1. Basic Principles

1. Overview of Mathematica Features. Mathematica as a Calculator.

You can get a lot of information from the Help Browser (to access it press F1 or use the Help menu).

One can use Mathematica just like a calculator: one types in formulas and Mathematica returns back their values. Just press SHIFT + ENTER (RETURN) to tell Mathematica to evaluate the input you have given it.

Example.

```
2 + 2
```

(press SHIFT + ENTER after putting the cursor after 2 + 2 to see the output)

```
2 + 2
```

```
4
```

With a text - based interface, you interact with Mathematica just by typing successive lines of input, and getting back successive lines of output on your screen.

At each stage, Mathematica prints a prompt of the form In[n] := to tell you that it is ready to receive input. When you have entered your input, Mathematica processes it, and then displays the result with a label of the form Out[n] =.
Different text - based interfaces use slightly different schemes for letting Mathematica know when you have finished typing your input.With some interfaces you press Shift - Return, while in others Return alone is sufficient.

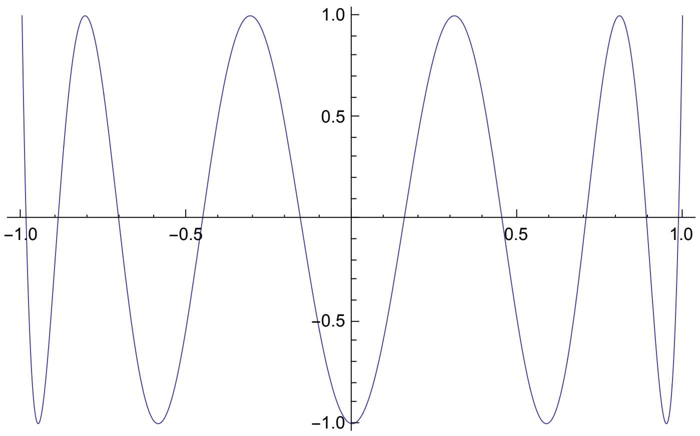
An important feature of Mathematica is its ability to handle formulas as well as numbers. Whenever you use Mathematica, you are accessing the world' s largest collection of computational algorithms. Mathematica knows about all the hundreds of special functions in pure and applied mathematics (e.g., Chebyshev polynomials, Bessel functions).

Example. The following function computes the 10th degree Chebyshev polynomial and the next one draws the function on the interval [-1, 1].

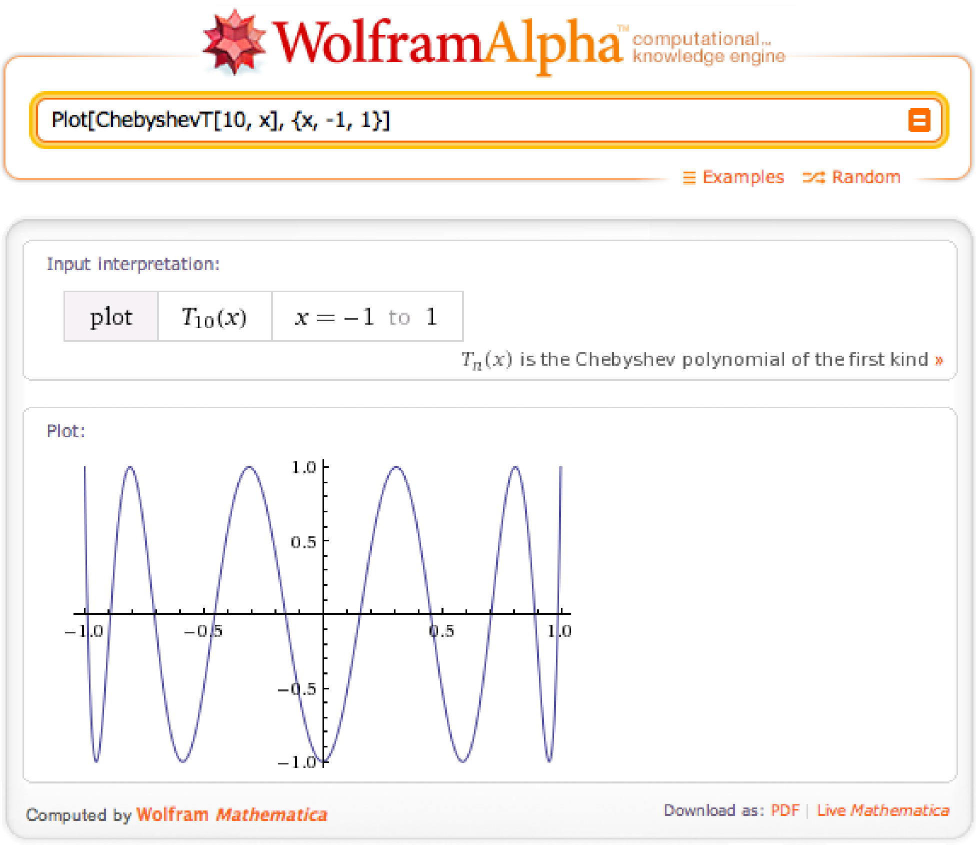
```
ChebyshevT[10, x]
```

$-1 + 50 x^2 - 400 x^4 + 1120 x^6 - 1280 x^8 + 512 x^{10}$

```
Plot[ChebyshevT[10, x], {x, -1, 1}]
```



Let's see what happens when we use the same input in WolframAl-
pha:



```
Plot[ChebyshevT[10, x], {x, -1, 1}]
```

Input interpretation:

plot	$T_{10}(x)$	$x = -1$ to 1

$T_n(x)$ is the Chebyshev polynomial of the first kind »

Plot:

Computed by Wolfram *Mathematica*

Download as: PDF | Live *Mathematica*

Note the two links at the lower right hand corner: Download as PDF and Live *Mathematica*. The first one is obvious. The second one needs the CDF Player plug-in to be installed.

In general, Mathematica notebooks allow importing and exporting of many formats. One can prepare even a slide show in *Mathematica*.

Kernel and FrontEnd

Mathematica consists of two independent computational environments called the FrontEnd and the Kernel, which communicate by means of a protocol called *MathLink*. The Kernel does all the computations. The FrontEnd is what you see in front of you, including the window, menu, etc. You can use many FrontEnds with one Kernel but the usual FrontEnd is what is know a notebook FrontEnd (there are also ASCII front ends you can run using a terminal interface).

The Kernel is the basic programing environment and in fact it can be used to completely control the FrontEnd. We will give a few examples, but we will not use much of this. For example:

```
nb1 = CreateDocument[{Plot[x^2, {x, -1, 1}]}]
```

```
NotebookObject[ Untitled-18 ]
```

```
ls = Notebooks[]
```

```
{NotebookObject[ Writing Assistant ],
 NotebookObject[ Foundations of Programming in Mathematica Part 1 ],
 NotebookObject[ Untitled-18 ], NotebookObject[ Untitled-13 ],
 NotebookObject[ Untitled-11 ], NotebookObject[ Untitled-15 ],
 NotebookObject[ NotebookClose – Wolfram Mathematica ],
 NotebookObject[ Installed Add-ons – Wolfram Mathematica ], NotebookObject[ Untitled-7 ],
 NotebookObject[ Untitled-6 ], NotebookObject[ Untitled-5 ],
 NotebookObject[ Untitled-4 ], NotebookObject[ Untitled-3 ],
 NotebookObject[ Untitled-2 ], NotebookObject[ Messages ]}
```

```
SelectedNotebook[]
```

```
NotebookObject[ Foundations of Programming in Mathematica Part 1 ]
```

```
SetSelectedNotebook[ls[[3]]]
```

```
NotebookObject[ Untitled-18 ]
```

The FrontEnd itself can also be “programmed” independently of the Kernel. This will be more important for us later, in building interfaces.

Mathematica notebooks

Mathematica is one of the largest single application programs ever developed, and it contains a vast array of algorithms and important technical innovations. Among these innovations is the concept of platform - independent interactive documents known as notebooks.

Every Mathematica notebook is a complete interactive document combining text, tables, graphics, calculations, and other elements. A Mathematica notebook consists of a list of cells, which you can group (sections etc).

Exercise. Click on different brackets on the right in this notebook with a right mouse button to find out the style being used.

Palettes and buttons provide a simple but fully customizable point - and - click interface to Mathematica (for Greek symbols, signs of integral, simple build - in functions, etc).
Recently Wolfram Research has expanded the concept of a notebook by introducing a new document format called CDF ("Computable Document Format") which unlike traditional notebooks allows interactive "dynamic" content.

The Unifying Idea of Mathematica

Mathematica is built on the powerful unifying idea that everything can be represented as a symbolic expression.

Main Features of Mathematica

Once one starts experimenting in Mathematica, one immediately notices some of its main features.
1. One important feature of Mathematica that differs from other computer languages, and from conventional mathematical notation, is that function arguments are enclosed in square brackets, not parentheses. Parentheses in Mathematica are reserved specifically for indicating the grouping of terms. There is obviously a need to distinguish giving arguments to a function from grouping terms together.
2. Names of built-in functions start with a capital letter.
3. Multiplication is represented either by * or by a space.
4. Powers are denoted by ^.
5. Numbers in scientific notation are entered, for example, as 2.5*^-4 or 2.5 10^-4.
6. There is a general convention in Mathematica that all function names are spelled out as full English words, unless there is a standard mathematical abbreviation for them. The great advantage of this scheme is that it is predictable. Once you know what a function does, you will usually be able to guess exactly what its name is. If the names were abbreviated, you would always have to remember which shortening of the standard English words was used.
7. Another feature of built - in Mathematica names is that they all start with capital letters. The capital letter convention makes it easy to distinguish built - in objects. If Mathematica used max instead of Max to represent the operation of finding a maximum, then you would never be able to use max as the name of one of your variables. In addition, when you read programs written in Mathematica, the capitalization of built - in names makes them easier to pick out.
8. N is a function that turns exact numbers and certain symbols into approximate numbers. For example:

```
N[Pi]
```

```
3.14159
```

```
N[Sqrt[2], 30]
```

```
1.41421356237309504880168872421
```

Hence N cannot be used for a function or a variable name. The same is true of some other symbols written with a capital letter (e.g. E,C). For that reason it is important to follow the convention that user defined symbols begin with a small letter.

A quick access to help information is achieved by typing the question mark :

```
? FullForm
```

FullForm[*expr*] prints as the full form of *expr*, with no special syntax. >>

```
? Part
```

expr[[*i*]] or Part[*expr*, *i*] gives the i^{th} part of *expr*.
expr[[−*i*]] counts from the end.
expr[[*i*, *j*, …]] or Part[*expr*, *i*, *j*, …] is equivalent to *expr*[[*i*]][[*j*]] ….
expr[[{i_1, i_2, …}]] gives a list of the parts i_1, i_2, … of *expr*.
expr[[*m* ;; *n*]] gives parts *m* through *n*.
expr[[*m* ;; *n* ;; *s*]] gives parts *m* through *n* in steps of *s*. >>

The quick access to help is also by highlighting the word and then pressing F1.

To get help for the command/operator you know you need to type ? and the command/operator. If you do not know the operator, search the Help Browser.

```
? >>
```

```
expr >> filename writes expr to a file. Put[expr1, expr2, ... ,
   "filename"] writes a sequence of expressions expri to a file. More...
```

Mathematica understands lists as {a, b, c} (in the full form it is List[a, b, c]). One can learn later on that many objects in *Mathematica* are written by using lists. For instance, a matrix can be inserted in the following way: go to the main menu; insert; tables/matrices:

$$\begin{matrix} \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \end{matrix}$$

Next one just needs to put the brackets and fill in the matrix elements by clicking on each empty square :

$$\begin{pmatrix} 1 & 2 & \square \\ \square & \square & \square \\ \square & \square & \square \end{pmatrix}$$

The result is :

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 5 & 6 & 7 \end{pmatrix}$$

The same matrix can be entered like this :

```
{{1, 2, 3}, {2, 3, 4}, {5, 6, 7}}

{{1, 2, 3}, {2, 3, 4}, {5, 6, 7}}
```

```
% // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 5 & 6 & 7 \end{pmatrix}$$

Here % means the last expression.

```
2 + 2

4
```

```
% + 4

8
```

```
%% + 2
```

6

Here %% means the last but one expression.

Example.

```
Plus[Power[x, 2], Sqrt[x]]
```

$\sqrt{x} + x^2$

The same can be entered by either using palettes or by the following sequence : Control key + 2 gives $\sqrt{\square}$; next one needs to type in "x"; this gives $\sqrt{x}$; next + x. To type in the square one can type in alt + 6 which gives $^{\square}$ and then type in 2 in the empty square.

One can type many symbols without using palettes. For instance, to type in π, one needs to press esc then type in pi then press esc once again.

Example.

The use of Ctrl+6:

x^2

esc+i+i+esc

ⅈ

esc+pi+esc

π

Alt+7 (applying to the blue bracket on the right): gives text in the notebook.

A very useful trick is the formula completion feature. Suppose, for example, you wish to use a function whose name begins with Plot but you can’t quite remember the rest of it. Just type in the beginning of the name and press the Control key (Command key on the Macintosh) and the letter K. You will see a pop up menu of all functions whose name begins with Plot. If you decide you want to use the function Plot3D you can type the name Plot3D and press Control and Shift keys together with the K key. You will see a template:

```
Plot3D[f, {x, xmin, xmax}, {y, ymin, ymax}]
```

An overview of programming techniques

For most of the more complex problems that one wants to solve with Mathematica, one has to create Mathematica programs oneself. Mathematica supports several styles of programming, and one is free to choose the one, one is most comfortable with. However, it turns out that no single type of programming suits all cases equally. As a result, it is useful to learn several different types of programming.

Traditional programming language such as C or Fortran use procedural programming (assignments and loops such as Do, For, While and so on). They also exist in *Mathematica*. But while any Mathematica program can, in principle, be written in a procedural way, this is rarely the best approach. In a symbolic system like Mathematica, functional and rule - based programming typically yields programs that are more efficient, and easier to understand.

Some types of programming :
Procedural Programming

List - based Programming (Many operations are automatically threaded over lists, a starting point to learn).
Functional Programming
Rule - Based Programming
Mixed Programming Paradigms

There are typically many different ways to formulate a given problem in Mathematica. In almost all cases, however, the most direct, precise and simple formulations will be best.

There are dozen of definitions of the factorial function (see later on).

Expressions

All objects in *Mathematica* programming language are expressions. For example

```
a + b
```

```
a + b
```

```
{2, 3, 5}
```

```
{2, 3, 5}
```

```
StringTake["hello", 4]
```

```
hell
```

```
Sin[x]
```

```
Sin[x]
```

```
Sin[π]
```

```
0
```

```
First[{a, b, c}]
```

```
a
```

are all different kinds of expressions. These expressions often look like mathematical formulas (more about that later on), which makes them more understandable and memorable to humans, but actually that have an internal form that is very simple and very consistent. It is called the “Full Form” of an expression and can be seen by applying the function FullForm to it (but there is a caveat, see below).

FullForm of expressions

Each expression is either an Atom or has the form

```
F[a1, a2,  ..., an]
```

where F is called the Head of the expression and a1, a2, are expressions. Examples of atoms are 2,a,3/4,3.2, "cat". Whether something is an atom can be tested with the function AtomQ:

```
AtomQ[2]
```

```
True
```

```
AtomQ[{2, 3}]
False
```

Expressions often do not look like their FullForms, for example a+b has FullForm:

```
FullForm[a + b]
Plus[a, b]
```

```
Head[a + b]
Plus
```

```
Head[a]
Symbol
```

```
Head[2]
Integer
```

```
FullForm[{a, b}]
List[a, b]
```

```
Head[{a, b}]
List
```

Note that atoms also have Head:

```
Head[2]
Integer
```

```
Head["cat"]
String
```

```
Head[cat]
Symbol
```

Note also that :

```
Head[x]
Symbol
```

```
x = 1
1
```

```
x + 2
3
```

```
Head[x]
Integer
```

Evaluation of x to 1 caused this to happen. You can see the original Head by preventing evaluating e.g.

```
Head[Unevaluated[x]]
```

```
Symbol
```

```
2 + 3
```

```
5
```

```
FullForm[Hold[2 + 3]]
```

```
Hold[Plus[2, 3]]
```

```
ReleaseHold[%]
```

```
5
```

The full form of

```
x = 1
```

is

```
FullForm[Hold[x = 1]]
```

```
Hold[Set[x, 1]]
```

```
Clear[x]
```

It is important to distinguish the assignment Set from Equal, which is usually written as == and has

```
FullForm[Hold[x == 1]]
```

```
Hold[Equal[x, 1]]
```

```
Clear[x]
```

```
x == 1
```

```
x == 1
```

```
x = 1;
```

```
x == 2
```

```
False
```

```
Clear[x]
```

Parts of Expressions

A very important skill is extracting parts of expressions. An expression is really a tree-like object, as can be seen using the function TreeForm:

```
? TreeForm
```

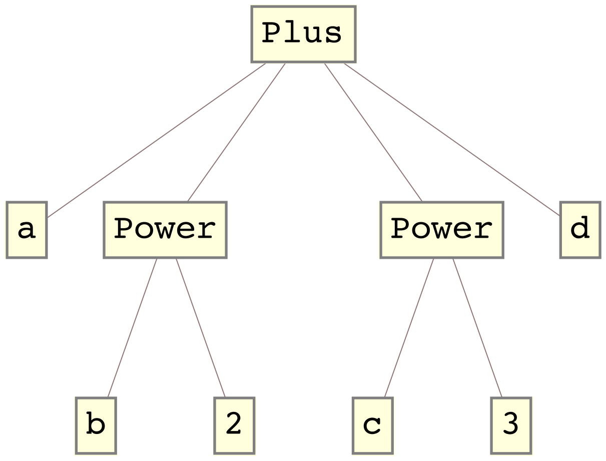
TreeForm[*expr*] displays *expr* as a tree with different levels at different depths.
TreeForm[*expr*, *n*] displays *expr* as a tree only down to level *n*. >>

$g = a + b^2 + c^3 + d;$

```
FullForm[g]
```

```
Plus[a, Power[b, 2], Power[c, 3], d]
```

```
TreeForm[g]
```



```
Level[g, {1}]
```

$\{a, b^2, c^3, d\}$

```
Level[g, {2}]

{b, 2, c, 3}

Part[g, 0]

Plus

g[[1]]

a

g[[2]]
```

b^2

```
g[[3]]
```

c^3

```
g[[3, 3]]
```

Part::partw : Part 3 of c^3 does not exist. ≫

$(a + b^2 + c^3 + d)[[3, 3]]$

```
g[[2, 2]]

2

g[[2, 0]]

Power
```

and so on.
You can also do this from the back :

```
g[[-2, 1]]

c
```

```
g
```

$a + b^2 + c^3 + d$

```
g[[1 ;; 3]]
```

$a + b^2 + c^3$

```
a + b^2 + c^3
g[[2 ;; 4]]
```

$b^2 + c^3 + d$

Now, here comes a very nice and important fact: you can change an expression by an assignment to a part of it. For example;

```
g[[1]] = x + y;
g
```

$b^2 + c^3 + d + x + y$

```
g[[3 ;; 4]] = z; g
```

$1 + b^2 + y + 2 z$

List, Vectors, Matrices, Tensors

A very important thing to notice that in *Mathematica* lists are just expressions with Head List:

```
m = {a, b, c, d};
Length[m]
```

```
4
```

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

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

A matrix is simply a list of lists of the same length:

```
mat = {{a, b, d}, {c, d, e}}
```

$$\begin{pmatrix} a & b & d \\ c & d & e \end{pmatrix}$$

```
mat[[1, 1]]
```

```
a
```

```
mat[[2, 2]]
```

```
d
```

```
mat[[All, 1]]
```

```
{a, c}
```

```
mat[[All, 2]]
```

```
{b, d}
```

```
mat[[1, All]]
```

```
{a, b}
```

```
mat[[2, All]]
```

```
{c, d}
```

We will later see how to easily create arbitrarily large matrices using the functions Table and Array.

Table$\left[i^2, \{i, 1, 10\}\right]$

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

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

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \\ 5 & 10 & 15 & 20 & 25 \end{pmatrix}$$

A note on forms of expressions.

We already know that a Mathematica expression often looks different to human eyes than its internal form (FullForm). However, the situation is made more complicated, by the fact that traditional mathematical notation is not unambiguous. Because of this and for reasons of history *Mathematica* has several “forms” of input and output. The first versions of *Mathematica* has only two forms: InputForm, which looked like a standard programming language (e.g. Fortran) way of writing mathematical formulas and OutputForm, which is a little more like usual mathematics and has become completely obsolete (it retained only for reasons of compatibility with very early *Mathematica* notebooks). Since then they have both been replaced by StandardForm and TraditionalForm. StandardForm retains the basic principles of InputForm but allows more usual mathematical expressions. TraditionalForm looks almost like the usual mathematical notation. One can convert between these forms using the Convert To sub menu in the Cell menu. One can also set the default forms for the Input and Output in the Preferences menu.

Basic principles of InputForm (and StandardForm)

1. All built in functions start with a capital letter.

2. Square brackets [] are used as function brackets.

3. (InputForm) The basic arithmetical operations are denoted by + (addition),* or space (multiplication) / (division), ^ (power).

4. There are the following inclusions InputForm ⊂ StandardForm ⊂ TraditionalForm but not in the opposite direction.

Links

http://reference.wolfram.com/Mathematica/guide/Expressions.html

```
http://reference.wolfram.com/mathematica/tutorial/FormsOfInputAndOutput.
  html
```

2. Working with Lists

One of the most common expressions in Mathematica are lists.

```
Solve[x^3 == 1, x]
```

$\{\{x \to 1\}, \{x \to -(-1)^{1/3}\}, \{x \to (-1)^{2/3}\}\}$

```
% // N
```

```
{{x → 1.}, {x → -0.5 - 0.866025 i}, {x → -0.5 + 0.866025 i}}
```

```
CoefficientList[a x^2 + b x + c, x]
```

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

```
Options[Plot]
```

```
{AlignmentPoint → Center, AspectRatio → 1/GoldenRatio, Axes → True,
 AxesLabel → None, AxesOrigin → Automatic, AxesStyle → {}, Background → None,
 BaselinePosition → Automatic, BaseStyle → {}, ClippingStyle → None,
 ColorFunction → Automatic, ColorFunctionScaling → True, ColorOutput → Automatic,
 ContentSelectable → Automatic, CoordinatesToolOptions → Automatic,
 DisplayFunction :→ $DisplayFunction, Epilog → {}, Evaluated → Automatic,
 EvaluationMonitor → None, Exclusions → Automatic, ExclusionsStyle → None,
 Filling → None, FillingStyle → Automatic, FormatType :→ TraditionalForm,
 Frame → False, FrameLabel → None, FrameStyle → {}, FrameTicks → Automatic,
 FrameTicksStyle → {}, GridLines → None, GridLinesStyle → {},
 ImageMargins → 0., ImagePadding → All, ImageSize → Automatic,
 ImageSizeRaw → Automatic, LabelStyle → {}, MaxRecursion → Automatic,
 Mesh → None, MeshFunctions → {#1 &}, MeshShading → None, MeshStyle → Automatic,
 Method → Automatic, PerformanceGoal :→ $PerformanceGoal,
 PlotLabel → None, PlotPoints → Automatic, PlotRange → {Full, Automatic},
 PlotRangeClipping → True, PlotRangePadding → Automatic, PlotRegion → Automatic,
 PlotStyle → Automatic, PreserveImageOptions → Automatic, Prolog → {},
 RegionFunction → (True &), RotateLabel → True, Ticks → Automatic,
 TicksStyle → {}, WorkingPrecision → MachinePrecision}
```

Let us also recall that the matrix is entered by using lists :

```
{{1, 2, 3}, {2, 3, 4}, {34, 5, 3}} // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 34 & 5 & 3 \end{pmatrix}$$

Let us learn how to generate lists and what basic operations one can perform with them. Another useful command is Table

```
Table[i^2 + 2, {i, -1, 2}]
```

```
{3, 2, 3, 6}
```

```
% // TableForm
```

```
3
2
3
6
```

One can generate not only numbers but also other expressions :

```
Array[a, 3]
```

```
{a[1], a[2], a[3]}
```

Some commonly used objects are already defined in Mathematica. For instance, the identity matrix :

```
IdentityMatrix[3] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

For the matrices *Mathematica* has a lot of build - in operations

```
Eigenvalues[{{1, 2}, {3, 2}}]
```

```
{4, -1}
```

```
Eigenvectors[{{1, 2}, {3, 2}}]
```

```
{{2, 3}, {-1, 1}}
```

Basic operations for the lists include the following :

```
{1, 2, 3} + {1, 2, 3}
```

```
{2, 4, 6}
```

```
{1, 2, 3} + 1
```

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

A scalar product is given by a dot

```
{a, b, c}.{s, d, f}
```

```
b d + c f + a s
```

However, one needs to be careful with length of the objects.

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

```
Thread::tdlen : Objects of unequal length in {1, 2, 3} + {1, 2, 3, 4} cannot be combined. >>
```

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

Other useful operations include

```
Prepend[{a, b, c}, d]
```

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

```
Append[{a, b, c}, d]
```

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

```
Union[{a, b, c}, {a, b, d}]
```

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

```
Join[{a, b, c}, {a, b, d}]
{a, b, c, a, b, d}
```

```
Take[{a, b, c, e}, 2]
{a, b}
```

Also have a look at commands Insert, Delete and many others in the help. The name of the command suggests unambiguously what it performs with a given list.

To get an element of the list one indicates its position.

```
{a, b, c}[[1]]
a
```

```
{a, b, c, d}[[-1]]
d
```

Here - 1 means the first element counted from the end.

```
{a, b, c, d}[[2]]
b
```

If you do not know how many elements are in the list, you can always verify this by using Length

```
Length[{a, b, v}]
3
```

```
Length[{a, b, {v, w}}]
3
```

A similar command for the dimension of the list is Dimensions

```
Dimensions[{a, b, {v, w}}]
{3}
```

```
Dimensions[{{a, b}, {v, w}}]
{2, 2}
```

This counts the elements of the first level in the list.

In applications one often encounters the problem to verify whether a given element is in the list and if so, one might require further its position.

```
Position[{{a, b, c}, {a, f, g}}, a]
{{1, 1}, {2, 1}}
```

Here Position takes account of the nesting of lists.

Since the lists can be nested, it is useful to know that they can always be flattened.

```
? Flatten
```

Flatten[*list*] flattens out nested lists.
Flatten[*list*, *n*] flattens to level *n*.
Flatten[*list*, *n*, *h*] flattens subexpressions with head *h*.
Flatten[*list*, {{s_{11}, s_{12}, …}, {s_{21}, s_{22}, …}, …}]
    flattens *list* by combining all levels s_{ij} to make each level *i* in the result. ≫

```
{{a, b, c}, {a, f, g}} // Flatten
```

```
{a, b, c, a, f, g}
```

To get rid of repeated elements one uses Union

```
% // Union
```

```
{a, b, c, f, g}
```

From a given list one can get a list of permutations and other lists of a given length with all elements of the original list

```
Permutations[{a, b, c}]
```

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

```
Tuples[{0, 1}, 3]
```

```
{{0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1}, {1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1, 1}}
```

```
Accumulate[{a, b, c, d, e, f}]
```

```
{a, a + b, a + b + c, a + b + c + d, a + b + c + d + e, a + b + c + d + e + f}
```

Apply and Map

```
? Apply
```

Apply[*f*, *expr*] or *f* @@ *expr* replaces the head of *expr* by *f*.
Apply[*f*, *expr*, *levelspec*] replaces heads in parts of *expr* specified by *levelspec*. >>

Let us form a new expression from the list and the other way round.

```
FullForm[{a, b, c}]
```

```
List[a, b, c]
```

```
Times @@ {a, b, c}
```

```
a b c
```

```
FullForm[a b c]
```

```
Times[a, b, c]
```

```
List @@ (a b c)
```

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

Another example is

```
Plus @@ {a, b, c}
```

```
a + b + c
```

A more complicated example is to generate a list of coefficients (maybe useful for polynomial expressions)

```
Subscript[A, #] & /@ Table[i, {i, 1, 10}]
```

$\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}\}$

Here /@ means Map.

```
? Map
```

Map[*f*, *expr*] or *f* /@ *expr* applies *f* to each element on the first level in *expr*.
Map[*f*, *expr*, *levelspec*] applies *f* to parts of *expr* specified by *levelspec*. ≫

Here there is a trivial example of forming a list.

```
Function[x, x^2] /@ Table[i, {i, 1, 10}]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

```
k = Table[i, {i, 1, 10}]
```

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

Therefore, the command /@ works as follows. It applies the function $x \longmapsto x^2$ to every element of the list k. (Here we meet an example of a pure function, the concept which will be discused below.)

Evaluation

A very important concept in *Mathematica* is that of evaluation. In *Mathematica* evaluation always takes place after you write some input and press Shift + Enter. The process of evaluation is quite complicated, and follows a definite sequence of steps. Understanding this process is important in advanced *Mathematica* programming and we will return to this in the future. Often the evaluation process takes place even if nothing seems to happen. For example:

```
FullForm[Hold[2/3]]
```

```
Hold[Times[2, Power[3, -1]]]
```

$$\mathbf{\frac{2}{3}}$$

$$\frac{2}{3}$$

```
FullForm[2 / 3]
```

```
Rational[2, 3]
```

```
2 + 3 I
```

$2 + 3\,i$

```
FullForm[Hold[2 + 3 I]] // InputForm
```

```
FullForm[Hold[2 + 3*I]]
```

```
FullForm[2 + 3 i]
```

```
Complex[2, 3]
```

```
AtomQ[Complex[2, 3]]
```

```
True
```

```
Head[Unevaluated[2 + 3 I]]
```

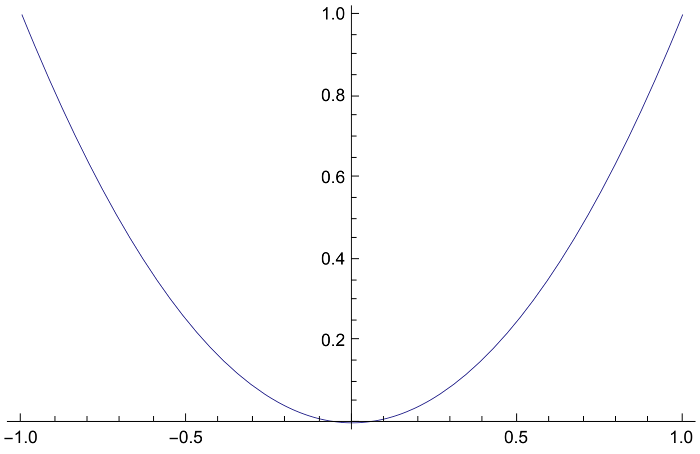
```
Plus
```

```
Head[2 + 3 I]
```

Complex

An interesting special case are graphics.

gr = Plot[x^2, {x, −1, 1}]



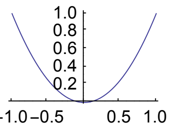
Short[InputForm[gr], 5]

```
Graphics[{{{}, {},
   {Hue[0.67, 0.6, 0.6], Line[{{-0.9999999591836735, 0.9999999183673486},
      << 272 >>, {0.9999999591836735, << 1 >>}}]}}}, {<< 6 >>}]
```

We see that a plot of a function is also a *Mathematica* Graphics object. One can therefore use the *Mathematica* programing language to control every detail of a graphic. Graphic programming in Mathematica is a whole big subject, but we will see a few examples later on.

It is possible to think of *Mathematica* as an algebraic object, somewhat like a ring, with partial addition and multiplication. This means that you can perform algebraic operations which are purely formal, for example, you can raise a number to the power of a graphic:

2^{gr}

2

In some situations arithmetical operations on objecs of different kind are defined, for example, it is possible to add a number (or a symbol) to a list:

{1, 2, 3} + 1

{2, 3, 4}

However, in certain cases, trying to perform such an operation on objects of different kind will cause a error message:

{1, 2, 3} + {1, 3}

```
Thread::tdlen : Objects of unequal length in {1, 2, 3} + {1, 3} cannot be combined. >>
```

{1, 3} + {1, 2, 3}

Here is an example of abstract algebraic manipulation performed on strings:

```
Distribute[("cat" + "dog") * "mouse"]
```

```
cat mouse + dog mouse
```

The evaluation loop.

When you enter an input expression *Mathematica*'s Kernel evaluates in a very definite order. Understanding this order is important for *Mathematica* programming. The evaluation order will be described carefully later once we learn about rules and patterns. However, the basic idea is this: *Mathematica* evaluates each part of the expression by turn, starting with the Head. It applies all rules it knows for the expression, first user defined then built in ones, until it can no longer find a rule. Then it stops and “returns” the result. (Sometimes this evaluation will not stop and we get into an infinite loop. Actually *Mathematica* will almost always detect such situations and will stop, unless we change the defaults to make it run for ever).

Programming using Patterns and Rules

Mathematica allows many different styles of programming. There is one style that, although not unique to *Mathematica*, distinguishes it from most other similar systems. This is the possibility of using "patterns" and "re-write rules" or just "rules".
The basic concepts in this kind of programming are "rule" and "pattern". Rules can be local and global.

Local Rules

A local rule always has the form

```
?Rule
```

lhs -> *rhs* or *lhs* → *rhs* represents a rule that transforms *lhs* to *rhs*. ≫

or

```
?RuleDelayed
```

lhs :> *rhs* or *lhs* :→ rhs represents a rule that transforms *lhs* to *rhs*, evaluating *rhs* only after the rule is used. ≫

Note that:

```
FullForm[lhs → rhs]
```

```
Rule[lhs, rhs]
```

```
FullForm[lhs :> rhs]
```

```
RuleDelayed[lhs, rhs]
```

The difference between Rule and RuleDelayed will be explained below. Most often Rule is used together with the function ReplaceAll (see also Replace):

?ReplaceAll

```
expr /. rules applies a rule or list of rules in an
   attempt to transform each sub part of an expression expr. More...
```

Here are some examples of using rules (in some of these examples the output appears in TraditionalForm).

```
x^2 + Sin[x y] + 3 /.{x → π, y → 2 π}
```

$3+\pi^2+\sin(2\pi^2)$

Now we use a more general rule. This time we use a "pattern"

```
? _
```

_ or Blank[] is a pattern object that can stand for any *Mathematica* expression.
_*h* or Blank[*h*] can stand for any expression with head *h*. ≫

```
x^2 + Sin[x y] + 3 /. _ → π
```

π

The reason for the above result is that ReplaceAll starts looking for a match starting at the top level of the expression and when it finds a match it stops looking for more. If we want to find a match at a different level we can use the function Replace with a level specification. For example, here we replace everything on level 3 of the expression with π.

```
Replace[x^2 + Sin[x y] + 3, _ → π, {3}]
```

$x^2+3+\sin(\pi^2)$

```
Level[x^2 + Sin[x y] + 3, {3}]
```

```
{x, y}
```

```
Replace[x^2 + Sin[x y] + 3, _ → π, {2}]
```

$3+\pi^\pi$

```
x^2 + Sin[x y] + 3 /. Sin[x_] :→ Sin[x^2]
```

$\sin(x^2 y^2)+x^2+3$

```
x^2 + Sin[x y] + 3 /. _?(# > 2 &) → π
```

$x^2+\sin(x\,y)+\pi$

```
x^2 + Sin[x y] + 3 /. a_ /; a > 2 → Pi
```

$x^2+\sin(x\,y)+\pi$

```
x^2 + Sin[x y] + 3 /. _?(AtomQ[#] &) → π
```

$\pi(\pi,\pi(\pi,\pi),\pi(\pi(\pi,\pi)))$

```
x^2 + Sin[x y] + 3 /. x_Times → π/4
```

$x^2+\frac{1}{\sqrt{2}}+3$

FullForm$\left[x^2 + \text{Sin}[x\, y] + 3\right]$

Plus[3, Power[x, 2], Sin[Times[x, y]]]

$x^2 + \text{Sin}[x\, y] + 3$ **/. _Power →** $\pi/4$

$\sin(x\, y) + \frac{\pi}{4} + 3$

These examples illustrate some of the very many ways of forming patterns in *Mathematica*. The most basic pattern is `x_` which stands for anything that is (locally) assigned the name x.

Here is an example where Rule and RuleDelayed give different answers:

***a* = 0;**

Sin[2] /. a_Integer → a^2

0

Sin[2] /. a_Integer :> a^2

sin(4)

Before using a rule it is a good idea to look at the FullForm of an expression. Here are some possible “traps”:

```
Clear[a]
```

$\sqrt{2} + \frac{1}{\sqrt{2}}$ **/.** $\sqrt{2} \to a$

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

FullForm$\left[\sqrt{2} + \frac{1}{\sqrt{2}}\right]$

Plus[Power[2, Rational[−1, 2]], Power[2, Rational[1, 2]]]

Unevaluated$\left[\sqrt{2} + \frac{1}{\sqrt{2}}\right]$ **/.HoldPattern**$\left[\sqrt{2}\right] \to a$

$a + \frac{1}{a}$

$\sqrt{2} + \frac{1}{\sqrt{2}}$ **/. 2^Rational[x_, y_] →** a^x

$a + \frac{1}{a}$

$\sqrt{2} + \frac{1}{\sqrt{2}}$ **/.** $\left\{\sqrt{2} \to a,\ \frac{1}{\sqrt{2}} \to 1/a\right\}$

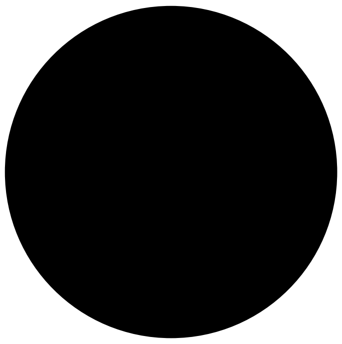
$a + \frac{1}{a}$

Rule based programming is very convenient when dealing with graphics.

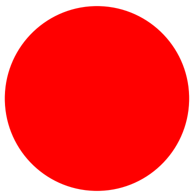
```
gr = Disk[{0, 0}, 1]
```

```
Disk[{0, 0}, 1]
```

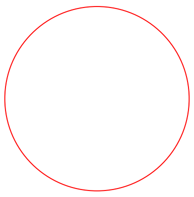
```
Graphics[gr, ImageSize → Small]
```



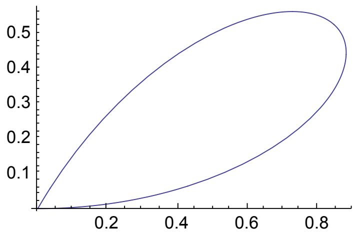
```
Graphics[{Red, gr}, ImageSize → Tiny]
```



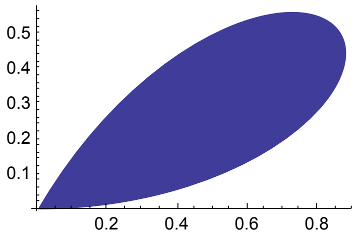
```
Graphics[{Red, gr}, ImageSize → {100, 100}] /. Disk → Circle
```



gr2 = PolarPlot$\left[\text{Sin}[3\,\theta], \left\{\theta, 0, \frac{\pi}{3}\right\}, \text{ImageSize} \to \text{Small}\right]$



```
gr2 /. Line → Polygon
```



Many *Mathematica* functions return a list of rules as the output.

rules = Solve[$x^3 + 3\,x + 4 == 0$, x]

$$\left\{\{x \to -1\}, \left\{x \to \frac{1}{2}\left(1 - i\sqrt{15}\right)\right\}, \left\{x \to \frac{1}{2}\left(1 + i\sqrt{15}\right)\right\}\right\}$$

Note that this is actually a list of lists, each containing one rule.

This is very convenient, because we can use ReplaceAll to substitute these rules into other formulas. For example:

x /. rules

$$\left\{-1, \frac{1}{2}\left(1 - i\sqrt{15}\right), \frac{1}{2}\left(1 + i\sqrt{15}\right)\right\}$$

$x^3 + 3\,x + 4$ /. rules // Simplify

```
{0, 0, 0}
```

Simplify[%]

```
{0, 0, 0}
```

Here is a similar example with FindRoot instead of Solve

FindRoot[$x\,e^{-x}$ == 0.2, {x, 0.1}]

$\{x \to 0.259171\}$

x Exp[$-x$] == 0.2 /. %

```
True
```

Links

http : // reference.wolfram.com/mathematica/tutorial/PatternsAndTransformationRules.html

Global Rules ("Functions")

Here is one way to define a "function" in *Mathematica*:

Clear[f]

f[x_] := x^2

```
f[I]
```

```
- 1
```

Clear[a]

f[a]

a^2

```
f[3]
```

```
9
```

?f

```
Global`f
```

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

```
DownValues[f]
```

```
{HoldPattern[f(x_)] :→ x^2}
```

Although people often call *f* defined in this way a function, actually it is only a "global rule". More precisely, when a definition of this kind is evaluated, Mathematica creates a rule for the symbol *f*, which it uses every time when *f* is used. The rule is stored as a DownValue of *f*:

```
DownValues[f]
```

```
{HoldPattern[f(x_)] :→ x^2}
```

```
{HoldPattern[f(x_)] :→ x^2}
```

Here x_ is a "pattern", which stands for "anything", with a temporary name "x". The rule says "change f(anything) to anything^2". HoldPattern prevents evaluation of f(x_) (which would otherwise be replaced by x_^{2} but f[x_] is treated as a pattern for matching purposes.

So what happens when we evaluate definitions of this kind is this: *Mathematica* makes certain rules, stores them, and then applies them in a certain order. Here is an example:

```
Clear["Global`*"]

f[x_Real] := 3

f[1] := 2

f[x_Symbol] := x^2

f[x_Integer] := 5

DownValues[f]
```

```
{HoldPattern[f(1)] :→ 2, HoldPattern[f(x_Real)] :→ 3,
  HoldPattern[f(x_Symbol)] :→ x^2, HoldPattern[f(x_Integer)] :→ 5}
```

```
DownValues[f] = Rest[DownValues[f]]
```

```
{HoldPattern[f(x_Real)] :→ 3, HoldPattern[f(x_Symbol)] :→ x^2, HoldPattern[f(x_Integer)] :→ 5}
```

```
DownValues[f]
```

```
{HoldPattern[f(x_Real)] :→ 3, HoldPattern[f(x_Symbol)] :→ x^2,
  HoldPattern[f(x_Integer)] :→ 5, HoldPattern[f(x_)] :→ 0}
```

```
f[2 / 3]
```

```
0
```

```
f[x_] := 0
```

```
f["cat"]
```

```
f(cat)
```

```
f[1]
```

```
5
```

```
f[x_] := 0
```

```
f["dog"]
```

```
0
```

```
Clear[f]
```

```
DownValues[f]
```

```
{}
```

```
f[2]
```

```
5
```

```
f[7]
```

```
5
```

```
f[1.1]
```

```
3
```

```
f[a]
```

a^2

```
f["cat"]
```

```
f(cat)
```

```
Map[f, {.5, 1, a}]
```

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

The order in which rules are applied by Mathematica is roughly determined by two facts; more specific rules are applied before more general rules, and rules of equal generality are applied in the order they are entered.

In addition to DownValues there are also OwnValues and UpValues (and some other Values) created as follows:

```
Clear[a]
```

```
a = 1; OwnValues[a]
```

```
{HoldPattern[a] :→ 1}
```

```
ClearAll[b];
```

```
b /: Sin[b] = 2;
```

```
b /: Cos[b] = 2;
```

$\cos^2(b) + \sin^2(b)$

```
8
```

```
UpValues[b]
```

```
{HoldPattern[cos(b)] :→ 2, HoldPattern[sin(b)] :→ 2}
```

When an expression is evaluated, *Mathematica* applies the rules contained in UpValues, DownValues, and so on in a certain order, after which it applies the built-in rules. It keeps evalutating the resulting expression until it stops changing. Note also that certain built in rules are applied by *Mathematica* automatically on evaluation but others require using a special function such as Simplify or FullSimplify. For example the transformation

Clear[*a*]

$\boldsymbol{a^n\, a^m}$

a^{m+n}

while

$\mathbf{\cos(\alpha)^2 + \sin(\alpha)^2}$

$\sin^2(\alpha) + \cos^2(\alpha)$

does not automatically simplify to 1 but

$\mathbf{Simplify}\left[\cos^2(b) + \sin^2(b)\right]$

8

Some simplifications only work with specific assumptions:

$\mathbf{Simplify}\left[\sqrt{x^2}\right]$

$\sqrt{x^2}$

$\mathbf{Assuming}\left[x \geq 0, \mathbf{Simplify}\left[\sqrt{x^2}\right]\right]$

x

$\mathbf{Assuming}\left[x <= 0, \mathbf{Simplify}\left[\sqrt{x^2}\right]\right]$

$-x$

Mathematica generally tries to apply any transformations it knows to an expression until it no longer changes. However, this is not the case when we use ReplaceAll. ReplaceAll looks for patterns in all the parts of an expression, but only looks for one match in each part. So, if we have only more than one rule, we may not obtain all the transformations we wish to get:

```
rules = {Log[x_ y_] :→ Log[x] + Log[y], Log[x_^k_] :→ k Log[x]};
```

$\mathtt{Log}\left[\sqrt{\mathtt{a}\left(\mathtt{b}\,\mathtt{c}^{\mathtt{d}}\right)^{\mathtt{e}}}\right]$ `/.rules`

$\frac{1}{2}\log\left(a\left(b\,c^d\right)^e\right)$

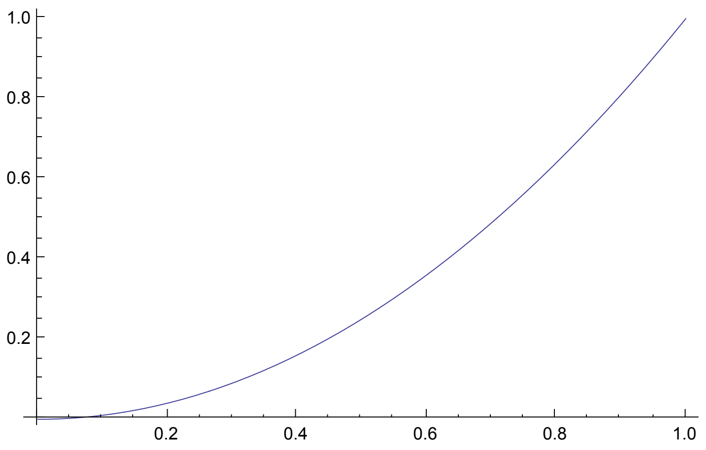
In order to obtain all transformations we should use ReplaceRepeated (//.) instead of ReplaceAll (/.).

$\mathtt{Log}\left[\sqrt{\mathtt{a}\left(\mathtt{b}\,\mathtt{c}^{\mathtt{d}}\right)^{\mathtt{e}}}\right]$ `//.rules`

$\frac{1}{2}(\log(a) + e\,(\log(b) + d\log(c)))$

Another important thing: Options of *Mathematica*'s functions are given as Rules.

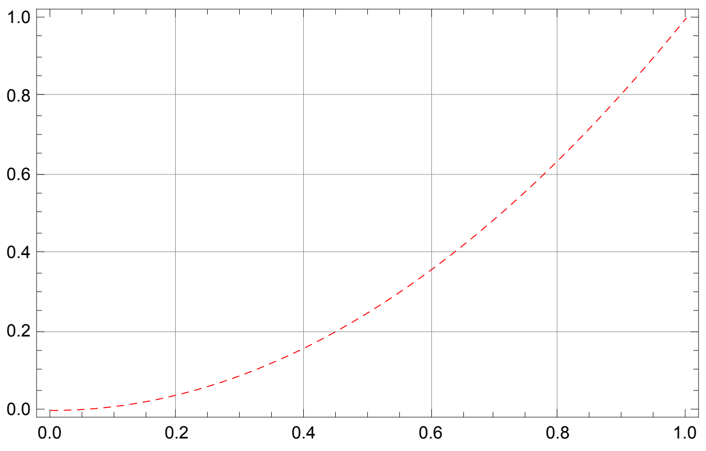
Plot[x^2, {x, 0, 1}]



Options[Plot]

```
{AlignmentPoint → Center, AspectRatio → 1/ϕ, Axes → True, AxesLabel → None,
  AxesOrigin → Automatic, AxesStyle → {}, Background → None, BaselinePosition → Automatic,
  BaseStyle → {}, ClippingStyle → None, ColorFunction → Automatic, ColorFunctionScaling → True,
  ColorOutput → Automatic, ContentSelectable → Automatic, DisplayFunction :→ $DisplayFunction,
  Epilog → {}, Evaluated → Automatic, EvaluationMonitor → None, Exclusions → Automatic,
  ExclusionsStyle → None, Filling → None, FillingStyle → Automatic, FormatType :→ TraditionalForm,
  Frame → False, FrameLabel → None, FrameStyle → {}, FrameTicks → Automatic, FrameTicksStyle → {},
  GridLines → None, GridLinesStyle → {}, ImageMargins → 0., ImagePadding → All, ImageSize → Automatic,
  LabelStyle → {}, MaxRecursion → Automatic, Mesh → None, MeshFunctions → {#1 &}, MeshShading → None,
  MeshStyle → Automatic, Method → Automatic, PerformanceGoal :→ $PerformanceGoal,
  PlotLabel → None, PlotPoints → Automatic, PlotRange → {Full, Automatic},
  PlotRangeClipping → True, PlotRangePadding → Automatic, PlotRegion → Automatic,
  PlotStyle → Automatic, PreserveImageOptions → Automatic, Prolog → {}, RegionFunction → (True &),
  RotateLabel → True, Ticks → Automatic, TicksStyle → {}, WorkingPrecision → MachinePrecision}
```

Plot[x^2, {x, 0, 1}, Axes → False, Frame → True, GridLines → Automatic, PlotStyle → {Red, Dashing[0.01]}]



The difference between := and =

The difference between :=and= is exactly the same as that between :→ and →. Note these Full-Forms:

FullForm[Hold[*a* = 3]]

Hold[Set[*a*, 3]]

FullForm[Hold[*a* := 3]]

Hold[SetDelayed[*a*, 3]]

Consider the following two definitions:

f[p_] := Expand[p]

g[p_] = Expand[p];

? =

```
lhs = rhs evaluates rhs and assigns the result to be
   the value of lhs. From then on, lhs is replaced by rhs whenever
   it appears. {l1, l2, ... } = {r1, r2, ... } evaluates the ri, and

   assigns the results to be the values of the corresponding li. More…
```

? :=

```
lhs := rhs assigns rhs to be the delayed value of
   lhs. rhs is maintained in an unevaluated form. When lhs appears,

   it is replaced by rhs, evaluated afresh each time. More…
```

If we apply them to an expression like $(a + b)^3$ we will get quite different results:

$f\left[(a+b)^3\right]$

$a^3 + 3\,b\,a^2 + 3\,b^2\,a + b^3$

$g\left[(a+b)^3\right]$

$(a+b)^3$

The reason is that = evaluates the right hand side before assigning the evaluated value to the left hand side, while := assigns the unevaluated right hand side to the left hand side.

Links

http : // reference.wolfram.com/mathematica/tutorial/ManipulatingValueLists.html

http : // reference.wolfram.com/mathematica/tutorial/ManipulatingOptions.html

Functions and Functional Programming

Pure Functions

In addition to functions defined by means of global rules *Mathematica* also has "genuine functions", defined as follows:

Function$\left[x, x^3\right]$[c]

c^3

Note that such a function does not need to have a name (so it is called an anonymous function), although we can of course give it a name:

```
f = Function[x, x^3];
```

```
f[3]
```

```
27
```

```
Clear[f]
```

```
OwnValues[f]
```

```
{HoldPattern[f] :→ Function[x, x^3]}
```

We can also, of course, in the same way, construct functions of several variables.

```
Function[{x, y}, x^3 + y^2][4, 2]
```

```
68
```

There are two problems with this approach. First, it is inconvenient to use letters for variable names. This problem is solved by using the notation #1, #2 ,... for the first, second, third etc., arguments. Thus:

```
Function[#1^2 + #2^2][1, 5]
```

```
26
```

Lastly, the word Function can be replaced by the shorthand & after the end of the function, as in

```
#1^2 + #2^2 &[2, 3]
```

```
13
```

Predicates (Boolean Functions)

A common class of functions are functions whose value are the Boolean constants True and False. Such functions are called predicates. Most built in *Mathematica* predicates have names that end in Q:

```
PrimeQ[25]
```

```
False
```

```
PrimeQ[18]
```

```
False
```

```
EvenQ[7]
```

```
False
```

Here are two ways of defining a predicate that test is a number is larger than 5:

```
LargerThanFive[n_] := n > 5
```

```
LargerThanFive[1]
```

```
False
```

```
LargerThanFive[p]
```

```
p > 5
```

Here is the same thing done using a pure function

```
# > 5 &[7]
```

```
True
```

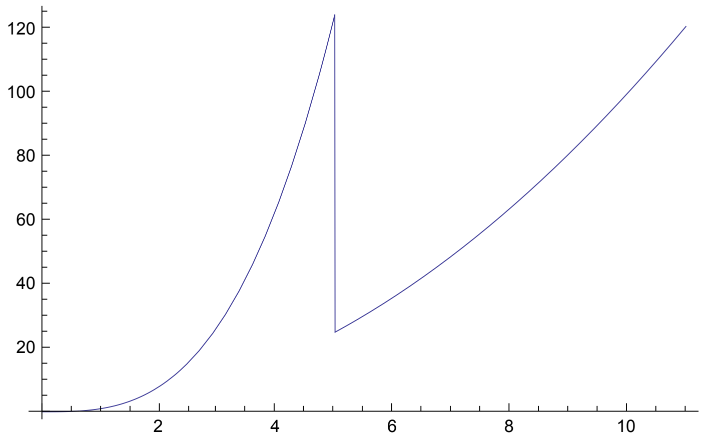
Such pure functions can be used in patterns:

```
Clear[f]
```

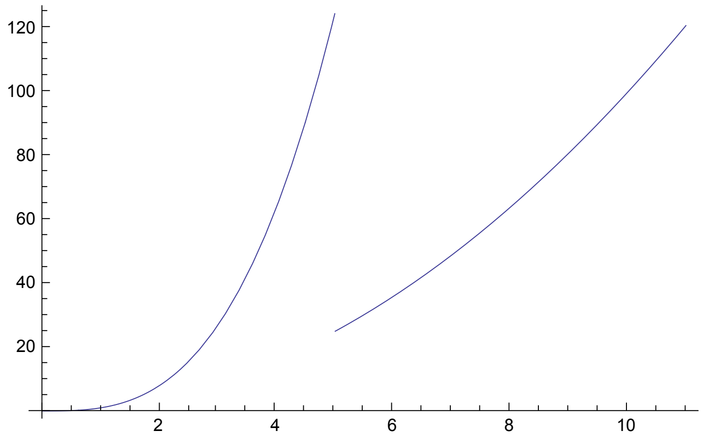
$f[\text{x\_} ? (\# > 5 \,\&)] := x^2$

$f[\text{x\_} ? (\# \leq 5 \,\&)] := x^3$

```
Plot[f[x], {x, 0, 11}]
```



```
Plot[f[x], {x, 0, 11}, Exclusions → {5}]
```



Functions that take Functions as arguments

In functional programming a very important role is played by functions that take functions as arguments. The most important of these are Map and Apply:

```
Clear[f]
```

```
Map[f, {a, b, c, d}]
```

$\{f(a), f(b), f(c), f(d)\}$

Map$\left[f, x^2\, y^3\right]$

$f\left(x^2\right) f\left(y^3\right)$

Map$\left[f, x^2\, y^3, \{1\}\right]$

$f\left(x^2\right) f\left(y^3\right)$

Map$\left[f, x^2\, y^3, \{2\}\right]$

$f(x)^{f(2)}\, f(y)^{f(3)}$

Apply[*f*, *g*[*x*, *y*]]

$f(x, y)$

Apply[Plus, *x* ∗ *y*]

$x + y$

Apply[Times, *x* + *y*]

$x\, y$

Apply[Times, Unevaluated[2 + 3]]

6

Apply[*f*, *g*[*h*[*x*], *k*[*y*]], {1}]

$g(f(x), f(y))$

Short notation:

Map[*f*, expr]

f **/@ expr**

Apply[*f*, expr]

f **@@ expr**

Attributes and Listability

The behavior of *Mathematica* functions and global rules is affected by so called Attributes. Each built-in function has some attributes, for example

Attributes[Sin]

{Listable, NumericFunction, Protected}

Attributes[Plus]

{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}

The most important attribute of functions is the attribute Listable. Let's explain briefly what it does.

Clear[*f*]

ls = Range[10]

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

```
Map[f, ls]
```

$\{f(1), f(2), f(3), f(4), f(5), f(6), f(7), f(8), f(9), f(10)\}$

If we give *f* the Attribute Listable we will not need to use Map.

```
SetAttributes[f, Listable]
```

```
f[ls]
```

$\{f(1), f(2), f(3), f(4), f(5), f(6), f(7), f(8), f(9), f(10)\}$

In addition

```
f[{a, b}, {c, d}]
```

$\{f(a, c), f(b, d)\}$

```
f[a, {b, c}]
```

$\{f(a, b), f(a, c)\}$

The attribute Listable of Plus is the reason for the following behavior:

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

$\{5, 7, 9\}$

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

$\{3, 4, 5, 6\}$

The attributes Orderless, Flat and OneIdentity are interesting, but complicated. Let's see an illustration

```
ClearAll[f]
```

```
f[a, b] /. f[b, x_] → g
```

$f(a, b)$

```
SetAttributes[f, Orderless]
```

```
f[a, b] /. f[b, x_] → g
```

g

```
f[a, b, c] /. f[a, f[b, c]] → g
```

$f(a, b, c)$

```
SetAttributes[f, Flat]
```

```
f[a, b, c] /. f[a, f[b, c]] → g
```

g

```
ClearAll[f]
```

Another group of important attributes are HoldFirst, HoldAll, HoldRest

```
Attributes[Set]
```

{HoldFirst, Protected, SequenceHold}

```
Attributes[SetDelayed]
```

```
{HoldAll, Protected, SequenceHold}
```

```
ClearAll[f]

SetAttributes[f, HoldFirst]

f[x_, y_] := (x = y^2)

x = 3;

f[x, 2];

x
```

```
4
```

Links

http : // reference.wolfram.com/mathematica/tutorial/PureFunctions.html

http : // reference.wolfram.com/mathematica/tutorial/ApplyingFunctionsToListsAndOtherExpression-
s.html

http : // reference.wolfram.com/mathematica/tutorial/Attributes.html

http : // reference.wolfram.com/mathematica/tutorial/SelectingPartsOfExpressionsWithFunctions.html