Chapter 1: Getting Started

This chapter provides a basic introduction to the use of Mathematica notebooks. You will learn how to enter text, how to perform basic numerical and symbolic computations, how to add graphics to a notebook, and how to create interactive expressions.

1.1 Notebooks and the Front End

The Mathematica software includes an interactive graphical user interface (GUI) called the *front end*. We use the front end to perform interactive computations with the Wolfram Language (WL) and to keep a record of these computations as notebooks. We can also add text and graphics to our notebooks.

1.1.1 Notebook Creation and Sharing

To create a new notebook, launch Mathematica and then pick from the welcome-screen menus `New Document > Notebook`. Notebook files are saved with a `.nb` extension. These notebooks can naturally be shared with anyone who has access to Mathematica or to the Wolfram Cloud. However you can also save both static and dynamic content in other useful and shareable formats.

The most useful format for sharing static content is probably the portable document format (PDF). Before producing a PDF, save your notebook. Then, pick from the front-end menus `File » SaveAs` and set `Save as Type` to `PDF Document`. (Alternatively, you can use the `Export` and `EvaluationNotebook` commands.)

For sharing dynamic content with people who do not have Mathematica, 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 supports some dynamic and interactive objects (including `Manipulate`, `Dynamic`, and `DynamicModule`).

Advanced

The computations in a notebook are performed by the Mathematica kernel. The front end and the kernel talk to each other via *MathLink*, a communications protocol. Programmers can use *MathLink* to communicate with Mathematica from other programs or applications.

1.1.2 Cells and Evaluation

A notebook is a collection of cells. After creating a new notebook, you can 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. Each cell has a style, which affects how it behaves. If you would prefer a visual introduction to these details, see the Wolfram screencast entitled Hands-on Start to Mathematica.

Click on a cell’s right edge to make it active, and then use the `Format » Style` menu to give it a style. (There are also keyboard shortcuts for the `Format` menu; see the Wolfram Language Tutorial entitled

Keyboard Shortcut Listing.) The default cell style is “Input”, which is for WL expressions that you wish to evaluate. 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`. You may also find the `Title`, `Section`, and `Subsection` styles to be useful. If you want to append a new cell with the same style you are currently using, press `ALT+ENTER`.

Text cells are intended for ordinary text. Give a cell a `Text` style when you want to provide discussion or descriptions of your calculations. The front end will do some basic formatting of this text, in a manner similar to a word processor. (For example, you can format your the font face of your text as *italic* or **bold**.)

Use the default `Input` style if your cell contains WL expressions that you want to evaluate. These expressions may be numerical or symbolic. Evaluate an `Input` cell by pressing `SHIFT+ENTER`. The result is placed a separate cell, which has the `Output` style.

This means that you can readily use a notebook as a simple calculator. Just enter a numerical expression into an `Input` cell, and evaluate it. For example, consider the following arbitrary arithmetic expression. (Remember, after entering an expression into an `Input` cell, press `SHIFT+ENTER` to evaluate it.)

```
2/4
```

$\frac{1}{2}$

By default, the notebook presents a “predictive interface”, which appears below the output when an expression is evaluated. Although it has the potential to be useful, as of version 11, this is considered by many users to be a source of surprising and unwanted behavior. To turn it off, go to the `Interface` tab in `Edit » Preferences`, and uncheck `Show Suggestions Bar after last output`.

1.1.3 Exact and Approximate Numbers

The result of the previous evaluation may have surprised you. WL distinguishes between exact numbers and approximate numbers. Exact numbers are roughly integers, rational numbers, a few special constants. Approximate numbers include the floating point approximations to real numbers. Computations done entirely with exact numbers return exact results. This is why our previous evaluation returned 1/2 instead of 0.5: this indicates that we evaluated an expression that used only exact numbers.

Computations with approximate numbers resemble the ordinary floating-point computations usually encountered in other programming languages. We usually create approximate numbers by including a decimal point. Expressions in both exact and approximate numbers will of course be approximate.

Expression	Result	Comment
2.0/4.0	0.5`	approximate
2/4.0	0.5`	approximate
2/4	$\frac{1}{2}$	exact

Complex numbers can also be exact or approximate. (For the imaginary part, produce the symbol *i* as `ESCiiESC`.) If either the real or imaginary part is approximate, then the entire number is approximate.

Here are a few examples.

Expression	Result	Comment
1.0/2+2/3ⅈ	0.5 + 0.666667 ⅈ	approximate
1/2+2.0/3ⅈ	0.5 + 0.666667 ⅈ	approximate
1/2+2/3ⅈ	$\frac{1}{2}+\frac{2 i}{3}$	exact

Occasionally it is desirable to do our computations with exact numbers. However, this can quickly become computationally costly. We can use the `N` command to turn an exact number into an approximate number. (This is like casting to a float value in some other programming languages.)

```
N[355/113] (* crude approximation to π *)
```

```
3.14159
```

Note the comment included with our computation. Input cells can contain comments, which we often use to explain an expression. Comments begin with an opening parenthesis and asterisk and end with an asterisk and closing parenthesis.

Advanced

FullForm and TreeForm

Each WL 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 occasionally have reason to consider this full form, especially when debugging. You can use the `HoldForm` and `FullForm` commands to display the full form of an expression. (The `FullForm` command returns the full form of the expression, and the `HoldForm` command prevents its evaluation.)

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

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

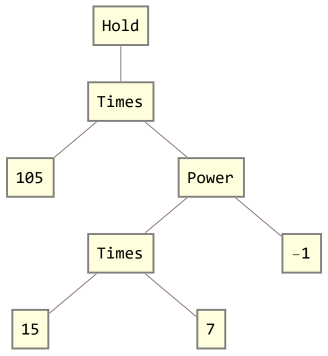
We can use `ReleaseHold` to evaluate the expression seen by `HoldForm`.

```
ReleaseHold[%]
```

```
1
```

The full form of an complex expression is sometimes difficult to parse visually, in which case it can be useful to produce a graphical representation of the expression tree. WL provides this functionality with the `TreeForm` command. Here is an example of the tree form of a simple calculation, where we use the `Hold` command to prevent the evaluation of the expression.

```
TreeForm[Hold[105 / (15 * 7)], ImageSize → Small]
```



1.1.4 Pretty Printing with Strings

Sometimes we want to produce nicely formatted output. WL provides strings as a basic data type. To create a string literal, just type text characters inside double quotes. Evaluating a string literal produces a string object. Its type is identified by its “head”, which as always we can produce with the `Head` command.

```
Head["The double quotes make this a string."]
```

```
String
```

String concatenation combines multiple strings into a single string. You can use the `StringJoin` command, but Mathematica also offers an unusual angle-bracket shorthand syntax for this.

```
"Strings can be" <> " concatenated."
```

```
Strings can be concatenated.
```

Note that the output display of a string does not include its quotes. We can force the quotes to display with the `InputForm` command.

```
InputForm["The double quotes make this a string."]
```

```
"The double quotes make this a string."
```

Often we want to create a control string and then substitute values into it. A control string is just a normal string, but it includes backticks to indicate the locations of substitutions to be made. For example,

```
s = "The sum of `` and `` is ``.";
```

We can use `StringTemplate` to create a template object from a control string. For example,

```
tmp = StringTemplate[s];
```

We can then use the template object repeatedly, to produce nicely formatted strings with varied substitutions.

Expression	Result
tmp[5,2,5+2]	The sum of 5 and 2 is 7.
tmp[5,3,5+3]	The sum of 5 and 3 is 8.

Advanced

Prior to version 10 of WL, this kind of templating was often handled with the `StringForm` command. By default, the value of a `StringForm` displays like a `String`, but the result of evaluating a `StringForm` expression has a head of `StringForm`, not `String`. This occasionally matters. You can convert any string form you produce to a string with the `ToString` command.

1.1.5 Errors

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. Furthermore, should you try to evaluate an expression that simply takes too much time, you can pick `Evaluation » Abort Evaluation` from the Mathematica 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 that Mathematica will work on it with the `TimeConstrained` command. (We will discuss Mathematica commands very soon.)

It is rare but possible to evaluate an expression that causes Mathematica to lock up, so be sure to save often. It is possible to ask Mathematica to autosave your notebook, but since half-written code may be useless or worse, you may find it more useful to save often by hand (e.g., with `CTRL+s`). Although very rare, it is also possible to put Mathematica in a state where it has trouble reloading your document. If this happens, try following Wolfram’s instructions for resetting Mathematica to its default configuration.

1.2 Constants, Commands, and Operators

All builtin WL names begin with a capital letter. It is therefore a convention that user-defined symbols should start with a lowercase letter.

1.2.1 Constants and Commands

WL includes a few special mathematical constants, documented by the Wolfram Language Tutorial entitled Mathematical Constants. Here is a list (in braces) of some of these special constants.

Expression	Result
Pi	π
E	ⅇ
I	ⅈ
Infinity	∞

When we evaluate one of these special constants, the result displays as a special symbol. In fact, it is often convenient to use these special symbols instead of the WL names. To enter the symbols, use the escape sequences `pi`, `ee`, `ii`, and `inf`. An escape sequence begins and ends by pressing the escape key. For example, `ESCpiESC` displays as π.

Unlike most programming languages, WL includes thousands of builtin commands. The name of each command begins with a capital letter. We apply a command by appending square brackets, within which we supply any needed arguments. (Any specific input value we supply to a command is called an *argument*.) Note that application uses brackets, not the parentheses often used in many other popular languages. Many commands can work with both exact numbers and approximate numbers, returning exact results when possible.

Expression	Result
`Sin[π]`	`0`
`Cos[π]`	`-1`

When we prefer an approximate result, we can use the `N` command to convert exact numbers or numerical mathematical constants into approximate numbers.

Expression	Result	Comment
`π`	`π`	exact
`N[π]`	``3.141592653589793` ``	approximate
`Sin[π/4]`	$\frac{1}{\sqrt{2}}$	exact
`N[Sin[π/4]]`	``0.7071067811865475` ``	approximate

The result of an evaluation is often an expressions that clearly can be simplified, but WL does not automatically assume that one prefers the simplified representation. We often can use the `Simplify` command to produce a simpler representation of a complicated expression. (When this does not produce a satisfactory result, we can ask for a greater effort at simplification by using the `FullSimplify` command.)

Expression	Result
`Sin[θ]^2+Cos[θ]^2`	$\text{Cos}[\theta]^2+\text{Sin}[\theta]^2$
`Simplify[Sin[θ]^2+Cos[θ]^2]`	`1`

1.2.2 Arithmetic Operators

Computer science uses the term *operator* somewhat differently than mathematics. In computer science, an operator is essentially a special symbol that behaves like a function but uses a special syntax. For example, WL provides the `Plus` command for addition, but it also provides the more familiar `+` infix operator. (A binary operator is called *infix* if it should be placed between its two operands.)

Expression	Result	Comment
`Plus[8,3]`	`11`	using command
`8+3`	`11`	using operator

WL includes the usual collection of infix arithmetic operators: +, -, *, /. Exponentiation is done with the ^ operator. Each infix operator is shorthand for a WL command: `Plus`, `Subtract`, `Times`, `Divide`, and `Power`. Operations on exact numbers return exact results.

Expression	Result
355+2	357
355-2	353
355*2	710
355/2	$\frac{355}{2}$
355^2	126025

As always, exact numbers can be converted to approximate numbers with the `N` command.

Expression	Result
N[355/2]	177.5`
N[355/113]	3.1415929203539825`

As in most programming languages, we can use parentheses to determine the order of evaluation in an expression.

Expression	Result
105/15*7	49
105/(15*7)	1

If we leave off parenthesis, the order of evaluation is determined by operator precedence and rules of operator associativity. Operator precedence is described in the Wolfram Language Tutorial entitled Operator Input Forms, but it is best to address any possible uncertainty on the part of future readers of your code by relying on parentheses.

Subtraction and division associate from the left. That means that we reading from left to right we perform operations as soon as possible.

Expression	Result
40-4-2	34
(40-4)-2	34
40-(4-2)	38
40/4/2	5
(40/4)/2	5
40/(4/2)	20

However, exponentiation associates from the right. This means that when we chain the exponentiation operator without using parenthesis, the entire expression to the right of the operator is treated as the operand.

Expression	Result
4^3^2	262144
(4^3)^2	4096
4^(3^2)	262144

Computation with numbers often holds one big surprise for novice programmers: negation is treated as an operation with low precedence. This means that you may have a surprising need for parentheses when working with negative numbers. In the following example, exponentiation takes precedence over negation, unless we force the order of evaluation by using parentheses. (To make things worse, not all programming languages follow this convention, and spreadsheet languages in particular often deviate from it.)

Expression	Result
-1^2	-1
(-1)^2	1

For more detail, see the Wolfram Language Tutorial entitled Arithmetic.

To draw attention to particular results (e.g., significant values in a statistical table), it is often convenient to add styling that renders certain expressions salient. WL allows you to style your output. However, keep in mind that a formatted object does not behave like the underlying object. Once a number is styled, for example, you can no longer simply do arithmetic with the result. Here is a simple example, where we apply a bold style to a number.

Expression	Result	Comment
six=Style[6,Bold]	**6**	a styled number
5+six	5 + **6**	addition not performed

1.2.3 Relational and Logical Operators

WL includes the usual collection of relational operators: <, ≤, ==, ≠, ≥, >. (These are infix shorthands for the `Less`, `LessEqual`, `Equal`, `GreaterEqual`, and `Greater` commands.) Expressions are evaluated before they are compared, and exact and approximate numbers compare in a natural fashion. The value of a numerical comparison is “boolean” (i.e., either `True` or `False`). For example,

Expression	Result
1<1	False
1≤1	True
1==3/3	True
1==1.0	True

WL provides the usual collection of logical operators: `!`, `&&`, and `||` are shorthand for the `Not`, `And`, and `Or` commands. Note the doubled characters for `And` (`&&`) and `Or` (`||`), in accord with many languages deriving from the C programming language. WL also offers somewhat less standard logical operations, including `Xor`, `Nand`, and `Nor`. We can use WL’s logical operators to combine multiple relational comparisons. However, chaining relational operators is permitted, and the resulting expressions can be shorter and more readable than with explicit use of logical operators.

Expression	Result
(1==1+0)&&(1+0==0+1)	True
1==1+0==0+1	True

The `And` and `Or` commands accept an arbitrary number of arguments, so we can use them to determine whether all or any comparisons are `True`.

Expression	Result
And[1<1,1≤1,1==1,1≠1,1≥1,1>1]	False
Or[1<1,1≤1,1==1,1≠1,1≥1,1>1]	True

As in many other languages, WL uses *short-circuit* evaluation of logical comparison, evaluating only as many arguments as needed to determine the value. (To reiterate: arguments may not all be evaluated before the function is.)

Material implication and biconditional equivalence are provided by the `Implies` and `Equivalent` commands, with their `⇒` and `⇔` shorthands. These behave according to their usual definitions, which can be somewhat counterintuitive on first encounter.

Expression	Result
0>1 ⇒ 3>2	True
0>1 ⇔ 3<2	True

In order to understand these results, recall that the behavior of logical operators is defined by their truth-table representations. We can use the `BooleanTable` command to create simple truth tables. Use `TableForm` with the `TableHeadings` option to add a header. (As always, turn to the Wolfram documentation for usage details for these commands. Additionally, we further illustrate the `TableForm` command in the next chapter.)

```
Clear[p, q]
props = {p, q, p && q, p || q, p ⇒ q, p ⇔ q};
btbl = BooleanTable[props, {p, q}];
TableForm[btbl, TableHeadings -> {None, hdrs}]
```

```
True    True    True    True    True    True
True    False   False   True    False   False
False   True    False   True    True    False
False   False   False   False   True    True
```

Advanced

Testing for Equality

WL uses “clever” numerical equality comparison, relying on numerical approximation to test for equality. Thus for example the following comparison evaluates to `True` in WL, whereas (due to rounding error) it would not in most programming languages.

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

```
True
```

If you really want to test for sameness (after evaluation) and not just numerical equality, use three equals signs (`===`, which is the shorthand for `SameQ`). For WL, exact numbers are not the same thing as approximate numbers.

Expression	Result
1===3/3	True
.3===.1+.2	True
1===1.0	False

For more detail, see the Wolfram Language Tutorial entitled Relational and Logical Operators and the Wolfram Language Guide entitled Boolean Computation.

1.3 User-Defined Symbols

WL defines a large number of symbols to be useful commands. We can also introduce new symbols whenever we wish. As a very useful convention, WL's builtin symbols always capitalize the first letter. To avoid potential conflicts with the builtin definitions, user-defined symbols should therefore always start with a lowercase letter.

You can use almost any combination of letters to define a symbol. A symbol cannot start with a digit, but you can use numeric characters anywhere else in your symbols. You should avoid using dollar signs in your symbols. You must not end a 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—at least, not without some special effort.

1.3.1 Setting and Unsetting the Values of Variables

We often introduce symbols as variable names: we 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. This should feel very familiar, except for the fact that the defining expression can be entirely symbolic (instead of numerical).

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

```
b + c
```

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

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

```
b + c
```

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

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

After using the equals sign (or `Set`) to make an assignment, the defined symbol will simply be replaced in subsequent expressions by its definition.

```
a * a
```

$(b + c)^2$

Assignments will propagate. An assignment creates a rule for rewriting future expressions. (Here we use a semicolon to separate two expressions. This creates a compound expression whose value is the value of the final subexpression.)

```
b = 1; a * a
```

```
(1 + 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`. (Closely related functionality is provided by the `Clear` command, which we will more commonly use.)

Expression	Result	Comment
a=b+c	b + c	set the value of \`a\`
a*a	$(b+c)^2$	symbolic multiplication
a=.		unset \`a\`, which now has no value
a*a	a^2	symbolic multiplication

In a notebook, you can access the previously computed value with a percent sign. (The percent sign is a shorthand for `Out[-1]`.) As an example, let us do a computation on multiple lines but only display the final result. First we compute a product by suppress printing the result.

```
2 * 3;
```

Although the output was suppressed, the value computed is still accessible.

```
% + 5
```

```
11
```

This can be useful for quick and dirty calculations in a notebook, if you are careful. However, be aware that a single compound expression has the value of the last expression evaluated, and you cannot use the percent sign to retrieve pieces of the compound expression evaluation.

```
2 * 3; % + 5 (* % retrieves previous expression, not subexpression! *)
```

```
16
```

Contexts

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 this has a special meaning in WL. (It is used to produce patterns, which we will explore latter on.) The only good choice from a standard American keyboard is the back tick. This actually creates a symbol with a “context”.

Every WL 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`
```

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]
```

```
my`
```

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

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

For more information, see the Wolfram Language Tutorial entitled Contexts.

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

Important

Recall that notebook computations are done by a 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 implies that all your open notebooks are sharing all your symbol definitions. As you start working in your Notebooks, remember that symbol definitions are shared: any symbol that you define in one notebooks is shared with all your open notebooks. You can reset a notebook's default context via the front-end menus (`Evaluation » Notebook's Default Context`).

Multiple Assignment and Clear

You can make many assignments at once by putting comma-separated variable names and assigned values in braces. (Comma-separated expressions within braces constitute a list of the expressions; we will soon discuss lists in more detail.) Since the right hand side of an assignment is evaluated before the assignment takes place, you can use multiple assignment to swap variable values. As we see from the table, you can clear many symbols and one go by providing them as arguments to `Clear`.

Expression	Result	Comment
{a01,a02}={1,2}	{1, 2}	set both values
{a01,a02}	{1, 2}	show both values
{a01,a02}={a02,a01}	{2, 1}	swap values
{a01,a02}	{2, 1}	show the new values
Clear[a01,a02]		unset multiple values
{a01,a02}	{a01, a02}	`a01` and `a02` were unset

Advanced

Use `ClearAll` instead of `Clear` if you want to clear `Attributes` and `Options` associated with a symbol. (We will discuss this more later.)

1.3.2 Variable Scope

By default, 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 WL expressions. For example, without defining x or y we can write utility as $u = \sqrt{x\,y}$. WL uses this expression to define u even though x and y remain undefined. Moreover, we can *subsequently* assign values to these variables, and it affects the values of u. If we then use the `Clear` command to clear the values of x and y, then we return to a definition of u in terms of undefined variables.

Expression	Result
`u=Sqrt[x*y]`	$\sqrt{x\,y}$
`x=5;y=20;u`	10
`Clear[x,y];u`	$\sqrt{x\,y}$

The default global scope is often convenient in a notebook setting, but it does pose hazards. For example, it means 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 occurrence 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 `Input` cells in order.)

For this and other reasons, you may want to ensure that some symbol definitions are local to an expression. You can accomplish this with the `With` command (for local constants) and the `Module` command (for local variables). Here we briefly illustrate how the use of `With` and `Module` can shield computations. For more details, see the section on functions in this chapter, and see the Wolfram Language Tutorial entitled How Modules Work.

Expression	Result	Comment
`t=0`	0	set a global value for \`t\`
`With[{t=25},Sqrt[t]]`	5	use a local constant named \`t\`
`t`	0	the global \`t\` was not affected
`Module[{t=1},t+=24;Sqrt[t]]`	5	use a local variable named \`t\`
`t`	0	the global \`t\` was not affected

Note the presence of multiple, semicolon separated expressions inside the `With` and `Module` expressions above. The value of the overall expression is the value of the last of these, which we call the terminal expression. Be sure that you do not append a semicolon to the terminal expression; that will suppress this value.

1.3.3 Delayed Assignment

We have seen that assignment (`Set`) immediately evaluates the right hand side and assigns the resulting value to the left hand side. The evaluation of the right-hand side is done once. Sometimes we want a very different behavior: we would like the right-hand side to remain unevaluated until the left-hand side symbol is evaluated. We can achieve this with the `SetDelayed` command or with the equivalent `:=` operator. The evaluation of the right-hand side is done anew each time the left-hand side symbol appears.

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

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

Expression	Result	Comment
x1=RandomInteger[100]	55	assign a value now
{x1,x1,x1}	{55, 55, 55}	the same each time
x2:=RandomInteger[100]		no value assigned yet
{x2,x2,x2}	{23, 62, 78}	can differ each time

Advanced

Set vs SetDelayed

The rest of the material in this section may appear a bit subtle on the first reading. When we use \`Set\`, the assigned expression is first evaluated and then assigned. If we make an assignment with a symbol that has not yet been assigned to, the symbol is its own value. In this case there is little difference between \`Set\` and \`SetDelayed\`. We can see this by looking directly at the definition via the \`OwnValues\` command, which shows us the rules that will be used when evaluating a symbol. If we look at the information for \`b1\` and \`b2\`, we see the different assignment operators that we used, but beyond that we do not learn much. However, we can learn more by using the \`OwnValues\` command. This shows us that the rules produced by our assignments were in fact identical.

Expression	Result	Comment
b1=a	a	assignment
b2:=a		delayed assignment
OwnValues[b1]	{HoldPattern[b1] :→ a}	the rule for \`b1\`
OwnValues[b2]	{HoldPattern[b2] :→ a}	the rule for \`b2\`

As a result, both \`b1\` and \`b2\` change in step when we change \`a\`.

Expression	Result
a=1	1
{b1,b2}	{1, 1}
a=2	2
{b1,b2}	{2, 2}

However, there is a radical difference between \`Set\` and \`SetDelayed\` when the symbol on the right has already been assigned to. We can see this difference when we look at the information for the two symbols. Again, \`Set\` will first evaluated the assigned expression, and then use the current value for assignment. In contrast, \`SetDelayed\` will wait until the moment of use to determine the value of the assigned expression. Notice how this shows up in the difference in the \`OwnValues\` of \`b1\` in the following case. We see this difference when we exam the own-values of the two symbols.

Expression	Result	Comment
a=3	3	first assign to \`a\`
b1=a	3	assignment
b2:=a		delayed assignment
a=2	2	set a value for \`a\`
OwnValues[b1]	{HoldPattern[b1] :→ 3}	new rule for \`b1\`
OwnValues[b2]	{HoldPattern[b2] :→ a}	same old rule for \`b2\`

The result is that now \`b1\` and \`b2\` do not move together as we change \`a\`.

Expression	Result
a=1	1
{b1,b2}	{3, 1}
a=2	2
{b1,b2}	{3, 2}

1.3.4 Manipulating Symbols

In addition to supporting familiar numerical computations, WL is a computer algebra system that allows us to do algebra entirely symbolically.

Symbolic Comparison

WL makes several different uses of variants of the equals sign. The most familiar uses are assignment (`=`) and equality testing (`==`). Less familiar are delayed assignment (`:=`) and identity testing (`===`). Equality and identity testing return boolean values (`False` or `True`).

Expression	Result	Comment
x1==1.0	False	equality testing (Equal}
1===1.0	False	identity testing (SameQ)

The double equals sign produces an equality comparison. (See the documentation for the `Equal` command.) The triple equals sign tests for the sameness or equivalence of the expressions. (See the documentation for the `SameQ` command.) Recall that exact numbers are not the same as approximate numbers.

We can also test equality and equivalence between symbolic expressions.

Expression	Result	Comment
a=b	b	assignment (Set)
a==b	True	equality test (Equal)
a===b	True	sameness test (SameQ)

Symbolic equations may not automatically be simplified, in which case the original equation is returned (possibly rearranged). We may be able to force simplification with the `Simplify` command (or the more computationally costly `FullSimplify` when necessary).

Expression	Result
Sin[θ]^2+Cos[θ]^2==1	$\mathrm{Cos}[\theta]^2 + \mathrm{Sin}[\theta]^2 == 1$
Simplify[Sin[θ]^2+Cos[θ]^2==1]	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. For this purpose, `Simplify` accepts a second argument, which states the assumptions under which the simplification is to take place. Note how the relational operators get repurposed to impose relations, not only to test for them.

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

```
True
```

A word of 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). (This may seem puzzling: how can a command can directly change the value of its argument? We take this up later when discussing the `HoldAll` attribute.) We can append `++` to a symbol as a shorthand for `Increment`. Each time the `Increment` command is evaluated, the value of its argument is changed.

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

```
False
```

Symbolic Algebra

First-time users of symbolic algebra system are often startled by the ability to evaluate expressions containing symbols to which no assignment has been made. To demonstrate, here we first use the `Clear` command to clear any definitions that might have been previously assigned to our symbols, then we symbolically solve a quadratic equation.

```
Clear[a, b, c, x]
```

$$\texttt{soln = Solve}\left[a * x^2 + b * x + c == 0, x\right]$$

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

Note that the multiple solutions are returned as a list of solution rules. Naturally, we would often like to know the numerical value of our solution for specific parameter values. One may of course assign values to the parameters and then solve the equation. In a symbolic algebra package, however, we often find ourselves instead making substitutions into a symbolic expression. We can accomplish this with the `ReplaceAll` command, which has a slash-dot infix-operator form (used here). The first argument is the expression for the replacement substitutions, and the second argument is a list of substitution rules.

```
soln /. {a → 1, b → 0, c → -1}
```

```
{{x → -1}, {x → 1}}
```

By relying on substitutions rather than assignment, we can easily look at the value of the solution for a different parameter set.

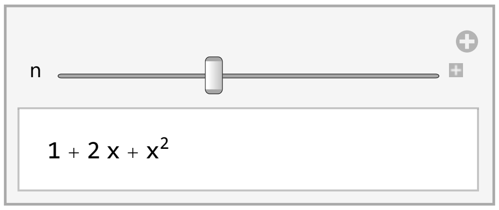
```
soln /. {a → 2, b → 5, c → 2}
```

$$\left\{\{x \to -2\}, \left\{x \to -\frac{1}{2}\right\}\right\}$$

Dynamic Manipulation

A fascinating and powerful feature of notebooks is their support for interactive expressions. Here is a simple example, using the `Manipulate` command. This command takes two arguments: an expression that involves a control variable, and a list describing the control. Here we create an interactive expansion of the expression $(1+x)^n$ for various values of n, from 0 to 5 (in increments of 1). We also provide a initial value of $n=2$.

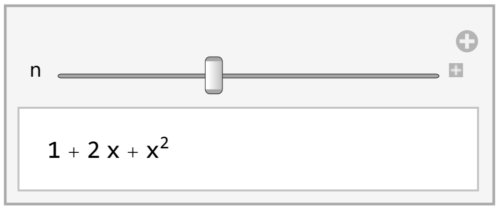
```
Clear[x, n]
Manipulate[Expand[(1 + x)^n], {{n, 2}, 0, 5, 1}]
```



```
1 + 2 x + x^2
```

Of course, interactivity is lost in static presentation formats (e.g., paper, most ebook formats, and most PDF files). However, one can distribute interactive content to anyone who has access to the free CDF player (or to the Wolfram Cloud). If a `Manipulate` object relies on defined symbols, we must use the `SaveDefinitions` option to ensure interactivity in the CDF player. Here is an example that uses a function definition. (We discuss function definition later in this chapter.)

```
ClearAll[f, x, n]
f = Function[{x, n}, (1 + x)^n];
Manipulate[Expand[f[x, n]], {{n, 2}, 0, 5, 1}, SaveDefinitions → True]
```



```
1 + 2 x + x^2
```

1.3.5 Information About Symbols

Mathematica's basic help facility is provided by its `Information` command, which provides basic information for any symbol. The help usually begins with a basic usage message and then provides some details about the symbol, including any options. For example, suppose we define the symbol `a` as follows:

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

Then evaluating `Information[a]` will print basic information about `a`, and we will learn that `a` is a global symbol that has been set equal `b+c`.

We often want information about WL commands. For convenience, we can prepend two question marks to the symbol and evaluate the resulting expression. For example, to get the available information for the `Information` command, we enter `??Information`. We will learn what the command does, along with some detailed information about its attributes, its options, and the default values for its options. In particular, we learn that `Information` has a `LongForm` option with a value of `True`.

The usage message is often all we need; this is available as an `Information` option by specifying `LongForm→False`. As shorthand, specify that we do not want the long-form information by using a single question mark instead of two. For example, evaluating `?Information` will print the short-form information for `Information`.

Advanced

Own Values

Any new name we introduce is becomes a WL symbol. We can see this with the `Head` command. A symbol with no associated values simply evaluates to itself.

Expression	Result
a	a
Head[a]	Symbol

When we assign a value to a symbol by defining a “rewrite rule” for the symbol. This rule is now associated with the symbol, and it will be applied whenever the symbol is evaluated. We might loosely say that this rule is the value owned by the symbol. We can use the `OwnValues` command to see exactly what rules define the symbol in this way.

Expression	Result
a=b+c	b + c
OwnValues[a]	{HoldPattern[a] :→ b + c}
OwnValues[b]	{}

We see that assignment has created a rule associated with the symbol. The notation `:→` means that application of the rule is delayed until the symbol is encountered during the evaluation of an expression. (See the documentation for `RuleDelayed` for more detail.) Since we did not assign to `b`, `OwnValues[b]` returns an empty list of rules.

When we unset a variable, this removes its own values.

Expression	Result
a=5	5
OwnValues[a]	{HoldPattern[a] :→ 5}
a=.	
OwnValues[a]	{}

For this application, `Clear` provides essentially the same functionality. However, `Clear` also accepts a sequence of names and clears them all.

Expression	Result
a=1;b=2	2
OwnValues[a]	{HoldPattern[a] :→ 1}
OwnValues[b]	{HoldPattern[b] :→ 2}
Clear[a,b]	
OwnValues[a]	{}
OwnValues[b]	{}

Advanced

Removing Symbols

Once you introduce a symbol, it persists. Even when we `Unset` (or `Clear`) a symbol, we do not remove it from the list of recognized symbols. We can see this list by using the `Names` and

`MemberQ` commands. (The `Names` command returns a list of names in the specified context, and the `MemberQ` command tests for elementhood.) If this persistence is for some reason unacceptable, we can `Remove` the symbol. However, be aware that the use of `Remove` has subtle repercussions: it affects all your previous definitions using the removed symbol.

Expression	Result	Comment
`a00=5`	5	set the value of `a00`
`Unset[a00]`		unset `a00`
`MemberQ[Names["Global`*"],"a00"]`	True	still in the list of names
`Remove[a00]`		remove `a00`
`MemberQ[Names["Global`*"],"a00"]`	False	no longer in the list of names

1.4 Writing Mathematics

Often we wish to include inline math in a `Text` cell; notebooks make this easy. We can also produce display formulae or output with nicely formatted math.

1.4.1 Typing Math in Text Cells

We usually create an inline-math cell` 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. You will notice that standard notebook styles apply 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 need to enter special symbols and formats. One simple approach is to pick from the front-end menus `Palettes » Basic Math Assistant`, which gives point and click access to a useful collection of symbols and typesetting formats. For faster entry, learn the keyboard shortcuts. In the Basic Math Assistant, if you hover your mouse over a special format, the keyboard shortcut for that format will appear. We can use the escape (ESC) key to delimit keyboard shortcuts (called aliases) to special symbols. You can also use full names for the symbols, surrounded by brack-ets, and preceded by a backslash. So we can enter an α either as `ESCaESC` or as `\[Alpha]`, and we can enter $\int$ as either `ESCintESC` or as `\[Integral]`.

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 equivalently, `CTRL+^` on an American keyboard) and `CTRL+-` (or equivalently, `CTRL+_` on an American keyboard) bring up the superscript and subscript templates. Now you can easily type expressions such as $y = \alpha_1 x^2$ in your `Text` cells. What about expressions like x_1^2, which have both a superscript and a subscript? You can start with a subscript, as above, and the press `ESC+5` to insert a superscript immediately above it.

We can also easily use LaTeX while we are typing text. If you know the LaTeX macro for the symbol you want, just enter it as an escaped alias. For example, just type `ESC\alphaESC` to enter an α. For more details, see the Wolfram Language Tutorial entitled Named Characters. Other useful online resources for math entry include the Wolfram Language Guide entitled Greek Letters, the Wolfram Language Guide entitled Special Characters, and the Wolfram Language Tutorial entitled Keyboard Shortcut

Listing.

1.4.2 Display Formula

Sometime we do not want to enter mathematics as part of a text paragraph; we would rather have it as a separate display. For this purpose, standard notebook styles include the `DisplayFormula` and `DisplayFormulaNumbered` cell styles.

We can create a list as braces-enclosed, comma-separated elements. We represent a matrix as a rectangular list of lists: every row (i.e., inner list) has the same number of elements. Usually you will want to type a matrix formula directly into a cell having the `DisplayFormula` style. Here is one way. Begin by entering your matrix equation and changing the cell style to `DisplayFormula`.

$\{\{a_{11}, a_{12}\}, \{a_{21}, a_{22}\}\} \{\{x_1\}, \{x_2\}\} == \{\{b_1\}, \{b_2\}\}$

Next, press `CTRL+SHIFT+t` (or pick `Cell > Convert To > TraditionalForm Display`) to change it to a traditional display style for mathematics.

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

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

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

Recall that when we plan 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 evaluation will display in a new cell, which is given an `Output` style. Sometimes we would like to display both the original expression and the result of its evaluation. We can use the `HoldForm` command to do this.

```
HoldForm[Sqrt[2.0]] == Sqrt[2.0]
```

$\sqrt{2.} == 1.41421$

The use of `HoldForm` 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. Here is a more interesting example from the Wolfram documentation.

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

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

It is easy to imagine wanting to include such an equation as display math in a document, using the `DisplayFormula` or `DisplayFormulaNumbered` cell style. These are where you will usually type stand-alone equations and expressions involving two-dimensional structures, such as matrices. One approach is to copy the equation we just produced into a new cell, which we give the `DisplayFormula` style. Perhaps a nicer approach is to use `CellPrint` with `ExpressionCell` to produce a `DisplayFormula` cell for us.

```
CellPrint@ExpressionCell[
  HoldForm[Integrate[x^2 E^-x^2, x]] == Integrate[x^2 E^-x^2, x],
  "DisplayFormula"]
```

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

1.4.3 Entering Expressions as LaTeX

This discussion is relevant only if you are familiar with LATEX. In that case, you may at times find it more convenient to enter math symbols using LATEX notation. This is easy: standard LATEX symbols can be entered without change, except that they must be escaped. For example, if you enter `ESC\alphaESC`, it will appear as α in your text.

One may also convert entire expressions. One way in an `Input` cell is to use the command `ToExpression["inputstring",TeXForm]`, where any backslashes in the input string must be doubled. Provide a third argument of `HoldForm` if you just want to typeset the expression. For example,

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

$$\sqrt{2.} == 1.41421$$

Since LATEX notation is not always unambiguous, you may have to experiment to learn conversion quirks. Consider, the following.

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

$$\int_0^3 2 x \, dx == 9$$

Note that we forced a space before the `dx`; without that, the evaluation will fail. If you run into such parsing difficulties, you gain useful hints by looking at the LATEX that WL produces for related expressions. Use the `TeXForm` command for this. For example, you could discover the need for a forced space before the variable of integration by examining the output of an expression like the following.

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

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

1.5 Drawing with Graphics Objects

Mathematica notebooks allow interactive drawing with standard GUI tools. More importantly, WL provides a powerful drawing language.

1.5.1 Interactive Drawing

To use the interactive drawing tools, just press `CTRL+1` to create a new empty graphic. (Or pick from the front-end menus: `Insert>Picture>NewGraphic`.) 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. The interface should be familiar from other interactive drawing applications: 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 Wolfram Language Guide entitled Graphics Interactivity & Drawing and the Wolfram Language Tutorial entitled Drawing Tools.)

For ease of modification and ready replication, it is better to use a drawing language. However, you can start with the drawing tools, and then use the `FullForm` command or the `InputForm` command to access the underlying drawing code.

1.5.2 Drawing Language

For technical drawings, use of a drawing language is often more useful than interactive drawing. A drawing language can be more precise and more replicable. WL includes a powerful drawing language. For two-dimensional graphics, we use the `Graphics` command to create a graphics object from a list of graphics primitives, such as `Point`, `Line`, `Arrow`, and `Text`. In this context, a point is produced from a list of two coordinates, while a line or arrow is produced from a list of two or more points.

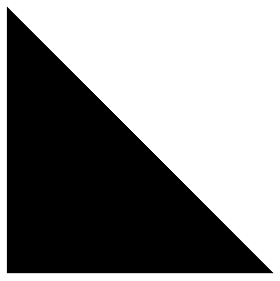
See the Wolfram Language Guide on Graphics Objects for a list of graphics objects. See the Wolfram Language Tutorial on Graphics Directives and Options for a discussion of graphics directives and options. The following examples assume that you have examined this documentation.

Basic needs in drawing include polygons, filled polygons, lines, arrows, points, and text. As a first simple example, we create a `Graphics` object that combines these components. Let us start by creating a triangle.

```
(* represent 2D locations with two-element lists *)
p1 = {0, 0}; p2 = {1, 0}; p3 = {0, 1}; p4 = {0.3, 0.3};
t1 = Triangle[{p1, p2, p3}];
```

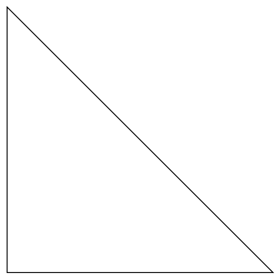
We have assigned to `t1` our description of a triangle. To produce a visual display of this description, we apply the `Graphics` command.

```
Graphics[t1, ImageSize → 144]
```



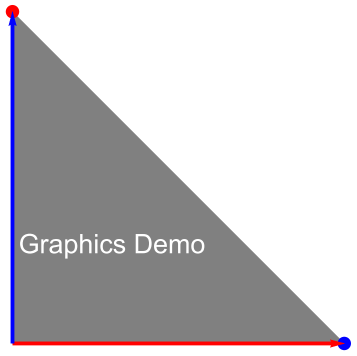
Here we also use the `ImageSize` option to set the image width to 144 points, which is equivalent to 2 inches. This option accepts some convenient predefined values, including `Tiny`, `Small`, and `Medium`. When we provide an integer, it is interpreted as the number PostScript points (72 per inch), at least when exporting to printable formats such as PDF. Note that `Graphics` displays `Triangle` as filled triangle. If we do not want the it to be filled, we need to specify `FaceForm[None]`, and if we want the edge to show, we need to specify the `EdgeForm`.

```
pts = {p1, p2, p3};
Graphics[{FaceForm[None], EdgeForm[Black], Triangle[pts]},
 ImageSize → 144]
```



We are ready for a more complex example, which illustrates how to set properties (such as color, size, or thickness) of graphics primitives since as `Point`, `Arrow`, or `Text`.   It is very lightly commented, since the syntax is simple and intuitive.

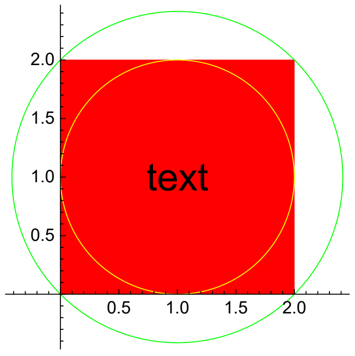
```
(* create a `Graphics` object from a list of graphics primitives *)
g1 = Graphics[
  {
   {Gray, Polygon[{p1, p2, p3}]}, (* fill a gray triangle *)
   {PointSize[Large], Red, Point[p3]},
   {PointSize[Large], Blue, Point[p2]},
   {Thick, Red, Arrow[{p1, p2}]},
   {Thick, Blue, Arrow[{p1, p3}]},
   {White, Text[Style["Graphics Demo", 14], p4]}
  },
  ImageSize → 180]
```



Note that `Polygon` displays as a filled polygon; we can change this as before by specifying the `Edgeform` and `FaceForm`. Since we are creating a triangle, we could equally well have used the `Triangle` command.

Axes can be added to a drawing as an option. WL provides some standard shapes (including rectangles and circles) as graphics primitives. The `Rectangle` command creates a rectangle; the two arguments are the coordinates of opposite corners. The `Circle` command creates a circle; the two arguments are the center point and the radius. See the documentation for details.

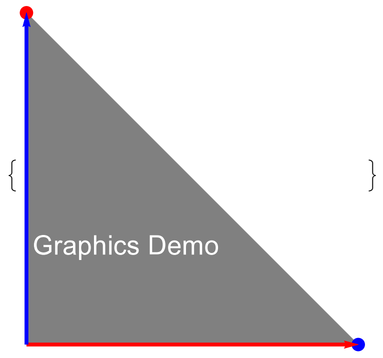
```
g2 = Graphics[{
    {Red, Rectangle[{0, 0}, {2, 2}]},
    {Yellow, Circle[{1, 1}, 1]},
    {Green, Circle[{1, 1}, Sqrt[2]]},
    Text[Style["text", 20], {1, 1}]
  }, Axes → True, ImageSize → 180]
```



Exporting and Importing Graphics

We can export and import graphics in many different formats. By default, the format is indicated by the file extension. Note that the PDF format supports multiple pages, so an imported PDF is given as a list (even when there is a single figure). Lists are discussed in the next chapter.

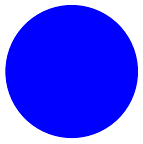
```
Export["/temp/temp.pdf", g1] ; (* WARNING: this will overwrite temp.pdf *)
Import["/temp/temp.pdf"]
```



1.5.3 Modifying and Combining Graphics Objects

Often we create a version of a drawing that we want to modify. We can accomplish this with the `Show` command. For example, suppose we wish to change the image size of a graphics object.

```
blueDisk = Graphics[{Blue, Disk[{0, 0}, 1]}, ImageSize → 180];
blueDisk02 = Show[blueDisk, ImageSize → 72]
```



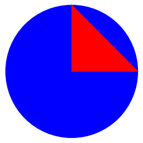
The `Show` command accomplishes this by prepending the new option value. This is a crucial concept: for any option, WL uses only the first value encountered. We can see this effect by using the `InputForm` command to view the representation in code of our graphics object.

```
InputForm[blueDisk02]
```

```
Graphics[{RGBColor[0, 0, 1], Disk[{0, 0}, 1]}, {ImageSize -> 72, ImageSize -> 180}]
```

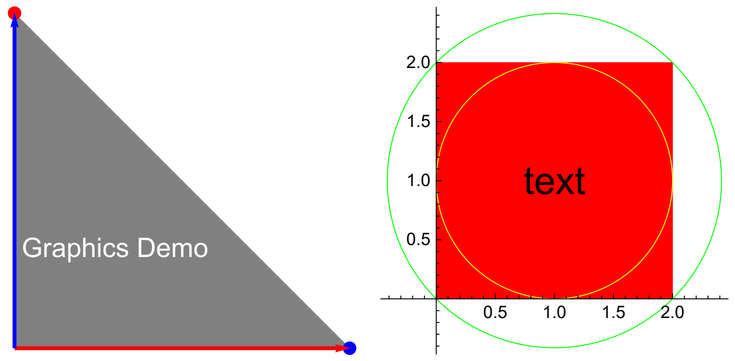
We can also use `Show` when we have constructed more than graphics objects and we want to combine them. To accomplish this, provide a list of the objects as the argument to `Show`. Object will be drawn in the same order that they are listed. Note that any options for the combined graphic will have the value first encountered, which gives the first object in your list considerable control over the display of the combined graphics.

```
redTriangle = Graphics[{Red, Triangle[{{0, 0}, {1, 0}, {0, 1}}]}];
Show[{blueDisk02, redTriangle}]
```



Graphics objects can also be combined in various grids. For example, we can use `GraphicsRow`, `GraphicsColumn`, and `GraphicsGrid` to make arrays of graphics.

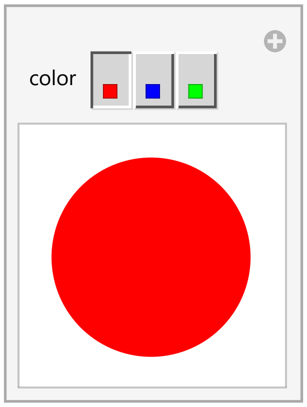
```
GraphicsRow[{g1, g2}]
```



Manipulating Graphics Objects

Our graphics objects are produced by normal WL expressions. So like any expression, graphics objects can also be dynamically manipulated. Here is a simple example, where the color of a disk can be chosen interactively.

```
Manipulate[
 Graphics[{color, Disk[]}, ImageSize → 108],
 {color, {Red, Blue, Green}}]
```

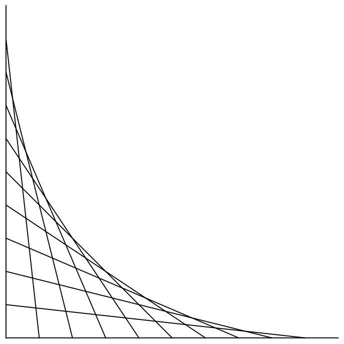


Advanced

1.5.4 Mixing Programming and Drawing

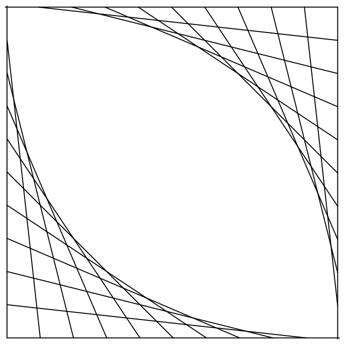
A mixture of drawing and programming can be very powerful. Here we provide a very simple illustration. Looking ahead to our discussion of lists, we use `Array` to create a list of pairs of pairs of numbers, where each pair of numbers represents a point. (We discuss the `Array` command later, in the section on list creation.) 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 two-dimensional result can be drawn by the `Graphics` command.

```
pointPairs = Array[x ↦ {{0, 10 - x}, {x, 0}}, 11, 0];
lines = Line[pointPairs];
g3 = Graphics[lines, ImageSize → Small]
```



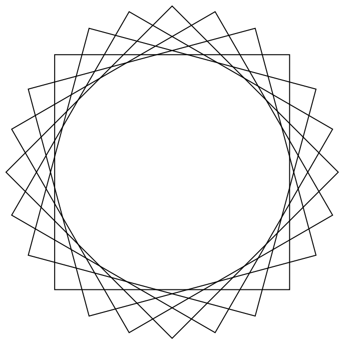
Graphics objects can be reused. For example, let us use the `Rotate` command to rotate our previous picture, and then use the `Show` command to combine two graphics objects into a single object. (For details, see the documentation for these commands.)

```
g4 = Graphics[Rotate[lines, π]];
Show[{g3, g4}, ImageSize → Small]
```



By default, the `Rotate` command rotates an object around the center of its bounding box. However, we can add a third argument, specifying a point around which to do our rotation. Here for example, we rotate the unit square around its midpoint through a range of angles. (The notation used here explained soon, in our discussion of the fundamentals of the WL language.)

```
rect = {EdgeForm[Black], FaceForm[None], Rectangle[{0, 0}, {1, 1}]};
rotater = angle ↦ Rotate[rect, angle, {0.5, 0.5}];
Graphics[rotater /@ (Range[6] * π / 12), ImageSize → Small]
```



Chapter 2: Language Fundamentals

2.1 Functions: Some Basic Considerations

A mathematical function pairs each element of one set (the domain) with a single element of another set (the codomain). This is a very precise and rather general concept, but we can loosely say that a function transforms inputs into outputs. Computationally, the word “function” is often used much more broadly, but in this section our computational functions will resemble mathematical functions: they will specify a particular transformation of an “input argument” into a new value that the function returns as its “output argument”.

2.1.1 Ordinary Function Definition

Use the `Function` command to define an ordinary function. Suppose for example that we want to create a function that returns x^2 for any numerical input x. We can create this in a very natural way.

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

Here `f1` is the name we arbitrarily chose for our function. The first argument to `Function` names our function parameter; this is the name we will use to represent any actual argument that we provide to our `f1` function. The second argument to `Function` is the function body; this expression specifies how the value returned by the function is related to the input argument.

We use the `Function` command to define a function. A function is an expression like any other, so we can associate a name to it with the `Set` command (here, by using the equals sign operator). Above, we set the name `f1` on the left to the function defined on the right. Once we assign a function to a name, we can use that name whenever we want to apply the function to inputs. We apply a our new function to an actual argument just as with builtin commands.

The basic syntax for function application uses brackets: the function is followed by brackets containing any input arguments (i.e., the expression to be transformed by the function). An input value is usually called an input argument. (Some authors call it the actual argument or even the actual parameter.) The core notation for function application consists of brackets, but prefix (atmark) and postfix (double-slash) notations are also available.

Expression	Result	Comment
f1[2]	4	standard notation
f1@3	9	prefix notation
4//f1	16	postfix notation

```
{{f1[2], 4, standard function application},
 {f1@3, 9, prefix notation}, {4//f1, 16, postfix notation}}
```

When we defined our function, we introduced a name (x) that we can use to refer to the function argument. This common practice should feel familiar from basic algebra: we often create names for the inputs our functions will receive. The names used in a function definition are usually called *parameters* or sometimes *formal parameters*. We will occasionally call them *argument names*. We use these names to conveniently specify the input-to-output transformation that takes place when we apply our function to an actual argument. The `Function` command ensures that the formal parameters (i.e., argument names) are names that are local to the function definition. This means that you do not need to worry about conflicts with any global variable having the "same" name.

WL commands naturally have the same flexible notation for application.

Expression	Result	Comment
`Sin[π]`	0	standard
`Sin@π`	0	prefix
`π//Sin`	0	postfix

Prefix and postfix notations sometimes add readability, especially when a sequence of operations is chained. For example, consider the following three equivalent representations of the sequential application of multiple functions. The first represents nested function application, which is also the standard form. The nesting arguably makes it hardest to read. The second uses postfix notation. The last uses prefix notation and is arguably easiest to read, although tastes may vary.

Expression	Result
`f[g[h[x]]]`	`f[g[h[x]]]`
`x//h//g//f`	`f[g[h[x]]]`
`f@g@h@x`	`f[g[h[x]]]`

A function of one variable is called unary. A function of two variables is called binary. A function of three variables is called ternary. More generally, we can speak of an *n*-ary function. We can use the `Function` command to specify functions in any number of variables; the first argument is then a list of formal parameters. If you want to apply a function to a list of arguments, you can use `Apply` or its double-atmark shorthand.

Expression	Result	Comment
`f2=Function[{x,y},x-y]`	`Function[{x, y}, x - y]`	function definition
`f2[3,5]`	-2	ordinary function application
`Apply[f2,{3,5}]`	-2	apply function to argument list
`f2@@{3,5}`	-2	double atmark shorthand

We occasionally encounter one additional notation for function application. For binary functions (i.e., functions that expect two arguments), we can use an alternative tilde-bracketed infix notation (although we will rarely do so). For example, the `Mod` command produces the remainder the from integer division. We can use the command in the usual way, or use the tilde-bracketed infix notation.

Expression	Result
`Mod[5,2]`	1
`5~Mod~2`	1

In the examples above, we set out to write functions to transform numbers, but they also behave sensibly with symbolic arguments. This ability feels natural from a mathematical standpoint, but it is a radical extension of the abilities of numerical languages.

Expression	Result	Comment
f1[2]	4	numerical computation
f1[a]	a^2	symbolic computation
f2[3,5]	-2	numerical computation
f2[a,b]	a - b	symbolic computation

Anonymous Functions

In our first two examples of function definition, we bound the names `f1` and `f2` to our functions so that we could easily reuse them. But these functions are usable even if we do not name them. Here is a simple demonstration:

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

```
4
```

Directly using anonymous functions is particularly useful whenever a function will only be used once, so that naming it is pointless. Later in this book, we will see many examples of the use of anonymous functions.

It is customary to refer to WL functions defined with `Function` as “pure” as a way to emphasize that they can be anonymous: we do not need to associate a name with them in order to use them. This customary usage use of the term ‘pure’ to describe such functions has the potential to sow confusion, since in computer science a function is called pure only if it has no side effects or behavioral dependencies. Real functional purity ensures a completely deterministic relationship between the function’s inputs and its output, in close analogy to a mathematical function. The functions defined above are also pure in this sense, and this is the sense in which we will use the term ‘pure function’ in this book. Breaking with customary WL usage, we use the term ‘ordinary function’ for a function defined with the `Function` command (or its shorthand notations). This captures their close resemblance to ordinary function definition in many other languages.

Advanced

Nevertheless, ordinary functions have some quirks from the perspective of other languages. For example, functions defined with the `Function` command will simply ignore any unneeded arguments. As a concrete illustration, a function of one variable will operate only on the first of a larger sequence of variables.

```
Function[x, x^2][1, 2, 3]
```

```
1
```

For the `Function` command, WL provides a convenient shorthand using the “mapsto” symbol familiar from mathematical notation. (You can produce the “mapsto” operator as `ESCfnESC`.) As before, we can apply an anonymous function to a value.

Expression	Result
$x \longmapsto x^2$	Function[x, x^2]
($x \longmapsto x^2$)[2]	4

In our function definitions up to now, we have introduced explicit names for the formal parameters. These names are local to the function definition, so this is generally harmless and can serve as a readability aid. But we are not required to supply names: each parameter has a slot number, which has a default label. Rather than name the formal parameters, we can instead refer to them by their default labels: `#1`, `#2`, etc. (Additionally, `#` is equivalent to `#1`.)

Expression	Result	Comment
`Function[#1^2]` → Function[$\#^2$]	$\#1^2$ &	one parameter
Function[$\#1^2$+$\#2^2$]	$\#1^2 + \#2^2$ &	two parameters

Let us apply that last function to two arguments.

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

```
5
```

Notice that the display of these function definitions is in an alternative notation, using a postfixed ampersand as a shorthand for the `Function` command. This representation of an anonymous function definition with hash marks (also called octothorpes) and an ampersand is called the short form. We may use this short-form notation to define our functions, and it is a common idiom in WL. It is somewhat more compressed, but it can be a little less readable. It is generally worth asking whether the gain from the compressed syntax offsets the cost of reduced readability, especially if you will share your code. Nevertheless, in this book we will often find it to be a useful and expressive idiom.

Advanced

Slots

The full name of `#n` is `Slot[n]`. So we could write our previous function with more effort as follows.

```
Slot[1] + Slot[2]^2 &;
```

WL additionally provides the `SlotSequence` command, which represents the entire sequence of arguments provided to an ordinary function. We usually use a double hash mark shorthand for `SlotSequence[]`. (For additional details,see the documentation.) For example, the following function multiplies all its arguments and adds the product to the sum of all the arguments.

```
Plus[##, Times[##]] &;
```

Local Variables

To improve the readability of our function definitions, we often want to declare local variables: variables that have meaning only inside the function body. We can use the `Module` command to do so. The `Module` command takes as its first argument a list of variable names, 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.

As a simple example, the following function definition uses `Module` to introduce to local variables in the function body. Notice how a comma separates the two arguments to `Module`, while semicolons are used to separate the three expressions that constitute the second argument.

```
x ↦ Module[{s, c},  (* declare auxiliary variables *)
   s = Sin[x]; c = Cos[x];  (* initialize aux vars *)
   s^2 + c^2  (* value of the module *)
  ];
```

We apply this function in the usual fashion.

```
%[Pi / 2]
```

```
1
```

We will refer to these “extra” local variables as auxiliary variables. Here, their only purpose is to enhance readability. The `Module` additionally allows the local variables to be initialized when declared, which can be even more readable. The following is an equivalent (and slightly cleaner) definition of the previous function.

```
x ↦ Module[{s = Sin[x], c = Cos[x]},  (* declare and initialize aux vars *)
   s^2 + c^2 (* value of the module *)
  ];
```

However, once we make this change, we see that we are no longer resetting the value of any of our auxiliary variables in the body of the module. Effectively, these are constants in the body of the module. In this case, we can use `With` (which simply replaces the specified symbols in its body) instead of `Module` (which introduces local variables). We will refer to the symbols introduced in a `With` command as auxiliary constants.

```
x ↦ With[{s = Sin[x], c = Cos[x]},  (* define auxiliary constants *)
   s^2 + c^2 (* value of the module *)
  ];
```

Advanced

2.1.2 Functions: Some Details

In this section we attend to a few nuances in the definitions of ordinary functions, including the use of compiled functions. This material is not used elsewhere in this chapter; you may skip it on a first reading and return to it as needed.

Partial Function Application

We partially apply a function when we supply some but not all of its arguments. For example, suppose we begin with a pure function of two variables.

```
f2 = {x, y} ↦ x^2 + y^2;
```

If we are given the value of one of these variables, then we are left with a function of one variable. Suppose we want to fix the first argument (in this case, identified by the variable *x*) at a constant value of 1.5. We can do that as follows.

```
f21 = f2[1.5, #] &;
```

The result is an ordinary function of a single variable. We can use it as usual.

```
f21[2]
6.25
```

Alternatively, we could decide we need our original function to have its second argument (in this case, *y*) fixed at 2.5.

```
f22 = f2[#, 2.5] &;
f22[2]
10.25
```

While the short-form notation for function definition sometimes hurts readability, in the case of partial function application it generally proves very readable. Its use in this setting is recommended. Nevertheless, it is of course possible to perform partial function application with other syntax. For example, we could have instead used the equivalent definition

```
f22 = x ↦ f2[x, 2.5];
f22[2]
10.25
```

In this case, the readability of the definition remains perfectly acceptable, so the choice is entirely a matter of taste. That said, it is customary to perform partial function application by using the short form.

The `Function` command has the `HoldAll` attribute. This means that its arguments are not evaluated when the function is created. To see an implication of this, we can look at the own values of our partially applied function `f21`.

```
OwnValues[f21]
{HoldPattern[f21] :→ (f2[1.5, #1] &)}
```

The important thing to note is that this definition explicitly involves the global name of the function `f2`. This means that if we change `f2`, then `f1` will also change. If we consider this kind of dependency to be considered undesirable, we can avoid it by using the `With` command in our definition to force the replacement of the symbol `f2` by its definition in the definition of `f21`. For example,

```
f21 = With[{f = f2}, f[#, 1.5] &];
```

Now if we look at the own values of `f21`, we find no use of the global symbol `f2`.

```
OwnValues[f21]
{HoldPattern[f21] :→ (Function[{x, y}, x^2 + y^2][#1, 1.5] &)}
```

Now, if we change the assignment to `f2` somewhere in our code, the function `f21` will be unaffected.

Piecewise Functions

Sometime we want to define a function that behaves differently on different parts of its domain. In the simplest case, we can use the `If` command. This command takes three arguments: a boolean condition, an expression to return if the condition evaluates to `True`, and an expression to return if the condition is `False`. For example, we can produce a function whose value is 1 in the interval [1 .. 2) and −1 everywhere else.

```
Function[x, If[1 ≤ x < 2, 1, -1]];
```

As another example, we can produce a continuous function from sin[x]/x by plugging the “hole” at $x = 0$ using an `If` command.

```
x ↦ If[x == 0, 1, Sin[x] / x];
```

If the function definition calls for multiple conditions, we can use the `Piecewise` command. The first argument to this command is a list that pairs expressions with the condition under which they apply, and `Piecewise` evaluates the first expression whose condition evaluates to `True`. We can also supply a second argument, which is a default value. As a first example, let us recreate our step function from above.

```
x ↦ Piecewise[{{1, 1 ≤ x < 2}}, -1]
```

$$\text{Function}\left[x, \begin{cases} 1 & 1 \le x < 2 \\ -1 & \text{True} \end{cases}\right]$$

Here we display WL’s evaluation of our function in order to discuss the meaning of `True` in the resulting expression. Obviously `True` always evaluates to `True`, so this determines the value that applies if no previous condition evaluates to `True`. This display also provides a nice match to a common mathematical notation. (In fact, we are allowed to enter our values and conditions using this notation; see the documentation of `Piecewise` for details.)

Here is a slightly more complicated example of using a piecewise definition for a function. This computes the marginal federal income-tax rates in the United States for the year 2016, as a function of income.

```
x ↦ Piecewise[{{0.10, 0 ≤ x < 9275}, {0.15, 9275 ≤ x < 37650}, {0.25, 37650 ≤ x, 91150},
     {0.28, 91150 ≤ x < 190150},
     {0.33, 190150 ≤ x < 413350}, {0.35, 413350 ≤ x < 415050}, {0.396, 415050 ≤ x}
   }];
```

Argument Passing

You will probably find it unsurprising that we cannot assign to a value. This example will not surprise you if you have any programming experience: we do not expect to be able to make an assignment to a literal value. The following fails and generates an error message.

```
0 = 1;
```

```
Set: Cannot assign to raw object 0.
```

Nevertheless, you may find it surprising that we cannot assign to a part of a value. The following fails with a related error message.

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

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

The behavior is potentially surprising, depending on your language experience. (For example, the equivalent operation would not be an error in Python.) We need an unprotected symbol (variable) on the left whenever we are making an assignment. So, in contrast, the following does not produce an 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 such errors is slightly more obscure: if you pass a symbol to a function, it will be evaluated before the function body is executed. (There are ways to work around this, e.g. using `HoldFirst`, which we do not discuss here.)

The default behavior of WL is that a function argument is passed as the value of the supplied expression. If you forget about this, you are likely to produce an error.

For example, contrast the following two lines of code. The first line associates a value with a variable and then increments the value associated with that variable.

```
x = 0; ++x
1
```

The next approach is not equivalent, because the function sees the value, not the variable name. As a result, we produce an error message.

```
(x ↦ ++x)[0];
```

```
PreIncrement: 0 is not a variable with a value, so its value cannot be changed.
```

Effectively, the function simply substitutes the values passes into the function body, using the rule established by the function definition. Essentially, all occurrences of the formal parameter are replaced with the passed value before the function is evaluated. (Using the terminology of [maeder-2006-cup]_, we might say that the formal parameters are pattern variables but not program variables.) As we have already seen, a value cannot be changed; we can only change what value a symbol is associated with. The bottom line is that your function cannot make any kind of assignment to your function parameters.

Here is one more related example. If you have understood the rest of this section, you should understand why this function produces the error it does.

```
Clear[x, s]
f = Function[x, x[[1]] = 1; x];
s = {0};
f[s]; (* error, since the value of s replaces x before function is executed *)
```

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

If we really need to manipulate a value in this way, we will have to assign it to a name within the function. For this, we can use local variables, which we introduce with the `Module` command.

Recursion

A function is recursive if it refers to itself in its definition. WL supports recursive functions. For example, here is a simple definition of a recursive computation of a factorial. (For now, we simply assume the input is a nonnegative integer.) Of course, this is just for exercise, since WL provides `Factorial` as a builtin function; we can even use the exclamation mark, common in mathematics, as a shorthand notation for `Factorial`.

```
f = n ↦ If[n == 0, 1, n * f[n - 1]];
```

One thing awkward about that formulation is that the function definition depends on a global variable: the name we have decided to give this function. This means that we have not actually produced a pure function, and as a result we lose referential transparency. For example, if we let a new symbol refer to this function but then change `f`, our new function will also change.

Expression	Result	Comment
g=f	Function[n, If[n == 0, 1, n f[n - 1]]]	refers to f
f=n↦1	Function[n, 1]	new f
g	Function[n, If[n == 0, 1, n f[n - 1]]]	still refers to f
g[5]	5	uses new f

Recall that `Slot[1]` refers to the first argument of a function. Interestingly, we can use `Slot[0]` to denote the function itself. If we are willing to do without naming our formal parameters, we can have a more elegant solution. This allows simple short-form recursive definitions. For example, here is a simple recursive definition of a factorial function.

```
f2 = If[# == 0, 1, # * #0[# - 1]] &;
```

This gives us more referential transparency, eliminating the previous problem of reliance on a global variable.

Expression	Result	Comment
g=f2	If[#1 == 0, 1, #1 #0[#1 - 1]] &	refers to #0
f2=n↦1	Function[n, 1]	new f2
g	If[#1 == 0, 1, #1 #0[#1 - 1]] &	still refers to #0
g[5]	120	still factorial

In sum, for recursive function definition, the use of slots is safest.

Inverse Functions

Suppose we have a strictly monotonic real-valued function of a real variable, $f : X \to \mathbb{R}$ where $X \subseteq \mathbb{R}$. Let $Y = f[X]$. Then there is an inverse function $f^{-1} : Y \to X$. WL allows us to provides the `InverseFunction` command to produce an inverse function.

We can apply `InverseFunction` to a raw symbol, but it is not very informative.

```
ClearAll[f]
InverseFunction[f]
```

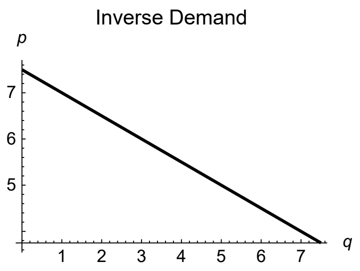
$f^{(-1)}$

Economists are often interested in the inverse demand function. Suppose we have quantity demanded as a downward sloping function of price. We most commonly produce the inverse function by setting quantity equal to the quantity demand and solving for price. This gives us price as a function of quantity, which is how economists often draw their demand curves. For example,

```
Clear[p, q]
soln = Solve[q == 15 - 2 p, p]
inverseDemand = p /. First[soln]
Plot[inverseDemand, {q, 0, 15/2}, PlotLabel → "Inverse Demand",
 AxesLabel → {q, p}, ImageSize → Small]
```

$\left\{\left\{p \to \frac{15-q}{2}\right\}\right\}$

$\frac{15-q}{2}$



This is a perfectly good approach, but we can alternatively use `InverseFunction`. The one caution is that if we define our function with an explicit variable, `InverseFunction` reuses the symbol for the bound variable in the inverse function. This is fine (it does convey any meaning, since it is bound), but it can feel a bit odd. As a result, we may find is more comfortable (however pointless) to replace the bound variable with a more meaningful name.

```
p ↦ 15 - 2 p
InverseFunction[%]
% /. {p → q}
```

Function[p, 15 - 2 p]

Function$\left[p, \frac{15-p}{2}\right]$

Function$\left[q, \frac{15-q}{2}\right]$

Alternatively, we can avoid using named variable altogether, simply relying on slots in our function definitions.

```
InverseFunction[5 - 2 # &]
%[q]
```

$\frac{5-\#1}{2}$ &

$\frac{5-q}{2}$

Function Closures

A closure can “recall” the environment it was created in. For example, functions can return functions, and the returned functions can close over the local variables of the creating function. Consider a function defined as $f[x] = x + n$. We say that n is a free variable: it occurs in the function body even though it is not in the signature and is not introduced as a local variable. That is, this function references a variable that is not local to the function definition. This function definition is incomplete until we specify the value of the variable n.

However, suppose we define this `plusn` in an environment where n is already defined and allow the returned function to refer to this pre-existing environment whenever it is called. The result is a closed expression, or closure. (Support for closures is language specific.) We also say that the returned function closes over n, meaning that n is actually not free but rather retains the value that it held in the enclosing environment.

Languages that support closures allow us to use functions to generate other functions. For example, we can define the following function `plusn` in terms of an inner function with free variable `n`.

```
ClearAll[plusn]
plusn = n ↦ (x ↦ x + n);
```

We can now use `plusn` to generated various closures; each closure carries with it the value taken by `n` at the time of creation. Let us create the following two closures and look at what we created.

```
plus2 = plusn[2]
plus3 = plusn[3]
```

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

Function[x$, x$ + 3]
```

Note that n has been replace by the value we specified when producing these closures; it is not a free variable. Also, 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. We see that `plus2` "recalls" that, when it was created, n had the value 2. Similarly, `plus3` "recalls" that, when it was created, n had the value 3.

```
plus2[5]
plus3[5]
```

```
7

8
```

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

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

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

7
```

We see that `plus2` "recalls" that, when it was created, n had the value 2. It carries around the environment that existed when it was created (the closure). We say that it closes over n.

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

Compiled Functions

When we define a normal function, it works equally well with numeric and symbolic arguments.

Expression	Result
`f=Function[{x},x^2]`	Function[{x}, x^2]
`f[2]`	4
`f[x]`	x^2

If we know that we will only call our function with numeric arguments, we may get a substantial speedup by creating a compiled function. The basic syntax for a compiled function is parallel to the basic syntax for a normal function, except that we use the `Compile` command instead of the `Function` command.

```
ClearAll[f];
f = Compile[{x}, x^2]
```

```
CompiledFunction[ WVM  Argument count: 1
                       Argument types: {_Real} ]
```

Note that the result is a compiled function. Function arguments are typed: they are `_Real` by default, and arguments are coerced to the specified type (if possible). We can now use our new function.

```
f[2.]
```

```
4.
```

Our compiled function is not intended work with symbolic values. However, if we misuse it by providing a symbolic argument, it provides a fallback behavior of raising a warning and returning a value

```
f[x]
```

```
CompiledFunction: Argument x at position 1 should be a machine-size real number.
```

x^2

The compiled function specifies an argument type. In this case, the type is `_Real`, which means it is an ordinary floating point number. If we want a different type, we can specify it.

```
ClearAll[f]
f = Compile[{{x, _Integer}}, x^2]
```

```
CompiledFunction[ WVM  Argument count: 1
                       Argument types: {_Integer} ]
```

For more information about compiled functions, see the Wolfram Language Tutorial entitled Compiling Wolfram Language Expressions. This is a powerful facility when needed.

A compiled functions can accept a list or a matrix (or even a list of matrices) as an argument, as long as all the elements are of a single numerical type. To produce this behavior, we have to specify the number of dimensions of the argument. Here is an example where we specify the input to be a one-dimensional integer list.

```
xsq03 = Compile[{{x, _Integer, 1}}, x^2];
xsq03[{1, 2}]
```

```
{1, 4}
```

Here is an example where we specify the input to be a two-dimensional real matrix.

```
xsq04 = Compile[{{x, _Real, 2}}, x^2];
xsq04[{{1, 2}, {3, 4}}]
```

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

Attributes and Messages

Advanced

We can add using information to a symbol with the `MessageName` command (or its double-colon shorthand) to tag a symbol with a message string. If we use the `usage` tag, this string can then be accessed in a notebook via Mathematica's help system (which we discussed above). For example,

```
f1::usage = "f1[x]  returns x*x";
? f1
```

Info2233714241632-5744334

```
f1[x] returns x*x
```

A function may be given attributes, messages, or defaults, and these are not cleared by `Clear`. However, the `ClearAll` command will fully clear the symbol, even of its attributes. (In addition, if you wish to clear the only the attributes, you can use the `ClearAttributes` command.) Here we illustrate these commands by setting and clearing the `Orderless` attribute on a symbol. A function that has the `Orderless` attribute will sort its arguments.

Expression	Result
SetAttributes[f,Orderless]	
f[b,a]	f[a, b]
Clear[f]	
f[b,a]	f[a, b]
ClearAll[f]	
f[b,a]	f[b, a]

Function Composition

Function composition combines two functions into one. The composition $f \circ g$ is equivalent to this: f is applied to the result of applying g, so that $(f \circ g)[x] = f[g[x]]$.The `Composition` command performs function composition, as does its slightly unusual `@*` shorthand.

```
ClearAll[f, g, h, x]
Composition[f, g]
%[x]
```

```
f@*g
```

```
f[g[x]]
```

We usually us the `@*` shorthand for the `Composition` command. We can compose as many functions as we wish:

```
(f@*g@*h)[x]
```

```
f[g[h[x]]]
```

Note that we did not have to specify the order in which the compositions are performed. This is because associativity is a basic law of composition.

```
f@*(g@*h) == (f@*g)@*h
```

```
True
```

As a simple example of function composition, let us create a function that sums the digits of a positive integer. We will use `IntegerDigits` to get the digits, and `Total` to sum them. First, consider a traditional approach to this problem.

```
sumDigits01 = x ↦ Total[IntegerDigits[x]];
sumDigits01[12 345]
```

```
15
```

Now use the `Composition` command to accomplish the same thing. Note how we can define a function without introducing a variable to represent its argument. This style of function definition is called “tacit” or “point-free”.

```
sumDigits02 = Total@*IntegerDigits;
sumDigits02[12 345]
```

```
15
```

In this simple case we do not see a great gain from relying on `Composition` and a point-free style, but it often proves to be a powerful technique.

Anonymous Functions as Lambdas

In this section we use anonymous functions to briefly discuss a few properties of lambda calculus, closely following section 11.1 of [maeder-2000-cup]_. This material will not be required elsewhere in this book and should be skipped by most readers.

The lambda calculus is a calculus of anonymous functions of one variable. It makes use of expressions in bound and unbound variables, functions, and function applications. Perhaps the simplest example is the identity function.

```
(x ↦ x)
```

```
Function[x, x]
```

This expression evaluates to a function (sometimes called a lambda abstraction), which is a rule for creating new expressions by substitution. On the right is an expression, which we may call the body of the function. In this case the function body involves one variable, *x*. On the left is the variable to be bound by the function in the function body, which is also *x* in this case. The variable bound by the function may be called the formal parameter of the function.

An application of the function to an argument produces a substitution of the argument for all occurrences of the formal parameter in the body of the function. (In discussion of lambda calculus, this substitution is called beta reduction.) We will use square brackets to indicate function application.

Expression	Result
`(x⟼x)[0]`	`0`
`(x⟼x)[1]`	`1`

Apparently function definition just produces a rule for how to substitute an argument into a function body. This is true, but it can help us think about this substitution process if we attempt to use the `ReplaceAll` command to do it. The tricky part is that we need to postpone evaluation of the formal parameter and the replacement expression until after after we complete our substitution. For example, recalling the slash-dot shorthand for `ReplaceAll`, the following exposes a problem.

```
x = 1;
2 x /. {x → 0}
```

```
2
```

The problem is that, because we first set a global value for *x*, the occurrences of *x* in our attempted substitution are evaluated before the substitution is attempted. (You can see this explicitly by using the `Trace` command.) We can work around this by using `Hold` and `HoldPattern` to avoid these evaluations. Our use of functions essentially does this for us and then releases the hold.

Expression	Result
`Hold[2x]/.{HoldPattern[x]→0}`	`Hold[2 0]`
`ReleaseHold[Hold[2 0]]`	`0`

Since the formal parameter is bound in a function body, its name becomes irrelevant in other contexts. The name of the formal parameter has meaning only in the context of the function definition. Equivalently, if we consistently substitute any other name, we will produce the same function. (In the lambda calculus, parameter renaming is called alpha conversion, and the resulting functions is said to be alpha-equivalent to the original.) As an example, let us rename the parameter in our identity function. (This is for illustration; we should not expect to be performing such renaming in useful code.)

```
ReleaseHold@Activate[Hold[(x ⟼ x)] /. {HoldPattern[x] → Inactive[y]}]
```

```
Function[y, y]
```

The important thing for us is not that we can do alpha conversion but rather that the choice of formal parameter generally does not matter. That said, it is clear that our substitution would not make sense if the function body already involved our new choice of parameter name. For example, the following clearly is not a proper parameter renaming.

```
ReleaseHold@Activate[Hold[(x ⟼ y)] /. {HoldPattern[x] → Inactive[y]}]
```

```
Function[y, y]
```

This fails because the new name we chose for the function parameter is already a free variable in the function body. As a result, we have converted a function that always returns *y* into an identity function. It is common to say that the free variable was captured by the substitution, and only capture-free substitutions are generally valid. From a WL programming perspective, we can readily avoid capture if our function bodies contain no free variables. A function with no free variables is sometimes said to be closed, and closed functions are sometimes called combinators.

We will briefly consider one more equivalence, sometimes known as extensionality reduction or eta reduction. This recognizes that $x \longmapsto f[x]$ is equivalent to f. For example, the following is just a complicated way to write our identity function.

```
(x ↦ (x ↦ x)[x]);
```

A function is called higher-order when it accepts functions as an argument. The lambda calculus is for functions of one variable, but if we use free variables, we can readily decompose any multivariate function into a sequence of univariate functions. (This decomposition is known as currying.) For example, suppose we want a function that produces the arithmetic mean of two numbers. We write a function that when applied to a number returns a new function that will average that argument with whatever argument is supplied to the new function. Here we build such a function, apply it to 0, and then apply the resulting function to 2 in order to produce the mean of 0 and 2.

```
(x ↦ (y ↦ (x + y) / 2))[0][2]
1
```

While this may be useful for mathematical logic, it quickly becomes notationally cumbersome. When we want multivariate functions, we usually define them directly rather than currying them.

2.1.3 Pattern-Based Function Definition

Suppose we want to define a function that squares its input. We have seen that we can define such a function as follows.

```
ClearAll[f01, f02, f03]
f01 = x ↦ x^2
Function[x, x^2]
```

Note the output generated by our definition, which is just our function definition, given in terms of the named parameter. We call this an ordinary function definition, while noting that in WL it is often called a pure function definition. Evaluation of our definition produces a function object, as we can verify with the `Head` command.

```
Head[f01]
Function
```

An extremely popular and powerful alternative approach to ordinary function definition is pattern-based function definition. This is usually accomplished by means of a delayed-evaluation assignment and pattern matching. In this approach to function definition, the function name and pattern-tagged formal parameters appear to the left of the delayed evaluation assignment operator `:=`, and the function definition in terms of the formal parameters appears on the right. The expression `x:pattern` represents a pattern (to the right of the colon) that we can refer to by the name `x`. A common pattern is the underscore, called a `Blank`, and it is a pattern that matches anything at all. When our pattern is simply a `Blank`, we are allowed to omit the colon preceding the pattern. This omission is conventional, but we include the colon here for completeness.

```
f02[x : _] := x^2
```

Note that because this definition involves *delayed* evaluation, it simply defines rule for a symbol and does not generate any output. We can see this difference when we look at the `Head` of the of the symbol we chose as our function name.

```
Head[f02]
```

```
Symbol
```

We will refer to this as a pattern-based function definition. A function that we define by explicit pattern matching has certain advantages: it can easily do type checking, enforce preconditions, provide default values, and incorporate optional arguments. As a simple example, here is another squaring function, which only works for integers and provides a default argument of `0`. The pattern `_Integer` only matches integers. The default value follows a second colon.

```
f03[x : _Integer : 0] := x^2
```

Expression	Result	Comment
f03[25]	625	squares an integer input
f03[]	0	provides a default argument
f03[1.5]	f03[1.5]	not defined for non-integer input

Pattern matching is an extremely powerful language facility. We provide a brief discussion in the chapter on Programming. For more details, see the Wolfram Language Tutorial entitled Introduction to Patterns. For a description of basic patterns, see the Wolfram Language Guide entitled Patterns.

Advanced

Down Values

Recall that assignment of a value to a symbol determines the own values of the symbol. However, we are often interested in assigning a value not directly to the symbol but rather indirectly. For example, we would like to find a value associated `f[x]`. The problem is that `f[x]` is not a symbol; is is a composite expression with `f` as its head. WL cannot associate a rewrite rule directly with a composite expression. The solution is do associate it with the head `f`, but to indicate that it applies when `f` is the head of a matching composite expression. We call such rules down values, and we can examine them with the `DownValues` command, which shows us the rewrite rules that will be used when our symbol occurs as the head of a composite expression.

Expression	Result
f[0]=0	0
DownValues[f]	{HoldPattern[f[0]] :→ 0}
f[0]	0
f[1]	f[1]

The replacement of a composite expression by a down value is based on pattern matching. Here we get a match for `f[0]`, so the down value is returned. However, we do not get a match for `f[1]`, so the composite expression is returned unevaluated.

We can create multiple down values by assigning to differing patterns. For example, let us add two more down values to `f`.

```
f[x_] = 99; f[1] = 1;
DownValues[f]
```

```
{HoldPattern[f[0]] :→ 0, HoldPattern[f[1]] :→ 1, HoldPattern[f[x_]] :→ 99}
```

When we look at the downvalues, we see them all. Interestingly, they are not in the order we assigned them. Instead, they are in the order in which they will be applied. WL always applies more specific patterns before less specific patterns. After accounting for specificity, the patterns are applied in the order created. For example,

Expression	Result
f[0]	0
f[y]	99

We can remove down values with `Unset` or `Clear`. There is an important difference: we use `Unset` to remove a single down value, while `Clear` removes them all at one go.

Expression	Result
f[0]=.	
DownValues[f]	{HoldPattern[f[1]] :→ 1, HoldPattern[f[x_]] :→ 99}
f[0]	99

With this background, you can understand that our pattern-based function definitions were making use of down values. For example, we can look at the down values of

```
DownValues[xsq03]
```

```
{}
```

Finally, to address a subtle point, we can ask what happens when a pattern-based function is evaluated. When

2.2 Lists

The list is a fundamental WL data type. We use lists to represent sets, vectors, and matrices. Additionally, nested lists are often used to store tabular data, and we can display such data in nicely formatted tables. This section also briefly discusses associations, which are WL's implementation of an associative array.

2.2.1 Introduction to Lists

A list is just an ordered collection of items. We may create a list by enumeration, by using a list creation command, or by combining or manipulating lists.

Creation by Enumeration

We can create a list by enumerating comma-separated elements within curly braces. The elements can be any expression, including undefined symbols. Note that an enumeration within braces such as `{a,b,c}` is just a convenient shorthand for `List[a,b,c]`. Programmers often use the term ‘syntactic sugar’ for convenient shorthand, and WL offers a great deal of syntactic sugar. To see that these are two different representations of the same expression, we can use the `FullForm` command.

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

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

We refer to the objects contained by the list as the elements of the list. Every list has a length equal to the number of its elements. A list can be empty, it which case it has a length of 0.

```
Length[{}]
```

```
0
```

A sequence of numbers in braces is an example of a list literal. (A “literal” is code that represents a fixed value, such as a number, a string, or even a list or function.) We are often interested in lists whose elements are numbers. For example, later on we will use lists of numbers to represent numeric vectors.

A list can contain any expression as an element, including another list.

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

When lists are nested, the outer list still has a length, as does each of the inner lists. If the nested lists give us a rectangular structure (like a matrix), we can use the `Dimensions` command to produce these dimensions.

Expression	Result
`Length[mA]`	`2`
`Dimensions[mA]`	`{2, 3}`

A rectangular list of lists can represent a matrix. (We discuss this in detail in the next chapter.) We can use the `MatrixForm` command to get a more intuitive visual display of a matrix. (A matrix form is different than the list it contains; it the list plus a display wrapper.)

```
MatrixForm[mA]
```

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

Element Access (Indexing)

Using double brackets, we can access the elements of a list with a unit-based index. (Indexes are called unit-based when they begin at 1.) To index backwards from the end of the list, use negative indexes.

Expression	Result	Comment
`lst01={1,2,3}`	`{1, 2, 3}`	create list
`lst01⟦1⟧`	`1`	access the first element
`lst01⟦-1⟧`	`3`	access the last element

We can also retrieve spans of elements with an index span specified as \`start;;stop\` or \`start;;stop;;step\`. (See the documentation for \`Span\`.)

Expression	Result
`x=Range[10]`	`{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`
`x⟦3;;9⟧`	`{3, 4, 5, 6, 7, 8, 9}`
`x⟦3;;9;;2⟧`	`{3, 5, 7, 9}`

We could even produced a reversed version of a list with this kind of indexing. (However, in practice we would just use the \`Reverse\` command.)

```
x⟦10 ;; 1 ;; -1⟧
```

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

List are also accepted as part specifications. We can can use a list to pick out elements in any order.

```
x⟦{7, 1, 3}⟧
```

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

The index list may repeat indexes as many times as desired.

```
x⟦{1, -1, 1, -1}⟧
```

```
{1, 10, 1, 10}
```

Although most commonly we use the the indexing brackets, they are actually a convenient shorthand for the \`Part\` command. For example, \`lst[[1]]\` is just a shorthand for \`Part[lst,1]\`. Occasionally it can be useful understand the relationship to \`Part\`. You can explore that with the \`FullForm\` command. Here are a few examples. (In these examples, \`lst\` has no parts, so Mathematica will issue a warning. You can use the \`Quiet\` command to silence these warnings.)

Expression	Result
`lst⟦1⟧//FullForm`	`Part[lst, 1]`
`lst⟦{1,2,3}⟧//FullForm`	`Part[lst, List[1, 2, 3]]`
`lst⟦1;;4;;2⟧//FullForm`	`Part[lst, Span[1, 4, 2]]`

For more details about indexing, see the Wolfram documentation for \`Part\`.

Changing List Elements

Lists are mutable. We can use indexing on the left side of an assignment to change the value of an element. We can even index with a list of indexes, which allows us to change multiple elements at one go.

Expression	Result
lst01=Range[3]	{1, 2, 3}
lst01⟦-1⟧=99	99
lst01	{1, 2, 99}
lst01⟦{1,-1}⟧={91,93}	{91, 93}
lst01	{91, 2, 93}

WL effectively copies lists on the right side of an 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 `All` as an index, allowing us to change all elements at one go.

```
lst02 = lst01;
lst01⟦All⟧ = 1;
lst01
lst02
```

```
{1, 1, 1}

{91, 2, 93}
```

We can also replace spans of elements with an index range specified as `start;;stop` or `start;;stop;;step`. (See the documentation for `Span`.)

```
x = Range[15];
x⟦1 ;; 15 ;; 2⟧ = 0;
x
```

```
{0, 2, 0, 4, 0, 6, 0, 8, 0, 10, 0, 12, 0, 14, 0}
```

Other commands for accessing specified elements include `First`, `Rest`, `Most`, `Last`, `Extract`, `Take`, and `Drop`. Read the Wolfram documentation for these, as we will return to them from time to time.

If we want a closer look at what is involved with this assignment with an index, we can again look at the `FullForm`. We find that `x⟦1⟧=99` is equivalent to `Part[x,1]=99`, so the double brackets are again a shorthand for `Part`.

```
HoldForm[FullForm[x⟦1⟧ = 99]]
```

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

Recall that this assignment will mutate an existing list. Often it is better to produce a new list: we can do that by using the `ReplacePart` command (rather than using `Set`). The `ReplacePart` command applies a rule to produce a new list based on the old one, but with appropriate replacements.

Expression	Result	Comment
x=Range[3]	{1, 2, 3}	create a list
xnew=ReplacePart[x,1→99]	{99, 2, 3}	create a new list
x	{1, 2, 3}	confirm old list is unchanged

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 user to make the right trade-offs.

Advanced

Although WL lists are mutable, they must be changed indirectly, using the symbol that refers to the list. (For example, the symbol `lst01` or `x`.) For example, the following produces an error:

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

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

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

2.2.2 List Creation

WL offers an enormous variety of ways to create and combine lists. Here we cover just a few of the most immediately needed facilities.

Constant Arrays, Ranges, and Function Iteration

If you plan to repeatedly enumerate a single value, you will find it faster to use `ConstantArray`. Symbolic constants are possible.

Expression	Result
ConstantArray[0,10]	{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
ConstantArray[a,10]	{a, a, a, a, a, a, a, a, a, a}

Lists are equal if there are equal elementwise. We can use `==` to do an equality comparison of lists. Here we use the `With` command to temporarily (i.e., locally) set `a=0`, so the equality comparison evaluates to `True`, but the any global value for `a` is not affected.

```
With[{a = 0}, Evaluate[ConstantArray[a, 20] == ConstantArray[0, 20]]]
```

```
True
```

Integer intervals are another common need, and we usually create them with `Range`. If we provide a single argument, it sets the maximum value of a positive integer range. There is an implicit minimum value of `1` and stepsize of `1`. If we provide two arguments, the first is taken as an starting value and the second as a maximum, with the stepsize still being `1`. Note that the range is inclusive: it contains the minimum and maximum values. We can also specify a stepsize as a third argument. If the stepsize is negative, then the second argument is reinterpreted as a minimum value. Here are some illustrative examples.

Expression	Result
Range[10]	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Range[-4,4]	{-4, -3, -2, -1, 0, 1, 2, 3, 4}
Range[-1,1,1/2]	$\{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\}$
Range[1,-1,-1/2]	$\{1, \frac{1}{2}, 0, -\frac{1}{2}, -1\}$

A common way to generate lists is by function iteration. We define a univariate function and specify a starting value, and they we repeatedly apply the function. We can use the `NestList` command for this.

```
Clear[f, x0]
NestList[f, x0, 3]
```

```
{x0, f[x0], f[f[x0]], f[f[f[x0]]]}
```

As a more concrete illustration, let us iterate a logistic function.

```
NestList[x ↦ 3.7 * x * (1 - x), 0.5, 5]
```

```
{0.5, 0.925, 0.256687, 0.705956, 0.768053, 0.659146}
```

The sequences generated by function iteration can have very interesting properties. Eventually we will explore some of these.

We have seen that creating lists in WL is trivial, but we should not lose sight of the fact that creating large lists can be costly in computational memory and time. Judicious use of mathematical reasoning can help us avoid unnecessary computations costs. For example, suppose we want to compute the sum of the first million positive integers. One approach to this problem is to create the a list holding the numbers and then find the total. An alternative is to recall that the sum of the first n positive integers is $n(n+1)/2$, which can be trivially computed. If we happen to forget this formula, we may be able to use WL to produce it. In this case, for example, we can try to produce the symbolic sum for an arbitrary positive integer n.

```
Clear[n]
Sum[i, {i, 1, n}]
```

$$\frac{1}{2} n \, (1 + n)$$

We can then use `ReplaceAll` (here, as its `/.` shorthand) to substitute any desired value for n into the formula. This will be much faster and less memory intensive than actually creating the underlying list of numbers.

```
% /. {n → 10^6}
```

```
500 000 500 000
```

Prepending and Appending Elements

Use the `Prepend` or the `Append` command to prepend or append an element to a list. These commands return a new list, without changing the value of the old list. In the following example, we illustrate this by sequentially evaluating the listed expressions.

Expression	Result
row=ConstantArray[0,4]	{0, 0, 0, 0}
Append[row,-1]	{0, 0, 0, 0, -1}
Prepend[row,1]	{1, 0, 0, 0, 0}
row	{0, 0, 0, 0}

When WL offers a facility for changing a list, it usually offers a related facility for producing a new list without mutating the old list. For example, we can use the `AppendTo` command to append an element to a list or the `Append` command to create an equivalent new list. Similarly, we can use the `PrependTo` command to prepend an element to a list or the `Prepend` command to create an equivalent new list.

If you want not to produce a new list but to change an existing list, use `AppendTo` or `PrependTo` instead. Once again, sequentially evaluate the expressions in the following table. This time we find the value of `row` has changed.

Expression	Result
row=ConstantArray[0,4]	{0, 0, 0, 0}
AppendTo[row,-1]	{0, 0, 0, 0, -1}
PrependTo[row, 1]	{1, 0, 0, 0, 0, -1}
row	{1, 0, 0, 0, 0, -1}

Suppose we append one list to another. The result is not a concatenation of the two lists; instead we end up with a list whose last element is in turn a list. To concatenate lists, we must instead use the `Join` command or the `Catenate` command.

Expression	Result
lst01={1,2,3};lst02={4,5,6};	
Append[lst01,lst02]	{1, 2, 3, {4, 5, 6}}
Join[lst01,lst02]	{1, 2, 3, 4, 5, 6}
Catenate[{lst01,lst02}]	{1, 2, 3, 4, 5, 6}

Creating Lists with `Array` and `Table`

We often build lists with the `Array` command, which takes three arguments: a function, the number of elements to produce, and a domain specification. The domain specification is an interval to subdivide ({xmin, xmax} or (xmax, xmin}). When the range is {1, *n*}, where *n* is the number of elements to produce, the domain specification can be omitted. Here are a few examples.

Expression	Result
Array[f,5]	{f[1], f[2], f[3], f[4], f[5]}
Array[i↦0,5]	{0, 0, 0, 0, 0}
Array[i↦i*i,5]	{1, 4, 9, 16, 25}
Array[i↦i*i,5,{1,3}]	$\{1, \frac{9}{4}, 4, \frac{25}{4}, 9\}$
Array[i↦i*i,5,{5,1}]	{25, 16, 9, 4, 1}
Array[i↦i/4,5,{0,4}]	$\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$

We are going to briefly explore a closely related command for list creation. The `Table` command is a fairly general facility for list creation. The first argument of `Table` command is an expression in a variable, and the second argument is a “standard iterator specification” in that variable. An iterator specification is just a list with a special format, which specifies a range of values for a variable. The `Table` command evaluates the first argument at each value in the specified range.The standard iterator specification echoes the signature of `Range`. So the `Table` command can accomplish the same tasks as `Range`, but slightly more verbosely.

```
Range[1, 20, 2] == Table[i, {i, 1, 20, 2}]
```

```
True
```

The iterator specification may take a few standard forms that can be anticipated from our knowledge of the `Range` command. The form with only a positive integer stop value, as in `{i,stop}`, will produce 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}`.

Expression	Result
Table[f[i],{i,5}]	{f[1], f[2], f[3], f[4], f[5]}
Table[0,{i,5}]	{0, 0, 0, 0, 0}
Table[i*i,{i,5}]	{1, 4, 9, 16, 25}
Table[i*i,{i,1,3,1/2}]	$\{1, \frac{9}{4}, 4, \frac{25}{4}, 9\}$
Table[i*i,{i,5,1,-1}]	{25, 16, 9, 4, 1}
Table[i/4,{i,0,4}]	$\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$

Some alternative specifications are allowed as the second argument. If we are not varying the value of the first expression, we can just specify the number of repetitions. Additionally, we can explicitly specify the range of values with a list, as in `{i,lst}`.

Expression	Result
Table[0,5]	{0, 0, 0, 0, 0}
Table[i*i,{i,Range[5,1,-1]}]	{25, 16, 9, 4, 1}

Advanced

Note that `Table` localizes the iterator variable with dynamic scoping, so extant expressions matching the iterator variable will be iterated. (Dynamic scope is too easy to lose track of and best avoided.)   For example:

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

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

2.2.3 Operating on Lists

Elementwise and Scalar Operations

We can use the standard arithmetic operators to do elementwise operations on lists. The following table lists some elementwise results for the lists va = $\{a_1, a_2, a_3\}$ and vb = $\{b_1, b_2, b_3\}$.

Elementwise Operations

Expression	Result
va	$\{a_1, a_2, a_3\}$
vb	$\{b_1, b_2, b_3\}$
va+vb	$\{a_1 + b_1, a_2 + b_2, a_3 + b_3\}$
va-vb	$\{a_1 - b_1, a_2 - b_2, a_3 - b_3\}$
va*vb	$\{a_1 b_1, a_2 b_2, a_3 b_3\}$
va/vb	$\{\frac{a_1}{b_1}, \frac{a_2}{b_2}, \frac{a_3}{b_3}\}$
va^vb	$\{a_1^{b_1}, a_2^{b_2}, a_3^{b_3}\}$

Scalar operations are also available. The following table provides some examples.

Scalar Operations

Expression	Result
va+2	$\{2+a_1, 2+a_2, 2+a_3\}$
va−2	$\{-2+a_1, -2+a_2, -2+a_3\}$
2*va	$\{2\,a_1, 2\,a_2, 2\,a_3\}$
va/2	$\{\frac{a_1}{2}, \frac{a_2}{2}, \frac{a_3}{2}\}$
2/va	$\{\frac{2}{a_1}, \frac{2}{a_2}, \frac{2}{a_3}\}$
va^2	$\{a_1^2, a_2^2, a_3^2\}$
2^va	$\{2^{a_1}, 2^{a_2}, 2^{a_3}\}$

Since lists support scalar multiplication and elementwise addition, we can use them to represent finite numerical vectors. We will return to this.

Maps

The basic facility for substitutive list creation is the `Map` command. The `Map` command applies a function to every element of a list. Once we create the appropriate range, the `Map` command can therefore duplicates functionality of the `Array` command.

On the one hand, `Array` is less general than `Map`, because the latter can map over any domain of values. On the other hand, when we are dealing with an appropriately structured domain of values, `Array` can be a bit more readable and may simultaneously be more compact. Furthermore, it does not require us to explicitly create the domain list.

We often produce lists by mapping over a range of values. The `Map` command applies a function to a list of values, thereby producing a new list of values.

Expression	Result
lst=Range[5]	{1, 2, 3, 4, 5}
Map[x⟼x^2,lst]	{1, 4, 9, 16, 25}
Map[♯^{2}&,lst]	{1, 4, 9, 16, 25}

The `Map` command has a popular infix shorthand: `/@`. It is conventional among WL users to map across lists with anonymous (unnamed) functions using this shorthand. Here are two variants of this approach to squaring the elements in a list.

Expression	Result
(x⟼x^2)/@lst	{1, 4, 9, 16, 25}
♯^{2}&/@lst	{1, 4, 9, 16, 25}

The second syntax, using slot designators, is actually the most common. However, it is probably the most obscure to someone new to WL. Can does not imply should, and slot notation should only be used when you have reason to expect it to be understood by the readers of your code. That said, from this point forward, we will feel free to use slot designators in illustrative code.

The `Map` command is extremely flexible. For example, we can use it to style list elements based on any criterion we wish. As an example, let us add bold styling to prime numbers, using the `If` command to condition the display style on primacy. (We can test for primacy with the `PrimeQ` command.) Prime numbers stand out visually in the resulting list of styled numbers.

```
If[PrimeQ[#], Style[#, Bold], Style[#, Gray]] & /@ Range[10]
```

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

When mapping across ranges, we often rely on the flexibility of the `Range` command instead of complicating the function to be mapped. Review the documentation of `Range` and then explain the following results.

Expression	Result
`f/@Range[5]`	`{f[1], f[2], f[3], f[4], f[5]}`
`(i↦0)/@Range[5]`	`{0, 0, 0, 0, 0}`
`(i↦i*i)/@Range[5]`	`{1, 4, 9, 16, 25}`
`(i↦i*i)/@Range[1,3,1/2]`	$\{1, \frac{9}{4}, 4, \frac{25}{4}, 9\}$
`(i↦i*i)/@Range[5,1,-1]`	`{25, 16, 9, 4, 1}`
`(i↦i/4)/@Range[0,4]`	$\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$

Advanced

Folds and Scans (`Fold` and `FoldList`)

The `Total` command sums up the elements of a list.

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

```
a + b + c
```

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

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

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

We are going to explore this observation in a bit more detail. A list has a “head” of `List`, which can be retrieved with the `Head` command or as part 0 of the list.

Expression	Result
`Head[{a,b,c}]`	`List`
`{a,b,c}〚0〛`	`List`

WL allows us to replace the head: we use the `Apply` command (or its shorthand, the `@@` operator). The `Apply` command simply replaces the “head” of an expression with a specified function. (This is not a simple application of a function to a single argument, which we could do with the `@` operator, but rather application of a function to the entire argument sequence in the list.)

Expression	Result
`Apply[f,{a,b,c}]`	`f[a, b, c]`
`f@@{a,b,c}`	`f[a, b, c]`

So, we can use `Apply` to replace the “head” of a list with `Plus` in order to find the total sum of list elements. Here we use the `@@` shorthand for `Apply`, which makes this a little easier to write.

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

```
a + b + c
```

We can produce the same result by using a fold. A fold repeatedly applies a binary operation to an accumulated value and a new element of a list, until all the list elements are used. Folds are common in functional programming languages. As part of its excellent support for functional programming, WL provides the `Fold` and `FoldList` commands. We first explore `FoldList`, because it produces a list of all the intermediate values produced by a fold, which is helpful for understanding how folds work.

Suppose we want the cumulative sum of a list. This new list has as its *i*-th element the sum of the first *i* elements of the original list. Accumulating all the intermediate values from repeatedly applying a binary operation is often called a scan, and WL provides this functionality with the `FoldList` command. (This kind of action is most often called a "scan"; however WL's `Scan` command does something quite different.) The arguments to `FoldList` are a binary function, an initializer (i.e., an initial value for the accumulator), and a list to scan.

```
ClearAll[f, x0, x1, x2]
FoldList[f, x0, {x1, x2}]
```

```
{x0, f[x0, x1], f[f[x0, x1], x2]}
```

As an example, let us fold `Plus` over the list `{a,b,c}`, with an intial value of 0 (the additive identity).

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

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

Note that the initializer is the first element of the `FoldList` result. We may wish to discard that element: the `Rest` command does exactly that. This leaves us with an ordinary cumulative sum.

```
Rest[flst]
```

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

Producing a cumulative sum is a very common need. For such common needs, WL often provides specialized and extremely efficient functions. This is no exception: the `Accumulate` command produces the cumulative sum of a list.

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

```
True
```

The last element of `flst` (our `FoldList` result) is the sum of all the list elements. We can retrieve the last element of a list with the `Last` command.

```
Last[flst]
```

```
a + b + c
```

However, if we only want the last element of a scan, it would be computationally wasteful to create the entire scan and then retrieve the final element. When we only want the final result of the scan produced by `FoldList`, we use the `Fold` command instead. As with `FoldList`, the arguments to `Fold` are a binary function, an initializer (i.e., an initial value for the accumulator), and a list to fold over. Folding is from left to right over the list elements.

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

```
a + b + c
```

The choice of binary operation and intializer is entirely up to us. If we use an arbitrary function, you can see that we first produce $f[x_0, a]$, and then we combine the result with the next list element (in this case, *b*), and then finally we combine that result with the last list element (in this case, *c*). We can similarly accumulate products or even powers.

Expression	Result
FoldList[f,x0,{a,b,c}]	{x0, f[x0, a], f[f[x0, a], b], f[f[f[x0, a], b], c]}
FoldList[Times,1,{a,b,c}]	{1, a, a b, a b c}
FoldList[Power,2,{a,b,c}]	$\{2, 2^a, (2^a)^b, ((2^a)^b)^c\}$

WL often provides many ways 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. Still, scans and folds have proved to be an extremely powerful programming construct, and if they are new to you, you should try to add them to your toolkit.

2.3 Nested Lists

Recall that the elements of a list can be any object, including other lists. A list inside another list is often called a nested list.

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.

Expression	Result	Comment
lstlst={{1,2},{3,4}}	{{1, 2}, {3, 4}}	list of lists
lstlst⟦1⟧⟦2⟧	2	element 2 of sublist 1
lstlst⟦1,2⟧	2	again: element 2 of sublist 1

2.3.1 Rectangular and Ragged Arrays

A nested list where all the inner lists have the same shape is called a rectangular array; otherwise it is a ragged array. Rectangular arrays of a constant value are most naturally produced with the `ConstantArray` command. For example, here is a rectangular array of zeros.

```
lstlst01 = ConstantArray[0, {2, 3}]
```

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

When an array is rectangular, we can use the `Dimensions` command to determine its shape. In this case, the first dimension is the length of the outermost list. The second dimension is the length of each of the inner lists.

```
Dimensions[lstlst01]
```

```
{2, 3}
```

In contrast, here is a ragged array: a list that contains three other lists, each of a different length. (As usual, we use the infix `/@` shorthand for the `Map` command.)

```
lstlst02 = ConstantArray[#, #] & /@ Range[3]
```

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

As we expect from our earlier discussion of list creation, we could use `Array` or `Table` instead of `Map` to create this nested list.

```
lstlst02 == Array[ConstantArray[#, #] &, 3] == Table[ConstantArray[i, i], {i, 3}]
```

```
True
```

We can also use `Range` to produce certain ragged arrays. It has the `Listable` attribute, so we do not need to map it.

```
Range@Range[1, 3]
```

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

We can also use `Range` to produce rectangular arrays by mapping it across a constant array of values.

```
Range /@ ConstantArray[3, 4]
```

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

For generating gridded coordinates, the `CoordinateBoundsArray` command plays the role of `Range` command in a multidimensional setting. The input is a nested list, where each inner list specifies the minimum and maximum values for the corresponding coordinate.

```
lstlstlst01 = CoordinateBoundsArray[{{1, 2}, {1, 3}}]
```

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

The result is a rectangular list of nested lists, where each innermost list can represent coordinates on a grid. This makes the point again that lists can contain any objects, including nested lists. Because of its regular structure, we again call this a “rectangular” array, but now there are more than two dimensions.

```
Dimensions[lstlstlst01]
```

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

2.3.2 Catenate, Partition, and Transpose

Given a list of lists, we can use the `Catenate` command to chain together the inner lists. If the original list had only one level of nesting, we end up with a flat list of all the elements.

```
flatlst = Catenate[lstlst02]
```

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

We can use `Partition` to create a rectangular array by partitioning a list into equal-length sublists. The arguments are a list to partition, and the length of the inner lists. (The result is always rectangular; left-over elements are discarded.)

```
Partition[flatlst, 2]
```

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

The `Partition` command offers functionality far beyond the simplest partitioning; see the documentation. One variant we will care about involves the specification of an “offset” as a third argument, where we can specify an offset specifying how far to shift to the right before starting the next sublist. We can therefore easily generate sublists of adjacent pairs.

```
a52 = Partition[flatlst, 2, 1]
```

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

This is a rectangular array: the five inner lists each contain two items. We say that the array has five rows and two columns; we also say that is has dimensions 5×2. We can retrieve the dimensions of a rectangular array with the `Dimensions` command.

```
Dimensions[a52]
```

```
{5, 2}
```

It is possible to swap the rows and columns of a rectangular array with the `Transpose` command.

```
Transpose[a52]
```

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

A hidden gem is `PartitionRagged`, which is currently in `Internal`.

```
ClearAll[f]
tri = Internal`PartitionRagged[Range[10], Range[4]]
```

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

We can use `PadRight` or `PadLeft` to turn a ragged array into a rectangular array. For example,

```
PadRight[tri]
```

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

Advanced: PartitionMap

We often want to process the lists that result from a partition. For example, suppose we want to add adjacent pairs.

```
Total /@ Partition[flatlst, 2, 1]
```

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

The computational problem with this approach is that we create the partitioned list, but we have no need for it. That is, this approach first creates a list of pairs and then maps a function across the pairs. If we need the pairs for no other purpose, it is computationally wasteful to create the list of pairs. This waste can matter when we are working with very large lists.

The `PartitionMap` command allows us to avoid this computational waste by combining the partition and mapping into a single operation. Currently this command is in the `Developer` package.

```
Developer`PartitionMap[Total, flatlst, 2, 1]
{3, 4, 5, 6, 6}
```

2.3.3 Creating Nested Lists with `Map`, `Array`, `Table`, and `Apply`

We can use any function as the first argument to `Map`, as long as that function needs a single argument. Since a function can return a list, we can eaily create nested lists with `Map`. As an application, let us pair each domain value with another value as follows.

```
ClearAll[f]
pairs = {#, f[#]} & /@ Range[5]
{{1, f[1]}, {2, f[2]}, {3, f[3]}, {4, f[4]}, {5, f[5]}}
```

Naturally, we expect to be able to accomplish the same thing with `Array` or `Table`.

```
pairs == Array[{#, f[#]} &, 5] == Table[{i, f[i]}, {i, 5}]
True
```

When we move to functions of more than one variable, `Array` and `Table` gain convenience relative to `Map`. For example, here is a function of two variables evaluated on a grid of values. Note that the second argument is now a pair of values, where each entry specifies the range for one coordinate.

```
Array[f, {2, 3}]
{{f[1, 1], f[1, 2], f[1, 3]}, {f[2, 1], f[2, 2], f[2, 3]}}
```

Recall that the `Array` command produces a list by applying a function (its first argument) to the evenly spaced input values (described by the second argument). We can also shift the index origin with a third argument: an “index origin” is the number from which to start the incremented indexes. In order to accomplish the same thing with `Table`, we specify a second iterator.

```
Table[f[i, j], {i, 2}, {j, 3}]
{{f[1, 1], f[1, 2], f[1, 3]}, {f[2, 1], f[2, 2], f[2, 3]}}
```

What about `Map`? Is it adequate to producing such lists? The answer is yes, but not elegantly. First of all, we need to generate the points in the domain. We can use `CoordinateBoundsArray` for this purpose. Once we have the desired points (as lists), we need to apply our function to the arguments represented by each innermost list. This is possible, since `Map` accepts a third argument, which allows us to specify the level at which we are applying our function. However, at that point the more straightforward way is to simply use `Apply` command, which also accepts a level specification.

```
Apply[f, CoordinateBoundsArray[{{1, 2}, {1, 3}}], {2}]
{{f[1, 1], f[1, 2], f[1, 3]}, {f[2, 1], f[2, 2], f[2, 3]}}
```

All told, at this point it looks simpler to use `Array` or `Table`. (However, see the discussion of `MapThread` below.) Clearly, for the construction of nested lists, the choice between `Table` and `Array` is often a matter of convenience. Expressions in `Array` are sometimes easier to read, and `Table` must use explicit expressions in the iterator variables. On the other hand, the use of explicit iterators in `Table` can enable the easier construction of non-rectangular nested lists. (These are often called "ragged arrays".) For example,

```
Table[f[i, j], {i, 3}, {j, i}]
```

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

```
lstlstlst = Table[{i, j}, {i, 1, 3}, {j, 1, i}]
```

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

Now we have two levels of nesting, because the items we are creating in our list of lists are themselves lists. If we really want just a list of the inner lists, we can use the `Catenate` command to flatten the first level of nesting.

```
Catenate[lstlstlst]
```

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

Advanced

2.3.4 Mapping with Multiple Arguments

Sometimes we wish to map with a function that takes multiple arguments. We can use `MapThread` for this. While the inputs to `Map` are a function and a list of arguments for that function, the inputs to `MapThread` are a function and a list of argument lists for that function.

```
ClearAll[f, a, b, c, x, y, z]
MapThread[f, {{a, b, c}, {x, y, z}}]
```

```
{f[a, x], f[b, y], f[c, z]}
```

As a closely related concern, we may want our manipulation of an element to depend on the location of the element. We can the `MapIndexed` command in this case; it provides the element index as a second argument to the function. The index is always provided as a list. (This is because we can apply the mapped function at different levels of a nested list; we will return to this observation eventually.)

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

```
{f[a, {1}], f[b, {2}], f[c, {3}]}
```

While the "index" is always a list, we can of course extract parts of this list in the usual ways. For example, we can use the `First` command to extract the first (and in this case, only) value in this list.

```
MapIndexed[#1^First[#2] &, {a, b, c}]
```

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

As an illustrative application, consider the computation of a Gini coefficient. A Gini coefficient of inequality is twice the area between the line of perfect equality and the Lorenz curve. (We discuss the Lorenz curve in the next section when we present line charts.) Roughly speaking, this is twice the mean distance of points on the Lorenz curve from the line of perfect equality.

```
gini[xs_List] := With[{cumsort = Accumulate[Sort[xs]], n = N@Length[xs]},
  2 * Mean[MapIndexed[First[#2] / n - #1 &, cumsort / Last[cumsort]]]]
```

MapThread

`MapThread` generalizes `Map` to functions that have multiple arguments. The arguments are given as a nested list of values, where the inner lists have a common length. The multivariate function takes one argument from each inner list, working through them in step.

Expression	Result
lst01={a,b,c,d}	{a, b, c, d}
lst02={1,2,3,4}	{1, 2, 3, 4}
MapThread[f,{lst01,lst02}]	{f[a, 1], f[b, 2], f[c, 3], f[d, 4]}

For example, we can zip together two lists as follows. (Ordinarily we would use the `Transpose` command for this.)

```
MapThread[List, {lst01, lst02}]
```

```
{{a, 1}, {b, 2}, {c, 3}, {d, 4}}
```

Or, we can make a list of rules like this:

```
MapThread[Rule, {lst01, lst02}]
```

```
{a → 1, b → 2, c → 3, d → 4}
```

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

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

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

As an aside, when a function has already been applied to list that contains its arguments, you can use `Thread` instead of `MapThread`. This can be used to thread equations. Somewhat surprisingly, `Thread` allows an argument to be a single constant, which it then repeats as needed.

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

It follows that we could also use `Thread` to zip together two lists by choosing `List` as our function to thread.

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

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

Nevertheless, `Thread` is less intuitive, and can usually be avoided in favor of `Map` or `MapThread`. Or, in the particular case of transposing a rectangular list of lists, we generally turn to the `Transpose` command.

2.3.5 Formatting Nested Lists as Tables

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

Simple Table Formatting with `TableForm`

We have seen how to use `Table`, `Array`, and `Map` to created nested lists. Rectangular lists can be nicely formatted as two-dimensional tables, using the `TableForm` command. Here is an example. (We use the `TableAlignments` option to right-align the text.)

```
tbl = Table[{x, x^2}, {x, 0, 5}];
TableForm[tbl, TableHeadings → {None, {"x", "x^2"}}, TableAlignments → Right]
```

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

We can use the `TableDirections` option to transpose the table display.

```
TableForm[tbl, TableHeadings → {None, {"x", "x^2"}},
 TableAlignments → Right, TableDirections → Row]
```

```
  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. Here is an example.

```
nvals = Range[3]; xvals = Range[0, 5];
tbl = Table[x^n, {x, xvals}, {n, nvals}];
TableForm[tbl, TableHeadings → {xvals, nvals}, TableAlignments → Right]
```

```
  |  1    2     3
------------------
0 |  0    0     0
1 |  1    1     1
2 |  2    4     8
3 |  3    9    27
4 |  4   16    64
5 |  5   25   125
```

Tables from External Data

Naturally, WL can build lists from external data. Here we provide a couple of examples.

First we need some external data. Let us store some numbers in “temp.csv”. CSV stands for comma-separated values, which is a common data exchange format. (See RFC 4180.) The `Export` command will recognize the intended format from the filename extension `.csv`, but here we include the format specification as a third argument, just to be explicit.

```
(* choose a filename (it must be safe to overwrite!): *)
fname = FileNameJoin[{$TemporaryDirectory, "temp.csv"}];
(* WARNING: this overwrites temp.csv! (but only in temp folder) *)
Export[fname, {{1, 2, 3}, {4, 5, 6}}, "CSV"];
```

Use `FilePrint` to take a look at our data.

```
FilePrint[fname]
```

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

Next we will read our data into a list. This is possible with `ReadList`, but the CSV format is more easily handled by `Import`.

```
datalist = Import[fname, "CSV"]
```

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

If we want it all as a single list, we can use `Flatten`.

```
Flatten[datalist]
```

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

Mathematica also provides web access to a variety of curated 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. (The special formatting of the country names reflects the fact that the list elements are country entities; we will return to this.)

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

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

Such entities may have many predefined properties. To take a fairly trivial example, we can access the country codes as an entity property. (See the documentation for details; also, we will explore this further in subsequent sections.)

```
CountryData[#, "CountryCode"] & /@ listG8
```

```
{CA, FR, DE, IT, JP, RU, GB, US}
```

Next we consider a more interesting example. We use `Map` to extract the two-letter country code and population for each G8 country, and we then display the result using `TableForm`. (For a more compact but still reasonably attractive display, we right-align the text and use the `TableSpacing` option to eliminate the spacing between rows.) One interesting thing to notice about Wolfram's country data is that it includes units (in this case, people).

```
listG8 = CountryData["G8"];
g8codes = CountryData[#, "CountryCode"] & /@ listG8;
g8pops = CountryData[#, "Population"] & /@ listG8;
TableForm[{g8codes, g8pops}, TableDirections → Row,
 TableAlignments → Right, TableSpacing → {1, 0}]
```

```
CA   35 309 555 people
FR   64 101 308 people
DE   81 625 599 people
IT   61 175 248 people
JP  126 225 259 people
RU  142 400 066 people
GB   63 556 184 people
US  322 422 965 people
```

We can also frame and label a table. Before illustrating this, let us extract a little more information about the G8 countries. Here we associate country names with their per capita GDPs (in dollars) using the `TableForm` command. Then we display a framed and labeled table, using the `Frame`, and `Labeled` commands.

```
g8GDPs = CountryData[#, "GDP"] & /@ listG8;
table = TableForm[{g8codes, g8GDPs / g8pops}, TableDirections → Row,
   TableHeadings → {{"Country", "GDP p.c."}, None}, TableSpacing → {1, 0}];
title = StringTemplate["GDP per capita (retrieved ``)"][
   DateString[{"MonthNameShort", "Year"}]];
Labeled[Framed[table], title, Top]
```

```
  GDP per capita (retrieved Sep2017)
Country  GDP p.c.
CA       $50 563.8 per person per year
FR       $44 136.3 per person per year
DE       $47 390.7 per person per year
IT       $35 000.5 per person per year
JP       $36 454.4 per person per year
RU       $13 066. per person per year
GB       $47 027.6 per person per year
US       $54 025.3 per person per year
```

The final result looks rather good for a quick and dirty table. Better looking tables are a bit more work and may require using the `Grid` command. However, keep in mind that, in contrast to exported graphics, the tables exported from a notebook often lose much information (including any coloration or shading). While this can be partially overcome by snipping a bitmap of the table, that approach will introduce a loss of scalability and will generally not be of publication quality. (On the other hand, many publication outlets may expect to reproduce the tables you supply, regardless of the format.)

Using `Grid` for Two-Dimensional Tables

We can achieve fine formatting control with the `Grid` command. Fine control can be a complex undertaking, but `Grid` also provides simple way to produce formatted two dimensional displays 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 use the `Background` option to shade specific rows, columns, or even items.

```
Grid[Prepend[gdppcG8, {"Country", "GDP per capita"}],
 Alignment → {{Left, Right}, Automatic},
 Dividers → {None, {1 → True, 2 → True, -1 → True}},
 Background → {None, {1 → Lighter[Gray, 0.9]}}
]
```

```
Prepend: Nonatomic expression expected at position 1 in Prepend[gdppcG8, {Country, GDP per capita}].
```

```
Grid[Prepend[gdppcG8, {Country, GDP per capita}], Alignment → {{Left, Right}, Automatic},
 Dividers → {None, {1 → True, 2 → True, -1 → True}}, Background → {None, {1 → □}}]
```

The bottom line is that `Grid` is much more flexible, but `TableForm` is simpler and is often adequate for common table formatting needs. For details, see the Wolfram Language Overview entitled Grids, Rows, and Columns.

2.4 Associations

In this section, we briefly introduce another core WL data type: the association. An association maps keys to values, with behavior similar to that of a hash table or associative list in other languages. This section covers the few features of associations that we will need in this book. However, associations are feature rich. See the Wolfram Language Guide entitled Associations for more details.

Creation by Explicit Rule Enumeration

An association is roughly a sequence of ordered pairs, each represented as a rule. We can create a rule with the `Rule` command, which has an arrow operator form. (Enter this arrow with two keystrokes: a hyphen, followed by the greater-than symbol.) Each rule has two parts, which we call the key and the value. When creating associations, it is often useful to use strings as keys.

Expression	Result
Rule["one",1]	one → 1
"two"->2	two → 2

We can create an association from a comma-separated sequence of rules. We use the `Association` command, usually as its special bracketing shortcut notation `<|...|>`. Note that the display of an association is much like a list of rules. (In fact, the `Association` command will also convert a list of rules to an association.) We can also convert an association to a list of rules with the `Normal` command. The strings in the resulting expressions normally display without their quotes.

Expression	Result
assoc01=<|"one"→1,"two"→2|>	⟨|one → 1, two → 2|⟩
Normal[assoc01]	{one → 1, two → 2}

Unlike an arbitrary list of rules, an association is a mapping. This means that it associates each key with a single value. If you reuse a key when creating an association, only the last value provided for a key will remain in the association.

Modifying Associations

We can create a new association by using the `Association` command to change or add to the key-value pairs (as rules) taken from an existing association. We can modify an association by using the `AssociateTo` command to change or add key-value pairs (as rules) to an existing association.

```
⟨|one → 1, two → 2|⟩
```

Expression	Result
assoc01=<|"one"→1,"two"→2|>	⟨|one → 1, two → 2|⟩
<|assoc01,{"two"->22,"three"->3}|>	⟨|one → 1, two → 22, three → 3|⟩
assoc01	⟨|one → 1, two → 2|⟩
AssociateTo[assoc01,{"two"->22,"three"->3}]	⟨|one → 1, two → 22, three → 3|⟩
assoc01	⟨|one → 1, two → 22, three → 3|⟩

It is nevertheless common to overwrite an existing value for a key or assign a new key-value pairs to an existing association using a function-like notation.

Expression	Result
assoc01["two"]=2	2
assoc01["three"]=33	33
assoc01	⟨|one → 1, two → 2, three → 33|⟩

Use the `Keys` command or the `Values` command to extract a list of keys or a list of values from an association. If we use `Part`, we find that parts of an association are the values, not the keys.

Expression	Result
Keys[assoc01]	{one, two, three}
Values[assoc01]	{1, 2, 33}
Part[assoc01,1]	1

We can also drop entries from an association with the `KeyDropFrom` command or similarly produce a new smaller association with `KeyDrop`. (It is not an error to drop keys that are not present.)

Expression	Result
assoc01=<|"a"->1,"b"->2,"c"->3|>	⟨|a → 1, b → 2, c → 3|⟩
KeyDrop[assoc01,{"a","c","d"}]	⟨|b → 2|⟩
assoc01	⟨|a → 1, b → 2, c → 3|⟩
KeyDropFrom[assoc01,{"c"}]	⟨|a → 1, b → 2|⟩
assoc01	⟨|a → 1, b → 2|⟩

We can perform various mapping operations on an association. We can use `Map` to produce a new association by mapping over the values, whereas `KeyMap` produces a new association by mapping over the keys. Functions operating on associations that have strings as keys can refer by key name to the values.

Expression	Result
assoc02=Map[v⟼v^2,assoc01]	⟨|a → 1, b → 4|⟩
assoc03=KeyMap[k⟼k<>"3",assoc01]	⟨|a3 → 1, b3 → 2|⟩
Function[#a3+#b3][assoc03]	3

The `KeyValueMap` command produces a list by applying a bivariate function to the key-value pairs of an association. For example, using the `<>` shorthand for `StringJoin`, which joins together two strings, we can do the following

```
tmp = StringTemplate["`` maps to ``"];
KeyValueMap[tmp, <|"a" → 1, "b" → 2|>]
```

```
{a maps to 1, b maps to 2}
```

2.5 Sorting

Numbers have a natural ordering. We often need to convert a list that is unsorted (i.e., out of order) into a sorted list. At times we wish to do more complex sorting, including sorting of data types other than numeric lists. This subsection provides an basic introduction to some of WL’s sorting facilities.

2.5.1 Sorting Lists

The `Sort` command sorts expressions in ascending order. This means that lists of numbers can easily be sorted in a natural way. The position that each element will have in the sorted list can be determined with the `Ordering` command, so the sorted list can also be produced by combining the `Part` and `Ordering` commands.

Expression	Result
lst = RandomInteger[100,5]	{36, 25, 17, 7, 79}
o=Ordering[lst]	{4, 3, 2, 1, 5}
lst⟦o⟧	{7, 17, 25, 36, 79}
Sort[lst]	{7, 17, 25, 36, 79}

While `Sort` is a natural choice for numerical lists, it is often the wrong choice for mathematical expressions that have not yet been coerced to a numeric value. Expressions will be sorted by special rules for expressions, not by the value of the expressions. When you want expressions to be sorted by their numerical value, you can use the `SortBy` command with `N` as its second argument. The second argument is a function that is applied to each list element, and the sort is based on the function values. The following examples illustrate this behavior.

Expression	Result	Comment
lst={1,Sin[3],Sqrt[3]}	$\{1, \text{Sin}[3], \sqrt{3}\}$	exact expressions
N[Sort[lst]]	{1., 1.73205, 0.14112}	expression sort!
Sort[N[lst]]	{0.14112, 1., 1.73205}	numeric sort
SortBy[lst,N]	$\{\text{Sin}[3], 1, \sqrt{3}\}$	numeric sort

Criterion-Based Sorting

More generally, we can use the `SortBy` command to sort a list based on any criteria we choose. As an example, we will sort the some countries by various criteria. Before doing this we briefly explore the idea of an entity, and entity property, and an entity property value.

WL curates data related to real-world entities, which are categorized into entity types. (For a list of entity types, enter `EntityValue[]`.) Specific entities are identified by type and name. One type of entity is “Country”. Let us get the country entity identified by the string “UnitedStates”. (For a list of all the country entities, enter `EntityValue[“Country”,”Entities”]`.)

```
us = Entity["Country", "UnitedStates"]
```

```
United States
```

Note that the output display of entities is distinctive. Once we have an entity, we can request information about its properties. (After entering the code above, get all the properties for the United States by entering `us[“Properties”]`.) The default value of each property can be characterized by an entity property object, formed by providing the `EntityProperty` command with an entity type and a property name. As a third argument we can provide a list of qualifier rules. As an example, let us create an entity property for country population, qualified to be for a particular year.

```
epPOP = EntityProperty["Country", "Population", {"Date" → DateObject[{2016}]}];
```

As an aside, it is possible to request some information about the source of this data. If you want to ask for the data source, you must use unqualified entity properties: `epPOP[“Source”]` will fail here because of our data qualification. However `First@EntityProperty[“Country”,”Population”]` will succeed; it provides a list of data source entities, each of which may offer additional data-source details as properties.

We can make lists of entities. For example, fetch all the countries in the G7 as follows.

```
listG7 = CountryData["G7"]
```

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

We can treat this list like any other. For example, we can use `SortBy` to sort it on any criterion we choose. Here we sort by population, largest to smallest.

```
popSortedG7 = SortBy[listG7, country ↦ -country[epPOP]]
```

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

As another example, we can sort these countries by per capita GDP. (Comparing GDPs across countries is difficult; for this exercise we work with the Wolfram data without worrying about its adequacy.) Our sorting function will fetch the GDP data and then divide it by the population data.

```
epGDP = EntityProperty["Country", "GDP", {"Date" → DateObject[{2016}]}];
getGDPpc = country ↦ country[epGDP] / country[epPOP];
gdppcSortedG7 = Reverse@SortBy[listG7, getGDPpc]
```

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

In the background, `SortBy` is clearly producing the comparison data for every country. Often we wish to persistently create all of those values as a new list. For this we can use `Map`. (Recall from our discussion of list creation that `Map` applies a function to every element of a list in order to produce a new list of values.) In the next example, we map each country to a list of values, thereby producing a nested list. With this nested list in hand, we can use `SortBy` with Part to sort by population. Here we again use the `TableForm` command to produce a readable display of our sorted data. As a slight refinement, we use `Framed` to place a frame around the table and `Labeled` to add a table caption.

```
table = TableForm[(ctry ↦ {ctry["Name"], getGDPpc[ctry]}) /@ gdppcSortedG7,
   TableHeadings → {None, {"Country", "GDP p.c."}}, TableSpacing → {0, 2}];
Labeled[Framed[table], "GDP Per Capita", Top]
```

GDP Per Capita

Country	GDP p.c.
United States	$56 496.4 per person per year
Germany	$42 616.6 per person per year
Canada	$42 567. per person per year
United Kingdom	$40 721.1 per person per year
Japan	$39 237.4 per person per year
France	$38 095.7 per person per year
Italy	$30 185.7 per person per year

2.5.2 Sorting Associations

Associations are ordered, and they can be reordered. In particular, associations can be sorted, either by their keys or by their values. To sort an association by its keys, use the `KeySort` command. (There is also a `KeySortBy` command for criterion-based sorting.)

```
assoc01 = <|10 → 1, 8 → 3, 9 → 2, 7 → 4|>;
KeySort[assoc01]
```

```
⟨|7 → 4, 8 → 3, 9 → 2, 10 → 1|⟩
```

It is interesting and perhaps somewhat unexpected that `KeySort` can also be used to sort a list of rules, with the returned value being an association. This implies that if there are any duplicate “keys” in the list of rules, only the last occurrence will be retained in the result.

```
rules01 = Append[Normal@assoc01, 9 → 5];
KeySort[rules01]
```

```
⟨|7 → 4, 8 → 3, 9 → 5, 10 → 1|⟩
```

To sort an association by its values, use the `Sort` command. (One may also use the `SortBy` command for criterion-based sorting.)

```
Sort[assoc01]
```

```
⟨|10 → 1, 9 → 2, 8 → 3, 7 → 4|⟩
```

Frequency Tables

In this section we combine our introductions to associations and to sorting to illustrate the creation of simple frequency tables.

Consider the following data, which might represent observed outcomes from an experiment.

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

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

Applying the `Counts` command to a list produces an association that maps each outcome to its frequency.

```
cts = Counts[data]
```

```
⟨|5 → 1, 4 → 2, 3 → 3, 2 → 2, 1 → 1|⟩
```

Recall that we can sort an association on either its values (with `Sort`) or its keys (with `KeySort`). Let us sort by key.

```
keySortedData = KeySort[cts]
```

```
⟨|1 → 1, 2 → 2, 3 → 3, 4 → 2, 5 → 1|⟩
```

Next, let us display an associated table, showing the frequency of each outcome. As of version 11 of Mathematica, `TableForm` does not accept an association. Our solution will be to use the `KeyValueMap` command to map the association to a list of pairs, which we already know how to display as a table. The `KeyValueMap` command makes this possible with minimal effort: we just map the `List` command across the key-value pairs.

```
ctsList = KeyValueMap[List, keySortedData]
```

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

This gives us our summary data as a list of pairs, which we can format with `TableForm`.

```
TableForm[ctsList, TableHeadings → {None, {"item", "count"}}]
```

```
item    count
1       1
2       2
3       3
4       2
5       1
```

Let us cover the same ground with a slightly more elaborate example. Suppose we repeatedly roll a six-sided die and then use the `Counts` command to produce an association from the faces to the number of occurrences.

```
rolls = RandomInteger[{1, 6}, 200];
facesCounts = Counts[rolls]
```

```
<|2 → 34, 6 → 35, 4 → 46, 5 → 30, 3 → 27, 1 → 28|>
```

Next, consider another use of `KeyValueMap`: we will pair each face value with the total value of the rolls of each face, measured as the face value multiplied by the associated number of rolls. Before making a table relating faces to these values, we sort our results. (Sorting a rectangular array of numbers produces a lexicographic sort, with the primary sort being on the first elements. Here we use that to sort by face value.)

```
facesValues = KeyValueMap[{k, v} ↦ {k, k * v}, facesCounts];
TableForm[Sort@facesValues,
 TableHeadings → {None, {"Face", "Total Value"}}, TableAlignments → Right]
```

```
Face    Total Value
   1             28
   2             68
   3             81
   4            184
   5            150
   6            210
```

Chapter 3: Basic Charts

3.1 Plots for Visualization

Plots are an important way to explore data and functions. WL provides extensive and powerful plotting facilities. This section introduces a few of the core plotting commands. (For more detail, refer to the Wolfram How To entitled Create Plots.) Future chapters will introduce additional chart types.

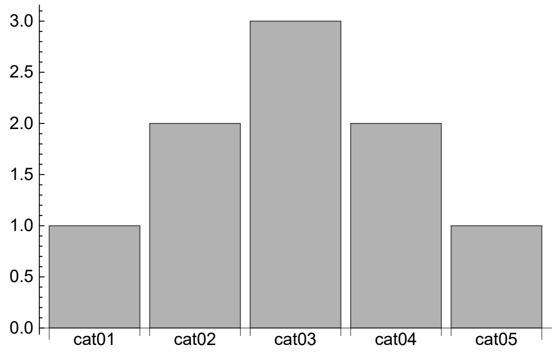
3.1.1 Basic Data Plots

A simple two-dimensional plot draws points in the Cartesian plane. Naturally, we cannot view the entire plane. Instead, we create viewing window appropriately scaled to accommodate our data.

Bar Charts

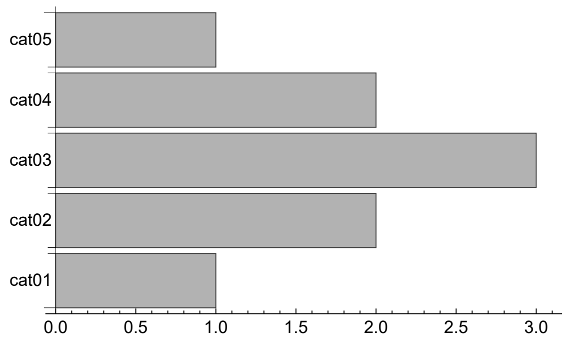
We often want to display the frequency (or relative frequency) with which categories of items occur. Due to the influence of spreadsheets, bar charts are often chosen for this purpose. For example, suppose we have the following category labels and associated observed frequencies.

```
ClearAll[labels, nums]
labels = {"cat01", "cat02", "cat03", "cat04", "cat05"};
freq01 = {1, 2, 3, 2, 1};
BarChart[freq01, ChartLabels → labels,
 PlotTheme → "Monochrome", ImageSize → 288]
```



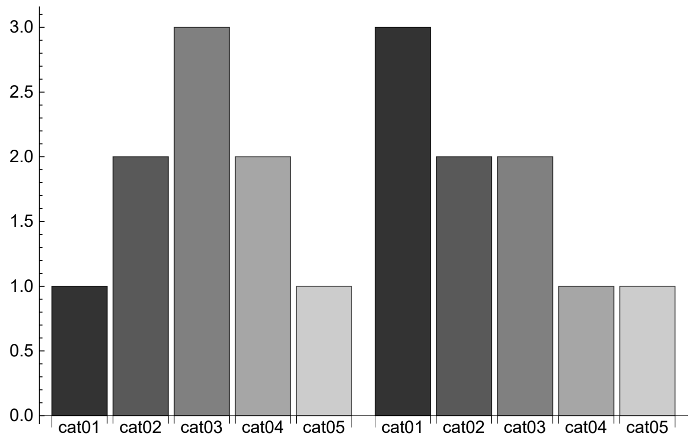
Sometimes horizontal bars are desirable. We can achieve the with the `BarOrigin` option. In this case, we may wish to use a

```
BarChart[freq01, ChartLabels → labels, BarOrigin → Left, ImageSize → 288]
```



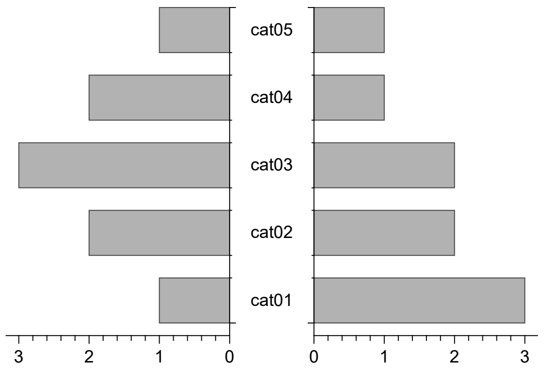
Often it is desirable to make attribute frequency comparisons across broader categories. The first argument to `BarChart` can be a list of lists, where each inner list provides the data for the broader category.

```
freq02 = {3, 2, 2, 1, 1};
BarChart[{freq01, freq02}, ChartLabels → labels, ImageSize → 360]
```



Although the default display is color coded to facilitate category comparison, these comparison may be easier with a paired bar chart. (To make enough space for the category labels, we will use the `BarSpacing` option.)

```
PairedBarChart[freq01, freq02, ChartLabels → labels, BarSpacing → {Large, Large, Large},
 PlotTheme → "Monochrome", ImageSize → 288]
```

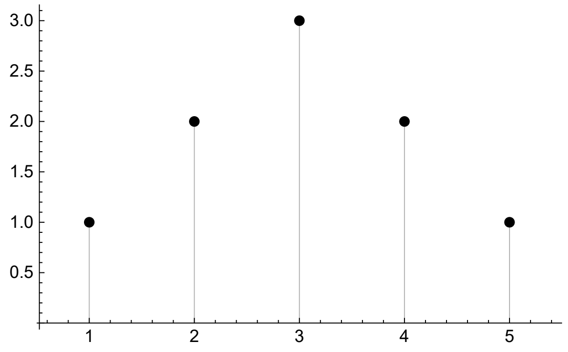


Frequency Plots

It is not uncommon to find bar charts used to create frequency charts. However, many people object to the choice of a bar chart for a simple frequency display, because it adds a lot of visual noise that does not communicate anything about the data. For example, the width and color of the bars is not informative. An alternative without these drawbacks is a simple stick-pin plot, which we can produce with the `ListPlot` command.

The data for the `ListPlot` command must be a list of points (i.e., a nested list, where each inner list is an ordered pair). In order to pair values with their frequencies of occurrence, we can zip them together with the `Transpose` command. If we feed `ListPlot` a list of points, it simply plots the points in standard Cartesian coordinates. However, we can use the `Filling` option to produce a nice stick-pin plot, which is appropriate for a frequency plot. (Here we use the `PlotRangePadding` option so that our plot does not crowd the axis.)

```
values = {1, 2, 3, 4, 5};
freqs = {1, 2, 3, 2, 1};
pts = Transpose@{values, freqs};
ListPlot[pts, Filling → Axis, PlotRangePadding → {0.5, Automatic}, ImageSize → 288]
```



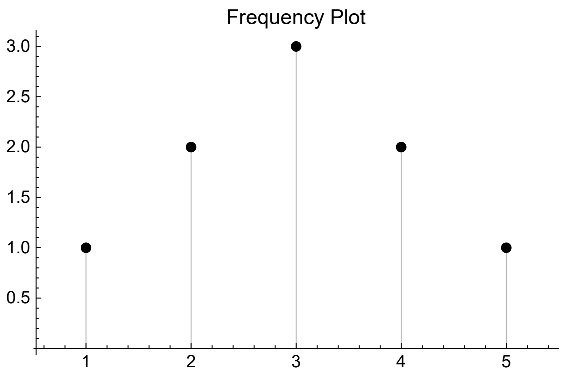
In the example so far, we have been given the values and their frequencies as two separate lists of data. Naturally, it is much more common that we start with a single list of raw data. Before we can produce our frequency plot, we need to find the unique items and pair each value with its frequency. This means we must first associate each possible item value with its count—the number of times it occurs. As we have seen in our discussion of associations, we can accomplish this pairing with the `Counts` command.

```
rawdata = {5, 4, 3, 4, 3, 2, 3, 2, 1};
cts01 = Counts[rawdata]
⟨|5 → 1, 4 → 2, 3 → 3, 2 → 2, 1 → 1|⟩
```

Recall that if we apply the `Counts` command to a list, it returns an association that maps the item values (which are the association's keys) to the item-value frequencies (which are the association's values). It is a pleasant convenience that we can apply `ListPlot` directly to an association. Before we do so, let us add a few options to our plot to clean it up. Along the horizontal axis, the only ticks we need are the values we are plotting. We can set this with the `Ticks` option. We will also use the `PlotRange` option to extend the horizontal axis a bit, for better appearance. Finally, we use the

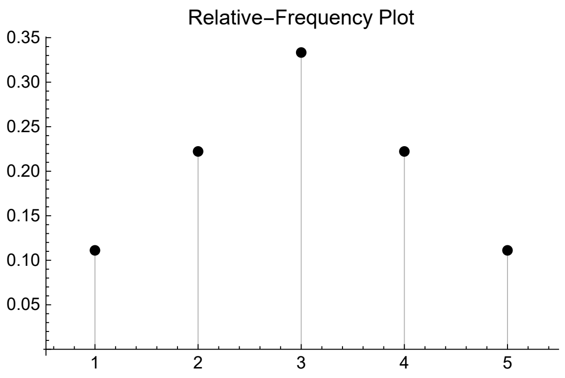
`PlotLabel` option to add a label to our plot, and we end up with a decent looking frequency plot. (Depending on the version you are working with, you may also want to specify the ticks explicitly.)

```
pinplotOptions = Sequence[
   Filling → Axis,
   PlotRangePadding → {0.5, Automatic},
   ImageSize → 288
  ];
ListPlot[cts01, pinplotOptions, PlotLabel → "Frequency Plot"]
```



Although it is visually similar, we may prefer a relative-frequency plot. This simply scales the absolute frequency by the total number of observations. Although we could naturally produce the number of observations as the length of the raw data, here we show that it can easily be produces as the sum over the values in our summary association.

```
nobs = Total[cts01];
ListPlot[cts / nobs, pinplotOptions, PlotLabel → "Relative-Frequency Plot"]
```



Producing an aesthetic and communicative plot is an art, and there is plenty of room for individual preferences to play a role. Plotting commands have many options; you can review the options for this command by evaluating `Options[ListPlot]`. Use these option freely to produce charts that are both useful and attractive.

Frequency Plots from Datasets

Adding to our earlier discussion of datasets, we now show how to produce a frequency plot from an existing dataset. In order to do so, we first create some artificial data; we might think of this as a collection of scores on two homework assignments. Each score will be a random integer between 1 and 4 (inclusive).

```
data02 =
  Dataset@Table[<|"hw1" → RandomInteger[{1, 4}], "hw2" → RandomInteger[{1, 4}]|>, 50];
```

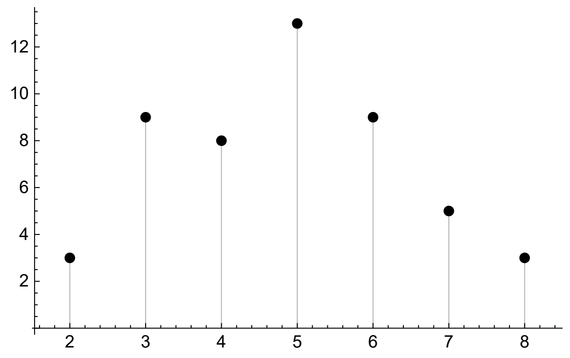
Next, we again make use of the flexibility of the `Query` command. This time, we recognize that its first argument can be an *aggregation operator*, such as `Counts` or `CountsBy`. We are going to determine the frequency of all the different totals across the two scores by creating an appropriate query and applying it to our dataset. This effectively gives us a frequency table as a dataset. (You can even apply `KeySort` to it if you wish.)

```
cts02 = Query[Counts, #hw1 + #hw2 &]@data02
```

7	5
4	8
3	9
2	3
5	13
6	9
8	3

Conveniently, we can apply `ListPlot` to the dataset produced by our query.

```
ListPlot[cts02, pinplotOptions]
```

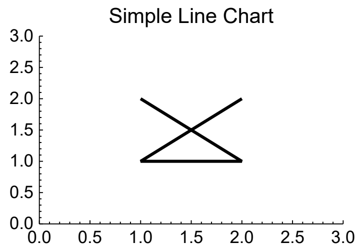


Run-Sequence Line Charts

chk

A basic line chart displays the information in a sequence of points by drawing straight line segments between each point. We usually produce line charts with the `ListLinePlot` command, which joints the data points with line segments. As with the `ListPlot` command, we can control the range of values displayed with the `PlotRange` option.

```
ListLinePlot[{{1, 2}, {2, 1}, {1, 1}, {2, 2}}, PlotRange → {{0, 3}, {0, 3}},
 PlotLabel → "Simple Line Chart", ImageSize → Small]
```

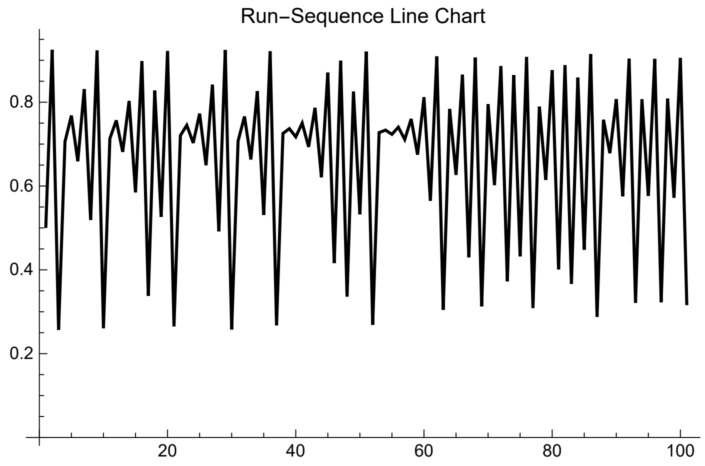


chk

A run-sequence line chart is a simple line chart based on a univariate sequence of data, where the order is significant. Points are formed by associating each index with the data value at that index. Chambers et al. (1983) describe run-sequence charts as a convenient summary of univariate data sets. (When the data are believed to be drawn from a probability distribution, outliers are easily detected, as are shifts in scale and location.)

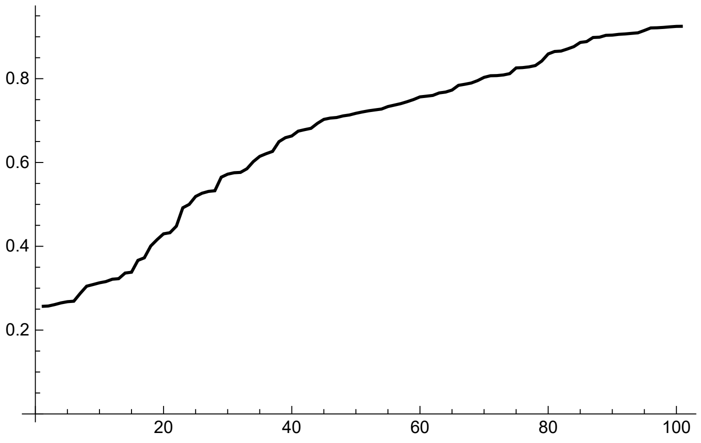
Here we produce a some univariate data by using `NestList` to iterate a logistic function. We then produce a run-sequence line chart of the data with the `ListLinePlot` command, using the `PlotLabel` option to add a label to the plot.

```
logisticData37 = NestList[x ↦ 3.7 * x * (1 - x), 0.5, 100]; (* list of arbitrary data *)
ListLinePlot[logisticData37, PlotLabel → "Run-Sequence Line Chart"]
```



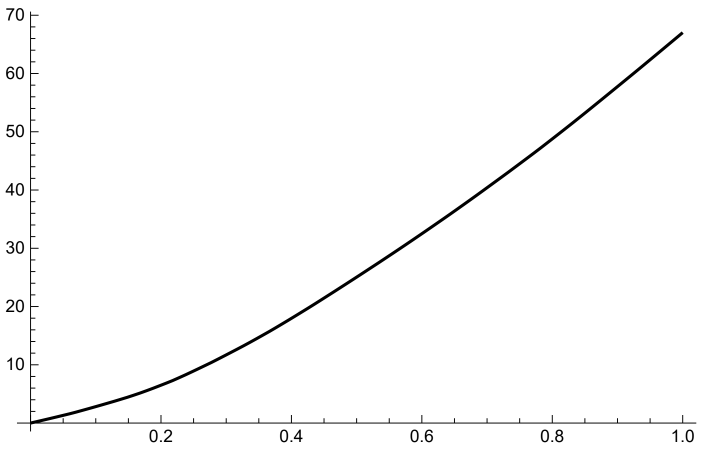
When the sequential values correspond to sequential points in time, the run-sequence plot is a time-series plot. (See below.) Other times, the order in the original data is less informative than the sorted data. While that is not true of our logistic plot data, we can still explore what happens if we sort it. Suppose for example that the data represented household wealths. We might wish to sort it to get a sense of the distribution of wealth.

```
sortedData = Sort[logisticData37];
ListLinePlot[sortedData]
```



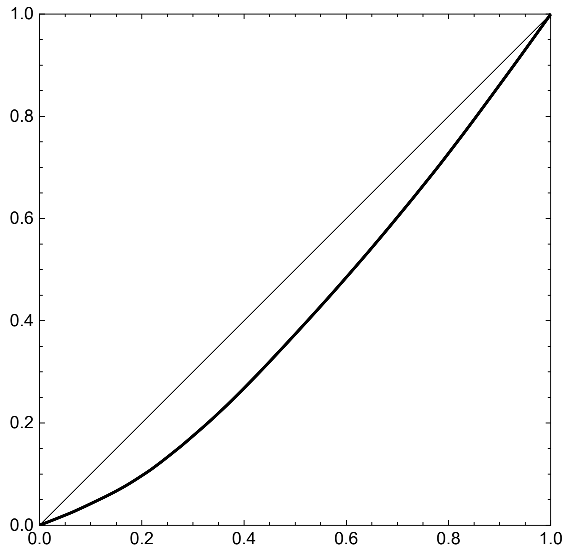
Of particular interest to economists has been the relationship between the least wealthy members of the population and their share of total wealth. If we form a cumulative sum of household wealth, we get a good representation of this relationship. (We prepend a 0 to the data so that we can meaningfully start at the origin.)

```
data = Accumulate[Prepend[sortedData, 0]];
ListLinePlot[data, DataRange → {0, 1}]
```



If we also scale cumulative total wealth to a proportion of the total, we get a representation of inequality known as a Lorenz curve. This maps each proportion of the population to the corresponding proportion of total wealth that it owns. In a perfectly egalitarian society, the Lorenz curve would lie along the 45 ° line. There is inequality to the extent that the Lorenz curve lies far from the line of perfect equality. (Here we use WL's `Prolog` option to add a 45 ° line to the plot, we set the aspect ratio of the plot to unity so that it is square, and we set the `PlotRangePadding` option to `None` in order to get a tight frame around the plot.)

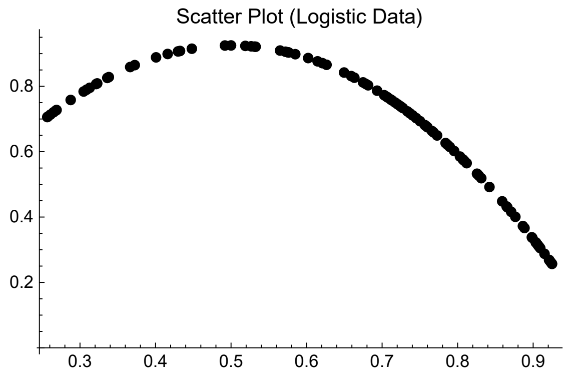
```
ListLinePlot[data / data[[-1]], DataRange → {0, 1},
 AspectRatio → 1, Frame → True, PlotRangePadding → None, ImageSize → 288,
 Prolog → Line[{{0, 0}, {1, 1}}]]
```



Scatter Plots vs Line Charts

A simple scatter plot displays a collection of points in Cartesian coordinates. The `ListPlot` command accepts a list of points and creates a scatter plot. Let us produce the points for our first scatter plot as all the adjacent pairs of points from our previous run-sequence data, which we can produce with the `Partition` command. (Review our previous discussion of the `Partition` command, if needed.)

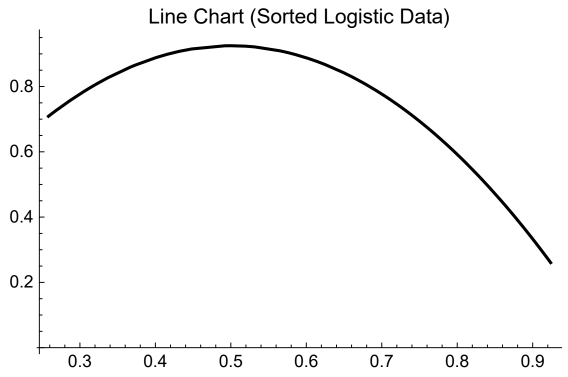
```
points = Partition[logisticData37, 2, 1];
ListPlot[points, ImageSize → 288,
 PlotLabel → "Scatter Plot (Logistic Data)"]
```



The difference between this scatter plot and our prior run-sequence plot, based on the same data, may prove rather startling on first encounter. Recall that data graphed above was produced by function iteration, which means that adjacent values are an input value and a corresponding output value. So this scatter plot actually illustrates the underlying function. This plot provides an entirely different perspective on the structure in the data than does our run-sequence plot. (In a later section, we elaborate on this observation by producing a cobweb plot.)

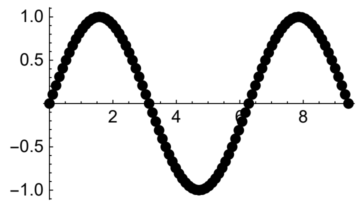
Points drawn from a function, as in this case, are often represented by a line chart. We can use \`ListLinePlot\` for this, as long as we first sort the points by the first coordinate. Luckily, the \`Sort\` command does lexicographic sorting on lists.

```
ListLinePlot[Sort@points, ImageSize → 288,
 PlotLabel → "Line Chart (Sorted Logistic Data)"]
```



Here is another comparison of two simple plots produced by \`ListPlot\` and \`ListLinePlot\`. We begin by generating some data, this time by applying the \`Sin\` function to a sequence of points from its domain. Since our first coordinates are already sorted, the two types of plot will yield similar results. This time, we use the \`Show\` command to combine our two plots, so that we have both lines and markers.

```
(* make up some data *)
domain = Range[0, 90] * Pi / 30;
sinPoints = Map[x ↦ {x, Sin[x]}, domain];
(* plot the data *)
sinPointsPlot = ListPlot[sinPoints, ImageSize → Small];
sinLinePlot = ListLinePlot[sinPoints, PlotStyle → Thin, ImageSize → Small];
Show[{sinPointsPlot, sinLinePlot}]
```



Linear vs Ratio Scales

The choice of scale along each axis can affect what we see in the data. WL offers convenience functions for changing the axes scales: \`ListLogPlot\` (for a ratio scale along the vertical axis, which linearizes the appearance of exponential functions), \`ListLogLinearPlot\` (for a ratio scale along the horizontal axis), and \`ListLogLogPlot\` (for a ratio scale along both axes, which linearizes the appearance of power functions).

The following example uses the life-table data that ships with Mathematica. We get the data as follows. We can evaluate \`ExampleData[]\` to see the main categories of example data. One of these categories is “Statistics”. We can evaluate \`ExampleData[“Statistics”]\` to find the datasets in this category. One of these is “USLifeTable2003”. We can retrieve that data as follows.

```
data = ExampleData[{"Statistics", "USLifeTable2003"}];
```

We need to know which columns to use. We could try looking at the original data:

```
ExampleData[{"Statistics", "USLifeTable2003"}, "Source"]
```

```
National Vital Statistics Reports, Vol. 54, No 14, April
  19, 2006. http://www.cdc.gov/nchs/data/nvsr/nvsr54/nvsr54_14.pdf
```

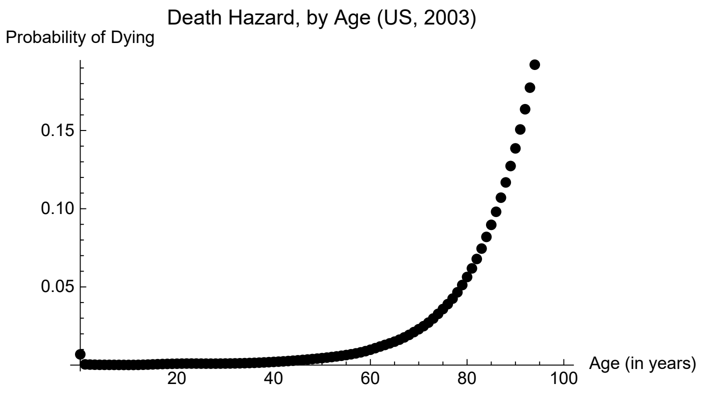
After a little browsing we can figure out what our data must be (Table 1, slightly reformated). However, there is an easier way: *Mathematica*’s example data are stored with helpful information, including column descriptions.

```
ExampleData[{"Statistics", "USLifeTable2003"}, "ColumnDescriptions"]
```

```
{Left endpoint of age interval., Right endpoint of age interval.,
 Probability of dying between AgeLower and AgeUpper.,
 Number surviving to AgeLower., Number dying between AgeLower and AgeUpper.,
 Person-years lived between AgeLower and AgeUpper.,
 Total number of person-years lived above AgeLower., Expectation of life at AgeLower.}
```

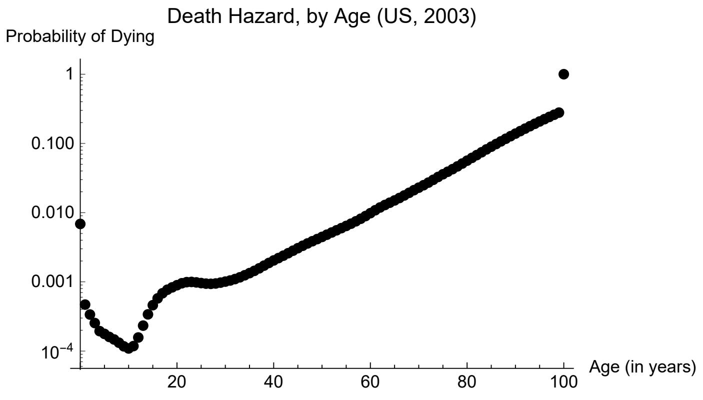
We are only going to use the first and third columns to plot the probability of dying against age.

```
plotdata = Map[x ↦ {x⟦1⟧, x⟦3⟧}, data]; (* get age and death prob *)
ListPlot[plotdata,
 AxesLabel → {"Age (in years)", "Probability of Dying"},
 PlotLabel → "Death Hazard, by Age (US, 2003)"]
```



Looking closely at the plot, it seems that some action at the tails is being concealed. We may try to expose that by adopting a ratio scale for the probability of dying, which is provided by `ListLogPlot`.

```
ListLogPlot[plotdata,
 AxesLabel → {"Age (in years)", "Probability of Dying"},
 PlotLabel → "Death Hazard, by Age (US, 2003)"]
```



Time-Series Line Charts

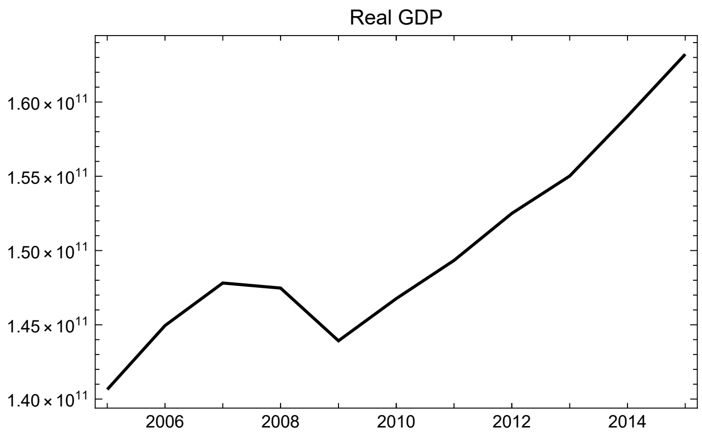
If sequential list indexes correspond to sequential points in time, the run-sequence plot is a time-series plot. In this case, we often use explicit dates or times as tick labels on the horizontal axis. Here we provide a minimal example. (For more details, see the documentation for the `DateListPlot` command.)

We can use `DateListPlot` to plot time series. Here, we begin by creating a series of date objects. We then use `Map` over these dates to produces lists of GDP and price-index entity properties, which we use fetch the associated GDP and price time series for one country (the US).

```
dates = Map[DateObject[{#}] &, 2004 + Range[11]];
gdpProps = Map[EntityProperty["Country", "GDP", {"Date" → #}] &, dates];
priceProps = Map[EntityProperty["Country", "PriceIndex", {"Date" → #}] &, dates];
```

Using these properties, we are going to fetch the data for a single country: the United States. (This requires an internet connection.) We then pair up each date with the associated real GDP, and we use `DateListPlot` to plot the resulting points.

```
ctry = Entity["Country", "UnitedStates"];
gdps = Map[ctry[#] &, gdpProps];
prices = Map[ctry[#] &, priceProps];
DateListPlot[Transpose[{dates, gdps / prices}], PlotLabel → "Real GDP"]
```



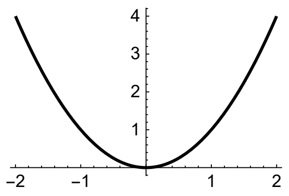
3.1.2 Function Visualization

For basic two-dimensional function plotting, we can use the `Plot` command. For three-dimensional function plotting, use the `Plot3D` command. These commands have many options for producing professional-quality charts. (Use `Options[Plot]` or `Options[Plot3D]` to list the options.) The available documentation is extensive and helpful.

Basic Function Plotting

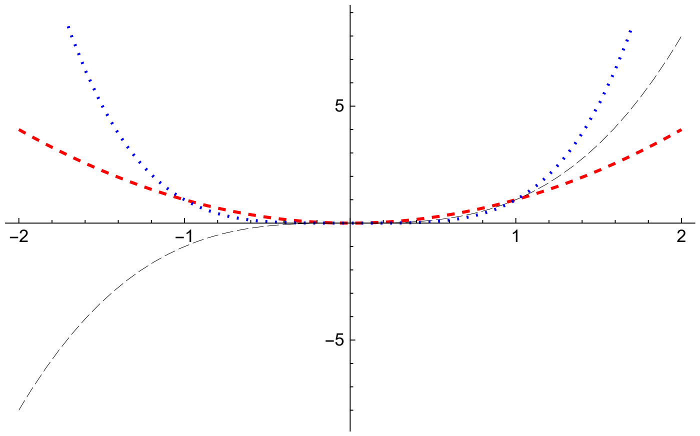
The `Plot` command provides simple 2D function plots. The first argument is an expression that involves a variable, and the second argument specifies the domain for that variable. The second argument is in the form {x,xmin,xmax}; in WL we usually call such expressions “iterators”, even when they apparently describe a continuous interval.

```
Plot[x², {x, -2, 2}, ImageSize → 144]
```



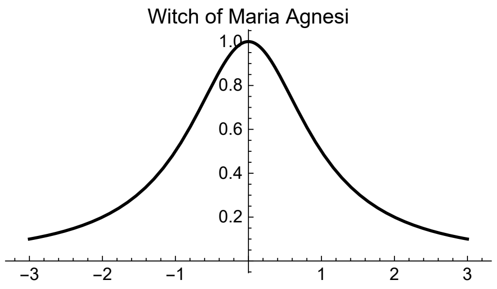
The first argument to the `Plot` command can be a list of expressions to be plotted together. The default behavior is to plot each expression with a different color, but you can also specify custom colors and other styles with the `PlotStyle` option.

```
Clear[x]
Plot[{x^2, x^3, x^4}, {x, -2, 2}, ImageSize → Medium,
 PlotStyle → {{Dashed, Red}, {Thin, Black}, {Dotted, Blue}}
]
```



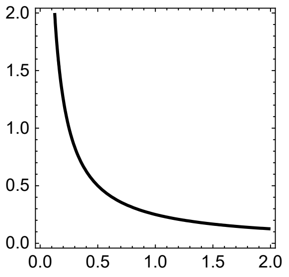
Functional relationships can also be plotted with the `ParametricPlot` command. The first argument must then be a list of two expressions, given a parametric representation of this relationship.

```
agnesi = ParametricPlot[{t, 1/(1 + t^2)}, {t, -3, 3}, ImageSize → 254,
  AspectRatio → 1/2, PlotLabel → "Witch of Maria Agnesi"]
```



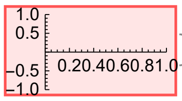
Functional relationships can also be implicitly defined by equations. Under suitable conditions, the expression $g(x, y) = 0$ implicitly defines y as a function of x (at least locally). However it may not always be possible to determine an explicit function f such that $y = f(x)$. Nevertheless, we can analyze the implicit relationship. WL allows us to plot these with the `ContourPlot` command (which replaces an earlier `ImplicitPlot` command). In the following example, we draw a level curve (also called a contour line or an isoline) for the expression $\sqrt{x\,y}$, which represents a functional relationship between y and x. This level curve could represent a firm’s isoquant or a consumer’s indifference curve.

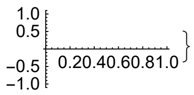
```
ContourPlot[Sqrt[x * y] == 0.5, {x, 0, 2}, {y, 0, 2},
 ImageSize → 144]
```



Often it is useful to combine plots, which is possible with the `Show` command. For example, consider our cobweb plot for the logistic map. It becomes much easier to understand if we add a function plot, so that we can see that each new value is determined by applying the logistic map to the previous value.

```
gLogisticMap02 = Plot[logisticMap[x], {x, 0, 1},
   PlotStyle → {Thin, Gray, Dashed}];
Show[{gCW02, gLogisticMap02}]
```

Show: Could not combine the graphics objects in Show[{gCW02,  }].

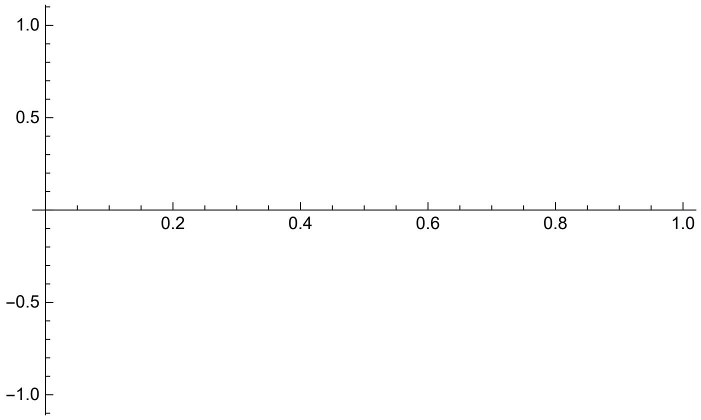
Show[{gCW02,  }]

Advanced

Adaptive Sampling

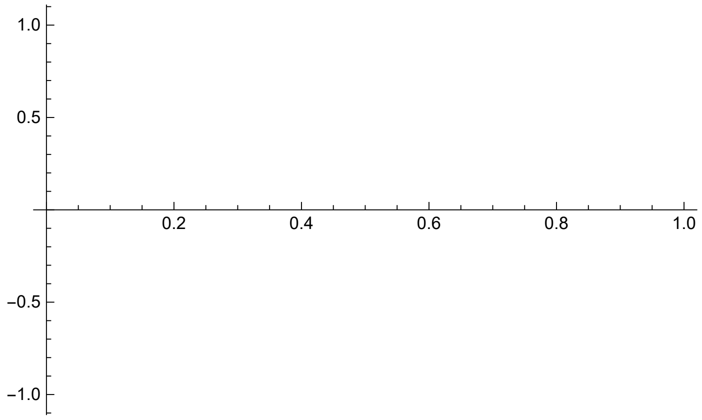
Function plots are drawn as a sequence of closely spaced, connected line segments. Many plotting programs simply evenly partition the domain over which we wish to plot. For example, if we plot a logistic map over the domain [0, 1] with 11 points, then the function will be evaluated at (0, 1/10, 2/10, ..., 1). We can get close to this outcome if we set `MacRecursion` to `0` while specifying the `PlotPoints`. (In order to make this easier to see, we also use the `Mesh` option to display all the plotted points.

```
g11a = Plot[logisticMap[x], {x, 0, 1}, PlotStyle → {Thin, Black},
  PlotPoints → 11, MaxRecursion → 0, Mesh → All]
```



However, this is not the default behavior of a plot. In order produce nicer looking plots, `Plot` uses an adaptive sampling algorithm. This means that more points are added wherever the function is changing curvature. We can see this by leaving `PlotPoints` and `MaxRecursion` at their default values. In the new function plot, we see the points tend to be more uniformly spaced when the function slope is not changing much. (If you want to see exactly which points are being plotted, you can look at the `InputForm` of the plot.)

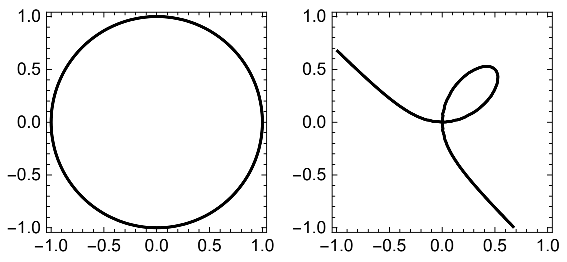
```
g11b = Plot[logisticMap[x], {x, 0, 1}, PlotStyle → {Thin, Black},
  Mesh → All]
```



Basic Curve Plotting

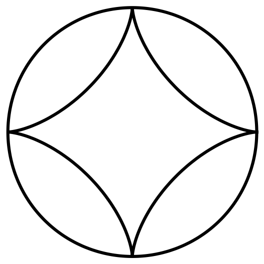
Level sets do not always produce functional relationships. We can still use the `ContourPlot` command to plot a level set.

```
GraphicsRow[{
  ContourPlot[x^2 + y^2 == 1, {x, -1, 1}, {y, -1, 1}],
  ContourPlot[x^3 + y^3 == x * y, {x, -1, 1}, {y, -1, 1}]
 }, ImageSize → 288]
```



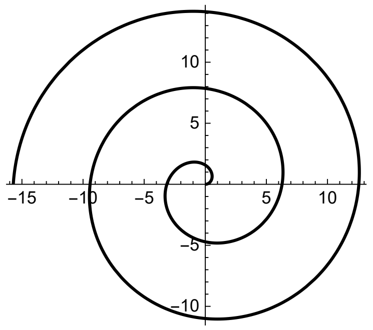
The `ParametricPlot` command provides a very general approach to curve plotting: it can be used to draw two-dimensional curves that do not represent functional relationships, such as a circle or an astroid. Here we use the `Show` command to combine a circle plot and an astroid plot into a single graph.

```
circle = ParametricPlot[{Cos[t], Sin[t]}, {t, 0, 2 π}, Axes → False];
astroid = ParametricPlot[{Cos[t]^3, Sin[t]^3}, {t, 0, 2 π}, Axes → False];
Show[{circle, astroid}, ImageSize → 144]
```



We can also plot the radius as a function of the angle, using `PolarPlot`. The simplest case may be when the radius is proportional to the angle. The produces the following plot, known as the spiral of Archimedes.

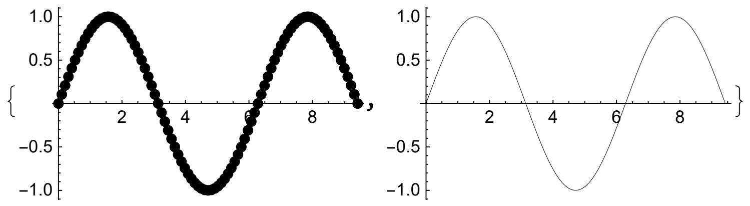
```
PolarPlot[θ, {θ, 0, 5 * π},
 ImageSize → 188]
```



Multiple Subplots and Shared Styles

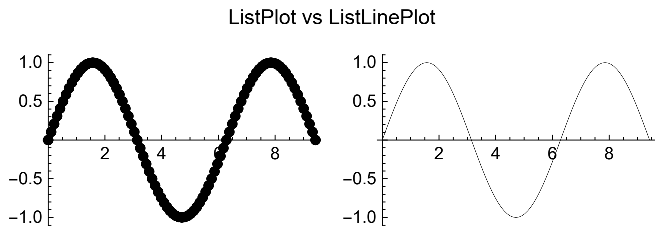
The `ListPlot` and `ListLinePlot` commands return graph objects that can be manipulated like other objects in WL. For example, you can make a list of them.

```
sinPlots = {sinPointsPlot, sinLinePlot}
```



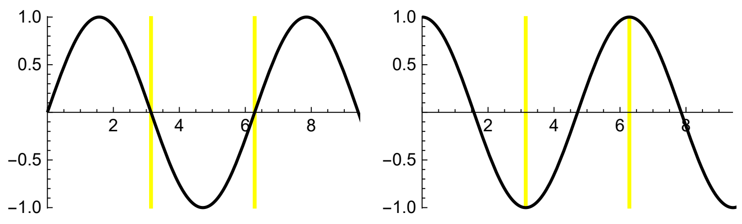
This can be very useful, but for pure display purposes, there are more aesthetic options.   Recall that we can use `GraphicsRow`, `GraphicsColumn`, and `GraphicsGrid` to group graphics objects. This applies to our charts, so we can convert a list of plots into a single chart with subfigures. For example,

```
GraphicsRow[sinPlots, PlotLabel → "ListPlot vs ListLinePlot"]
```



Fine control over plots can be achieved with the many options, which you can examine by giving the command `Options[Plot]`. As with `ListPlot` and `ListLine` plot, if you have a style you would like to apply to multiple plots, you can collect the options in a named `Sequence`, say `myoptions`, and then use it when you wish. A common need is the `ImageSize` option, illustrated above. Here we use the `ImageSize` option to specify the width and height of each plot, in points. In the next example, we do this, and we illustrate several other options: `PlotRange`, `GridLines`, and `GridLinesStyle`. These are fairly self explanatory, but as usual the Mathematica documentation is extensive and high quality.

```
myoptions = Sequence[
   ImageSize → 180, PlotRange → {{0, 3 * Pi}, {-1.01, 1.01}},
   GridLines → {{Pi, 2 Pi}, {}}, GridLinesStyle → Directive[Thick, Yellow]
  ];
GraphicsRow[{
  Plot[Sin[x], {x, 0, 10}, Evaluate@myoptions],
  Plot[Cos[x], {x, 0, 10}, Evaluate@myoptions]
 }]
```



Advanced

An alert reader may have notice a subtle difference from `ListPlot` in our use of `myoptions` with `Plot`: we had to use `Evaluate` to evaluate it. This is because, unlike `ListPlot` or `ListLinePlot`, the `Plot` command has an attribute called `HoldAll` which prevents evaluation of its arguments. To see the attributes of a command, use the `Attributes` command. (All attributes are fully covered in the Mathematica documentation; also see the Wolfram Language Tutorial entitled Attributes.)

```
Attributes[Plot]
```

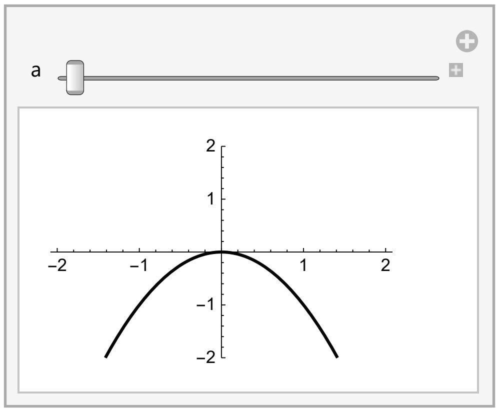
```
{HoldAll, Protected, ReadProtected}
```

Advanced

Manipulable Plots

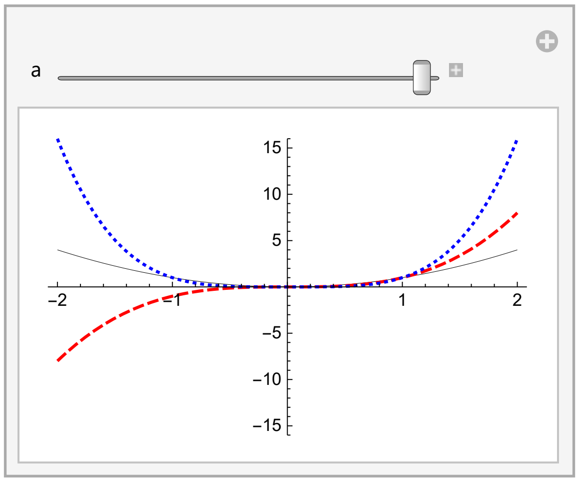
Use the `Manipulate` command for dynamic exploration of function plots. This command adds a control which allows user interaction with the manipulated expression. The expression can be any WL expression, including a plot. So, for example, we can parameterize a function plot, and allow user manipulation of the parameter value. Here is a simple example, plotting a quadratic.

```
Clear[a, x]
Manipulate[
 Plot[a * x^2, {x, -2, 2}, PlotRange → {-2, 2}, ImageSize → Small],
 {a, -1, 1}]
```



Manipulable plots can be as complex as we wish. Here is a slightly more complex example, plotting three monomials conditional on a user-manipulable coefficient.

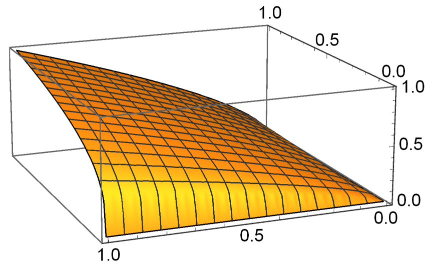
```
Clear[a, x]
Manipulate[
 Plot[{a * x^2, a * x^3, a * x^4}, {x, -2, 2}, PlotRange → {-16, 16},
  ImageSize → 250, PlotStyle → {{Thin, Black}, Red, Blue}],
 {{a, 1}, -1, 1}]
```



Introduction to 3D Function Plotting

The `Plot3D` command provides simple three-dimensional surface plots of functions. The first argument is an expression that involves two variables, and the second and third arguments specify the domains for those variables. The second and third arguments are a standard WL domain specification (e.g., {x,xmin,xmax}). We can specify many additional options in order to adjust the plot characteristics. The most important of these may be `ViewPoint`, which sets the point in space from which the plot is viewed.

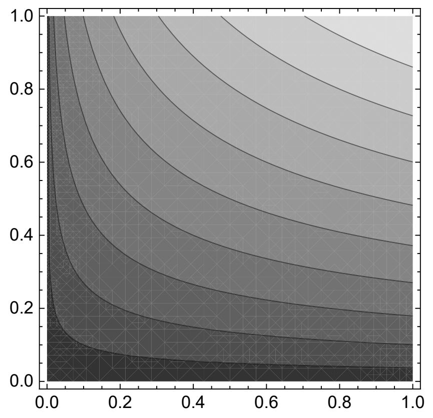
```
Clear[k, n]
cd = k^α * n^(1 - α) /. {α → 0.3}; (* Cobb-Douglas function *)
Plot3D[cd, {k, 0, 1}, {n, 0, 1}, ViewPoint → {-5, 2, 1.5},
 PlotTheme → "Default", ImageSize → 216]
```



Three-dimension plots can be rotated with a mouse to produce the most useful perspective on the image. If you would like to ensure that you can reproduce this perspective in the future, you can determine the options you have selected interactively with the `AbsoluteOptions` command. For example, manipulate the above graphic to get the perspective you prefer, and then select and copy the result to the clipboard. In a new cell, type `AbsoluteOptions[]` and then paste your graphic from the clipboard into this command. When you evaluate this expression, WL will produce all the options set for the graphic. Pay particular attention to the `ViewPoint` and the `ViewCenter`. In order to develop your understanding of these options, consider viewing Wolfram's free training course "Graphics & Visualization: Advanced 3D Graphics" (written by Yu-Sung Chang).

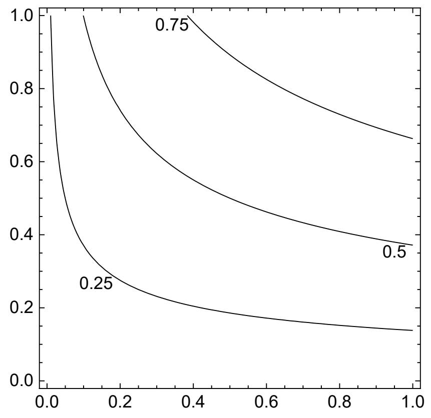
Sometimes it is convenient to view three dimensional functions as contours.

```
ContourPlot[cd, {k, 0, 1}, {n, 0, 1}, ImageSize → 216]
```



As usual with WL plotting commands, extensive customization is possible. For example, the following example turns off the contour shading, picks the contours we want to see, and adds labels to these contours. For details, see the documentation for `ContourPlot` and its options.

```
ContourPlot[cd, {k, 0, 1}, {n, 0, 1},
 Contours → {0.25, 0.5, 0.75},
 ContourLabels → (Text[#3, {#1, #2}, {1, 1}] &),
 ContourShading → None, ImageSize → 216]
```

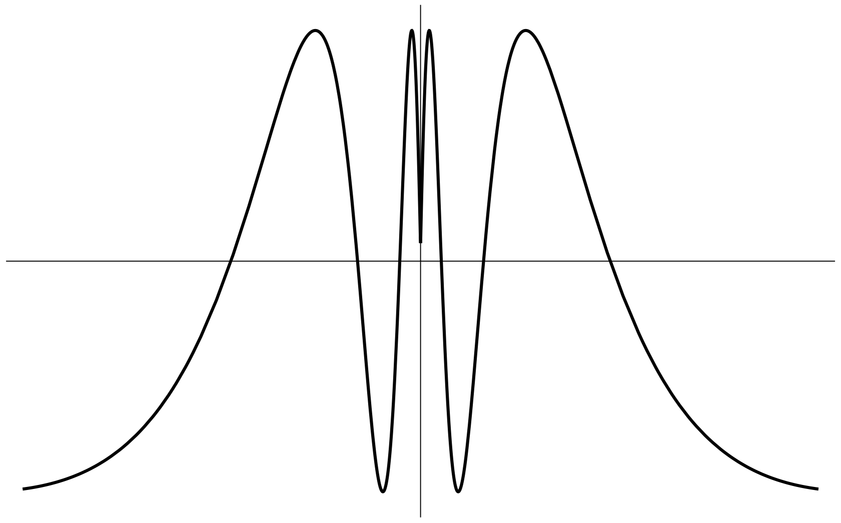


Advanced

3.1.3 Insets

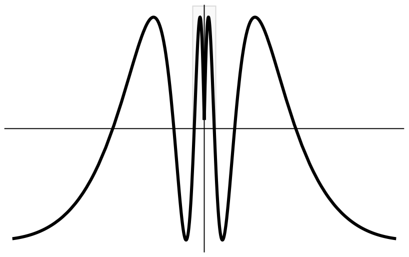
In this subsection, we consider a fairly common but somewhat advanced plotting need: the addition of insets to a plot. WL provides the `Inset` command for this. Here we implement the basic idea of making a magnified plot over a subset of the plot domain in order to show greater detail. So let us begin with a function plot that has some detail that is perhaps a bit difficult to discriminate visually.

```
ClearAll[f]
f = x ↦ -Sin[1 / (.08 + Abs[x])]; (* function to plot *)
gmain = Plot[f[x], {x, -0.5, 0.5}, PlotRange → All, PlotStyle → Black,
  Ticks → None, ImageSize → 432]
```



We will take a closer look at the plot around 0. To illustrate, let us highlight the region we will examine more closely. Use the `Prolog` option to add background to the region we will emphasize. We can make a separate plot representing the highlighted region. One option is to display these side by side, using `GraphicsRow`.

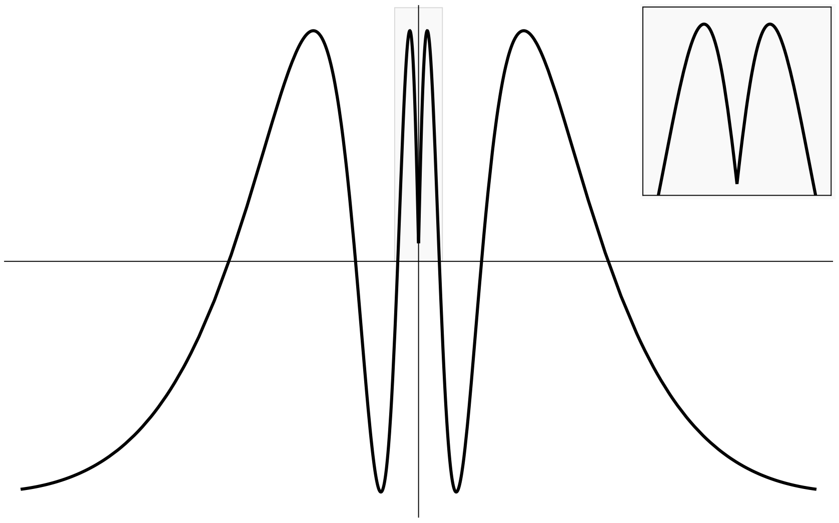
```
{rectll, rectur} = {{-0.03, 0}, {0.03, 1.1}}; (* bounds for gray rectangle *)
rect = {FaceForm[Lighter[Gray, 0.95]], EdgeForm[LightGray], Rectangle[rectll, rectur]};
gzoom =
  Plot[f[x], {x, First@rectll, First@rectur}, PlotRange → {Last@rectll, Last@rectur},
   Axes → False, Frame → True, FrameTicks → None, Background → Lighter[Gray, 0.95]];
GraphicsRow[{Show[gmain, Prolog → rect], ginset}, ImageSize → 432]
```



```
ginset
```

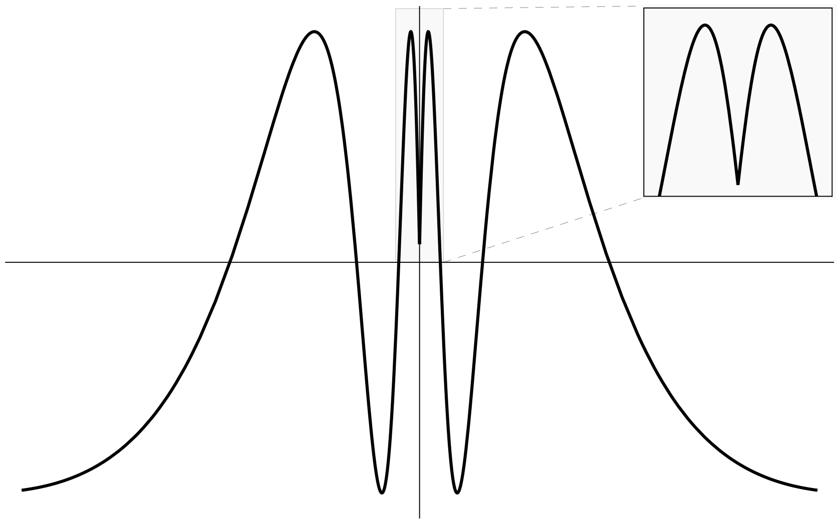
A more popular option is to use an inset in the original graph, using the `Inset` command, so that we can continue to display it at full size. This is going to require a bit of trial and error, because we want to place the inset so that it minimally interferes with the main plot. Here we decide to place the inset in the top right corner. We accomplish this using `Scaled`, which provides coordinates scaled from 0 to 1 across the entire plot range. For example, the scaled coordinates (1, 1) designate the top right corner of the plot range.

```
insetSize = {100, 100}; (* inset width and height, in points *)
insetXY = Scaled[{1, 1}]; (* location of inset, in scaled coordinates *)
inset = Inset[Show[gzoom, ImageSize → insetSize, AspectRatio → Full, ImagePadding → 1],
   insetXY, {Right, Top}];
Show[gmain, Prolog → {rect, inset}]
```



The result at this point is understandable and rather acceptable. However, a common convention is to supply an explicit visual link between the inset and the region it represents, usually by drawing connecting lines. This poses some difficulties because WL does not supply us with the bounding box for the inset. However, because we provided a location along with a height and a width for our inset, we can compute the bounding box ourselves. To do so, we take advantage of the `Offset` command, which locates an offset (in points) relative to a location, which we can specify in scaled coordinates.

```
(* compute the points anchoring the lines *)
pt1a = rectur;
pt1b = Offset[{-1, 0} * insetSize, insetXY];
pt2a = {First@rectur, Last@rectll};
pt2b = Offset[{-1, -1} * insetSize, insetXY];
lines = {Thin, Gray, Dashed, Line[{{pt1a, pt1b}, {pt2a, pt2b}}]};
(* add the lines to the plot *)
Show[gmain, Prolog → {lines, rect, inset}]
```



Chapter 4: Basic Uses of Lists: Sets, Vectors, and Matrices

This chapter introduces some basic uses of lists. In WL, we can readily use lists to represent sets, vectors, and matrices.

4.1 Lists as Sets

WL does not have a separate set type. Instead, it defines set operations on lists. One may also use associations for certain set features, but we will focus on the traditional use of lists.

4.1.1 Elementhood and Inclusion

A set is specified by fully describing the elements that belong to it. Some sets include others, in the sense of containing all of the elements of the others. Some sets even contain other sets as elements. This section explores various notions on containment and inclusion. It also introduces basic set operations and set relations. For more detail, see the Wolfram Language Guide entitled Operations on Sets.

Elementhood

Like a list, a mathematical set is a collection of elements. However, lists differ substantially from mathematical sets. A computational list is really a kind of sequence: the order of the elements is significant, as are duplicate elements. In contrast, a there is no notion of element order or duplication in a mathematical set.

We can usually ignore the order of elements when we use a list to represent a set. Sometimes we can also ignore the possibility of duplicates. For example, duplicates do not change the results of elementhood testing, even if it may be slower to test elementhood on a list that contains duplicates. When duplicate elements are inconvenient, we may explicitly discard duplicates with the `DeleteDuplicates` command. This returns a new list, which does not contain any duplicate elements. (The original list is unchanged.) For example, suppose `s1` below is supposed to represent a set, so that we want to ignore the duplicate elements. Then we can create a new representation of the same set, `s2`, by deleting all the duplicate elements.

Expression	Result
`s1={1,2,2,3,3,3}`	`{1, 2, 2, 3, 3, 3}`
`s2=DeleteDuplicates[s1]`	`{1, 2, 3}`
`s1`	`{1, 2, 2, 3, 3, 3}`

Sometimes we want to know the cardinality of a set, which is roughly speaking the number of elements it contains. We can use the `Length` command to determine the number of elements in a list, but if we want the result to represent the cardinality of a set, we must first delete duplicates. We might implement this observation in the following function. (This definition uses function composition and a point-free style. If this looks unfamiliar, you may wish to review our earlier discussion of function composition.)

```
cardinality = Length@*DeleteDuplicates;
```

Try out our new `cardinality` function:

Expression	Result
s1	{1, 2, 2, 3, 3, 3}
cardinality[s1]	3

We are often interested in whether or not a set contains a specific item. An elementhood test determines if a set contains a specified item. For example, given a list of numbers, we may wish to determine whether a particular number is in the list. Recalling our discussion of `Apply` (with its `@@` shorthand) and `Map` (with its `/@` shorthand), here is a very naive implementation of such a test.

```
contains01 = {set, item} ↦ Or @@ (# === item & /@ set);
```

Mathematically, this seems perfectly proper. Let us conduct some simple tests of `contains01` on a list of randomly generated integers between to check that it works as it should.

Expression	Result
lst01=RandomInteger[100,5]	{89, 27, 27, 16, 21}
contains01[lst01,101]	False
contains01[lst01,Last@lst01]	True

However, the second test reveals a computational problem with this elementhood test: we do a first iteration over the list to produce a Boolean list from the desired comparison, and then we iterate over this Boolean list to do the `Or` comparisons. We used a very short list, so this hardly matters. But real-world applications often involve long lists. Iterating twice when once would do becomes computationally costly. If we recall our discussion of `Fold`, we can find a better (although still naive) implementation of an elementhood test.

```
contains02 = {set, item} ↦ Fold[#1 || #2 === item &, False, set];
```

Let us conduct some simple tests of `contains02` our a list of randomly generated integers between to check that it works as it should.

Expression	Result
contains02[lst01,101]	False
contains02[lst01,First@lst01]	True

This is better: now we only iterate through the list once. But the second test illustrates why this is still a naive approach: even though we could stop after testing a single item, our implementation will fold over all the items in the list. We would like to “short circuit” the evaluation: stopping as soon as we encounter the item. WL provides for this with the `AnyTrue` command. This will “short circuit” the evaluation of the tests as soon as one item tests true.

```
contains03 = {set, item} ↦ AnyTrue[lst, # === item &];
```

We have finally produced a decent solution for a single elementhood test. Of course, since membership testing is such a common need, we should suspect that WL already provides an elementhood test. In fact, WL provides the `MemberQ` command. This takes two arguments: a list (representing a set), and a proposed element of the set. Here are some example uses.

Expression	Result
s1	{1, 2, 2, 3, 3, 3}
MemberQ[s1,101]	False
MemberQ[s1,#]&/@{0,1,1.0}	{False, True, False}

The last example uses `Map` (in the `/@` operator) to perform many elementhood tests at one go. Note that the `MemberQ` command tests for equivalence, not just for equality. (Recall that equal exact and approximate numbers are not equivalent.) As long was this is the behavior we desire in an elementhood test, we can just use `MemberQ`. If we prefer to use an equality test, we now know how to write our own elementhood test.

Advanced

Associations as Set Representations

Sometime we have many items we wish to test for elementhood in a given set. If we have access to all the items at once, we can gather them together and use the `Intersection` command. (See below.) If gathering the items into a list is costly or problematic, it can pay to incur the cost of creating a hash table of the set elements. Element lookup is much faster in a hashtable than in a list. Although WL does not have an explicit set type, we can use an association list for this. Here we illustrate how to do this.

For convenience, we use `AssociationMap` to create an association where the keys are our list elements and the values are all `True`. To illustrate, we will work with a list of 100 integers chosen randomly in [1..1000]. (The resulting set may have fewer than 100 members.) We will take advantage of the `Lookup` command command, which allows a default value as the third argument. In this example, we test whether 10 is in the set of keys. If it is found by `Lookup`, we will get the associated value, which is `True`. If it is not found, we will get the default value that we supplied, which s `False`.

```
data = RandomInteger[1000, 100];
s3 = AssociationMap[True &, data];
Lookup[s3, 10, False]
```

```
False
```

Advanced

Elementhood for Special Sets

Usually we use the `MemberQ` command to test for elementhood. However, for some special sets, we can use the `Element` command (or its `∈` shorthand). Here are a few examples; see the documentation for details. Note that approximate numbers are never assumed to be exact representations, and WL expresses the resulting ambiguity by simply returning the expression.

noclass, nopdf

	Integers	Rationals	Reals	Complexes
1	True	True	True	True
1.	1. ∈ Integers	1. ∈ Rationals	True	True
1. + 0. ⅈ	1. + 0. ⅈ ∈ Integers	1. + 0. ⅈ ∈ Rationals	1. + 0. ⅈ ∈ Reals	True

Quantification

We are often interested in claims about some or all of the elements in a set. These claims are often stated in terms of predicates. For example, we might claim that all numbers in a set *S* are positive. If we let *p*[*x*] mean that element *x* is a positive number, then we may write this claim as $\forall x \in S,\ p[x]$. Or we might claim that there is at least one number in *S* that is positive. We might write this claims as $\exists x \in S,\ p[x]$. How would we test such claims?

Given our previous discussion of the `Map` command, it may appear to be an obvious candidate for this task. If we map a predicate over our set elements, the result is a boolean list (i.e., every value is either `True` or `False`). We can then ask whether all or any the elements are true. To determine if all are true, we can use the `Apply` command with the `And` command. (From the documentation, we see that the `And` command accepts more than two arguments.) Similarly, to determine if at least one is true, we can use the `Apply` command with the `Or` command. Here is a simple example, using the `@@` shorthand operator form of `Apply`. (We use WL's `Positive` predicate to test for positivity.)

Expression	Result
`n5=Range[5]`	`{1, 2, 3, 4, 5}`
`tests=Positive/@n5`	`{True, True, True, True, True}`
`And@@tests`	`True`
`Or@@tests`	`True`

This approach offers a nice match to the mathematical thinking about quantification over finite sets: every outcome is true iff their conjunction is true, while at least one outcome is true iff their disjunction is true. However, a moment's reflections suggests that this use of `Map` is computationally inefficient: it creates an entire list of intermediate results, where we need only produce the final result. Since the `And` and `Or` commands are naturally considered to be generalizations of the underlying binary logical operators, it may occur to use to use `Fold` to avoid this inefficiency. (Recall that `Fold` reduces a list to a single value by repeatedly applying a binary function.) We will deal with the corner case of an empty set as follows: all elements of an empty set satisfy the the predicate, but there does not exist any element in the empty set that satisfies the predicate. (If this feels odd, please review material implication.)

Expression	Result
`Fold[{b,x}⟼And[b,Positive[x]],True,n5]`	`True`
`Fold[{b,x}⟼Or[b,Positive[x]],False,n5]`	`True`

While this solution is much more satisfactory, it still fails to "shortcut" the evaluation as soon as an answer is determined. WL offers builtin solutions that do shortcut, in the form of the `AllTrue` and `AnyTrue` commands.

Expression	Result
`AllTrue[n5,Positive]`	`True`
`AnyTrue[n5,Positive]`	`True`

Advanced

If we want to see that this approach really uses appropriate shortcutting, we can trace the evaluation.

```
Trace[AnyTrue[n5, Positive]]
```

```
{{n5, {1, 2, 3, 4, 5}}, AnyTrue[{1, 2, 3, 4, 5}, Positive], {Positive[1], True}, True}
```

Inclusion (Subsets)

We say set *A* includes set *B* if every element of *B* is also an element of *A*. In this case, we also say *A* is a superset of *B* and write $A \supseteq B$. More commonly, we say *B* is a subset of *A* and write $B \subseteq A$.

As an exercise, let us build our own inclusion test, using the `MemberQ` command for testing element-hood. Working directly with the definition of subset, we might use `AllTrue` with an appropriate element-hood test to come up with the following test of whether `set1` includes `set2`. (Review our earlier discussion of the `Function`, `AllTrue`, and `MemberQ` commands, if necessary.)

```
includes = {set1, set2} ↦ AllTrue[set2, MemberQ[set1, #] &];
```

Let us try out our new `includes` function to make sure it works as expected.

Expression	Result
s1=Range[5]	{1, 2, 3, 4, 5}
s2=Range[2]	{1, 2}
includes[s1,s2]	True
includes[s2,s1]	False

By definition, the empty set is a subset of every set, as is the set itself. To be careful, we should also test these corner cases.

Expression	Result
includes[s1,{}]	True
includes[s1,s1]	True

As expected, our `includes` function works just fine. The exercise of creating it is useful for reinforcing the concept of inclusion, but when it comes to common needs, very good implementations often exist within WL. In this case, the `SubsetQ` command fits the bill. Given the name of this function, perhaps the order of arguments is surprising: as with the `includes` function that we created above, `SubsetQ` tests whether its second argument is a subset of its first argument.

Expression	Result
SubsetQ[s1,s2]	True
SubsetQ[s2,s1]	False

We can use the `Timing` command to compare the computational speed of the two approaches. To show this, we will work with larger sets of values, creating a set s_1 as a sizeable range and then creating a second set s_2 as a random sample from this range. (This ensures that $s_2 \subseteq s_1$.) We find that the builtin command executes much faster than our custom function. (This will almost always be the case.)

```
s1 = Range[10000];  s2 = RandomSample[s1, 1000];
```

Expression	Result
`Timing[includes[s1,s2]]`	`{0.5625, True}`
`Timing[SubsetQ[s1,s2]]`	`{0., True}`

Bitwise Operations

Readers who have studied regression analysis are likely to have encountered indicator variables, also known as dummy variables. An indicator variable takes on a value of 1 if an observation belongs in a specified subset of all observations and a value of 0 otherwise. A value of 1 indicates that the observation belongs to the subset. We often generate the values of an indicator variable by applying an indicator function to a dataset.

An indicator function for a set is just an elementhood test that returns 0 and 1 instead of `False` and `True`. The `Boole` command turns `False` to 0 and `True` to 1, so we can leverage the existing `MemberQ` command to produce an indicator function corresponding to any subset. For the moment, we will abstract from any underlying predicate, and build an indicator function based on the subset itself. Here we define a function whose return value is also a function.

```
indicatorFactory = set ↦ Boole[MemberQ[set, #]] &;
```

Our `indicatorFactory` returns a *closure*: a function that keeps track of the set that was used to create it. (We might say that a function produced this way captures the set that was used to create it.) So we can can use `indicatorFactory` to create an indicator function for any set we wish; it is a factory for such functions. Here are two examples.

Expression	Result
`n5=Range[5]`	`{1, 2, 3, 4, 5}`
`indicatorFactory[{1,2,3}]/@n5`	`{1, 1, 1, 0, 0}`
`indicatorFactory[{3,4,5}]/@n5`	`{0, 0, 1, 1, 1}`

If we index the elements of a set, then we can use a bit array to represent any subset of that set. A natural way to index set elements in WL is to simply index each element by its position in a list. We can think create a bit array -- a list of zeros and ones -- that is the same length as the universal set, with a one or zero in each position to indicate whether or not the element is in the subset under consideration. Furthermore, we can give an integer representation to a bit array by considering it to be a base-2 representation of an integer.

For example, the decimal number 28 has the base-2 representation $11\,100_2$. We can therefore use the base-2 digits of the decimal number 28 to pick from an indexed set the elements of the set that correspond to nonzero elements of the digit list.

Expression	Result
`r5=Range[5]`	`{1, 2, 3, 4, 5}`
`ind=IntegerDigits[28,2]`	`{1, 1, 1, 0, 0}`
`Pick[r5,ind,1]`	`{1, 2, 3}`

Now for any set, we have a way to represent each subset with a single integer. We do need to be careful to generate the right number of digits, which must equal the number of elements in our set. (The

`IntegerDigits` command takes a third argument that controls this.) As a bonus, this gives us one nice way to pick a random subset of any set: we simply generate a random integer and then use the 1s in its base-2 representation to determine elementhood.

Set Operations and Venn Diagrams

The set operation commands `Union`, `Intersection`, and `Complement` conveniently discard duplicates from each set. Here is a table illustrating the results of these operations. (Enter the operator for union as `ESCunESC`; etner the operator for intersection as `ESCinterESC`.)

Expression	Result
s1={1,2,2,3,3,3}	{1, 2, 2, 3, 3, 3}
s2={3,3,3,4,4,5}	{3, 3, 3, 4, 4, 5}
s1⋃s2	{1, 2, 3, 4, 5}
s1⋂s2	{3}
s1~Complement~s2	{1, 2}

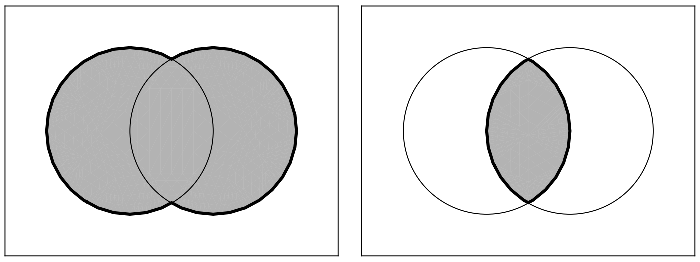
WL does not currently (version 11.0) offer a `SymmetricDifference` command, but for this you can use the union of the set differences.

```
Complement[s1, s2] ⋃ Complement[s2, s1]
```

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

Venn diagrams are useful for illustrating simple set operations. Here is an illustration, using the `RegionPlot` command. WL allows us to use a disk as a region, and we can form region unions and intersections with `RegionUnion` and `RegionIntersection`. Here we offer an illustration, during which we again utilize the `Sequence` command, which can be used to store a collection of shared options in a variable. (Remember, you must `Evaluate` the `options` variable inside your plot command.) Here we additionally make use of the `Epilog` option, which adds items to our graphic object after everything else is drawn. (We want the circles drawn last, so that they show up on top of the shaded regions.)

```
d1 = Disk[{3, 3}, 2]; d2 = Disk[{5, 3}, 2];
options = Sequence[
   PlotRange → {{0, 8}, {0, 6}}, AspectRatio → Automatic,
   FrameTicks → None,
   Epilog → {Circle @@ d1, Circle @@ d2}
  ];
GraphicsRow[{
  RegionPlot[RegionUnion[d1, d2], Evaluate@options],
  RegionPlot[RegionIntersection[d1, d2], Evaluate@options]
 }]
```



4.1.2 Set Building

Set building is a common need in both theoretical and empirical work. In this section we lay out some general principles, which we then use to explore dataset queries.

Selective Set Building

We often represent sets with set builder notation: $\{f(x) \mid x \in X,\ p(x)\}$. Here X is the set of elements that we consider (the domain of discourse), p is a predicate over this domain, and f is a transformation over this domain. This representation has two specializations: substitutive set building, and selective set building. In substitutive set building, p always returns `True` and can therefore be ignored. In selective set building, f is the identity function and can therefore be ignored.

Substitute set building is sometimes called transformation or recoding. For finite sets given a list representation, we effectively covered substitutive set building when we discussed the `Map` command. We need only be aware of the possibility that different elements may have common substitutions, which we can address with the `DeleteDuplicates` command. (When working with datasets, we will often retain some kind of unique record identifier, so that exact duplicates are never created.)

In this section, we cover selective set building, often called predicative set building, selection, or *filtering*. We will again use lists to represent sets. Sometimes we have a set from which we want a sublist that meets an inclusion criterion. We would like to filter out all the elements that do not satisfy this criterion. If the list we are filtering is a proper representation of a set (i.e., if it contains no duplicates), then the result of the filtering operation will also contain no duplicates. In this case, list filtering is equivalent to predicative set building.

In fact, the `Map` command is capable of selective set building: we can substitute `Nothing` for unwanted elements, and all instances of `Nothing` will be removed automatically from our result. For example, if we might select from a set of numbers only those less than 10 as follows.

```
(x ↦ If[x < 10, x, Nothing]) /@ Range[50]
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

As another example, suppose we want to find the primes in a set of numbers. For concreteness, let this set be the first 10 positive integers. Use the `PrimeQ` command to test for primacy. (Most of WL’s univariate predicates end in “Q”, with a small number of exceptions for sign determination: `Negative`, `NonNegative`, `Positive`, and `NonPositive`.) We again use `Map` to produce conditional selection. Recall that this is possible because the special `Nothing` value is dropped from the result.

```
If[PrimeQ[#], #, Nothing] & /@ Range[10]
{2, 3, 5, 7}
```

However, we ordinarily use `Map` like this only when we want to manipulate the selected values—e.g., when we also want to do substitutive set building. WL offers two specialized approaches to selection: ordinary filtering with the `Select` command, and pattern-based filtering with the `Cases` command. Ordinary filtering applies a predicate (i.e., a boolean function) to each element; elements evaluating to `False` do not make it through the filter. Pattern-based filtering tests each element to determine if it matches a pattern. To maximize comparability, our pattern below includes a predicate representing a boolean test, introduced by a question mark. (For more details, see the documentation for the `PatternTest` command. We could alternatively use a named pattern with the predicate as a constraint; for details, see the Wolfram Language Tutorial entitled Putting Constraints on Patterns.) We end up with two very similar ways of doing the same job of selective filtering.

nopdf, noclass

```
... Select: Nonatomic expression expected at position 1 in Select[set01, PrimeQ].
```

Expression	Result
Select[set01,PrimeQ]	Select[set01, PrimeQ]
Cases[set01,_?PrimeQ]	{}

Both `Select` and `Cases` have an operator form. In the present example, the choice of form is purely a matter of taste.

nopdf, noclass

```
... Select: Nonatomic expression expected at position 1 in Select[PrimeQ][set01].
```

Expression	Result
Select[PrimeQ]@set01	Select[PrimeQ][set01]
Cases[_?PrimeQ]@set01	{}

We see that `Select` and `Cases` are often close substitutes, so that the choice is largely a matter of taste. `Select` is perhaps more likely to seem familiar, as similar filtering facilities are available in many languages. For numerical lists, `Select` often seems to be the more natural choice.

Criterion-based filtering is a basic need in data manipulation. Use the `Select` command or the `Cases` command to construct smaller sets from larger sets based on an inclusion criterion. For example, we might retrieve data from a life table, and be interested only in the data for children under the age of 12. Mathematica includes a sample life table with its example data, `USLifeTable2003`. The data is provided as a list of lists, where each inner list is a record for a one-year age group. The first two fields are the minimum and maximum age for that age group, so we can select only those records where the first field is less than 12.

```
lt`data = ExampleData[{"Statistics", "USLifeTable2003"}];
lt`u12 = Select[lt`data, x ↦ x⟦1⟧ < 12] (* the desired records *);
```

The `Select` command applies the provided function to each element, one at a time. If you have many items to check, and if you can vectorize the operation, it can be much faster (although more memory intensive) to use `Pick`. The `Pick` command picks out items from one list based on a boolean-valued selector list. Here we illustrate the use of `Pick` to produce the same data. In this particular case,while the use of `Pick` is faster than the use of `Select` above, that gain is eliminated by the construction of the selector.

```
selector = #⟦1⟧ < 12 & /@ lt`data;
lt`u12 == Pick[lt`data, selector] (* confirm we get same result *)
```

```
True
```

Dataset Transformations

This section provides a very basic introduction to datasets. For more details see Wolfram (2017; section 45).

A dataset is often structured as a collection of records, where each record consists of multiple fields, and each field has an identifier and a value. We say that such data is in a tabular format (or more simply that it is a table). In WL, this structure is very naturally modeled as a list of associations, where each association is a single record, and where the keys of the association are the field identifiers. The `Dataset` command creates a unified structure from such lists of associations, which can be readily queried and manipulated. (We can use the `Normal` command to reverse this process, creating a list of associations from such a dataset.)

```
assoc01 = <|"first" → "John", "last" → "Doe", "h1" → 50, "h2" → 30|>;
assoc02 = <|"first" → "Jane", "last" → "Doe", "h1" → 30, "h2" → 50|>;
assoc03 = <|"first" → "Signor", "last" → "Rossi", "h1" → 46, "h2" → 45|>;
data01 = Dataset[{assoc01, assoc02, assoc03}]
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
Signor	Rossi	46	45

Ordinarily, we do not build up our datasets by hand. One typical approach is to use the `SemanticImport` command to create data sets from external data files, such as CSV files. (See the WL documentation for details.) To illustrate this concept, here we use the closely related `SemanticImportString` command, which operates directly on strings.

```
data02 = SemanticImportString["first,last,h1,h2
John,Doe,50,30
Jane,Doe,30,50,
Signor,Rossi,46,45"]
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
Signor	Rossi	46	45

Given a dataset, we often wish to work with a smaller dataset comprising only those records that meet some criterion. This process is called *selective set building*, *filtering*, or *subsetting*. We can use the `Select` command to perform dataset subsetting by applying it to a dataset and an appropriate predicate (i.e., a function that returns `True` iff a record satisfies our criterion). For example, suppose we would like to keep only those records where the `h1` and `h2` fields sum to less than 90. Recalling that functions operating on associations that have strings as keys can refer by key name to the values., we can proceed as follows.

```
Select[data01, #h1 + #h2 < 90 &]
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50

The `Select` command also allows for the construction of a selection operator, which can subsequently be applied to the dataset. (This is particularly convenient if we wish to use the same criterion to subset several datasets.)

```
selop01 = Select[#h1 + #h2 < 90 &];
selop01@data01
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50

Given a dataset, we often want to perform a *query*, which retrieves data from the dataset. We can use the `Query` command to produce a query operator, which we apply to a dataset to produce a subset or a modification. The query syntax is extremely flexible, and queries may be very complex. Here we consider only a few simple queries.

As a first example of a query, let us select a subset of the fields from every record. This is the simplest example of *substitutive set building* or *mapping*. Using the `Query` command, we first characterize the records from which we want data (here, all of them), and then we list the columns we want (by name).

```
query01 = Query[All, {"h2", "h1"}];
ds03 = query01@data01
```

h2	h1
30	50
50	30
45	46

We can always discard the headers by selecting just the values.

```
ds04 = Values@ds03
```

30	50
50	30
45	46

The `Values` command produces a headerless rectangular dataset, which is essentially a rectangular list of lists, rather than a list of associations. Use the `Normal` command to produce the underlying list of lists.

```
Normal@ds04
```

```
{{30, 50}, {50, 30}, {45, 46}}
```

One interesting aspect of datasets is that `LinearModelFit` can be used as a query operator.

```
Query[LinearModelFit[#, x, x] &][ds04]
```

```
FittedModel[ 76.6154 - 0.830769 x ]
```

```
ds04[LinearModelFit[#, x, x] &]
```

```
FittedModel[ 76.6154 - 0.830769 x ]
```

chk

Later on we will discuss this further. You may wish to read the Wolfram Language How To entitled Perform a Linear Regression.

Let us try a new query, which combines substitutive and selective set building. Instead of using all records, we may use only records that satisfy some criterion. Here we illustrate that by replacing `All` with the select operator we already developed above.

```
query02 = Query[selop01, {"h2", "h1"}];
query02@data01
```

h2	h1
30	50
50	30

One of the most interesting possibilities arises from remembering that when defining functions that will be applied to associations we can make direct use of the keys. This allows us to easily create new data sets by applying functions to the records of an existing data set. This slightly more complex example of substitutive set building is sometimes known as *recoding*. Here we illustrate producing a new dataset that holds only names and totals by mapping over our original dataset. (Recall the slash-atmark short-hand for the `Map` command.)

```
sub01 = <|"first" → #first, "last" → #last, "total" → #h1 + #h2|> &;
sub01 /@ data01
```

first	last	total
John	Doe	80
Jane	Doe	80
Signor	Rossi	91

As noted before, the `Query` command is remarkably flexible. We can provide as its second argument a function to apply to each record. This means we can again easily combine selective and substitutive set building.

```
query03 = Query[selop01, sub01];
query03@data01
```

first	last	total
John	Doe	80
Jane	Doe	80

We can also use substitutive set building to add fields. Here, we append a “total” column to the dataset.

```
sub02 = Append[#, "total" → #h1 + #h2] &;
sub02 /@ data01
```

first	last	h1	h2	total
John	Doe	50	30	80
Jane	Doe	30	50	80
Signor	Rossi	46	45	91

We can also sort the data set on arbitrary criteria. (The next section contains some details about sorting.)

Sorting Datasets

In this section we will discuss how to reorder the rows of a dataset by sorting the records based on some criterion. Recall our first dataset.

```
data01
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
Signor	Rossi	46	45

Use `SortBy` to sort on any function of a record. Here is an example of sorting by the total score. (A numerical sort is ascending, so we add a minus sign to move the highest score to the top.)

```
SortBy[- (#h1 + #h2) &]@data01
```

first	last	h1	h2
Signor	Rossi	46	45
Jane	Doe	30	50
John	Doe	50	30

We will briefly address one subtlety of sorting in WL: by default, ties are broken based on what the WL calls the canonical order of the expressions. This means that tied rows may appear to be unexpectedly reordered. Here is an example of sorting on a column that includes ties.

```
SortBy[#last &]@data01
```

first	last	h1	h2
Jane	Doe	30	50
John	Doe	50	30
Signor	Rossi	46	45

To force a stable sort, where tied rows to retain their relative position, we can put braces around the sorting criterion.

```
SortBy[{#last &}]@data01
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
Signor	Rossi	46	45

4.1.3 Permutations and Combinations

A permutation of a set is a sequence of all the set elements, each occurring once. So order matters. A combination is just a subset: order does not matter. A *k*-combination is a subset of size *k*. A *k*-permutation is a permutation of a *k*-combination.

Power Set

The collection of all the subsets of a set *S* is called the power set of *S*. As we should expect, WL already implement powerset building for us: it provides the `Subsets` command. Nevertheless, working through some possible implementations pays multiple dividends. First, it provides a good illustration of some basic approaches to WL programming. Second, it more generally develops algorithmic insights that are important for working in any language. Finally and most importantly, it strengthens mathematical insight into the nature of the powerset and the subsets it contains. Readers who already have mastery in all these areas can skip to the next section.

The powerset is never empty, since even the empty set has one subset (which is the empty set). We can develop a simple iterative approach to producing the powerset. Begin with the empty set. Then iterate over the elements of the set, doing the following each time: create new subsets by appending the new element to all the existing subsets. Here is one approach to appending an element to each list in a list of lists and then joining the original and new list. (This approach provides a nice example of the operator form of the `Append` command; see the documentation for details.)

```
augment = {sets, elem} ↦ Join[sets, Append[elem] /@ sets];
```

For example, suppose $S = \{x, y, z\}$ and we wish to construct the powerset. We begin with the empty set. Then we add the element x to each of our existing subsets. This add a single subset, $\{x\}$, which we get by augmenting the empty set with the element x. Next we will augment the two subsets we have so far with the next element, y. This gives us two new subsets, $\{y\}$ and $\{x, y\}$. We continue in this fashion, doubling the number of subsets created each time we consider a new element. Here is a table of the successive outcomes.

```
{{}}
{{}, {x}}
{{}, {x}, {y}, {x, y}}
{{}, {x}, {y}, {x, y}, {z}, {x, z}, {y, z}, {x, y, z}}
```

A traditional approach to this problem would be to use an explicit procedural loop.

```
Module[{result = {{}}},
 Do[result = augment[result, elem], {elem, {x, y, z}}];
 result
]
```

```
{{}, {x}, {y}, {x, y}, {z}, {x, z}, {y, z}, {x, y, z}}
```

This works fine, but it is not the most idiomatic approach in WL. Note that each iteration produces a new collection of subsets from two inputs: an existing collection of subsets, and a new element. Recalling our earlier discussion of the `Fold` command, this is an ideal application.

```
Fold[augment, {{}}, {x, y, z}]
```

```
{{}, {x}, {y}, {x, y}, {z}, {x, z}, {y, z}, {x, y, z}}
```

Regardless of the implementation details, we see that each time we consider a new element, we double the number of subsets. There a set S will have $2^{\sharp S}$ subsets (where $\sharp S$ is the cardinality of S. This number grows very rapidly: a set of twenty elements has more than a million subsets. Even though we often have cause to talk about powersets, it is rare that we want to explicitly produce all the subsets of a set.

The smallest subset is the empty set: it has no elements. There is only one set this size. The largest subset of a finite set S is S itself: it contains all the elements. There is only one subset of this size. However, there may be multiple subsets of any intermediate size. We refer to a subset with k elements as a k-subset. It turns out that counting up the number of subsets of any particular size is an interesting and important problem.

Permutations

WL provides facilities for computing permutations. Use the `Permutations` command to produce all the permutations.

```
Clear[a, b, c, d, e]
Permutations[{a, b, c}]
```

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

We need to be a bit cautious when using the `Permutations` command. A set of cardinality *n* has *n*! permutations. This number grows very rapidly.

```
10!
```

```
3 628 800
```

We can produce the *k*-permutations by providing `{k}` as a second argument to `Permutations`.

```
Permutations[{a, b, c}, {2}] (* all permutations of length 2 *)
```

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

We can use `FactorialPower` to find the number of *k*-permutations. (This is also known as the falling factorial.)

```
FactorialPower[3, 2]
```

```
6
```

For any given value of *k* we can get a symbolic representation of the falling factorial if we use `FunctionExpand`:

```
FunctionExpand[FactorialPower[n, 2]]
```

```
(-1 + n) n
```

Use `RandomSample` to generate a random permutation of a set.

```
RandomSample[{a, b, c, d, e}]
```

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

Provide a second argument to `RandomSample` to generate a random *k*-permutation of a set.

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

```
{e, b}
```

Advanced

The `Permutations` command allows repeated elements, but they are treated as indistinguishable. Swapping indistinguishable elements does not produce a new permutation.

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

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

Combinations

Given a set, we can produce all of its subsets with the `Subsets` command. This is called the “power set”.

```
powerset = Subsets[{1, 2, 3}]
```

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

One word of warning: despite its name, the `Subsets` command does not discard any duplicate elements in its argument.

To get all the subsets of a given size, we could produce them all and then filter based on their size.

This uses normal set building, which we illustrate here with `Select`.

```
Select[powerset, lst ↦ (Length[lst] == 2)]
```

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

However, this is a computationally expensive approach. A set of cardinality *n* has 2^n subsets, and this number grows very quickly.

```
2^64
```

```
18 446 744 073 709 551 616
```

WL provides a computationally superior way. The `Subsets` command accepts a second argument of `{k}` (note the braces) to produce *k*-combinations:

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

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

To compute the number of *k*-combinations in a set of size *n*, we could create the subsets and then count them. But computationally it is much better to use the `Binomial` command.

```
Binomial[10, 5]
```

```
252
```

The binomial command produces the value of $C[n, k]$, which is the coefficient on x^k we find when expanding $(1 + x)^n$. For example:

n	$(1 + x)^n$	Coefficients
0	1	{1}
1	$1 + x$	{1, 1}
2	$1 + 2x + x^2$	{1, 2, 1}
3	$1 + 3x + 3x^2 + x^3$	{1, 3, 3, 1}
4	$1 + 4x + 6x^2 + 4x^3 + x^4$	{1, 4, 6, 4, 1}
5	$1 + 5x + 10x^2 + 10x^3 + 5x^4 + x^5$	{1, 5, 10, 10, 5, 1}

Formatting Fun: Pascal’s Triangle

Exemplifying Stigler’s law of eponymy, the triangular arrangement of the binomial coefficients known as Pascal’s triangle was developed by the 10th century Persian mathematician Al-Karaji. The *n*-th row contains the binomial coefficients $C[n - 1, i]$ for $i = 0, \ldots, n - 1$.

```
pascalRow[rownum_Integer?Positive] :=
 Array[Binomial[rownum - 1, #] &, rownum, 0]
```

We can use this function to generate as many rows as we wish of Pascal’s triangle.

```
ClearAll[pascalRows]
pascalRows[nrows_] := pascalRow /@ Range[nrows]
```

In order to produce a triangular layout, we can use the `Grid` command.

```
Grid[pascalRows[10], Alignment → Right]
```

```
1
1 1
1 2  1
1 3  3  1
1 4  6  4   1
1 5 10 10   5   1
1 6 15 20  15   6  1
1 7 21 35  35  21  7  1
1 8 28 56  70  56 28  8 1
1 9 36 84 126 126 84 36 9 1
```

The default layout by `Grid` actually looks pretty good, but add somewhat more consistent formatting to the triangle by basing column widths on the largest integer represented. We will use the `ItemSize` option to accomplish this.

```
Grid[pascalRows[10], Alignment → Right, ItemSize → All]
```

```
  1
  1   1
  1   2   1
  1   3   3   1
  1   4   6   4   1
  1   5  10  10   5   1
  1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
  1   8  28  56  70  56  28   8   1
  1   9  36  84 126 126  84  36   9   1
```

Now for some extra formatting fun: let us try to produce a more traditional pyramidal arrangement of the table. The `Grid` command is very flexible, so we will push it a bit to see if it can meet our needs. Essentially, we want the same triangle, but with a more traditional formatting. To do this, we will make each row a separate grid, centered in a column of such grids. We will find the cell sizes for these grids by playing around with the possibilities, rather than from first principles.

```
ClearAll[pascalPyramid]
pascalPyramid[nrows_] := Module[{max, cellWd},
  max = Binomial[nrows, Quotient[nrows, 2]]; (* maximum entry *)
  cellWd = 0.25 * IntegerLength[max] + 1;  (* ad hoc cell width *)
  Column[
   Grid[{#}, ItemSize → {cellWd, 1.5}, Alignment → Center] & /@ pascalRows[nrows],
   Center
  ]
 ]
pascalPyramid[10]
```

```
                    1
                 1     1
              1     2     1
           1     3     3     1
        1     4     6     4     1
     1     5    10    10     5     1
  1     6    15    20    15     6     1
1     7    21    35    35    21     7     1
   1     8    28    56    70    56    28     8     1
1     9    36    84   126   126    84    36     9     1
```

4.2 Lists as Vectors

WL lists support elementwise addition and scalar multiplication. Because of this, a list of *N* numerical elements give us a natural computational representation of an *N*-vector. The elements of an *N*-vector are sometimes called components or coordinates.

4.2.1 Introduction to Vectors

Mathematically, an *N*-vector is more than just a list of numbers. A vector is an element of a vector space. A vector space is essentially a set whose elements (the vectors) are closed under linear combinations. (We will leave most of the details to your math texts.) We will be primarily concerned with *N*-vectors of real numbers. In WL, scalar multiplication and elementwise addition are normal properties of lists, so we do not need a separate object type to represent *N*-vectors.

Numerical and Symbolic Vectors

Here are some two-dimensional numerical examples. (Evaluate these expressions successively.)

Expression	Result
v1={1,2}	{1, 2}
2*v1	{2, 4}
v2={2,1}	{2, 1}
v1+v2	{3, 3}
2*v1+3*v2	{8, 7}

The vector elements are called “coordinates”. In contrast with the elements of a set, the order and multiplicity of the coordinates affect the definition of a vector. For example, despite being built from the same set of numbers, v_1 and v_2 represent two different vectors.

Recall that WL uses unit-based indexing: the first coordinate has an index of 1. Our vectors are just lists, we we access individual coordinates with the `Part` command (here, using the double bracket shorthand).

```
v1⟦1⟧
```

```
1
```

Here are some three-dimensional numerical examples. (Evaluate these expressions successively.)

Expression	Result
v1={1,2,3}	{1, 2, 3}
2*v1	{2, 4, 6}
v2={3,2,1}	{3, 2, 1}
v1+v2	{4, 4, 4}
2*v1+3*v2	{11, 10, 9}

In this book, we will often need finite-dimensional vectors. Any list of numbers or symbols can represent a vector, so all the tools we developed for creating lists are applicable to creating vectors. For example, for the construction of symbolic vectors, the `Array` command can be useful.

```
ClearAll[s, x, y]
xs = Array[x# &, 5]
ys = Array[y# &, 5]
s * xs (* scalar multiplication *)
xs + ys (* vector addition *)
```

$\{x_1, x_2, x_3, x_4, x_5\}$

$\{y_1, y_2, y_3, y_4, y_5\}$

$\{s\, x_1, s\, x_2, s\, x_3, s\, x_4, s\, x_5\}$

$\{x_1 + y_1, x_2 + y_2, x_3 + y_3, x_4 + y_4, x_5 + y_5\}$

Recall from our discussion of list construction that an indexed symbol is not itself a symbol, it is an expression. While you can conveniently use such an expression to represent a value, ordinarily you will not assign a value to it. This becomes clearer if we examine the full form of a subscripted symbol: it is an expression.

FullForm[x_1]

```
Subscript[x, 1]
```

Testing for Vectorhood

The `VectorQ` command provides a narrow test for vectorhood: it only tests that a list contains no other lists. In the following simple example, the third list is not a vector because it contains another list.

```
{vA = Array[a# &, 5], Append[vA, 6], Append[vA, {}]}
VectorQ /@ %
```

```
{{a1, a2, a3, a4, a5}, {a1, a2, a3, a4, a5, 6}, {a1, a2, a3, a4, a5, {}}}
```

```
{True, True, False}
```

The `VectorQ` Command accepts a second argument, which is a test to apply to each element. For example, to test whether we have a numerical vector, we can use the `NumberQ` command as our second argument, or we could use the `Positive` command to test for positivity. We can use any builtin test of properties, or construct our own test. For example, we could require that all elements lie in the unit interval:

```
VectorQ[
 {0.1, 1.0, 1},  (* first arg: test this for vectorhood *)
 x ↦ 0 < x ≤ 1   (* second arg: additional elementwise requirement *)
]
```

```
True
```

Visualizing Vectors (2D & 3D)

In the Cartesian coordinate system, each coordinate represents a distance from the origin along a coordinate axis. This leads to two standard graphical representations of 2-tuples and 3-tuples: as simple markers (e.g., filled discs) centered on the represented point, or as the end of an arrow whose tail is at the origin. The second representation is often used to indicate that the tuples are vectors (i.e., that they can be added and scaled).

In the following two-dimensional illustration, we draw an arrow to indicate that we are dealing with a vector. Two-dimensional real vectors have a familiar representation with standard Cartesian coordinates: each vector is represented by an ordered pair of numbers, which measure the distances along the horizontal and vertical axes.

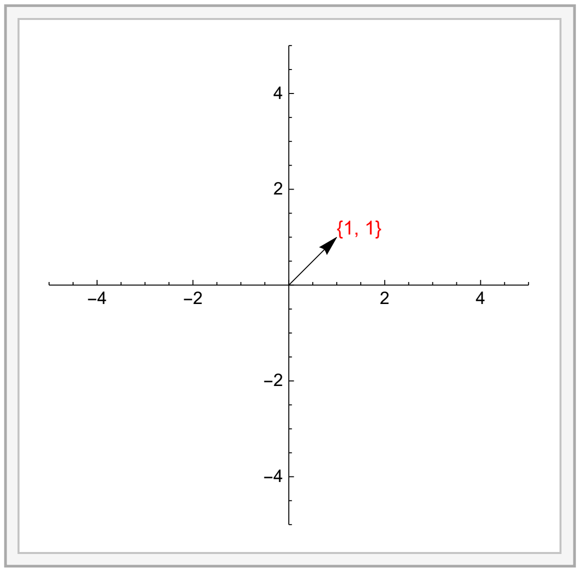


Figure 1: Visualize 2D Vector

Constructing a visual representation of three-dimensional Euclidean vectors is a bit more challenging, when we must project them onto a two-dimension surface. Still, standard conventions for providing visual clues make this somewhat possible. Here we frame our three-dimensional vectors with a cube, which provides visual clues about how far the vector extends in each dimension.

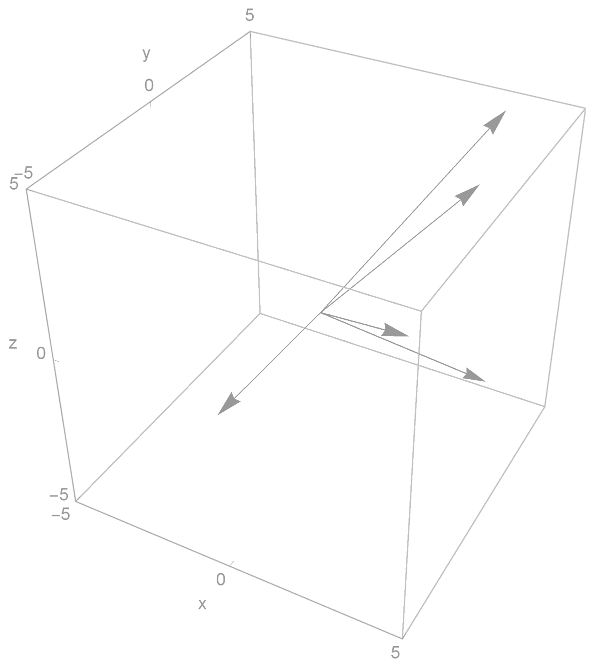


Figure 2: Visualize 3D Vectors

Visualizing Scalar Multiplication and Vector Addition

Scalar multiplication literally scales the vector: it changes its length (and possibly reverses its direction). We define scalar multiplication as the result of multiplying each coordinate by the scalar.

```
ClearAll[s, v]
s * Array[v# &, 5]  (* scalar multiplication *)
```

```
{s v1, s v2, s v3, s v4, s v5}
```

We can easily visualize scalar multiplication in two-dimensional Euclidean space: it produces a proportional change in the length of the vector. If the scalar is negative, the direction of the vector is reversed.

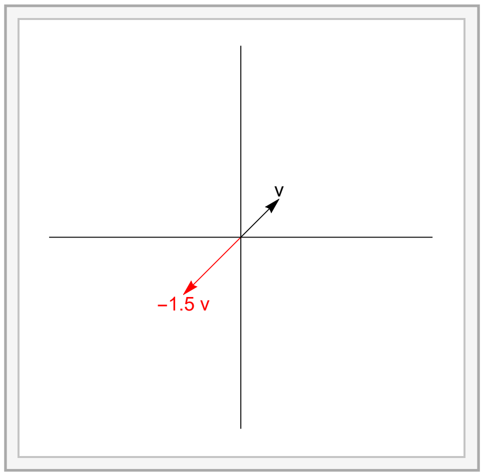


Figure 3: Scalar Multiplication

From our definition of scalar multiplication, we can see that it has the following properties:

Expression	Result	Comment
1 v==v	True	scalar identity
s2 (s1 v) == (s2 s1) v	True	associative scaling

Two two vectors from a vector space can be added. We define the addition of *n*-vectors to be elementwise: corresponding coordinates are added to produce a new vector. Here we use `Array` to create two vectors, which we then add.

```
ClearAll[vx, vy]
vx = Array[x# &, 4];
vy = Array[y# &, 4];
vx + vy  (* vector addition is elementwise *)
```

```
{x1 + y1, x2 + y2, x3 + y3, x4 + y4}
```

Vector addition satisfies the usual properties of an additive group.

Expression	Result	Comment
0+v==v	True	additive identity exists
v+(-v)==0	True	inverses exist
v1+(v2+v3)==(v1+v2)+v3	True	associativity
v1+v2==v2+v1	True	commutativity

We also see that for any vector *v* we can produce its additive inverse −*v* by means of the scalar multiplication −1 *v*.

```
v + (-1) v == 1 v + (-1) v == (1 - 1) v == 0 v == 0
True
```

The result of adding two two-dimensional real vectors can be represented as “completing a parallelogram”. Here two vectors are represented by two black arrows. Treating these as two sides to a parallelogram, the dashed lines complete the parallelogram. The light red arrow represents the result of adding the two vectors.

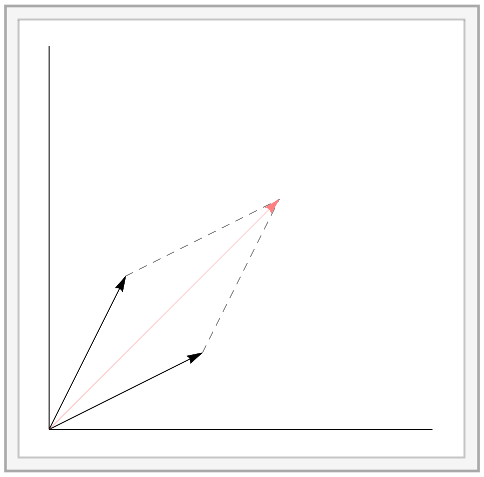


Figure 4: Vector Addition

Scalars behave like ordinary numbers, so they can be added and multiplied. We therefore end up with the following two distributive laws.

Expression	Result	Comment
s(v1+v2)==s v1+s v2//Simplify	True	field-sum distribution
(s1+s2)v==s1 v+s2 v//Simplify	True	scalar-sum distribution

Linear Combination

A finite weighted sum of vectors is called a linear combination of the vectors. If the weights sum to unity, it is also called an affine combination. An affine combination with nonnegative weights is called a convex combination.

Here we create two 2-vectors, then scale them, then add them -- all purely symbolically.

```
ClearAll[s, v]
{v1, v2} = Array[v_## &, {2, 3}]
s_1 * v1 + s_2 * v2  (* linear combination of v1 and v2 *)
```

$\{\{v_{1,1}, v_{1,2}, v_{1,3}\}, \{v_{2,1}, v_{2,2}, v_{2,3}\}\}$

$\{s_1 v_{1,1} + s_2 v_{2,1}, s_1 v_{1,2} + s_2 v_{2,2}, s_1 v_{1,3} + s_2 v_{2,3}\}$

Here is a similar exercise, but using numbers instead of symbols.

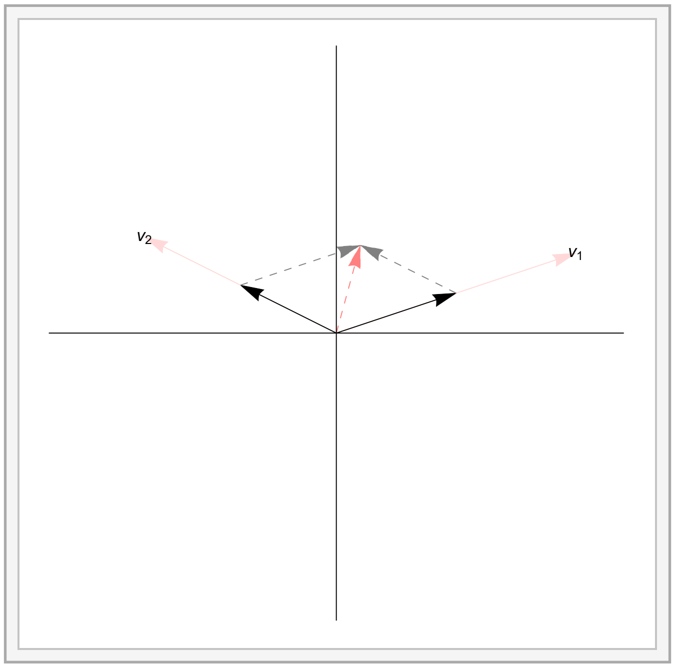
```
ClearAll[s1, s2, v1, v2]
{s1, s2} = {1, 2}
{v1, v2} = {{3, 4, 5}, {6, 7, 8}}
s1 * v1 + s2 * v2  (* linear combination of v1 and v2 *)
```

```
{1, 2}
```

```
{{3, 4, 5}, {6, 7, 8}}
```

```
{15, 18, 21}
```

In the following illustration, we produce a linear combination (i.e., weighted sum) of two vectors. First, two vectors (v_1 and v_2) are scaled, and then the results are added.



Vector Space, Span, and Linear Independence

A vector space is a set of vectors that is closed under linear combination. This means that any linear combination of vectors in the set produces another vector in the set.

xmpl

Consider the following set of vectors: any vector x, along with all the vectors that can be formed from it by scalar multiplication. If we add any two vectors from this set, we get another vector in this set.

Let V be a collection of vectors. The linear span of V is the set of all the vectors that can formed as linear combinations of vectors in V. The span of a collection of vectors is clearly a vector space.

Any vector in the span of V is said to be linearly dependent on V. Any vector not in the span of V is said to be linearly independent of V. A collection of vectors such that no vector can be written as a linear combination of the others is also called linearly independent. Otherwise, the collection is linearly dependent.

xmpl

Let $V = \{v_1, v_2\}$ where $v_1 = (1, 2)$ and $v_2 = (2, 4)$. Then $v_3 = (7, 10)$ is linearly independent of V: we cannot find a linear combination of v_1 and v_2 that is equal to v_3. That is, we cannot find two scalars (s_1, s_2) such that $v_3 = s_1 v_1 + s_2 v_2$.

If we try to find solutions when none exist, we will get an empty set of solutions. For example, let us attempt to use `Solve` on the following equation.

```
Clear[s1, s2]
Solve[s1 * {1, 2} + s2 * {2, 4} == {7, 10}, {s1, s2}]
```

```
{}
```

In contrast, $v_4 = (7, 14)$ is linearly dependent on V. For example, $v_4 = 3 v_1 + 2 v_2$. Let us try to show this using `Solve`.

```
Solve[s1 * {1, 2} + s2 * {2, 4} == {7, 14}, {s1, s2}]
```

```
Solve: Equations may not give solutions for all "solve" variables.
```

$$\left\{\left\{s2 \to \frac{7}{2} - \frac{s1}{2}\right\}\right\}$$

What happened? A collection of vectors V is said to be linearly independent if none of the vectors lies in the span of the others. The problem is that V is itself linearly dependent, so there is not a unique solution to this problem. The `Solve` command returns an answer that represents all the solutions, including our previously proposed solution $(s_1, s_2) = (3, 2)$.

xmpl

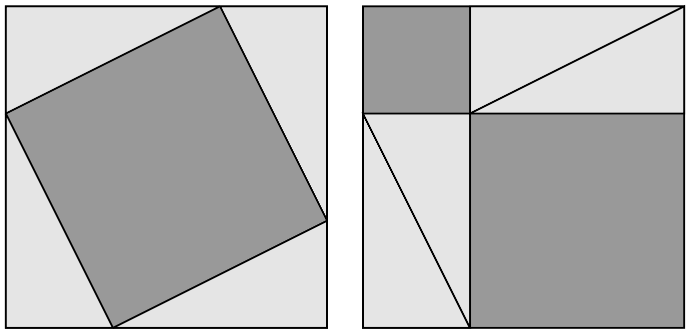
Let $V = \{v_1, v_2\}$ where $v_1 = (1, 2)$ and $v_2 = (3, 4)$. Then $v_3 = (7, 10)$ is linearly dependent on V, because we can find a linear combination of v_1 and v_2 that is equal to v_3. That is, we can find two scalars (s_1, s_2) such that $v_3 = s_1 v_1 + s_2 v_2$. Specifically, we find $(s_1, s_2) = (1, 2)$.

```
Solve[s1 * {1, 2} + s2 * {3, 4} == {7, 10}, {s1, s2}]
(* (7,10) is in the span of (1,2) and (3,4) *)
```

```
{{s1 → 1, s2 → 2}}
```

Euclidean Length and Normalization

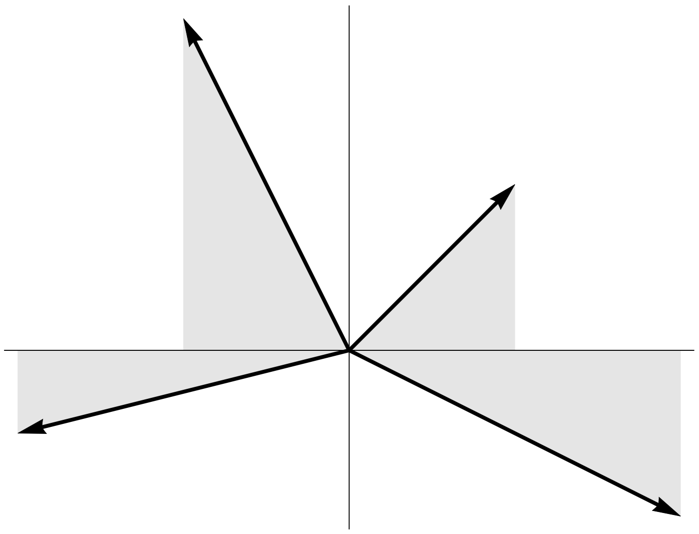
Next we explore the idea of the length of a vector. To motivate our concept of length, we work with vectors in the Cartesian plane and apply the Pythagorean theorem. The longest side of a right triangle is called the hypotenuse; the other two sides are called legs (or catheti). The Pythagorean theorem says that the squared length of the hypotenuse is the sum of the squared lengths of the legs. The following figure provides a classical proof by rearrangement of this theorem.



Pythagorean Theorem

With the Pythagorean theorem in mind, we note that associated with a real vector (*x*, *y*) >> 0 is a right triangle with vertexes at (0, 0), (*x*, *y*), and (*x*, 0). The last of these is called the projection of the vector onto the *x*-axis. (We could equally well project onto the *y*-axis.) Here is a fairly arbitrary collection of examples, where we use the `Triangle` command to create the filled triangles and the `Arrow` command to create the arrows.

```
vectors = {{1, 1}, {-1, 2}, {-2, -0.5}, {2, -1}}; (* arbitrary *)
g1 = Graphics[{
     {GrayLevel[0.9], Triangle[{{0, 0}, #, {First[#], 0}}] & /@ vectors},
     {Thick, Arrow[{{0, 0}, #}] & /@ vectors}
   }, Axes → True, Ticks -> None];
Labeled[g1, "Vectors with Associated Right Triangles"]
```



Vectors with Associated Right Triangles

Applying the Pythagorean theorem, we conclude that the vector (*x*, *y*) has "length" equal to $\sqrt{x^2 + y^2}$. For example, the vector (1, 1) has a length of $\sqrt{2}$. This terminology is a bit misleading: a vector is just a point, so it does not inherently have length. Rather, we have found the distance from the origin to this point, which is the length of the arrow we draw to represent the vector. When we want to remove any ambiguity in terminology, we refer to this value as the *norm* of the vector—more specifically, the

Euclidean norm of the vector. The need to compute vector norms is very common, and the WL provides the `Norm` command to compute it.

```
Norm[{1, 1}]
```

$\sqrt{2}$

A vector with a unit norm is called a unit vector. A vector whose elements are all zero except for one that is one is called a standard unit vector. While we can readily create standard unit vectors as needed, WL provides the `UnitVector` command to do so. This takes two arguments: the length of the vector, and the location of the unit element.

Expression	Result
UnitVector[2,1]	{1, 0}
UnitVector[2,2]	{0, 1}
UnitVector[3,1]	{1, 0, 0}
UnitVector[3,2]	{0, 1, 0}
UnitVector[3,3]	{0, 0, 1}

Our formula for the two-dimensional Euclidean norm is the square root of the sum of the squared elements. This can be extended to any real *N*-vector. The `Norm` command correspondingly handles *N*-vectors of any length. For example,

Expression	Result
xs=Array[x#&,5]	$\{x_1, x_2, x_3, x_4, x_5\}$
Norm[xs]	$\sqrt{\text{Abs}[x_1]^2+\text{Abs}[x_2]^2+\text{Abs}[x_3]^2+\text{Abs}[x_4]^2+\text{Abs}[x_5]^2}$
xs=UnitVector[4,4]	{0, 0, 0, 1}
Norm[xs]	1

At first glance, this result may appear to include some redundancy: why is the `Abs` command used before squaring? There is a good reason: we have not yet specified that that we are dealing with a real vector. We can provide this assurance with the second argument of the `Simplify` command, which is where we can specify assumptions relevant to the expression being simplified. Here, we insist that all of the vector elements are real numbers by creating an appropriate list of assumptions.

```
xsReal = Element[#, Reals] & /@ xs
```

```
{True, True, True, True}
```

With these assumptions, the `Norm` command produces the “expected” result.

```
Simplify[Norm[xs], xsReal]
```

```
1
```

By examining the algebra of the Euclidean norm, we will find that it is absolutely homogenous, subadditive, and separating. (Only subadditivity requires any work to understand, and we will come back to it.)

Property Name	Expression
absolutely homogeneous:	$\lVert\lambda\ x\rVert == \lvert\lambda\rvert\ \lVert x\rVert$
subadditive:	$\lVert x+y\rVert \leq \lVert x\rVert + \lVert y\rVert$
separating:	$\lVert x\rVert == 0 \Rightarrow x == 0$

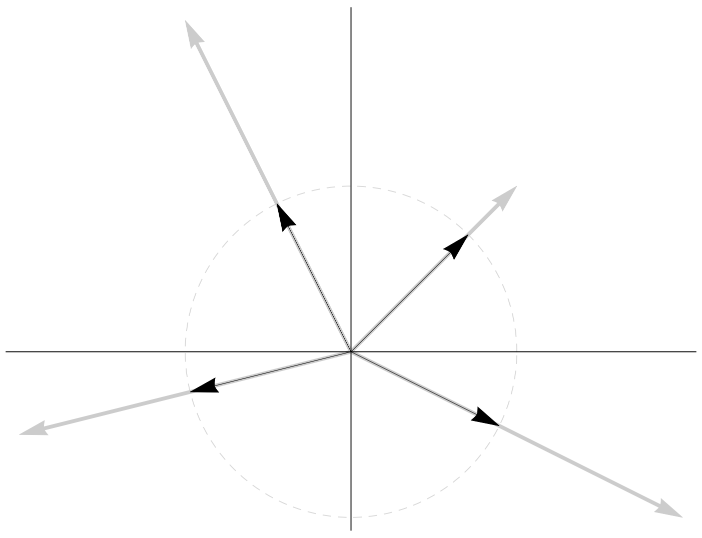
We sometimes use the term ‘functional’ to designate a function from a vector space to the real numbers. We can create many functionals that satisfy these three properties, which define what we mean by a norm. So there are many norms besides the Euclidean norm. However, in this book we will usually be interested in the Euclidean norm.

Note that the three definitional properties imply that a norm is a positive definite functional. (That is, it is positive for every argument, apart from the zero vector, which has a norm of 0.) If we begin with a nonzero vector *x* and divide each coordinate by Norm[*x*], then we produce a new vector in the same direction that has a norm of 1. We say that we have normalized *x* to produce a codirectional unit vector. WL provides the `Normalize` command to do this normalization.

```
Normalize[{1, 1}]
```

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

The next figure illustrates some vectors (in gray) with their normalizations (in black), along with a unit circle to aid length comparisons.



Normalized Vectors

Hadamard Product and Scalar Product

We next consider the scalar product of two *N*-vectors. Suppose we first do element-by-element multiplication of two real vectors and then sum the elements of the result: this produces what is known as the *scalar product* of our two vectors. The scalar product of two vectors in $\mathbb{R}^N$ is the result of multiplying the corresponding coordinates and summing those products.

$$a.b = \sum_{i=1}^{N} a_i\, b_i \qquad (1)$$

Suppose for example that we have a vector of prices and a vector of quantities purchased for 5 different goods. If we multiply together the corresponding elements of the vectors, we get a vector containing the expenditures on each of the 5 goods. This is the Hadamard product of the two vectors. If we then form the total of these expenditures, we get the total expenditure on the 5 goods. This is the scalar product of the two vectors.

Since vectors in WL are lists, all list operations are available if you wish to do non-standard vector manipulations. For example, you can perform elementwise multiplication simply by using the multiplication operator (which is equivalent to the `Times` command). For two conformable vectors, this produces the elementwise product (which is also called the Hadamard product). Once we have the elementwise product, we can sum up all the terms with the `Total` command to produce the scalar product.

Expression	Result
ps=Array[p#&,5]	$\{p_1, p_2, p_3, p_4, p_5\}$
qs=Array[q#&,5]	$\{q_1, q_2, q_3, q_4, q_5\}$
ps*qs	$\{p_1 q_1, p_2 q_2, p_3 q_3, p_4 q_4, p_5 q_5\}$
Total[ps*qs]	$p_1 q_1 + p_2 q_2 + p_3 q_3 + p_4 q_4 + p_5 q_5$

The scalar product has many uses, and WL provides the `Dot` command to compute it. It is idiomatic in WL to use the infix shorthand for the `Dot` command, which is simply a dot (produced as a period, or full stop). Indeed, it is common refer to the scalar product as the dot product, and we will most often adopt this usage. Using the shorthand notation, we can represent the dot product of *p* and *q* very simply.

```
ps.qs
```

$p_1 q_1 + p_2 q_2 + p_3 q_3 + p_4 q_4 + p_5 q_5$

Here is a step by step numerical illustration of the same concepts, using 2-vectors.

Expression	Result
v1={1,2}	{1, 2}
v2={3,4}	{3, 4}
v3=v1*v2	{3, 8}
Total[v3]	11

Now that we are in possession of the scalar product, we have another perspective on the computation of the Euclidean norm. It is the square root of the scalar product of a vector with itself.

```
Sqrt[ps.ps]
```

$\sqrt{p_1^2 + p_2^2 + p_3^2 + p_4^2 + p_5^2}$

Advanced: Inner Product

For vectors in $\mathbb{R}^N$, the dot product is an inner product. An inner product of two vectors is often represented as $< v1, v2 >$. For real vectors, you can use WL's `Inner` command to produce the dot product. This command has very general functionality. It takes four arguments, the first and last of which are the functions to sequentially apply to the list arguments. For example, our standard scalar product of real vectors can be written as follows.

```
Clear[a, b, c, d]
Inner[Times, {a, b}, {c, d}, Plus]
```

$a c + b d$

An inner product maps two vectors to a number while meeting certain constraints, which we will discuss in more detail latter. For vectors of real numbers one of these constraints is symmetry:

$< v1, v2 >$ `==` $< v2, v1 >$. However, WL's `Inner` command does *not* enforce such constraints. For example, let us replace `Times` with `Power`. We find that symmetry is violated, but WL does not complain about this. Thus the `Inner` command can be used to produce inner products, but it can be used in other ways as well.

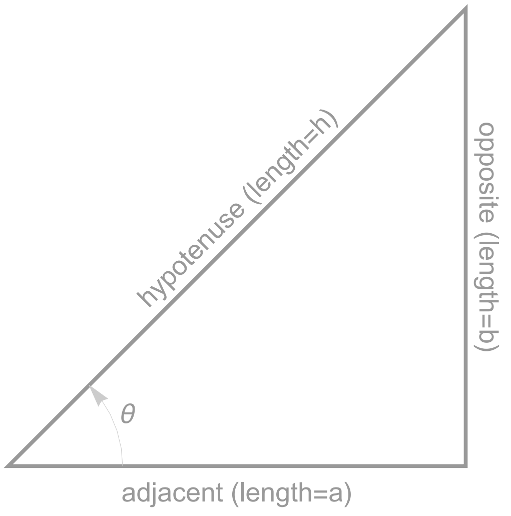
```
Clear[a, b, c, d]
Inner[Power, {a, b}, {c, d}, Plus]
Inner[Power, {c, d}, {a, b}, Plus]
```

$a^c + b^d$

$c^a + d^b$

Trigonometric Functions

Recall that the interior angles of any triangle sum to π radians (i.e., half a turn, or 180°). In a right triangle, one of the interior angles is $\pi/4$ radians (i.e., a quarter turn, or 90°). The other two angles must therefore be smaller than this (i.e., must be acute angles). Given a right triangle, knowing one more angle determines the other. Triangles with a common set of angles are called similar, and similar triangles have common relative lengths of their three sides. So given one acute angle, the relative lengths of the sides of a right triangle are functions of that angle, which are called "trigonometric" functions of the angle. Here we illustrate a simple right triangle with one of the acute angles labeled as θ. The sides relative to this angle are also labeled: the adjacent side (with length a), the opposite side (with length b), and the hypotenuse (with length h). The Pythagorean Theorem implies that $h^2 = a^2 + b^2$.



Right Triangle

In this setting, we can define the trigonometric functions in terms of these sides. (We will soon extend these definitions to other angles.) The cosine, cosecant, and cotangent of the angle θ are the sine, secant, and tangent of the complementary angle. (Thus the "co" prefix.)

name	ratio	WL command	equivalent
sine	b/h	Sin[θ]	Cos[π/2-θ]
cosine	a/h	Cos[θ]	Sin[π/2-θ]
secant	h/a	Sec[θ]	Csc[π/2-θ]
cosecant	h/b	Csc[θ]	Sec[π/2-θ]
tangent	b/a	Tan[θ]	Cot[π/2-θ]
cotangent	a/b	Cot[θ]	Tan[π/2-θ]

Recall that similar triangles have the same angles and the same relative lengths of sides. This allows us to illustrate the trigonometric functions of theta within the unit circle. For an angle of *θ* (counter-clockwise from the positive *x*-axis), we see that we can represent the cosine and sine as projections on the *x*-axis and *y*-axis.

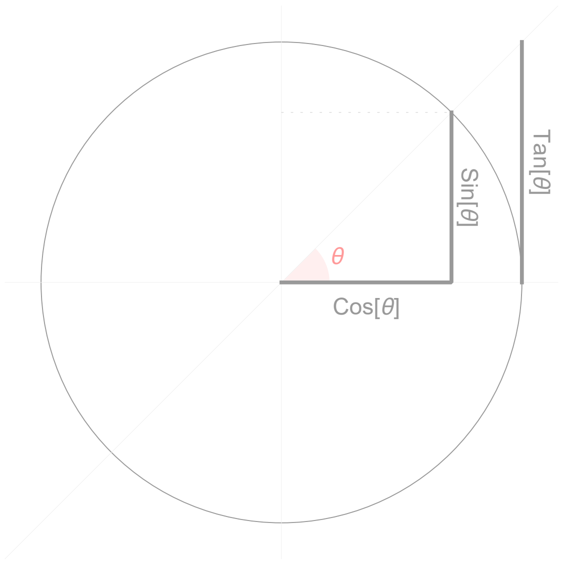


Figure 5: Some Trigonometric Functions

We will use this understanding to extend our definitions of the sine and cosine functions to all angles. For example:

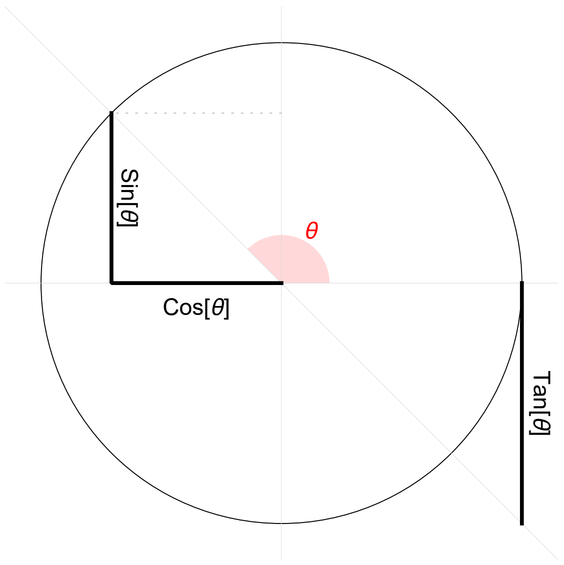
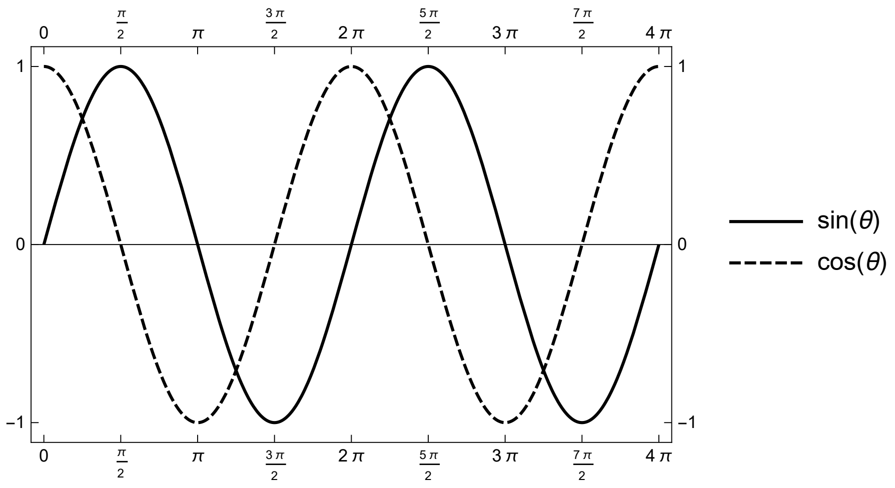


Figure 6: Trig Function with Obutuse Angles

An implication is that the trigonometric functions are periodic. Here we illustrate this by plotting the sine and cosine functions. We also take this opportunity to introduce the possibility of placing plots in a ticked frame, with the `Frame` and `FrameTicks` options. The latter allows us to specify a pair of lists, the ticks along the *x*-axis and *y*-axis. As a convenience, we use the `Subdivide` command to evenly divide the interval from 0 to 4 into 8 pieces. Finally, we use the `PlotLegends` option to add a plot legend.

```
Plot[{Sin[θ], Cos[θ]}, {θ, 0, 4 π},
 Frame → True,
 FrameTicks → {π * Subdivide[0, 4, 8], {-1, 0, 1}},
 PlotLegends → "Expressions"]
```



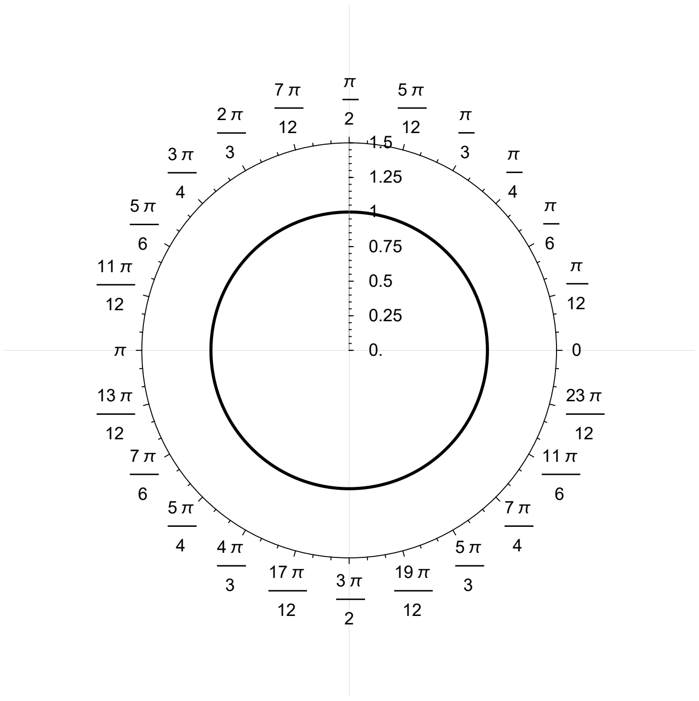
The angle between two vectors is the angle between the rays from the origin through the points. Equivalently, it is the angle between the arrows we often draw to represent our vectors. This description is a bit ambiguous. To remove this ambiguity, unless explicitly stated otherwise, we restrict our meaning to positive angles of half a turn or less. Then the angle between codirectional vectors is 0, the angle between contradirectional vectors is half a turn (π radians), and the angle between vectors that are not colinear is their interior angle, which is less than half a turn.

4.2.2 Angles

Our discussion of trigonometric functions suggests an alternative way to characterize a vector. Instead of directly stating its Cartesian coordinates, we can describe the vector with an angle (counter-clockwise from the positive *x*-axis) and a length. This representation is called the polar coordinates of the vector. We are going to briefly explore the relationship between Cartesian coordinates and polar coordinates.

Angles are measured in a wide variety of units, all relative to the circumference of a circle. The most basic measure is the turn: 1 turn is 1 circumference. The most popular measure is the Sumerian degree: 1 turn is 360 degrees. The most commonly used measure in science and mathematics is the radian: 1 turn is 2 π radians. A radian is the length of a circle's radius relative to its circumference. In WL, angles are measured in radians. Using Cartesian axes in standard position, these angles are traditionally measured counterclockwise from the positive *x*-axis. Here we illustrate these measurements by using WL's `PolarPlot` command to plot a unit circle along with polar coordinates. (Recall that polar coordinates characterize a point in the Cartesian plane with a radius and an angle of rotation.) As a minor refinement, we use the `Prolog` option to add to our polar plot two lines representing the traditional Cartesian axes.

```
PolarPlot[1, {θ, 0, 2 π}, PolarAxes → True,
 PlotRange → {{-2, 2}, {-2, 2}},
 Prolog → {Thin, LightGray, InfiniteLine[{{0, 0}, #}] &/@{{1, 0}, {0, 1}}}
]
```



We will also be interested in characterizing the angle between any two vectors.

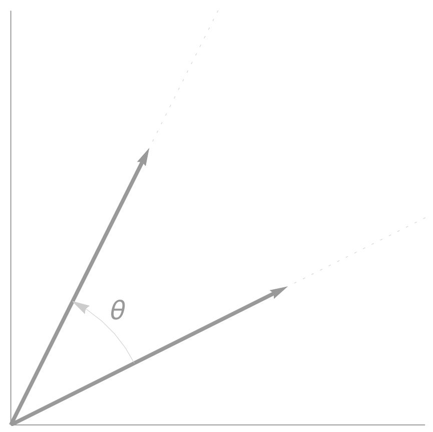
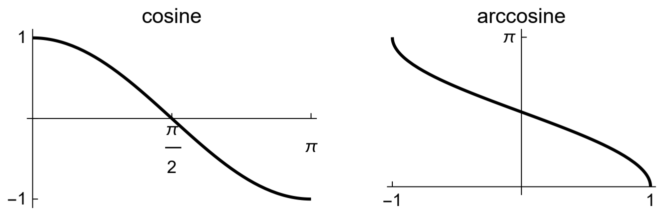


Figure 7: Angle Between Two Vectors

It turns out our prior work on the scalar product and norms helps with this. This cosine of the angle between the two vectors (*u* and *v*) is the dot product of the normalized vectors.

$$\mathbf{Cos}[\theta] == \frac{\mathbf{u}}{\|\mathbf{u}\|}.\frac{\mathbf{v}}{\|\mathbf{v}\|};$$

In order to use the vectors to solve for this angle, we must first address the periodic nature of the cosine function. We would like to just use the inverse of the cosine function to solve for the angle, but because the cosine function is not monotone its inverse is not a function. We therefore create an inverse function for a restricted version of the cosine function, which is strictly decreasing on a domain of [0 .. π]. We call the inverse of this function the arccosine function: it returns the arc (in radians) required to produce the cosine value (in [0 .. 1]).



We are now able to solve for the angle between any two nonzero real *N*-vectors. For example, if we define the function

```
realVectorAngle = {u, v} ↦ ArcCos[Normalize[u].Normalize[v]];
```

then we can do computation such as

Expression	Result
`realVectorAngle[{1,2},{-1,-2}]`	π
`realVectorAngle[{1,2},{-2,1}]`	$\frac{\pi}{2}$

This is a common enough need that WL builds in this functionality with its `VectorAngle` command. Additionally, it is evident from out plots that the meaning of a zero dot product is that our vectors are orthogonal.

Consider a budget constraint example, with free disposal, no saving, and no labor supply constraint. Each period a consumer supplies labor n to the market at wage w in order to pay for consumption x at the price p. The basic budget constraint is therefore $w\,n == p\,x$, or equivalently $(w, p).(-n, x) == 0$. This means that the price vector (w, p) is orthogonal to the consumption vector $(-n, x)$.

4.2.3 Projection and the Dot Product

Consider two nonzero vectors, u and v. We are going to construct the "orthogonal projection" of the vector u onto the vector v by dropping a perpendicular from the point u to the line through v. The result of this projection, which we will call p, is obviously a scaled version of v. Therefore, $p = s\,v/\text{Norm}[v]$ for some scalar s, which we sometimes call the scalar projection of u on v. We are going to determine that $s = (u.v)/\|v\|$.

```
realScalarProjection = {u, v} ↦ u.Normalize[v];
realOrthogonalProjection = {u, v} ↦ realScalarProjection[u, v] * Normalize[v];
```

Looking at the graph, we see that our representation of the projection constructs a right triangle with the hypotenuse represented by u and $\cos[\theta] = \text{Norm}[p]/\text{Norm}[u]$, so that $\text{Norm}[p] = \cos[\theta]\,\text{Norm}[u]$. Recalling our expression for $\cos[\theta]$, we have

$$\|p\| == \cos[\theta]\ \|u\| == \frac{u.v}{\|u\|\ \|v\|}\|u\| == \frac{u.v}{\|v\|} \tag{2}$$

$$p = \frac{\|p\|}{\|v\|}v == \frac{u.v}{\|v\|^2}v == \frac{u.v}{v.v}v \tag{3}$$

$$p = \frac{\|p\|}{\|v\|}v == \|p\|\frac{v}{\|v\|} == \cos[\theta]\frac{\|u\|}{\|v\|}v \tag{4}$$

Here is a graphical example.

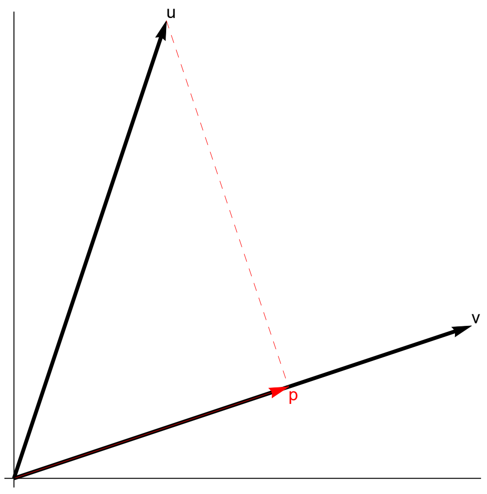


Figure 8: Orthogonal Projection (Acute Angle)

If the two vectors form an obtuse angle, then the scalar projection will be negative. The projection vector will be colinear, but it will not be codirectional.

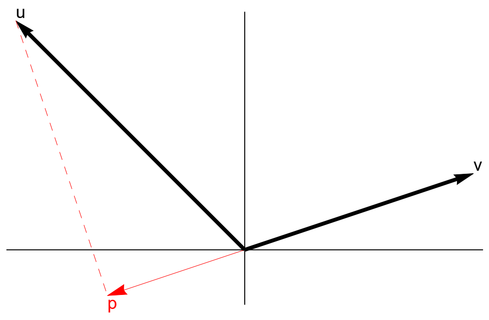


Figure 9: Orthogonal Projection (Obtuse Angle)

The same principles apply with higher dimensional vectors.

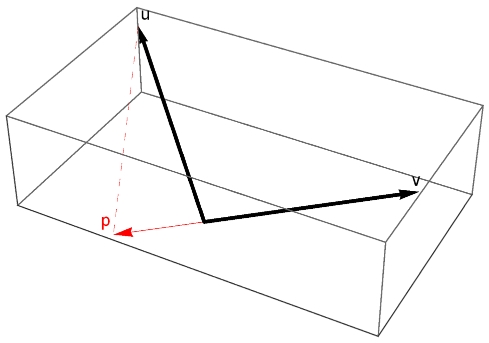


Figure 10: Orthogonal Projection (3D)

Orthogonal projections are common enough that WL builds this functionality into its `Projection` command.

4.2.4 Linear Equations

WL can solve large systems of linear equations. In this section, we provide some background.

Null Space

Suppose we have a linear map $f : X \to Y$. The null space of f comprises the elements of X that map to 0_Y. As a simple example, consider a real N-vector a, which represents a linear map $\mathbb{R}^N \to \mathbb{R}$. The null space of the vector a is the set of vectors that are transformed to 0 by a: $\text{null}(a) = \{x \mid a.x = 0\}$.

The WL command `NullSpace` takes a matrix argument and returns a basis for the nullspace of that matrix.

```
mA = Transpose[{{1, 2}, {3, 4}}]; (* put our vectors in the columns *)
nA = NullSpace[mA]
```

```
{}
```

```
v1 = {1, 2}; v2 = {3, 6};
mA = Transpose[{v1, v2}]; (* put our vectors in the columns *)
nA = NullSpace[mA]
```

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

This gives us weights for a linear combination of our vectors that equals the zero vector.

```
-3 * v1 + 1 * v2
```

```
{0, 0}
```

```
mA.Transpose[nA]
```

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

Suppose we want to characterize the null space of the vector (1,2). One approach is to use Solve to find the equation of the null space:

```
Clear[x1, x2]
v1 = {1, 2};
Solve[v1.{x1, x2} == 0, {x1, x2}]
```

```
Solve: Equations may not give solutions for all "solve" variables.
```

$$\left\{\left\{x2 \to -\frac{x1}{2}\right\}\right\}$$

Another approach is to use WL's NullSpace command, which returns a basis for the null space. However, this only works on matrices, so we would have to wrap the vector in a list:

```
NullSpace[{v1}]
```

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

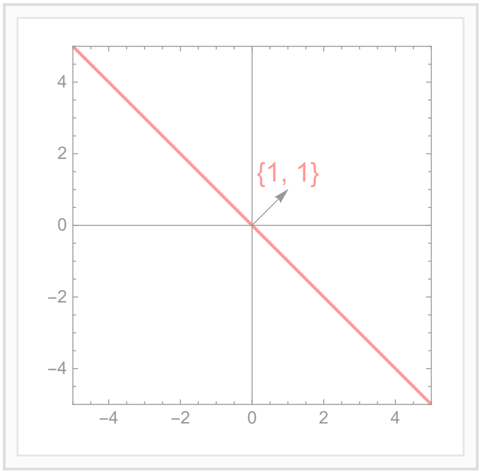
```
m1 = {{1}, {2}}
m1 // MatrixForm
NullSpace[m1]
```

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

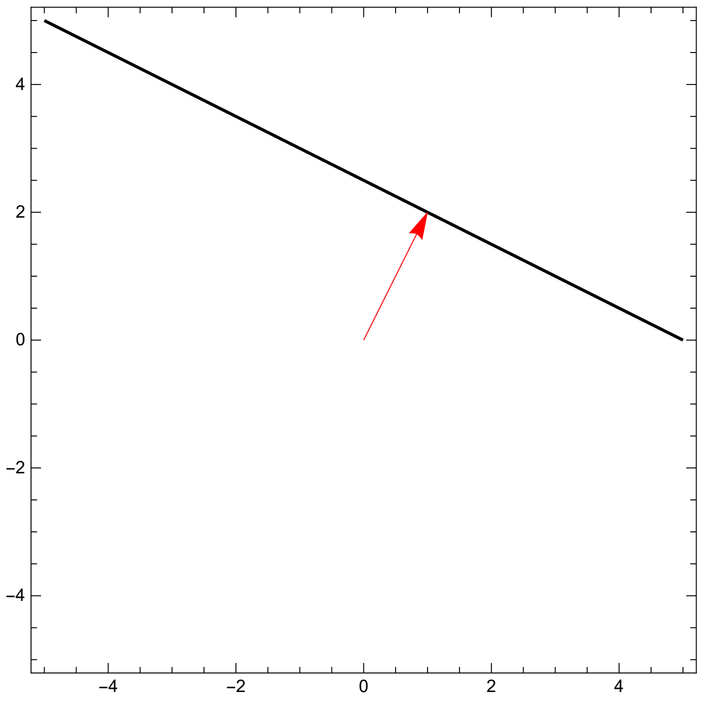
$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

```
{}
```

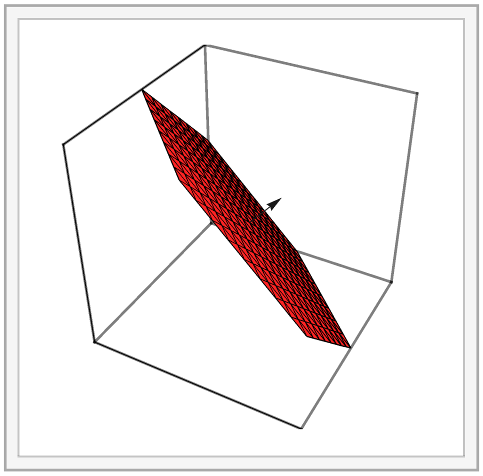
Illustrate Null Space: 2D



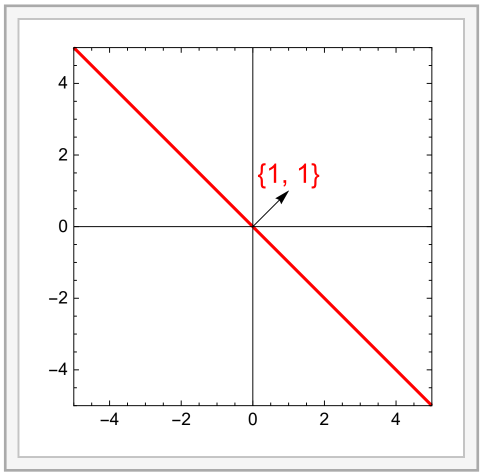
```
v = {1, 2}
ContourPlot[v.{x1, x2} == 5, {x1, -5, 5}, {x2, -5, 5}, Epilog → {Red, Arrow[{{0, 0}, v}]}]
{1, 2}
```



Illustrate Nullspace: 3D

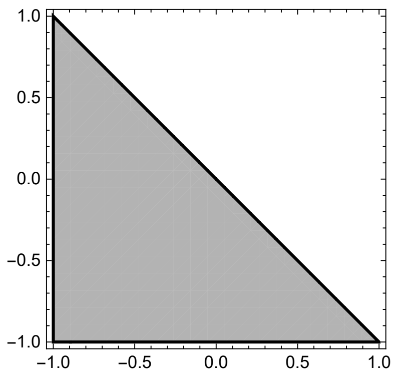


Affine Spaces: Illustrating Hyperplanes



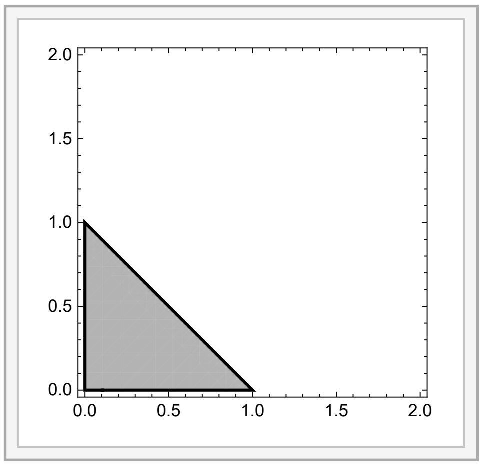
Inequalities

```
RegionPlot[x + y ≤ 0, {x, -1, 1}, {y, -1, 1}, ImageSize → 200]
```



Budget Constraints

```
Manipulate[RegionPlot[px * x + py * y ≤ w, {x, 0, 2}, {y, 0, 2}, ImageSize → 200],
 {px, 1, 5}, {py, 1, 5}, {{w, 1}, 0, 2},
 Paneled → True,
 Method → {"ShowControls" → False}]
```



4.2.5 Linear Functions

Lines

Consider any two distinct points, p_1 and p_2. The line segment from p_1 to p_2 is the shortest path between the two points, and the length of this path is the distance between the two points. This line segment is the set of points that can be produced as convex combinations of p1 and p2. The line through p1 and p2 is the set of points that can be produced as affine combinations of the two points. That is, any point p on the line can be expressed as

$$p = (1 - t)\, p_1 + t\, p_2 = p_1 + t\, (p_2 - p_1)$$

This is called the parametric form of the line. We will refer to the point $(p_2 - p_1)$ as a “direction vector” for the line. To illustrate this, suppose our two distinct points are in the Cartesian plane: $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. Then we have

$$x = x_1 + t(x_2 - x_1)$$
$$y = y_1 + t(y_2 - y_1)$$

which implies

$$(y_2 - y_1)(x - x_1) = (x_2 - x_1)(y - y_1)$$

To see this, note that we can rewrite our system as

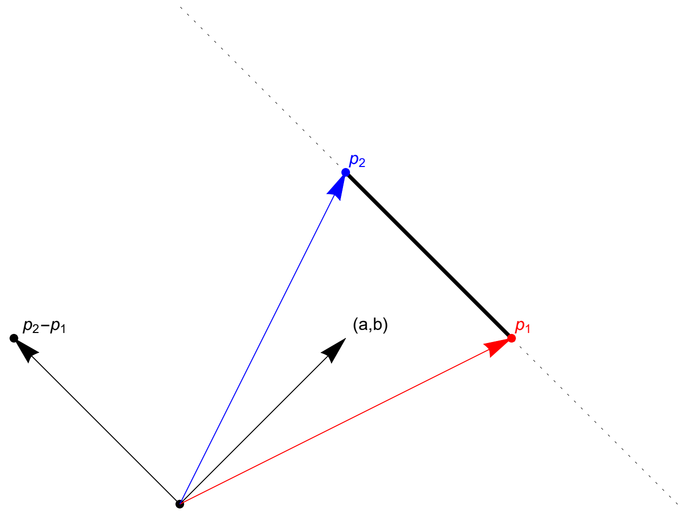
$$(y_2 - y_1)(x - x_1) = (y_2 - y_1)(x_2 - x_1)\,t$$
$$(x_2 - x_1)(y - y_1) = (x_2 - x_1)(y_2 - y_1)\,t$$

This line is commonly represented by the equation

$$a\,x + b\,y + c = 0$$

where e.g. $a = y_2 - y_1$, $b = x_1 - x_2$, and $c = y_1(x_2 - x_1) - x_1(y_2 - y_1) = y_1 x_2 - x_1 y_2$.

```
{1, 1}
```



If $x_1 \neq x_2$ we define the slope m of the line by

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Otherwise the slope is undefined. Two distinct lines in a plane are “parallel” if they do not intersect; the minimum distance from one to the other is constant. Parallel lines have a common slope. Two parallel lines cut the horizontal axis at the same angle, denoted by θ. The slope and the angle are related by

$$m = \mathrm{Tan}[\theta]$$
$$\theta = \arctan(m)$$

$$m = \frac{y2 - y1}{x2 - x1}$$

```
ArcTan[-1, 1]
```

```
-1
```

$$\frac{3\pi}{4}$$

Two lines in a plane are “perpendicular” if their intersection creates common adjacent angles (i.e., right angles). The product of their slopes is −1 (if defined). The dot product of their direction vectors is 0.

4.3 Lists of Lists as Matrices

This section focuses on matrix creation and manipulation. We represent a matrix as a rectangular list of lists: each inner list is a row of the matrix. (So be sure you have read the section on nested lists before reading the present section.) We can create matrices by explicitly enumerating the elements of each row, and the elements may be symbolic as well as numeric.

```
Clear[a, b, c, d]
mA = {{a, b}, {c, d}};
```

To produce a traditional display of the matrix, we use the `MatrixForm` command. This is often visually convenient, but be aware that the value returned by this command is a matrix form, not a matrix. That is, the value is a display wrapper, and so it will not respond correctly to matrix operations.

```
MatrixForm[mA]
```

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

Remember that WL uses capital letters for built-in commands, and some of these (C,D,E,I,K,N, and O) are single letter commands. So although traditional textbook notation uses capital letters for matrices, this is a risky habit for WL input. It is therefore a convention for each user defined symbol to begin with a lower-case letter.

4.3.1 Matrix Basics

A matrix is a rectangular array: each row has the same number of elements. We can use the `MatrixQ` command to determine whether an expression is in fact a rectangular array. The `MatrixQ` command accepts a second argument, which is used to test each element of the matrix. For example, we may wish to test whether we are dealing with a numeric matrix.

Expression	Result
MatrixQ[{{1,2},{3,4,5}}]	False
MatrixQ[{{1,2},{3,4}}]	True
MatrixQ[{{1,2},{3,4}},NumberQ]	True
MatrixQ[{{a,b},{c,d}}]	True
MatrixQ[{{a,b},{c,d}},NumberQ]	False

Since our representation of a matrix is just a rectangular list of lists, you can create matrix rows any way you create lists. You can use the `Dimensions` command to retrieve the shape of your matrix as a list containing the number of rows and the number of columns. You can find the smallest and largest elements with `Min` and `Max`, or both together with `MinMax`.

Expression	Result
lst01={1,2,3}	{1, 2, 3}
lst02={2,3,4}	{2, 3, 4}
mA={lst01,lst02}	{{1, 2, 3}, {2, 3, 4}}
MatrixQ[mA]	True
Dimensions[mA]	{2, 3}
MinMax[mA]	{1, 4}

Convenient Matrix Entry in a Notebook

In a notebook, one can enter expressions in a two-dimensional layout by using `CTRL+,` to add columns and `CTRL+ENTER` to add rows. By default, this produces a list of lists of placeholders. The placeholders allow for convenient keyboard entry of matrix elements, since you can use `TAB` to move from placeholder to placeholder.

Another easy way to type matrices is to pick the Basic Math Assistant palette, scroll down to the first typesetting tab, and click the matrix template. (Note that by default matrices display rounded brackets.) This will give you a 2 by 2 matrix template. Again, use `CTRL+ENTER` to add a row; use `CTRL+,` to add a column. Here is a matrix created in this way inline in a text cell: $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.

For typing efficiency, you should only use the Basic Math Assistant to create matrices when you forget the keyboard shortcuts. If you hover your mouse over the template in the Math Assistant, you will see the keyboard shortcuts for matrix creation.

Creating Matrices with WL Commands

When creating matrices from scratch, we usually use the `Array` command. (Recall from our discussion of lists that `Table` is generally a possible alternative to `Array`.) This is convenient whenever we can describe a matrix as a function of its indexes, even if the function is purely symbolic. (Here we use the double-slash postfix notation to display the result with `MatrixForm`.)

```
ClearAll[f, a]
Array[f, {2, 3}] // MatrixForm
```

```
( f[1, 1]  f[1, 2]  f[1, 3] )
( f[2, 1]  f[2, 2]  f[2, 3] )
```

We can therefore use the `Array` command to conveniently create common symbolic matrices. Here we illustrate this with the `SlotSequence` command, using its double-hash (`##`) shorthand notation. In this particular case, we are using `SlotSequence` to supply all the function arguments to the `Subscript` function.

Array[$a_{\#\#}$ &, {2, 3}] // MatrixForm

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}$$

For the creation of many common matrix types, WL offers specialized commands. For example, we could create a 2×3 matrix of constants as `Array[a&,{2,3}]`, but it is more common to use `ConstantArray`.

```
ConstantArray[a, {2, 3}] // MatrixForm
```

$$\begin{pmatrix} a & a & a \\ a & a & a \end{pmatrix}$$

Similarly, one can readily create a 3×3 scalar matrix as `Array[{r,c}⟼If[r==c,a,0],{3,3}]`, but it is more conventional to make use of `DiagonalMatrix` and `ConstantArray`.

```
DiagonalMatrix[ConstantArray[a, 3]] // MatrixForm
```

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

The `DiagonalMatrix` accepts a single argument, the diagonal elements as a list. The list members can be symbolic or numeric.

```
DiagonalMatrix[{1, 2, 3}] // MatrixForm
```

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

The special case where our scalar matrix has a value of 1 along the diagonal produces an identity matrix. (We explore this concept of identity below.) We can use the `IdentityMatrix` command in this case.

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

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

As a slightly more complex example, let us create another 3×5 matrix, this time with elements that are the first fifteen positive integers.

```
m3by5 = Array[{r, c} ↦ (r - 1) * 5 + c, {3, 5}];
m3by5 // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

In this particular case, it would be more common to use the `Partition` and `Range` commands.

```
m3by5 == Partition[Range[15], 5]
```

```
True
```

Advanced

As an implementation detail, matrices created with `Array` are filled in row-major order. Here is an easy way to see that: we will create our subscripted indexes by incrementing a counter.

```
ClearAll[a]
Module[{i = 1}, Array[a_i++ &, {2, 3}]] // MatrixForm
```

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \end{pmatrix}$$

Following our earlier discussion of variable scope, here we use `Module` to localize (and initialize) the variable *i* in this expression. This ensures that we do not alter any already existing global value of *i* when we assign new values inside the module. Note that this approach discards the input arguments provided during construction and instead depends on the array being filled in row-major order, which is true but undocumented. It is usually a bad idea to rely on undocumented features, however, so we will not pursue this further. If we want a matrix as above, we can always find an alternative. For example,

```
Partition[Array[a_# &, 6], 3] // MatrixForm
```

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \end{pmatrix}$$

4.3.2 Simple Matrix Operations

This section will focus on matrix creation and manipulation. Remember that by convention, user defined symbols begin with a lower-case letter.

Indexing into Matrices

Recall that a matrix is just a rectangular list of lists and that the `Array` command is therefore often useful for matrix creation.

```
mA = Array[{r, k} ↦ 10 * r + k, {3, 5}, {0, 1}];
mA // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$

From a multidimensional list, we can extract a part of a part in the obvious ways (e.g., sequential double brackets). But as a convenient shorthand, we can just use a comma-separated list of indexes. Use an index of `All` to select all rows or all columns.

Expression	Result	Comment
mA⟦2⟧	{11, 12, 13, 14, 15}	second row
mA⟦2⟧⟦3⟧	13	the 2,3 element
mA⟦2,3⟧	13	the 2,3 element (shorthand)
mA⟦All,3⟧	{3, 13, 23}	the third column

One can even arbitrarily select from rows or, more surprisingly, columns. Use a list of arbitrary indexes to select an arbitrary set of columns or rows. (You can even change the order or include repetitions.)

```
mA⟦All, {3, 1, 2}⟧ // MatrixForm (* reordered columns *)
```

$$\begin{pmatrix} 3 & 1 & 2 \\ 13 & 11 & 12 \\ 23 & 21 & 22 \end{pmatrix}$$

If you want a subset whose indexes can be described by `Range`, you should use `Span`, which has `;;` as a special infix notation. With the infix notation, we write `Span[i,j,k]` as `i;;j;;k`. If we omit `i` it is assumed to be 1. If we omit `j` it is assumed to be `All`. If we omit `k` it is assumed to be 1, but then we must omit the second double semicolon.

```
mA⟦All, ;; ;; 2⟧ // MatrixForm (* all rows, every other column *)
```

$$\begin{pmatrix} 1 & 3 & 5 \\ 11 & 13 & 15 \\ 21 & 23 & 25 \end{pmatrix}$$

Basic Vector Operations on Matrices

In WL, any rectangular list of lists can be treated as a matrix. The basic vector operations on matrices are very natural: add and subtract matrices with the usual + and - operators, and do scalar multiplication by premultiplying your matrix by any scalar. Here are some examples with 2×2 matrices.

m1	2*m1	mA	mA+m1	mA−m1
$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	$\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$

It is common to refer to a matrix with only one row as a row vector and a matrix with only one column as a column vector. As with any other matrices, one can perform the usual vector operations with row vectors and column vectors.

Matrix Multiplication

The binary operation commonly called “matrix multiplication” produces a new matrix $C = A.B$ as follows: the element $c_{r,k}$ (in row r and and column k) of the product is produced by multiplying corresponding elements in row r of A and column k of B, and then summing the products. This requires that the number of columns in A equal the number of rows in B, so that we can match up the elements. We can use the `Dot` command to perform matrix multiplication, or we can use a “dot” (i.e., a period or full-stop) as an infix shorthand for the `Dot` command.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} . \begin{pmatrix} a & b \\ c & d \end{pmatrix} == \begin{pmatrix} a+2c & b+2d \\ 3a+4c & 3b+4d \end{pmatrix}$$

The shorthand syntax for matrix multiplication may be a bit of a surprise for new users. WL allows use of either a space or an asterisk to produce an elementwise (Hadamard) product. Use of a simple space is always considered an implicit `Times` operator, never an implicit `Dot` operator. So one must use a "dot" to produce matrix multiplication. Here are examples with two 2×2 matrices that emphasize this difference.

mA	mB	mA*mB	mA mB	mA.mB
$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$	$\begin{pmatrix} a & 2b \\ 3c & 4d \end{pmatrix}$	$\begin{pmatrix} a & 2b \\ 3c & 4d \end{pmatrix}$	$\begin{pmatrix} a+2c & b+2d \\ 3a+4c & 3b+4d \end{pmatrix}$

To reiterate, matrix multiplication is a dot product. The (r, k)-th element of the product matrix is formed from the r-th row of the premultiplying matrix and the k-th column of the postmultiplying matrix. For example, we can form the (1, 2)-th element of the product matrix as the dot product of the first row of the premultiplying matrix and the second column of the postmultiplying matrix.

```
mA⟦1⟧.mB⟦All, 2⟧ == (mA.mB)⟦1, 2⟧
```

```
True
```

This means that, for the matrices to be conformable for multiplication, the premultiplying matrix must have as many columns as the postmultiplying matrix has rows. It also means that the order of the operands affects the result: matrix multiplication is generally not commutative. (However, matrix multiplication is always associative.)

Advanced

The infix operators * and . provide a shorthand for the WL commands `Times` and `Dot`. In fact, WL syntax allows you to use these binary commands as infix operators by surrounding them with tildes. However, this usage is relatively rare.

```
mA * mB === mA ~Times~ mB (* element-by-element multiplication *)
mA.mB === mA ~Dot~ mB (* matrix multiplication *)
```

```
True

True
```

When we repeated multiply a matrix by itself, we produce a matrix power. in WL, we can produce matrix powers with the `MatrixPower` command, which is based on repeated application of the matrix product.

```
MatrixPower[{{a, b}, {c, d}}, 2] // MatrixForm
```

$$\begin{pmatrix} a^2 + b\,c & a\,b + b\,d \\ a\,c + c\,d & b\,c + d^2 \end{pmatrix}$$

You cannot use the caret (`^`) operator, since that is a shorthand for `Power` and not for `MatrixPower`. The `Power` command can be applied to matrices (because it has the `Listable` attribute), but it operates elementwise. That is, it just raises each matrix element to the specified power.

```
{{a, b}, {c, d}}^2 // MatrixForm
```

$$\begin{pmatrix} a^2 & b^2 \\ c^2 & d^2 \end{pmatrix}$$

Identity, Transpose, and Inverse

Recall that a diagonal matrix is a square matrix with zeros everywhere except possibly along its main diagonal. A scalar matrix is a diagonal matrix where all the diagonal elements have the same value. An identity matrix is a diagonal matrix where the diagonal elements are 1. An identity matrix is thus a square matrix that has ones on its diagonals, all other elements being zero. We can use `IdentityMatrix[n]` to produce an $n \times n$ identity matrix. Here we produce a list of the first four identity matrices, and we then use the `Row` command to nicely display them as a single row. (The `Spacer` option allows us to push them apart a bit, as a nicer display.)

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

An identity matrix plays the role of an identity for matrix multiplication. Matrix-multiplication by an identity matrix has “no effect”, in the sense that $A.I == I.A == A$. Consider for example the following matrix multiplications.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} == \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.\begin{pmatrix} a & b \\ c & d \end{pmatrix} == \begin{pmatrix} a & b \\ c & d \end{pmatrix}.\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The number 1 is the multiplicative identity for numbers. If there is a number x^{-1} such that $1 == x\,x^{-1} == x^{-1}\,x$, we say that the number x has a multiplicative inverse Every real number except 0 has a multiplicative inverse. Similarly, a square matrix A has a multiplicative inverse if we can find a matrix A^{-1} such that $I == A.A^{-1} == A^{-1}.A$. (From this definition, we see that only square matrices can have inverses, and we see that A is also the inverse of the matrix A^{-1}.) We can use the `Inverse` command to produce a matrix inverse, when one exists.

```
mA = {{a, b}, {c, d}};
Inverse[mA] // MatrixForm
```

$$\begin{pmatrix} \frac{d}{-b\,c+a\,d} & -\frac{b}{-b\,c+a\,d} \\ -\frac{c}{-b\,c+a\,d} & \frac{a}{-b\,c+a\,d} \end{pmatrix}$$

If we premultiply or postmultiply a matrix by its inverse, we get the identity matrix.

```
IdentityMatrix[2] == Inverse[mA].mA == mA.Inverse[mA] // Simplify
```

```
True
```

A square matrix is called “singular” if it has no inverse. Our symbolic matrix is singular if $a\,d == b\,c$, but WL nevertheless produces the generally useful symbolic result. It even allows multiplication by this inverse. This is an example of how a duty of mathematical thinking still falls upon the user: you will not be warned of the limitations of this symbolic solution. However, you will be warned if you try to use the `Inverse` command on a singular matrix.

```
Inverse[{{1, 2}, {1, 2}}];
```

```
Inverse: Matrix {{1, 2}, {1, 2}} is singular.
```

Just as we can understand the solution of the single real equation $a\,x == b$ for the variable x as resulting from multiplying both sides of the equation by a^{-1}, we can understand the solution of the matrix equality $A\,x == b$ for the vector variable x as resulting from premultiplying both sides of the equation by A^{-1}. In a textbook setting, we will often represent solutions in just this way. In the single equation case, we need $a \neq 0$ for this solution procedure to work. When working with the matrix equality, we similarly need the coefficient matrix A to be nonsingular. Recalling our earlier discussion of linear dependence, we can say that a necessary and sufficient condition for a square matrix to be nonsingular is that its columns be linearly independent. We will explore this further in a later chapter. Note that if A and B are nonsingular, then $(A.B)^{-1} == B^{-1}.A^{-1}$. Finally, if we find inverses B and C for a matrix A, the $B.A.C$ evaluates to both B (since $A.C == I$) and to C (since $B.A == I$), so $B == C$: the inverse of a matrix is unique.

A commonly needed matrix operation is transposition: switching the rows and columns. (If we do this twice, we get back our original matrix.)

The transpose of $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is $\begin{pmatrix} a & c \\ b & d \end{pmatrix}$.

A symmetric matrix is a matrix that is unchanged by transposition. In a symmetric matrix, the *r*, *k*-th element equals the *k*, *r*-th element. Evidently, a symmetric matrix must be square. Every diagonal matrix, including every identity matrix, is symmetric.

We use the `Transpose` command to produce a matrix transpose. Observe that $(A.B)^{\tau} == B^{\tau}.A^{\tau}$. It follows that $(A^{\tau})^{-1} == (A^{-1})^{\tau}$ for any nonsingular matrix A (since an identity matrix is symmetric). For example,

```
mA = {{a, b}, {c, d}};
Inverse@Transpose[mA] == Transpose@Inverse[mA]
```

```
True
```

If we matrix-multiply a matrix and its transpose, the result is a symmetric matrix.

```
(mS = mA.Transpose[mA]) // MatrixForm
```

$$\begin{pmatrix} a^2 + b^2 & a\,c + b\,d \\ a\,c + b\,d & c^2 + d^2 \end{pmatrix}$$

In this 2×2 case we can easily confirm symmetry by inspection. In general, we must check programmatically that a matrix is equal to its transpose.

```
mS == Transpose[mS]  (* test for symmetry *)
```

```
True
```

It is possible that the transpose of a matrix is also its inverse. Such matrices are called “orthogonal”, although they might better be called orthonormal. The columns of an orthogonal matrix are unit vectors that are orthogonal to each other. Naturally, an identity matrix is an orthogonal matrix, as is any matrix created by permuting the rows of an identity matrix. We will have more to say about orthogonal matrices later on.

4.3.3 Some Linear Transformations in Two Dimensions

We can use a 2×2 real matrix to represent a linear transformation of each point in the Cartesian plane to another such point.

General Linear Transformation $(\mathbb{R}^2 \to \mathbb{R}^2)$

The following represents a general linear transformation of 2-vectors to 2-vectors.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} . \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a\,x + b\,y \\ c\,x + d\,y \end{pmatrix}$$

Note that the zero vector always maps to the zero vector. We are going to visualize some special transformations by looking at transformations of the points of the unit square. The unit square has corners at the following points: (0, 0), (1, 0), (1, 1), and (0, 1). Let us create a 2 by 4 matrix, where each column represents one of those corners, and then plot those four points. The resulting matrix is

$$mS = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix};$$

We can plot these four points with `ListPlot`.

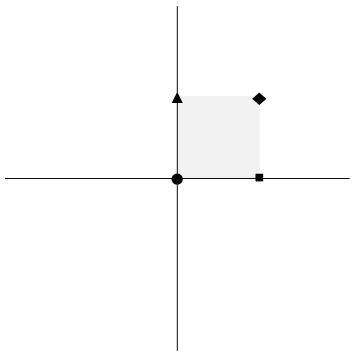


Figure 11: Corners of the Unit Square

We can premultiply this representation of the unit square by any 2 by 2 matrix, and each column of the result represents a transformed point.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & a & a+b & b \\ 0 & c & c+d & d \end{pmatrix}$$

The first column of the result is of course the zero vector. The next is the first column of our transformation matrix. Then comes the sum of the two columns of our transformation matrix. And last, we find the second column of our transformation matrix.

Scale Transformations

Scale transformations uniformly scale all first coordinates and uniformly scale all second coordinates. Any diagonal matrix can be interpreted in terms of such scale transformations. The entries along the diagonal are called the scaling factors.

Let $\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$ be our two–dimensional scale–transformation matrix.

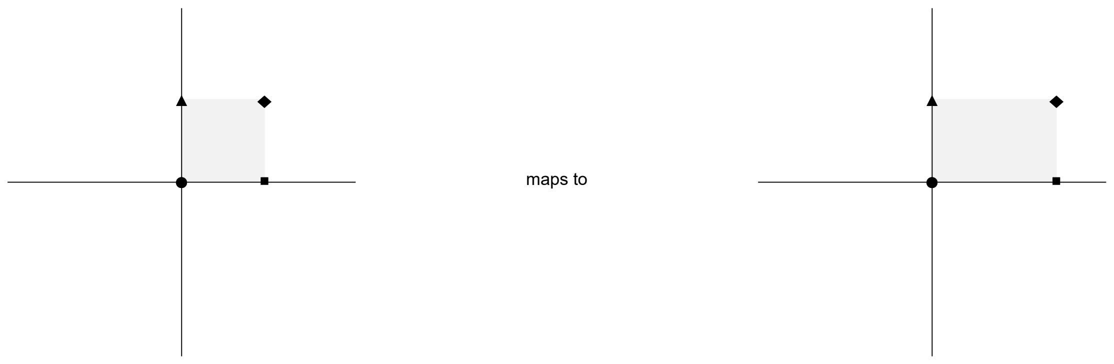
$$\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\, s_x \\ y\, s_y \end{pmatrix}$$

$$\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \text{msq} = \{\{s_x, 0\}, \{0, s_y\}\}.\text{msq}$$

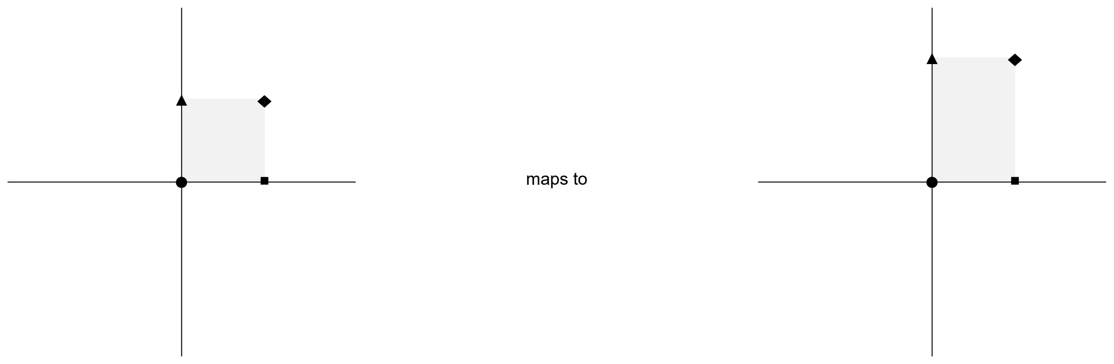
Examples of Scale Transformations

A scale transformation simply rescales the *x*-values, the *y*-values, or both.

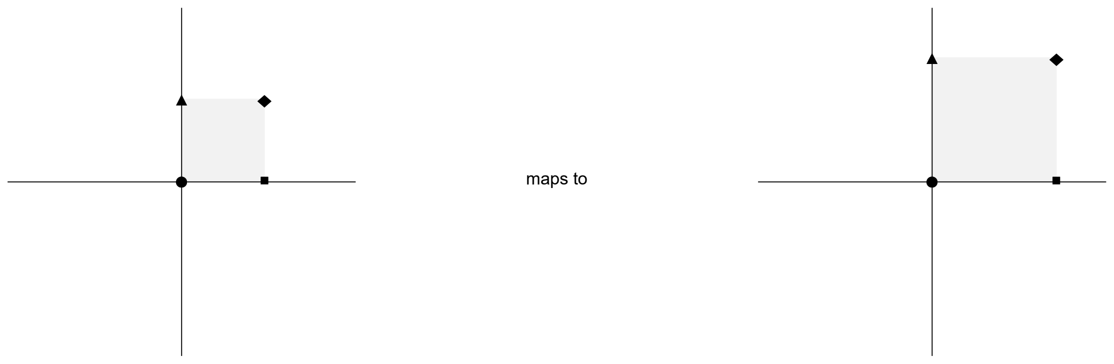
The matrix $\begin{pmatrix} 1.5 & 0 \\ 0 & 1 \end{pmatrix}$ will scale first coordinates by 150%:



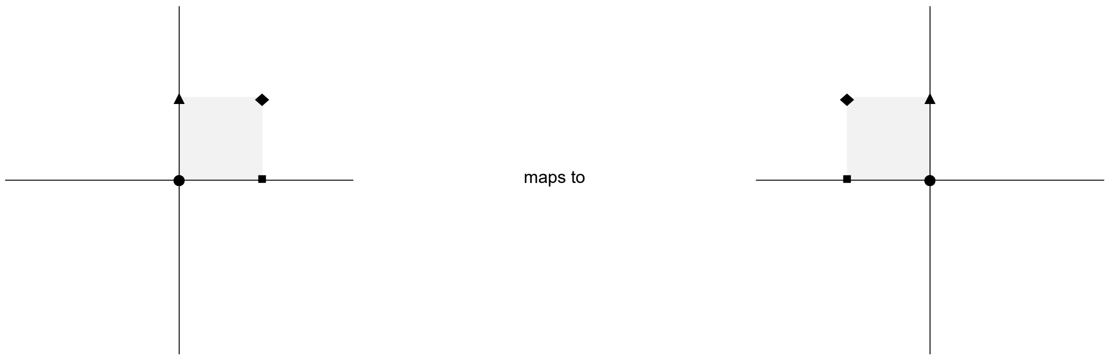
The matrix $\begin{pmatrix} 1 & 0 \\ 0 & 1.5 \end{pmatrix}$ will scale second coordinates by 150%:



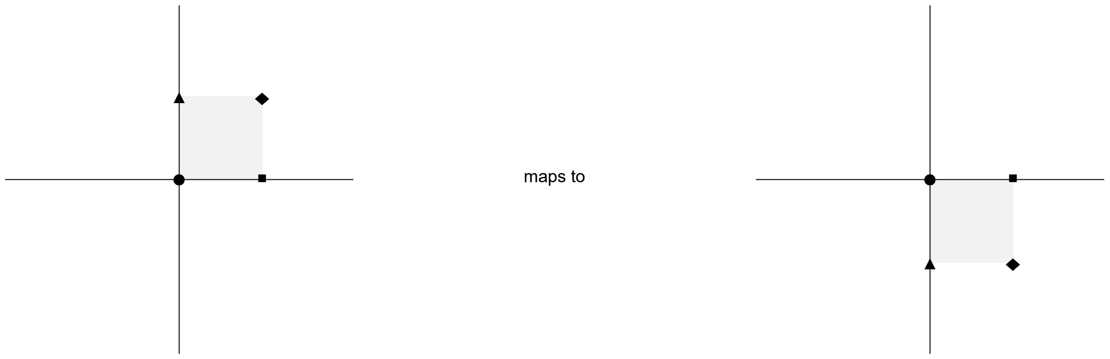
The matrix $\begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$ will scale the both coordinates by 150%:



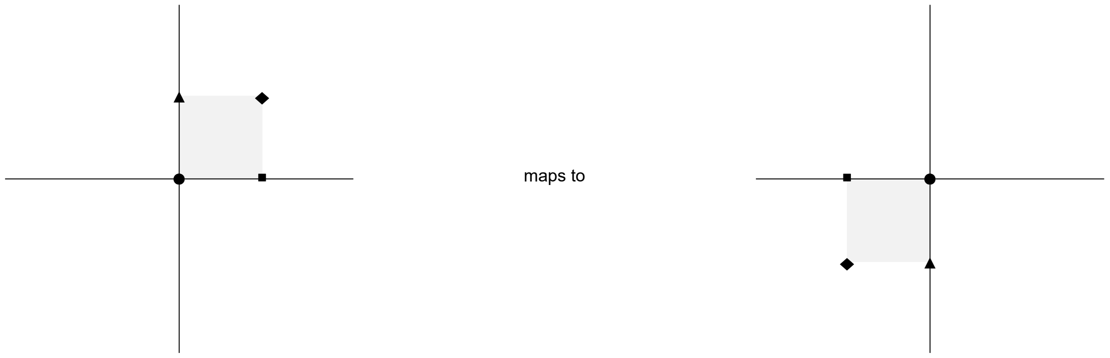
The matrix $\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$ will scale first coordinates by −100%:



The matrix $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ will scale second coordiantes by −100%:



The matrix $\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$ will scale both coordinates by −100%:



Shear Transformations

A shear transformation in any direction proportionally displaces each point in that (signed) direction. Create a matrix for shearing as follows.

```
ClearAll[a, b, c, x, y]
(mT = {{1, b}, {c, 1}}) // MatrixForm
```

$$\begin{pmatrix} 1 & b \\ c & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & b \\ c & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + b\,y \\ c\,x + y \end{pmatrix}$$

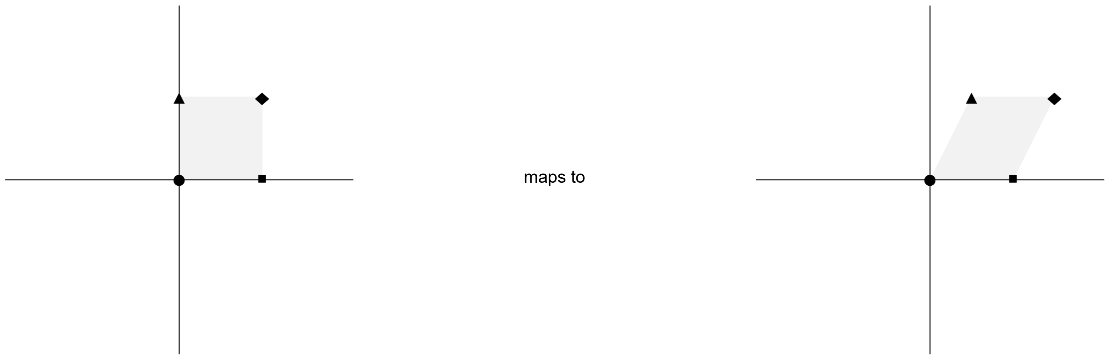
Here *b* determines the horizontal shearing, and *c* determines the vertical shearing. A horizontal shear transformation has $c = 0$; a vertical shear transformation has $b = 0$.

A horizontal shear transformation sets c=0: $\begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix}$

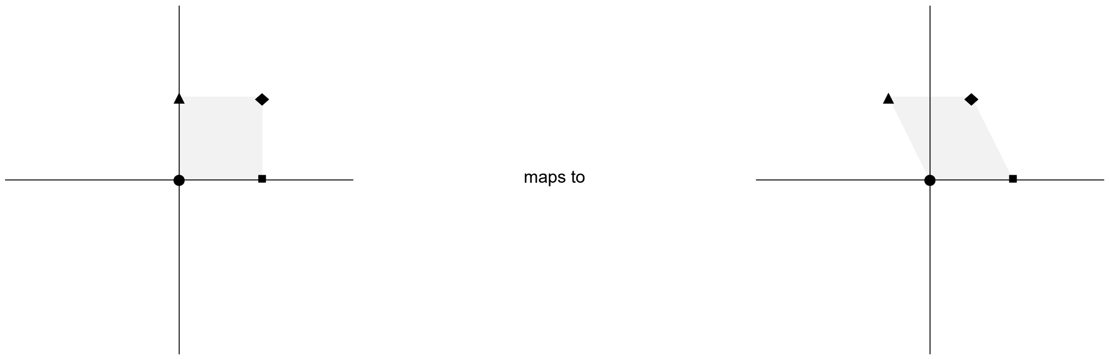
A vertical shear transformation sets b=0: $\begin{pmatrix} 1 & 0 \\ c & 1 \end{pmatrix}$

Examples of Shear Transformations

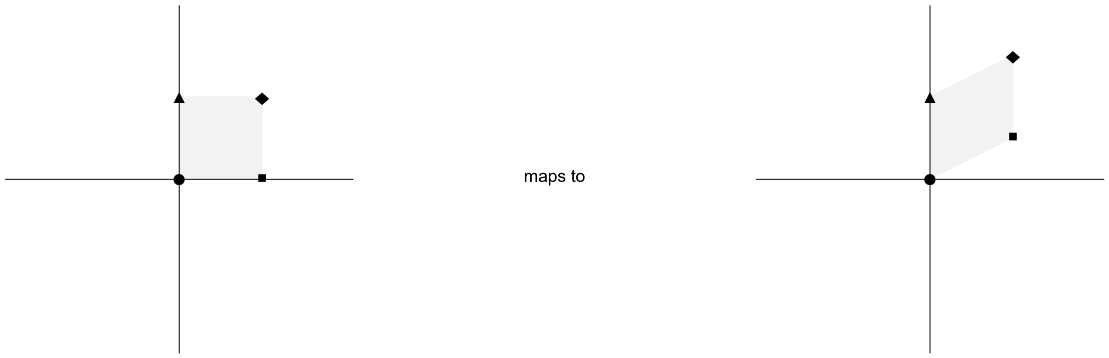
The matrix $\begin{pmatrix} 1 & 0.5 \\ 0 & 1 \end{pmatrix}$ produces a horizontal shear to the right:



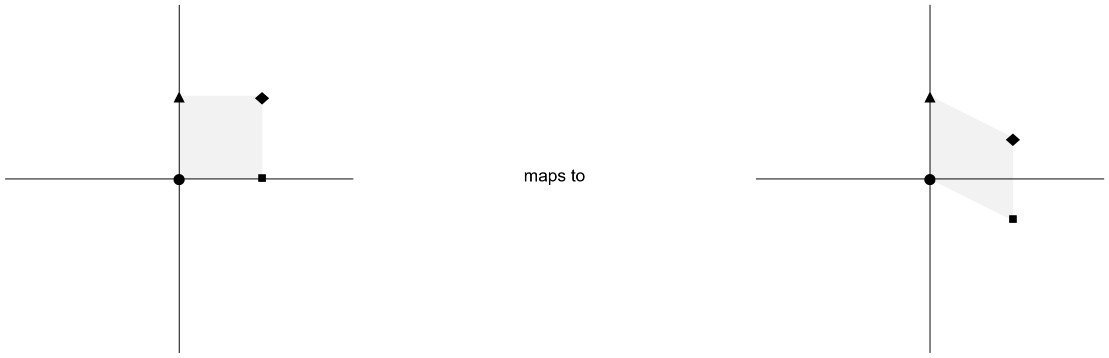
The matrix $\begin{pmatrix} 1 & -0.5 \\ 0 & 1 \end{pmatrix}$ produces a horizontal shear to the left:



The matrix $\begin{pmatrix} 1 & 0 \\ 0.5 & 1 \end{pmatrix}$ produces a vertical shear upwards:



The matrix $\begin{pmatrix} 1 & 0 \\ -0.5 & 1 \end{pmatrix}$ produces a vertical shear downwards:



Looking at the transformation of the entire unit square, we see that either a horizontal or a vertical shear transformation will leave unchanged the total area of the transformed polygon.

4.3.4 Orthogonal Transformations

A matrix is called “orthogonal” if its transpose is its multiplicative inverse: $A^{-1} = A^{\tau}$. That is, the columns of of an orthogonal matrix are an orthonormal set of vectors. The associated linear transformation transformation is called an orthogonal transformation. An orthogonal transformation preserves length. It also preserves the angle between vectors. Simple reflections through either axis or through the origin provide examples of orthogonal transformations. Rotation by a fixed angle around the origin is another example. We can produce more orthogonal transformations by combing reflection and rotation.

Permutation Matrices

We produce can produce permutation matrices by shuffling the rows of an identity matrix. Here are all 6 of the 3×3 permutation matrices.

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

Note that transposition of a permutation matrix produces a permutation matrix. (In this particular case, all but two of the permutation matrices are symmetric.)

Exercise: show for each of these permutation matrices that its transpose is its inverse. Many of these permutation matrices are symmetric, in which case the matrix is also its own inverse. Such matrices are called involutory, and they are a kind of square root for the identity matrix. Note that the permutation matrices we produce via a single row exchange are involutory.

Since a permutation matrix results from reordering the rows of an identity matrix, it represents the linear transformation that performs the same reordering of coordinates on the vector it transforms. For example, let us rearrange the rows of the 4×4 identity matrix as {4, 1, 2, 3}. Then the matrix should transform $\{x_1, x_2, x_3, x_4\}$ to $\{x_4, x_1, x_2, x_3\}$.

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} . \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} == \begin{pmatrix} x_4 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Rotation Transformations

Out discussion of rotations transformations will focus on the two-dimensional case. Given a point (x, y), we will rotate it around the origin by θ radian (counterclockwise). This produces a new point $(\cos[\theta]\, x - \sin[\theta]\, y, \sin[\theta]\, x + \cos[\theta]\, y)$. For any given θ, we can represent this transformation as by the following matrix:

$$\begin{pmatrix} \text{Cos}[\theta] & -\text{Sin}[\theta] \\ \text{Sin}[\theta] & \text{Cos}[\theta] \end{pmatrix}$$

For example, the vector (1, 0) is transformed to $(\cos[\theta], \sin[\theta])$.

```
RotationMatrix[θ].{1, 0}
```

```
{Cos[θ], Sin[θ]}
```

We can also use the `RotationMatrix` command to determine the rotation matrix required to get one vector to point in the same direction as another.

```
Simplify[RotationMatrix[{{1, 0}, {Cos[θ], Sin[θ]}}], θ > 0]
```

```
{{Cos[θ], -Sin[θ]}, {Sin[θ], Cos[θ]}}
```

By inspecting the rotation matrix, we see that we can similarly produce a clockwise rotation through an angle of θ radians with the transpose of this matrix. The two operations are clearly inverses, so their composition should yield an identity matrix. Let us try it.

```
mR = RotationMatrix[θ];
mR.mRᵀ // MatrixForm
```

$$\begin{pmatrix} \mathrm{Cos}[\theta]^2 + \mathrm{Sin}[\theta]^2 & 0 \\ 0 & \mathrm{Cos}[\theta]^2 + \mathrm{Sin}[\theta]^2 \end{pmatrix}$$

That might not be quite what you expected. However, if we recall the trigonometric identity $1 = \mathrm{Cos}[\theta]^2 + \mathrm{Sin}[\theta]^2$, then we can see that the result is the 2×2 identity matrix. We can use the `Simplify` command to force recognition of this identity.

```
mR.mRᵀ // Simplify // MatrixForm
```

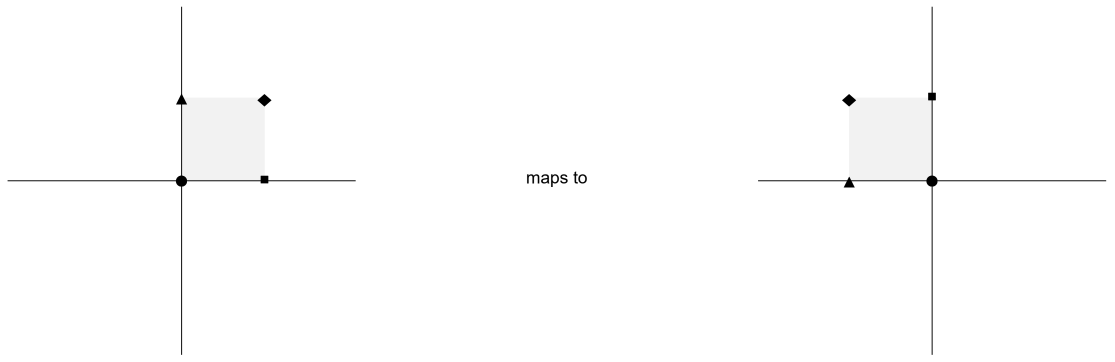
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Let us examine a simple rotation example. Suppose we want to map points in $\mathbb{R}^2$ to points rotated counter-clockwise by $\pi/2$ radians (90 degrees). Use the `ReplaceAll` command to replace θ with $\pi/2$ in matrix R. This gives us the following transformation matrix.

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

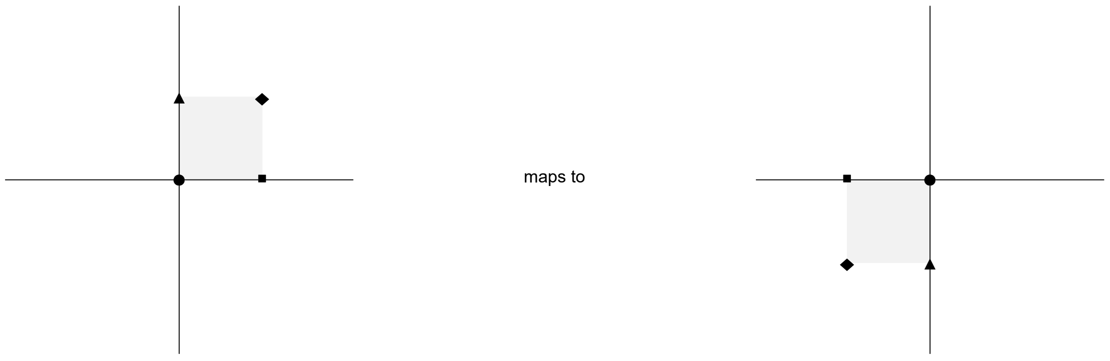
Examining the matrix, we see that when applied to a point (x, y) it will swap the x and y coordinates after changing the sign of the y coordinate. So the point (x, y) rotates to the point ($-y$, x). For example, (1, 2) would become (−2, 1). The dot product of these two vectors is 0. Recall from our discussion of vectors that the dot product of two perpendicular vectors is 0. This is true of our two vectors, and it is also true of any scalar multiples of these vectors.

The matrix $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$ produces a 90 degree counterclockwise rotation:



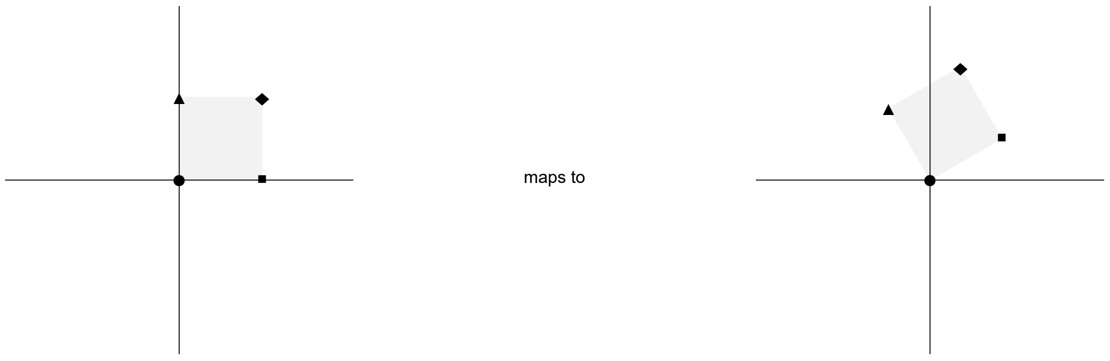
If we apply this same transformation again, we get another counter-clockwise rotation of $\pi/2$ radians (90 degrees). We can produce the matrix for this composite transformation either directly (by setting $\theta = \pi$) or by multiplying our previous matrix by itself. (Examining the transformation matrix for a rotation of π radians, we recognize from our earlier discussion that it is also a particular scale transformation.)

The matrix $\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$ produces a 180 degree counterclockwise rotation:



As a final example, set $\theta = \pi/6$.

The matrix $\begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix}$ produces a 30 degree counterclockwise rotation:



4.3.5 Useful Matrix Manipulations

WL has powerful facilities for creating new matrices from existing matrices.

Reshaping Matrices

We have already encountered the most basic reshaping: the use of `Partition` to produce a matrix from an array.

```
(m12 = Partition[Range[12], 4]) // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

More generally, we can reshape a matrix with `ArrayReshape`. The new matrix is filled in row-major order (i.e., row by row).

```
ArrayReshape[m12, {2, 6}] // MatrixForm
```

```
( 1  2  3   4   5   6 )
( 7  8  9  10  11  12 )
```

A word of caution: if the original array has more elements than the new shape can accommodate, the excess elements are silently dropped.

```
ArrayReshape[m12, {2, 5}] // MatrixForm
```

```
( 1  2  3  4   5 )
( 6  7  8  9  10 )
```

If the original array has fewer elements than the new shape needs, the excess elements are silently padded as 0s. Or we can specify that the padding be some specified value.

```
ArrayReshape[m12, {2, 7}, 99] // MatrixForm
```

```
( 1  2   3   4   5   6   7 )
( 8  9  10  11  12  99  99 )
```

Elementwise Transformation

In working with lists, we met the `Map` command. Recall that `Map` accepts a function and a list as arguments, and applies the function to each element of the list. If we try this with a matrix, `Map` naturally applies the function to each inner list, since these are the elements of the outer list.

```
Clear[f]
mA = Partition[Range[6], 3];
Map[f, mA]
```

```
{f[{1, 2, 3}], f[{4, 5, 6}]}
```

Often our goal is quite different: rather than apply the function to a each list, we wish to apply it to each matrix element. Fortunately, `Map` accepts a third argument, which allows us to specify the level at which we wish to do our mapping.

```
Map[f, mA, {2}] // MatrixForm
```

```
( f[1]  f[2]  f[3] )
( f[4]  f[5]  f[6] )
```

Other times we may have an altogether different goal: to treat each inner list as the sequence of arguments for a function. For this we are again in luck: the `Apply` command also accepts a third argument, which provides a level specification. This means that instead of replacing the head of the matrix with our function, we can replace the head of each matrix row (i.e., each inner list).

```
Apply[f, mA, {1}]
```

```
{f[1, 2, 3], f[4, 5, 6]}
```

Since it is a common need, there is a shorthand for this particular case: `@@@` (three consecutive atmarks). Here is a very simple example, where we replace the head of each row with the `Plus` command, thereby producing the total of the row elements. (As an aside, we would usually perform this particular computation by mapping `Total` across the rows. But either approach is acceptable.)

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

```
{6, 15}
```

Enlarging Matrices

We often want to add rows or columns to a matrix. Since a matrix is just a list of lists, we can naturally append a vector as a row with the `Append` command.

mA = $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$; v = {a, b};

```
Append[mA, v] // MatrixForm
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ a & b \end{pmatrix}$$

Appending a vector as a column requires a bit more thought. Probably the most conceptually straightforward method is to append an item to each row, which we can accomplish with `MapThread`.

```
MapThread[Append, {mA, v}] // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & a \\ 3 & 4 & b \end{pmatrix}$$

Matrices with the same number of columns can be stacked vertically with `Join`.

mA = $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$; mB = $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$;

```
Join[mA, mB] // MatrixForm
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ a & b \\ c & d \end{pmatrix}$$

Matrices with the same number of rows can be stacked horizontally by adding a level specification as a third argument to `Join`. (One could alternatively `MapThread[Join,{mA,mB}]`.)

```
Join[mA, mB, 2] // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & a & b \\ 3 & 4 & c & d \end{pmatrix}$$

More generally, matrices can be stacked by making a “matrix of matrices” and then using the `ArrayFlatten` command. For example, we can stack two matrices horizontally by using them as the row elements of a matrix of matrices.

```
ArrayFlatten[{{mA, mB}}] // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & a & b \\ 3 & 4 & c & d \end{pmatrix}$$

Or we can stack them vertically by using them as the column elements of a matrix of matrices.

```
ArrayFlatten[{{mA}, {mB}}] // MatrixForm
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ a & b \\ c & d \end{pmatrix}$$

We can stack both horizontally and vertically at the same time. For example, let us create a 2×2 matrix of diagonal matrices and then flatten this to get a single matrix.

```
matmat = Partition[DiagonalMatrix[{#, #}] & /@ {a, b, c, d}, 2];
ArrayFlatten[matmat] // MatrixForm
```

$$\begin{pmatrix} a & 0 & b & 0 \\ 0 & a & 0 & b \\ c & 0 & d & 0 \\ 0 & c & 0 & d \end{pmatrix}$$

The `ArrayFlatten` command is even more flexible that this: if the dimensions are unambiguous, you can conveniently use a single constant to represent an entire constant matrix.

```
(mD = ArrayFlatten[{{mA, x}, {y, mB}}]) // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & x & x \\ 3 & 4 & x & x \\ y & y & a & b \\ y & y & c & d \end{pmatrix}$$

Another command that is sometimes useful for stacking matrices is `KroneckerProduct`. This command takes two matrices as inputs, and it uses each element of the first matrix to scalar multiply the second matrix. This operation sometimes proves useful in statistics and econometrics.

$\texttt{mA} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}; \texttt{mB} = \begin{pmatrix} a & b \\ c & d \end{pmatrix};$

```
KroneckerProduct[mA, mB] // MatrixForm
```

$$\begin{pmatrix} a & b & 2a & 2b \\ c & d & 2c & 2d \\ 3a & 3b & 4a & 4b \\ 3c & 3d & 4c & 4d \end{pmatrix}$$

If the first matrix is a constant array, we get tiling. For example, if the first input is a unit matrix, we get tiling of the second matrix.

```
KroneckerProduct[ConstantArray[1, {2, 3}], {{a, b}, {c, d}}] // MatrixForm
```

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

As an aside, we can use the `KroneckerProduct` command to produce the outer product of two arrays.

```
KroneckerProduct[{1, 2}, {a, b, c}] // MatrixForm
```

$$\begin{pmatrix} a & b & c \\ 2a & 2b & 2c \end{pmatrix}$$

This is more compact than the typical textbook representation of the vector outer product as the result of matrix multiplication of a column vector and a row vector. However, we can take that approach if we prefer.

Unstacking Matrices

Stacks of square matrices can be unstacked using the `Partition` command. To partition a matrix *A* into $n \times n$ blocks use `Partition[A,{n,n}]`.

```
(mDu = Partition[mD, {2, 2}]) // MatrixForm
```

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & \begin{pmatrix} x & x \\ x & x \end{pmatrix} \\ \begin{pmatrix} y & y \\ y & y \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{pmatrix}$$

We access the submatrices in the expected fashion.

```
mDu⟦1, 1⟧ // MatrixForm
```

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

More complex unstacking can be handled by `PartitionRagged`, which is currently in `Internal`.

```
mDu2 = Internal`PartitionRagged[mD, {{3, 1}, {3, 1}}]
MatrixForm@Map[MatrixForm, mDu2, {2}]
mDu2[[1, 1]] // MatrixForm
```

```
{{{{1, 2, x}, {3, 4, x}, {y, y, a}}, {{x}, {x}, {b}}}, {{{y, y, c}}, {{d}}}}
```

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 & x \\ 3 & 4 & x \\ y & y & a \end{pmatrix} & \begin{pmatrix} x \\ x \\ b \end{pmatrix} \\ \begin{pmatrix} y & y & c \end{pmatrix} & \begin{pmatrix} d \end{pmatrix} \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & x \\ 3 & 4 & x \\ y & y & a \end{pmatrix}$$

In the process of unstacking arrays, we come to a concrete realization that WL arrays can have more than two dimensions. In fact, the number of dimensions is arbitrary. Three dimensional arrays quite often prove useful -- for example, in image processing. However, this book mostly ignores them.

We may find it useful to unstack matrices in order to produce an inverse. For example, a block diagonal matrix is a matrix that can be constructed with square matrices along it diagonal and zeros everywhere else. For example,

```
(mC = ArrayFlatten[{{mA, 0}, {0, mB}}]) // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{pmatrix}$$

In such cases, we can form the matrix inverse by inverting the blocks along the diagonal.

```
mCI = ArrayFlatten[{{Inverse@mA, 0}, {0, Inverse@mB}}];
mCI.mC // Simplify // MatrixForm
```

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

Principal Submatrices

Submatrices formed by retaining only the elements from a subset of rows and from the same subset of indexes for the columns are called the principle submatrices of our original matrix. The number of rows (and columns) retained is the order of the submatrix. Consider an $n \times n$ matrix *A* and a positive integer $k \leq n$. From our work on combinations, we know there are *C*[*n*, *k*] *k*-th order principle submatrices of *A*. The total number of principle submatrices is therefore $2^n - 1$.

We have already seen that we can index matrix rows and columns with lists of indexes, so we can easily form all the principal submatrices. Here we create a function to do that work for us when given a matrix as an input. We use `Subsets` to create all the index subsets, and we use `Map` to successively use these subsets as row and column indexes for our matrix. We take advantage of the documented order in which `Subsets` produces its subsets, so we need not worry about index sequencing, and we use `Rest` to drop the empty set of indexes.

```
principleSubmatrices[m_?MatrixQ] := With[{n = Min@Dimensions[m]},
   m⟦#, #⟧ /@ Rest@Subsets[Range[n]]
  ];
```

```
mA = Array[a_## &, {3, 3}];
Column[MatrixForm /@ principleSubmatrices[mA]]
```

```
Part: The expression #1 cannot be used as a part specification.
```

$\{\{a_{1,1}, a_{1,2}, a_{1,3}\}, \{a_{2,1}, a_{2,2}, a_{2,3}\}, \{a_{3,1}, a_{3,2}, a_{3,3}\}\}⟦\#1, \#1⟧[\{1\}]$
$\{\{a_{1,1}, a_{1,2}, a_{1,3}\}, \{a_{2,1}, a_{2,2}, a_{2,3}\}, \{a_{3,1}, a_{3,2}, a_{3,3}\}\}⟦\#1, \#1⟧[\{2\}]$
$\{\{a_{1,1}, a_{1,2}, a_{1,3}\}, \{a_{2,1}, a_{2,2}, a_{2,3}\}, \{a_{3,1}, a_{3,2}, a_{3,3}\}\}⟦\#1, \#1⟧[\{3\}]$
$\{\{a_{1,1}, a_{1,2}, a_{1,3}\}, \{a_{2,1}, a_{2,2}, a_{2,3}\}, \{a_{3,1}, a_{3,2}, a_{3,3}\}\}⟦\#1, \#1⟧[\{1, 2\}]$
$\{\{a_{1,1}, a_{1,2}, a_{1,3}\}, \{a_{2,1}, a_{2,2}, a_{2,3}\}, \{a_{3,1}, a_{3,2}, a_{3,3}\}\}⟦\#1, \#1⟧[\{1, 3\}]$
$\{\{a_{1,1}, a_{1,2}, a_{1,3}\}, \{a_{2,1}, a_{2,2}, a_{2,3}\}, \{a_{3,1}, a_{3,2}, a_{3,3}\}\}⟦\#1, \#1⟧[\{2, 3\}]$
$\{\{a_{1,1}, a_{1,2}, a_{1,3}\}, \{a_{2,1}, a_{2,2}, a_{2,3}\}, \{a_{3,1}, a_{3,2}, a_{3,3}\}\}⟦\#1, \#1⟧[\{1, 2, 3\}]$

The *k*-th order leading principal submatrix includes the elements that are in the first *k* rows and columns of our matrix. Since we can construct submatrices by indexing with spans, we can easily construct all the leading principle submatrices of any matrix. For example, assuming our matrix is square we can use the following `kthLeading` function.

```
kthLeading = {m, k} ↦ m⟦1 ;; k, 1 ;; k⟧;
```

Exercise: create a function named `allLeading` that uses our `kthLeading` function to produce all the leading principle submatrices of any square matrix argument. Check that you can produce the following leading principal submatrices of a 5×5 matrix (or larger).