

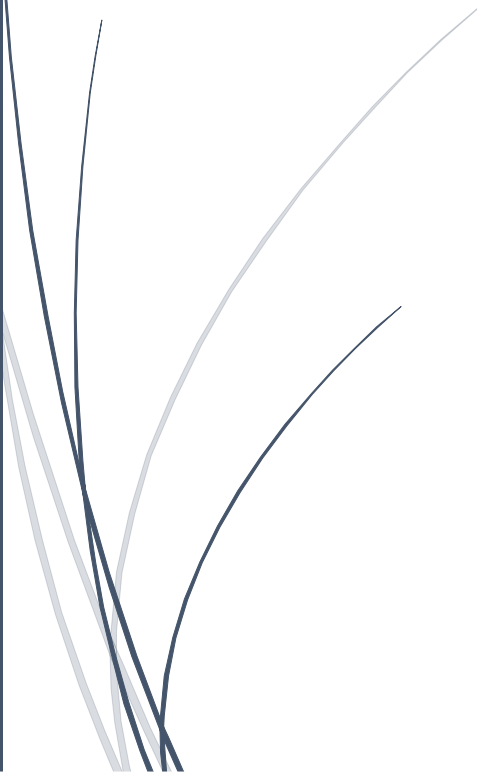
A thick dark blue vertical bar on the left side of the page, with a blue arrow pointing right from its center.

Convex Optimization for Machine Learning

with Mathematica Applications

CHAPTER 1

MATHEMATICA NOTEBOOKS

Several thin, curved lines in dark blue and light gray, resembling stylized grass or reeds, growing from the bottom left corner.

M. M. Hammad
Department of Mathematics
Faculty of Science
Damanhour University
Egypt

CHAPTER 1

MATHEMATICA NOTEBOOKS

1.1 Mathematica Notebooks, Palettes, Packages, and Help

Mathematica is a computer algebra system that performs numeric, symbolic, and graphical computations. Although Mathematica can be used as a programming language, its high-level structure is more appropriate for performing sophisticated operations through the use of built-in functions. For example, Mathematica can find limits, derivatives, integrals, and determinants, as well as plot the graph of functions and perform symbolic computations. The number of built-in functions in Mathematica is enormous. Our goals in this introductory chapter are modest. Namely, we introduce a small subset of Mathematica commands necessary to explore Mathematica discussed in this book.

A notebook is a document which allows us to interact with Mathematica. Each notebook is divided up into a sequence of individual units called cells, each containing a specific type of information such as text, graphics, input or output. Text cells contain information to be read by the user but contain no executable Mathematica commands. The following cell, displaying $(\text{In}[1]:2^{20})$, is an example of an input cell containing executable Mathematica commands. Mathematica computes the value of 2^{20} and the results of the calculation are displayed as $(\text{Out}[1]:1048576)$ in an output cell. When we create a new cell, the default cell type is an input cell. Suppose instead, we want to create a text cell. To do this, use the mouse to click in an area where we want to create a new cell and a horizontal line will appear. Then from the Format menu, select Style and then Text. A new text cell will then be created as soon as we begin typing. We can experiment with creating other types of cells by selecting a cell style of our choice, after first choosing Format and Style from the menu.

A palette is similar to a set of calculator buttons, providing shortcuts to entering commands and symbols into a notebook. The name of a useful palette is “Basic Math Assistant Input” and it can be found by selecting Palettes menu and then Basic Math Assistant. After opening Basic Math Assistant, drag it to the right side of the screen and resize the notebook, if necessary, so that both the notebook and palette are visible in non-overlapping windows. To demonstrate the usefulness of palettes, suppose we wish to calculate $\sqrt{804609}$. The Mathematica command for computing the square root of n is `Sqrt[n]`. The following input cell was created by typing in the information exclusively from the keyboard.

```
Input      : Sqrt[804609]
Output     : 897
```

A quicker and more natural way of entering $\sqrt{804609}$ can be accomplished by clicking on the square root button $\sqrt{\square}$ in the palette and then entering 804609.

```
Input      :  $\sqrt{804609}$ 
Output     : 897
```

Note that, many of Mathematica's functions are available at startup, but additional specialized functions are available from add-in packages. You can load a built-in or installed package in two ways, with the `Needs[]` function or with the symbols `<<`. The package name has quotation marks if you use the `Needs[]` function, but does not with `<<`. Package names are always indicated with a backwards apostrophe at the end of the name.

```
Input      Needs["PackageName`"]
Input      <<PackageName`
```

There are four important tricks to keep in mind to help with Mathematica:

- 1- If you want to know something about a Mathematica function or procedure, just type `?` followed by a Mathematica command name, and then enter the cell to get information on that command.

Input : `?FactorInteger`

After you press Enter Mathematica responds:

Output : `FactorInteger[n]` gives a list of the prime factors of the integer `n`, together with their exponents.

- 2- Mathematica also can finish typing a command for you if you provide the first few letters. Here is how it works: After typing a few letters choose Complete Selection from the Edit menu. If more than one completion is possible, you will be presented with a pop-up menu containing all of the options. Just click on the appropriate choice.
- 3- If you know the name of a command, but have forgotten the syntax for its arguments, type the command name in an input cell, then choose Make Template from the Edit menu. Mathematica will paste a template into the input cell showing the syntax for the simplest form of the command. For example, if you were to type `Plot`, and then choose Make Template, the input cell would look like this:

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

- 4- The Wolfram Documentation is the most useful feature imaginable; learn to use it and use it often. Go to the Help menu and choose Wolfram Documentation. A window will appear displaying the documentation home page.

1.2 Basic Concepts

Arithmetic Operations

Mathematica can be thought of as a sophisticated calculator, able to perform exact as well as approximate arithmetic computations. You can always control grouping the arithmetic computations by explicitly using parentheses. The following list summarizes the Mathematica symbols used for addition, subtraction, multiplication, division and powers.

<code>x+y+z</code>	gives the sum of three numbers.
<code>x*y*z</code> , <code>xx*yxz</code> , or <code>x y z</code>	represents a product of terms.
<code>x-y</code>	is equivalent to <code>x + (-1 * y)</code> .
<code>x^y</code>	gives <code>x</code> to the power <code>y</code> .
<code>x/y</code>	is equivalent to <code>x y⁻¹</code> .

Example 1.1

Input	: <code>2.3+5.63</code>
Output	: <code>7.93</code>
Input	: <code>2.4/8.9^2</code>
Output	: <code>0.0302992</code>
Input	: <code>2*3*4</code>
Output	: <code>24</code>
Input	: <code>(3+4)^2-2(3+1)</code>
Output	: <code>41</code>

Precedence of common operators is generally defined so that "higher-level" operations are performed first. For simple expressions, operations are typically ordered from highest to lowest in the order: 1. Parenthesization, 2. Factorial, 3. Exponentiation, 4. Multiplication and division, 5. Addition and subtraction. Consider the expression $3 \times 7 + 2^2$. This expression has value $(3 \times 7) + (2^2) = 25$.

Mathematica has several built-in constants. The three most commonly used are π , e and i . You can find each of these constants on the Basic Math Assistant palette. Some built-in constants are listed below.

I	$(i = \sqrt{-1}).$
E	$(2.71828).$
Pi	$(\pi = 3.14159).$

Relational and Logical Operators

Relational and logical operators are instrumental in program flow control. They are used in Mathematica to test various conditions involving variables and expressions. The relational operators are listed below.

<code>lhs==rhs</code>	returns True if lhs and rhs are identical.
<code>lhs!=rhs</code> or <code>lhs≠rhs</code>	returns False if lhs and rhs are identical.
<code>x>y</code>	yields True if x is determined to be greater than y.
<code>x>=y</code> or <code>x≥y</code>	yields True if x is determined to be greater than or equal to y.
<code>x<y</code>	yields True if x is determined to be less than y.
<code>x<=y</code> or <code>x≤y</code>	yields True if x is determined to be less than or equal to y.

Logical operators are used to negate or combine relational expressions. The standard logical operators are listed below.

<code>e₁&&e₂&&...</code>	is the logical AND function. It evaluates its arguments in order, giving False immediately if any of them are False, and True if they are all True.
<code>e₁ e₂ ...</code>	is the logical OR function. It evaluates its arguments in order, giving True immediately if any of them are True, and False if they are all False.
<code>!expr</code>	is the logical NOT function. It gives False if expr is True, and True if it is False.

Example 1.2

```

Input      : 10<7
Output     : False
Input      : Pi^E<E^Pi
Output     : True
Input      : 2+2==4
Output     : True
Input      : x^2==1+x
Output     : x^2==1+x
Input      : a!=b
Output     : a≠b
Input      : 1!=2
Output     : True
Input      : 1>2||Pi>3
Output     : True
Input      : 2>1&&Pi>3
Output     : True
Input      : (3 < 5) || (4 < 5)
Output     : True
Input      : (3 < 5) && !(4 > 5)
Output     : True

```

1.3 Elementary Functions

In this section we discuss some of the more commonly used functions Mathematica offers. The Wolfram Language has about 5000 built-in functions. All have names in which each word starts with a capital letter. Remember that the argument of a function must be contained within square brackets, []. Arguments to functions are always separated by commas.

Common Functions

Log[z]	gives the natural logarithm of z (logarithm to base e).
Log[b, z]	gives the logarithm to base b.
Exp[z]	gives the exponential of z.
Sqrt[z] or \sqrt{z}	gives the square root of z.
N[expr]	gives the numerical value of expr.
Abs[z]	gives the absolute value of the real or complex number z.
Floor[x]	gives the greatest integer less than or equal to x.

Trigonometric Functions

Sin[z]	gives the sine of z.
Cos[z]	gives the cosine of z.
Tan[z]	gives the tangent of z.

Hyperbolic Functions

Sinh[z]	gives the hyperbolic sine of z.
Cosh[z]	gives the hyperbolic cosine of z.
Tanh[z]	gives the hyperbolic tangent of z.

Numerical Functions

IntegerPart[x]	integer part of x.
FractionalPart[x]	fractional part of x.
Round[x]	integer x closest to x.
Max[x ₁ , x ₂ , ...]	the maximum of x ₁ , x ₂ , ...
Min[x ₁ , x ₂ , ...]	the minimum of x ₁ , x ₂ , ...
Re[z]	the real part Re z.
Im[z]	the imaginary part Im z.
Conjugate[z]	the complex conjugate z*.

Combinatorial Functions

n!	factorial $n(n-1)(n-2) \times \dots \times 1$.
n!!	double factorial $n(n-2)(n-4) \times \dots \times 1$.
Binomial[n, m]	binomial coefficient $\binom{n}{m} = \frac{(n!)}{(m!(n-m)!)}$.
Multinomial[n ₁ , n ₂ , ...]	multinomial coefficient $\frac{(n_1+n_2+\dots)}{(n_1!n_2!\dots)}$.

Example 1.3

Input	: Log[10, 1000]
Output	: 3
Input	: Exp[I Pi/5]
Output	: E^((I Pi)/5)
Input	: Sin[Pi/3]
Output	: $\sqrt{3}/2$
Input	: Sinh[1.4]
Output	: 1.9043
Input	: N[1/7]
Output	: 0.142857
Input	: Floor[2.4]

```

Output      : 2
Input       : 30!
Output      : 265252859812191058636308480000000
Input       : Binomial[n,2]
Output      : 1/2 (-1+n) n
Input       : Multinomial[6,5]
Output      : 462

```

Sum and Product Functions

Sums and products are of fundamental importance in mathematics, and Mathematica makes their computation simple. Unlike other computer languages, initialization is automatic and the syntax is easy to apply, particularly if the Basic Math Assistant Input palette is used. Any symbol may be used as the index of summation. Negative increments are permitted wherever increment is used.

<code>Sum[f, {i, i_{max}}]</code>	evaluates the $\sum_{i=1}^{i_{\max}} f$.
<code>Sum[f, {i, i_{min}, i_{max}}]</code>	starts with $i = i_{\min}$.
<code>Sum[f, {i, i_{min}, i_{max}, di}]</code>	uses steps di .
<code>Sum[f, {i, {i₁, i₂, ...}}]</code>	uses successive values i_1, i_2, \dots .
<code>Sum[f, {i, i_{min}, i_{max}}, {j, j_{min}, j_{max}}, ...]</code>	evaluates the multiple sum $\sum_{i=1}^{i_{\max}} \sum_{j=1}^{j_{\max}} f$.
<code>Product[f, {i, i_{max}}]</code>	evaluates the $\prod_{i=1}^{i_{\max}} f$.
<code>Product[f, {i, i_{min}, i_{max}}]</code>	starts with $i = i_{\min}$.
<code>Product[f, {i, i_{min}, i_{max}, di}]</code>	uses steps di .
<code>Product[f, {i, {i₁, i₂, ...}}]</code>	uses successive values i_1, i_2, \dots .
<code>Product[f, {i, i_{min}, i_{max}}, {j, j_{min}, j_{max}}, ...]</code>	evaluates the multiple sum $\prod_{i=1}^{i_{\max}} \prod_{j=1}^{j_{\max}} f$.

Example 1.4

```

Input      : Sum[i^2, {i, 10}]
Output     : 385
Input      : Sum[i^2, {i, 1, n}]
Output     : 1/6 n (1+n) (1+2n)
Input      : Sum[1/i^6, {i, 1, Infinity}]
Output     : π^6/945
Input      : Sum[1/(j^2 (i + 1)^2), {i, 1, Infinity}, {j, 1, i}]
Output     : π^4/120
Input      : Product[i^2, {i, 1, 6}]
Output     : 518400
Input      : Product[i^2, {i, 1, n}]
Output     : (n!)^2
Input      : Product[2^j+i, {i, 1, p}, {j, 1, i}]
Output     : 2^(1/2 p (1+p)^2)

```

Limit and Series Functions

<code>Limit[expr, x→x₀]</code>	finds the limiting value of <i>expr</i> when <i>x</i> approaches <i>x₀</i> .
<code>Series[f, {x, x₀, n}]</code>	generates a power series expansion for <i>f</i> about the point $x=x_0$ to order $(x - x_0)^n$.

Example 1.5

```

Input      : Limit[(Sin[x])/x, x->0]
Output     : 1
Input      : Limit[(1+x^n)^n, n->Infinity]
Output     : E^x

```

```

Input      : Series[Exp[x], {x, 0, 10}]
Output     : 1+ x + x^2/2 + x^3/6 + x^4/24 + x^5/120 + x^6/720 + x^7/5040 + x^8/40320
            + x^9/362880 + x^10/3628800 + O[x]^11
Input      : Series[f[x], {x, a, 3}]
Output     : f[a]+f'[a] (x-a)+1/2 f''[a] (x-a)^2+1/6 f^(3)[a] (x-a)^3+ O[x-a]^4
Input      : Series[x^x, {x, 0, 4}]
Output     : 1+Log[x]x+1/2 Log[x]^2 x^2+1/6 Log[x]^3 x^3+1/24 Log[x]^4 x^4+O[x]^5

```

Differentiation Function

$D[f, x]$ gives the partial derivative $\frac{\partial}{\partial x}f$.
 $D[f, x, y, \dots]$ gives the derivative $\frac{\partial}{\partial x} \frac{\partial}{\partial y} \dots f$.
 $D[f, \{x, n\}]$ gives the multiple derivative $\frac{\partial^n}{\partial x^n}f$.

Example 1.6

```

Input      : D[x^n, x]
Output     : n x^(-1+n)
Input      : D[Sin[x]^10, {x, 4}]
Output     : 5040 Cos[x]^4 Sin[x]^6-4680 Cos[x]^2 Sin[x]^8+280 Sin[x]^10
Input      : D[(Sin[xy])/((x^2+y^2)), x, y]
Output     : -(2 x^2 Cos[xy])/(x^2+y^2)^2 -(2 y^2 Cos[x y])/(x^2+y^2)^2 +(Cos[x
            y])/(x^2+y^2 )+(8 x y Sin[x y])/(x^2+y^2)^3 -(x y Sin[x y])/(x^2+y^2 )
Input      : D[x f[x] f'[x], x]
Output     : f[x] f' [x] + x f' [x]^2+ x f[x] f'' [x]
Input      : D[Sin[x] Cos[x+y], x, y]
Output     : -Cos[x+y] Sin[x]-Cos[x] Sin[x+y]
Input      : D[ArcCoth[x], {x, 2}]
Output     : 2 x/ (1-x^2)^2

```

Integration Functions

$\text{Integrate}[f, x]$ gives the indefinite integral $\int f dx$.
 $\text{Integrate}[f, \{x, x_{\min}, x_{\max}\}]$ gives the definite integral $\int_{x_{\min}}^{x_{\max}} f dx$.
 $\text{Integrate}[f, \{x, x_{\min}, x_{\max}\}, \{y, y_{\min}, y_{\max}\}, \dots]$ gives the multiple integral $\int_{x_{\min}}^{x_{\max}} dx \int_{y_{\min}}^{y_{\max}} dy \dots f$.

Example 1.7

```

Input      : Integrate[1/((x^3+1)), x]
Output     : (ArcTan[(-1+2 x)/√3])/√3+1/3 Log[1+x]-1/6 Log[1-x+x^2]
Input      : ∫Sqrt[x+ Sqrt[x]]dx
Output     : 1/12 √(√x+x) (-3+2 √x+8 x)+1/8 Log[1+2√x+2√(√x+x)]
Input      : Integrate[1/((x^4+x^2+1)), {x, 0, Infinity}]
Output     : π/(2√3)
Input      : Integrate[x/(Sqrt[1-x]), {x, 0, 1}]
Output     : 4/3
Input      : Integrate[1/((2+x^2) Sqrt[4+3 x^2]), {x, -Infinity, Infinity}]
Output     : ArcCosh[√(3/2)]
Input      : Integrate[x^2+y^2, {x, 0, 1}, {y, 0, x}]
Output     : 1/3

```

Algebraic Operations

Mathematica has many functions for transforming algebraic expressions. The following list summarizes them.

<code>Simplify[expr]</code>	performs a sequence of algebraic and other transformations on <code>expr</code> and returns the simplest form it finds.
<code>Expand[expr]</code>	expands out products and positive integer powers in <code>expr</code> .
<code>Factor[expr]</code>	factors a polynomial over the integers.
<code>Together[expr]</code>	puts terms in a sum over a common denominator, and cancels factors in the result.
<code>ExpandAll[expr]</code>	expands out all products and integer powers in any part of <code>expr</code> .
<code>FunctionExpand[expr]</code>	tries to expand out special and certain other functions in <code>expr</code> when possible reducing compound arguments to simpler ones.
<code>Reduce[expr, vars]</code>	reduces the statement <code>expr</code> by solving equations or inequalities for <code>vars</code> and eliminating quantifiers.

Example 1.8

```
Input      : Simplify[Sin[x]^2+Cos[x]^2]
Output     : 1
Input      : Expand[(1+x)^10]
Output     : 1+10 x+45 x^2+120 x^3+210 x^4+252 x^5+210 x^6+120 x^7+45 x^8+10 x^9+x^10
Input      : Factor[x^10-1]
Output     : (-1+x) (1+x) (1-x+x^2-x^3+x^4) (1+x+x^2+x^3+x^4)
Input      : Together[x^2/(x^2-1)+x/(x^2-1)]
Output     : (x/((-1+x)))
Input      : ExpandAll[1/(1+x)^3+Sin[(1+x)^3]]
Output     : (1/((1+3 x+3 x^2+x^3)))+Sin[1+3 x+3 x^2+x^3]
Input      : FunctionExpand[Sin[24 Degree]]
Output     : -(1/8) Sqrt[3] (-1-Sqrt[5]) -1/4 Sqrt[1/2 (5-Sqrt[5])]
```

Solving Equations

Solutions of general algebraic equations may be found using the `Solve` command. `Solve` always tries to give you explicit formulas for the solutions to equations. However, it is a basic mathematical result that, for sufficiently complicated equations, explicit algebraic formulas in terms of radicals cannot be given. If you have an algebraic equation in one variable, and the highest power of the variable is at most four, then the Wolfram Language can always give you formulas for the solutions. However, if the highest power is five or more, it may be mathematically impossible to give explicit algebraic formulas for all the solutions. You can also use the Wolfram Language to solve sets of simultaneous equations. You simply give the list of equations and specify the list of variables to solve for. Not all algebraic equations are solvable by Mathematica, even if theoretical solutions exist. If Mathematica is unable to solve an equation, it will represent the solution in a symbolic form. For the most part, such solutions are useless, and a numerical approximation is more appropriate. Numerical approximations are obtained with the command `NSolve`.

<code>Solve[lhs==rhs, x]</code>	solve an equation for <code>x</code> .
<code>Solve[{lhs1==rhs1, lhs2==rhs2, ...}, {x, y, ...}]</code>	solve a set of simultaneous equations for <code>x, y, ...</code> .
<code>Eliminate[{lhs1==rhs1, lhs2==rhs2, ...}, {x, ...}]</code>	eliminate <code>x, ...</code> in a set of simultaneous equations.
<code>Reduce[{lhs1==rhs1, lhs2==rhs2, ...}, {x, y, ...}]</code>	give a set of simplified equations, including all possible solutions.
<code>NSolve[expr, vars]</code>	attempts to find numerical approximations to the solutions of the system <code>expr</code> of equations or inequalities for the variables <code>vars</code> .
<code>FindRoot[f, {x, x0}]</code>	searches for a numerical root of <code>f</code> , starting from the point <code>x = x0</code> .

Example 1.9

```
Input      : Solve[x^2+ax+1==0,x]
Output     : {{x→1/2 (-a-√(-4+a^2)),{x→1/2 (-a+√(-4+a^2))}}
Input      : Solve[a x+y==7&& b x-y==1,{x,y}]
Output     : {{x→8/(a+b),y→-(a-7 b)/(a+b)}}
Input      : Eliminate[{x==2+y,y==z},y]
Output     : 2+z==x
Input      : NSolve[x^5-2x+3==0,x,Reals]
Output     : {{x→-1.42361}}
Input      : Reduce[x^2-y^3==1,{x,y}]
Output     : y==(-1+x^2)^(1/3)||y==(-1)^(1/3)(-1+x^2)^(1/3)||y==(-1)^(2/3)(-1+x^2)^(1/3)
Input      : FindRoot[Sin[x]+Exp[x],{x,0}]
Output     : {x→-0.588533}
```

Some Notes

1- In doing calculations, you will often need to use previous results that you have got. In the Wolfram Language, % always stands for your last result.

%	the last result generated.
%%	the next-to-last result.
% n	the result on output line Out[n].
Out[n]	is a global object that is assigned to be the value produced on the n th output line.

Example 1.10

```
Input      : 77^2
Output     : 5929
Input      : %+1
Output     : 5930
Input      : 3 % + %^2 + %%
Output     : 35188619
Input      : % 2+% 3
Output     : 175943095
```

2- When you write programs in the Wolfram Language, there are various ways to document your code. As always, by far the best thing is to write clear code, and to name the objects you define as explicitly as possible. Sometimes, however, you may want to add some "commentary text" to your code, to make it easier to understand. You can add such text at any point in your code simply by enclosing it in matching (* and *). Notice that in the Wolfram Language, "comments" enclosed in (* and *) can be nested in any way.

(<i>*text*</i>)	a comment that can be inserted anywhere in Wolfram Language code.
-------------------	---

Example 1.11

```
Input      : If[a > b,(*then*) p,(*else*) q]
Output     : If[a > b, p, q]
```

3- Although Mathematica is a powerful calculating tool, it has its limits. Sometimes it will happen that the calculations you tell Mathematica to do are too complicated or may be the output produced is too long. In these cases, Mathematica could be calculating for too long to get an output so you might want to stop these calculations. To abort a calculation: go to "Kernel" and select "Abort evaluation". It can take long to abort a calculation. If the computer does not respond an alternative is to close down the Kernel. By doing this you do not lose the data displayed in your

notebooks, but you do lose all the results obtained so far from the Kernel, so in case you are running a series of calculations, you would have to start again. To close down the Kernel: go to "Kernel" and select "Quit Kernel" and then "Local". Closing down the Kernel is not a practice that is done only when you want to stop a calculation. Sometimes, when you have been using Mathematica for a long time you forget about the definitions and calculations that you have done before (maybe you have defined values for variables or functions, for example). Those definitions can clash with the calculations you are doing, so you might want to close down the Kernel and start your new calculations from scratch. In general, it is a good idea to close down the Kernel after you have finished with a series of calculations, so that when you move to a different problem your new calculations do not interact with the previous ones.

1.4 Variables and Functions

When you do long calculations, it is often convenient to give names to your intermediate results. Just as in standard mathematics, or in other computer languages, you can do this by introducing named variables. It is very important to realize that values you assign to variables are permanent. Once you have assigned a value to a particular variable, the value will be kept until you explicitly remove it. The value will, of course, disappear if you start a whole new Wolfram Language session.

<code>x=value</code>	assign a value to the variable x.
<code>x=y=value</code>	assign a value to both x and y.
<code>x=.or Clear[x]</code>	remove any value assigned to x.
<code>{x,y}={value₁,value₂}</code>	assign different values to x and y.
<code>{x,y}={y,x}</code>	interchange the values of x and y.

Example 1.12

```

Input      : x=5
Output     : 5
Input      : x^2
Output     : 25
Input      : x=7+4
Output     : 11

```

In Mathematica one can substitute an expression with another using rules. In particular one can substitute a variable with a value without assigning the value to the variable.

<code>lhs:=rhs</code>	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.
<code>expr/.rules</code>	applies a rule or list of rules in an attempt to transform each subpart of an expression expr.
<code>lhs->rhs or lhs→rhs</code>	represents a rule that transforms lhs to rhs.

Example 1.13

```

Input      : f[x_] := Expand[x^2]
Input      : f[a + b]
Output     : a^2 + 2 a b + b^2
Input      : g[x_] = Expand[x^2]
Output     : x^2
Input      : g[a+b]
Output     : (a + b)^2
Input      : x+y/.x→2

```

```

Output      : 2+y
Input       : x+y/.{x -> a, y -> b}
Output      : a+b
Input       : x^2 + y/.x -> y/.y -> x
Output      : x + x^2
Input       : x + 2y/.{x -> y, y -> a}
Output      : 2a + y

```

The last example reveals that Mathematica goes through the expression only once and replaces the rules. If we need Mathematica to go through the expression again and replace any expression which is possible until no substitution is possible, one uses //. . In fact /. and //. are shorthand for Replace and ReplaceRepeated, respectively.

Example 1.14

```

Input       : {x,x^2,a,b} /. x->3
Output      : {3,9,a,b}
Input       : x + 2y//.{x -> y, y -> a}
Output      : 3a
Input       : x + 2 y //. {x -> b, y -> a, b -> c}
Output      : 2 a + c
Input       : x + 2 y /. {x -> b, y -> a, b -> c}
Output      : 2 a + b
Input       : Sin[x]/.Sin->Cos
Output      : Cos[x]

```

There are many functions that are built into the Wolfram Language. Here we discuss how you can add your own simple functions to the Wolfram Language. As a first example, consider adding a function called f which squares its argument. The Wolfram Language command to define this function is $f[x]:=x^2$. The names like f that you use for functions in the Wolfram Language are just symbols. Because of this, you should make sure to avoid using names that begin with capital letters, to prevent confusion with built-in Wolfram Language functions. You should also make sure that you have not used the names for anything else earlier in your session.

f[x]=value	definition for a specific expression x.
f[x_]=value	definition for any expression, referred to as x.
Clear[f]	clear all definitions for f.
Function[x,body]	is a pure function with a single formal parameter x.
Function[{x ₁ ,x ₂ ,...},body]	is a pure function with a list of formal parameters.

The _ (referred to as "blank") on the left-hand side is very important.

Example 1.15

```

Input       : f[x_]:=x^2
Input       : f[a+1]
Output      : (1+a)^2
Input       : f[4]
Output      : 16
Input       : f[3 x+x^2]
Output      : (3 x+x^2)^2
Input       : Expand[f[(x+1+y)]]
Output      : 1+2 x+x^2+2 y+2 x y+y^2
Input       : Function[u, 3 + u][x]
Output      : 3 + x
Input       : Function[{u, v}, u^2 + v^4][x, y]
Output      : x^2 + y^4

```

One can define functions of several variables. Here is a simple example defining $f(x, y) = \sqrt{x^2 + y^2}$

Example 1.16

```
Input[1] : f[x_, y_] := Sqrt[x^2 + y^2]
Input[2] : f[3, 4]
Output[2] : 5
```

Some Notes

- 1- There are four kinds of bracketing used in the Wolfram Language. Each kind of bracketing has a very different meaning.

(term)	parentheses for grouping.
f[x]	square brackets for functions.
{a,b,c}	curly braces for lists.
v[[i]]	double brackets for indexing (Part[v,i]).

- 2- Compound expression

expr ₁ ;expr ₂ ;expr ₃	do several operations and give the result of the last one.
expr ₁ ;expr ₂ ;	do the operations but print no output.
expr;	do an operation but display no output.

Example 1.17

```
Input [1] : x=4;y=6;z=y+6
Output[1] : 12
Input [2] : a=2;b=3;a+b
Output[2] : 5
```

- 3- Particularly when you write procedural programs in the Wolfram Language, you will often need to modify the value of a particular variable repeatedly. You can always do this by constructing the new value and explicitly performing an assignment such as x=value. The Wolfram Language, however, provides special notations for incrementing the values of variables, and for some other common cases.

i++	increment the value of i, by 1 returning the old value of i.
i--	decrement the value of i, by 1 returning the old value of i.
++i	pre-increment i, returning the new value of i.
--i	pre-decrement i, returning the new value of i.
i+=di	add di to the value of i and returns the new value of i.
i-=di	subtract di from i and returns the new value of i.
x*=c	multiply x by c.
x/=c	divide x by c.

Example 1.18

```
Input [1] : k = 1; k++
Output[1] : 1
Input [2] : k
Output[2] : 2
Input [3] : k = x
Output[3] : x
Input [4] : k++
Output[4] : x
Input [5] : k
Output[5] : 1+x
Input [6] : k = 1; ++ k
Output[6] : 2
Input [7] : k
```

```

Output[7]      : 2
Input [8]      : k = 1; k--
Output[8]      : 1
Input [9]      : k
Output[9]      : 0
Input [10]     : k = 1; k -= 5
Output[10]     : -4
Input [11]     : k
Output[11]     : -4

```

- 4- Primarily there are three equalities in Mathematica, $=$, $:=$, $==$. There is a fundamental differences between $=$ and $:=$. Study the following examples:

Example 1.19

Input [1] : $x = 5; y = x + 2;$	Input [1] : $x = 5; y := x + 2;$
Input [2] : y	Input [2] : y
Output[2] : 7	Output[2] : 7
Input [3] : $x = 10$	Input [3] : $x = 10$
Output[3] : 10	Output[3] : 10
Input [4] : y	Input [4] : y
Output[4] : 7	Output[4] : 12
Input [5] : $x = 15$	Input [5] : $x = 15$
Output[5] : 15	Output[5] : 15
Input [6] : y	Input [6] : y
Output[6] : 7	Output[6] : 17

From the examples, it is clear that when we define $y=x+2$ then y takes the value of $x+2$ and this will be assigned to y . No matter if x changes its value, the value of y remains the same. In other words, y is independent of x . But in $y:=x+2$, y is dependent on x , and when x changes, the value of y changes too. Namely using $:=$ then y is a function with variable x . Finally the equality $==$ is used to compare:

Example 1.20

```

Input [1] : 5==5
Output[1] : True
Input [2] : 3==5
Output[2] : False

```

1.5 Lists

Lists are extremely important objects. In doing calculations, it is often convenient to collect together several objects, and treat them as a single entity. Lists give you a way to make collections of objects. Lists are sequences of Mathematica objects separated by commas and enclosed by curly brackets. A list such as $\{3,5,1\}$ is a collection of three objects. But in many ways, you can treat the whole list as a single object. You can, for example, do arithmetic on the whole list at once, or assign the whole list to be the value of a variable.

Defining your own lists is easy. You can, for example, type them in full, like this:

```

Input      : oddList = {81, 3, 5, 7, 9, 11, 13, 15, 17}
Output     : {81, 3, 5, 7, 9, 11, 13, 15, 17}

```

Alternatively, if (as here) the list elements correspond to a rule of some kind, the command `Table` can be used, like this:

```

Input      : oddList = Table[2 n + 1, {n, 0, 8}]
Output     : {1, 3, 5, 7, 9, 11, 13, 15, 17}

```

The functions for obtaining elements of lists are

First[list]	the first element.
Last[list]	the last element.
Part[list,n] or list[[n]]	the nth element.
Part[list,-n] or list[[-n]]	the nth element from the end.
Part[list,{n1,n2,...}] or list[[{n1,n2,...}]]	the list of the n1th, n2th, ... elements.
Take[list,n]	the list of the first n elements.
Take[list,-n]	the list of the last n elements.
Take[list,{m,n}]	the list of the mth through nth elements.
Rest[list]	list without the first element.
Most[list]	list without the last element.
Drop[list,n]	list without the first n elements.
Drop[list,-n]	list without the last n elements.
Drop[list,{m,n}]	list without the mth through nth elements.

Example 1.21

```

Input      : First[{a, b, c}]
Output     : a
Input      : First[{a, b}, {c, d}]
Output     : {a, b}
Input      : Last[{a, b, c}]
Output     : c
Input      : {a, b, c, d, e, f}[[3]]
Output     : c
Input      : {{a, b, c}, {d, e, f}, {g, h, i}}[[2, 3]]
Output     : f
Input      : Take[{a, b, c, d, e, f}, 4]
Output     : {a, b, c, d}
Input      : Rest[{a, b, c, d}]
Output     : {b, c, d}
Input      : Most[{a, b, c, d}]
Output     : {a, b, c}
Input      : Drop[{a, b, c, d, e, f}, 2]
Output     : {c, d, e, f}

```

Some functions for inserting, deleting, and replacing list and sublist elements are

Prepend[list,elem]	insert elem at the beginning of list.
Append[list,elem]	insert elem at the end of list.
Insert[list,elem,i]	insert elem at position i in list.
Insert[list,elem,{i,j,...}]	insert elem at position {i,j,...} in list.
Insert[list,elem,{{i1,j1,...},{i2,...},...}]	insert elem at positions {i1,j1,...},{i2,...}, ... in list.
Delete[list,i]	delete the element at position i in list.
Delete[list,{i,j,...}]	delete the element at position {i,j,...} in list.
Delete[list,{{i1,j1,...},{i2,...},...}]	delete elements at positions {i1,j1,...},{i2,...}, ... in list.
ReplacePart[list,elem,i]	replace the element at position i in list with elem.
ReplacePart[list,elem,{i,j,...}]	replace the element at position {i,j,...} with elem.
ReplacePart[list,elem,{{i1,j1,...},{i2,...},...}]	replace elements at positions {i1,j1,...},{i2,...}, ... with elem.

Example 1.22

```
Input      : Prepend[{a, b, c, d}, x]
Output     : {x, a, b, c, d}
Input      : Append[{a, b, c, d}, x]
Output     : {a, b, c, d, x}
Input      : Insert[{a, b, c, d, e}, x, 3]
Output     : {a, b, x, c, d, e}
Input      : Insert[{a, b, c, d, e}, x, -2]
Output     : {a, b, c, d, x, e}
Input      : Delete[{a, b, c, d}, 3]
Output     : {a, b, d}
Input      : Delete[{a, b, c, d}, {{1}, {3}}]
Output     : {b, d}
Input      : ReplacePart[{a, b, c, d, e}, 3 -> xxx]
Output     : {a, b, xxx, d, e}
Input      : ReplacePart[{a, b, c, d, e}, {2 -> xx, 5 -> yy}]
Output     : {a, xx, c, d, yy}
```

Some functions for rearranging lists are

Sort[list]	sort the elements of list into canonical order.
Union[list]	give a sorted version of list, in which all duplicated elements have been dropped.
Reverse[list]	reverse the order of the elements in list.
RotateLeft[list]	cycle the elements in list one position to the left.
RotateLeft[list,n]	cycle the elements in list n positions to the left.
RotateRight[list]	cycle the elements in list one position to the right.
RotateRight[list,n]	cycle the elements in list n positions to the right.
Permutations[list]	generate a list of all possible permutations of the elements in list.
Partition[list,n]	partition list into nonoverlapping sublists of length n.
Partition[list,n,d]	generate sublists with offset d.
Split[list]	split list into sublists consisting of runs of identical elements.
Transpose[list]	transpose the first two levels in list.
Flatten[list]	flatten out nested lists.
Flatten[list,n]	flatten out the top n levels.
FlattenAt[list,i]	flatten out a sublist that appears as the ith element of list.
FlattenAt[list,{i,j,...}]	flatten out the element of list at position {i,j, ...}.
FlattenAt[list,{{i1,j1,...},{i2,j2,...}}]	flatten out elements of list at several positions.
Join[list1,list2,...]	concatenate lists together.
Union[list1,list2,...]	give a sorted list of all the distinct elements that appear in any of the lists.

Example 1.23

```
Input      : Sort[{d, b, c, a}]
Output     : {a, b, c, d}
Input      : Sort[{4, 1, 3, 2, 2}, Greater]
Output     : {4, 3, 2, 2, 1}
Input      : Union[{1, 2, 1, 3, 6, 2, 2}]
Output     : {1, 2, 3, 6}
Input      : Union[{a, b, a, c}, {d, a, e, b}, {c, a}]
Output     : {a, b, c, d, e}
Input      : Reverse[{a, b, c, d}]
Output     : {d, c, b, a}
```

```

Input      : RotateLeft[{a, b, c, d, e}, 2]
Output     : {c, d, e, a, b}
Input      : RotateRight[{a, b, c, d, e}, 2]
Output     : {d, e, a, b, c}
Input      : Permutations[{a, b, c}]
Output     : {{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a}}
Input      : Partition[{a, b, c, d, e, f}, 2]
Output     : {{a, b}, {c, d}, {e, f}}
Input      : Flatten[{{a, b}, {c, {d}, e}, {f, {g, h}}}]
Output     : {a, b, c, d, e, f, g, h}
Input      : Transpose[{{a, b, c}, {x, y, z}}]
Output     : {{a, x}, {b, y}, {c, z}}
Input      : Join[{a, b, c}, {x, y}, {u, v, w}]
Output     : {a, b, c, x, y, u, v, w}

```

Vectors and matrices in the Wolfram Language are simply represented by lists and by lists of lists, respectively. Functions for generating lists are `Range[]`, `Table[]` and `Array[]`.

Vectors

Mathematica has many functions for generating vectors. The following list summarizes them.

<code>{e₁, e₂, ...}</code>	is a list of elements.
<code>Range[n]</code>	create the list $\{1, 2, 3, \dots, n\}$.
<code>Range[n₁, n₂]</code>	create the list $\{n_1, n_1 + 1, \dots, n_2\}$.
<code>Range[n₁, n₂, dn]</code>	create the list $\{n_1, n_1 + dn, \dots, n_2\}$.
<code>Table[f, {i, n}]</code>	build a length-n vector by evaluating f with $i = 1, 2, \dots, n$.
<code>Length[list]</code>	give the number of elements in list.
<code>List[[i]]</code> or <code>Part[list, i]</code>	give the i^{th} element in the vector list.

Example 1.24

```

Input      : List[a, b, c, d]
Output     : {a, b, c, d}
Input      : v = {x, y}
Output     : {x, y}
Input      : Range[4]
Output     : {1, 2, 3, 4}
Input      : Range[x, x+4]
Output     : {x, 1+x, 2+x, 3+x, 4+x}
Input      : Table[i^2, {i, 10}]
Output     : {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
Input      : Length[{a, b, c, d}]
Output     : 4
Input      : {5, 8, 6, 9}[[2]]
Output     : 8

```

<code>c v</code>	multiply a vector v by a scalar.
<code>a.b</code>	dot product of two vectors a.b.
<code>Cross[a, b]</code>	cross product of two vectors (also input as $a \times b$).
<code>Norm[v]</code>	Euclidean norm of a vector v.
<code>Normalize[v]</code>	gives the normalized form of a vector v.
<code>Orthogonalize [{v₁, v₂, ...}]</code>	gives an orthonormal basis found by orthogonalizing the vectors v _i .

Example 1.25

```
Input      : {a,b,c}.{x,y,z}
Output     : ax+by+cz
Input      : Cross[{a,b,c},{x,y,z}]
Output     : {-cy+bz,cx-az,-bx+ay}
Input      : Norm[{x,y,z}]
Output     :  $\sqrt{\text{Abs}[x]^2+\text{Abs}[y]^2+\text{Abs}[z]^2}$ 
Input      : Normalize[{1,5,1}]
Output     : {1/(3 $\sqrt{3}$ ),5/(3 $\sqrt{3}$ ),1/(3 $\sqrt{3}$ )}
Input      : Orthogonalize[{{1,0,1},{1,1,1}}]
Output     : {{1/ $\sqrt{2}$ ,0,1/ $\sqrt{2}$ },{0,1,0}}
```

Matrix

Mathematica has many functions for generating matrices. The following list summarizes them.

<code>{{a,b},{c,d}}</code>	matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$.
<code>Table[f,{i,m},{j,n}]</code>	build an $m \times n$ matrix by evaluating f with i ranging from 1 to m and j ranging.
<code>List[[i,j]]</code> or <code>Part[list,i,j]</code>	give the i,j th element in the matrix list.
<code>DiagonalMatrix[list]</code>	generate a square matrix with the elements in list on the main.
<code>Dimensions[list]</code>	give the dimensions of a matrix represented by list.
<code>Column[list]</code>	display the elements of list in a column.
<code>c m</code>	multiply a matrix m by a scalar.
<code>a.b</code>	dot product of two matrices $a.b$.
<code>Inverse[m]</code>	matrix inverse m .
<code>MatrixPower[m,n]</code>	gives the n^{th} power of a matrix m .
<code>Det[m]</code>	Determinant m .
<code>Tr[m]</code>	Trace m .
<code>Transpose[m]</code>	Transpose m .
<code>Eigenvalues[m]</code>	Eigenvalues m .
<code>Eigenvalues[m,k]</code>	gives the first k eigenvalues of m .
<code>Eigenvectors[m]</code>	Eigenvectors m .
<code>Eigenvectors[m,k]</code>	gives the first k eigenvectors of m .
<code>Eigensystem[m]</code>	gives a list {values,vectors} of the eigenvalues and eigenvectors of the square matrix m .
<code>Eigensystem[m,k]</code>	gives the eigenvalues and eigenvectors for the first k eigenvalues of m .

Example 1.26

```
Input      : m = {{a, b},{c, d}}
Output     : {{a, b},{c, d}}
Input      : m[[1]]
Output     : {a, b}
Input      : m[[1, 2]]
Output     : b
Input      : v = {x, y}
Output     : {x, y}
Input      : m.v
Output     : {ax+by,cx+dy}
Input      : m.m
Output     : {{a2+bc, ab+bd},{ac+cd, bc+d2}}
Input      : s = Table[i+j, {i,3},{j,3}]
Output     : {{2, 3, 4},{3, 4, 5},{4, 5, 6}}
Input      : DiagonalMatrix[{a, b, c}]
```

```

Output      : {{a, 0, 0}, {0, b, 0}, {0, 0, c}}
Input       : Det[m]
Output      : -bc+ad
Input       : Transpose[m]
Output      : {{a, c}, {b, d}}
Input       : h = Table[1/(i+j-1), {i, 3}, {j, 3}]
Output      : {{1, 1/2, 1/3}, {1/2, 1/3, 1/4}, {1/3, 1/4, 1/5}}
Input       : Inverse[h]
Output      : {{9, -36, 30}, {-36, 192, -180}, {30, -180, 180}}
Input       : r = Table[i+j+1, {i, 3}, {j, 3}]
Output      : {{3, 4, 5}, {4, 5, 6}, {5, 6, 7}}
Input       : Eigenvalues[r]
Output      : {1/2 (15+√249), 1/2 (15-√249), 0}
Input       : Eigenvalues[{{a, b}, {c, d}}]
Output      : {1/2 (a+ d- √(a^2 + 4 b c- 2 a d + d^2 )), 1/2 (a+ d+√(a^2+ 4 b c - 2 a
d + d^2 ))}
Input       : (*Largest 5 eigenvalues:*)
Output      : Eigenvalues[Table[N[1/(i + j + 1)], {i, 50}, {j, 50}], 5]
Output      : {1.56835, 0.320271, 0.0506625, 0.00728517, 0.000970997}
Input       : (*Three smallest eigenvalues*)
Output      : Eigenvalues[Table[N[1/(i + j + 1), 20], {i, 10}, {j, 10}], -3]
Output      : {1.14381991477649725×10-10, 1.04594763489401510×10-12,
4.43502120490729759×10-15}
Input       : Eigenvectors[{{a, b}, {c, d}}]
Output      : {{-(-a+ d+ √(a^2 + 4 b c- 2 a d + d^2 ))/2 c, 1}, {-(-a+d-√(a^2 + 4 b c -
2 a d + d^2 ))/ 2 c, 1}}
Input       : Eigenvectors[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]
Output      : {{-(-7-√33)/(2 (11 + 2 √33)), -(-29 - 5 √33)/(4 (11 + 2 √33)), 1}, {-(-7
- √33)/(2 (-11+2 √33)), -(29 - 5 √33)/(4 (-11 + 2 √33)), 1}, {1, -2, 1}}
Input       : N[%]
Output      : {{0.283349, 0.641675, 1.}, {-1.28335, -0.141675, 1.}, {1., -2., 1.}}
Input       : (*Eigenvectors computed using numerical methods*)
Output      : Eigenvectors[N[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]]
Output      : {{-0.231971, -0.525322, -0.818673}, {-0.78583, -0.0867513, 0.612328},
{0.408248, -0.816497, 0.408248}}
Input[1]    : m = Table[N[1/(i + j + 1)], {i, 10}, {j, 10}];
              : (* Eigenvectors corresponding to the eigenvalues with smallest
              : magnitude*)
Input[2]    : vecs = Eigenvectors[m, -2]
Output      : {{0.000199278, -0.0052208, 0.0482359, -0.213117, 0.487952, -0.525743,
0.0417668, 0.478158, -0.437717, 0.12549}, {-0.0000188938, 0.000663109, -
0.00844045, 0.054171, -0.200069, 0.451308, -0.63202, 0.536247, -0.252425,
0.050584}}
Input       : (*The corresponding eigenvalues*)
Output      : Eigenvalues[m, -2]
Output      : {1.045927766943346×10-12, 4.450005850477068×10-15}
Input       : (*Find the eigenvectors corresponding to the 4 largest eigenvalues, or
as many as there are if fewer*)
Output      : Eigenvectors[Table[N[1/(i + j + 1)], {i, 3}, {j, 3}], UpTo[4]]
Output      : {{-0.703153, -0.549268, -0.451532}, {-0.668535, 0.29444, 0.68291},
{0.242151, -0.782055, 0.574241}}
Input       : Eigensystem[N[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]]
Output      : {{16.1168, -1.11684, -1.30368×10-15}, {{-0.231971, -0.525322, -0.818673},
{-0.78583, -0.0867513, 0.612328}, {0.408248, -0.816497, 0.408248}}}
Input       : Eigensystem[{{a, b}, {c, d}}]
Output      : {{1/2 (a + d - √(a^2 + 4 b c - 2 a d + d^2 )), 1/2 (a + d + √(a^2 + 4 b
c - 2 a d + d^2 ))}, {{-(-a + d + √(a^2 + 4 b c - 2 a d + d^2 ))/2 c, 1},
{-(-a + d - √(a^2 + 4 b c - 2 a d + d^2 ))/2 c, 1}}}
Input       : (* Find the 4 largest eigenvalues and their corresponding eigenvectors,
or as many as there are if fewer*)
Output      : Eigensystem[Table[N[1/(i + j + 1)], {i, 3}, {j, 3}], UpTo[4]]
Output      : {{0.657051, 0.0189263, 0.000212737}, {{-0.703153, -0.549268, -0.451532},
{-0.668535, 0.29444, 0.68291}, {0.242151, -0.782055, 0.574241}}}

```

```

Input      : (*Zero vectors are used when there are more eigenvalues than independent
              eigenvectors*)
              : Eigenvectors[{{2, 1, 0}, {0, 2, 0}, {0, 0, 1}}]
Output     : {1, 0, 0}, {0, 0, 0}, {0, 0, 1}

```

Array

```

Array[f,n]          generates a list of length n, with elements f[i].
Array[f,n,r]        generates a list using the index origin r.
Array[f,n,{a,b}]    generates a list using n values from a to b.
Array[f,{n1,n2,...}] generates an  $n_1 \times n_2 \times \dots$  array of nested lists, with elements f[i1,i2,...].

```

Example 1.27

```

Input [1] : Array[f, 10]
Output[1] : {f[1], f[2], f[3], f[4], f[5], f[6], f[7], f[8], f[9], f[10]}
Input [2] : Array[f, {3, 2}]
Output[2] : {{f[1, 1], f[1, 2]}, {f[2, 1], f[2, 2]}, {f[3, 1], f[3, 2]}}

```

Layout & Tables

Print[expr]	prints expr as output.
MatrixForm[list]	prints with the elements of list arranged in a regular array.
TableForm[list]	prints with the elements of list arranged in an array of rectangular cells.
Grid[{{expr ₁₁ , expr ₁₂ , ...}, {expr ₂₁ , expr ₂₂ , ...}, ...}]	is an object that formats with the expr _{ij} arranged in a two-dimensional grid.
Row[expr ₁ , expr ₂ , ...]	is an object that formats with the expr _i arranged in a row, potentially extending over several lines.
Row[list, s]	inserts s as a separator between successive elements.
Column[expr ₁ , expr ₂ , ...]	is an object that formats with the expr _i arranged in a column, with expr ₁ above expr ₂ , etc.
Multicolumn[list, cols]	is an object that formats with the elements of list arranged in a grid with the indicated number of columns.
Multicolumn[list, {rows, Automatic}]	formats as a grid with the indicated number of rows.

Example 1.28

```

Input [1] : MatrixForm[{{1, 2}, {3, 4}}]
Output[1] :
      ( 1 2 )
      ( 3 4 )

Input [2] : MatrixForm[Table[1/(i + j), {i, 4}, {j, 4}]]
Output[2] :
      1/2  1/3  1/4  1/5
      1/3  1/4  1/5  1/6
      ( 1/4  1/5  1/6  1/7 )
      1/5  1/6  1/7  1/8

Input [3] : TableForm[Table[1/(i + j), {i, 4}, {j, 4}]]
Output[3] :
      1/2  1/3  1/4  1/5
      1/3  1/4  1/5  1/6
      1/4  1/5  1/6  1/7
      1/5  1/6  1/7  1/8

Input [4] : Grid[{{a, b, c}, {x, y, z}}]
Output[4] :
      a  b  c
      x  y  z

Input [5] : Grid[{{a, b, c}, {x, y^2, z^3}}, Frame -> All]

```

```

Output[5]      :


|   |     |     |
|---|-----|-----|
| a | b   | c   |
| x | y^2 | z^3 |


Input [6]      : Row[{aaa, b, cccc}]
Output[6]      : aaabcccc
Input [7]      : Row[{aaa, b, cccc}, "----"]
Output[7]      : aaa----b----cccc
Input [8]      : Column[{1, 12, 123, 1234}]
Output[8]      :
1
12
123
1234
Input [9]      : Column[{1, 22, 333, 4444}, Frame -> True]
Output[9]      :


|      |
|------|
| 1    |
| 22   |
| 333  |
| 4444 |


Input [10]     : Multicolumn[Range[50], {6, Automatic}]
Output[10]     :
1  7  13  19  25  31  37  43  49
2  8  14  20  26  32  38  44  50
3  9  15  21  27  33  39  45
4  10 16  22  28  34  40  46
5  11 17  23  29  35  41  47
6  12 18  24  30  36  42  48

```

1.6 2-D and 3-D Graphing

The graph of a function offers tremendous insight into the function's behavior and can be of great value in the solution of problems in mathematics. One of the outstanding features of Mathematica is its graphing capabilities. Mathematica contains functions for 2-D and 3-D graphing of functions, lists, and arrays of data.

Basic Plotting

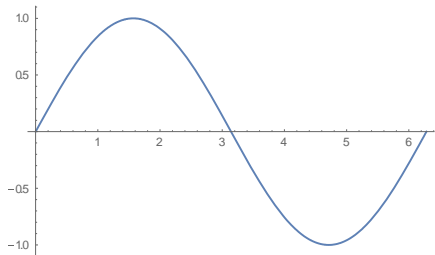
<code>Plot[f, {x, x_{min}, x_{max}}]</code>	plot f as a function of x from x_{\min} to x_{\max} .
<code>Plot[{f₁, f₂, ...}, {x, x_{min}, x_{max}}]</code>	plot several functions together.

Example 1.29

```

Input      : Plot[Sin[x], {x, 0, 2 Pi}]
Output     :

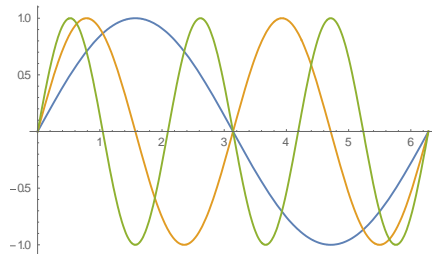
```



```

Input      : Plot[{Sin[x], Sin[2 x], Sin[3 x]}, {x, 0, 2 Pi}]
Output     :

```

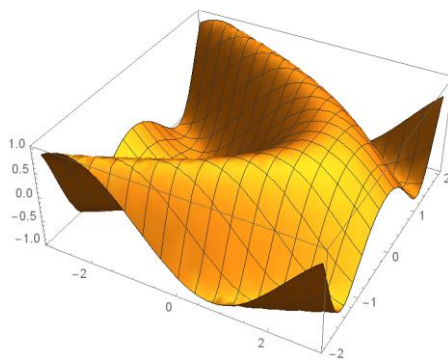


3D Plot

`Plot3D[f,{x,xmin,xmax},{y,ymin,ymax}]` make a three-dimensional plot of f as a function of the variables x and y .

Example 1.30

Input : `Plot3D[Sin[x+y^2],{x,-3,3},{y,-2,2}]`
 Output :



Plotting Lists of Data

`ListPlot[{y1, y2, ...}]`

plot y_1, y_2, \dots at x values 1, 2, ...

`ListPlot[{{x1, y1}, {x2, y2}, ...}]`

plot points $(x_1, y_1), \dots$

`ListLinePlot[list]`

join the points with lines.

`ListPlot3D[array]`

generates a three-dimensional plot of a surface representing an array of height values.

`ListPlot3D[{{x1, y1, z1}, {x2, y2, z2}, ...}]`

generates a plot of the surface with heights z_i at positions x_i, y_i .

`ListPlot3D[{data1, data2, ...}]`

plots the surfaces corresponding to each of the $data_i$.

`ListPointPlot3D[array]`

generates a 3D scatter plot of points with a 2D array of height values.

`ListPointPlot3D[{{x1, y1, z1}, {x2, y2, z2}, ...}]`

generates a 3D scatter plot of points with coordinates x_i, y_i, z_i

`ListPointPlot3D[{data1, data2, ...}]`

plots several collections of points, by default in different colors.

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

makes a density plot of f as a function of x and y .

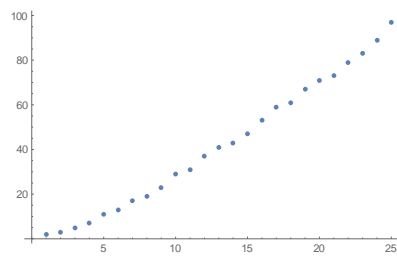
`ContourPlot[f,{x,xmin,xmax},{y,ymin,ymax}]`

generates a contour plot of f as a function of x and y .

Example 1.31

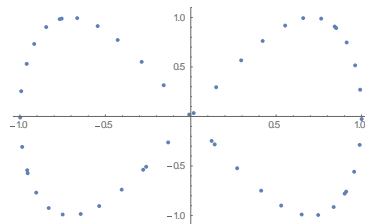
Input : `ListPlot[Prime[Range[25]]]`

Output :



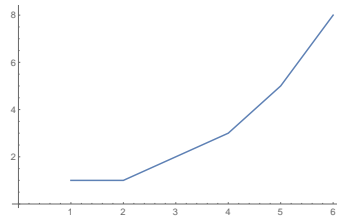
Input : `ListPlot[Table[{Sin[n], Sin[2n]}, {n, 50}]]`

Output :



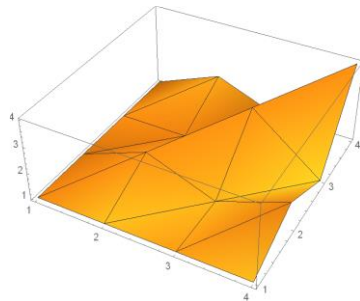
Input : `ListLinePlot[{1, 1, 2, 3, 5, 8}]`

Output :



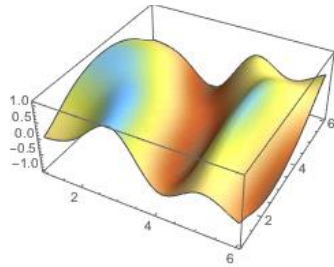
Input : `ListPlot3D[{{1, 1, 1, 1}, {1, 2, 1, 2}, {1, 1, 3, 1}, {1, 2, 1, 4}}, Mesh -> All]`

Output :

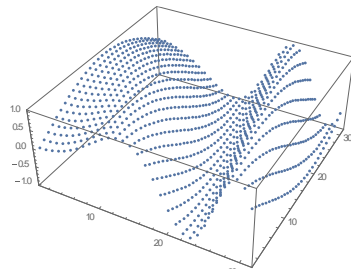


Input : `data= Table[Sin[j^2 + i],{i, 0, Pi, Pi/5},{j, 0, Pi, Pi/5}];
ListPlot3D[data, Mesh->None, InterpolationOrder->3,
ColorFunction->"SouthwestColors"]`

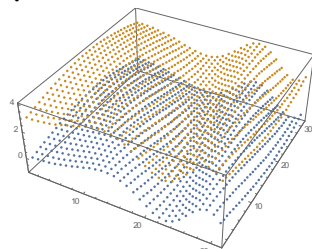
Output :



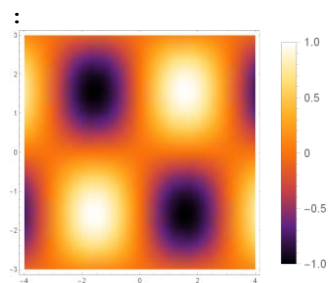
Input : ListPointPlot3D[Table[Sin[j^2 + i], {i, 0, 3, 0.1}, {j, 0, 3, 0.1}]]
 Output :



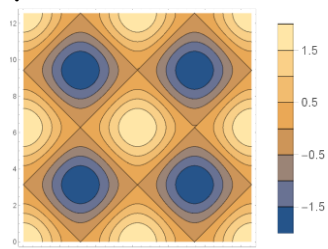
Input : ListPointPlot3D[{Table[Sin[j^2 + i], {i, 0, 3, 0.1}, {j, 0, 3, 0.1}],
 Table[Sin[j^2 + i] + 3, {i, 0, 3, 0.1}, {j, 0, 3, 0.1}]]
 Output :



Input : DensityPlot[Sin[x] Sin[y], {x, -4, 4}, {y, -3, 3}, ColorFunction ->
 "SunsetColors", PlotLegends -> Automatic]
 Output :



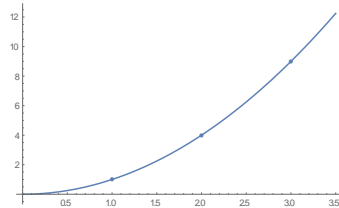
Input : ContourPlot[Cos[x] + Cos[y], {x, 0, 4 Pi}, {y, 0, 4 Pi}, PlotLegends ->
 Automatic]
 Output :



<code>Show[plot₁,plot₂,...]</code>	combine several plots.
<code>GraphicsGrid[{plot₁,plot₂,...},...]</code>	draw an array of plots.
<code>GraphicsRow[{plot₁,plot₂,...}]</code>	draw several plots side by side.
<code>GraphicsColumn[{plot₁,plot₂,...}]</code>	draw a column of plots.

Example 1.32

Input : `Show[Plot[x^2,{x, 0, 3.5}], ListPlot[{1, 4, 9}]]`
Output :

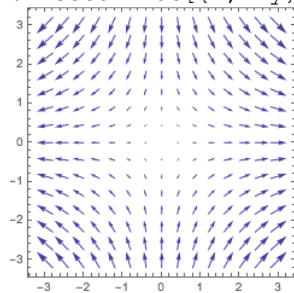


Vector Fields Plots

<code>VectorPlot[{v_x, v_y }, {x, x_{min}, x_{max} }, {y, y_{min}, y_{max} }]</code>	generates a vector plot of the vector field $\{v_x, v_y\}$ as a function of x and y .
<code>VectorPlot3D[{v_x, v_y, v_z }, {x, x_{min}, x_{max} }, {y, y_{min}, y_{max} }, {z, z_{min}, z_{max} }]</code>	generates a 3D vector plot of the vector field $\{v_x, v_y, v_z\}$ as a function of x , y , and z .
<code>VectorDensityPlot[{v_x, v_y }, s], {x, x_{min}, x_{max} }, {y, y_{min}, y_{max} }]</code>	generates a vector plot of the vector field $\{v_x, v_y\}$ as a function of x and y , superimposed on a density plot of the scalar field s .

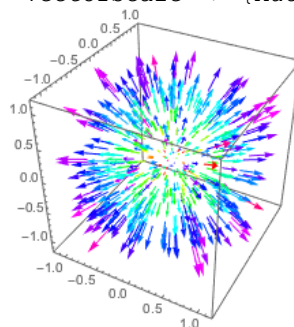
Example 1.33

Input : `VectorPlot[{x, -y}, {x, -3, 3}, {y, -3, 3}]`
Output :

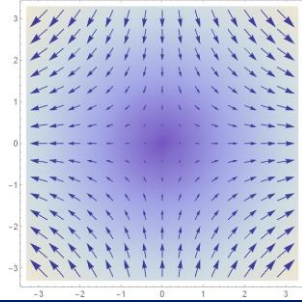


Output : `VectorPlot3D[{x, y, z}, {x, -1, 1}, {y, -1, 1}, {z, -1, 1},
VectorPoints -> points, PlotRange -> All,
VectorScale -> {Automatic, Scaled[0.5]}, VectorColorFunction -> Hue]`

Input



Output : `VectorDensityPlot[{x, -y}, {x, -3, 3}, {y, -3, 3}]`
 Input



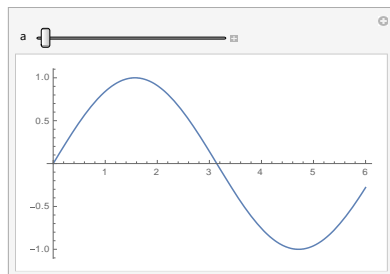
1.7 Manipulate

The single command `Manipulate` lets you create an astonishing range of interactive applications with just a few lines of input. The output you get from evaluating a `Manipulate` command is an interactive object containing one or more controls (sliders, etc.) that you can use to vary the value of one or more parameters. The output is very much like a small applet or widget: it is not just a static result, it is a running program you can interact with.

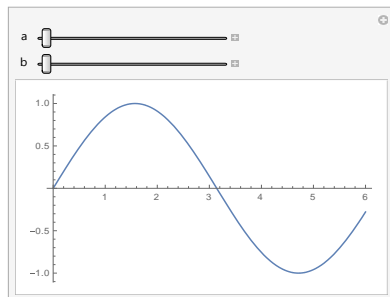
<code>Manipulate[expr, {u, u_{min}, u_{max}}]</code>	generates a version of <code>expr</code> with controls added to allow interactive manipulation of the value of <code>u</code> .
<code>Manipulate[expr, {u, u_{min}, u_{max}, du}]</code>	allows the value of <code>u</code> to vary between <code>u_{min}</code> and <code>u_{max}</code> in steps <code>du</code> .
<code>Manipulate[expr, {{u, u_{init}}, u_{min}, u_{max}, ...}]</code>	takes the initial value of <code>u</code> to be <code>u_{init}</code> .

Example 1.34

Input : `Manipulate[Plot[Sin[x (1 + a x)], {x, 0, 6}], {a, 0, 2}]`
 Output :



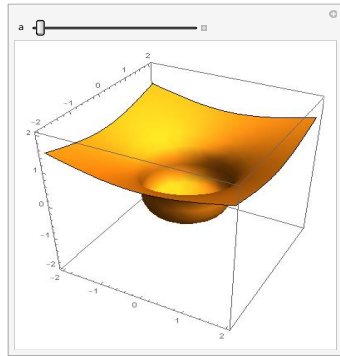
Input : `Manipulate[Plot[Sin[a x + b], {x, 0, 6}], {a, 1, 4}, {b, 0, 10}]`
 Output :



Input : `Manipulate[ContourPlot3D[x^2 + y^2 + a z^3 == 1, {x, -2, 2}, {y, -2, 2}, {z, -2, 2},`

Output

```
Mesh -> None], {a, -2, 2}]
:
```

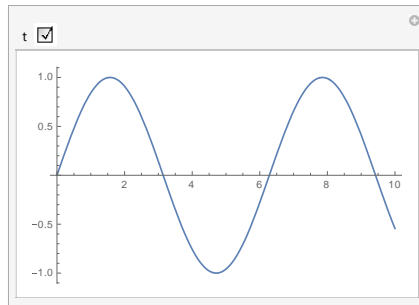


Input

```
: Manipulate[
  Plot[If[t, Sin[x], Cos[x]], {x, 0, 10}], {t, {True, False}}
```

Output

```
:
```

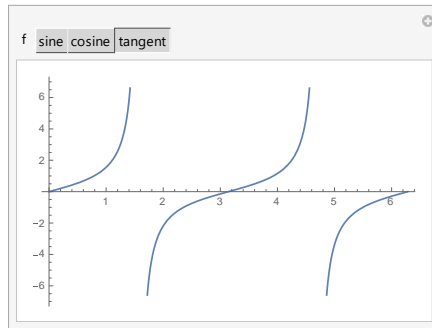


Input

```
: Manipulate[
  Plot[f[x], {x, 0, 2 Pi}], {f, {Sin -> "sine", Cos -> "cosine",
  Tan -> "tangent"}}
```

Output

```
:
```

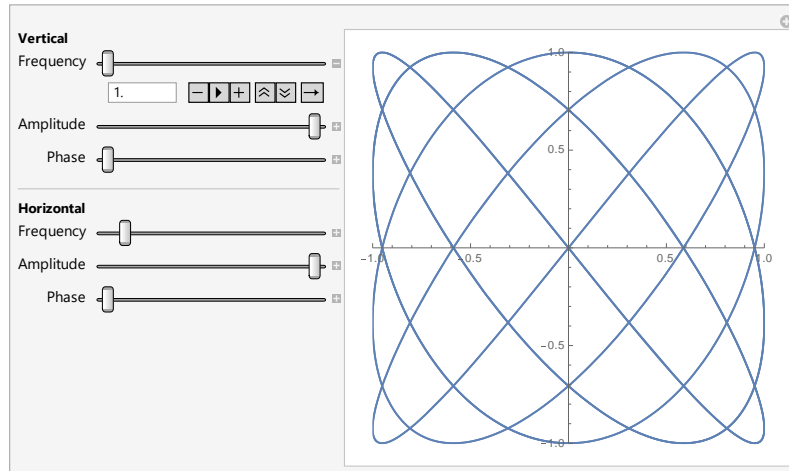


Input

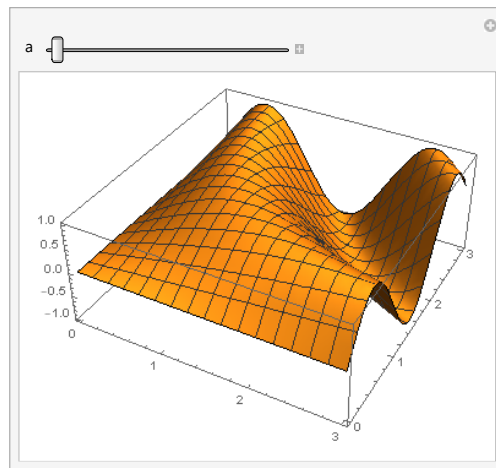
```
: Manipulate[
  ParametricPlot[{a1 Sin[n1 (x + p1)], a2 Cos[n2 (x + p2)]}, {x, 0,
  20 Pi}, PlotRange -> 1, PerformanceGoal -> "Quality",
  Style["Vertical", Bold, Medium], {{n1, 1, "Frequency"}, 1, 4},
  {{a1, 1, "Amplitude"}, 0, 1}, {{p1, 0, "Phase"}, 0, 2 Pi}, Delimiter,
  Style["Horizontal", Bold, Medium], {{n2, 5/4, "Frequency"}, 1,
  4}, {{a2, 1, "Amplitude"}, 0, 1},
  {{p2, 0, "Phase"}, 0, 2 Pi}, ControlPlacement -> Left]
```

Output

```
:
```



Input : Manipulate[Plot3D[Sin[x y + a], {x, 0, 3}, {y, 0, 3}], {a, 0, 1}]
 Output :



1.8 Control Structure

Most of the programming languages use control structures to control the flow of a program. The control structures include decision-making and loops. Decision-making is done by applying different conditions in the program. If the conditions are true, the statements following the condition are executed. The values in a condition are compared by using the comparison operators. The loops are used to run a set of statements several times until a condition is met. If the condition is true, the loop is executed. If the condition becomes false, the loop is terminated, and the control passes to the next statement that follows the loop block.

Conditional Statements

Programmers often need to check the status of a computed intermediate result to branch the program to such or such another block of instructions to pursue the computation. Several examples of the branching condition structures are given next.

```
lhs:=rhs/;test
If[test,then,else]
```

is a definition to be used only if test yields True.
 evaluate then if test is True, and else if it is False.

<code>Which[test₁,value₁,test₂,...]</code>	evaluate the test ₁ in turn, giving the value associated with the first one that is True.
<code>Switch[expr,form₁,value₁,form₂,...]</code>	compare expr with each of the form _i , giving the value associated with the first form it matches.
<code>Switch[expr,form₁,value₁,form₂,...,_,def]</code>	use def as a default value.
<code>Piecewise[{{value₁,test₁},{value₁,test₁},...}]</code>	represents a piecewise function with values value _i in the regions defined by the conditions test _i .
<code>Piecewise[{{value₁,test₁},...],def]</code>	give the value corresponding to the first test ₁ which yields True.

Note that, `If[condition, t, f]` is left unevaluated if condition evaluates to neither True nor False. `If[condition, t]` gives Null if condition evaluates to False.

Example 1.35

```

Input      : If[7>8,x,y]
Output     : y
Input      : If[7>8,Print[x], Print[y]]
Output     : y
Input      : x=2; If[x==0, Print["x is 0"], Print["x is different from 0"]]
Output     : x is different from 0
Input      : x = 3;
            y = 0;
            If[x > 1, y = Sqrt[x], y = x^2];
            Print[y]
            m := If[x > 5, 1, 0];
            Print[m]
Output     : √3
            0
Input      : a=2; Which[a==1,x,a==2,b]
Output     : b
Input      : expr=3; Switch[expr, 1, Print["expr is 1"], 2, Print["expr is 2"], 3,
            Print["expr is 3"], _, Print["expr has some other value"]]
Output     : expr is 3
Input      : k = 2;
            n = 0;
            Switch[k, 1, n = k + 10, 2, n = k^2 + 3, _, n = -1];
            Print[n]
            k = 5;
            n := Switch[k, 1, k + 10, 2, k^2 + 3, _, -1];
            Print[n]
Output     : 7
            -1
Input      : Piecewise[{{x^2,x<0},{x,x>0}}]
Output     :
            {
              x^2  x<0
              x    x==0
              0    true
Input      : Piecewise[{{Sin[x]/x,x<0},{1,x==0}},-x^2/100+1]
Output     :
            {
              Sinx/x    x<0
              1         x==0
              1-x^2/100 true
Input      : x = 4;
            If[x>0, y=Sqrt[x], y=0]
Output     : 2
Input      : x=-4;
            y=Which[x>0,1/x,x<-3,x^2, True, 0]
Output     : 16
Input      : a=-4; b=4;

```

```

y=Switch[a^2, ab, 1.0/a, b^2, 1.0/b, _, 01]
Output      : 0.25

```

Looping Statements

Mathematica has several looping functions, the most common of which is Do[.].

Do [expr, n]	evaluates expr n times.
Do [expr, {i, i _{max} }]	evaluates expr with the variable i successively taking on the values 1 through i _{max} (in steps of 1).
Do [expr, {i, i _{min} , i _{max} }]	starts with i = i _{min}
Do [expr, {i, i _{min} , i _{max} , di}]	uses steps di.
Do [expr, {i, {i ₁ , i ₂ , ...}}]	uses the successive values i ₁ , i ₂ , ...
Do [expr, {i, i _{min} , i _{max} }, {j, j _{min} , j _{max} }, ...]	evaluates expr looping over different values of j etc. for each i.

Example 1.36

```

Input      : Do[Print[5^k], {k, 3, 7}]
Output     :
            125
            625
            3125
            15625
            78125
Input      : t=x; Do[Print[t=1/(1+kt)], {k, 2, 6, 2}]
Output     :
            1/(1+2 x)
            1/(1 + 4/(1 + 2 x))
            1/(1 + 6/(1 + 4/(1 + 2x)))
Input      : Do[Print[{i,j}], {i, 4}, {j, i}]
Output     :
            {1,1}
            {2,1}
            {2,2}
            {3,1}
            {3,2}
            {3,3}
            {4,1}
            {4,2}
            {4,3}
            {4,4}
Input      : sum=0;
            Do[Print[ sum=sum+i], {i, 1, 4}];
            sum
Output     :
            1
            3
            6
            10
            10
Input      : fact = 1;
            Do[Print[ fact = fact*i], {i, 1, 4}];
            fact
Output     :
            1
            2
            6
            24
            24

```

```

Input      : Do[
              Do[
              Do[
                Print["i= ", i , " j= ", j, " k= ", k], {i, 1, 2}]
              , {j, 1, 2}]
            , {k, 1, 2}]
Output     :
i= 1 j= 1 k= 1
i= 2 j= 1 k= 1
i= 1 j= 2 k= 1
i= 2 j= 2 k= 1
i= 1 j= 1 k= 2
i= 2 j= 1 k= 2
i= 1 j= 2 k= 2
i= 2 j= 2 k= 2

Input      : sum=0;
            Do[Print[i,"",sum=sum+i^2],{i,1,6,2}];
            sum
Output     : 1,1
            3,10
            5,35
            35

Input      : Do[
              Do[
                If[Sqrt[i^2 + j^2] ∈ Integers, Print[i, " ", j]],
                {j, i, 10}],
              {i, 1, 10}]
Output     : 3   4
            6   8

Input      : Do[Print[k!], {k,3}]
Output     : 1
            2
            6

Input      : Do[Print[k, " ", k^2, " ", k^3], {k, 3}]
Output     : 1   1   1
            2   4   8
            3   9  27

Input      : Do[Print[k, " squared is ", k^2], {k, 5}]
Output     : 1 squared is 1
            2 squared is 4
            3 squared is 9
            4 squared is 16
            5 squared is 25

Input      : Print["k k^2"]
            Print["-----"]
            Do[Print[k, " ", k^2], {k, 5}]
Output     : k k^2
            -----
            1 1
            2 4
            3 9
            4 16
            5 25

Input      : Do[Print[k], {k, 1.6, 5.7, 1.2}]
Output     : 1.6
            2.8
            4.
            5.2

Input      : Do[Print[k], {k,3(a+b), 8(a+b), 2(a+b)}]
Output     :
            3 (a+b)
            5 (a+b)
            7 (a+b)

```

<code>Nest[f,expr,n]</code>	apply f to expr n times.
<code>FixedPoint[f,expr]</code>	start with expr, and apply f repeatedly until the result no longer changes.
<code>NestList[f,expr,n]</code>	gives a list of the results of applying f to expr 0 through n times.
<code>While[test,body]</code>	evaluate body repetitively, so long as test is.
<code>For[start,test,incr,body]</code>	executes start, then repeatedly evaluates body and incr until test fails to give True.
<code>Break[]</code>	exits the nearest enclosing Do, For, or While.

Example 1.37

```

Input      : Nest[f,x,3]
Output     : f[f[f[x]]]
Input      : Nest[Function[t,1/(1+t)],x,3]
Output     : 1/(1 + 1/(1 + 1/(1 + x)))
Input      : NestList[f,x,4]
Output     : {x,f[x],f[f[x]],f[f[f[x]]],f[f[f[f[x]]]]}
Input      : NestList[Cos,1.0,10]
Output     : {1., 0.540302, 0.857553, 0.65429, 0.79348,
0.701369, 0.76396, 0.722102, 0.750418, 0.731404, 0.744237}
Input      : FixedPoint[Function[t, Print[t]; Floor[t/2]], 67]
Output     :
           67
           33
           16
            8
            4
            2
            1
            0
            0
Input      : n=17; While[n= Floor[n/2]; n !=0, Print[n]]
Output     :
           8
           4
           2
           1
Input      : n=1; While[n<4, Print[n]; n=n+1]
Output     :
           1
           2
           3
Input      : Do[Print[i]; If[i>2, Break[]], {i,10}]
Output     :
           1
           2
           3
Input      : For[i=1;t=x,i^2<10,i=i+1,t=t^2+i; Print[t]]
Output     :
           1+x^2
           2+(1+x^2)^2
           3+(2+(1+x^2)^2)^2
Input      : For[sum=0.0; x=1.0, (1/x) > 0.15, x=x+1, sum=sum+1/x;Print[sum]]
Output     :
           1.
           1.5
           1.83333
           2.08333
           2.28333
           2.45

```

1.9 Modules, Blocks, and Local Variables

Global Variables are those variables declared in Main Program and can be used by Subprograms. Local Variables are those variables declared in Subprograms. The Wolfram Language normally assumes that all your variables are global. This means that every time you use a name like x , the Wolfram Language normally assumes that you are referring to the same object. Particularly when you write subprograms, however, you may not want all your variables to be global. You may, for example, want to use the name x to refer to two quite different variables in two different subprograms. In this case, you need the x in each subprogram to be treated as a local variable. You can set up local variables in the Wolfram Language using modules. Within each module, you can give a list of variables which are to be treated as local to the module.

<code>Module[{x, y, ...}, body]</code>	a module with local variables x, y, \dots
<code>Module[{x=x₀, y=y₀, ...}, body]</code>	a module with initial values for local variables.

Example 1.38

```
Input      : k=25
Output     : 25
Input      : Module[{k}, Do[Print[k, " ", 2^k], {k, 3}]]
Output     :
            1  2
            2  4
            3  8
Input      : k
Output     : 25
```

Thus, we can create programs as series of modules, each performing a specific task. For subtasks, we can imbed modules within other modules to form a hierarchy of operations. The most common method for setting up modules is through function definitions,

Example 1.39

```
Input      : k = 25;
            integerPowers[x_Integer] := Module[{k}, Do[Print[k, " ", x^k], {k, 3}]]
            integerPowers[k]
Output     : 1 2
            2 625
            3 15625
Input      : k
Output     : 25
Input[1]   : t = 17
Output[1]  : 17
Input[2]   : Module[{t}, t = 8; Print[t]]
Output[2]  : 8
Input[3]   : t
Output[3]  : 17
Input[4]   : g[u_] := Module[{t = u}, t += t/(1 + u)]
Input[5]   : g[a]
Output[5]  : a + a/(1 + a)
Input[6]   : h[x_] := Module[{t}, t^2 - 1 /; (t = x - 4) > 1]
Input[7]   : h[10]
Output[7]  : 35
```

The format of Module is `Module[{var1, var2, ...}, body]`, where $var1, var2, \dots$ are the variables we localize, and `body` is the body of the function. The value returned by Module is the value returned by the last operator in the body (unless an explicit `Return[]` statement is used within the body of Module. In this case, the argument of `Return[arg]`

is returned). In particular, if one places the semicolon after this last operator, nothing (Null) is returned. As a variant, it is acceptable to initialize the local variables in the place of the declaration, with some global values: `Module[{var1 = value1, var2, ...}, body]`. However, one local variable (say, the one "just initialized" cannot be used in the initialization of another local variable inside the declaration list. The following would be a mistake: `Module[{var1 = value1, var2 = var1, ...}, body]`. Moreover, this will not result in an error, but just the global value for the symbol `var1` would be used in this example for the `var2` initialization (this is even more dangerous since no error message is generated and thus we don't see the problem). In this case, it would be better to do initialization in steps: `Module[{var1=value1,var2,...}, var2=var1;body]`, that is, include the initialization of part of the variables in the body of `Module`. One can use `Return[value]` statement to return a value from anywhere within the `Module`. In this case, the rest of the code (if any) inside `Module` is slipped, and the result value is returned.

To show how this is done, the following is an example module which will simulate a single gambler playing the game until the goal is achieved or the money is gone.

Example 1.40

```
Input      : GamblersRuin[ a_, c_, p_ ] :=
            Module[ {ranval,var1,var2,var3},
            var1 = a;
            var2 = c;
            var3 = p;
            While[ 0 < var1 < var2,
            ranval = Random[];
            If[ ranval < var3, var1 = var1 + 1, var1 = var1 - 1 ] ];
            Return[var1==var2] ]
```

There are several things to notice in the example. First, this is the same thing we have done in the past to define a function. That is, we have a function name `GamblersRuin` with three input variables, `a`, `c`, and `p`. The operator `:=` is used to start the definition. Secondly the function involves the Mathematica command `Module`. This just tells Mathematica to perform all the commands in the module (like a subroutine in Fortran or method in C++). There are some special features we need to understand in the `Module` command. The `Module` command has two arguments. The first argument is a list of all the local variables that will only be used inside the module. In the example above the local variable list is `{ranval,var1,var2,var3}`. These variables are only used in the module and are cleared once the module is done executing. The second argument is all the commands that will be executed each time the module is called. There are some assignment commands at the beginning that are used to make things cleaner. The module uses temporary variables so that the values of the input variables are not overwritten when the module executes.

The last command is added to our list of input lines to return a result from the work done by the `Module`. Without this we would never get any results from our calculation. Any recognized variable type or structure within Mathematica can be returned by a `Module`. In this example, the returned value is the result of testing two variables in the code for equality. The code fragment `var1==var2` tests to determine if the variables, `var1` and `var2`, are equal. If the two are equal the line outputs `True` and if the two are not equal, the line outputs `False`.

Again, all but the last command must be ended by a semicolon. This is to make sure that the commands are separated in the execution. Commands separated by blank space will be considered as terms to be multiplied together. Leaving out the semicolon will give rise to lots of error messages, wrong results or both.

Modules in Mathematica allow one to treat the names as local. When one uses `Block` then the names are global, but the values are local.

<code>Block[{x,y,...},body]</code>	evaluate expr using local values for <code>x,y, ...</code>
<code>Block[{x=x0,y=y0,...},body]</code>	assign initial values to <code>x,y</code> ; and evaluate ... as above.

`Block[]` is automatically used to localize values of iterators in iteration constructs such as `Do`, `Sum` and `Table`. `Block[]` may be used to pack several expressions into one unity.

Example 1.41

```
Input      : Clear[x,t,a]
Input      : x^2 + 3
Output     : 3+x2
Input      : Block[ {x = a + 1}, %]
Output     : 3+(1+a)2
Input      : x
Output     : x
Input      : t = 17
Output     : 17
Input      : Module[ {t}, Print[t]]
Output     : t$505
Input      : t
Output     : 17
Input      : Block[ {t}, Print[t] ]
Output     : t
```