An Overview of *Mathematica*

General Information

OBJECTIVE: Learn the basic features, characteristics, and language structure of *Mathematica*.

The working environment for *Mathematica* is called a notebook. This Overview is an example of a *Mathematica* notebook. Each notebook is divided into cells as indicated by the blue brackets on the right margin of the page. A cell that contains executable commands is called an input cell and has a small triangle at the upper end of the cell bracket. If you start to type in a *Mathematica* notebook, by default the program understands the typing as an executable command and so will open a new input cell. To type other kinds of cells, like the titles above or the text in this cell, for example, pull down the Format menu at the top of the page and select Style.

You can type several commands in a single cell by separating the commands with semicolons and/or by pressing the return key (i.e., Enter on the QWERTY keypad). If a semicolon follows a command, its output is suppressed. If you wish to type a command in a new cell, you can press the down arrow key or mouse click a new cell. A new cell will start where a horizontal line appears across the notebook page.

To execute a *Mathematica* command or group of commands in a cell, place the cursor anywhere in the cell containing the command(s) to be executed and click the left mouse button. Typing Shift-Enter on the QWERTY keyboard or pressing Enter on the numeric keypad executes the command(s) in the cell. All of the commands in a cell are executed in sequence from the first to the last.

In mathematical formulas, use of the times symbol (*) for multiplication is optional. However, if two or more symbol names are to be multiplied they must be separated by spaces so that *Mathematica* does not interpret the symbols together as a different symbol name. If a symbol pre-multiplies a constant, it must also be separated from the constant by a space for the same reason.

There are several palettes available for typing standard mathematical forms in commands and in text cells. These are found in the File (Palettes) pull-down menu. The BasicInput palette is one of the most useful.

The Help pull-down menu is extremely useful when using *Mathematica*. Formats for *Mathematica* commands are usually given with several examples that can be executed right in the Help window.

In the Introduction section of each module, there is a set of **Technology Guidelines** contained in a cell that is normally closed. Until you get used to *Mathematica*, you should open the cell and read the guidelines before continuing with the rest of the module. The following shows what the guidelines say.

▪ Technology Guidelines

NOTE: If you have just finished a module, restart *Mathematica* or close the *Kernel* before executing a new module.
TO OPEN CELLS, put your cursor on the right cell bracket and double click.
INITIALIZATION CELLS
When asked if you want to ". . . automatically evaluate all the initialization cells in the notebook . . . ," respond by pressing the "Yes" button.
TO STOP AN EXECUTION
Select the *Kernel* pull-down menu and click on *Abort Evaluation.*
ORDER OF EXECUTION
Execute cells in the order given. Do not skip any Input cells within a given notebook.
SAVING NOTEBOOKS
You can save anytime to any directory you choose, and it is wise to save often.
However, before you do your final save, delete all your output by selecting the

Delete All Output selection under the *Kernel* pull-down menu.

EXPERIENCING MAJOR PROBLEMS

Save if appropriate, then shut down *Mathematica* and start it up again.

Built-In Commands

Mathematica commands consist of a word or a string of words followed immediately (i.e., no spaces) by a series of arguments enclosed in a pair of square brackets **[ ]**. The following are examples.

In[279]:=

```
Sin[Pi / 4]
```

Out[279]=

$$\frac{1}{\sqrt{2}}$$

In[280]:=

```
Sqrt[16]
```

Out[280]=

```
4
```

In[281]:=

```
Abs[x]
```

Out[281]=

```
Abs[x]
```

In[282]:=

```
ExpandNumerator[(a - b)^2 / (c + d)^3]
```

Out[282]=

$$\frac{a^2 - 2\,a\,b + b^2}{(c + d)^3}$$

In[283]:=

```
Solve[x^2 == 2, x]
```

Out[283]=

$$\{\{x \to -\sqrt{2}\}, \{x \to \sqrt{2}\}\}$$

Note that the symbol **==** (pronounced "double equals") is used to represent mathematical equality. This is necessary because the symbol **=** is reserved for making assignments to symbol names. (See the **Assignment Commands** section later in this Overview.)

Words in the command string typically begin with a single capital letter that is followed by lowercase letters. The command in the fourth example consists of two words. In most multiple-word commands, each word begins with a capital letter, and there are no spaces between the words. Also, there is no space between the word in the command string and the opening square bracket that begins the argument list.

In the first four preceding commands, the list of arguments contains only one item, whereas, in the **Solve[ ]** command, the list of arguments contains two items, the equation to solve and the symbol that is to be isolated in the solution. When a command has

more than one argument, they are separated by commas.

Lists

Another important structure in *Mathematica* is the list. A list is any ordered array of things. In *Mathematica*, lists are enclosed in a pair of curly brackets **{ }**, and the items in the list are separated by commas. The following are examples.

In[284]:=

```
{a, b, c, d}
```

Out[284]=

```
{a, b, c, d}
```

In[285]:=

```
{a, a^2, a^3, a^4}
```

Out[285]=

$$\{a, a^2, a^3, a^4\}$$

In[286]:=

```
{{1, 2}, {2, 3}, {3, 4}}
```

Out[286]=

```
{{1, 2}, {2, 3}, {3, 4}}
```

The last list consists of three items, and each of these items is a list of two items.

A list of lists can also be thought of as a matrix or an array. The **MatrixForm[ ]** command shows a list of lists in more standard matrix form.

In[287]:=

```
MatrixForm[{{1, 2}, {2, 3}, {3, 4}}]
```

Out[287]//MatrixForm=

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{pmatrix}$$

Each list inside the list is a row of the matrix.

Note that while the MatrixForm of a list makes it more readable, this form won't work in calculations. For use in calculations, a list must be in the curly bracket { } form.

Another useful way to display and view the information in a list is with the **TableForm[ ]** command.

In[288]:=

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

Out[288]//TableForm=

1 2 3 4 5
6 7 8 9 10

Row and column labels can be added with a **TableHeadings** specification inside the **TableForm[ ]** command.

In[289]:=

```
TableForm[{{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}},
 TableHeadings →
  {{"Row A", "Row B"},
   {"Col 1", "Col 2", "Col 3", "Col 4",
    "Col 5"}}]
```

Out[289]//TableForm=

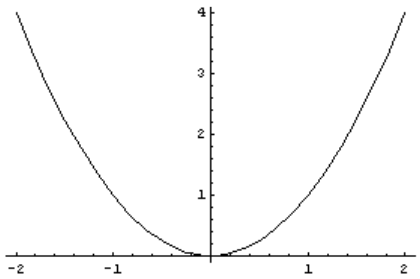
	Col 1	Col 2	Col 3	Col 4	Col 5
Row A	1	2	3	4	5
Row B	6	7	8	9	10

Like the **MatrixForm** of a list, the **TableForm** cannot be used in calculations. For use in calculations, a list must be in curly bracket { } form.

Sometimes one or more of the arguments of a *Mathematica* command is a list. For example, the following **Plot[ ]** command has two arguments, the first is the function to plot, and the second is a list that specifies the domain for the graph.

In[290]:=

```
Plot[x^2, {x, -2, 2}]
```



Out[290]=

```
- Graphics -
```

The **Table[ ]** command is very useful for generating a list when you know a formula for the elements in the list, as in the following examples.

In[291]:=

```
Table[2*i, {i, 1, 5}]
```

Out[291]=

```
{2, 4, 6, 8, 10}
```

In[292]:=

```
Table[2, {i, 1, 10}]
```

Out[292]=

```
{2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
```

In the **Table[ ]** command, the first argument is a function of the index variable. This function generates the elements in the list for the corresponding values of the index variable; the second argument is called an iterator. An iterator is a list that contains the symbol for the index variable, the beginning value, the ending value, and an optional increment or step size. If no value is specified for the increment, the default value is 1. Iterators are used in many other *Mathematica* commands.

In the second example above, the index **i** is not used in the formula. You can use an alternate form for this second case, as shown in the next example. For the iterator, you simply specify the number of elements in the list.

In[293]:=

```
Table[2, {10}]
```

Out[293]=

```
{2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
```

You can generate lists of lists in a variety of ways. For example, the formula for the elements in the list can be a function of two indices.

In[294]:=

```
Table[i * j, {i, 1, 5}, {j, -2, 2}]
```

Out[294]=

```
{{-2, -1, 0, 1, 2}, {-4, -2, 0, 2, 4},
 {-6, -3, 0, 3, 6}, {-8, -4, 0, 4, 8}, {-10, -5, 0, 5, 10}}
```

Or you can generate the same list like this.

In[295]:=

```
Table[{-2 * i, -i, 0, i, 2 * i}, {i, 1, 5}]
```

Out[295]=

```
{{-2, -1, 0, 1, 2}, {-4, -2, 0, 2, 4},
 {-6, -3, 0, 3, 6}, {-8, -4, 0, 4, 8}, {-10, -5, 0, 5, 10}}
```

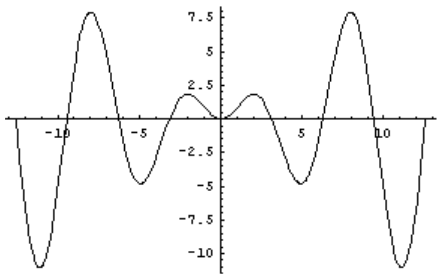
2-D Plots

There are numerous two-dimensional plot commands in *Mathematica* that we use to generate graphical displays of functions and data.

The **Plot[ ]** command is used to graph functions of the form y=f(x).

In[296]:=

```
Plot[x * Sin[x], {x, -4 * Pi, 4 * Pi}]
```



Out[296]=

```
- Graphics -
```

The first argument of the **Plot[ ]** command is the function to graph, and the second is a list that contains the independent variable,

plus the left and right ends of the domain over which you wish to graph the function. These first two arguments are required, but, as you will see, you can add other specifications.

Note that a semicolon (;) at the end of a *Mathematica* command normally suppresses display of the output from the command. In the first of the two commands that follow we omit the semicolon and in the second one we do not.

In[297]:=

```
√4
```

Out[297]=

```
2
```

In[298]:=

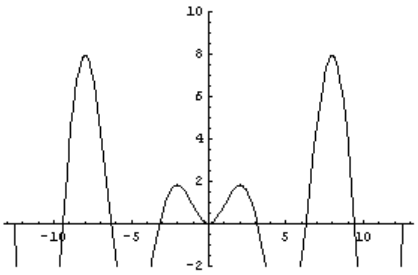
```
√4 ;
```

After a graphics command, however, a semicolon does not suppress printing of the graphics. It is usually best to include the semicolon after graphics commands to save memory. We do so in the remainder of this part.

By default, *Mathematica* selects the range (vertical limits) for the graph. However, you can override this by specifying your own **PlotRange** as follows.

In[299]:=

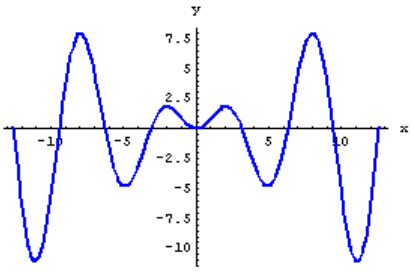
```
Plot[x * Sin[x], {x, -4 * Pi, 4 * Pi},
 PlotRange → {-2, 10}];
```



You can format a graph by specifying a wide variety of options after the first two required arguments in the **Plot[ ]** command.

In[300]:=

```
Plot[x * Sin[x], {x, -4 * Pi, 4 * Pi},
 AxesLabel → {"x", "y"},
 PlotStyle → {RGBColor[0, 0, 1],
   Thickness[0.01]}];
```

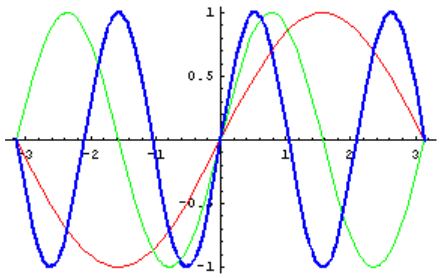


The **PlotStyle** option is used to change the attributes of a graph, such as the color and line thickness, as shown in the preceding cell. If you want help in selecting a color, place the cursor where you want the **RGBColor[ ]** indicator, or highlight an existing indicator (like the one in the command above), and then pull down the Input menu and select Color-Selector. When you choose a color from the palette, the corresponding **RGBColor[ ]** indicator is inserted where you placed the cursor. The arguments of **RGBColor[ ]** are three numbers that can each range from 0 to 1. The first number specifies the amount of red in the color, the second number the amount of green, and the third number the amount of blue. For example, **RGBColor[1, 0, 0]** gives full red and no green or blue.

The following command plots three functions on the same graph; each curve is a different color, and the third one is thicker.

In[301]:=

```
Plot[{Sin[x], Sin[2*x], Sin[3*x]},
 {x, -Pi, Pi},
 PlotStyle→{{RGBColor[1, 0, 0]},
   {RGBColor[0, 1, 0]},
   {RGBColor[0, 0, 1], Thickness[0.01]}}];
```



Note that inside the **PlotStyle** list, the attributes for each of the three graphs are specified in separate lists.

The **ListPlot[ ]** command is used to plot a list of values.

In[302]:=

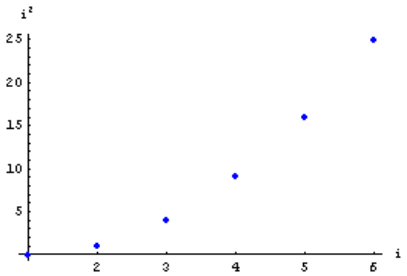
```
plotvalues = Table[i^2, {i, 0, 5}]
```

Out[302]=

```
{0, 1, 4, 9, 16, 25}
```

In[303]:=

```
ListPlot[plotvalues,
 AxesLabel→{"i", "i^2"},
 PlotStyle→{RGBColor[0, 0, 1],
   PointSize[0.02]}];
```



In the preceding example, the *x*-coordinate values of the plot points are taken to be 1, 2, . . . by default.

You can also use the **ListPlot[ ]** command to plot lists of values where both coordinates are specified.

In[304]:=

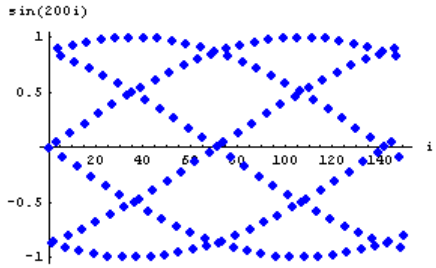
```
plotvalues = Table[{i, Sin[200. * i]},
  {i, 0, 150}]
```

Out[304]=

```
{{0, 0}, {1, -0.873297}, {2, -0.850919}, {3, 0.0441824}, {4, 0.89397},
 {5, 0.82688}, {6, -0.0882786}, {7, -0.912896}, {8, -0.801225}, {9, 0.132202},
 {10, 0.93004}, {11, 0.774005}, {12, -0.175868}, {13, -0.945367},
 {14, -0.745274}, {15, 0.21919}, {16, 0.958847}, {17, 0.715087}, {18, -0.262084},
 {19, -0.970455}, {20, -0.683504}, {21, 0.304466}, {22, 0.980168},
 {23, 0.650585}, {24, -0.346254}, {25, -0.987966}, {26, -0.616397},
 {27, 0.387365}, {28, 0.993835}, {29, 0.581004}, {30, -0.42772}, {31, -0.997763},
 {32, -0.544476}, {33, 0.467239}, {34, 0.999742}, {35, 0.506885}, {36, -0.505846},
 {37, -0.999769}, {38, -0.468305}, {39, 0.543465}, {40, 0.997843},
 {41, 0.428809}, {42, -0.580022}, {43, -0.993968}, {44, -0.388476},
 {45, 0.615447}, {46, 0.988152}, {47, 0.347385}, {48, -0.649669},
 {49, -0.980406}, {50, -0.305614}, {51, 0.682623}, {52, 0.970746},
 {53, 0.263247}, {54, -0.714244}, {55, -0.959189}, {56, -0.220366},
 {57, 0.744469}, {58, 0.945759}, {59, 0.177055}, {60, -0.773241}, {61, -0.930482},
 {62, -0.133397}, {63, 0.800503}, {64, 0.913388}, {65, 0.0894796},
 {66, -0.826201}, {67, -0.894509}, {68, -0.045387}, {69, 0.850285},
 {70, 0.873884}, {71, 0.00120577}, {72, -0.872709}, {73, -0.851552},
 {74, 0.0429778}, {75, 0.893429}, {76, 0.827557}, {77, -0.0870775},
 {78, -0.912403}, {79, -0.801946}, {80, 0.131007}, {81, 0.929596},
 {82, 0.774768}, {83, -0.174681}, {84, -0.944973}, {85, -0.746077},
 {86, 0.218013}, {87, 0.958504}, {88, 0.71593}, {89, -0.26092}, {90, -0.970164},
 {91, -0.684383}, {92, 0.303317}, {93, 0.979928}, {94, 0.651501},
 {95, -0.345122}, {96, -0.987779}, {97, -0.617346}, {98, 0.386253},
 {99, 0.993701}, {100, 0.581985}, {101, -0.426629}, {102, -0.997682},
 {103, -0.545487}, {104, 0.466172}, {105, 0.999714}, {106, 0.507924},
 {107, -0.504805}, {108, -0.999794}, {109, -0.46937}, {110, 0.542452},
 {111, 0.997921}, {112, 0.429898}, {113, -0.579039}, {114, -0.9941},
 {115, -0.389587}, {116, 0.614496}, {117, 0.988337}, {118, 0.348515},
 {119, -0.648752}, {120, -0.980643}, {121, -0.306762}, {122, 0.681742},
 {123, 0.971034}, {124, 0.26441}, {125, -0.713399}, {126, -0.959529},
 {127, -0.221542}, {128, 0.743664}, {129, 0.94615}, {130, 0.178241},
 {131, -0.772476}, {132, -0.930923}, {133, -0.134592}, {134, 0.799779},
 {135, 0.913878}, {136, 0.0906805}, {137, -0.825521}, {138, -0.895048},
 {139, -0.0465915}, {140, 0.84965}, {141, 0.87447}, {142, 0.00241155},
 {143, -0.87212}, {144, -0.852184}, {145, 0.0417731}, {146, 0.892886},
 {147, 0.828233}, {148, -0.0858762}, {149, -0.911909}, {150, -0.802665}}
```

In[305]:=

```
ListPlot[plotvalues,
 AxesLabel → {"i", "sin(200i)"},
 PlotStyle → {RGBColor[0, 0, 1],
   PointSize[0.025]}];
```

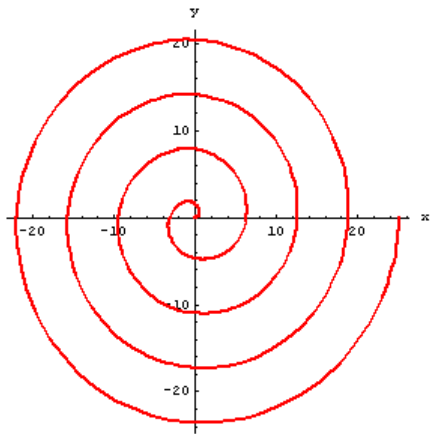


Just as you can in the **Plot[ ]** command, you can specify formatting options in the **ListPlot[ ]** command, as shown in the preceding cell.

There is one more two-dimensional plot command that you may find useful. It is **ParametricPlot[ ]**. You can use this command to plot curves that are specified in parametric form. For example, if the *x*-coordinates of the points on a curve are given by x=t cos *t* and the *y*-coordinates by y=t sin t, then the curve can be graphed like this.

In[306]:=

```
ParametricPlot[{t * Cos[t], t * Sin[t]},
 {t, 0, 8 * Pi}, AxesLabel → {"x", "y"},
 PlotStyle → {RGBColor[1, 0, 0],
   Thickness[0.010]}, AspectRatio → 1];
```



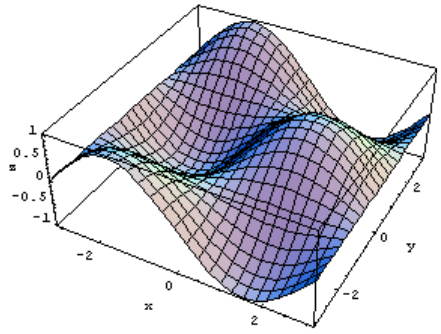
3-D Plots

There are numerous *Mathematica* commands for plotting three-dimensional functions and data. In this section, we introduce two of them and in later modules we will introduce more.

The **Plot3D[ ]** command is used to plot functions of two independent variables, that is, functions of the form z=f(x,y).

In[307]:=

```
Plot3D[Sin[x] * Cos[y], {x, -Pi, Pi},
 {y, -Pi, Pi}, AxesLabel → {"x", "y", "z"}]
```



Out[307]=

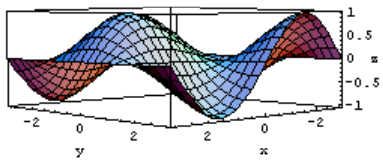
```
- SurfaceGraphics -
```

The first three arguments of the **Plot3D[ ]** command are required They are: 1) the function to graph, 2) a list that specifies the first independent variable and its limits for the graph, and 3) a second list that specifies the second independent variable and its limits. As is the case with two-dimensional graphics, you can specify a number of formatting options after the first three arguments in the command.

To change the viewpoint for the graph, you can include a **ViewPoint** indicator as an option in the **Plot3D[ ]** command.

In[308]:=

```
Plot3D[Sin[x]*Cos[y], {x, -Pi, Pi},
 {y, -Pi, Pi}, AxesLabel→{"x", "y", "z"},
 ViewPoint -> {2.354, 2.427, 0.127}]
```



Out[308]=

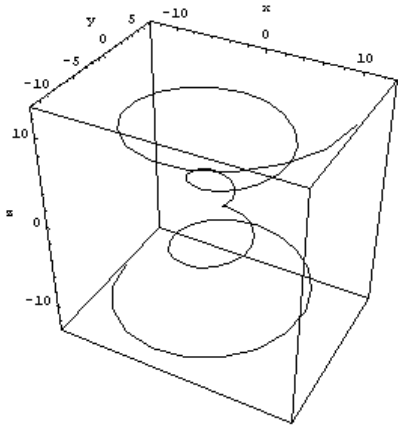
```
- SurfaceGraphics -
```

If you want help in selecting a viewpoint, place the cursor where you want the **ViewPoint → { }** indicator, or highlight an existing indicator (like the one in the command above), and then pull down the Input menu and select 3D ViewPoint Selector. When you select a viewpoint and press the "Paste" button, the corresponding **ViewPoint → { }** indicator is inserted where you placed the cursor. If you highlight an existing **ViewPoint → { }** indicator, be sure that you highlight all of it, including the part in curly brackets.

The second three-dimensional plot command is **ParametricPlot3D[ ]**. You can use this command to plot three-dimensional curves and surfaces that are specified in parametric form. For example, if the *x*-coordinates of the points on a curve are given by x=t cos *t*, the *y*-coordinates by y=t sin t, and the *z*-coordinates by z=t, then the curve can be graphed as follows.

In[309]:=

```
ParametricPlot3D[{t*Cos[t], t*Sin[t], t},
 {t, -4*Pi, 4*Pi},
 AxesLabel→{"x", "y", "z"}];
```



Assignment Commands

Mathematica has two assignment commands that are used to assign items (values, lists, commands, graphs, etc) to symbols for later use or reference. The first assignment command is the "equals" (**=**) and the other is the "colon-equals" (**:=**).

The equals assignment evaluates the expression on the right side of the **=** sign and then assigns the result of the evaluation to the symbol on the left side of the **=** sign. The following are examples.

In[310]:=

```
x = 2
```

Out[310]=

```
2
```

In[311]:=

```
y = x + 2
```

Out[311]=

```
4
```

In[312]:=

```
z = y^3
```

Out[312]=

```
64
```

You can check to see what has been assigned to a symbol by typing a question mark (?) followed by the symbol name.

In[313]:=

```
? x
```

```
Global`x

        x=2
```

In[314]:=

```
? y
```

Global`y

y=4

In[315]:=

```
? z
```

Global`z

z=64

The colon-equals does not evaluate the expression on the right side of the **:=**. It simply assigns the unevaluated expression to the symbol name on the left side of the **:=**. Whenever the symbol name is referred to later on in the session, *Mathematica* then evaluates the expression that was assigned to the symbol name. For this reason, the colon-equals is called a *delayed assignment*. The following are some examples.

In[316]:=

```
a := 2
```

In[317]:=

```
b := a + 2
```

In[318]:=

```
c := b^3
```

Note that no output is displayed when you use the colon-equals assignment. If you check to see what has been assigned to these symbols, you see that the expressions are saved rather than the results of their evaluations.

In[319]:=

```
? a
```

Global`a

a:=2

In[320]:=

```
? b
```

Global`b

b:=a+2

In[321]:=

```
? c
```

Global`c

$c := b^3$

Now, if you use any of these symbols, the expression is evaluated and the result of the evaluation is used in place of the symbol. Here are some examples.

In[322]:=

```
a
```

Out[322]=

```
2
```

In[323]:=

```
3 * b^2
```

Out[323]=

```
48
```

In[324]:=

```
c
```

Out[324]=

```
64
```

In[325]:=

```
w = a + b + c
```

Out[325]=

```
70
```

The **Clear[ ]** command removes assignments that you have made to symbols.

In[326]:=

```
Clear[x]
```

If you check the symbol name, you see that after it is cleared it is treated as a symbol with no assignments.

In[327]:=

```
? x
```

```
Global`x
```

If you use the symbol in an expression it will be treated simply as a symbol with its name.

In[328]:=

```
x + 2
```

Out[328]=

```
2 + x
```

Note, however, that **y** is still **4**. This is because the **=** assignment was used to define **y,** and as a result the expression **x+2** was evaluated giving **4**, and this value was stored instead of the expression **x+2**.

In[329]:=

```
y
```

Out[329]=

```
4
```

If you want to clear **a**, **b**, and **c**, type the following command.

In[330]:=

```
Clear[a, b, c]
```

In[331]:=

```
? a
```

Global`a

In[332]:=

```
? b
```

Global`b

In[333]:=

```
? c
```

Global`c

It is extremely important to keep close track of the symbol names that you use in a *Mathematica* session. If you use a new symbol name in an expression or a command, and you are not absolutely sure that no assignments have been previously made to the symbol, you should first use the **Clear[ ]** command to erase all previous assignments to the new symbol name. Failure to do this can produce misleading and confusing results. There is more information about this in the **Symbol Definitions and Pattern Matching** section later in this Overview.

There is another item that is related to the issue of symbol-name management. If you finish a *Mathematica* notebook or one of our modules and you wish to start another one, it is best to save your work and then shut down *Mathematica* and restart it before beginning the new module or notebook. This is because *Mathematica* retains any assignments that you have made to symbol names, even after you close a notebook or module, and carries these over to the new one with the potential for conflicts. Shutting down *Mathematica* before starting a new notebook or module ensures that all symbol names are free and clear.

Special Packages

While there are many built-in commands that are available for use as soon as you open a *Mathematica* notebook, there are many more specialized commands that are contained in packages. These packages are not automatically available in a *Mathematica* notebook, and in order to use the specialized commands that they contain, you have to read them into computer memory. You do this in an input cell by typing << followed by the package name and then executing the command. Here is an example.

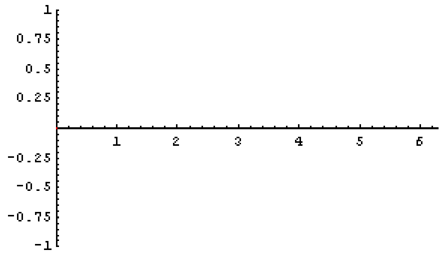
In[334]:=

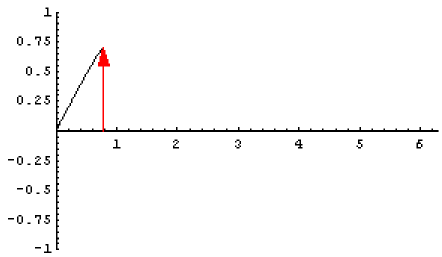
```
<< Graphics`Arrow`
```

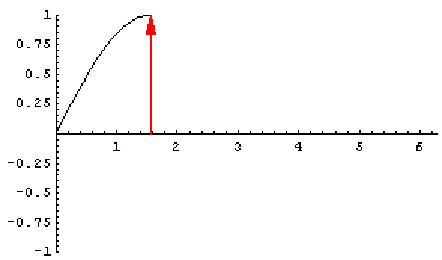
Note that the ` is a backward blip. The **Graphics`Arrow`** package contains commands for drawing arrows in graphics cells. Here is an example that illustrates the use of the **Arrow[ ]** command (highlighted in red) that is available in the **Graphics`Arrow`** package.

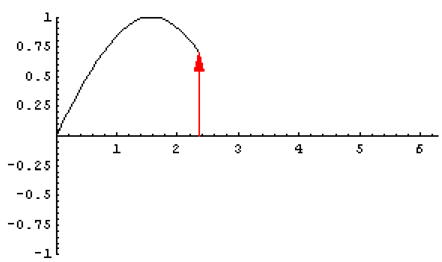
In[335]:=

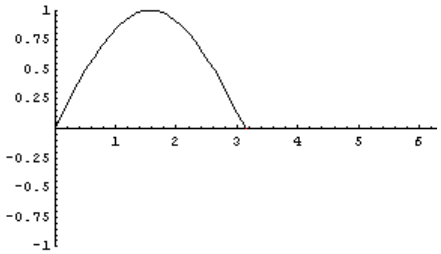
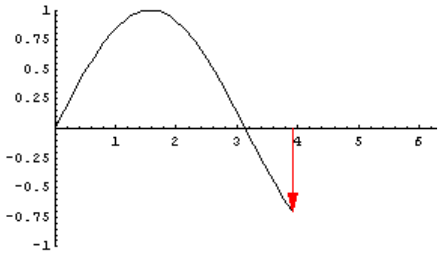
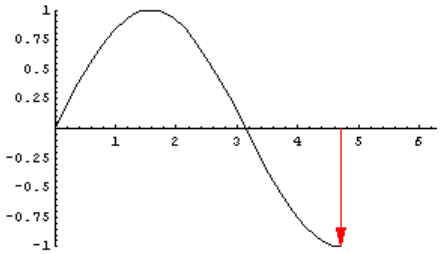
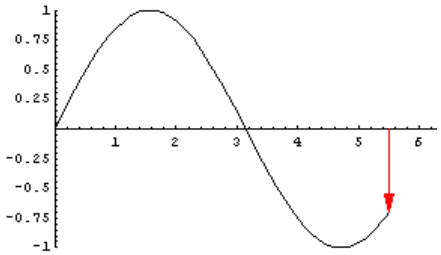
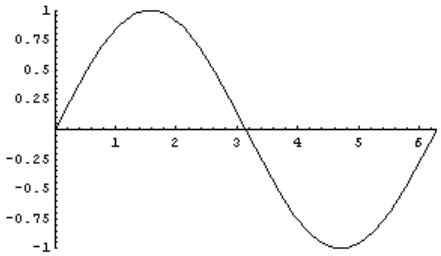
```
p1 := Plot[Sin[x], {x, -0.01, a},
  PlotRange → {{0, 2 π}, {-1, 1}},
  Epilog → {RGBColor[1, 0, 0],
    Arrow[{a, 0}, {a, Sin[a]}]}]

Do[p1, {a, 0, 2 π, π/4}];
```









The first command in the cell above uses a delayed assignment of the **Plot[ ]** command to the symbol name **p1**. Inside the **Plot[ ]** command, the **Epilog→{ }** option allows us to add things, like arrows, to the graph of the sin x. The **RGBColor[1, 0, 0]** specification will make the arrows red. The second command in the cell is the built-in **Do[ ]** command. It causes the **Plot[ ]** command that was assigned to **p1** to execute repeatedly as the value for **a** varies from 0 to 2π, incremented by steps of $\frac{\pi}{4}$.

To find out what packages are available to read into memory, pull down the Help menu, click the Add-On button, and select Standard Packages.

Before you use any of the specialized commands available in a package, be sure to read in the package first. A package needs to be read in only once during a *Mathematica* session. If you try to execute a package command before you have read in the package, the command won't work. Reading in the package after you have tried to execute one of its commands will result in error messages. To fix this problem, you must do one of two things: 1) save your work, close *Mathematica* and restart it; or 2) use the **Remove[ ]** command to remove the commands that you tried to execute before loading the package. If you choose to restart *Mathematica*, be sure to re-execute all the commands in the package that preceded the one that gave you problems. The commands in the following cells illustrate how to use **Remove[ ]**. First we try to execute a command from the **Graphics`FilledPlot`** package before we load the package. Nothing happens because *Mathematica* does not recognize the command.

In[337]:=

```
FilledPlot[Sin[x], {x, 0, 2 π}];
```

Next, we load the package and try to re-execute the command, but now we get error messages.

In[338]:=

```
<< Graphics`FilledPlot`
```

```
FilledPlot::shdw : Symbol FilledPlot appears in multiple
   contexts {Graphics`FilledPlot`, Global`}; definitions in context
   Graphics`FilledPlot` may shadow or be shadowed by other definitions. More…
```

If we then try to re-execute **FilledPlot[ ]**, it will not work.

In[339]:=

```
FilledPlot[Sin[x], {x, 0, 2 π}];
```

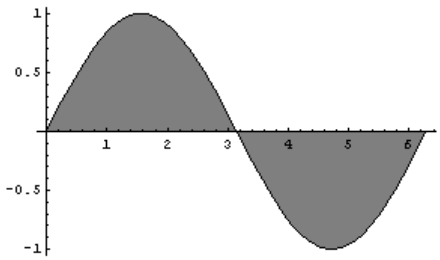
Finally, after we **Remove[ ]** our previous use of **FilledPlot[ ]**, the **PlotFilled[ ]** command should work.

In[340]:=

```
Remove[FilledPlot];
```

In[341]:=

```
FilledPlot[Sin[x], {x, 0, 2 π}];
```



Making Your Own Commands

As a *Mathematica* user and programmer, you can design and construct your own commands.

To avoid conflicts with built-in *Mathematica* commands, you should consistently begin the names of your commands with lowercase letters.

To create a command, use an assignment statement (usually with the colon-equals, **:=**). On the left of the **:=**, place the name you want to use for your command, followed by a set of single square brackets **[ ]** that enclose the list of arguments for the command. On the right of the **:=**, state the operations that your command is to perform on inputs that will be supplied for the arguments whenever the command is used. Suppose, for example, that you are going to use the function **x^2*Sin[x]** over and over again in a program. You may wish to create a new function and make it a user-defined *Mathematica* command, as illustrated by the following definition.

In[342]:=

```
myfunction[x_] := x^2 * Sin[x]
```

The underscore _ after the **x** tells *Mathematica* that this is the definition of a new function rather than a call of an existing function. The underscore _ after the **x** also tells *Mathematica* that, in the expression on the right of the **:=**, **x** is a dummy, stand-in variable that will be replaced by real input in the argument list, whenever **myfunction[ ]** is called. The following examples illustrate what *Mathematica* does with **myfunction[ ]**.

In[343]:=

```
myfunction[2]
```

Out[343]=

```
4 Sin[2]
```

In[344]:=

```
myfunction[2.0]
```

Out[344]=

```
3.63719
```

In[345]:=

```
myfunction[r]
```

Out[345]=

$$r^2 \text{Sin}[r]$$

In[346]:=

```
myfunction[a + b]
```

Out[346]=

$$(a + b)^2 \text{Sin}[a + b]$$

In[347]:=

```
myfunction[{2, 4}]
```

Out[347]=

```
{4 Sin[2], 16 Sin[4]}
```

The last example illustrates an important feature of most *Mathematica* commands: they are "listable." This means that if you put in a list as an input to a command, *Mathematica* performs the command operation(s) on each of the elements in the list and returns a list of the results. This even works for simple operations like addition, multiplication, or raising a number to a power.

In[348]:=

```
{1, 2, 3} + {4, 5, 6}
```

Out[348]=

```
{5, 7, 9}
```

In[349]:=

```
{1, 2, 3, 4} * {-2, 2, 3, 5}
```

Out[349]=

```
{-2, 4, 9, 20}
```

In[350]:=

```
{-2, 3, 5, 9}^2
```

Out[350]=

```
{4, 9, 25, 81}
```

You can design commands that perform more than one operation, typically using the **Block[ ]** command to set them up. The **Block[ ]** command requires a minimum of two arguments. The first argument is a list of local symbols (sometimes it is empty), and the second is a series of commands that you want your command to execute whenever it is called or used. The following example generates a list of decaying numbers, plots the list, and returns the sequence as an output of the module. The input is the length of the sequence, **n**.

In[351]:=

```
buildplot[n_] := Block[{plotlist},
      plotlist =
   Table[0.8^(i - 1), {i, 1, n}];
      ListPlot[plotlist,
   PlotStyle -> {PointSize[0.02]}];
      Return[plotlist];]
```

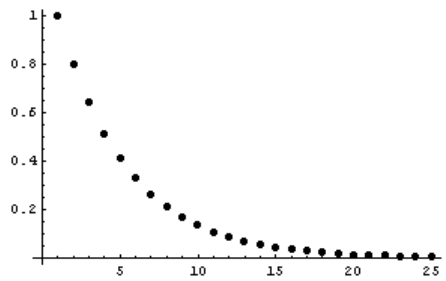
In **buildplot[ ]**, the list of local symbols contains the symbol **plotlist**. Making a symbol local to a command means that any assignments you make to the symbol inside the block will not be available outside the block. If you type **?plotlist** after you execute the command, you will find that nothing has been assigned to this symbol. We will check it out below, after we execute the **buildplot[ ]** command.

The series of commands forms the second argument of the **Block[ ]** command. Semicolons are used to separate the individual commands in the series of commands inside the block.

You can execute the **buildplot[ ]** command by using its name and providing an input value for the argument.

In[352]:=

```
buildplot[25]
```



Out[352]=

```
{1, 0.8, 0.64, 0.512, 0.4096, 0.32768, 0.262144, 0.209715,
 0.167772, 0.134218, 0.107374, 0.0858993, 0.0687195, 0.0549756,
 0.0439805, 0.0351844, 0.0281475, 0.022518, 0.0180144, 0.0144115,
 0.0115292, 0.00922337, 0.0073787, 0.00590296, 0.00472237}
```

Now we check the symbol name **plotlist**.

In[353]:=

```
?plotlist
```

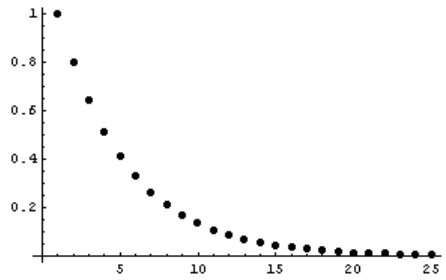
Global`plotlist

The list that was assigned to **plotlist** inside the block is not available in the global environment outside the block.

The **Return[plotlist]** command inside the block returns the values stored in the local **plotlist** symbol. If you want to save the output list of values for later use, you can assign it to a symbol name.

In[354]:=

```
decay = buildplot[25]
```



Out[354]=

```
{1, 0.8, 0.64, 0.512, 0.4096, 0.32768, 0.262144, 0.209715,
 0.167772, 0.134218, 0.107374, 0.0858993, 0.0687195, 0.0549756,
 0.0439805, 0.0351844, 0.0281475, 0.022518, 0.0180144, 0.0144115,
 0.0115292, 0.00922337, 0.0073787, 0.00590296, 0.00472237}
```

If you use the symbol name **decay**, you will get the list of numbers that the **buildplot[ ]** command returned as output.

In[355]:=

```
decay
```

Out[355]=

```
{1, 0.8, 0.64, 0.512, 0.4096, 0.32768, 0.262144, 0.209715,
 0.167772, 0.134218, 0.107374, 0.0858993, 0.0687195, 0.0549756,
 0.0439805, 0.0351844, 0.0281475, 0.022518, 0.0180144, 0.0144115,
 0.0115292, 0.00922337, 0.0073787, 0.00590296, 0.00472237}
```

Note what happens when you use the colon-equals assignment to the symbol **delaydecay** and then use the symbol **delaydecay**.

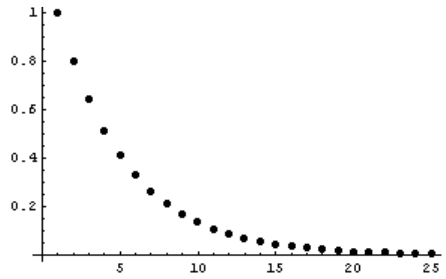
In[356]:=

```
delaydecay := buildplot[25]
```

There is no output when you execute this command because of the delayed evaluation; output is generated only when an expression is evaluated. With the delayed assignment, **buildplot[25]** is assigned to **delaydecay**, unevaluated, hence no output is displayed. However, if you type **delaydecay**, the **buildplot[25]** command that was assigned to **delaydecay** is evaluated, and output is displayed.

In[357]:=

```
delaydecay
```



Out[357]=

```
{1, 0.8, 0.64, 0.512, 0.4096, 0.32768, 0.262144, 0.209715,
 0.167772, 0.134218, 0.107374, 0.0858993, 0.0687195, 0.0549756,
 0.0439805, 0.0351844, 0.0281475, 0.022518, 0.0180144, 0.0144115,
 0.0115292, 0.00922337, 0.0073787, 0.00590296, 0.00472237}
```

Instead of evaluating **buildplot[25]** and storing the output in **delaydecay**, the colon-equal assignment stores the unevaluated **buildplot[25]** command in **delaydecay** and executes it whenever **delaydecay** is used.

Parts of Lists

Curly brackets **{ }** enclose the elements of a list, square brackets **[ ]** enclose the series of arguments in a *Mathematica* command, and regular parentheses ( ) are the symbols of inclusion for algebraic operations.

A pair of double square brackets **[[ ]]** is used to specify a part or element of a list. Consider the following examples.

In[358]:=

```
odds = {1, 3, 5, 7, 9, 11, 13, 15}
```

Out[358]=

```
{1, 3, 5, 7, 9, 11, 13, 15}
```

To retrieve the third element of **odds**, use a part specification after the name of the list.

In[359]:=

```
odds[[3]]
```

Out[359]=

```
5
```

For nested lists, that is, a list of lists, use multiple part specifications like **[[3, 2]]** to retrieve elements. The first part number points to an individual element in the outermost list, which may itself be a list. The second part number points to an element

inside the inner list. The following examples illustrate this.

In[360]:=

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

Out[360]=

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

The following command retrieves the fourth list from **oddeven**.

In[361]:=

```
oddeven[[4]]
```

Out[361]=

```
{7, 8}
```

To retrieve the second element of the third list in **oddeven**, type:

In[362]:=

```
oddeven[[3, 2]]
```

Out[362]=

```
6
```

In *Mathematica*, you can attach a part specification to just about anything. The reason for this is that almost every *Mathematica* entity (symbol, command, graphic, etc.) in its basic form consists of a list with a **Head** in front of it. To see the **Head[*list*]** form of a *Mathematica* expression, use the **FullForm[ ]** command. For example, consider the basic operation of addition.

In[363]:=

```
FullForm[a + b]
```

Out[363]//FullForm=

```
Plus[a, b]
```

This shows that the basic form for addition in *Mathematica* is a list of arguments (the addends) with the head **Plus**. The head specifies the operation that is to be performed on the arguments.

Even lists have this form.

In[364]:=

```
FullForm[odds]
```

Out[364]//FullForm=

```
List[1, 3, 5, 7, 9, 11, 13, 15]
```

In this case the head is the command, **List**.

The **Head[*list*]** format is the standard form for just about every expression or command in *Mathematica.* Sometimes, as in the case of addition, the **Head[*list*]** form is hidden from the user. When you type the more conventional form of a command, like **a+b**, *Mathematica* translates it to **Plus[a, b]**. You can type any *Mathematica* command in **Head[*list*]** form if you wish, but conventional math notation is often times more natural.

Because nearly all expressions and commands are in **Head[*list*]** form, you can attach part specifications to just about any *Mathematica* entity. Consider the following examples.

In[365]:=
```
(a + b)[[1]]
```
Out[365]=
```
a
```
In[366]:=
```
(3 * a + 2 * b)[[2]]
```
Out[366]=
```
2 b
```
In[367]:=
```
(3 * a + 2 * b)[[2, 2]]
```
Out[367]=
```
b
```

The subscript 0 retrieves the head of a command.

In[368]:=
```
(3 * a + 2 * b)[[0]]
```
Out[368]=
```
Plus
```
In[369]:=
```
(3 * a + 2 * b)[[2, 0]]
```
Out[369]=
```
Times
```
In[370]:=
```
odds[[0]]
```
Out[370]=
```
List
```

If you are not sure how an element is embedded in a command, you can use **FullForm[ ]** to find out.

In[371]:=
```
FullForm[(3 * a + 2 * b)]
```
Out[371]//FullForm=
```
Plus[Times[3, a], Times[2, b]]
```

Rules and the Slash-Dot Command

Sometimes it is desirable to evaluate an expression in *Mathematica* by temporarily replacing the symbols in the expression with

values or other symbols without permanently assigning the replacement values/symbols to the symbol names. This can be accomplished with the **/.** (pronounced slash-dot) command followed by a list of replacement rules. Note that the slash is a forward slash. The **/.** is also called the **ReplaceAll** command. The following are examples.

In[372]:=

```
Clear[x, y, f];

f = x^2 + y^2
```

Out[373]=

$$x^2 + y^2$$

In[374]:=

```
f /. {x → 1, y → 2}
```

Out[374]=

```
5
```

In[375]:=

```
f /. {x → Sin[θ], y → Cos[θ]}
```

Out[375]=

$$\text{Cos}[\theta]^2 + \text{Sin}[\theta]^2$$

You can accomplish the same thing with the following command.

In[376]:=

```
ReplaceAll[f, {x → Sin[θ], y → Cos[θ]}]
```

Out[376]=

$$\text{Cos}[\theta]^2 + \text{Sin}[\theta]^2$$

The expressions **x →1** and **y→2** are rules. (You can find the arrow on the BasicInput palette or you can type a minus sign followed by a greater than sign.) Rules are different from assignments (=) in that they do not become definitions associated with the symbols **x** and **y**, as is the case with assignments. You can verify this by checking out the definitions that are associated with **x** and **y**.

In[377]:=

```
? x
```

```
Global`x
```

In[378]:=

```
? y
```

```
Global`y
```

The symbols **x** and **y** are clear. The symbol **x** was replaced with the number **1,** and **y** with **2,** in the evaluation of **f**, but the numbers were not permanently assigned to **x** and **y**. In addition, the symbol definition of **f** is left unchanged.

In[379]:=

```
? f
```

Global`f

$$f = x^2 + y^2$$

This is in contrast to evaluation of **f** after values are assigned to the variables **x** and **y**, as shown by the following commands.

In[380]:=

```
x = 2;
```

In[381]:=

```
y = 2;
```

In[382]:=

```
f
```

Out[382]=

```
8
```

In[383]:=

```
? x
```

Global`x

x=2

In[384]:=

```
? y
```

Global`y

y=2

In this case, **x** is permanently assigned the value **1** and **y** the value **2**, that is, until we **Clear[x, y]**. The definition of **f** is still unchanged, however.

In[385]:=

```
? f
```

Global`f

$$f = x^2 + y^2$$

Here is another example.

In[386]:=

```
Clear[x, y];

equations = {x + 2 * y == 3, 3 * x + 4 * y == 2}
```

Out[387]=

```
{x + 2 y == 3, 3 x + 4 y == 2}
```

Note that **==** is used in *Mathematica* to represent mathematical equality. This is necessary because the **=** symbol is reserved for assignments to symbol names (See the **Assignment Commands** section).

In[388]:=

```
solution = Solve[equations, {x, y}]
```

Out[388]=

$$\left\{\left\{x \to -4,\ y \to \frac{7}{2}\right\}\right\}$$

In[389]:=

```
equations /. solution
```

Out[389]=

```
{{True, True}}
```

The last command verifies that the solutions found satisfy the two equations.

Retrieving Output from Commands

If you type a *Mathematica* command, the output is usually displayed directly after the command.

In[390]:=

```
Solve[x^2 == 1, x]
```

Out[390]=

```
{{x → -1}, {x → 1}}
```

Note that **==** is used in *Mathematica* to represent mathematical equality. This is necessary because the **=** symbol is reserved for assignments to symbol names (See the **Assignment Commands** section).

Typing a semicolon (;) after the command will suppress printing the output.

In[391]:=

```
Solve[x^2 == 1, x];
```

The semicolon is used for two purposes. One is to separate individual commands in a **Block[ ]** that includes a series of *Mathematica* commands, and the other is to suppress printing the output of any single command.

To see the standard *Mathematica* format for the output of the **Solve[ ]**, you can assign the output to a new symbol name and use the **FullForm[ ]** command.

In[392]:=

```
soln = Solve[x^2 == 1, x]
```

Out[392]=

```
{{x → -1}, {x → 1}}
```

In[393]:=

```
FullForm[soln]
```

Out[393]//FullForm=

```
List[List[Rule[x, -1]], List[Rule[x, 1]]]
```

The solution is a list of two lists. In each of the inner lists, there is a rule, which is itself a list with two elements. The second element of each rule is a solution. Therefore, to retrieve the two solutions, you could type the following:

In[394]:=

```
soln[[1, 1, 2]]
```

Out[394]=

```
-1
```

In[395]:=

```
soln[[2, 1, 2]]
```

Out[395]=

```
1
```

Once you know the **FullForm** of the output, you can subscript the command directly, as illustrated by the following examples.

In[396]:=

```
firstsoln = Solve[x^2 == 1, x][[1, 1, 2]]
```

Out[396]=

```
-1
```

In[397]:=

```
secondsoln = Solve[x^2 == 1, x][[2, 1, 2]]
```

Out[397]=

```
1
```

This latter approach is not as efficient as the one preceding it because it requires solving the equation twice whereas the first approach only solves it once.

Symbol Definitions and Pattern Matching

Mathematica evaluates and simplifies expressions through a process of pattern matching. To understand how this works, suppose that you want to construct a *Mathematica* function called **crazy[ ]** that subtracts two numbers and then divides the result by their sum. You can do this with the following command.

In[398]:=

```
crazy[x_, y_] := (x - y) / (x + y)
```

Once you define this new function, *Mathematica* will match it with any command of the form **crazy[item1, item2]** and perform the designated operation on the two items. For example,

In[399]:=

```
crazy[house, garage]
```

Out[399]=

$$\frac{-\text{garage} + \text{house}}{\text{garage} + \text{house}}$$

Now suppose that when both arguments are zero, you want the function **crazy[ ]** to return the value 1. You can do this by adding a new pattern definition to the description of **crazy[ ]** as follows:

In[400]:=

```
crazy[0, 0] = 1
```

Out[400]=

```
1
```

If you ask *Mathematica* what it knows about **crazy[ ]**, it will tell you that it has recorded both of your definitions.

In[401]:=

```
?crazy
```

Global`crazy

crazy[0,0]=1

$$\text{crazy}[x\_, y\_] := \frac{x - y}{x + y}$$

From now on, whenever you use the symbol **crazy**, *Mathematica* will check all of the forms that have been assigned to the symbol **crazy**, starting with the more specific forms and proceeding to the more generic, until it finds a match. If it finds a match, it performs the operation that corresponds to the matched form of **crazy**. Note in the list above that **crazy[0,0]** is more specific than **crazy[x_,y_]**; it is therefore placed first in the list of forms associated with the symbol **crazy**. *Mathematica* simply begins at the top of its list of definitions for the symbol **crazy** and searches down the list until it finds the first match. If it does not find a match, then it treats the unrecognized form as a new symbol. The new symbol is not associated with an operation and, therefore, is left unevaluated. These features are illustrated by the following examples.

In[402]:=

```
crazy[0, 0]
```

Out[402]=

```
1
```

In[403]:=

```
crazy[r, s]
```

Out[403]=

$$\frac{r - s}{r + s}$$

In[404]:=

```
crazy[x, y, z]
```

Out[404]=

```
crazy[x, y, 64]
```

***Mathematica* doesn't forget!** To understand this, consider the following scenario. You assign values to **p[1]**, **p[2]**, etc. as follows:

In[405]:=

```
Do[p[i] = i^2, {i, 1, 5}]
```

You check the assignments by typing,

In[406]:=

```
?p
```

Global`p

p[1]=1
p[2]=4
p[3]=9
p[4]=16
p[5]=25
$p[x\_] := x^3$

Later you decide to build a new function that takes an input value and cubes it. However, you forget that you already used the symbol **p** for other things and inadvertently call your new function **p**, which you define as follows:

In[407]:=

```
p[x_] := x^3
```

Now you use your new function, **p[x]**, to calculate 2^3.

In[408]:=

```
p[2]
```

Out[408]=

```
4
```

This is obviously not the output you expected. The problem is that *Mathematica* has remembered what you told it before. The form **p[2]=4** is a more specific definition than **p[x_]=x^3**, and so when you ask for **p[2]** the first match it finds is **p[2] = 4**. If you ask *Mathematica* what it knows about the symbol **p**, you will see the problem.

In[409]:=

```
?p
```

Global`p

p[1]=1
p[2]=4
p[3]=9
p[4]=16
p[5]=25
$p[x\_] := x^3$

If you were to open a new notebook or module without entirely closing down *Mathematica*, it would still retain all of the previous definitions of **p**.

Sometimes this can be very perplexing when you are working in *Mathematica.* Consequently, it is extremely important to keep

close track of the symbols you use, and when you are done with a symbol you should use the **Clear[ ]** command to erase its definitions. If you define a new function and it gives you unexpected or erroneous results, it is sometimes helpful to clear the function name, and then try to redefine it.

It is good practice to clear the name of the new function before you define it. Put the **Clear[ ]** command in the same cell as the function definition. For example,

In[410]:=

```
Clear[p];

p[x_] := x^3
```

Now the function **p** should work correctly.

In[412]:=

```
p[2]
```

Out[412]=

```
8
```

Another thing that can cause problems is defining a function using a symbol name with an argument and then later redefining the symbol without an argument before clearing the original definition. The following commands illustrate what can happen.

In[413]:=

```
Clear[g, x];

g[x_] = x^2
```

Out[414]=

$$x^2$$

In[415]:=

```
g = Sin[2 x]
```

Out[415]=

```
Sin[2 x]
```

In[416]:=

```
g[2]
```

Out[416]=

```
Sin[2 x][2]
```

In[417]:=

```
?g
```

Global`g

g=Sin[2 x]

$g[x\_] = x^2$

For the symbol **g[2]**, the first match that is found is for **g** and this symbol is replaced with **Sin[2x]** giving the output, Sin[2x][2].

Once you understand how *Mathematica* evaluates expressions by pattern matching, you will find that this can be a very powerful tool. It is the basic process that *Mathematica* uses when it evaluates expressions.

About *Mathematica*

Throughout the modules you will see hyperlink buttons that look like this:

Each of these buttons links to a text cell containing information about the *Mathematica* commands and formats that are being used nearby in the module. If you are interested in learning more about *Mathematica*, you can click the button to read the information in the text cell, or you can elect to skip over it so as not to interrupt the flow of the module. Each text cell has a Go Back link that takes you back to the location in the module where you clicked the button.

Created by Mathematica (August 8, 2005)