Getting Started

from ***Essential Mathematica for Students of Science***

© James J. Kelly, 1998

Last revised: January, 2006

This notebook has been modified for PHGN 384 by Alex Yuffa, Abram Van Der Geest; Todd G. Ruskell; updated for Mma 7.x, 8.x by DM Wood, May 2012

To open a section, double click on a cell bracket at the right side of the screen that has a hook on its bottom end like this ↿.   ⟹

The Structure of *Mathematica*

The two basic parts of the *Mathematica* system are the **front end** and the **kernel.** The front end interacts with the user while the kernel performs computations. These two parts are separate, but communicate with each other via the *MathLink* protocol.

A **notebook document**, such as this, contains commands that direct the actions of the kernel, output from those calculations, and documentation. A well-written notebook document provides a clear and structured account of your work and evolves as you refine your solution to a problem. Even though a notebook might resemble a sheaf of scratch paper during the intermediate stages of an investigation, by the time your problem approaches completion it should more closely resemble a research paper in which the problem is clearly stated and introduced, and the solution is developed in a clear and orderly fashion with enough documentation for any informed reader to follow easily. However, an important difference between a *Mathematica* notebook and either a research report or a set of notes is that the notebook is a live document in which improvements or extensions can be made without extensive rewriting.

Starting this tutorial

1. Click in this notebook and type Cntl - A

2. Select the drop - down menu item Cell -> Grouping -> Open all subgroups

3. Select the drop-down menu item Evaluation->Evaluate Notebook

Steps 1 and 2 display the entire CONTENTS of the Notebook (text comments, formatting, and commands to Mathematica), while Step 3 actually runs the notebook, allowing Mathematica to run all the commands from beginning to end.

Running the kernel

Starting the kernel

The kernel does not automatically start when you run the front end. After all, the front end is responsible for display and editing functions which do not require action by the kernel. Although you can start the kernel explicitly by using the menu sequence

Evaluation ↦ Start Kernel ↦ Local

one normally waits until there is work for the kernel to do, such as an arithmetic calculation. For example, type 2+2 into an input cell and then press the keystroke combination **Shift Enter**, which passes the input to the kernel for evaluation. Evaluation can also be initiated using the **Enter** key on the numeric keypad. After evaluation, the kernel passes the result back to the front end, which then creates an output cell containing the result.

```
2 + 2
```

Note: You will probably notice that the first evaluation that you perform may require a surprisingly long time — why doesn't the computer perform the above arithmetic evaluation much faster than you could by inspection? The evaluation is actually very fast, but before that evaluation is performed the kernel must be loaded and initialized, which is not done until an evaluation is requested. Hence, practically all of the time apparently needed to produce the first result is actually spent on the more arduous task of loading the kernel. If you are close enough to your disk drive, you will probably hear it churning as the kernel is loaded and initialized. Reevaluation of this expression will be practically instantaneous. Check by either selecting the input cell using the cell bracket or clicking within the cell and then repeating the keystroke combination **Shift Enter**. Alternatively, the **Enter** key on the numeric keypad should also execute a selection.

Note that we discussed explicitly starting the kernel using a menu sequence, we specified the local kernel. It is possible to run the front end on one machine and the kernel on another. It is also possible to run kernels on several machines simultaneously, but we will not use those more advanced techniques in this course.

Quitting the kernel

Sometimes it is necessary to interrupt or abort an evaluation or occasionally even to shut down the kernel completely. Unless the system crashes, these operations do not affect the front end and you will not lose your work. The menu sequence

Evaluation ↦ Quit Kernel ↦ Local

shuts down the kernel. *Try it now; we haven't done anything important yet.*

Shutting down the kernel removes all symbols from memory. When you restart you will probably have to execute some or all of the preceding cells before testing a revised version of the last command attempted in the previous session. If your notebook is organized well, executing a few sections rather than the entire notebook may be sufficient to establish the necessary definitions.

Saving notebooks

It is good practice with any computer program to save your work periodically, especially after having solved a difficult problem or having written a lot of text. Also, if you are concerned that some code you have written may not work properly, it is wise to save your previous work in case something bad happens.

It is good practice to save your work frequently!

Delete all output before saving your work, Cell ↦ Delete All Output

New notebooks are created with default names like ***untitled-1.nb***, ***untitled-2.nb***, etc. Of course, you would prefer a more descriptive name and can have it by using File ↦ Save As from the tool bar or the shortcut Shift-Ctrl-S and then filling in the dialog box that appears. Note that this operation only saves the notebook that is currently active. *Now would be a good time to save your working notebook.*

Command completion and templates

Command completion and templates can be obtained either from the **Input** menu or using shortcut keys Ctrl-K or Shift-Ctrl-K, respectively. For example, try typing `Be` followed by the command completion shortcut (Ctrl-K). You should then see a list of possibilities, from which you could select `BesselK` either with the mouse or arrow keys. Then typing the template shortcut (Shift-Ctrl-K) will produce a dummy argument list. Of course, you will then need to replace the dummy arguments to do a real calculation. Note, however, that the command templates do not specify optional arguments.

Although some people like command completion, others consider it to be a useless frill.

Help!

Help Browser

Probably the most important source of help is the **Documentation Center** provided with *Mathematica* itself. The **Documentation Center** can be accessed from the menu bar and its use should be self-explanatory. To search all resources, simply type the name of the function or keyword in the dialog box. Alternatively, you can browse the various categories shown in the **Documentation Center**.

▼ **Use the Documentation Center to discover the definition of Fibonacci polynomials. Obtain the 10th Fibonacci polynomial.**

▼ **Use the Documentation Center to find the syntax for Integrate. Then evaluate the integral of x ^ 3 / (Exp[x] + 1) from 0 to infinity.**
Note: The integral may take some time to compute.

Throughout the course materials there are several commands highlighted in blue. If you put your cursor in the command (any command in a text or input section works) and push F1 *Mathematica* opens a window containing the Help Browser entry for that command. The Help Browser will then provide the definition of the function, its options, and details of its syntax and properties. In many cases there will also be examples provided and links to related information throughout the *Mathematica Book*. Using this method it will not be necessary to provide such details within these notebooks themselves. For example, by placing your cursor within the word **Range**, one finds that `Range[nmin,nmax]` produces a list of integers between `nmin` and `nmax` (Try it!).

Many parts of the Documentation Center include statements which you can modify or execute without actually changing anything in your system. Don't worry — you won't break it! You can also copy cells from the browser into your own notebook.

▼ **Find the examples of Integrate in the reference guide. Execute some of the input statements by clicking within the cell and pressing Shift Enter simultaneously. The Enter key from the numeric keypad should also work.**

Information escape

Quick help on a built-in function or on a symbol that you have created can be obtained by typing `?symbol` or, for more information, `??symbol`.

```
? Fibonacci
```

```
?? Fibonacci
```

Assuming that you already know the definition for Fibonacci numbers or polynomials, this method provides the information needed to use the built-in function. However, if you do not know that definition, more complete information can be obtained using the Documentation Center. A suitable hyperlink is often included with the output.

Information about the available functions can be obtained by using wildcards. Thus, we find the following symbols with names beginning with Plot. Notice that clicking on any of these links provides a brief explanation of the function and a link ">>" to more detailed information.

```
? Plot*
```

```
? *Plot*
```

▼ **Obtain a list of functions that modify the Form of an expression. Hint: These commands end in Form. Read the brief descriptions for a few of these. Some will have obvious function but others may appear somewhat obscure.**

```
? *Form
```

Web resources

You can also obtain useful information from the ***Mathematica*** web site. Another valuable resource is the ***Mathematica Information Center*** , which provides an extensive collection of applications catalogued by topic. Clicking on a hyperlink to a web resource should activate a web browser if one is available on your system. Notice that if you point at a hyperlink without clicking, the URL is displayed in the lower left corner of the window containing the notebook.

Input conventions

Any written or programming language has a set of conventions governing the preparation of acceptable text and *Mathematica* is no exception.

Symbols

■ Built-in symbols

First note that *Mathematica* is case-sensitive, meaning that a symbol named *y* is distinct from a symbol named *Y*. All built-in symbols are assigned names that begin with capital letters. *Mathematica* generally names its symbols quite descriptively with names closely related to the function of each symbol. For example, the function designed to simplify algebraic expressions is called **`Simplify`**. Sometimes the names become very long, but most users find it easier to type a verbose name than to remember abbreviations which can be chosen almost arbitrarily. To avoid confusing your variables with names reserved by *Mathematica* it is good practice to choose names that begin with lowercase letters, but if you also choose long and descriptive variable names that is usually no longer necessary.

Information about a symbol can be obtained by typing **`?name`** where **`name`** is replaced by the desired symbol and may include the wildcard character **`*`** if you are uncertain of the complete name.

```
? Bessel*
```

```
? BesselJ
```

■ Rules for naming symbols

- □ names must begin with a letter, but may include numbers anywhere after the first character
- □ names cannot use _, +, -, /, :, ?, * or other operators
- □ names cannot include embedded spaces
- □ names cannot duplicate a reserved name. For example, a common conflict occurs with *N*, which is reserved for numerical evaluation; hence, the following assignment fails.

```
N = 2
```

```
? N
```

- □ names beginning with **`$`** indicate a default or system parameter. For example, **`$MachinePrecision`** indicates the default number of decimal digits used to represent floating-point numbers.

```
? $MachinePrecision
```

- □ names may include special characters, such as Greek letters, but do not differentiate between fonts.

■ Reserved names

A brief list of the reserved names most likely to generate conflicts (at least for physics applications) is given here. Other reserved names are longer function names which are rarely confused with user-generated symbols.

`N`	numerical evaluation, confused with number
`I`	imaginary unit $\sqrt{-1}$
`E`	exponential constant *e* (base of natural logarithms), confused with energy
`C`	constant of integration
`D`	partial derivative

■ Special characters

A nice feature of the notebook interface is its ability to display most special characters you might desire. For example, the built-in symbol `Pi` can be entered using the standard alphabet

```
Pi
```

it is returned as a Greek character upon output. If you prefer to enter π in expressions, the *escape sequence* ⎋pi⎋ will yield the desired result — just press the escape key, the letters pi, and then another escape key.

▼ Use escape keys to produce the following symbols: π, Π, ϕ, θ, $\int$.

Beware of characters which have the similar or identical appearances on screen but different internal representations. For example, below we test whether the Greek letter μ, entered as ⎋mu⎋, matches the symbol used to represent the prefix micro, entered as ⎋mi⎋.

```
MatchQ[μ, µ]
```

▼ Obtain information about MatchQ.

Naming things

It is easy to assign a value to a symbol or to assign a name to a complicated expression or result that we may need to use again — simply use the assignment operator in the form `name=expression`.

```
d = 3
```

Thus, the statement

```
d
```

simply returns the value assigned to `d`. Similarly, the assignment

```
quadratic = a x^2 + b x + c;
```

creates a symbol that represents a quadratic expression.

```
quadratic
```

The *Mathematica* function associated with this type of assignment is `Set`.

```
Set[Joe, "an obnoxious fool"];
```

```
Joe
```

Multiple assignments of several symbols to the same value can be performed by stringing together assignments.

```
a = b = c = d = π^ⅈ
```

```
N[b]
```

Name conflicts

Once a value is set it remains until reset or cleared. A common error is to forget that a value has been assigned to a symbol and later to reuse that symbol in another expression where it should be a variable. Therefore, it is good practice to clear symbols before using them or after you are finished with them.

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

Caution: symbols are associated with an entire session, not with a particular notebook. When several notebooks are open, symbols defined in one are defined in all. Conflicts can arise if you are not careful. It is good practice to place the following statement near the beginning of each notebook. If you then evaluate an entire notebook, you should get a fresh start free of conflicts arising from prior work on other notebooks.

```
ClearAll["Global`*"]
```

which clears the definitions, values, and attributes of all symbols created by the user (technically, from the `Global` context). **Remove** removes the names also, but usually is not necessary.

It is good practice to clear symbols before using them!

ClearAll["Global`*"] before each new problem on your homework.

I also like to include the statement

```
Off[General::spell, General::spell1]
```

to suppress some annoying warning messages that occur when two symbols are created with names sufficiently similar that *Mathematica* worries about possible typographical errors. On the other hand, you might appreciate such warnings if your typing or proofreading skills are not so strong; in that case, just omit this command.

Arithmetic notation

Basic Arithmetic Operators

operation	operator	example	function
negation	-	-a	Times[-1, a]
addition	+	a + b	Plus[a, b]
subtraction	-	a - b	Plus[a, -b]
multiplication	* or space	a * b or a b	Times[a, b]
division	/	a / b	Times[a, Power[b, -1]]
power	^	a^b	Power[a, b]
root		a^(1 / b)	Power[a, Power[b, -1]]
factorial	!	a!	Factorial[a]
contraction	.	a.b	Dot[a, b]

The four basic arithmetic operations of addition, subtraction, multiplication, and division are indicated by the usual operators +, −, *, and /. In addition, the power operator is ^ and the factorial operator is !. These operators are summarized in the accompanying table. The last column gives the function form for these operations. Note that independent functions for negation, subtraction, division, and roots are not used for internal representation of these functions because they are actually just special cases of addition, multiplication, and powers.

Mathematica often provides several equivalent methods for entering the same expression. For example, multiplication can be entered either using the * operator or using a space between two variable names. Thus, even though we use * to enter the following input, the output is given with a space because that is the form preferred by most aficionados.

```
a * b
```

Similarly, the quotient may be entered as a/b,

```
a / b
```

but is printed in standard form as $\frac{a}{b}$. The form a/b is described as one-dimensional input because all characters occupy the same line, while the form $\frac{a}{b}$ is described as two-dimensional.

▼ **Compute $87.3^{-0.3}$.**

▼ **Evaluate $(-1.0)^{1/4}$. Check your result. Is it unique?**
Hint: To check if it's unique write (-1.0)^(1 / 4) as $e^{i(\pi+2\pi*n)/4}$ for n = 0,1,2,3.

Delimiters

Delimiters are punctuation marks used to group terms in an expression or to separate distinct subexpressions. These delimiters are summarized in the accompanying table, along with simple examples.

Delimiters

delimiter		purpose	example
parentheses	( )	grouping	`(a + b) * c`
brackets	[ ]	functions	`Sqrt[a]`
braces	{ }	lists	`{a, b, c}`
double brackets	[[ ]]	part of expression	`list[[2]]`
comma	,	separate elements	`{a, b, c}`
semicolon	;	separate expressions	`a * b; b - c`
quotes	" "	text strings	"Have a nice day!"
comment	(* *)	documentation	(* this does ... *)

■ Parentheses are used to group terms

Standard rules of precedence require multiplication and division to be evaluated before addition and subtraction. Compare the following arithmetic expressions.

```
2 + 2 * 3 - 4 / 5 + 1
```

```
(2 + 2 * 3 - 4) / 5 + 1
```

In the first version the multiplication and division operations were evaluated first, followed by the addition and subtraction operations. In the second version parentheses were used to group some of the terms so that the grouped expression was evaluated before the division was performed. The standard rules of precedence apply within the parentheses. Also note that groupings may be nested to any order, with the evaluation proceeding from the innermost to the outermost grouping.

Although standard algebra permits square brackets or braces to be used for grouping, these delimiters are reserved by *Mathematica* for other purposes. Thus, the following expression (after the correct **Clear** statement) is invalid

```
Clear[a, b, c, d];
a / [b - c] + d
```

and should have been entered as

```
a / (b - c) + d
```

■ Square brackets are used to indicate functional dependencies

Square brackets are used to contain the arguments of a function. Although traditional notation uses parentheses, having the same delimiter serve two such different purposes would be prone to ambiguity, especially for a machine! Those of you who have tried to teach algebra have probably observed the confusion that can arise from an expression like $g(z+2)$ — is this equivalent to $(z+2)\,g$ or is it a function $g(x)$ evaluated for $x \to z+2$? Without additional information, the intended interpretation is unclear. It would be difficult to teach a machine subtle rules of context when intelligent humans often cannot resolve such ambiguities without additional information.

```
Sqrt[1.5]
```

▼ Compare Sqrt[1.5] with Sqrt(1.5). Explain the result with parentheses.

Note that in many of the exercises we will present mathematical expressions in traditional notation, but you will need to re-express those expressions in *Mathematica* notation — we do this to promote familiarity with the ambiguity-free notation employed by *Mathematica*. We also discourage you from using cut-and-paste operations to form input from exercise text — it will be good practice to type it yourself.

▼ Evaluate $\sin(\pi/4)$ both as written here and in the correct form for *Mathematica* input.

■ Comments

Comments or explanation can be included within input cells by using comment delimiters `(*` and `*)` to bracket your documentation. The bracketed text may span several lines. You can even place comments within executable expressions.

```
(* The following expressions use variables
        to illustrate the rules of arithmetic precedence. *)

Clear[a, b, c];
{
    (* division *)

    a / b / c, (a / b) / c, a / (b / c),

    (* powers *)

    a^b^c, a^(b^c), (a^b)^c, a^-b, -a^b, (-a)^b
}

(* Compare with previous numerical exercises. *)
```

For simple input it is usually better to place explanation in a separate text cell, but when an input cell contains a long, complicated program it is better to document each vital step using comment delimiters within the input cell itself; we will encounter these situations soon.

■ Several statements in the same input cell

An input cell may contain a single instruction or several instructions separated by semicolons — the semicolon is the delimiter used to separate complete expressions. The input cell below contains three instructions in a single cell.

```
a = 2;
b = 3;
a * b
```

Notice that the two expressions terminated by semicolons did not produce printed output, but were executed. The final instruction did produce output because it did not end with a semicolon; if it had, that would be equivalent to separating that expression from an additional expression which happens to be blank and hence gives no output. The following cell calculates an expression, but does not print output.

```
c = a * b * 12;
```

We can inspect that result later simply by entering the symbol **c** in an input cell that is not terminated by a semicolon.

```
c
```

Also note that it is not necessary for these instructions to occupy different lines or to be completed on a given line.

```
a = 2; b = 3; a b
```

```
c =
     2; d =
 -8;
c d
```

However, do not omit the semicolons (or use commas) because the results will then depend upon parsing and evaluation order and may be unpredictable. Also, be careful about using explicit carriage returns before a statement is complete because *Mathematica* generally evaluates an expression as soon as a complete expression is found while scanning the input cell from the beginning to the end. Thus, explicit carriage returns or multiple expressions without proper termination might produce complete expressions prematurely that generate unintended results. For example, the following input cell is ambiguous.

```
b = 2
a =
     3
          b
```

According to the input conventions, the following cell should be equivalent but gives different results.

```
b = 2;
a = 3 b
```

Trust *Mathematica*'s automatic line-wrapping and use the proper delimiters.

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

Some students develop the habit of including several statements in the same input cell, without using semicolons, so that several outputs are produced. However, I would like to discourage that habit because it can be error prone when those statements are long and line-wrapping becomes an issue. There is nothing wrong with entering several statements in the same input cell provided that those statements are separated by semicolons and only the last one, without a semicolon, is expected to produce output that we want to see. I do that frequently. However, if you want to see the output for more than one statement, it is better to use separate input cells for each.

Exact versus approximate numbers

Mathematica distinguishes between exact numbers, such as integers and rational numbers, on the one hand, and approximate real numbers on the other. When presented with exact input, *Mathematica* will always endeavor to return exact output. Thus,

```
1 / 2
```

is returned as an exact rational number. Similarly, an expression like

```
E^π
```

is returned unevaluated because the symbols **E** and **π** both represent exact, albeit irrational, numbers rather than decimal approximations. Conversely, an expression like

```
1. / 2.
```

is returned as an approximate real number because the numerator and/or denominator were entered in the form of approximate real numbers and *Mathematica* declines to interpret these as integers. The use of decimal points in the input distinguishes these cases. However, often we would prefer decimal output. The built-in function **N** stands for numerical evaluation and converts all exact numbers to approximate real numbers. `N[expr]` chooses the number of digits for itself, while `N[expr,m]` attempts to return a result with *m*-digit precision.

$$\frac{8!^2}{13^9}$$

$$N\left[\frac{8!^2}{13^9}\right]$$

```
N[π, 10]
```

Note that `N[expr,m]` does not necessarily print all *m* digits; *Mathematica* makes some aesthetic choices in printing its output. Use `ScientificForm` or `NumberForm` to control the output format.

$$\text{ScientificForm}\left[N\left[e^{2\pi}\right], 10\right]$$

$$\text{NumberForm}\left[N\left[e^{2\pi}\right], 10\right]$$

Sometimes an approximate result includes a vanishingly small residual. For example, if we evaluate $(-2.)^{1/4}$ and check the result by exponentiation, we see an imaginary term with vanishingly small coefficient.

$$a = (-2.)^{1/4}$$

```
a^4
```

This insignificant imaginary term arises because one can never be certain with finite-precision arithmetic that the imaginary part really does vanish exactly. (You know it, but the machine is not that smart!) Numerically insignificant terms can be removed using the function `Chop`.

```
Chop[a^4]
```

▼ **Compute the number of seconds in a year divided by π as a decimal number.**

▼ **Compare i^i and $e^{-\pi/2}$ numerically.**

▼ **The number $e^{\pi\sqrt{163}}$ is very close to an integer — determine the closest integer. Hint: Use Floor and Ceiling functions.**

Lists

■ Basic syntax

Braces or curly brackets are used to enclose lists and the elements of a list are separated by commas. For example, `Range[n]` produces a list of integers between 1 and *n*. Note that we prepend a lower-case letter to distinguish our symbol from the *Mathematica* `List` function.

```
aList = Range[10]
```

A matrix is simply a list of rows, where each row is a list with the same length.

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

▼ Produce a list of numbers between 0.15 and 1.25 with a spacing of 0.1.

▼ Evaluate {a,b,c}.{x,y,z} using both the operator form and the function form (Dot). Explain the result. Hint: Clear[a,b,c,x,y,z]

▼ Evaluate {{m11,m12,m13},{m21,m22,m23},{m31,m32,m33}}.{x,y,z} and interpret the result.

▼ Compute the eigenvalues of the matrix $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ using the built-in function Eigenvalues.

Mathematica provides a large number of built-in functions to manipulate the structure of lists. For example, `Reverse.`

```
Reverse[aList]
```

simply reverses the ordering of a list and `Transpose`

```
Transpose[aMatrix]
```

transposes a matrix.

A matrix can be displayed in the traditional grid form using the function `MatrixForm`.

```
MatrixForm[ Transpose[aMatrix] ]
```

Note that an object created using `MatrixForm` is not actually a matrix. `MatrixForm` is an example of a *formatting function* designed to print an object in a specialized output form. Thus,

```
test = MatrixForm[aMatrix]
```

cannot be manipulated using functions which expect matrix input. The expression

```
Transpose[test]
```

is returned unevaluated because the input is not of the correct type. To verify that an object is a matrix, one can use the query

```
MatrixQ[test]
```

```
MatrixQ[aMatrix]
```

`MatrixQ` is an example of a query function, ending in the letter Q used to verify that an object has the desired structure.

▼ Obtain a (partial) list of query functions ending in Q. Try a few.

▼ Obtain a (partial) list of formatting functions. Hint: These functions end in Form.

■ Elements of a list

Double square brackets are used to identify parts of an expression. Thus, `aList[[n]]` returns the n^{th} member of a list.

```
aList[[3]]
```

A particular element of a matrix is obtained by specifying its row and column within the *part* specification [[*row,column*]].

```
aMatrix[[2, 3]]
```

A more appealing version of the double bracket is obtained using the escape sequences ESC[[ESC or ESC]]ESC. *Type this yourself:*

```
aMatrix⟦3, 1⟧
```

▼ Print the second eigenvalue of aMatrix by itself using its part specification.

■ Iterators

Many functions require a list of uniformly spaced input values — such lists are created by iterators. An *iterator* is a list of the form `{variable,minimum,maximum,step}`. For example, `Table[f,iterator]` evaluates a table of expressions for each element of the iteration list. The expression may have a numerical or symbolic value.

```
Table[x, {x, 1, 11, 2}]
```

```
Table[BesselJ[n, x], {n, 0, 4}]
```

Several variants of the iterator notation are recognized, as summarized in the table below.

Iterator notation

`{imax}`	**iterate *imax* times without changing any values**
`{i, imax}`	**yields *i* between 1 and *imax* in unit steps**
`{i, imin, imax}`	**yields *i* between *imin* and *imax* in unit steps**
`{i, imin, imax, istep}`	**yields *i* between *imin* and *imax* in increments of *istep***

▼ **Produce a list of the first 10 LegendreP polynomials. Use the formatting function ColumnForm to improve the appearance of your output. Hint: The first LegendreP polynomial is LegendreP[0,x]**

■ Applying functions to lists

A very important function that facilitates computation with lists is `Apply[f,list]` which applies a function `f` to a list of arguments. The shortcut for this function is `f@@list`. For example, we can calculate the sum of even integers between 2 and 100 by applying the `Plus` function to a list of even integers generated by `Range` or `Table`.

```
Apply[Plus, Range[2, 100, 2]]
```

```
Plus @@ Table[n, {n, 2, 100, 2}]
```

▼ **Evaluate $\sum_{n=0}^{nmax}\left(-\frac{1}{2}\right)^n$ for *nmax=10* using Apply and a list produced by Table. Hint: Apply[...,Table[...,{n,0,10}]]**
Compare with the analytical result (obtained with Sum) for *nmax*=∞. Hint: Sum[...,{...,...,Infinity}]

▼ **Find a series approximation to Exp[5.] . Hint: $e^x=\sum_{n=0}^{\infty}\frac{x^n}{n!}$ and Apply[...,Table[...,{...,...,100}]]**

Closely related is the function `Map[f,list]`, with shortcut `f/@list`, which maps a function onto each element of a list.

```
Clear[f];
aList = Range[10]
```

```
Map[Sqrt, aList]
```

```
f /@ aList
```

These examples could have been formulated using `Table` because the elements of the list are equally spaced, but `Map` is more useful when there is no clear relationship between members of the list.

```
f /@ {4, π, q, Exp[x]}
```

▼ **Using Map and Apply compute the product of the square roots of the elements of the following list: {3,π,x,Sin[y]}. Try it with both shortcut and function notation.**
Hint: Apply[...,Map[Sqrt,...]]

Relational operators

It is useful to distinguish between two types of relational operators: arithmetic and logical, depending upon the nature of their arguments. Arithmetic relations compare two or more numerical expressions while logical relations compare two or more logical statements. In either case, the result is either `True` or `False` when it is possible to evaluate the relationship definitively or the expression is returned unevaluated when insufficient information is available for a definite evaluation. Note that `True` and `False` are reserved symbols. The following tables give a description of each relational operator, the corresponding *Mathematica* function, the keystroke sequence, and typeset examples. With the exception of `Not`, which is a unary function that takes only a single argument, these functions take a sequence of two or more arguments.

■ Arithmetic and algebraic relationships

Arithmetic Relational Operators

relationship	function	operator	example
equal	`Equal`	`==` or ESC `==` ESC	`p == q` or `p == q`
unequal	`Unequal`	`!=` or ESC `!=` ESC	`p != q` or `p ≠ q`
greater than	`Greater`	`>`	`p > q`
less than	`Less`	`<`	`p < q`
greater or equal	`GreaterEqual`	`>=` or ESC `>=` ESC	`p >= q` or `p ≥ q`
less or equal	`LessEqual`	`<=` or ESC `<=` ESC	`p >= q` or `p ≤ q`

Since each relationship in the following expression is true, the net result is true also.

$$2 * 2 == 2^2 == \sqrt{16} \geq 2$$

However, because the truth of the next expression cannot be ascertained without specifying the symbols, the expression is returned in the form of an unevaluated equation.

```
Clear[a, b];
a == b
```

Note that the operator that tests equality is a double-equal **==**, while **=** represents assignment of a symbol to an expression.

Symbolic identities can also be tested using `Equal`, but it is often necessary to employ `Simplify` or related symbolic manipulation functions to ensure more than a superficial comparison between two sides of the proposed equation. For example, the expression

```
Sin[x] Cos[x] == 1/2 Sin[2 x]
```

remains unevaluated whereas

```
Simplify[Sin[x] Cos[x] == 1/2 Sin[2 x]]
```

yields a definite answer. Without using the mathematical knowledge encoded in the simplification functions, it is not immediately obvious that the two sides of this expression agree.

▼ **Is $\frac{1}{1+\tan^2 x} = \cos^2 x$?**

▼ **Is $\tan\frac{x}{2} = \sqrt{\frac{1-\cos x}{1+\cos x}}$? Under what conditions can a definite evaluation be made. Hint: There is a sign ambiguity in the square root. Complications arise when handling multivalued functions.**

■ Equations (Equal or ==)

An equation is represented by a double equals sign, == or the typeset form ESC==ESC. Hence, the expression

```
eq1 = x^2 - 3 x - 4 == 0
```

produces a named equation. This simple algebraic equation can be solved using the built-in function **Solve**. The result is a list of replacement rules; we will learn more about rules and equation solving soon.

```
Solve[eq1, x]
```

An equation is simply a special case of a logical relationship between two expressions and uses the same relational operator as does symbolic logic.

Built-in functions

Standard mathematical functions

Practically all standard mathematical functions (arithmetic, trigonometric, exponential, logarithmic, hyperbolic, etc.) are provided. The names are capitalized and are either standard abbreviations, such as `Sin`, or are spelled completely, such as `Factorial`. A brief summary of these functions is given in the accompanying table. Representative argument sequences have been included for some functions, but we have not attempted to be exhaustive at this stage.

Standard Mathematical Functions

exponential	`Exp[x]`
natural logarithm	`Log[x]`
logarithm to base b	`Log[b, x]`
trigonometric	`Sin, Cos, Tan, Sec, Csc, Cot`
inverse trig	`ArcSin, ArcCos, ArcTan,`
	`ArcSec, ArcCsc, ArcCot`
hyperbolic trig	`Sinh, Cosh, Tanh, Sech, Csch, Coth`
inverse hyperbolic trig	`ArcSinh, ArcCosh, ArcTanh,`
	`ArcSech, ArcCsch, ArcCoth`

▼ **Compute $\log_{10} e$.**

▼ **Evaluate $\sin\left(\tan^{-1}(x)\right)$. [Remember to use *Mathematica* input conventions.]**

By default, trigonometric functions assume that the argument is expressed in radians. For those of us who think in degrees (who doesn't?), the built-in symbol `Degree` converts from degrees to radians and can be included as a factor in the argument.

```
Sin[30 Degree]
```

▼ **Use Degree to obtain the number of radians in 32.5°.**

Plotting

Mathematica provides a vast array of plotting and graphics tools, but for now it is sufficient to introduce only the most basic. A basic plot of a function of a single variable is produced by the command

```
Plot[f[x],{x,xmin,xmax}]
```

where `f[x]` is any built-in or user-defined function which evaluates to a real number given a numerical argument `x` between `xmin` and `xmax`.

```
Plot[ArcSin[x], {x, -1, 1}]
```

It is often necessary to wrap user-defined functions inside the `Evaluate` command to ensure that proper numerical results are obtained for plotting. For example, the *Mathematica* function `D[f[x],x]` evaluates the partial derivative of `f` with respect to **x**, where `f[x]` is any expression involving **x**. The **Evaluate** command is used in the following example to ensure that the derivative is evaluated before substitution of **x**. (See what happens if **Evaluate** is removed!)

```
Clear[f];
f[x] = D[ArcSin[x], x];
Plot[Evaluate[f[x]], {x, -1, 1}]
```

A list of functions is plotted by providing that list to **Plot.**

```
Plot[Evaluate[Table[Sin[2 n π x], {n, 1, 4}]], {x, 0, 1}]
```

Note that **Evaluate** is needed to construct the list before attempting to plot.

▼ **Plot the hyperbolic tangent for arguments between -5 and 5.**

▼ **Plot the positive branch of the inverse hyperbolic secant function over its entire domain.
Hint: ?ArcSech**

▼ **Plot the Airy function (AiryAi) for an illustrative range of real arguments.
Hint: ?AiryAi**

▼ **Plot the first four Bessel functions, BesselJ[n,x], for {x,0,10} together.
Hint: Plot[Table[...,{n,0,3}],{x,0,10}]**

Using functions

Listable functions

Many functions are listable; that is, they can be applied to each member of a list. As we will soon discover, this feature can often be used to perform calculations much more efficiently than by using traditional loop commands, such as `Do`.

For example, `Prime[n]` returns the n^{th} prime number and can be applied to a list to give a list of primes.

```
primeList = Prime[ Range[100] ]
```

A list of the first 10 square roots is obtained from the following command.

```
Sqrt[Range[10]]
```

Listable functions with two arguments, such as `Plus`, can receive either a scalar and a list or two lists of the same structure. Given a scalar and a list, a listable function of two arguments is evaluated for each element of the list paired with the scalar. Given two lists of the same dimensions, a listable function of two arguments is evaluated using corresponding members of the two lists. In both cases the output is a list of results.

```
1 + Sqrt[Range[10]]
```

```
Sqrt[Range[10]] Range[10]
```

▼ **Display the first 20 powers of 2.**
Hint: Range[1,20]

▼ **Evaluate the following expressions and explain the results. (Clear variables first!)**
a) x {a, b, c, d}   b) x^{a, b, c, d}
c) {a, b, c, d} {e, f, g, h}   d) {a, b, c, d} / {e, f, g, h}

▼ **Revelation 13.18: "Let him that hath understanding count the number of the beast, for it is the number of a man and his number is 666." Show that the sum of the squares of the first seven primes is the *beast number*.**
Hint: Apply[...,Prime[...]^2]

▼ **Create a list of reciprocals for the first 10 square roots. Then multiply this list with the square roots themselves.**
Hint: reciprocals=1/Map[...,...]

▼ **Evaluate the following expressions and explain the results.**
a) BesselJ[Range[0,4],x]
b) Plot[BesselJ[Range[0,4],x],{x,0,10}]

Creating functions

We will soon need to create our own functions that accept input, perform some calculations or manipulations, and return results. The efficient design of functions, or program modules, involves many considerations and it will take considerable time to become proficient at this task.

Labels that masquerade as functions

Suppose that we assign to some expression a name that looks like a function. For example, the assignment

```
Clear[f, x];
f[x] = 2 x^2 - x + 1
```

appears to create a simple quadratic function. However, although we can use this definition verbatim in other expressions

```
f[x]^2
```

we soon find that this definition is quite limited. When we give this function a numerical argument,

```
f[2]
```

it fails to return a numerical result; in fact, it remains unevaluated. The problem is that we supplied a definition for the expression `f[x]`,

```
?f
```

but did not supply a definition for `f[2]`. *Mathematica* evaluates expressions by matching patterns and does not automatically assume that `f[2]` matches the pattern `f[x]`. In this sense, `f[x]` behaves more like a symbolic name than like a function; it is a symbol that masquerades as a function.

```
MatchQ[f[2], f[x]]
```

In fact, we are free to give independent definitions for `f[2]` or `f[y]` which need not have any relationship to the definition for `f[x]`.

```
f[2] = "The sun will rise tomorrow."
```

```
f[y] = Exp[y]
```

The symbol `f` is now associated with a collection of assignments that may or may not be related.

```
?f
```

Patterns as dummy arguments

If we want *Mathematica* to recognize `f[2]` or `f[y]` as examples of a more general function `f[x]`, we must define our function using a *pattern argument*, as follows.

```
Clear[g];
g[x_] = 2 x^2 - x + 1
```

The additional underscore attached to the argument **x_** is used on the *lhs* to indicate that any expression can be used as the argument of this function and that the argument will be named **x** for the purpose of evaluating the *rhs*. In other words, **x** is a dummy argument. However, the underscore is not used on the *rhs* — once the argument is given, it is referred to as **x** in the subsequent definition of the function. With this version of the function, any argument is substituted for **x_**.

```
{g[2], g[y], g[z^2], g[p + q]}
```

In *Mathematica* jargon, the underscore refers to a *blank pattern* which matches any expression; attaching a pattern to a symbol in the form **x_** names the pattern **x** for later use.

```
MatchQ[2, x_]
```

```
MatchQ["Pink Elephants on Parade.", x_]
```

▼ Create the function $e^{-x} \sin x$ using a pattern argument.

▼ Write a function with two arbitrary arguments which returns the ratio between their difference and their sum. Be sure to test it!

Sometimes it is advantageous to use patterns for one or more arguments and explicit labels for others. In the following example, a pattern is used for a numerical argument, which can assume arbitrary values, while an explicit symbol is used to identify the system of units, of which there are only a small number of choices.

▼ **Write a function which converts between Fahrenheit and Celsius temperature. The input should identify its scale and the output should be the other scale — in other words, the same function should operate in both directions depending upon the input presented.**
Hint: functionName[temp_,scale1] and functionName[temp_,scale2]

Virtues of procrastination

Another very useful technique for creating functions is provided by the `SetDelayed` function or the `:=` operator. The right-hand side *rhs* of a delayed assignment of the form `lhs:=rhs` is not actually evaluated until the left-hand side *lhs* is requested. Consider the following functions.

```
Clear[f, g];
g[x_] = Expand[x^2 - 1];
f[x_] := Expand[x^2 - 1]
```

Both functions are defined using pattern or dummy arguments, but `f` uses delayed and `g` immediate assignment. Both functions give the same results for many arguments

```
{g[y], f[y]}
```

but different behavior is observed for more complicated arguments.

```
{g[x + y], f[x + y]}
```

The difference arises from the evaluation schemes for the two functions. Although we included `Expand` in the definition of `g`, it has no effect because *rhs* is evaluated immediately and is already expanded as much as its definition permits. Thus, `Expand` no longer appears in the definition stored for `g`.

```
? g
```

Conversely, when delayed evaluation is employed, the *rhs* remains unevaluated until an argument is presented. Thus, `Expand` remains within the definition of `f` and can act upon the argument.

```
? f
```

Although the difference between `Set` and `SetDelayed` may appear trivial for these simple functions, it is often crucial to the proper behavior of a function. The best choice often depends upon the intended behavior of the function, but I find that delayed assignment is usually better.

▼ **The built-in function $D[f, x] = \frac{\partial f}{\partial x}$ evaluates partial derivatives. Write and test a function which evaluates $\frac{\partial(1/f)}{\partial x}$ for any f and x. Compare delayed with immediate assignment and comment. Hint: One of these functions will always give 0.**

▼ **Create a function that transforms a pair of Cartesian coordinates $\{x, y\}$ to polar form $\{r, \theta\}$ using a quadrant-sensitive form of ArcTan that accepts two arguments. Test your function on the following points, {1,2},{-2,-3}, {0,1},{-3,0}?**
Hint: cartToPolar[{x_,y_}]:={radius,angle}=
Define another function which performs the inverse transformation and test it also.
Hint: polarToCart[{r_,θ_}]:={x,y}

Replacement rules

Basic syntax

One of the most powerful features of a symbolic manipulator is its ability to replace one expression with another using replacement rules defined by the user. A replacement rule is given in the form `OldExpression->NewExpression`; a prettier version of the arrow, →, is entered as ESC->ESC. Note that *Mathematica* will often substitute the pretty version even if you do not use ESC. The usual syntax for applying a replacement rule to an expression is to use the replacement operator `/.` after an expression as follows:

```
expr/.rule
```

Alternatively, we could used the function syntax

```
ReplaceAll[expr,rule]
```

For example, we can a define rule, named `rule1`, which replaces the symbol a by its square as follows.

```
rule1 = a → a^2
```

The effect of this rule is illustrated below.

```
5 a /. rule1
```

```
Exp[a] /. rule1
```

```
ReplaceAll[Exp[a], rule1]
```

Be aware, though, that *Mathematica* will apply your rules whether or not they make sense.

```
badRule = Sin[x] → 2
```

```
a + b Sin[x]^2 /. badRule
```

(Actually, there is nothing wrong with `badRule` if x is complex.)

▼ **Perform the change of variables $x \to y^{-1}$ on the following expression: x^2 Exp$\left[-x^{-2}\right]$.**

Rules using patterns

Let us reconsider `rule1`.

```
rule1
```

This is fine is we only need to replace a by a^2, but what if we want to replace b or c? This can be accomplished by expressing the replacement rule in terms of patterns rather than explicit expressions. The pattern **x_** stands for any expression.

```
rule2 = x_ → x^2
```

```
q /. rule2
```

```
Sin[q] /. rule2
```

```
a x^2 + b x + c /. rule2
```

▼ **Define a rule which expresses exponential functions in terms of hyperbolic sine and cosine functions (note: Exp[x]=Sinh[x] +Cosh[x]). Test your rule on the expression x Exp$\left[-y^2\right]$.**
Hint: {Exp[...]→Sinh[...]+Cosh[...]}

▼ **Define a rule which transforms a pair of Cartesian coordinates {x_,y_} into polar form {r,θ} using a**

quadrant-sensitive form of ArcTan that accepts two arguments. Test your rule on the following points, {1,2},{-2,-3}, {0,1},{-3,0}?
Hint: ruleFromCartToPolar={x_,y_}→{...,...};
Define another rule which performs the inverse transformation and test.
Hint: ruleFromPolarToCart={r_,θ_}→{...,...};

Lists of rules

We can collect a set of rules in the form of a list and assign a name to that set. For example, the following rules can be used to manipulate expressions involving logarithmic functions. The first replaces the log of a product of two factors by the sum of their logs. The second pulls an exponent outside. To completely simplify all possible logarithmic expressions, we would need to produce a more complete set of rules using other features of pattern matching, but this simple set will suffice to illustrate the principle.

```
myRules = {Log[a_ b_] → Log[a] + Log[b], Log[a_^n_] → n Log[a]};
```

```
Log[x y] /. myRules
```

```
Log[x^2] /. myRules
```

▼ **Perform the change of variables x→Sin[θ] on the following expression: $\frac{x\,dx}{1-x^2}$. Be sure to make the corresponding replacement of the differential in the numerator also, and simplify the result.**
Hint: Simplify[.../.{...,dx→Cos[θ]*dθ}]

Repeated replacement

Since our rules for simplifying logarithms seem to work on the most basic examples, let's try something a little more interesting.

```
Log[x y z] /. myRules
```

```
Log[x y z^2] /. myRules
```

```
Log[x^2 y z^2] /. myRules
```

```
Log[x / y] /. myRules
```

Unfortunately, none of these expressions was expanded completely. The problem is that `ReplaceAll` examines each subexpression, applies the first relevant rule to that subexpression, and then moves on to the next subexpression. However, we often encounter situations in which several rules are pertinent to a particular subexpression, or another rule pertains to the result of an earlier rule acting on a subexpression. Although one could apply the rules several times, it is often more efficient to employ `ReplaceRepeated`, with the `//.` operator notation.

```
Log[x y z] //. myRules
```

```
Log[x y^2 z^3] //. myRules
```

```
Log[x / y] //. myRules
```

`ReplaceRepeated` applies all pertinent rules to each subexpression until the result stops changing. In this manner we achieve more complete expansion of these more complicated expressions.

⚠ **However, one must be careful to ensure that a set of rules is not circular or infinitely recursive or else ReplaceRepeated might produce an endless loop requiring manual abortion.**

```
a //. x_ → x^2
```

```
$Aborted
```

Notice that we used the **Cell Properties** menu to specify that the input cell above cannot be evaluated. If it were, the evaluation would not terminate and we would have to abort the evaluation manually, resulting in the output value `$Aborted` indicated above.

▼ **Stirling's approximation suggests Log[n!]≈n Log[n]-n. Use this approximation, together with rules for logarithms, to evaluate Log$\left[\frac{n!}{m!\,(n-m)!}\right]$.**
Hints:
newRule1={Log[n_!]→Stirling's Approximation};
Log[...]//.myRules//.newRule1

Delayed rules

As with assignments, it is sometimes better to delay the application of a rule until after an explicit argument is provided. For example,

```
TrigFactor[Sin[a/b] + Sin[b/a]]
```

produces an expression whose arguments are not simplified. One might try to simplify those arguments using

```
TrigFactor[Sin[a/b] + Sin[b/a]] /. {f_[x_] → f[Together[x]]}
```

but the rule is applied too early. The solution is to use a delayed rule, indicated by `:>` or ESC:>ESC, so that `TrigFactor` is performed before the replacement is applied. Also, note that `Simplify` does not have the desired effect; it would undo the result of `TrigFactor`.

```
TrigFactor[Sin[a/b] + Sin[b/a]] /. {f_[x_] :→ f[Together[x]]}
```

▼ **The phenomenon of *beats* occurs when two waves of comparable amplitude and slightly different frequencies are added together. Let $\psi = \text{Cos}[\omega_1 t] + \text{Cos}[\omega_2 t]$. Use TrigFactor to transform the expression for ψ into the product of two terms and a delayed rule to simplify the arguments. Identify the beat (modulation) frequency.**
Hints:
TrigFactor[ψ]/.{Cos_[...]:→Cos[Simplify[...]]}
Beat frequency = |A-B| if 2*Cos[$\frac{t}{2}$(A-B)]Cos[$\frac{t}{2}$(A+B)]

Results given as rules

Many functions report their output in the form of replacement rules. For example, the function Solve returns the solution to an equation or system of equations in the form of replacement rules.

```
solutions = Solve[a x^2 + b x + c == 0, x]
```

Particular cases are then evaluated using replacement rules.

```
x /. solutions /. {a → 1, b → 2, c → -1}
```

Frequently we need to apply a replacement rule only to parts of a list that meet certain criteria. To accomplish this we use the `Condition` operator `/;`

```
? Condition
```

For example, to replace negative values of the list `{1,-2,3,-5,-4}` by their positive values, we do

```
{1, -2, 3, -5, -4} /. {x_ /; x < 0 → -x}
```

▼ **Take square root only of the positive values in {3, -3, 5, -5, 4} list.**
Hint: {3, -3, 5, -5, 4}/.{...→Sqrt[...]}

▼ **Compute Exp[x] for the negative solution to the equation $x^3 - 2x^2 + 1 == 0$.**
Hints:
?Cases
Cases[...,x_/;x<0→Exp[x]]

▼ **Evaluate and simplify the product of the roots of the quadratic equation $a x^2 + b x + c == 0$.**
Hints:
roots=...;
Simplify[Apply[...,roots]]

Options

The behavior of many functions can be modified by choosing options. The usual syntax is to call a function `f[args,opts]` where `args` represents the set of positional arguments and `opts` represents a subset of the available options. Options are specified in the form of replacement rules. The set of options that are available for a particular function or object and their default values can be listed using the `Options` function. If an optional argument specifies the particular option or list of options, the defaults for those will be returned.

```
Options[Plot]
```

```
Options[Plot, AspectRatio]
```

```
? GoldenRatio
```

```
Options[Plot, {Frame, FrameLabel, PlotLabel, GridLines}]
```

```
samplePlot =
    Plot[ BesselJ[2, x], {x, 0, 100},
  Frame → True, FrameLabel → {"x", "BesselJ"},
  PlotLabel → "Sample Plot", GridLines → Automatic]
```

▼ **Plot the gamma function for real arguments between, say, -5 and +5. Add axis labels and a title to your figure using appropriate options of Plot.**
Hint: AxesLabel

▼ **Use** Factor **to factor the expression $(x^2 + 1)$. Your first attempt will probably not achieve the desired result. Try using the** GaussianIntegers **option. Of course, you should find out which options are available and use Help to find out what this option is supposed to do!**

Using replacement rules to evaluate functions

Suppose that we define a function with an explicit argument.

```
Clear[f, x];
f[x] = 1 / Sin[x];
```

Although we cannot evaluate this function for $x = 0.2$ by simply requesting `f[0.2]`

```
f[0.2]
```

because there is no definition for `f[0.2]`,

```
?f
```

we can use a replacement rule to substitute the desired value for the argument.

```
f[x] /. x → 0.2
```

Similarly, a symbolic replacement gives a related function.

```
f[x] /. x → 1 / y
```

It is usually better to use a replacement rule instead of a definition (`Set`) to substitute values or to transform expressions because definitions fundamentally change symbolic expressions, whereas substitution by replacement merely evaluates a particular instance. For example, if we define **`g[y]`** to have essentially the same functional dependence as **`f[x]`** with **`x`** replaced by **`y`**

```
Clear[g, y];
g[y] = f[x] /. x → y
```

but then assign a value to **`y`** using `Set`

```
y = 2
```

the functional dependence of **`g[y]`** is lost until **`y`** is cleared. It is much cleaner to evaluate substitutions using replacement rules.

```
g[y]
```

```
Clear[x, y, z]
```

Replace versus ReplaceAll

Notice that the operator **`/.`** is equivalent to **`ReplaceAll`** rather than to **`Replace`**; what is the difference between these versions of the replacement function? **`Replace`** acts only on complete expressions, whereas **`ReplaceAll`** applies the rule to subexpressions as well. Thus,

```
ReplaceAll[Exp[a], a → I k x]
```

replaces the argument of the exponential function with the desired interpretation, whereas

```
Replace[Exp[a], a → I k x]
```

does not reach inside the exponential to apply the replacement rule to the subexpression represented by the symbol **`a`**. Nevertheless, we can use **`Replace`** if we give it rules which act on complete expressions.

```
Replace[Exp[a], Exp[a] → Exp[I k x]]
```

However, we tend to use this version of the replacement function rather infrequently.

Using packages

What are packages?

One of the most important features of *Mathematica* is that it is an extensible system. In addition to the built-in functions available to any *Mathematica* session, one can add new features by loading a package that contains the necessary definitions. A *package* is a special kind of *Mathematica* file that contains the definitions and specifies the rules for manipulating symbols and functions relevant to a particular purpose. When such a package is loaded, new features become available. There are several kinds of packages.

standard packages — are provided and supported by the vendor as standard parts of the *Mathematica* system. However, instead of using memory to store the definitions and code for all functions that are available, those that are needed less frequently than the built-in functions are collected as sets of packages that can be loaded as needed for particular applications.

extra packages — are written by third parties but distributed by the vendor. Although these packages are considered sufficiently useful for general distribution, they are not necessarily as well supported as the standard packages.

commercial packages — are developed and sold independently. These packages provide extensive tools for specialized applications in science, engineering, or finance.

user packages — many users find it useful to collect their best code in packages that can be used for many of their own applications. Some of these packages can evolve toward commercial status. We will discuss package writing near the end of the course.

Contexts

Just as the context determines the meaning of words in human languages, particularly when the word has several possible meanings, *Mathematica* symbols are defined within a *context* also. When the kernel is loaded we start with the **`System``** and **`Global``** contexts. In addition, there may be several other contexts available depending upon the contents of the start-up folder.

The list of contexts which are presently available is given by the following system variable.

```
$ContextPath
```

Symbols encountered by *Mathematica* are interpreted by searching the context path for information about that symbol. Thus, a new symbol that we introduce, such as **x**, will usually be associated with the **Global`** context, as indicated by the response **Global`x** to the inquiry **?x**.

```
x = 2
```

```
? x
```

The context in which a symbol was defined is returned by the function **Context**.

```
Context[x]
```

System variables established by the kernel are generally found in the **System`** context. For example, the context in which **$ContextPath** was defined is

```
Context[$ContextPath]
```

Loading a package usually adds one or more new contexts and introduces a set of new symbols defined within those contexts.

It should be clear now why we use place statements like **Clear["Global`*"]** or **ClearAll["Global`*"]** at the beginning of notebooks to clear the symbols created by the user within other notebooks. Hopefully the syntax also appears clear at this point.

Loading a package

The best way to load a package is to use the << operator, which reads the package if it is not yet on the context path. The argument is of the form **Package`**. Note that the back-quote(s) is essential. For example, the following statement loads the package which can transform units from one form to another.

```
Remove[Celsius, Fahrenheit]; (* explained later*)
<< Units`
```

When a package is loaded, it introduces new symbols that are defined in the *context* of that package. The list of contexts which are presently available is given by the following system variable.

```
$ContextPath
```

Loading the `Units` packages introduced several new contexts based upon the package names.

The symbols defined by a particular context can be listed using the information operator and a wild card after the context name; thus, after we load `PhysicalConstants`` package many physical constants will become available.

```
<< PhysicalConstants`
```

We can obtain more detailed information about a particular symbol also.

```
?? PlanckConstant
```

The `Units` package provides a function `Convert` which can perform a change of units. Note that the physical constants include units in the form of tags consisting of factors that are symbols for the appropriate units.

```
FullForm[PlanckConstantReduced]
```

```
? Joule
```

```
Convert[Joule, ElectronVolt]
```

```
Convert[PlanckConstantReduced, ElectronVolt Second]
```

```
Convert[PlanckConstantReduced SpeedOfLight, ElectronVolt Angstrom]
```

▼ **The gravitational acceleration g at the earth's surface is $G M / R^2$ where G is the gravitational constant and M and R are the mass and radius of the earth. Convert g from units of Meter / Second^2 to Feet / Second^2.**
Hint: ?*Gravity*

▼ **What do MKS and CGS do?**

▼ **Use ConvertTemperature to express 0 Fahrenheit in Celsius units. (Obtain syntax with ?). Note that this function avoids the difficulties with 0 that you may have encountered in earlier exercises.**

▼ **What is the relationship between parsecs and light-years?**
Hints:
?Parsec*
?*Year

```
? Curl
```

Avoiding shadows (mod DMW 2012)

```
ClearAll[x, y, z]
```

Suppose that you attempt to use a function before the relevant package has been loaded. Executing a statement which contains that function creates a symbol by that name in the `Global`` context, but does not perform the desired function. For example, Div (shorthand for the vector calculus function divergence) is found in VectorCalculus`, which has not been loaded yet.

```
vfunc := {2 x z + y^3, x^2 + z^2 - y, -3 x + y Sin[z]}
```

```
Div[vecfunc, Cartesian]
```

Remember-- when Mathematica returns the same thing you typed in, it means it does not recognize what you asked for.

```
? Div
```

If we now load the correct package FIRST, there is a conflict between the global symbol we created and the symbol defined by the package.

```
<< VectorAnalysis`
```

In *Mathematica* jargon, our symbol casts a *shadow* over the symbol of the same name defined within another context; hence, *Mathematica* does not know which version to use unless the context is specified. Although we can include the context when using the function,

```
SetCoordinates[Cartesian[x, y, z]]
```

Ah! Now Mathematica has recognized that we are setting the coordinate system to be cartesian, with variables x, y, and z.

```
VectorAnalysis`Div[vfunc]
```

we probably would rather use the short symbol name without the verbose context. The solution is to **`Remove`** the symbol from the global context so only **`Div`** from **`VectorAnalysis`** context remains.

```
Remove[Global`Div]
```

Having loaded the package above and defined the coordinate system, we can just use Div

```
? Div
```

```
Div[vfunc]
```

Exercises

▼ Gaussian genius

There is a story, perhaps apocryphal, that shows the mathematical genius of Gauss as a young school boy. Supposedly his teacher asked the class to sum the integers between 1 and 100. After a minute or so, while the other students toiled, Gauss appeared to be staring out the window daydreaming. When confronted, Gauss said that he had already obtained the answer, using the following method.

a) Produce a list of integers between 1 and 100.

b) Produce a second list in reverse order. Hint: Reverse[listFromPartA]

c) Add the two lists term by term and deduce a general formula for $\sum_{k=1}^{n} k$ by inspection.

Hint: Apply[...,{...,...}]

▼ Hexadecimal representation

Express the number of seconds in a year in hexadecimal form. [Hint: use **Help** to find functions which manipulate bases and note that hexadecimal form describes a number represented in base 16.]

▼ Count random primes

Produce a list of 100 random integers between 1 and 10^9 and determine the number that are prime (do not count by hand!). [Hint: consult **Help** and use `RandomInteger`, `PrimeQ`, and `Count`.] Try it several times and comment on results.

▼ A table of symbolic results

Produce a table containing factorizations of $1 - x^n$, for $1 \leq n \leq 10$ in convenient form.

Hint: Table[Factor[...,...→True],{...,1,10}]//...Form

▼ Golden Ratio

The Golden Ratio arises in the following problem from Euclid's *Geometry*. Suppose that a line segment *AC* is divided in segments *AB* and *BC* such that $\overline{AC}:\overline{AB} = \overline{AB}:\overline{BC}$. Determine this ratio. [Use *Mathematica,* don't do it by hand!]

Hint: http://en.wikipedia.org/wiki/Golden_ratio

▼ Beats

The phenomenon of *beats* occurs when two waves of comparable amplitude and slightly different frequencies are added together. Let $\psi = \text{Cos}[t] + \text{Cos}[(1+\delta)\, t]$, where $\delta \ll 1$, represent the superposition of two waves for which the time variable is expressed in units of inverse frequency.

Display the beat phenomenon by plotting ψ for $\delta \to 0.1$ over an appropriate interval. [Hint: you may wish to use the `PlotPoints` option if your plot appears somewhat ragged.]

▼ Compare product and series representations of exponential function

This problem compares product and series representations of the exponential function.

a) Evaluate limit of $\left(1 + \frac{x}{n}\right)^n$ as n goes to infinity using the *Mathematica* `Limit` function.

b) How large does *n* have to be to obtain convergence to better than 1% when $x = 2$?

Hint: Use `Plot` to find a guess value for `FindRoot`

c) Make a table of symbolic expressions for $\left(1+\frac{x}{n}\right)^n$ covering n between 1 and 10. Use **`Expand`** and **`TableForm`** to display the results in a useful fashion. Compare the product with $n = 10$ with the Taylor series for e^x around x=0.

Hint: For Talyor series use **`Series`** function.

▼ Table versus Do: The wondering Freshman

■ Generating data

```
f[x_] := x^3 - 4 x^2 + 2 x + 0.1;
data = Table[{i, f[i]}, {i, 0, 10, 0.005}];
Length[data]
```

■ Exercise

a) The list of data provided is the linear position of a lost freshman on their first day. Using the Table and ListPlot functions to plot their velocity for the same period of time (make sure that it plots from 0 to 10 seconds). Why do you need to start the table generating v at step 2?

Hint: Numerical programers often uses the definition of the velocity at a specific time step i as $v_i = \frac{x_i - x_{i-1}}{\Delta t}$ where Δt is the size of the timestep between intervals (0.005 seconds).

b) Many of you might know some programming in a procedural programming language. In most procedural languages you would use a do loop for this type problem. To do a comparison do the same problem using the Do command:

i) Look up the Do command in the Help Index. In procedural programming loops are used any time a command will be executed multiple times usually with small changes. Other examples of loops are For, While, DoWhile etc.

ii) Do loops do not generate output. Why not? So a list must be defined in advance to store any data generated. You will want to initialize a list to take the output since you cannot efficiently generate a list inside a Do loop in *Mathematica*. Make a list to hold your generated data (make sure it can hold the data from both the horizontal and vertical axes for the plot).

iii) Create a Do loop to generate the velocity of the freshman. Plot the result.

c) Now, repeat both the Table method and the do loop method, but this time use the Timing[] command to determine which takes longer. Remember to include the initialization of the list for the Do loop to be equivalent to Table command. While the difference is small this is very small set of data (How long?). It is not unusual to have data sets of matrices of 100,000 by 100,000 so that small difference adds up.