Applied Mathematical Economics with *Mathematica*

Author: Alan G. Isaac
Institution: American University

Initialization

Front Matter

Preface

This booklet is intended to support the use of *Mathematica* in courses on Mathematical Economics. No prior programming experience is assumed; no prior *Mathematica* experience is assumed. Built-in *Mathematica* commands are introduced more-or-less on an as-needed basis, rather than systematically. Some commands are treated in detail, but I make no effort to reproduce the excellent and extensive *Mathematica* documentation. Instead, I introduce readers to the *Mathematica* help system, and I occasionally point to online resources.

The mathematics presented herein is not self-contained. This is not a textbook in Mathematical Economics. Rather, I use *Mathematica* to provide illustrations of some mathematical tools typically covered in a first course in Mathematical Economics. Core goals include the following:

- introduce the basic use of *Mathematica* notebooks for computation, plotting, and symbolic algebra
- encourage readers to improve their mathematical intuition with concrete applications and graphical explorations
- familarize readers with the computational application of basic mathematical tools to economic problems

This booklet approaches these goals by providing economics-relevant applications of the following:

- linear algebra
- univariate and multivariate calculus
- linear and nonlinear comparative statics
- univariate and mulitvariate optimization

Use of *Mathematica* provides substantial benefits to teachers and students in introductory Mathematical Economics. Courses introducing mathematical tools to economists often cover a wide range of topics at a speed that can intimidate students. Concrete applications of the mathematical concepts help students learn, but these often require tedious calculations that consume classroom time as well as the mental energy of students. The use of *Mathematica* delegates some of this tedium to the software and

greatly increases the speed with which examples can be developed and problems solved. This allows greater emphasis on core mathematical practices.

This booklet was written using *Mathematica* version 10 for Windows. Users on other operating systems should have no difficulties as long as they have a recent version of *Mathematica*. However, I use a few commands that did not exist in earlier versions, and sometimes of the algorithms associated with a command improve in later versions. Therefore readers are encouraged to work the examples with version 10 or later.

Some sections or passages are marked as “advanced”. This material can be skipped on a first reading by those new to *Mathematica*, but those with some *Mathematica* exposure should find them accessible even on a first reading.

Mathematica and the Notebook: A Brief Introduction

This chapter provides a basic introduction to the use of *Mathematica* notebooks to enter text and to perform basic numerical and symbolic computations.

Notebooks and the Front End

Mathematica comes with an interactive graphical user interface (GUI) called the *front end*. We use the front end to perform interactive computations and to keep a record of these computations as notebooks. To create a new notebook, pick from the *front end* menus File > New > Notebook.

Evaluating Numerical Expressions

Cells

A *Mathematica* notebook is a collection of cells. After starting the *Mathematica* front end and creating a new notebook, just start typing to create a notebook cell. You will see that the cell is delimited by a bracket on its right side. Press the down arrow to leave this cell. A horizontal line will appear, indicating an insertion point for a new cell. Type some more and a new cell is created.

Cell Styles

Click on a cell’s right edge to make it active, and then use the Format > Style menu to give it a style. The default cell style is Input, which is for *Mathematica* expressions that you wish to evaluate, but you can change the style at any time by using the Format menu. We will most often be creating one of the following Cell types: Input, DisplayFormula, and Text. We will also use Title, Section, and Subsection.

There are also keyboard shortcuts for the Format menu. (See https://reference.wolfram.com/language/tutorial/KeyboardShortcutListing.html#1484044289).

Text cells can contain ordinary text. Give a cell a Text style when you want to provide discussion or

descriptions of your calculations. The *Mathematica* front end will do some basic formatting of this text, in a manner similar to how a word processor handles text.

In contrast, use the default Input style if your cell contains *Mathematica* expressions that you want to evaluate.

If you want to create a new cell with the same style your are currently using, press Alt+Enter.

Evaluating Input Cells

After entering any *Mathematica* expression into an Input cell, press `Shift+Enter` to evaluate it. The result is placed in its own cell, which has the Output style. For example, consider an arbitrary arithmetic expression.

```
105 / (15 * 7)
```

```
1
```

This example shows that you can easily use *Mathematica* as a calculator. Note the familiar use of parentheses to determine the order of evaluation in the Input expression. (Q: What is the result if you remove them?)

We often want to explain an expression. We can do this with comments, which open with an opening parenthesis and asterisk and close with an asterisk and closing parenthesis, `(* like this *)`.

```
355 / 113.  (* crude approximation to π *)
```

```
3.14159
```

You may place more than one expression in a single Input cell. If you want to compute the value of an expression but not display it, end the expression with a semicolon. If you want to refer to the last computed value, use a percent sign. As an example of both, let us do a computation on multiple lines but only display the final result.

```
2 * 3;
% + 5
```

```
11
```

Mathematica allows you to add styles to your results. However, keep in mind that a formatted object does not behave the same as the underlying object. Once a number is styled, for example, you will not be able simply to do arithmetic with the result.

```
Style[2 * 3, Red]
% + 5 (* addition will not be performed *)
```

```
6
```

```
5 + 6
```

Numbers

Mathematica distinguishes between exact numbers (roughly, integers and rationals) and approximate numbers (roughly, floating point numbers). Computations done with exact numbers return exact results.

```
5 * (3 / 7)
```

$$\frac{15}{7}$$

We can use the `N` command to turn an exact number into an approximate number.

```
N[%]
```

```
2.14286
```

Mathematica names a few special numbers. (Like Mathematica commands, these names all start with a capital letter.)

```
{Pi, E, I, Infinity, -Infinity}
```

```
{π, e, i, ∞, -∞}
```

Use `N` to turn these into approximate numbers (when possible):

```
N[%]
```

```
{3.14159, 2.71828, 0. + 1. i, ∞, -∞}
```

Numerical Comparison with Relational Operators

Mathematica has the usual collection of comparison operators: `<`, `≤`, `==`, `≠`, `≥`, `>`. The value of a numerical comparison is "boolean" (i.e., either `True` or `False`). E.g.,

```
1 < 1
```

```
False
```

Expressions are evaluated before they are compared, and numbers of different types compare in a natural fashion.

```
1 == 1.0
1 == 3 / 3
```

```
True
```

```
True
```

We can use the `And` and `Or` commands to determine whether all or any comparisions are `True`.

```
And[1 < 1, 1 ≤ 1, 1 == 1, 1 ≠ 1, 1 ≥ 1, 1 > 1]
Or[1 < 1, 1 ≤ 1, 1 == 1, 1 ≠ 1, 1 ≥ 1, 1 > 1]
```

```
False
```

```
True
```

Each comparison operator is shorthand for a *Mathematica* command. For example, the infixed double equal sign `==` is shorthand for `Equal`.

Relational operators can be chained:

```
1 == 1 + 0 == 0 + 1
3 > 2 > 1
```

```
True
```

```
True
```

Advanced: Mathematica numerical equality are “clever”, relying on numerical approximation to test inequality. Thus for example the following comparision evaluates to `True` in Mathematica, whereas (due to rounding error) it would not in most languages.

```
.3 == .1 + .2
```

```
True
```

Advanced: If you really want to test for sameness (after evaluation) and not just numerical equality, use three equals signs: ===. For *Mathematica*, exact numbers are not the same thing as approximate numbers.

```
1 === 3 / 3
.3 === .1 + .2
1 === 1.0
```

```
True
```

```
True
```

```
False
```

Stopping an Evaluation

Do not be afraid to evaluate expressions in *Mathematica*. If you make a mistake that results in an invalid expression, *Mathematica* will tell you about your mistake. If you evaluate an expression that is taking too much time, you can pick Evaluation > Abort Evaluation from the Mathematic menus. (You can alternatively enter Alt+. to abort an evaluation, i.e., bring it to a full stop.) If you want to evaluate an expression that you fear may take a very long time, you can limit the amount of time *Mathematica* will work on it with the TimeConstrained command.

It is rare but possible to enter an expression that causes *Mathematica* to lock up, so be sure to save often. It is possible to ask *Mathematica* to autosave your work, but since half-written code may be useless or worse, you may find it more useful to save often by hand (ctrl+s).

Full Form

Each *Mathematica* expression has an equivalent “full form”, which is what the front end actually sends to the kernel for evaluation. For example, our expression `105/(15*7)` is really a convenient shorthand for **Times[105,Power[Times[15,7],-1]]**. We will seldom have reason to consider this full form, but at times it is useful for debugging. You can use the `Hold` and `FullForm` commands to discover it. (The `Hold` command prevents evaluation of the expression, and `FullForm` returns its full form.)

```
HoldForm[FullForm[105 / (15 * 7)]]
```

```
Times[105, Power[Times[15, 7], -1]]
```

```
HoldForm[FullForm[{1 < 1, 1 ≤ 1, 1 == 1, 1 ≥ 1, 1 > 1}]]
```

```
List[Less[1, 1], LessEqual[1, 1], Equal[1, 1], GreaterEqual[1, 1], Greater[1, 1]]
```

Symbols

Rules for the Creation of Symbols

We will often want to give names to *Mathematica* objects, using user-defined symbols. You can use almost any combination of letters to define a symbol. By convention, *Mathematica*'s system-defined symbols capitalize the first letter, so in order to avoid name clashes you should not capitalize the first letter of your new symbols. A symbol cannot start with a digit, but you can use numeric characters anywhere else in your symbols.

Sometimes we would like to include a non-alphanumeric character in order to visually break a symbol into parts. The standard choice in many languages is the underscore, but that has a special meaning in *Mathematica*. (It is used to produce patterns.) The only choice from a standard American keyboard is the back tick. (This actually creates a symbol with a "context", as we will discuss shortly.) Be sure not to end your symbol with a dollar sign nor with a dollar sign followed by digits, as *Mathematica* uses these forms for special purposes. (For details, see the documentation entitled How Modules Work.) Also note that superscripts and subscripts do not become part of a symbol name.

Manipulating Symbols

You can manipulate symbols without associating them with any values.

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

$$\left\{\left\{x \to \frac{-b-\sqrt{b^2-4\,a\,c}}{2\,a}\right\},\ \left\{x \to \frac{-b+\sqrt{b^2-4\,a\,c}}{2\,a}\right\}\right\}$$

Note how we first used `Clear` to clear any definitions that might have been assigned to these symbols.

Setting and Unsetting the Values of Variables

When we introduce symbols as variable names, and we often wish to assign values to them. Use a single equal sign (the assignment operator) to define a symbol to be equivalent to the value of a defining expression.

```
Clear[a, b, c]
a = b + c
```

```
b + c
```

The equals sign is actually shorthand for the `Set` command, so the following is equivalent. (But we usually use the equals sign.)

```
Clear[a, b, c]
Set[a, b + c]
```

```
b + c
```

Notice that an assignment expression has a value, produced by the evaluation of the right hand side. Assignment returns this value, and it is displayed as output. You can suppress that output by putting a semi-colon at the end of the line.

```
a = b + c;
```

After such an assignment, the defined symbol will simply be replaced in subsequent expressions by its definition. (We have associated a “rewrite rule” with the symbol `a`, which will be applied whenever *Mathematica* encounters `a`.)

```
a * a
```

```
(b + c)^2
```

An assignment persists until you make a new assignment or `Unset` the symbol. In the latter case, the symbol still exists but no longer refers to its previous definition. Mathematica provides the shorthand `=.` for `Unset`.

```
Clear[a, b, c]
a = b + c;  (* Set the value of `a` *)
a * a
a =.   (* Unset `a`, which now has no value *)
a * a
```

```
(b + c)^2
```

```
a^2
```

Advanced: Once you create a symbol, it persists. When we `Unset` a symbol, we do not remove it from Mathematica’s list of recognized symbols. We can see this list by using the `Names` command. If this is for some reason unacceptable, we can `Remove` the symbol.

```
a00 = 5;
Unset[a00] (* `a00` is still in the list of names: *)
MemberQ[Names["Global`*"], "a00"]
Remove[a00](* `a00` is no longer in the list of names: *)
MemberQ[Names["Global`*"], "a00"]
```

```
True
```

```
False
```

Multiple Assignment and Clear

You can make many assignments at once by putting comma-separated variable names and assigned values in braces. Since the right hand side of an assignment is evaluated before the assignment takes

place, you can use multiple assignment to swap variable values.

```
{a01, a02} = {1, 2};     (* set both values *)
{a01, a02}               (* show both values *)
{a01, a02} = {a02, a01}; (* swap values *)
{a01, a02}
```

```
{1, 2}
```

```
{2, 1}
```

You can `Unset` many symbols by providing them as arguments to `Clear`.

```
Clear[a01, a02]
{a01, a02}
```

```
{a01, a02}
```

Advanced: Use ClearAll instead of Clear if you want to clear Attributes and Options associated with a symbol. You can also Remove a symbol, but this has subtle repercussions. (It will affect all your previous definitions using that symbol.)

Advanced: You can use the OwnValues command to examine the assignment to a variable. We see that assignment has created a rule associated with that variable. For example, if we have made an assignment to `a` but not to `b` we find a rule in `OwnValues[a]` but none in `OwnValues[b]`.

```
Clear[a, b, c]
a = b + c
OwnValues[a]
OwnValues[b]
```

```
b + c
```

```
{HoldPattern[a] :> b + c}
```

```
{}
```

Delayed Assignment

We have seen that assignment (`Set`) immediately evaluates the right hand side and assigns this value to the left hand side. The evaluation of the right-hand side is done once.

```
Clear[a, b];
a = 0; b = a; a = 1; b
```

```
0
```

Sometimes we want different behavior: the right hand side is not evaluated until the left hand side symbol is used. *Mathematica* allows us to achieve this with the `SetDelayed` command, or the equivalent := operator. The evaluation of the right hand side is done anew each time the left-hand side symbol appears.

```
a = 0; b := a; a = 1; b
```

```
1
```

Here is another way to see the different between assignment and delayed assignment.

```
x1 = RandomInteger[100]
{x1, x1, x1, x1} (* the same each time *)
x2 := RandomInteger[100]
{x2, x2, x2, x2} (* can differ each time *)
```

```
44
```

```
{44, 44, 44, 44}
```

```
{9, 40, 90, 80}
```

Advanced: Notice the difference in the `OwnValues` of `b` in the following cases. Related to this, in contrast to `Set`, the `SetDelayed` command returns a `Null` value.

```
ClearAll[a, b1, b2, b3]
{a = 2, b1 = a};
OwnValues[b1]
{a := 2, b2 = a};
OwnValues[b2]
{a := 2, b3 := a};
{OwnValues[b1], OwnValues[b2], OwnValues[b3]}
{b1, b2, b3}
a = 3
{b1, b2, b3}
```

```
{HoldPattern[b1] :↦ 2}
```

```
{HoldPattern[b2] :↦ 2}
```

```
{{HoldPattern[b1] :↦ 2}, {HoldPattern[b2] :↦ 2}, {HoldPattern[b3] :↦ a}}
```

```
{2, 2, 2}
```

```
3
```

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

Equality vs. Assignment

Note that *Mathematica* has several different uses of the equals sign.

```
x1 = 2 (* assignment *)
x2 := Random[] (* delayed assignment *)
1 == 1.0 (* equality testing *)
1 === 1.0 (* identity testing *)
```

```
2
```

```
True
```

```
False
```

Variable Scope

By default *Mathematica* symbols have global scope: they are available anywhere in your notebook. For users accustomed to traditional programming languages, this has some surprising implications, which trace to our ability to manipulate undefined symbols in *Mathematica* expressions. For example, without defining *x* or *y* we can write utility as

```
Clear[x, y]
U = x^0.5 y^0.5
```

$$x^{0.5}\, y^{0.5}$$

Mathematica uses this expression to define *U* but *x* and *y* remain undefined. Moreover, we can then subsequenctly assign values to these varaibles, and it affects the values of *U*.

```
x = 20; y = 30;
U
```

```
24.4949
```

If we change the values of *x* or *y*, it changes the value of *U*.

```
x = 50;
U
```

```
38.7298
```

If we clear the values of *x* and *y*, then we return to a definition of *U* in terms of undefined variables.

```
Clear[x, y]
U
```

$$x^{0.5}\, y^{0.5}$$

The default global scope is often convenient, but it does mean that if you re-evaluate an expression earlier in a notebook after you change (later in the notebook) any variable in the expression, this will affect the evaluation of that expression. That is, the order of occurence in a notebook need not signal the order in which expressions have been evaluated. (You can pick Evaluate Notebook from the Evaluation menu if you want to evaluate all the evaluatable cells in order.) For this and other reasons, you may want to ensure that some variables are local to an expression. You can accomplish this with the Module command. (For more details, see the *Mathematica* documentation entitled How Modules Work.)

```
t = 5;
Module[{t = 0}, t += 1; Print[t]]
Print[t]
```

```
1
```

```
5
```

Comparison

We can test equality between symbolic expressions:

```
Clear[a, b]
a = b;  (* assignment *)
a == b   (* equality test *)
```

```
True
```

Symbolic equations may not automatically be simplified. In this case the original equation is returned.

```
Sin[θ]^2 + Cos[θ]^2 == 1
```

```
Cos[θ]^2 + Sin[θ]^2 == 1
```

We may be able to force simplification with the `Simplify` command (or the more costly `FullSimplify` if necessary).

```
Simplify[%]
```

```
True
```

If we restrict their values, we may even be able to simplify to a boolean value size comparisons between symbols that have not been assigned numerical values. Note how the relational operators get reused to impose relations, not only to test for them.

```
Simplify[x < y, x < 3 && y > 4]
```

```
True
```

Warning: Expressions that appear identical may not be identical, if they have side effects. For example, the `Increment` command increases the value of a symbol by 1 (and returns the old value). (There is a puzzle as to how a command can directly change the value of its argument, which we will take up later when discussing the `HoldAll` attribute.) We can append `++` to a symbol as a shorthand for `Increment`.

```
a = 1;
a++ == a++
a
```

```
False
```

```
3
```

Help

Mathematica's basic help facility is its `Information` command, which provides help for any symbol. The help begins with a basic "usage message" and the provides some details about the symbol, including any options.

```
a = b + c;
Information[a]
```

Info-aeb39697-4ef6-44ec-96c5-bedbc86d17f4

```
Global`a
```

Info-aeb39697-4ef6-44ec-96c5-bedbc86d17f4

```
a = b + c
```

Here we learn that `a` is a global symbol that has been set equal `b+c`.

We often want information about Mathematica commands. To make this much shorter to type this as an "input escape", prepending two question marks to the symbol.

```
?? Information
```

Info-3cd93d23-7d8e-44c4-9bc3-f96744387842

Information[*symbol*] prints information about a symbol. ≫

Info-3cd93d23-7d8e-44c4-9bc3-f96744387842

```
Attributes[Information] = {HoldAll, Protected, ReadProtected}

Options[Information] = {LongForm → True}
```

We often only want the usage message, which is available as an option by specifying LongForm->False. To specify that we do not want the long-form information, we can alternatively use a single question mark instead of two.

```
? Information
```

Info-40bca079-4b37-4ce1-9f77-cc24162448b4

Information[*symbol*] prints information about a symbol. ≫

Additional Observations

Warning: As you start working in your Notebooks, remember that symbol definitions are global. Computations are done by the *Mathematica* kernel. If you open multiple notebooks during a single *Mathematica* session, the notebooks will share a single kernel by default. (You can change this.) This means that all your open notebooks are sharing all your global definitions!

Aside: to completely remove all your global symbols, it is safest just to use the Quit[] command. This quits the kernel and then starts a new *Mathematica* session.

Advanced: The front end and the kernel talk to each other via *MathLink*, a communications protocol. Programmers can use *MathLink* to communicate with *Mathematica* from their programs or other applications.

Contexts

Every *Mathematica* symbol has a context, which is part of its full name. When you create a new symbol

at a notebook prompt, it is ordinarily part of the Global` context. You can determine the context of a symbol with the `Context` command.

```
Context[a]
```

```
Global`
```

IMPORTANT: by default, symbols in the global context are available to all your open notebooks! You can set a Notebook's Default Context via the Evaluation menu.

A context name always ends with a backtick, called a context mark. You can create a new context at any time simply by including a new context name when defining a symbol.

```
Context[my`a]
Context[yr`a]
```

```
my`
```

```
yr`
```

```
a + my`a + yr`a
```

```
my`a + yr`a + b + c
```

For more information, see the *Mathematica* documentation entitled Contexts.

Entering Math

Typing Math in Text Cells

Often we wish to include inline math in a Text cell. We do this by creating an inline-math cell, usually by pressing Ctrl+9. A lightly colored box appears, and you can begin typing your math. When you are done typing your math, press ctrl-0 to exit. On a standard American keyboard, the numbers 9 and 0 are associated with opening and closing parentheses, which provides a nice mnemonic for math entry. Notice that *Mathematica* applies different formatting to the inline-math cell than it does to ordinary text. Here is an example: $y = f(x)$.

In order to type mathematics, you will also want to learn how to enter special symbols and formats. One simple approach is to pick from the menus Palettes > Basic Math Assistant, which gives point and click access to a useful collection of symbols and typesetting formats. For faster entry, you will want to learn keyboard shortcuts. *Mathematica* allows use of the escape (esc) key to delimit keyboard shortcuts (called aliases) to special symbols. You can also use full names for the symbols, surrounded by brackets, and preceded by a backslash. So we can enter an α either as [esc]a[esc] or as \[Alpha], and we can enter $\int$ as either [esc]int[esc] or as \[Integral]. Finally, we often want to type superscripts or subscripts: rather than rely on the Basic Math Assistant, it is good to learn the keyboard shortcuts: Ctrl+6 (or alternatively, Ctrl+^ on an American keyboard) and Ctrl+- (or equivalently, Ctrl+_ on an American keyboard) bring up the superscript and subscript templates. Now you can type expressions such as $y = \alpha_1 x^2$ in your Text cells.

In the Basic Math Assistant, if you hover your mouse over a special format, *Mathematica* will display the keyboard shortcut for that format.

Some online resources:
http://reference.wolfram.com/mathematica/guide/GreekLetters.html
http://reference.wolfram.com/mathematica/guide/SpecialCharacters.html
http://reference.wolfram.com/mathematica/tutorial/KeyboardShortcutListing.html

HoldForm

When we want *Mathematica* to evaluate a mathematical expression, we type that expression into a cell that has the default `Input` style. When that cell is active, we can evaluate the expression by pressing `Shift+Enter`. The result of the evaulation will display in a new cell, which is given an `Output` style. Here is an example from the Wolfram documentation.

```
HoldForm[Integrate[x^2 E^-x^2, x]] == Integrate[x^2 E^-x^2, x]
```

$$\int x^2\, e^{-x^2}\, dx == -\frac{1}{2}\, e^{-x^2}\, x + \frac{1}{4}\sqrt{\pi}\ \text{Erf}[x]$$

Note the use of HoldForm, which prevents the evaluation of the expression on the left. As a result we can display both the expression we want to evaluate and its evaluation. This is the kind of thing we might want to include in our text. *Mathematica* distinguishes display math from inline math. Display math is placed in a separate cell with style DisplayFormula or DisplayFormulaNumbered. This is where you will usually type stand-alone equations and expressions involving two-dimensional structures, such as matrices. So let us copy it into a new cell, which we give the DisplayFormula style.

$$\int x^2\, e^{-x^2}\, dx == -\frac{1}{2}\, e^{-x^2}\, x + \frac{1}{4}\sqrt{\pi}\ \text{Erf}[x]$$

Matrices in a Display Formula

Usually you will want to type a matrix formula directly into a cell having the DisplayFormula style. Here is one way. Begin by creating a DisplayFormula cell and entering your matrix equation.

```
{{a11, a12}, {a21, a22}} {{x1}, {x2}} == {{b1}, {b2}}
```

Next, press Ctrl+Shift+t to change it to traditional display:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

Display formula are not automatically center-aligned. To change the alignment, pick from the menu Format > Text Alignment > Align Center.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

Advanced: Entering Math as LaTeX

If you are familiar with LaTeX, you may at time find it more convenient to enter math expressions using LaTeX notation. *Mathematica* allows you to do this with ToExpression[“inputstring”,TeXForm], where any backslashes in the input string must be doubled.

```
ToExpression["\\sqrt{2.0}", TeXForm]
```

```
1.41421
```

Since LaTeX notation is not always unambiguous, *Mathematica* can be very touchy about how you use it. For example, ToExpression[“\\int_0^3 2*x dx”,TeXForm] will fail. But if we force a space, *Mathematica* will recognize (and evaluate) the expression.

```
ToExpression["\\int_0^3 2*x \, dx", TeXForm]
```

```
9
```

If you want to see the LaTeX that *Mathematica* associcates with an expression, use TeXForm.

```
TeXForm[Integrate[f[x], {x, 0, 3}]]
```

```
\int_0^3 f(x) \, dx
```

Drawing with Graphics Objects

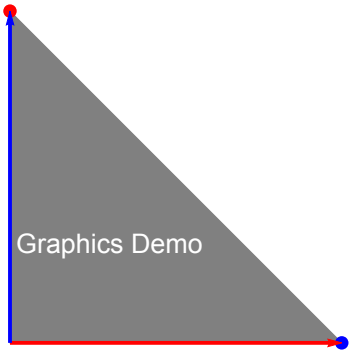
Interactive Drawing

Mathematica allows for interactive drawing in your notebooks. From the *Mathematica* menus pick Insert>Picture>NewGraphic, or just press Ctrl+1, to create a new empty graphic. Then press Ctrl+D to bring up the Drawing Tools palette. At the top of the palette you can find a variety of objects to place in your drawing. You can set the properties (e.g., color and line thickness) before drawing the object, or you can left-click the object after drawing it and change its properties. (For more details, see the *Mathematica* documentation entitled Graphics Interactivity & Drawing and the documentation entitled Drawing Tools.)

However, for ease of modification and ready replication, it is a good habit to draw with code rather than with the interactive tools.

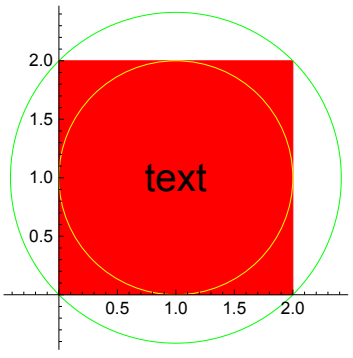
Example: Polygon, Point, Arrow, and Text

```
p1 = {0, 0}; p2 = {1, 0}; p3 = {0, 1};
g1 = Graphics[
   {
    {Gray, Polygon[{p1, p2, p3}]},
    {PointSize[Large], Red, Point[p3]},
    {PointSize[Large], Blue, Point[p2]},
    {Thick, Red, Arrow[{p1, p2}]},
    {Thick, Blue, Arrow[{p1, p3}]},
    {White, Text[Style["Graphics Demo", 14], {0.3, 0.3}]}
   },
   ImageSize → Small]
```



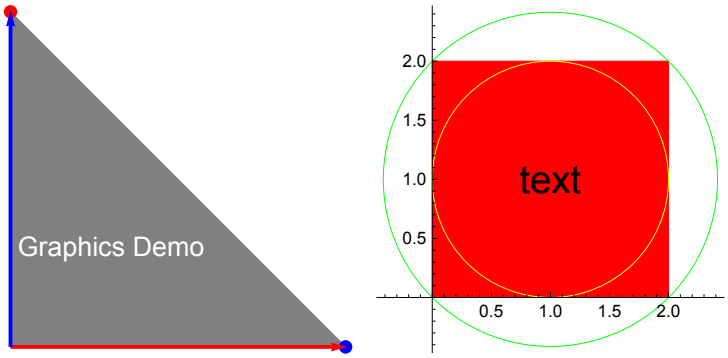
Example: Rectangle, Circle, Axes

```
g2 = Graphics[{
    {Red, Rectangle[{0, 0}, {2, 2}]},
    {Yellow, Circle[{1, 1}, 1]},
    {Green, Circle[{1, 1}, Sqrt[2]]},
    Text[Style["text", 20], {1, 1}]
   }, Axes → True, ImageSize → Small]
```



Example: Combining Graphics Objects

```
g3 = GraphicsRow[{g1, g2}]
Export["c:/temp/temp.pdf", g3]  (* WARNING: this will overwrite temp.pdf *)
```




```
c:/temp/temp.pdf
```

Advanced: Programmatic Drawing

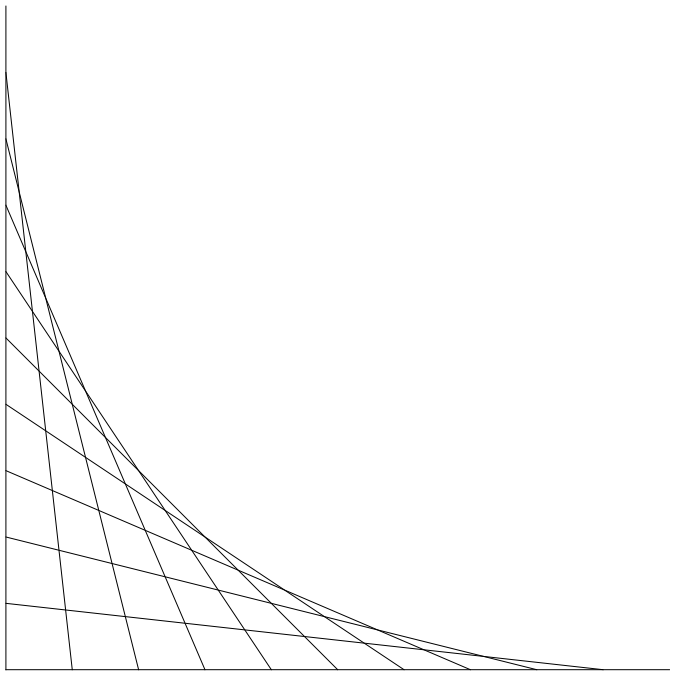
We can also generate graphics programmatically, which is often more useful. (It is more precise and more replicable.) For two-dimensional graphics, we create a Graphics object from a list of graphics primitives such as Point, Line, Arrow, and Text. In this context, point is produced from a list of two coordinates, a line or arrow is produced from a list of two or more points.

As an example, let us use `Array` to create a list of pairs of pairs of numbers. Each pair of numbers represents a point. Each pair of pairs therefore represents a line. The `Line` command can accept this array as a description of a collection of lines, and the result can be drawn by the `Graphics` command.

```
Array[{{0, 10 - #}, {#, 0}} &, 11, 0]
Line[%]
Graphics[%]
```

```
{{{0, 10}, {0, 0}}, {{0, 9}, {1, 0}}, {{0, 8}, {2, 0}},
 {{0, 7}, {3, 0}}, {{0, 6}, {4, 0}}, {{0, 5}, {5, 0}}, {{0, 4}, {6, 0}},
 {{0, 3}, {7, 0}}, {{0, 2}, {8, 0}}, {{0, 1}, {9, 0}}, {{0, 0}, {10, 0}}}
```

```
Line[{{{0, 10}, {0, 0}}, {{0, 9}, {1, 0}}, {{0, 8}, {2, 0}},
  {{0, 7}, {3, 0}}, {{0, 6}, {4, 0}}, {{0, 5}, {5, 0}}, {{0, 4}, {6, 0}},
  {{0, 3}, {7, 0}}, {{0, 2}, {8, 0}}, {{0, 1}, {9, 0}}, {{0, 0}, {10, 0}}}]
```



See http://reference.wolfram.com/mathematica/guide/GraphicsObjects.html for more graphics objects. See http://reference.wolfram.com/mathematica/tutorial/GraphicsDirectivesAndOptions.html for graphics directives and options.

Sharing Notebooks

Notebook files are saved with a .nb extension. These are readily shared with anyone who has *Mathematica*. However *Mathematica* allows you to share both static and dynamic content with others.

The most useful format for sharing static content is probably the portable document format (PDF). First, save your notebook. Then, pick from the *Mathematica* menus File>SaveAs>Save as Type> PDF Document. (Alternatively, you can Export the EvaluationNotebook.)

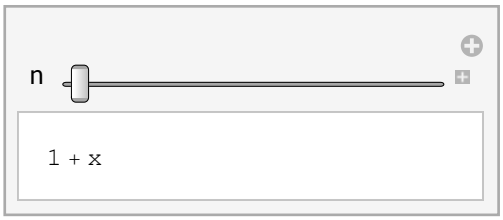
For sharing dynamic content with people who do not have *Mathematica*, in version 8 and above you can save your notebook in the computable document format (CDF). These documents can be opened in the free Wolfram CDF Player. The static content of an ordinary notebook is viewable in the CDF player, but

the CDF format additionally produces Manipulate objects that are fully interactive in the player. (Also supported are Dynamic and DynamicModule.)

Manipulate Example

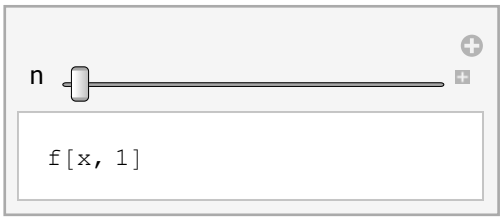
Here is a simple example of the use of Manipulate.

```
Clear[x]
Manipulate[Expand[(1 + x)^n], {n, 1, 5, 1}]
```



If your Manipulate object relies on defined symbols, you will need to use the SaveDefinitions option to ensure interactivity in the CDF player. Here is an example that uses a function definition. (We give a fuller discussion of function defintion in a later section.)

```
ClearAll[f, x, n]
f[x_, n_] := (1 + x)^n
Manipulate[Expand[f[x, n]], {n, 1, 5, 1}, SaveDefinitions → True]
```



Exercise

Create a new notebook and then do the following.
(a) From the menus, pick Format > Style Title and then enter the title “My First Notebook”.
(b) Create a Section cell and enter the subtitle “Text and Expressions”.
(c) Create a Text cell, enter your name, and right justify the cell contents. (Look under the Format>Alignment menu.)
(d) Create a Section cell and enter "Just Text".
(e) Create a Text cell and enter the text “My nicely formatted text cell.”.
(e) Create a Section cell and enter the section heading “Just Computation”.
(f) Create an Input cell (the default style), enter an expression, and evaluate it.

Some Useful Resources

Remember, the *Mathematica* documentation is generally excellent, and it is available both in the Note-

book and online. Nevertheless, here are a few supplementary resources.

- Input Syntax: http://reference.wolfram.com/mathematica/tutorial/InputSyntax.html
- Herbert Halpern's introduction: https://math.uc.edu/~halpern/Mathematicafall09/Ho/Mathematicabook24.pdf
- Wolfram's *Mathematica* tutorials: https://www.wolfram.com/learningcenter/tutorialcollection/
- Pitfalls for *Mathematica* beginners: http://mathematica.stackexchange.com/questions/18393/what-are-the-most-common-pitfalls-awaiting-new-users
- Leonid Shifrin's *Mathematica* Programming: http://www.mathprogramming-intro.org/

Functions: Some Basic Considerations

Mathematically, a function maps elements of one set to elements of another set. This is a very general concept. Computationally, functions are usually much more restricted in scope. For example, a function will often specify a particular transformation of an "input argument" into a new value that the function returns.

Pure Functions

Suppose that we want a function to return x^2 for any input x. Mathematica allows us to create this in a very natural way with the `Function` command: first we give names to the inputs. (The names are called parameters or formal parameters; the input values are called arguments or actual parameters.) Then we use these names to specify the transformation of input(s) into output(s).

```
f1 = Function[x, x^2]
```

```
Function[x, x^2]
```

Note that the formal parameters are names that are local to the function definition. You do not need to worry about any global variable having the "same" name.

Our function can now be applied to different arguments, numerical or symbolic.

```
f1[2]
f1[s]
```

```
4
```

```
s^2
```

The basic considerations are that simple. If we want a function of more than one variable, we can provide a list of parameter names.

```
f2 = Function[{x, y}, x^2 + y^2]
f2[2, 3]
```

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

```
13
```

We bound the names `f1` and `f2` to our functions so that we could use them later. But these functions

are usable even if we do not name them. Here is a simple example:

```
Function[x, x^2][2]
```

```
4
```

Anonymous functions are useful whenever a function will only be used once, so that naming it is a needless expense. (We will see a number of examples of this later on.)

Alternative Notation for Pure Functions

Mathematica provides a convenient shorthand equivalent using a familiar mathematical notation. (You can produce the “mapsto” arrow as `[Esc]fn[Esc]`.)

```
f3 = x ↦ x^2
f3[2]
f3[s]
```

```
Function[x, x^2]
```

```
4
```

```
s^2
```

In our function definitions up to now, we have given names to the function parameters. These names are local to the function definition, so this is usually harmless and serves as an aid to reading. But we are not required to name our arguments. Instead we can refer to them by their slot numbers. This allows us to skip naming the formal parameters and instead refer to them by their default labels: #1, #2, etc. (Also, # is equivalent to #1.)

Here is a function pure function with one formal parameter and another with two formal parameters.

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

```
#1^2 &
```

```
#1^2 + #2^2 &
```

Notice that *Mathematica* represents these function definition in an alternative notation, using a postfixed ampersand. We are also free to use this notation directly. It is a bit more compressed but perhaps a little less readable. It is a very common Mathematica usage. However, especially if you will share your code, it is worth asking whether the gain from the compressed syntax offsets the cost of reduced readability.

Local Variables

Often we want to declare local variables inside our function definition. In *Mathematica* we do this with the `Module` command, which takes a list of variable names as its first argument, which are then treated as local in the expression that is its second argument. The second argument can be a compound expression, where subexpressions are separated by semicolons. The last expression evaluated is the value of the module.

```
Function[x, Module[{s, c}, s = Sin[x]; c = Cos[x]; s^2 + c^2]]
%[Pi / 2]
```

```
Function[x, Module[{s, c}, s = Sin[x]; c = Cos[x]; s^2 + c^2]]
```

```
1
```

`Module` also allows the local variables to be initialized when declared.

```
Function[x, Module[{s = Sin[x], c = Cos[x]}, s^2 + c^2]]
%[Pi / 4]
```

```
Function[x, Module[{s = Sin[x], c = Cos[x]}, s^2 + c^2]]
```

```
1
```

Advanced: Argument Passing

The default behavior of Mathematica is that function arguments are passed as the value of the expression. If you forget about this, you are likely to run into the following error message:

Set::setps {...} in the part assignment is not a symbol

Here is a simple way to produce this error:

```
{0}[[1]] = 1;
```

```
Set::setps : {0} in the part assignment is not a symbol. >>
```

However the following does not produced this error:

```
x = {0}; x[[1]] = 1; x
```

```
{1}
```

The difference is that `Set` changes the value associated with a symbol (in this case, `x`). You need a symbol to associate with the value.

Unfortunately, the place where you are most likely to encounter this error is slightly more obscure: if you pass a symbol to a function, it will be evaluated before the function body is exected. (There are ways to work around this, e.g. using `HoldFirst`, which we do not discuss.)

```
Clear[x, s]
s = {0}
Function[x, x[[1]] = 1]
%[s]  (* problem: s will be evaluated before function is executed *)
```

```
{0}
```

```
Function[x, x[[1]] = 1]
```

```
Set::setps : {0} in the part assignment is not a symbol. >>
```

```
1
```

Advanced: Closures

Functions can return functions, and the returned functions can close over the local variables of the creating function. (A closure can “recall” the environment it was created in.)

```
ClearAll[plusn]
plusn = n ↦ (x ↦ x + n)
plus2 = plusn[2]
plus2[5]
```

```
Function[n, Function[x, x + n]]
```

```
Function[x$, x$ + 2]
```

```
7
```

We see that `plus2` “recalls” that, when it was created, *n* had the value 2.

Note *Mathematica's* special use of the dollar sign for internally defined variable names. Do not use the dollar sign in your own variable names.

Functions via Pattern Matching and Delayed Evaluation

A popular approach to function definition is through a delayed-evaluation assignment. The function name and underscore-tagged formal parameters appear to the left, then the delayed evaluation assignment operator `:=`, and finally the function definition on the right. Note that because we use delayed evaluation, this definition simply defines a symbol and does not generate any output.

```
ClearAll[xsq03]
xsq03[x_] := x^2
xsq03[33]
```

```
1089
```

We can see this difference when we look at the Head or FullForm of the different expressions.

```
Map[Head, {xsq01, xsq02, xsq03}]
Map[FullForm, {xsq01, xsq02, xsq03}]
```

```
{Symbol, Symbol, Symbol}
```

```
{xsq01, xsq02, xsq03}
```

Nevertheless, as we have seen, all three approaches give us a usable “function”.

There are subtle advantages to the different forms. A pure function can be readily added to an expression that needs a function “on the fly”. However, the delayed evaluation syntax allows easy inclusion of default values for a function paramater, given after a colon. (Pattern matching can also be used for type checking, but we do not cover that here.)

```
xsq04[x_: 2] := x * x
xsq04[]
```

```
4
```

Advanced: More Closures

We can also return closures from functions defined by delayed evaluation.

```
ClearAll[plusn]
plusn[n_] := (x ↦ x + n)
plus2 = plusn[2] (* produces a closure *)
plus2[5]
```

```
Function[x$, x$ + 2]
```

```
7
```

We see that `plus2` “recalls” that, when it was created, *n* had the value 2.

Note again *Mathematica's* special use of the dollar sign for internally defined variable names.

Lists

The list is a fundamental *Mathematica* data type. We use it to represent sets, vectors, and matrices.

Quick Introduction

A list is just an ordered collection of items.

Enumeration

We can create a list by enumerating comma-separated elements within curly braces.

```
lst01 = {1, 2, 3}
lst02 = {4, 5, 6}
```

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

```
{4, 5, 6}
```

The numbers are the elements of the list. We can join lists to make a bigger list:

```
Join[lst01, lst02]
```

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

A list can be empty:

```
lst = {}
```

```
{}
```

A list can contain anything as its elements, including other lists. For example, we represent a matrix as a list of lists.

```
mA = {lst01, lst02}
```

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

We can use the `MatrixForm` command to get a more intuitive visual display of a matrix.

```
MatrixForm[mA]
```

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

Note: `{a,b,c}` is just shorthand for `List[a,b,c]`. We can use `FullForm` to see this:

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

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

List Arithmetic

Arithmetic operations on lists are elementwise. So is exponentiation.

```
lst01 + lst02
lst01 - lst02
lst01 * lst02
lst01 / lst02
lst01 ^ lst02
```

```
{5, 7, 9}
```

```
{-3, -3, -3}
```

```
{4, 10, 18}
```

$$\left\{\frac{1}{4}, \frac{2}{5}, \frac{1}{2}\right\}$$

```
{1, 32, 729}
```

Scalar operations are also possible.

```
lst01 + 2
lst01 - 2
lst01 * 2
lst01 / 2
lst01 ^ 2
```

```
{3, 4, 5}
```

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

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

$$\left\{\frac{1}{2}, 1, \frac{3}{2}\right\}$$

```
{1, 4, 9}
```

```
2 + lst02
2 - lst02
2 * lst02
2 / lst02
2 ^ lst02
```

```
{6, 7, 8}
```

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

```
{8, 10, 12}
```

$$\left\{\frac{1}{2}, \frac{2}{5}, \frac{1}{3}\right\}$$

```
{16, 32, 64}
```

Indexing

Using double brackets, we can access the elements of a list with a unit-based index. To index backwards from the end of the list, use negative indexes.

```
lst01 = {1, 2, 3}
lst02 = {4, 5, 6}
lst01[[1]]
lst01[[-1]]
```

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

```
{4, 5, 6}
```

```
1
```

```
3
```

Lists can contain lists. We can index the outer list to get the inner lists. We can then index the inner lists to get their elements. There is also a shorthand for this nested indexing: separate the levels at which you are indexing by commas.

```
lstlst = {lst01, lst02}
lstlst[[1]][[2]]
lstlst[[1, 2]]
```

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

```
2
```

```
2
```

Note: `lst[[1]]` is just a shorthand for `Part[lst,1]`. For more details about indexing, see the documentation for `Part`.

Digression on Summing and Accumulating (Fold and FoldList)

Use `Total` to sum up the elements of a list.

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

```
a + b + c
```

This is equivalent to using the list elements as arguments to `Plus`.

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

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

We are going to digress a bit and explore this observation in some detail. A *Mathematica* list has a "head" of list, which is available as part 0 of the list.

```
Head[{a, b, c}]
{a, b, c}[[0]]
```

```
List
```

```
List
```

Mathematica allows us to replace the head, using the `Apply` command (equivalent to the `@@` operator). The Apply command simply replaces the "head" of an expression with a specified function.

```
Clear[f, a, b]
Apply[f, {a, b}]
```

```
f[a, b]
```

We can use `Apply` to replace the "head" of a list with `Plus` in order to find the total sum of list elements. Recall that *Mathematica* provides the `@@` shorthand for `Apply`, which makes this a little easier to write (but perhaps a little less understandable).

```
Apply[Plus, {a, b, c}]
% === Plus @@ {a, b, c}
```

```
a + b + c
```

```
True
```

Now for one more digression. Let us do exactly the same thing using folds. Basically, a fold repeatedly apply a binary operation, using an accumulated value and successive elements of a list. We have to provide an initial value for the accumulator. Folds are common in functional programming languages; showing its functional emphasis, *Mathematica* supports folds.

```
ClearAll[f, x0]
Fold[f, x0, {x1, x2, x3}]
```

```
f[f[f[x0, 2], 0.628989], x3]
```

For example, we can form the total sum of list elements by folding `Plus` over the list with a `0` initial value for the accumlator.

```
Fold[Plus, 0, {a, b, c}]
```

```
a + b + c
```

This is a reminder that in *Mathematica* there are often many way to accomplish a single goal. Usually the best choice to make will be the one that you will find easiest to read when you return to your code.

Exercise: use `Fold`, `Times`, and `Range` to produce 10!.

```
Fold[Times, 1, Range[10]] == 10!
```

```
True
```

We can use `Accumulate` to produce the cumulative sum. The last term is the same a folding `Plus` over the list, but we also get intermediate terms.

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

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

`Accumulate` is a special example of folding a list and keeping the intermediate values. This is exactly what the `FoldList` command does. There is one important difference: `FoldList` includes an initial value (that we provide). But we can use `Rest` to discard this initial value.

```
Rest[FoldList[Plus, 0, {a, b, c}]]
```

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

We can similarly accumulate products.

```
Rest[FoldList[Times, 1, {a, b, c}]]
```

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

List Creation

Constant Arrays and Ranges

Mathematica provides powerful facilities for the creation and manipulation of lists.

If you plan to repeatedly enumerate a single value, you may find it faster to use `ConstantArray`. We can use `==` to do an equality comparison of lists.

```
list01 = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
list02 = ConstantArray[0, 10];
list01 == list02
```

```
True
```

Integer intervals are a common need, and we usually create them with `Range`. For example, let us create the first twenty natural numbers.

```
nat20 = Range[20]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

You can also specify a minimum as well as a maximum value for a range of integers. Note that the

range is inclusive: it contains the minimum and maximum values. A stepsize can be given as a third argument.

```
Range[5, 20]
Range[5, 20, 2]
```

```
{5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

```
{5, 7, 9, 11, 13, 15, 17, 19}
```

The arguments to `Range` do not have to be integers.

```
Range[1.5, 10.5, 0.5]
```

```
{1.5, 2., 2.5, 3., 3.5, 4., 4.5, 5., 5.5, 6., 6.5, 7., 7.5, 8., 8.5, 9., 9.5, 10., 10.5}
```

Prepending and Appending Elements

You can Append or Prepend to a list. These commands return a new list, without changing the value of the old list.

```
row = ConstantArray[0, 8]
Append[row, -1]
Prepend[row, 1]
row
```

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

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

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

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

If you want not to produce a new list but to change an existing list, use `AppendTo` and `PrependTo` instead.

```
row = ConstantArray[0, 8];
AppendTo[row, -1];
PrependTo[row, 1];
row
```

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

Suppose I append a list to an empty list:

```
lst = {}
AppendTo[lst, row]
```

```
{}
```

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

Note that lst has one element, which is a list.

```
Length[lst]
```

```
1
```

List Creation using Table

The `Table` command is a more general facility for list creation. It therefore can readily accomplish the same task as `Range`, but slightly more verbosely.

```
Range[20] == Table[i, {i, 20}]
```

```
True
```

The Table command takes as its first argument an expression in a variable and, as its second argument, an “iterator” in that variable. An iterator is just a list with a special format. The iterator may take a few basic forms. The form with only a positive integer stop value, as in `{i,stop}`, will produce numbers natural numbers up to stop. You can also specify a start and stop value, as in `{i,start,stop}`. The default is a unit increment, but you can change that by specifying a third argument, as in `{i,start,stop,increment}`. An alternative is to offer an explicit list of values, as in `{i,mylist}`.

```
Table[0, {5}]
Table[i, {i, 5}]
Table[i, {i, 6, 10}]
Table[i, {i, 1.5, 10.5, 0.5}]
lst = Range[5, 1, -1]; Table[i * i, {i, lst}]
```

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

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

```
{6, 7, 8, 9, 10}
```

```
{1.5, 2., 2.5, 3., 3.5, 4., 4.5, 5., 5.5, 6., 6.5, 7., 7.5, 8., 8.5, 9., 9.5, 10., 10.5}
```

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

The first argument to `Table` can be any expression you want. For example, to produce a list of lists, the first argument can be a list of outputs.

```
tbl = Table[{x, x^2}, {x, 0, 5}]
```

```
{{0, 0}, {1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}}
```

The `Table` command produces nested lists when given multiple iterators. Here for example is a simple list of lists.

```
Table[i * j, {i, {-1, 1}}, {j, 3}]
```

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

Advanced: Note that `Table` localizes the iterator with dynamic scoping, so expressions matching the iterator variable will be iterated. (I do not recommend relying on dynamic scoping; it is too easy to lose track of it.) Example:

```
y := Sin[i * Pi / 2]
Table[y, {i, 4}]
```

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

List Creation from External Data

Naturally *Mathematica* can build lists from external data. For convenience, *Mathematica* ships with some sample data. Here we simply replicate the example at https://reference.wolfram.com/language/tutorial/ReadingTextualData.html. First we take a look at the data:

```
FilePrint["ExampleData/numbers"]
```

```
11.1   22.2    33.3
44.4   55.5    66.6
```

Then we read it into a list:

```
ReadList["ExampleData/numbers", Number]
```

```
{11.1, 22.2, 33.3, 44.4, 55.5, 66.6}
```

Mathematica also includes web access to a variety of data sets. For example, `CountryData[]` produces a list of countries in the database. We can use this same function to produce a variety of sublists. For example, we query for the membership of the EU, the G8, or the G20. These are returned as lists.

```
listG8 = CountryData["G8"]
```

{ Canada , France , Germany , Italy , Japan , United Kingdom , United States }

Note: The special formatting reflects the fact that the list elements are country entities. Such entities may have many predefined properties. See the documentation for details, but the following example provides a hint at what is possible.

```
us = Entity["Country", "UnitedStates"];
gdp2014 = EntityProperty["Country", "GDP", {"Date" → DateObject[{2014}]}];
pop2014 = EntityProperty["Country", "Population", {"Date" → DateObject[{2014}]}];
us[gdp2014] / us[pop2014]
```

```
$54 025.3 per person per year
```

As a final example, *Mathematica* can of course read standard data formats, such as comma-separated values (CSV) files. To illustrate this, let us create one and then read it in.

```
data = RandomInteger[100, {4, 3}] (* make some data *)
(* choose a filename (it must be safe to overwrite!): *)
fname = FileNameJoin[{$TemporaryDirectory, "temp.csv"}];
Export[fname, data]; (* export the data to file *)
FilePrint[fname] (* look at what we stored *)
newdata = Import[fname]; (* import the stored data *)
newdata == data (* check that the data are unchanged *)
```

```
{{12, 62, 47}, {95, 19, 59}, {19, 4, 78}, {65, 64, 80}}
```

```
12,62,47
95,19,59
19,4,78
65,64,80
```

```
True
```

List Manipulation

Mathematica provides powerful list manipulation facilities. Here we touch on a few of the most common ones.

Accessing and Changing List Elements

You can access list elements with doubled brackets. *Mathematica* uses unit based indexing of the list elements; negative indexes count from the end of the list. Here we make a new list out of elements of the old list.

```
lst01 = Range[4];
{lst01[[1]], lst01[[-1]]}
```

```
{1, 4}
```

Mathematica lists are mutable. We can use indexing on the left side of an assignment to change the value of an element. *Mathematica* effectively copies lists on assignment, so in the following example `lst01` is changed but `lst02` is not changed. (I.e., `lst01` and `lst02` refer to two different lists.) Note the use of a list of indexes to change multiple elements.

```
lst01 = Range[4]; lst02 = lst01;
lst01[[{1, -1}]] = {91, 94};
lst01
lst02
```

```
{91, 2, 3, 94}
```

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

We can retrieve and replace contiguous elements with an index range specified as `start;;stop` or `start;;stop;;step`.

```
x = Range[15]
x[[4 ;; 10]]
x[[4 ;; 10 ;; 2]]
x[[4 ;; 10 ;; 2]] = 0
x
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

```
{4, 5, 6, 7, 8, 9, 10}
```

```
{4, 6, 8, 10}
```

```
0
```

```
{1, 2, 3, 0, 5, 0, 7, 0, 9, 0, 11, 12, 13, 14, 15}
```

We can retrieve and replace multiple parts by using a list of indexes.

```
x = Range[10]
x[[{1, 1, 3, 7}]]
x[[{1, 2, 3}]] = {21, 22, 23}
x
```

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

```
{1, 1, 3, 7}
```

```
{21, 22, 23}
```

```
{21, 22, 23, 4, 5, 6, 7, 8, 9, 10}
```

Other commands for accessing specified elements include `First`, `Rest`, `Most`, `Last`, `Extract`, `Take`, and `Drop`.

Advanced: Accessing and Changing List Elements

The indexing brackets are actually a shorthand for the `Part` command. But most commonly, we use the the indexing brackets.

```
Clear[lst]
lst[[1]] // FullForm // Quiet
```

```
Part[lst, 1]
```

If we want a closer look at what is involved with this assignment with an index, we can again look at the `FullForm`.

```
Clear[x]
HoldForm[FullForm[x[[1]] = 99]]
```

```
Set[Part[x, 1], 99]
```

Assignment with an index mutates an existing list. When a new list is preferred, use `ReplacePart`, which applies a rule (rather than using `Set`).

```
x = {0};
xnew = ReplacePart[x, 1 → 99]
x (* unchanged! *)
```

```
{99}
```

```
{0}
```

Always creating a new list is safer, because it removes ambiguity about the contents of a list. However, it is computationally more costly, especially if many changes will be sequentially made to a large list. It is up to the usef to make the right trade-offs.

Advanced: Although Mathematica lists are mutable, they must be changed indirectly, using the symbol that refers to the list. (In this case, the symbol `lst01`.) For example, the following produces an error:

```
{0}[[1]] = 1;
```

```
Set::setps : {0} in the part assignment is not a symbol. >>
```

This matters when passing lists to functions, because the default behavior is to evaluate any argument before evaluating the function.

Reversing, Sorting, Accumulating

Lists can be reversed, sorted, and accumulated (as a cumulative sum).

```
myList = RandomInteger[20, 5]  (* a list of 5 random numbers *)
mySortedList = Sort[myList]
myReversedList = Reverse[mySortedList]
myCumulativeSum = Accumulate[mySortedList]
```

```
{3, 7, 12, 0, 2}
```

```
{0, 2, 3, 7, 12}
```

```
{12, 7, 3, 2, 0}
```

```
{0, 2, 5, 12, 24}
```

By using `SortBy`, we can sort a list on arbitrary criteria. For example, we can use `SortBy` to sort the G8 countries by population. (We will reverse the sign on population in order to sort from greatest to least.)

```
popSortedG8 = SortBy[listG8, country ↦ -country[pop2014]]
```

{ United States , Japan , Germany , France , United Kingdom , Italy , Canada }

Map and MapThread

Often we wish to apply some function to every element of a list in order to produce a new list of values. For this we use `Map`, for which `/@` is an alternate infix syntax.

```
ClearAll[f]
Map[f, {x1, x2, x3}]
f /@ {x1, x2, x3}
```

```
{f[2], f[0.769947], f[x3]}
```

```
{f[2], f[0.844339], f[x3]}
```

Recall that *Mathematica* allows the creation of anonymous (unnamed) functions. This practice is very common when using `Map`.

```
Map[x ↦ x^2, Range[20]]
(x ↦ x^2) /@ Range[20]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
```

Recall that *Mathematica* allows us to designate parameter slots instead of naming our paramters. Here is our list of squares once again, produced with this syntax.

```
Map[#^2 &, nat20]
#^2 & /@ nat20
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
```

However, can does not imply should, and the notation in the last example undercuts readability for most people.

If the function used by `Map` produces a list, then we get a list of lists as our result.

```
Map[ctry ↦ {ctry[[2]], ctry[gdp2014] / 10^12}, popSortedG8]
```

```
{{UnitedStates, $17.419 per year},
 {Japan, $4.60146 per year}, {Germany, $3.85256 per year},
 {France, $2.82919 per year}, {UnitedKingdom, $2.94189 per year},
 {Italy, $2.14434 per year}, {Canada, $1.78666 per year}}
```

`MapThread` generalizes `Map` to functions that have multiple arguments. The arguments are given as a list of lists of values, where the inner lists have a common length.

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

```
{f[2, y1], f[0.96874, y2], f[x3, y3]}
```

For example, we can make a list of rules like this:

```
MapThread[Rule, {{x1, x2, x3}, {y1, y2, y3}}]
```

```
{2 → y1, 0.989897 → y2, x3 → y3}
```

`MapThread` can do much more than this; see the documentation. To give just one example, we can apply a list of functions to a list of arguments.

```
ClearAll[f, g, h, x, y, z]
MapThread[{f, x} ↦ f[x], {{f, g, h}, {x, y, z}}]
```

```
{f[x], g[y], h[z]}
```

Aside: If a function has already been applied to list of arguments, you can use `Thread` instead of `MapThread`. This can be used to thread equations. Somewhat surprisingly, `Thread` allows an argument to be a constant.

```
Thread[f[{x1, x2, x3}, {y1, y2, y3}]]
Thread[{x1, x2, x3} == {y1, y2, y3}]
Thread[{x1, x2, x3} == 0]
Thread[0 == {y1, y2, y3}]
```

```
{f[2, y1], f[0.629764, y2], f[x3, y3]}
```

```
{2 == y1, 0.491755 == y2, x3 == y3}
```

```
{False, False, x3 == 0}
```

```
{0 == y1, 0 == y2, 0 == y3}
```

Exercise: noting that {x,y} is the same as List[x,y], use `MapThread` or `Thread` to transpose a rectangular list of lists.

```
Thread[{{x1, x2, x3}, {y1, y2, y3}}]
MapThread[List, {{x1, x2, x3}, {y1, y2, y3}}]
```

```
{{2, y1}, {0.13913, y2}, {x3, y3}}
```

```
{{2, y1}, {0.101357, y2}, {x3, y3}}
```

```
gdpG8 = Map[c ↦ c[gdp2014], listG8];
popG8 = Map[c ↦ c[pop2014], listG8];
gdppcG8 = MapThread[{c, g, p} ↦ {c[[2]], Round[g / p]}, {listG8, gdpG8, popG8}]
```

```
{{Canada, $50 600 per person per year}, {France, $44 136 per person per year},
 {Germany, $47 198 per person per year}, {Italy, $35 052 per person per year},
 {Japan, $36 454 per person per year}, {UnitedKingdom, $46 288 per person per year},
 {UnitedStates, $54 025 per person per year}}
```

From Lists of Lists to Formatted Tables

We often want to present data in tabular form. A list of lists can be nicely formatted with `TableForm` or `Grid`.

TableForm

The `Table` command can produce a list of lists, using two expressions. Two-dimensional tables can be nicely formatted with the `TableForm` command. Here is an example.

```
tbl = Table[{x, x^2}, {x, 0, 5}]
TableForm[tbl, TableHeadings → {None, {"x", "x^2"}}]
TableForm[tbl, TableHeadings → {None, {"x", "x^2"}}, TableDirections → Row]
```

```
{{0, 0}, {1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}}
```

x	x^2
0	0
1	1
2	4
3	9
4	16
5	25

x	0	1	2	3	4	5
x^2	0	1	4	9	16	25

Recall that the `Table` command also accepts multiple iterators, producing nested lists of lists. Here is an example.

```
ivals = Range[3];
xvals = Range[0, 10];
tbl = Table[x^i, {i, ivals}, {x, xvals}];
formattedTable =
 TableForm[tbl, TableHeadings → {ivals, xvals}, TableAlignments → Right]
```

	0	1	2	3	4	5	6	7	8	9	10
1	0	1	2	3	4	5	6	7	8	9	10
2	0	1	4	9	16	25	36	49	64	81	100
3	0	1	8	27	64	125	216	343	512	729	1000

We may want our tables to be framed and labeled.

```
ySortedG8 = SortBy[gdppcG8, x ↦ -x[[2]]];  (* sort G8 by GDP per capita *)
table = TableForm[ySortedG8, TableHeadings → {None, {"Country", "GDP p.c."}}];
Labeled[Framed[table], "G8 GDP per capita for 2014"]
```

Country	GDP p.c.
UnitedStates	$54 025 per person per year
Canada	$50 600 per person per year
Germany	$47 198 per person per year
UnitedKingdom	$46 288 per person per year
France	$44 136 per person per year
Japan	$36 454 per person per year
Italy	$35 052 per person per year

G8 GDP per capita for 2014

For additional options, read the documentation for the `Table` command.

Using `Grid` for Two-Dimensional Tables

We can achieve fine formatting control with the `Grid` command. (Read the documentation.) But it still provides simple way to produce a two dimensional display of nested lists.

```
mydata = Partition[Range[15], 3];
Grid[mydata]
```

```
 1   2   3
 4   5   6
 7   8   9
10  11  12
13  14  15
```

We can format gridded data, aligning it and adding dividers.

```
mydata02 = Prepend[mydata, {"header1", "header2", "header3"}];
Grid[mydata02, Alignment → Right, Dividers → {None, 2 → True}]
```

header1	header2	header3
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15

We can even shade specific rows.

```
Grid[Prepend[gdppcG8, {"Country", "GDP p.c."}],
 Alignment → Left,
 Dividers → {None, 2 → True},
 Background → {None, 1 → Lighter[Gray, 0.9]}
]
```

Country	GDP p.c.
Canada	$50 600 per person per year
France	$44 136 per person per year
Germany	$47 198 per person per year
Italy	$35 052 per person per year
Japan	$36 454 per person per year
UnitedKingdom	$46 288 per person per year
UnitedStates	$54 025 per person per year

Plots

Plots are an important way to explore data and functions. *Mathematica* provides extensive and powerful plotting facilities. This section provides only a basic introduction to some core plotting functions.

From Lists to Plots