Wolfram *Mathematica*® Tutorial Collection

NOTEBOOKS AND DOCUMENTS

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

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

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

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

Printed in the United States of America.

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

©2008 Wolfram Research, Inc.

All rights reserved. No part of this document may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright holder.

Wolfram Research is the holder of the copyright to the Wolfram *Mathematic*a software system ("Software") described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, “look and feel,” programming language, and compilation of command names. Use of the Software unless pursuant to the terms of a license granted by Wolfram Research or as otherwise authorized by law is an infringement of the copyright.

Wolfram Research, Inc. and Wolfram Media, Inc. ("Wolfram") make no representations, express, statutory, or implied, with respect to the Software (or any aspect thereof), including, without limitation, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Wolfram does not warrant that the functions of the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. As such, Wolfram does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury or significant loss.

Mathematica, *MathLink*, and *MathSource* are registered trademarks of Wolfram Research, Inc. *J/Link*, *MathLM*, *.NET/Link*, and web*Mathematica* are trademarks of Wolfram Research, Inc. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Macintosh is a registered trademark of Apple Computer, Inc. All other trademarks used herein are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc.

Contents

Notebook Interface

Notebook Interfaces . . . 1
Doing Computations in Notebooks . . . 4
Notebooks as Documents . . . 7
Working with Cells . . . 12
The Option Inspector . . . 21
Notebook History Dialog . . . 23

Input and Output in Notebooks

Entering Greek Letters . . . 28
Entering Two-Dimensional Input . . . 30
Editing and Evaluating Two-Dimensional Expressions . . . 36
Entering Formulas . . . 38
Entering Tables and Matrices . . . 43
Subscripts, Bars and Other Modifiers . . . 45
Non-English Characters and Keyboards . . . 47
Other Mathematical Notation . . . 48
Forms of Input and Output . . . 50
Mixing Text and Formulas . . . 53
Displaying and Printing *Mathematica* Notebooks . . . 54
Setting Up Hyperlinks . . . 55
Automatic Numbering . . . 56
Exposition in *Mathematica* Notebooks . . . 57
Named Characters . . . 58

Textual Input and Output

How Input and Output Work . . . 61
The Representation of Textual Forms . . . 62
The Interpretation of Textual Forms . . . 64
Short and Shallow Output . . . 67
String-Oriented Output Formats . . . 70
Output Formats for Numbers . . . 74
Tables and Matrices . . . 79
Styles and Fonts in Output . . . 91
Representing Textual Forms by Boxes . . . 92
String Representation of Boxes . . . 97
Converting between Strings, Boxes and Expressions . . . 102

The Syntax of the *Mathematica* Language 106
Operators without Built-in Meanings 111
Defining Output Formats 114
Low-Level Input and Output Rules 116
Generating Unstructured Output 118
Formatted Output 121
Requesting Input 135
Messages 136
International Messages 141
Documentation Constructs 142

Manipulating Notebooks

Cells as *Mathematica* Expressions 145
Notebooks as *Mathematica* Expressions 148
Manipulating Notebooks from the Kernel 152
Manipulating the Front End from the Kernel 166
Front End Tokens 167
Executing Notebook Commands Directly in the Front End 169
The Structure of Cells 170
Styles and the Inheritance of Option Settings 171
Options for Cells 175
Text and Font Options 181
Options for Expression Input and Output 186
Options for Notebooks 189
Global Options for the Front End 193

Mathematical and Other Notation

Mathematical Notation in Notebooks 194
Special Characters 199
Names of Symbols and Mathematical Objects 206
Letters and Letter-like Forms 209
Operators 220
Structural Elements and Keyboard Characters 229

Notebook Interface

Using a Notebook Interface

If you use your computer via a purely graphical interface, you will typically double-click the *Mathematica* icon to start *Mathematica*. If you use your computer via a textually based operating system, you will typically type the command `mathematica` to start *Mathematica*.

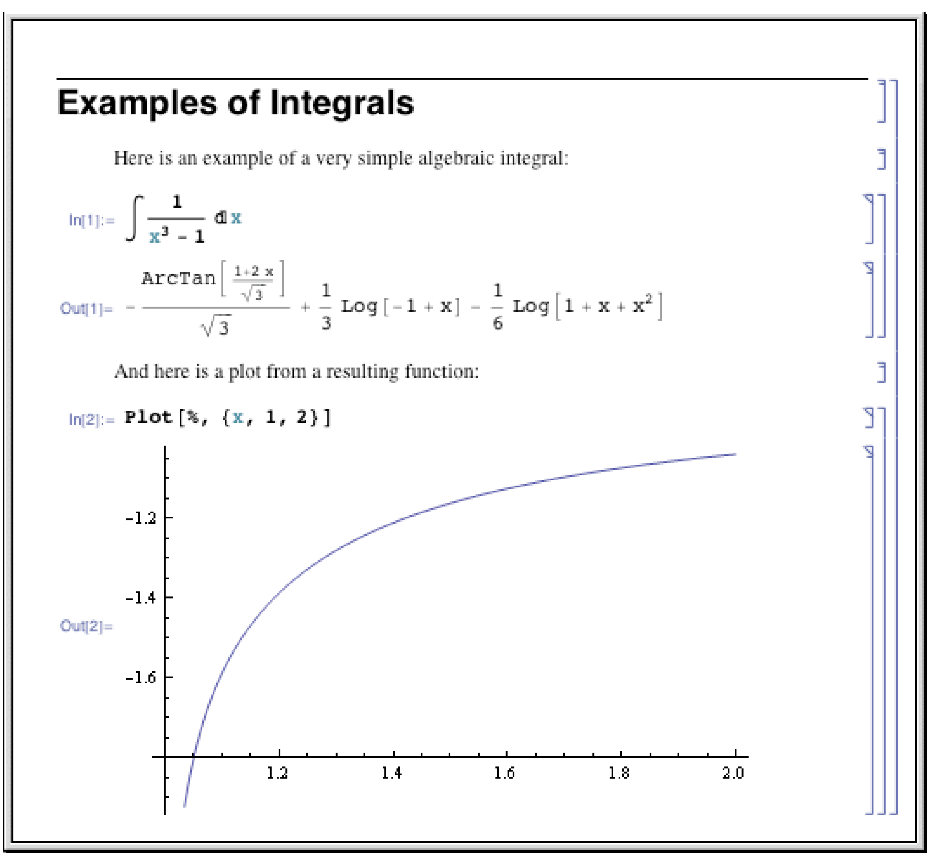
use an icon or the **Start** menu	graphical ways to start *Mathematica*
`mathematica`	the shell command to start *Mathematica*
text ending with Shift+Return	input for *Mathematica* (Shift+Return on some keyboards)
choose the **Exit** menu item	exiting *Mathematica* (**Quit** on some systems)

Running *Mathematica* with a notebook interface.

In a "notebook" interface, you interact with *Mathematica* by creating interactive documents.

The notebook front end includes many menus and graphical tools for creating and reading notebook documents and for sending and receiving material from the *Mathematica* kernel.

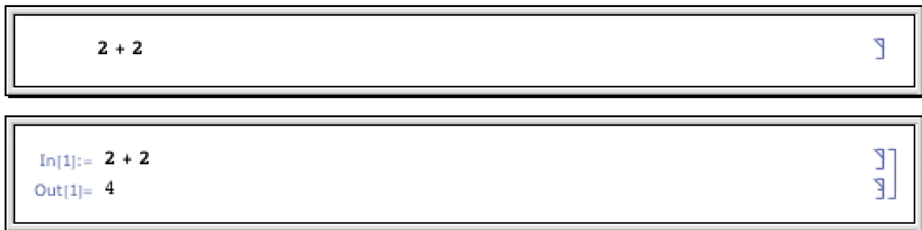
A notebook mixing text, graphics and *Mathematica* input and output.



When Mathematica is first started, it displays an empty notebook with a blinking cursor. You can start typing right away. *Mathematica* by default will interpret your text as input. You enter *Mathematica* input into the notebook, then type Shift+Return to make *Mathematica* process your input. (To type Shift+Return, hold down the Shift key, then press Return.) You can use the standard editing features of your graphical interface to prepare your input, which may go on for several lines. Shift+Return tells *Mathematica* that you have finished your input. If your keyboard has a numeric keypad, you can use its Enter key instead of Shift+Return.

After you send *Mathematica* input from your notebook, *Mathematica* will label your input with `In[n]:=`. It labels the corresponding output `Out[n]=`. Labels are added automatically.

You type 2 + 2, then end your input with Shift+Return. *Mathematica* processes the input, then adds the input label `In[1]:=`, and gives the output.



The output is placed below the input. By default, input/output pairs are grouped using rectangular cell brackets displayed in the right margin.

In *Mathematica* documentation, "dialogs" with *Mathematica* are shown in the following way:

With a notebook interface, you just type in 2 + 2. *Mathematica* then adds the label `In[1]:=`, and prints the result.

In[1]:= **2 + 2**

Out[1]= 4

You should realize that notebooks are part of the "front end" to *Mathematica*. The *Mathematica* kernel which actually performs computations may be run either on the same computer as the front end, or on another computer connected via a network. Sometimes, the kernel is not even started until you actually do a calculation with *Mathematica*.

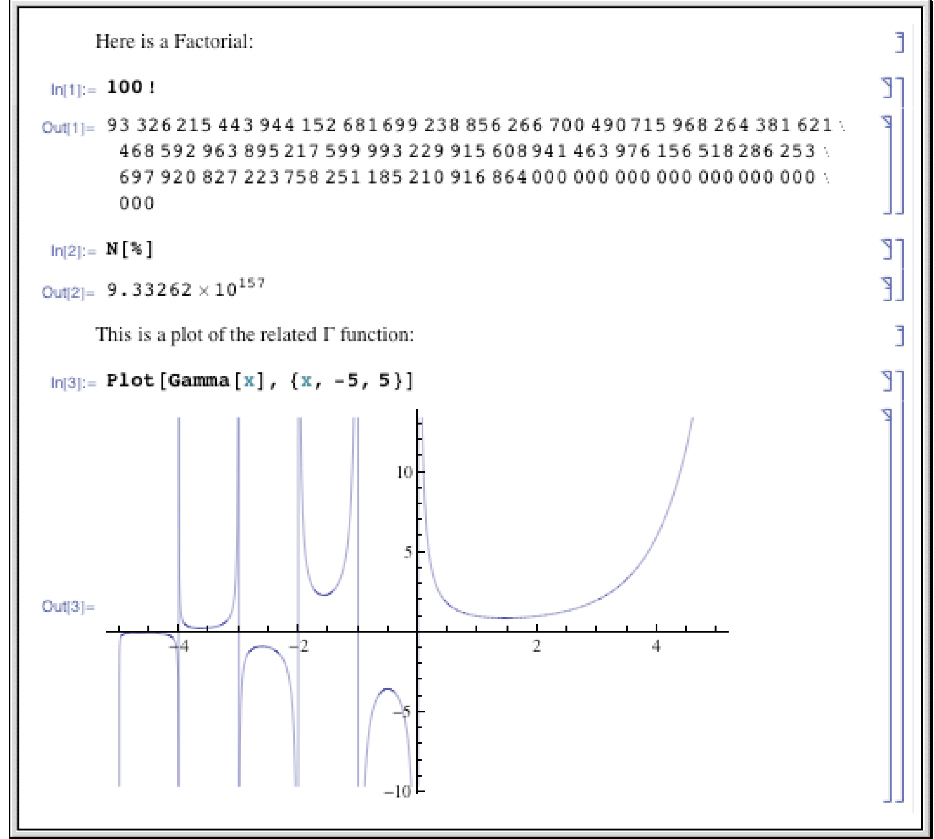
The built-in *Mathematica* Documentation Center (**Help ▸ Documentation Center**), where you might be reading this documentation, is itself an example of a *Mathematica* notebook. You can evaluate and modify examples in place, or type your own examples.

In addition to the standard textual input, *Mathematica* supports the use of generalized, non-textual input such as graphics and user interface controls, freely mixed with textual input.

To exit *Mathematica*, you typically choose the **Exit** menu item in the notebook interface.

Doing Computations in Notebooks

A typical *Mathematica* notebook containing text, graphics and *Mathematica* expressions. The brackets on the right indicate the extent of each cell.



Mathematica notebooks are structured interactive documents that are organized into a sequence of *cells*. Each cell may contain text, graphics, sounds or *Mathematica* expressions in any combination. When a notebook is displayed on the screen, the extent of each cell is indicated by a bracket on the right.

The notebook front end for *Mathematica* provides many ways to enter and edit the material in a notebook. Some of these ways will be standard to whatever computer system or graphical interface you are using. Others are specific to *Mathematica*.

Shift + Return	send a cell of input to the *Mathematica* kernel

Doing a computation in a *Mathematica* notebook.

Once you have prepared the material in a cell, you can send it as input to the *Mathematica* kernel simply by pressing Shift+Return. The kernel will send back whatever output is generated, and the front end will create new cells in your notebook to display this output. Note that if you have a numeric keypad on your keyboard, then you can use its Enter key as an alternative to Shift+Return.

Here is a cell ready to be sent as input to the *Mathematica* kernel.

```
3^100
```

The output from the computation is inserted in a new cell.

```
In[1]:= 3^100
Out[1]= 515 377 520 732 011 331 036 461 129 765 621 272 702 107 522 001
```

Most kinds of output that you get in *Mathematica* notebooks can readily be edited, just like input. Usually *Mathematica* will convert the output cell into an input cell when you first start editing it.

Once you have done the editing you want, you can typically just press Shift+Return to send what you have created as input to the *Mathematica* kernel.

Here is a typical computation in a *Mathematica* notebook.

```
In[1]:= Integrate[Sqrt[x + 1] / Sqrt[x - 1], x]
```

$$\text{Out[1]=}\ \frac{(-1+x)\sqrt{1+x}+2\sqrt{-1+x}\ \text{ArcSinh}\left[\frac{\sqrt{-1+x}}{\sqrt{2}}\right]}{\sqrt{\frac{-1+x}{1+x}}\ \sqrt{1+x}}$$

If you start editing the output cell, *Mathematica* will automatically change it to an input cell.

```
In[1]:= Integrate[Sqrt[x + 1] / Sqrt[x - 1], x]
```

$$\frac{(-1+x)\sqrt{1+x}+2\sqrt{-1+x}\ \text{ArcSinh}\left[\frac{\sqrt{-1+x}}{\sqrt{2}}\right]}{\sqrt{\frac{-1+x}{1+x}}\ \sqrt{1+x}}$$

After you have edited the output, you can send it back as further input to the *Mathematica* kernel.

```
In[1]:= Integrate[Sqrt[x + 1] / Sqrt[x - 1], x]
```

$$\text{In[2]:=}\ \frac{(-1+x)\sqrt{1+x}+2\sqrt{-1+x}\,\mathrm{D}\left[\mathrm{ArcSinh}\left[\frac{\sqrt{-1+x}}{\sqrt{2}}\right],x\right]}{\sqrt{\frac{-1+x}{1+x}}\sqrt{1+x}}\ \text{// Simplify}$$

$$\text{Out[2]=}\ \frac{x^2}{\sqrt{\frac{-1+x}{1+x}}\,(1+x)}$$

When you do computations in a *Mathematica* notebook, each line of input is typically labeled with `In[n] :=`, while each line of output is labeled with the corresponding `Out[n] =`.

There is no reason, however, that successive lines of input and output should necessarily appear one after the other in your notebook. Often, for example, you will want to go back to an earlier part of your notebook, and reevaluate some input you gave before.

It is important to realize that in most cases wherever a particular expression appears in your notebook, it is the line number given in `In[n] :=` or `Out[n] =` which determines when the expression was processed by the *Mathematica* kernel. Thus, for example, the fact that one expression may appear earlier than another in your notebook does not mean that it will have been evaluated first by the kernel. This will only be the case if it has a lower line number.

Each line of input and output is given a label when it is evaluated by the kernel. It is these labels, not the position of the expression in the notebook, that indicate the ordering of evaluation by the kernel.

```
Results:

In[2]:= s^2 + 2
Out[2]= 146

In[4]:= s^2 + 2
Out[4]= 10 002

Settings for s:

In[1]:= s = 12
Out[1]= 12

In[3]:= s = 100
Out[3]= 100
```

The exception to this rule is when an output contains the formatted results of a `Dynamic` or `Manipulate` function. Such outputs will reevaluate in the kernel on an as-needed basis long after the evaluation which initially created them. See "Dynamic Interactivity Language" for more information on this functionality.

As you type, *Mathematica* applies syntax coloring to your input using its knowledge of the structure of functions. The coloring highlights unmatched brackets and quotes, undefined global symbols, local variables in functions and various programming errors. You can ask why *Mathematica* colored your input by selecting it and using the **Why the Coloring?** item in the **Help** menu.

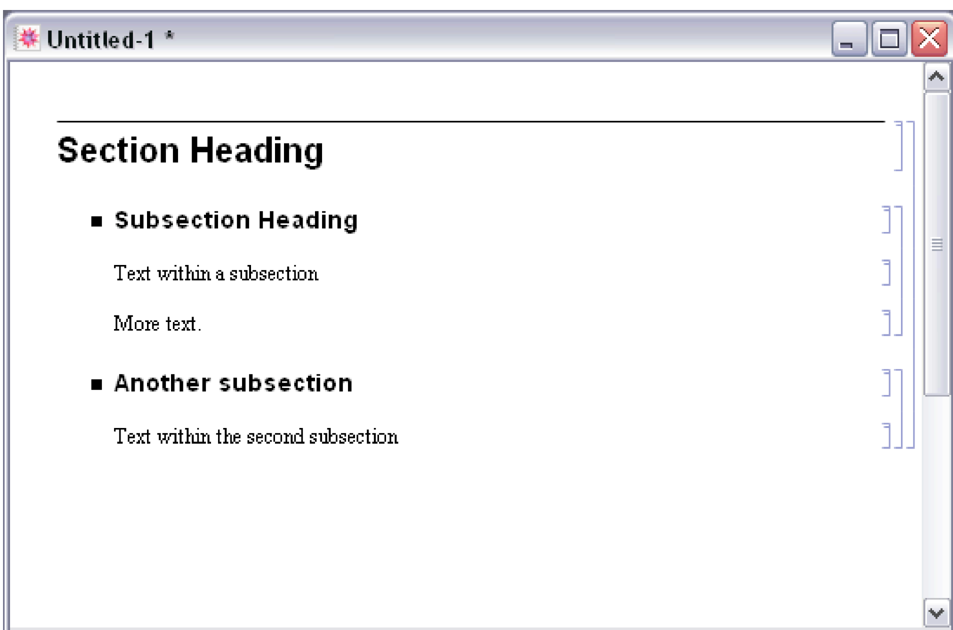
If you make a mistake and try to enter input that the *Mathematica* kernel does not understand, then the front end will produce a beep and emphasize any syntax errors in the input with color. In general, you will get a beep whenever something goes wrong in the front end. You can find out the origin of the beep using the **Why the Beep?** item in the **Help** menu.

Notebooks as Documents

Mathematica notebooks allow you to create documents that can be viewed interactively on screen or printed on paper.

Particularly in larger notebooks, it is common to have chapters, sections and so on, each represented by groups of cells. The extent of these groups is indicated by a bracket on the right.

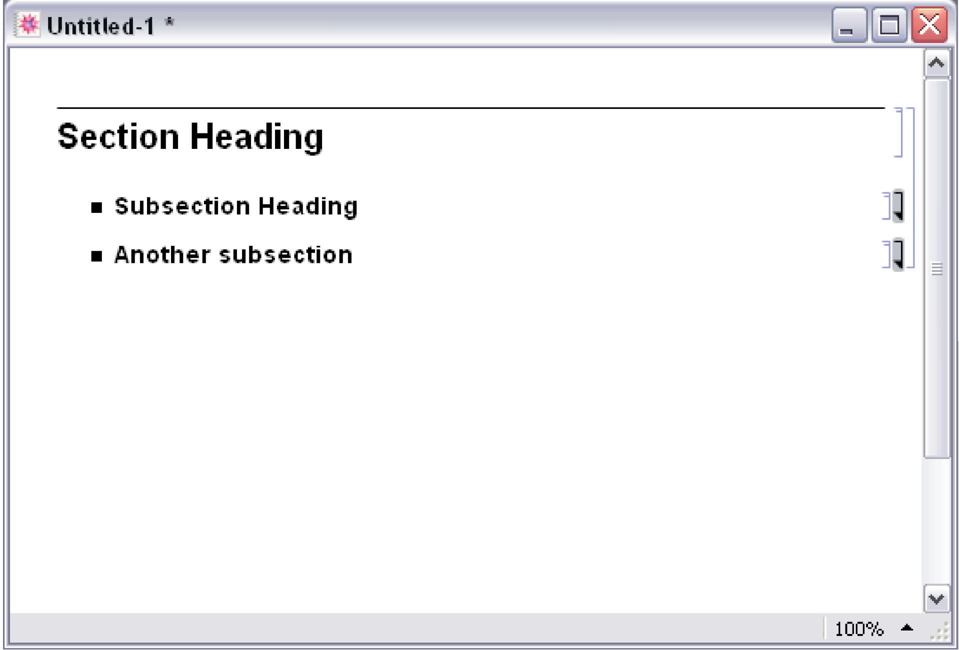
The grouping of cells in a notebook is indicated by nested brackets on the right.



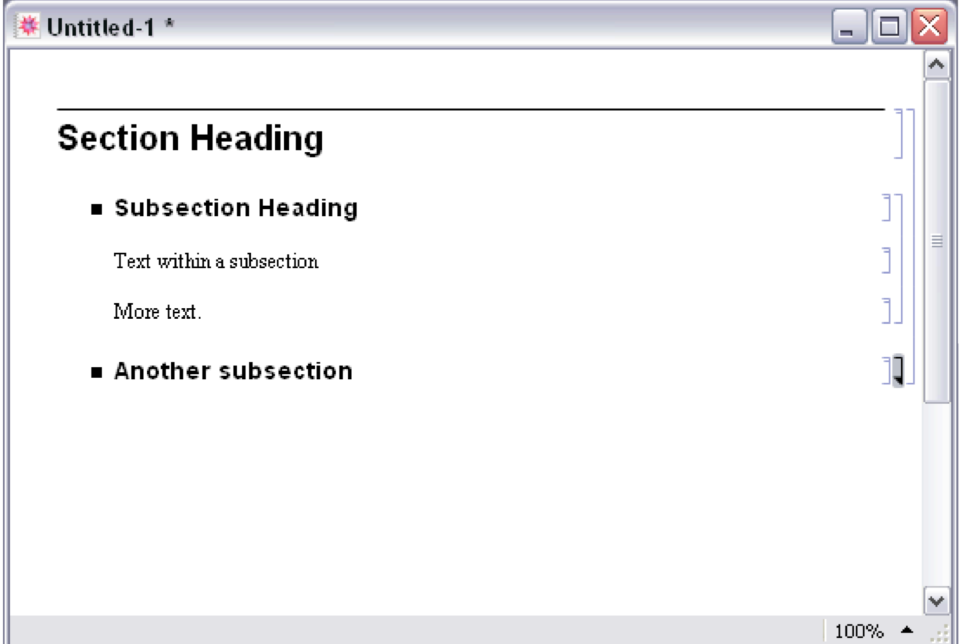
A group of cells can be either *open* or *closed*. When it is open, you can see all the cells in it explicitly. But when it is closed, you see only the cell around which the group is closed. Cell groups are typically closed around the first or *heading* cell in the group, but you can close a group around any cell in that group.

Large notebooks are often distributed with many closed groups of cells, so that when you first look at the notebook, you see just an outline of its contents. You can then open parts you are interested in by double-clicking the appropriate brackets.

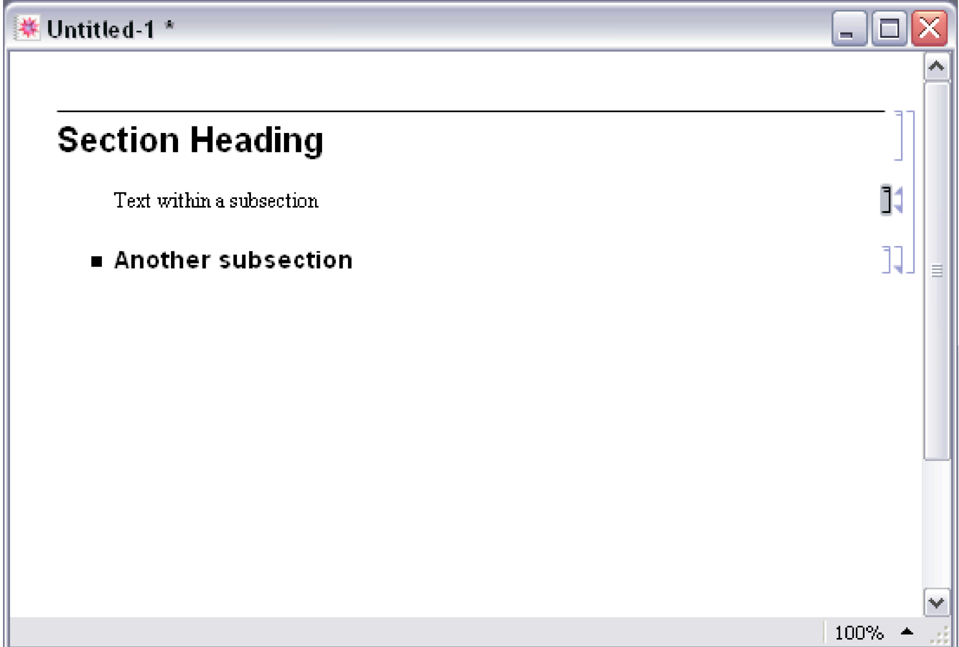
Double-clicking the bracket that spans a group of cells closes the group, leaving only the first cell visible.



When a group is closed, the bracket for it has an arrow at the bottom. Double-clicking this arrow opens the group again.

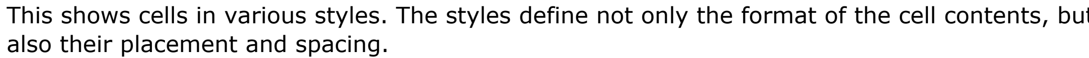


Double-clicking the bracket of a cell that is not the first of a cell group closes the cell group around that cell and creates a bracket with up and down arrows (or only an up arrow if the cell was the last in the group).

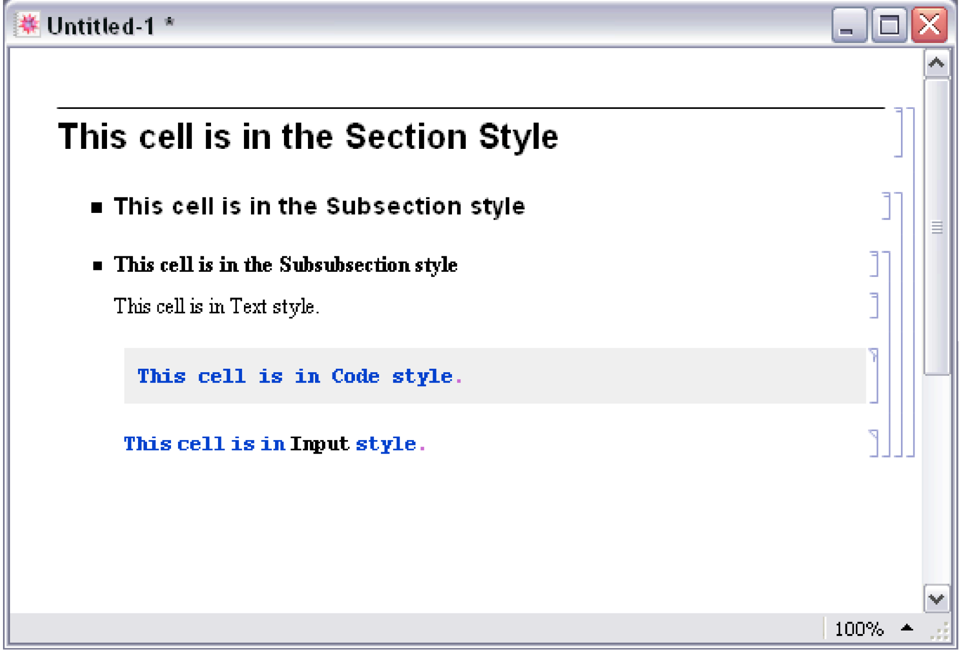


Each cell within a notebook is assigned a particular *style* which indicates its role within the notebook. Thus, for example, material intended as input to be executed by the *Mathematica* kernel is typically in `Input` style, while text that is intended purely to be read is typically in `Text` style.

The *Mathematica* front end provides menus and keyboard shortcuts for creating cells with different styles, and for changing styles of existing cells.



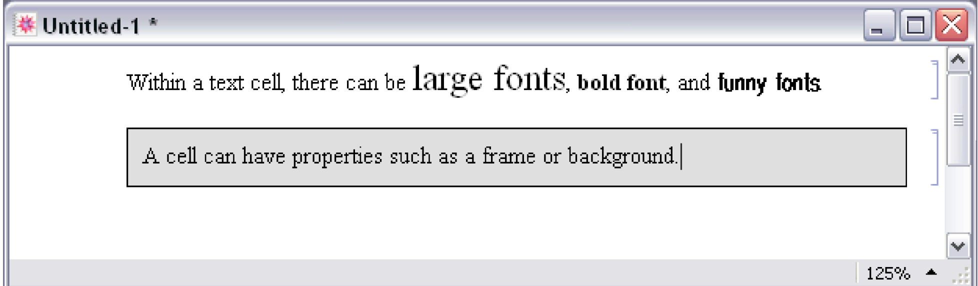
This shows cells in various styles. The styles define not only the format of the cell contents, but also their placement and spacing.



By putting a cell in a particular style, you specify a whole collection of properties for the cell, including for example how large and in what font text should be given.

The *Mathematica* front end allows you to modify such properties, either for complete cells, or for specific material within cells.

Even within a cell of a particular style, the *Mathematica* front end allows a wide range of properties to be modified separately.



Ordinary *Mathematica* notebooks can be read by non-*Mathematica* users using the free product, *Mathematica Player*, which allows viewing and printing, but does not allow computations of any

Manipulate

kind to be performed. This product also supports notebook player files (.nbp), which have been specially prepared by Wolfram Research to allow interaction with dynamic content such as the output of `Manipulate`. For example, all the notebook content on The Wolfram Demonstrations Project site is available as notebook player files.

Mathematica front end	creating and editing *Mathematica* notebooks
Mathematica kernel	doing computations in notebooks
Mathematica Player	reading *Mathematica* notebooks and running Demonstrations

Programs required for different kinds of operations with notebooks.

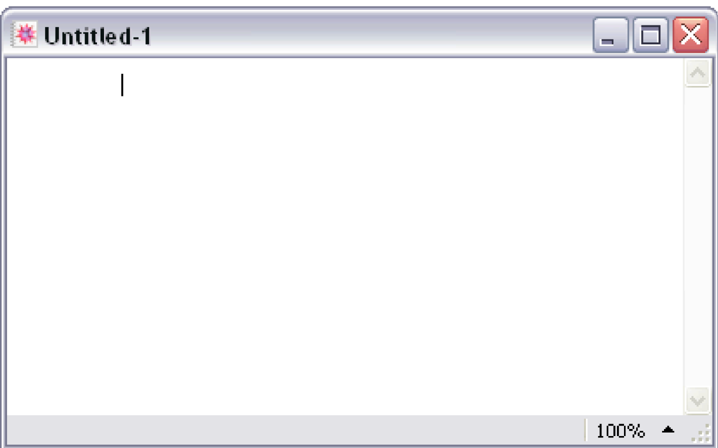
Working with Cells

Mathematica notebooks consist of sequences of cells. The hierarchy of cells serves as a structure for organizing the information in a notebook, as well as specifying the overall look of the notebook.

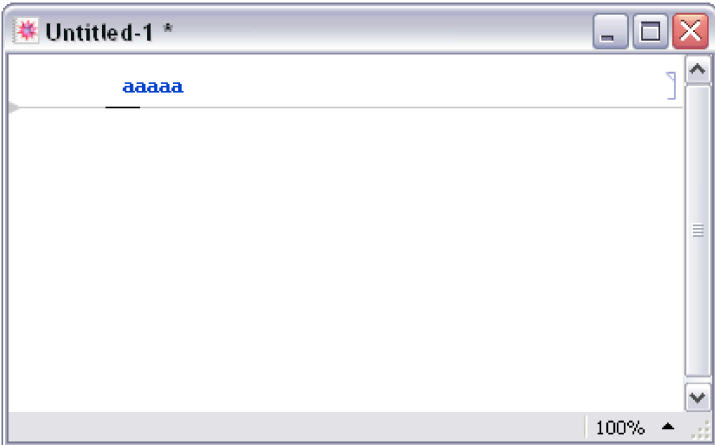
Font, color, spacing, and other properties of the appearance of cells are controlled using stylesheets. The various kinds of cells associated with a notebook's stylesheet are listed in **Format ▸ Style**. *Mathematica* comes with a collection of color and black-and-white stylesheets, which are listed in the **Format ▸ Stylesheet** menu.

In a New Session:

When *Mathematica* is first started, it displays an empty notebook with a blinking cursor. You can start typing right away.



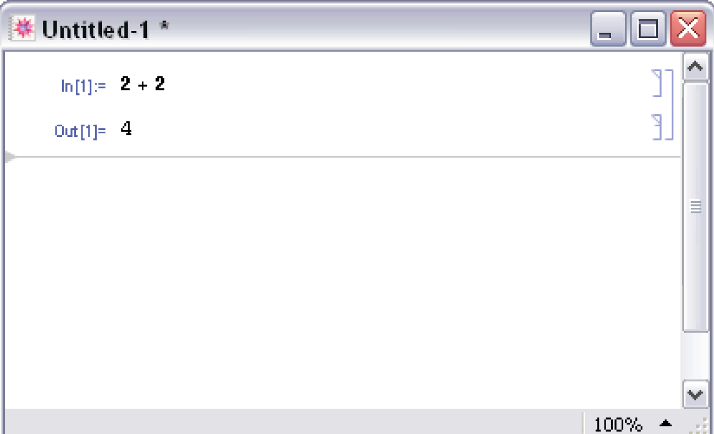
The insertion point is indicated by the cell insertion bar, a solid gray line with a small black cursor running horizontally across the notebook. The cell insertion bar is the place where new cells will be created, either as you type or programmatically. To set the position of the insertion bar, click in the notebook.



To Create a New Cell:

Move the pointer in the notebook window until it becomes a horizontal I-beam.

Click, and a cell insertion bar will appear; start typing. By default, new cells are *Mathematica* input cells.



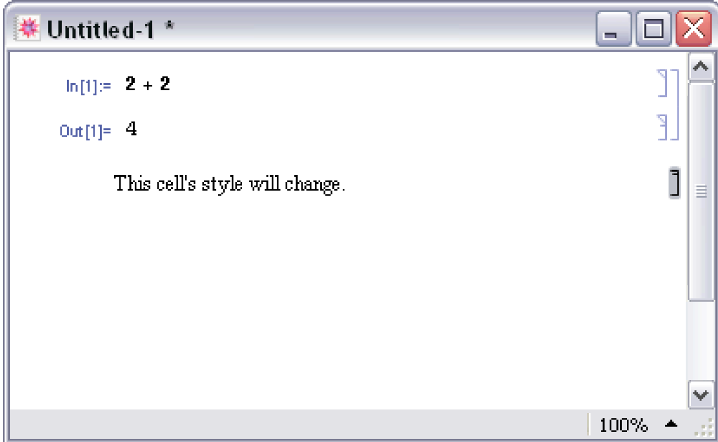
To Create a New Cell to Hold Ordinary Text:

Click in the notebook to get a cell insertion bar. Choose **Format** ▸ **Style** ▸ **Text** or use the keyboard shortcut Cmd+7.

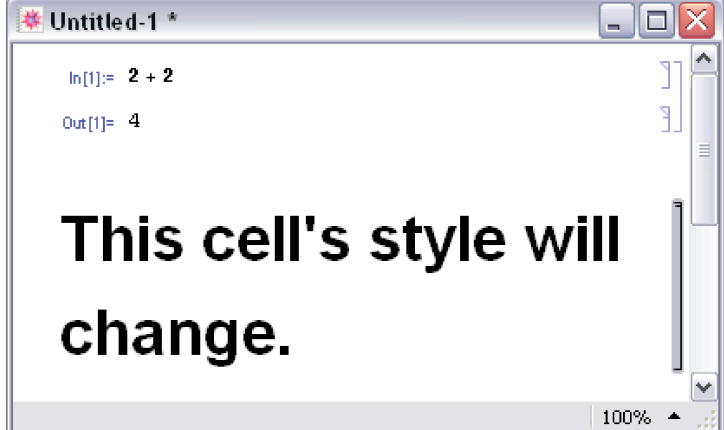
When you start typing, a text cell bracket appears.

To Change the Style of a Cell:

Click the cell bracket. The bracket is highlighted.

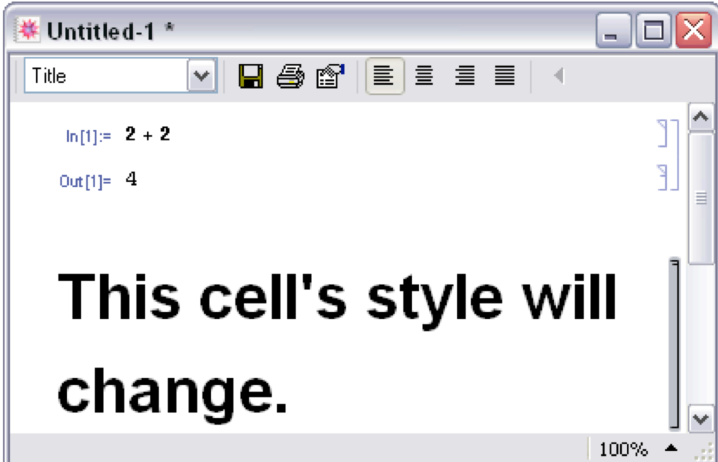


Select a style from **Format** ▸ **Style**. The cell will immediately reflect the change.

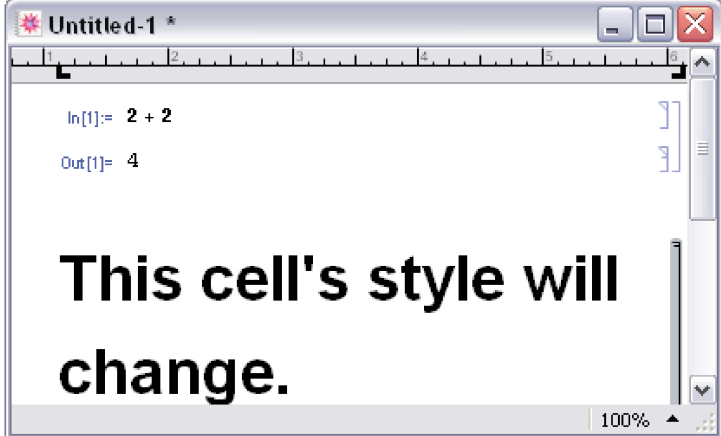


Alternatively, you can simultaneously press Cmd with one of the numbered keys, 0 through 9, to select a style.

Choose **Window ▸ Show Toolbar** to get a toolbar at the top of the notebook.

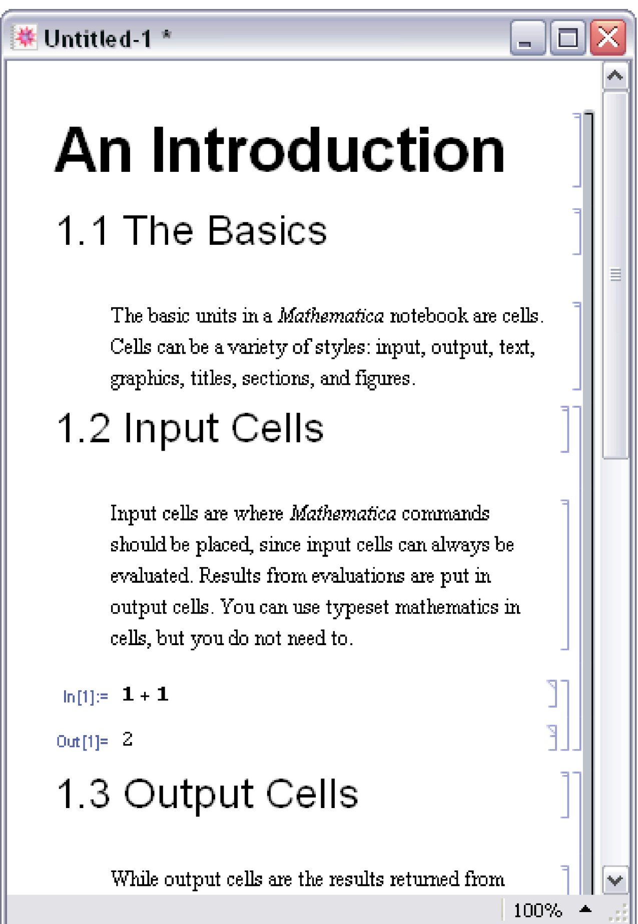


Choose **Window ▸ Show Ruler** to get a ruler at the top of the notebook.

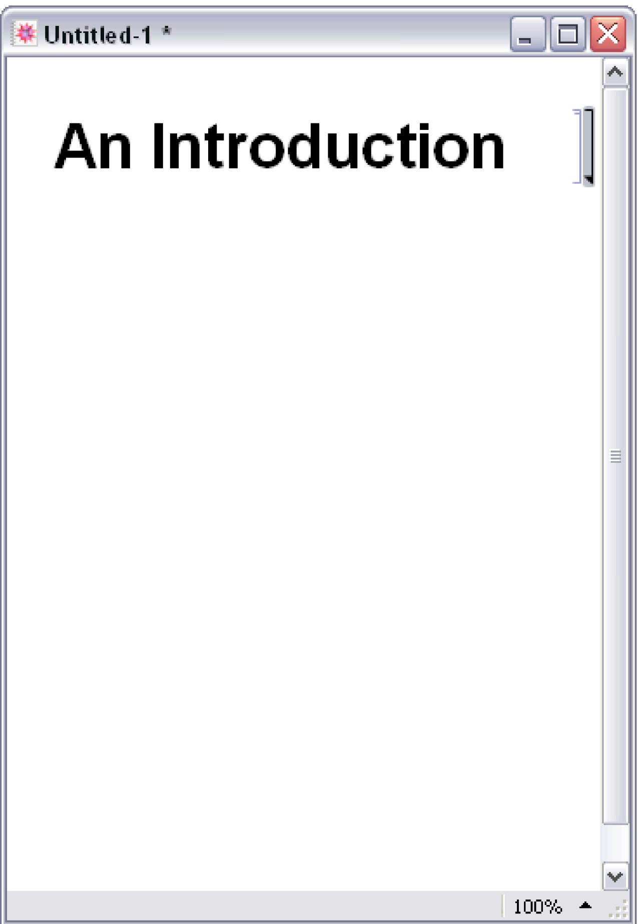


To Close a Group of Cells:

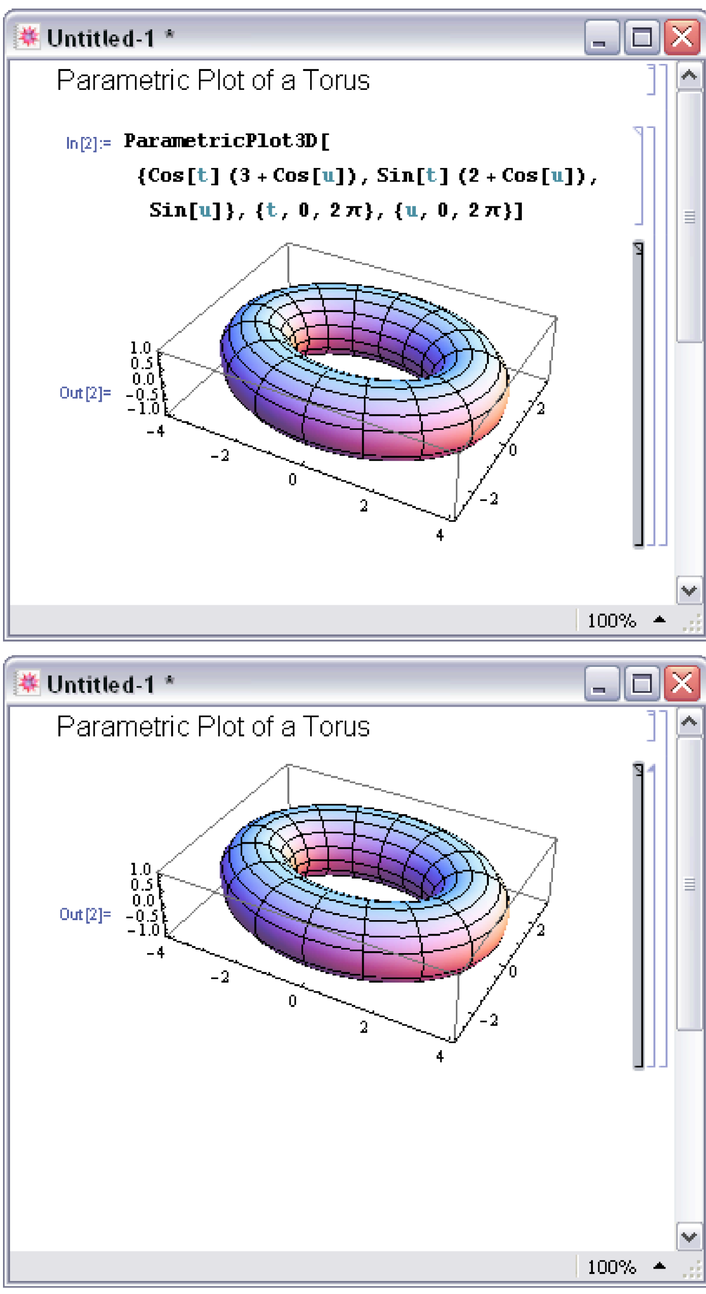
Double-click the outermost cell bracket of the group.



When a group is closed, only the first cell in the group is displayed by default. The group bracket is shown with a triangular flag at the bottom.



To specify which cells remain visible when the cell group is closed, select those cells and double-click to close the group. The closed group bracket is shown with triangular flags at the top and bottom if the visible cells are within a cell group, or with a triangular flag at the top if they are at the end of a cell group.



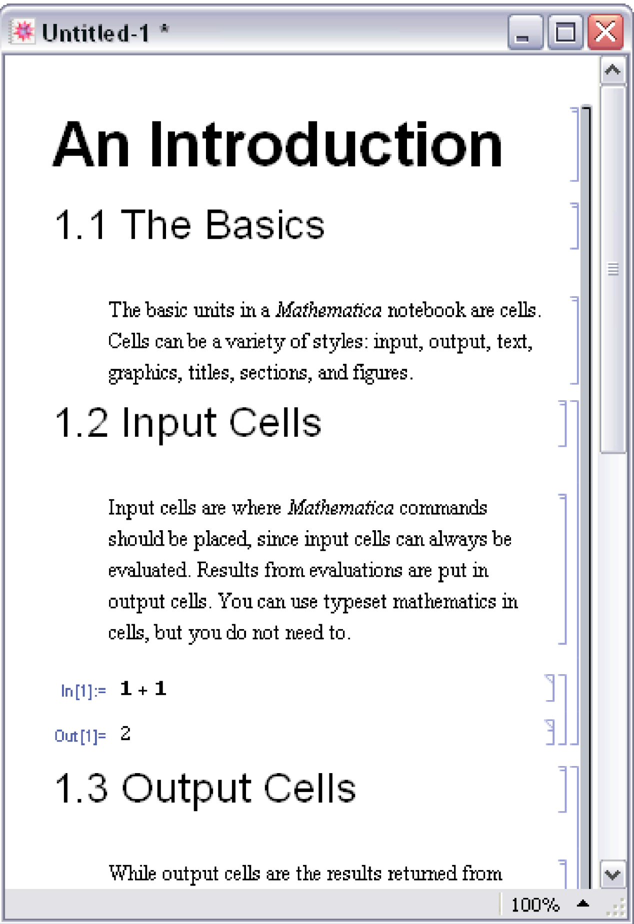
To Open a Group of Cells:

Double-click a closed group's cell bracket.

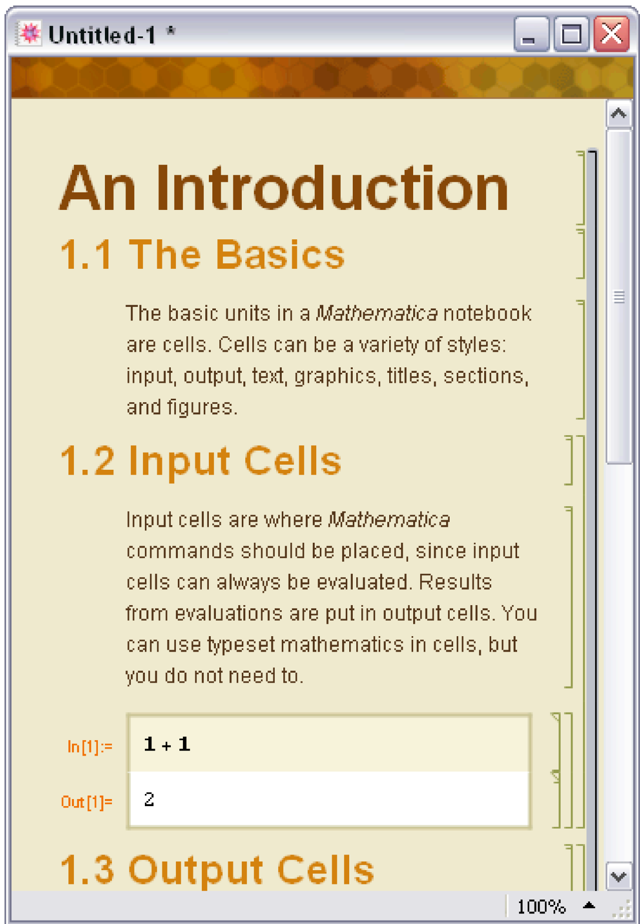
To Print a Notebook:

Choose **File** ▸ **Print**. The notebook style will be automatically optimized for printing.

To Change the Overall Look of a Notebook:



Choose **Format ▸ Stylesheet**. Select a stylesheet from the menu. All cells in the notebook will change appearance, based on the definitions in the new stylesheet.



Use **Format ▸ Edit Stylesheet** to customize stylesheets for *Mathematica* notebooks.

Changes to a notebook that only involve opening or closing cell groups will not cause the front end to ask you if you want to save such changes when you close the notebook before saving. To save these changes, use **File ▸ Save** before you close the notebook or quit *Mathematica*.

To close a notebook, click the **Close** button in the title bar. You will be prompted to save any unsaved changes.

On Windows, to close notebooks without being prompted to save, hold down the Shift key when clicking the **Close** box.

The Option Inspector

Introduction

Many aspects of the *Mathematica* front end, such as the styles of cells, the appearance of notebooks, or the parameters used in typesetting, are controlled by options. For example, text attributes such as size, font, and color each correspond to a separate option. You can set options by directly editing the expression for a cell or notebook. But in most cases it is simpler to use the Option Inspector.

The Option Inspector is a special tool for viewing and modifying option settings. It provides a comprehensive listing of all front end options, grouped according to their function. You can specify not only the setting for an option, but also the level at which it will take effect: globally, for an entire notebook, or for a selection.

To use the Option Inspector, choose **Format ▸ Option Inspector**. This brings up a dialog box with two popup menus on top. The popup menu on the left specifies the level at which options will take effect. The popup menu on the right allows you to choose if you want the options listed by category, alphabetically, or as text.

Inheritance of Options

The Option Inspector allows you to set the value of an option on three different levels. In increas ing order of precedence, the levels are as follows.

Global Preferences - settings for the entire application

Selected Notebook - settings for an entire notebook

Selection - settings for the current selection, e.g. for a group of cells, a single cell, or text within a cell

The levels lower in the hierarchy inherit their options from the level immediately above them. For example, if a notebook has the option `Editable` set to `True`, by default all cells in the notebook will be editable.

You can, however, override the inherited value of an option by explicitly changing its value. For example, if you do not want a particular cell in your notebook to be editable, you can select the cell and set `Editable` to `False`. This inheritance property of options provides you with a great deal of control over the behavior of the front end, since you can set any option to have different values at each level, as required.

Note: At each level, only the options that can be set at that level are listed in the Option Inspector. All other options appear dimmed, indicating that they cannot be changed unless you go to a higher or lower level.

Searching for an Option

To search for a specific option, begin typing its name in the text field. The Option Inspector goes to the first matching option. Press Enter to go to the next matching item on the list. (On Macintosh, the Option Inspector displays all matching options at once).

Each line in the list of options gives the option name followed by its current value. You can change the option's value by choosing from the popup menu next to the option setting, or by selecting the option and clicking the value, typing over it, and pressing Enter.

When you start *Mathematica* for the first time, the values of all the options are set to their default values. Each time you modify one of the options, a symbol appears next to it, indicating that the value has been changed. Clicking the symbol resets the option to its default value.

Setting Options: An Example

Suppose you want to draw a frame around a cell. The option that controls this property of a cell is called `CellFrame`.

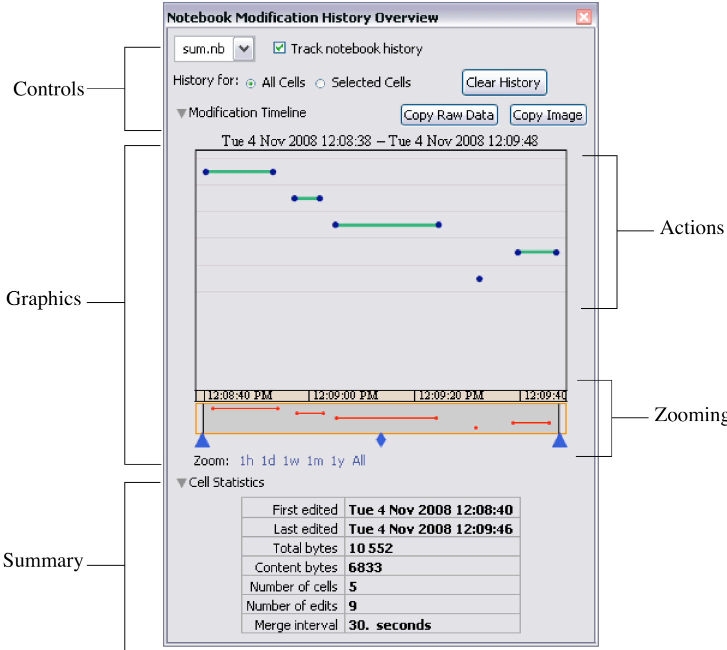
To Draw a Frame around a Cell:

1. Select the cell by clicking the cell bracket.
2. Choose **Format** ▸ **Option Inspector** to open the Option Inspector window.
3. Choose **Selection** from the first popup menu.
4. Click **Cell Options** ▸ **Display Options**. This gives a list of all options that control how a cell is displayed in the notebook.
5. Type `True` into the value field next to the option `CellFrame`. An icon appears next to the option, indicating that its value has been changed. The cell that you selected now has a frame drawn around it.

Alternatively, you can begin typing "cellframe" in the text field. This leads you directly to the `CellFrame` option without having to search by category. This feature provides a useful way to locate an option if you are unsure of the category it belongs in.

Notebook History Dialog

This dialog displays information regarding the editing times of the input notebook. This is a "live" dialog that dynamically updates as changes are made to the notebook. It can be accessed through **Cell** ▸ **Notebook History**.



The time information is saved in each cell of the notebook, in the form of a list of numbers and/or pairs of numbers.

```
Cell[
BoxData["123"], "Input",
 CellChangeTimes->{{3363263352.09502, 3363263354.03695}, 3363263406.22268,
3363263441.939}
 ]
```

Each number represents the exact time of an edit, in absolute time units. A list of pairs indicates multiple edits that have occurred during this interval.

Consecutive edits are recorded as an interval if they happen within a set time period. This period is determined by `CellChangeTimesMergeInterval`, which can be set through the Option Inspector or the **Advanced** section of the **Preferences** dialog. The default is 30 seconds.

The notebook history tracking feature can be turned off at the global level by using the **Preferences** dialog or by setting `TrackCellChangeTimes` to `False`.

Features

Controls

Notebook Chooser Popup Menu

This popup menu allows users to choose from all current open notebooks. The chosen notebook will be brought to the front, making it the new input notebook.

Track History Checkbox

This checkbox enables or disables the notebook history tracking feature for the input notebook.

All/Selected Cells Radio Buttons

These radio buttons allow the graphics display to show information associated only with the selected cells or all cells in the input notebook.

Clear History Button

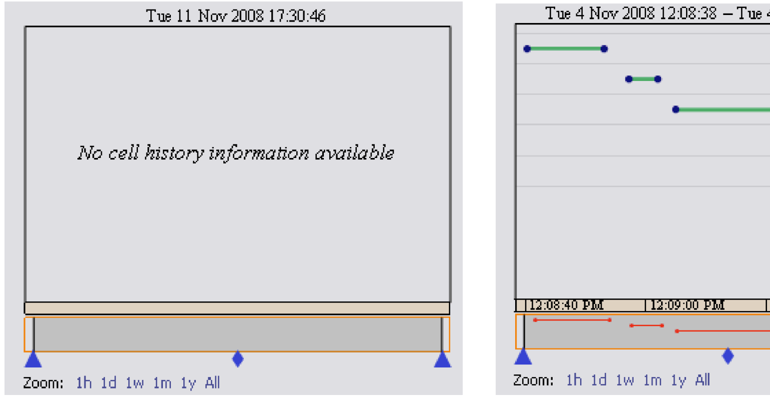
This button will clear the stored edit time information from all currently displayed cells. This operation cannot be undone.

Copy Buttons

The **Copy Raw Data** button will copy the raw data (in the form of a list of numbers and/or pairs of numbers) from currently displayed cells to the clipboard.

The **Copy Image** button will copy the currently displayed graphics to the clipboard. All dynamic features, except tooltips, are stripped from the copied graphics. This includes the zooming features.

Graphics

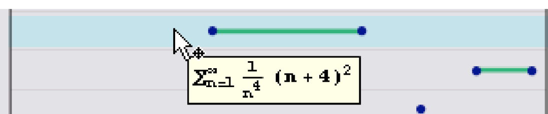


The graphics display plots cells versus time. Each cell in the notebook corresponds to each row on the y axis. The corresponding edit times are plotted as points, while edit intervals are represented by lines.

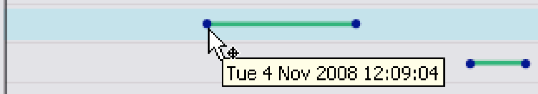
Mouse Events

As you mouse over the graphics, the mouse tooltip may provide some useful details for the following elements:

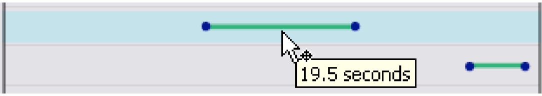
- Each row on the y axis will display the corresponding cell's contents.



- Points will display the exact time of the edit (which corresponds to the computer clock at the time of edit).

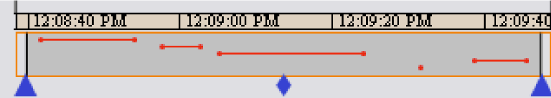


- Lines will display the length of the edit interval (this value may be greater than the `CellChangeTimesMergeInterval` value).



Clicking a highlighted row will select the corresponding cell in the input notebook if and only if the selection-only checkbox is unchecked.

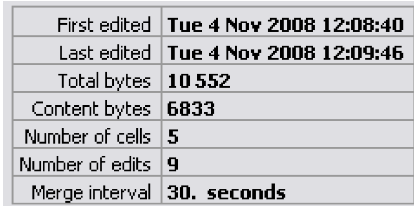
Zooming



The graphics display comes with a couple of zooming features for the time axis:

- The blue triangles at the bottom can be dragged to change the plotted time interval. Use the middle diamond to pan the graphics using the same time interval.
- Clicking any time label blocks will zoom into that interval of time. With this feature, users can actually zoom down to the last second (which may be out of range with the previous zoom feature).
- Clicking the shaded area will undo the last zoom action. Click outside the shaded area to revert to showing the entire time interval.

Summary



First edited	**Tue 4 Nov 2008 12:08:40**
Last edited	**Tue 4 Nov 2008 12:09:46**
Total bytes	**10 552**
Content bytes	**6833**
Number of cells	**5**
Number of edits	**9**
Merge interval	**30. seconds**

The summary is a concise, overall display of relevant cell information. This display also respects the setting of the selection-only checkbox.

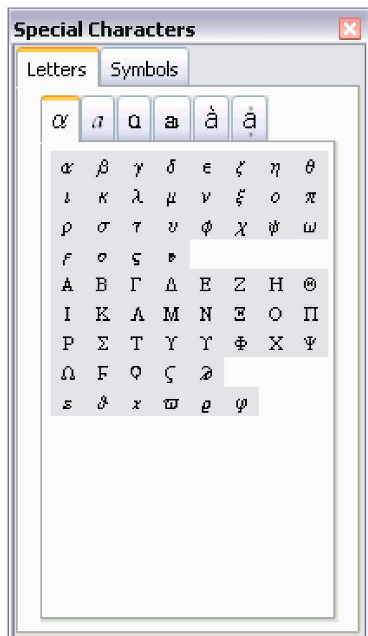
Input and Output in Notebooks

Entering Greek Letters

click on α	use a button in a palette
`\[Alpha]`	use a full name
Esc `a` Esc or Esc `alpha` Esc	use a standard alias (shown below as Esc a Esc)
Esc `\alpha` Esc	use a $\TeX$ alias
Esc `α` Esc	use an HTML alias

Ways to enter Greek letters in a notebook.

Here is a palette for entering common Greek letters.



You can use Greek letters just like the ordinary letters that you type on your keyboard.

```
In[1]:= Expand[(α + β)^3]
```

Out[1]= $\alpha^3 + 3\,\alpha^2\,\beta + 3\,\alpha\,\beta^2 + \beta^3$

There are several ways to enter Greek letters. This input uses full names.

In[2]:= **Expand[(α + β)^3]**

Out[2]= $\alpha^3 + 3\alpha^2\beta + 3\alpha\beta^2 + \beta^3$

	full name	aliases		full name	aliases
α	\[Alpha]	Esc a Esc, Esc alpha Esc	Γ	\[CapitalGamma]	Esc G Esc, Esc Gamma Esc
β	\[Beta]	Esc b Esc, Esc beta Esc	Δ	\[CapitalDelta]	Esc D Esc, Esc Delta Esc
γ	\[Gamma]	Esc g Esc, Esc gamma Esc	Θ	\[CapitalTheta]	Esc Q Esc, Esc Th Esc, Esc Theta Esc
δ	\[Delta]	Esc d Esc, Esc delta Esc	Λ	\[CapitalLambda]	Esc L Esc, Esc Lambda Esc
ϵ	\[Epsilon]	Esc e Esc, Esc epsilon Esc	Π	\[CapitalPi]	Esc P Esc, Esc Pi Esc
ζ	\[Zeta]	Esc z Esc, Esc zeta Esc	Σ	\[CapitalSigma]	Esc S Esc, Esc Sigma Esc
η	\[Eta]	Esc h Esc, Esc et Esc, Esc eta Esc	ϒ	\[CapitalUpsilon]	Esc U Esc, Esc Upsilon Esc
θ	\[Theta]	Esc q Esc, Esc th Esc, Esc theta Esc	Φ	\[CapitalPhi]	Esc F Esc, Esc Ph Esc, Esc Phi Esc
κ	\[Kappa]	Esc k Esc, Esc kappa Esc	Χ	\[CapitalChi]	Esc C Esc, Esc Ch Esc, Esc Chi Esc
λ	\[Lambda]	Esc l Esc, Esc lambda Esc	Ψ	\[CapitalPsi]	Esc Y Esc, Esc Ps Esc, Esc Psi Esc
μ	\[Mu]	Esc m Esc, Esc mu Esc	Ω	\[CapitalOmega]	Esc O Esc, Esc W Esc, Esc Omega Esc
ν	\[Nu]	Esc n Esc, Esc nu Esc			
ξ	\[Xi]	Esc x Esc, Esc xi Esc			
π	\[Pi]	Esc p Esc, Esc pi Esc			
ρ	\[Rho]	Esc r Esc, Esc rho Esc			
σ	\[Sigma]	Esc s Esc, Esc sigma Esc			
τ	\[Tau]	Esc t Esc, Esc tau Esc			
ϕ	\[Phi]	Esc f Esc, Esc ph Esc, Esc phi Esc			
φ	\[CurlyPhi]	Esc j Esc, Esc cph Esc, Esc cphi Esc			
χ	\[Chi]	Esc c Esc, Esc ch Esc, Esc chi Esc			
ψ	\[Psi]	Esc y Esc, Esc ps Esc, Esc psi Esc			
ω	\[Omega]	Esc o Esc, Esc w Esc, Esc omega Esc			

Commonly used Greek letters. TeX aliases are not listed explicitly.

Note that in *Mathematica* the letter π stands for `Pi`. None of the other Greek letters have special meanings.

π stands for `Pi`.

```
In[3]:= N[π]
Out[3]= 3.14159
```

You can use Greek letters either on their own or with other letters.

```
In[4]:= Expand[(Rαβ + Ξ)^4]
```

$$\text{Out[4]=}\quad R\alpha\beta^4 + 4\,R\alpha\beta^3\,\Xi + 6\,R\alpha\beta^2\,\Xi^2 + 4\,R\alpha\beta\,\Xi^3 + \Xi^4$$

The symbol $\pi\alpha$ is not related to the symbol π.

```
In[5]:= Factor[πα^4 - 1]
```

$$\text{Out[5]=}\quad (-1 + \pi\alpha)\,(1 + \pi\alpha)\,\left(1 + \pi\alpha^2\right)$$

Entering Two-Dimensional Input

When *Mathematica* reads the text `x^y`, it interprets it as `x` raised to the power `y`.

```
In[1]:= x^y
```

$$\text{Out[1]=}\quad x^y$$

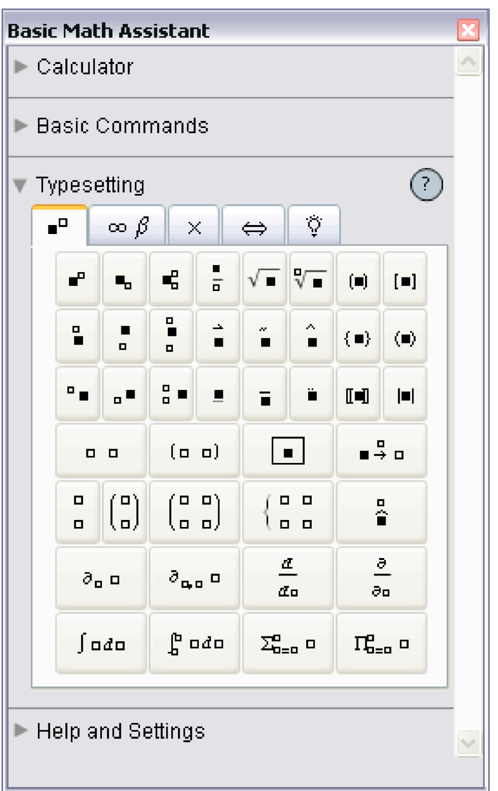
In a notebook, you can also give the two-dimensional input x^y directly. *Mathematica* again interprets this as a power.

$$\text{In[2]:=}\quad x^y$$

$$\text{Out[2]=}\quad x^y$$

One way to enter a two-dimensional form such as x^y into a *Mathematica* notebook is to paste this form into the notebook by clicking the appropriate button in the palette.

Here is a palette for entering some common two-dimensional notations.



There are also several ways to enter two-dimensional forms directly from the keyboard.

x Ctrl+^ *y* Ctrl+Space	use control keys that exist on most keyboards
x Ctrl+6 *y* Ctrl+Space	use control keys that should exist on all keyboards

Ways to enter a superscript directly from the keyboard.

You type Ctrl+^ by holding down the Control key, then pressing the ^ key. As soon as you do this, your cursor will jump to a superscript position. You can then type anything you want and it will appear in that position.

When you have finished, press Ctrl+Space to move back down from the superscript position. You type Ctrl+Space by holding down the Control key, then pressing the Space bar.

This sequence of keystrokes enters x^y.

In[3]:= **x Ctrl+^ y**

Out[3]= x^y

Here the whole expression y + z is in the superscript.

In[4]:= **x Ctrl+^ y + z**

Out[4]= x^{y+z}

Pressing Ctrl+Space takes you down from the superscript.

In[5]:= **x Ctrl+^ y Ctrl+Space + z**

Out[5]= $x^y + z$

You can remember the fact that Ctrl+^ gives you a superscript by thinking of Ctrl+^ as just a more immediate form of ^. When you type x^y, *Mathematica* will leave this one-dimensional form unchanged until you explicitly process it. But if you type x Ctrl+^ y then *Mathematica* will immediately give you a superscript.

On a standard English-language keyboard, the character ^ appears as the shifted version of 6. *Mathematica* therefore accepts Ctrl+6 as an alternative to Ctrl+^. Note that if you are using something other than a standard English-language keyboard, *Mathematica* will almost always accept Ctrl+6 but may not accept Ctrl+^.

x Ctrl+_ *y* Ctrl+Space	use control keys that exist on most keyboards
x Ctrl+- *y* Ctrl+Space	use control keys that should exist on all keyboards

Ways to enter a subscript directly from the keyboard.

Subscripts in *Mathematica* work very much like superscripts. However, whereas *Mathematica* automatically interprets x^y as x raised to the power y, it has no similar interpretation for x_y. Instead, it just treats x_y as a purely symbolic object.

This enters y as a subscript.

In[6]:= **x Ctrl+_ y**

Out[6]= x_y

Here is the usual one-dimensional *Mathematica* input that gives the same output expression.

In[7]:= **Subscript[x, y]**

Out[7]= x_y

x Ctrl+/ *y* Ctrl+Space	use control keys

How to enter a built-up fraction directly from the keyboard.

This enters the built-up fraction $\frac{x}{y}$.

In[8]:= **x Ctrl+/ y**

Out[8]= $\frac{x}{y}$

Here the whole y + z goes into the denominator.

In[9]:= **x Ctrl+/ y + z**

Out[9]= $\frac{x}{y+z}$

But pressing Ctrl+Space takes you out of the denominator, so the + z does not appear in the denominator.

In[10]:= **x Ctrl+/ y Ctrl+Space + z**

Out[10]= $\frac{x}{y} + z$

Mathematica automatically interprets a built-up fraction as a division.

In[11]:= $\frac{8888}{2222}$

Out[11]= 4

Ctrl+@ x Ctrl+Space	use control keys that exist on most keyboards
Ctrl+2 x Ctrl+Space	use control keys that should exist on all keyboards

Ways to enter a square root directly from the keyboard.

This enters a square root.

In[12]:= **Ctrl+@ x + y**

Out[12]= $\sqrt{x+y}$

Ctrl+Space takes you out of the square root.

In[13]:= **Ctrl+@ x Ctrl+Space + y**

Out[13]= $\sqrt{x}+y$

Here is the usual one-dimensional *Mathematica* input that gives the same output expression.

In[14]:= **Sqrt[x] + y**

Out[14]= $\sqrt{x}+y$

Ctrl+^ or Ctrl+6	go to the superscript position
Ctrl+_ or Ctrl+-	go to the subscript position
Ctrl+@ or Ctrl+2	go into a square root
Ctrl+% or Ctrl+5	go from subscript to superscript or vice versa, or to the exponent position in a root
Ctrl+/	go to the denominator for a fraction
Ctrl+Space	return from a special position

Special input forms based on control characters. The second forms given should work on any keyboard.

This puts both a subscript and a superscript on `x`.

In[15]:= **x Ctrl+^ y Ctrl+% z**

Out[15]= x_z^y

Here is another way to enter the same expression.

In[16]:= **x Ctrl+_ z Ctrl+% y**

Out[16]= x_z^y

The same procedure can be used to enter a definite integral.

In[17]:= **Esc** int **Esc Ctrl+_ 0 Ctrl+% 1 Ctrl+Space f[x] Esc** dd **Esc x**

Out[17]= $\int_0^1 f[x]\, dx$

In addition to subscripts and superscripts, *Mathematica* also supports the notion of underscripts and overscripts—elements that go directly underneath or above. Among other things, you can use underscripts and overscripts to enter the limits of sums and products.

x Ctrl+Plus *y* Ctrl+Space or *x* Ctrl+= *y* Ctrl+Space	create an underscript $\underset{y}{x}$
x Ctrl+& *y* Ctrl+Space or *x* Ctrl+7 *y* Ctrl+Space	create an overscript $\overset{y}{x}$

Creating underscripts and overscripts.

Here is a way to enter a summation.

In[18]:= **Esc** sum **Esc Ctrl+Plus x=0 Ctrl+% n Ctrl+Space f[x]**

Out[18]= $\sum_{x=0}^{n} f[x]$

Editing and Evaluating Two-Dimensional Expressions

When you see a two-dimensional expression on the screen, you can edit it much as you would edit text. You can for example place your cursor somewhere and start typing. Or you can select a part of the expression, then remove it using the Delete key, or insert a new version by typing it in.

In addition to ordinary text editing features, there are some keys that you can use to move around in two-dimensional expressions.

Ctrl+.	select the next larger subexpression
Ctrl+Space	move to the right of the current structure
→	move to the next character
←	move to the previous character

Ways to move around in two-dimensional expressions.

This shows the sequence of subexpressions selected by repeatedly typing Ctrl+..

$$-\frac{\text{ArcTan}\left[\frac{1+2x}{\sqrt{3}}\right]}{\sqrt{3}}+\frac{\text{Log}[-1+x]}{3}-\frac{\text{Log}[1+x+x^2]}{6}+\frac{\text{Log}[-1+x+x^2+x^3]}{9}$$

$$-\frac{\text{ArcTan}\left[\frac{1+2x}{\sqrt{3}}\right]}{\sqrt{3}}+\frac{\text{Log}[-1+x]}{3}-\frac{\text{Log}[1+x+x^2]}{6}+\frac{\text{Log}[-1+x+x^2+x^3]}{9}$$

$$-\frac{\text{ArcTan}\left[\frac{1+2x}{\sqrt{3}}\right]}{\sqrt{3}}+\frac{\text{Log}[-1+x]}{3}-\frac{\text{Log}[1+x+x^2]}{6}+\frac{\text{Log}[-1+x+x^2+x^3]}{9}$$

$$-\frac{\text{ArcTan}\left[\frac{1+2x}{\sqrt{3}}\right]}{\sqrt{3}}+\frac{\text{Log}[-1+x]}{3}-\frac{\text{Log}[1+x+x^2]}{6}+\frac{\text{Log}[-1+x+x^2+x^3]}{9}$$

$$-\frac{\text{ArcTan}\left[\frac{1+2x}{\sqrt{3}}\right]}{\sqrt{3}}+\frac{\text{Log}[-1+x]}{3}-\frac{\text{Log}[1+x+x^2]}{6}+\frac{\text{Log}[-1+x+x^2+x^3]}{9}$$

$$-\frac{\text{ArcTan}\left[\frac{1+2x}{\sqrt{3}}\right]}{\sqrt{3}}+\frac{\text{Log}[-1+x]}{3}-\frac{\text{Log}[1+x+x^2]}{6}+\frac{\text{Log}[-1+x+x^2+x^3]}{9}$$

Shift+Return	evaluate the whole current cell
Shift+Ctrl+Enter (Windows/Unix/Linux) or Cmd+Return (Mac OS X)	evaluate only the selected subexpression

Ways to evaluate two-dimensional expressions.

In most computations, you will want to go from one step to the next by taking the whole expression that you have generated, and then evaluating it. But if for example you are trying to manipulate a single formula to put it into a particular form, you may instead find it more convenient to perform a sequence of operations separately on different parts of the expression.

You do this by selecting each part you want to operate on, then inserting the operation you want to perform, then using Shift+Ctrl+Enter for Windows/Unix/Linux or Cmd+Return for Mac OS X.

Here is an expression with one part selected.

```
{Factor[x^4 - 1], Factor[x^5 - 1], Factor[x^6 - 1],
   Factor[x^7 - 1]}
```

Pressing Shift+Ctrl+Enter (Windows/Unix/Linux) or Cmd+Return (Mac OS X) evaluates the selected part.

```
{Factor[x^4 - 1], (-1 + x) (1 + x + x^2 + x^3 + x^4), Factor[x^6 - 1],
   Factor[x^7 - 1]}
```

The **Basic Commands ▸ y=x** tab in the **Basic Math Assistant**, **Classroom Assistant**, and **Writing Assistant** palettes also provides a number of convenient operations which will transform in place any selected subexpression.

Entering Formulas

character	short form	long form	symbol
π	Esc p Esc	\[Pi]	Pi
∞	Esc inf Esc	\[Infinity]	Infinity
°	Esc deg Esc	\[Degree]	Degree

Special forms for some common symbols.

This is equivalent to `Sin[60 Degree]`.

```
In[1]:= Sin[60 °]
```

Out[1]= $\frac{\sqrt{3}}{2}$

Here is the long form of the input.

```
In[2]:= Sin[60 °]
```

Out[2]= $\frac{\sqrt{3}}{2}$

You can enter the same input like this.

```
In[3]:= Sin[60 ⋮deg⋮]
```

Out[3]= $\frac{\sqrt{3}}{2}$

Here the angle is in radians.

In[4]:= **Sin**$\left[\frac{\pi}{3}\right]$

Out[4]= $\frac{\sqrt{3}}{2}$

special characters	*short form*	*long form*	*ordinary characters*
$x \leq y$	*x* Esc <= Esc *y*	*x* \[LessEqual] *y*	*x* <= *y*
$x \geq y$	*x* Esc >= Esc *y*	*x* \[GreaterEqual] *y*	*x* >= *y*
$x \neq y$	*x* Esc != Esc *y*	*x* \[NotEqual] *y*	*x* != *y*
$x \in y$	*x* Esc el Esc *y*	*x* \[Element] *y*	Element[*x*,*y*]
$x \to y$	*x* Esc -> Esc *y*	*x* \[Rule] *y*	*x* -> *y*

Special forms for a few operators. "Operator Input Forms" gives a complete list.

Here the replacement rule is entered using two ordinary characters, as ->.

```
In[5]:= x / (x + 1) /. x -> 3 + y
```

Out[5]= $\frac{3 + y}{4 + y}$

This means exactly the same.

```
In[6]:= x / (x + 1) /. x → 3 + y
```

Out[6]= $\frac{3 + y}{4 + y}$

As does this.

```
In[7]:= x/(x+1) /. x Esc->Esc 3 + y
```

Out[7]= $\frac{3 + y}{4 + y}$

When you type the ordinary-character form for certain operators, the front end automatically replaces them with the special-character form. For instance, when you type the last three examples, the front end automatically substitutes the → character for ->.

The special arrow form → is by default also used for output.

```
In[8]:= Solve[x^2 == 1, x]
Out[8]= {{x → -1}, {x → 1}}
```

special characters	*short form*	*long form*	*ordinary characters*
$x \div y$	*x* Esc div Esc *y*	*x* \[Divide] *y*	*x* / *y*
$x \times y$	*x* Esc * Esc *y*	*x* \[Times] *y*	*x* * *y*
$x \times y$	*x* Esc cross Esc *y*	*x* \[Cross] *y*	Cross[*x*,*y*]
$x == y$	*x* Esc == Esc *y*	*x* \[Equal] *y*	*x* == *y*
$x = y$	*x* Esc l = Esc *y*	*x* \[LongEqual] *y*	*x* == *y*
$x \wedge y$	*x* Esc && Esc *y*	*x* \[And] *y*	*x* && *y*
$x \vee y$	*x* Esc || Esc *y*	*x* \[Or] *y*	*x* || *y*
$\neg x$	Esc ! Esc *x*	\[Not] *x*	! *x*
$x \Rightarrow y$	*x* Esc => Esc *y*	*x* \[Implies] *y*	*x* => *y*
$x \cup y$	*x* Esc un Esc *y*	*x* \[Union] *y*	Union[*x*,*y*]
$x \cap y$	*x* Esc inter Esc *y*	*x* \[Intersection] *y*	Intersection[*x*,*y*]
xy	*x* Esc , Esc *y*	*x* \[InvisibleComma] *y*	*x* , *y*
fx	*f* Esc @ Esc *x*	*f* \[InvisibleApplication] *x*	*f* @ *x* or *f*[*x*]
$x\frac{y}{z}$	*x* Esc + Esc $\frac{y}{z}$	*x* \[ImplicitPlus] $\frac{y}{z}$	*x* + *y* / *z*

Some operators with special forms used for input but not output.

Mathematica understands ÷, but does not use it by default for output.

```
In[9]:= x ÷ y
```

```
Out[9]= x
        —
        y
```

Many of the forms of input discussed here use special characters, but otherwise just consist of ordinary one-dimensional lines of text. *Mathematica* notebooks, however, also make it possible to use two-dimensional forms of input.

two-dimensional	*one-dimensional*	
x^y	x^y	power
$\frac{x}{y}$	x/y	division
$\sqrt{x}$	`Sqrt`[x]	square root
$\sqrt[n]{x}$	x^(1/n)	n^{th} root
$\sum_{i=i_{min}}^{i_{max}} f$	`Sum`[f, {i, i_{min}, i_{max}}]	sum
$\prod_{i=i_{min}}^{i_{max}} f$	`Product`[f, {i, i_{min}, i_{max}}]	product
$\int f\, dx$	`Integrate`[f, x]	indefinite integral
$\int_{x_{min}}^{x_{max}} f\, dx$	`Integrate`[f, {x, x_{min}, x_{max}}]	definite integral
$\partial_x f$	`D`[f, x]	partial derivative
$\partial_{x,y} f$	`D`[f, x, y]	multivariate partial derivative
z^*	`Conjugate`[x]	complex conjugate
m^{T}	`Transpose`[m]	transpose
$m^{\dagger}$	`ConjugateTranspose`[m]	conjugate transpose
$expr_{[[i,j,\ldots]]}$	`Part`[$expr$, i, j, …]	part extraction

Some two-dimensional forms that can be used in *Mathematica* notebooks.

You can enter two-dimensional forms using any of the mechanisms discussed in "Entering Two-Dimensional Input". Note that upper and lower limits for sums and products must be entered as overscripts and underscripts—not superscripts and subscripts.

This enters an indefinite integral. Note the use of Esc dd Esc to enter the “differential d”.

In[10]:= **Esc** int **Esc** **f[x]** **Esc** dd **Esc** **x**

Out[10]= $\int f[x]\, dx$

Here is an indefinite integral that can be explicitly evaluated.

In[11]:= $\int \mathbf{Exp}[-x^2]\, dx$

Out[11]= $\frac{1}{2} \sqrt{\pi}\, \mathrm{Erf}[x]$

Here is the usual *Mathematica* input for this integral.

In[12]:=
```
Integrate[Exp[-x^2], x]
```

Out[12]= $\frac{1}{2}\sqrt{\pi}$ Erf[x]

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

Some special characters used in entering formulas. "Mathematical and Other Notation" gives a complete list.

You should realize that even though a summation sign can look almost identical to a capital sigma it is treated in a very different way by *Mathematica*. The point is that a sigma is just a letter; but a summation sign is an operator which tells *Mathematica* to perform a `Sum` operation.

Capital sigma is just a letter.

In[13]:=
```
a + Σ^2
```

Out[13]= $a+\Sigma^2$

A summation sign, on the other hand, is an operator.

In[14]:= **Esc** sum **Esc** **Ctrl+ +** **n=0** **Ctrl+ %** **m** **Ctrl+ Space** **1/f[n]**

Out[14]= $\sum_{n=0}^{m}\frac{1}{f[n]}$

Much as *Mathematica* distinguishes between a summation sign and a capital sigma, it also distinguishes between an ordinary d, the "partial d" ∂ that is used for taking derivatives, and the special "differential d" d that is used in the standard notation for integrals. It is crucial that

d

d d

you use the differential d—entered as Esc dd Esc—when you type in an integral. If you try to use an ordinary d, *Mathematica* will just interpret this as a symbol called d—it will not understand that you are entering the second part of an integration operator.

This computes the derivative of x^n.

```
In[15]:= ∂x x^n
Out[15]= n x^(-1+n)
```

Here is the same derivative specified in ordinary one-dimensional form.

```
In[16]:= D[x^n, x]
Out[16]= n x^(-1+n)
```

This computes the third derivative.

```
In[17]:= ∂x,x,x x^n
Out[17]= (-2 + n) (-1 + n) n x^(-3+n)
```

Here is the equivalent one-dimensional input form.

```
In[18]:= D[x^n, x, x, x]
Out[18]= (-2 + n) (-1 + n) n x^(-3+n)
```

Entering Tables and Matrices

The *Mathematica* front end provides an **Insert ▸ Table/Matrix** submenu for creating and editing arrays with any specified number of rows and columns. Once you have such an array, you can edit it to fill in whatever elements you want.

Mathematica treats an array like this as a matrix represented by a list of lists.

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

Putting parentheses around the array makes it look more like a matrix, but does not affect its interpretation.

```
        ( a  b  c )
In[2]:= ( 1  2  3 )
Out[2]= {{a, b, c}, {1, 2, 3}}
```

Using `MatrixForm` tells *Mathematica* to display the result of the `Transpose` as a matrix.

In[3]:= **MatrixForm[Transpose[$\begin{pmatrix} a & b & c \\ 1 & 2 & 3 \end{pmatrix}$]]**

Out[3]//MatrixForm= $\begin{pmatrix} a & 1 \\ b & 2 \\ c & 3 \end{pmatrix}$

Ctrl+,	add a column
Ctrl+Enter	add a row
Tab	go to the next □ or ■ element
Ctrl+Space	move out of the table or matrix

Entering tables and matrices.

Note that you can use Ctrl+, and Ctrl+Enter to start building up an array, and particularly for small arrays this is often more convenient than using the **New** menu item in the **Table/Matrix** submenu. The **Table/Matrix** menu items typically allow you to make basic adjustments, such as drawing lines between rows or columns.

Entering a `Piecewise` expression is a special case of entering a table.

Enter the `\[Piecewise]` character and press Ctrl+, to get a template of placeholders for two cases.

In[4]:= **f[x_] :=** $\begin{cases} \square & \square \\ \square & \square \end{cases}$

Fill in the placeholders to complete the piecewise expression.

In[5]:= **f[x_] :=** $\begin{cases} 0 & x < 0 \\ 1 & x = 0 \end{cases}$

To add additional cases, use Ctrl+Enter.

In[6]:= **f[x_] :=** $\begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \square & \square \end{cases}$

You can make an element in a table span over multiple rows or columns by selecting the entire block that you want the element to span and using the **Insert ▸ Table/Matrix ▸ Make Spanning** menu command. To split a spanning element into individual components, use **Insert ▸ Table/Matrix ▸ Split Spanning**.

To make the top element span across both columns, first select the row.

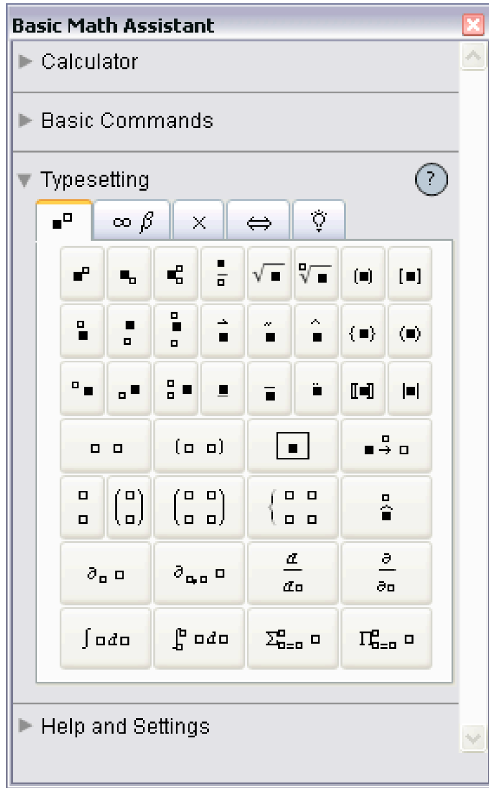
In[7]:=
```
x □
y z
```

Now use the **Make Spanning** menu command.

In[8]:=
```
 x
y z
```

Subscripts, Bars and Other Modifiers

Here is a typical palette of modifiers.



Mathematica allows you to use any expression as a subscript.

In[1]:= **Expand**$[(1 + x_{1+n})^4]$

Out[1]= $1 + 4\ x_{1+n} + 6\ x_{1+n}^2 + 4\ x_{1+n}^3 + x_{1+n}^4$

Unless you specifically tell it otherwise, *Mathematica* will interpret a superscript as a power.

In[2]:= **Factor**$[x_n^4 - 1]$

Out[2]= $(-1 + x_n)\ (1 + x_n)\ (1 + x_n^2)$

Ctrl+_ or Ctrl+-	go to the position for a subscript
Ctrl++ or Ctrl+=	go to the position underneath
Ctrl+^ or Ctrl+6	go to the position for a superscript
Ctrl+& or Ctrl+7	go to the position on top
Ctrl+Space	return from a special position

Special input forms based on control characters. The second forms given should work on any keyboard.

This enters a subscript using control keys.

In[3]:= **Expand[(1 + x Ctrl+_ 1+n Ctrl+Space)^4]**

Out[3]= $1 + 4\ x_{1+n} + 6\ x_{1+n}^2 + 4\ x_{1+n}^3 + x_{1+n}^4$

Just as Ctrl+^ and Ctrl+_ go to superscript and subscript positions, so also Ctrl+& and Ctrl+= can be used to go to positions directly above and below. With the layout of a standard English-language keyboard Ctrl+& is directly to the right of Ctrl+^ while Ctrl+= is directly to the right of Ctrl+_.

key sequence	*displayed form*	*expression form*
x Ctrl+& _	$\overline{x}$	OverBar[*x*]
x Ctrl+& Esc vec Esc	$\vec{x}$	OverVector[*x*]
x Ctrl+& ~	$\tilde{x}$	OverTilde[*x*]
x Ctrl+& ^	$\hat{x}$	OverHat[*x*]
x Ctrl+& .	$\dot{x}$	OverDot[*x*]
x Ctrl+= _	$\underline{x}$	UnderBar[*x*]

Ways to enter some common modifiers using control keys.

Here is $\overline{x}$.

In[4]:= **x Ctrl+& _ Ctrl+Space**

Out[4]= $\overline{x}$

You can use $\overline{x}$ as a variable.

In[5]:= **Solve[a^2 == %, a]**

Out[5]= $\{\{a \to -\sqrt{\overline{x}}\}, \{a \to \sqrt{\overline{x}}\}\}$

Non-English Characters and Keyboards

If you enter text in languages other than English, you will typically need to use various additional accented and other characters. If your computer system is set up in an appropriate way, then you will be able to enter such characters directly using standard keys on your keyboard. But however your system is set up, *Mathematica* always provides a uniform way to handle such characters.

	full name	*alias*		*full name*	*alias*
à	\[AGrave]	⋮a`⋮	ø	\[OSlash]	⋮o/⋮
å	\[ARing]	⋮ao⋮	ö	\[ODoubleDot]	⋮o"⋮
ä	\[ADoubleDot]	⋮a"⋮	ù	\[UGrave]	⋮u`⋮
ç	\[CCedilla]	⋮c,⋮	ü	\[UDoubleDot]	⋮u"⋮
č	\[CHacek]	⋮cv⋮	ß	\[SZ]	⋮sz⋮, ⋮ss⋮
é	\[EAcute]	⋮e'⋮	Å	\[CapitalARing]	⋮Ao⋮
è	\[EGrave]	⋮e`⋮	Ä	\[CapitalADoubleDot]	⋮A"⋮
í	\[IAcute]	⋮i'⋮	Ö	\[CapitalODoubleDot]	⋮O"⋮
ñ	\[NTilde]	⋮n~⋮	Ü	\[CapitalUDoubleDot]	⋮U"⋮
ò	\[OGrave]	⋮o`⋮			

Some common European characters.

Here is a function whose name involves an accented character.

```
In[1]:= Lamé[x, y]
Out[1]= Lamé[x, y]
```

This is another way to enter the same input.

```
In[2]:= Lam⋮e'⋮[x, y]
Out[2]= Lamé[x, y]
```

You should realize that there is no uniform standard for computer keyboards around the world, and as a result it is inevitable that some details of what has been said in this tutorial may not apply to your keyboard.

In particular, the identification for example of Ctrl+6 with Ctrl+^ is valid only for keyboards on which ^ appears as Shift+6. On other keyboards, *Mathematica* uses Ctrl+6 to go to a superscript position, but not necessarily Ctrl+^.

Regardless of how your keyboard is set up you can always use palettes or menu items to set up superscripts and other kinds of notation. And assuming you have some way to enter characters such as \, you can always give input using full names such as `\[Infinity]`.

Other Mathematical Notation

Mathematica supports an extremely wide range of mathematical notation, although often it does not assign a pre-defined meaning to it. Thus, for example, you can enter an expression such as x ⊕ y, but *Mathematica* will not initially make any assumption about what you mean by ⊕.

Mathematica knows that ⊕ is an operator, but it does not initially assign any specific meaning to it.

```
In[1]:= {17 ⊕ 5, 8 ⊕ 3}
Out[1]= {17⊕5, 8⊕3}
```

This gives *Mathematica* a definition for what the ⊕ operator does.

```
In[2]:= x_ ⊕ y_ := Mod[x + y, 2]
```

Now *Mathematica* can evaluate ⊕ operations.

```
In[3]:= {17 ⊕ 5, 8 ⊕ 3}
Out[3]= {0, 1}
```

	full name	*alias*		*full name*	*alias*
⊕	\[CirclePlus]	⋮c+⋮	⟶	\[LongRightArrow]	⋮-->⋮
⊗	\[CircleTimes]	⋮c*⋮	⟷	\[LeftRightArrow]	⋮<->⋮
±	\[PlusMinus]	⋮+-⋮	↑	\[UpArrow]	
∧	\[Wedge]	⋮^⋮	⇌	\[Equilibrium]	⋮equi⋮
∨	\[Vee]	⋮v⋮	⊢	\[RightTee]	
≃	\[TildeEqual]	⋮~=⋮	⊃	\[Superset]	⋮sup⋮
≈	\[TildeTilde]	⋮~~⋮	⊓	\[SquareIntersection]	
∼	\[Tilde]	⋮~⋮	∈	\[Element]	⋮elem⋮
∝	\[Proportional]	⋮prop⋮	∉	\[NotElement]	⋮!elem⋮
≡	\[Congruent]	⋮===⋮	∘	\[SmallCircle]	⋮sc⋮
≳	\[GreaterTilde]	⋮>~⋮	∴	\[Therefore]	
≫	\[GreaterGreater]		|	\[VerticalSeparator]	⋮|⋮
≻	\[Succeeds]		∣	\[VerticalBar]	⋮_|⋮
▷	\[RightTriangle]		∖	\[Backslash]	⋮\⋮

A few of the operators whose input is supported by *Mathematica*.

Mathematica assigns built-in meanings to ≥ and ⩾, but not to ≳ or ≫.

```
In[4]:= {3 ≥ 4, 3 ⩾ 4, 3 ≳ 4, 3 ≫ 4}
Out[4]= {False, False, 3 ≳ 4, 3 ≫ 4}
```

There are some forms which look like characters on a standard keyboard, but which are interpreted in a different way by *Mathematica*. Thus, for example, \[Backslash] or ⋮\⋮ displays as ∖ but is not interpreted in the same way as a \ typed directly on the keyboard.

The ∖ and ∧ characters used here are different from the \ and ^ you would type directly on a keyboard.

```
In[5]:= {a ⋮\⋮ b, a ⋮^⋮ b}
Out[5]= {a ∖ b, a ∧ b}
```

Most operators work like ⊕ and go in between their operands. But some operators can go in other places. Thus, for example, ⋮<⋮ and ⋮>⋮ or \[LeftAngleBracket] and \[RightAngleBracket] are effectively operators which go around their operand.

The elements of the angle bracket operator go around their operand.

```
In[6]:= 〈 1 + x 〉
Out[6]= 〈1 + x〉
```

	full name	alias		full name	alias
ℓ	\[ScriptL]	⋮scl⋮	Å	\[Angstrom]	⋮Ang⋮
ℰ	\[ScriptCapitalE]	⋮scE⋮	ℏ	\[HBar]	⋮hb⋮
ℜ	\[GothicCapitalR]	⋮goR⋮	£	\[Sterling]	
ℤ	\[DoubleStruckCapitalZ]	⋮dsZ⋮	∠	\[Angle]	
ℵ	\[Aleph]	⋮al⋮	•	\[Bullet]	⋮bu⋮
∅	\[EmptySet]	⋮es⋮	†	\[Dagger]	⋮dg⋮
µ	\[Micro]	⋮mi⋮	♮	\[Natural]	

Some additional letters and letter-like forms.

You can use letters and letter-like forms anywhere in symbol names.

```
In[7]:= {ℜ∅, ∠ABC}
Out[7]= {ℜ∅, ∠ABC}
```

∅ is assumed to be a symbol, and so is just multiplied by a and b.

```
In[8]:= a ∅ b
Out[8]= a b ∅
```

Forms of Input and Output

Here is one way to enter a particular expression.

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

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

Here is another way to enter the same expression.

```
In[2]:= Plus[Power[x, 2], Sqrt[y]]
```

Out[2]= $x^2 + \sqrt{y}$

With a notebook front end, you can also enter the expression directly in this way.

In[3]:= $x^2 + \sqrt{y}$

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

Mathematica allows you to output expressions in many different ways.

In *Mathematica* notebooks, expressions are by default output in `StandardForm`.

```
In[4]:= x^2 + Sqrt[y]
```

Out[4]= $x^2 + \sqrt{y}$

`OutputForm` uses only ordinary keyboard characters and is the default for text-based interfaces to *Mathematica*.

```
In[5]:= OutputForm[x^2 + Sqrt[y]]

Out[5]//OutputForm=  2
                    x  + Sqrt[y]
```

`InputForm` yields a form that can be typed directly on a keyboard.

```
In[6]:= InputForm[x^2 + Sqrt[y]]

Out[6]//InputForm= x^2 + Sqrt[y]
```

`FullForm` shows the internal form of an expression in explicit functional notation.

```
In[7]:= FullForm[x^2 + Sqrt[y]]

Out[7]//FullForm= Plus[Power[x, 2], Power[y, Rational[1, 2]]]
```

`FullForm[`*expr*`]`	the internal form of an expression
`InputForm[`*expr*`]`	a form suitable for direct keyboard input
`OutputForm[`*expr*`]`	a two-dimensional form using only keyboard characters
`StandardForm[`*expr*`]`	the default form used in *Mathematica* notebooks

Some output forms for expressions.

Output forms provide textual representations of *Mathematica* expressions. In some cases these textual representations are also suitable for input to *Mathematica*. But in other cases they are intended just to be looked at, or to be exported to other programs, rather than to be used as input to *Mathematica*.

`TraditionalForm` uses a large collection of ad hoc rules to produce an approximation to traditional mathematical notation.

```
In[8]:= TraditionalForm[x^2 + Sqrt[y] + Gamma[z] EllipticK[z]]
```

Out[8]//TraditionalForm=

$x^2 + K(z)\,\Gamma(z) + \sqrt{y}$

`TeXForm` yields output suitable for export to TeX.

```
In[9]:= TeXForm[x^2 + Sqrt[y]]
Out[9]//TeXForm= x^2+\sqrt{y}
```

`MathMLForm` yields output in MathML.

```
In[10]:= MathMLForm[x^2 + Sqrt[y]]
Out[10]//MathMLForm=
<math>
 <mrow>
  <msup>
   <mi>x</mi>
   <mn>2</mn>
  </msup>
  <mo>+</mo>
  <msqrt>
   <mi>y</mi>
  </msqrt>
 </mrow>
</math>
```

`CForm` yields output that can be included in a C program. Macros for objects like `Power` are included in the header file `mdefs.h`.

```
In[11]:= CForm[x^2 + Sqrt[y]]
Out[11]//CForm= Power(x,2) + Sqrt(y)
```

`FortranForm` yields output suitable for export to Fortran.

```
In[12]:= FortranForm[x^2 + Sqrt[y]]
Out[12]//FortranForm=
x**2 + Sqrt(y)
```

`TraditionalForm[`*expr*`]`	traditional mathematical notation
`TeXForm[`*expr*`]`	output suitable for export to TEX
`MathMLForm[`*expr*`]`	output suitable for use with MathML on the web
`CForm[`*expr*`]`	output suitable for export to C
`FortranForm[`*expr*`]`	output suitable for export to Fortran

Output forms not normally used for *Mathematica* input.

"Low-Level Input and Output Rules" discusses how you can create your own output forms. You should realize however that in communicating with external programs it is often better to use *MathLink* to send expressions directly than to generate a textual representation for these expressions.

- Exchange textual representations of expressions.
- Exchange expressions directly via *MathLink*.

Two ways to communicate between *Mathematica* and other programs.

Mixing Text and Formulas

The simplest way to mix text and formulas in a *Mathematica* notebook is to put each kind of material in a separate cell. Sometimes, however, you may want to embed a formula within a cell of text, or vice versa.

Ctrl+(or Ctrl+9	begin entering a formula within text, or text within a formula
Ctrl+) or Ctrl+0	end entering a formula within text, or text within a formula

Entering a formula within text, or vice versa.

Here is a notebook with formulas embedded in a text cell.

This is a text cell, but it can contain formulas such as $\int \frac{1}{x^3-1}\, dx$ or $-\frac{\log(x^2+x+1)}{6} - \frac{\tan^{-1}\left(\frac{2x+1}{\sqrt{3}}\right)}{\sqrt{3}} + \frac{\log(x-1)}{3}$. The formulas flow with text.

Mathematica notebooks often contain both formulas that are intended for actual evaluation by *Mathematica*, and ones that are intended just to be read in a more passive way.

When you insert a formula in text, you can use the **Convert to StandardForm** and **Convert to TraditionalForm** menu items within the formula to convert it to `StandardForm` or `TraditionalForm`. `StandardForm` is normally appropriate whenever the formula is thought of as a *Mathematica* program fragment.

In general, however, you can use exactly the same mechanisms for entering formulas, whether or not they will ultimately be given as *Mathematica* input.

You should realize, however, that to make the detailed typography of typical formulas look as good as possible, *Mathematica* automatically does things such as inserting spaces around certain operators. But these kinds of adjustments can potentially be inappropriate if you use notation in very different ways from the ones *Mathematica* is expecting. In such cases, you may have to make detailed typographical adjustments by hand.

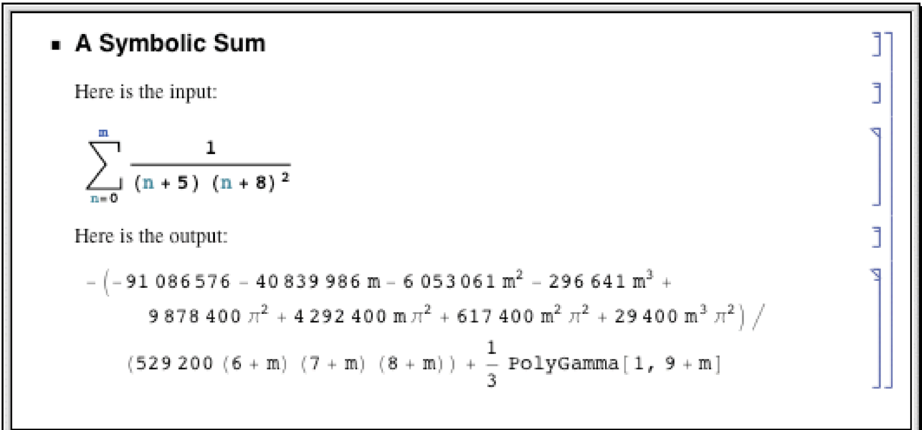
Displaying and Printing *Mathematica* Notebooks

Depending on the purpose for which you are using a *Mathematica* notebook, you may want to change its overall appearance. The front end allows you to specify independently the styles to be used for display on the screen and for printing. Typically you can do this by choosing appropri ate items in the **Format** menu.

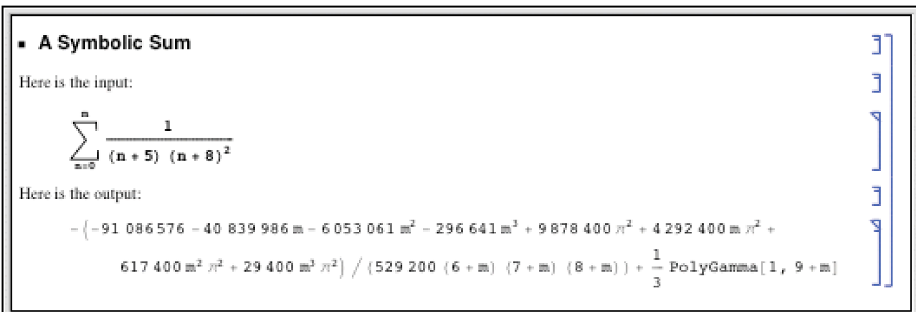
`ScreenStyleEnvironment`	styles to be used for screen display
`PrintingStyleEnvironment`	styles to be used for printed output
`Working`	standard style definitions for screen display
`Presentation`	style definitions for presentations
`SlideShow`	style definitions for displaying presentation slides
`Printout`	style definitions for printed output

Front end settings that define the global appearance of a notebook.

Here is a typical notebook as it appears in working form on the screen.



Here is a preview of how the notebook would appear when printed out.



Setting Up Hyperlinks

Insert ▸ Hyperlink	menu item to make the selected object a hyperlink
`Hyperlink["`*uri*`"]`	generate as output a hyperlink with the label and destination set as *uri*
`Hyperlink["`*label*`","`*uri*`"]`	generate as output a hyperlink with the label *label* and the destination *uri*
`Hyperlink[{"`*file*`.nb",None}]`	generate as output a hyperlink to the specified notebook
`Hyperlink[{"`*file*`.nb","`*tag*`"}]`	generate as output a hyperlink to the cell tagged as *tag* in the specified notebook

Methods for generating hyperlinks.

A hyperlink is a special kind of button which jumps to another part of a notebook when it is pressed. Typically hyperlinks are indicated in *Mathematica* by blue text.

To set up a hyperlink, just select the text or other object that you want to be a hyperlink. Then choose the menu item **Insert ▸ Hyperlink** and fill in the specification of where you want the destination of the hyperlink to be.

The destination of a hyperlink can be any standard web address (URI). Hyperlinks can also point to notebooks on the local file system, or even to specific cells inside those notebooks. Hyperlinks which point to specific cells in notebooks use cell tags to identify the cells. If a particular cell tag is used for more than one cell in a given notebook, then the hyperlink will go to the first instance of a cell with that cell tag.

A hyperlink can be generated in output by using the *Mathematica* command `Hyperlink`. These hyperlinks can be copied and pasted into text or used in a larger interface being generated by *Mathematica*.

This command generates a hyperlink to the web.

```
In[1]:= Hyperlink["Wolfram Research, Inc.", "http://www.wolfram.com"]
Out[1]= Wolfram Research, Inc.
```

Automatic Numbering

- Choose a cell style such as `DisplayFormulaNumbered`.
- Use the **Insert ▶ Automatic Numbering** menu item, with a counter name such as `Section`.

Two ways to set up automatic numbering in a *Mathematica* notebook.

Using the DisplayFormulaNumbered style

These cells are in `DisplayFormulaNumbered` style. `DisplayFormulaNumbered` style is available in stylesheets such as `"Report"`.

$$\int \frac{x}{x+1} \, dx \qquad (1)$$

$$\int \frac{\mathrm{Sin}[x]}{x+1} \, dx \qquad (2)$$

$$\int \frac{\mathrm{Log}[x] + \mathrm{Exp}[x]}{x+1} \, dx \qquad (3)$$

Using the AutomaticNumbering menu item

The input for each cell here is exactly the same, but the cells contain an element that displays as a progressively larger number as one goes through the notebook.

- **1. A Section**
- **2. A Section**
- **3. A Section**

Exposition in *Mathematica* Notebooks

Mathematica notebooks provide the basic technology that you need to be able to create a very wide range of sophisticated interactive documents. But to get the best out of this technology you need to develop an appropriate style of exposition.

Many people at first tend to use *Mathematica* notebooks either as simple worksheets containing a sequence of input and output lines, or as onscreen versions of traditional books and other printed material. But the most effective and productive uses of *Mathematica* notebooks tend to lie at neither one of these extremes, and instead typically involve a fine-grained mixing of *Mathematica* input and output with explanatory text. In most cases the single most important factor in obtaining such fine-grained mixing is uniform use of the *Mathematica* language.

One might think that there would tend to be four kinds of material in a *Mathematica* notebook: plain text, mathematical formulas, computer code, and interactive interfaces. But one of the key ideas of *Mathematica* is to provide a single language that offers the best of both traditional mathematical formulas and computer code.

In `StandardForm`, *Mathematica* expressions have the same kind of compactness and elegance as traditional mathematical formulas. But unlike such formulas, *Mathematica* expressions are set up in a completely consistent and uniform way. As a result, if you use *Mathematica* expressions, then regardless of your subject matter, you never have to go back and reexplain your basic notation: it is always just the notation of the *Mathematica* language. In addition, if you set up your explanations in terms of *Mathematica* expressions, then a reader of your notebook can immediately take what you have given, and actually execute it as *Mathematica* input.

If one has spent many years working with traditional mathematical notation, then it takes a little time to get used to seeing mathematical facts presented as `StandardForm` *Mathematica* expressions. Indeed, at first one often has a tendency to try to use `TraditionalForm` whenever possible, perhaps with hidden tags to indicate its interpretation. But quite soon one tends to evolve to a mixture of `StandardForm` and `TraditionalForm`. And in the end it becomes clear that `StandardForm` alone is for most purposes the most effective form of presentation.

In traditional mathematical exposition, there are many tricks for replacing chunks of text by fragments of formulas. In `StandardForm` many of these same tricks can be used. But the fact

that *Mathematica* expressions can represent not only mathematical objects but also procedures, algorithms, graphics, and interfaces increases greatly the extent to which chunks of text can be replaced by shorter and more precise material.

Named Characters

Mathematica provides systemwide support for a large number of special characters. Each character has a name and a number of shortcut aliases. They are fully supported by the standard *Mathematica* fonts.

Interpretation of Characters

The interpretations given here are those used in `StandardForm` and `InputForm`. Most of the interpretations also work in `TraditionalForm`.

You can override the interpretations by giving your own rules for `MakeExpression`.

Letters and letter-like forms	used in symbol names
Infix operators	e.g. $x \oplus y$
Prefix operators	e.g. $\neg x$
Postfix operators	e.g. $x!$
Matchfix operators	e.g. $\langle x \rangle$
Compound operators	e.g. $\int f \, dx$
Raw operators	operator characters that can be typed on an ordinary keyboard
Spacing characters	interpreted in the same way as an ordinary space
Structural elements	characters used to specify structure; usually ignored in interpretation
Uninterpretable elements	characters indicating missing information

Types of characters.

The precedences of operators are given in "Operator Input Forms".

Infix operators for which no grouping is specified in the listing are interpreted so that for example $x \oplus y \oplus z$ becomes `CirclePlus[x, y, z]`.

Naming Conventions

Characters that correspond to built-in *Mathematica* functions typically have names corresponding to those functions. Other characters typically have names that are as generic as possible.

Characters with different names almost always look at least slightly different.

`\[Capital...]`	uppercase form of a letter
`\[Left...]` and `\[Right...]`	pieces of a matchfix operator (also arrows)
`\[Raw...]`	a printable ASCII character
`\[...Indicator]`	a visual representation of a keyboard character

Some special classes of characters.

style	`Script`, `Gothic`, etc.
variation	`Curly`, `Gray`, etc.
case	`Capital`, etc.
modifiers	`Not`, `Double`, `Nested`, etc.
direction	`Left`, `Up`, `UpperRight`, etc.
base	`A`, `Epsilon`, `Plus`, etc.
diacritical mark	`Acute`, `Ring`, etc.

Typical ordering of elements in character names.

Aliases

Mathematica supports both its own system of aliases, as well as aliases based on character names in TeX and SGML or HTML. Except where they conflict, character names corresponding to plain TeX, LaTeX and AMSTeX are all supported. Note that TeX and SGML or HTML aliases are not given explicitly in the list of characters below.

Esc `xxx` Esc	ordinary *Mathematica* alias
Esc `\xxx` Esc	TeX alias
Esc `&xxx` Esc	SGML or HTML alias

Types of aliases.

The following general conventions are used for all aliases:

- Characters that are alternatives to standard keyboard operators use these operators as their aliases (e.g. Esc -> Esc for →, Esc && Esc for ∧).
- Most single-letter aliases stand for Greek letters.
- Capital-letter characters have aliases beginning with capital letters.
- When there is ambiguity in the assignment of aliases, a space is inserted at the beginning of the alias for the less common character (e.g. Esc -> Esc for \[Rule] and Esc ␣-> Esc for \[RightArrow]).
- ! is inserted at the beginning of the alias for a Not character.
- TeX aliases begin with a backslash \.
- SGML aliases begin with an ampersand &.
- User-defined aliases conventionally begin with a dot or comma.

Font Matching

The special fonts provided with *Mathematica* include all the characters given in this listing. Some of these characters also appear in certain ordinary text fonts.

When rendering text in a particular font, the *Mathematica* notebook front end will use all the characters available in that font. It will use the special *Mathematica* fonts only for other characters.

A choice is made between Times-like, Helvetica-like (sans serif) and Courier-like (monospaced) variants to achieve the best matching with the ordinary text font in use.

Textual Input and Output

How Input and Output Work

Input	convert from a textual form to an expression
Processing	do computations on the expression
Output	convert the resulting expression to textual form

Steps in the operation of *Mathematica*.

When you type something like `x^2` what *Mathematica* at first sees is just the string of characters `x`, `^`, `2`. But with the usual way that *Mathematica* is set up, it immediately knows to convert this string of characters into the expression `Power[x, 2]`.

Then, after whatever processing is possible has been done, *Mathematica* takes the expression `Power[x, 2]` and converts it into some kind of textual representation for output.

Mathematica reads the string of characters `x`, `^`, `2` and converts it to the expression `Power[x, 2]`.

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

Out[1]= x^2

This shows the expression in Fortran form.

```
In[2]:= FortranForm[%]

Out[2]//FortranForm= x**2
```

`FortranForm` is just a "wrapper": the value of `Out[2]` is still the expression `Power[x, 2]`.

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

Out[3]= x^2

It is important to understand that in a typical *Mathematica* session `In[n]` and `Out[n]` record only the underlying expressions that are processed, not the textual representations that happen to be used for their input or output.

If you explicitly request a particular kind of output, say by using `TraditionalForm[`*expr*`]`, then what you get will be labeled with `Out[`*n*`] // TraditionalForm`. This indicates that what you are seeing is *expr* `// TraditionalForm`, even though the value of `Out[`*n*`]` itself is just *expr*.

Mathematica also allows you to specify globally that you want output to be displayed in a particular form. And if you do this, then the form will no longer be indicated explicitly in the label for each line. But it is still the case that `In[`*n*`]` and `Out[`*n*`]` will record only underlying expressions, not the textual representations used for their input and output.

This sets `t` to be an expression with `FortranForm` explicitly wrapped around it.

```
In[4]:= t = FortranForm[x^2 + y^2]
Out[4]//FortranForm= x**2 + y**2
```

The result on the previous line is just the expression.

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

Out[5]= $x^2 + y^2$

But `t` contains the `FortranForm` wrapper, and so is displayed in `FortranForm`.

```
In[6]:= t
Out[6]//FortranForm= x**2 + y**2
```

Wherever `t` appears, it is formatted in `FortranForm`.

```
In[7]:= {t^2, 1 / t}
```

Out[7]= $\left\{\text{x ** 2 + y ** 2}^2,\ \frac{1}{\text{x ** 2 + y ** 2}}\right\}$

The Representation of Textual Forms

Like everything else in *Mathematica* the textual forms of expressions can themselves be represented as expressions. Textual forms that consist of one-dimensional sequences of characters can be represented directly as ordinary *Mathematica* strings. Textual forms that involve subscripts, superscripts and other two-dimensional constructs, however, can be represented by nested collections of two-dimensional boxes.

One-dimensional strings	`InputForm`, `FullForm`, etc.
Two-dimensional boxes	`StandardForm`, `TraditionalForm`, etc.

Typical representations of textual forms.

This generates the string corresponding to the textual representation of the expression in `InputForm`.

```
In[1]:= ToString[x^2 + y^3, InputForm]
Out[1]= x^2 + y^3
```

`FullForm` shows the string explicitly.

```
In[2]:= FullForm[%]
Out[2]//FullForm= "x^2 + y^3"
```

Here are the individual characters in the string.

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

Here is the box structure corresponding to the expression in `StandardForm`.

```
In[4]:= ToBoxes[x^2 + y^3, StandardForm]
Out[4]= RowBox[{SuperscriptBox[x, 2], +, SuperscriptBox[y, 3]}]
```

Here is the `InputForm` of the box structure. In this form the structure is effectively represented by an ordinary string.

```
In[5]:= ToBoxes[x^2 + y^3, StandardForm] // InputForm
Out[5]//InputForm= \(x\^2 + y\^3\)
```

If you use the notebook front end for *Mathematica*, then you can see the expression that corresponds to the textual form of each cell by using the **Show Expression** menu item.

Here is a cell containing an expression in `StandardForm`.

$$\frac{1}{2\left(1+x^2\right)} + \mathrm{Log}[x] - \frac{\mathrm{Log}\left[1+x^2\right]}{2}$$

Here is the underlying representation of that expression in terms of boxes, displayed using the **Show Expression** menu item.

```
Cell[BoxData[
 RowBox[{
  FractionBox["1",
   RowBox[{"2",
    RowBox[{"(",
     RowBox[{"1", "+",
      SuperscriptBox["x", "2"]}], ")"}]}]], "+",
  RowBox[{"Log", "[", "x", "]"}], "-",
  FractionBox[
   RowBox[{"Log", "[",
    RowBox[{"1", "+",
     SuperscriptBox["x", "2"]}], "]"}], "2"]}]], "Input"]
```

`ToString[`*expr*`,`*form*`]`	create a string representing the specified textual form of *expr*
`ToBoxes[`*expr*`,`*form*`]`	create a box structure representing the specified textual form of *expr*

Creating strings and boxes from expressions.

The Interpretation of Textual Forms

`ToExpression[`*input*`]`	create an expression by interpreting strings or boxes

Converting from strings or boxes to expressions.

This takes a string and interprets it as an expression.

In[1]:= **ToExpression["2 + 3 + x/y"]**

Out[1]= $5+\frac{x}{y}$

Here is the box structure corresponding to the textual form of an expression in `StandardForm`.

In[2]:= **ToBoxes[2 + x^2, StandardForm]**

Out[2]= RowBox[{2, +, SuperscriptBox[x, 2]}]

`ToExpression` interprets this box structure and yields the original expression again.

In[3]:= **ToExpression[%]**

Out[3]= $2+x^2$

In any *Mathematica* session, *Mathematica* is always effectively using `ToExpression` to interpret the textual form of your input as an actual expression to evaluate.

If you use the notebook front end for *Mathematica*, then the interpretation only takes place when the contents of a cell are sent to the kernel, say for evaluation. This means that within a notebook there is no need for the textual forms you set up to correspond to meaningful *Mathematica* expressions; this is only necessary if you want to send these forms to the kernel.

`FullForm`	explicit functional notation
`InputForm`	one-dimensional notation
`StandardForm`	two-dimensional notation

The hierarchy of forms for standard *Mathematica* input.

Here is an expression entered in `FullForm`.

```
In[4]:= Plus[1, Power[x, 2]]
Out[4]= 1 + x^2
```

Here is the same expression entered in `InputForm`.

```
In[5]:= 1 + x^2
Out[5]= 1 + x^2
```

And here is the expression entered in `StandardForm`.

In[6]:= $1 + x^2$

Out[6]= $1 + x^2$

Built into *Mathematica* is a collection of standard rules for use by `ToExpression` in converting textual forms to expressions.

These rules define the *grammar* of *Mathematica*. They state, for example, that $x + y$ should be interpreted as `Plus[x, y]`, and that x^y should be interpreted as `Power[x, y]`. If the input you give is in `FullForm`, then the rules for interpretation are very straightforward: every expression consists just of a head followed by a sequence of elements enclosed in brackets. The rules for `InputForm` are slightly more sophisticated: they allow operators such as `+`, `=`, and `->`, and understand the meaning of expressions where these operators appear between operands. `StandardForm` involves still more sophisticated rules, which allow operators and operands to be arranged not just in a one-dimensional sequence, but in a full two-dimensional structure.

Mathematica is set up so that `FullForm`, `InputForm` and `StandardForm` form a strict hierarchy: anything you can enter in `FullForm` will also work in `InputForm`, and anything you can enter in `InputForm` will also work in `StandardForm`.

If you use a notebook front end for *Mathematica*, then you will typically want to use all the features of `StandardForm`. If you use a text-based interface, however, then you will typically be able to use only features of `InputForm`.

When you use `StandardForm` in a *Mathematica* notebook, you can enter directly two-dimensional forms such as x^2 or annotated graphics. But `InputForm` allows only one-dimensional forms.

If you copy a `StandardForm` expression whose interpretation can be determined without evaluation, then the expression will be pasted into external applications as `InputForm`. Otherwise, the text is copied in a linear form that precisely represents the two-dimensional structure using `\!\(...\)`. When you paste this linear form back into a *Mathematica* notebook, it will automatically "snap" into two-dimensional form.

`ToExpression[`*input*`,`*form*`]`	attempt to create an expression assuming that *input* is given in the specified textual form

Importing from other textual forms.

`StandardForm` and its subsets `FullForm` and `InputForm` provide precise ways to represent any *Mathematica* expression in textual form. And given such a textual form, it is always possible to convert it unambiguously to the expression it represents.

`TraditionalForm` is an example of a textual form intended primarily for output. It is possible to take any *Mathematica* expression and display it in `TraditionalForm`. But `TraditionalForm` does not have the precision of `StandardForm`, and as a result there is in general no unambiguous way to go back from a `TraditionalForm` representation and get the expression it represents.

Nevertheless, `ToExpression[`*input*`, TraditionalForm]` takes text in `TraditionalForm` and attempts to interpret it as an expression.

This takes a string and interprets it as `TraditionalForm` input.

```
In[7]:= ToExpression["f(6)", TraditionalForm]
Out[7]= f[6]
```

In `StandardForm` the same string would mean a product of terms.

```
In[8]:= ToExpression["f(6)", StandardForm]
Out[8]= 6 f
```

When `TraditionalForm` output is generated as the result of a computation, the actual collection of boxes that represent the output typically contains special `Interpretation` objects or other specially tagged forms which specify how an expression can be reconstructed from the `TraditionalForm` output.

The same is true of `TraditionalForm` that is obtained by explicit conversion from `StandardForm`. But if you edit `TraditionalForm` extensively, or enter it from scratch, then *Mathematica* will have to try to interpret it without the benefit of any additional embedded information.

Short and Shallow Output

When you generate a very large output expression in *Mathematica*, you often do not want to see the whole expression at once. Rather, you would first like to get an idea of the general structure of the expression, and then, perhaps, go in and look at particular parts in more detail.

The functions `Short` and `Shallow` allow you to see "outlines" of large *Mathematica* expressions.

`Short[`*expr*`]`	show a one-line outline of *expr*
`Short[`*expr*`,`*n*`]`	show an *n*-line outline of *expr*
`Shallow[`*expr*`]`	show the "top parts" of *expr*
`Shallow[`*expr*`,{`*depth*`,`*length*`}]`	show the parts of *expr* to the specified depth and length

Showing outlines of expressions.

This generates a long expression. If the whole expression were printed out here, it would go on for 23 lines.

```
In[1]:= t = Expand[(1 + x + y)^12];
```

This gives a one-line "outline" of `t`. The `<<  >>` indicates the number of terms omitted.

```
In[2]:= Short[t]
```

Out[2]//Short= $1 + 12\,x + 66\,x^2 + 220\,x^3 + 495\,x^4 + \ll 81 \gg + 132\,x\,y^{10} + 66\,x^2\,y^{10} + 12\,y^{11} + 12\,x\,y^{11} + y^{12}$

When *Mathematica* generates output in a textual format such as `OutputForm`, it first effectively writes the output in one long row. Then it looks at the width of text you have asked for, and it chops the row of output into a sequence of separate "lines". Each of the "lines" may of course contain superscripts and built-up fractions, and so may take up more than one actual line on your output device. When you specify a particular number of lines in `Short`, *Mathematica* takes this to be the number of "logical lines" that you want, not the number of actual physical lines on your particular output device.

Here is a four-line version of `t`. More terms are shown in this case.

```
In[3]:= Short[t, 4]
```

Out[3]//Short= $1 + 12\,x + 66\,x^2 + 220\,x^3 + 495\,x^4 + 792\,x^5 + 924\,x^6 + 792\,x^7 + 495\,x^8 + 220\,x^9 + 66\,x^{10} + 12\,x^{11} + x^{12} + 12\,y + 132\,x\,y + \ll 61 \gg + 495\,y^8 + 1980\,x\,y^8 + 2970\,x^2\,y^8 + 1980\,x^3\,y^8 + 495\,x^4\,y^8 + 220\,y^9 + 660\,x\,y^9 + 660\,x^2\,y^9 + 220\,x^3\,y^9 + 66\,y^{10} + 132\,x\,y^{10} + 66\,x^2\,y^{10} + 12\,y^{11} + 12\,x\,y^{11} + y^{12}$

`Short` works in other formats too, such as `StandardForm` and `TraditionalForm`. When using these formats, linewrapping is determined by the notebook interface when displaying the output rather than by the kernel when creating the output. As a result, setting the number of lines generated by `Short` can only approximate the actual number of lines displayed onscreen.

You can use `Short` with other output forms, such as `InputForm`.

```
In[4]:= Short[InputForm[t]]

Out[4]//Short= 1 + 12*x + 66*x^2 + 220*x^3 + 495*x^4 + <<83>> + 12*y^11 + 12*x*y^11 + y^12
```

`Short` works by removing a sequence of parts from an expression until the output form of the result fits on the number of lines you specify. Sometimes, however, you may find it better to specify not how many final output lines you want, but which parts of the expression to drop. `Shallow[`*expr*`, {`*depth*`,` *length*`}]` includes only *length* arguments to any function, and drops all subexpressions that are below the specified depth.

`Shallow` shows a different outline of `t`.

```
In[5]:= Shallow[t]

Out[5]//Shallow= 1 + 12 x + 66 Power[<<2>>] + 220 Power[<<2>>] + 495 Power[<<2>>] + 792 Power[<<2>>] +
  924 Power[<<2>>] + 792 Power[<<2>>] + 495 Power[<<2>>] + 220 Power[<<2>>] + <<81>>
```

This includes only 10 arguments to each function, but allows any depth.

```
In[6]:= Shallow[t, {Infinity, 10}]
```

Out[6]//Shallow= $1 + 12\,x + 66\,x^2 + 220\,x^3 + 495\,x^4 + 792\,x^5 + 924\,x^6 + 792\,x^7 + 495\,x^8 + 220\,x^9 + \ll 81 \gg$

`Shallow` is particularly useful when you want to drop parts in a uniform way throughout a highly nested expression, such as a large list structure returned by `Trace`.

Here is the recursive definition of the Fibonacci function.

```
In[7]:= fib[n_] := fib[n - 1] + fib[n - 2]; fib[0] = fib[1] = 1
Out[7]= 1
```

This generates a large list structure.

```
In[8]:= tr = Trace[fib[8]];
```

You can use `Shallow` to see an outline of the structure.

```
In[9]:= Shallow[tr]
Out[9]//Shallow= {fib[≪1≫], Plus[≪2≫], {{≪2≫}, ≪1≫, ≪1≫, {≪7≫}, {≪7≫}, ≪1≫, ≪1≫},
  {{≪2≫}, ≪1≫, ≪1≫, {≪7≫}, {≪7≫}, ≪1≫, ≪1≫}, Plus[≪2≫], 34}
```

`Short` gives you a less uniform outline, which can be more difficult to understand.

```
In[10]:= Short[tr, 4]
Out[10]//Short= {fib[8], fib[8 - 1] + fib[8 - 2], {{8 - 1, 7}, fib[7], ≪3≫, 13 + 8, 21}, {≪1≫}, 21 + 13, 34}
```

When generated outputs in the notebook interface are exceedingly large, *Mathematica* automatically applies `Short` to the output. This user interface enhancement prevents *Mathematica* from spending a lot of time generating and formatting the printed output for an evaluation which probably generated output you did not expect.

Typically, an assignment like this would have a semicolon at the end.

```
In[11]:= lst = Range[10^6]
```

```
Out[11]=
A very large output was generated. Here is a sample of it:

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, ≪999964≫,
 999983, 999984, 999985, 999986, 999987, 999988, 999989, 999990, 999991,
 999992, 999993, 999994, 999995, 999996, 999997, 999998, 999999, 1000000}

[Show Less] [Show More] [Show Full Output] [Set Size Limit...]
```

The buttons in the user interface allow you to control how much of the output you see. The size threshold at which this behavior takes effect is determined by the byte count of the output expression. That byte count can be set in the **Preferences** dialog of the notebook interface, which is opened by the **Set Size Limit** button.

String-Oriented Output Formats

"*text*"	a string containing arbitrary text

Text strings.

The quotes are not included in standard *Mathematica* output form.

```
In[1]:= "This is a string."
Out[1]= This is a string.
```

In input form, the quotes are included.

```
In[2]:= InputForm[%]
Out[2]//InputForm= "This is a string."
```

You can put any kind of text into a *Mathematica* string. This includes non-English characters, as well as newlines and other control information. "Strings and Characters" discusses in more detail how strings work.

`StringForm["cccc``cccc", x1, x2, ...]`	output a string in which successive `` are replaced by successive x_i
`StringForm["cccc`i`cccc", x1, x2, ...]`	output a string in which each `i` is replaced by the corresponding x_i

Using format strings.

In many situations, you may want to generate output using a string as a "template", but "splicing" in various *Mathematica* expressions. You can do this using `StringForm`.

This generates output with each successive `` replaced by an expression.

```
In[3]:= StringForm["x = ``, y = ``", 3, (1 + u)^2]
Out[3]= x = 3, y = (1 + u)^2
```

You can use numbers to pick out expressions in any order.

```
In[4]:= StringForm["{`1`, `2`, `1`}", a, b]
Out[4]= {a, b, a}
```

The string in `StringForm` acts somewhat like a "format directive" in the formatted output statements of languages such as C and Fortran. You can determine how the expressions in `StringForm` will be formatted by wrapping them with standard output format functions.

You can specify how the expressions in `StringForm` are formatted using standard output format functions.

```
In[5]:= StringForm["The `` of `` is ``.", TeXForm, a / b, TeXForm[a / b]]

               a
Out[5]= The TeXForm of — is \frac{a}{b}.
               b
```

You should realize that `StringForm` is only an output format. It does not evaluate in any way. You can use the function `ToString` to create an ordinary string from a `StringForm` object.

`StringForm` generates formatted output in standard *Mathematica* output form.

```
In[6]:= StringForm["Q: `` -> ``", a, b]

Out[6]= Q: a -> b
```

In input form, you can see the actual `StringForm` object.

```
In[7]:= InputForm[%]

Out[7]//InputForm= StringForm["Q: `` -> ``", a, b]
```

This creates an ordinary string from the `StringForm` object.

```
In[8]:= InputForm[ToString[%]]

Out[8]//InputForm= "Q: a -> b"
```

`StringForm` allows you to specify a "template string", then fill in various expressions. Sometimes all you want to do is to concatenate together the output forms for a sequence of expressions. You can do this using `Row`.

`Row[{`$expr_1$`,`$expr_2$`,`...`}]`	give the output forms of the $expr_i$ concatenated together
`Row[`*list*`,` *s*`]`	insert *s* between successive elements
`Spacer[`*w*`]`	a space of *w* points which can be used in `Row`
`Invisible[`*expr*`]`	a space determined by the physical dimensions of *expr*

Output of sequences of expressions.

`Row` prints as a sequence of expressions concatenated together.

```
In[9]:= Row[{"[x = ", 56, "]"}]
Out[9]= [x = 56]
```

`Row` also works with typeset expressions.

```
In[10]:= Row[{"[y = ", Subscript[a, b], "]"}]
```
Out[10]= [y = a_b]

`Row` can automatically insert any expression between the displayed elements.

```
In[11]:= Row[{a, b, c, d}, "⟷"]
Out[11]= a⟷b⟷c⟷d
```

`Spacer` can be used to control the precise spacing between elements.

```
In[12]:= Row[{"x", Spacer[10], "y"}]
Out[12]= x  y
```

`Column[{`*expr*$_1$`,`*expr*$_2$`,…}]`	a left-aligned column of objects
`Column[`*list*`,`*alignment*`]`	a column with a specified horizontal alignment (`Left`, `Center` or `Right`)
`Column[`*list*`,`*alignment*`,`*s*`]`	a column with elements separated by *s* x-heights

Output of columns of expressions.

This arranges the two expressions in a column.

```
In[13]:= Column[{a + b, x^2}]
```
Out[13]= a + b
x^2

`Defer[`*expr*`]`	give the output form of *expr*, with *expr* maintained unevaluated
`Interpretation[`*e*`,`*expr*`]`	give an output which displays as *e*, but evaluates as *expr*

Output of unevaluated expressions.

Using text strings and functions like `Row`, you can generate pieces of output that do not necessarily correspond to valid *Mathematica* expressions. Sometimes, however, you want to generate

`Defer`

output that corresponds to a valid *Mathematica* expression, but only so long as the expression is not evaluated. The function `Defer` maintains its argument unevaluated, but allows it to be formatted in the standard *Mathematica* output form.

`Defer` maintains 1 + 1 unevaluated.

```
In[14]:= Defer[1 + 1]
Out[14]= 1 + 1
```

The `Defer` prevents the actual assignment from being done.

```
In[15]:= Defer[x = 3]
Out[15]= x = 3
```

When the output of `Defer` is evaluated again, which might happen by modifying the output or by using copy and paste, it will evaluate normally.

The following output was copied from the previous output cell into an input cell.

```
In[16]:= x = 3
Out[16]= 3
```

It is also possible to produce output whose appearance has no direct correlation to how it evaluates by using `Interpretation`. This method is effectively used by *Mathematica* when formatting some kinds of outputs where the most readable form does not correspond well to the internal representation of the object. For example, `Series` always generates an `Interpretation` object in its default output.

Although this output displays as `y`, it will evaluate as `x`.

```
In[17]:= Interpretation[y, x]
Out[17]= y
```

Copying and pasting the previous output will reference the value earlier assigned to `x`.

```
In[18]:= 2 y
Out[18]= 6
```

Output Formats for Numbers

`ScientificForm[`*expr*`]`	print all numbers in scientific notation
`EngineeringForm[`*expr*`]`	print all numbers in engineering notation (exponents divisible by 3)
`AccountingForm[`*expr*`]`	print all numbers in standard accounting format

Output formats for numbers.

These numbers are given in the default output format. Large numbers are given in scientific notation.

```
In[1]:= {6.7^-4, 6.7^6, 6.7^8}
```

Out[1]= $\{0.00049625,\ 90\,458.4,\ 4.06068\times10^{6}\}$

This gives all numbers in scientific notation.

```
In[2]:= ScientificForm[%]
```

Out[2]//ScientificForm=

$\{4.9625\times10^{-4},\ 9.04584\times10^{4},\ 4.06068\times10^{6}\}$

This gives the numbers in engineering notation, with exponents arranged to be multiples of three.

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

Out[3]//EngineeringForm=

$\{496.25\times10^{-6},\ 90.4584\times10^{3},\ 4.06068\times10^{6}\}$

In accounting form, negative numbers are given in parentheses, and scientific notation is never used.

```
In[4]:= AccountingForm[{5.6, -6.7, 10.^7}]
```

Out[4]//AccountingForm=

```
{5.6, (6.7), 10000000.}
```

`NumberForm[`*expr*`,`*tot*`]`	print at most *tot* digits of all approximate real numbers in *expr*
`ScientificForm[`*expr*`,`*tot*`]`	use scientific notation with at most *tot* digits
`EngineeringForm[`*expr*`,`*tot*`]`	use engineering notation with at most *tot* digits

Controlling the printed precision of real numbers.

Here is π^9 to 30 decimal places.

In[5]:=
```
N[Pi^9, 30]
```

Out[5]=
```
29 809.0993334462116665094024012
```

This prints just 10 digits of π^9.

In[6]:=
```
NumberForm[%, 10]
```

Out[6]//NumberForm=
```
29809.09933
```

This gives 12 digits, in engineering notation.

In[7]:=
```
EngineeringForm[%, 12]
```

Out[7]//EngineeringForm=

$29.8090993334 \times 10^3$

option name	*default value*	
`DigitBlock`	`Infinity`	maximum length of blocks of digits between breaks
`NumberSeparator`	`{",", " "}`	strings to insert at breaks between blocks of digits to the left and right of a decimal point
`NumberPoint`	`"."`	string to use for a decimal point
`NumberMultiplier`	`"\[Times]"`	string to use for the multiplication sign in scientific notation
`NumberSigns`	`{"-", ""}`	strings to use for signs of negative and positive numbers
`NumberPadding`	`{"", ""}`	strings to use for padding on the left and right
`SignPadding`	`False`	whether to insert padding after the sign
`NumberFormat`	`Automatic`	function to generate final format of number
`ExponentFunction`	`Automatic`	function to determine the exponent to use

Options for number formatting.

All the options in the table except the last one apply to both integers and approximate real numbers.

All the options can be used in any of the functions `NumberForm`, `ScientificForm`, `EngineeringForm` and `AccountingForm`. In fact, you can in principle reproduce the behavior of any one of these functions simply by giving appropriate option settings in one of the others. The default option settings listed in the table are those for `NumberForm`.

Setting `DigitBlock -> n` breaks digits into blocks of length n.

```
In[8]:= NumberForm[30!, DigitBlock -> 3]
Out[8]//NumberForm= 265,252,859,812,191,058,636,308,480,000,000
```

You can specify any string to use as a separator between blocks of digits.

```
In[9]:= NumberForm[30!, DigitBlock -> 5, NumberSeparator -> " "]
Out[9]//NumberForm= 265 25285 98121 91058 63630 84800 00000
```

This gives an explicit plus sign for positive numbers, and uses | in place of a decimal point.

```
In[10]:= NumberForm[{4.5, -6.8}, NumberSigns -> {"-", "+"}, NumberPoint -> "|"]
Out[10]//NumberForm=
{+4|5, -6|8}
```

When *Mathematica* prints an approximate real number, it has to choose whether scientific notation should be used, and if so, how many digits should appear to the left of the decimal point. What *Mathematica* does is first to find out what the exponent would be if scientific notation were used, and one digit were given to the left of the decimal point. Then it takes this exponent, and applies any function given as the setting for the option `ExponentFunction`. This function should return the actual exponent to be used, or `Null` if scientific notation should not be used.

The default is to use scientific notation for all numbers with exponents outside the range -5 to 5.

```
In[11]:= {8.^5, 11.^7, 13.^9}
Out[11]= {32 768., 1.94872×10^7, 1.06045×10^10}
```

This uses scientific notation only for numbers with exponents of 10 or more.

```
In[12]:= NumberForm[%, ExponentFunction -> (If[-10 < # < 10, Null, #] &)]
Out[12]//NumberForm=
{32768., 19487171., 1.06045×10^10}
```

This forces all exponents to be multiples of 3.

```
In[13]:= NumberForm[%, ExponentFunction -> (3 Quotient[#, 3] &)]
Out[13]//NumberForm=
{32.768×10^3, 19.4872×10^6, 10.6045×10^9}
```

Having determined what the mantissa and exponent for a number should be, the final step is to assemble these into the object to print. The option `NumberFormat` allows you to give an arbitrary function which specifies the print form for the number. The function takes as arguments three strings: the mantissa, the base, and the exponent for the number. If there is no exponent, it is given as `""`.

This gives the exponents in Fortran-like "e" format.

```
In[14]:= NumberForm[{5.6^10, 7.8^20}, NumberFormat -> (SequenceForm[#1, "e", #3] &)]

Out[14]//NumberForm=
         {3.03305e7, 6.94852e17}
```

You can use `FortranForm` to print individual numbers in Fortran format.

```
In[15]:= FortranForm[7.8^20]

Out[15]//FortranForm=
         6.94851587086215e17/
```

`PaddedForm[`*expr*`,`*tot*`]`	print with all numbers having room for *tot* digits, padding with leading spaces if necessary
`PaddedForm[`*expr*`,{`*tot*`,`*frac*`}]`	print with all numbers having room for *tot* digits, with exactly *frac* digits to the right of the decimal point
`NumberForm[`*expr*`,{`*tot*`,`*frac*`}]`	print with all numbers having at most *tot* digits, exactly *frac* of them to the right of the decimal point
`Column[{`$expr_1$`,`$expr_2$`,…}]`	print with the $expr_i$ left aligned in a column

Controlling the alignment of numbers in output.

Whenever you print a collection of numbers in a column or some other definite arrangement, you typically need to be able to align the numbers in a definite way. Usually you want all the numbers to be set up so that the digit corresponding to a particular power of 10 always appears at the same position within the region used to print a number.

You can change the positions of digits in the printed form of a number by "padding" it in various ways. You can pad on the right, typically adding zeros somewhere after the decimal. Or you can pad on the left, typically inserting spaces in place of leading zeros.

This pads with spaces to make room for up to 7 digits in each integer.

```
In[16]:= PaddedForm[{456, 12 345, 12}, 7]

Out[16]//PaddedForm=
         {     456,    12345,       12}
```

This creates a column of integers.

```
In[17]:= PaddedForm[Column[{456, 12345, 12}], 7]

Out[17]//PaddedForm=
    456
  12345
     12
```

This prints each number with room for a total of 7 digits, and with 4 digits to the right of the decimal point.

```
In[18]:= PaddedForm[{-6.7, 6.888, 6.99999}, {7, 4}]

Out[18]//PaddedForm=
{  -6.7000,    6.8880,    7.0000}
```

In `NumberForm`, the 7 specifies the maximum precision, but does not make *Mathematica* pad with spaces.

```
In[19]:= NumberForm[{-6.7, 6.888, 6.99999}, {7, 4}]

Out[19]//NumberForm=
{-6.7000, 6.8880, 7.0000}
```

If you set the option `SignPadding -> True`, *Mathematica* will insert leading spaces *after* the sign.

```
In[20]:= PaddedForm[{-6.7, 6.888, 6.99999}, {7, 4}, SignPadding -> True]

Out[20]//PaddedForm=
{-  6.7000,    6.8880,    7.0000}
```

Only the mantissa portion is aligned when scientific notation is used.

```
In[21]:= PaddedForm[Column[{6.7 × 10^8, 48.7, -2.3 10^-16}], {4, 2}]

Out[21]//PaddedForm=
  6.70×10^8
 48.70
 -2.30×10^-16
```

With the default setting for the option `NumberPadding`, both `NumberForm` and `PaddedForm` insert trailing zeros when they pad a number on the right. You can use spaces for padding on both the left and the right by setting `NumberPadding -> {" ", " "}`.

This uses spaces instead of zeros for padding on the right.

```
In[22]:= PaddedForm[{-6.7, 6.888, 6.99999}, {7, 4}, NumberPadding -> {" ", " "}]

Out[22]//PaddedForm=
{  -6.7   ,    6.888 ,    7.    }
```

BaseForm[*expr*, *b*]	print with all numbers given in base *b*

Printing numbers in other bases.

This prints a number in base 2.

```
In[23]:= BaseForm[2 342 424, 2]
Out[23]//BaseForm= 1000111011111000011000₂
```

In bases higher than 10, letters are used for the extra digits.

```
In[24]:= BaseForm[242 345 341, 16]
Out[24]//BaseForm= e71e57d₁₆
```

BaseForm also works with approximate real numbers.

```
In[25]:= BaseForm[2.3, 2]
Out[25]//BaseForm= 10.010011001100110011₂
```

You can even use BaseForm for numbers printed in scientific notation.

```
In[26]:= BaseForm[2.3 × 10^8, 2]
Out[26]//BaseForm= 1.1011011010110000101₂ × 2^27
```

"Digits in Numbers" discusses how to enter numbers in arbitrary bases, and also how to get lists of the digits in a number.

Tables and Matrices

Column[*list*]	typeset as a column of elements
Grid[*list*]	typeset as a grid of elements
TableForm[*list*]	print in tabular form

Formatting lists as tables and matrices.

Here is a list.

```
In[1]:= Table[(i + 45)^j, {i, 3}, {j, 3}]
Out[1]= {{46, 2116, 97 336}, {47, 2209, 103 823}, {48, 2304, 110 592}}
```

`Grid` gives the list typeset in a tabular format.

In[2]:=
```
Grid[%]
```

Out[2]=
```
46  2116   97 336
47  2209  103 823
48  2304  110 592
```

`TableForm` displays the list in a tabular format.

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

Out[3]//TableForm=
```
46  2116  97 336
47  2209  103 823
48  2304  110 592
```

`Grid` and `Column` are wrappers which do not evaluate, but typeset their contents into appropriate forms. They are typesetting constructs and require a front end to render correctly.

`Column` is a shorthand for a `Grid` with one column.

In[4]:=
```
Column[Range[5]]
```

Out[4]=
```
1
2
3
4
5
```

The `FullForm` of a `Grid` or `Column` demonstrates that the head is inert.

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

Out[5]//FullForm=
```
Column[List[1, 2, 3, 4, 5]]
```

All of these wrappers can be used to present any kind of data, including graphical data.

In[6]:=
```
Grid[{{"disk", Graphics[Disk[], ImageSize → 25]},
  {"square", Graphics[Rectangle[], ImageSize → 25]}}]
```

Out[6]=
```
disk    ●
square  ■
```

`PaddedForm[Column[`*list*`],`*tot*`]`	print a column with all numbers padded to have room for *tot* digits
`PaddedForm[Grid[`*list*`],`*tot*`]`	print a table with all numbers padded to have room for *tot* digits
`PaddedForm[Grid[`*list*`],{`*tot*`,`*frac*`}]`	put *frac* digits to the right of the decimal point in all approximate real numbers

Printing tables of numbers.

Here is a list of numbers.

```
In[7]:= fac = {10!, 15!, 20!}

Out[7]= {3 628 800, 1 307 674 368 000, 2 432 902 008 176 640 000}
```

`Column` displays the list in a column.

```
In[8]:= Column[fac]

        3 628 800
Out[8]= 1 307 674 368 000
        2 432 902 008 176 640 000
```

This aligns the numbers by padding each one to leave room for up to 20 digits.

```
In[9]:= PaddedForm[Column[fac], 20]

                                  3628800
Out[9]//PaddedForm=         1307674368000
                      2432902008176640000
```

In this particular case, you could also align the numbers using the `Alignment` option.

```
In[10]:= Column[fac, Alignment -> {Right}]

                         3 628 800
Out[10]=         1 307 674 368 000
         2 432 902 008 176 640 000
```

This lines up the numbers, padding each one to have room for 8 digits, with 5 digits to the right of the decimal point.

```
In[11]:= PaddedForm[Column[{6.7, 6.888, 6.99999}], {8, 5}]

Out[11]//PaddedForm=
   6.70000
   6.88800
   6.99999
```

`SpanFromLeft`	span from the element on the left
`SpanFromAbove`	span from the element above
`SpanFromBoth`	span from the element above and to the left

Symbols used to represent spanning in `Grid`.

`Grid` takes a rectangular matrix as its first argument. Individual elements of the `Grid` can span across multiple rows, columns, or a rectangular subgrid by specifying the areas to be spanned. The spanning element is always specified in the upper left-hand corner of the spanning area, and the remaining area is filled in with the appropriate spanning symbols.

This shows a spanning row, where the spanning portion is filled in using `SpanFromLeft`.

```
In[12]:= Grid[{{"t", SpanFromLeft, SpanFromLeft, SpanFromLeft}, {"a", "b", "c", "d"}}]

Out[12]=    t
         a b c d
```

Similarly, a column can be spanned using `SpanFromAbove`.

```
In[13]:= Grid[{{"t", "a"}, {SpanFromAbove, "b"}}]

Out[13]= t a
           b
```

When specifying a rectangular spanning area, `SpanFromBoth` is used in every element which is both below and to the right of the spanning element.

```
In[14]:= Grid[{{"t", SpanFromLeft, "a"},
           {SpanFromAbove, SpanFromBoth, "b"}, {"c", "d", "e"}}]

Out[14]=   t   a
               b
         c d e
```

option	*default value*	
`Background`	`None`	what background colors to use
`BaselinePosition`	`Automatic`	what to align with a surrounding text baseline
`BaseStyle`	`{}`	base style specifications for the grid
`Frame`	`None`	where to draw frames in the grid
`FrameStyle`	`Automatic`	style to use for frames

Some options which affect the behavior of a `Grid` as a whole.

The `Frame` option can specify a frame around the entire `Grid`.

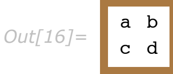
```
In[15]:= Grid[{{"a", "b"}, {"c", "d"}}, Frame → True]
```

```
Out[15]= a b
         c d
```

This uses `FrameStyle` to change the appearance of a frame.

```
In[16]:= Grid[{{"a", "b"}, {"c", "d"}}, Frame → True,
          FrameStyle → {Brown, AbsoluteThickness[5]}]
```

Out[16]=



This uses `Background` to specify a background color for the entire `Grid`.

```
In[17]:= Grid[{{"a", "b"}, {"c", "d"}}, Background → Pink, Frame → True]
```

```
Out[17]= a b
         c d
```

The position of a `Grid` relative to its surroundings can be controlled using the `BaselinePosition` option.

```
In[18]:= Row[{"A matrix:", Grid[{{1, 2}, {3, 4}}, BaselinePosition → Top]}]
```

```
Out[18]= A matrix: 1 2
                   3 4
```

This aligns the bottom of the grid with the baseline.

```
In[19]:= Row[{"A matrix:", Grid[{{1, 2}, {3, 4}}, BaselinePosition → Bottom]}]
```

```
                   1 2
Out[19]= A matrix: 3 4
```

This sets the base style of the entire `Grid` to be the Subsection style.

```
In[20]:= Grid[{{"a", "bit"}, {"of", "text"}}, BaseStyle → {"Subsection"}]
```

Out[20]=

a ***bit***

of ***text***

`Column` is a shorthand for specifying a `Grid` with one column. Since the two functions are similar, the same options can be used for each one.

This sets some `Grid` options for `Column`.

```
In[21]:= Column[{1, 2, 3, 4}, Background → Pink, Frame → True]

Out[21]= 1
         2
         3
         4
```

option	default value	
`Alignment`	`{Center, Baseline}`	horizontal and vertical alignment of items
`Dividers`	`None`	where to draw divider lines in the grid
`ItemSize`	`Automatic`	width and height of each item
`ItemStyle`	`None`	styles for columns and rows
`Spacings`	`{0.8,0.1}`	horizontal and vertical spacings

Some options which affect the columns and rows of a `Grid`.

The options for `Grid` which affect individual rows and columns all share a similar syntax. The options can be specified as $\{x, y\}$, where x applies to all of the columns and y applies to all of the rows; x and y can be single values, or they can be a list of values which represent each column or row in turn.

With no `Alignment` setting, elements align to the center horizontally and on the baseline vertically.

```
In[22]:= Grid[{{"ten", 10!}, {"twenty", 20!}}]

Out[22]=  ten           3 628 800
         twenty  2 432 902 008 176 640 000
```

This changes the horizontal alignment of columns to be on the right.

```
In[23]:= Grid[{{"ten", 10!}, {"twenty", 20!}}, Alignment → {Right, Baseline}]

Out[23]=    ten                  3 628 800
         twenty  2 432 902 008 176 640 000
```

This sets the horizontal alignment of each column separately.

```
In[24]:= Grid[{{"ten", 10!}, {"twenty", 20!}}, Alignment → {{Left, Right}, Baseline}]

Out[24]= ten                     3 628 800
         twenty  2 432 902 008 176 640 000
```

When `Background` or `ItemStyle` options specify distinct settings for rows and columns, the front end will attempt to combine the settings where the rows and columns overlap.

This shows how the green row combines with columns of various colors.

```
In[25]:= Grid[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}},
          Background → {{Orange, None, Cyan}, {None, Green, None}}]
```

```
Out[25]= 1 2 3
         4 5 6
         7 8 9
```

This example shows how `ItemStyle` can combine styles specified in both rows and columns.

```
In[26]:= Grid[{{1, 2}, {3, 4}}, ItemStyle → {{Red, Automatic}, {Bold, Italic}}]
```

```
Out[26]= 1 2
         3 4
```

To repeat an individual row or column specification over multiple rows or columns, wrap it in a list. The repeated element will be used as often as necessary. If you wrap multiple elements in a list, the entire list will be repeated in sequence.

The red divider is repeated.

```
In[27]:= Grid[{{1, 2, 3, 4, 5, 6}, {7, 8, 9, 10, 11, 12}},
          Dividers → {{None, {Red}, None}, None}]
```

```
Out[27]= 1 | 2 | 3 |  4 |  5 |  6
         7 | 8 | 9 | 10 | 11 | 12
```

Here, red and black dividers are repeated in sequence.

```
In[28]:= Grid[{{1, 2, 3, 4, 5, 6}, {7, 8, 9, 10, 11, 12}},
          Dividers → {{None, {Red, Black}, None}, None}]
```

```
Out[28]= 1 | 2 | 3 |  4 |  5 |  6
         7 | 8 | 9 | 10 | 11 | 12
```

The `ItemSize` and `Spacings` options take their horizontal measurements in ems and their vertical measurements in line heights based upon the current font. Both options also can take a `Scaled` coordinate, where the coordinate specifies the fraction of the total cell width or window height. The `ItemSize` option also allows you to request as much space as is required to fit all of the elements in the given row or column by using the keyword `Full`.

This makes all of the items 3 ems wide and 1 line height tall.

```
In[29]:= Grid[{{1, 2}, {3, 4}}, Dividers → All, ItemSize → {3, 1}]
```

Out[29]=

1	2
3	4

The same example in a new font size will show at a different size.

```
In[30]:= Style[Grid[{{1, 2}, {3, 4}}, Dividers → All, ItemSize → {3, 1}], FontSize → 18]
```

Out[30]=

1	2
3	4

The buttons in this example will always be sized to be a quarter of the width of the cell.

```
In[31]:= Grid[{{Button["Left button"], Button["Right button"]}},
          ItemSize → {{{Scaled[0.25]}}, Full}]
```

Out[31]= Left button Right button

The first and last settings for `Spacings` specify one-half of the top and bottom space.

```
In[32]:= Grid[{{1, 2}, {3, 4}}, Frame → True, Spacings → {{1, 0, 1}, {1, 1, 1}}]
```

Out[32]=

1 2
3 4

option	*default value*	
`Alignment`	`{Center, Baseline}`	horizontal and vertical alignment of items
`Background`	`None`	what background colors to use
`BaseStyle`	`{}`	base style specifications for the item
`Frame`	`None`	where to draw frames around the item
`FrameStyle`	`Automatic`	style to use for frames
`ItemSize`	`Automatic`	width and height of each item

Some options for `Item`.

Many of the settings which can be applied to entire rows and columns can also be applied individually to the elements of a `Grid` or `Column` by using the `Item` wrapper. `Item` allows you to change these settings at the granularity of a single item. Settings which are specified at the `Item` level always override settings from the `Grid` or `Column` as a whole.

This sets item-specific options for the lower left-hand element.

```
In[33]:= Grid[{{1, 2}, {Item[3, Background → LightGreen, Frame → True], 4}}]
```

Out[33]=

1	2
3	4

The Background setting for Item overrides the one for the Column.

```
In[34]:= Column[{1, 2, Item[3, Background → Pink]}, Background → Yellow]

         1
Out[34]= 2
         3
```

Most of the options to Item take the same settings as their Grid counterparts. However, the Alignment and ItemSize options, which allow complex row and column settings in Grid, take only the {*horizontal*, *vertical*} setting in Item.

This specifies a larger item area and how the text should be aligned within it.

```
In[35]:= Column[{Item["Some aligned text", Frame → True,
            ItemSize → {15, 3}, Alignment → {Center, Bottom}], "caption"}]

Out[35]=     Some aligned text
         caption
```

The width value of the ItemSize option is used to determine line breaking.

```
In[36]:= Column[{Item[N[Pi, 20], ItemSize → {10, 2}], "digits of pi"}]

         3.141592653589793\
Out[36]=  2385
         digits of pi
```

The ItemSize here specifies a minimum height of 2 line heights, but the item is larger.

```
In[37]:= Column[{Item[N[Pi, 50], ItemSize → {10, 2}], "digits of pi"}]

         3.141592653589793\
          238462643383279\
Out[37]=  502884197169399\
          3751
         digits of pi
```

Formatting Higher-Dimensional Data

Column supports one-dimensional data, and Grid supports two-dimensional data. To print arrays with an arbitrary number of dimensions, you can use TableForm.

Here is the format for a 2×2 array of elements a[*i*, *j*].

```
In[39]:= TableForm[Array[a, {2, 2}]]

Out[39]//TableForm= a[1, 1]  a[1, 2]
                    a[2, 1]  a[2, 2]
```

Here is a 2×2×2 array.

```
In[40]:= TableForm[{Array[a, {2, 2}], Array[b, {2, 2}]}]

                    a[1, 1]  a[2, 1]
                    a[1, 2]  a[2, 2]
Out[40]//TableForm= b[1, 1]  b[2, 1]
                    b[1, 2]  b[2, 2]
```

And here is a 2×2×2×2 array.

```
In[41]:= TableForm[
          {{Array[a, {2, 2}], Array[b, {2, 2}]}, {Array[c, {2, 2}], Array[d, {2, 2}]}}]

                    a[1, 1]  a[1, 2]  b[1, 1]  b[1, 2]
                    a[2, 1]  a[2, 2]  b[2, 1]  b[2, 2]
Out[41]//TableForm= c[1, 1]  c[1, 2]  d[1, 1]  d[1, 2]
                    c[2, 1]  c[2, 2]  d[2, 1]  d[2, 2]
```

In general, when you print an n-dimensional table, successive dimensions are alternately given as columns and rows. By setting the option `TableDirections -> {`dir_1`, `dir_2`, …}`, where the dir_i are `Column` or `Row`, you can specify explicitly which way each dimension should be given. By default, the option is effectively set to `{Column, Row, Column, Row, …}`.

The option `TableDirections` allows you to specify explicitly how each dimension in a multidimensional table should be given.

```
In[42]:= TableForm[{Array[a, {2, 2}], Array[b, {2, 2}]},
          TableDirections -> {Row, Row, Column}]

Out[42]//TableForm= a[1, 1]  a[2, 1]  b[1, 1]  b[2, 1]
                    a[1, 2]  a[2, 2]  b[1, 2]  b[2, 2]
```

`TableForm` can handle arbitrary "ragged" arrays. It leaves blanks wherever there are no elements supplied.

`TableForm` can handle "ragged" arrays.

```
In[43]:= TableForm[{{a, a, a}, {b, b}}]

Out[43]//TableForm= a    a    a
                    b    b
```

You can include objects that behave as "subtables".

```
In[44]:= TableForm[{{a, {{p, q}, {r, s}}, a, a}, {{x, y}, b, b}}]

                         p q
                    a    r s    a    a
Out[44]//TableForm=
                    x
                    y    b      b
```

You can control the number of levels in a nested list to which `TableForm` goes by setting the option `TableDepth`.

This tells `TableForm` only to go down to depth 2. As a result `{x, y}` is treated as a single table entry.

```
In[45]:= TableForm[{{a, {x, y}}, {c, d}}, TableDepth -> 2]

Out[45]//TableForm= a {x, y}
                    c d
```

option name	*default value*	
`TableDepth`	`Infinity`	maximum number of levels to include in the table
`TableDirections`	`{Column,Row, Column,…}`	whether to arrange dimensions as rows or columns
`TableAlignments`	`{Left,Bottom, Left,…}`	how to align the entries in each dimension
`TableSpacing`	`{1,3,0,1,0,…}`	how many spaces to put between entries in each dimension
`TableHeadings`	`{None,None,…}`	how to label the entries in each dimension

Options for `TableForm`.

With the option `TableAlignments`, you can specify how each entry in the table should be aligned with its row or column. For columns, you can specify `Left`, `Center` or `Right`. For rows, you can specify `Bottom`, `Center` or `Top`. If you set `TableAlignments -> Center`, all entries will be centered both horizontally and vertically. `TableAlignments -> Automatic` uses the default choice of alignments.

Entries in columns are by default aligned on the left.

```
In[46]:= TableForm[{a, bbbb, cccccccc}]

                    a
Out[46]//TableForm= bbbb
                    cccccccc
```

This centers all entries.

```
In[47]:= TableForm[{a, bbbb, cccccccc}, TableAlignments -> Center]

                       a
Out[47]//TableForm=   bbbb
                    cccccccc
```

You can use the option `TableSpacing` to specify how much horizontal space there should be between successive columns, or how much vertical space there should be between successive rows. A setting of 0 specifies that successive objects should abut.

This leaves 6 spaces between the entries in each row, and no space between successive rows.

```
In[48]:= TableForm[{{a, b}, {ccc, d}}, TableSpacing -> {0, 6}]

Out[48]//TableForm=
a        b
ccc      d
```

`None`	no labels in any dimension
`Automatic`	successive integer labels in each dimension
`{{`lab_{11}`,`lab_{12}`,…},…}`	explicit labels

Settings for the option `TableHeadings`.

This puts integer labels in a 2×2×2 array.

```
In[49]:= TableForm[Array[a, {2, 2, 2}], TableHeadings -> Automatic]

Out[49]//TableForm=
  | 1                2
--+------------------------------
  | 1 | a[1, 1, 1]   1 | a[1, 2, 1]
1 | 2 | a[1, 1, 2]   2 | a[1, 2, 2]
  | 1 | a[2, 1, 1]   1 | a[2, 2, 1]
2 | 2 | a[2, 1, 2]   2 | a[2, 2, 2]
```

This gives a table in which the rows are labeled by integers, and the columns by a list of strings.

```
In[50]:= TableForm[{{a, b, c}, {ap, bp, cp}},
          TableHeadings -> {Automatic, {"first", "middle", "last"}}]

Out[50]//TableForm=
  | first  middle  last
--+--------------------
1 | a      b       c
2 | ap     bp      cp
```

This labels the rows but not the columns. `TableForm` automatically drops the third label since there is no corresponding row.

```
In[51]:= TableForm[{{2, 3, 4}, {5, 6, 1}},
          TableHeadings -> {{"row a", "row b", "row c"}, None}]

Out[51]//TableForm=
row a | 2  3  4
row b | 5  6  1
```

Styles and Fonts in Output

`Style[`*expr*`,`*options*`]`	print with the specified style options
`Style[`*expr*`,"`*style*`"]`	print with the specified cell style

Specifying output styles.

The second x^2 is here shown in boldface.

```
In[1]:= {x^2, Style[x^2, FontWeight -> "Bold"]}

Out[1]= {x², x²}
```

This shows the word `text` in font sizes from 10 to 20 points.

```
In[2]:= Table[Style["text", FontSize -> s], {s, 10, 20}]

Out[2]= {text, text, text, text, text, text, text, text, text, text, text}
```

This shows the text in the Helvetica font.

```
In[3]:= Style["some text", FontFamily -> "Helvetica"]

Out[3]= some text
```

`Style` allows an abbreviated form of some options. For options such as `FontSize`, `FontWeight`, `FontSlant` and `FontColor`, you can include merely the setting of the option.

Options are specified here in a short form.

```
In[4]:= Style["text", 20, Italic]

Out[4]= text
```

option	*typical setting(s)*	
`FontSize`	`12`	size of characters in printer's points
`FontWeight`	`"Plain"` or `"Bold"`	weight of characters
`FontSlant`	`"Plain"` or `"Italic"`	slant of characters
`FontFamily`	`"Courier"`, `"Times"`, `"Helvetica"`	font family
`FontColor`	`GrayLevel[0]`	color of characters
`Background`	`GrayLevel[1]`	background color for characters

A few options that can be used in `Style`.

If you use the notebook front end for *Mathematica*, then each piece of output that is generated will by default be in the style of the cell in which the output appears. By using `Style[`*expr*`, "`*style*`"]` however, you can tell *Mathematica* to output a particular expression in a different style.

Here is an expression output in the style normally used for section headings.

```
In[5]:= Style[x^2 + y^2, "Section"]
```

Out[5]= $x^2 + y^2$

"Cells as *Mathematica* Expressions" describes in more detail how cell styles work. By using `Style[`*expr*`, "`*style*`", `*options*`]` you can generate output that is in a particular style, but with certain options modified.

Representing Textual Forms by Boxes

All textual and graphical forms in *Mathematica* are ultimately represented in terms of nested collections of *boxes*. Typically the elements of these boxes correspond to objects that are to be placed at definite relative positions in two dimensions.

Here are the boxes corresponding to the expression `a + b`.

```
In[1]:= ToBoxes[a + b]
Out[1]= RowBox[{a, +, b}]
```

`DisplayForm` shows how these boxes would be displayed.

```
In[2]:= DisplayForm[%]
Out[2]//DisplayForm= a + b
```

`DisplayForm[`*boxes*`]`	show *boxes* as they would be displayed

Showing the displayed form of boxes.

This displays three strings in a row.

```
In[3]:= RowBox[{"a", "+", "b"}] // DisplayForm
Out[3]//DisplayForm= a + b
```

This displays one string as a subscript of another.

```
In[4]:= SubscriptBox["a", "i"] // DisplayForm
```

Out[4]//DisplayForm= a_i

This puts two subscript boxes in a row.

```
In[5]:= RowBox[{SubscriptBox["a", "1"], SubscriptBox["b", "2"]}] // DisplayForm
```

Out[5]//DisplayForm= $a_1 b_2$

"*text*"	literal text
RowBox[{*a*,*b*,…}]	a row of boxes or strings *a*, *b*, …
GridBox[{{a_1,b_1,…},{a_2,b_2,…},…}]	a grid of boxes $\begin{matrix} a_1 & b_1 & \ldots \\ a_2 & b_2 & \ldots \\ \vdots & \vdots & \end{matrix}$
SubscriptBox[*a*,*b*]	subscript a_b
SuperscriptBox[*a*,*b*]	superscript a^b
SubsuperscriptBox[*a*,*b*,*c*]	subscript and superscript a_b^c
UnderscriptBox[*a*,*b*]	underscript $\underset{b}{a}$
OverscriptBox[*a*,*b*]	overscript $\overset{b}{a}$
UnderoverscriptBox[*a*,*b*,*c*]	underscript and overscript $\overset{c}{\underset{b}{a}}$
FractionBox[*a*,*b*]	fraction $\frac{a}{b}$
SqrtBox[*a*]	square root $\sqrt{a}$
RadicalBox[*a*,*b*]	b^{th} root $\sqrt[b]{a}$

Some basic box types.

This nests a fraction inside a radical.

```
In[6]:= RadicalBox[FractionBox[x, y], n] // DisplayForm
```

Out[6]//DisplayForm= $\sqrt[n]{\frac{x}{y}}$

This puts a superscript on a subscripted object.

```
In[7]:= SuperscriptBox[SubscriptBox[a, b], c] // DisplayForm
```

Out[7]//DisplayForm= ${a_b}^c$

This puts both a subscript and a superscript on the same object.

```
In[8]:= SubsuperscriptBox[a, b, c] // DisplayForm
```

Out[8]//DisplayForm= a_b^c

`FrameBox[`*box*`]`	render *box* with a frame drawn around it
`GridBox[`*list*`,RowLines->True]`	put lines between rows in a `GridBox`
`GridBox[`*list*`,ColumnLines->True]`	put lines between columns
`GridBox[`*list*`,RowLines->{True,False}]`	put a line below the first row, but not subsequent ones

Inserting frames and grid lines.

This shows a fraction with a frame drawn around it.

```
In[9]:= FrameBox[FractionBox["x", "y"]] // DisplayForm
```

Out[9]//DisplayForm= $\boxed{\frac{x}{y}}$

This puts lines between rows and columns of an array.

```
In[10]:= GridBox[Table[i + j, {i, 3}, {j, 3}],
           RowLines -> True, ColumnLines -> True] // DisplayForm
```

Out[10]//DisplayForm=

2	3	4
3	4	5
4	5	6

And this also puts a frame around the outside.

```
In[11]:= FrameBox[%] // DisplayForm
```

Out[11]//DisplayForm=

2	3	4
3	4	5
4	5	6

`StyleBox[`*boxes*`,`*options*`]`	render *boxes* with the specified option settings
`StyleBox[`*boxes*`,"`*style*`"]`	render *boxes* in the specified style

Modifying the appearance of boxes.

`StyleBox` takes the same options as `Style`. The difference is that `Style` is a high-level function that applies to an expression to determine how it will be displayed, while `StyleBox` is the corresponding low-level function in the underlying box structure.

This shows the string `"name"` in italics.

```
In[12]:= StyleBox["name", FontSlant -> "Italic"] // DisplayForm
Out[12]//DisplayForm=
name
```

This shows `"name"` in the style used for section headings in your current notebook.

```
In[13]:= StyleBox["name", "Section"] // DisplayForm
Out[13]//DisplayForm=
name
```

This uses section heading style, but with characters shown in gray.

```
In[14]:= StyleBox["name", "Section", FontColor -> GrayLevel[0.5]] // DisplayForm
Out[14]//DisplayForm=
name
```

If you use a notebook front end for *Mathematica*, then you will be able to change the style and appearance of what you see on the screen directly by using menu items. Internally, however, these changes will still be recorded by the insertion of appropriate `StyleBox` objects.

`FormBox[`*boxes*`,`*form*`]`	interpret *boxes* using rules associated with the specified form
`InterpretationBox[`*boxes*`,`*expr*`]`	interpret *boxes* as representing the expression *expr*
`TagBox[`*boxes*`,`*tag*`]`	use *tag* to guide the interpretation of *boxes*
`ErrorBox[`*boxes*`]`	indicate an error and do not attempt further interpretation of *boxes*

Controlling the interpretation of boxes.

This prints as `x` with a superscript.

```
In[15]:= SuperscriptBox["x", "2"] // DisplayForm
Out[15]//DisplayForm=
x^2
```

It is normally interpreted as a power.

```
In[16]:= ToExpression[%] // InputForm
Out[16]//InputForm= x^2
```

This again prints as `x` with a superscript.

```
In[17]:= InterpretationBox[SuperscriptBox["x", "2"], vec[x, 2]] // DisplayForm
Out[17]//DisplayForm=
x^2
```

But now it is interpreted as `vec[x, 2]`, following the specification given in the `InterpretationBox`.

```
In[18]:= ToExpression[%] // InputForm
Out[18]//InputForm= vec[x, 2]
```

If you edit the boxes given in an `InterpretationBox`, then there is no guarantee that the interpretation specified by the interpretation box will still be correct. As a result, *Mathematica* provides various options that allow you to control the selection and editing of `InterpretationBox` objects.

option	*default value*	
`Editable`	`Automatic`	whether to allow the contents to be edited
`Selectable`	`True`	whether to allow the contents to be selected
`Deletable`	`True`	whether to allow the box to be deleted
`DeletionWarning`	`False`	whether to issue a warning if the box is deleted
`BoxAutoDelete`	`False`	whether to strip the box if its contents are modified
`StripWrapperBoxes`	`False`	whether to remove `StyleBox` etc. from within *boxes* in `TagBox[`*boxes*`, …]`

Options for `InterpretationBox` and related boxes.

`TagBox` objects are used to store information that will not be displayed but which can nevertheless be used by the rules that interpret boxes. Typically the *tag* in `TagBox[`*boxes*, *tag*`]` is a symbol which gives the head of the expression corresponding to *boxes*. If you edit only the arguments of this expression then there is a good chance that the interpretation specified by the `TagBox` will still be appropriate. As a result, `Editable -> True` is effectively the default setting for a `TagBox`.

The rules that *Mathematica* uses for interpreting boxes are in general set up to ignore details of formatting, such as those defined by `StyleBox` objects. Thus, unless `StripWrapperBoxes -> False`, a red `x`, for example, will normally not be distinguished from an ordinary black `x`.

A red `x` is usually treated as identical to an ordinary one.

```
In[19]:= ToExpression[StyleBox[x, FontColor -> RGBColor[1, 0, 0]]] == x
Out[19]= True
```

String Representation of Boxes

Mathematica provides a compact way of representing boxes in terms of strings. This is particularly convenient when you want to import or export specifications of boxes as ordinary text.

This generates an `InputForm` string that represents the `SuperscriptBox`.

```
In[1]:= ToString[SuperscriptBox["x", "2"], InputForm]
Out[1]= \(x\^2\)
```

This creates the `SuperscriptBox`.

```
In[2]:= \(x \^ 2\)
Out[2]= SuperscriptBox[x, 2]
```

`ToExpression` interprets the `SuperscriptBox` as a power.

```
In[3]:= ToExpression[%] // FullForm
Out[3]//FullForm= Power[x, 2]
```

It is important to distinguish between forms that represent just raw boxes, and forms that represent the *meaning* of the boxes.

This corresponds to a raw `SuperscriptBox`.

```
In[4]:= \(x \^ 2\)
Out[4]= SuperscriptBox[x, 2]
```

This corresponds to the power that the `SuperscriptBox` represents.

```
In[5]:= \!\(x \^ 2\)
```

Out[5]= x^2

The expression generated here is a power.

```
In[6]:= FullForm[\!\(x \^ 2\)]
Out[6]//FullForm= Power[x, 2]
```

`\(`*input*`\)`	raw boxes
`\!\(`*input*`\)`	the meaning of the boxes

Distinguishing raw boxes from the expressions they represent.

If you copy the contents of a `StandardForm` cell into another program, such as a text editor, *Mathematica* will generate a `\!\(...\)` form where necessary. This is done so that if you subsequently paste the form back into *Mathematica*, the original contents of the `StandardForm` cell will automatically be re-created. Without the `\!`, only the raw boxes corresponding to these contents would be obtained.

With default settings for options, `\!\(...\)` forms pasted into *Mathematica* notebooks are automatically displayed in two-dimensional form.

`"\(`*input*`\)"`	a raw character string
`"\!\(`*input*`\)"`	a string containing boxes

Embedding two-dimensional box structures in strings.

Mathematica will usually treat a `\(...\)` form that appears within a string just like any other sequence of characters. But by inserting a `\!` you can tell *Mathematica* instead to treat this form like the boxes it represents. In this way you can therefore embed box structures within ordinary character strings.

Mathematica treats this as an ordinary character string.

```
In[7]:= "\( x \^ 2 \)"
Out[7]= \( x \^ 2 \)
```

The `!\` tells *Mathematica* that this string contains boxes.

```
In[8]:= "\!\( x \^ 2 \)"
```

Out[8]= x^2

You can mix boxes with ordinary text.

```
In[9]:= "box 1: \!\(x\^2\); box 2: \!\(y\^3\)"
```

Out[9]= box 1: x^2; box 2: y^3

`\(`*box*$_1$`,`*box*$_2$`,`…`\)`	`RowBox[`*box*$_1$`,`*box*$_2$`,`…`]`
box$_1$`\^`*box*$_2$	`SuperscriptBox[`*box*$_1$`,`*box*$_2$`]`
box$_1$`_` *box*$_2$	`SubscriptBox[`*box*$_1$`,`*box*$_2$`]`
box$_1$`_` *box*$_2$`\%` *box*$_3$	`SubsuperscriptBox[`*box*$_1$`,`*box*$_2$`,`*box*$_3$`]`
box$_1$`\&` *box*$_2$	`OverscriptBox[`*box*$_1$`,`*box*$_2$`]`
box$_1$`\+`*box*$_2$	`UnderscriptBox[`*box*$_1$`,`*box*$_2$`]`
box$_1$`\+`*box*$_2$`\%` *box*$_3$	`UnderoverscriptBox[`*box*$_1$`,`*box*$_2$`,`*box*$_3$`]`
box$_1$`\/`*box*$_2$	`FractionBox[`*box*$_1$`,`*box*$_2$`]`
`\@`*box*	`SqrtBox[`*box*`]`
`\@`*box*$_1$`\%`*box*$_2$	`RadicalBox[`*box*$_1$`,`*box*$_2$`]`
form`` \` `` *box*	`FormBox[`*box*`,`*form*`]`
`*`*input*	construct boxes from *input*

Input forms for boxes.

Mathematica requires that any input forms you give for boxes be enclosed within `\(` and `\)`. But within these outermost `\(` and `\)` you can use additional `\(` and `\)` to specify grouping.

Here ordinary parentheses are used to indicate grouping.

```
In[10]:= \(x \/ (y + z)\) // DisplayForm
```

Out[10]//DisplayForm=

$$\frac{x}{(y + z)}$$

Without the parentheses, the grouping would be different.

```
In[11]:= \(x \/ y + z\) // DisplayForm
Out[11]//DisplayForm=
x
— + z
y
```

\(and \) specify grouping, but are not displayed as explicit parentheses.

```
In[12]:= \(x \/ \(y + z\)\) // DisplayForm
Out[12]//DisplayForm=
  x
─────
y + z
```

The inner \(and \) lead to the construction of a RowBox.

```
In[13]:= \(x \/ \(y + z\)\)
Out[13]= FractionBox[x, RowBox[{y, +, z}]]
```

When you type aa + bb as input to *Mathematica*, the first thing that happens is that aa, + and bb are recognized as being separate "tokens". The same separation into tokens is done when boxes are constructed from input enclosed in \(… \). However, inside the boxes each token is given as a string, rather than in its raw form.

The RowBox has aa, + and bb broken into separate strings.

```
In[14]:= \(aa + bb\) // FullForm
Out[14]//FullForm= RowBox[List["aa", "+", "bb"]]
```

Spaces around the + are by default discarded.

```
In[15]:= \(aa + bb\) // FullForm
Out[15]//FullForm= RowBox[List["aa", "+", "bb"]]
```

Here two nested RowBox objects are formed.

```
In[16]:= \(aa + bb / cc\) // FullForm
Out[16]//FullForm= RowBox[List["aa", "+", RowBox[List["bb", "/", "cc"]]]]
```

The same box structure is formed even when the string given does not correspond to a complete *Mathematica* expression.

```
In[17]:= \(aa + bb /\) // FullForm
Out[17]//FullForm= RowBox[List["aa", "+", RowBox[List["bb", "/"]]]]
```

Within `\(...\)` sequences, you can set up certain kinds of boxes by using backslash notations such as `\^` and `\@`. But for other kinds of boxes, you need to give ordinary *Mathematica* input, prefaced by `*`.

This constructs a `GridBox`.

```
In[18]:= \(\*GridBox[{{"a", "b"}, {"c", "d"}}]\) // DisplayForm
Out[18]//DisplayForm=
a b
c d
```

This constructs a `StyleBox`.

```
In[19]:= \(\*StyleBox["text", FontWeight -> "Bold"]\) // DisplayForm
Out[19]//DisplayForm=
text
```

`*` in effect acts like an escape: it allows you to enter ordinary *Mathematica* syntax even within a `\(...\)` sequence. Note that the input you give after a `*` can itself in turn contain `\(...\)` sequences.

You can alternate nested `*` and `\(...\)`. Explicit quotes are needed outside of `\(...\)`.

```
In[20]:= \(x + \*GridBox[{{"a", "b"},
            {\(c \^ 2\), \(d \/ \*GridBox[{{"x", "y"}, {"x", "y"}}]\)}}]\) // DisplayForm
Out[20]//DisplayForm=
```

$$x + \begin{array}{cc} a & b \\ c^2 & \frac{d}{\begin{array}{cc} x & y \\ x & y \end{array}} \end{array}$$

`\!\(`*input*`\)`	interpret input in the current form
`\!\(`*form*`` \` ``*input*`\)`	interpret input using the specified form

Controlling the way input is interpreted.

In a `StandardForm` cell, this will be interpreted in `StandardForm`, yielding a product.

```
In[21]:= \!\(c (1 + x)\)
Out[21]= c (1 + x)
```

The backslash backquote sequence tells *Mathematica* to interpret this in `TraditionalForm`.

```
In[22]:= \!\(TraditionalForm \` c (1 + x)\)
Out[22]= c[1 + x]
```

When you copy the contents of a cell from a notebook into a program such as a text editor, no explicit backslash backquote sequence is usually included. But if you expect to paste what you get back into a cell of a different type from the one it came from, then you will typically need to include a backslash backquote sequence in order to ensure that everything is interpreted correctly.

Converting between Strings, Boxes and Expressions

`ToString[`*expr*`,`*form*`]`	create a string representing the specified textual form of *expr*
`ToBoxes[`*expr*`,`*form*`]`	create boxes representing the specified textual form of *expr*
`ToExpression[`*input*`,`*form*`]`	create an expression by interpreting a string or boxes as input in the specified textual form
`ToString[`*expr*`]`	create a string using `OutputForm`
`ToBoxes[`*expr*`]`	create boxes using `StandardForm`
`ToExpression[`*input*`]`	create an expression using `StandardForm`

Converting between strings, boxes and expressions.

Here is a simple expression.

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

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

This gives the `InputForm` of the expression as a string.

```
In[2]:= ToString[x^2 + y^2, InputForm]
Out[2]= x^2 + y^2
```

In `FullForm` explicit quotes are shown around the string.

```
In[3]:= FullForm[%]
Out[3]//FullForm= "x^2 + y^2"
```

This gives a string representation for the `StandardForm` boxes that correspond to the expression.

```
In[4]:= ToString[x^2 + y^2, StandardForm] // FullForm
Out[4]//FullForm= "\!\(x\^2 + y\^2\)"
```

ToBoxes yields the boxes themselves.

```
In[5]:= ToBoxes[x^2 + y^2, StandardForm]
Out[5]= RowBox[{SuperscriptBox[x, 2], +, SuperscriptBox[y, 2]}]
```

In generating data for files and external programs, it is sometimes necessary to produce two-dimensional forms which use only ordinary keyboard characters. You can do this using `OutputForm`.

This produces a string which gives a two-dimensional rendering of the expression, using only ordinary keyboard characters.

```
In[6]:= ToString[x^2 + y^2, OutputForm]
Out[6]=  2    2
        x  + y
```

The string consists of two lines, separated by an explicit `\n` newline.

```
In[7]:= FullForm[%]
Out[7]//FullForm= " 2    2\nx  + y"
```

The string looks right only in a monospaced font.

```
In[8]:= Style[%, FontFamily -> "Times"]
Out[8]= 2  2
        x + y
```

If you operate only with one-dimensional structures, you can effectively use `ToString` to do string manipulation with formatting functions.

This generates a string corresponding to the `OutputForm` of `StringForm`.

```
In[9]:= ToString[StringForm["``^10 = ``", 4, 4^10]] // InputForm
Out[9]//InputForm= "4^10 = 1048576"
```

`InputForm`	strings corresponding to keyboard input
`StandardForm`	strings or boxes corresponding to standard two-dimensional input (default)
`TraditionalForm`	strings or boxes mimicking traditional mathematical notation

Some forms handled by `ToExpression`.

This creates an expression from an InputForm string.

```
In[10]:= ToExpression["x^2 + y^2"]
```

Out[10]= $x^2 + y^2$

This creates the same expression from StandardForm boxes.

```
In[11]:= ToExpression[RowBox[{SuperscriptBox["x", "2"], "+", SuperscriptBox["y", "2"]}]]
```

Out[11]= $x^2 + y^2$

In TraditionalForm these are interpreted as functions.

```
In[12]:= ToExpression["c(1 + x) + log(x)", TraditionalForm]
Out[12]= c[1 + x] + Log[x]
```

ToExpression[*input*, *form*, *h*]	create an expression, then wrap it with head *h*

Creating expressions wrapped with special heads.

This creates an expression, then immediately evaluates it.

```
In[13]:= ToExpression["1 + 1"]
Out[13]= 2
```

This creates an expression using StandardForm rules, then wraps it in Hold.

```
In[14]:= ToExpression["1 + 1", StandardForm, Hold]
Out[14]= Hold[1 + 1]
```

You can get rid of the Hold using ReleaseHold.

```
In[15]:= ReleaseHold[%]
Out[15]= 2
```

SyntaxQ["*string*"]	determine whether a string represents syntactically correct *Mathematica* input
SyntaxLength["*string*"]	find out how long a sequence of characters starting at the beginning of a string is syntactically correct

Testing correctness of strings as input.

`ToExpression` will attempt to interpret any string as *Mathematica* input. But if you give it a string that does not correspond to syntactically correct input, then it will print a message, and return `$Failed`.

This is not syntactically correct input, so `ToExpression` does not convert it to an expression.

```
In[16]:= ToExpression["1 +/+ 2"]

ToExpression::sntx: Syntax error in or before "1 +/+ 2".
                                                    ^

Out[16]= $Failed
```

`ToExpression` requires that the string correspond to a *complete Mathematica* expression.

```
In[17]:= ToExpression["1 + 2 + "]

ToExpression::sntxi:   Incomplete expression; more input is needed.

Out[17]= $Failed
```

You can use the function `SyntaxQ` to test whether a particular string corresponds to syntactically correct *Mathematica* input. If `SyntaxQ` returns `False`, you can find out where the error occurred using `SyntaxLength`. `SyntaxLength` returns the number of characters which were successfully processed before a syntax error was detected.

`SyntaxQ` shows that this string does not correspond to syntactically correct *Mathematica* input.

```
In[18]:= SyntaxQ["1 +/+ 2"]

Out[18]= False
```

`SyntaxLength` reveals that an error was detected after the third character in the string.

```
In[19]:= SyntaxLength["1 +/+ 2"]

Out[19]= 3
```

Here `SyntaxLength` returns a value greater than the length of the string, indicating that the input was correct so far as it went, but needs to be continued.

```
In[20]:= SyntaxLength["1 + 2 + "]

Out[20]= 10
```

The Syntax of the *Mathematica* Language

Mathematica uses various syntactic rules to interpret input that you give, and to convert strings and boxes into expressions. The version of these rules that is used for `StandardForm` and `InputForm` in effect defines the basic *Mathematica* language. The rules used for other forms, such as `TraditionalForm`, follow the same overall principles, but differ in many details.

`a`, `xyz`, $\alpha\beta\gamma$	symbols
`"some text"`, `"α+β"`	strings
`123.456`, $3.\times10^{45}$	numbers
`+`, `->`, `≠`	operators
`(*comment*)`	input to be ignored

Types of tokens in the *Mathematica* language.

When you give text as input to *Mathematica*, the first thing that *Mathematica* does is to break the text into a sequence of *tokens*, with each token representing a separate syntactic unit.

Thus, for example, if you give the input `xx+yy-zzzz`, *Mathematica* will break this into the sequence of tokens `xx`, `+`, `yy`, `-` and `zzzz`. Here `xx`, `yy` and `zzzz` are tokens that correspond to symbols, while `+` and `-` are operators.

Operators are ultimately what determine the structure of the expression formed from a particular piece of input. The *Mathematica* language involves several general classes of operators, distinguished by the different positions in which they appear with respect to their operands.

prefix	`!x`	`Not[x]`
postfix	`x!`	`Factorial[x]`
infix	`x+y+z`	`Plus[x,y,z]`
matchfix	`{x,y,z}`	`List[x,y,z]`
compound	`x/:y=z`	`TagSet[x,y,z]`
overfix	$\hat{x}$	`OverHat[x]`

Examples of classes of operators in the *Mathematica* language.

Operators typically work by picking up operands from definite positions around them. But when a string contains more than one operator, the result can in general depend on which operator picks up its operands first.

Thus, for example, a * b + c could potentially be interpreted either as (a * b) + c or as a * (b + c) depending on whether * or + picks up its operands first.

To avoid such ambiguities, *Mathematica* assigns a *precedence* to each operator that can appear. Operators with higher precedence are then taken to pick up their operands first.

Thus, for example, the multiplication operator * is assigned higher precedence than +, so that it picks up its operands first, and a * b + c is interpreted as (a * b) + c rather than a * (b + c).

The * operator has higher precedence than +, so in both cases `Times` is the innermost function.

```
In[1]:= {FullForm[a * b + c], FullForm[a + b * c]}
Out[1]= {Plus[Times[a, b], c], Plus[a, Times[b, c]]}
```

The // operator has rather low precedence.

```
In[2]:= a * b + c // f
Out[2]= f[a b + c]
```

The @ operator has high precedence.

```
In[3]:= f@a * b + c
Out[3]= c + b f[a]
```

Whatever the precedence of the operators you are using, you can always specify the structure of the expressions you want to form by explicitly inserting appropriate parentheses.

Inserting parentheses makes `Plus` rather than `Times` the innermost function.

```
In[4]:= FullForm[a * (b + c)]
Out[4]//FullForm= Times[a, Plus[b, c]]
```

Extensions of symbol names	*x*_ , #2 , *e*::*s* , etc.
Function application variants	*e*[*e*] , *e*@@*e* , etc.
Power-related operators	$\sqrt{e}$, *e*^*e* , etc.
Multiplication-related operators	∇*e* , *e*/*e* , *e*⊗*e* , *ee* , etc.
Addition-related operators	*e*⊕*e* , *e*+*e* , *e*⋃*e* , etc.
Relational operators	*e*==*e* , *e*∼*e* , *e*⩽*e* , *e*⧏*e* , *e*∈*e* , etc.
Arrow and vector operators	*e*⟶*e* , *e*↗*e* , *e*⇌*e* , *e*⥟*e* , etc.
Logic operators	$\forall_e e$, *e*&&*e* , *e*⋁*e* , *e*⊢*e* , etc.
Pattern and rule operators	*e*.. , *e*|*e* , *e*->*e* , *e*/.*e* , etc.
Pure function operator	*e*&
Assignment operators	*e*=*e* , *e*:=*e* , etc.
Compound expression	*e*;*e*

Outline of operators in order of decreasing precedence.

The table in "Operator Input Forms" gives the complete ordering by precedence of all operators in *Mathematica*. Much of this ordering, as in the case of * and +, is determined directly by standard mathematical usage. But in general the ordering is simply set up to make it less likely for explicit parentheses to have to be inserted in typical pieces of input.

Operator precedences are such that this requires no parentheses.

```
In[5]:= ∀x ∃y x ⊗ y ≻ y ⋀ m ≠ 0 ⇒ n ⧐̸ m
Out[5]= Implies[∀x (∃y x⊗y ≻ y) && m ≠ 0, n ⧐̸ m]
```

`FullForm` shows the structure of the expression that was constructed.

```
In[6]:= FullForm[%]
Out[6]//FullForm= Implies[And[ForAll[x, Exists[y, Succeeds[CircleTimes[x, y], y]]], Unequal[m, 0]],
  NotRightTriangleBar[n, m]]
```

Note that the first and second forms here are identical; the third requires explicit parentheses.

```
In[7]:= {x -> #^2 &, (x -> #^2) &, x -> (#^2 &)}
Out[7]= {x → #1^2 &, x → #1^2 &, x → (#1^2 &)}
```

flat	*x*+*y*+*z*	*x*+*y*+*z*
left grouping	*x*/*y*/*z*	(*x*/*y*)/*z*
right grouping	*x*^*y*^*z*	*x*^(*y*^*z*)

Types of grouping for infix operators.

`Plus` is a `Flat` function, so no grouping is necessary here.

```
In[8]:= FullForm[a + b + c + d]
Out[8]//FullForm= Plus[a, b, c, d]
```

`Power` is not `Flat`, so the operands have to be grouped in pairs.

```
In[9]:= FullForm[a^b^c^d]
Out[9]//FullForm= Power[a, Power[b, Power[c, d]]]
```

The syntax of the *Mathematica* language is defined not only for characters that you can type on a typical keyboard, but also for all the various special characters that *Mathematica* supports.

Letters such as γ, $\mathcal{L}$ and $\aleph$ from any alphabet are treated just like ordinary English letters, and can for example appear in the names of symbols. The same is true of letter-like forms such as ∞, $\hbar$ and ⅃.

But many other special characters are treated as operators. Thus, for example, $\oplus$ and ⥿ are infix operators, while $\neg$ is a prefix operator, and $\langle$ and $\rangle$ are matchfix operators.

$\oplus$ is an infix operator.

```
In[10]:= a ⊕ b ⊕ c // FullForm
Out[10]//FullForm= CirclePlus[a, b, c]
```

$\times$ is an infix operator which means the same as `*`.

```
In[11]:= a × a × a × b × b × c
Out[11]= a^3 b^2 c
```

Some special characters form elements of fairly complicated compound operators. Thus, for example, $\int f \, ⅆx$ contains the compound operator with elements $\int$ and ⅆ.

The $\int$ and ⅆ form parts of a compound operator.

```
In[12]:= ∫ k[x] ⅆx // FullForm
Out[12]//FullForm= Integrate[k[x], x]
```

No parentheses are needed here: the "inner precedence" of $\int \ldots d$ is lower than `Times`.

```
In[13]:= ∫ a[x] b[x] ⅆx + c[x]
Out[13]= c[x] + ∫ a[x] b[x] ⅆx
```

Parentheses are needed here, however.

```
In[14]:= ∫ (a[x] + b[x]) ⅆx + c[x]
Out[14]= c[x] + ∫ (a[x] + b[x]) ⅆx
```

Input to *Mathematica* can be given not only in the form of one-dimensional strings, but also in the form of two-dimensional boxes. The syntax of the *Mathematica* language covers not only one-dimensional constructs but also two-dimensional ones.

This superscript is interpreted as a power.

In[15]:= x^{a+b}

Out[15]= x^{a+b}

$\partial_x f$ is a two-dimensional compound operator.

In[16]:= $\partial_x x^n$

Out[16]= $n\,x^{-1+n}$

$\sum$ is part of a more complicated two-dimensional compound operator.

In[17]:= $\sum_{n=1}^{\infty} \frac{1}{n^s}$

Out[17]= `Zeta[s]`

The $\sum$ operator has higher precedence than $+$.

In[18]:= $\sum_{n=1}^{\infty} \frac{1}{n^s} + n$

Out[18]= `n + Zeta[s]`

Operators without Built-in Meanings

When you enter a piece of input such as `2 + 2`, *Mathematica* first recognizes the `+` as an operator and constructs the expression `Plus[2, 2]`, then uses the built-in rules for `Plus` to evaluate the expression and get the result `4`.

But not all operators recognized by *Mathematica* are associated with functions that have built-in meanings. *Mathematica* also supports several hundred additional operators that can be used in constructing expressions, but for which no evaluation rules are initially defined.

You can use these operators as a way to build up your own notation within the *Mathematica* language.

The ⊕ is recognized as an infix operator, but has no predefined value.

```
In[1]:= 2 ⊕ 3 // FullForm
Out[1]//FullForm= CirclePlus[2, 3]
```

In `StandardForm`, ⊕ prints as an infix operator.

```
In[2]:= 2 ⊕ 3
Out[2]= 2 ⊕ 3
```

You can define a value for ⊕.

```
In[3]:= x_ ⊕ y_ := Mod[x + y, 2]
```

Now ⊕ is not only recognized as an operator, but can also be evaluated.

```
In[4]:= 2 ⊕ 3
Out[4]= 1
```

$x \oplus y$	`CirclePlus[x,y]`
$x \approx y$	`TildeTilde[x,y]`
$x \therefore y$	`Therefore[x,y]`
$x \leftrightarrow y$	`LeftRightArrow[x,y]`
∇x	`Del[x]`
$\square x$	`Square[x]`
$\langle x, y, \ldots \rangle$	`AngleBracket[x,y,...]`

A few *Mathematica* operators corresponding to functions without predefined values.

Mathematica follows the general convention that the function associated with a particular operator should have the same name as the special character that represents that operator.

\[Congruent] is displayed as ≡.

```
In[5]:= x ≡ y
Out[5]= x ≡ y
```

It corresponds to the function `Congruent`.

```
In[6]:= FullForm[%]
Out[6]//FullForm= Congruent[x, y]
```

x \[*name*] *y*	*name*[*x*, *y*]
\[*name*] *x*	*name*[*x*]
\[Left *name*] *x*,*y*,… \[Right *name*]	*name*[*x*, *y*, …]

The conventional correspondence in *Mathematica* between operator names and function names.

You should realize that even though the functions `CirclePlus` and `CircleTimes` do not have built-in evaluation rules, the operators ⊕ and ⊗ do have built-in precedences. "Operator Input Forms" lists all the operators recognized by *Mathematica*, in order of their precedence.

The operators ⊗ and ⊕ have definite precedences—with ⊗ higher than ⊕.

```
In[7]:= x ⊗ y ⊕ z // FullForm
Out[7]//FullForm= Mod[Plus[z, CircleTimes[x, y]], 2]
```

x_y	`Subscript[x, y]`
x_+	`SubPlus[x]`
x_-	`SubMinus[x]`
x_*	`SubStar[x]`
x^+	`SuperPlus[x]`
x^-	`SuperMinus[x]`
x^*	`SuperStar[x]`
$x^\dagger$	`SuperDagger[x]`
$\overset{y}{x}$	`Overscript[x, y]`
$\underset{y}{x}$	`Underscript[x, y]`
$\overline{x}$	`OverBar[x]`
$\vec{x}$	`OverVector[x]`
$\tilde{x}$	`OverTilde[x]`
$\hat{x}$	`OverHat[x]`
$\dot{x}$	`OverDot[x]`
$\underline{x}$	`UnderBar[x]`

Some two-dimensional forms without built-in meanings.

Subscripts have no built-in meaning in *Mathematica*.

In[8]:= $x_2 + y_2$ **// InputForm**

Out[8]//InputForm= `Subscript[x, 2] + Subscript[y, 2]`

Most superscripts are however interpreted as powers by default.

In[9]:= $x^2 + y^2$ **// InputForm**

Out[9]//InputForm= `x^2 + y^2`

A few special superscripts are not interpreted as powers.

In[10]:= $x^\dagger + y^+$ **// InputForm**

Out[10]//InputForm= `SuperDagger[x] + SuperPlus[y]`

Bar and hat are interpreted as `OverBar` and `OverHat`.

In[11]:= $\overline{x} + \hat{y}$ **// InputForm**

Out[11]//InputForm= `OverBar[x] + OverHat[y]`

Defining Output Formats

Just as *Mathematica* allows you to define how expressions should be evaluated, so also it allows you to define how expressions should be formatted for output. The basic idea is that whenever *Mathematica* is given an expression to format for output, it first calls `Format[expr]` to find out whether any special rules for formatting the expression have been defined. By assigning a value to `Format[expr]` you can therefore tell *Mathematica* that you want a particular kind of expression to be output in a special way.

This tells *Mathematica* to format `bin` objects in a special way.

```
In[1]:= Format[bin[x_, y_]] := MatrixForm[{{x}, {y}}]
```

Now `bin` objects are output to look like binomial coefficients.

```
In[2]:= bin[i + j, k]
```

```
Out[2]= ( i + j )
        (   k   )
```

Internally, however, `bin` objects are still exactly the same.

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

```
Out[3]//FullForm= bin[Plus[i, j], k]
```

`Format[`$expr_1$`]:=`$expr_2$	define $expr_1$ to be formatted like $expr_2$
`Format[`$expr_1$`,` *form*`]:=`$expr_2$	give a definition only for a particular output form

Defining your own rules for formatting.

By making definitions for `Format`, you can tell *Mathematica* to format a particular expression so as to look like another expression. You can also tell *Mathematica* to run a program to determine how a particular expression should be formatted.

This specifies that *Mathematica* should run a simple program to determine how `xrep` objects should be formatted.

```
In[4]:= Format[xrep[n_]] := StringJoin[Table["x", {n}]]
```

The strings are created when each `xrep` is formatted.

```
In[5]:= xrep[1] + xrep[4] + xrep[9]
```

```
Out[5]= x + xxxx + xxxxxxxxx
```

Internally however the expression still contains `xrep` objects.

```
In[6]:= % /. xrep[n_] -> x^n
```

Out[6]= $x + x^4 + x^9$

`Prefix[`*f*`[`*x*`],`*h*`]`	prefix form *h x*
`Postfix[`*f*`[`*x*`],`*h*`]`	postfix form *x h*
`Infix[`*f*`[`*x*`,`*y*`,…],`*h*`]`	infix form *x h y h …*
`Prefix[`*f*`[`*x*`]]`	standard prefix form *f*@*x*
`Postfix[`*f*`[`*x*`]]`	standard postfix form *x* // *f*
`Infix[`*f*`[`*x*`,`*y*`,…]]`	standard infix form *x*~*f*~*y*~*f*~…
`PrecedenceForm[`*expr*`,`*n*`]`	an object to be parenthesized with a precedence level *n*

Output forms for operators.

This prints with `f` represented by the "prefix operator" <>.

```
In[7]:= Prefix[f[x], "<>"]
Out[7]= <> x
```

Here is output with the "infix operator" <>.

```
In[8]:= s = Infix[{a, b, c}, "<>"]
Out[8]= a <> b <> c
```

By default, the "infix operator" <> is assumed to have "higher precedence" than ^, so no parentheses are inserted.

```
In[9]:= s^2
```

Out[9]= $(a <> b <> c)^2$

When you have an output form involving operators, the question arises of whether the arguments of some of them should be parenthesized. As discussed in "Special Ways to Input Expressions", this depends on the "precedence" of the operators. When you set up output forms involving operators, you can use `PrecedenceForm` to specify the precedence to assign to each operator. *Mathematica* uses integers from 1 to 1000 to represent "precedence levels". The higher the precedence level for an operator, the less it needs to be parenthesized.

Here <> is treated as an operator with precedence 100. This precedence turns out to be low enough that parentheses are inserted.

```
In[10]:= PrecedenceForm[s, 100]^2

Out[10]= (a <> b <> c)^2
```

When you make an assignment for Format[*expr*], you are defining the output format for *expr* in all standard types of *Mathematica* output. By making definitions for Format[*expr*, *form*], you can specify formats to be used in specific output forms.

This specifies the TeXForm for the symbol x.

```
In[11]:= Format[x, TeXForm] := "{\\bf x}"
```

The output format for x that you specified is now used whenever the TeX form is needed.

```
In[12]:= TeXForm[1 + x^2]

Out[12]//TeXForm= x^2+1
```

Low-Level Input and Output Rules

MakeBoxes[*expr*, *form*]	construct boxes to represent *expr* in the specified form
MakeExpression[*boxes*, *form*]	construct an expression corresponding to *boxes*

Low-level functions for converting between expressions and boxes.

MakeBoxes generates boxes without evaluating its input.

```
In[1]:= MakeBoxes[2 + 2, StandardForm]

Out[1]= RowBox[{2, +, 2}]
```

MakeExpression interprets boxes but uses HoldComplete to prevent the resulting expression from being evaluated.

```
In[2]:= MakeExpression[%, StandardForm]

Out[2]= HoldComplete[2 + 2]
```

Built into *Mathematica* are a large number of rules for generating output and interpreting input. Particularly in StandardForm, these rules are carefully set up to be consistent, and to allow input and output to be used interchangeably.

It is fairly rare that you will need to modify these rules. The main reason is that *Mathematica* already has built-in rules for the input and output of many operators to which it does not itself assign specific meanings.

Thus, if you want to add, for example, a generalized form of addition, you can usually just use an operator like ⊕ for which *Mathematica* already has built-in input and output rules.

This outputs using the ⊕ operator.

```
In[3]:= CirclePlus[u, v, w]
Out[3]= u⊕v⊕w
```

Mathematica understands ⊕ on input.

```
In[4]:= u ⊕ v ⊕ w // FullForm
Out[4]//FullForm= CirclePlus[u, v, w]
```

In dealing with output, you can make definitions for `Format[`*expr*`]` to change the way that a particular expression will be formatted. You should realize, however, that as soon as you do this, there is no guarantee that the output form of your expression will be interpreted correctly if it is given as *Mathematica* input.

If you want to, *Mathematica* allows you to redefine the basic rules that it uses for the input and output of all expressions. You can do this by making definitions for `MakeBoxes` and `MakeExpression`. You should realize, however, that unless you make such definitions with great care, you are likely to end up with inconsistent results.

This defines how `gplus` objects should be output in `StandardForm`.

```
In[5]:= gplus /: MakeBoxes[gplus[x_, y_, n_], StandardForm] :=
         RowBox[{MakeBoxes[x, StandardForm],
           SubscriptBox["⊕", MakeBoxes[n, StandardForm]], MakeBoxes[y, StandardForm]}]
```

`gplus` is now output using a subscripted ⊕.

```
In[6]:= gplus[a, b, m + n]
Out[6]= a ⊕m+n b
```

Mathematica cannot however interpret this as input.

```
In[7]:= a ⊕m+n b

   Syntax::sntxi: Incomplete expression; more input is needed.
```

This tells *Mathematica* to interpret a subscripted ⊕ as a specific piece of `FullForm` input.

```
In[8]:= MakeExpression[RowBox[{x_, SubscriptBox["⊕", n_], y_}], StandardForm] :=
         MakeExpression[RowBox[{"gplus", "[", x, ",", y, ",", n, "]"}], StandardForm]
```

Now the subscripted ⊕ is interpreted as a `gplus`.

```
In[9]:= a ⊕m+n b // FullForm
Out[9]//FullForm= gplus[a, b, Plus[m, n]]
```

When you give definitions for `MakeBoxes`, you can think of this as essentially a lower-level version of giving definitions for `Format`. An important difference is that `MakeBoxes` does not evaluate its argument, so you can define rules for formatting expressions without being concerned about how these expressions would evaluate.

In addition, while `Format` is automatically called again on any results obtained by applying it, the same is not true of `MakeBoxes`. This means that in giving definitions for `MakeBoxes` you explicitly have to call `MakeBoxes` again on any subexpressions that still need to be formatted.

- Break input into tokens.
- Strip spacing characters.
- Construct boxes using built-in operator precedences.
- Strip `StyleBox` and other boxes not intended for interpretation.
- Apply rules defined for `MakeExpression`.

Operations done on *Mathematica* input.

Generating Unstructured Output

The functions described in "Textual Input and Output Overview" determine *how* expressions should be formatted when they are printed, but they do not actually cause anything to be printed.

In the most common way of using *Mathematica* you never in fact explicitly have to issue a command to generate output. Usually, *Mathematica* automatically prints out the final result that it gets from processing input you gave. Sometimes, however, you may want to get *Mathematica* to print out expressions at intermediate stages in its operation. You can do this using the function `Print`.

`Print[`$expr_1$`,`$expr_2$`,…]`	print the $expr_i$, with no spaces in between, but with a newline (line feed) at the end

Printing expressions.

`Print` prints its arguments, with no spaces in between, but with a newline (line feed) at the end.

```
In[1]:= Print[a, b]; Print[c]

ab

c
```

This prints a table of the first five integers and their squares.

```
In[2]:= Do[Print[i, " ", i^2], {i, 5}]

1 1

2 4

3 9

4 16

5 25
```

`Print` simply takes the arguments you give, and prints them out one after the other, with no spaces in between. In many cases, you will need to print output in a more complicated format. You can do this by giving an output form as an argument to `Print`.

This prints the matrix in the form of a table.

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

1 2
3 4
```

Here the output format is specified using `StringForm`.

```
In[4]:= Print[StringForm["x = ``, y = ``", a^2, b^2]]
```

$x = a^2, y = b^2$