et;, :eta;</code>	Θ	<code>\[CapitalTheta]</code>	<code>:O;, :Th;, :Theta;</code>
θ	<code>\[Theta]</code>	<code>:q;, :th;, :theta;</code>	I	<code>\[CapitalIota]</code>	<code>:I;, :Iota;</code>
ϑ	<code>\[CurlyTheta]</code>	<code>:cq;, :cth;, :ctheta;</code>	K	<code>\[CapitalKappa]</code>	<code>:K;, :Kappa;</code>
ι	<code>\[Iota]</code>	<code>:i;, :iota;</code>	Λ	<code>\[CapitalLambda]</code>	<code>:L;, :Lambda;</code>
κ	<code>\[Kappa]</code>	<code>:k;, :kappa;</code>	M	<code>\[CapitalMu]</code>	<code>:M;, :Mu;</code>
χ	<code>\[CurlyKappa]</code>	<code>:ck;, :ckappa;</code>	N	<code>\[CapitalNu]</code>	<code>:N;, :Nu;</code>
λ	<code>\[Lambda]</code>	<code>:l;, :lambda;</code>	Ξ	<code>\[CapitalXi]</code>	<code>:X;, :Xi;</code>
μ	<code>\[Mu]</code>	<code>:m;, :mu;</code>	O	<code>\[CapitalOmicron]</code>	<code>:Om;, :Omicron;</code>
ν	<code>\[Nu]</code>	<code>:n;, :nu;</code>	Π	<code>\[CapitalPi]</code>	<code>:P;, :Pi;</code>
ξ	<code>\[Xi]</code>	<code>:x;, :xi;</code>	P	<code>\[CapitalRho]</code>	<code>:R;, :Rho;</code>
\omicron	<code>\[Omicron]</code>	<code>:om;, :omicron;</code>	Σ	<code>\[CapitalSigma]</code>	<code>:S;, :Sigma;</code>
π	<code>\[Pi]</code>	<code>:p;, :pi;</code>	T	<code>\[CapitalTau]</code>	<code>:T;, :Tau;</code>
ϖ	<code>\[CurlyPi]</code>	<code>:cp;, :cpi;</code>	Υ	<code>\[CapitalUpsilon]</code>	<code>:U;, :Upsilon;</code>
ρ	<code>\[Rho]</code>	<code>:r;, :rho;</code>	Υ	<code>\[CurlyCapitalUpsilon]</code>	<code>:cu;, :cUpsilon;</code>
ϱ	<code>\[CurlyRho]</code>	<code>:cr;, :crho;</code>			
σ	<code>\[Sigma]</code>	<code>:s;, :sigma;</code>			
ς	<code>\[FinalSigma]</code>	<code>:fs;</code>			
τ	<code>\[Tau]</code>	<code>:t;, :tau;</code>			
υ	<code>\[Upsilon]</code>	<code>:u;, :upsilon;</code>			

<i>form</i>	<i>full name</i>	<i>aliases</i>	<i>form</i>	<i>full name</i>	<i>aliases</i>
ϕ	<code>\[Phi]</code>	<code>:f;, :ph;, :phi;</code>	Υ	<code>\[CurlyCapitalUpsilon]</code>	<code>:cU;, :cUpsilon;</code>
φ	<code>\[CurlyPhi]</code>	<code>:j;, :cph;, :cphi;</code>	Φ	<code>\[CapitalPhi]</code>	<code>:F;, :Ph;, :Phi;</code>
χ	<code>\[Chi]</code>	<code>:c;, :ch;, :chi;</code>	X	<code>\[CapitalChi]</code>	<code>:C;, :Ch;, :Chi;</code>
ψ	<code>\[Psi]</code>	<code>:y;, :ps;, :psi;</code>	Ψ	<code>\[CapitalPsi]</code>	<code>:Y;, :Ps;, :Psi;</code>
ω	<code>\[Omega]</code>	<code>:o;, :w;, :omega;</code>	Ω	<code>\[CapitalOmega]</code>	<code>:O;, :W;, :Omega;</code>
\digamma	<code>\[Digamma]</code>	<code>:di;, :digamma;</code>	F	<code>\[CapitalDigamma]</code>	<code>:Di;, :Digamma;</code>
κ	<code>\[Koppa]</code>	<code>:ko;, :koppa;</code>	\mathbb{Q}	<code>\[CapitalKoppa]</code>	<code>:Ko;, :Koppa;</code>
ς	<code>\[Stigma]</code>	<code>:sti;, :stigma;</code>	\mathbb{S}	<code>\[CapitalStigma]</code>	<code>:Sti;, :Stigma;</code>
\simeq	<code>\[Sampi]</code>	<code>:sa;, :sampi;</code>	\mathbb{D}	<code>\[CapitalSampi]</code>	<code>:Sa;, :Sampi;</code>

The complete collection of Greek letters in *Mathematica*.

You can use Greek letters as the names of symbols. The only Greek letter with a built-in meaning in `StandardForm` is π , which *Mathematica* takes to stand for the symbol `Pi`.

Note that even though π on its own is assigned a built-in meaning, combinations such as π^2 or $x\pi$ have no built-in meanings.

The Greek letters Σ and Π look very much like the operators for sum and product. But as discussed above, these operators are different characters, entered as `\[Sum]` and `\[Product]` respectively.

Similarly, ϵ is different from the \in operator `\[Element]`, and μ is different from μ or `\[Micro]`.

Some capital Greek letters such as `\[CapitalAlpha]` look essentially the same as capital English letters. *Mathematica* however treats them as different characters, and in `TraditionalForm` it uses `\[CapitalBeta]`, for example, to denote the built-in function `Beta`.

Following common convention, lower-case Greek letters are rendered slightly slanted in the standard fonts provided with *Mathematica*, while capital Greek letters are unslanted. On Greek systems, however, *Mathematica* will render all Greek letters unslanted so that standard Greek fonts can be used.

Almost all Greek letters that do not look similar to English letters are widely used in science and mathematics. The *capital xi* Ξ is rare, though it is used to denote the cascade hyperon particles, the grand canonical partition function and regular language complexity. The *capital epsilon* ε is also rare, though it is used to denote $b\bar{b}$ particles, as well as the vernal equinox.

Curly Greek letters are often assumed to have different meanings from their ordinary counterparts. Indeed, in pure mathematics a single formula can sometimes contain both curly and ordinary forms of a particular letter. The curly pi ϖ is rare, except in astronomy.

The *final sigma* ς is used for sigmas that appear at the ends of words in written Greek; it is not commonly used in technical notation.

The *digamma* \digamma , *koppa* κ , *stigma* σ and *sampi* \wp are archaic Greek letters. These letters provide a convenient extension to the usual set of Greek letters. They are sometimes needed in making correspondences with English letters. The digamma corresponds to an English w, and koppa to an English q. Digamma is occasionally used to denote the digamma function $\text{PolyGamma}[x]$.

Variants of English Letters

form	full name	alias	form	full name	alias
ℓ	$\text{\textbackslash [ScriptL]}$	<code>:scL:</code>	\mathbb{C}	$\text{\textbackslash [DoubleStruckCapitalC]}$	<code>:dsC:</code>
\mathcal{E}	$\text{\textbackslash [ScriptCapitalE]}$	<code>:scE:</code>	\mathbb{R}	$\text{\textbackslash [DoubleStruckCapitalR]}$	<code>:dsR:</code>
\mathcal{H}	$\text{\textbackslash [ScriptCapitalH]}$	<code>:scH:</code>	\mathbb{Q}	$\text{\textbackslash [DoubleStruckCapitalQ]}$	<code>:dsQ:</code>
\mathcal{L}	$\text{\textbackslash [ScriptCapitalL]}$	<code>:scL:</code>	\mathbb{Z}	$\text{\textbackslash [DoubleStruckCapitalZ]}$	<code>:dsZ:</code>
\mathcal{C}	$\text{\textbackslash [GothicCapitalC]}$	<code>:goC:</code>	\mathbb{N}	$\text{\textbackslash [DoubleStruckCapitalN]}$	<code>:dsN:</code>
\mathcal{H}	$\text{\textbackslash [GothicCapitalH]}$	<code>:goH:</code>	\mathfrak{I}	$\text{\textbackslash [DotlessI]}$	
\mathcal{I}	$\text{\textbackslash [GothicCapitalI]}$	<code>:goI:</code>	\mathfrak{J}	$\text{\textbackslash [DotlessJ]}$	
\mathcal{R}	$\text{\textbackslash [GothicCapitalR]}$	<code>:goR:</code>	\wp	$\text{\textbackslash [WeierstrassP]}$	<code>:wp:</code>

Some commonly used variants of English letters.

By using menu items in the notebook front end, you can make changes in the font and style of ordinary text. However, such changes are usually discarded whenever you send input to the *Mathematica* kernel.

Script, gothic and double-struck characters are, however, treated as fundamentally different from their ordinary forms. This means that even though a c that is italic or a different size will be considered equivalent to an ordinary c when fed to the kernel, a double-struck C will not.

Different styles and sizes of C are treated as the same by the kernel. But gothic and double-struck characters are treated as different.

```
In[9]:= C + c + C + c + c
Out[9]= 3 C + c + c
```

In standard mathematical notation, capital script and gothic letters are sometimes used interchangeably. The double-struck letters, sometimes called blackboard or openface letters, are conventionally used to denote specific sets. Thus, for example, \mathbb{C} conventionally denotes the set of complex numbers, and \mathbb{Z} the set of integers.

Dotless i and j are not usually taken to be different in meaning from ordinary i and j ; they are simply used when overscripts are being placed on the ordinary characters.

`\[WeierstrassP]` is a notation specifically used for the Weierstrass P function `WeierstrassP`.

full names	aliases	
<code>\[ScriptA] - \[ScriptZ]</code>	<code>:sca: - :scz:</code>	lowercase script letters
<code>\[ScriptCapitalA] - \[ScriptCapitalZ]</code>	<code>:scA: - :scZ:</code>	uppercase script letters
<code>\[GothicA] - \[GothicZ]</code>	<code>:goa: - :goz:</code>	lowercase gothic letters
<code>\[GothicCapitalA] - \[GothicCapitalZ]</code>	<code>:goA: - :goZ:</code>	uppercase gothic letters
<code>\[DoubleStruckA] - \[DoubleStruckZ]</code>	<code>:dsa: - :dsz:</code>	lowercase double-struck letters
<code>\[DoubleStruckCapitalA] - \[DoubleStruckCapitalZ]</code>	<code>:dsA: - :dsZ:</code>	uppercase double-struck letters
<code>\[FormalA] - \[FormalZ]</code>	<code>:\$a: - :\$z:</code>	lowercase formal letters
<code>\[FormalCapitalA] - \[FormalCapitalZ]</code>	<code>:\$A: - :\$Z:</code>	uppercase formal letters

Complete alphabets of variant English letters.

Formal Symbols

Symbols represented by formal letters, or formal symbols, appear in the output of certain functions. They are indicated by gray dots above and below the English letter.

DifferentialRoot automatically chooses the names for the function arguments.

`In[83]:= root = DifferentialRootReduce[Cos]`

`Out[83]= DifferentialRoot[Function[{y, x}, {y[x] + y'[x] == 0, y[0] == 1, y'[0] == 0}]]`

Formal symbols are Protected, so they cannot be accidentally assigned a value.

Trying to modify a formal symbol fails.

In[2]:= $\dot{y} = 0$

Set::wrsym : Symbol \dot{y} is Protected. >>

Out[2]= 0

In[3]:= \dot{y}

Out[3]= \dot{y}

This means that expressions depending on formal symbols will not be accidentally modified.

In[4]:= $\text{root}[[1, 2]]$

Out[4]= $\{\dot{y}[\dot{x}] + \dot{y}''[\dot{x}] = 0, \dot{y}[0] = 1, \dot{y}'[0] = 0\}$

Specific values for formal symbols can be substituted using replacement rules.

Verify that the defining equations hold for cosine.

In[5]:= $\text{root}[[1, 2]] /. \dot{y} \rightarrow \cos$

Out[5]= {True, True, True}

Formal symbols can be temporarily modified inside of a Block because Block clears all definitions associated with a symbol, including Attributes. Table works essentially like Block, thus also allowing temporary changes.

Assign a temporary value to \dot{y} :

In[6]:= $\text{Block}[\{\dot{y} = \cos\}, \text{root}[[1, 2]]]$

Out[6]= {True, True, True}

In most situations modifying formal symbols is not necessary. Since in DifferentialRoot formal symbols are used as names for the formal parameters of a function, the function should simply be evaluated for the actual values of arguments.

Evaluating the function substitutes x for \dot{x} and y for \dot{y} .

In[7]:= $\text{root}[[1]][y, x]$

Out[7]= $\{y[x] + y''[x] = 0, y[0] = 1, y'[0] = 0\}$

It is possible to define custom typesetting rules for formal symbols.

Use coloring to highlight formal symbols.

```
In[84]:= MakeBoxes[\[FormalX], ___] := TagBox["x", \[FormalX] &, AutoDelete -> True,
  BaseStyle -> {FontColor -> RGBColor[.6, .4, .2], ShowSyntaxStyles -> False}]
MakeBoxes[\[FormalY], ___] := TagBox["y", \[FormalY] &, AutoDelete -> True,
  BaseStyle -> {FontColor -> RGBColor[.6, .4, .2], ShowSyntaxStyles -> False}]
```

```
In[86]:= root
```

```
Out[86]= DifferentialRoot[Function[{y, x}, {y[x] + y''[x] == 0, y[0] == 1, y'[0] == 0}]]
```

The formatting rules were attached to `MakeBoxes`. Restore the original formatting:

```
In[87]:= FormatValues@MakeBoxes = {};
```

```
In[88]:= root
```

```
Out[88]= DifferentialRoot[Function[{y, x}, {y[x] + y''[x] == 0, y[0] == 1, y'[0] == 0}]]
```

Hebrew Letters

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>
\[Aleph]	\[Aleph]	:al:	\[Gimel]	\[Gimel]
\[Bet]	\[Bet]		\[Dalet]	\[Dalet]

Hebrew characters.

Hebrew characters are used in mathematics in the theory of transfinite sets; \aleph_0 is for example used to denote the total number of integers.

Units and Letter-Like Mathematical Symbols

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
μ	\[Micro]	:mi:	\circ	\[Degree]	:deg:
\\O	\[Mho]	:mho:	\emptyset	\[EmptySet]	:es:
\\AA	\[Angstrom]	:Ang:	∞	\[Infinity]	:inf:
\\hbar	\[HBar]	:hb:	e	\[ExponentialE]	:ee:
\\cent	\[Cent]	:cent:	i	\[ImaginaryI]	:ii:
\\sterling	\[Sterling]		j	\[ImaginaryJ]	:jj:
\\euro	\[Euro]	:euro:	π	\[DoubledPi]	:pp:
\\yen	\[Yen]		γ	\[DoubledGamma]	:gg:

Units and letter-like mathematical symbols.

Mathematica treats \circ or \[Degree] as the symbol `Degree`, so that, for example, 30° is equivalent to 30 `Degree`.

Note that μ , \\AA and \emptyset are all distinct from the ordinary letters μ (\[Mu]), \\AA (\[CapitalARing]) and \emptyset (\[CapitalOSlash]).

Mathematica interprets ∞ as `Infinity`, e as `E`, and both i and j as `I`. The characters `e`, `i` and `j` are provided as alternatives to the usual uppercase letters `E` and `I`.

π and γ are not by default assigned meanings in `StandardForm`. You can therefore use π to represent a pi that will not automatically be treated as `Pi`. In `TraditionalForm` γ is interpreted as `EulerGamma`.

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
∂	\[PartialD]	:pd:	\sum	\[Sum]	:sum:
d	\[DifferentialD]	:dd:	\prod	\[Product]	:prod:
\\D	\[CapitalDifferentialD]	:DD:	T	\[Transpose]	:tr:
∇	\[Del]	:del:	H	\[HermitianConjugate]	:hc:
Δ	\[DifferenceDelta]	:diffd:	\bigtriangleup	\[DiscreteShift]	:shift:
			\bigcirc	\[DiscreteRatio]	:dratio:

Operators that look like letters.

∇ is an operator while \hbar , \circ and ¥ are ordinary symbols.

```
In[1]:= {\nabla f, \hbar^2, 45^\circ, 5000 \text{\textyen}} // FullForm
Out[1]//FullForm= List[Del[f], Power[\text{HBar}, 2], Times[45, Degree], Times[5000, \text{Yen}]]
```

Shapes, Icons and Geometrical Constructs

form	full name	alias	form	full name	alias
■	\[FilledVerySmallSquare]	:fvssq:	○	\[EmptySmallCircle]	:esci:
□	\[EmptySmallSquare]	:essq:	●	\[FilledSmallCircle]	:fsci:
■	\[FilledSmallSquare]	:fssq:	○	\[EmptyCircle]	:eci:
□	\[EmptySquare]	:esq:	●	\[GrayCircle]	:gci:
■	\[GraySquare]	:gsq:	●	\[FilledCircle]	:fci:
■	\[FilledSquare]	:fsq:	△	\[EmptyUpTriangle]	
□	\[DottedSquare]		▲	\[FilledUpTriangle]	
□	\[EmptyRectangle]		▽	\[EmptyDownTriangle]	
■	\[FilledRectangle]		▼	\[FilledDownTriangle]	
◇	\[EmptyDiamond]		★	\[FivePointedStar]	:*5:
◆	\[FilledDiamond]		★	\[SixPointedStar]	:*6:

Shapes.

Shapes are most often used as “dingbats” to emphasize pieces of text. But *Mathematica* treats them as letter-like forms, and also allows them to appear in the names of symbols.

In addition to shapes such as `\[EmptySquare]`, there are characters such as `\[Square]` which are treated by *Mathematica* as operators rather than letter-like forms.

form	full name	alias	form	full name	aliases
✿	\[MathematicaIcon]	:math:	☺	\[HappySmiley]	::) :, ::-) :
🌐	\[KernelIcon]		☹	\[NeutralSmiley]	::- :
💡	\[LightBulb]		☹	\[SadSmiley]	::-(:
⚠	\[WarningSign]		Ѡ	\[FreakedSmiley]	::-@:
🐺	\[WatchIcon]		🐺	\[Wolf]	:wf:, :wolf:

Icons.

You can use icon characters just like any other letter-like forms.

```
In[1]:= Expand[(\theta + \mathbb{A})^4]
Out[1]= \theta^4 + 4 \theta^3 \mathbb{A} + 6 \theta^2 \mathbb{A}^2 + 4 \theta \mathbb{A}^3 + \mathbb{A}^4
```

<i>form</i>	<i>full name</i>	<i>form</i>	<i>full name</i>
\angle	<code>\[Angle]</code>	\triangleleft	<code>\[SphericalAngle]</code>
\llcorner	<code>\[RightAngle]</code>	\triangleup	<code>\[EmptyUpTriangle]</code>
\measuredangle	<code>\[MeasuredAngle]</code>	\varnothing	<code>\[Diameter]</code>

Notation for geometrical constructs.

Since *Mathematica* treats characters like \angle as letter-like forms, constructs like $\angle BC$ are treated in *Mathematica* as single symbols.

Textual Elements

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
$-$	<code>\[Dash]</code>	<code>:---</code>	$'$	<code>\[Prime]</code>	<code>:'</code>
$--$	<code>\[LongDash]</code>	<code>:---</code>	$''$	<code>\[DoublePrime]</code>	<code>:''</code>
\bullet	<code>\[Bullet]</code>	<code>:bu:</code>	$`$	<code>\[ReversePrime]</code>	<code>:`</code>
\P	<code>\[Paragraph]</code>		$``$	<code>\[ReverseDoublePrime]</code>	<code>:``</code>
$\$$	<code>\[Section]</code>		$``$	<code>\[LeftGuillemet]</code>	<code>:g<<</code>
?	<code>\[DownQuestion]</code>	<code>:d?:</code>	$``$	<code>\[RightGuillemet]</code>	<code>:g>></code>
!	<code>\[DownExclamation]</code>	<code>:d!:</code>	\dots	<code>\[Ellipsis]</code>	<code>:....</code>

Characters used for punctuation and annotation.

<i>form</i>	<i>full name</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
\circledC	<code>\[Copyright]</code>	\dagger	<code>\[Dagger]</code>	<code>:dg:</code>
\circledR	<code>\[RegisteredTrademark]</code>	\ddagger	<code>\[DoubleDagger]</code>	<code>:ddg:</code>
TM	<code>\[Trademark]</code>	\clubsuit	<code>\[ClubSuit]</code>	
\flat	<code>\[Flat]</code>	\diamond	<code>\[DiamondSuit]</code>	
\natural	<code>\[Natural]</code>	\heartsuit	<code>\[HeartSuit]</code>	
\sharp	<code>\[Sharp]</code>	\spadesuit	<code>\[SpadeSuit]</code>	

Other characters used in text.

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
—	\[HorizontalLine]	:hline:	—	\[UnderParenthesis]	:u(:
	\[VerticalLine]	:vline:	—	\[OverParenthesis]	:o(:
...	\[Ellipsis]	:...:	—	\[UnderBracket]	:u{:
...	\[CenterEllipsis]		—	\[OverBracket]	:o{:
:	\[VerticalEllipsis]		—	\[UnderBrace]	:u{:
.	\[AscendingEllipsis]		—	\[OverBrace]	:o{:
,	\[DescendingEllipsis]				

Characters used in building sequences and arrays.

The under and over braces grow to enclose the whole expression.

In[1]:= `Underoverscript[Expand[(1 + x)^4], ~, ~]`

Out[1]=
$$\overbrace{\underbrace{1 + 4x + 6x^2 + 4x^3 + x^4}_{\sim}}$$

Extended Latin Letters

Mathematica supports all the characters commonly used in Western European languages based on Latin scripts.

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
à	\[Agrave]	:a`:	À	\[CapitalAGrave]	:A`:
á	\[AAcute]	:a':	Á	\[CapitalAAcute]	:A':
â	\[AHat]	:a^:	Â	\[CapitalAHat]	:A^:
ã	\[ATilde]	:a~:	Ã	\[CapitalATilde]	:A~:
ä	\[ADoubleDot]	:a":	Ä	\[CapitalADoubleDot]	:A":
å	\[ARing]	:ao:	Å	\[CapitalARing]	:Ao:
ã	\[ABar]	:a-:	Ã	\[CapitalABar]	:A-:
ă	\[ACup]	:au:	Ă	\[CapitalACup]	:Au:
æ	\[AE]	:ae:	Æ	\[CapitalAE]	:AE:
ć	\[CACute]	:c':	Ć	\[CapitalCACute]	:C':
ç	\[CCedilla]	:c,::	Ç	\[CapitalCCedilla]	:C,::
č	\[CHacek]	:cv:	Č	\[CapitalCHacek]	:Cv:
è	\[Egrave]	:e`:	È	\[CapitalEgrave]	:E`:
é	\[Eacute]	:e':	É	\[CapitalEacute]	:E':
ë	\[Ebar]	:e-:	Ë	\[CapitalEbar]	:E-:
ê	\[EHat]	:e^:	Ê	\[CapitalEHat]	:E^:
ë	\[EDoubleDot]	:e":	Ë	\[CapitalEDoubleDot]	:E":

ě	\[ECup]	:eu:	Ě	\[CapitalECup]	:Eu:
ѝ	\[IGrave]	:i`:	Ѝ	\[CapitalIGrave]	:I`:
ѝ	\[IAcute]	:i':	Ѝ	\[CapitalIAcute]	:I':
ѝ	\[IHat]	:i^:	Ѝ	\[CapitalIHat]	:I^:
ѝ	\[IDoubleDot]	:i":	Ѝ	\[CapitalIDoubleDot]	:I":
Ѝ	\[ICup]	:iu:	Ѝ	\[CapitalICup]	:Iu:
ð	\[Eth]	:d-:	ð	\[CapitalEth]	:D-:
ȝ	\[LSlash]	:l/:	ȝ	\[CapitalLSlash]	:L/:
ñ	\[NTilde]	:n~:	Ñ	\[CapitalNTilde]	:N~:
ð	\[OGrave]	:o`:	ð	\[CapitalOGrave]	:O`:
ó	\[OAcute]	:o':	ó	\[CapitalOAcute]	:O':
ô	\[OHat]	:o^:	ô	\[CapitalOHat]	:O^:
õ	\[OTilde]	:o~:	õ	\[CapitalOTilde]	:O~:
ö	\[ODoubleDot]	:o":	ö	\[CapitalODoubleDot]	:O":
ó	\[ODoubleAcute]	:o'':	ó	\[CapitalODoubleAcute]	:O'':
ø	\[OSlash]	:o/:	ø	\[CapitalOSlash]	:O/:
œ	\[OE]	:oe:	œ	\[CapitalOE]	:OE:
š	\[SHacek]	:sv:	š	\[CapitalSHacek]	:Sv:
ù	\[UGrave]	:u`:	Ù	\[CapitalUGrave]	:U`:
ú	\[UAcute]	:u':	Ú	\[CapitalUAcute]	:U':
û	\[UHat]	:u^:	Û	\[CapitalUHat]	:U^:
ü	\[UDoubleDot]	:u":	Ü	\[CapitalUDoubleDot]	:U":
ú	\[UDoubleAcute]	:u'':	ӹ	\[CapitalUDoubleAcute]	:U'':
ÿ	\[YAcute]	:y':	Ӻ	\[CapitalYAcute]	:Y':
þ	\[Thorn]	:thn:	þ	\[CapitalThorn]	:Thn:
ß	\[SZ]	:sz:, :ss:			

Variants of English letters.

Most of the characters shown are formed by adding diacritical marks to ordinary English letters. Exceptions include \[SZ] ß, used in German, and \[Thorn] þ and \[Eth] ð, used primarily in Old English.

You can make additional characters by explicitly adding diacritical marks yourself.

`char Ctrl+& mark Ctrl+Space` add a mark above a character

`char Ctrl++ mark Ctrl+Space` add a mark below a character

Adding marks above and below characters.

<i>form</i>	<i>alias</i>	<i>full name</i>
'	(keyboard character)	\[RawQuote]
'	:':	\[Prime]
'	(keyboard character)	\[RawBackquote]
'	:`:	\[ReversePrime]
.	(keyboard characters)	
^	(keyboard character)	\[RawWedge]
o	:esci:	\[EmptySmallCircle]
.	(keyboard character)	\[RawDot]
~	(keyboard character)	\[RawTilde]
_	(keyboard character)	\[RawUnderscore]
ˇ	:hc:	\[Hacek]
ˇ	:bv:	\[Breve]
ˇ	:dbv:	\[DownBreve]
”	:`':	\[DoublePrime]
,	:cd:	\[Cedilla]

Diacritical marks to add to characters.

Operators

Basic Mathematical Operators

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
×	\[Times]	:*:	×	\[Cross]	:cross:
÷	\[Divide]	:div:	±	\[PlusMinus]	:+-:
√	\[Sqrt]	:sqrt:	⋮	\[MinusPlus]	:--+:

Some operators used in basic arithmetic and algebra.

Note that the \times for \[Cross] is distinguished by being drawn slightly smaller than the \times for \[Times].

$x \times y$	Times [x,y]	multiplication
$x \div y$	Divide [x,y]	division
\sqrt{x}	Sqrt [x]	square root
$x \times y$	Cross [x,y]	vector cross product
$\pm x$	PlusMinus [x]	(no built-in meaning)
$x \pm y$	PlusMinus [x,y]	(no built-in meaning)
$\mp x$	MinusPlus [x]	(no built-in meaning)
$x \mp y$	MinusPlus [x,y]	(no built-in meaning)

Interpretation of some operators in basic arithmetic and algebra.

Operators in Calculus and Algebra

form	full name	alias	form	full name	alias
∇	\[Del]	:del:	\int	\[Integral]	:int:
∂	\[PartialD]	:pd:	\oint	\[ContourIntegral]	:cint:
d	\[DifferentialD]	:dd:	$\oint\oint$	\[DoubleContourIntegral]	
\sum	\[Sum]	:sum:	$\oint\oint\oint$	\[CounterClockwiseContourIntegral]	:ccint:
\prod	\[Product]	:prod:	$\oint\oint\oint\oint$	\[ClockwiseContourIntegral]	:ccint:

Operators used in calculus.

form	full name	aliases	form	full name	alias
$*$	\[Conjugate]	:co:, :conj:	†	\[ConjugateTranspose]	:ct:
$^\top$	\[Transpose]	:tr:	$^\text{H}$	\[HermitianConjugate]	:hc:

Operators for complex numbers and matrices.

Logical and Other Connectives

form	full name	aliases	form	full name	alias
\wedge	\[And]	:&&;, :and:	\Rightarrow	\[Implies]	:>:
\vee	\[Or]	: ;, :or:	\therefore	\[RoundImplies]	
\neg	\[Not]	:!, :not:	\because	\[Therefore]	:tf:
\in	\[Element]	:el:	\vdash	\[Because]	
\forall	\[ForAll]	:fa:	\dashv	\[RightTee]	
\exists	\[Exists]	:ex:	\dashv	\[LeftTee]	
\nexists	\[NotExists]	:!ex:	\models	\[DoubleRightTee]	
\oplus	\[Xor]	:xor:	\ni	\[DoubleLeftTee]	
$\overline{\wedge}$	\[Nand]	:nand:	\ni	\[SuchThat]	:st:
$\overline{\vee}$	\[Nor]	:nor:	$ $	\[VerticalSeparator]	: :
			:	\[Colon]	::

Operators used as logical connectives.

The operators \wedge , \vee and \neg are interpreted as corresponding to the built-in functions `And`, `Or` and `Not`, and are equivalent to the keyboard operators `&&`, `||` and `!`. The operators \forall , \exists and \nexists correspond to the built-in functions `Xor`, `Nand` and `Nor`. Note that \neg is a prefix operator.

$x \Rightarrow y$ and $x \ni y$ are both taken to give the built-in function `Implies[x, y]`. $x \in y$ gives the built-in function `Element[x, y]`.

This is interpreted using the built-in functions `And` and `Implies`.

In[1]:= $3 < 4 \wedge x > 5 \Rightarrow y < 7$

Out[1]= `Implies[x > 5, y < 7]`

Mathematica supports most of the standard syntax used in mathematical logic. In *Mathematica*, however, the variables that appear in the quantifiers \forall , \exists and \nexists must appear as subscripts. If they appeared directly after the quantifier symbols then there could be a conflict with multiplication operations.

\forall and \exists are essentially prefix operators like ∂ .

In[2]:= $\forall_x \exists_y \phi[x, y] // \text{FullForm}$

Out[2]//FullForm= `ForAll[x, Exists[y, \[Phi][x,y]]]`

Operators Used to Represent Actions

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
◦	\[SmallCircle]	:sc:	^	\[Wedge]	:^:
⊕	\[CirclePlus]	:c+:	∨	\[Vee]	:v:
⊖	\[CircleMinus]	:c-:	⊸	\[Union]	:un:
⊗	\[CircleTimes]	:c*:	⊹	\[UnionPlus]	
⊙	\[CircleDot]	:c.:	⊺	\[Intersection]	:inter:
◊	\[Diamond]	:dia:	⊻	\[SquareIntersection]	
·	\[CenterDot]	:..:	⊼	\[SquareUnion]	
*	\[Star]	:star:	⊽	\[Coproduct]	:coprod:
⋮	\[VerticalTilde]		⊿	\[Cap]	
＼	\[Backslash]	:\\:	⊾	\[Cup]	
			□	\[Square]	:sq:

Operators typically used to represent actions. All the operators except \[Square] are infix.

Following *Mathematica*'s usual convention, all the operators in the table are interpreted to give functions whose names are exactly the names of the characters that appear in the operators.

The operators are interpreted as functions with corresponding names.

```
In[3]:= x ⊕ y ~ z // FullForm
Out[3]//FullForm= CirclePlus[x, Cap[y, z]]
```

All the operators in the table above, except for □, are infix, so that they must appear in between their operands.

Bracketing Operators

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
	\[LeftFloor]	:lf:	(\[LeftAngleBracket]	:<:
	\[RightFloor]	:rf:)	\[RightAngleBracket]	:>:
[\[LeftCeiling]	:lc:		\[LeftBracketingBar]	:l :
]	\[RightCeiling]	:rc:		\[RightBracketingBar]	:r :
[\[LeftDoubleBracket]	:[[:		\[LeftDoubleBracketingBar]	:l :
]	\[RightDoubleBracket]	:]]:		\[RightDoubleBracketingBar]	:r :

Characters used as bracketing operators.

\lfloor x \rfloor	Floor[x]
\lceil x \rceil	Ceiling[x]
m[[i,j,...]]	Part[m,i,j,...]
\langle x,y,...\rangle	AngleBracket[x,y,...]
\{x,y,...\}	BracketingBar[x,y,...]
\ x,y,...\	DoubleBracketingBar[x,y,...]

Interpretations of bracketing operators.

Operators Used to Represent Relations

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
==	\[Equal]	:==:	≠	\[NotEqual]	:!=:
=	\[LongEqual]	:l=:	≢	\[NotCongruent]	:!====:
≡	\[Congruent]	:=====:	≵	\[NotTilde]	:!~:
~	\[Tilde]	:~:	≶	\[NotTildeTilde]	:!~~:
≈	\[TildeTilde]	:~~~:	≷	\[NotTildeEqual]	:!~=:
≃	\[TildeEqual]	:~=:	≸	\[NotTildeFullEqual]	:!~==:
⊑	\[TildeFullEqual]	:~==:	≹	\[NotEqualTilde]	:!=~:
≔	\[EqualTilde]	:=~:	≻	\[NotHumpEqual]	:!h=:
≕	\[HumpEqual]	:h=:	≺	\[NotHumpDownHump]	
≖	\[HumpDownHump]		≻	\[NotCupCap]	
≘	\[CupCap]		≼	\[Proportional]	:prop:
≙	\[DotEqual]	::	≽	\[Proportion]	

Operators usually used to represent similarity or equivalence.

The special character `==` (or `\[Equal]`) is an alternative input form for `==`. `#` is used both for input and output.

In[4]:= {`a == b`, `a = b`, `a != b`, `a # b`}

Out[4]= {`a == b`, `a == b`, `a != b`, `a # b`}

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
<code>≥</code>	<code>\[GreaterEqual]</code>	<code>:≥=:</code>	<code>≤</code>	<code>\[NotGreaterEqual]</code>	<code>:!≥=:</code>
<code>≤</code>	<code>\[LessEqual]</code>	<code>:≤=:</code>	<code>≥</code>	<code>\[NotLessEqual]</code>	<code>:!≤=:</code>
<code>≥</code>	<code>\[GreaterSlantEqual]</code>	<code>:≥/:</code>	<code>≤</code>	<code>\[NotGreaterSlantEqual]</code>	<code>:!≥/:</code>
<code>≤</code>	<code>\[LessSlantEqual]</code>	<code>:≤/:</code>	<code>≥</code>	<code>\[NotLessSlantEqual]</code>	<code>:!≤/:</code>
<code>≡</code>	<code>\[GreaterFullEqual]</code>	<code>≡</code>	<code>≢</code>	<code>\[NotGreaterFullEqual]</code>	
<code>≡</code>	<code>\[LessFullEqual]</code>	<code>≡</code>	<code>≢</code>	<code>\[NotLessFullEqual]</code>	
<code>≥</code>	<code>\[GreaterTilde]</code>	<code>:≥~:</code>	<code>≤</code>	<code>\[NotGreaterTilde]</code>	<code>:! >~:</code>
<code>≤</code>	<code>\[LessTilde]</code>	<code>:≤~:</code>	<code>≥</code>	<code>\[NotLessTilde]</code>	<code>:! <~:</code>
<code>≥</code>	<code>\[GreaterGreater]</code>	<code>≥</code>	<code>≤</code>	<code>\[NotGreaterGreater]</code>	
<code>≤</code>	<code>\[LessLess]</code>	<code>≤</code>	<code>≥</code>	<code>\[NotLessLess]</code>	
<code>≥</code>	<code>\[NestedGreaterGreater]</code>	<code>≥</code>	<code>≤</code>	<code>\[NotNestedGreaterGreater]</code>	
<code>≤</code>	<code>\[NestedLessLess]</code>	<code>≤</code>	<code>≥</code>	<code>\[NotNestedLessLess]</code>	
<code>≥</code>	<code>\[GreaterLess]</code>	<code>≥</code>	<code>≤</code>	<code>\[NotGreaterLess]</code>	
<code>≤</code>	<code>\[LessGreater]</code>	<code>≤</code>	<code>≥</code>	<code>\[NotLessGreater]</code>	
<code>≥</code>	<code>\[GreaterEqualLess]</code>	<code>≥</code>	<code>≤</code>	<code>\[NotGreater]</code>	<code>:!≥:</code>
<code>≤</code>	<code>\[LessEqualGreater]</code>	<code>≤</code>	<code>≥</code>	<code>\[NotLess]</code>	<code>:!≤:</code>

Operators usually used for ordering by magnitude.

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
<code>⊂</code>	<code>\[Subset]</code>	<code>:sub:</code>	<code>⊄</code>	<code>\[NotSubset]</code>	<code>:!sub:</code>
<code>⊃</code>	<code>\[Superset]</code>	<code>:sup:</code>	<code>⊅</code>	<code>\[NotSuperset]</code>	<code>:!sup:</code>
<code>⊆</code>	<code>\[SubsetEqual]</code>	<code>:sub=:</code>	<code>⊄</code>	<code>\[NotSubsetEqual]</code>	<code>:!sub=:</code>
<code>⊇</code>	<code>\[SupersetEqual]</code>	<code>:sup=:</code>	<code>⊅</code>	<code>\[NotSupersetEqual]</code>	<code>:!sup=:</code>
<code>∈</code>	<code>\[Element]</code>	<code>:el:</code>	<code>∉</code>	<code>\[NotElement]</code>	<code>:!el:</code>
<code>∉</code>	<code>\[ReverseElement]</code>	<code>:mem:</code>	<code>∉</code>	<code>\[NotReverseElement]</code>	<code>:!mem:</code>

Operators used for relations in sets.

<i>form</i>	<i>full name</i>	<i>form</i>	<i>full name</i>
>	\[Succeeds]	*	\[NotSucceeds]
<	\[Precedes]	†	\[NotPrecedes]
≥	\[SucceedsEqual]	‡	\[NotSucceedsEqual]
≤	\[PrecedesEqual]	§	\[NotPrecedesTilde]
≥	\[SucceedsSlantEqual]	¶	\[NotSucceedsSlantEqual]
≤	\[PrecedesSlantEqual]	¤	\[NotPrecedesSlantEqual]
≈	\[SucceedsTilde]	≠	\[NotSucceedsTilde]
≈	\[PrecedesTilde]	≠	\[NotPrecedesEqual]
▷	\[RightTriangle]	▷	\[NotRightTriangle]
◁	\[LeftTriangle]	◁	\[NotLeftTriangle]
▷	\[RightTriangleEqual]	▷	\[NotRightTriangleEqual]
◁	\[LeftTriangleEqual]	◁	\[NotLeftTriangleEqual]
▷	\[RightTriangleBar]	▷	\[NotRightTriangleBar]
◁	\[LeftTriangleBar]	◁	\[NotLeftTriangleBar]
⊉	\[SquareSuperset]	⊉	\[NotSquareSuperset]
⊈	\[SquareSubset]	⊈	\[NotSquareSubset]
⊉	\[SquareSupersetEqual]	⊉	\[NotSquareSupersetEqual]
⊈	\[SquareSubsetEqual]	⊈	\[NotSquareSubsetEqual]

Operators usually used for other kinds of orderings.

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
	\[VerticalBar]	⋮⋮	✗	\[NotVerticalBar]	⋮!⋮
	\[DoubleVerticalBar]	⋮⋮ ⋮⋮	✗	\[NotDoubleVerticalBar]	⋮!⋮⋮

Relational operators based on vertical bars.

Operators Based on Arrows and Vectors

Operators based on arrows are often used in pure mathematics and elsewhere to represent various kinds of transformations or changes.

→ is equivalent to ->.

In[5]:= $x + y / . x \rightarrow 3$

Out[5]= $3 + y$

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>	<i>alias</i>
\rightarrow	<code>\[Rule]</code>	<code>:>:</code>	\Rightarrow	<code>\[Implies]</code>	<code>:>:</code>
\Rightarrow	<code>\[RuleDelayed]</code>	<code>::>:</code>	\Rightarrow	<code>\[RoundImplies]</code>	

Arrow-like operators with built-in meanings in *Mathematica*.

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>
\longrightarrow	<code>\[RightArrow]</code>	<code>:>:</code>	\uparrow	<code>\[UpArrow]</code>
\longleftarrow	<code>\[LeftArrow]</code>	<code>:<:</code>	\downarrow	<code>\[DownArrow]</code>
\longleftrightarrow	<code>\[LeftRightArrow]</code>	<code>:<->:</code>	\updownarrow	<code>\[UpDownArrow]</code>
\longrightarrow	<code>\[LongRightArrow]</code>	<code>:-->:</code>	\uparrow	<code>\[UpTeeArrow]</code>
\longleftarrow	<code>\[LongLeftArrow]</code>	<code>:<--:</code>	\downarrow	<code>\[DownTeeArrow]</code>
\longleftrightarrow	<code>\[LongLeftRightArrow]</code>	<code>:<->:</code>	\updownarrow	
\rightarrow	<code>\[ShortRightArrow]</code>		\uparrow	<code>\[UpArrowBar]</code>
\leftarrow	<code>\[ShortLeftArrow]</code>		\downarrow	<code>\[DownArrowBar]</code>
\longrightarrow	<code>\[RightTeeArrow]</code>		\upuparrows	<code>\[DoubleUpArrow]</code>
\longleftarrow	<code>\[LeftTeeArrow]</code>		\downdownarrows	<code>\[DoubleDownArrow]</code>
\longrightarrow	<code>\[RightArrowBar]</code>		\updownarrows	<code>\[DoubleUpDownArrow]</code>
\longleftarrow	<code>\[LeftArrowBar]</code>			
\Longrightarrow	<code>\[DoubleRightArrow]</code>	<code>:>:</code>	$\overrightarrow{}$	<code>\[RightArrowLeftArrow]</code>
\Longleftarrow	<code>\[DoubleLeftArrow]</code>	<code>:<:</code>	$\overleftarrow{}$	<code>\[LeftArrowRightArrow]</code>
\Longleftrightarrow	<code>\[DoubleLeftRightArrow]</code>	<code>:<->:</code>	$\overleftrightarrow{}$	<code>\[UpArrowDownArrow]</code>
\Longrightarrow	<code>\[DoubleLongRightArrow]</code>	<code>:==>:</code>	\overuparrow	<code>\[DownArrowUpArrow]</code>
\Longleftarrow	<code>\[DoubleLongLeftArrow]</code>	<code>:<==:</code>	\overdownarrow	<code>\[LowerRightArrow]</code>
\Longleftrightarrow	<code>\[DoubleLongLeftRightArrow]</code>	<code>:<==>:</code>	$\overleftarrow{}$	<code>\[LowerLeftArrow]</code>
			\nearrow	<code>\[UpperLeftArrow]</code>
			\searrow	<code>\[UpperRightArrow]</code>

Ordinary arrows.

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>
→	\[RightVector]		⋮vec⋮	↑ \[LeftUpVector]
↙	\[LeftVector]		↓	\[LeftDownVector]
↔	\[LeftRightVector]		↗	\[LeftUpDownVector]
→→	\[DownRightVector]		↖	\[RightUpVector]
→←	\[DownLeftVector]		↙	\[RightDownVector]
→→→	\[DownLeftRightVector]		↙↗	\[RightUpDownVector]
→→→→	\[RightTeeVector]		↖↖	\[LeftUpTeeVector]
→→→→→	\[LeftTeeVector]		↖↖↖	\[LeftDownTeeVector]
→→→→→→	\[DownRightTeeVector]		↖↖↖↖	\[RightUpTeeVector]
→→→→→→→	\[DownLeftTeeVector]		↖↖↖↖↖	\[RightDownTeeVector]
→→→→→→→→	\[RightVectorBar]		↑↑↑↑↑↑↑↑	\[LeftUpVectorBar]
→→→→→→→→→	\[LeftVectorBar]		↓↓↓↓↓↓↓↓	\[LeftDownVectorBar]
→→→→→→→→→→	\[DownRightVectorBar]		↖↖↖↖↖↖↖↖	\[RightUpVectorBar]
→→→→→→→→→→→	\[DownLeftVectorBar]		↙↙↙↙↙↙↙↙	\[RightDownVectorBar]
→→→→→→→→→→→→	\[Equilibrium]	⋮equi⋮	↑↑↑↑↑↑↑↑↑↑	\[UpEquilibrium]
→→→→→→→→→→→→→	\[ReverseEquilibrium]		↓↓↓↓↓↓↓↓↓↓	\[ReverseUpEquilibrium]

Vectors and related arrows.

All the arrow and vector-like operators in *Mathematica* are infix.

In[6]:= x ≠ y || z

Out[6]= $\mathbf{x} \geq \mathbf{y} \wedge \mathbf{z}$

<i>form</i>	<i>full name</i>	<i>alias</i>	<i>form</i>	<i>full name</i>
\vdash	<code>\[RightTee]</code>	<code>:rt;</code>	\vDash	<code>\[DoubleRightTee]</code>
\dashv	<code>\[LeftTee]</code>	<code>:lt;</code>	\vDash	<code>\[DoubleLeftTee]</code>
\vdash	<code>\[UpTee]</code>	<code>:ut;</code>		
\dashv	<code>\[DownTee]</code>	<code>:dt;</code>		

Tees.

Structural Elements and Keyboard Characters

<i>full name</i>	<i>alias</i>	<i>full name</i>	<i>alias</i>
\[InvisibleComma]	Esc , Esc	\[AlignmentMarker]	Esc am Esc
\[InvisibleApplication]	Esc @ Esc	\[NoBreak]	Esc nb Esc
\[InvisibleSpace]	Esc is Esc	\[Null]	Esc null Esc
\[ImplicitPlus]	Esc + Esc		

Invisible characters.

In the input there is an invisible comma between the 1 and 2.

In[1]:= m12

Out[1]= m1,2

Here there is an invisible space between the x and y, interpreted as multiplication.

In[2]:= FullForm[xy]

Out[2]//FullForm= Times[x, y]

\[Null] does not display, but can take modifications such as superscripts.

In[3]:= f[x, ^a]

Out[3]= f[x, ^a]

The \[AlignmentMarker] does not display, but shows how to line up the elements of the column.

In[4]:= Grid[{{"b + c + d"}, {"a + b + c"}}, Alignment -> ""] // DisplayForm

*Out[4]//DisplayForm=
$$\begin{array}{c} b + c + d \\ a + b + c \end{array}$$*

The \[ImplicitPlus] operator is used as a hidden plus sign in mixed fractions.

In[5]:= 1 $\frac{2}{3}$

Out[5]= $\frac{5}{3}$

full name	alias	full name	alias
\[VeryThinSpace]	Esc_ Esc	\[NegativeVeryThinSpace]	Esc-_ Esc
\[ThinSpace]	Esc__ Esc	\[NegativeThinSpace]	Esc__ Esc
\[MediumSpace]	Esc___ Esc	\[NegativeMediumSpace]	Esc___ Esc
\[ThickSpace]	Esc____ Esc	\[NegativeThickSpace]	Esc____ Esc
\[InvisibleSpace]	Esc_is Esc	\[NonBreakingSpace]	Esc_nbs Esc
\[NewLine]		\[IndentingNewLine]	Esc_nl Esc

Spacing and newline characters.

form	full name	alias	form	full name	alias
■	\[SelectionPlaceholder]	Esc_sp1 Esc	□	\[Placeholder]	Esc_p1 Esc

Characters used in buttons.

In the buttons in a palette, you often want to set up a template with placeholders to indicate where expressions should be inserted. `\[SelectionPlaceholder]` marks the position where an expression that is currently selected should be inserted when the contents of the button are pasted. `\[Placeholder]` marks other positions where subsequent expressions can be inserted. The `Tab` key will take you from one such position to the next.

form	full name	alias	form	full name	alias
_	\[SpaceIndicator]	Esc_space Esc	.	\[RoundSpaceIndicator]	
←	\[ReturnIndicator]	Esc_ret Esc	[CTRL]	\[ControlKey]	Esc_ctrl
RET	\[ReturnKey]	Esc_ret Esc	[CMD]	\[CommandKey]	Esc_cmc
ENTER	\[EnterKey]	Esc_ent Esc	[\[LeftModified]	Esc[_ Es
ESC	\[EscapeKey]	Esc_esc Esc]	\[RightModified]	Esc_] Es
⋮	\[AliasIndicator]	Esc_esc Esc	⌘	\[CloverLeaf]	Esc_cll

Representations of keys on a keyboard.

In describing how to enter input into *Mathematica*, it is sometimes useful to give explicit representations for keys you should press. You can do this using characters like `←` and `ESC`. Note that `_` and `.` are actually treated as spacing characters by *Mathematica*.

This string shows how to type α^2 .

`In[6]:= "ESCaESC [CTRL]^2 [CTRL_]"`

`Out[6]= ESCaESC [CTRL]^2 [CTRL_]`

<i>form</i>	<i>full name</i>	<i>form</i>	<i>full name</i>
\	\[Continuation]	-	\[SkeletonIndicator]
<<	\[LeftSkeleton]	\square	\[ErrorIndicator]
>>	\[RightGuillemet]		

Characters generated in *Mathematica* output.

Mathematica uses a \[Continuation] character to indicate that the number continues onto the next line.

In[7]:= **80 !**

```
Out[7]= 71 569 457 046 263 802 294 811 533 723 186 532 165 584 657 342 365 752 577 109 445 058 227 039 255 480 148 842 668 944 867 280 814 080 000 000 000 000 000 000 000
```

<i>form</i>	<i>full name</i>	<i>form</i>	<i>full name</i>
\[RawTab]	/	\[RawSlash]	
\[NewLine]	:	\[RawColon]	
\[RawReturn]	;	\[RawSemicolon]	
\[RawSpace]	<	\[RawLess]	
\[RawExclamation]	=	\[RawEqual]	
\[RawDoubleQuote]	>	\[RawGreater]	
\[RawNumberSign]	?	\[RawQuestion]	
\[RawDollar]	@	\[RawAt]	
\[RawPercent]	[\[RawLeftBracket]	
\[RawAmpersand]	\	\[RawBackslash]	
\[RawQuote]]	\[RawRightBracket]	
\[RawLeftParenthesis]	^	\[RawWedge]	
\[RawRightParenthesis]	_	\[RawUnderscore]	
\[RawStar]	`	\[RawBackquote]	
\[RawPlus]	{	\[RawLeftBrace]	
\[RawComma]		\[RawVerticalBar]	
\[RawDash]	}	\[RawRightBrace]	
\[RawDot]	~	\[RawTilde]	

Raw keyboard characters.

The fonts that are distributed with *Mathematica* contain their own renderings of many ordinary keyboard characters. The reason for this is that standard system fonts often do not contain appropriate renderings. For example, \wedge and \sim are often drawn small and above the centerline, while for clarity in *Mathematica* they must be drawn larger and centered on the centerline.