Chapter 3: Programming in Mathematica

Programming in Mathematica

A program (code) is a sequence of instructions to solve some problem. In Mathematica, we input each instruction and press the “return” key. After all instructions are typed in, we press the “enter” key to execute the sequence.

Individual instructions in the code can be assignment statements, iteration statements (loops), conditional (if), input or output statements, functions, or other programming constructs.

Looping Constructs (Iteration)

Allows repeated evaluation of expressions.

Functions **Do**, **For**, and **While** are similar to looping statements in high-level programming languages.

- **Do Function**
  - Has general forms:

Do[body, {*k*, *kstart*, *kstop*, *dk*}]

  - evaluates *body* repeatedly with *k* varying from *kstart* to *kstop* in steps of *dk*. Can omit *dk*, or both *kstart* and *dk*; default values are 1.

  - body contains one or more expressions separated by semicolons.

Do[body, {*n*}]

  - *body* is evaluated *n* times.

The **Do** function generates no output, it simply evaluates the expression in the body, repeating the evaluation for the specified number of iterations.

Using the **Print** function, we can output the value of an expression at each iteration.

Example Do Statements:

1. List of factorials (using factorial operator I).

```
Do[Print[k!], {k,3}]
1
2
6
```

2. List of negative powers of 2.

```
Do[Print[2^k], {k,0,-2,-1}]
1
1
-                    {obtain results as rational numbers
2
1
-
4
```

3. Table of powers.

```
Do[Print[k," ",k^2" ",k3], {k,3}]
1    1    1        {character strings of two blank spaces each
2    4    8             are used to separate columns
3    9    27
```

4. Another table with character strings.

```
Do[Print[k," squared is ",k^2], {k,5}]


1 squared is 1
2 squared is 5
3 squared is 9
4 squared is 16
5 squared is 25
```

5. A table with column headings.

```
Print["k   k^2"]
Print["_______"]
Do[Print[k,"   ",k^2],{k,5}]
```

k	k^2
1	1
2	4
3	9
4	16
5	25

(A better way to produce tables is to set up lists and use the Table function, which aligns columns. We will discuss this method after we take a look at list structures in general.)

6. Loop variable does not have to be an integer.

- **It can be a floating-point number, as in**

```
Do[Print[k],{k,1.6, 5.7,1.2}]

1.6
2.8
4.
5.2
```

- It can include units, as in

```
Do[Print[k], {k, 2cm, 9cm, 3cm}]

2cm
5cm
8cm
```

- Or it can be an expression, as in

```
Do[Print[k], {k,3(a+b), 8(a+b), 2(a+b)}]

3 (a + b)
5 (a + b)
7 (a + b)
```

7. Nested Loops.

```
Do[Print[{i,j}],{i,4},{j,i-1}]

(2,1)
(3,1)
(3,2)
(4.1)
(4,2)
(4.3)
```

(At i=1, j cannot vary from 1 to 0 in steps of 1; i.e., jstart is bigger than jend.)

8. Body of **Do** Function can contain multiple expressions, separated by semicolons.

```
x=10.0;          {semicolon here suppresses output
Do[Print[x];x=Sqrt[x], {3}]

10.
3.16228
1.77828
```

Output from **Print** function simply produces text on the screen (called a "side effect"), and does not return a value that can be referenced by other functions.

Referencing a **Print** function produces a "Null" result' i.e.,

```
Sqrt[Print[2]]

2
Sqrt[Null]
```

Do function is useful for loops that are to be repeated a fixed number of times. But in many cases, we do not know in advance how many times we want to repeat a loop.

In Mathematica, have two general loop functions that allow a loop to be ended when a particular condition is satisfied.

- **For Function**

Has general form:

For[*initial, test, incr, body*]

- first, the *initial* statements are processed, then the test condition is evaluated; if *test* is true, the *body* is processed, then *incr* is processed. The sequence *test-body-incr* is repeatedly processed until *test* is false.

Example:

Evaluate s*um = sum + 1/x*, starting at *x=1* and continuing as long as *1/x* > 0.15.

```
For[sum=0.0; x=1.0, (1/x) > 0.15, x=x+1,
              sum=sum+1/x;Print[sum]]
```

```
1
1.5
1.83333
2.08333
2.28333
2.45
```

(Note: semicolon used as delimiter between statements in *initial* and in *body.*)

- **While Function**

  Has general form

While[*test*, *body*]

  -where body is a set of statements that are repeatedly processed
  unit test is evaluated to be false.

  Similar to **For** function, except initialization must be given as separate statements.

Example While loops:

```
n=25; While[(n=Floor[n/2])>0, Print[n]]

12
6
3
1
```

```
sum=0.0;
x=1.0;
While[1/x>0.15, sum=sum+1/x;
Print[sum]; x=x+1]

1
1.5
1.83333
2.08333
2.28333
2.45
```

(Note: semicolons after the initialization for sum and x suppress the output for these two statements.)

Mathematica provides abbreviated forms for incrementing variables.

For example, we can use **x++** in place of **x=x+1** (as in C programming language) in the previous example:

```
sum=0.0;
x=1.0;
While[1/x>0.15, sum=sum+1/x;
              Print[sum]; x++]
```

The following table lists abbreviated forms for incrementing and making other assignments to the value of a variable in Mathematica.

x++	Increments in value of x by 1.
x--	Decrements the value of x by 1.
x += dx	Adds dx to the value of x.
x -= dx	Subtracts dx from the value of x.
x *= a	Multiplies the value of x by a.
x /= a	Divides the value of x by a.

Test conditions in **For** and **While** functions are Logical Expressions -- evaluate to TRUE or FALSE.

Form logical expressions with the following operators:

Relational Operator	Type of Test
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Logical (Boolean) Operators	Operation
!	not (“flips” logic value)
&&	and (True if all exprs True)
||	or (True if at least one is True)
Xor[expr1, expr2, . . .	**exclusive or** (True if only one expr True)
===	**equivalence** (True if all exprs have same logic value)
=!=	**nonequivalence** (True if exprs have opposite logic values)

Simple Logical Expressions

- formed with relational operators.

```
3 > 5
False

3 <= 5
True

4+2 == 5+1
True

a != b
a != b
```

(If no values have been assigned to variables a and b, the result cannot be evaluated as True or False.)

```
a=5; b=5; a != b
False
```

Can also form sequence of logical tests as

```
4 < 7 <= 12
True

2 != 4 != 2
False
```

(not all values unequal)

Compound Logical Expressions

- formed with Logical (Boolean) operators and combinations of simple expressions.

```
(3 < 5) || (4 < 5)
True

Xor[3<5, 4<5]
False

(3 < 5) && !(4 > 5)
True

(3 < 4) === (4 < 5) === (5 < 6)
True
```

The following functions can be used in place of the equality and equivalence operators.

Equal[a,b]	a == b
Unequal[a,b]	a != b
SameQ[a,b]	a === b
UnsameQ[a,b]	a =!= b

Note: Avoid testing floating-point variables for equality. Why?

Summation Function

- Has general forms:

Sum[*f*, {*k*, *kstart*, *kstop*}]	$\Sigma_{k\,=\,kstart}^{kstop} f$
Sum[*f*, {*k*, *kstart*, *kstop*, *dk*}]	- with incr. dk
Sum[*f*, {*kstart*, *kstop*}, {*j*, *jstart*, *jstop*}]	- nested sums

Example - Previous summation problem can be accomplished with Sum Function:

```
Sum[1/k, {k, 1,10}]
```

$\frac{7381}{2520}$ (exact result)

```
N[Sum[1/k, {k,1,10}]]
```

2.92897 (decimal approximation)

Product Function

Product[*f*, {*k*, *kstart*, *kstop*}]	$\Pi_{k\,=\,kstart}^{kstop} f$

General-purpose programming languages usually do not contain sum and product functions.)

Conditionals (Decision Statements)

High-level programming languages generally make decisions as to which processing steps should be executed next using an **if** structure, but some languages provide other conditionals as well.

Mathematica provides the functions If, Which, and Switch for decision making.

- **If Function**

If [*test*, *true-stmts*, *false-stmts*]

- if the *test* is true, the *true-stmts* are executed; if the *test* is false, the *false-stmts* are executed.

Example:

```
x = 4;
If[x>0, y=Sqrt[x], y=0]
```

```
2
```

In Mathematica it is possible that a test can be neither true nor false, since values may not have been assigned to the variables involved in the test. To handle this possibility, we can use the function:

If [*test*, *true-stmts*, *false-stmts*, *neither*]

- if the *test* is true, the *true-stmts* are executed; if the *test* is false, the *false-stmts* are executed; otherwise the neither statements are executed.

- **Which Function**

Which[*test1*, *value1*, *test2*, *value2*, . . .]

- returns the first *valuek* associated with the first true *testk*, as tests are evaluated left to right.

Example:

```
x=-4;
y=Which[x>0,1/x,x<-3,x^2, True, 0]
```

```
16
```

(The third test, “True”, is the default in this example.)

- **Switch Function**

Switch[*expr*, *form1*, *value1*, *form2*, *value2*, . . . __, *defva*l]

- evaluates *expr* and compares it to each *formk* in left-to-right order, and returns corresponding *valuek* when a match is found. The underscore(_) above is an optional default form (matches anything).

Example:

```
a=-4; b=4;
y=Switch[a^2, ab, 1.0/a, b^2, 1.0/b, _, 01]
```

```
0.25
```

(For this example, the evaluation of a^2 matches the evaluation b^2, so the floating-point value of 1/b is returned.)

User-Defined Functions

In addition to the built-in functions provided in the function library, programming languages generally allow users to define other functions useful in particular applications.

We define functions in Mathematica with statements of the form:

fcnName[a*rg1_, arg2_, . . .] := body*

- where body specifies the computations that must be performed to evaluate the function. This can be a single expression or a series of expressions, separated by commas and enclosed in parentheses, involving arithmetic operations, loops, if statements, etc.

The underscore (_) after each argument name allows that argument to be replaced with any other name or expression.

Without the underscore (called “blank”), an argument cannot be replaced with any other name or expression.

Example: `binPower[x_] := (1+x)^2` {single expression

```
y4 = binPower[4]
25

y4yab = y4 binPower[a+b]
25 (1 + a + b)^2
```

It is good practice to clear a variable name before defining it as a function to be sure all previous values and definitions are removed.

For example, functions can be defined with multiple rules as in

```
invSquare[0] = 0;                (immediate assignment)
invSquare[x_] := (1/x)^2         (delayed assignment)
```

With the Clear function, we erase all values (immediate assignments) and definitions (delayed assignments) given to a name. (The operation "=." only removes individual value assignments.)

Example: Redefining binPower as a function of two variables.

```
Clear[binPower];        {could also clear other variables
binPower[x_, n_] := (1+x)^n

y23 = binPower[2,3]
27

ya2Exp = Expand[binPower[a, 2]]
1 + 2 a + a^2
```

Just as with built-in functions, we can get info on user-defined functions by typing ?fcnName. E.g.,

```
?binPower

Global‘binPower

binPower[x_, n_] ;= (1 + x)^n
```

Recursive Functions

- are defined in terms of themselves.

Example: Fibonacci Numbers

(occur in many natural objects; for example, ratio of
successive radii of spirals in seashells and flowers)

```
Clear[fib];
fib[0] = fib[1] = 1;
fib[n_] := fib[n-1] + fib[n-2]
```

(assuming n is nonnegative integer)

```
fourthFib = fib[4]
```

```
4
```

The function **fib** is an example of repeated procedure calls. When we invoke the function with the value 4, it must call itself to compute values for **fib[3]** and **fib[2]**, and so on.

Another common example of a recursive function is factorial (of course, in Mathematica, we can simply write `n!` or use Gamma function which is an extension of factorial to real numbers):

```
Clear[factorial];
factorial[0] = 1;
factorial[n_1] := n factorial[n-1]
```

(assuming n is nonnegative integer)

```
fact5 = factorial[5]
```

```
120
```

Recursion is often a natural way to program a problem. Some languages, particularly older languages, such as FORTRAN 1 through FORTRAN 77, do not allow recursion.

If a function is to be called repeatedly, we can reduce calculation time by saving all previously computed values. This is done in Mathematica by repeating the function name after the SetDelayed Symbol (:=).

Example: Fibonacci Series

```
Clear[fib];
fib[0] = fib[1] = 1;
fib[n_] := fib[n] = fib[n-1] + fib[n-2]
```

(i.e., function fib now defined as an assignment statement.)

Now, any time we call this function, it saves all values it calculates. Then if it is called again with a value it has already computed, it simply does a table lookup.

Tradeoff: Uses more memory.

```
fib4 = fib[4]
5

?fib
Global‘fib
fib[0] = 1
fib[1] = 1          (stored values for fib [n])
fib[2] = 2
fib[3] = 3
fib[4] = 5
fib[n_] := fib[n] = fib[n = 1]
                        + fib[n - 2]
```

To ensure correct input to functions, it is always a good idea to do data checking to be sure that input values are in the proper range (program is “robust”).

Example: Input to Fibonacci Function must be positive.

```
Clear[fib];
fib[n_] :=
  If[n < 0,
    Print[“Input to fib must be
      nonnegative integer.”],
    If[(n==0)||(n==1),
       fib[n] = 1,
       fib[n] = fib[n-1] + fib[n-2]
    ]  (assignment stmts cause values to be saved,
    ]        otherwise just list what is to right of =)
```

This example also illustrates an indentation scheme that can be used to facilitate reading of the code. This is a major feature of good programming practice, and it is especially important as blocks of code get longer.

In this scheme, the expression is on the next line after the function definition line. To identify components of the if blocks, we have the test, *true-stmts*, and f*alse-stmts* each on separate lines. We easily identify the beginning and end of each if block by the alignment of brackets.

For all programming assignments, you must use a consistent indentation scheme. You can use the examples in the lecture notes as a guide, but your scheme does not have to be exactly the same.

We can also modify the **`fib`** function to check for integer input using the Round function (how?).

High-level programming languages provide type declaration statements for variable names. Mathematica specifies type declarations (Integer, Real, Complex, String, List, etc.) by appending the required type to the variable name.

For example, we can restrict **fib** to integer values with the definition:

```
Clear[fib];
fib[n_Integer] :=
  If[n < 0,
    Print[“Input to fib must be
      nonnegative integer.”],
    If[(n==0)||(n==1)
        fib[n] = 1,
        fib[n]=fib[n-1]+fib[n-2]
      ]
    ]
```

This function would now produce the following output:

```
fiveFib = fib[5]
8

minus3Fib = fib[-3]
Input to fib must be nonnegative integer

realNumFib = fib[2.5]
fib[2.5]
```

(Indicates that 2.5 cannot be evaluated by this function,
but give no instructions to user.)

Another method for restricting data range in a function definition is to specify the allowable range using the Condition operator(/;).

We can include the condition for the data range following the type declaration:

```
Clear[fib];
fib[n_Integer /; n >= 0] :=
    If[(n==0)||(n==1),
      fib[n] = 1,
      fib[n]=fib[n-1]+fib[n-2]
    ]
```

Or, we can place the restriction on the data range after the function definition:

```
Clear[fib];
fib[n_Integer :=
    If[(n==0)||(n==1),
      fib[n] = 1,
      fib[n]=fib[n-1]+fib[n-2]
    ]   /; n >= 0
```

If we would like to return a message when an incorrect data type or data value is passed to a function, we can use **`If`** functions to make the necessary checks and return an appropriate message.

When we set up a function definition using a series of expressions, the general form is:

fcnName[*arg1_type1, arg2_type2, . . .*]
:= (*expr1; expr2, . . .*)

- where statements are separated by semicolons.

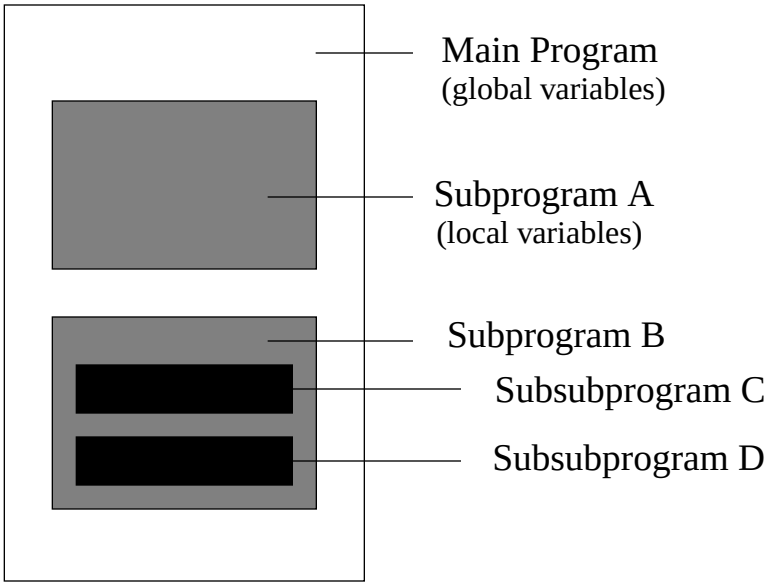
Examples:

```
Clear[fcn1]
fcn1[a_,b_] := (c=a^2; b^3+c)

y = fcn1[2,2]    (variable y is assigned last value calculated
12                  in the body of fcn1, unless the last expression
                    is terminated with a semicolon)

Clear[arithFcn]
arithFcn[a_Real,b_Real] :=
    (sumab = a + b; diffba = b - a;
      prodab = a * b
    )

arithFcn[5.2,-3.1]
-16.12           {result from last expression: a*b

sumab
2.1

diffba
-8.3
```

Procedural Programming

- involves writing blocks of code sequences that contain operations such as assignment statements, loops, conditionals, etc.

Block-Structured Languages allow programs to be “modularized” into a set of independent program units:



Global Variables:

- are those variables declared in outer blocks that can be used by inner subblocks. E.g., variables in Main Prog can be used by Subprograms A, B.

Local Variables:

- are those variables declared in subblocks. They are only known locally; i.e., to the declaring block and its subblocks. For example, variables declared in B are known to B, C, and D, but not to Main or Subprogram A. Local variables override any global variables of the same name.

Subprograms are called “subroutines”, or “functions”, or “procedures”, etc., depending on the particular language and the nature of the subprogram.

For example, a FORTRAN subprogram is either a “subroutine” or a “function”; Pascal uses “procedures” and “functions”; the C language uses only “functions”.

Modules and Local Variables in Mathematica

Module[{*var1*,*var2*, . . .}, body]

- creates a block of code with local variables *var1*, *var2*, etc.,
which exist only during execution of the module.

Example:

```
k=25          (global assignment statement)
25

Module[{k},Do[Print[k,"    ",2^k], {k,3}] ]
1    2
2    4        (here k is local variable with final value 3)
3    8

k             (but the global value of k is still 25)
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, E.g.,

```
Clear[integerPowers];
integerPowers[x_Integer] :=
    Module[{k}, Do[Print[k,"    ",x^k],{k,3}] ]
integerPowers[k]
1   2          {where global variable k still has value 25
2   625
3   15625
```

And local variables can be assigned initial values.

```
Clear[invSum];
invSum[x0_Real /;x0>0, xEnd_Real] :=
    Module[{sum=0.0,x=x0},
          While[(1/x)>xEnd,
                sum=sum+1/x; x++
               ];
          sum
                  ]

y=invum[1.0, 0.15]
2.45

y=invSum[1.0, 0.10]
2.82897

y=invSum[1, 0.1]          {cannot evaluate: x0 is assigned an
invSum[, 0.1]                   integer value, instead of Real

y=invSum[1.0, 3]          {cannot evaluate: xEnd is assigned
invSum[1., 3]                 an integer value, instead of Real

y=invSum[-1.0, 0.1]       {cannot evaluate: x0 is assigned
invSum[-1., 0.1]              a negative value
```

Programs are also easier to understand when they contain documentation, as well as modularization.

Note: comments specified as:

(* comment *)

Example:

```
Clear[invSum];
(* fcn to calculate sum of 1/x for specified
   initial x and final 1/x values *)
invSum[x0_Real /;x0>0, xEnd_Real' :=
  Module[{sum=0.0,x=x0}, (* initialization *)
          While[(1/x)>xEnd,
                  sum=sum+1/x; x++
               ];
          sum
        ]
```

Good Programming Practice:

Modularize programs.
Avoid use of global variables.
Avoid use of goto and similar constructs (spaghetti code).
Include Documentation (Comments)

Advantages - Programs are then:

Easier to Understand
Easier to Modify

Input-Output in Programming Languages

Input commands typically available for

- Interactive Input
- File Input

using names such as “Read”, “Input”, “Get”, etc.,

Output commands direct output data to specific devices and files (screen display, printer or plotter output, disk file output, etc.).

Typical names for output commands in high-level languages are “Write”, “Print”, “Put”, etc.

Associated with input-output commands are “format” specifications, such as

- Total number of digits or characters.
- Representation for decimal numbers (e.g., scientific notation)
- Number of decimal places.
- Spacing between data values.
- Line spacing.
- New page - for output.
- Headings - for output.

etc.

Interactive Input in Mathematica

Input can be requested at any point in a program with either of the functions:

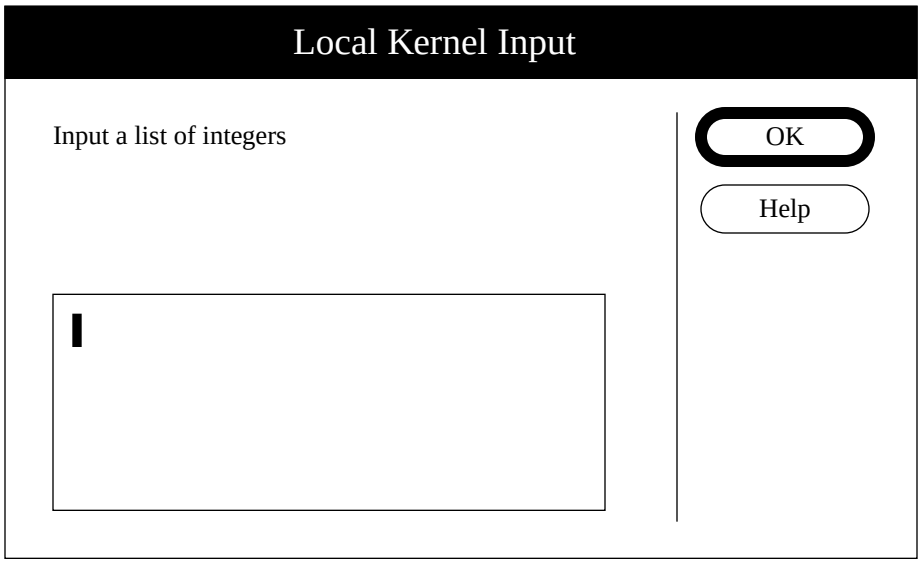
Input[“*prompt*”]	{for any expression or data type
InputString[“*prompt*”]	{string input only

- where the prompt is optional, but should be included
to explain to the user what type of input is needed.

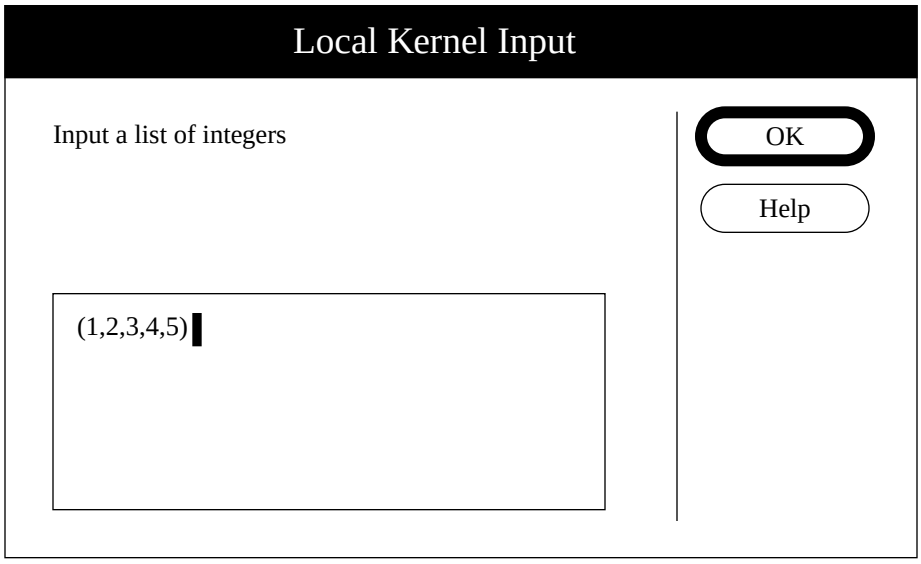
With a notebook interface, an input function usually generates a dialog box, showing the prompt and a blinking cursor.

The following example shows a typical dialog box displayed by an input function.

```
intList=Input[“Input a lislt of integers.”]
```



We then type in the required input at the position of the cursor (remembering that a list must be enclosed in brackets {}):



After typing the input, we click on the OK button (or press “return” or “enter”) to close the dialog box. The output line is then generated following the input stmt:

```
intList=Input[“Input a list of Integers”]
(1, 2, 3, 4, 5)
```

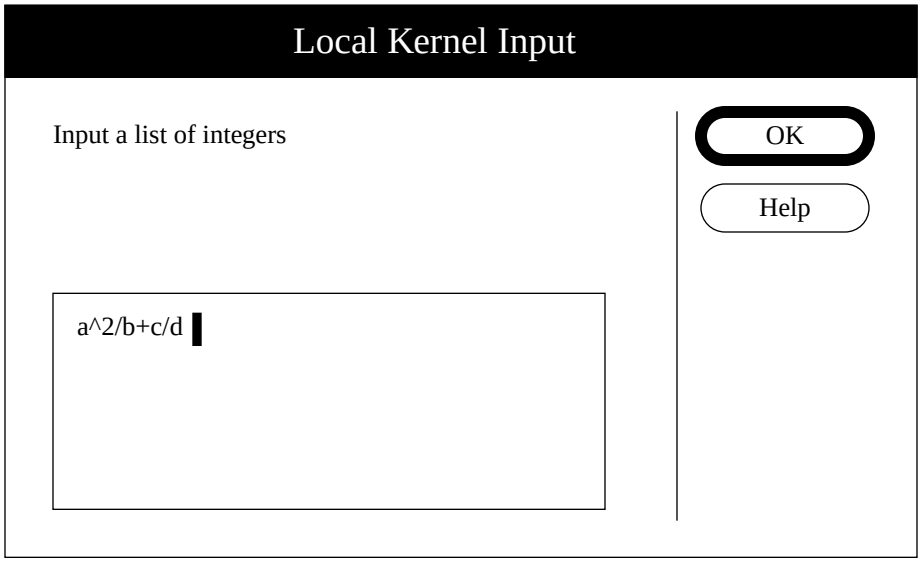
The InputString function is similar; e.g.,

```
str=InputString[“Type in any 5 characters.”]
abcde
```

(If the input is typed as “abcde”, then the output line will include the quotes -- which is a 7-character string.)

Any expression can be entered with the Input function. For example:

```
expr = Input[“Input any expression.”]
```



which produces the Mathematica output line:

$$\frac{a^2}{b} + \frac{c}{d}$$

Similarly for other data types: e.g.,

```
realNo=Input[“Input a floating-pt number.”]
28.957
```

The function InputForm converts an expression into a form suitable for input, editing, or text manipulations. E.g., $\frac{a^2}{b} + \frac{c}{d}$

```
InputForm[%]
a^2/b + c/d
```
(Similarly, OutputForm would return expr to standard 2D math form above.)

```
str=InputString[“Input a 5 char string.”]
abcde

InputForm[%]
“abcde”
```
(If input string is typed as “abcde”, then InputForm produces “\”abcde\”” - where \” represents the character “In the string.)

Conversion functions are also available for changing an expression to a valid form for FORTRAN, C, and T_EX:

```
y = x^2/(a c)
```

$\frac{a^2}{ac}$

```
FortranForm[y]
x**2/(a*c)

CForm[y]
Power(x,2)/(a*c)

TeXForm[y]
(({x^2})\over {a\,c})
```

File I-O in Mathematica

Files are used to store various types of information, such as a program file, data file, or graphics image file.

expr >> filename - write *expr* to file as plain text.

expr >>> filename - append *expr* to file.

<< filename - read in a file, evaluate each expr, display list item in output form.

!!filename - display contents of a file in Mathematical input form.

(Operators >>, >>>, and << are analogous to Unix shell operators >, >>, and <.)

Examples: Storing expressions, definitions, results, etc.

```
Expand[(a+b)^2] >>exprFile    {operator >> erases file if it
expr1 = <<exprFile                 already existed
a^2 + 2 a b + b^2

!!exprFile
a^2 + 2*a*b + b^2

expr2=Factor[<<exprFile] >>>exprFile
!!exprFile
a^2 + 2*a*b + b^2

<<exprFile  {can suppress output by setting last file item
(a + b)^2          to Null
```

- Operator << can be used with a program file to read in and evaluate the statements in the program.
- Operators >> and << can also be written as
  Put[“*filename*”] and *Get*[“*filename*”]

We can display file items in various formats using the following function:

ReadList[“*filename*”, *type*]

- displays a list of the file items in the specified type (if appropriate):

Type	Description
Byte	integer codes for each character in file.
Character	lists the characters in file.
Number	lists numbers in file (exact or approx).
Real	lists the floating-point values of numbers.
Word	lists “words”, separated by spaces, etc.
Record	lists “records”, separated by lines, etc.
String	lists strings in file.
Expression	lists expressions in file.

Examples:

ReadList[“exprFile”, Expression]

$a^2 + 2\,a\,b + b\ ,\ (a + b)^2$

ReadList[“exprFile”, String]

```
{a^2 + 2*a*b + b^2, (a + b)^2}
```

(same as Record, for this file)

ReadList[“exprFile”, Word]

```
{a^2, +, 2*a*b, +, b^2, (a, +, b)^2}
```

```
ReadList["exprFile", Character]
(a, ^, 2,  , +,  , 2, *, a, *, b,
  +,  , b, ^, 2, , (, a, , +,  , b, '),
  ^, 2, )

ReadList["exprFile", Byte]
(97, 94, 50, 32, 43, 32, 50, 42, 97,
  42, 98, 32, 43, 32, 98, 94, 50, 13,
  40, 97, 32, 43, 32, 98, 41, 94, 50,
  13)
```

```
"A three-word sentence." >>charFile
ReadList["charFile",Word]
{"A, three-word, sentence."}
```

```
Do[k^2 >>>sqsFile, {k,5}
!!sqsFile
1
4
9
16
25
```

(Note: To produce a numeric data file, the file must be created with Table or other functions that generate a list

(The file could be created with another program; i.e., word processor)

```
<<sqsFile
25

ReadList["sqsFile", Number]
{1, 4, 9, 16, 25}

ReadList["sqsFile", Real]
{1., 4., 9., 16., 25.}
```

We can also save function definitions and variable assignments with

Save[“fi*lename*”, f*cn*1, *fcn2*, . . . *var*1, *var*2, . . .]

Examples:

```
f[x] := (a+x)^3

a = 8
8

Save[“defsFile”, f]

!!defsFile
f[x_] := (a + x)^3
a = 8
```

(Both the function and the value 8 for parameter a are saved -- provided we make the assignment before saving f.)

```
y = m x + b

Save[“defsFile”, y]

!!defsFile
f[x_] := (a + x)^3
a = 8
y = b + m*x
```

Single items can be read from a file with the following sequence of operations:

OpenRead[“*filename*”] - opens a file in “read” mode.

Read[“*filename*”, *type*] - read an item from file.

Skip[“*filename*”, *type*, *n*] - skip n items of specified type.

Close[“*filename*”] - close file.

When file is opened, a file pointer is set to beginning of file. As each item of file is read or skipped, pointer advances to next file item. After last item is read, Read function returns “EndOfFile”.

Example: Reading file items into a list.

Consider the file of squares:

```
ReadList[“sqsFile”, Number]

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

The following code segment transfers the file items one-by-one to sqsList, using the Append function:

```
OpenRead[“sqsFile”];
num=Read[“sqsFile”, Number];
sqsList={}; (* initialize to empty list *)
While[num != EndOfFile”,
      sqsList=Append[sqsList, num];
      num=Read[“sqsFile”, Number]
    ];
Close[“sqsFile”];
sqsList

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

The following code section transfers every other item of the sqsFile to the list sqsList:

```
OpenRead[“sqsFile”];
num=Read[“sqsFile”, Number];
sqsList={}; (* initialize to empty list *)
While[num != “EndOfFile”,
      sqsList=Append[sqsList, num];
      Skip[“sqsFile”, Number];
      num=Read[“sqsFile”, Number]
    ];
Close[“sqsFile”];
sqsList

{1, 9, 25}
```

Similar functions are available for writing to files. They are low-level operations corresponding to >> and >>>.

OpenWrite[“*filename*”] - opens (and erases) a file in “write” mode.

OpenAppend[“f*ilename”*] - opens a file for appending.

Write[“*filename”*, *expr*] - write an expr to a file.

Close[“*filenam*e”] - close file.

Additional file functions:

DeleteFile[“*filename”*] - delete the file.

Copy[“*file* 1”, “*file* 2”] - copyes items in file 1 to file 2.

Directory[] - gives current working directory.

SetDiretory[“*dirname”*] - set current working dir.

FileNames[] - lists files in current working dir.

File Names

Within the current working directory, we can reference a file directly with its *filename.*

When the file is in a directory below the current working directory, we can reference the file through the hierarchy of file names:

*dirname*1 ‘*dirname*2 ‘ . . . ‘*filename*

where *dirname*2 is a directory (folder)
within *dirname*1, etc.

Thus, to reference a file that is not in the current working directory, we can either change the current directory or use the hierarchical filename above.

Number Representations

- **Decimal Number System** (Base 10)

  Digits: 0, 1, 2, . . . , 9

  e.g.,

  $$352.7 = 3 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 + 7 \times 10^{-1}$$

- **Binary Number system** (Base 2)

Digits: 0, 1

$$101.11 = 1 + 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$
$$= 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5.75_{10}$$

- **Octal Number System** (Base 8)

  Digits: 0, 1, 2, . . . 7

  Useful for representing binary information. Starting from right, replace each group of three binary digits with equivalent octal representation. E.g.,

  $$101110010111_2 = 5627_8 \qquad (= 2967_{10})$$

- **Hexadecimal Number System** (Base 16)

  Digits: 0, 1, 2, . . . , 9, A, B, C, D, E, F

  Can use upper or lower case letters. Useful for representing groups of four binary digits. Eg.,

  $$101110010111_2 = B97_{16}$$

Number Representations in Mathematics

Using the appropriate digits, n, we can represent a number in any base b as

b^^nnnnnn

Examples:

2^^101 (= 5 in base 10)
8^^101 (= 65 in base 10)
16^^101 (= 257 in base 10)

16^^1F(= 31 in base 10)

Mathematica provides the following function for converting a given number to its equivalent in base b:

BaseForm[number, b]

Examples:

```
BaseForm[400, 2]
```
{when no base is specified, the number is assumed to be a base 10 representation

110010000_2

```
BaseForm[65, 8]
```
101_8

```
BaseForm[16^^101, 10]
```
257