

2. Expressions and Evaluation



2.1 Introduction

2.2 Expressions

- Objectives
- What is an expression?
- But not everything looks like an expression!
- Ways of looking at expressions
- Finding the **Length** of an expression
- Exercises

2.3 Evaluation

- Objectives
- Mathematica evaluates expressions
- Holding evaluation
- Exercises

2.4 Assignment

- Objectives
- Using **Set**
- Using **SetDelayed**
- Exercises

2.5 Lists and Tables

- Objectives
- Lists
- Listability
- Algebraic and calculus operations on lists
- Constructing lists with **Table**
- Efficiency in list manipulation
- Exercises

2.6 Problems

- Problem 1: Exploring the validity of expressions
- Problem 2: Exploring the evaluation sequence with **Unique**
- Problem 3: Exploring the difference between **Set** and **SetDelayed**
- Problem 4: Exploring expressions with **FullForm**
- Problem 5: Generating data with **Table**

2.1 Introduction

The aim of this module is to understand, and gain a facility in using, three concepts basic to writing high level programs. These are *expressions*, *evaluation*, and *assignment*. You will also understand how to use *lists* and *tables*, particularly for handling data.

2.2 Expressions

Objectives

To understand the concept of a Mathematica *expression*.

To understand that *any* Mathematica program is simply an expression.

To understand the difference between the *displayed* form and the *internal* form of an expression.

To be able to obtain the internal form of any expression.

What is an expression?

A Mathematica *expression* is any string of symbols of the form

$$\square[\square, \square, \square, \dots]$$

where the \square are placeholders in which we can write either pure symbols, *or other expressions*. That is, expressions can be *nested*.

This is the internal form of *everything* in Mathematica. Its simplicity is what makes programming in Mathematica extremely powerful.

$$\text{Everything is an } \textit{expression}$$

◆ Examples

The following are examples of expressions because they conform to the form above

$$\mathbf{\text{Sin}[x]}$$

$$\mathbf{\text{f}[x, y, z]}$$

$$\mathbf{\text{f}[x, \text{Sin}[x], \text{Sin}[\text{Sin}[x]]]}$$

For example, the skeleton of the last expression is

$$\square[\square, \square[\square], \square[\square[\square]]]$$

■ Terminology

The things inside the square brackets are called the *elements* of the expression. The thing at the front of the square brackets is called the *head* of the expression.

For example in $\mathbf{f[x, y, z]}$, \mathbf{f} is the head and $\mathbf{x, y, z}$, are the elements.

■ Atoms

Pure symbols like $\mathbf{f, x, y, z, Sin}$, from which the most basic expressions are built are called *atoms*. They are expressions which have no definition in terms of other expressions.

There are three types of atoms, *numbers*, *symbols*, and *strings*.

An *atomic number* is either integer or real.

A *symbol* is any sequence of letters and integers *not starting with an integer*.

A *string* is any sequence of characters between double quotes.

For example $\mathbf{2, 3.14159, x, Sin, x33, "Hello"}$ are atoms, but $\mathbf{-2, \frac{2}{3}, 33x}$ are not atoms.

Note that $\mathbf{x33}$ will be interpreted by Mathematica as the symbol with this name, but $\mathbf{33x}$ will be interpreted as $\mathbf{33}$ times \mathbf{x} .

Mathematica is *case-sensitive*. So the upper and lower case forms of a letter are considered to be *different* symbols.

But not everything looks like an expression!

The *internal* representation of everything as expressions makes programming in Mathematica extremely powerful as we shall see later. However, it also makes it very difficult to read. It is easier for us to read expressions in mathematical form.

So the designers of Mathematica have arranged for it to *display* expressions in mathematical form wherever possible. We can also *enter* the expressions in this form.

■ FullForm

The Mathematica command **FullForm** will take any expression in its displayed form, and generate the full internal form.

This is an important command to understand as it shows you how all programs (and all of mathematics!) can be written as nested expressions.

You will often need to find the **FullForm** of a displayed expression so that you can pull it apart and manipulate it (that is, program with it).

For example, we would normally write the sum of two numbers \mathbf{a} and \mathbf{b} as $\mathbf{a+b}$. To find out how Mathematica represents this internally as an expression we enter **FullForm[a+b]**.

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

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

This shows us that the internal representation of $a+b$ as an expression is **Plus[a,b]**. When you program, visualizing $a+b$ as really **Plus[a,b]** enables you to bring to bear a whole range of programming tools which have been designed to operate on *any* expression.

◆ Examples

If something does not appear to be in the form of an expression, you can use **FullForm** to find out what it is. Here are some simple examples:

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

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

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

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

DISPLAY	FULLFORM
$a + b + c$	Plus[a, b, c]
$a b c$	Times[a, b, c]
$a - b$	Plus[a, Times[-1, b]]
a^n or a^n	Power[a, n]
a/b or $\frac{a}{b}$	Times[a, Power[b, -1]]
$\{a, b, c\}$	List[a, b, c]
$a \rightarrow b$ or $a \rightarrow b$	Rule[a, b]
$a == b$ or $a == b$	Equal[a, b]
$a >= b$ or $a \geq b$	GreaterEqual[a, b]

(The alternatives in the first column are alternative keyboard entry methods)

■ Special symbols

The convention for expressing special symbols not directly available from the keyboard is to enclose their full name in $\backslash[]$. For example the **FullForm** of α is $\backslash[\text{Alpha}]$. These are still atoms.

```
FullForm[{ $\alpha$ ,  $\mathbb{D}$ ,  $\mathbb{B}$ ,  $\square$ }]
```

```
List[\[Alpha], \[DoubleStruckCapitalD],  
      \[GothicCapitalB], \[Placeholder]]
```

Ways of looking at expressions

In programming, and in mathematics, there are various types of objects that we want to use, and various types of operations that we want to apply to those objects. Mathematica treats all of these objects and operations alike as expressions.

For example, the classic mathematical concept of *function* $f(x, y, \dots)$ is clearly able to be written as the expression $f[x, y, \dots]$.

In programming a command, for example **Expand** to expand an expression $(x + y)^3$, can also be written as an expression.

```
Expand[(x + y)3]  
x3 + 3 x2 y + 3 x y2 + y3
```

Sometimes you may want to define a type of *object*, say **Engine**, which has as contents certain information about the engine, for example its bore and stroke. You could write this as an expression

```
Engine[Bore, Stroke, ...]
```

INTERPRETATION	EXAMPLES
Function [arguments]	Sin [x], f [x, y]
Command [parameters]	Expand [(x + y) ³]
Operator [operands]	Plus [x, y], Times [x, y]
Object [contents]	List [a, b, c]

◆ Example

In Mathematica, a typical inbuilt object type is the list. Lists have the form $\{\square, \square, \square, \dots\}$. They are essential in programming for collecting together into one object a set of objects of any type, particularly data. The list $\{a, b, c\}$ is represented internally as **List**[a, b, c].

Mathematica also uses lists to represent vectors and matrices. *Vectors* are represented directly by lists. A *matrix* is represented by a list of lists. For example $\{\{1, 0, -1\}, \{-1, 2, 2\}, \{3, 0, -2\}\}$ represents a matrix. We can *display* it in the form we are used to by using **MatrixForm**.

```
MatrixForm[\{\{1, 0, -1\}, \{-1, 2, 2\}, \{3, 0, -2\}\}]
```

$$\begin{pmatrix} 1 & 0 & -1 \\ -1 & 2 & 2 \\ 3 & 0 & -2 \end{pmatrix}$$

Finding the Length of an expression

The *length* of the expression is simply equal to the number of elements or arguments. For example the length of $f[x, y, z]$ is 3.

We compute the length of an expression by using the function **Length**:

```
Length[f[x, y, z]]  
3
```

Note that because there are still only 3 arguments in an expression like `f[{x1, y1}, {x2, y2}, {x3, y3}]`, its length is also 3.

```
Length[f[{x1, y1}, {x2, y2}, {x3, y3}]]
3
```

Exercises

◆ Exercise: Finding the FullForm of an expression

Find the **FullForm** of the elements of the following list.

```
{a -  $\frac{2b}{c}$ , {{{x, y}}}, a[b[c[d, e], f], g], x == Sin[x]}
```

◆ Exercise: Determining the validity of an expression

Divide the elements in the following list into two lists **A**: those which are valid expressions, and **B**: those which are not valid expressions.

```
{1, x, y[x, y, z], = 2, zed[],
{x, y}, →, 3.14159, a + i b, [], Sin[ $\frac{\pi}{2}$ ], 0 = x,
Tan[θ], a == b, (), Log[4, 3], □, j2a, 2 aj, {},
x0, x :=>, 4 > 3, □□, √π, 2[x], c[c[c[c[c]]]],
Log[2,], {, }, c[, c], Sin[, 2], {a, b}^{c,d}};
```

Hint: Try entering each element separately into Mathematica. If it returns a result without complaint, it is a valid expression. If it complains, perhaps saying that more input is needed, it is not a valid expression.

2.3 Evaluation

Objectives

To understand the way in which Mathematica deals with the input you give it.

To understand how to prevent evaluation from occurring so that you can see the internal form of an expression.

Mathematica evaluates expressions

Mathematica has just *one* operation: *evaluation of expressions*.

Mathematica runs a program (expression) that you give it, simply by looking at the set of rules it has (both its inbuilt rules, and those that you might have given it) and applying any that are applicable.

◆ Example

Mathematica has a lot of inbuilt rules for simplifying expressions. For example, whenever *Mathematica* encounters **Sin**[π] in an expression, it replaces it by **0**, because it has an inbuilt rule that tells it to do so.

```
□[□, □[□], □[Sin[ $\pi$ ]]]
□[□, □[□], □[0]]
```

Programming in *Mathematica* is nothing more than adding to *Mathematica*'s set of rules. And of course, these rules are themselves just expressions!

■ The order of evaluation

Mathematica's evaluation cycle evaluates heads and arguments from left to right. For example the contents of the boxes in □[□, □[□], □[□[□]]] would be evaluated in the order **1[2, 3[4], 5[6[7]]]**.

Once it has done this it will go back to see if the result of its evaluations have generated anything new which it recognizes it has a rewrite rule for. If it does, it repeats the process. If it does not it terminates and returns its result.

Holding evaluation

If we want to see the **FullForm** of an expression that will be evaluated to something different when we enter it into Mathematica, we have to put a *hold* on the evaluation. We can do this with **HoldForm**. For example if we ask for **FullForm**[**2+2**], Mathematica will evaluate **2+2** to get **4**, then give us **FullForm**[**4**] (which is just **4**). By wrapping **HoldForm** around an expression that we do not want evaluated, we can see its full internal form.

```
FullForm[2 + 2]
4
HoldForm[FullForm[2 + 2]]
Plus[2, 2]
```

You can release the hold on the evaluation by applying **ReleaseHold**.

```
ReleaseHold[%]
```

```
4
```

Use `HoldForm[FullForm[...]]` to see the internal representation of a function.

◆ **Example**

```
FullForm[Integrate[1/x, {x, 1, 2}]]
```

```
Log[2]
```

```
HoldForm[FullForm[Integrate[1/x, {x, 1, 2}]]]
```

```
Integrate[Times[1, Power[x, -1]], List[x, 1, 2]]
```

```
ReleaseHold[%]
```

```
Log[2]
```

Exercises

◆ **Exercise: Using HoldForm to find the FullForm of an expression**

Find the **FullForm** of each of the expressions in the following list.

```
{x == Sin[x], Integrate[(x^3 + x) dx, Solve[x^3 + x + 1 == 0, x],  
Plot[Sin[x], {x, 0, 100}], N[Pi, 1000], Table[n!, {n, 0, 100}]}
```

◆ **Exercise: Exploring subscripts and superscripts**

Determine the displayed (output form) of the symbols in the list below.

```
{Subscript[x, i], Subscript[x, _], Subscript[x, i, j],  
OverBar[x], OverDot[x], SuperDagger[A], Underscript[B, 7]}
```

◆

2.4 Assignment

Objectives

- To understand the concept of assignment in programming.
- To understand the difference between (immediate) assignment and delayed assignment.
- To be able to use assignment and delayed assignment operations correctly.

Using Set

Assignment is simply a way of giving an expression a name, so that when you call the name, you get the expression. (If you assign the name Fido to your dog, then when you call Fido, you actually get your dog ...). More precisely, the name acts as an "alias" for the expression, so that whatever you do with the name, you effectively do with the expression, because as soon as you enter the name into *Mathematica* it replaces it with the expression.

Assignment is probably the most important programming operation. It is represented in almost all computing languages by the mathematical symbol =. If you have an expression, say $\{\{x_1, y_1\}, \{x_2, y_2\}\}$, and you want to assign the name (symbol) **P** to it, you write

$$\mathbf{P} = \{\{x_1, y_1\}, \{x_2, y_2\}\}$$

$$\{\{x_1, y_1\}, \{x_2, y_2\}\}$$

On entering $\mathbf{P} = \{\{x_1, y_1\}, \{x_2, y_2\}\}$, we see that *Mathematica* returns the expression (the right hand side), because this is the *value* of **P**.

If now, we enter **P**, *Mathematica* will evaluate it, and return its value $\{\{x_1, y_1\}, \{x_2, y_2\}\}$.

$$\mathbf{P}$$

$$\{\{x_1, y_1\}, \{x_2, y_2\}\}$$

Be very careful here. The programming usage for the = sign is much more specialized than the many different ways it is used in mathematics.

An assignment, for example, $\mathbf{x} = \mathbf{y}$, does not look like an expression. To see what the internal representation of $\mathbf{x} = \mathbf{y}$ is, we can ask for the **FullForm**. But because the evaluation is immediate we need also to use **HoldForm**.

$$\mathbf{HoldForm}[\mathbf{FullForm}[\mathbf{x} = \mathbf{y}]]$$

$$\mathbf{Set}[\mathbf{x}, \mathbf{y}]$$

Hence the command to *set* (assign) the value of **x** to be equal to **2** is written **Set[x, 2]**.

◆ Example

We can generate a random number between 0 and 1 with the function **Random[]** (note there are no arguments). Let us assign a random number to be the value of **x**:

```
x = Random[ ]  
0.126236
```

Now, whenever we use **x** we get this number.

```
{x, x, x}  
{0.126236, 0.126236, 0.126236}
```

■ Unsetting an assignment

If you want to *unset* an assignment for **x** you use **Unset[x]**, or in shorthand: **x = .**. For example if we enter **x = 2**, next time we enter **x**, we get **2**. Now enter **x = .**, then next time we enter **x** we just get **x** back, because we have removed the assignment.

△ Keep your desk clean with □ = .

For otherwise, later on in your programming, you may find strange things happening, because you had forgotten that a symbol you are using now had been assigned a value before.

Using SetDelayed

In an assignment using **= (Set)** the right hand side is evaluated *once only* at the time *when the assignment is entered*.

However, there is an alternative type of assignment in which the right hand side is evaluated and the assignment made *only whenever the left hand side is asked for*.

This type of assignment is called *delayed assignment* and is entered using **:=** (that is, a colon, followed immediately by an equals). To see the Mathematica internal form of this we again use **FullForm**.

```
HoldForm[FullForm[x := y]]  
SetDelayed[x, y]
```

Although there may not immediately appear to be that much difference between **Set** and **SetDelayed** assignments, they can be crucial. Imagine you were a stock-broker and you got an order from a client to evaluate the value of a portfolio of shares and then sell immediately, assigning the value to a certain account. The client may get a very different result if you do not action the order at the time, but wait until the client wishes to retrieve the money out of the account before doing so. The value of the portfolio may well have changed dramatically in the meantime!

A simple way of showing the difference between **Set** and **SetDelayed** is to use the **Random[]** function again. We enter

```
x := Random[]
```

Note that there is no output, because Mathematica has simply stored the expression for use at a later time, that is, *next* time you ask for the value of **x**.

Now, if we ask for the value of **x** three times, we will get three *different* answers, since the right hand side will be evaluated afresh *each* time the value of **x** is asked for. That is, a new random number will be generated each time.

```
{x, x, x}  
{0.839136, 0.0140678, 0.917866}
```

Exercises

◆ Exercise: Exploring assignment and delayed assignment

Define **e1 = Expand[(x + 1)³]** and **e2 := Expand[(x + 1)³]**.
Then define **x = (a+b)**. Now enter the symbols **e1** and **e2**.
Understand why the two results *look* different.

◆

2.5 Lists and Tables

Objectives

To understand the concept of a list.
To understand the concept of listability.
To be able to generate and manipulate lists efficiently.

Lists

As we have already seen a list of elements or data **a, b, c, ...** is represented in Mathematica by **{a, b, c, ...}** whose **FullForm** is **List[a, b, c, ...]**.

You can have lists of lists (for example, data pairs, point coordinates, matrices), or lists of lists of lists ..., to any level of nesting.

Listability

Most of the inbuilt functions, when given a list as an argument, will map the function over the elements of the list. This property of the function is called *listability*. For example, entering **Log[{a,b,c}]** returns **{Log[a],Log[b],Log[c]}**. This mapping is made at all levels of the list. For example

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

Algebraic and calculus operations on lists

Algebraic and calculus operations propagate through lists. Here are some examples

INPUT	OUTPUT
$\{a, b, c\} + \{x, y, z\}$	$\{a + x, b + y, c + z\}$
$\{a, b, c\} \{x, y, z\}$	$\{a x, b y, c z\}$
$\frac{\{a,b,c\}}{\{x,y,z\}}$	$\{\frac{a}{x}, \frac{b}{y}, \frac{c}{z}\}$
$\{a, b, c\}^3$	$\{a^3, b^3, c^3\}$
$\{a, b, c\}^{\{x,y,z\}}$	$\{a^x, b^y, c^z\}$
$2 \{a, b, c\} + 1$	$\{1 + 2 a, 1 + 2 b, 1 + 2 c\}$
$D[\{x, x^2, x^3\}, x]$	$\{1, 2 x, 3 x^2\}$
$\text{Integrate}[\{x, x^2, x^3\}, x]$	$\{\frac{x^2}{2}, \frac{x^3}{3}, \frac{x^4}{4}\}$

Constructing lists with Table

The **Table** function is used for *constructing* lists. You can see all the things you can do with **Table** by entering **?Table**. Here is a table of 50 random integers between 0 and 100:

```
Table[Random[Integer, {0, 100}], {50}]
{99, 55, 85, 23, 4, 14, 73, 81, 76, 75, 18,
 12, 7, 28, 11, 59, 54, 46, 57, 21, 38, 98, 37, 84,
 22, 99, 29, 40, 32, 40, 33, 21, 99, 43, 45, 43, 34,
 82, 32, 87, 12, 15, 38, 28, 13, 7, 75, 87, 60, 56}
```

Here is a table of expressions:

```
Table[i Sin[j x] + k, {i, 1, 2}, {j, 2, 3}, {k, 1, 3}]
{{{1 + Sin[2 x], 2 + Sin[2 x], 3 + Sin[2 x]},
 {1 + Sin[3 x], 2 + Sin[3 x], 3 + Sin[3 x]}},
 {{1 + 2 Sin[2 x], 2 + 2 Sin[2 x], 3 + 2 Sin[2 x]},
 {1 + 2 Sin[3 x], 2 + 2 Sin[3 x], 3 + 2 Sin[3 x]}}}
```

Efficiency in list manipulation

It is much more computationally efficient to treat lists of elements *as a whole*, and manipulate them as a whole, rather than take them apart and reconstruct new ones. We will see ways of doing this later, but for the meantime we note the important principle for efficient programming, which John Gray calls *the fundamental dictum of Mathematica programming*:

Treat mathematical structures as wholes.
Never tear them apart and rebuild them again.

Exercises

◆ Exercise: Generating lists

Generate a list of integers from 1 to 20 using the **Range** function, and a list of powers of **x** from 1 to 20 using the **Table** function. Then construct a third list consisting of the products of corresponding elements in the first two lists.

◆ Exercise: Exploring listability

Enter $\mathbf{x} = \{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$. Construct the lists given by the elements of the list below:

$$\{2 \mathbf{x}, \mathbf{x}^2, \mathbf{x}^2 + 2 \mathbf{x} + 1, \frac{\mathbf{x}}{\mathbf{x}}, \mathbf{x}^{\mathbf{x}}, \mathbf{Sin}[\mathbf{x}]^2 + \mathbf{Cos}[\mathbf{x}]^2\}$$

◆ Exercise: Exploring list processing functions

Teach yourself how to use other list processing functions like **Length**, **Flatten**, **Reverse**, **Transpose**, **First**, **Rest**, **Last**, and **Take**.

2.6 Problems

Problem 1: Exploring the validity of expressions

Explore which of the elements of the list below are valid expressions.

$$\{\mathbf{f}[], \mathbf{f}[\mathbf{g}][\mathbf{h}], [\mathbf{x}], [], \mathbf{f}[\mathbf{g}][\mathbf{h}][\mathbf{i}]\}$$

◆

Problem 2: Exploring the evaluation sequence with `Unique`

`Unique[x]` creates a new symbol name each time it is used in the same session. Use this to test the evaluation sequence of an expression of the following form:

$$\{\square, \square[\square, \square], \square[\square, \square[\square[\square[\square]]]]\}$$

Find out about `Unique` by entering `?Unique`.

Hint: Write the expression with `Unique[x]` in place of each placeholder. Or, more compactly, set `u := Unique[x]`, and write the expression with `u` in place of each placeholder.

Observe the evaluation sequence.



Problem 3: Exploring the difference between `set` and `setDelayed`

Define `e1 = Flatten[{x, y}]` and `e2 := Flatten[{x, y}]`. Then define `x = {a, b}`. Now enter the symbols `e1` and `e2`. Understand why the two results are different.



Problem 4: Exploring expressions with `FullForm`

Even the essential internal definition of a graph is an expression. Plot the graph of `Sin[x]` from `0` to `8π`, and assign this plot to the symbol `p`, say.

Now find the `FullForm` of `p`, and note that it is made up mostly of points (the plot points) and rules (the plot options). We will cover rules in the next module.



Problem 5: Generating data with `Table`

You are a sound engineer who is developing a program to process sound data, and you want to generate some (artificial) test data consisting of a list of pairs, the first of which is a frequency in Hz represented by an integer ranging from 100 to 2000 in steps of 100, while the second is the corresponding sound pressure level in Pa, represented by a random number between 0 and 0.01. Use the `Table` function to generate this list.

