HANDS-ON START TO WOLFRAM *MATHEMATICA*®

and Programming with the Wolfram Language™

SECOND EDITION

Cliff Hastings   Kelvin Mischo   Michael Morrison

WolframMedia
Champaign

Table of Contents

INTRODUCTION
How to Use This Book

Create Examples while Reading

This book is meant to be an active companion during the process of learning how to use Mathematica®. The main body of the text will certainly provide insights into how Mathematica works, but the examples should be retyped as a starting point for individual exploration. Each chapter contains discussion, tips and a description of Mathematica functionality, along with actual examples that serve as starting points. Each chapter ends with additional exercises to emphasize comprehension, which can be used as an assignment to students or simply to work through on your own.

No matter what format this book is viewed in, it is recommended that readers have Mathematica on the desktop or Mathematica Online™ immediately accessible to type the examples and work through the exercises. It is recommended that as readers work through the book, they save a new file for each chapter in Wolfram Notebook™ format (.nb), either locally or in the Wolfram Cloud™, for future reference.

Any text in this type of styled box is meant to be a tip by the authors. The advice is meant to pass along experience gained from teaching thousands of people how to use Mathematica.

Part I: The Complete Overview Is Required Reading

All new Mathematica users should work through chapters one through nine first to obtain the necessary basis of knowledge for the rest of the book. These chapters will be of value to intermediate Mathematica users by filling in gaps in knowledge that can result from using Mathematica only for a narrowly defined set of tasks, or by broadening the horizons of users who may have learned Mathematica from an older version.

Part II: Extending Knowledge Is Suggested Reading

Once **Part I: The Complete Overview** is finished, the rest of the chapters can be read in order as a complete book or can be read in the order that most appeals to the reader.

Chapter 1: The Very Basics is designed to give you experience with typing commands in Mathematica. Knowing what commands to use and when to use those commands will be discussed in subsequent chapters; the purpose of the first chapter is simply to provide initial practice and immersion in using Mathematica.

Chapter 2: A Sample Project in Mathematica is meant to show the scope of Mathematica and how it can be applied to quickly explore a real-world problem. The goal of the chapter is not to understand the details of the commands but rather the thought process for building upon each step, and the process of working toward an interesting final result. Subsequent chapters will explain the individual commands in much more detail. They will provide the necessary building blocks of knowledge to create similar analyses while using Mathematica fluidly and fluently.

Mathematica on the Desktop and Mathematica Online

This book is about Mathematica and is primarily written from the perspective of using Mathematica on a desktop computer. A different product, Mathematica Online, provides a way to use Mathematica via a web browser, and it can also be used to follow along with the book's examples. This book is written from the perspective of using Mathematica on the desktop, so there may be times when the process for doing something, like navigating a menu, may not be exactly the same as the process in Mathematica Online. For cases where there are dramatic differences between the desktop version of Mathematica and Mathematica Online—such as working with slide shows, stylesheets, palettes and parallel computing—the text contains notes to make the reader aware of these differences. For the vast majority of examples, however, there will be no difference in entering the commands in Mathematica on the desktop or Mathematica Online.

Getting Access to Mathematica

If you do not currently have access to Mathematica, you can request a trial license from the Wolfram website at www.wolfram.com/mathematica/trial and use that to work along with the book.

Part I

The Complete Overview

CHAPTER 1

The Very Basics

First Interactions with Mathematica

Although Mathematica has functionality that spans many specialized areas, knowledge of the entire software package is not required to get started. Often it is the simple things in Mathematica that are the most impressive, especially when users are starting out.

Subsequent chapters in this book will explain why commands in Mathematica produce certain output and will also explain the scope of the system. This chapter is designed to be used as practice, since by typing commands into Mathematica, one can become acquainted with the workflow. Many people learn by doing, and this is the precise spirit of this chapter, which will provide some repetition to make the other chapters in the book more meaningful.

A common theme in this book is that Mathematica uses the Wolfram Language, and Wolfram Language commands all follow the same rules. A certain intuition develops for these rules, making it easy to apply commands to new situations. This chapter will help explain some commands and give some brief general descriptions to aid in developing this intuition.

Launch Mathematica and create a new notebook by clicking the **File** menu, choosing **New** and then selecting **Notebook**. A blank document with a horizontal cursor will appear. This is called a notebook. The horizontal cursor means that Mathematica is ready to be given a command. Type 10! to enter a calculation. When finished, evaluate the command by pressing Shift+Enter. Alternatively, inputs can be evaluated by pressing the Enter key on the numeric keypad if the keyboard being used has such a keypad. Mathematica will take the input, perform the designated operation and return the result of 3 628 800. Once a command has finished, the mouse or arrow keys can be used to place the cursor below the result, at which time Mathematica is ready to receive a new command. With these brief instructions, recreate the following examples.

If you get stuck during this section, it might be easier to watch a video of someone else typing commands into Mathematica and to mirror those actions. You can visit the Wolfram website (wolfr.am/hostm) to watch the Hands-on Start to Mathematica video series, which is a subset of the content of this book. In fact, this book was written in response to requests from people who watched the videos and wanted a more thorough introduction.

Type the following command to divide 717 by 3.

```
717/3
```

```
239
```

Find the exact answer to 718 divided by 3.

```
718/3
```

$$\frac{718}{3}$$

Find the approximate answer to 718 divided by 3.

```
N[718/3]
```

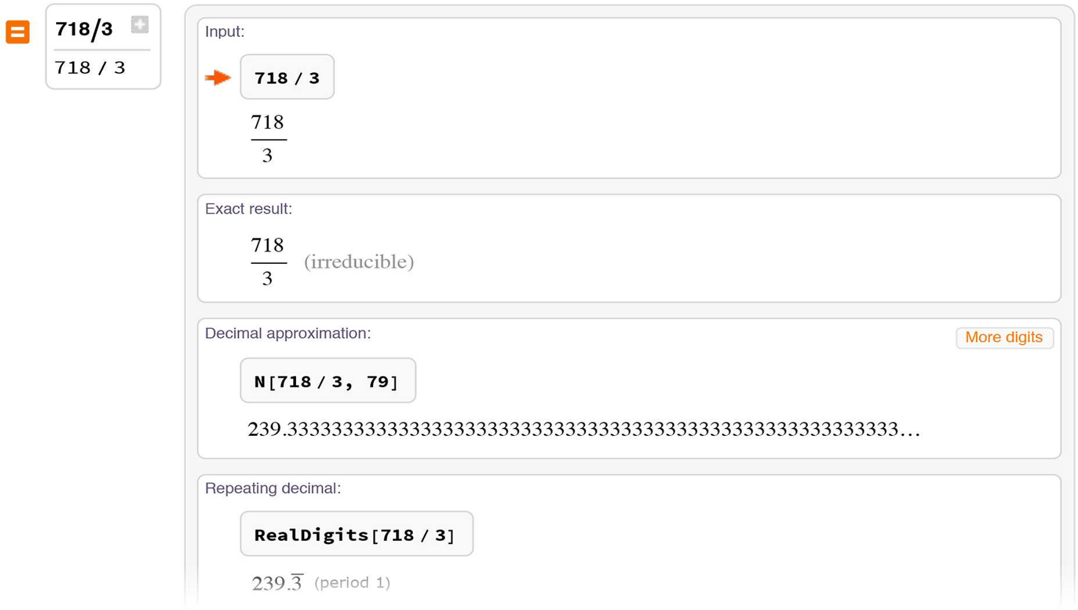
```
239.333
```

Find the approximate answer to 718 divided by 3 rounded to 5 digits.

```
N[718/3, 5]
```

```
239.33
```

Use free-form input to calculate 718 divided by 3. Free-form input is invoked by pressing the = key, and then the rest of the command can be typed and evaluated. See related calculations by clicking the plus icon after evaluating.



718/3

718 / 3

Input:

718 / 3

$\frac{718}{3}$

Exact result:

$\frac{718}{3}$ (irreducible)

Decimal approximation:

N[718 / 3, 79]

239.33333333333333333333333333333333333333333333333333333333…

Repeating decimal:

RealDigits[718 / 3]

$239.\overline{3}$ (period 1)

Assign the value 5 to a variable named **a**.

```
a = 5
```

```
5
```

Calculate $3\,a + 1$, where **a** is already defined as 5.

```
3 a + 1
```

```
16
```

Clear the variable definition of **a**, which will make **a** undefined.

```
Clear[a]
```

Expand the algebraic expression $(a + 5)\,(a + 9)$.

```
Expand[(a + 5) (a + 9)]
```

$$45 + 14\,\mathrm{a} + \mathrm{a}^2$$

Solve the equation $2\,x - 7 = 0$ for x.

```
Solve[2 x - 7 == 0, x]
```

$$\left\{\left\{\mathrm{x} \to \frac{7}{2}\right\}\right\}$$

Notice that two equal signs (**==**) were used in this command. The reason for that will be discussed in **Chapter 6: Fundamentals of the Wolfram Language**.

Solve the equation $2\,x - 7 = 0$ for x and find a numeric approximation of the result.

```
NSolve[2 x - 7 == 0, x]
```

```
{{x → 3.5}}
```

Use free-form input to solve the equation $2x - 7 = 0$ for x.

```
solve 2x−7=0
↳ Result
Reduce[−7 + 2*x == 0, x]
```

$$x == \frac{7}{2}$$

Solve the two equations with two unknowns, $2x - y = 0$ and $3x - 2y = 0$, for both x and y.

```
Solve[{2 x − 7 == 0, 3 x − 2 y == 0}, {x, y}]
```

$$\left\{\left\{x \to \frac{7}{2}, y \to \frac{21}{4}\right\}\right\}$$

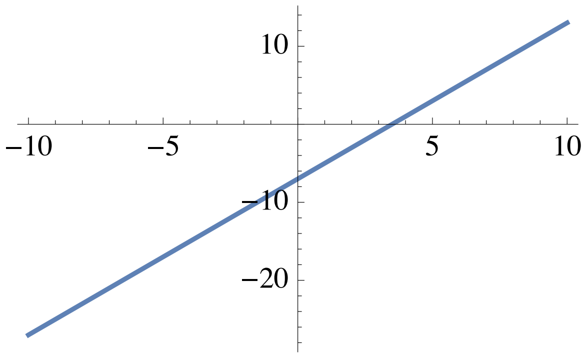
Solve the equation $a x^2 + b x + c = 0$ for x.

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

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

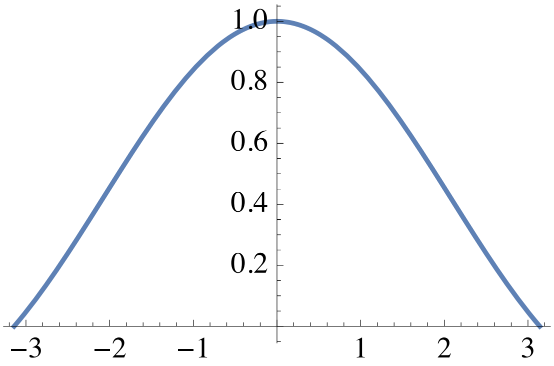
Plot the equation $y = 2x - 7$, where x goes from -10 to 10.

```
Plot[2 x − 7, {x, −10, 10}]
```

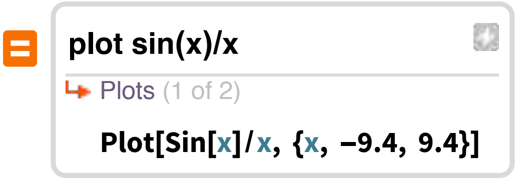


Plot sin(x) / x, where x goes from $-\pi$ to π.

```
Plot[Sin[x]/x, {x, -Pi, Pi}]
```



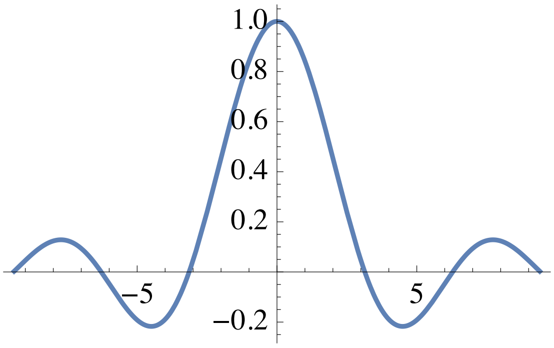
Use free-form input to plot sin(x) / x.



plot sin(x)/x

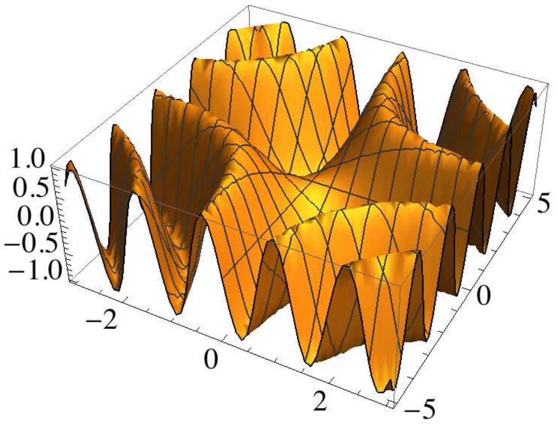
Plots (1 of 2)

```
Plot[Sin[x]/x, {x, -9.4, 9.4}]
```

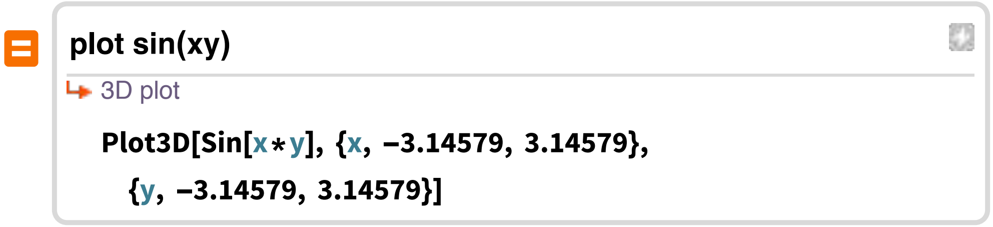


Plot sin(x y) in three dimensions, where x goes from -3 to 3 and y goes from -5 to 5.

```
Plot3D[Sin[x*y], {x, -3, 3}, {y, -5, 5}]
```



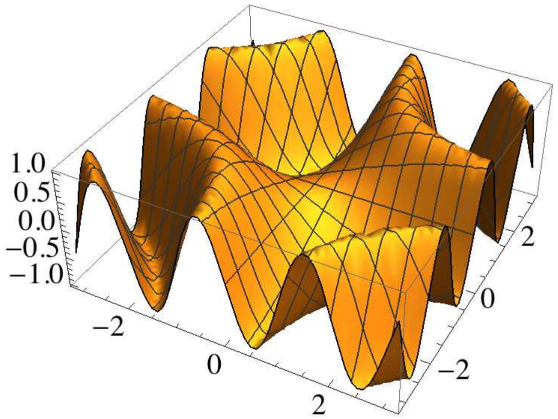
Use free-form input to plot sin(*x y*) in three dimensions.



plot sin(xy)

↳ 3D plot

```
Plot3D[Sin[x*y], {x, −3.14579, 3.14579},
  {y, −3.14579, 3.14579}]
```



Create a table of values for i^2, where *i* goes from 1 to 5.

```
Table[i^2, {i, 1, 5}]
```

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

Table is used to create a list of values, and it is one of the most used Wolfram Language commands. You will see plenty of examples of **Table** in this book, especially in the chapters on working with data.

The % symbol is used to represent the last calculation or the last output cell. Note that this is the last calculation that was evaluated, which may or may not be the calculation directly above the new calculation. Find the total of the values in the list above by using % to refer to the previous result.

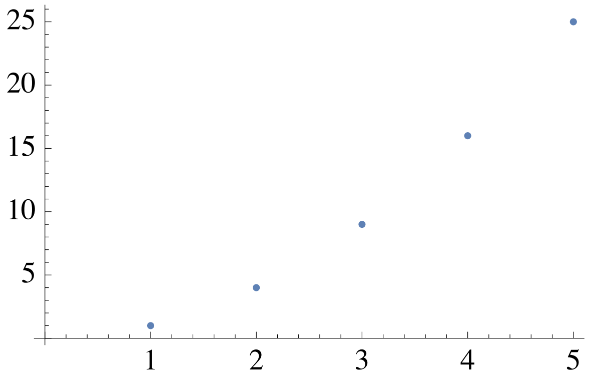
```
Total[%]
```

```
55
```

> If you are a Microsoft Excel user, you might have been thinking that **Sum** would be the name of the command to add up a list of numbers. There is a **Sum** command in the Wolfram Language, but it is used for mathematical summation and requires some parameters to be given for the index. If you already have a list of values, like in the previous example, then **Total** is the command you use to add them up.

Visualize the table of values for i^2, where i goes from 1 to 5.

```
ListPlot[Table[i^2, {i, 1, 5}]]
```

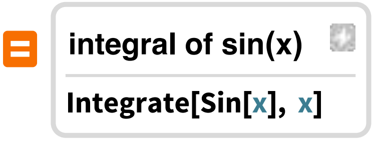


Calculate the indefinite integral of sin(x) with respect to x.

```
Integrate[Sin[x], x]
```

```
-Cos[x]
```

Use free-form input to calculate the indefinite integral of sin(x) with respect to x.



```
integral of sin(x)
Integrate[Sin[x], x]
```

```
-Cos[x]
```

Define a variable **mat1**, which is a list of three sublists, and end the statement with a semicolon to suppress the output.

```
mat1 = {{1, 2, 3}, {3, 5, 7}, {4, 6, 8}};
```

Calculate the determinant of that matrix.

```
Det[mat1]
```

```
0
```

Clear the variable definition of **mat1**, which will make **mat1** undefined.

```
Clear[mat1]
```

CHAPTER 2

A Sample Project in Mathematica

The Scope of Mathematica

Mathematica has been well-known for over 25 years as a computational system for research projects at the level of higher education or industry work. The roots of the company go back to research work in particle physics (one of Stephen Wolfram's many areas of interest), and Mathematica was designed to do mathematics that were impractical or impossible to do by hand. Mathematica continues to excel in this area.

But the core of Mathematica has always been broad: it does not favor one type of data over another, and it does not favor one type of calculation over another. Everything can be represented as a symbolic expression, which gives Mathematica great power and allows it to achieve results that are not possible in other systems. Mathematica is built upon the Wolfram Language, the same language that powers other Wolfram technologies like Wolfram Development Platform and Wolfram Data Science Platform.

It takes some time to learn Mathematica, and this text will illustrate how to get started and how to really start speaking the Wolfram Language. This language is worth learning because while it is convenient to use one of the many preexisting applications built into Mathematica, it is more convenient to be able to build your own, which is something that even non-programmers can do. And it is more convenient still to speak the Wolfram Language and to be able to apply it to any question or task that arises in work or everyday life.

What Part of the World Has the Highest Life Expectancy?

Consider the question, what part of the world has the highest life expectancy?

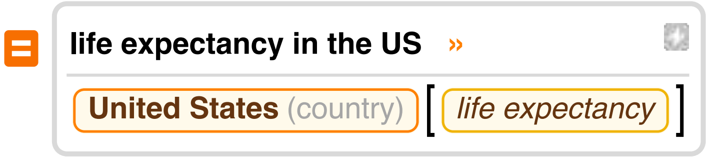
It is possible to search the internet for an answer to this question, and while there are certainly many websites that list this information, most of these websites do not provide underlying data, leaving the user unable to perform any further investigation. Even when some data can be located, it may only be a summary and not the full underlying dataset, making it difficult to explore why life expectancies are higher in some countries than in others.

Mathematica's integrated access to Wolfram Knowledgebase data, discussed in more detail later in the book, is a useful way to find and import a relevant dataset in a form that is ready for immediate computation.

Unlike **Chapter 1: The Very Basics**, which focused on typing and learning how to do a few things in Mathematica, this chapter will move very quickly and outline the complete thought process of how Mathematica can be used to explore an open-ended question. The explanation for the commands here is brief, but subsequent chapters will show how all the individual functions and concepts work together. This chapter serves as a representative example of the fluidity of exploration Mathematica provides, once users have a grasp on the basics.

Getting Started

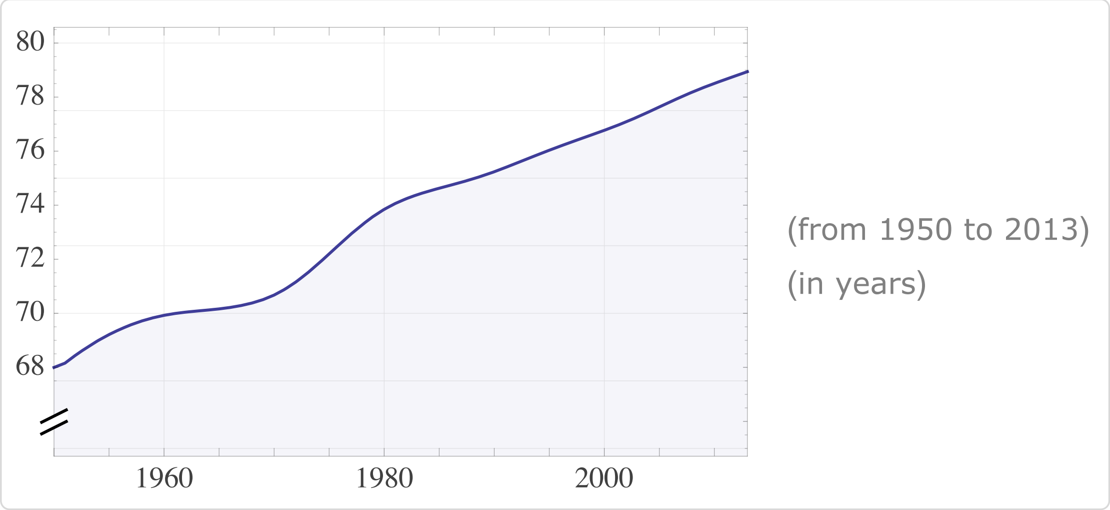
Mathematica's free-form input is an easy way to look up information for many different subject areas. Free-form input can be used to look at life expectancy for a particular country, such as the United States.



```
life expectancy in the US »
United States (country) [ life expectancy ]
```

```
78.941 yr
```

The results can be expanded by clicking the plus icon to show additional data that may be of interest, like a plot of life expectancy in the United States over time.



While looking at the expanded results, a command name was shown that can be evaluated to look up the life expectancy for the United States in a more precise way.

```
CountryData["UnitedStates", "LifeExpectancy"]
```

```
78.941 yr
```

> Free-form input is useful for looking up data or performing initial calculations because you can start using it immediately, without knowing anything about the underlying syntax or structure of the Wolfram Language. In general, users will start their Mathematica experience by using free-form input but will transition to using the Wolfram Language directly once a level of comfort is achieved.

Instead of querying each country one at a time, getting a complete list of life expectancies by country would be much more efficient. That list can be constructed by defining a new variable, **data**, which consists of pairs of country names and corresponding life expectancies. A command called **DeleteCases** is used to remove data for which one or more values was missing, and the **Short** command is used to only print a subset of the results to the screen.

```
data = DeleteCases[
     Table[{i, CountryData[i, "LifeExpectancy"]}, {i, CountryData[All]}], {_, _Missing}];
Short[data]
```

```
{{Afghanistan, 60.947 yr}, {Albania, 77.392 yr}, {Algeria, 71. yr},
 {American Samoa, 73.72 yr}, {Andorra, 82.51 yr}, {Angola, 51.899 yr},
 {Anguilla, 80.65 yr}, {Antigua and Barbuda, 75.954 yr},
 {Argentina, 76.305 yr}, {Armenia, 74.561 yr}, {Aruba, 75.455 yr},
 {Australia, 82.496 yr}, ≪207≫, {Uruguay, 77.23 yr},
 {Uzbekistan, 68.241 yr}, {Vanuatu, 71.626 yr}, {Venezuela, 74.633 yr},
 {Vietnam, 75.945 yr}, {Wallis and Futuna Islands, 78.2 yr},
 {West Bank, 74.54 yr}, {Western Sahara, 67.764 yr},
 {Yemen, 63.112 yr}, {Zambia, 58.105 yr}, {Zimbabwe, 59.871 yr}}
```

Since we used the **Short** command, Mathematica prints << 207 >> to indicate that there are 207 pieces of data that are not shown in the current list of results. Even if you do not explicitly use **Short**, there may be times when Mathematica will automatically decide to only print a subset of results if they are too long. In such instances, you will be provided with a dialog menu that allows you to see more results or the full list of results.

The output in this book is formatted as **StandardForm**, which is the default style for output generated in Mathematica. If the output in Mathematica does not match the output in this book, it is likely that Mathematica is configured to display results in **TraditionalForm** instead of **StandardForm**. Open the **Preferences** menu, choose **Evaluation** and make sure **StandardForm** is set as the format type of new output cells.

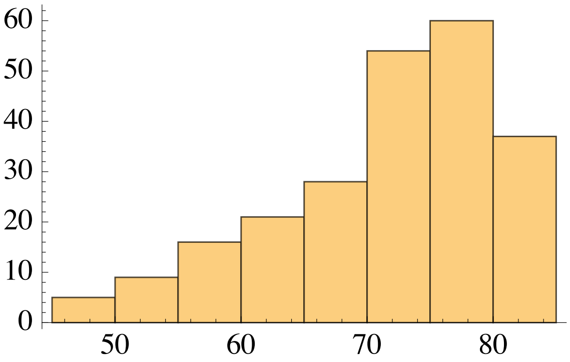
Having all the data in one place is useful, but filtering the data can be even more useful to explore this topic. For example, the **SortBy** function can be used to sort the list with respect to life expectancy from least to greatest.

```
Short[SortBy[data, Last]]
```

```
{{Sierra Leone, 45.561 yr},{Botswana, 47.572 yr},
  {Swaziland, 49. yr},{Lesotho, 49.446 yr},
  {Democratic Republic of the Congo, 49.963 yr},
  {Central African Republic, 50.179 yr},{Mozambique, 50.25 yr},
  {Chad, 51.182 yr},{Angola, 51.899 yr},{Nigeria, 52.506 yr},
  {Equatorial Guinea, 53.062 yr},{Burundi, 54.104 yr},
  ≪207≫,{Hong Kong, 81.86 yr},{San Marino, 81.97 yr},
  {Iceland, 82.086 yr},{Spain, 82.1 yr},{Singapore, 82.322 yr},
  {Italy, 82.385 yr},{Australia, 82.496 yr},{Andorra, 82.51 yr},
  {Switzerland, 82.604 yr},{Japan, 83.58 yr},{Macau, 84.36 yr}}
```

A histogram is also an effective way to look for patterns in the list of life expectancies. The ⟦ ⟧ symbols can be entered with the escape sequences Esc [[Esc for the left bracket and Esc]] Esc for the right bracket.

```
Histogram[data⟦All, 2⟧]
```



It is straightforward to look at larger trends as well. For example, with a few commands, one can look at the average life expectancy for a group of countries by repeating what was done previously but looking at groups of countries instead of all of the countries at once. Then, the average of each group's life expectancies can be calculated.

```
dataAfrica = DeleteCases[
    Table[{i, CountryData[i, "LifeExpectancy"]}, {i, CountryData["Africa"]}], {_, _Missing}];
Mean[dataAfrica⟦All, 2⟧]
```

```
60.6803 yr
```

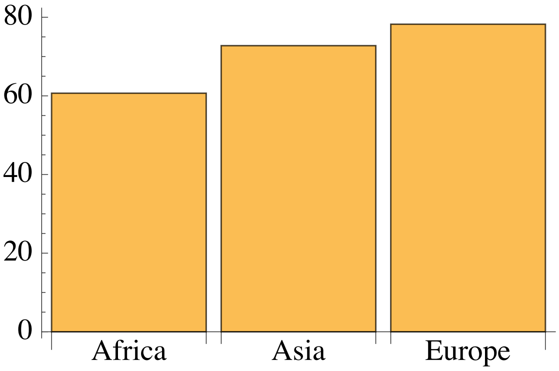
```
dataAsia = DeleteCases[
    Table[{i, CountryData[i, "LifeExpectancy"]}, {i, CountryData["Asia"]}], {_, _Missing}];
Mean[dataAsia⟦All, 2⟧]
```

```
72.7732 yr
```

```
dataEurope = DeleteCases[
    Table[{i, CountryData[i, "LifeExpectancy"]}, {i, CountryData["Europe"]}], {_, _Missing}];
Mean[dataEurope⟦All, 2⟧]
```

```
78.2237 yr
```

```
BarChart[{Mean[dataAfrica⟦All, 2⟧], Mean[dataAsia⟦All, 2⟧], Mean[dataEurope⟦All, 2⟧]},
  ChartLabels → {"Africa", "Asia", "Europe"}]
```

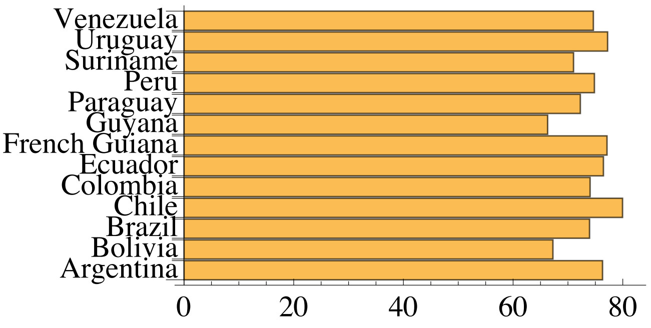


Choosing to view a breakdown by country for a particular continent is just as straightforward. The same approach as before is followed.

```
dataSouthAmerica = DeleteCases[
     Table[{i, CountryData[i, "LifeExpectancy"]}, {i, CountryData["SouthAmerica"]}], {_, _Missing}];
Take[dataSouthAmerica, 3]
```

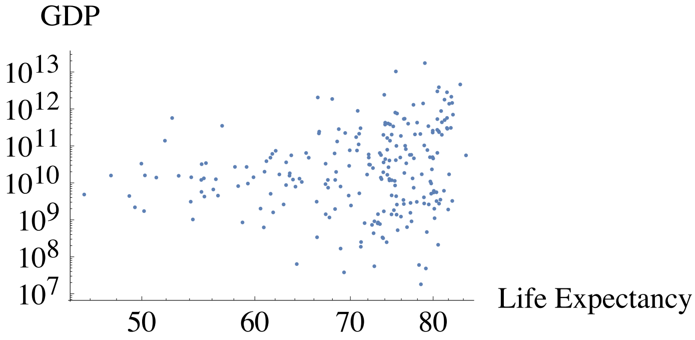
```
{{Argentina, 76.305 yr}, {Bolivia, 67.26 yr}, {Brazil, 73.937 yr}}
```

```
BarChart[dataSouthAmerica⟦All, 2⟧, ChartLabels → dataSouthAmerica⟦All, 1⟧,
  BarOrigin → Left]
```



Exploration in Mathematica often leads to other, more interesting questions. Instead of simply visualizing life expectancies as a simple measurement of magnitude, other data can be used to explore whether there is a relationship between life expectancy and some other property related to a country. For example, there might be a relationship between life expectancy and GDP that is just as easy to visualize as the life expectancy data was by itself.

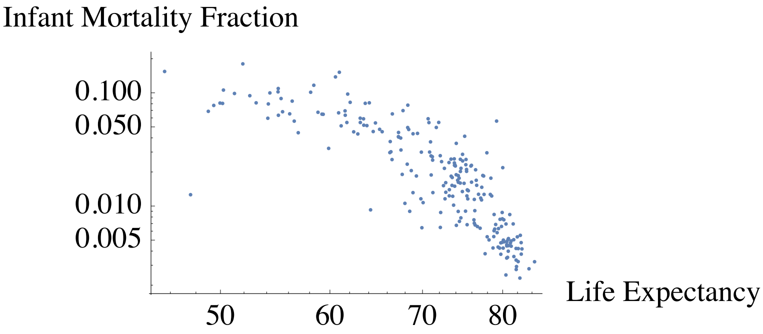
```
data = Table[Tooltip[{CountryData[i, "LifeExpectancy"], CountryData[i, "GDP"]},
      CountryData[i, "Name"]], {i, CountryData[]}];
ListLogLogPlot[data, AxesLabel → {"Life Expectancy", "GDP"}]
```



You may have noticed that we had already defined the variable **data** in the beginning of this section. When the preceding command is evaluated, the old definition of **data** is discarded and the new definition is assigned. For these examples, this redefinition is fine, but sometimes it is better practice to define unique variable names as they are needed, instead of recycling existing ones. Defining, redefining and clearing variables is covered in more detail in **Chapter 6: Fundamentals of the Wolfram Language**.

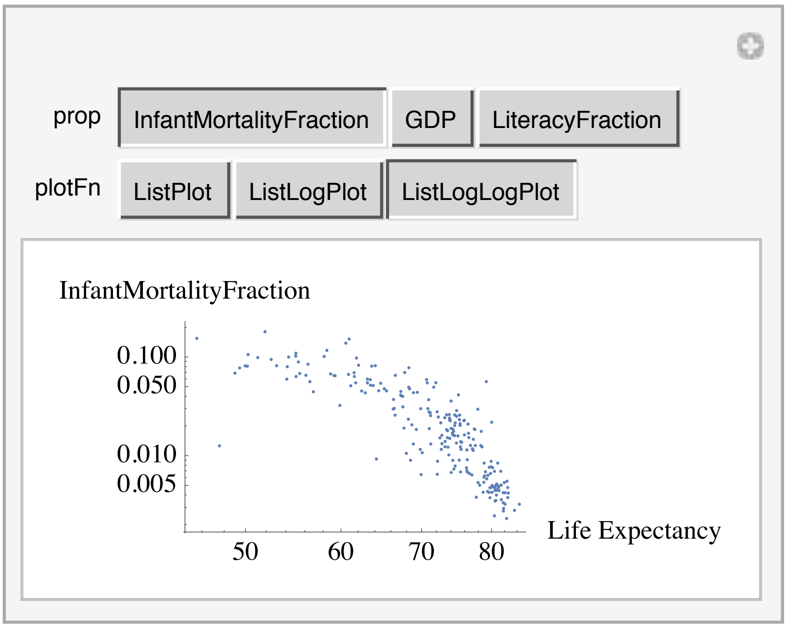
The data is clustered, but no clear pattern stands out. With the same series of commands, it is easy to explore a different hypothesis. Instead of looking at the relationship between GDP and life expectancy, perhaps the infant mortality fraction has a more direct relationship with life expectancy.

```
data =
   Table[
     Tooltip[{CountryData[i, "LifeExpectancy"], CountryData[i, "InfantMortalityFraction"]},
       CountryData[i, "Name"]], {i, CountryData[]}];
ListLogLogPlot[data, AxesLabel → {"Life Expectancy", "Infant Mortality Fraction"}]
```



This is looking more interesting! Swapping different properties into the series of commands above is simple enough, but luckily, there is an incredibly easy way to do this interactively in Mathematica by adding a single command into the mix. **Manipulate** can be used to create an interactive model from any expression by introducing some parameters; in turn, Mathematica will create an interactive model that shows the result of manipulating those parameters, giving users immediate, real-time feedback.

```
Manipulate[
  plotFn[Table[Tooltip[{CountryData[i, "LifeExpectancy"], CountryData[i, prop]},
       CountryData[i, "Name"]], {i, CountryData[All]}],
    AxesLabel → {"Life Expectancy", prop}],
  {prop, {"InfantMortalityFraction", "GDP", "LiteracyFraction"}},
  {{plotFn, ListLogLogPlot}, {ListPlot, ListLogPlot, ListLogLogPlot}},
  SaveDefinitions → True]
```



Only a few commands are needed to create a user interface that reads in data for all countries around the world, plots life expectancy against a user-chosen property and allows the user to toggle between three different types of plots for trend-spotting—not bad for a single expression!

This example demonstrates the power and flexibility that is the essence of Mathematica. It is easy to start with something simple, iteratively (and interactively) layer on more commands quickly and easily explore open-ended topics. Mathematica is unique in this regard, thanks to its principles of automation and integration, which means that things just work, providing users with more time for deeper analysis or to ask more probing questions.

The rest of the chapters in this section should be read in sequential order in order to achieve a foundational understanding of how to use Mathematica. After that, the remaining chapters can be read according to readers' specific interests.

A Bookkeeping Note

Each chapter clears the variable definitions used in the chapter. This is a good practice to avoid any variable conflicts when working in a new Mathematica notebook or session.

```
Clear[data, dataAfrica, dataAsia, dataEurope, dataSouthAmerica]
```

CHAPTER 3

Input and Output

Introduction

Mathematica is an interactive environment where a typical workflow is to enter commands, evaluate those commands to see the results and then build up to more complicated and sophisticated programs or blocks of code. This workflow encourages exploration, since immediate feedback can be obtained once a command or piece of code is executed, making it an excellent environment for both well-defined and exploratory work.

A basic understanding of how to enter input and receive output is important, but the core ideas are very simple and will be well understood by the end of the chapter. To make things even easier, Mathematica can accept input in several different forms, including giving commands in natural language, which makes it incredibly accessible for beginners to get started.

First Evaluations

Mathematica provides a variety of methods for interacting with the system. At the most basic level, commands are entered as input, and then `Shift`+`Enter` is pressed to execute the commands. By default, Mathematica expects to receive a command when typing into a brand-new document, and input will be entered wherever the cursor is located.

Commands can be given with traditional calculator syntax, such as the use of the caret symbol to represent exponentiation and the use of the forward slash to represent division.

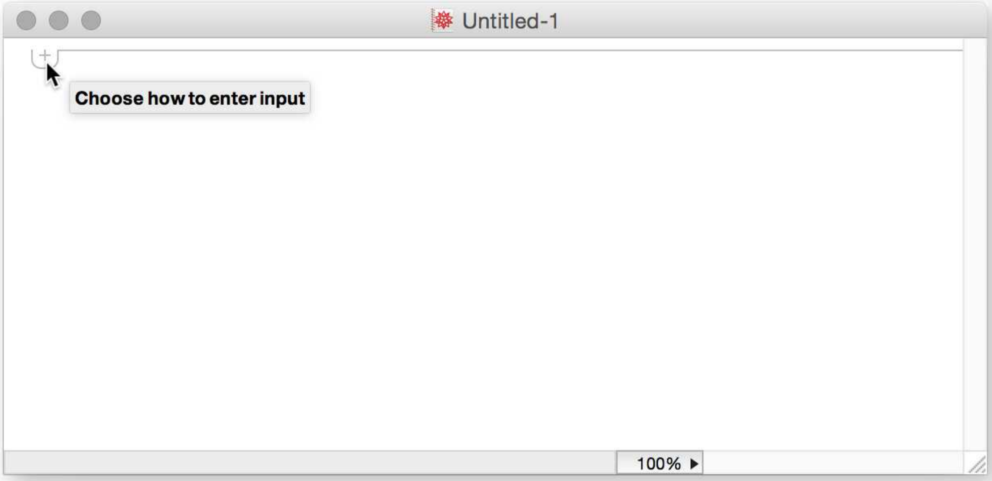
```
2^100
```

```
1 267 650 600 228 229 401 496 703 205 376
```

Input can also be entered in the form of typeset expressions. As an alternative to what was just given, the expression 2^{100} can be entered by typing the number 2 and then pressing `Ctrl`+`6` to create a little template box for an exponent, at which point the 100 can be entered and the expression can be evaluated using the same `Shift`+`Enter` key press.

When a command is evaluated, the input is preserved, and both the input and output are labeled and displayed unless suppressed by the user. The input and output are each displayed in a cell, and a Mathematica document, called a notebook, is comprised of cells.

Input can also be entered using the Cell Insertion Assistant, which is displayed as an icon when the cursor is between cells, or at the top or bottom of a notebook.



If **Wolfram Language input** is selected from the Cell Insertion Assistant popup menu, an input cell is created in expectation of a command being given, and the cursor is placed inside the new cell, waiting to receive input.

Both of these examples of entering input—typing to create a new cell and using the Cell Insertion Assistant to select Wolfram Language input—do exactly the same thing by creating an input cell to contain Wolfram Language commands. Using the Wolfram Language directly is the most common way to tell Mathematica to do something, but there are some alternate ways to enter input, and they will be discussed shortly.

Navigating around Notebooks

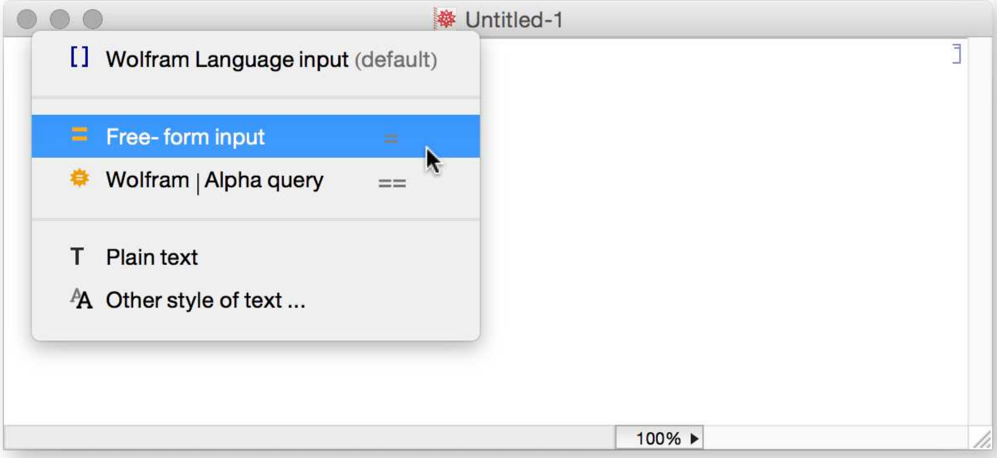
Before discussing other ways to enter input, it is important to be mindful of the position of the cursor. When the mouse pointer is between cells, it will display as a horizontal I-beam, and when clicked, the cursor will display as a horizontal bar with the Cell Insertion Assistant icon on the left. The cursor may also display as a horizontal bar when placed before the first cell of the notebook or after the last cell of the notebook.

When the mouse pointer is inside a cell, then it will display as a vertical I-beam, and clicking will insert the cursor into the cell at that location, allowing for input or editing. The cursor can also be moved within and between cells with the arrow keys on the keyboard.

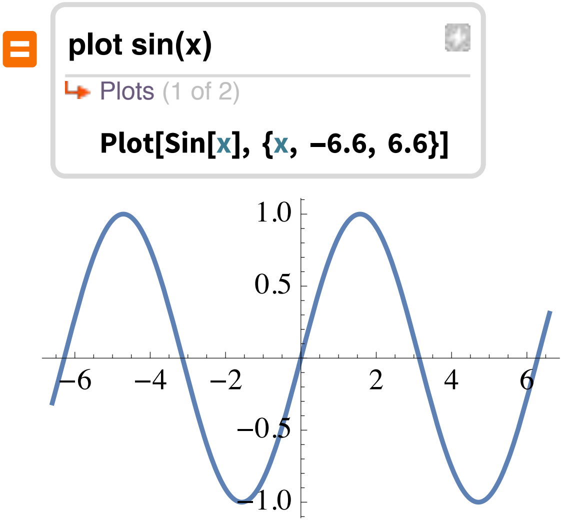
There is a global setting for the size of content displayed within notebooks. The bottom of a notebook window has a menu that displays the current magnification setting for a notebook, and clicking this number opens a popup window that can be used to display the content from 50% to 300% of its normal size.

Free-Form Input

Mathematica has a unique capability to accept free-form input, which means that commands can be entered using plain English. Free-form input can be entered by selecting **Free-form input** from the Cell Insertion Assistant popup menu. Or, if the cursor is displayed as a horizontal bar—meaning it is between cells, so typing will create a brand-new *input* cell in that location—then an equal sign can be typed to start a *free-form input* cell in that location instead of a Wolfram Language input cell.



When free-form input is used, Mathematica will display an orange icon to the left of the free-form input content to help differentiate between regular input cells and free-form input cells. Once a free-form command is entered, Shift+Enter is used to evaluate the command, the same way that a Wolfram Language command is evaluated. When free-form input is evaluated, a parser translates the natural language into Wolfram Language commands (and looks up data, if that is relevant). The result of performing those commands is returned, possibly along with some Wolfram Language syntax as well.

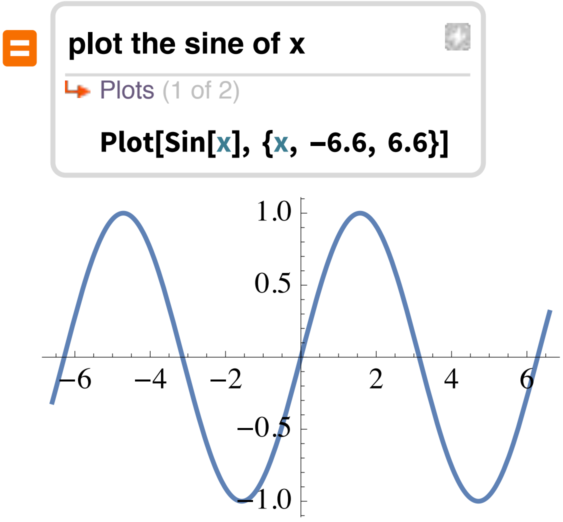


plot sin(x)

Plots (1 of 2)

```
Plot[Sin[x], {x, −6.6, 6.6}]
```

Free-form input requires an internet connection, since the natural-language parser resides on a server at Wolfram and not in a local installation of desktop Mathematica. Having the parser reside on a server means that new versions can be rolled out on a regular basis, giving users more and better parses over time. At the time of this writing, however, the parsing technology is already so effective and robust that free-form input can be given in many different ways to achieve the same result. The following examples illustrate this flexibility by asking for the plot of sin(*x*) in different ways.

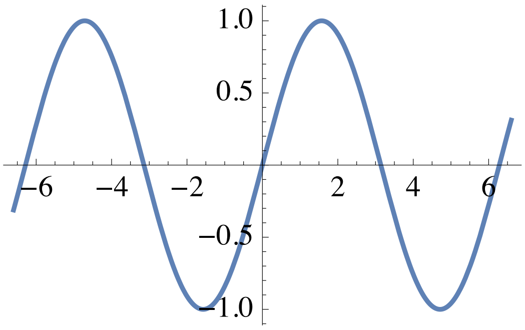


plot the sine of x

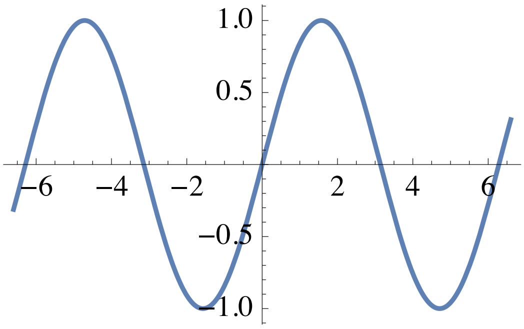
Plots (1 of 2)

```
Plot[Sin[x], {x, −6.6, 6.6}]
```

```
graph of sinx
↳ Plots (1 of 2)
   Plot[Sin[x], {x, -6.6, 6.6}]
```



```
picture of sine curve
↳ Plots (1 of 2)
   Plot[Sin[x], {x, -6.6, 6.6}]
```



Free-form input can be used to perform many operations, such as calculations.

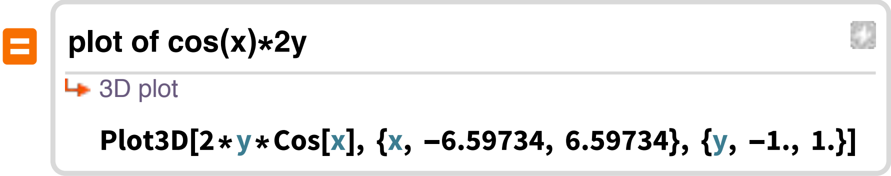
```
100 digits of pi
N[Pi, 100]
```

```
3.141592653589793238462643383279502884197169399375105820974944592307816406
  286208998628034825342117068
```

```
integral of 1/(x^3+1)
Integrate[1/(x^3 + 1), x]
```

$$\frac{\text{ArcTan}\left[\frac{-1+2\,x}{\sqrt{3}}\right]}{\sqrt{3}}+\frac{1}{3}\text{Log}[1+x]-\frac{1}{6}\text{Log}\left[1-x+x^2\right]$$

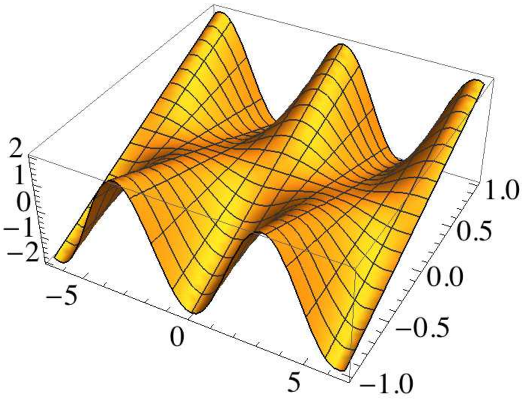
Free-form input can graph functions and visualize mathematical surfaces.

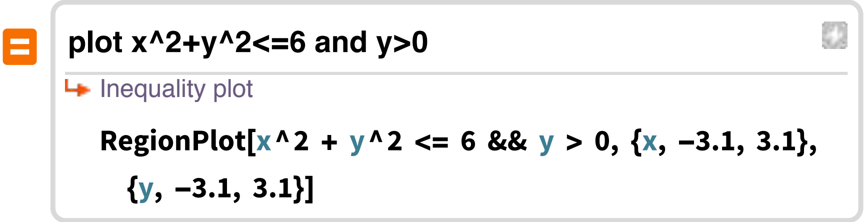


plot of cos(x)*2y

↳ 3D plot

```
Plot3D[2*y*Cos[x], {x, −6.59734, 6.59734}, {y, −1., 1.}]
```

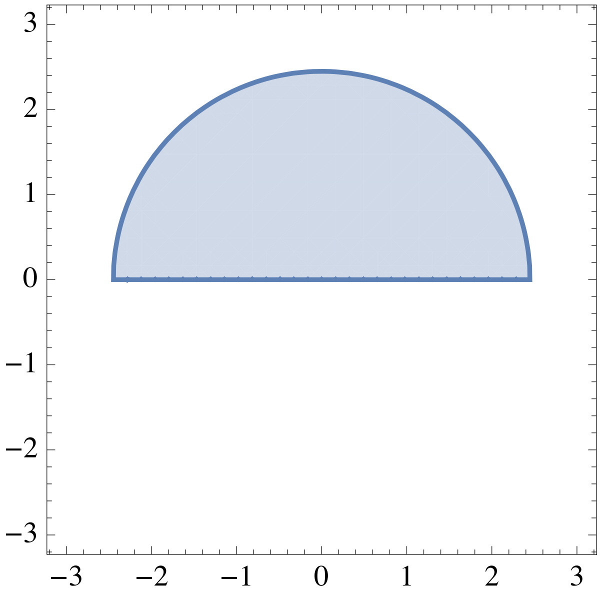




plot x^2+y^2<=6 and y>0

↳ Inequality plot

```
RegionPlot[x^2 + y^2 <= 6 && y > 0, {x, −3.1, 3.1},
  {y, −3.1, 3.1}]
```



Free-form input can be used to look up data for a variety of knowledge domains.

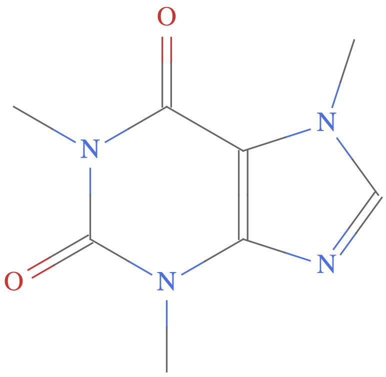
```
number of turkeys in turkey
↳ Result
```

```
2990  (2014 estimate)
```

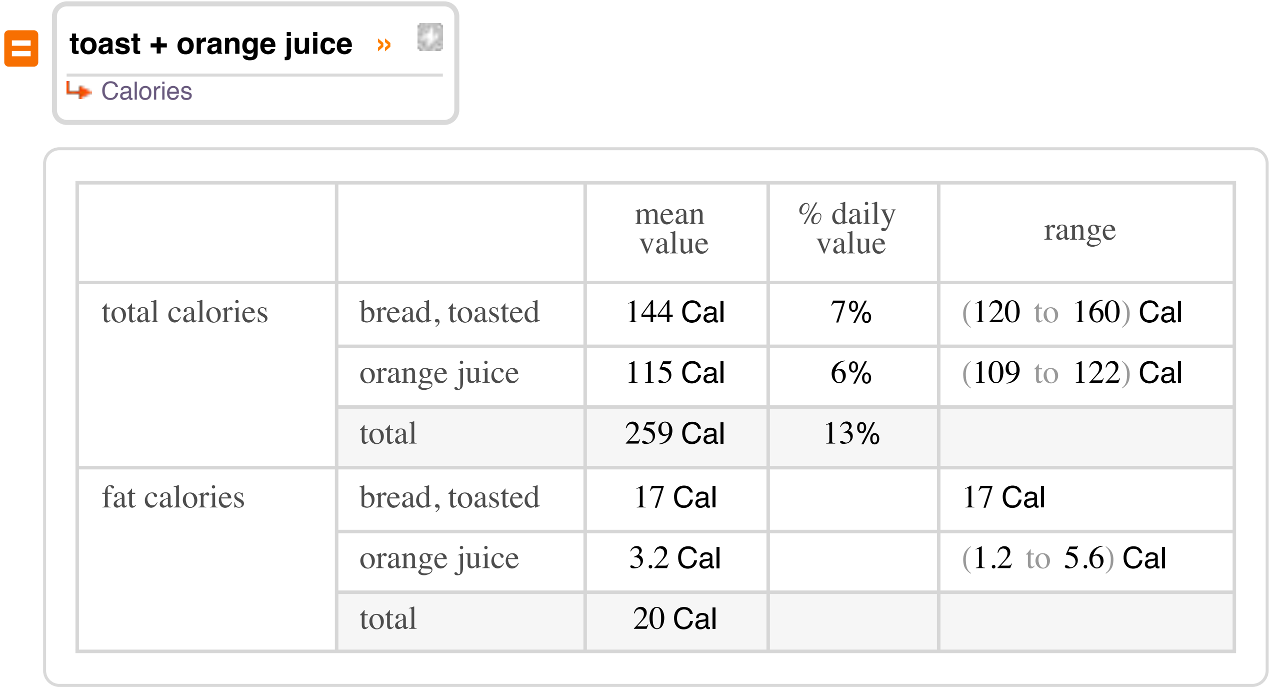
```
population of the United States
CountryData["UnitedStates", "Population"]
```

```
322 422 965 people
```

```
caffeine molecule
ChemicalData["Caffeine"]
```



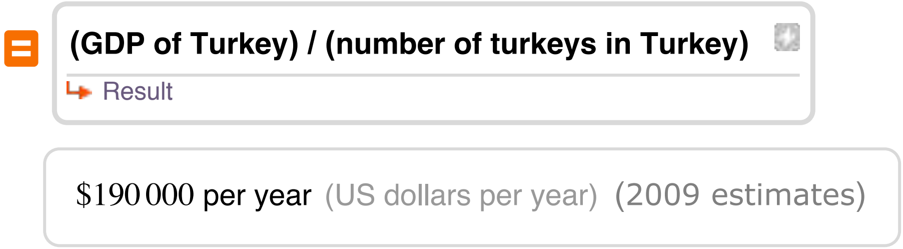
Free-form input can even look up data and then perform computations with the results.



toast + orange juice »

Calories

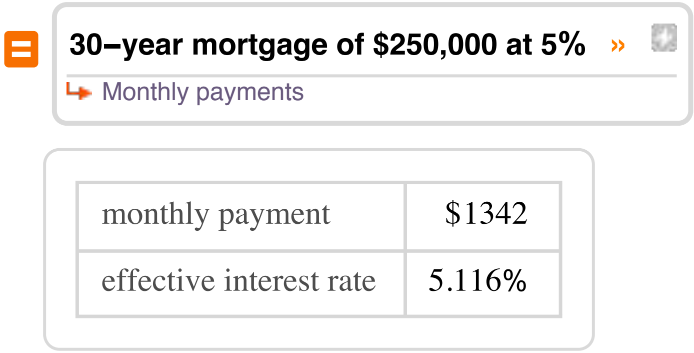
		mean value	% daily value	range
total calories	bread, toasted	144 Cal	7%	(120 to 160) Cal
	orange juice	115 Cal	6%	(109 to 122) Cal
	total	259 Cal	13%	
fat calories	bread, toasted	17 Cal		17 Cal
	orange juice	3.2 Cal		(1.2 to 5.6) Cal
	total	20 Cal		



(GDP of Turkey) / (number of turkeys in Turkey)

Result

$190 000 per year (US dollars per year) (2009 estimates)



30–year mortgage of $250,000 at 5% »

Monthly payments

monthly payment	$1342
effective interest rate	5.116%

Free-form input may return many results, although only a single result may be shown by default. Click the "Show all results" icon, which looks like a gray plus sign at the top right of a free-form input pod, to see additional calculations and results that may be of interest.

Point-and-Click Palettes

For those who like visual menus, Mathematica contains a variety of palettes to make entering input easier. Available palettes can be found under the **Palettes** menu. For example, the **Basic Math Assistant** palette facilitates entry of mathematical typesetting, as well as commands related to algebra, calculus, linear algebra and visualization.

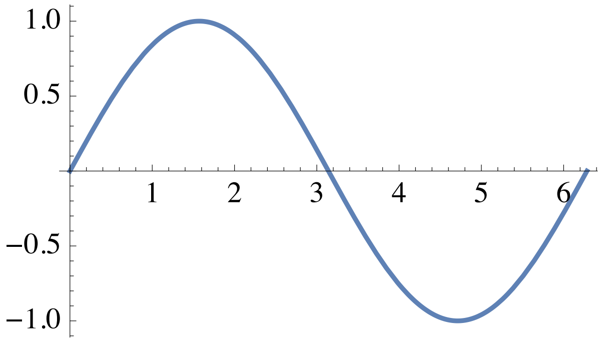
> You can use the **CreatePalette** command to construct your own custom palette, which is handy if you find yourself doing the same operations and typesetting constructions over and over again.

Many of the buttons on the **Basic Math Assistant** palette, as well as the other Assistant palettes available from the **Palettes** menu, provide command templates when they are clicked. For example, navigating to the **2D** tab of the **Basic Commands** section and clicking the **Plot** button yields the following.

```
Plot[function, {var, min, max}]
```

Such a template provides the appropriate syntax for the command name and only requires the user to enter the remaining arguments before evaluating the command. The arguments can be entered with the keyboard (and Tab can be used to jump between the placeholders) or by clicking buttons in the palette.

```
Plot[Sin[x], {x, 0, 2 π}]
```



Using palettes can streamline the process of entering input for a new user, since command templates can be filled in quickly and intuitively. Palettes also provide an excellent interface when working with technologies that favor a mouse-driven experience, such as interactive whiteboards.

Entering Wolfram Language Commands Directly

While free-form input and palettes provide easy ways to enter commands, most users prefer to leverage Mathematica's power by using the Wolfram Language directly. This is actually very easy to do, thanks to a very consistent language design that really only requires you to remember three main rules.

1. Wolfram Language commands begin with capital letters.
2. Function arguments are enclosed by square brackets: [ ].
3. Lists, which are also used to store domains and ranges, are enclosed by curly braces: { }.

In regard to the first rule, all Wolfram Language commands begin with capital letters, and if a command name is comprised of multiple words, like **ListPlot**, then the first letter of each word is capitalized. There is no space between the words in the command name. In addition, commands are generally written as full English words, although some exceptions are made to fit with the conventional naming of certain mathematical functions. A few commands like **N** are abbreviated for the sake of simplicity.

Regarding the second and third rules, it is important to realize the special meaning that square brackets and curly braces have within the Wolfram Language. Since these symbols are used for denoting function arguments and lists, respectively, they cannot be used for other purposes, such as for grouping mathematical expressions. Instead, mathematical expressions that require multiple levels of grouping can use sets of nested parentheses.

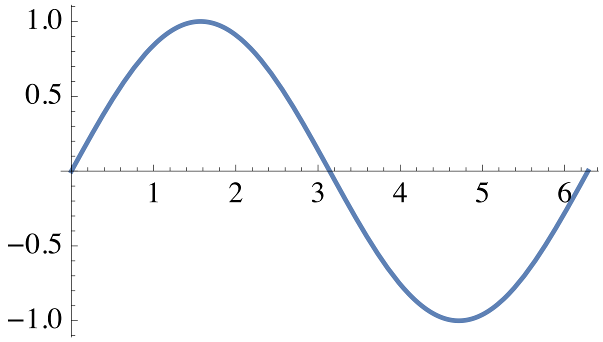
Since all Wolfram Language functions are capitalized, a good practice to use when defining your own functions is to start the function names with lowercase letters. That way, it is very easy to distinguish between what is a built-in Wolfram Language command and what is a function that you have defined yourself.

These rules make it very easy to identify Wolfram Language commands when looking at a notebook or reading through code, even if the command name itself is unfamiliar. For example, in the following input, it is obvious that **Expand** is a Wolfram Language command, since it is capitalized and has a pair of square brackets enclosing an argument. (An argument is just terminology to describe what is going to be operated on by the command.)

```
Expand[(a + b)^10]
```

Oftentimes, a single input cell will contain multiple Wolfram Language commands, but these same rules can be used to identify separate commands and their respective arguments.

```
Plot[Sin[x], {x, 0, 2 π}]
```

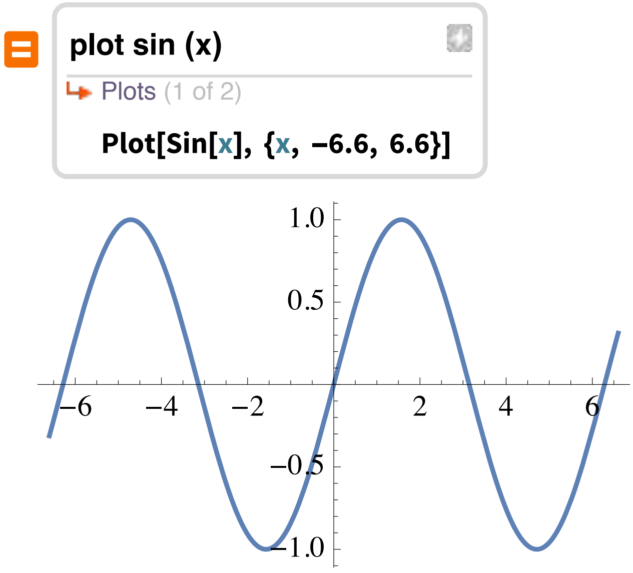


Here it can be seen that two commands are being used: **Plot** and **Sin**. The **Sin** command has a single argument, which is **x**, and its form **Sin[x]** illustrates the rules about command names starting with capital letters and square brackets surrounding function arguments. The **Plot** command has two arguments: **Sin[x]** and **{x,0,2π}**, the second of which illustrates the rule that lists are enclosed with curly braces.

In the preceding example, you can figure out where the **Sin** command "ends" by triple-clicking the command name, which will highlight the entire command, including its opening and closing brackets and all of its arguments inside the brackets. This is really useful if you are working with a block of code and need to quickly see where one command ends and the next one begins.

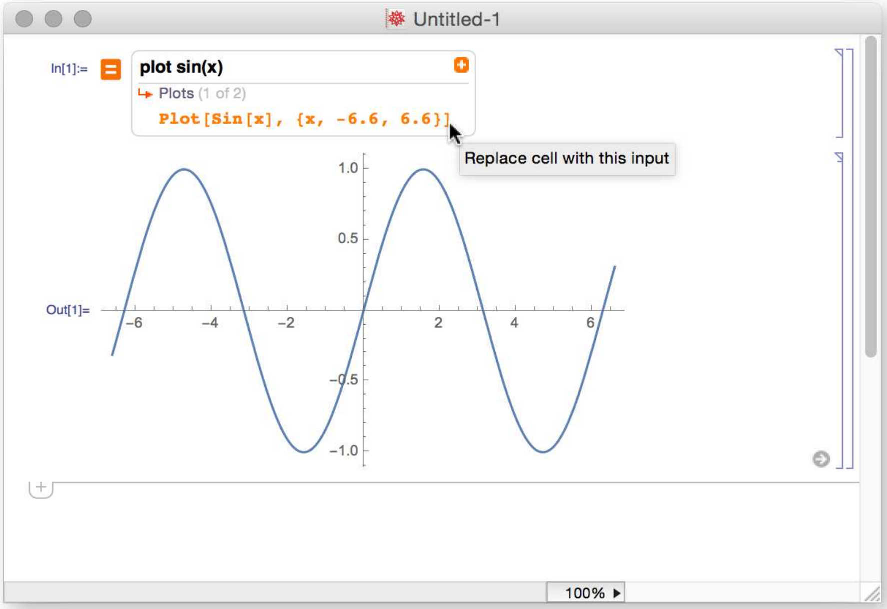
Transitioning from Free-Form Input to Formal Syntax

Using free-form input is a natural choice for new users, since it does not require any knowledge of syntax. Direct use of the Wolfram Language, however, provides a more powerful platform for performing computations and developing ideas, and it is the method that most users gravitate to over time. Free-form input provides some facility to help accelerate this transition by providing formal Wolfram Language syntax, when available, for free-form input that was entered.



```
plot sin (x)
Plots (1 of 2)
Plot[Sin[x], {x, −6.6, 6.6}]
```

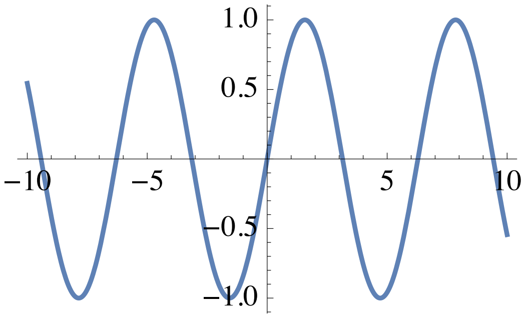
When evaluated, the free-form input cell contains two pieces of input: the original free-form command and the Wolfram Language equivalent of the command. When the pointer is hovered over the Wolfram Language syntax, a popup window indicates that the free-form input cell can be replaced with the formal syntax by clicking.



```
In[1]:= plot sin(x)
        Plots (1 of 2)
        Plot[Sin[x], {x, -6.6, 6.6}]
Out[1]=
```

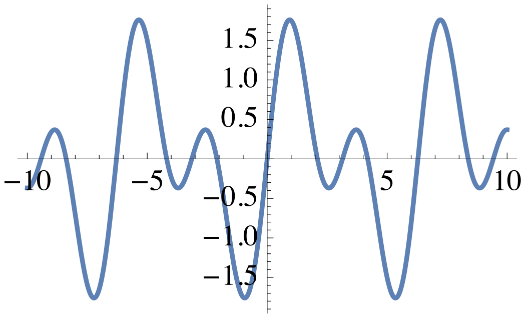
Replacing free-form input with Wolfram Language syntax allows the command to be edited and modified, and since Wolfram Language command names are so readable, it is usually very obvious how the command can be modified to provide a different result. For example, changing the values of – 6.6 and 6.6 allows the sine function to be plotted along a different domain.

```
Plot[Sin[x], {x, −10, 10}]
```



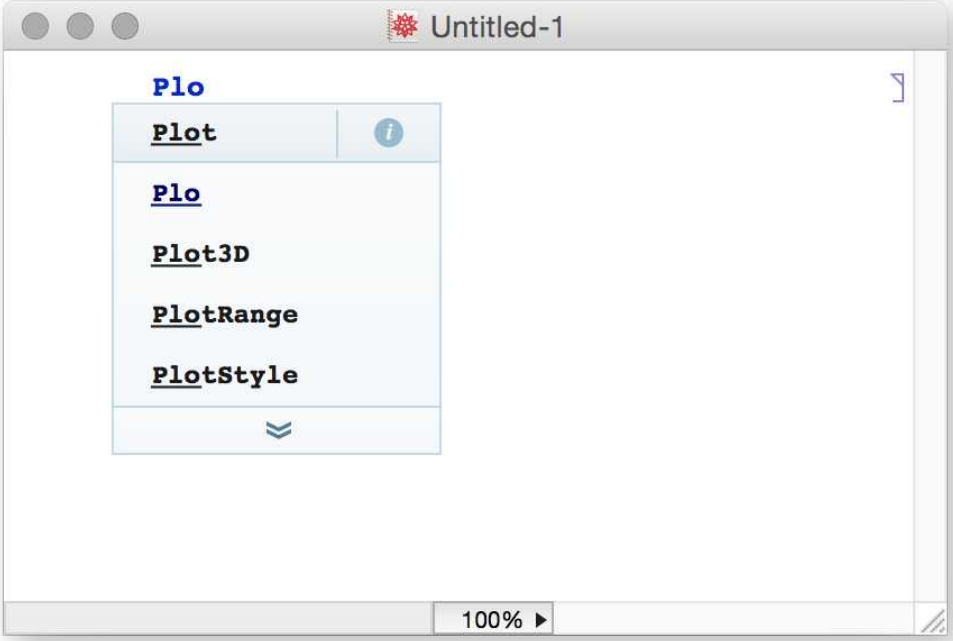
Other function arguments can be altered as well, like the mathematical expression that is being plotted.

```
Plot[Sin[x] + Sin[2 x], {x, −10, 10}]
```



Using Autocompletion and Command Templates

Autocompletion assistance is available when entering Wolfram Language commands directly. As commands are typed into input cells, popup menus list possible matches for the command name being typed. A command name can be selected from the list of suggestions by pressing Enter or Tab or by clicking the mouse. For example, by typing the letters **Plo**, the **Plot** command can be selected from the list.



Once a command is selected, the double-chevrons icon can be clicked to select a template for the command, or the information icon can be clicked to jump to the relevant documentation for the command. Repeating the previous example of typing the letters **Plo**, selecting the **Plot** command and then choosing the first template yields the following.

Plot[f, {x, x_{min}, x_{max}}]

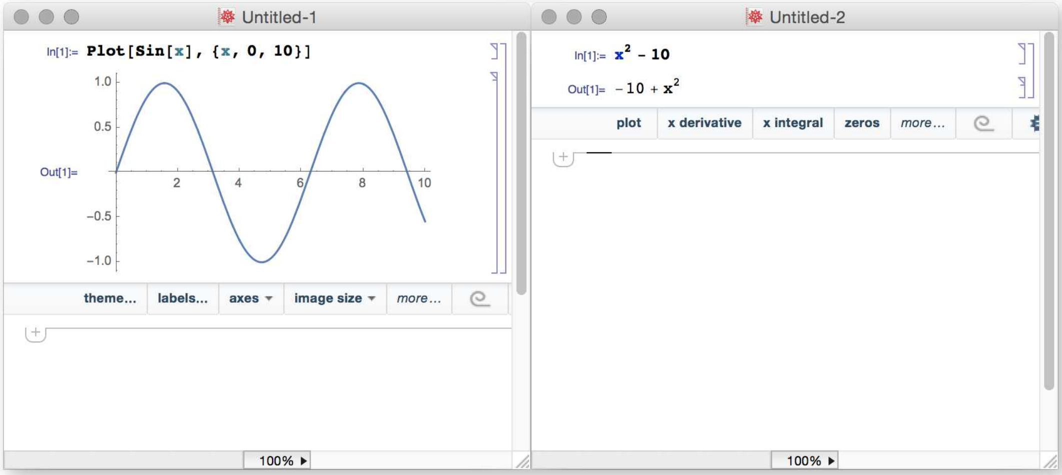
Templates are a quick and convenient way to recall the syntax for a command without leaving the current cell and interrupting the task at hand, and since many Wolfram Language commands are generally very readable in terms of what they do, getting a template is sometimes all that is needed to continue working without interruption.

For those who like keyboard shortcuts: once a command name, like **Plot**, is typed into an input cell, then the Ctrl+Shift+K or Cmd+Shift+K keyboard shortcut can be used to bring up the template list.

Automatic Suggestions for Next Calculations

Mathematica also provides the Suggestions Bar to suggest useful calculations based on the last output. The Suggestions Bar appears when the cursor is in an output cell or after an output cell, and it disappears when the cursor is not near output.

When displayed, the Suggestions Bar will show a variety of commands that can be taken as a next step. These suggestions are context aware, so output from a **Plot** command may have suggestions for changing the style and visual appearance of the plot, while output that is mathematical in nature may have suggestions for taking derivatives, integrals or finding zeros.



```
In[1]:= Plot[Sin[x], {x, 0, 10}]
Out[1]=
```

```
In[1]:= x^2 - 10
Out[1]= -10 + x^2
```

Any of these suggestions can be applied by simply clicking. For example, choosing to find the zeros by clicking that suggestion will create a new input cell that contains the precise Wolfram Language command (**Solve**) to perform the operation, and this input cell is also automatically evaluated to show its corresponding result.

```
x^2 – 10
```

$-10 + x^2$

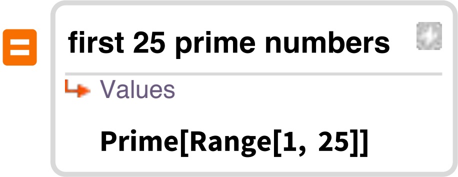
```
Solve[−10 + x^2 == 0, x]
```

$\{\{x \to -\sqrt{10}\}, \{x \to \sqrt{10}\}\}$

The Suggestions Bar does not lock users into a single path for calculations, because it is possible to return to a previous output and choose a different operation from the Suggestions Bar's list of choices. If this approach is followed, previous results will not be overwritten, but new input and output cell pairs will be created.

> The Suggestions Bar is kind of like free-form input: If you do not know exactly what you want to do, then it generally can do a pretty good job of helping you along. One main difference is that it can be very easy to build up a series of compound calculations by using the Suggestions Bar, while free-form input excels more in situations where a single result is returned.

The Suggestions Bar can also be used to combine a series of Wolfram Language commands into a single compound statement, essentially creating a short program by simply pointing and clicking. For example, use free-form input to find the first 25 prime numbers.



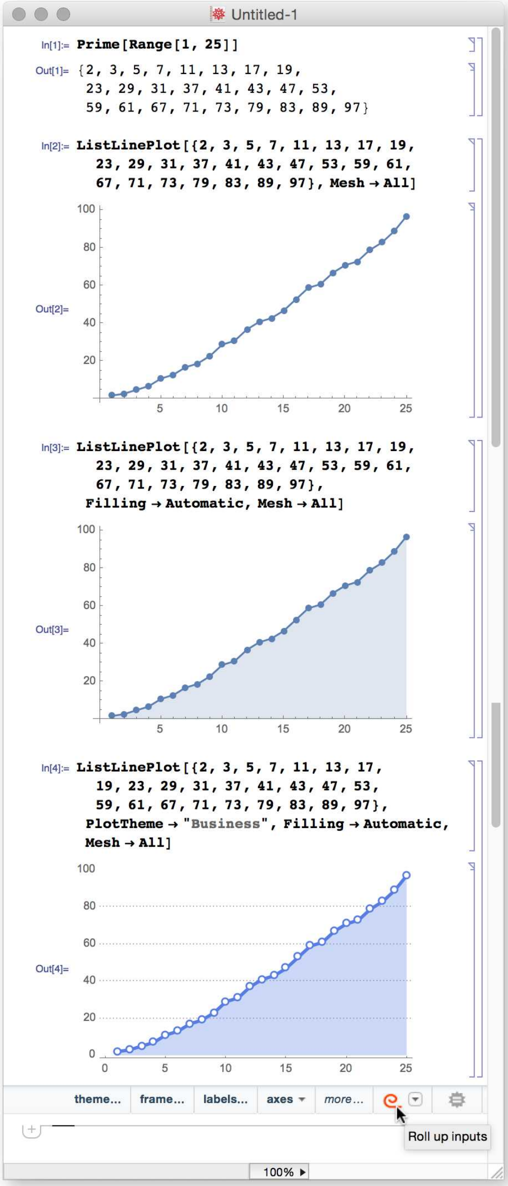
first 25 prime numbers

Values

```
Prime[Range[1, 25]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
```

Click the Wolfram Language syntax to discard the free-form input and to keep the **Prime[Range[1, 25]]** command. Using the Suggestions Bar, selections can be made to plot the points and a line through the points and then to add filling and change the theme to the **Business** setting. When finished, the button to roll up inputs can be clicked, which will condense all of the sequential inputs into a single input that can be evaluated to arrive at the final result.

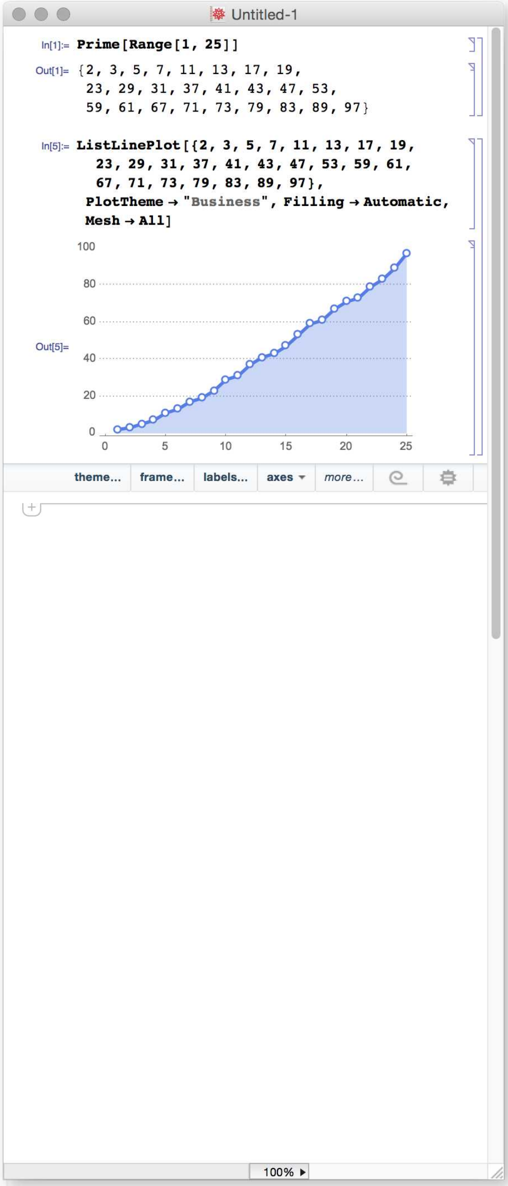


```
In[1]:= Prime[Range[1, 25]]
Out[1]= {2, 3, 5, 7, 11, 13, 17, 19,
  23, 29, 31, 37, 41, 43, 47, 53,
  59, 61, 67, 71, 73, 79, 83, 89, 97}

In[2]:= ListLinePlot[{2, 3, 5, 7, 11, 13, 17, 19,
   23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
   67, 71, 73, 79, 83, 89, 97}, Mesh → All]
Out[2]=

In[3]:= ListLinePlot[{2, 3, 5, 7, 11, 13, 17, 19,
   23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
   67, 71, 73, 79, 83, 89, 97},
  Filling → Automatic, Mesh → All]
Out[3]=

In[4]:= ListLinePlot[{2, 3, 5, 7, 11, 13, 17,
   19, 23, 29, 31, 37, 41, 43, 47, 53,
   59, 61, 67, 71, 73, 79, 83, 89, 97},
  PlotTheme → "Business", Filling → Automatic,
  Mesh → All]
Out[4]=
```



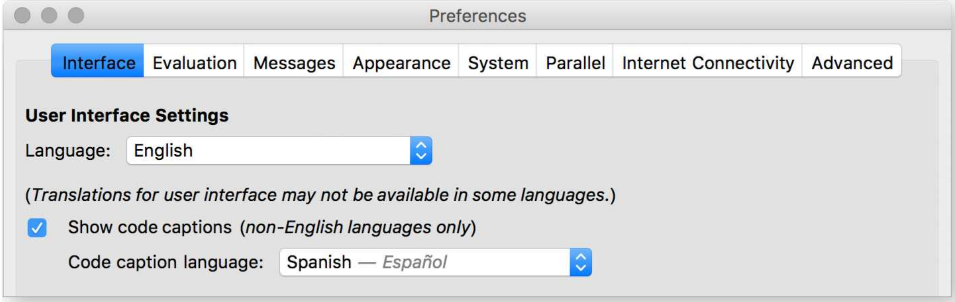
```
In[1]:= Prime[Range[1, 25]]
Out[1]= {2, 3, 5, 7, 11, 13, 17, 19,
  23, 29, 31, 37, 41, 43, 47, 53,
  59, 61, 67, 71, 73, 79, 83, 89, 97}

In[5]:= ListLinePlot[{2, 3, 5, 7, 11, 13, 17, 19,
   23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
   67, 71, 73, 79, 83, 89, 97},
  PlotTheme → "Business", Filling → Automatic,
  Mesh → All]
Out[5]=
```

If you do not like the Suggestions Bar, there is an icon all the way to the right to hide it. This can be useful when you know what you are doing and do not need the assistance. You can bring the Suggestions Bar back by clicking the arrow icon all the way to the right.

Code Captions

The examples shown so far have been in English. Code captions can be used to aid in the understanding of Wolfram Language by supplementing English inputs with another desired language. The **Interface** section of the **Preferences** menu provides a list of available languages for code captions.



User Interface Settings

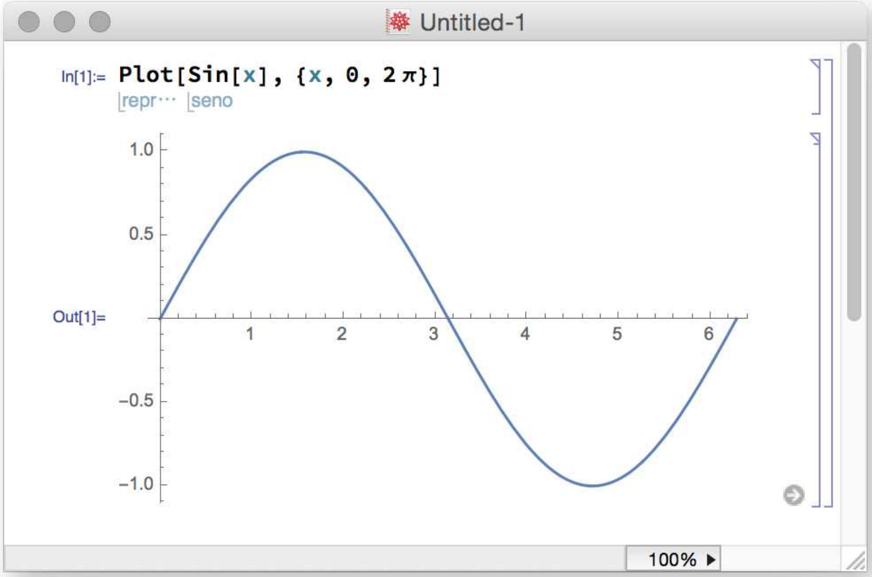
Language: English

(*Translations for user interface may not be available in some languages.*)

☑ Show code captions (*non-English languages only*)

Code caption language: Spanish — *Español*

Once code captions are enabled, input cells will contain a translation to the desired language under each function name. The code caption is truncated if the translation is longer than the English function name, and the ellipses that indicate truncation can be moused over to display the full translation.



```
In[1]:= Plot[Sin[x], {x, 0, 2 π}]
        repr⋯ seno
Out[1]=
```

Output Matches Input

When given a command, Mathematica will attempt to return the most exact result possible. The form of this output will match the form of the input, so giving Mathematica exact input will result in Mathematica returning exact output. Giving Mathematica approximate input will result in approximate output. When a mixed input is given, such as a computation that contains both exact and approximate numbers, the result that is returned will be approximate.

For example, computing a division of two integers will return an exact result.

$$\mathbf{\frac{1234}{2468}}$$

$$\frac{1}{2}$$

A fraction can be entered with two-dimensional typesetting by using the **Basic Math Assistant** from the **Palettes** menu. The fraction button in the **Basic Math Assistant** palette will also show you the keyboard shortcut that can be used, which is Ctrl+/ in this case.

Mathematica performs this division and automatically simplifies the result, but the result is still returned as an exact quantity.

In contrast, computing a division of two approximate numbers will return an approximate result.

$$\mathbf{\frac{1.35}{2.46}}$$

$$0.54878$$

And computing a division of an exact number and an approximate number will also return an approximate result.

$$\mathbf{\frac{135.7}{246}}$$

$$0.551626$$

Sometimes you know that you will want to receive an approximate result even though you are giving Mathematica exact input, like when you want a decimal approximation of dividing two integers. In situations like this, you can force Mathematica to give an approximate result by appending a decimal place to the end of one of the integers. Input like **1234.0/2468**, or even **1234./2468** (without the trailing zero), will force Mathematica to return an approximate result.

Converting between Exact and Approximate Results

While exact results are useful, sometimes it can be desirable to receive an approximate result. One way to do that is to introduce a decimal place in the input, which will force Mathematica to return an approximate result. There is also a Wolfram Language command named **N** that can be used to approximate an exact quantity. To calculate an approximation of $\frac{1234}{5678}$, just pass the fraction as an argument to the **N** command.

$$\mathbf{N}\left[\frac{1234}{5678}\right]$$

```
0.21733
```

> Almost all Wolfram Language commands have names that are full English words. There are a few, however, that are used so often that they have abbreviated forms. **N** is one of them; some of the others are **D**, for differentiation, and **Det**, for determinant.

The **N** command also accepts an optional second parameter, which specifies the desired number of digits of precision to be returned by the approximation.

$$\mathbf{N}\left[\frac{1234}{5678}, 100\right]$$

```
0.2173300457907713983797111659034871433603381472349418809439943642127509⋱.
  865093342726312081718915111095
```

Sometimes a user may not know that approximate results are desirable until a result is received. In those cases, the shorthand for the last output, **%**, can be quite handy. This shorthand is a quick way to reference the previous output.

$$\int_0^1 \mathrm{Sin}[x]\, dx$$

```
1 – Cos[1]
```

```
N[%]
```

```
0.459698
```

Similarly, the **Postfix** operator, which has the shorthand symbol **//**, can be used to apply **N** as the final step in a command or sequence of commands.

$$\int_0^1 \mathbf{Sin}[x]\, dx \;\mathbf{//\; N}$$

```
0.459698
```

Of course, wrapping **N** around the original input is also an acceptable practice.

$$\mathbf{N}\left[\int_0^1 \mathbf{Sin}[x]\, dx\right]$$

```
0.459698
```

> Postfix operations can be an easy way to apply a final touch to a result, and you can nest multiple postfix operations on top of one another. You should stay away from using postfix operations, however, when you are assigning results to variables, because you might end up accidentally storing the "form" of the result instead of the content, which means the variable might not act as expected when you reference it in other calculations.

Conclusion

Thanks to free-form input, users can start doing sophisticated things in Mathematica immediately. User-assistance features like automatic code completion and the Suggestions Bar also help users transition to direct use of the Wolfram Language, and from there, the sky is pretty much the limit of what can be accomplished. The next chapter will address the structure of Mathematica notebooks and the use of them for word processing and containing technical information.

Exercises

1. Create a new notebook. At the top, a horizontal I-beam will be presented that shows it is ready for you to type. Enter 705 divided by 3, and evaluate the calculation to get a result.
2. Use free-form input to calculate the sum of *n*, where *n* goes from 1 to 10.

3. Use the Suggestions Bar to find the largest nontrivial divisor of your result from Exercise 2.
4. Using the Wolfram Language, plot the expression $x + 5$, where x goes from 1 to 10.
5. Using the Suggestions Bar, modify the graphic by applying a "Scientific" plot theme.
6. Using the free-form input, ask Mathematica to add the 107th prime number and the 108th prime number. (Reminder: upon evaluating your free-form input, you will not only get the answer but also the Wolfram Language input for the free-form input you typed.)
7. Use free-form input to find out whether 611 is larger than 3^6.
8. Use free-form input to plot $\sin(x) + \cos(y)$.
9. Change the command from Exercise 8 to plot $\sin(x) * \cos(y)$ instead. (Reminder: the Wolfram Language syntax returned in a free-form input pod can be clicked to discard the free-form input and to modify the underlying Wolfram Language command.)
10. Use the Suggestions Bar to remove the mesh from the result in Exercise 9.

CHAPTER 4

Word Processing and Typesetting

Introduction

Mathematica is best known as a computation system, but its document-based workflow also makes it very good at creating documents with nicely formatted text and equations. Mathematica is an excellent choice for creating technical documents that contain textual exposition along with graphs, figures, code and even interactive elements. Unlike multiple tools that only accomplish specific tasks, Mathematica has the capability to serve as a single environment for exploration, experimentation and documentation, from the beginning of a project to the end.

Both Mathematica on the desktop and Mathematica Online make for great document creation tools. This chapter is written with Mathematica on the desktop in mind and discusses some topics, like palettes and stylesheets, that are unique to that environment. Other topics, like adding styled cells to a notebook, are applicable to both environments, although the user interface elements may be slightly different on each.

For an example of a technical document created with Mathematica, look no further than this book! This entire book was created using Mathematica, with each chapter saved as a separate notebook and then all the chapters assembled into a single document for printing.

The Structure of Notebooks

Notebooks are comprised of cells, which hold pieces of information and form a structural basis for documents in Mathematica. Some examples of cells were shown in the previous chapter, where input cells were used to hold commands, and output cells were used to hold the results of performing those commands. There are other types of cells available in Mathematica, and these are generally referred to as cell styles.

Cell styles include input, output, title, subtitle, section, subsection, subsubsection, text and item, but there are many other choices available. Most of these cell styles are meant to hold plain text, but some of them, like input and output, are designed to hold commands or results from evaluating commands. Cell styles also contain definitions that govern appearance, so text within a default title cell will look large and red, while text within a default text cell will be small and black. Definitions for cell styles are controlled by a notebook's stylesheet, and editing the stylesheet can be an easy way to quickly change the appearance of a document; this will be discussed later in this chapter.

Cells are automatically grouped according to a hierarchy. This grouping is illustrated by the brackets at the right side of a notebook. Each bracket corresponds to a cell, and larger, spanning brackets correspond to groups of cells. Cell brackets can be double-clicked to collapse or expand cell content, which is useful for hiding input, hiding output or hiding a group of cells. When a cell group is collapsed, a little triangle appears in the cell group's bracket to let users know that more content is available, and that content can be shown by double-clicking the bracket with the triangle.

A notebook can contain a lot of hierarchical content, like title, section, subsection, subsubsection and text cells. This can make it harder to choose the right cell brackets to double-click if you want to hide or show cell content. One way to make this easier is to open Mathematica's **Preferences** menu, navigate to the **Interface** tab and tick the box for **Show open/close icon for cell groups**. This will add a little triangle icon to the left of cells that are at the top level of a cell group, and single-clicking this triangle will hide or show all of the cells in that cell group. This can be a lot easier than trawling through a bunch of nested cell brackets to find the right ones to double-click.

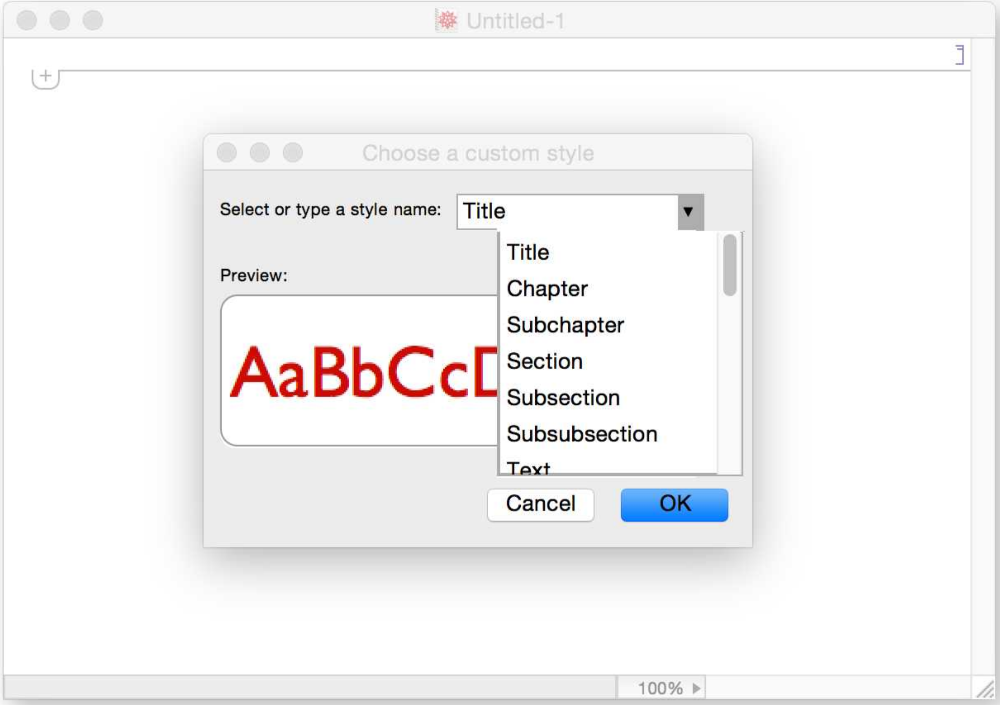
Adding Plain Text Cells

The easiest way to add plain text to a notebook is to click the Cell Insertion Assistant and choose **Plain text** from the popup menu. Making that selection creates a new cell and the cursor is placed inside the cell, ready to accept text.

A second way to create a text cell is to use the **Format** menu, from which different styles can be chosen. To create a new text cell, place the cursor in the desired location—either between existing cells or at the top or bottom of a notebook—and click the **Format** menu, select **Style** and choose **Text**. The **Style** submenu also lists a keyboard shortcut to create a text cell, which is Alt+7 or Cmd+7, depending on the operating system that is being used.

Adding Styled Cells

Styled cells, like title cells and section cells, can be added to a notebook using the Cell Insertion Assistant. Place the cursor in the desired location, click the Cell Insertion Assistant icon and choose **Other style of text...**; this will open a dialog box with a drop-down menu of choices for title, chapter, subchapter, section, subsection and a variety of other styles of cells.



Select or type a style name: Title

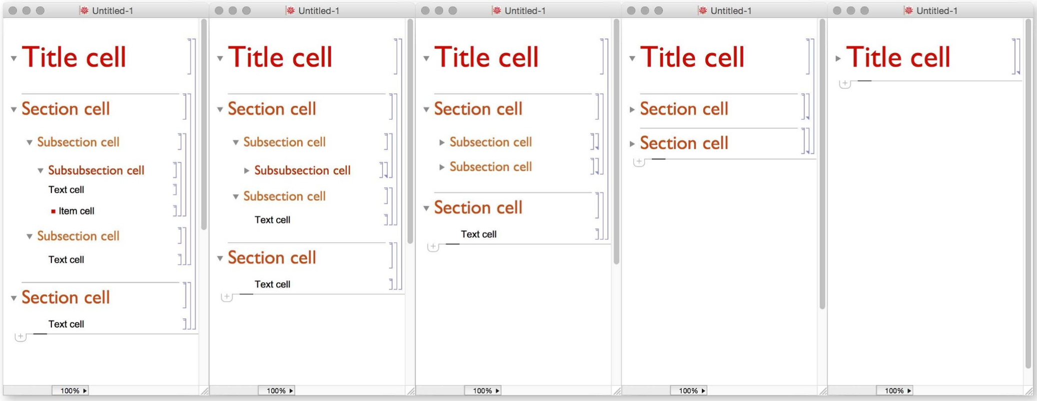
Preview: AaBbCcD

- Title
- Chapter
- Subchapter
- Section
- Subsection
- Subsubsection
- Text

A second way to create styled cells is to use the **Format** menu, select **Style** and choose the appropriate cell style. As with text cells, the **Style** submenu also lists the keyboard shortcuts to create the most commonly used cell styles.

The keyboard shortcuts for the most commonly used cell styles start with Alt+1 or Cmd+1, which creates title cells, and end with Alt+9 or Cmd+9, which creates input cells. The keyboard shortcuts are also arranged according to the cell hierarchy, with title cells at the top of the list and input cells at the bottom of the list. If you are only going to memorize a few of them, you should know Alt+1 or Cmd+1 for title cells, Alt+4 or Cmd+4 for section cells and Alt+7 or Cmd+7 for text cells—with just those three, you can quickly create a nicely structured notebook.

Each of these types of styled cells will have a slightly different appearance, and each will have a different position in the hierarchy of how the notebook is organized into cell groups. For example, a title cell has the highest position in the hierarchy, so any other cell style will be grouped below the title cell, and if that cell group is collapsed, only the title cell will show. The following screen shot shows an example of what happens when certain types of cell groupings (subsubsections, subsections, sections and titles) are collapsed.



You can cut and copy entire cells by selecting cell brackets and then using the **Edit** menu or the canonical keyboard shortcuts for cutting and copying. When you paste a cell, its cell style will be pasted as well, and then you can overwrite the content in the new cell.

Cell styles can also be applied to existing cells by highlighting their cell brackets and selecting a new style from the **Style** submenu, or by applying a keyboard shortcut.

In the course of creating a document, you may decide you want to change all of one style of cell to another style. In such situations, you can press the `Alt` or `Option` key and click the bracket of a particular style of cell. This will select all the cells of that style in your notebook, allowing you to quickly switch them to a new cell style by using the **Format** menu or a keyboard shortcut.

Adding Typesetting

Notebooks can include typeset expressions, both as input for commands and as part of plain text cells and styled text cells. To see a listing of what typesetting tools are available, go to the **Palettes** menu and choose one of the assistant palettes (**Basic Math Assistant**, **Classroom Assistant**, **Writing Assistant**). Each of those palettes has a **Typesetting** section with buttons for available forms, symbols, operators and icons.

The **Classroom Assistant** palette contains all the functionality of the **Basic Math Assistant** and the **Writing Assistant** palettes, and then some. The **Classroom Assistant** palette is the palette to open if you want the most functionality available at your disposal, and it is useful even if you are not using Mathematica in a classroom setting or for academic work.

When the cursor is inside a cell, clicking a button on the palette will insert templates for positional elements like subscripts and superscripts, or symbols for things like Greek letters and special characters. As the pointer hovers over the buttons in the palette, a popup window will provide more details, along with keyboard shortcuts, if available. For example, the keyboard shortcut to create a two-dimensional fraction is Ctrl+/, while the symbol π can be entered using the escape sequence Esc pi Esc, which means, press the Esc key, then type "pi" (without quotes) and press the Esc key again. Mathematica will interpret that key sequence to format the result as the Greek letter.

You can actually type the Greek letter π using Esc p Esc. Other letters in the Greek alphabet follow the same pattern, so Esc a Esc creates α, Esc b Esc creates β and so on. Along the same lines, the capital versions of Greek letters can be entered a similar way: Esc P Esc creates Π, Esc D Esc creates Δ and so on.

Customizing Text

When a cell of a particular style—like a text cell—is created, Mathematica applies default styling to any content within that cell. That styling can be overridden, giving users control over choices like font, size, face and color.

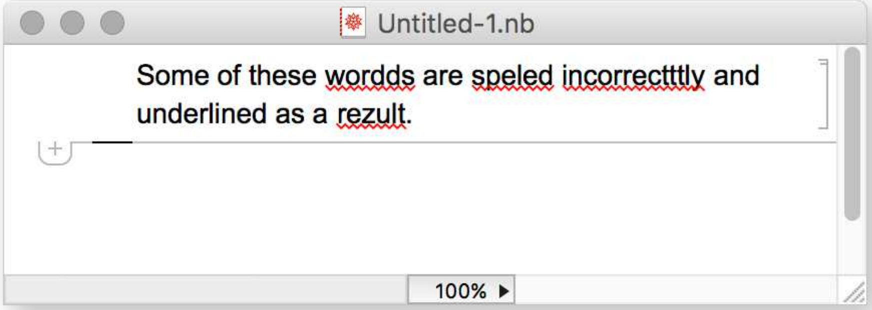
To change the style of a particular piece of text, either highlight the text directly or highlight its cell bracket to select all of the content of the cell. Once selected, the **Format** menu can be used to change the font, face, size, text color and background color.

If you override the default cell styles for a particular cell, those style definitions will travel along with it when copied and pasted. Customizing one cell and then copying and pasting it can be a quick way to transfer that style to other cells, but for any large-scale changes, stylesheets should be used instead.

Checking Spelling

Mathematica's built-in spellchecker can be used to review a document's contents. The **Check Spelling** command from the **Edit** menu can be used to access Mathematica's built-in dictionary, which is scientifically focused to suit the typical user and can be extended as needed.

In addition, Mathematica will automatically underline words that do not appear in its default dictionary. This behavior helps to highlight errors in spelling or typing prior to running the built-in spellchecker.



Some of these wordds are speled incorrectttly and underlined as a rezult.

This default behavior can be turned off using the **Preferences** menu. Navigate to the **Interface** section and uncheck **Check spelling as you type**.

Showing Page Breaks

The content of a cell may be displayed on a single line if the content is short. For cells that have longer content, Mathematica will automatically wrap words at the edge of the notebook window. When notebook windows are resized, Mathematica will automatically adjust and wrap the cell content in an appropriate manner for the new window size.

When a notebook is printed, however, Mathematica will be forced to display the cell content according to the paper size the user selects. To view how word wrapping will look in the fixed dimensions of a paper printout (or if printed to a file, like PDF), click the **Format** menu, select **Screen Environment** and choose **Printout**. This will show the notebook document as it would appear with fixed dimensions.

To see where page breaks occur, click the **File** menu, select **Printing Settings** and then choose **Show Page Breaks**. Page breaks are displayed as thick gray bars that run the width of the notebook window. Mathematica chooses optimal locations for page breaks, but users can adjust their content to accommodate these breaks by breaking large cells into multiple smaller cells or resizing graphics. Users can also manually insert page breaks by placing the cursor in a specific location and using the **Insert** menu to select **Page Break**.

It is always possible to toggle back to the original view by deselecting the **Show Page Breaks** option and changing the **Screen Environment** back to **Working**.

For many users, it is perfectly fine to ignore screen environments and page breaks and let Mathematica handle that for you. If you print something out and then notice you want to tweak it, you can go back and make the desired changes—but many people never change these settings.

Working with Stylesheets

A stylesheet in Mathematica is a special notebook used to define systematized formatting rules. Stylesheets are incredibly useful because they allow users to define what a particular style of cell should look like; once those definitions are made, then each cell will adhere to that style. An alternative to using stylesheets is to manually set the typeface, color, font size and other options for each cell individually. That can be fine for shorter documents, but for longer documents, or a collection of documents, a stylesheet can be a real timesaver.

Do you have to learn how to define your own stylesheet? Of course not. Mathematica has different built-in stylesheets that you can use, including the default one applied to all new notebooks.

Linked Stylesheets and Embedded Stylesheets

Stylesheets can be either linked or embedded. A linked stylesheet resides *outside* of a notebook, such as in a directory on a computer, and can be referenced by many different notebooks. A linked stylesheet is very powerful, since any changes within the stylesheet will be applied to each of the notebooks that links to it. However, in order to share a notebook with a linked stylesheet, the stylesheet must be shared as well, and this can present some challenges with distribution.

An embedded stylesheet resides *within* a notebook. This means that any changes to the stylesheet will only affect that single notebook. Embedded stylesheets allow great flexibility since notebooks can be shared with others without the need to also distribute a separate stylesheet; instead, just the notebook itself needs to be shared, and its style definitions travel along with it.

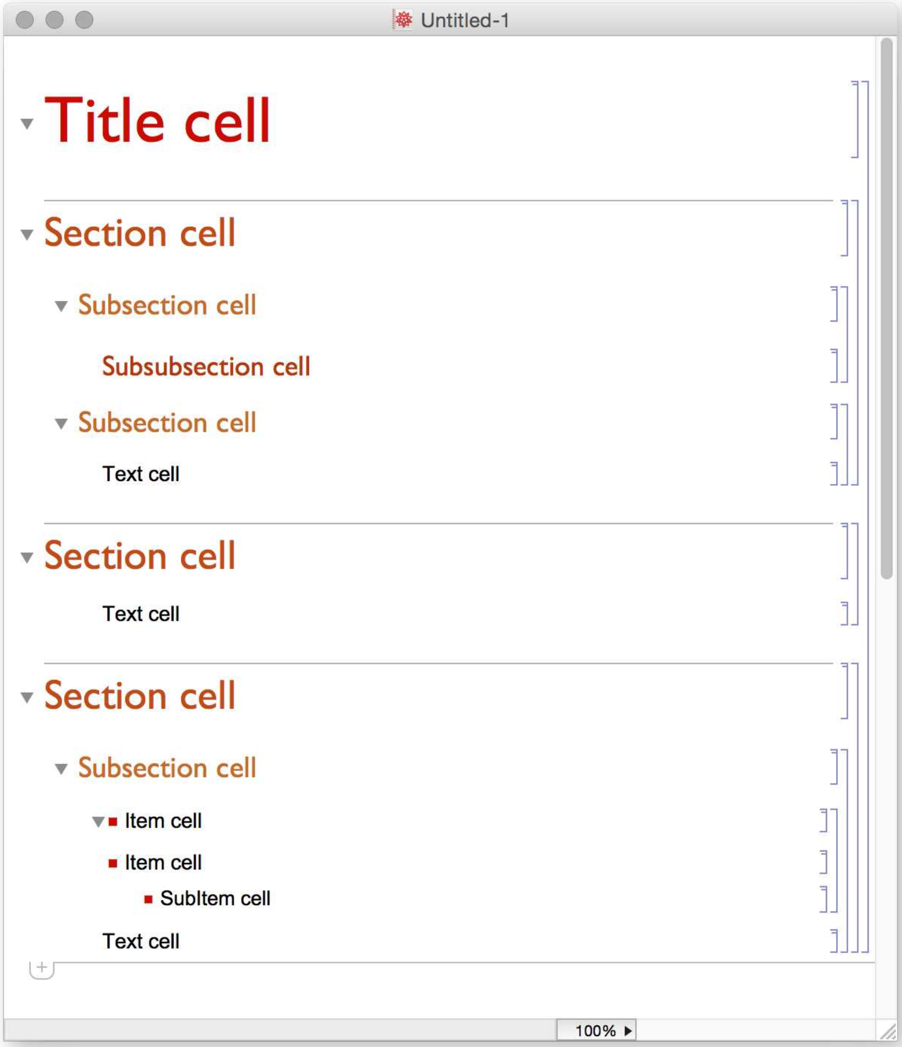
Using embedded stylesheets is a preferred approach for many users because it is easier to share a document with an embedded stylesheet than with a linked stylesheet.

Default Cell Styles and Built-in Stylesheets

This chapter has discussed the various types of default cell styles that Mathematica uses, such as title, section, subsection, text, input and output cell styles, and the hierarchical structure that determines how cells are grouped together. These cell groups can be collapsed or expanded to hide or show content, which is very useful both when working in and presenting from a notebook.

The following examples do not have input cells to be retyped and evaluated, but you should follow along by recreating the documents shown in the screen shots and applying the same operations that are detailed in the text.

The appearance of built-in cell styles is governed by the notebook's stylesheet. When a notebook is created, it uses a default stylesheet, which is why all section cells, for example, are the same color and size and have a border at the top. The following screen shot shows an example of a Mathematica notebook that uses the styling defined by the default stylesheet.



Untitled-1

Title cell

Section cell

Subsection cell

Subsubsection cell

Subsection cell

Text cell

Section cell

Text cell

Section cell

Subsection cell

▪ Item cell

▪ Item cell

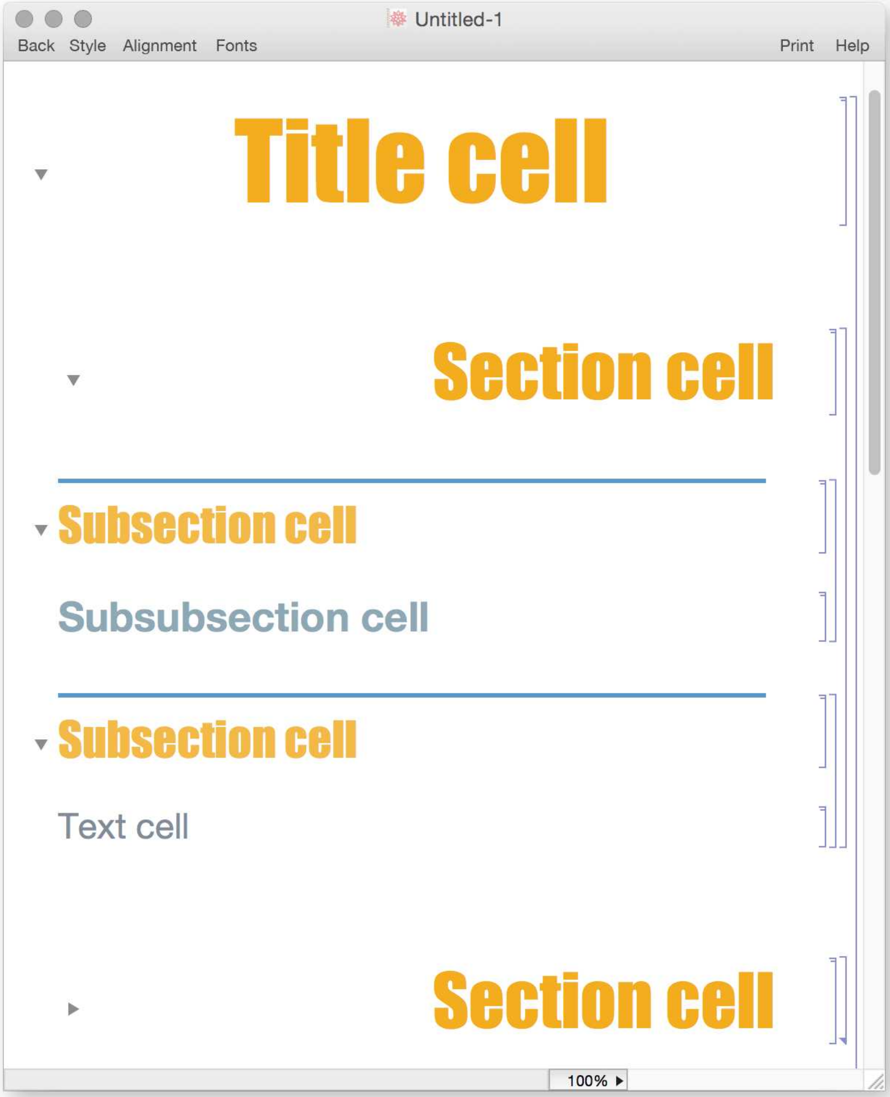
▪ SubItem cell

Text cell

100%

A notebook's stylesheet can be changed by clicking the **Format** menu, selecting **Stylesheet** and making a choice from the available options. The **Stylesheet Chooser** (located in the same **Stylesheet** submenu) can be used to see some of the various stylesheet options side by side, and from this window, choices can be made to apply a stylesheet to the active notebook or to a brand-new notebook.

Some of the most visually dynamic stylesheets are not available from the **Stylesheet Chooser** but are available from the **Format** ▶ **Stylesheet** ▶ **SlideShow** submenu. Applying the Sunrise stylesheet to the notebook from the preceding example creates a much different appearance than the default one.



Title cell

Section cell

Subsection cell

Subsubsection cell

Subsection cell

Text cell

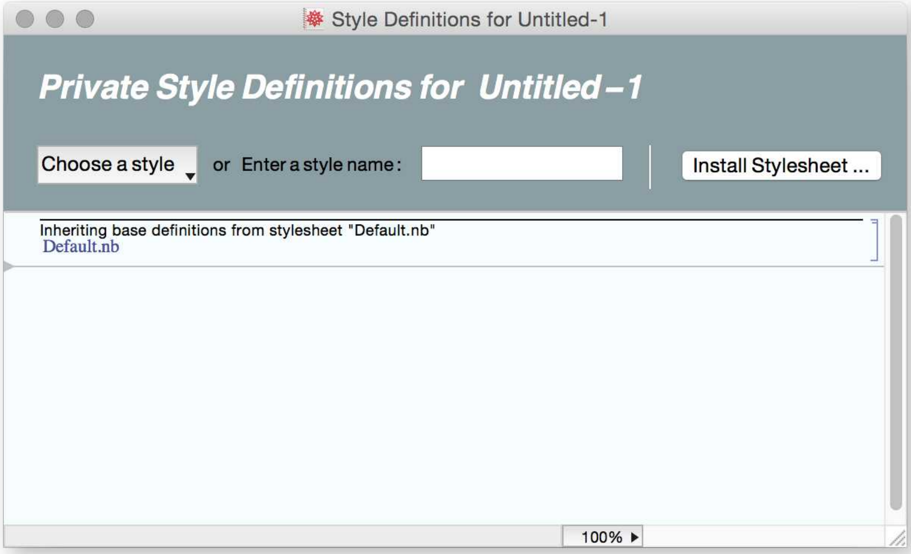
Section cell

Notice how application of the stylesheet affected many parts of the notebook, including text color and size, background color, indentation and choice of font. This same principle can be applied to custom, user-defined stylesheets, where changing a few definitions can cascade dramatic changes throughout a notebook.

What stylesheet you should select is a matter of personal taste. The authors find the default stylesheet to be an attractive choice, but that does not mean you cannot use, say, the Sunrise stylesheet instead.

Defining a Custom Stylesheet

Creating a custom stylesheet for a notebook is simple. The first step is to create a new notebook or to open an existing notebook. Once the notebook is created or opened, choose **Edit Stylesheet** from the **Format** menu to see the style definitions for the notebook. The style definitions will be displayed in a new window, and the content of this window will be mostly blank.



Style Definitions for Untitled-1

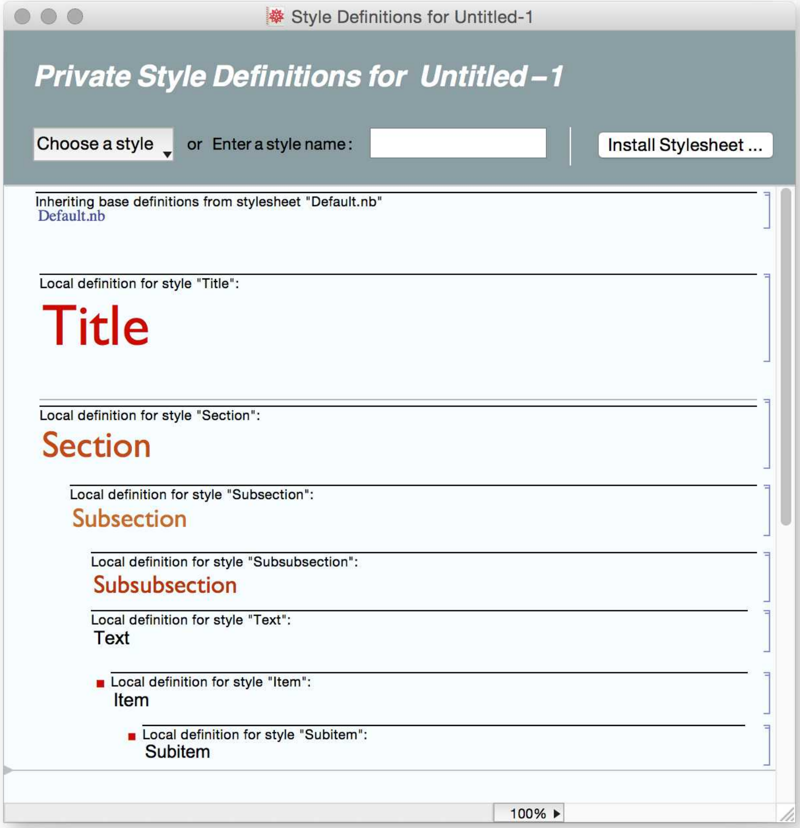
Private Style Definitions for Untitled–1

Inheriting base definitions from stylesheet "Default.nb"
Default.nb

The first cells (or in this case, the first cell) will indicate where the base definitions are inherited from. This is because Mathematica supports cascading stylesheets, so one stylesheet can reference the definitions of another stylesheet, and so on. New notebooks use Mathematica's default styling, and those definitions are stored in the Default.nb file that is linked from the first cell.

To change the styling for a certain type of cell, it must be selected, so that it appears in the style definitions notebook. A cell style can be chosen from the drop-down menu or by entering its name in the input field and pressing Enter. Once chosen, a definition cell for that cell style will appear in the notebook window.

The following screen shot shows an example of selecting the styles for title, section, subsection, subsubsection, text, item and subitem cells.



Style Definitions for Untitled-1

Private Style Definitions for Untitled–1

Choose a style ▾ or Enter a style name: Install Stylesheet ...

Inheriting base definitions from stylesheet "Default.nb"
Default.nb

Local definition for style "Title":
Title

Local definition for style "Section":
Section

Local definition for style "Subsection":
Subsection

Local definition for style "Subsubsection":
Subsubsection

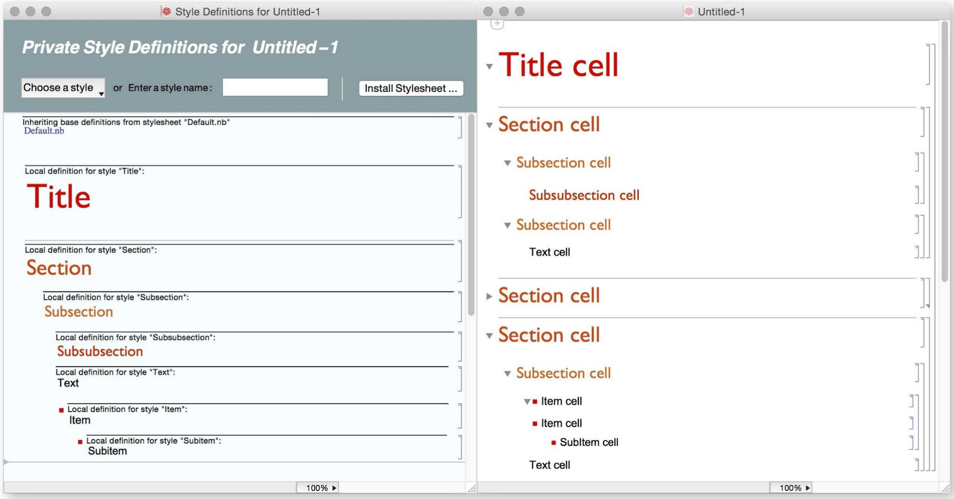
Local definition for style "Text":
Text

▪ Local definition for style "Item":
Item

▪ Local definition for style "Subitem":
Subitem

100%

Seven definition cells are displayed, one for each of the styles that was chosen. Note that the appearance of these definition cells matches the appearance of the cells in the notebook itself: in both instances, title cells are large and red, section cells are brown with a gray top border and so on. This is very obvious if the windows are viewed side by side.



Style Definitions for Untitled-1

Private Style Definitions for Untitled–1

Choose a style ▾ or Enter a style name: Install Stylesheet ...

Inheriting base definitions from stylesheet "Default.nb"
Default.nb

Local definition for style "Title":
Title

Local definition for style "Section":
Section

Local definition for style "Subsection":
Subsection

Local definition for style "Subsubsection":
Subsubsection

Local definition for style "Text":
Text

▪ Local definition for style "Item":
Item

▪ Local definition for style "Subitem":
Subitem

100%

Untitled-1

Title cell

Section cell

Subsection cell

Subsubsection cell

Subsection cell

Text cell

Section cell

Section cell

Subsection cell

▪ Item cell

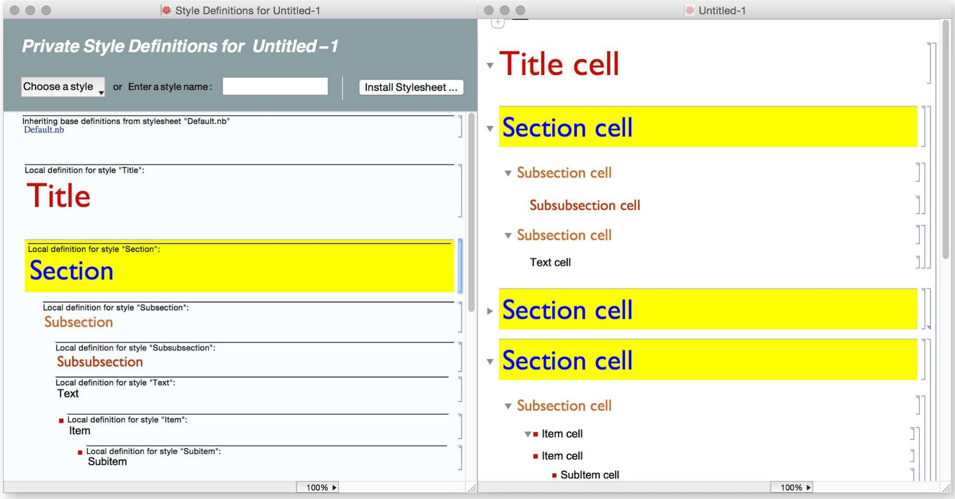
▪ Item cell

▪ SubItem cell

Text cell

100%

Any changes made to the definition cells in the style definitions window will be applied to all corresponding cells of that style within the notebook. To change a definition cell, highlight its cell bracket and then use any of the usual methods already discussed, such as the **Format** menu or palettes like the **Writing Assistant**. For example, changing the definition cell for the section style so that its text is blue and 36 point with a background color of yellow will change all the section cells in the notebook to those same settings. The result will look like the following screen shot.



Style Definitions for Untitled-1

Private Style Definitions for Untitled – 1

Choose a style or Enter a style name: Install Stylesheet ...

Inheriting base definitions from stylesheet "Default.nb"
Default.nb

Local definition for style "Title":
Title

Local definition for style "Section":
Section

Local definition for style "Subsection":
Subsection

Local definition for style "Subsubsection":
Subsubsection

Local definition for style "Text":
Text

Local definition for style "Item":
Item

Local definition for style "Subitem":
Subitem

100%

Untitled-1

Title cell

Section cell

Subsection cell

Subsubsection cell

Subsection cell

Text cell

Section cell

Section cell

Subsection cell

Item cell

Item cell

SubItem cell

100%

Style definitions can be cleared through several different methods. In the style definitions window, the cell bracket of a cell can be highlighted and then **Clear Formatting** can be chosen from the **Format** menu. This will clear any local style definitions, and the cell will assume the appearance as specified by its stylesheet.

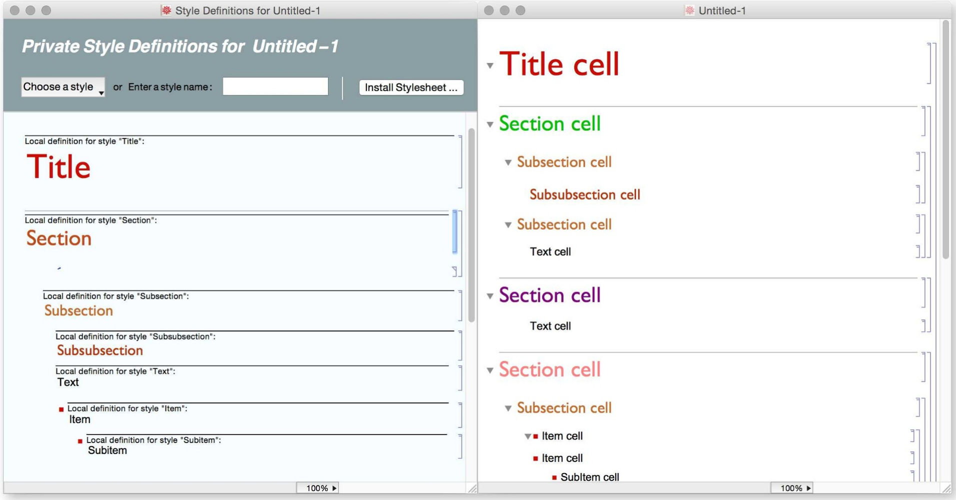
Another way to reset the style definitions for a particular style is to highlight its cell bracket in the style definitions window and press the Delete key. Once deleted, the style for that cell type will revert back to the default definition.

> If you delete a definition cell from the style definitions window, you will need to re-choose that definition cell if you later decide you want to make changes to that cell style. You can re-choose the cell the same way it was originally chosen, by using the drop-down menu in the style definitions window or by typing a style name into the input field.

Overriding Style Definitions

Stylesheets are very powerful tools for defining a set of rules to govern the overall appearance of a notebook. There may be times, however, when it is desirable to make some additional changes—not to all cells of a certain type, but to a particular cell. Mathematica allows users to do this by overriding style definitions for individual cells or cell contents. This behavior is possible because styles in Mathematica cascade, starting with stylesheets and ending at the cellular level. This allows the user to have fine-grained control over the appearance of notebooks by setting global options for a notebook with a stylesheet but overriding those options when necessary.

The following screen shot shows an example of overriding styles. The stylesheet definitions for a notebook are displayed on the left, and the notebook on the right shows that the section cells have had each of their text colors changed by selecting the cells and changing their colors with the **Format** menu.


Style Definitions for Untitled-1
Private Style Definitions for Untitled–1
Choose a style
or Enter a style name:
Install Stylesheet ...
Local definition for style "Title":
Title
Local definition for style "Section":
Section
Local definition for style "Subsection":
Subsection
Local definition for style "Subsubsection":
Subsubsection
Local definition for style "Text":
Text
Local definition for style "Item":
Item
Local definition for style "Subitem":
Subitem
100%
Untitled-1
Title cell
Section cell
Subsection cell
Subsubsection cell
Subsection cell
Text cell
Section cell
Text cell
Section cell
Subsection cell
Item cell
Item cell
SubItem cell
100%

If you create a new notebook and customize many individual cells to look a certain way, do not be surprised if editing the stylesheet does not do anything; you may have already made changes at a very specific level (to a cell or its contents), and these changes will take precedence over changes made to a more general level (the stylesheet).

Moving Style Definitions from One Stylesheet to Another

If a notebook has a custom stylesheet, those style definitions can be moved to other notebooks with a couple of different methods. Perhaps the easiest is to open a notebook with the custom stylesheet, save the notebook under a new name and then delete all the content. Deleting a notebook's content does not affect its stylesheet, so this series of operations will have created a blank document into which new content can be entered, and which will be styled using the desired stylesheet.

Another option is to open a notebook and its corresponding style definitions. Once the style definitions window is open, individual cells can be selected by highlighting their cell brackets, or all cells can be selected by using **Select All** from the **Edit** menu. Once selected, the style definition cells can be copied, which will also copy their formatting. Once they have been copied, open a different notebook (either a brand-new one or an existing one), and then open the new notebook's style definitions. Once the new notebook's style definitions window is open, paste the style definition cells from the clipboard. This process allows users to move individual cell styles or the complete contents of a stylesheet from one document to another.

Conclusion

Mathematica's document-centric workflow provides users with a single environment to both perform their work and document its results. Thanks to built-in cell styles and their native hierarchy, notebooks are cleanly organized and easily customizable for users' particular needs. Stylesheets allow users to quickly make global and systematic changes to the appearance of their notebooks by changing the styling definitions for particular types of cells. Mathematica should be seen as the immediate tool of choice when a need to create a technical or scientific document arises.

Exercises

1. Create a new notebook. Add a new title cell with text that says "My First Notebook."
2. Highlight the word "First" and change the text to be blue and bold.
3. Place the cursor below the title cell and use the Cell Insertion Assistant to create a new section cell that contains the text "Initial Calculations."
4. Add a subsection cell at the bottom of the notebook that says "Solving an equation."
5. Below the subsection cell, create a free-form input cell to solve the equation $12\,x + 24 = 0$.
6. Add a text cell at the bottom of the notebook to say "In the exampel above, x is -2." (Before you get too concerned, the misspelling is on purpose.)
7. In formatting the final document, you may wish to remove the free-form input and replace it with the Wolfram Language instead. Do that now to clean up the input/output of the calculation.
8. Hide the input cell so that only the result from the calculation is shown.
9. Use the spellchecker to correct any misspellings in the document.
10. Collapse the cell group containing the title cell, so that only the title cell is shown.

CHAPTER 5

Presenting with Slide Shows

Introduction

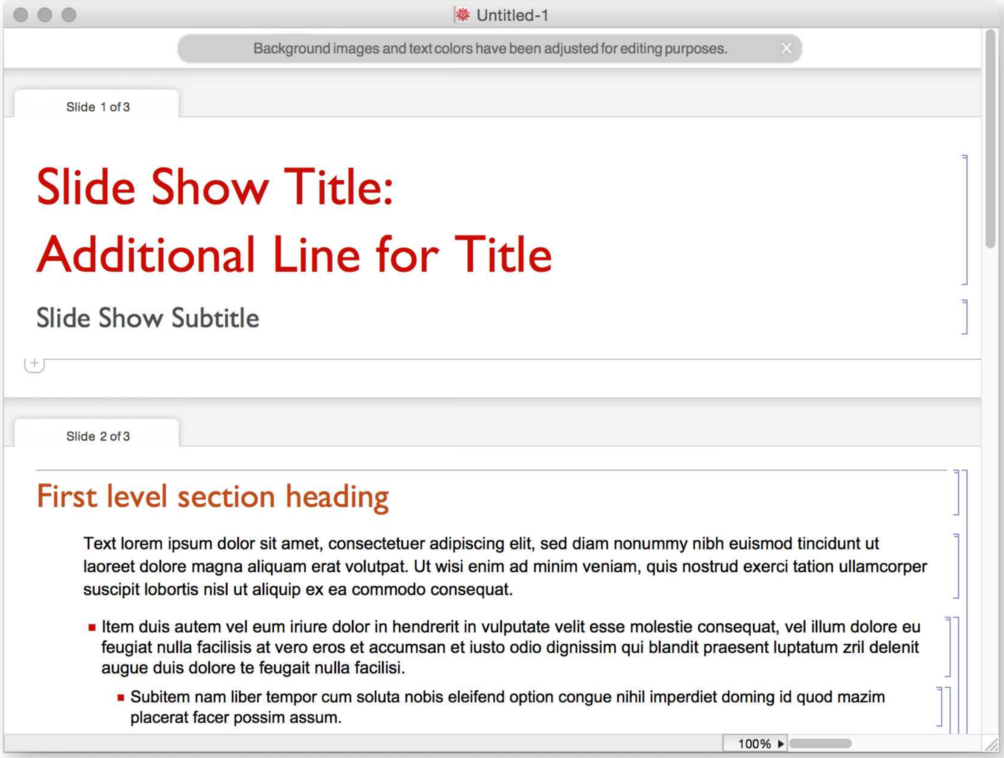
In addition to creating documents like reports, articles, course handouts and books, Mathematica can also create slide shows. Since slide shows are created as notebooks, they can include text, typesetting, graphics, code and interactive elements. Instead of using one piece of software for computation, another for graphics, a different one for word processing and typesetting and yet another one to create a presentation, Mathematica provides a single environment for the entire technical workflow.

This chapter is written with Mathematica on the desktop in mind and discusses some topics that are unique to that environment. Other topics, like using grids to align elements, are applicable to both Mathematica on the desktop and Mathematica Online.

Creating a New Slide Show

To create a new slide show, click the **File** menu, select **New** and choose **Slide Show**. A window appears with styled templates to choose from. Choosing a template will open a slide show notebook with some placeholder content, and the **Slide Show** palette will be opened as well.

The Slide Show palette can be reopened at any time by choosing it from the **Palettes** menu.



Background images and text colors have been adjusted for editing purposes.

Slide Show Title:
Additional Line for Title

Slide Show Subtitle

First level section heading

Text lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

- Item duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zril delenit augue duis dolore te feugait nulla facilisi.
  - Subitem nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum.

A slide show is just a Mathematica notebook that has some special cells to separate the slides from one another. When a new slide show is created, these slide delimiter cells say "Slide 1 of 3," "Slide 2 of 3" and so on. The placeholder content can be overwritten or deleted, and new content can be added.

To add more slides, place the cursor at the top of the notebook, between cells or at the bottom of the notebook, and then click the **New Slide Template** or **Blank Slide** button on the **Slide Show** palette. The **New Slide Template** button will insert a slide delimiter and a few cells with placeholder text, while the **Blank Slide** button will simply insert a slide delimiter.

You can also copy and paste a cell containing a slide delimiter. It might be hard to see the cell brackets for a cell with a slide delimiter because they are somewhat covered up with a gray color, and the cell brackets may not appear until you mouse over their position to the far right of the window, but they can be selected and then copied and pasted like other cells.

As more slide delimiter cells are added, the other slide delimiters are automatically renumbered to give users a sense of where the slides are in relation to one another and to the slide show as a whole.

Once the slide show content is complete, the slide show can be viewed by changing the screen environment. The default screen environment for new notebooks, including slide show notebooks, is **Working**. The **Working** screen environment shows all the content in the notebook at once.

Slide shows are designed to be presented by setting the screen environment to **Slide Show**. This will change the notebook so that only one slide is displayed at a time, and a navigation bar will be placed at the top of the window. The screen environment can be set by clicking the **Start Presentation** button on the **Slide Show** palette or by making the appropriate selection from the **View Environment** button in the palette. Clicking the **Start Presentation** button will not only change the screen environment to **SlideShow**, but it will also place the slide show in full screen mode. A slide show can be ended by clicking the **End Presentation** button in the palette.

The screen environment can also be changed by clicking the **Format** menu and using the **Screen Environment** submenu. This method can also be used to toggle a notebook between the **Working** and **SlideShow** screen environments without needing to open the **Slide Show** palette.

When a notebook has its screen environment set to **SlideShow**, the notebook will display a navigation bar at the top of the window. The navigation bar has buttons to jump to the first slide, go back one slide, go forward one slide and jump to the last slide. In addition, a drop-down menu lists the number of the active slide and the total number of slides. This drop-down menu can be clicked to jump to a particular slide. The content for the choices in this menu is taken from the content in the first cell in each slide.

Another feature of the **SlideShow** screen environment is that it automatically hides cell brackets, which helps the focus of the presentation be purely on the content. Cell brackets will reappear once the pointer hovers over their location at the right-hand side of the notebook; once displayed, they can be selected and manipulated as normal. As the pointer moves away from the right-hand side of the notebook, the cell brackets will automatically disappear.

The **SlideShow** screen environment will turn off highlighting for incorrectly spelled words and will completely hide the Suggestions Bar and the Image Editing toolbar. You will not be able to use these tools while the screen environment is set to **SlideShow**, but they will become available again once the screen environment is changed back to **Working**.

Creating a Slide Show from an Existing Notebook

It is common to work in a notebook and then create a slide show from the notebook as the last step. The **Slide Show** palette has a utility to automate the creation of a new slide show from an existing notebook.

To create a slide show from an existing notebook, open the notebook in Mathematica. Then, click the **Palettes** menu to select **Slide Show**, and click the **Slide Show from Current Document** button. A menu will appear with a list of the types of cells from the original notebook, like title, section and text. Choose the types of cells that should mark the start of a new slide and then click the **Insert** button. A new slide show will be created, with slide delimiter cells placed at selected positions.

A common choice for slide delimiters is section cells, but the right choice will really depend on the structure of your particular notebook.

For notebooks that do not have a systematic structure, slide breaks can be inserted manually by clicking the **Blank Slide** button in the **Slide Show** palette.

You will want to make sure your notebook has the screen environment set to **Working** before adding slide breaks; the **Working** view allows you to see all the content at once, which makes choosing the locations of the slide breaks easier.

If a cell containing a slide delimiter is deleted, then the slides that were previously separated by the delimiter will be merged into a single cell.

Another tip when adding slide breaks to an existing notebook: be sure to click at the very top of the document and add a slide break to mark the beginning of the presentation; failure to do this will result in incorrect numbering of slides in the top-right corner, with the first slide being displayed as slide 0 instead of slide 1.

Presentation Tips

Creating a Table of Contents

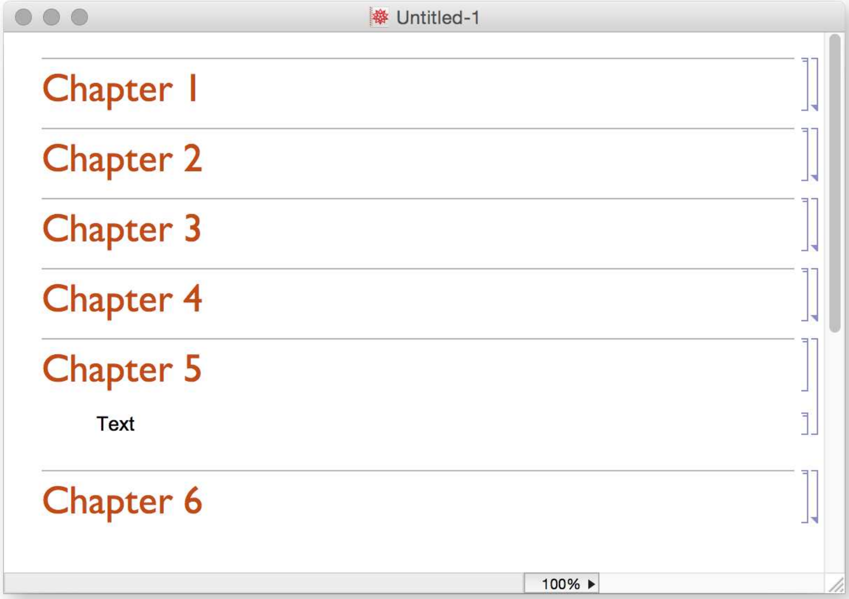
The **Slide Show** palette can be used to generate a table of contents for a slide show notebook. Once a notebook with slide delimiters is opened, clicking the **Table of Contents** button on the **Slide Show** palette will create a new window with a table of contents for the slide show notebook. This table of contents window has buttons to move forward and backward and has links to jump to specific slides or sections within slides.

In some ways, the table of contents is just like the navigation bar at the top of a slide show: both have navigation buttons and ways to jump to specific slides, with the **Slide 1**, **Slide 2**, etc. buttons in the table of contents and the **Slide 1 of *n*** drop-down menu in the navigation bar. The table of contents also creates links for section cells and other types of cells. The table of contents can be useful for driving some slide show notebooks, but for long notebooks with many section cells, using the navigation bar at the top of the slide show itself can be easier.

One good use of the table of contents: If you have your computer display set to an extended-desktop mode, so that the projector shows one screen while your computer display shows another screen, then you can have your slide show opened on the projector and the table of contents opened on your local screen. This allows you to drive the presentation by using the navigation controls in the table of contents.

Collapsing Cell Groups

To avoid having an audience read ahead or get overloaded with text, a common practice is to minimize cells and reveal the content only when discussing that specific topic. For example, double-clicking the cell bracket for a section cell group will toggle the cell group open or closed, giving users the ability to hide any information in that section until it is time to discuss that material in the presentation. Clicking the **Cell** menu, choosing **Grouping** and selecting **Open/Close Group** can be used to open or close a cell group, and that menu item also displays the relevant keyboard shortcut for that operation.



Untitled-1

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Text

Chapter 6

Since the **SlideShow** screen environment automatically hides cell brackets, it can be really useful to set your preferences to show the open/close group icon for cell groups. Open the **Preferences** menu and go to the **Interface** tab to tick the box for **Show open/close icon for cell groups**. This will put a small triangle icon to the left of all the cell groups, and clicking this icon will open or close that cell group. This setting is useful for notebooks other than slide shows, and you may want to keep it on all the time.

Arranging Content in Grids

Since a slide show in Mathematica is just a special way to view a notebook, slides can be arbitrarily long. This allows users to vertically scroll through a slide's contents, if necessary, before advancing to the next slide. This is very different from other presentation software in which slides have a fixed height and width.

However, the amount of content that can be shown at one time *will* be a fixed size, and this size will be based on the resolution of the projector used to display the slide show. As such, it can be useful to arrange content using commands like **Row**, **Column** and **Grid**. These commands can be used to display content like plots, images and text in customized layouts.

The **Row** command takes a list of elements and arranges these elements in a single row. An optional second argument is interspersed as a separator between objects, and the **Spacer** command is very useful in this regard. **Spacer** takes a numerical argument and prints a spacer that is a certain number of printer's points wide. The following command creates a row of three items, each of which is separated by a spacer of 10 points.

```
Row[{"Item 1", "Item 2", "Item 3"}, Spacer[10]]
```

```
Item 1    Item 2    Item 3
```

The **Column** command follows a similar syntax as **Row**, with a list of expressions passed as its argument, and an output that displays those expressions in a single column.

Column[{ **, "An image of a tree."}]**

An image of a tree.

Column also allows the user to specify the spacing between elements. To do this, however, the **Column** command must receive three arguments: the list of expressions, an alignment setting for the expressions and then a numerical value for the vertical spacing. The following example will create a column where the elements are center aligned, and the setting of 3 determines the vertical spacing between elements.

Column[{ **, "An image of a tree."}, Center, 3]**

An image of a tree.

Column does not use **Spacer** because **Spacer** is only used for creating horizontal space. You might be thinking, well, can **Row** just take a number for determining spacing between elements? The reason it does not is because you might want to separate the elements of a row with something *other* than a space, like a symbol, as follows: **Row[{"a","b","c"},✵]**. This command will return output as **a✵b✵c**, but that would not be possible if only blank space were allowed to be placed between row elements.

The **Grid** command can be used to create two-dimensional layouts. **Grid** takes a list of lists as its argument: the first sublist becomes the first row of the grid, the second sublist becomes the second row of the grid and so on. Then, the position of the elements within each sublist determines in which column the element is displayed. This is most easily seen with a simple example.

```
Grid[{
    {a, b, c},
    {d, e, f},
    {g, h, i}
  }]
```

```
a  b  c
d  e  f
g  h  i
```

The first list, **{a, b, c}**, becomes the first row, with **a** in the first column, **b** in the second column and **c** in the third column.

Grid has many options available for changing things like the alignment of elements, whether a frame is drawn around the grid and whether dividers are placed between individual elements. The following command creates a grid where the content is aligned to the left, a frame is drawn around the entire grid and dividers are placed between each element of the grid.

```
Grid[{
    {"Image", },
    {"Color space", "RGB"},
    {"Dimensions", "{1080,720}"}
  }, Alignment → Left, Frame → True, Dividers → All]
```

Image	
Color space	RGB
Dimensions	{1080,720}

Grid has many options to change the final appearance of a grid. You can change the background colors of elements (individually, for entire rows or columns, or for alternating rows and columns), draw borders around elements to highlight items, have elements that span multiple columns and so on. The list is too long to explain here, but the documentation for **Grid** is extensive and has many examples of how to apply these options.

It can be useful to wrap the **Style** command around a **Grid** command if the grid contains a great deal of text. This is an easy way to apply styling specifications for all the text in a grid instead of setting the style for each piece of text individually.

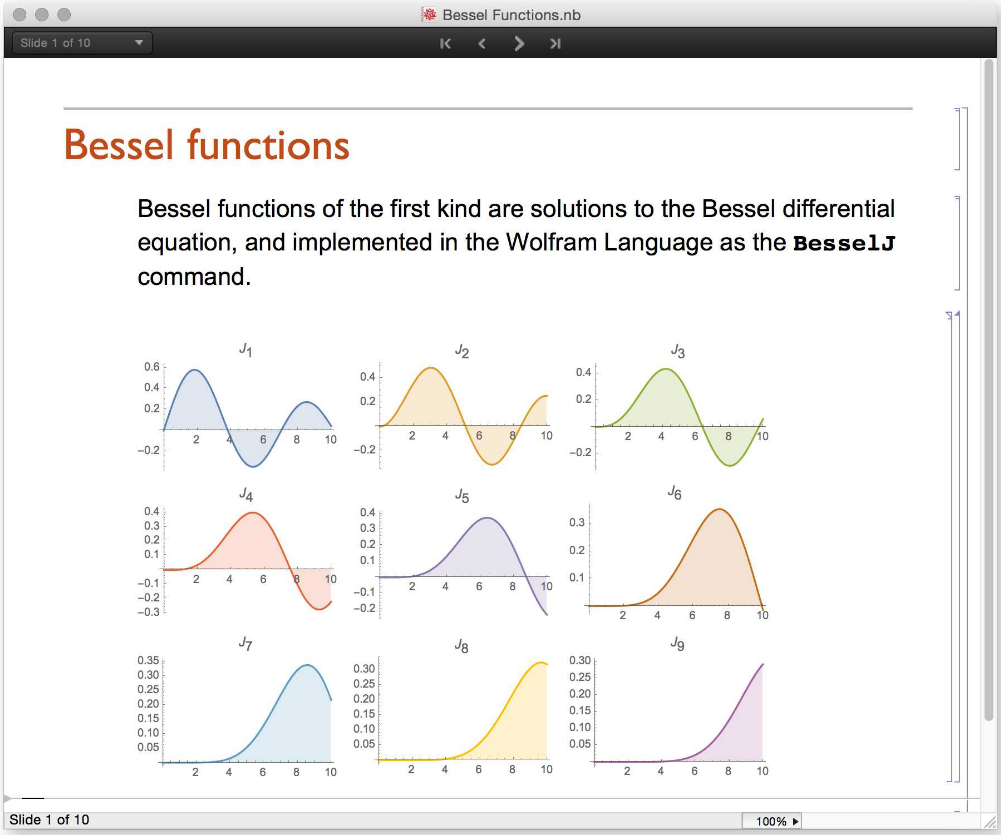
```
Style[Grid[{
    {Style["Image", Bold, Red],
```

```
    },
    {"Color space", "RGB"},
    {"Dimensions", "{1080,720}"}
  }, Alignment → Left, Frame → True, Dividers → All],
 FontFamily → "Times"]
```

Image	
Color space	RGB
Dimensions	{1080,720}

Notice how the **Style** command applied to the entire grid to change the font to Times, but the word "Image" is still bold and colored in red, thanks to the additional **Style** command used on that content.

Since the purpose of these functions is to create nice layouts, minimizing the input cells is normally appropriate; that allows the focus of the presentation to be on the content instead of the code. The following screen shot shows an example of a presentation where the input cell for the **Grid** command is collapsed; only the output is shown, resulting in a clean appearance.



Bessel functions

Bessel functions of the first kind are solutions to the Bessel differential equation, and implemented in the Wolfram Language as the **BesselJ** command.

Could you just use **Row** and **Column** instead of grid? You can; those commands can be nested, so a single **Column** command could contain multiple **Row** commands to create a two-dimensional layout. However, **Grid** is a more robust function for 2D layout and has more options for controlling how the output is displayed. If you want to create a two-dimensional layout, you should use **Grid**.

Resizing Content

When a notebook has its screen environment set to **SlideShow**, the text of the notebook is automatically enlarged to accommodate being displayed on a projector. Displayed content can also be magnified by clicking the magnification button that appears at the bottom right-hand side of a notebook. This button has choices to display the content at 50%, 75%, 100%, 125%, 150%, 200% and 300% of the original size.

The magnification settings are mostly to increase the size of text, including the text displayed in graphics, like tick labels for a plot. As the magnification setting is increased, the size of the graphics themselves will also increase, but only up to a point. If the magnification is set to a value that is quite large, the graphics may actually *decrease*, as some automatic resizing of graphics can happen to accommodate the larger text. However, any individual graphic can be resized by clicking it and then dragging its orange bounding box.

The **Slide Show** palette has a **Presentation Size** button that can be used to resize a notebook window to fit typical projector resolutions, like 800 × 600 or 1024 × 768. Choosing a setting from this button will change the window size of the active notebook to see how content will fit (or not fit) when displayed at certain resolutions.

Most content, like text and graphics, is usually perfectly fine when resized. One type of content that can get problematic, however, is tabular displays of information. If you have a grid with many rows and columns, displaying it in a window like 800 × 600 can be challenging. There are two things that can help. First, the notebook window will have vertical and horizontal scrollbars (unless they were actively turned off), so the content can be scrolled through. Second, the **Pane** command can be wrapped around content, like a **Grid** command, to restrict its output to display as a certain size. **Pane** can also have an option set to **Scrollbars → True**, which will put scrollbars around the content displayed in the pane; this allows you to scroll through the content itself without having to scroll through the notebook.

Setting Window Elements and Transition Effects

There are various slide show-related elements that can be set for notebooks, like whether the navigation toolbar is displayed, whether scrollbars are displayed and whether a magnification popup menu is available. These elements can be set by using the **Slide Show** palette. Once a notebook is open, use the **Slide Show** palette to select or deselect the desired window elements for the notebook.

The settings for these elements only work when the screen environment is set to **SlideShow**, so if you are clicking the buttons and nothing seems to be happening in the notebook, you might need to make sure the correct screen environment is set.

Another button on the **Slide Show** palette allows users to apply a transition effect as slides are changed. To choose a transition effect, open the notebook and then use the **Slide Show** palette to apply the setting. When the notebook is placed into the **SlideShow** screen environment, the effect will be shown as the navigation bar is used to move through the slide deck.

Conclusion

Slide shows have the same types of cells and capabilities as other notebooks, making Mathematica an ideal environment for presenting technical information. Slide shows can have a mixture of text, calculations, graphics and interactive models that can be changed or altered in front of an audience. Instead of being limited to preplanned and static content like other presentation software, Mathematica's live environment can support dynamic and interactive presentations, making it easier to engage audience members and respond to their questions on the fly.

Exercises

1. Create a row of text that consists of the strings "apple," "banana" and "orange," with a space of five printer's points between each element.
2. Create a table of values for *t*, where *t* goes from 0 to 5, and place the result in a column.
3. Create a list of 10 random integers ranging from 1 to 10, and place the result in a column.
4. Create a list of plots of sin(*x t*), where *x* goes from 0 to 10 and *t* goes from 1 to 6, and place the result in a column.
5. Create a 4 × 4 grid of the first 16 prime numbers, where the first four prime numbers are placed in the first row, the next four prime numbers are placed in the second row and so on. (Hint: recall that **Prime[Range[5,8]]** creates a list of the 5th through 8th prime numbers, and **Prime[Range[9,12]]** creates a list of the 9th through 12th prime numbers.)

6. Open the **Slide Show** palette and then place the cursor in the notebook that holds the results from Exercises 1 through 5. Click the **Slide Show from Current Document** button in the palette, and make the selection to insert a slide break at each input cell. Add a new slide to the beginning of the slide show that contains a section cell with the text "Chapter 5 Exercises." Use the **View Environment** button to change the setting to **SlideShow**.
7. Use the **Slide Show** palette to create a table of contents for the slide show that was created in Exercise 6.
8. Use the **View Environment** button to change the setting to **Working**. Add a section cell that says "Exercise 1" to the top of the appropriate slide, and so on for the slides related to Exercises 1 through 5.
9. Below the section cell on the first slide, add a subsection cell for your name, and then add a subsubsection cell below that for the date.
10. Use the **View Environment** button to change the setting to **SlideShow**. Scroll through the slides to ensure they are numbered 1 through 6, and rebuild the table of contents.

CHAPTER 6

Fundamentals of the Wolfram Language

Introduction

Mathematica is based on the Wolfram Language, a language designed to provide the broadest collection of commands and knowledge for a wide variety of areas. Calculations can often be written in several different styles, with advantages and disadvantages in each scenario. This chapter focuses on conventions and shortcuts in the Wolfram Language to make calculations shorter, clearer or easier to understand. There are many such shortcuts, and this chapter outlines the most useful ones for new users.

Tips for Quickly Creating Input

Mathematica's Predictive Interface, palettes and free-form input capabilities greatly minimize the opportunity to make typing mistakes. Since typos are inevitable, though, there are additional features to help minimize them further.

Mathematica uses automatic syntax coloring for recognized names of both built-in Wolfram Language commands and user-defined variables and functions. When Mathematica recognizes the name of a command, the name will be displayed in black, like the **Expand** command in the following example.

```
Expand[(x + 1)^10]
```

When Mathematica does not recognize a name, it will be displayed in bright blue. If the name is not recognized, it could be because the command name may be misspelled (such as for a Wolfram Language function) or because the symbol may not yet be defined (for user-defined functions and variables). In the following example, the misspelled command name is shown in blue, along with the symbol **x**, since it has no current definition or value associated with it.

```
Expandd[(x + 1)^10]
```

Syntax coloring is also used to give indication of missing or excess arguments. Missing arguments are indicated with red carets. The following example shows an input cell that is missing the second argument required by the **Table** command, which contains instructions on how many copies to create for a given expression.

```
Table[i‸]
```

Syntax coloring may also indicate the presence of too many arguments. The **Range** command is used to generate a list of values according to some iteration specifications. **Range** has several syntactical forms, but the most verbose takes three arguments, which are an initial value, a maximum value and a step size. For example, **Range** is used here to create a list of numbers from 2 to 12 in steps of 3.

```
Range[2, 12, 3]
```

```
{2, 5, 8, 11}
```

If an extra argument is given to **Range**, however, the excess pieces will be displayed in red text.

```
Range[2, 12, 3, 1]
```

> Syntax coloring for excess arguments works with **Range** because it does not accept any options, so anything beyond the three accepted arguments is flagged as excess information. On the other hand, you will not see arguments passed to **Plot** show up in red, because **Plot** can take options. However, if you pass excess, non-applicable arguments to **Plot**, you *will* receive an error message about options being expected in the position where the excess arguments are.

Unbalanced bracketing can trip up new users, but this mistake is mitigated through syntax coloring. Unbalanced brackets are colored in purple prior to evaluation to give a visual indication that something is wrong. Unbalanced brackets are also highlighted when an expression is unsuccessfully evaluated, as well as an error message being sent.

```
Expand[(x + 1)^10
```

When a closing bracket is typed into an expression that has a matching open bracket, the brackets will flash to indicate the pair is complete, and the brackets will also change from purple to black.

There is a menu item to enter a pair of matching parentheses, brackets or braces, which can be found by clicking the **Insert** menu, selecting **Typesetting** and then selecting the desired symbol. That menu also shows the relevant keyboard shortcut that can be used to create a pair of matching symbols.

In the latest versions of Mathematica, the process of typing any command, like the **Expand** function, is made even easier through Code Assist, which shows commands or symbol names that match what has been typed. The arrow keys can be used to navigate through the autocompletion menu, and pressing Enter will complete the selection according to the highlighted menu item. The pointer can be used to click a name on the autocompletion menu as well. If the Complete Selection window is active, it can be dismissed by pressing the Esc key.

If autocompletion really bothers you, then turn it off: just head to the **Preferences** menu, go to the **Interface** tab and uncheck the **Enable autocompletion with a popup delay** box. You might want to first change the delay from the default setting of 0 seconds to something more palatable to your personal style; that way, the popup menu does not appear quite as quickly, but it can still be there when you need it.

Once a command name is selected using Code Assist, a second menu appears with a pair of chevrons. This icon can be clicked to select a template for the command, with multiple templates available if the command accepts more than one syntactic form.

Once a command name is typed, the Make Template functionality can be invoked by clicking the **Edit** menu and selecting **Make Template** or by using the keyboard shortcut listed in that same menu.

Once selected, a template is pasted into the input cell with placeholders to be filled in. Users can advance to the next open placeholder by pressing the Tab key.

You might start with an input cell containing a statement that works, but then add to it and inadvertently introduce a syntax error. Clicking the **Edit** menu and choosing **Undo** is a lifesaver in these sorts of cases. Mathematica supports multiple levels of undo and as a consequence of this, it also visually indicates when an input cell no longer matches its corresponding output cell. When input and output cells no longer match, the contents of the output cell are grayed out to let you know that the input has been altered since the output cell was produced.

Insight into How Mathematica Computes

In addition to recognizing typos and errors in syntax, it is equally useful to know how Mathematica interprets a given input. A very simple example would be comparing two arithmetic statements to see how Mathematica computes the result.

```
2 × 3 + 4 × 5

 26
```

```
2 (3 + 4) 5

 70
```

In the first example, Mathematica computes the multiplication first and then adds the two integers, which follows mathematical conventions. In the second example, three numbers end up being multiplied together after the initial addition step. By changing the numbers to symbols, it is easy to confirm the order of the calculations.

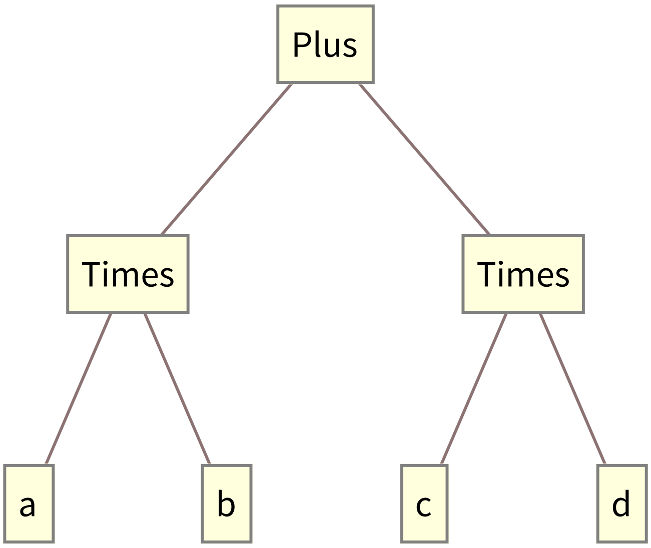
The **×** symbol printed between the numbers in the first example was not typed; it was automatically printed to show that the two numbers are being multiplied together. In Mathematica, two numbers or two symbols are multiplied together if there is either a space between them or an asterisk between them. Thus, if you want to multiply the symbol **y** by the symbol **z**, you can enter **y z** (with a space), **z y** (with a space), **y*z** or **z*y**. What you want to avoid in such a situation is putting the symbols directly together—like **yz** or **zy**—because Mathematica will treat that as an entirely new symbol instead of performing the multiplication that was expected.

The commands **FullForm** and **TreeForm** can be used to see how Mathematica symbolically represents expressions. **FullForm** prints the result as a linearly formatted command, while **TreeForm** gives a graphical view of the underlying symbolic representation.

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

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

```
TreeForm[a b + c d]
```



Mousing over **Times** or **Plus** in the **TreeForm** output shows the specific calculation at that step. Although the **Times** and **Plus** commands are not very commonly used as input, that is how mathematical operations are internally represented, following a language design principle that every expression in the Wolfram Language can be represented symbolically.

In this particular case, retracing Mathematica's symbolic representation of the expression is fairly straightforward, but **TreeForm** can be useful for examining involved expressions. Generally, if you wrap one function around another, Mathematica will perform the innermost calculation first, then work outward. In the preceding case, **Plus** is calculated first for **b** and **c**, with **Times** being calculated second.

FullForm and **TreeForm** both show alternate forms of output for calculations. **TraditionalForm** is another alternate form of output and is used more often than **FullForm** or **TreeForm**. **TraditionalForm** is a function that can be wrapped around an expression to create output that is more in line with the typesetting and formatting found in a typical math or science textbook. Later chapters will discuss the benefits of using **TraditionalForm**.

You can experiment with how **TraditionalForm** looks for various outputs by opening the **Preferences** menu, going to the **Evaluation** tab and choosing **TraditionalForm** as the setting for **Format type of new output cells**. The output cells in this book are formatted as **StandardForm** unless **TraditionalForm** is explicitly invoked, so if you do change this preference setting to try a few inputs, make sure to toggle back to **StandardForm** before proceeding.

Defining Variables

When working in Mathematica, it is common for calculations to build on one another and to use results from those calculations in subsequent calculations. For example, the simple calculation (10 + 20 − 15) might be a part of a larger calculation where that quantity is used as an exponent.

$$2^{(10+20-15)}+3^{(10+20-15)}+4^{(10+20-15)}$$

```
1 088 123 499
```

To avoid repetition or retyping each instance of (10 + 20 − 15) above, this value can be computed, stored in a variable and then referenced when needed. This can save time and also create more readable code by focusing on general forms instead of specific cases.

In the Wolfram Language, the equal sign is used to assign values to variables. In order to make an assignment, the assignment must be typed in an input cell and then evaluated. Evaluating the following command will assign the value 10 to the symbol **x**.

```
x = 10
```

```
10
```

Now whenever the symbol **x** is encountered in a command, Mathematica will automatically substitute the value 10 in its place.

$$2^{x}$$

```
1024
```

Note that the double equal sign is a completely different convention and tests equality. The following example can be interpreted as asking if 10 and 5 are equivalent and will give either True or False as the output. The double equal sign (**==**) is used to represent equations and the single equal sign (**=**) is used for storing values in variables.

```
x == 5

  False
```

Returning to the first example of this section, the value of (10 + 20 − 15) can be stored in a variable, like **a**, and then used in subsequent calculations.

```
a = 10 + 20 − 15

  15
```

Note that variables can be letters, words, Greek letters or anything not already defined in the Wolfram Language.

```
a = 10 + 20 − 15

  15
```

```
2^a + 3^a + 4^a

  1 088 123 499
```

It is possible to replace the definition for **a** with another value and then go back to the other calculation and reevaluate it to get a new result.

```
a = 12

  12
```

```
2^a + 3^a + 4^a

  17 312 753
```

Note that Mathematica notebooks do not calculate as a top-to-bottom page; this means that if a variable assignment is made in part of the notebook, and then a command that references that variable is later evaluated in a different part of the notebook—including a cell that might be at the very top of the notebook—that command will evaluate using whatever is the then-current definition for that variable. The numbered In and Out cell labels on input and output cells can help identify the order in which commands have been evaluated.

In fact, when a symbol or function is defined, it is stored not just for this document, but for the entire Mathematica session. If you define the variable **a** in one notebook and then create a new notebook, **a** can be evaluated to return its current value. (There are scoping mechanisms, like **Module**, that can be used to create local variables, but you can read about that in a later chapter.)

It is good practice to clear variable definitions and avoid using multiple definitions for a single symbol; this helps to prevent confusion and inadvertently using the wrong definition for a variable. The **Clear** command can be used to clear the value of a symbol. Using **Clear** does not produce any output, but a visual indication that it has performed its duty is that the previously recognized symbol will change in color from black to blue, since blue symbolizes an unrecognized symbol or name.

```
Clear[a]
```

While we are on the topic of good practices: It is recommended that as you create your own variables and functions, you use a lowercase letter for the beginning of their names. This will allow you (and your audience) to distinguish between your own functions and the names of Wolfram Language functions, since the latter always start with capital letters.

Creating Compound Expressions

Mathematica's interactive nature makes it easy for users to build up calculations piece by piece, which can create a series of input cells that work together to produce a final result. Sometimes having separate cells to create such building blocks is desired, as it gives the author an opportunity to intersperse textual explanations among the input. Other times, it is more useful to condense a series of related inputs into a single cell, which can be done with compound expressions.

Compound expressions can be created a couple of different ways. The first method is to place each command on a separate line in a single cell; when that cell is evaluated, Mathematica will treat each line as a separate command, evaluate the commands in order from top to bottom and create corresponding output cells. The following command assigns a value to the variable **a** and then uses **a** in a calculation; the command is entered by typing the first line, pressing Enter to add a line break, typing the second line and then evaluating by pressing Shift+Enter.

```
a = 10 + 20 − 18
2^a + 3^a + 4^a
```

```
12

17 312 753
```

When you evaluate the preceding command, you should see that the input cell is labeled with an In label, but two output cells are created: the first output cell has an Out label that matches the In label, and the second output cell has a different Out label. This is because the compound expression is in a single input cell, but each subexpression of the compound expression creates a different piece of output, which Mathematica labels differently.

The second method to create a compound expression is to use the semicolon to separate commands. The semicolon serves two purposes: it allows multiple expressions in a compound expression to reside on the same line, and it also suppresses the output of an expression. The preceding example can be entered as a compound expression in this manner.

```
a = 10 + 20 − 18; 2^a + 3^a + 4^a
```

```
17 312 753
```

In this method, the expressions are evaluated from left to right. This time, only a single output cell was created, since a semicolon was used to suppress the output of the first expression.

I know what you are thinking: Can you create a compound expression that both uses the semicolon (to suppress intermediate outputs) and places each expression on a separate line for the sake of readability? I do not think you will be surprised to hear that the answer is yes. In that case, the calculations are performed one line at a time, from top to bottom, and from left to right on each line. As a side note, the three of us authors each prefers a different style, so we could not come to a consensus to only suggest a single approach!

There is no limit to the length of compound expressions; a single compound expression in a cell could contain an entire program, with multiple variable assignments, function definitions and commands to be evaluated.

> The semicolon can be used to suppress output even when a single expression is given. For example, a cell might contain a variable assignment, like **a = 5**. Putting a semicolon at the end of that assignment prevents Mathematica from printing an output cell with the value 5, which might be viewed as redundant.

Creating Lists of Values for Representing Data

Lists are central constructs in the Wolfram Language. Many commands operate on lists directly or create output in the form of a list. Lists are used for simple operations, like creating tables of values, and complex operations, like reading in multidimensional structured data files so that further operations and analysis can be performed.

Since lists will be used for many examples in this text, a brief description of the **Table** command is necessary, since it is one of the most useful commands for creating lists. **Table** defines a pattern to create a list of values and is far more efficient than having to type each value manually. For example, a simple list containing the square of the first 10 integers can be manually entered by enclosing the values in a pair of curly braces.

$$\{1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2\}$$

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

It is much more efficient, however, to let Mathematica do the work by instructing **Table** to create a list of values. This is done by giving **Table** a pattern and an index to iterate over the pattern. Since this example concerns creating a list of squares of the first 10 integers, that is the pattern given to **Table**, and then some specifications for the index are given. In this case, the index, **i**, starts at 1 and ends at 10.

$$\text{Table}[i^2, \{i, 1, 10\}]$$

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Table has several different forms. For example, a step size can be passed as a fourth element in the second argument to further control the iteration. Giving a step size of 1 will produce an output identical to that of the preceding example.

$$\text{Table}[i^2, \{i, 1, 10, 1\}]$$

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

However, using a different step size will produce a different list of values. Here, a step size of 2 is used to generate only squares of odd integers.

```
Table[i^2, {i, 1, 10, 2}]
```

```
{1, 9, 25, 49, 81}
```

Similarly, there is a form for **Table** where only a single value for the iterator is given. In this form, Mathematica assumes that the value is a maximum and that the iteration runs from 1 to the maximum in steps of 1. The following form of the **Table** command will print a list of the squares of the first 10 integers.

```
Table[i^2, {i, 10}]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Table even has a form where it just prints *n* copies of an expression. You can see this in action if you evaluate something like **Table[π, 100]**, which will create a list of 100 instances of π.

These multiple syntactical forms are mentioned simply to familiarize readers with the **Table** command should they encounter a use of the command outside of this text. Since this book is intended for beginning users of Mathematica, however, the most specific syntax for **Table**, where iterators are given a starting value, an ending value and a step size, will generally be used for any examples.

It is important to note that **Table** can also create multidimensional lists by using a list as the pattern instead of a single element as the pattern. For example, to create a list of pairs of the form (x, x^2), where x goes from 1 to 10 in steps of 1, the following syntax is used.

```
Table[{i, i^2}, {i, 1, 10, 1}]
```

```
{{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}, {6, 36}, {7, 49}, {8, 64}, {9, 81}, {10, 100}}
```

While the examples of lists so far have concentrated on storing numeric values, a list can hold any combination of elements: expressions, formulas and even graphics. Along the same lines, variables can also hold different types of information: numbers, symbols, graphics and lists, just to name a few.

Mixing Text with Calculations

Although it is useful to intersperse text cells among input and output cells, sometimes it can be useful to work with text directly by incorporating it into commands. For example, when creating a program that alters the value of a variable, it may be useful to output statements that reference what the current value of the variable is at different points in the calculation. In order to accomplish this task, a basic understanding of strings is required.

Blocks of text used in calculations are called strings, which is a useful term to differentiate between text cells and text-related calculations. Mathematica has a variety of ways to manipulate strings as needed. Strings are denoted by placing their content between sets of quotation marks. An input cell can have strings that act just like any other Mathematica expression, and it is common to see strings used as settings for things like function options.

Strings are used within input cells to delineate between textual elements and expressions that should be evaluated, like mathematical operations. Without quotation marks, Mathematica will treat the textual elements as symbols to be evaluated. The following example shows this behavior.

```
π squared is N[π^2]
```

```
31.0063 is squared
```

Mathematica treats the words "is" and "squared" as symbols with unknown values. Since π is a known symbol, Mathematica can compute with it, and so it computes the result of multiplying **π** by **squared** by **is** by **N[π^2]**, which returns the result of **31.00627668 is squared**.

One command that operates on strings is **StringJoin**, which takes multiple strings and joins them together to create a single string as output.

```
StringJoin["The first part ", "and the second part."]
```

```
The first part and the second part.
```

Mathematica can also take an expression and turn it into a string with the command **ToString**. This approach is useful when a variable holds a numeric or symbolic value but needs to be converted to a string for the purposes of creating a label. For example, **ToString** is used to create a string that holds the value of **N[π^2]**.

```
ToString[N[π^2]]
```

```
9.8696
```

The result looks like a number, but the **FullForm** command can be used to verify that it is, in fact, a string.

```
FullForm[%]
```

```
"9.8696"
```

The quotation marks enclosing the number indicate that it is a string.

The results of **ToString** can be combined with **StringJoin** to create an expression that references the value of a variable. This can be especially useful when creating labels that automatically update as the value of the variable changes, and it is discussed in **Chapter 7: Creating Interactive Models with a Single Command**.

```
StringJoin["π squared is: ", ToString[N[π^2]]]
```

```
π squared is: 9.8696
```

StringJoin also has a shorthand form, **<>**, which can be used when joining strings inline.

```
"π squared is: " <> ToString[N[π^2]]
```

```
π squared is: 9.8696
```

Working with Units

The symbolic structure of the Wolfram Language provides an easy way to work with units, and Mathematica has access to a rich set of data to understand units and work with them seamlessly within both calculations and visualizations.

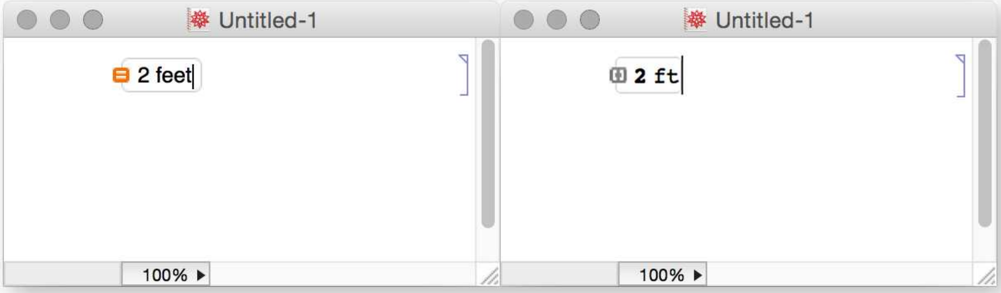
Some users start out by using symbols to denote units, such as the use of **m** and **s** in the following calculation.

```
10 m
──── 5 s
 s
```

```
50 m
```

Mathematica automatically performs a simplification for the values of **s** and **1/s** and returns a result of 50 **m**. This may seem useful at first glance, but the **m** in this case does not mean anything. Luckily, Mathematica provides a units framework for those who want to seriously work with units for computation and conversion.

Units can be discovered by using free-form input. In such cases, it is useful to use the inline version of free-form input, which is invoked through the Ctrl+= key combination when in an input cell. Once invoked, a free-form input pod will appear. Text can be typed into this pod, and then the arrow keys can be pressed or the pointer clicked to move the cursor outside the pod. Once the cursor is moved, the results from the free-form input cell will appear, and in the case of units, this means printing both the quantity and the unit inside the pod. The following screen shots show typing an expression in an inline free-form input cell and the result that is displayed after Mathematica reconciles the free-form expression into an actual quantity and unit.



2 feet

2 ft

Once a quantity is unitized, it can be used in calculations. When units of like measure are used for calculations, Mathematica will automatically resolve the results into a common unit.

2 ft ✓ + 3 m ✓

$\frac{1504}{127}$ ft

If you try to perform a calculation with unlike units, you will get an error message that the units are incompatible.

Units also have a formal specification, which uses the **Quantity** command. For those who would prefer to work with the Wolfram Language directly instead of using free-form input as a starting point, that is easily done. **Quantity** is used to associate a unit with a particular magnitude. For example, to create a quantity of 2 feet, use the following command.

```
Quantity[2, "Feet"]
```

```
2 ft
```

The preceding example that added 2 feet to 3 meters can be replicated using Wolfram Language syntax with the **Quantity** command instead of using free-form input.

```
Quantity[2, "Feet"] + Quantity[3, "Meters"]
```

```
1504
──── ft
 127
```

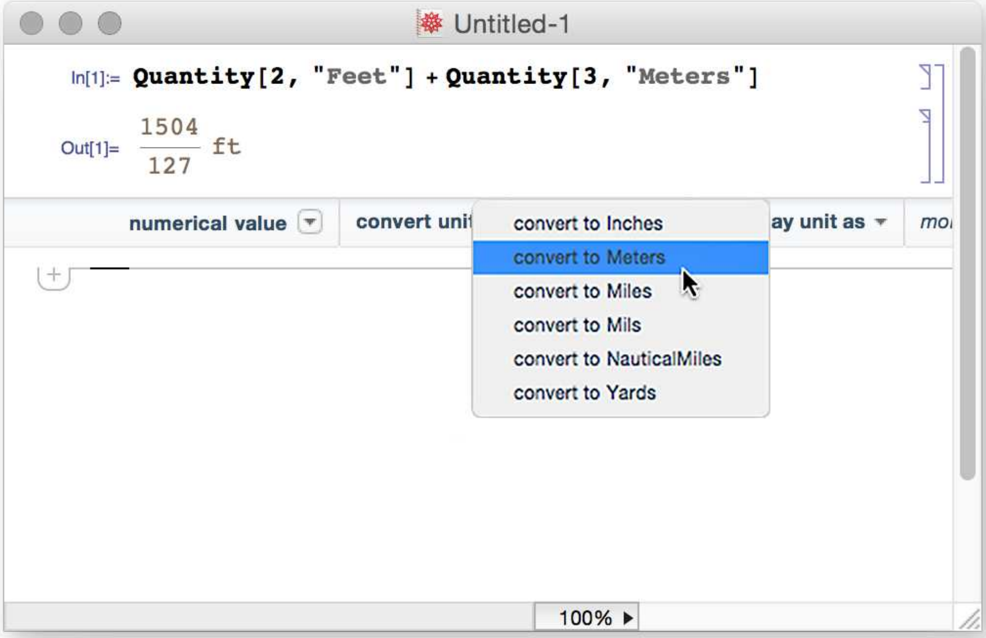
Quantities can be converted to other units with the **UnitConvert** command. The first argument to **UnitConvert** is the quantity to convert, and the second argument is the unit to convert the quantity to.

```
UnitConvert[%, "Meters"]
```

```
2256
──── m
 625
```

Units are given in their plural forms, so ask for "Meters" instead of "Meter," and so on.

When working with units, the Suggestions Bar may provide suggestions for unit conversions. For example, users can select to convert a unit and choose from suggested units, or options to convert to SI units may be presented as well. Once a selection is made, the **UnitConvert** command is used to perform the conversion.



```
In[1]:= Quantity[2, "Feet"] + Quantity[3, "Meters"]

        1504
Out[1]= ---- ft
        127
```

When Mathematica accesses data from the Wolfram Knowledgebase—such as when free-form input is used—that data uses units intelligently for single calculations. For example, you can use free-form input to evaluate "0.25 miles < height of the Empire State Building," and the data is automatically converted to the appropriate unit to make this comparison possible and to return a result. (In case you were wondering, the answer is no.)

Dates can also be specified with the **Quantity** function. For example, 7 days can be added to 2 weeks by using the units in a calculation.

```
Quantity[7, "days"] + Quantity[2, "weeks"]

 21 days
```

The Wolfram Language includes many specialized functions related to dates and time. For example, **Today** will return the current date with formatting specific to a date object.

```
Today

Sat 27 Aug 2016
```

A date object represents a date and is formatted in a special way. Dates can also be represented by and output as lists, with year, month, day, hour, minute and second values specified individually.

```
DateList[DateObject[{2016, 7, 15}]]
```

```
{2016, 7, 15, 0, 0, 0.}
```

Special functions for working with dates and times do not require explicit use of the **Quantity** function. For example, to add 7 days to the current day, **DatePlus** assumes that the appropriate unit is days.

```
DatePlus[Today, 7]
```

```
Sat 3 Sep 2016
```

However, **Quantity** can also be used with the **DatePlus** function if desired.

```
DatePlus[Today, Quantity[7, "days"]]
```

```
Sat 3 Sep 2016
```

```
DatePlus[Today, Quantity[4, "weeks"]]
```

```
Sat 24 Sep 2016
```

A useful date-computation function is **DayName**, which returns the day of the week for a certain date. The following example adds 5 months to the current day and outputs the day of the week for the resulting date.

```
DayName[DatePlus[Today, Quantity[5, "months"]]]
```

```
Friday
```

Defining Functions

Just like users can define their own variables, so too can they define their own functions. Creating custom functions can reduce the need for repetitive typing or copying and pasting, and also allows users to package a series of calculations that can be invoked with a single command.

Preceding examples have shown how to assign values to symbols. This type of assignment uses an equal sign and is known as immediate assignment, since the current value on the right-hand side of the equation is immediately assigned to the symbol on the left-hand side of the equation.

There is another type of assignment, known as delayed assignment. Delayed assignments are created by using the **:=** symbol, which is entered as a colon followed by an equal sign. With delayed assignment, a pattern is defined on the left-hand side, and values that match the pattern are substituted into the right-hand side, but only at the time that the variables are passed to the function in question. Delayed assignment is recommended for user-defined functions, and it is the convention that this book will follow in all of its examples.

> If you are really curious about the details of immediate versus delayed assignment, then you can read the entire tutorial devoted to this very topic in the documentation. We will explain how to use the documentation before this first set of chapters is over.

To define a function, a function name (or symbol) has to be given, along with square brackets to surround its arguments. Function arguments need to be denoted with symbols that end with underscore characters, like this: **x_**. Then the delayed assignment symbol is given, and finally, the operation (or operations) the function needs to perform is placed on the right-hand side. For example, to define a function **f** that takes a single argument **x** and returns the result of squaring that argument, the function definition is as follows.

```
f[x_] := x^2
```

The underscore character represents a pattern and allows Mathematica to match any expression that is given as **x**. This allows **f** to be used with any type of value; there is no need to specify a type for this argument as is the case with some other languages.

Now that it is defined, the function **f** can be used with specific values, and the syntax is comparable to any built-in Mathematica function. For example, an integer value can be passed to calculate its square.

```
f[2]
```

```
4
```

But since the **x_** stands for any expression, other types of arguments can be passed to **f** as well, whether they are symbolic, real numbers or even lists of values.

```
f[π]
```

π^2

```
f[1.2345]
```

```
1.52399
```

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

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

User-defined functions can also have multiple arguments. The following example defines a function that accepts two inputs, each of which can be any type of expression.

```
h[a_, b_] := a^b
```

This function can also be used just like any built-in function.

```
h[10, 10]
```

```
10 000 000 000
```

And just like any other function, if you pass the wrong number of elements, then the function will not evaluate, and the result will be the unevaluated expression. You can try this on your own if you evaluate the function definition for **h** as shown and then try to evaluate **h[10, 10, 10]**.

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[x, a, f, h]
```

Conclusion

The Wolfram Language is designed to cover a wide range of subject areas and is unique for its paradigm of knowledge-based computation. However, only an understanding of the basics is necessary for users to efficiently and effectively use many parts of the system. Subsequent chapters will build on the material outlined in this chapter to apply these techniques to other areas, such as graphics, working with data and programming.

Exercises

1. Assign a value of 7 to a variable named **t**.
2. Now that **t** is defined, evaluate $3\,t^2 + 2\,t + 1$.
3. Create a list of values of the form $t + 1$, where t ranges from 1 to 10.
4. Write a three-line compound expression and set the variable **v** equal to 6.5 on line 1, set the variable **w** equal to 7.1 on line 2 and set the variable **answer** to be the solution for $w^2 - v^2$ on line 3. In this compound expression, be sure to suppress the output of lines 1 and 2 so that only the final result is shown.
5. Write a compound expression where you define a function **f** that takes a single variable **z** as input and returns **Sin[z]**. The definition for **f** should be placed on line 1 of the compound expression. On line 2 of the compound expression, create a variable named **myResponse** and assign to it the value of evaluating the function **f** at 3. On line 3 of the compound expression, numerically approximate the value of **myResponse** to five digits. Suppress the output from line 1, but show the output for lines 2 and 3 of the compound expression.
6. Write a two-line compound expression. The first line should use the **Table** command and the function **f** defined in Exercise 5 to create a list of the results of applying **f** to the first 10 integers. The second line should numerically approximate the result from line 1 to two digits of precision. The output of line 1 should be suppressed, and the **%** operator should be used for the command in line 2.
7. Use free-form input to convert 20 ounces to kilograms.
8. Use Wolfram Language commands to convert 20 ounces to kilograms.
9. A Japanese elevator has a weight limit of 600 kg. Convert this quantity into pounds using either free-form input or Wolfram Language commands directly.
10. Use the Suggestions Bar with the previous output to request the magnitude of the result; this will discard the associated unit and return just the numeric value, which might be useful for further, non-unit calculations. (Reminder: The cursor must be directly below an output cell for the Suggestions Bar to appear. If your cursor is in the right place but the Suggestions Bar is not displayed, it may have been turned off. You can turn it back on by clicking the arrow icon at the far right of an output cell.)

CHAPTER 7

Creating Interactive Models with a Single Command

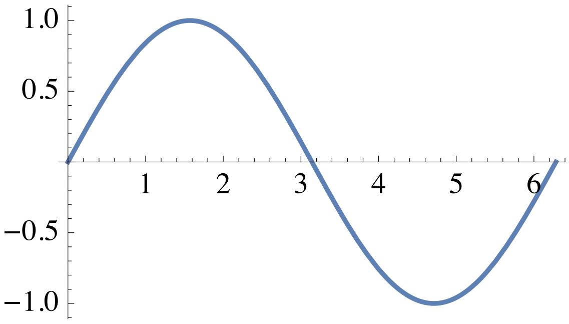
Introduction

One of the most exciting features of Mathematica is the ability to create interactive models with a single command called **Manipulate**. The core idea of **Manipulate** is very simple: wrap it around an existing expression and introduce some parameters; Mathematica does the rest in terms of creating a useful interface for exploring what happens when those parameters are manipulated. This single command is a powerful tool for learning and teaching about phenomena and for creating models and simulations to support research activities.

Building a First Model

A common workflow is to start with something static, such as a plot, and then to make it interactive using **Manipulate**. Take the following plot as an example, which plots $\sin(x)$ from 0 to $2\,\pi$.

```
Plot[Sin[x], {x, 0, 2 π}]
```



The goal may be to compare the curve of $\sin(x)$ with the curve of $\sin(2\,x)$, the curve of $\sin(3\,x)$ and so on. In other words, to examine the behavior of $\sin(f\,x)$ when f is varied among a large quantity of numbers. **Manipulate** provides an easy way to perform this investigation by constructing an interactive model to explore this behavior.

To begin, it is important to know that using **Manipulate** requires three components:

1. **Manipulate** command
2. Expression to manipulate by changing certain parameters
3. Parameter specifications

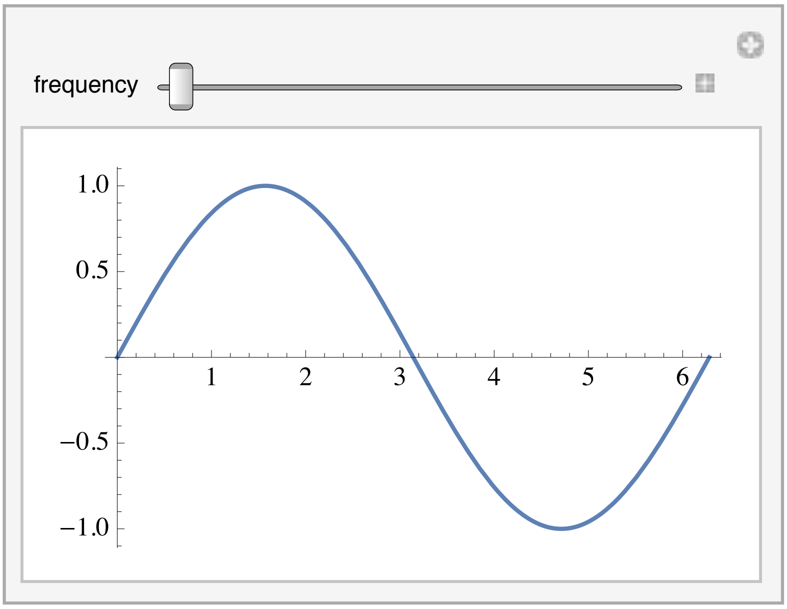
An easy way to keep track of these components is to write commands involving **Manipulate** as follows.

Manipulate[
expression to manipulate,
parameter specifications]

This approach keeps each component on a separate line and provides an easy way to keep track of each separate component.

For the example introduced above, the **Manipulate** command might be as follows.

```
Manipulate[
  Plot[Sin[frequency * x], {x, 0, 2 π}],
  {frequency, 1, 5}]
```



The result is an interactive model with a slider bar that can be clicked and dragged to interactively explore what happens as the value of **frequency** is changed. This specific model can be quite useful for explaining concepts of periodicity and frequency and was built from a single line of code—pretty impressive, and a representative example of the power of **Manipulate**.

The plus icon immediately to the right of the slider bar can be clicked to open an Animation Controls menu for that controller. Animation Controls can be used to animate the model, incrementally step through different values for the parameter or assign a particular value to the parameter through the use of an input field.

You do not have to follow this multiline convention; you could put a **Manipulate** command on a single line, like:

```
Manipulate[Plot[Sin[f*x],{x,0,2π}],{f,1,5}]
```

To some, it reads more cleanly to have the command on one line; to others, having the components on different lines makes the code more readable. Choose the style that makes the most sense to you.

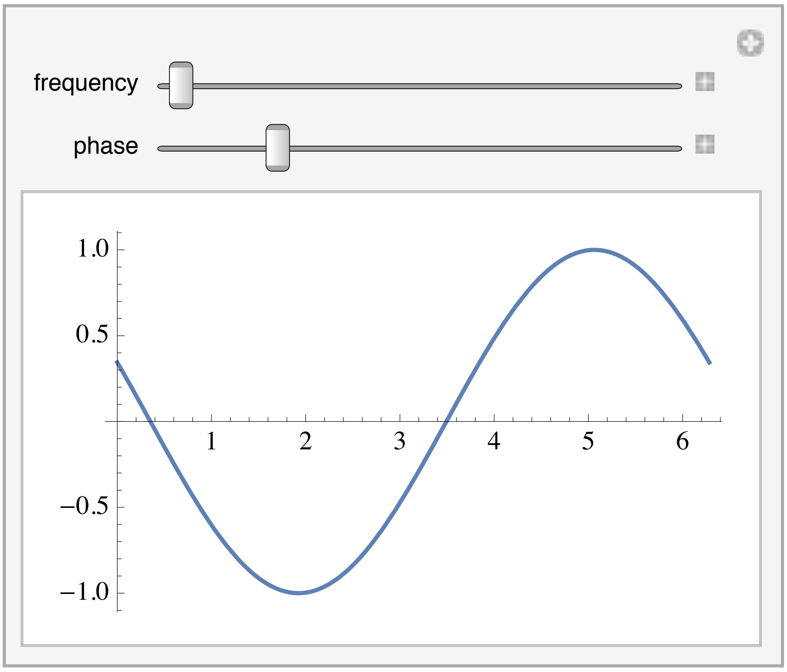
Building Models with Multiple Controls

Manipulate can be used to construct interactive models with an arbitrary number of controllers. To control a model with multiple parameters, simply introduce the new parameters and their corresponding parameter specifications. With two parameters, the basic outline changes to the following.

Manipulate[
  expression to manipulate,
  first parameter specifications,
  second parameter specifications]

The previous example can be expanded by introducing a new parameter, **phase**, along with a range of values for the minimum and maximum of this new parameter. Mathematica will automatically create separate controllers for each parameter and label them accordingly.

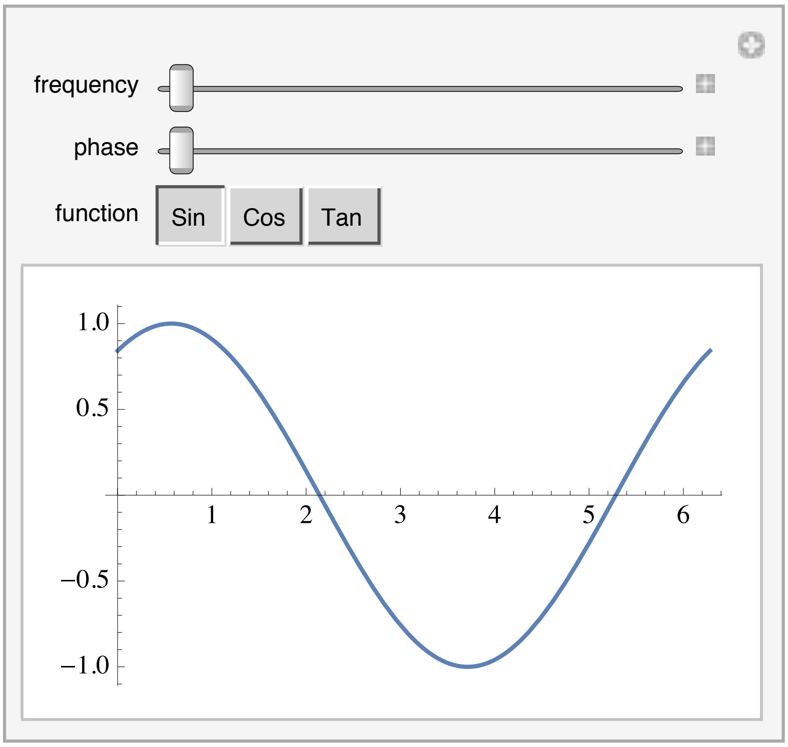
```
Manipulate[
  Plot[Sin[frequency*x + phase], {x, 0, 2 π}],
  {frequency, 1, 5},
  {phase, 1, 10}]
```



Manipulate can be used to give parameters a list of discrete choices instead of a continuous range for their values. For example, the **Sin** command can be replaced by a new parameter called **function**, and then a list of choices can be given as the parameter specification for **function**.

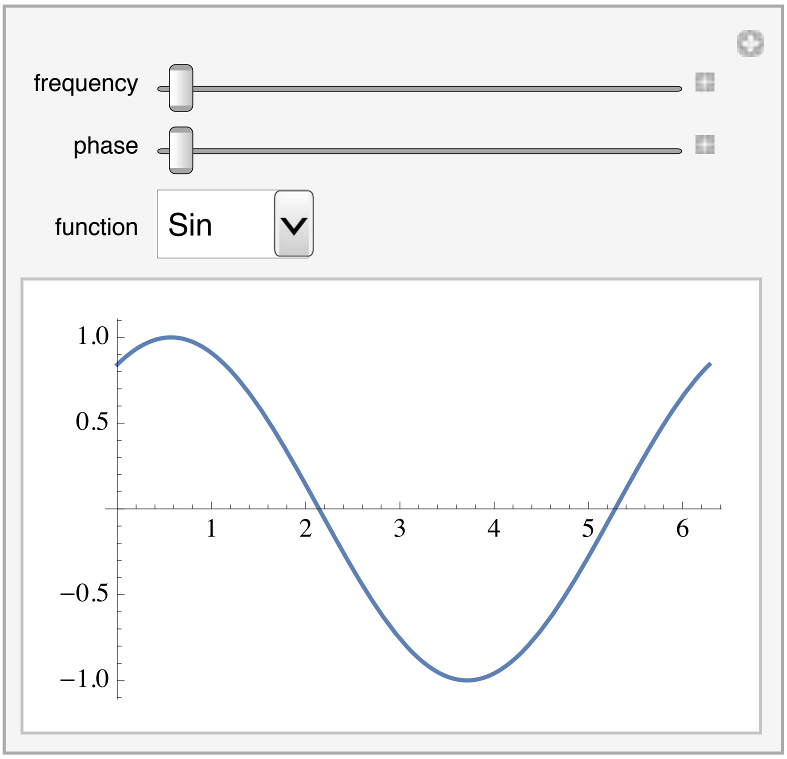
Since curly braces are used to denote lists, this will create a parameter specification with a nested list: the outermost list contains the parameter name and the specification, and the specification itself is a list that contains discrete choices—in this case, **Sin**, **Cos** and **Tan**—for the parameter to assume.

```
Manipulate[
  Plot[function[frequency*x + phase], {x, 0, 2 π}],
  {frequency, 1, 5},
  {phase, 1, 10},
  {function, {Sin, Cos, Tan}}]
```



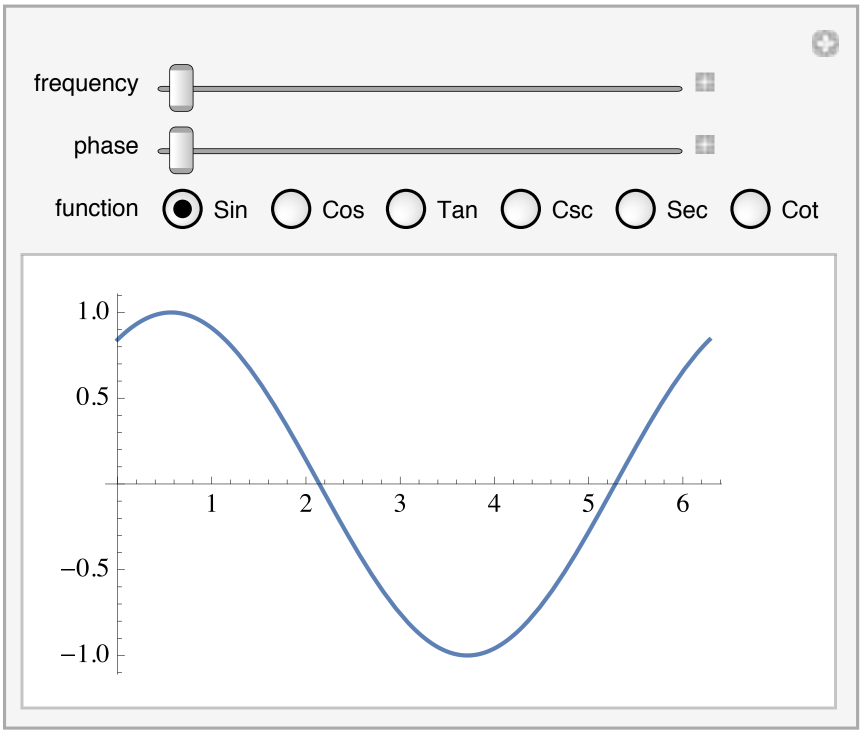
Mathematica has built-in heuristics to select appropriate controller types based on the parameter specifications that have been given. For example, giving a long list of choices causes **Manipulate** to display the controller as a drop-down menu instead of a list of buttons.

```
Manipulate[
  Plot[function[frequency*x + phase], {x, 0, 2 π}],
  {frequency, 1, 5},
  {phase, 1, 10},
  {function, {Sin, Cos, Tan, Csc, Sec, Cot}}]
```



Like most everything in Mathematica, the output from commands can be customized through the use of options. If you want to force Mathematica to use a particular control type, the **ControlType** option can be used with values such as **Setter**, **Slider** and **RadioButtonBar**. For example, if you add **ControlType→ RadioButtonBar** between the two closing curly braces in the last parameter specification in the preceding example, Mathematica will create a row of radio buttons to set the value of **function** instead of giving you a drop-down menu.

```
Manipulate[
  Plot[function[frequency*x + phase], {x, 0, 2 π}],
  {frequency, 1, 5},
  {phase, 1, 10},
  {function, {Sin, Cos, Tan, Csc, Sec, Cot}, ControlType → RadioButtonBar}]
```



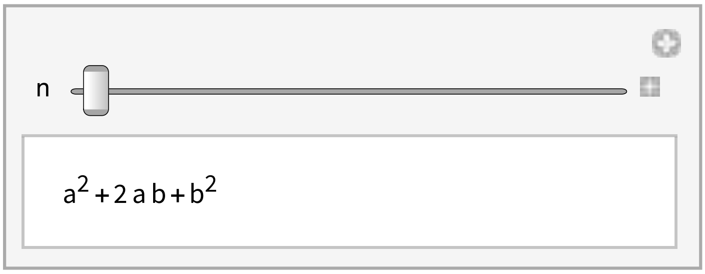
frequency

phase

function ● Sin ○ Cos ○ Tan ○ Csc ○ Sec ○ Cot

The **Manipulate** command is not restricted to graphical manipulation and can be used with any Mathematica expression. For example, symbolic expressions can be manipulated just as easily as graphical expressions.

```
Manipulate[
  Expand[(a + b)^n],
  {n, 2, 10, 1}]
```



n

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

In the preceding example, the range **{n, 2, 10, 1}** was used to restrict the values of **n** to be from 1 to 10 in increments of 1, since exponentiation is not defined for noninteger values.

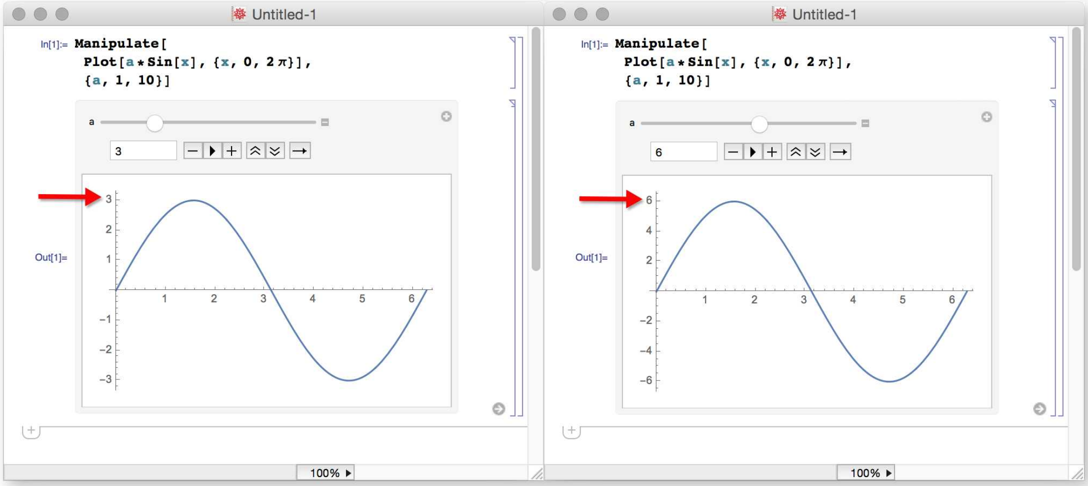
Some Tips for Creating Useful Models

The default results returned by **Manipulate** are generally very useful and do not require any special customization. However, there are a few important points to be aware of, so we will discuss them here in order to help you avoid potential problems.

The Importance of PlotRange

The default behavior of commands like **Plot** is to automatically choose an appropriate viewing window unless a specific range is given. This means that when **Manipulate** is used to change the value of a parameter, which has a resulting effect of changing the appearance of a plot, the plot will immediately be redrawn with a new viewing window. The end result is that manipulating a parameter may appear to change the axes for the plot rather than the plot itself.

The following screen shot shows an example of this behavior. On the left, the value of the parameter **a** is set to 3, and the plot axes are automatically chosen to fully display the behavior of the plot. On the right, the value of **a** is set to 6, and the plot is drawn accordingly.



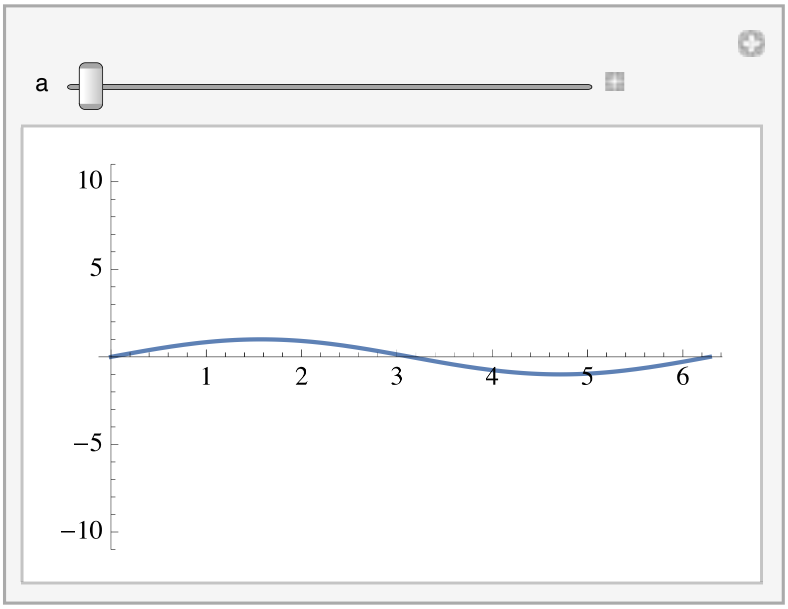
```
In[1]:= Manipulate[
          Plot[a*Sin[x], {x, 0, 2π}],
          {a, 1, 10}]
```

```
In[1]:= Manipulate[
          Plot[a*Sin[x], {x, 0, 2π}],
          {a, 1, 10}]
```

This behavior can be avoided by specifying an explicit range to plot over. This can be accomplished by using the **PlotRange** option for the **Plot** command, which forces the plot to be drawn with the specific plot range the user provides. **PlotRange** takes a list as its argument (remember: lists are enclosed by curly braces), where the first element of the list is the minimum value for the plot range, and the second element of the list is the maximum value for the plot range.

The arrow (→) in the **PlotRange** option is constructed by using the hyphen (**-**) and the greater-than symbol (**>**), which Mathematica then formats into the arrow.

```
Manipulate[
 Plot[a * Sin[x], {x, 0, 2 π},
  PlotRange → {−11, 11}],
 {a, 1, 10}]
```



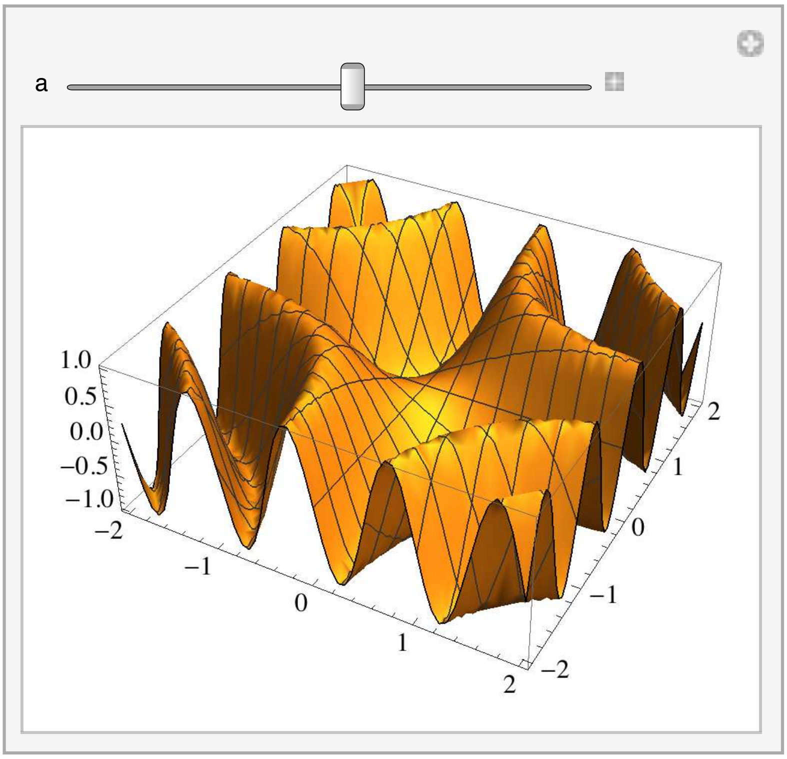
In the preceding example, the **Plot** function now spans two lines as a result of adding the **PlotRange** option. Notice how the **PlotRange** line is nicely indented to show that it is part of the **Plot** statement, while the list with the amplitude parameter is indented to show that it is an argument that belongs with the **Manipulate** command. You should experiment with deleting and adding extra line breaks like this based on your preference for how the code looks.

Since the plot range is now fixed, adjusting **a** appears to stretch or flatten the plot, which may be the desired behavior for this model to show.

Optimizing Performance for 3D Graphics

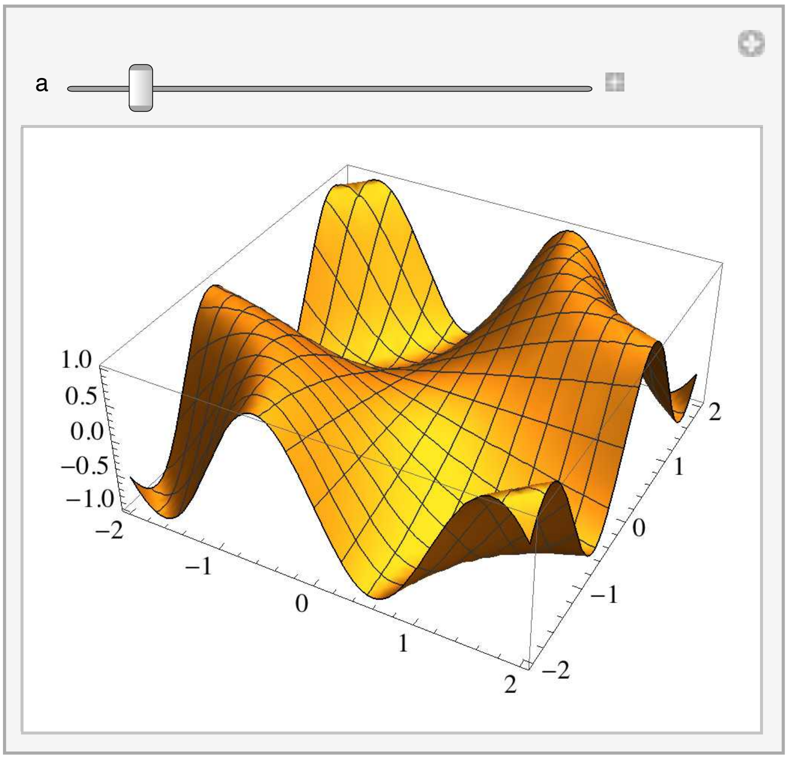
When 3D graphics are manipulated with controllers like slider bars, they may appear jagged while the controllers are being moved, and then smooth again when the controllers are released. The following example shows this behavior in action.

```
Manipulate[
  Plot3D[Sin[a x y], {x, −2, 2}, {y, −2, 2}],
  {a, 1, 5}]
```



Mathematica's default behavior is to optimize the performance while the controller is being moved, and then to optimize the appearance once the controller is released. This allows a fast interaction between users and the controllers, and nicely rendered results when finished. However, if rendering is more important than fast interaction, then the use of options like **PerformanceGoal** can be handy.

```
Manipulate[
  Plot3D[Sin[a x y], {x, −2, 2}, {y, −2, 2}, PerformanceGoal → "Quality"],
  {a, 1, 5}]
```



Now when the slider bar is dragged, the appearance of the plot remains smooth. The tradeoff is that the slider bar may be slightly less responsive than it was in the preceding example.

Labeling Controllers and Displaying Current Values

Manipulate creates a unique controller for each parameter that can be manipulated. By default, Mathematica will use the name of the parameter when it labels its corresponding controller, so if the parameter is named **frequency**, then "frequency" is what the label for the controller will say.

There are times, though, when it is desirable to name the parameter one thing and to have the controller label display something else. A user might do this to save on keystrokes: use a short variable name, like **f**, for a parameter, but then label the control for **f** with something different, like "frequency," to improve readability of the model.

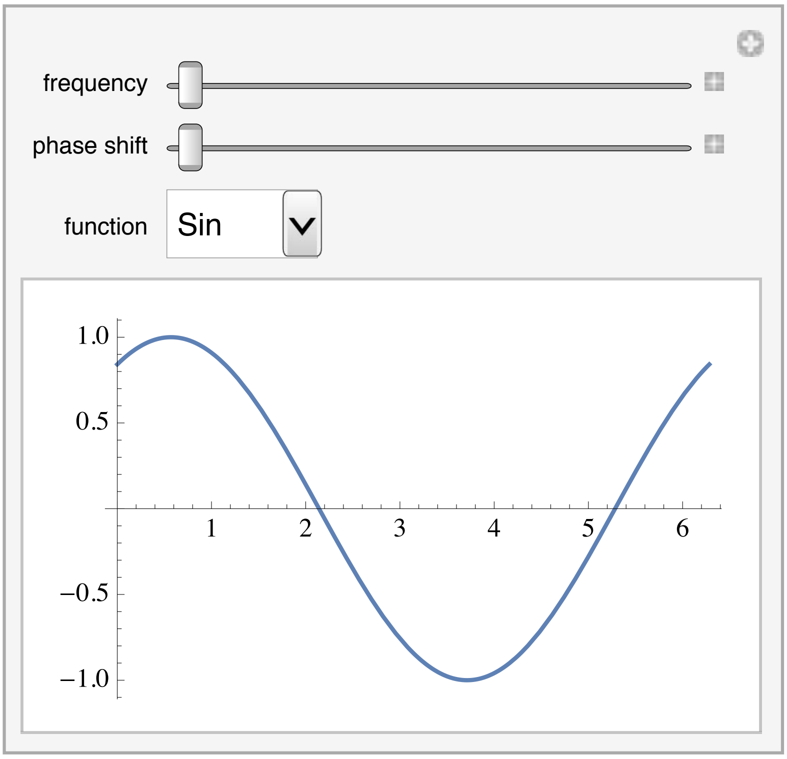
Labeling is also useful in situations where the label is comprised of multiple words. Since a parameter in Mathematica has to be a single symbol without spaces, a parameter cannot be named something like **phase shift**. However, a parameter could be named **ps**, and then the label corresponding to the controller for **ps** could be given as "phase shift."

To label a controller, a set of nested braces is used in the parameter specification, and values are entered as follows.

```
{{parameter, initial value, "parameter label"}, minimum, maximum}
```

Using this idea, an example from earlier in this chapter could be modified to use different parameter names and labels for each of the controllers.

```
Manipulate[
  Plot[fn[f*x + ps], {x, 0, 2 π}],
  {{f, 1, "frequency"}, 1, 5},
  {{ps, 1, "phase shift"}, 1, 10},
  {{fn, Sin, "function"}, {Sin, Cos, Tan, Csc, Sec, Cot}}]
```

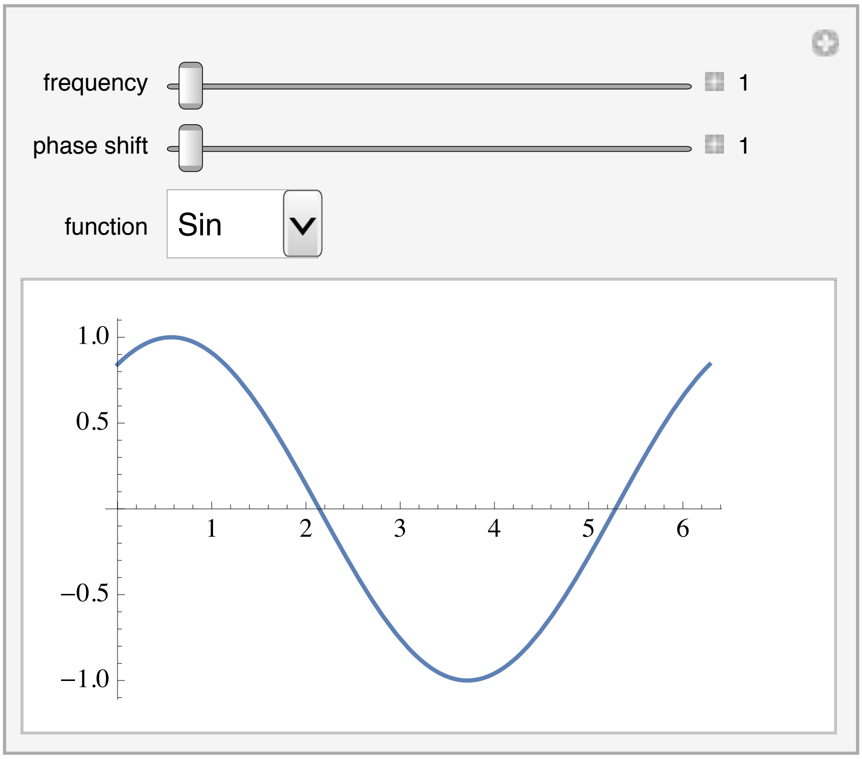


The labels appear to the left of each controller, and the actual names of the parameters—in this case, **f**, **ps** and **fn**—are not visible in the output at all.

You do not have to make the initial value of the parameter the same as the lower bound of the controller; you can set the initial value to be, say, 3 for a controller that ranges from 1 to 5.

Another useful option to set for the controllers is **Appearance→"Labeled"**, which will display the current value of the parameter to the right of its Animation Controls button. (There is no need to set this option for the **fn** parameter, since the function name is already displayed *within* the controller as part of the buttons.)

```
Manipulate[
  Plot[fn[f * x + ps], {x, 0, 2 π}],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"},
  {{ps, 1, "phase shift"}, 1, 10, Appearance → "Labeled"},
  {{fn, Sin, "function"}, {Sin, Cos, Tan, Csc, Sec, Cot}}]
```

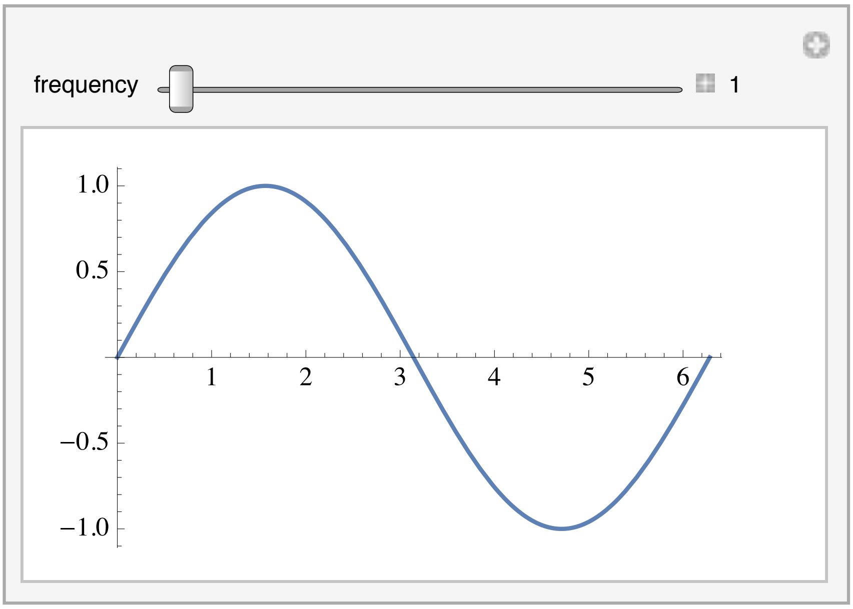


Creating an Interactive Plot Label

While labeling individual controllers in a **Manipulate** can be useful, it can also be desirable to create an interactive plot label that takes all of these labels into consideration and prints a single expression, like the equation of the function being graphed.

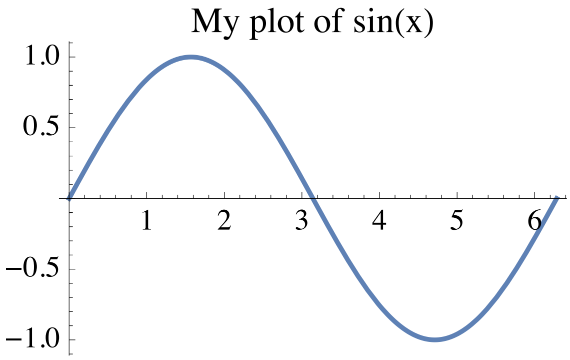
The following example plots **Sin[f*x]**, where **f** is a manipulable parameter. The controller for **f** uses the **Appearance→"Labeled"** option setting to print its values to the right of the controller, which is helpful, but the user is still required to examine the code to ascertain exactly what function is being plotted.

```
Manipulate[
  Plot[Sin[f*x], {x, 0, 2 π}],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"}]
```



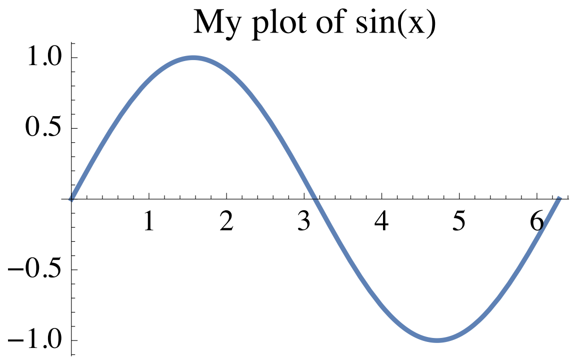
Creating an interactive plot label can make the function being plotted more obvious. First, a quick explanation of the **PlotLabel** option is necessary. **PlotLabel** is an option for **Plot** (and other plotting commands) that prints a label at the top of the plot. **PlotLabel** expects a string to be passed as its option setting. A string in Mathematica is enclosed with quotation marks.

```
Plot[Sin[x], {x, 0, 2 π}, PlotLabel → "My plot of sin(x)"]
```



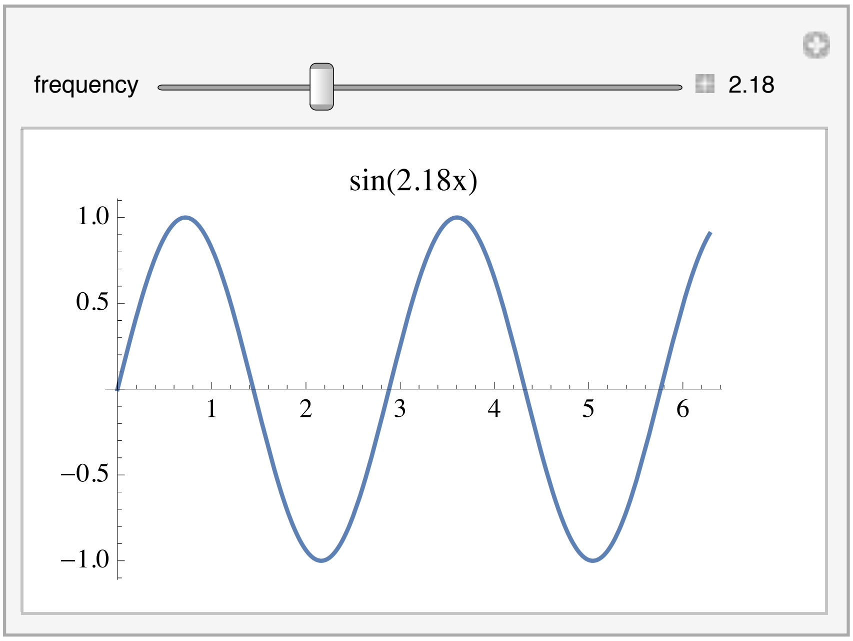
Strings can also be joined together with the **<>** operator. This is useful when constructing a single string from multiple pieces of information that might be coming from different places.

```
Plot[Sin[x], {x, 0, 2 π}, PlotLabel → "My plot of " <> "sin(x)"]
```



To create an interactive plot label, the **PlotLabel** option has to be hooked up to the same parameters as the **Manipulate** command. By using the same parameter symbol name, when the plot is manipulated, its plot label will simultaneously update. However, the **PlotLabel** option expects a string, and parameters in **Manipulate** commands are generally not strings. A trick is to use the **ToString** command to convert an expression to a string, and then to use **<>** to hook multiple strings together.

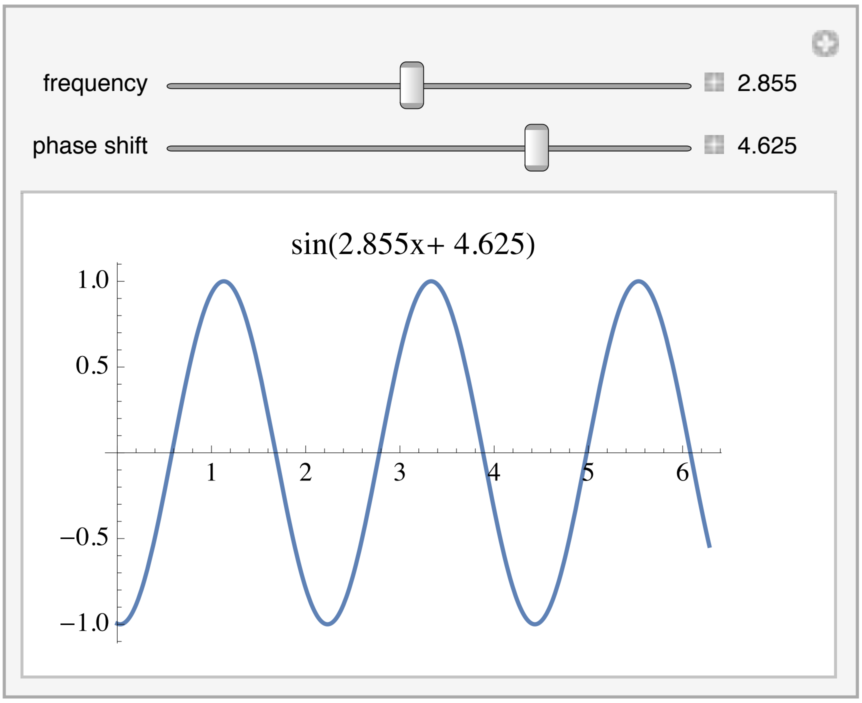
```
Manipulate[
  Plot[Sin[f * x], {x, 0, 2 π}, PlotLabel → "sin(" <> ToString[f] <> "x)"],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"}]
```



frequency 2.18

The same approach can be used to create an interactive plot label that updates based on the values of several manipulable parameters.

```
Manipulate[
  Plot[Sin[f*x + ps], {x, 0, 2 π},
    PlotLabel → "sin(" <> ToString[f] <> "x+ " <> ToString[ps] <> ")"],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"},
  {{ps, 1, "phase shift"}, 1, 6, Appearance → "Labeled"}]
```

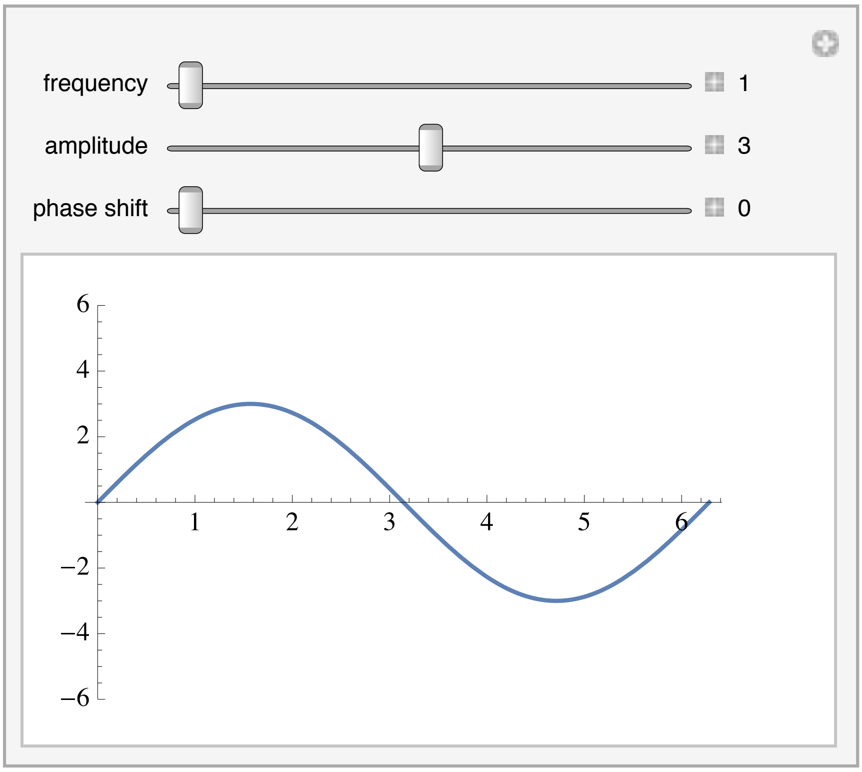


The <> symbol is actually shorthand for a command named **StringJoin**, but since it is used so often, the symbolic shorthand form exists. There are other commands like this in Mathematica with symbolic shorthand forms, so if you see a symbol you do not recognize, you can search the documentation to find the corresponding formal command name.

Hiding Code

Manipulate commands especially lend themselves to hidden code because the input that created the model is usually not as important as the model itself. Like other situations where hidden input is desirable, simply double-click the cell bracket containing the output (the interactive model created by **Manipulate**) to hide the corresponding input.

```
Manipulate[
  Plot[a * Sin[f * x + ps], {x, 0, 2 π}, PlotRange → 6],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"},
  {{a, 3, "amplitude"}, 1, 5, Appearance → "Labeled"},
  {{ps, 0, "phase shift"}, 0, 2 π, Appearance → "Labeled"}]
```



frequency 1

amplitude 3

phase shift 0

Obfuscating code can be taken one step further by deleting the input entirely or by copying and pasting just the output (the interactive model) into a separate notebook. In many cases, the interactive model will still function when the notebook is opened, although it will not be operational if it references a function or data that is no longer available at the time of future reuse. The next section outlines ways to make **Manipulate** statements self-contained so that they include all necessary definitions.

Double-clicking the output to hide the input is much more common than deleting it. If you keep the input intact, you can add minor edits later quite easily. If the input is deleted, you would likely have to start over and recreate the **Manipulate** statement.

Remembering User-Defined Functions

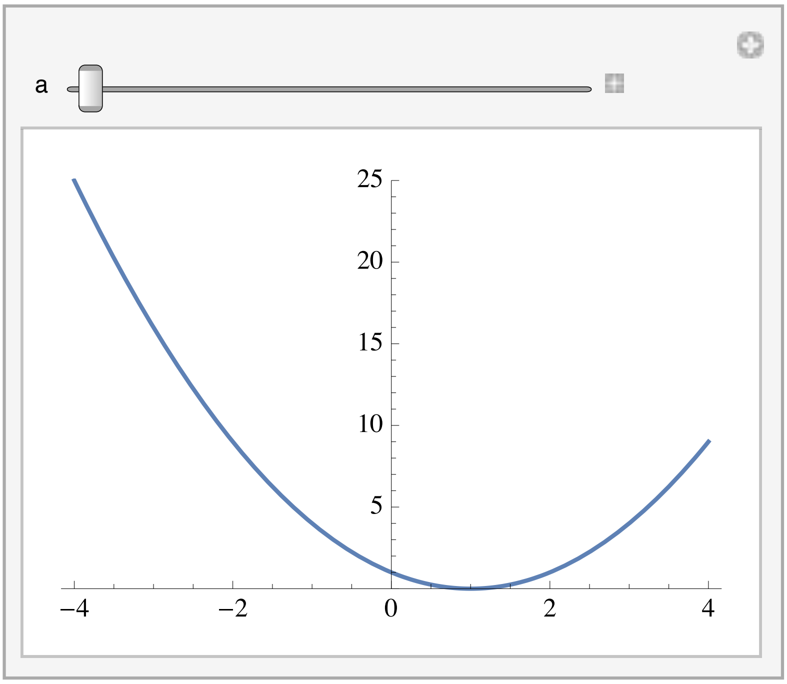
While the examples so far have utilized Wolfram Language functions, the **Manipulate** command can be used with any expression, including user-defined functions. Once a function is defined, then **Manipulate** can operate on it. As an example, the function **f[x]** is defined as follows.

```
f[x_] := 2 x^2 + 2 x + 1
```

Remember, you can typeset an exponent using the Ctrl+6 keyboard shortcut or by using one of the palettes to create a typesetting template.

And now this function can be used with **Manipulate**.

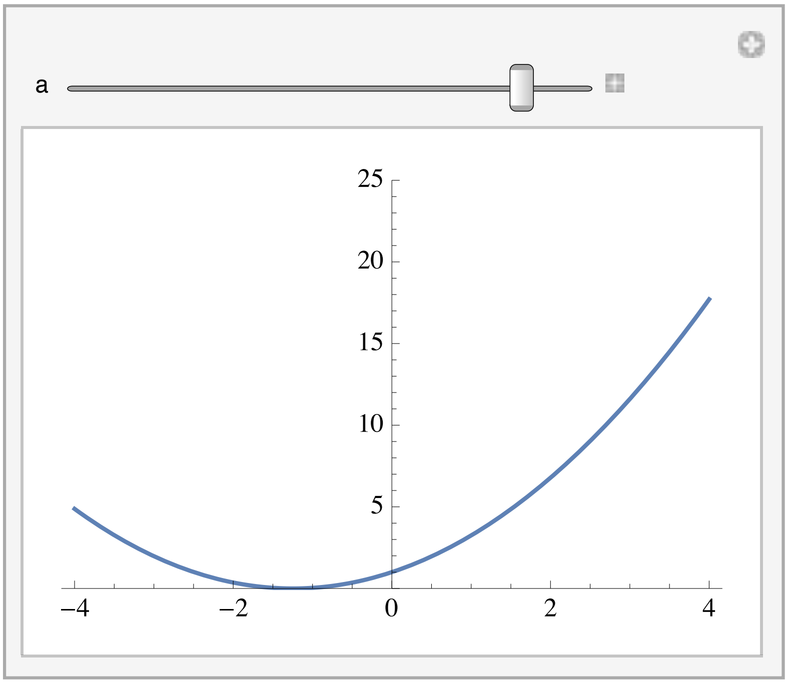
```
Manipulate[
 Plot[f[a * x], {x, −4, 4}, PlotRange → {0, 25}],
 {a, −1, 1}]
```



If the output cell of the above expression—the interactive model created by **Manipulate**—was copied to a new notebook, and the Mathematica session was ended, and the new notebook was reopened later, then the interactive model would no longer function because Mathematica would not remember the definition of the function **f**.

There are two different strategies that can be employed to use **Manipulate** with user-defined functions. The first is to use **Initialization**, which allows the definition of symbols to be performed when the **Manipulate** command is evaluated. The syntax for **Initialization** uses **RuleDelayed**, which is more commonly input using the escape sequence Esc :> Esc and automatically converted to :→ as its shorthand form. In the following example, the **Initialization** option setting is placed on its own line for the sake of clarity.

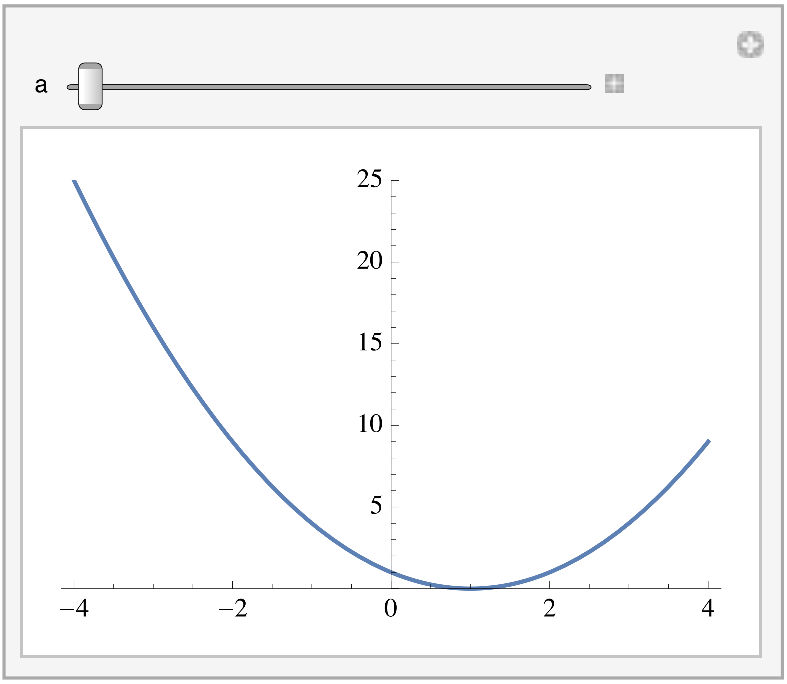
```
Manipulate[
  Plot[f[a * x], {x, −4, 4}, PlotRange → {0, 25}],
  {a, −1, 1},
  Initialization :→ (f[x_] := 2 x^2 + 2 x + 1)]
```



Now if the output (or the input and output) is copied into a new document, saved and reopened at a later time, the **Manipulate** model will work. The function definition for **f** in the **Initialization** option is the initial state for the **Manipulate** function. If the function **f** is redefined in another section of the notebook, that new definition will apply to the **Manipulate** object as well.

A second approach is to use the **SaveDefinitions** option, which will save the current definitions for every symbol in the **Manipulate** command; these saved definitions will travel with the interactive model, even when copied and pasted to a new notebook.

```
Manipulate[
  Plot[f[a * x], {x, −4, 4}, PlotRange → {0, 25}],
  {a, −1, 1},
  SaveDefinitions → True]
```



As before, if the output is now copied into a new document, saved and reopened at a later time, the **Manipulate** model will work; it "remembers" the definition for the function **f** since it was told to save the definitions.

In general, using **Initialization** is good if you might want the recipient of your document to see the underlying commands used to construct your interactive model. If you would prefer to hide that information from your audience, then **SaveDefinitions** can be a better approach.

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[f]
```

Conclusion

The use of **Manipulate** to communicate ideas is quite popular with Mathematica users, since the results can be immediately understood without the audience having to understand or even see any Wolfram Language commands. A good understanding and generous use of **Manipulate** can go a long way in explaining ideas, illustrating concepts and simulating phenomena—and all with a single command!

Exercises

1. Create a **Manipulate** statement to vary $x^2 + 1$, where x is an integer ranging from 1 to 10.

2. Similarly to Exercise 1, create a **Manipulate** statement to produce a list of values of the form $\{x, x^2 + 1, x^3 + 1\}$, where x is an integer ranging from 1 to 10.

3. Create a **Manipulate** statement to show the list of $\{x, x^2 + 1, x^3 + 1\}$ and then add a fourth element to the list that is an expression that answers whether $x^2 > 2x + 1$. As before, use the same integer range of 1 to 10 for the variable x.

4. Use the Wolfram Language to create a plot of $x^2 + 3x - 1$ over the domain from -5 to 5.

5. Use **Manipulate** to visualize the behavior of $x^2 + 3x - 1$ when a constant c is used to multiply x^2, and where c ranges from 1 to 20.

6. When moving the slider from the example in Exercise 5, remember that Mathematica is choosing the optimal plot range as the slider is moved. Use **PlotRange** to introduce a fixed plot range of -5 to 100.

7. Copy the input from Exercise 6 and add a second constant d to change $3x$ to $3dx$, where d also ranges from 0 to 20.

8. Copy the input from Exercise 7 and add another function so that you are visualizing both $cx^2 + 3dx - 1$ and $2cx^2 - dx + 3$. (Reminder: to visualize two functions on the same set of axes, place the functions in a list.)

9. Use the **ThermometerGauge** command to create a gauge illustrating the temperature of 10 on a scale of 0 to 50.

10. Now use **Manipulate** to create a model of the temperature $10x$, where x can be changed from 0 to 5.

CHAPTER 8

Sharing Mathematica Notebooks

Introduction

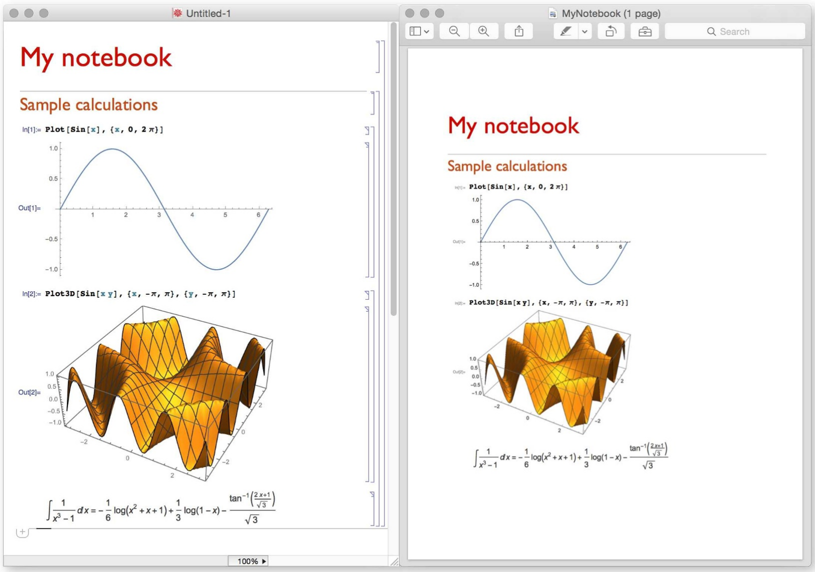
Mathematica provides a broad collection of commands and knowledge for exploring concepts. Sharing results is a central part of any concept exploration, and Mathematica's capabilities for sharing work are diverse and support many types of collaborations. Mathematica can be used to share static documents, but it has the unique capability of sharing notebooks that include live calculations.

Authoring versus Viewing Notebooks

The default document type in Mathematica is a notebook. Notebooks are platform independent, meaning that they look and run the same on a wide variety of operating systems. This allows users to share their materials without needing to do anything special to accommodate viewing and execution of those files on computers with different operating systems. When sharing documents with other Mathematica users, notebook format (.nb) provides the most flexibility.

Delivering Content in Static Formats

PDF is a common file format used to share documents, and Mathematica supports saving a complete notebook as a PDF. The styling of all text and calculations will be retained in the PDF document. The following screen shots show a notebook on the left, and on the right is the PDF generated by saving the notebook.



My notebook

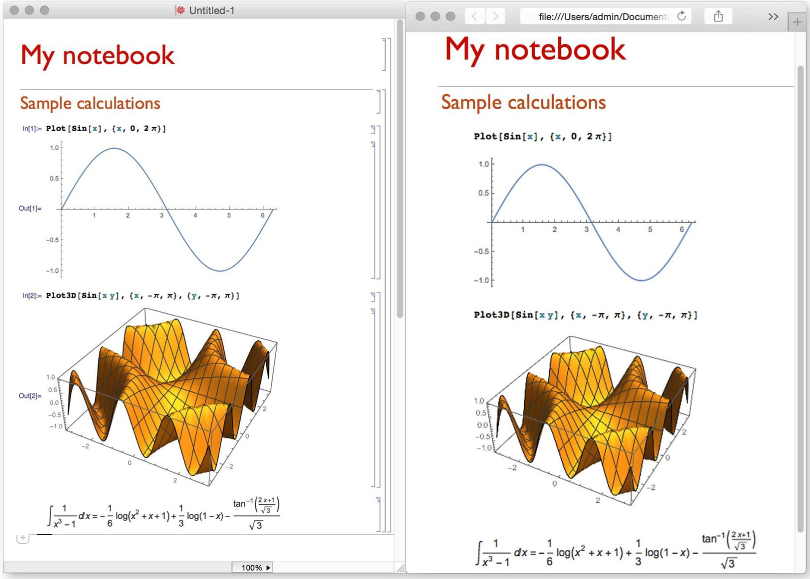
Sample calculations

```
In[1]:= Plot[Sin[x], {x, 0, 2 π}]
Out[1]=
In[2]:= Plot3D[Sin[x y], {x, -π, π}, {y, -π, π}]
Out[2]=
```

$$\int \frac{1}{x^3-1}\,dx = -\frac{1}{6}\log(x^2+x+1) + \frac{1}{3}\log(1-x) - \frac{\tan^{-1}\left(\frac{2x+1}{\sqrt{3}}\right)}{\sqrt{3}}$$

While this chapter does not have commands to retype, you can replicate the examples shown in the screen shots and then save as various file types to see the results.

Sometimes sharing a notebook as a webpage is preferable to sharing as a PDF. A notebook can be saved as HTML, allowing static content to be posted on webpages for anyone to view. When notebooks are saved as HTML, graphics and typesetting components are automatically saved as GIF images, and content and style files are organized into a system of folders. The resulting folders and files can be uploaded to a web server and made available for immediate viewing. The following screen shot shows a notebook on the left, and on the right is the webpage generated by saving the notebook.



My notebook

Sample calculations

```
In[1]:= Plot[Sin[x], {x, 0, 2 π}]
Out[1]=
In[2]:= Plot3D[Sin[x y], {x, -π, π}, {y, -π, π}]
Out[2]=
```

$$\int \frac{1}{x^3-1}\,dx = -\frac{1}{6}\log(x^2+x+1) + \frac{1}{3}\log(1-x) - \frac{\tan^{-1}\left(\frac{2x+1}{\sqrt{3}}\right)}{\sqrt{3}}$$

When notebooks are saved as HTML, the HTML document mirrors the exact appearance of the Mathematica notebook. This means that visual effects, like the **In** and **Out** cell labels, will be present in the HTML version if they were present in the original notebook at the time it was saved. The previous example was saved and reopened in Mathematica to exclude the **In** and **Out** cell labels. In addition, if cell groups were expanded or collapsed, or if the input was unevaluated for graphics, this information will be excluded from the HTML version since the HTML output is a static snapshot of the notebook in its current form.

> The options to share documents in this chapter are limited to what a typical Mathematica user can do with a desktop license or a Mathematica Online account. However, there are other Wolfram products, like webMathematica, that provide more specialized functionality for more intricate projects.

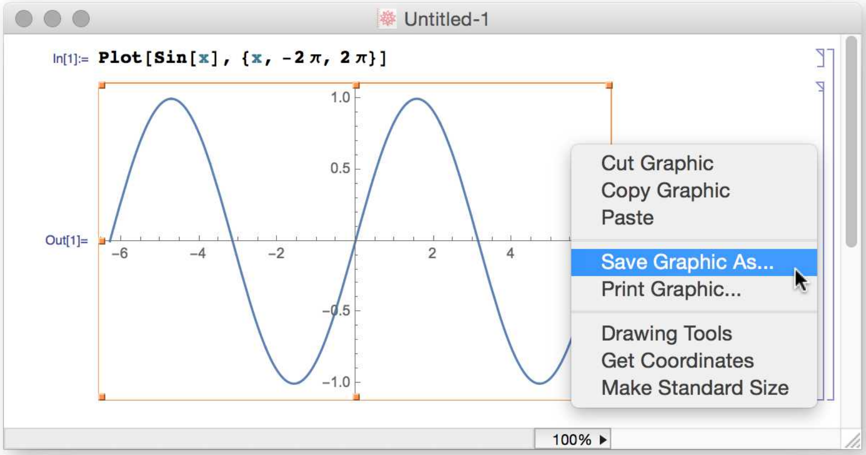
Mathematica supports saving notebooks as LaTeX, a markup language popular for technical and scientific papers and journal articles. When a notebook is saved as LaTeX, markup language source code is generated, including statements to include relevant libraries, depending on the formatting and content of the saved document. In addition, graphics are automatically saved as EPS files.

It is important to note that Mathematica does not include a LaTeX compiler, so the source code that is generated will need to be compiled with a different program in order to generate the final output as something like a PDF.

Delivering Individual Graphics

If a mix of software packages is used for a certain project, it might be preferable to share only a specific graph or chart rather than an entire document. Mathematica's support for export makes it an excellent platform for creating high-quality graphics and sharing only specific graphics as images.

The easiest way to save a graphical result from Mathematica is to right-click, which will open a context menu that allows the output to be saved in a variety of formats.



```
In[1]:= Plot[Sin[x], {x, -2 π, 2 π}]
```

Exporting is not covered in much detail here; you can read more about that in **Chapter 19: Importing and Exporting Data**.

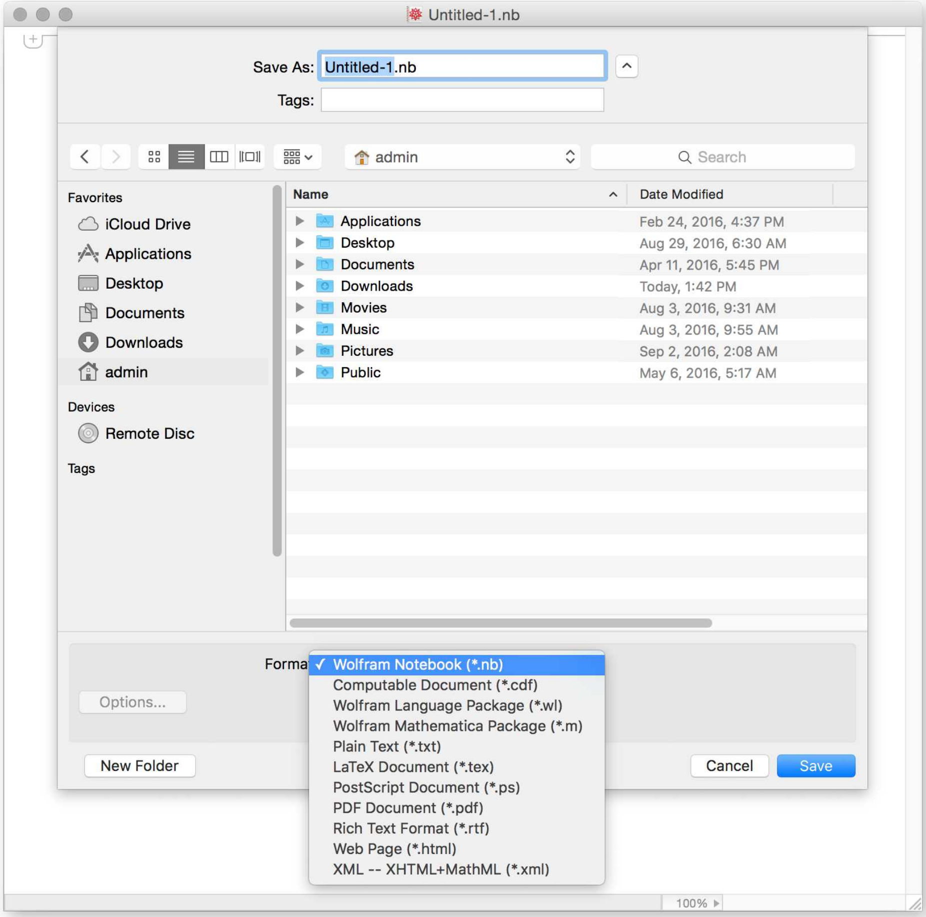
A graphic that includes annotations from **Drawing Tools** can be exported with this same right-clicking method.

Although graphics are most commonly exported as individual elements, it is just one of several capabilities for exporting individual elements of a document. For example, you can also use the **TeXForm** command to take a mathematical expression and generate the markup language for it. This can be handy for generating something like a two-dimensional table layout, which can be much easier to create with Mathematica, compared to directly in TeX.

Delivering Interactive Content with Computable Document Format (CDF)

Sharing static documents is a typical approach with many software packages, but Mathematica adds the unique capability of sharing notebooks that include live calculations and interactive elements. This means that viewers of a shared notebook can rerun calculations on their own and explore an idea in much more depth than what would be available in a static document.

Wolfram CDF Player is a free application that allows users to interact with notebooks that use the CDF file extension (.cdf).



Untitled-1.nb

Save As: Untitled-1.nb

Tags:

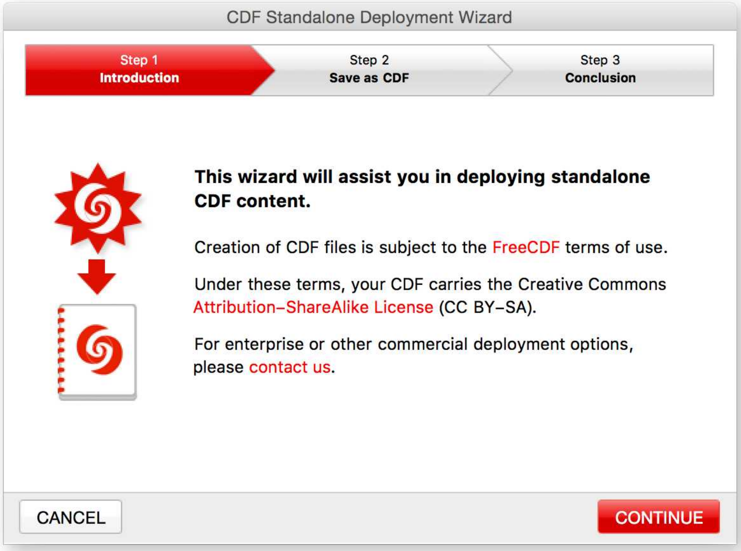
Name	Date Modified
Applications	Feb 24, 2016, 4:37 PM
Desktop	Aug 29, 2016, 6:30 AM
Documents	Apr 11, 2016, 5:45 PM
Downloads	Today, 1:42 PM
Movies	Aug 3, 2016, 9:31 AM
Music	Aug 3, 2016, 9:55 AM
Pictures	Sep 2, 2016, 2:08 AM
Public	May 6, 2016, 5:17 AM

Format: ✓ Wolfram Notebook (*.nb)

Mathematica users can author materials, save them as CDF files and distribute those files. The recipients of those files can view and interact with mouse-driven controls, like sliders, after installing the free CDF Player. CDF technology allows Mathematica-based content to be shared with a larger audience, an audience that can actually interact with and drive results from a document—as opposed to sending static documents that can only be read.

Although the typical convention is to share CDF files (.cdf) for use with the Wolfram CDF Player application, and to share notebook files (.nb) for use with Mathematica, both CDF and notebook files can be opened and edited in Mathematica.

In addition to saving the entire notebook in CDF format, a wizard is available to deploy CDF files. This deployment process gives greater control to the author in terms of locking down content and preventing editing, as well as options to change some appearance elements. CDFs can be deployed by clicking the **File** menu, opening the **CDF Export** submenu and choosing **Standalone**. The CDF deployment wizard has three steps for standalone documents that are designed to be viewed in Mathematica or CDF Player.



This wizard will assist you in deploying standalone CDF content.

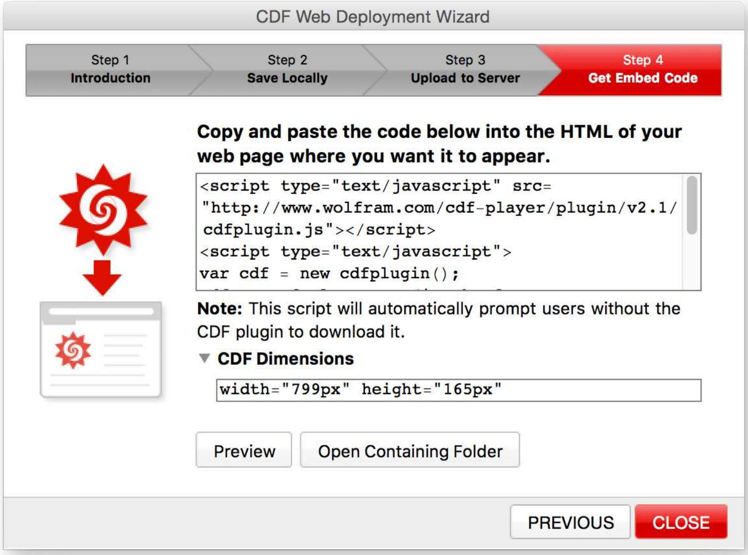
Creation of CDF files is subject to the FreeCDF terms of use.

Under these terms, your CDF carries the Creative Commons Attribution–ShareAlike License (CC BY–SA).

For enterprise or other commercial deployment options, please contact us.

CDFs can also be posted within a website to make them easier to share with a large group. The CDF deployment wizard generates web-embeddable CDFs by clicking the **File** menu, opening the **CDF Export** submenu and choosing **Web Embeddable**.

The deployment process is similar to deploying a standalone CDF file, except for an additional step to specify whether the CDF file will reside in the same directory as the HTML webpage in which it will be embedded, or whether the CDF file will reside in a different web directory. The wizard will create a deployed version of the CDF file and provide JavaScript code that can be pasted into the source code of the webpage to display the embedded CDF. Once the generated CDF file is uploaded to the appropriate web server and the JavaScript code is placed into the appropriate webpage, any visitor to that webpage can view and interact with the content.



Copy and paste the code below into the HTML of your web page where you want it to appear.

```
<script type="text/javascript" src=
"http://www.wolfram.com/cdf-player/plugin/v2.1/
cdfplugin.js"></script>
<script type="text/javascript">
var cdf = new cdfplugin();
```

Note: This script will automatically prompt users without the CDF plugin to download it.

▼ **CDF Dimensions**

```
width="799px" height="165px"
```

After the CDF file is posted within a website, the viewer of the CDF file still needs to install CDF Player (or Mathematica) to power the live calculations.

Similarly to saving a notebook as HTML, a deployed CDF document will retain the same content as the source Mathematica file at the time of deployment. It is a good idea to review notebooks before deploying as CDFs to make sure that cell groups are collapsed or expanded as desired before launching the deployment wizard.

Standard installations of Mathematica and CDF Player include a browser plugin that allows viewing of CDF content embedded in webpages. An example of a website with embedded CDF content is the Wolfram Demonstrations Project at: demonstrations.wolfram.com.

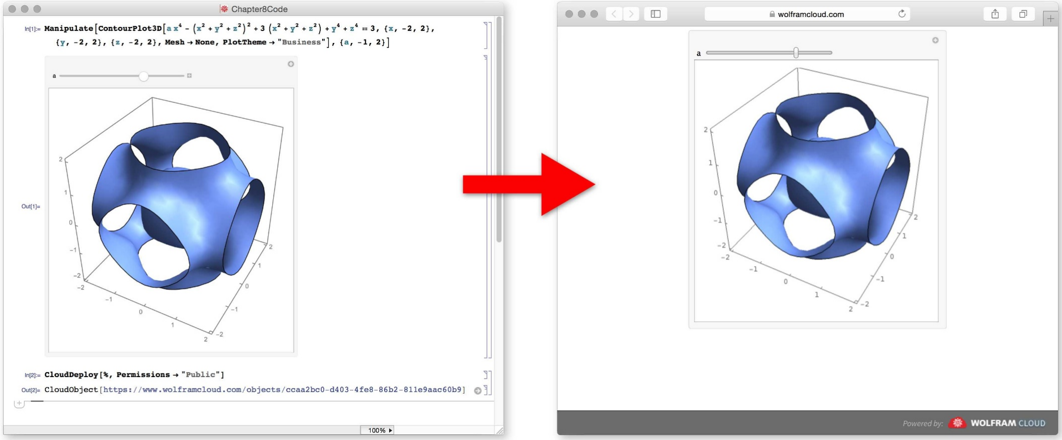
Updating a CDF file is easy when it is embedded into a website. Updating one file on a web server is more convenient than updating each and every copied file residing on many individual machines. For organizations that use web-based tools, like course management systems, the ability to embed interactive content into webpages without forcing visitors to leave the online environment can also be very attractive.

Finally, some special tools are available for embedding CDF content into specific frameworks, like a plugin that can be used to post CDFs to WordPress blogs. These tools are easily found by searching the Wolfram website.

Interactive Content with CDF in the Cloud

All users of CDF files must first install CDF Player or Mathematica in order to view Mathematica-based content. CDF in the Cloud is a newer technology that simplifies the process of sharing files by using the Wolfram Cloud to power the calculations via the web. Since nothing needs to be installed for CDF in the Cloud to work, this makes sharing simple and practical. CDF in the Cloud is similar to CDF in that both provide ways to share Mathematica-based work with people who may not have Mathematica. And by eliminating any installation process, CDF in the Cloud files can include live calculations viewable on tablets or mobile devices.

The **CloudDeploy** function is used to create CDF in the Cloud files. A very common approach is to use this command with **Manipulate** statements to create interactive models that run in the cloud.



```
In[1]:= Manipulate[ContourPlot3D[a x^4 - (x^2 + y^2 + z^2)^2 + 3 (x^2 + y^2 + z^2) + y^4 + z^4 == 3, {x, -2, 2},
          {y, -2, 2}, {z, -2, 2}, Mesh → None, PlotTheme → "Business"], {a, -1, 2}]

In[2]:= CloudDeploy[%, Permissions → "Public"]
Out[2]= CloudObject[https://www.wolframcloud.com/objects/ccaa2bc0-d403-4fe8-86b2-811e9aac60b9]
```

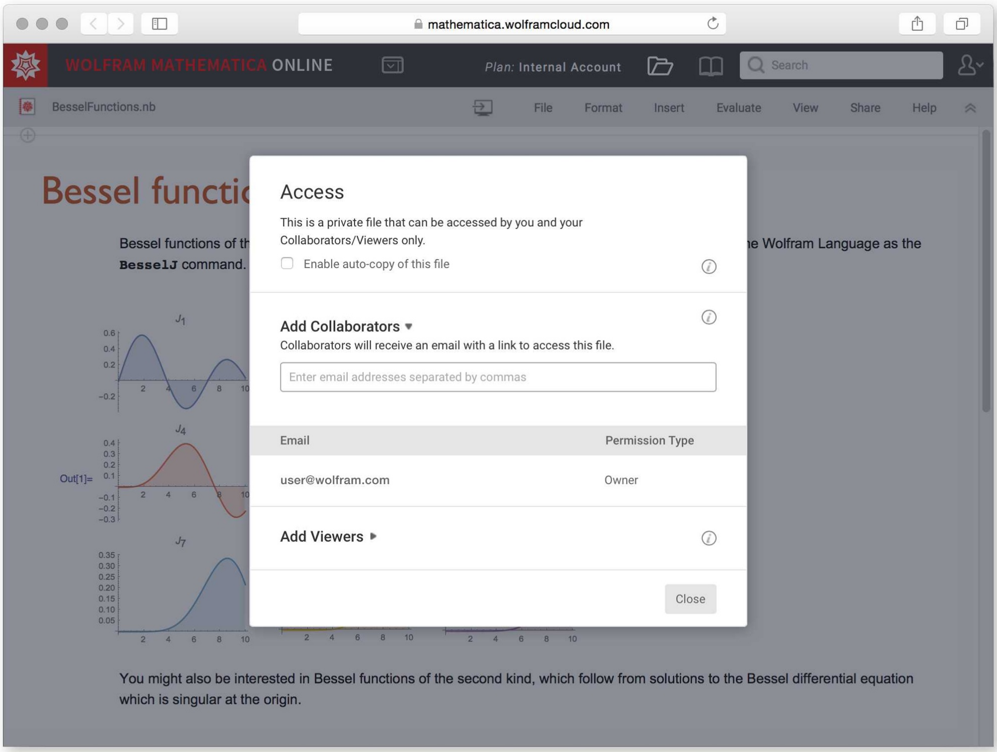
Although licensing is outside of the scope of this book, the concept of Cloud Credits is relevant to CDF in the Cloud. Cloud Credits govern the licensing for CDF in the Cloud documents, and each calculation that is performed by a user of a CDF in the Cloud costs a certain number of Cloud Credits. Eligible Mathematica users have some Cloud Credits built into their licenses, but additional Cloud Credits can be purchased.

Saving Notebooks to the Cloud

The default notebook file format is easy to share with other users who will be editing or adding content. Instead of storing documents locally and sending them to collaborators via email or other channels, users of Mathematica Online can easily share notebooks stored in their Wolfram Cloud accounts. Both Mathematica on the desktop and Mathematica Online use the same notebook file format. Mathematica users can save their notebooks to the Wolfram Cloud by clicking the **File** menu and selecting **Save to Wolfram Cloud...**, which opens a dialog window that displays a file browser for their Wolfram Cloud storage. Mathematica Online users with notebooks stored on their local machines can log in to Mathematica Online and use the file operations buttons in the sidebar to upload notebooks to their Wolfram Cloud storage.

Sharing and Collaborating in the Cloud

The owner of a Mathematica Online notebook can use the **Share** menu to add collaborators to the document. The owner can choose whether each collaborator has read or write access. If write access is granted, then any changes made to the notebook, either by the owner or a collaborator, are saved. Collaborators need to have their own Mathematica Online subscriptions in order to view and edit files shared with them.



Bessel functi

Bessel functions of th … e Wolfram Language as the `BesselJ` command.

Access

This is a private file that can be accessed by you and your Collaborators/Viewers only.

☐ Enable auto-copy of this file

Add Collaborators ▾

Collaborators will receive an email with a link to access this file.

Enter email addresses separated by commas

Email	Permission Type
user@wolfram.com	Owner

Add Viewers ▸

Close

You might also be interested in Bessel functions of the second kind, which follow from solutions to the Bessel differential equation which is singular at the origin.

If you do not want a notebook to be edited, but you want your collaborators to be able to continue the work you have already started, you should ask them to duplicate the file by clicking the **File** menu and choosing **Duplicate**. This will allow your collaborator to have their own copy of the notebook, and then they can make changes or experiment without overwriting your original document.

The owner of a Mathematica Online notebook can also add viewers to a notebook. Viewers do not need to have a Mathematica Online subscription, but they will need to sign up for a free account before they can view the content of the shared notebook. Mathematica Online users receive a certain number of free viewer accounts that they can associate with their notebooks, with the option to purchase more accounts as needed.

Conclusion

Mathematica has uniquely flexible options to share work, ranging from sharing very rich notebook files with other Mathematica users to sharing mouse-driven applications to be run with the free CDF Player. Other options outside the scope of this book include running a Mathematica kernel on a file server with a web browser as the interface, using Wolfram Language code as an API linked to another program or having a private cloud of Wolfram technology hosted for your organization.

Exercises

1. Create a new notebook with a section cell that contains the text "Chapter 8: Sharing Mathematica Documents." Add an input cell to solve the equation $3\,x + 1 = 7\,x - 9$ for x. Add another input cell to plot $\frac{\sin(x)}{x}$, where x goes from -10 to 10.
2. Add a **Manipulate** command to plot $\frac{\sin(x\,y)}{x}$, where x goes from -10 to 10 and y goes from 1 to 5, and include the appropriate option to restrict the plot range from -1 to 5.
3. Save the notebook from Exercise 2 as a PDF and open the resulting file to view its contents.
4. Save the notebook from Exercise 2 as HTML and open the resulting webpage with a web browser. (Note: you do not need to upload the files to a web server; you can view them on your machine locally.)
5. Save the plot of $\frac{\sin(x)}{x}$ as a JPEG file.
6. Save the notebook from Exercise 2 to the Wolfram Cloud.
7. Log in to Mathematica Online and open the file that was saved in Exercise 6.
8. Add a viewer to the file that was saved in Exercise 6.
9. Add a collaborator to the file that was saved in Exercise 6.
10. Remove the viewers and collaborators who were added in Exercises 8 and 9.

CHAPTER 9

Finding Help

Introduction

Mathematica is extremely easy to get started with, and new users can quickly master the basics. Since Mathematica is built upon the Wolfram Language, however, there are thousands of commands available, spanning areas as diverse as mathematics, physical science, life science, engineering, computer science, data science, economics, business and most other major technical disciplines. Because there are so many commands available, Mathematica has a comprehensive documentation system that contains hundreds of thousands of examples. This makes the documentation an excellent place to grab a quick example of how to do something, as well as a destination for detailed information about the intricacies of functions and notes about their implementation.

Getting Help while Working

Mathematica's Code Assist technology makes help only a click away by providing command completion, function templates and quick access to the documentation as function names are typed. There may be times when a quick refresher on the syntax for a command is needed, and for those instances, the **?** operator can be used to retrieve a brief snippet about the command in question.

```
?Plot
```

Plot[f, {x, x_{min}, x_{max}}] generates

a plot of f as a function of x from x_{min} to x_{max}.

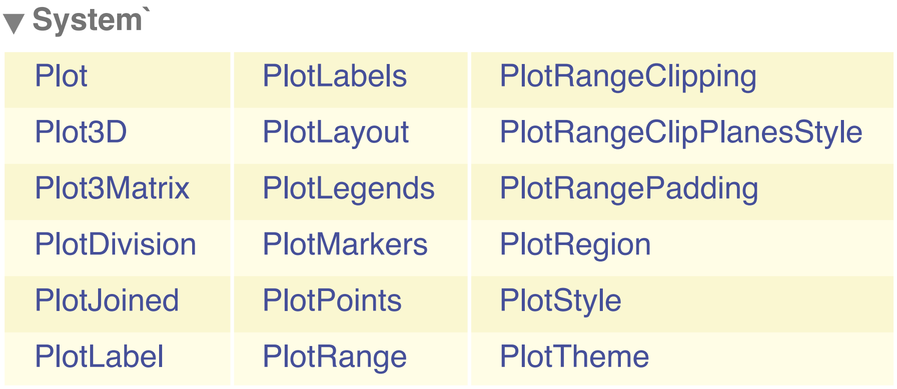
Plot[{f_1, f_2, …}, {x, x_{min}, x_{max}}] plots several functions f_i.

Plot[{…, w[f_i], …}, …] plots f_i with features defined by the symbolic wrapper w.

Plot[…, {x} ∈ reg] takes the variable x to be in the geometric region reg. >>

The results returned by the **?** operator are similar to the ones displayed by Code Assist when it shows available command templates, but **?** can also be used with the ***** wild card operator to find lists of commands that match the given pattern, such as to find all of the commands that start with the letters **Plot**.

```
?Plot*
```



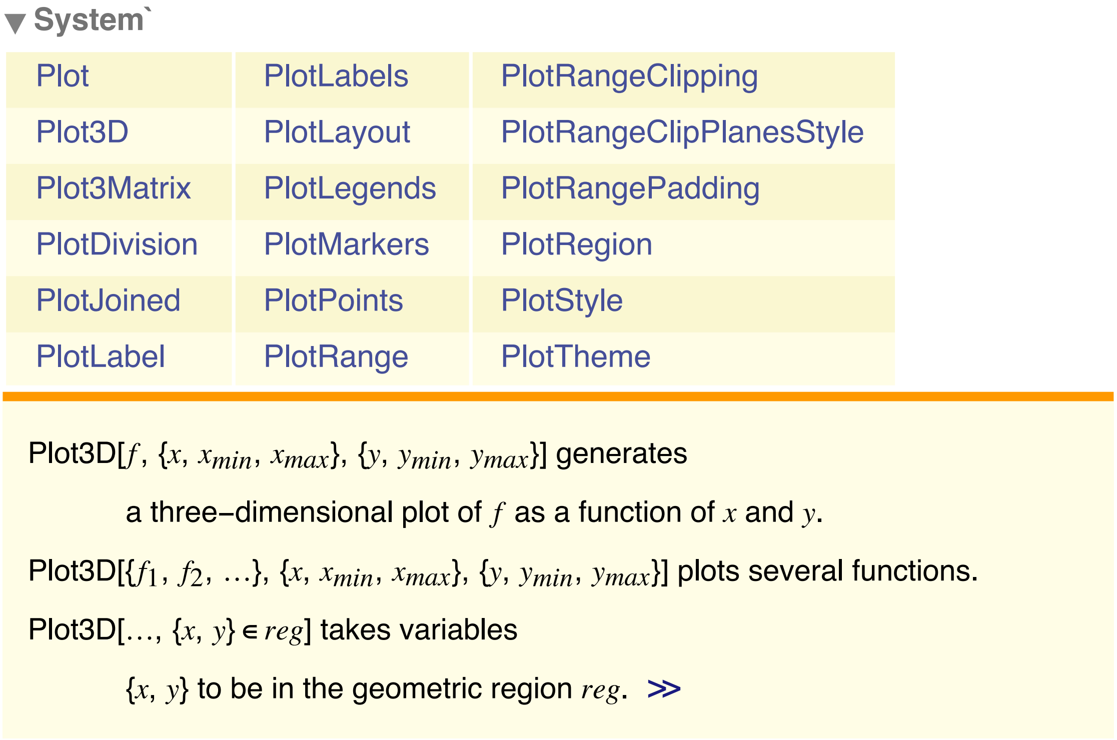
▼ System`

Plot	PlotLabels	PlotRangeClipping
Plot3D	PlotLayout	PlotRangeClipPlanesStyle
Plot3Matrix	PlotLegends	PlotRangePadding
PlotDivision	PlotMarkers	PlotRegion
PlotJoined	PlotPoints	PlotStyle
PlotLabel	PlotRange	PlotTheme

You can use multiple wild cards in this type of search, so evaluating **?*Plot*** will show all commands that start with **Plot** (like **Plot3D**), end with **Plot** (like **ListPlot**) or have **Plot** somewhere in the middle (like **ListPlot3D**).

Clicking one of the symbol names returned from such a search (like **Plot3D**) will print a definition box, similarly to evaluating **?** and that symbol name directly (like **?Plot3D**).

```
?Plot*
```



▼ System`

Plot	PlotLabels	PlotRangeClipping
Plot3D	PlotLayout	PlotRangeClipPlanesStyle
Plot3Matrix	PlotLegends	PlotRangePadding
PlotDivision	PlotMarkers	PlotRegion
PlotJoined	PlotPoints	PlotStyle
PlotLabel	PlotRange	PlotTheme

Plot3D[f, {x, x_{min}, x_{max}}, {y, y_{min}, y_{max}}] generates

a three–dimensional plot of f as a function of x and y.

Plot3D[{f_1, f_2, …}, {x, x_{min}, x_{max}}, {y, y_{min}, y_{max}}] plots several functions.

Plot3D[…, {x, y} ∈ reg] takes variables

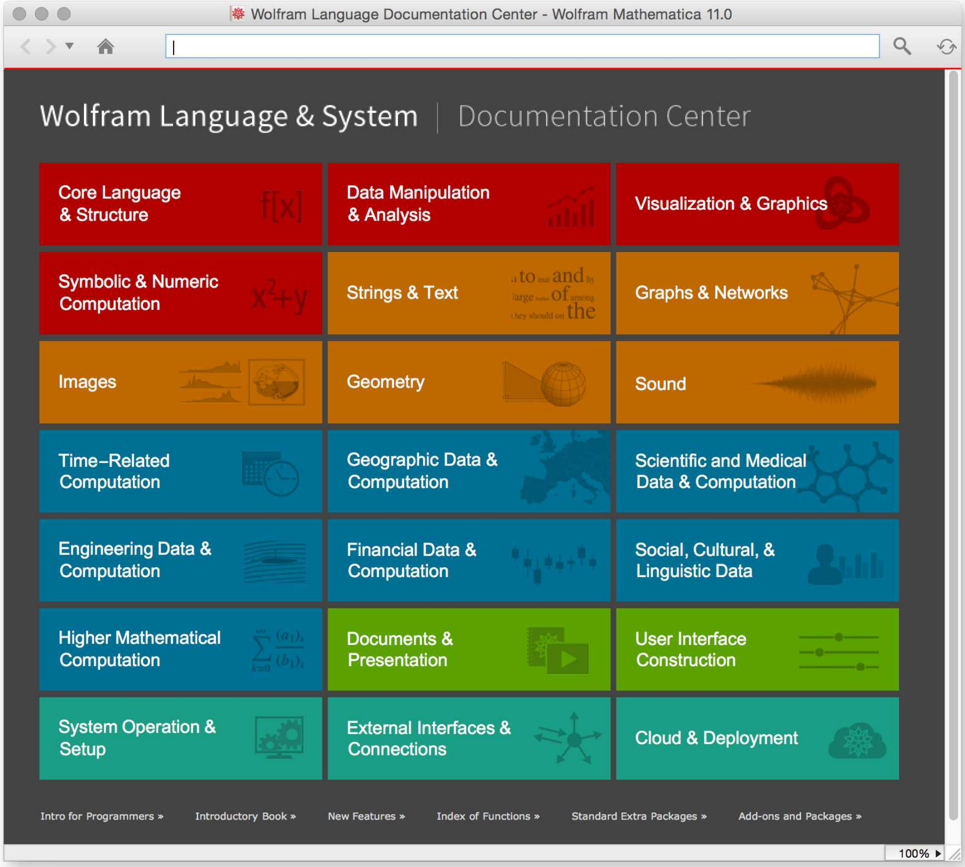
{x, y} to be in the geometric region reg. >>

Clicking the chevrons icon in a definition box will open a new window to show the documentation for the specific function.

> Another way to open the documentation for a specific function is to highlight the function name (or to place the cursor at the end of its name), click the **Help** menu and choose **Find Selected Function**. There is also a keyboard shortcut for this, which can be seen when the **Help** menu is opened.

Navigating the Documentation

While it can be very useful to open a specific command's documentation, it is also important to understand the general organizational structure of the documentation system so that it can be used to browse information even when a specific command name is not known. To view the documentation, click the **Help** menu and choose **Wolfram Documentation**. A home page with tiled areas of functionality is displayed, and these tiles can be clicked to reveal links to more specific information. There is also a search bar at the top to look for command names, general topics or keywords of interest.


Wolfram Language Documentation Center - Wolfram Mathematica 11.0
Wolfram Language & System | Documentation Center
Core Language & Structure
Data Manipulation & Analysis
Visualization & Graphics
Symbolic & Numeric Computation
Strings & Text
Graphs & Networks
Images
Geometry
Sound
Time-Related Computation
Geographic Data & Computation
Scientific and Medical Data & Computation
Engineering Data & Computation
Financial Data & Computation
Social, Cultural, & Linguistic Data
Higher Mathematical Computation
Documents & Presentation
User Interface Construction
System Operation & Setup
External Interfaces & Connections
Cloud & Deployment
Intro for Programmers »
Introductory Book »
New Features »
Index of Functions »
Standard Extra Packages »
Add-ons and Packages »
100%

There are four main types of Documentation Center pages: function pages, guide pages, tutorial pages and "How tos."

- Every command in Mathematica has a function page with a syntax definition and often includes several examples of how to use that function.
- Guide pages are topical in nature and provide links to commands relevant to that particular topic.
- Tutorial pages explain how to use Mathematica for particular tasks and read more like textbooks than the reference manual approach that function pages use.
- "How tos" are concise, step-by-step examples of performing specific operations in Mathematica.

Function pages are very important to understanding how a particular command works. Function pages can be accessed directly through Code Assist menus, by clicking the chevrons in the definition box returned from the **?** operator, by browsing guide pages or by using the search bar in the documentation window.

> If you know the exact name for a Wolfram Language command, typing it into the documentation's search bar and pressing Return or clicking the search icon will open that particular command's function page. It is important that the exact name, including capitalization, is used. Searching for "Plot" without the quotes will open the function page for the **Plot** command, but searching for "plot" will return a list of search results that include that phrase.

Each function page starts with a definition of the syntax for the function, followed by a Details and Options section. The Details and Options section is aptly named, providing details that may include notes about implementation of the function, a list of the default values for that function's options and notes about subtleties related to execution of the command. While this section contains a wealth of knowledge, many new users can safely forgo a thorough examination of this material until a specific need arises.

The next part of a function page is the Examples section, which may include subsections such as Basic Examples, Scope, Generalizations & Extensions, Options, Applications, Properties & Relations and Neat Examples.

Basic Examples is an excellent way to get a quick snapshot of how the function works, with examples that typically illustrate the multiple ways that a function can be used. For example, the Basic Examples section for the **Plot** command shows how to plot a single function, how to plot several functions with a legend and how to create a plot with filling.

The sections for Scope and Generalizations & Extensions provide more detailed information typically related to advanced uses.

The Options section details what settings can be used to control function behavior, from the algorithms used for execution to customizing the appearance of the results that are returned. The sheer number of examples usually displayed in this section makes it an excellent source of fodder for learning how to use Mathematica.

The remaining sections containing examples—Applications, Properties & Relations and Neat Examples—typically serve as reference material for more advanced users.

> The Applications and Neat Examples sections have some incredibly interesting and cool examples that showcase Mathematica's power and flexibility. If you find yourself on a function page in the documentation, you should take a quick peek at these sections, if only to gain a greater appreciation for what is possible with Mathematica.

The remaining sections on the function page may provide links to related function pages, tutorials, guide pages and links to the Wolfram website. Guide pages in particular can be useful for new users who do not yet know the commands available for an area of computation, serving as an index to functions and related tutorials. (Guide pages are displayed when clicking the tiles on the documentation home page to pick a particular topic of interest.)

Interacting with the Examples

Besides being extremely comprehensive, Mathematica's documentation is also interactive: documentation pages are simply Mathematica notebooks, meaning the examples contained therein can be evaluated and changed. However, any changes made to the documentation will not be saved once the window is closed. This makes the documentation an excellent sandbox for trying out new commands and approaches, and once a satisfactory result is attained, the input and/or output cells can be copied and pasted to a new notebook.

However, sometimes care must be taken when copying content from the documentation. Since the definitions in the documentation are not shared with other notebooks, it is important to copy all relevant cells (including cells where variables or functions are defined) when copying them from the documentation to another notebook.

Additional Sources for Help

The documentation for Mathematica is available in product as well as online at reference.wolfram.com.

For those who prefer lecture-based learning, Wolfram Training offers both online and onsite training courses for Mathematica and other Wolfram technology from their website: www.wolfram.com/training.

Wolfram Community provides a forum where users can exchange ideas, offer assistance and solicit feedback on their work: community.wolfram.com.

Conclusion

Thanks to free-form input and Code Assistance technology, it is incredibly easy for new users to get started with Mathematica. For users who learn by imitation, the documentation can provide an excellent source of material to draw examples from, while also serving as a comprehensive and detailed reference for beginners and veteran users alike.

Exercises

1. Go to the documentation, click the "Symbolic & Numeric Computation" tile and then select "Numbers & Precision." Once there, choose the first function, which is **N**. This will open the function page for the **N** command. Once there, modify the first example from **N[1/7]** to **N[4/13]** and recalculate.
2. Create a new cell immediately following your output from Exercise 1 and enter the command to numerically evaluate 4 / 13 to 10 digits of precision. (Note that an example in the reference page illustrates this use of **N** as well.)
3. Next, create a new input cell and find the numerical approximation of the square root of 5 to 15 digits of precision. The function for square root is **Sqrt**, or you can use the **Basic Math Assistant** palette or the keyboard shortcut Ctrl+2 to enter a typeset representation of the square root command.
4. Create a new input cell to find the numerical approximation of a list of the form {sin(1), sin(2)} to two digits of precision. (Remember: curly braces are used to create a list.)
5. Although documentation pages are editable, no work is saved. However, the cells can be copied and pasted to a new notebook and then saved. Do this for your work thus far and save your notebook.

6. Return to the documentation home page, click the "Core Language & Structure" tile, and then choose "Lists." From this guide page, click the **Table** command to open its function page. Copy the first input/output cell pair from the Basic Examples section into a new notebook and evaluate.
7. Add a text cell to the top of the notebook created in Exercise 6 that reads "A table of sin(*x*) values, where *x* ranges from 1 to 20."
8. Return to the input cell containing the **Table** command and create a new variable named **myTable** to store the results of the command.
9. Create a new input cell and use the **N** command to compute an approximation of **myTable** to two digits of precision.
10. Visualize the values in **myTable** by using the **ListPlot** command. (If you need help, then find the function page for **ListPlot**, using the methods described in this chapter.)

Part II

Extending Knowledge

CHAPTER 10

2D and 3D Graphics

Introduction

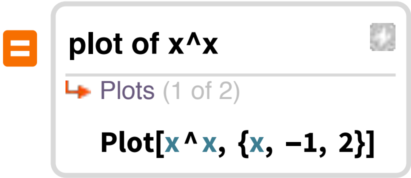
The Wolfram Language includes many commands to visualize functions, equations and data, providing results that are both useful and aesthetically pleasing. This chapter will introduce different ways of creating graphics and discuss some of the most commonly used visualization commands.

Visualizing Univariate Functions

Using Free-Form Input

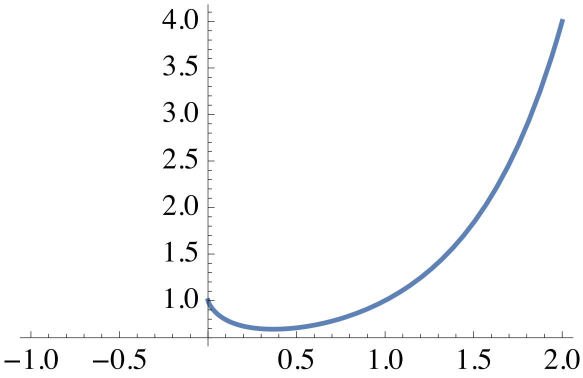
Although the documentation is an effective way to explore the full scope of what visualization commands are available, free-form input is a good starting point for creating simple plots. By default, a free-form input statement will return a single output, which is often the desired result for a simple visualization.

Just like in earlier chapters, type the equal sign when starting a new input cell to invoke free-form input. Free-form input takes everyday language as input and returns a corresponding output, along with the equivalent Wolfram Language command for performing that task.

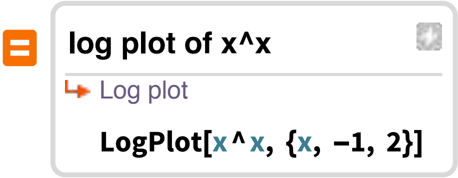


```
plot of x^x
```

```
Plot[x^x, {x, −1, 2}]
```



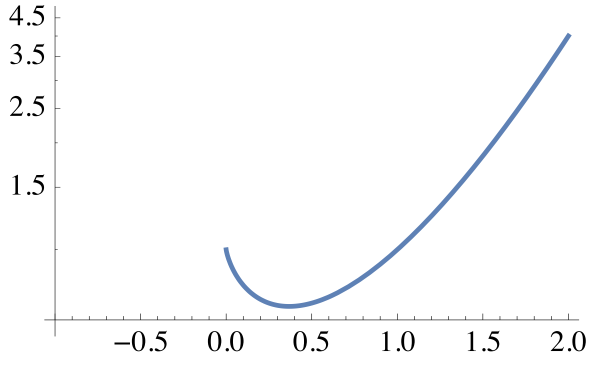
Specifying a phrase like "log plot" is all the change that is necessary to achieve a different output using an entirely different command.



log plot of x^x

Log plot

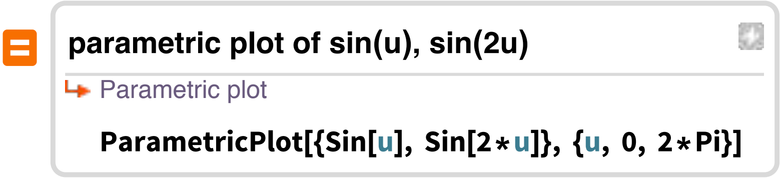
```
LogPlot[x^x, {x, −1, 2}]
```



It can be useful to enter typeset expressions into a free-form input cell. Open the **Classroom Assistant** palette by selecting it from the **Palette** menu and then navigate to the **Calculator** section, which has buttons to paste mathematical symbols like π, and operations like square roots and exponents.

Both the **Plot** and **LogPlot** commands require an expression that contains a single independent variable, along with a domain to plot over. When free-form input is used, Mathematica automatically chooses the domain unless instructed otherwise.

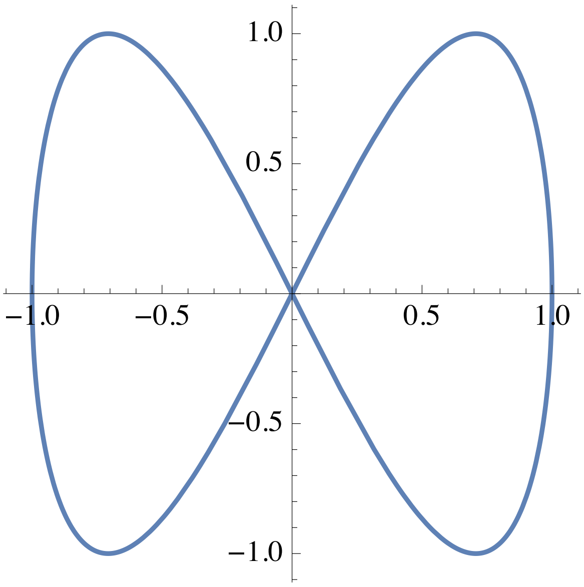
Free-form input can also be used for other types of plots, including parametric plots. Two functions are required for a parametric plot: the first function specifies the *x* coordinate of each point and the second function specifies the *y* coordinate of each point.



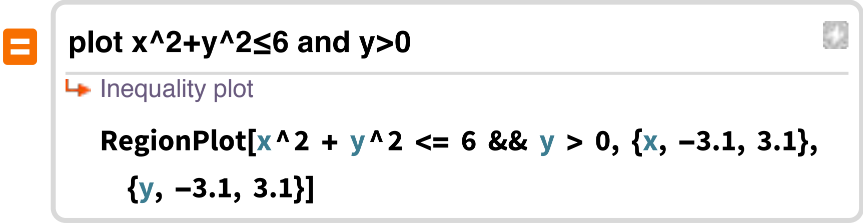
parametric plot of sin(u), sin(2u)

Parametric plot

```
ParametricPlot[{Sin[u], Sin[2*u]}, {u, 0, 2*Pi}]
```



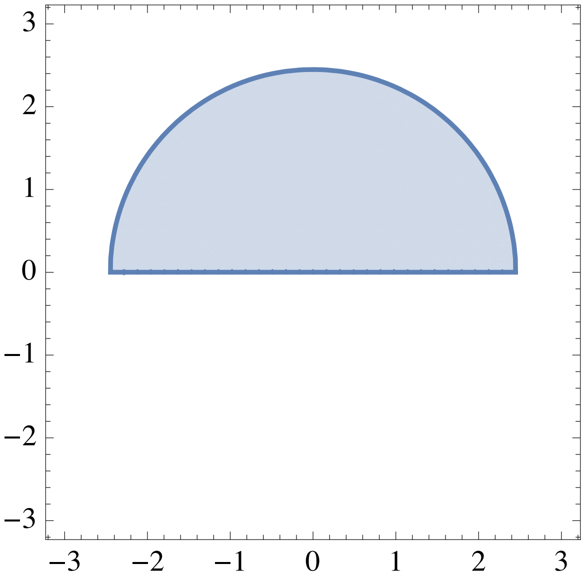
It is also possible to graph regions defined by inequalities using free-form input. The following free-form expression shows the equivalent Wolfram Language syntax, which uses the **RegionPlot** command to visualize this region.



plot x^2+y^2≤6 and y>0

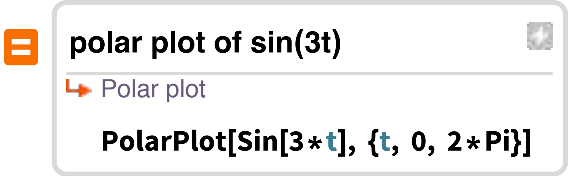
Inequality plot

```
RegionPlot[x^2 + y^2 <= 6 && y > 0, {x, −3.1, 3.1},
  {y, −3.1, 3.1}]
```



RegionPlot accepts a logical combination of expressions constructed with Boolean operators. In Mathematica, the symbol **&&** represents the logical AND function, and the symbol || represents the logical OR function. If you click the Wolfram Language syntax to discard the free-form input in the preceding evaluation, change the logical AND (**&&**) to logical OR (||) and reevaluate, the result will look quite different.

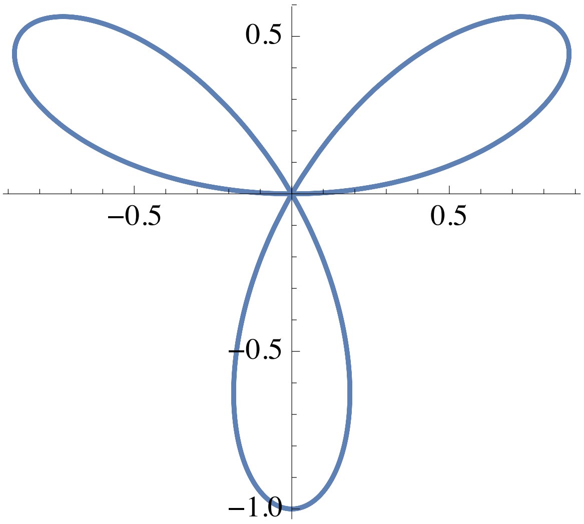
Free-form input can also be used to create polar plots by specifying a parameter of interest. Mathematica will make any additional choices as necessary to create an interesting and attractive visualization. This can be seen in the following example, in which the domain for the parameter *t* was automatically selected to create a nice-looking result.



polar plot of sin(3t)

Polar plot

```
PolarPlot[Sin[3*t], {t, 0, 2*Pi}]
```



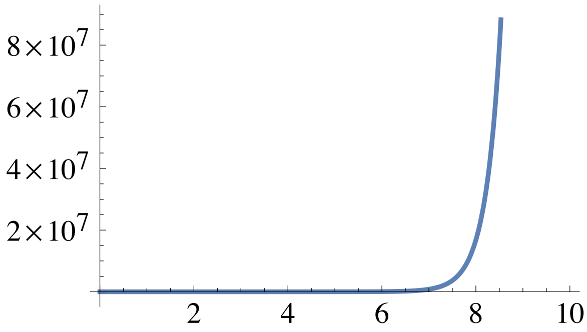
Recall that clicking the plus icon in the upper right-hand corner displays additional results provided by Wolfram|Alpha. Many results given by Wolfram|Alpha have interactive elements, and the results for this particular expression include a model with a slider to trace **t** from 0 to 2 π. Click the pod to select it and press Shift+Enter. This will replace the original result with the interactive application.

Typing Commands Directly

Once users have familiarity with the commands they need, those commands can be entered directly using the Wolfram Language. Even though many different and specialized visualization commands are available, all of the commands follow a similar syntax, making it easy to try out new functions in an intuitive manner.

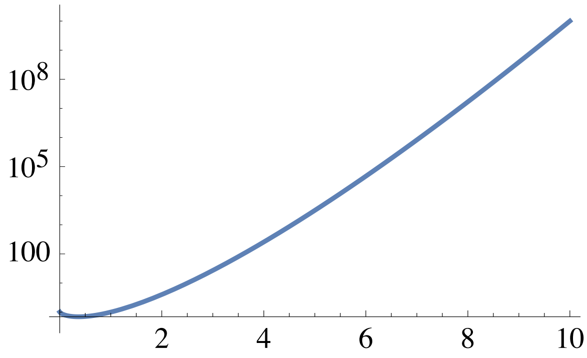
For example, by now readers should have a firm grasp of the **Plot** command, which requires an expression or function to plot, along with the desired domain to plot over. The domain is given in the form of a list that contains the independent variable, the minimum and the maximum.

```
Plot[x^x, {x, 0, 10}]
```



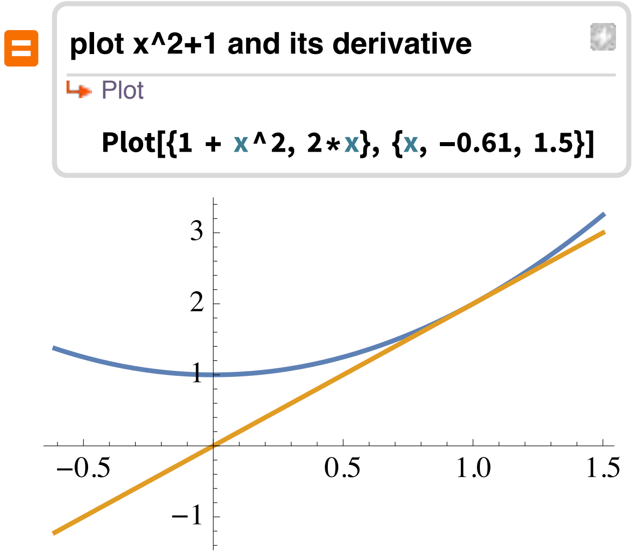
Since **Plot** and **LogPlot** use the same syntax, all that needs to be done is to change **Plot** to **LogPlot** to create a different result.

```
LogPlot[x^x, {x, 0, 10}]
```



Plotting Multiple Functions Together

Free-form input can also be used to plot multiple functions on the same set of axes: just start a free-form input cell and give a list of functions to visualize. As before, Mathematica will make the necessary choices, such as selecting an appropriate viewing window, and the result will use different colors for each function to help differentiate them.



plot x^2+1 and its derivative

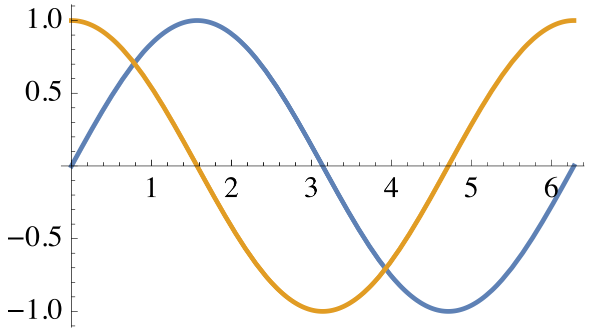
↳ Plot

```
Plot[{1 + x^2, 2*x}, {x, -0.61, 1.5}]
```

Free-form input is obviously useful and has many advantages for a new user, but it does not give you full control over the results. Therefore, learning to directly use the Wolfram Language commands that are relevant to your work will give you much more power and flexibility.

The Wolfram Language command returned by the free-form input in the preceding example shows the syntax for visualizing multiple curves using the **Plot** command. Instead of visualizing a single function, **Plot** can visualize multiple functions if they are placed in a list. The following example shows the result of plotting sin(x) and cos(x) on the same set of axes.

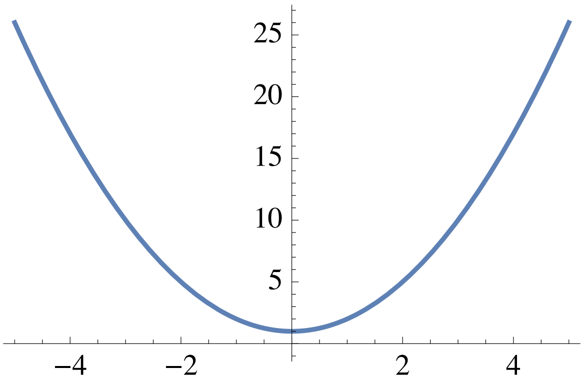
```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 π}]
```



Plotting User-Defined Functions

Many of the plotting examples thus far have focused on plotting mathematical functions, like **Sin**. Mathematica can also plot arbitrary expressions, including user-defined functions. For example, to plot the expression $x^2 + 1$ from -5 to 5, those arguments are passed to the **Plot** command in the required form.

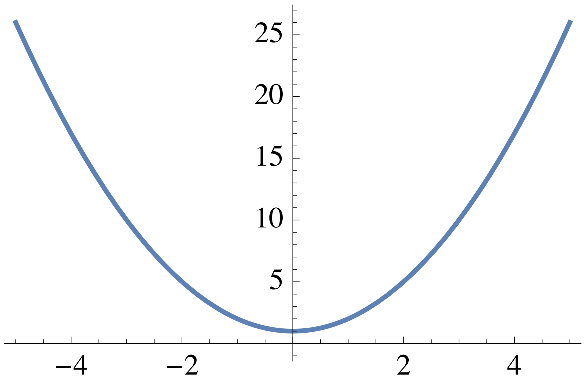
```
Plot[x^2 + 1, {x, −5, 5}]
```



There are times when users have defined their own custom functions, and those user-defined functions can also be passed as arguments to plotting commands. Using a single-variable user-defined function with **Plot** works just the same as using a single-variable Wolfram Language function with **Plot**.

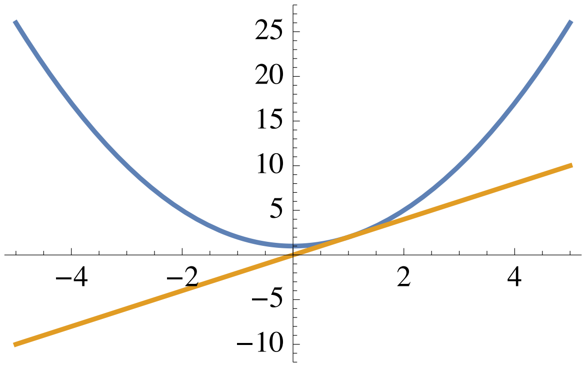
```
f[x_] := x^2 + 1
```

```
Plot[f[x], {x, −5, 5}]
```



User-defined functions can be operated on by other Wolfram Language commands, so they can be used for integration or differentiation. A previous example showed how free-form input could be used to visualize a curve and its derivative on the same set of axes. Now that the function **f[x]** is defined, the same visualization for **f[x]** and **f'[x]** can be created by placing them in a list and passing them to **Plot**.

```
Plot[{f[x], f'[x]}, {x, −5, 5}]
```

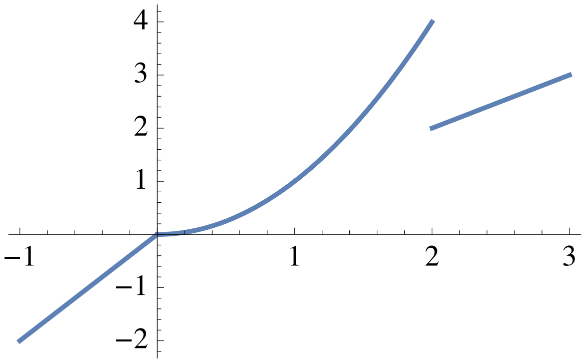


Piecewise functions can be defined using the command **Piecewise** or by using two-dimensional input to typeset a traditional notation for piecewise functions. The simplest way to create a two-dimensional piecewise function is to open the **Basic Math Assistant** palette, expand the **Calculator** section and click the **Advanced** button at the top. A button to create a piecewise function is displayed, and hovering over the button shows a tooltip that displays the keyboard shortcut (Ctrl+Enter) that can be used to add additional rows of conditions.

$$h[x\_] := \begin{cases} 2x & x < 0 \\ x^2 & 0 \le x \le 2 \\ x & x > 2 \end{cases}$$

As a reminder, the palettes referenced in this book may not be available in Mathematica Online, so if that is the environment you are working in, the best approach is to define the function **h[x]** using the **Piecewise** command with one-dimensional input.

```
Plot[h[x], {x, −1, 3}]
```



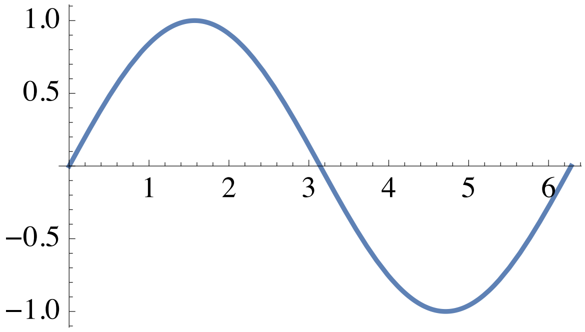
Underlying Plotting Technology

Mathematica includes routines to optimize results that are aesthetically pleasing, with a minimum amount of input required from the user. The graphics results generated by Mathematica are both useful and attractive.

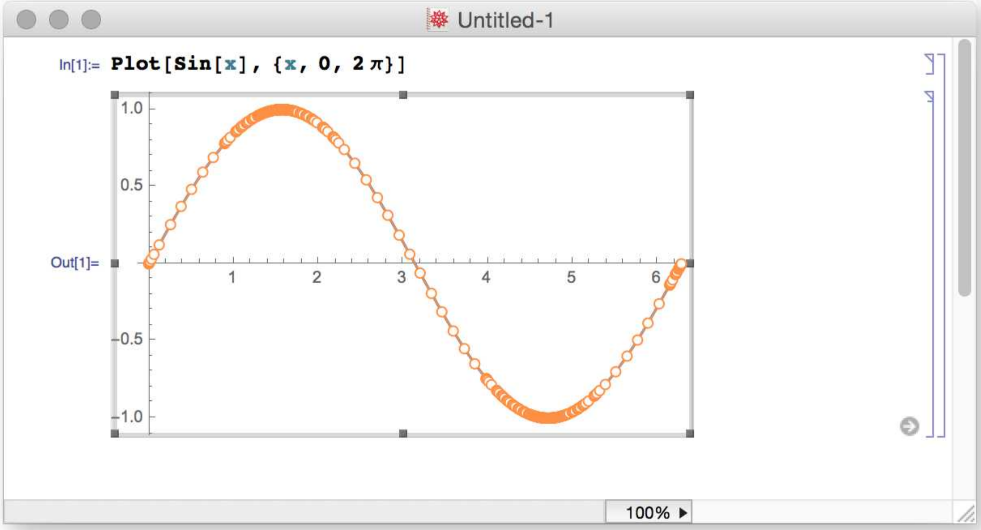
There are times when you might want to override Mathematica's decisions for how graphics look. If that is of interest, then you will want to read the later chapter on styling and customizing graphics, which details the various methods that can be used to control the appearance of graphical output and gives a brief summary of the most common and useful options.

Mathematica graphics can also be altered after rendering is complete. For example, once plotted, a curve of a function can be selected by double-clicking. Once selected, the curve can be clicked and dragged to move it while keeping the rest of the graphic, like the axes, stationary.

```
Plot[Sin[x], {x, 0, 2 π}]
```

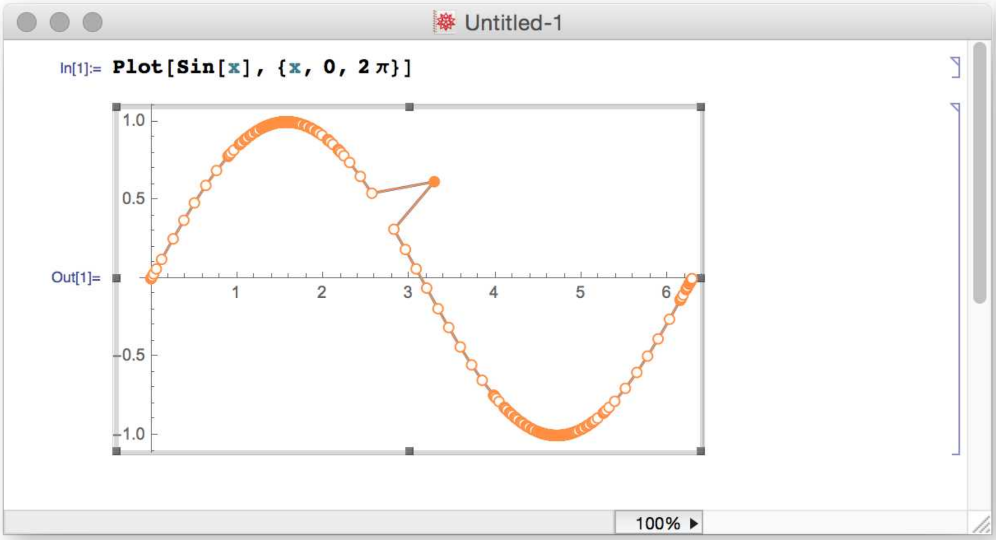


Triple-clicking can be used to expose even more of the underlying structure of a 2D graphics object. For the preceding example, triple-clicking shows the computational mesh that is used to render the curve.



```
In[1]:= Plot[Sin[x], {x, 0, 2 π}]
Out[1]=
```

The mesh shows that adaptive sampling is used, meaning that Mathematica uses more points when the curve is changing, to give a smooth and accurate representation of the plotted function. This gives a high-quality plot with the minimum rendering time. And just as double-clicking allows the plot as a whole to be moved, triple-clicking allows a single point of the mesh to be moved.

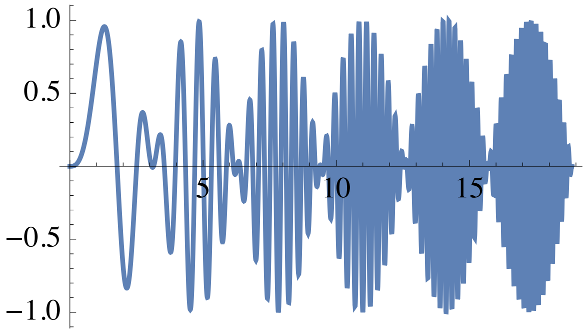


In[1]:= Plot[Sin[x], {x, 0, 2 π}]

If the result of a plot is altered—by moving the curve as a whole or by manipulating points in the mesh—then Mathematica breaks the link between the original input and the current output. You can see an example of this in the preceding screen shot, where the input and output cells no longer have a single parent cell bracket to group them together. In such a situation, if the input cell is evaluated again, a new output cell containing an unedited plot is created. This will not overwrite the edited version of the plot, so you will have two instances of (different) outputs but only a single input.

Adaptive sampling is an effective way to visualize complicated functions, especially when the plot has dense regions. Thanks to its adaptive sampling capabilities, Mathematica is able to generate attractive results even when plotting highly oscillatory functions like $\sin(x)\sin(x^2)$.

```
Plot[Sin[x] Sin[x^2], {x, 0, 6 π}]
```

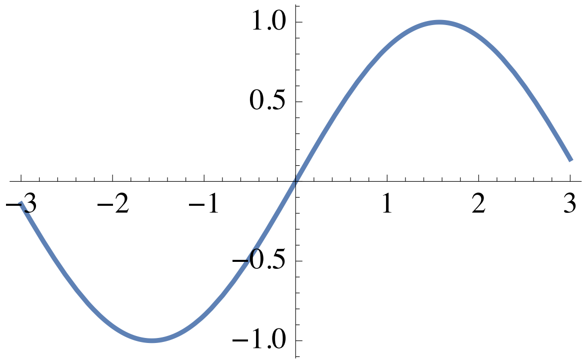


Most of the examples in this book use *x* as the independent variable when plotting a function or an expression. Mathematica expressions can use arbitrary variables, so you can just as easily use *p* instead of *x* in the preceding example. There is one important point to note, however: any symbol you choose for your variable should be undefined. If you try to use **Pi** for the variable in the preceding example, you will not get the same result, since **Pi** is a symbol that is already defined.

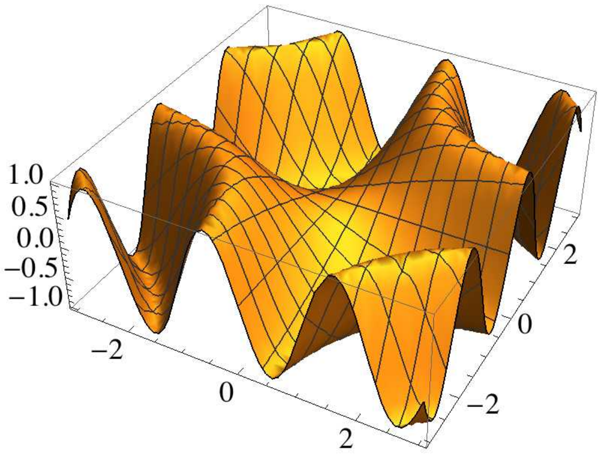
Visualizing Multivariate Functions and Expressions

The plotting commands to visualize 3D objects are very similar to their 2D analogs, in terms of both naming and syntax. The primary difference between a 2D plotting command and a 3D plotting command is that 3D plotting commands require specification of two independent variables and their corresponding domains. However, once users understand how to use a 2D command like **Plot**, then it is simple to extend that knowledge to a 3D command like **Plot3D**.

```
Plot[Sin[x], {x, -3, 3}]
```



```
Plot3D[Sin[x y], {x, -3, 3}, {y, -3, 3}]
```

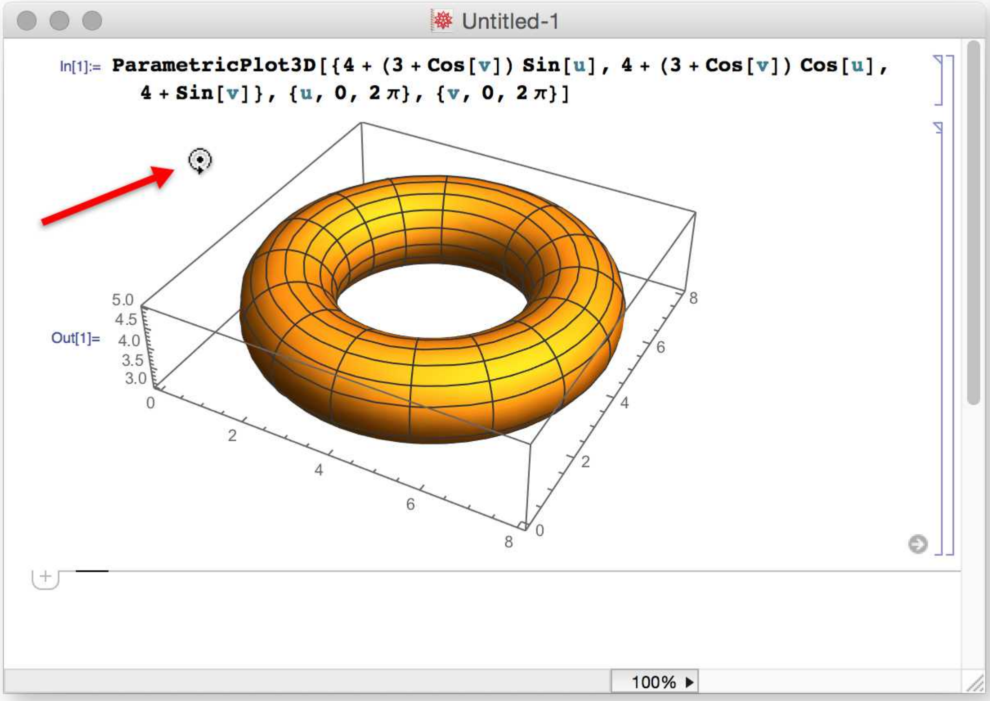


While 2D graphics allow interactivity by moving curves and manipulating the computational mesh, 3D graphics allow interactivity through rotation, panning and zooming. When the pointer is over a 3D object, it will display as a pair of twisty arrows, indicating that the object can be clicked and dragged to rotate the graphic.

Panning a graphic is also possible. When the twisty arrows are displayed and the Shift key is pressed, a set of axes is displayed, indicating that the object can be clicked and dragged to move its location within the graphics bounding box.

It is also possible to zoom into a graphics output. When the twisty arrows are displayed and the Alt key is pressed, a zoom icon is displayed, indicating that the object can be zoomed by clicking and dragging.

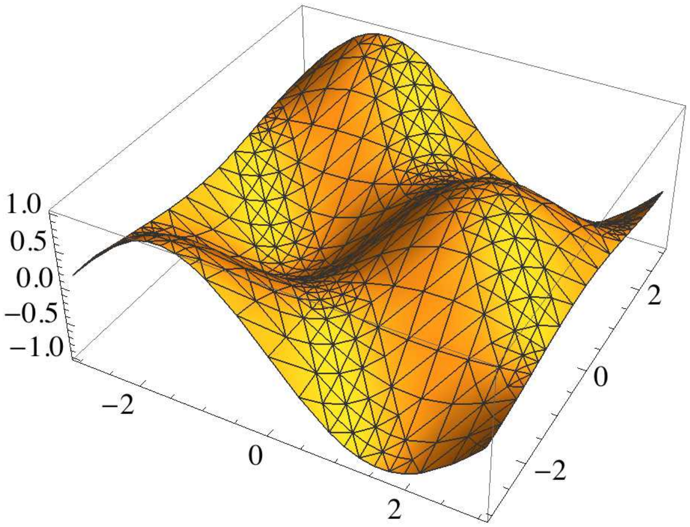
Some graphics have an additional rotation option, which is indicated by a circle-dot icon when the cursor is placed in the corner of the graphic. In this particular case, when clicking in the area shown below, the graphic can be rotated with one axis fixed.



```
In[1]:= ParametricPlot3D[{4 + (3 + Cos[v]) Sin[u], 4 + (3 + Cos[v]) Cos[u],
          4 + Sin[v]}, {u, 0, 2 π}, {v, 0, 2 π}]
```

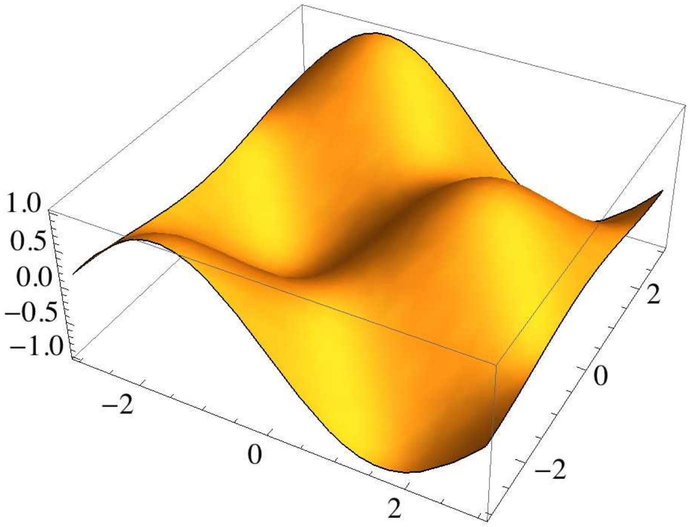
Similarly to the adaptive sampling displayed for 2D plots, surface plots are also rendered with polygons of varying size, depending on how much detail is needed for dense regions. The following plot shows the same surface plot as an earlier example, but shows all of the subdivisions that are used to render the surface instead of the mesh that is displayed by default. (The directive **Mesh → All** is an option for **Plot3D** that specifies how much of the computational mesh to show. This topic will be discussed in more detail in the next chapter.)

```
Plot3D[Sin[x] Cos[y], {x, −3, 3}, {y, −3, 3}, Mesh → All]
```



Another possible setting is **Mesh → None**, which will completely remove the mesh from the rendered graphic.

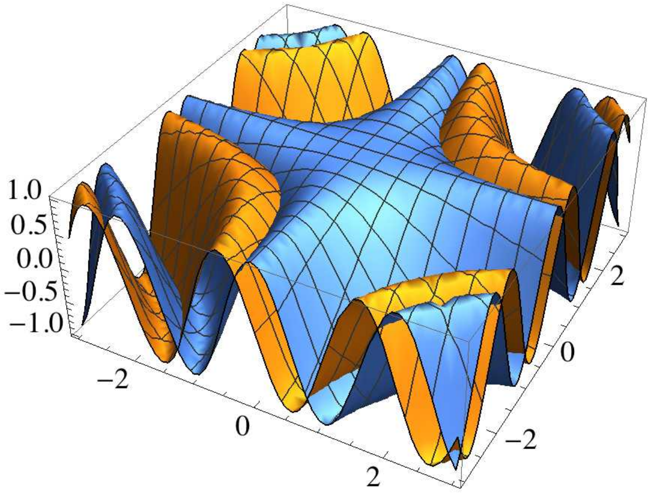
```
Plot3D[Sin[x] Cos[y], {x, −3, 3}, {y, −3, 3}, Mesh → None]
```



Plotting Multiple 3D Surfaces Together

Just like multiple curves can be plotted on the same set of axes by placing them in a list, multiple surfaces can be plotted by placing them in a list and passing them as the argument to the **Plot3D** command. As expected, Mathematica will color these surfaces differently to more easily differentiate them.

```
Plot3D[{Sin[x y], Cos[x y]}, {x, −3, 3}, {y, −3, 3}]
```

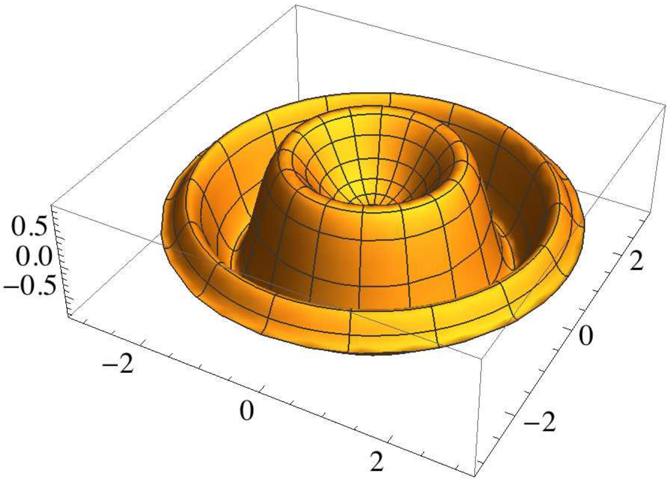


Remember: you can use **x y** (with a space), **y x** (with a space), **x*y** or **y*x** to indicate multiplication of two variables together—but **xy** and **yx**, with no spaces, are considered new variables named **xy** and **yx**, respectively.

Other Types of Visualization

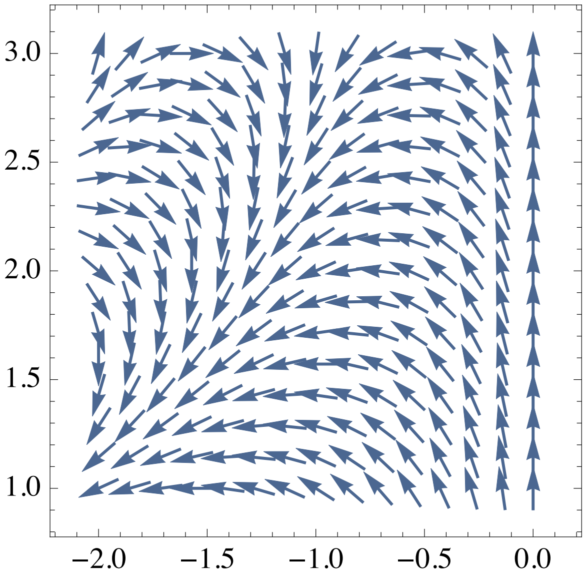
Some Wolfram Language commands are designed to create plots without a need to perform the intermediary calculations that might otherwise be necessary to visualize a particular relationship. **RevolutionPlot3D** is one such command and takes a 2D equation to generate a surface of revolution. The following example takes a curve and rotates it around the *z* axis.

```
RevolutionPlot3D[Sin[t] Sin[t^2], {t, 0, π}]
```



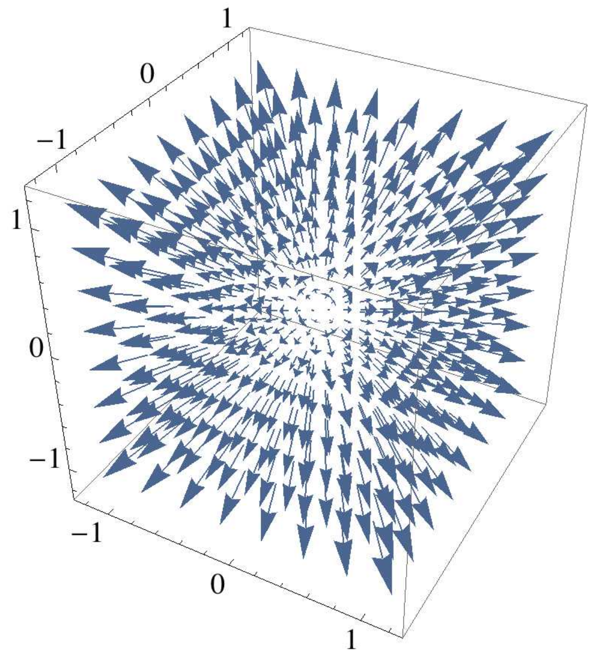
The Wolfram Language has plotting commands to visualize other specialized relationships. Plotting a vector field can be done using the **VectorPlot** function. The syntax for **VectorPlot** is very similar to the other plotting commands: the vector field is the first argument, and the domains for the appropriate variables are passed as the second and third arguments, respectively.

```
VectorPlot[{Sin[x y], Cos[x y]}, {x, −2, 0}, {y, 1, 3}]
```



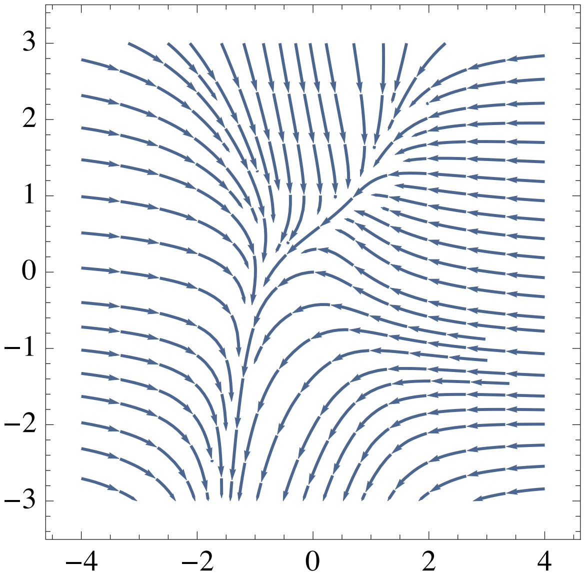
3D vector fields follow this syntax as well, with the addition of specifying the domain for the third variable.

```
VectorPlot3D[{x, y, z}, {x, −1, 1}, {y, −1, 1}, {z, −1, 1}]
```



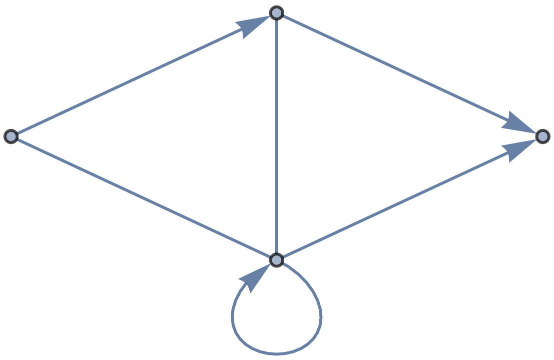
Another function, **StreamPlot**, visualizes the streamlines that show the direction of the vector field at each point. This provides useful information on the direction of the vectors in addition to the magnitude.

```
StreamPlot[{-1 - x^3 + y, x - y^2}, {x, -4, 4}, {y, -3, 3}]
```



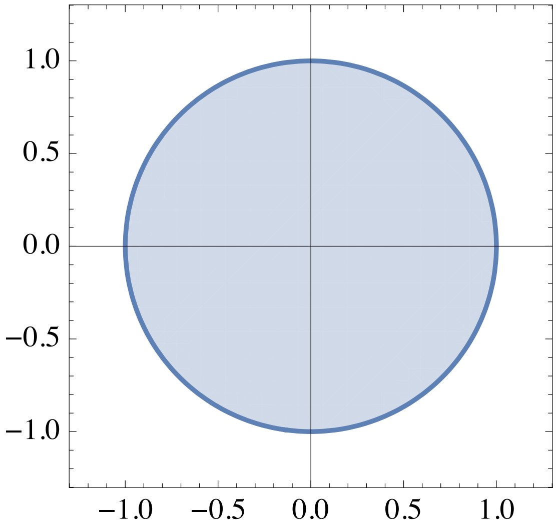
Mathematica can plot graphs and networks using commands like **Graph**. When graphs are plotted, the layout is automatically computed to be aesthetically pleasing. Graphs can be plotted with undirected edges, which are specified using the escape sequence Esc ue Esc, and directed edges, which are specified by using the escape sequence Esc de Esc. A graph is visualized by giving a list of edge specifications to the **Graph** command.

```
Graph[{1 •—• 2, 2 •—• 3,  3 •→ 1, 2 •→ 4, 1 •→ 4, 2 •→ 2}]
```



Regions like those defined by inequalities can be visualized using **RegionPlot**. The **RegionPlot** command will create a region that is filled in for values where the inequalities are true. For example, a disk can be defined using the inequality $x^2 + y^2 \leq 1$, and this inequality can be passed as an argument to **RegionPlot**.

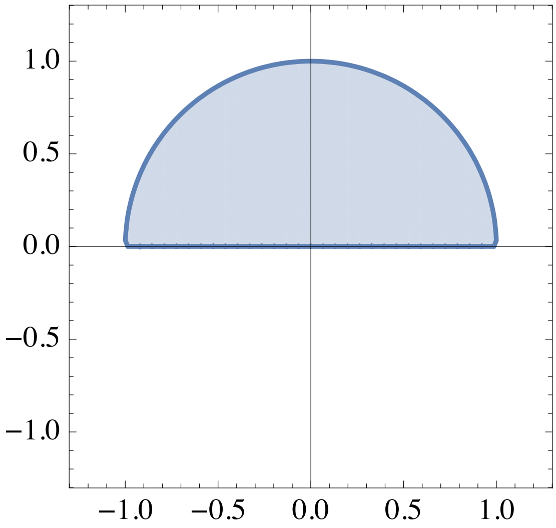
```
RegionPlot[x^2 + y^2 ≤ 1, {x, −1.25, 1.25}, {y, −1.25, 1.25}, Axes → True]
```



In an input cell, the ≥ symbol can be typed by entering **>** and then **=**, and Mathematica will reformat the characters into the single symbol **≥**. Another option is to use the escape sequence Esc>=Esc, which will print a **≥** in both input cells and other types of cells, like text cells.

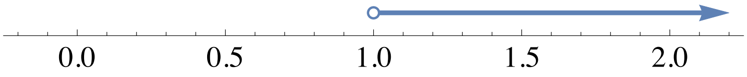
A region can be defined by multiple inequalities joined together with Boolean operators like **&&** for logical AND and || for logical OR. To plot the region where $x^2 + y^2 \leq 1$ and $y \geq 0$, the **&&** operator is used to join the two inequalities together.

```
RegionPlot[x^2 + y^2 ≤ 1 && y ≥ 0, {x, −1.25, 1.25}, {y, −1.25, 1.25}, Axes → True]
```



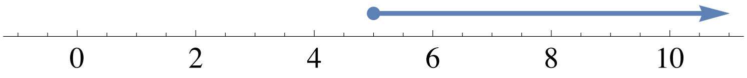
The **NumberLinePlot** command is useful for graphing points or intervals on a number line. One form of its syntax is similar to **RegionPlot**, with the first argument being an inequality and the second being a domain to plot over. For example, plotting the interval $x > 1$ from 0 to 2 will print a number line with an open circle to indicate that 1 is not included in the interval. An arrow is printed on the number line to indicate that there is no upper bound on x.

```
NumberLinePlot[x > 1, {x, 0, 2}]
```



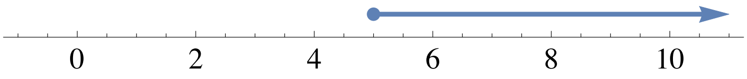
Using an inequality like $x \geq 5$ will print the number line with a closed circle to indicate that 5 is included in the interval.

```
NumberLinePlot[x ≥ 5, {x, 0, 10}]
```



NumberLinePlot can be used with the **Interval** command to specify intervals in a more formalized way. This approach obviates the need for the user to give a specific domain to plot over, which is required when an inequality is given. The preceding example used an inequality to plot the interval $[5, \infty)$ over a specific domain, and the following example uses the **Interval** command to plot the same interval but lets Mathematica choose the domain to display for viewing purposes.

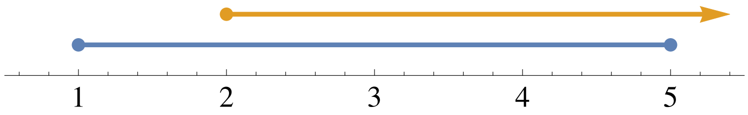
```
NumberLinePlot[Interval[{5, Infinity}]]
```



Infinity is a named symbol in Mathematica and its name and the symbol ∞ can be used interchangeably. There are other such symbols, too; one example is **Pi**, which also can be used interchangeably with its symbol π.

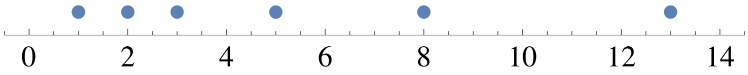
Multiple intervals can be plotted simultaneously by placing them in a list.

```
NumberLinePlot[{Interval[{1, 5}], Interval[{2, ∞}]}]
```



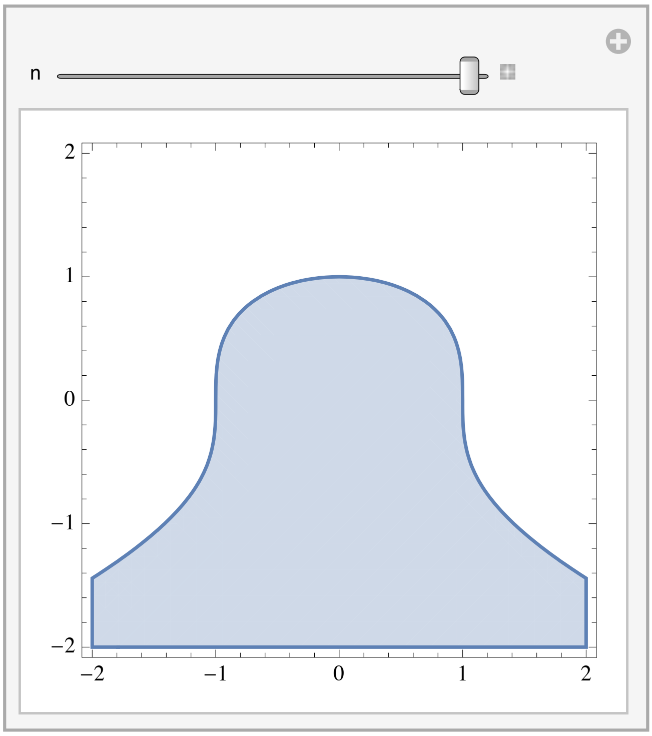
Besides inequalities and intervals, **NumberLinePlot** accepts points for its argument. The first numbers of the Fibonacci sequence could be printed using **NumberLinePlot**.

```
NumberLinePlot[{1, 1, 2, 3, 5, 8, 13}]
```



Of course, any of these visualization commands can be used with **Manipulate** to further explore the properties that govern their behavior, such as the effect that choosing different bounds has on the result when plotting a region.

```
Manipulate[
  RegionPlot[x^2 + y^3 < n, {x, -2, 2}, {y, -2, 2}],
  {n, -1, 1}]
```



Clear is used to remove all variable and function definitions from this chapter.

```
Clear[f, h]
```

Conclusion

This chapter outlined some of the most well-known commands used for visualizing functions and surfaces in two and three dimensions. There are, however, many other commands available, which can be found in the documentation.

The graphics output from Mathematica is both aesthetically pleasing and useful, making Mathematica an obvious choice for creating visualizations for presentations and publications. The next chapters will discuss how to visualize data and how to customize all types of graphics.

Exercises

1. Use free-form input to visualize $3x^2 + 7x - 9$.
2. Use free-form input to show the graph of the derivative of $3x^2 + 7x - 9$.
3. Use free-form input to visualize $3x^2 + 5y^2$.
4. Use the Wolfram Language to plot $\sin(x)/x$, where x goes from -10 to 10.
5. Use the Wolfram Language to plot $3x - 5$ and $x^2 + 1$ on the same set of axes, and where x goes from -10 to 10.
6. Use the Wolfram Language to visualize a 3D plot of $\sin(x)\cos(x)\sin(y)$, where x and y both go from $-\pi$ to π.
7. Use the Wolfram Language to create a vector plot of $3\sin(x y^2)$ and $-3\cos(x y^2)$, where both x and y go from -1.5 to 1.5.
8. Use the Wolfram Language to create a network described by the following rules: $1 \to 2, 1 \to 3, 1 \to 4, 2 \to 3, 2 \to 5, 3 \to 1, 3 \to 3, 3 \to 5, 4 \to 1, 4 \to 5$ and $5 \to 5$.
9. Use the Wolfram Language to create a number line plot to illustrate x less than $1/2$, where x goes from -5 to 5.
10. Use the Wolfram Language to create a **Manipulate** of a number line plot to illustrate x less than or equal to the parameter **number**, where **number** goes from 0 to 5 and x goes from -5 to 5.

CHAPTER 11
Visualizing Data

Introduction

The life expectancy example at the beginning of this book generated quite a few different lists that were used to compare properties of various countries. In addition, a few functions were used to plot the data points and create a histogram. These two functions, however, represent only a small sampling of the available data visualization commands in the Wolfram Language. Developing an understanding of the structure and scope of data visualization commands is useful for determining the best approach to visualizing any of a wide variety of datasets.

Visualize a One-Dimensional List of Numbers

In Mathematica, curly braces are used to represent lists, regardless of the type of elements in the list. Lists can be created several different ways: by using a command like **Table**, which creates a list based on a pattern; by importing values from an external file; or by typing in values directly.

> A lot of people use Mathematica to work with their own data, so there are entire chapters later on dedicated to the topics of importing and working with data.

To create a list manually, just create a pair of curly braces and place the values inside, separated by commas.

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

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

It is common to store lists in variables; this allows the lists to be easily referenced in subsequent calculations. For example, by assigning a list to the symbol **data**, this variable can be used in other calculations or commands where the list is needed.

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

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

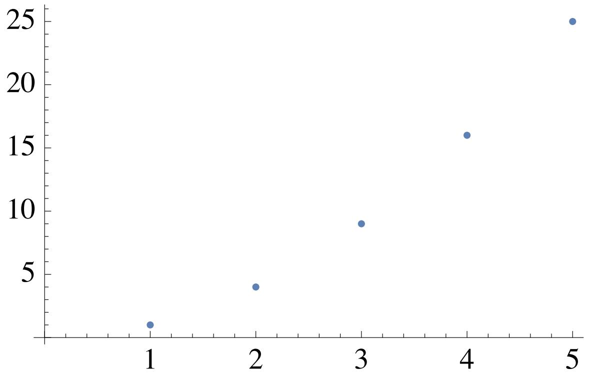
Now that **data** is defined, evaluating this symbol shows the values assigned to it.

```
data
```

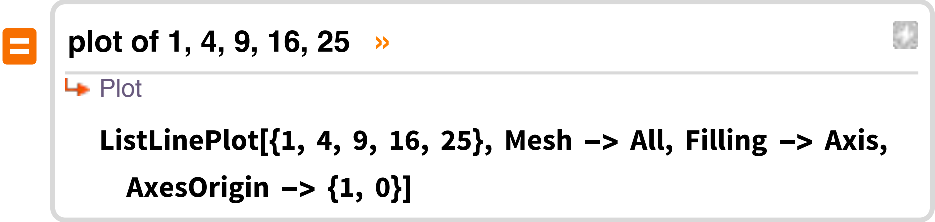
```
{1, 4, 9, 16, 25}
```

Just as the Wolfram Language has many commands available to visualize all types of mathematical functions and surfaces, so too does it have many commands available to visualize lists and datasets. One of the most common commands to visualize data is **ListPlot**, which displays the data as individual points.

```
ListPlot[data]
```

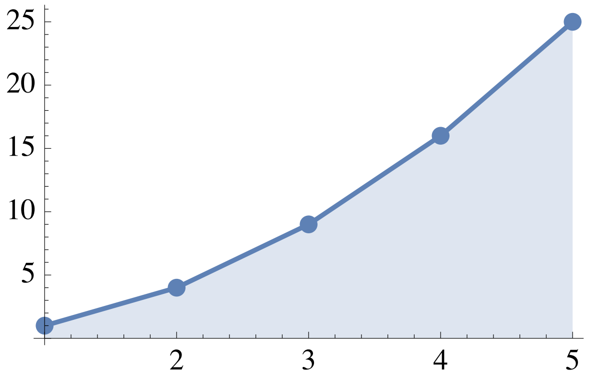


Free-form input can also be used to visualize data by including the values in the natural language command, as shown in the following example.



plot of 1, 4, 9, 16, 25 »

```
ListLinePlot[{1, 4, 9, 16, 25}, Mesh -> All, Filling -> Axis,
  AxesOrigin -> {1, 0}]
```

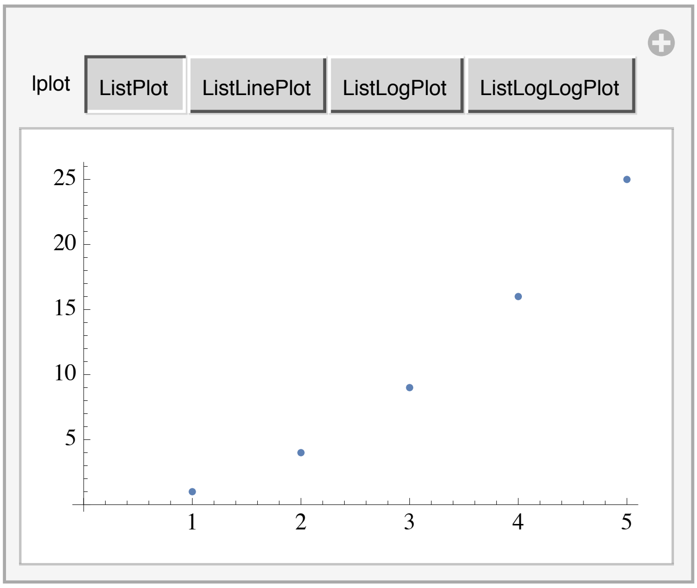


In this instance, the free-form input expression was parsed to use the **ListLinePlot** command instead of **ListPlot**, and a few options were invoked as well. **ListPlot** and **ListLinePlot** are similar, with the former plotting the values as points and the latter drawing in a line to connect the data points instead.

> You may notice that the plot options returned by free-form input look like **Mesh -> All** instead of **Mesh → All**. These are equivalent expressions and both will work. If an option is typed using the hyphen and greater-than symbol, Mathematica will automatically format those two symbols into the → symbol in **StandardForm**, which is Mathematica's default form for input cells. There are times when an expression might be shown as **InputForm** instead, which prints **->** instead of **→**. For the sake of this book, the difference is not really important except to point out that options like **Mesh -> All** and **Mesh → All** will work the same way.

Because the syntax is identical for **ListPlot**, **ListLinePlot** and other similar commands, a **Manipulate** statement is a handy way to explore the output produced by various data plotting commands. Since all the data commands require a list of values for their argument, the **Initialization** option is once again used with **Manipulate** to define some values.

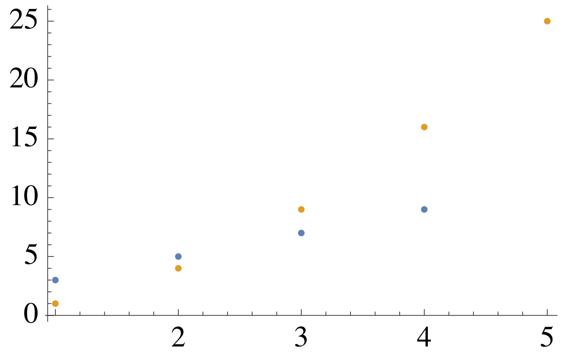
```
Manipulate[
  lplot[data],
  {lplot, {ListPlot, ListLinePlot, ListLogPlot, ListLogLogPlot}},
  Initialization :→ (data = {1, 4, 9, 16, 25})
]
```



A reminder, since we seem to be on the topic of symbols: You can create the ⧴ symbol for the **Initialization** option by using the Esc **:>** Esc sequence. Of course, you may have guessed that you could use **:>** instead of ⧴, and you would be correct.

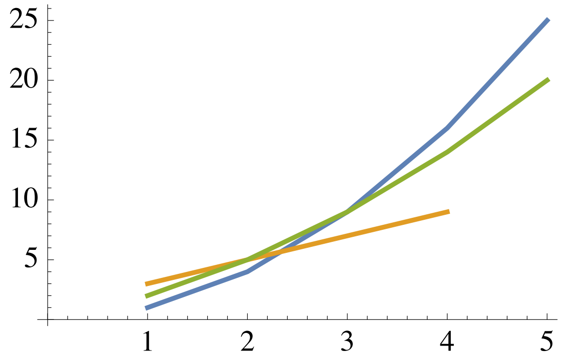
While the examples thus far have concentrated on visualizing a single dataset, the visualization commands can also be used to visualize multiple datasets. Multiple datasets can be constructed by placing several different lists into a single "parent" list that encompasses them all. This larger list can be passed to a command like **ListPlot** to visualize each of the sublists as a separate dataset. The following example shows the use of **ListPlot** to plot two datasets, which are automatically given different colors to easily tell them apart.

```
ListPlot[{{3, 5, 7, 9}, {1, 4, 9, 16, 25}}]
```



Nested lists can also be constructed from variables, where each variable represents a sublist. The following example shows the creation of three lists, each of which is assigned to a variable. The three lists are placed into a single list by referencing their variable names, and this is passed as the argument to a plotting command.

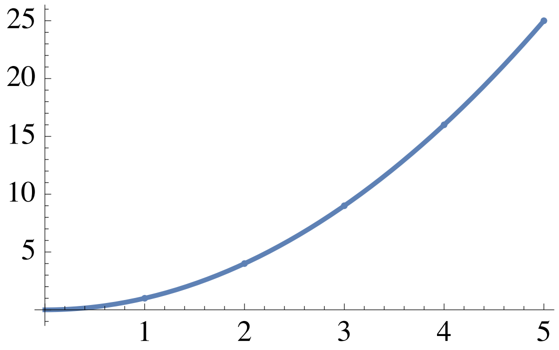
```
dataset1 = {1, 4, 9, 16, 25};
dataset2 = {3, 5, 7, 9};
dataset3 = {2, 5, 9, 14, 20};
ListLinePlot[{dataset1, dataset2, dataset3}]
```



This is another example of the automatic coloring applied to plots produced in Mathematica. The default styling for graphics is both aesthetically pleasing and useful, but it is possible to customize graphics to choose your own colors and styling elements. You can read more about that in **Chapter 12: Styling and Customizing Graphics**.

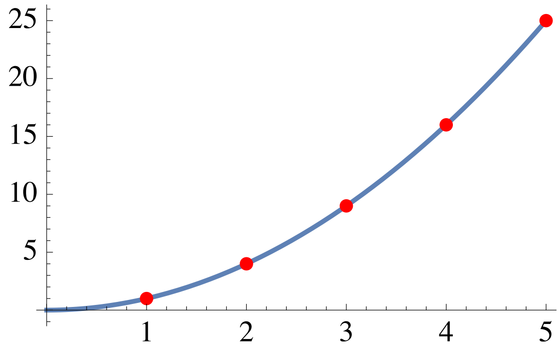
Many graphics commands can accept lists as their arguments to plot multiple functions, datasets or surfaces on a single set of axes. Sometimes, however, it is useful to create a series of individual plots and then combine them later on. The **Show** command can be used in such instances. **Show** can take multiple graphics as input and combine them into a single graphical output. The following example depicts a use of **Show** to combine a data visualization from **ListPlot** and a function visualization from **Plot**.

```
Show[Plot[x^2, {x, 0, 5}], ListPlot[{1, 4, 9, 16, 25}]]
```



All right, we know we just said that options for graphics are coming up in the next chapter, and they are, but the preceding example illustrates a situation in which it might be really useful to make the data points in **ListPlot** a different color from the curve drawn by **Plot**. We can add an option to **ListPlot** called **PlotStyle**, which will allow us to color the data points in red and change their size to medium, in order to make them stand out.

```
Show[Plot[x^2, {x, 0, 5}], ListPlot[{1, 4, 9, 16, 25}, PlotStyle → {Red, PointSize[Medium]}]]
```

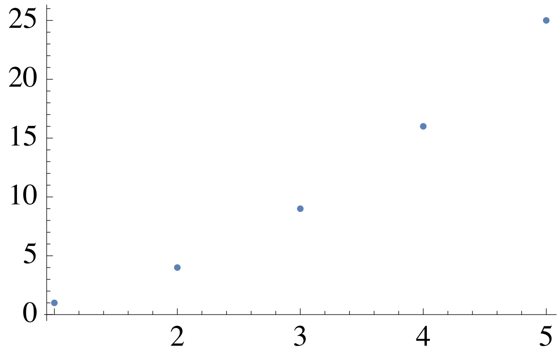


Visualize a Two-Dimensional List of Numbers

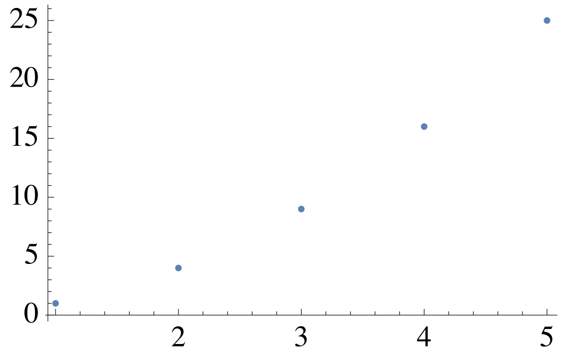
The examples in this chapter thus far have used one-dimensional datasets, but many measurements are commonly represented as two-dimensional datasets. The data visualization commands in the Wolfram Language are designed to work with both one- and multidimensional data.

When a command like **ListPlot** is given a one-dimensional list, it is assumed that the list contains *y* values that correspond to *x* values 1, 2 and so on. **ListPlot** also accepts a list of (*x*, *y*) pairs instead of single height values for *y* coordinates. The following two examples produce equivalent output.

```
ListPlot[{1, 4, 9, 16, 25}]
```

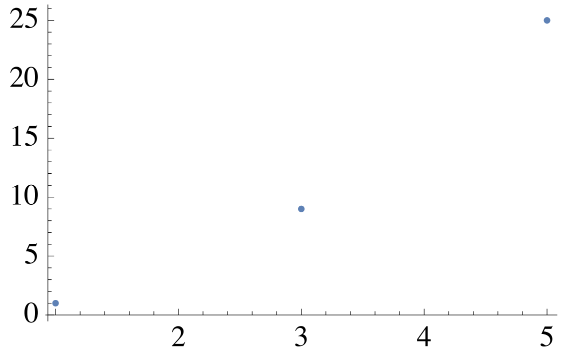


```
ListPlot[{{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}}]
```



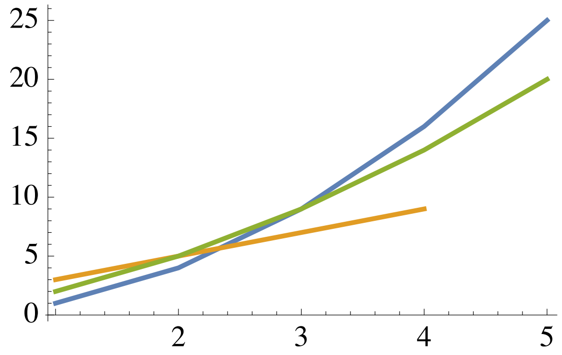
This more verbose form of **ListPlot**, however, is able to plot lists where the data is not sequential, such as a dataset of the form (1, 1), (3, 9), (5, 25).

```
ListPlot[{{1, 1}, {3, 9}, {5, 25}}]
```



The other commands from the previous section, like **ListLinePlot**, also accept two-dimensional datasets as input. The following example stores three datasets as variables and then places those variables into a list to be passed as the argument to the **ListLinePlot** command.

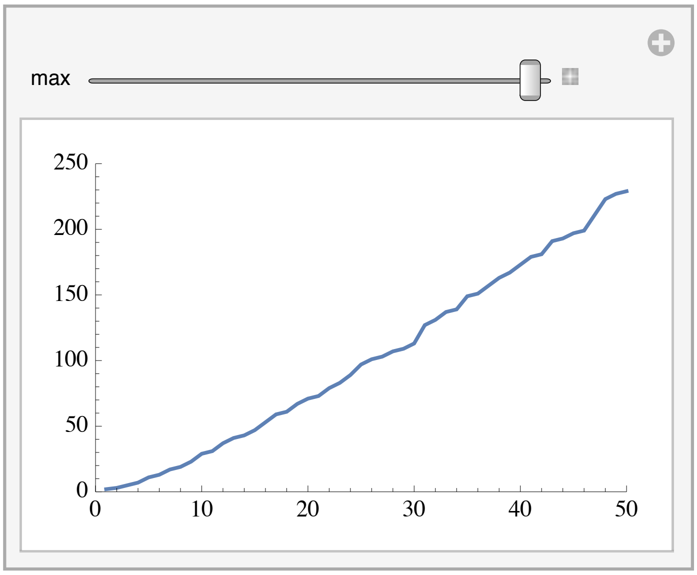
```
dataset1 = {{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}};
dataset2 = {{1, 3}, {2, 5}, {3, 7}, {4, 9}};
dataset3 = {{1, 2}, {2, 5}, {3, 9}, {4, 14}, {5, 20}};
ListLinePlot[{dataset1, dataset2, dataset3}]
```



You do not have to put your datasets in variables; you could copy and paste them all into a massive list for **ListLinePlot**, but that might get tedious to copy and paste, and it might create a really long input cell that takes up a lot of space.

Just like any other Wolfram Language functions, data visualization commands can be used with **Manipulate** to create interactive models for exploring behavior related to data. One such use of **Manipulate** might be to examine what happens as more and more values from a dataset are plotted. The following example shows how to construct a list of ordered pairs, where the *x* value is an integer and the *y* value is the x^{th} prime, like (5, 11), since 11 is the 5^{th} prime. (Primes are found using the **Prime** command.)

```
Manipulate[
  ListLinePlot[Table[{n, Prime[n]}, {n, 1, max, 1}], PlotRange → {{0, 50}, {0, 250}}],
  {max, 1, 50, 1}]
```



Recall that Animation Controls can be expanded for a **Manipulate** slider by clicking the plus icon to the right of the controller. Animation Controls include an input field where you can type in a specific value for that parameter. What happens if you choose a value that is outside the predefined range? The slider bar will be highlighted with red coloring to let you know that the value for the controller is outside the given range.

Visualize a Three-Dimensional List of Numbers

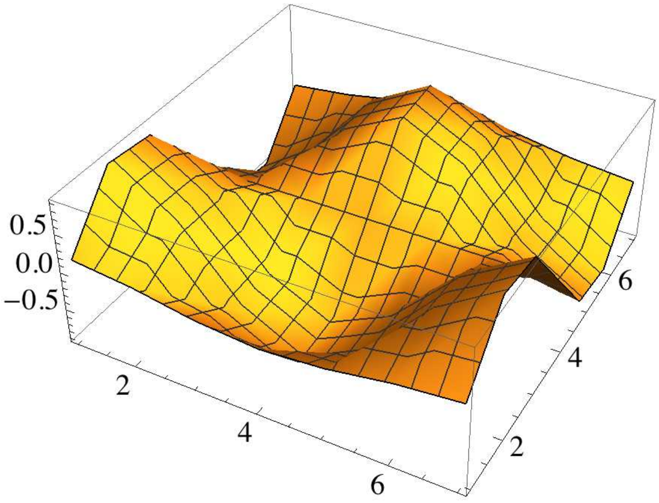
Visualizing lists or datasets in three dimensions is just as easy as visualizing them in one or two dimensions. Many of the plotting commands shown thus far have 3D equivalents, like **ListPlot** and **ListPlot3D**.

Since this section will work with larger datasets than the previous sections in this chapter, the **Table** command is used as an efficient way to generate example datasets, as in the following command.

```
data3 =
    Table[
      N[Sin[x] Cos[y]],
      {x, −3, 3, 1},
      {y, −3, 3, 1}];
```

The **data3** variable can be passed to the **ListPlot3D** command to visualize a 3D surface based on the values from that dataset, and this 3D surface has the same interactivity (rotation, panning, zooming) as other objects.

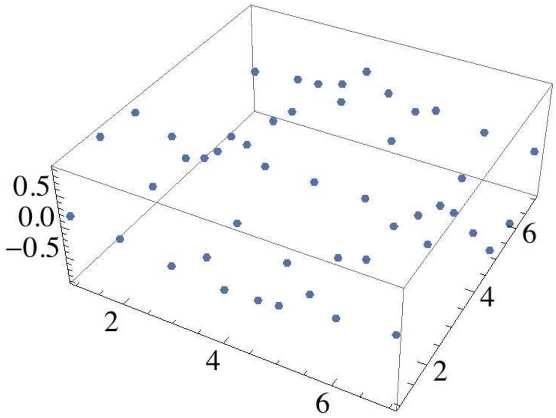
```
ListPlot3D[data3]
```



The plot above does not look smooth, and that is to be expected since a small number of points are being plotted. Reducing the step size from 1 to 0.1 in the **Table** statement that is assigned to **data3** will render more points, which in turn makes the graphic look smoother.

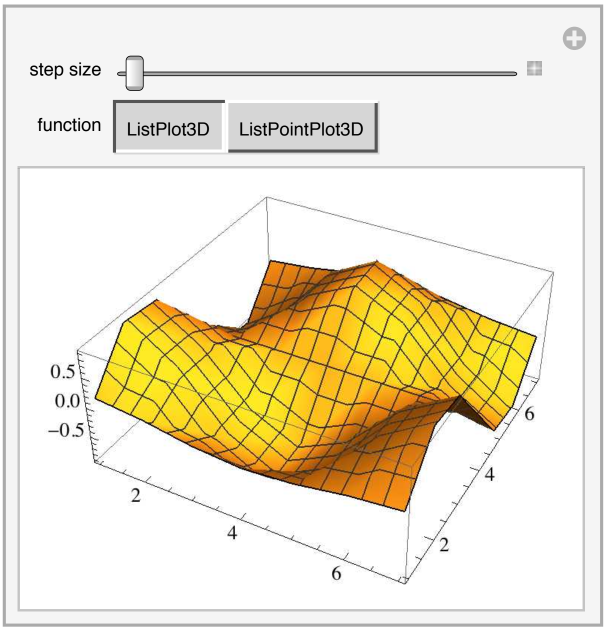
To visualize only the discrete data points and not a connecting mesh between the points, **ListPointPlot3D** can be used. The same color is applied to each point, since each represents an element of the same nested list.

```
ListPointPlot3D[data3]
```



As with previous examples, **Manipulate** provides an easy way to explore various plotting functions for 3D data. The **Table** example from earlier in this chapter is expanded to include a fourth argument to specify step size when creating the dataset. The following result allows users to explore what happens to the visualizations as more points are sampled.

```
Manipulate[
  func[Table[N[Sin[x] Cos[y], 10], {x, −3, 3, incr}, {y, −3, 3}]],
  {{incr, 1, "step size"}, 1, 0.05},
  {{func, ListPlot3D, "function"}, {ListPlot3D, ListPointPlot3D}}]
```

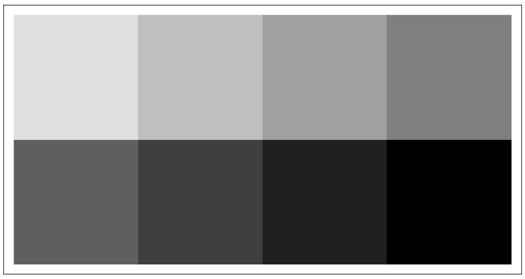


The **Manipulate** statement earlier in the chapter defined the variable **data**, which was used for the list plots and included an **Initialization** option. This **Manipulate** statement does not use any variable definitions and uses a **Table** function to generate the dataset inline instead. Since there is no variable being defined, no **Initialization** option is necessary for this **Manipulate** statement.

Visualize a Matrix

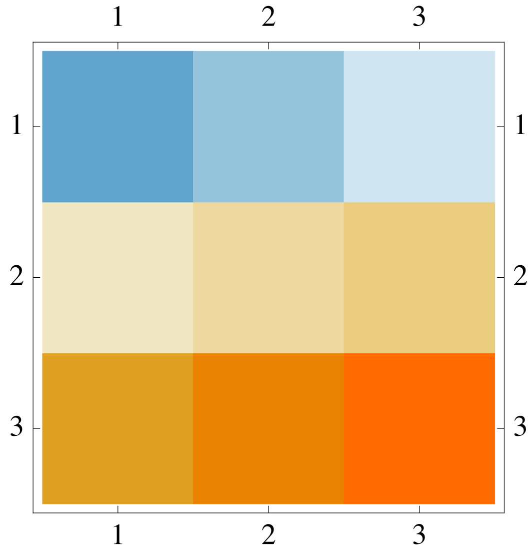
Vectors and matrices have special visualization commands. **ArrayPlot** draws a representation of an array, coloring squares that represent larger values with darker colors.

```
ArrayPlot[{{1, 2, 3,  4}, {5, 6, 7, 8}}]
```



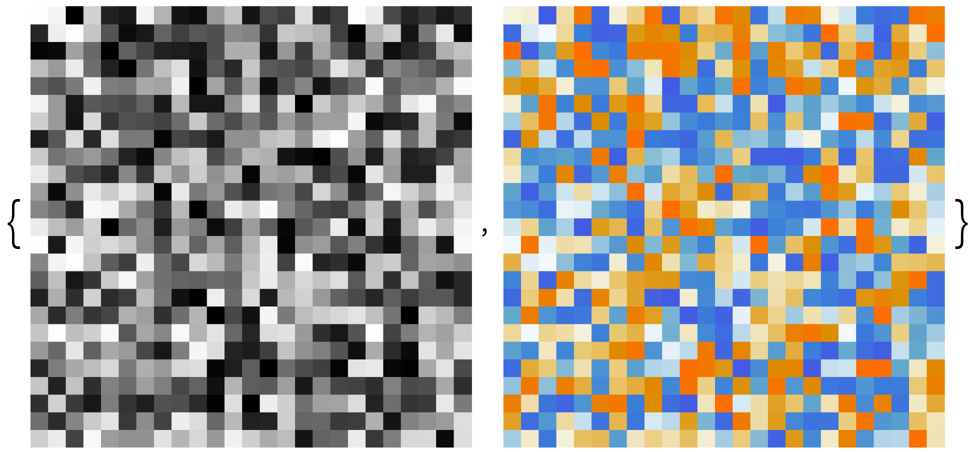
MatrixPlot follows a similar logic, where the value of an element in a matrix determines the coloration of that position in the plot, with negative values shown in cool tones, like blue, and positive values shown in warm tones, like orange. The higher the magnitude of a value, the more intense its corresponding color is for that position.

```
MatrixPlot[{{−10, −5, −1}, {2, 4, 6}, {20, 30, 40}}]
```



If an understanding of magnitude is the only consideration, then **ArrayPlot** is a fine choice; if the sign of the values is important, however, then **MatrixPlot** is the logical choice, since the coloration provides that additional information. The following example uses the **RandomInteger** command to generate a 25 × 25 matrix of random integers ranging from values – 100 to 100 and then visualizes the same dataset plotted with **ArrayPlot** and **MatrixPlot** side by side.

```
d = RandomInteger[{–100, 100}, {25, 25}];
{ArrayPlot[d, Frame → False], MatrixPlot[d, Frame → False]}
```



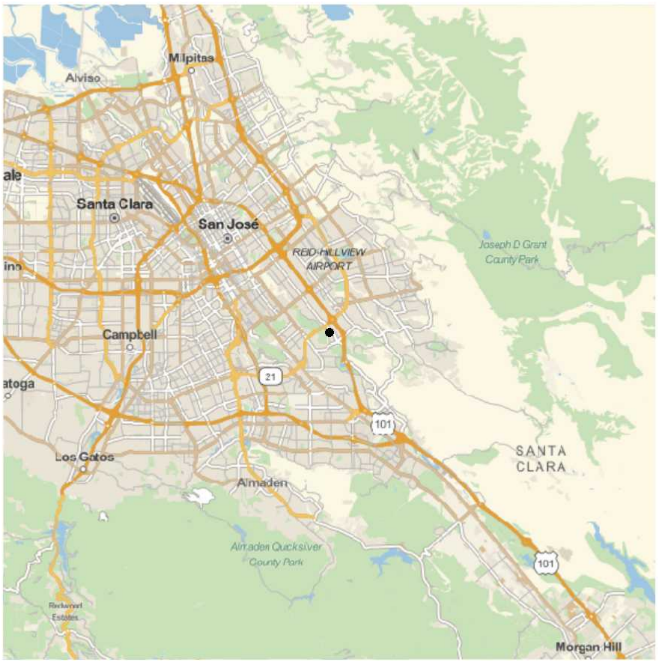
RandomInteger, **RandomReal**, **RandomPrime** and other commands are ideal for generating sets of random values. Remember that you can find a list of commands and symbols that match a given string, like "Random", by evaluating **?*Random*** in an input cell.

Graphics Related to Geography

Mathematica can create graphics to display geographical information, including highlighting cities, countries and areas of interest. The function **GeoGraphics** is a function for visualizing specialized data and is closely linked to the curated data that can be accessed from Mathematica.

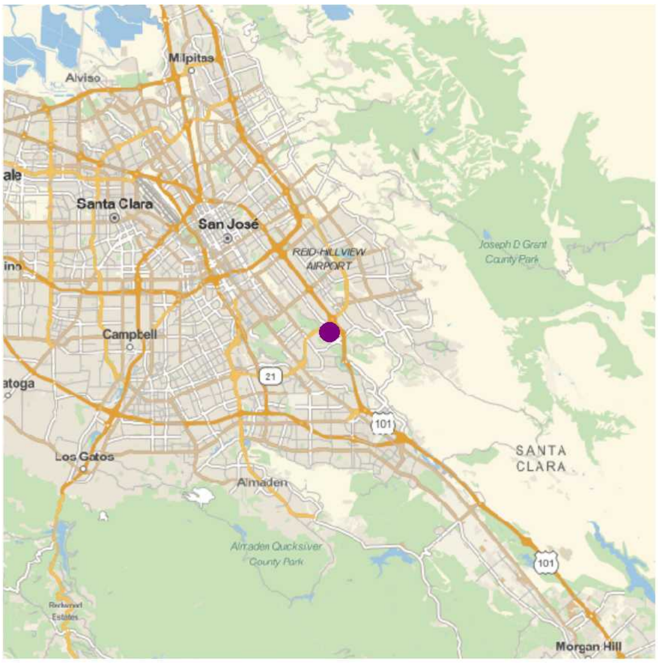
The following example uses **GeoPosition** to represent the latitude and longitude for San Jose, California, and then uses **GeoGraphics** to create a map.

```
GeoGraphics[Point[GeoPosition[{37.2969, −121.819}]]]
```



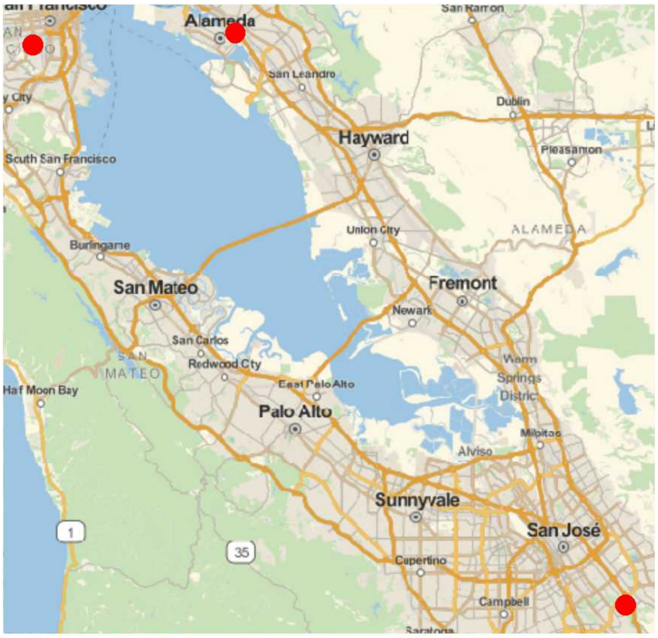
The dot on the map may blend into the background. **GeoGraphics** can accept options to change the color of a point or the size of the point on the map. Adding curly brackets is necessary to create a list of the form {*color*, *size*, *geoposition*}.

```
GeoGraphics[{Purple, PointSize[Large], Point[GeoPosition[{37.2969, −121.819}]]}]
```



Within the list in **GeoGraphics**, several sets of geopositions can be specified. The following is a map of the approximate areas of the California cities San Francisco, San Jose and Oakland.

```
GeoGraphics[{Red, PointSize[Large], Point[GeoPosition[{37.7699, −122.226}]],
    Point[GeoPosition[{37.2969, −121.819}]],
    Point[GeoPosition[{37.7599, −122.437}]]}]
```

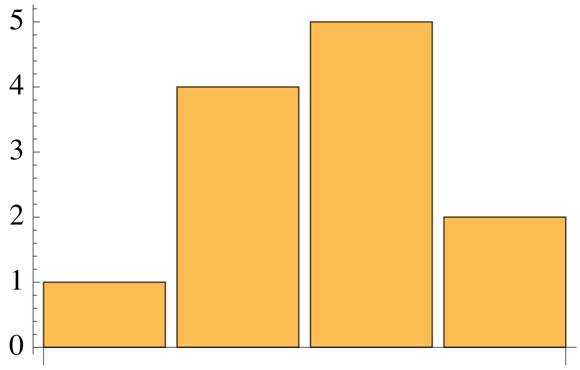


Chapter 13: Creating Figures and Diagrams with Graphics Primitives outlines how to work with graphics primitives like points and lines. These graphics primitives can be used with **GeoGraphics** to create customized maps.

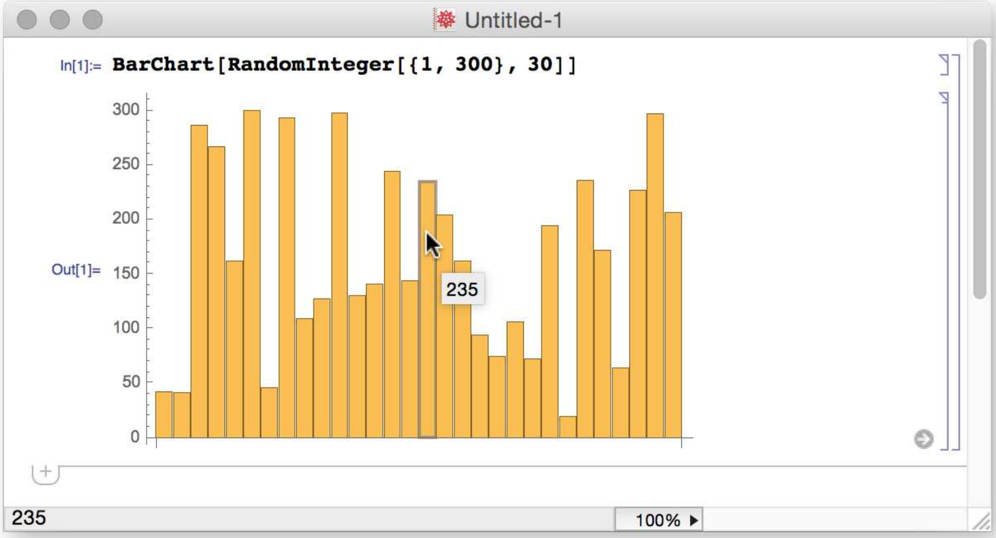
Visualize Data with Charts

The Wolfram Language has many useful charting commands that Mathematica can use to create bar charts, pie charts, bubble charts, box-and-whisker charts and more. In addition, many of these charting functions include features to give additional information, like tooltips that appear when the pointer hovers over parts of the chart. For example, **BarChart** can take a one-dimensional dataset for its argument to create a bar chart with varying bar lengths.

```
BarChart[{1, 4, 5, 2}]
```



When the pointer hovers over the bars in the output, a small tooltip window appears to list the height value of the bar. In the preceding example, tooltips are probably not necessary, since there are only a few bars and the heights are easily ascertained, but in a more complicated bar chart, as shown in the following screen shot, the tooltips can become much more useful.



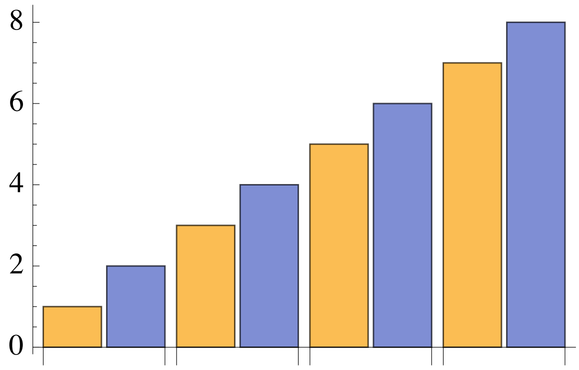
```
In[1]:= BarChart[RandomInteger[{1, 300}, 30]]
Out[1]=
```

235

Free-form input can also be used to create simple charts. It can be a great starting point, but you will want to use the Wolfram Language directly as you start to create more sophisticated or complicated graphics.

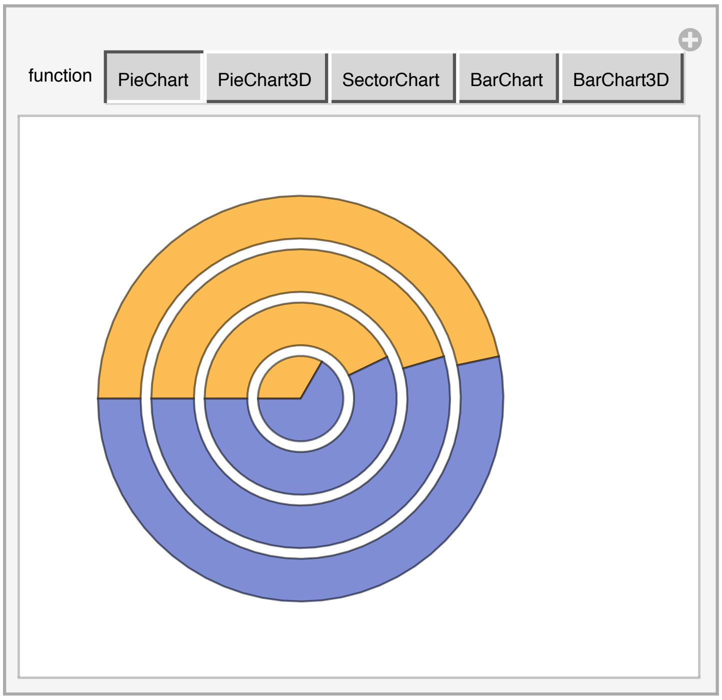
BarChart also accepts nested lists as input. Each sublist is interpreted as a list of elements that correspond to multiple datasets. In the following example, the values 1, 3, 5 and 7 are interpreted as elements from one dataset, and the values 2, 4, 6 and 8 are interpreted as elements from another dataset. As a result, the bars for the odd values are printed in one color, and the bars for the even values are printed in a different color.

```
BarChart[{{1, 2}, {3, 4}, {5, 6}, {7, 8}}]
```



Other common charts like pie charts are available, along with 3D versions of each command. As before, a **Manipulate** can be created to explore the output from applying each command to the same dataset.

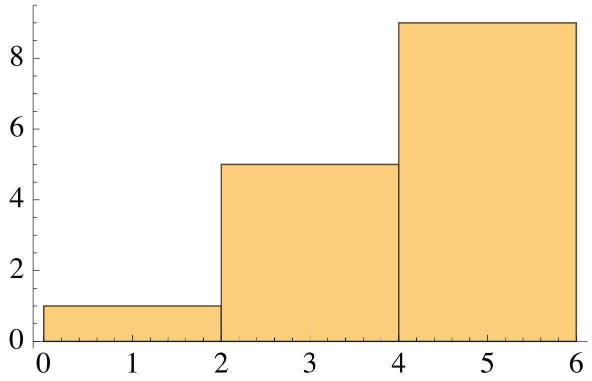
```
Manipulate[
  function[{{1, 2}, {3, 4}, {5, 6}, {7, 8}}],
  {function, {PieChart, PieChart3D, SectorChart, BarChart, BarChart3D}}]
```



You might be curious why **SectorChart3D** is not in the list. **SectorChart** takes lists of pairs, and **SectorChart3D** takes lists of triples, so they cannot use the same dataset. **PieChart** and **BarChart** (and their 3D analogs) can take lists of arbitrary length, since they are interpreted as multiple datasets.

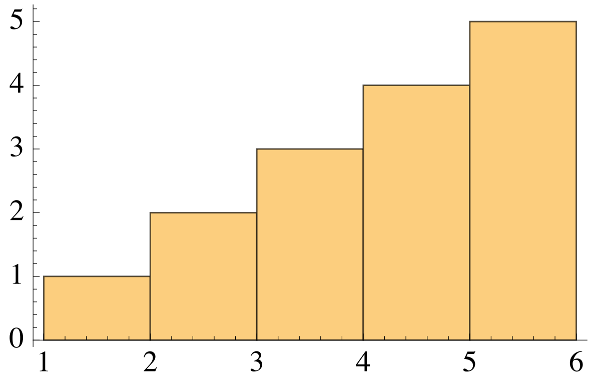
Another common data visualization command is **Histogram**, which takes a list of values, bins the values and charts the results. The following example takes a dataset and shows the quantity of values between 0 and 2, the quantity of values between 2 and 4 and the quantity of values between 4 and 6 as three separate bars. Like the other charting commands shown thus far, **Histogram** will show a tooltip with a bar's value when the pointer hovers over the bar.

```
Histogram[{1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5}]
```



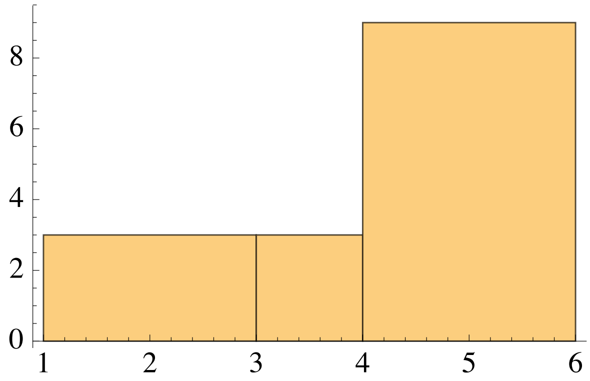
Histogram chooses the bin width automatically unless a second argument is given to control the bin width. The following example charts the quantity of values from 1 to 6 in increments of 1.

```
Histogram[{1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5}, {1, 6, 1}]
```



The bin widths do not have to be equal. The following example chooses unequal widths of 1 to 3, 3 to 4 and 4 to 6. The resulting visualization represents these unequal bin sizes with wider rectangles.

```
Histogram[{1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5}, {{1, 3, 4, 6}}]
```



Manipulate can be used to adjust the bin widths of a histogram, which can be useful with a large dataset where it is difficult to guess the most useful bin width.

Similar to a histogram, **WordCloud** is a function that can be used to visualize the most common elements in a list. **WordCloud** takes a list of any type of elements and displays the elements in a format where their size is based on the number of occurrences in the list. The most commonly-recurring elements are large, and elements that occur less frequently are small.

In the following example, the variable **cloudNum** is used to store a list of 10 random integers.

```
cloudNum = RandomInteger[{1, 10}, 10]
```

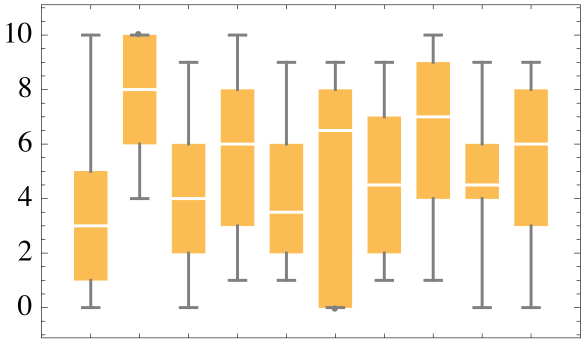
```
{6, 9, 6, 3, 7, 2, 5, 10, 7, 3}
```

```
WordCloud[cloudNum]
```

Although **WordCloud** works with lists containing any type of expressions, this function is most commonly used with lists containing strings. If you are interested in processing and visualizing text, such as from a spreadsheet, then you will want to be sure to read **Chapter 19: Importing and Exporting Data**.

A final example of a charting command is **BoxWhiskerChart**, which has tooltips to show the minimum, 25% quantile, median, 75% quantile and maximum of the dataset presented when mousing over the individual bars in the chart. This type of visualization eliminates the need to compute those measures separately. The following example uses **RandomInteger** to construct a dataset of ten rows and ten columns, with each value being a random integer between 0 and 10, and visualizes the result as a box-and-whisker chart.

```
BoxWhiskerChart[RandomInteger[{0, 10}, {10, 10}]]
```



Clear is used to remove all variable and function definitions from this chapter.

```
Clear[data, dataset1, dataset2, dataset3, data3, d, cloudNum]
```

Conclusion

There are many different data visualization commands in the Wolfram Language, used for projects from creating simple bar charts to visualizing scatter plots and 3D surfaces described by datasets. The common thread that ties them all together is that the consistent language design of the Wolfram Language means that one dataset can be passed to several different visualization commands to create the specific graphical output that is required by the problem at hand. The dynamic elements of data visualization commands—interactivity, tooltips, automatic computation of statistics—make Mathematica an ideal platform for data visualization work.

Exercises

1. Use free-form input to visualize the dataset {1, 6, 10, 13, 15}.
2. Use the resulting Wolfram Language expression from Exercise 1 as a starting point to visualize just the points of the dataset without a line connecting the points, and also remove the filling and place the axes origin at (0, 0).
3. Define the variable **res1** as {{0, 0}, {1, 1}, {2, 3}, {3, 6}, {5, 15}}, define the variable **res2** as {{0, 3}, {1, 7}, {2, 10}, {3, 13}, {4, 15}, {5, 16}} and suppress the output from both variable assignments. Then, use the Wolfram Language to create a single plot that visualizes both datasets.
4. Use the Wolfram Language to create a **Manipulate** statement to visualize the dataset {{1, 1}, {2, *i*}, {3, 10}}, where *i* can be varied from 1 to 15. Include an option to specify the plot range between 0 and 20.
5. Use the Wolfram Language to define a variable named **points17** that contains a dataset of points of the form $i^3 - i^2 + 1$, where *i* ranges from −50 to 50, and suppress the output from this variable assignment. Then, use a Wolfram Language command to visualize the set of points.
6. Use the Wolfram Language to define a variable named **points21** that contains a dataset of points of the form $\cos(a) + \sin(b)$, where *a* and *b* range from −10 to 10, and suppress the output from this variable assignment. Then, use a Wolfram Language command to visualize the set of points and include an option to remove the mesh from the graphic.
7. Repeat Exercise 6 but use a step size of 0.1 when creating the dataset of points. This will create 10 times the number of data points, which in turn will create a much smoother graphic when the dataset is visualized.
8. Create a histogram to visualize the dataset {1, 6, 10, 13, 15, 25}, and specify a bin width of 12.

9. Use the dataset from Exercise 8 and bin the values into bins of 1 to 5, 5 to 15 and 15 to 30.

10. Use the **ThermometerGauge** and **Manipulate** commands to visualize a gauge for the expression 6 *x*, where the gauge values range from 0 to 20 and *x* can range from 1 to 3 in steps of 0.1.

CHAPTER 12

Styling and Customizing Graphics

Introduction

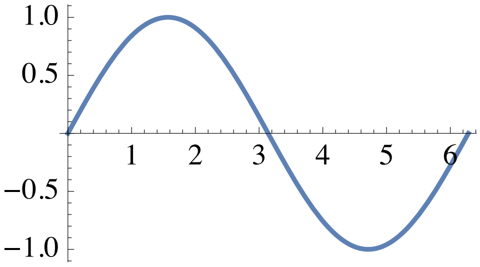
Many of the graphics examples so far have used Mathematica's default options for styling the results. While the default styling is designed to be useful and very aesthetically pleasing, certain graphics may require a degree of customization to emphasize information, highlight features or conform to style guidelines. This chapter outlines the scope for how graphics may be customized, including changing colors, themes, tick marks, labels, frames and viewing windows, or adding legends, grid lines, filling or textures.

Using Options with Graphics

Mathematica gives users a great deal of control over the appearance of results. This control is harnessed through the use of options, which are additional arguments to commands that can be used to control the behavior of results, like changing colors, themes and styling elements.

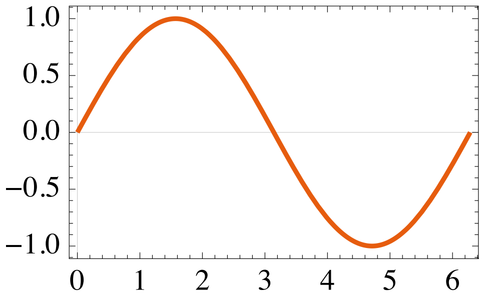
For example, the default appearance of the plot of sin(x) from 0 to 2 π looks as follows.

```
Plot[Sin[x], {x, 0, 2 π}]
```



The use of an option, like **PlotTheme**, can change the default color. Options are added to the command syntax and are separated from the other arguments by commas. Options take the form **OptionName→OptionValue**, like this example that applies a scientific theme to the appearance of a plot.

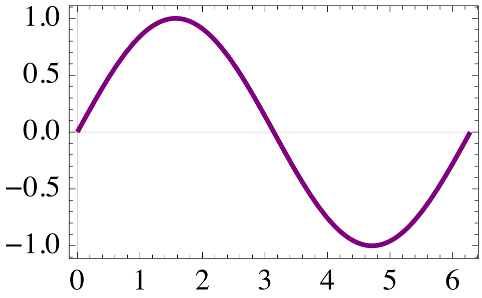
```
Plot[Sin[x], {x, 0, 2 π}, PlotTheme → "Scientific"]
```



As you typed the preceding command, you were presented with a visual list of choices for different types of plot themes. Mathematica's Code Assistance features extend to autocompletion on option values, which is incredibly handy when adding options to plots or other types of commands.

Multiple options can be used with a single command. For example, to change the appearance of the curve itself, the option **PlotStyle** can be used to specify that the curve should be purple instead of the default chosen by the plot theme. When multiple options are added to a plot, they are separated by commas.

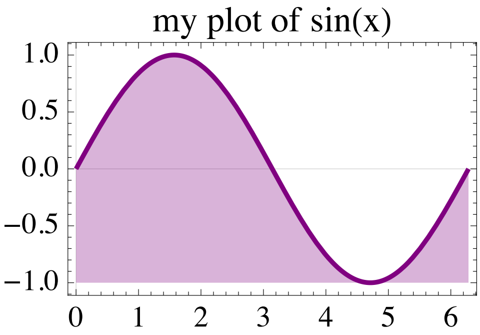
```
Plot[Sin[x], {x, 0, 2 π}, PlotTheme → "Scientific",
  PlotStyle → Purple]
```



You may be wondering why the option setting for **PlotTheme** is given as a string (**"Scientific"**), but the option setting for **PlotStyle** is not (**Purple**). Some option settings are names, and they are given in the form of strings. Others, like **Purple**, are actually symbols that represent something. You can check this by evaluating **Purple** in an input cell; you will see that its output is a swatch representing RGB values for the color purple.

Notice how the other elements of this particular **PlotTheme** setting, like drawing a frame around the plot, were maintained even though a second option, **PlotStyle**, was used to override the color. And still more options can be applied, such as to add filling and a plot label.

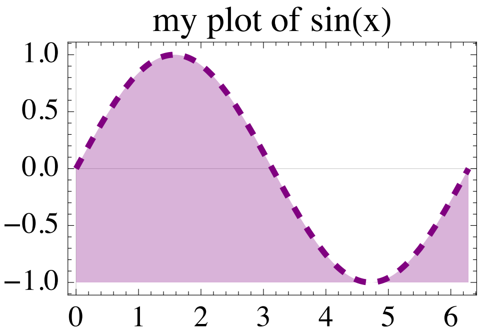
```
Plot[Sin[x], {x, 0, 2 π}, PlotTheme → "Scientific",
  PlotStyle → Purple, Filling → Bottom, PlotLabel → "my plot of sin(x)"]
```



If you add lots of options to a command, you may find it easier to introduce line breaks to separate each option, or group of options, on separate lines. That is perfectly fine and a matter of personal preference. Mathematica will ignore whitespace in input cells, so feel free to format your code in a manner that makes the most sense to you.

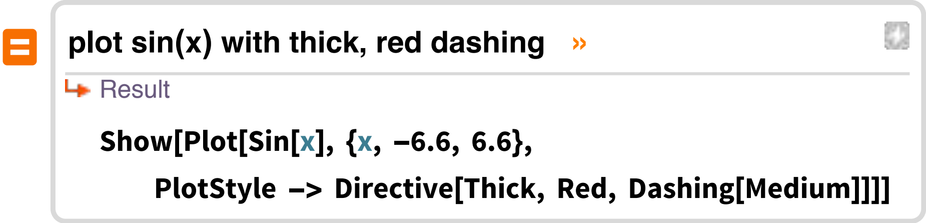
Some options can take multiple arguments themselves, in which case they are enclosed in a list. One example is **PlotStyle**, which allows the user to specify different types of styling controls, like color, thickness and dashing, in a single command. The desired options values are passed in the form of a list enclosed by curly braces.

```
Plot[Sin[x], {x, 0, 2 π}, PlotTheme → "Scientific",
  PlotStyle → {Purple, Thick, Dashed}, Filling → Bottom, PlotLabel → "my plot of sin(x)"]
```



Similar to **Purple**, **Dashed** and **Thick** are symbols themselves, and evaluating them in an input cell will reveal their underlying structure. This chapter will also outline how to choose specific shades of color, thickness and dashing styles to employ an even higher degree of customization.

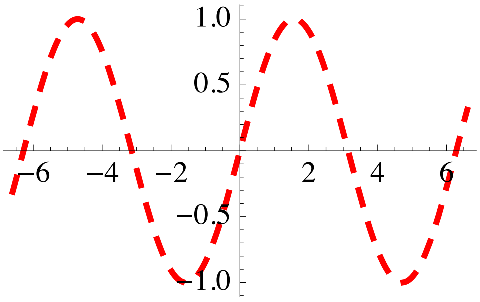
Free-form input can also be used to specify options to a limited degree. The following example uses free-form input to adjust the appearance of a plot.



plot sin(x) with thick, red dashing »

Result

```
Show[Plot[Sin[x], {x, −6.6, 6.6},
    PlotStyle −> Directive[Thick, Red, Dashing[Medium]]]]
```



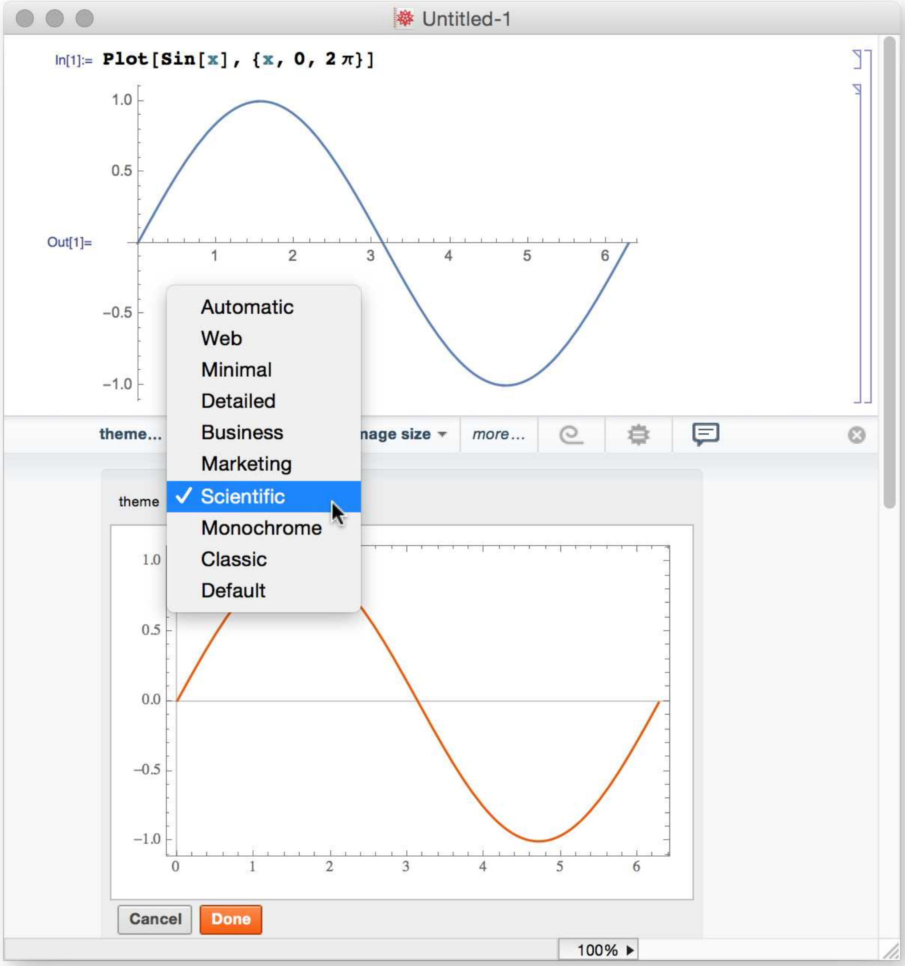
Not every option can be specified using free-form input, but it can sometimes work for simple things like colors and dashing. Of course, once the syntax for a specific Wolfram Language option name is known, it is generally better to use that option directly.

Multiple approaches can be used to apply options to graphics, and four of them will be discussed in this chapter: interactive customization with the Suggestions Bar, interactive customization with Drawing Tools, creating option templates with the Assistant palettes and direct use of options with Wolfram Language commands.

Interactive Customization with the Suggestions Bar

As introduced in an earlier chapter, the Suggestions Bar automatically recommends additional commands to apply once an operation is performed. When an output contains graphics, the Suggestions Bar may recommend common styling options, such as adding a frame, changing the axes or styling the plot itself.

A common option for plots is **PlotTheme**, which changes multiple styling elements at the same time. Changing the plot theme is one of the suggestions that may be returned by the Suggestions Bar when the output of a plot is selected. Clicking **"theme..."** in the list of suggestions brings up a menu from which different settings for **PlotTheme** can be previewed, and once a desired one is chosen, clicking **Done** will apply the option and print its Wolfram Language syntax.



```
In[1]:= Plot[Sin[x], {x, 0, 2 π}]
```

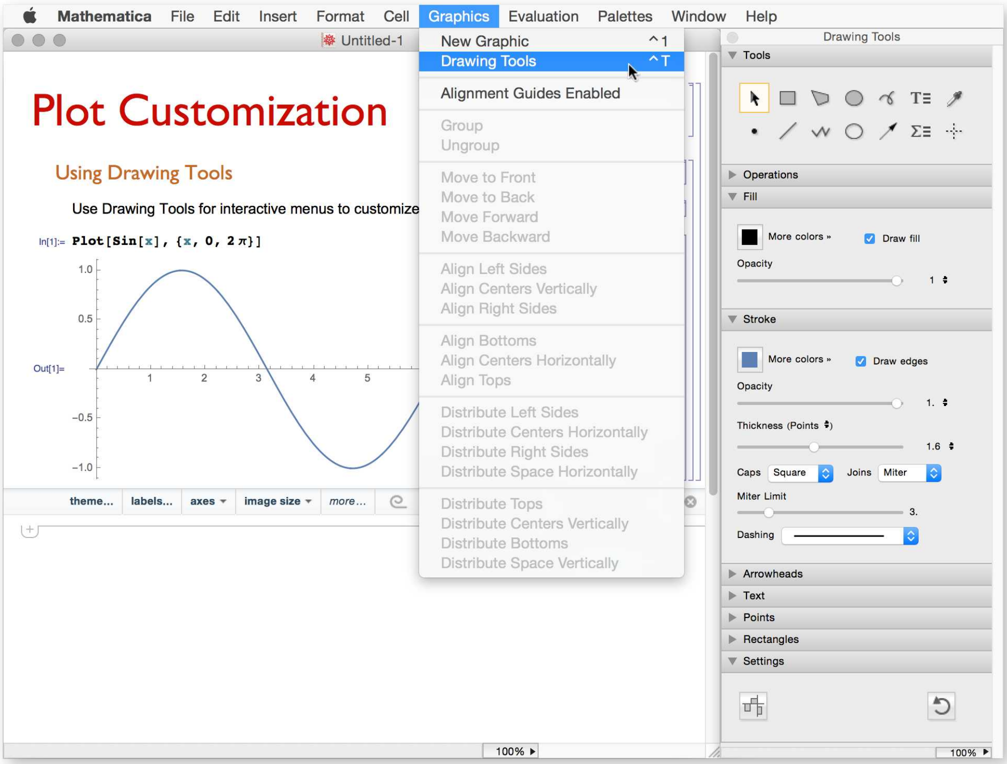
theme: Scientific

Some suggestions from the Suggestions Bar will open interactive menus to add elements like labels, and others will have choices for changing axes and background colors, adjusting sizes, and changing thickness and dashing of curves.

You may notice that applying some graphics options from the Suggestions Bar will result in the creation of new cells that use the **Show** command. If you end up with a series of input and output cells and you wish to condense them, you can click the icon to roll up inputs, which will compress the inputs into a single cell with all of the options applied.

Interactive Customization with Drawing Tools

For those who prefer a point-and-click experience, the **Drawing Tools** palette provides an excellent interface for customizing graphics. Click the **Graphics** menu and choose **Drawing Tools** to launch a palette with a variety of buttons for operations that can be used to edit an existing graphic, such as by changing its color and thickness.



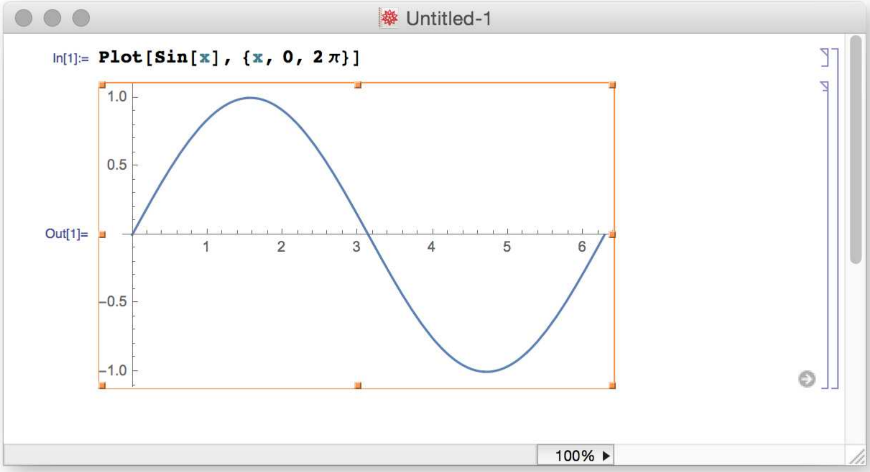
Plot Customization

Using Drawing Tools

Use Drawing Tools for interactive menus to customize

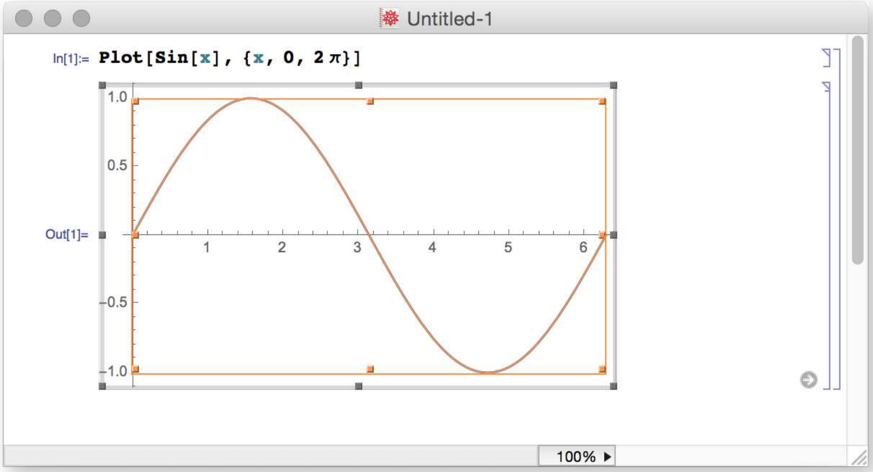
```
In[1]:= Plot[Sin[x], {x, 0, 2π}]
```

To edit an existing graphic, it must first be selected. Single-clicking a graphic will draw an orange bounding box around the entire output with square markers on the perimeter that can be used for resizing, but single-clicking will not select the graphic for editing. Double-clicking a graphic will select it for editing; in the case of a plot, this means double-clicking the curve itself.



```
In[1]:= Plot[Sin[x], {x, 0, 2π}]
```

Notice the orange bounding box that surrounds the entire graphic when the graphic is single-clicked.



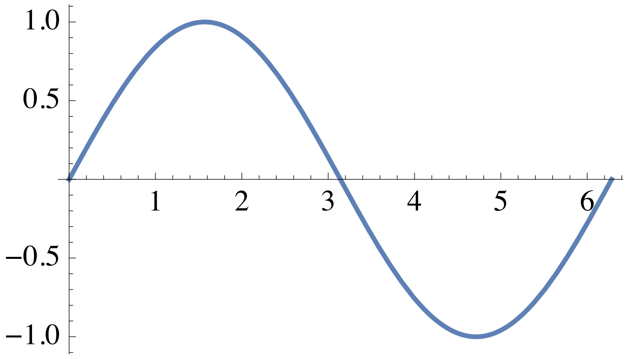
```
In[1]:= Plot[Sin[x], {x, 0, 2π}]
```

Notice both the orange and gray bounding boxes that surround and highlight the selected graphic (in this case, the plot) when the graphic is double-clicked.

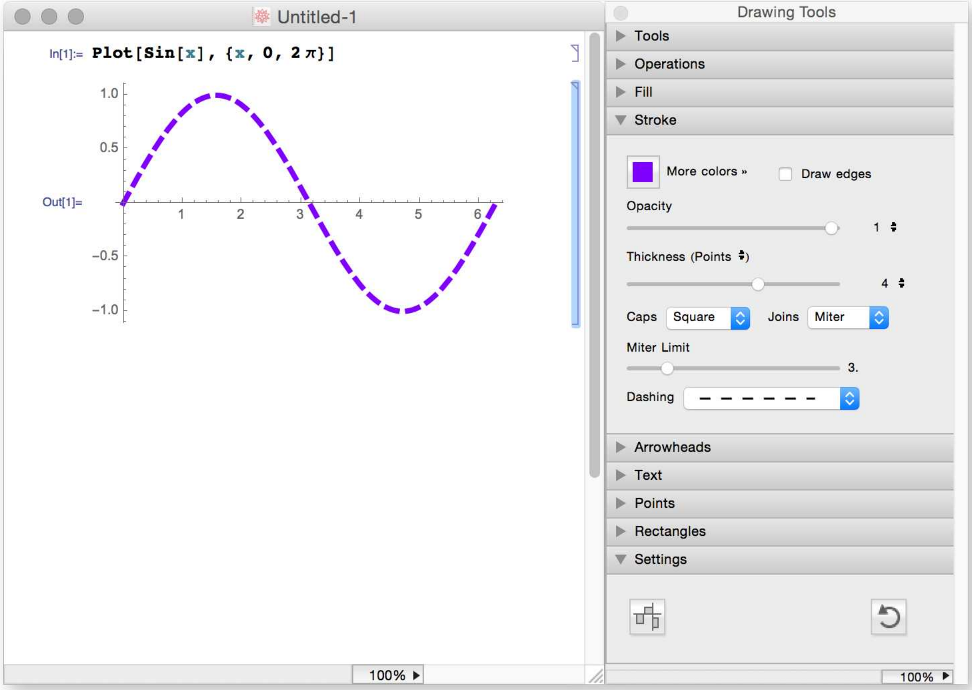
What is the benefit of single-clicking a graphic if it can only be edited after double-clicking? Double-clicking allows editing, which includes customization with **Drawing Tools**, but it is also possible to inadvertently move curves or objects in the graphic such that they no longer correspond to the original equation. Single-clicking only selects the graphic as a whole and not any of the individual elements; as such, if you single-click, you cannot accidentally move the graphic from its original position. A good rule is: if you want to copy and paste, then single-click; if you want to edit elements, then double-click.

Once the graphic is selected, **Drawing Tools** can be used to customize its appearance. For example, the tools in the **Stroke** section can be used to change color, opacity, thickness and dashing.

```
Plot[Sin[x], {x, 0, 2 π}]
```



Using the **Drawing Tools** palette, it is straightforward to change the above plot so that it is purple, thicker and dashed. The resulting plot will resemble the following screen shot.



```
In[1]:= Plot[Sin[x], {x, 0, 2 π}]
```

□ Draw edges

Opacity 1

Thickness (Points) 4

Caps Square Joins Miter

Miter Limit 3.

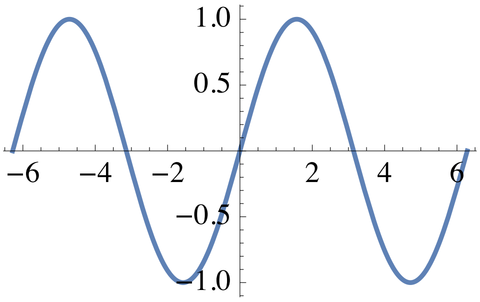
Dashing – – – – – – –

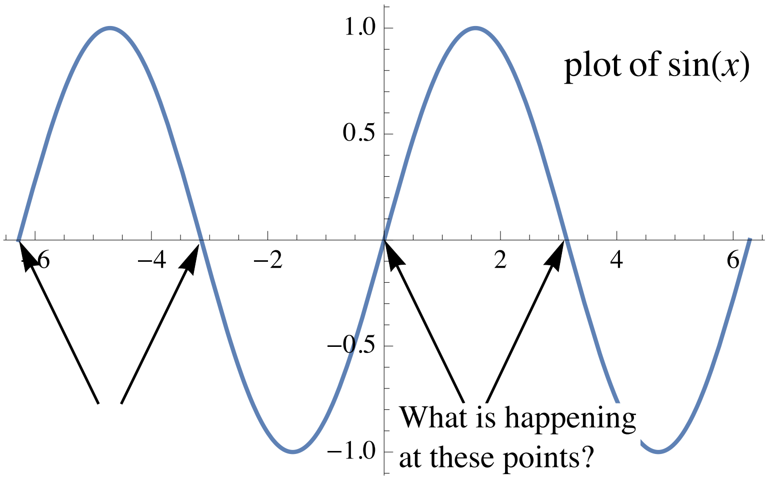
Graphics can be reset to their default appearance by selecting them and clicking the reset button in the **Settings** section of **Drawing Tools**. This comes in quite handy, especially since changed settings like thickness will persist and be applied to new components added to the graphic; if that happens, selecting the newly added component and clicking the reset button will apply default settings, which will remove customization, like dashing.

What are the advantages to using the Suggestions Bar instead of **Drawing Tools**, or vice versa? One advantage of the Suggestions Bar is that it prints the Wolfram Language syntax for the options it applies, while **Drawing Tools** does not.

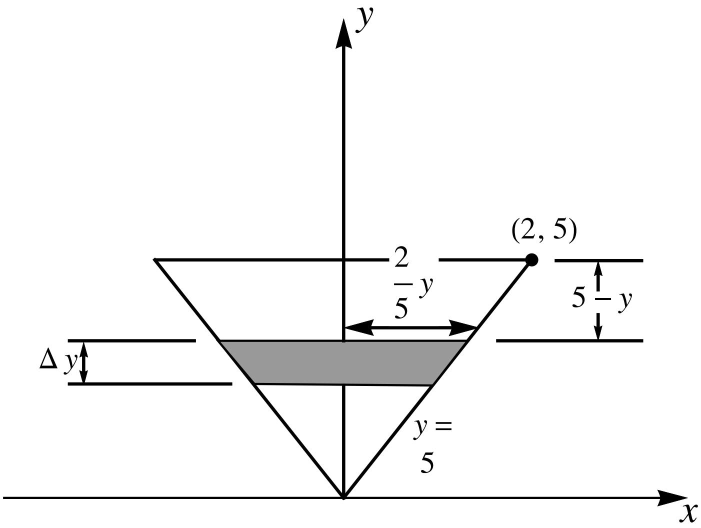
Besides editing graphics, **Drawing Tools** can be used to add other graphical and textual components to graphics. This allows graphics to be interactively annotated with arrows, descriptions and mathematical typesetting, which can be extremely useful when preparing graphics for handouts, reports and publications. The following example shows a plot that has been annotated by adding arrows and text elements with **Drawing Tools**.

```
Plot[Sin[x], {x, −2 π, 2 π}]
```





Drawing Tools can also be used to construct entire diagrams or figures. To make a new graphic, choose **New Graphic** from the **Graphics** menu to create a blank drawing area. Resize the canvas if desired and use **Drawing Tools** to add and edit components. The following shows an example of a diagram constructed with **Drawing Tools**.



When adding lines in a new drawing, hold down the Shift key when dragging to draw perfectly straight lines. Also pay attention to the blue lines that pop up as you drag objects around; these are alignment guides and they can help you align, center and arrange components. (If you do not see the alignment guides, open the **Settings** section of **Drawing Tools** and click the alignment guides button on the left to turn them on.)

Each component added by **Drawing Tools** can be individually selected for further editing, moving or deletion. This makes it easy to tweak an existing figure by changing elements or adding new ones.

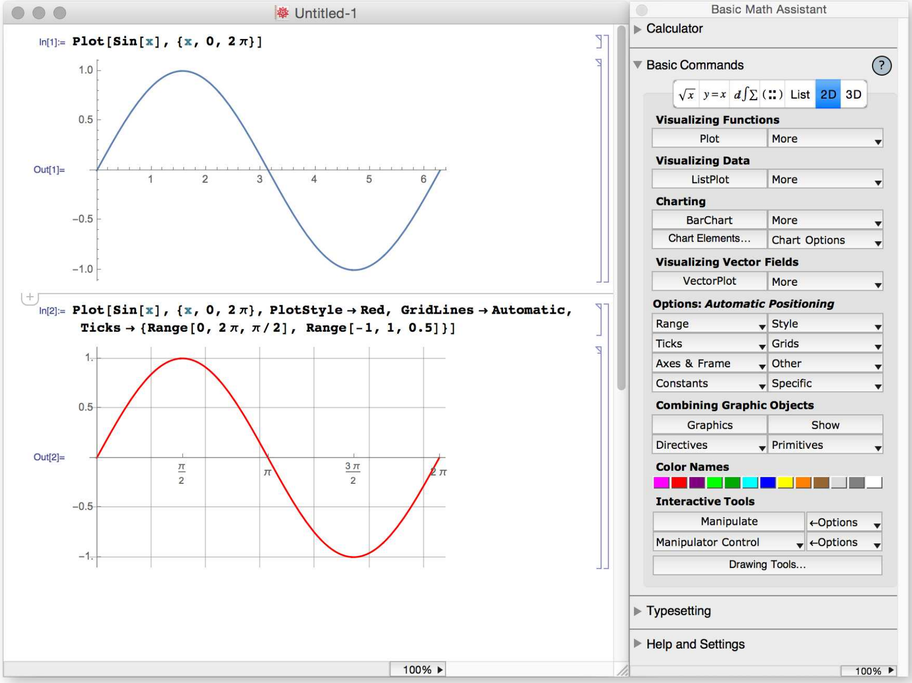
If you are having trouble selecting a component because it is too close to another one, select the component you are not interested in and use the tools in the **Operations** section to send it to the back; this will reorder the layers so that the component you *do* want to select is closer to the front and therefore more easily selectable.

Option Templates with the Assistant Palettes

So far, examples have shown how to use the Suggestions Bar and **Drawing Tools** to apply options to graphics. Another method is to use palettes to create option templates. Much like how palettes provide an intermediate step between using free-form input and typing Wolfram Language commands directly, palettes can play a similar role between the interactive editing capabilities of **Drawing Tools** and the finer-grained control available by applying options directly.

It is remarkably easy to customize a plot with either the **Basic Math Assistant** or the **Classroom Assistant** palette: simply create the plot, place your cursor anywhere in the cell containing the plotting command, navigate to the appropriate **2D** or **3D** section of the palette and select a choice from the drop-down buttons in the **Options** section.

The following shows an example of two plots: the first without any options, and the second with options applied by making selections from the **Options** section of the **Basic Math Assistant** palette.



```
In[1]:= Plot[Sin[x], {x, 0, 2 π}]
```

```
In[2]:= Plot[Sin[x], {x, 0, 2 π}, PlotStyle → Red, GridLines → Automatic,
         Ticks → {Range[0, 2 π, π / 2], Range[-1, 1, 0.5]}]
```

Why use palettes to paste option templates? For those who like visual menus, the palette gives a nice presentation of some of the most commonly used options. Unlike the Suggestions Bar, the buttons in the palette are always the same, regardless of the type of output that is available, so if you prefer the comfort of seeing the same menu, then the palettes are for you.

Using Options Directly with Wolfram Language Commands

The most effective way to customize graphical output is by specifying options directly with the Wolfram Language. This provides more fine-grained control over appearance elements and can be used for all options—not just the ones presented in the menus by the Suggestions Bar, **Drawing Tools** or palettes.

Finding Options

It is easy to find what options are available for a particular command: a complete list is given by evaluating the **Options** command for a particular function, which shows both the available options and their default values. For example, here are the first 10 options available for **Plot**.

```
Take[Options[Plot], 10]
```

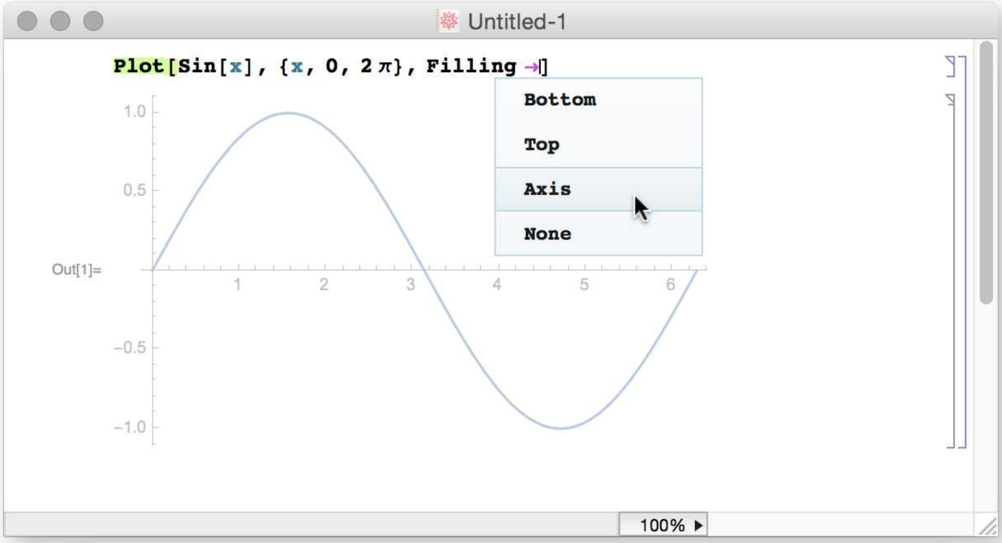
```
                                              1
{AlignmentPoint → Center, AspectRatio → ───────────, Axes → True,
                                         GoldenRatio
  AxesLabel → None, AxesOrigin → Automatic, AxesStyle → {}, Background → None,
  BaselinePosition → Automatic, BaseStyle → {}, ClippingStyle → None}
```

Take is a useful command that displays the first *n* elements of a list. Here we use it to display only the first 10 options, rather than the rather long complete list of 60 options available for the **Plot** command. You can read more about **Take** in **Chapter 17: Linear Algebra**.

In general, though, a more comprehensive list of options is provided by the relevant function page in the documentation. Navigate to a function page for a command, such as **Plot**, scroll down and expand the Options section. A listing of relevant options will be available, and expanding a specific option will show examples of applying that option.

Remember that a quick way to navigate to any command's function page in the Documentation Center is by highlighting the command name, clicking the **Help** menu and choosing **Find Selected Function**.

Mathematica's Code Assistance is also useful when adding options to commands; it will suggest the most common settings for options as they are typed into an input cell, and these suggestions can be clicked to paste them into the input cell.



```
Plot[Sin[x], {x, 0, 2 π}, Filling →]
```

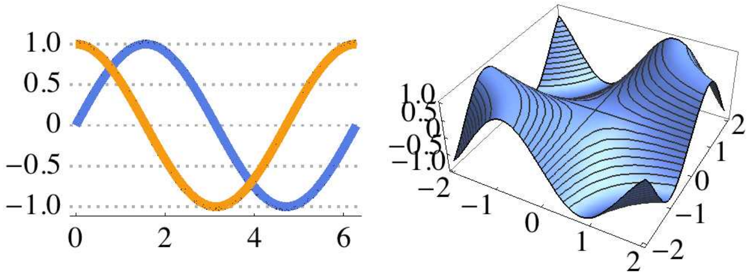
Some Useful Options for Customizing Plots

It would be impossible to highlight every useful option for plotting without replicating entire sections of the documentation. That said, an overview of some of the most commonly used and useful options will be provided with the hope that they inspire readers to continue exploring on their own.

PlotTheme

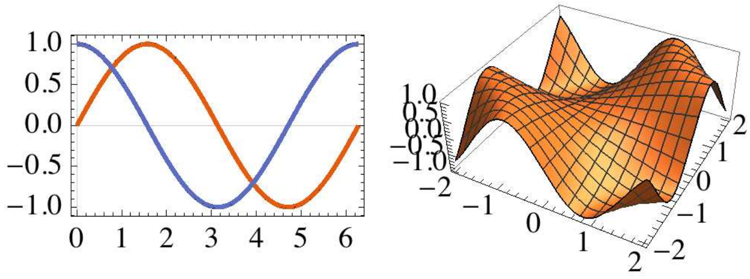
The **PlotTheme** option can be used to change the overall appearance of a plot and may affect attributes like color, thickness, axes and frames. This option is incredibly useful because the same theme can be applied to different types of plots to achieve a consistent appearance among the results without having to fiddle with individual option settings for each command. For example, here are two plots that use the **"Business"** setting.

```
GraphicsRow[{
    Plot[{Sin[x], Cos[x]}, {x, 0, 2 π}, PlotTheme → "Business"],
    Plot3D[Sin[x y], {x, −2, 2}, {y, −2, 2}, PlotTheme → "Business"]
 }]
```



And here are two plots that use the **"Scientific"** setting.

```
GraphicsRow[{
    Plot[{Sin[x], Cos[x]}, {x, 0, 2 π}, PlotTheme → "Scientific"],
    Plot3D[Sin[x y], {x, −2, 2}, {y, −2, 2}, PlotTheme → "Scientific"]
  }]
```



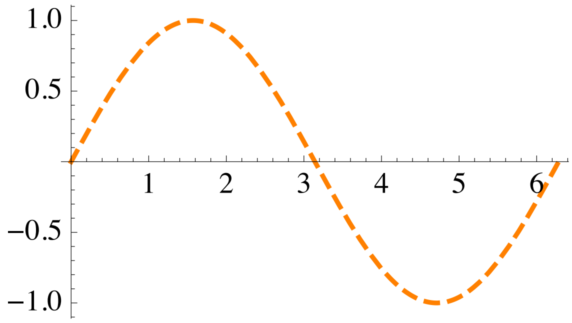
More options can be applied to a plot that has the **PlotTheme** option set, so specific features can be changed to further customize results, if desired.

If you want to switch to the default styling that was used in previous versions of Mathematica, the **PlotTheme** → "**Classic**" option setting can be used.

PlotStyle

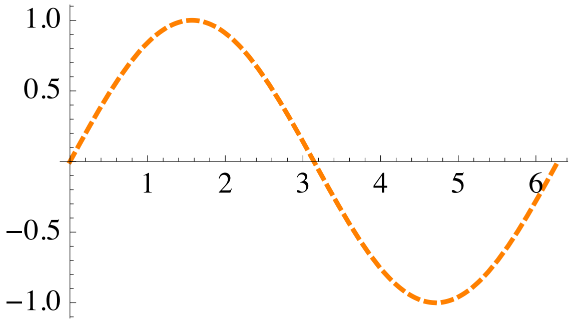
The **PlotStyle** option can be used to control several different features like dashing, color and thickness. These features can often be specified with symbolic forms or numerical values. For example, the following uses special names—**Orange** and **Dashed**—for the option settings.

```
Plot[Sin[x], {x, 0, 2 π}, PlotStyle → {Orange, Dashed}]
```



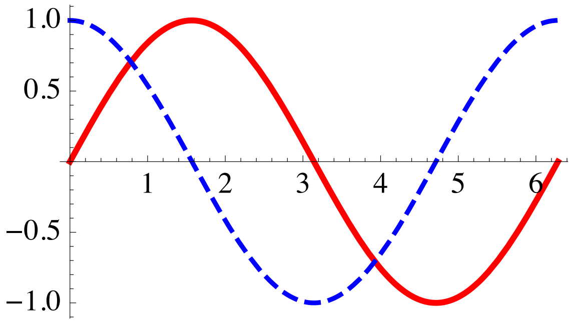
Alternatively, passing numeric values to the more general counterparts for these symbols—**RGBColor** for **Orange** and **Dashing** for **Dashed**—can be used instead.

```
Plot[Sin[x], {x, 0, 2 π}, PlotStyle → {RGBColor[1, 0.5, 0], Dashing[{0.017, 0.017}]}]
```



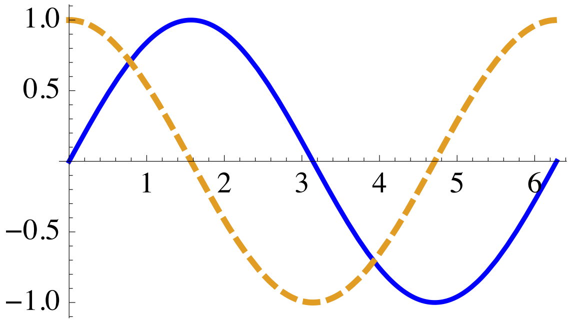
An extremely common desire is to have multiple plots styled differently but displayed on the same set of axes. This can be accomplished by using **PlotStyle** and taking care to group each set of attributes into its own list. Mathematica will take each sublist and apply the changes to the corresponding curve in the order that the arguments were provided. In the following example, the curve of sin(x) will be red and thick, and the curve of cos(x) will be blue and dashed.

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 π}, PlotStyle → {{Red, Thick}, {Blue, Dashed}}]
```



A common mistake is for new users to want to group "like" elements together, so the preceding command might be mistakenly written as follows.

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 π}, PlotStyle → {{Red, Blue}, {Thick, Dashed}}]
```

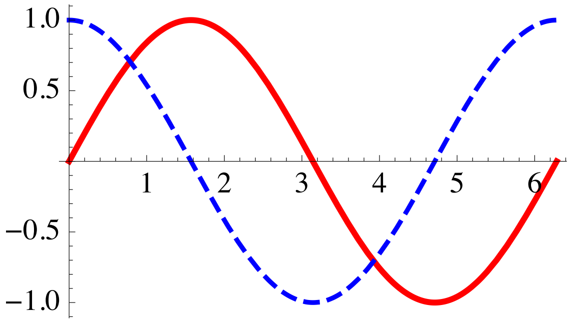


A user might think Mathematica would take the list of colors and apply red to the first curve and blue to the second curve, and that a similar logic would be used to make the first curve thick and the second curve dashed. However, Mathematica interprets the first sublist, **{Red, Blue}**, as a directive to make the curve of sin(x) red and blue, and to resolve this contradiction, the final argument is used to determine the color of the plot. Similarly, the second sublist, **{Thick, Dashed}**, is interpreted as a directive to make the curve of cos(x) thick and dashed; these directives are not contradictory and thus both are applied.

If you leave Mathematica's default styling alone or if you use the **PlotThemes** option, you will end up with two differently colored curves.

For readers who try to avoid the bookkeeping that can accompany multiple sets of brackets, the **Directive** command can provide some relief. Recall the previous example where the curve of sin(x) was red and thick and the curve of cos(x) was blue and dashed. This same result can be achieved as follows by using **Directive**.

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 π}, PlotStyle → {Directive[Red, Thick], Directive[Blue, Dashed]}]
```



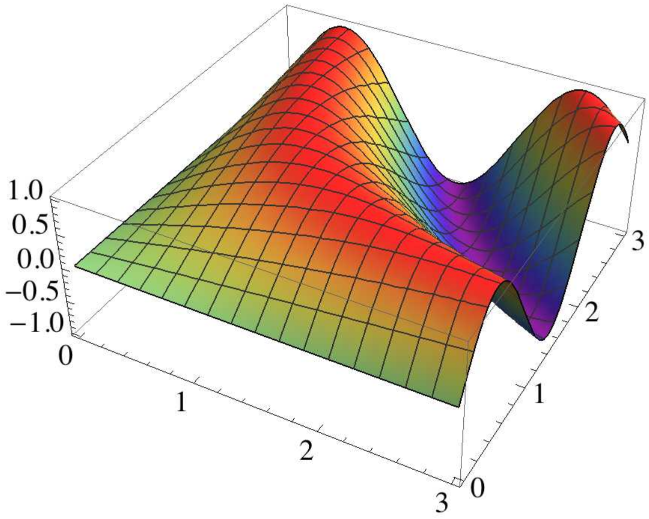
In this case, since each set of options is grouped together inside a **Directive** statement, some may find the syntax a bit cleaner.

> What if you always want to use the same set of plot options? There are multiple different approaches to this problem. You can create a list of common options and then pass that variable to a plotting command, you can create your own custom plotting command with the settings you like hard-coded into the command or you can use the **SetOptions** command to change the default settings for commands like **Plot**, which means the same options will be applied to the output each time **Plot** is used.

ColorFunction

The **ColorFunction** option is used to specify a function for coloring graphics. While it can be used for 2D graphics, it really shines when used with 3D graphics, like those returned by **Plot3D**.

```
Plot3D[Sin[x y], {x, 0, 3}, {y, 0, 3}, ColorFunction → ColorData["Rainbow"]]
```

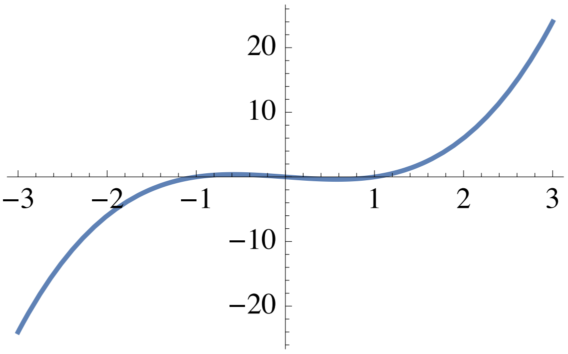


ColorFunction can be defined by a user, but there are many predefined functions ready for use. These predefined functions can be seen by clicking the **Palettes** menu and choosing **Color Schemes**.

PlotRange *and* AxesOrigin

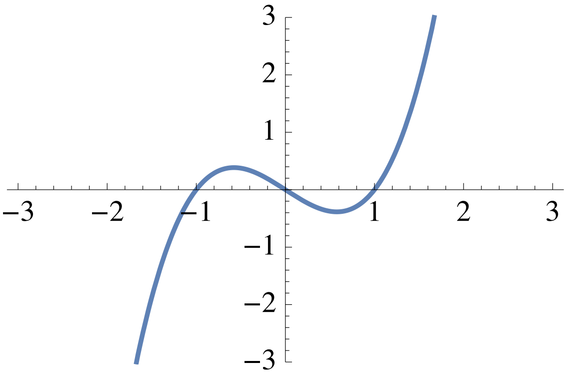
Many plotting commands require minimum information, like the expression to plot and a domain to plot over. Mathematica takes care of the rest, such as choosing the viewing range. The default result is often satisfactory, but there may be situations where enforcing control through options can yield more desirable results. Notice the plot range chosen for the following example.

```
Plot[x^3 - x, {x, -3, 3}]
```



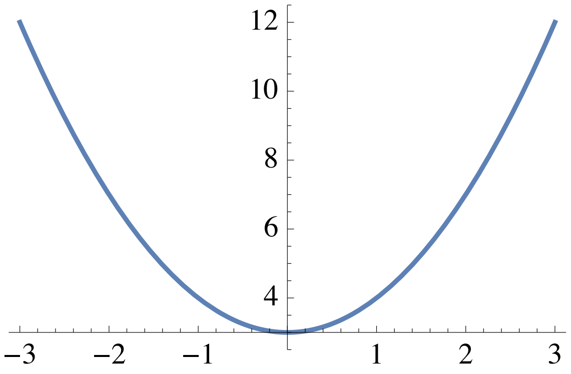
If the intent is to highlight the three roots of $x^3 - x$, then a constrained viewing window would be more useful, and this can be specified with the use of **PlotRange**.

```
Plot[x^3 - x, {x, -3, 3}, PlotRange → {-3, 3}]
```



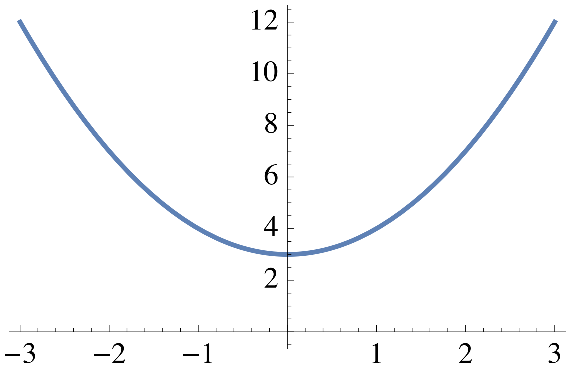
Another option with a related purpose is **AxesOrigin**. This comes in particularly handy for emphasizing whether a curve crosses through or near the point (0, 0). A casual reader might glance at the following plot and assume the parabola crosses through (0, 0).

```
Plot[x^2 + 3, {x, −3, 3}]
```



However, forcing the axes to be drawn at (0, 0) reveals that this is not true.

```
Plot[x^2 + 3, {x, −3, 3}, AxesOrigin → {0, 0}]
```



This same behavior can also be exposed by using **PlotRange**.

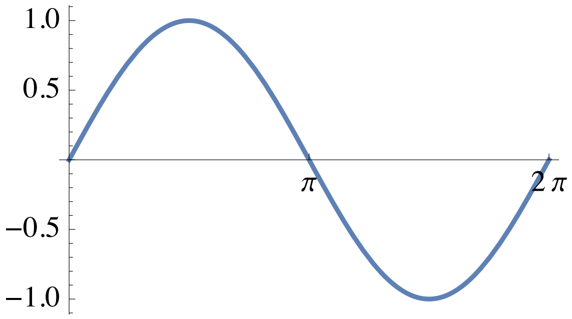
Why is the origin of the plot not always (0,0)? Mathematica chooses the best view of a plot, and it is common for the critical areas in a plot to be unrelated to the origin of the graph. Always displaying the origin at (0, 0) would likely make other critical areas too small to be noticeable.

Ticks

It can be helpful to change the default tick marks to employ a different spacing metric or unit of measure, such as ticks that correspond to units of π when plotting a trigonometric function. The **Ticks** option will accept many different syntactic forms, but the most control is afforded when a list of specific values for the x and y directions is given.

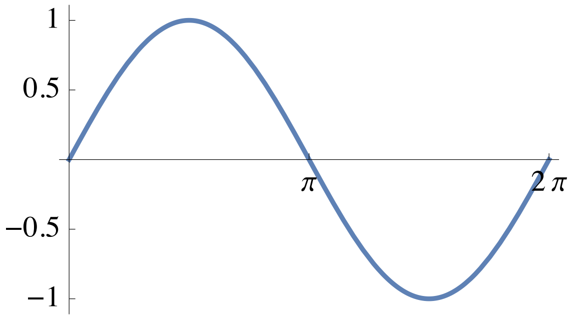
It is possible to customize the tick marks on just one axis. For example, the following plot shows tick marks at positions π and $2\,\pi$ on the x axis, while Mathematica automatically selects the tick marks for the y axis.

```
Plot[Sin[x], {x, 0, 2 π}, Ticks → {{π, 2 π}, Automatic}]
```



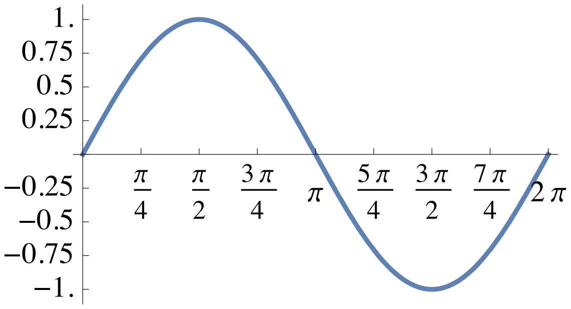
Giving the specifications for the ticks on both axes is accomplished by passing two lists to the **Ticks** command: the first list contains the positions for the ticks on the x axis, and the second list contains the positions for the ticks on the y axis.

```
Plot[Sin[x], {x, 0, 2 π}, Ticks → {{π, 2 π}, {−1, −0.5, 0.5, 1}}]
```



Instead of having to type a list of values, the **Range** command can be used to generate a list from a start value to an end value in steps of a certain size. **Range** is a great command to use when generating ticks to appear at evenly spaced intervals.

```
Plot[Sin[x], {x, 0, 2 π}, Ticks → {Range[0, 2 π, π/4], Range[−1, 1, 0.25]}]
```

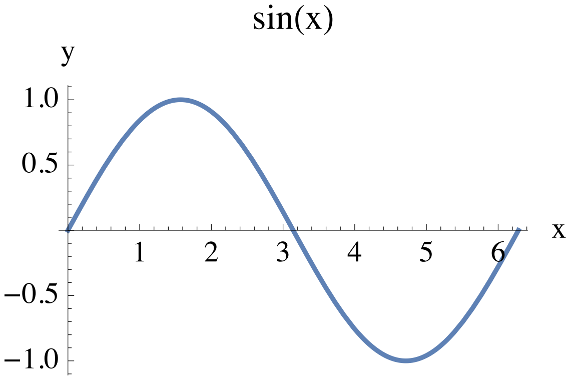


Other commands, like **Table**, can be used to create more sophisticated lists for tick positions.

PlotLabel *and* **AxesLabel**

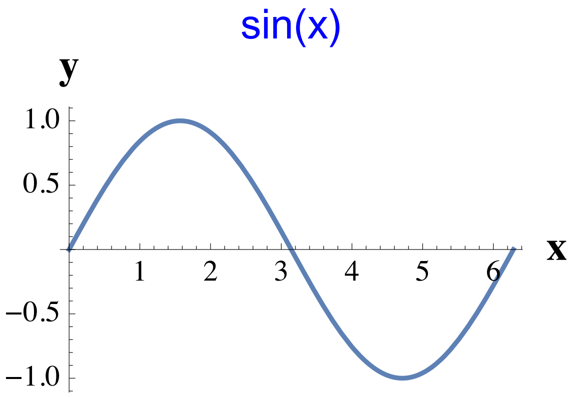
PlotLabel can be used to give an overall label for a plot, and **AxesLabel** can be used to place specific labels at the end of each axis. **PlotLabel** takes a single argument, while **AxesLabel** takes a variable number of arguments depending on how many axes are available for labeling. Both options expect to receive strings for their option settings.

```
Plot[Sin[x], {x, 0, 2 π}, PlotLabel → "sin(x)", AxesLabel → {"x", "y"}]
```



If customization of the label itself is desired, then the **Style** command can be used to control font family, size, background, text color and many other options. This versatility makes **Style** a natural choice to couple with **PlotLabel** and **AxesLabel** when customizing graphics. **Style** expects a string as its argument, and then it accepts option values to control the appearance of that text. In this example, the plot label is changed to use 14 point blue Arial, and the axes label is set to be 14 point and bold.

```
Plot[Sin[x], {x, 0, 2 π},
  PlotLabel → Style["sin(x)", FontSize → 14, FontFamily → "Arial", FontColor → Blue],
  AxesLabel → {Style["x", FontSize → 14, Bold], Style["y", FontSize → 14, Bold]}]
```

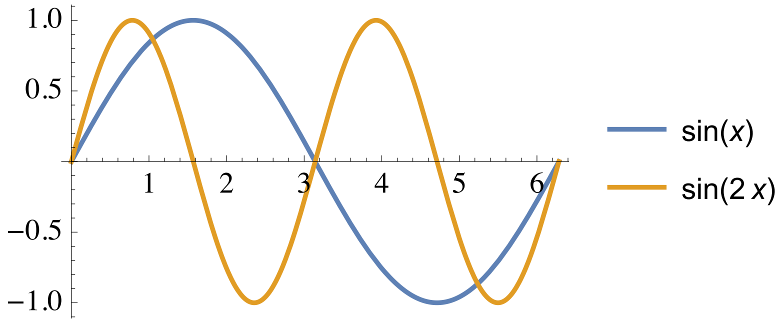


Style lets you cheat a little on the option values. Instead of typing out **FontColor → Blue** and **FontSize → 14**, you can just use **Blue** and **14** instead. **Style** will automatically interpret these values as settings to change the color and size of the text.

PlotLegends

When multiple curves are plotted, it can be desirable to easily distinguish between them. Mathematica's default styling will automatically color different functions differently, and the **PlotLegends** option can also be used to create a legend that labels each curve. Various settings can be given to **PlotLegends**, but a very common one is **"Expressions"**, which will create legend labels based on the **TraditionalForm** of the expression being plotted. The end result is a nice-looking legend with minimal direction from the user.

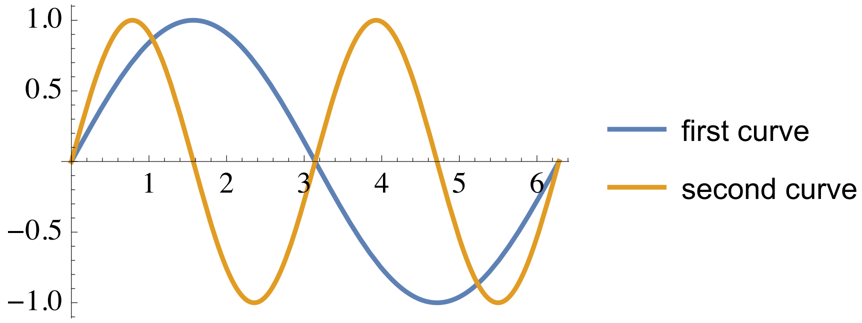
```
Plot[{Sin[x], Sin[2 x]}, {x, 0, 2 π},
  PlotLegends → "Expressions"]
```



TraditionalForm is useful for all sorts of things, including changing the appearance of input and output cells. A cell can be converted to **TraditionalForm** by clicking the **Cell** menu, choosing **Convert To** and then selecting **TraditionalForm**. To change a cell back to its default appearance, click the **Cell** menu, choose **Convert To** and then select **StandardForm.**

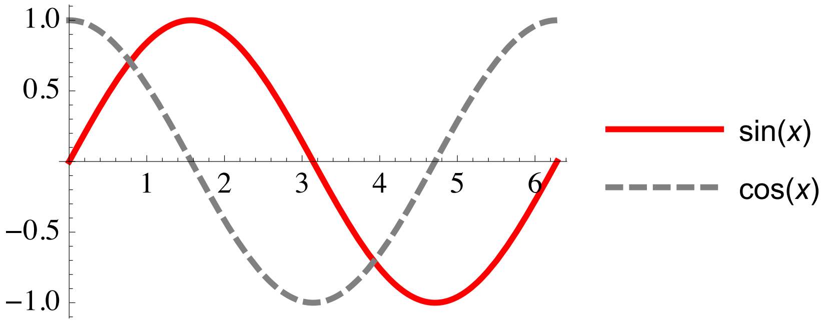
PlotLegend also accepts explicit settings for its labels. If multiple curves are plotted, then multiple labels can be defined by placing them in a list.

```
Plot[{Sin[x], Sin[2 x]}, {x, 0, 2 π},
  PlotLegends → {"first curve", "second curve"}]
```



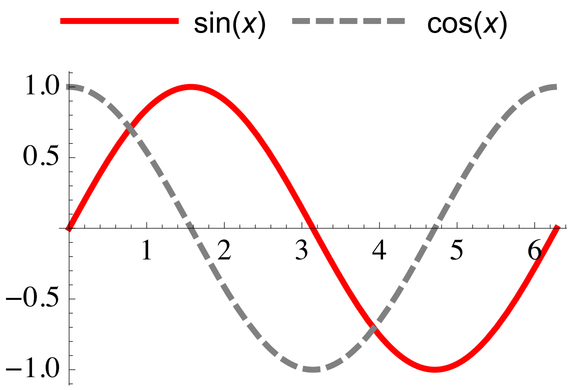
The styling for the legends is based on the styling for the plots themselves, so if the plots are customized, the plot legend styling will automatically match.

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 π},
  PlotStyle → {Directive[Red, Thick], Directive[Gray, Thick, Dashed]},
  PlotLegends → "Expressions"]
```



Legends can be highly customized and stylized themselves. New users will likely be most interested in changing the location of the legend, which can be adjusted by wrapping the option setting in **Placed**. In the following example, **Placed** is used to move the legend to the top of the plot.

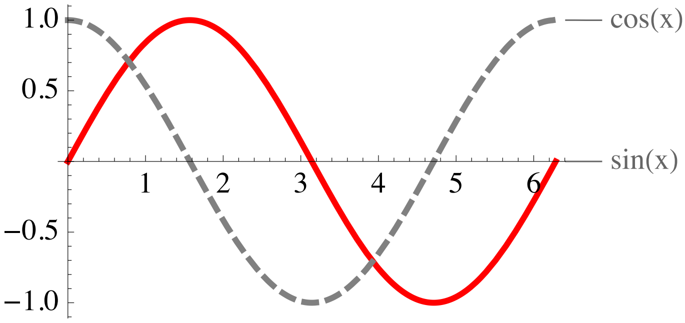
```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 π},
  PlotStyle → {Directive[Red, Thick], Directive[Gray, Thick, Dashed]},
  PlotLegends → Placed["Expressions", Top]]
```



Other possible settings for **Placed** include **Bottom** and **Left**.

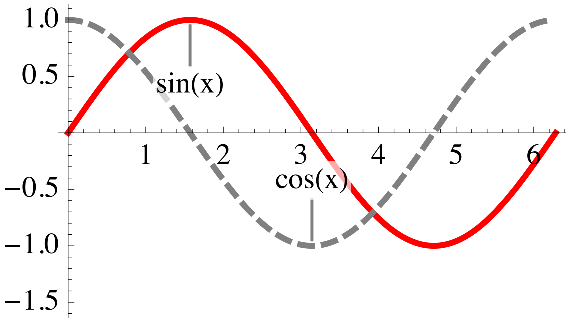
As an alternative to creating a legend, the function **Callout** is useful for creating a label at a specific position within a graphic. **Callout** is not listed as an option near the end of the **Plot** statement, but instead is wrapped around the first element, which is the function that is being plotted. **Callout** takes two arguments: the function or data that is being plotted, and then an expression—most commonly a string of text—for the label.

```
Plot[{Callout[Sin[x], "sin(x)"], Callout[Cos[x], "cos(x)"]}, {x, 0, 2 π},
  PlotStyle → {Directive[Red, Thick], Directive[Gray, Thick, Dashed]}]
```



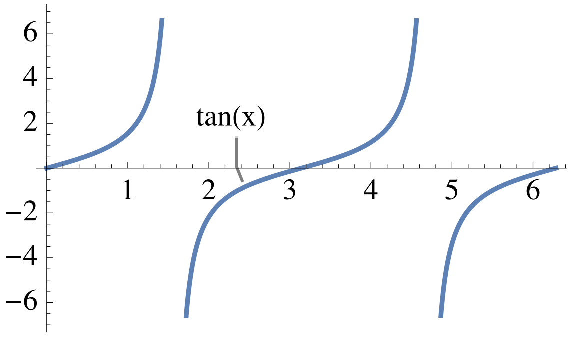
Callout has an optional third argument to specify placement, and as a matter of convenience, uses the same named values (**Top**, **Bottom**, **Left** and **Right**) available to the **PlotLegends** option.

```
Plot[{Callout[Sin[x], "sin(x)", Top], Callout[Cos[x], "cos(x)", Bottom]}, {x, 0, 2 π},
  PlotStyle → {Directive[Red, Thick], Directive[Gray, Thick, Dashed]}]
```



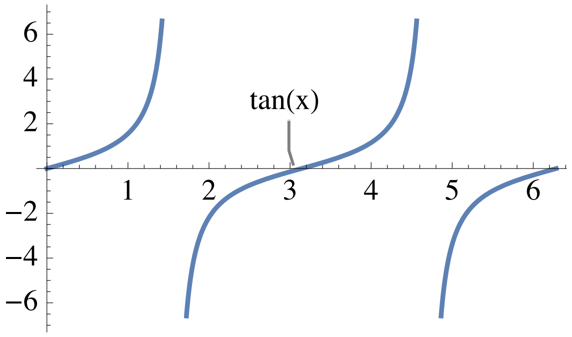
Automation is a common theme in this book. When using a named value like **Bottom** or **Right** as the third argument of **Callout**, Mathematica chooses an aesthetically pleasing area of the plot for the label. However, the third argument for **Callout** can also be a specific value, allowing for exact placement of the label. In the following example, the function and label have been changed to show the plot of the tangent of x with a callout used to label the function where $x = 2.5$.

```
Plot[Callout[Tan[x], "tan(x)", 2.5], {x, 0, 2 π}]
```



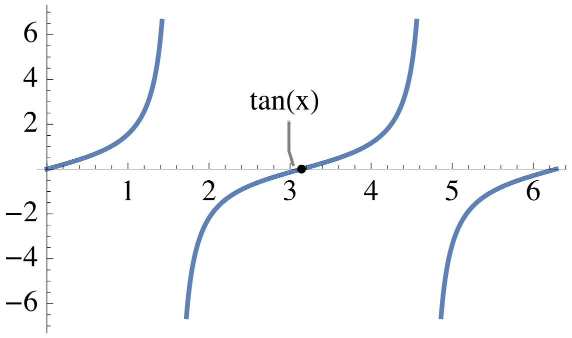
The third argument can be given as a symbol that corresponds to a value, such as π.

```
Plot[Callout[Tan[x], "tan(x)", π], {x, 0, 2 π}]
```



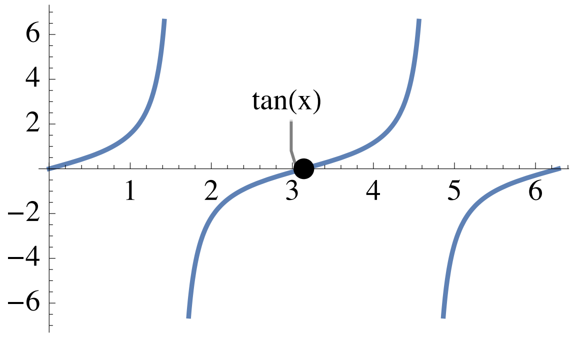
This chapter has contained many examples of functions and surfaces being plotted with commands such as **Plot** and **Plot3D**. An additional command, **Epilog**, allows shapes such as points and lines to be added to this visualization output. **Epilog** works with objects called graphics primitives, which will be discussed in more detail in a later chapter. For now, **Epilog** will be used to render a point at coordinates (π, 0) on the same tangent line plotted in the previous example.

```
Plot[Callout[Tan[x], "tan(x)", π], {x, 0, 2 π}, Epilog → Point[{π, 0}]]
```



The size of the point can be changed by using **PointSize**. Since multiple commands are now being used with the **Epilog** option, they are surrounded by curly braces to turn them into a list.

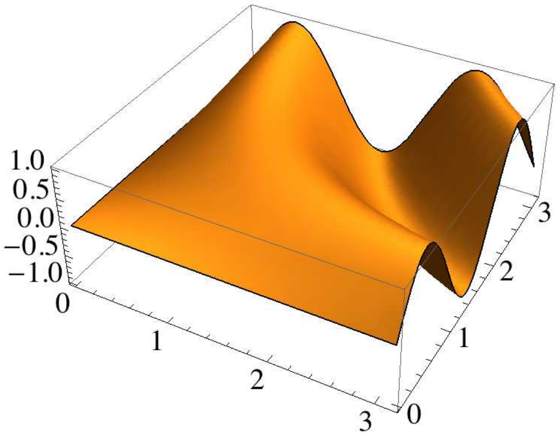
```
Plot[Callout[Tan[x], "tan(x)", π], {x, 0, 2 π}, Epilog → {PointSize[Large], Point[{π, 0}]}]
```



Mesh *and* Opacity

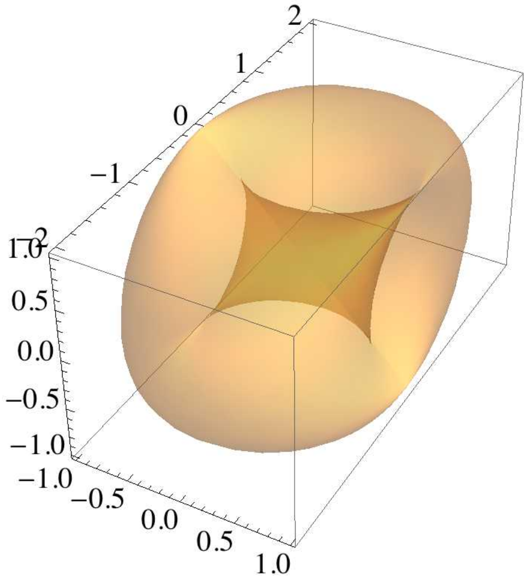
Many 3D graphics display a mesh by default. While this can be useful for assessment, it can sometimes get in the way of close inspection of the surface or may need to be removed for aesthetic reasons. The mesh is easily removed by setting **Mesh → None**.

```
Plot3D[Sin[x y], {x, 0, π}, {y, 0, π}, Mesh → None]
```



Another useful option when working with 3D surfaces is **Opacity**. This option allows the user to set the transparency for a graphic, with 0 being fully transparent and 1 being fully opaque. **Opacity** can be used to see inside objects, and this effect is amplified if the mesh is turned off.

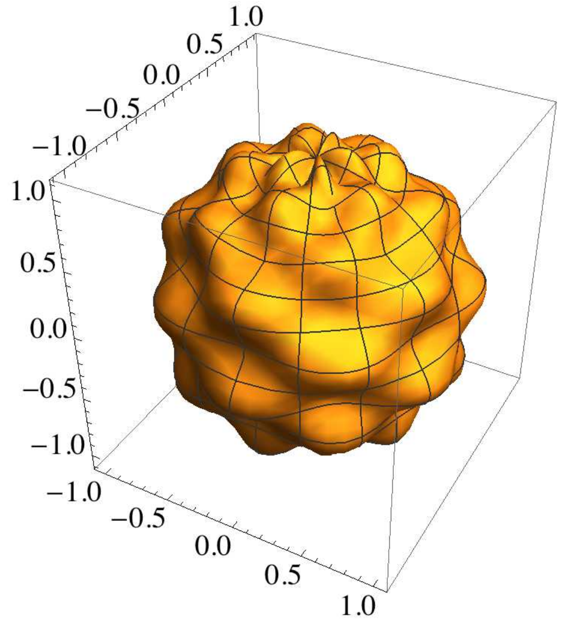
```
ParametricPlot3D[{Cos[u], Sin[u] + Cos[v], Sin[v]}, {u, 0, 2 π}, {v, −π, π},
  PlotStyle → Opacity[0.3], Mesh → None]
```



Texture

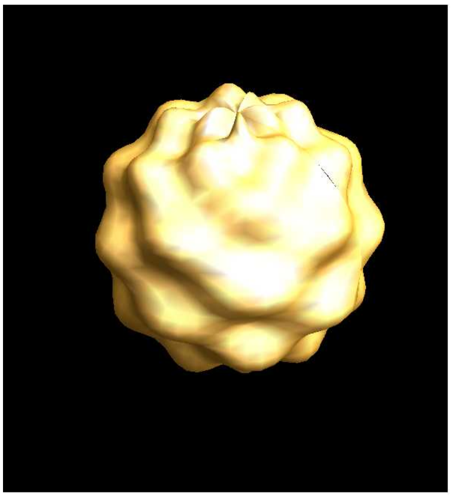
Mathematica supports texture mapping, which allows a user to specify a texture to be placed over a surface. Texture mapping can have a dramatic effect on plots, especially when combined with other options.

```
SphericalPlot3D[1 + Sin[5 ϕ] Sin[10 θ]/10, {θ, 0, π}, {ϕ, 0, 2 π}]
```



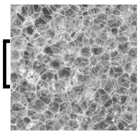
The same surface is now plotted but with options set to remove the axes, to remove the mesh, to remove the bounding box, to change the background color and to adjust the lighting to be neutral.

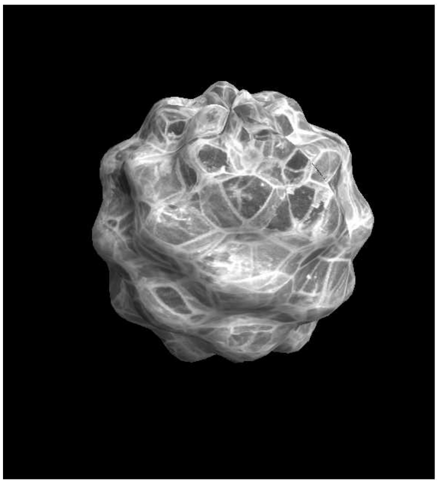
```
SphericalPlot3D[1 + Sin[5 ϕ] Sin[10 θ]/10, {θ, 0, π}, {ϕ, 0, 2 π}, Axes → False,
  Mesh → None, Boxed → False,
  Background → Black, Lighting → "Neutral"]
```



Finally, the **Texture** command is used with the **PlotStyle** option to overlay a graphic on the surface of the plot.

```
SphericalPlot3D[1 + Sin[5 ϕ] Sin[10 θ]/10, {θ, 0, π}, {ϕ, 0, 2 π}, Axes → False,
  Mesh → None, Boxed → False, Background → Black, Lighting → "Neutral",
  PlotStyle → {Texture[ ]}]
```





You can drag and drop images into Mathematica notebooks and use them directly as input.

A complete list of built-in black and white textures can be found by evaluating **ExampleData["Texture"]**, and a specific texture can then be accessed by passing its values to the **ExampleData** command.

```
ExampleData["Texture"]
```

```
{{Texture, Bark}, {Texture, Bark2}, {Texture, Bark3}, {Texture, Bricks}, {Texture, Bricks2},
  {Texture, Bricks3}, {Texture, Bricks4}, {Texture, Bricks5}, {Texture, Bricks6},
  {Texture, Bricks7}, {Texture, Bricks8}, {Texture, BrickWall}, {Texture, Bubbles},
  {Texture, Bubbles2}, {Texture, Bubbles3}, {Texture, Cloth}, {Texture, Cloth2},
  {Texture, Cloth3}, {Texture, Grass}, {Texture, Grass2}, {Texture, Grass3},
  {Texture, Grass4}, {Texture, Grass5}, {Texture, Gravel}, {Texture, Herringbone},
  {Texture, Herringbone2}, {Texture, Herringbone3}, {Texture, HexagonalHoles},
  {Texture, Leather}, {Texture, Leather2}, {Texture, Leather3}, {Texture, MetalGrates},
  {Texture, Mosaic}, {Texture, Mosaic2}, {Texture, Mosaic3}, {Texture, Mosaic4},
  {Texture, Mosaic5}, {Texture, Mosaic6}, {Texture, Pigskin}, {Texture, Pigskin2},
  {Texture, Pigskin3}, {Texture, Raffia}, {Texture, Raffia2}, {Texture, Raffia3},
  {Texture, Sand}, {Texture, Sand2}, {Texture, Sand3}, {Texture, Sand4},
  {Texture, Sand5}, {Texture, Sand6}, {Texture, Shingles}, {Texture, Shingles2},
  {Texture, Straw}, {Texture, Straw2}, {Texture, Straw3}, {Texture, TileRoof},
  {Texture, Wall}, {Texture, Water}, {Texture, Water2}, {Texture, Water3},
  {Texture, Wood}, {Texture, Wood2}, {Texture, Wood3}, {Texture, WoodFence}}
```

```
ExampleData[{"Texture", "Bark"}]
```

Similarly, color textures can be found by evaluating **ExampleData["ColorTexture"]**.

Defining a Function to Use Plot Options

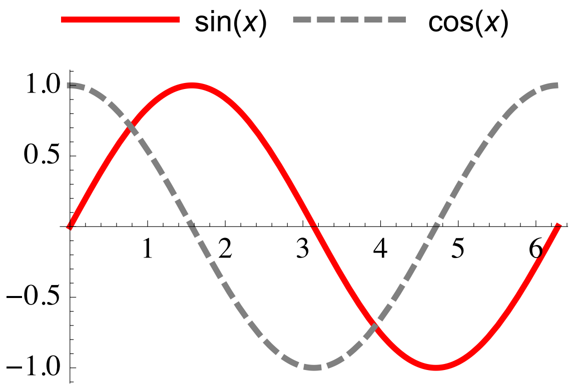
It is common to experiment with plot styling until a desired list of styles is reached. At that point, it can be useful to recycle the same set of options for a series of new plots. One way to accomplish this is to define a new function that stores the values of these plot options.

For example, if a series of plots will each have the same plot legend and styling to distinguish the two curves, the following function can be used to easily duplicate the plot options.

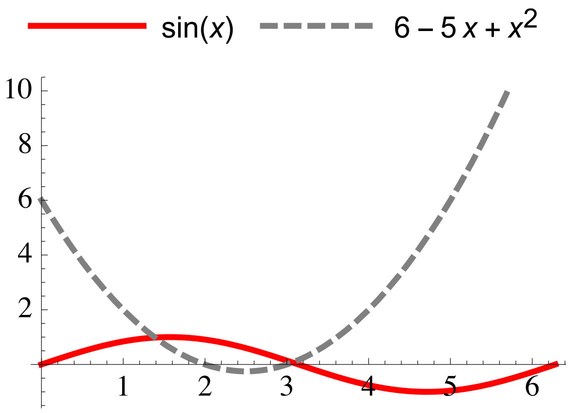
```
myPlot[eq1_, eq2_] :=
  Plot[{eq1, eq2}, {x, 0, 2 π},
    PlotStyle → {Directive[Red, Thick], Directive[Gray, Thick, Dashed]},
    PlotLegends → Placed["Expressions", Above]]
```

The function can then be used instead of **Plot** for situations in which two function plots need to be visualized.

```
myPlot[Sin[x], Cos[x]]
```



```
myPlot[Sin[x], x^2 - 5 x + 6]
```



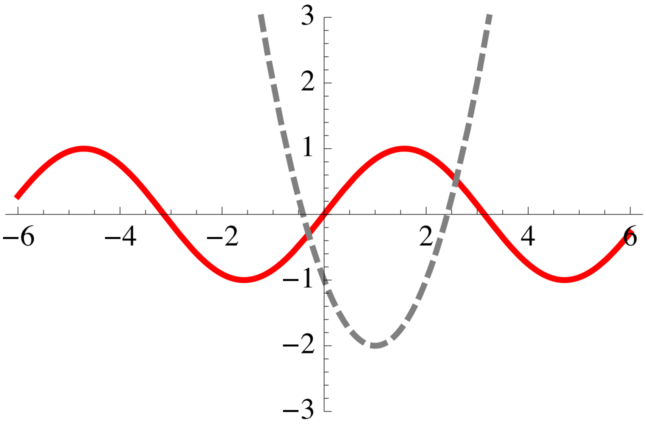
This approach does have a potential downside: since the domain is hard-coded into the definition for **myPlot**, that can be limiting when using **myPlot** to visualize certain functions.

An alternate approach is to store the values of the option settings in a list, and then to pass that list when a function is used. For example, the values for **PlotStyle** can be stored as a list and assigned to a variable named **mySettings**.

```
mySettings = PlotStyle → {Directive[Red, Thick], Directive[Gray, Thick, Dashed]};
```

Now whenever those settings need to be used, they can be passed to the command as an option. There is one trick: the **Evaluate** command needs to be used, but ignore that for now.

```
Plot[{Sin[x], x^2 − 2 x − 1}, {x, −6, 6}, PlotRange → 3, Evaluate[mySettings]]
```



Clear is used to remove all variable and function definitions from this chapter.

```
Clear[myPlot, mySettings]
```

Conclusion

Throughout this chapter, four different approaches to customizing graphics have been explored. Users should employ whichever approach or combination of approaches appeals to their sensibilities. And remember: Mathematica's default output is often quite good, so leaving that output alone is perfectly acceptable, too.

Exercises

1. Use the Wolfram Language to create a plot of $x^3 + 5\,x^2 - 11$, where x goes from -6 to 3, and add an option so that the plot is filled to the axis.
2. Use free-form input to create a plot of $\frac{\sin(x)}{x}$ such that the curve is thick and red.
3. Use the Wolfram Language to create a plot of the curves $\frac{\sin(x)}{3}$ and $\frac{\cos(x)}{5}$, where x goes from -10 to 10. Then use the Suggestions Bar to remove the axes, remove the frame and roll up the code once finished to create a single pair of input and output cells.
4. Use the Wolfram Language to plot the curves of $x^2 - 5$, $2\,x^2 + 3\,x - 5$, and $x + 2$ on the same set of axes, and where x goes from -5 to 5. Then add a plot legend with labels of "1," "2" and "3."
5. Use the appropriate Wolfram Language command to plot $\sin(x^2 - y)$, where both x and y go from $-\pi$ to π, and use options to remove the mesh, remove the bounding box, remove the axes and increase the number of plot points to 100.
6. Use free-form input to visualize the region represented by $x^3 - x^2 + y^3 - 4\,y^2 - 3\,y < 5$, and change the color of the region to orange.
7. Use the Wolfram Language to make a density plot of $\sin(3\,x)$ multiplied by $\sin(y - 5)$, and use the appropriate option and setting to automatically determine a plot legend.
8. Use the **Graph** command to create a network based on the rules $a \to b$, $a \to c$, $b \to a$, $b \to 3$, $c \to d$, $c \to e$, $d \to e$, $e \to a$, $e \to d$ and $e \to e$, and use options to show the directed edges and label the vertices.
9. Use the **ListPlot** command to create a visualization of the points {0, 0}, {1, 0}, {2, 1}, {3, 3}, {4, 3}, {5, 7}, {6, 13}, {7, 15}, {8, 16}, {9, 12} and {13, 0}, and use options to label the axes as "seconds" and "temp," change the points to red and set the plot range to go from 0 to 30.
10. Use the Wolfram Language to create a 3D bar chart of ten groups of five random real numbers, where the numbers range from 0 to 5, and then use an option to stack the bars in the chart.

CHAPTER 13

Creating Figures and Diagrams with Graphics Primitives

Introduction

The preceding chapters have focused on using Mathematica to visualize expressions, equations, surfaces and data, but the Wolfram Language also has a rich set of graphics primitives that can be used to create figures, diagrams and models. Since these primitives are Wolfram Language commands themselves, they can be programmed to create graphical representations of phenomena.

> You might be thinking, what about using **Drawing Tools** to make diagrams, like we learned about in the last chapter? That is true, but that was a very different approach. **Drawing Tools** provides an interactive interface to *manually* create a diagram. Using graphics primitives allows you to *programmatically* create a diagram. There are pros and cons to each approach, and you should use the one that fits your working style the best. One big advantage of using graphics primitives, however, is that you have really fine-grained control over the appearance of objects, and you can recreate your examples more easily when they are created with graphics primitives.

Working with 2D Primitives

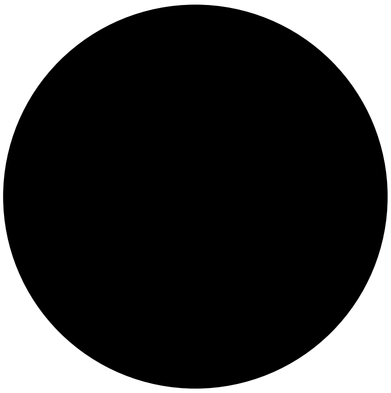
A variety of 2D graphics primitives are available: points, lines, arrows, polygons, disks, circles, rectangles, joined curves, filled curves, splined curves, triangles and many more. These primitives typically take a set of coordinates as an argument, with some additional arguments, depending on the specific primitive that is used. Some primitives can be used with empty arguments, in which case Mathematica will assume that the necessary coordinate pair is (0, 0). For example, if the **Disk** command is not passed any arguments, it will create a unit disk centered at (0, 0).

```
Disk[]
```

```
Disk[{0, 0}]
```

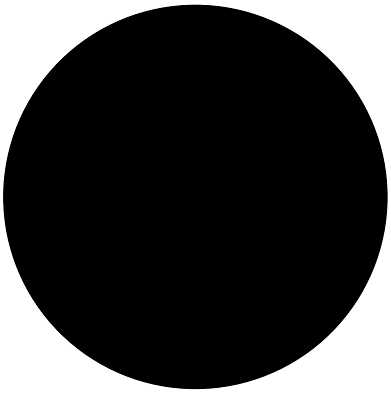
The output from evaluating a graphics primitive is not very interesting by itself. When primitives are wrapped in the **Graphics** command, though, they will be rendered and displayed as graphical output. For example, using the same **Disk** command from the preceding example with **Graphics** will create a graphical representation of that unit disk.

```
Graphics[Disk[]]
```



A different form of **Disk** can be used to specify its radius and its center position. The argument for the center position is given as a list of (x, y) values and the radius is given as a number. The following example creates a disk of radius 2 centered at $(0, -3)$.

```
Graphics[Disk[{0, –3}, 2]]
```

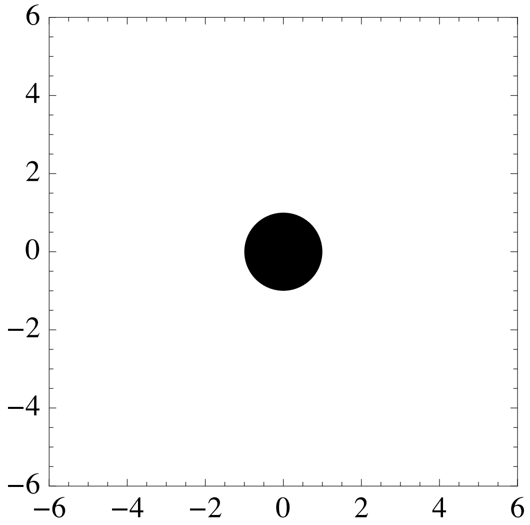


At first glance, this output looks identical to the first; this is because in both cases, Mathematica has automatically chosen a viewing window to display the graphic. This is the same behavior exhibited for commands like **Plot**.

This viewing window can be controlled with the **PlotRange** option. As was discussed in **Chapter 12: Styling and Customizing Graphics**, setting the **PlotRange** for a specific range of values allows users to control what viewing window is used. It can also be helpful to include the option **Frame → True** to draw a labeled frame with tick marks, which are useful for understanding the magnitude and position of the **Disk** as it relates to the viewing window.

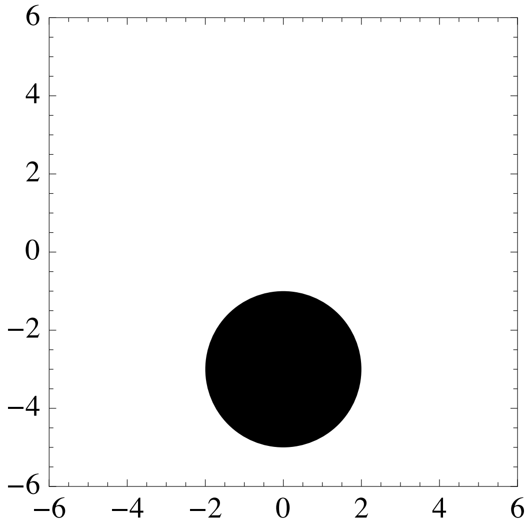
The following example uses that approach to visualize the unit disk centered at (0, 0).

```
Graphics[Disk[], PlotRange → 6, Frame → True]
```



And that same approach is used to visualize the disk of radius 2 centered at (0, −3).

```
Graphics[Disk[{0, −3}, 2], PlotRange → 6, Frame → True]
```

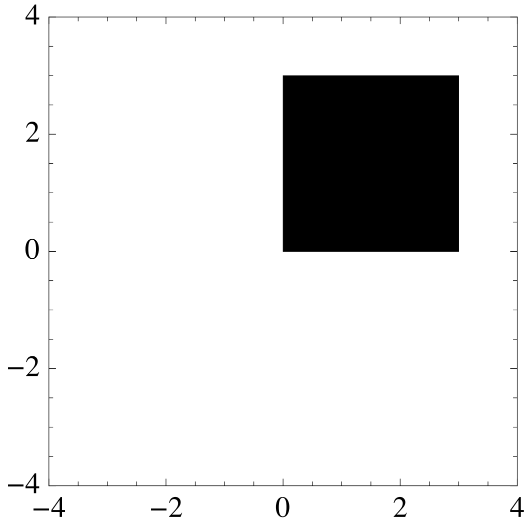


Now it is obvious that these disks are, in fact, quite different.

You might be wondering what the difference is between the **Disk** primitive and the **Circle** primitive. **Disk** is filled, while **Circle** is not.

Another 2D graphics primitive is **Rectangle**, which has a couple of different syntactical forms. The most common form takes two lists as its arguments: the first describes the (*x*, *y*) coordinates for the bottom-left corner of the rectangle and the second describes the (*x*, *y*) coordinates for the top-right corner of the rectangle. The same approach with **PlotRange** and **Frame** can be used to give some additional perspective on how the rectangle relates to its surrounding space.

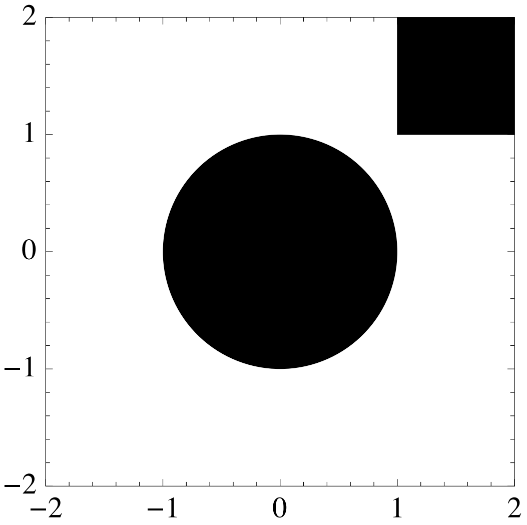
```
Graphics[Rectangle[{0, 0}, {3, 3}], PlotRange → 4, Frame → True]
```



Creating Diagrams with Multiple Primitives

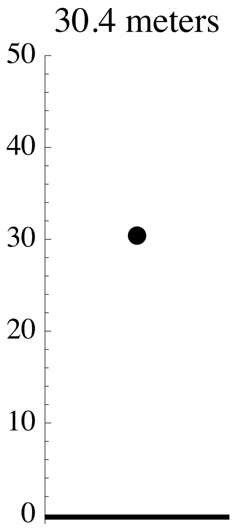
Multiple graphics primitives are often used as building blocks to construct figures, diagrams and models. To construct a diagram, pass the **Graphics** command a list of multiple primitives. The primitives will be rendered and displayed as a single graphical output.

```
Graphics[{Disk[{0, 0}, 1], Rectangle[{1, 1}, {2, 2}]}, PlotRange → 2, Frame → True]
```



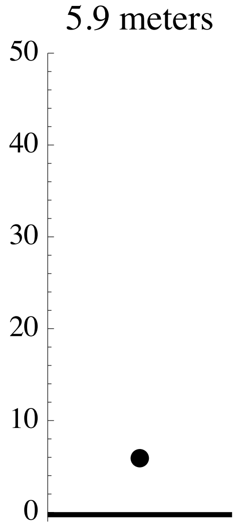
As an example, graphics primitives can be used to create a diagram to visualize a concept like simple free-fall motion. The following example shows a ball's final height if the ball starts at a height of 50 meters and falls for $t = 2$ seconds, by constructing a model using **Graphics**, **Disk** and **Rectangle**.

```
t = 2;
     1
d = ─ (−9.8) t^2 + 50;
     2
Graphics[{Disk[{10, d}], Rectangle[{0, −0.5}, {20, 0}]}, PlotRange → {{0, 20}, {−1, 50}},
  Axes → {False, True},
  PlotLabel → ToString[d] <> " meters"]
```



Creating a diagram for a new situation, such as $t = 3$, is as simple as changing the value of t and reevaluating.

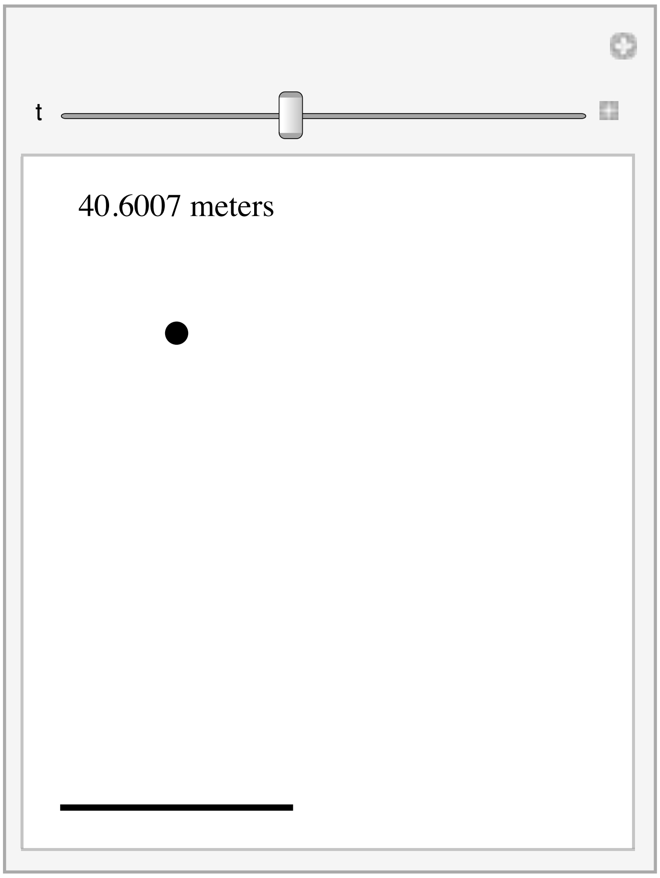
```
t = 3;
    1
d = — (−9.8) t^2 + 50;
    2
Graphics[{Disk[{10, d}], Rectangle[{0, −0.5}, {20, 0}]}, PlotRange → {{0, 20}, {−1, 50}},
  Axes → {False, True},
  PlotLabel → ToString[d] <> " meters"]
```



You could, of course, manually create these diagrams using **Drawing Tools**, but you would have to make a new diagram each time—and even with copying and pasting, that might be a little cumbersome.

This particular example could be even more interesting if packaged as a function and then manipulated, which is shown in the following block of code.

```
DynamicModule[{d, t},
 Manipulate[Graphics[{Disk[{10, d[t]}], Rectangle[{0, −0.5}, {20, 0}]},
   PlotRange → {{0, 20}, {−1, 50}},
   PlotLabel → ToString[d[t]] <> " meters"],
  {t, 0, 3.2},
  Initialization :→ (d[t_] := 1/2 (−9.8) t^2 + 50)]]
```

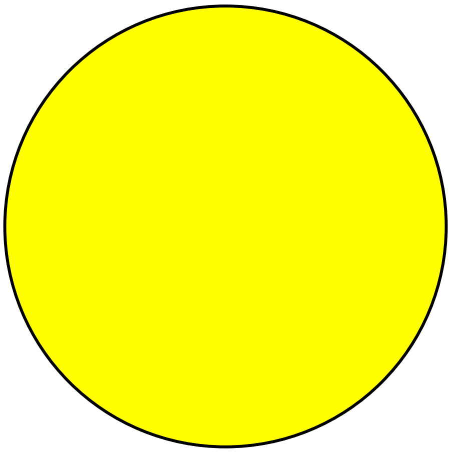


In the preceding example, **DynamicModule** is used to create local variables for **d** and **t**. This allows **d** and **t** to be used as if they are undefined, even though they were previously defined by other examples. An alternate would be to use **Clear** to remove the values of **d** and **t** before evaluating the **Manipulate** statement, but sometimes you may want to preserve your variables for later use and just temporarily recycle them for a different purpose; in those situations, you can use **DynamicModule** to create new local definitions for those variables.

Styling 2D Graphics Primitives

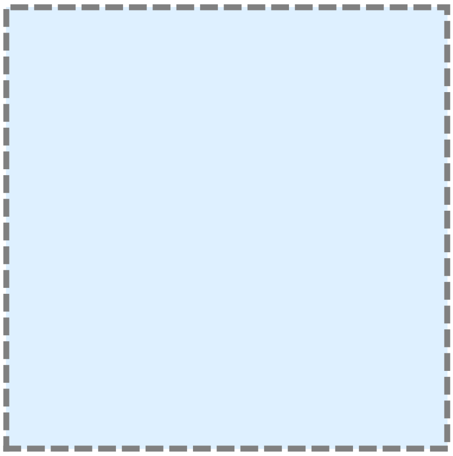
Like other Wolfram Language commands, the output of graphics primitives can be customized using options to change styling and appearance elements like color and edge form. The following example uses options to draw a yellow disk with a black edge.

```
Graphics[{Yellow, EdgeForm[{Black}], Disk[]}]
```



Some options accept multiple values, so a light blue triangle can be drawn with a gray, thick and dashed edge, as in the following example.

```
Graphics[{LightBlue, EdgeForm[{Gray, Thick, Dashed}], Rectangle[]}]
```

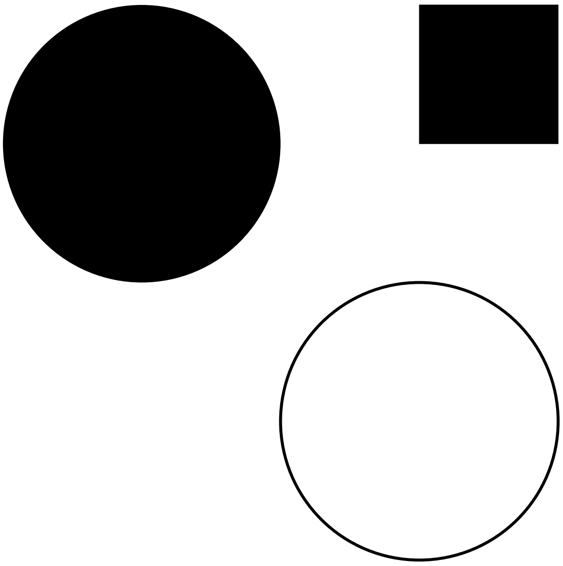


> These options, like **Thick** and **Dashed**, may seem familiar. Some of the same options used to control the style of plots can also be applied to graphics primitives. This is one of the great things about the Wolfram Language: since the language design is so consistent, once you know how to use one command or option, you can apply that same approach to other situations.

Since it is common for multiple graphics primitives to be used with a single **Graphics** command, it is important to know how to style those primitives appropriately. When the **Graphics** command is used, it will render the first graphics primitive passed to it according to the style specifications that have been given, and if none have been specified, then the default values are used; the next graphics primitive will be rendered the same way, and so on. This means that it is common to see arguments passed to **Graphics** that alternate between styling directives and graphics primitives when a different appearance is desired for each primitive.

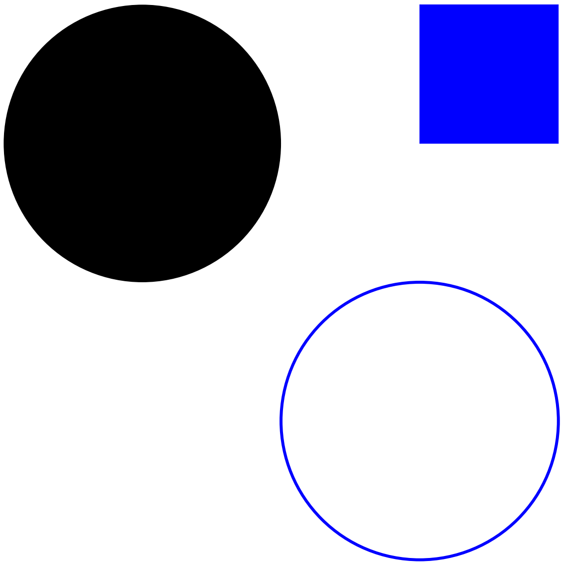
The following example draws three graphics primitives with default styling.

```
Graphics[{Disk[{−1, 1}, 1], Rectangle[{1, 1}, {2, 2}], Circle[{1, −1}, 1]}]
```



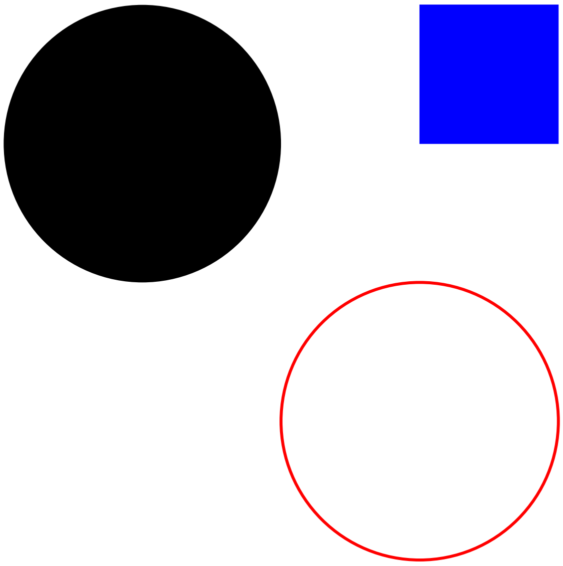
Now, an option is introduced after the **Disk** primitive but before the **Rectangle** and **Circle** primitives. This means that the disk will be drawn with default styling, but the rectangle and circle will be drawn in blue. The following example illustrates this behavior.

```
Graphics[{Disk[{−1, 1}, 1], Blue, Rectangle[{1, 1}, {2, 2}], Circle[{1, −1}, 1]}]
```



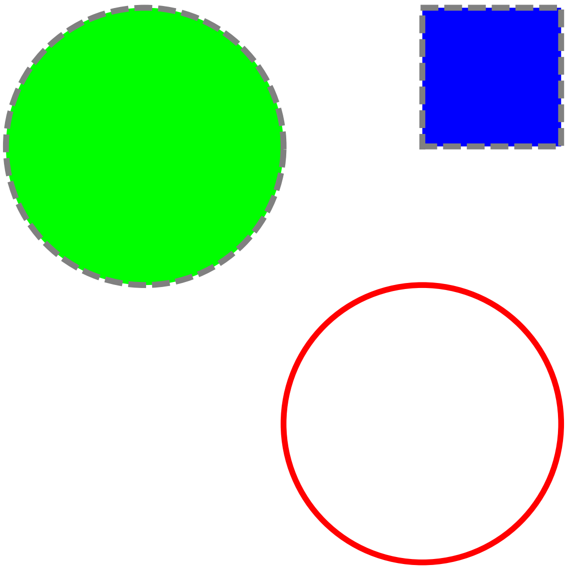
Once a styling option is introduced, it is applied to all primitives that follow it, unless a new setting for that same option is introduced. If a new styling option is introduced, then that new option setting will be applied to any primitives that follow it. In the following example, the **Blue** option is introduced and applied to the **Rectangle** primitive, but then the **Red** option is introduced, so that applies to the **Circle** graphics primitive that follows it.

```
Graphics[{Disk[{−1, 1}, 1], Blue, Rectangle[{1, 1}, {2, 2}], Red, Circle[{1, −1}, 1]}]
```



If non-competing style options are used, then multiple options will be combined when drawing a graphics primitive. In the following example, the **EdgeForm** option is used to draw a thick, gray, dashed edge around the **Disk** and **Rectangle** primitives. Different colors are interspersed throughout the **Graphics** command, which draw the disk in green, the rectangle in blue and the circle in red. In particular, note how both **EdgeForm** and **Blue** settings are applied to the rectangle, even though they were introduced at different parts of the **Graphics** command.

```
Graphics[{EdgeForm[{Thick, Dashed, Gray}], Green, Disk[{−1, 1}, 1], Blue,
    Rectangle[{1, 1}, {2, 2}], Red, Thickness[0.01], Circle[{1, −1}, 1]}]
```

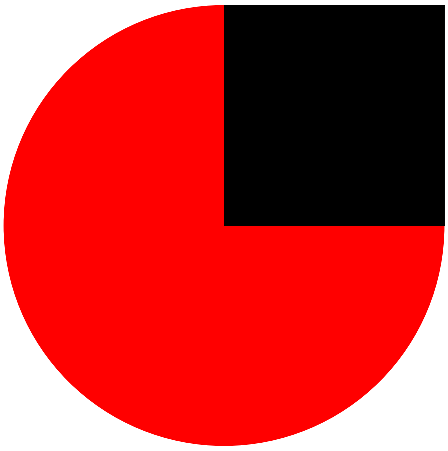


You may be wondering why the circle does not have a gray, thick, dashed edge. This is because **EdgeForm** does not work with **Circle**; since the circle does not have an edge, per se, that command does not apply. However, the edge of the circle can be colored (as in the preceding example, where it is red), and it can also allow a thickness to be specified (using something like **Thick** or **Thickness[0.01]**). The difference is that the thickness setting is applied to the entire graphic—which, in the case of a circle, just happens to be its border.

The preceding examples show that it is easy to change the styling for a list of graphics primitives, and that a styling option is applied to any applicable graphics primitives that appear in the list after the styling option. Introducing a styling option that precedes each graphics primitive can provide a different style for each shape being displayed.

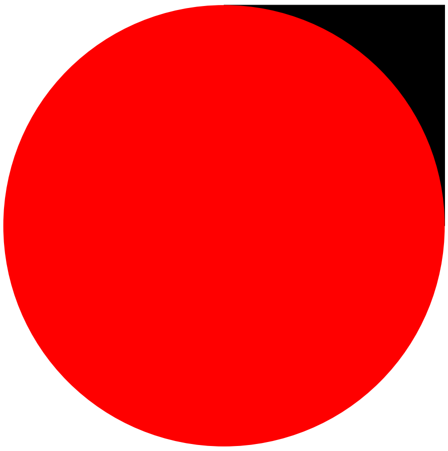
It has been shown that styling options are applied to graphics primitives sequentially, and along the same lines, graphics primitives are rendered in the order that they are given. If there are primitives that overlap, the resulting output is dependent on the ordering of the graphics primitives. The first primitive is rendered as the lowest layer, the second primitive is rendered on top of that and so on. The following example shows two graphics primitives that overlap; the disk precedes the rectangle in the **Graphics** command, so the disk is rendered first, and then the rectangle—which in this particular example is a square—is rendered on top of the disk.

```
Graphics[{Red, Disk[{0, 0}, 1], Black, Rectangle[{0, 0}, {1, 1}]}]
```



In the following example, the ordering of the primitives is reversed; the rectangle is rendered first, and the disk is rendered on top of the rectangle.

```
Graphics[{Black, Rectangle[{0, 0}, {1, 1}], Red, Disk[{0, 0}, 1]}]
```



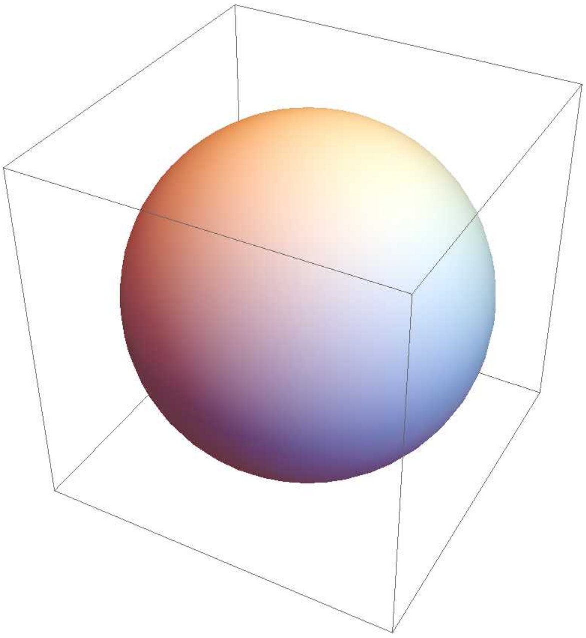
In the example earlier in the chapter that created a model for free-fall motion, rendering the falling object last is key; the falling object is the most important aspect of the example, so it should not pass behind other graphics primitives that are also being rendered. Primitives in **Graphics** statements can be reordered by copying and pasting, and the buttons in the **Operations** section of **Drawing Tools** can also be used to reorder layers.

Working with 3D Primitives

Several of the 2D primitives, like points, lines, polygons and arrows, can be used to construct 3D graphics. In addition, there are special 3D primitives for spheres, cylinders, cones, cuboids and tubes. Similar to the 2D primitives, the 3D primitives typically take a set of coordinates as an argument, with additional arguments depending on the specific primitive that is used.

Just like the **Graphics** command is wrapped around primitives to render them in 2D, the **Graphics3D** command is used to render 3D primitives.

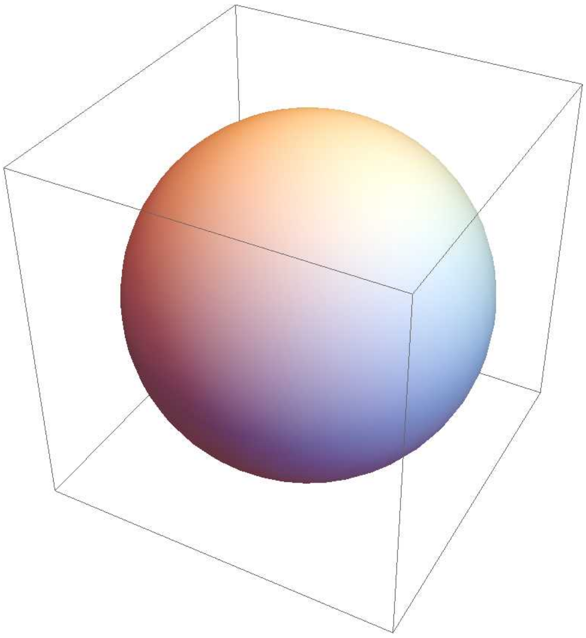
```
Graphics3D[Sphere[]]
```



Output from **Graphics3D** can be rotated, panned and zoomed, just like the output from 3D plotting commands. As a quick refresher: use click and drag to rotate when the pointer icon hovers over the 3D object and changes to twisty arrows, hold down the Shift key while dragging to pan the graphic and hold down the Alt key while dragging to zoom in and out.

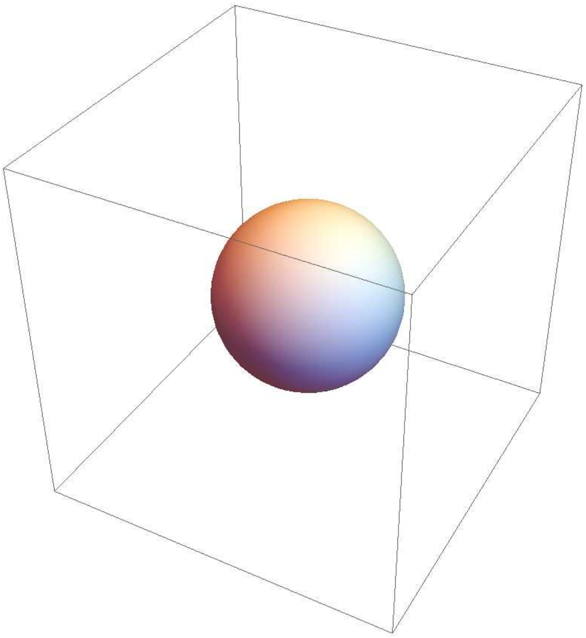
As before, it is important to be cognizant of the coordinate system and to realize that unless given explicit instructions, Mathematica will automatically choose an appropriate viewing window when rendering 3D graphics. The following example shows a sphere of diameter 0.5 centered at the point (0, 0, 0).

```
Graphics3D[Sphere[{0, 0, 0}, 0.5]]
```



The same technique of using **PlotRange** to set a specific viewing window can be used. Since this sphere is being rendered in three dimensions, the setting for **PlotRange** takes a minimum and maximum value for all three dimensions.

```
Graphics3D[Sphere[{0, 0, 0}, 0.5], PlotRange → {{-1, 1}, {-1, 1}, {-1, 1}}]
```



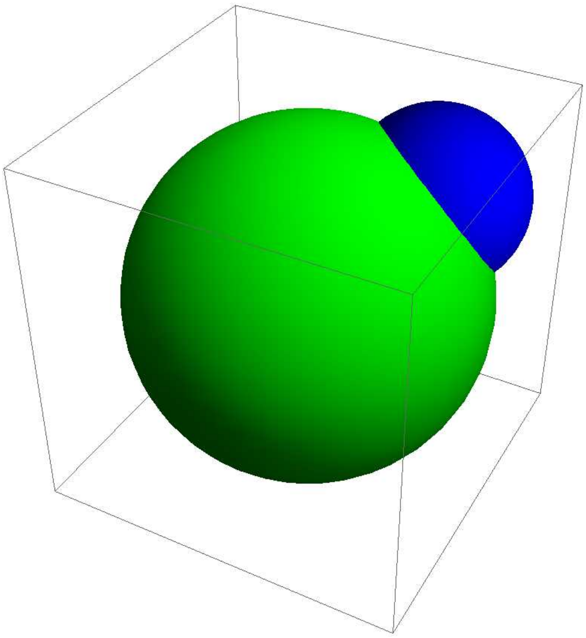
When we were exploring 2D graphics, the **Frame** option was used to give a sense of how the object related to its surrounding space. **Frame** is not an available option for **Graphics3D**, but the axes can be turned on, which can provide helpful information. The **Graphics3D** command has a default setting of **Axes→False**, but using **Axes→True** will turn them back on and print some numerical values, which can be used for reference.

Styling 3D Graphics Primitives

The same order of operations applies to styling both 2D and 3D graphics: primitives are rendered in the order they are presented, and styling options apply to any primitives that come thereafter unless overridden by a different options setting.

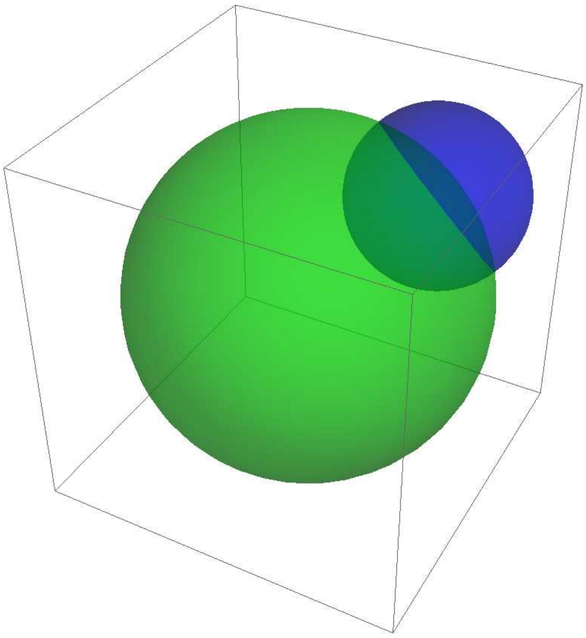
Where 2D graphics overlap due to layering, 3D graphics that overlap can appear to merge into one another. What follows is an example of this phenomenon, where the blue sphere is specified (and thus rendered) first, and then the green sphere is specified and rendered, making it appear as if the spheres have merged together.

```
Graphics3D[{Blue, Sphere[{0, 0, 0}, 0.5], Green, Sphere[{–0.5, –0.5, –0.5}, 1]}]
```



A particularly useful option for 3D graphics is **Opacity**, which can be used to see the underlying structure of graphics when they overlap. When the **Opacity** option is added to the statement that produced the preceding graphic, it creates an output that clearly shows the two spheres as separate objects.

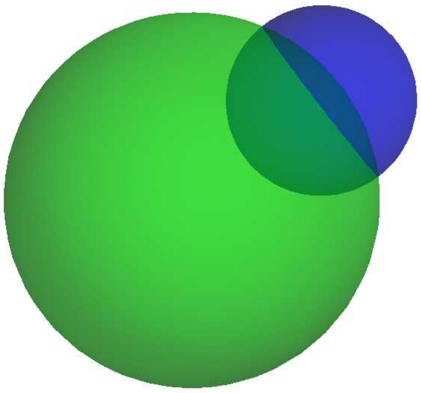
```
Graphics3D[{Blue, Opacity[0.5], Sphere[{0, 0, 0}, 0.5], Green, Sphere[{–0.5, –0.5, –0.5}, 1]}]
```



The same setting of **Opacity[0.5]** could have been introduced after the **Green** option and before the second **Sphere** is specified, but doing so is redundant if the same opacity is to be applied to both graphics. If you want to specify different opacity settings, though—including **Opacity[1]**, which makes a primitive completely solid—then add them before each primitive that you want to give a different setting.

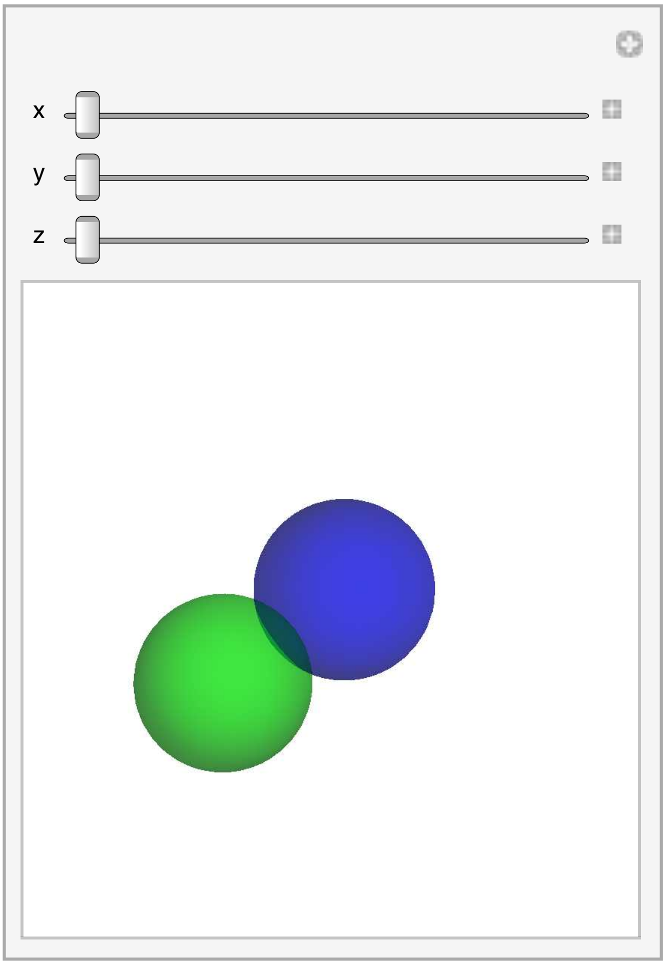
By default, 3D graphics are returned with a bounding box; this box can be a helpful way to orient the user to the relationship between the primitives and the viewing area. When using graphics primitives to construct models, however, the bounding box can get in the way. It can be removed by setting **Boxed → False**, as in the following example.

```
Graphics3D[{Blue, Opacity[0.5], Sphere[{0, 0, 0}, 0.5], Green, Opacity[0.5],
    Sphere[{−0.5, −0.5, −0.5}, 1]}, Boxed → False]
```



Like all Wolfram Language commands, graphics primitives are tightly integrated with the rest of the language. This integration makes it easy to use graphics primitives to illustrate the behavior between objects and mathematical values. The following **Manipulate** statement illustrates this; it allows the center of the green sphere to be moved by changing the values of its coordinates.

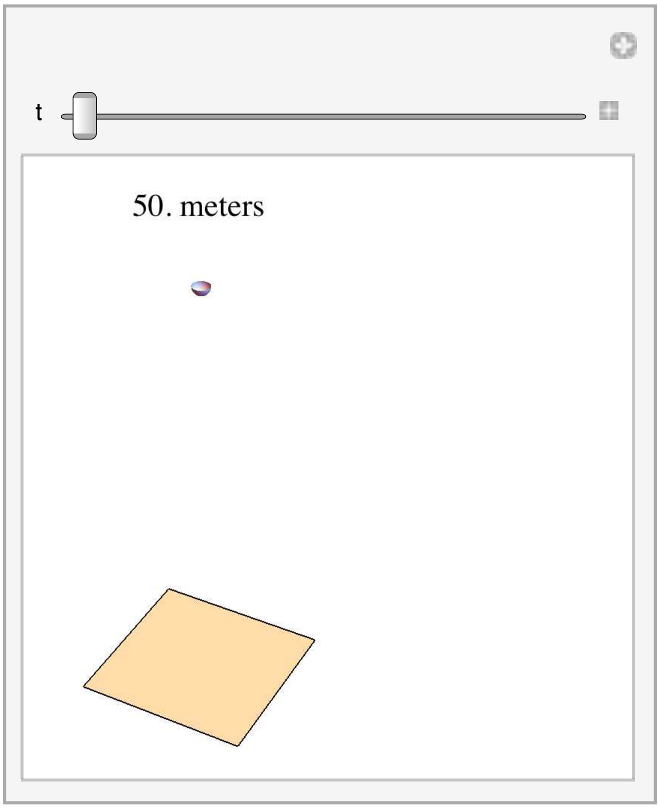
```
Manipulate[
  Graphics3D[{Blue, Opacity[0.5], Sphere[{0, 0, 0}, 1], Green, Opacity[0.5],
      Sphere[{x, y, z}, 1]},
    PlotRange → 2, Boxed → False],
  {x, −1, 1},
  {y, −1, 1},
  {z, −1, 1}
]
```



An earlier example used 2D graphics primitives to illustrate the behavior of dropping an object from a certain height. Since the language for 2D and 3D graphics primitives is so similar, this example can easily be changed to 3D by changing the primitives and adding any necessary arguments.

For example, **Disk** is changed to **Sphere**, and **Rectangle** is changed to **Polygon**. The **PlotRange** option setting is also updated to specify the viewing window in three dimensions.

```
DynamicModule[{d, t},
  Manipulate[
    Graphics3D[{Sphere[{10, 10, d[t]}],
        Polygon[{{0, 0, 0}, {0, 20, 0}, {20, 20, 0}, {20, 0, 0}}]},
      PlotRange → {{0, 20}, {0, 20}, {0, 50}}, PlotLabel → ToString[d[t]] <> " meters",
      Boxed → False],
    {t, 0, 3.2},
    Initialization :→ (d[t_] := 1/2 (-9.8) t^2 + 50)]]
```



Clear is used to remove all variable and function definitions from this chapter.

```
Clear[d, t]
```

Conclusion

Graphics primitives can be used to create diagrams, figures and models that range from very simple to sophisticated and complex. Since graphics primitives are Wolfram Language commands and tightly integrated with the rest of the system, graphics primitives can be used to programmatically generate graphical output suitable for modeling and simulating phenomena and for creating diagrams and figures.

Exercises

1. Use free-form input to create a triangle with lengths of 3, 4 and 5.
2. Use the Wolfram Language to create the same triangle, in terms of shape and dimensions, as shown in Exercise 1. Place the leftmost point at coordinate position (0, 0).
3. Use the Wolfram Language to change the color of the triangle in Exercise 2 from black to pink.
4. Use the Wolfram Language to change the border of the triangle in Exercise 3 to black.
5. Use the Wolfram Language to add a blue circle, centered at (3, 4) and with a radius of 1, to the triangle from Exercise 4.
6. Use the Wolfram Language to create a plot of $3\,x$, where x goes from $-\pi$ to π.
7. Use the **Epilog** and **Point** commands to draw a point on the plot of $3\,x$ at (0, 0).
8. Use the **PointSize** option to change the size of the **Point** graphics primitive.
9. Change the color of the point to red.
10. Use **Manipulate** to create an interactive model where the x value of the point can be changed from $-\pi$ to π. (Hint: Do not try to create manipulable parameters to set both the x and y values of the point. Instead, focus on the x value and then use that to calculate the y value using the $3\,x$ function.)

CHAPTER 14

Algebraic Manipulation and Equation Solving

Introduction

The Wolfram Language has a variety of commands for algebraic manipulation operations like expansion and factoring of polynomials, addition of fractions with unlike denominators and collection of terms with like variables. Following the language conventions that have already been discussed, these commands have names that describe exactly what they do, making it intuitive for new users to find the right commands for their needs. This chapter will discuss some of the most common functions for algebraic manipulation and equation solving, and it will also detail some methods for extracting results from the output that is returned by these commands.

Basic Algebraic Operations

Mathematica is a wonderful tool for handling the bookkeeping that can become so tedious when working with calculations by hand. Mathematica automatically simplifies and resolves expressions as needed, such as in the following example where *b* is canceled.

$$\frac{2\,a\,b}{b\,c}$$

$$\frac{2\,a}{c}$$

There are times when an expression may need to be transformed from one representation to another. For example, when given a list of symbolic quantities to multiply, Mathematica may return the result in simplest form—which may just happen to be the form the expression was already in. See the following for an example of this behavior in action.

```
(a + b) (a + c) (b + c)
```

```
(a + b) (a + c) (b + c)
```

In this situation, the **Expand** command can be used to expand the result of multiplying expressions, such as the ones in this example.

$$\textbf{Expand}[(a + b)\,(a + c)\,(b + c)]$$

$$a^2\,b + a\,b^2 + a^2\,c + 2\,a\,b\,c + b^2\,c + a\,c^2 + b\,c^2$$

Now that the expression is expanded, Mathematica will leave it alone unless it is instructed to change its representation into a different form. The **Factor** command can be used to go the other direction by factoring a polynomial.

$$\textbf{Factor}\left[a^2\,b + a\,b^2 + a^2\,c + 2\,a\,b\,c + b^2\,c + a\,c^2 + b\,c^2\right]$$

$$(a + b)\,(a + c)\,(b + c)$$

A common mistake is for new users to want to use **Factor** to calculate the prime factors of an integer. If you want to calculate the prime factors for an integer, the command you need to use is **FactorInteger**.

There are other commands that can be used to put expressions into different forms. **Together** is used to add fractions with unlike denominators, and **Apart** is used to break expressions into terms with minimal denominators.

$$\textbf{Together}\left[\frac{1}{(x + 1)} + \frac{1}{(x - 1)}\right]$$

$$\frac{2\,x}{(-1 + x)\,(1 + x)}$$

$$\textbf{Apart}\left[\frac{2\,x}{(-1 + x)\,(1 + x)}\right]$$

$$\frac{1}{-1 + x} + \frac{1}{1 + x}$$

You may have noticed that the initial expression given to **Together** was of the form $\frac{1}{(x+1)} + \frac{1}{(x-1)}$, and the result returned by **Apart** was of the form $\frac{1}{-1+x} + \frac{1}{1+x}$. This happens because of rules that govern how **StandardForm** renders expressions. Toggle the result to **TraditionalForm** using the methods previously discussed (for example, by clicking the **Cell** menu, choosing **Convert To** and selecting **TraditionalForm**) to see the difference in how the expression is presented.

Another command that is very useful when working with polynomials is **Collect**, which collects terms together that share the same powers for a certain specified symbol. For example, to collect all of the terms together of $a\,x^2 + b\,x^2\,y + c\,x\,y$ that involve x, invoke the **Collect** command as follows.

```
Collect[a x^2 + b x^2 y + c x y, x]
```

```
c x y + x^2 (a + b y)
```

And to collect all of the terms that involve y instead, change the second argument from x to y.

```
Collect[a x^2 + b x^2 y + c x y, y]
```

```
a x^2 + (c x + b x^2) y
```

Collect can also be given a list of symbols as its second argument in order to collect multiple sets of terms together. The following command collects all terms involving x separately from all terms involving y, and both are collected independently of other terms, like those involving $x\,y$.

```
Collect[a^2 y + 2 a b y + b^2 y + 2 a x y + 2 b x y + x^2 y + c^2 x^2 y + 2 c d x^2 y + d^2 x^2 y,
  {x, y}]
```

```
(a^2 + 2 a b + b^2) y + (2 a + 2 b) x y + (1 + c^2 + 2 c d + d^2) x^2 y
```

Since Mathematica handles algebraic computation and manipulation so effortlessly, it can be easy to scale up to quite complicated expressions rather quickly. As these results are used for further computation, the complexity of the results may increase. When that happens, a good remedy can be found by employing a simplification command like **Simplify** or **FullSimplify**. Both of these commands will accept an expression and will attempt to return a simplified form of that expression.

```
Simplify[Sin[x]^2 + Cos[x]^2]
```

```
1
```

```
FullSimplify[(a^2 + 2 a b + b^2) y + (2 a + 2 b) x y + (1 + c^2 + 2 c d + d^2) x^2 y]
```

```
((a + b)^2 + 2 (a + b) x + (1 + (c + d)^2) x^2) y
```

> What is the difference between **Simplify** and **FullSimplify**? Both commands have the goal of taking an expression and returning a simpler form, but **FullSimplify** will attempt more transformations and substitutions and may take more time to complete for a very complicated expression. One approach that many people use is to try **Simplify** first, and if it does not work, then try **FullSimplify** instead.

Both **Simplify** and **FullSimplify** can be given assumptions to further refine their attempts at simplification. For example, if **Simplify** is used with $\sqrt{x^2}$, it does not return any simpler forms than the input it was given.

```
Simplify[Sqrt[x^2]]
```

$$\sqrt{x^2}$$

However, if some additional information is passed to **Simplify**, like the fact that $x > 0$, a simpler form can be found.

```
Simplify[Sqrt[x^2], x > 0]
```

```
x
```

There are also specialized expand and simplification functions, like **TrigExpand** and **TrigReduce**, which work on trigonometric functions.

```
TrigExpand[Sin[x^2]*Cos[2 x]]
```

```
Cos[x]^2 Sin[x^2] - Sin[x]^2 Sin[x^2]
```

Basic Equation Solving

Many commands are available for equation solving, from solving over specific domains to returning general solutions to finding roots of equations. For new users, becoming familiar with just a few of the most common commands is sufficient to solve many different types of problems.

The **Solve** command does exactly what its name implies: it solves a given system of equations or inequalities for a particular variable or list of variables. To solve for an equation in one variable, just pass **Solve** the expression of interest and the variable to solve for.

```
Solve[x^2 + 2 x - 1 == 0, x]
```

```
{{x → -1 - √2}, {x → -1 + √2}}
```

Remember: when testing equality with commands like **Solve**, the double equal sign (==) is used. The single equal sign (=) is reserved for assigning values to variables, so if you try to use it in a command like **Solve**, you will receive an error message.

To solve for multiple variables, pass **Solve** a system of equations and a list of variables by placing the system of expressions in one list and the variables in another list.

```
Solve[{2 x + y == 12, x + 4 y == 34}, {x, y}]
```

```
{{x → 2, y → 8}}
```

Using Results from Solve

One of the most common challenges new users face is how to take results returned by **Solve** (and other commands that return results in a similar way) and put them into a more familiar form. The output from the **Solve** command is a list of rules, so when **Solve** returned the output of **{{x→2, y→8}}** for the preceding example, it means that a value of 2 for *x* and 8 for *y* would satisfy the equation. That can be verified by substituting these values back into the original system of equations.

```
{2 (2) + 8 == 12, 2 + 4 (8) == 34}

 {True, True}
```

To use these results, with *x* having a value of 2 and *y* having a value of 8, users can manually create variables for *x* and *y* and assign values to them. That is fine for simple cases, but what if there are too many variables to retype, or the results are long and complicated? In such cases, it is better to learn how to work with rules so users can automate the process of extracting the values on the right-hand side of the arrows in order to use those values in further calculations.

> You may be thinking, could Mathematica just return a list of variable assignments for commands like **Solve**? Could it just return a cell with **x = 2** and **y = 8** as the output? There are probably many, many good reasons why Mathematica does not do this, but an obvious one from our perspective is, what if you, the user, had already defined **x** and **y** in your notebook? Then using **Solve** would overwrite those values and likely cause all kinds of pandemonium.

The **ReplaceAll** command is the easiest way to get at the right-hand side of a rule or list of rules. **ReplaceAll** takes two arguments: an expression to transform and a rule or list of rules to apply. In the following example, the variable **x** is transformed into the value 2, which is then substituted back into the list, and then the result of that substitution is displayed.

```
ReplaceAll[{x, x + 1, x + 2}, x → 2]

 {2, 3, 4}
```

ReplaceAll also has a shorthand notation in the form of **/.**, which is quite commonly used. This allows the replacement operation to be appended almost as an afterthought; this is useful when a result contains a list of rules, at which point the **/.** operator and a list of replacement rules can be given to perform a transformation.

```
x /. x → y^2
```

```
y^2
```

> A way to mentally parse the preceding statement is to think of **/.** as "such that" and **→** as "goes to." Then you can interpret the last input as "take this expression *such that x goes to y*^{2}," which may help with your mental interpretation of the **/.** symbol.

Since **ReplaceAll** can be used with an expression that contains a list of rules, that means it can be used with the results returned by **Solve**. First, review an example of how output is returned by this command.

```
Solve[x^2 + 2 x – 1 == 0, x]
```

```
{{x → –1 – √2}, {x → –1 + √2}}
```

Now, the **ReplaceAll** command is used to get the right-hand values of the rules with a single input.

```
x /. Solve[x^2 + 2 x – 1 == 0, x]
```

```
{–1 – √2, –1 + √2}
```

If it is desirable to store the result as a variable that can be used later, the variable assignment can also become part of the same input.

```
results = x /. {{x → –1 – √2}, {x → –1 + √2}}
```

```
{–1 – √2, –1 + √2}
```

The values are now stored in the **results** variable and can be used in other calculations.

```
results * √3
```

$$\left\{\sqrt{3}\left(-1-\sqrt{2}\right),\sqrt{3}\left(-1+\sqrt{2}\right)\right\}$$

The same general approach can be extended to situations where multiple values are returned by **Solve**, by placing the variable names in a list and then applying the appropriate **ReplaceAll** command.

```
result = {x, y} /. Solve[{2 x + y == 12 && x + 4 y == 34}, {x, y}]
```

```
{{2, 8}}
```

When using **Solve** to find the solution to a system of equations, the result is returned as a list of lists in which each solution set is in a separate list. When there is only a single solution set, as in the preceding example, the extra set of braces is not needed. It can be discarded using the **First** command, which takes the first element of a list.

```
First[result]
```

```
{2, 8}
```

> A variable definition can be used on the right-hand side of a new variable definition to update the value of the variable. You can change the preceding input to **result=First[result]**; this will grab the first element from the list and then assign that value back to the **result** variable.

Other Commands for Solving Equations

Solve is great for finding solutions to equations and systems of equations, but there are some types of problems that are better suited to other, more specialized commands. For example, when **Solve** is used to find solutions to the equation $x^2 - y^3 = 1$, it returns a couple of results, but it also prints a helpful note that it may not have given all solutions.

```
Solve[x^2 - y^3 == 1, {x, y}]
```

```
••• Solve: Equations may not give solutions for all "solve" variables.
```

$$\left\{\left\{x\to-\sqrt{1+y^3}\right\},\left\{x\to\sqrt{1+y^3}\right\}\right\}$$

Commands like **Reduce** use different approaches and may be used to obtain completely general results.

```
Reduce[x^2 - y^3 == 1, {x, y}]
```

```
y == (-1 + x^2)^(1/3) || y == -(-1)^(1/3) (-1 + x^2)^(1/3) || y == (-1)^(2/3) (-1 + x^2)^(1/3)
```

The preceding result is quite readable even in **StandardForm**, but try converting it to **TraditionalForm** to see an even nicer, typeset version. As a reminder, one way you can convert a cell to **TraditionalForm** is by highlighting its cell bracket, right-clicking and choosing **TraditionalForm** from the **Convert To** menu.

If **Reduce** gives completely general results, then why would one ever want to use **Solve**? One reason may be the form in which results are returned. Since **Solve** returns a finite number of solutions, it returns them as a list of rules, which can be easily manipulated to extract the values from the right-hand side. **Reduce**, on the other hand, returns a closed-form solution in the form of an equivalence statement; this makes its results very readable but it is not as obvious to new users how to extract the results.

Another command for equation solving is **FindRoot**, which searches for a root near some given starting values of each variable. The following looks for a root of $\sin(x^2) - \cos(x)$ by starting a search near values when $x = \pi$.

```
FindRoot[Sin[x^2] - Cos[x], {x, π}]
```

```
{x → 3.29304}
```

Since **FindRoot** only searches for a root, it can find solutions for classes of problems that **Solve** is not suited for, like systems with an infinite number of solutions. However, **Reduce** has also been shown to be able to find infinite solution sets, so why use **FindRoot** at all? Again, it goes back to the desired form of the results. **Reduce** is great for delivering a general statement about a solution set, but if a numerical value is needed, then **FindRoot** delivers in a way that **Reduce** does not.

The following example shows that **Solve** returns a warning when looking for a solution to $x + e^x = \frac{1}{2}$.

```
Solve[x + ⅇ^x == 1/2, x]
```

```
    Solve: Inverse functions are being used by Solve, so some solutions may
      not be found; use Reduce for complete solution information.
 {{x → 1/2 (1 - 2 ProductLog[√ⅇ])}}
```

Reduce can be used to find the general solution.

```
Reduce[x + ⅇ^x == 1/2, x]
```

```
 C[1] ∈ Integers && x == 1/2 - ProductLog[C[1], √ⅇ]
```

And **FindRoot** returns the first numerical root found near the value $x = 0$.

```
FindRoot[x + ⅇ^x == 1/2, {x, 0}]
```

```
 {x → -0.266249}
```

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[results, result]
```

Conclusion

Manipulating algebraic expressions and solving equations is a broad topic with many available built-in Wolfram Language commands. While this chapter only briefly touched on a few of them, other commands and more specialized applications of the commands can be explored in more detail via the documentation.

Exercises

1. Use the Wolfram Language to factor the expression $x^3 + 3x^2 + x + 3$.

2. Use the **TraditionalForm** command to create a typeset version of the results from Exercise 1.

3. Use free-form input to solve the equation $x^2 + 7x - 8 = 0$.

4. Use free-form input to solve the system of equations $3x + 5y = 8$ and $x - 2y = 7$.

5. Use the Wolfram Language to numerically approximate the results from Exercise 4 to 10 digits.

6. Use the Wolfram Language to solve the equation $x^2 + 5x + 9 = 14$.

7. Use the Wolfram Language to retrieve the right-hand-side values of x from the results of Exercise 6.

8. Use the Wolfram Language to numerically approximate the results from Exercise 7 to four digits.

9. Use the **Reduce** command to find the general solution set for $3x^4 - 5y^3 = 11$.

10. Use the Wolfram Language to create a single input that plots $3\sin(3x) + 2\cos(x^2)$, where x goes from -5 to 5, and that also uses the **FindRoot** command to find the solution nearest to $x = 0$.

CHAPTER 15

Calculus

Introduction

One of the most common courses taught with Mathematica is calculus, from high school classes preparing for AP exams to universities, where Mathematica is a critical component of calculus sequences. Mathematica allows the exploration of topics that are untenable for pencil and paper computation, and the methods used by the calculus commands in the Wolfram Language can return results for essentially all expressions that have closed-form representations.

This chapter will introduce basic functionality related to commands for differentiation, limits and integration.

Differentiation

There are two commands used to take derivatives in Mathematica: **D**, which takes partial derivatives, and **Dt**, which takes total derivatives. Both commands expect to receive a function for their first argument, and then the variable (or variables) of interest. For example, **D** can be used to differentiate $x^2 \sin(x)$ with respect to x.

```
D[x^2 Sin[x], x]
```

```
x^2 Cos[x] + 2 x Sin[x]
```

Multiple derivatives can be computed by passing a list as the second parameter. The list should contain the variable along with the degree to specify what derivative should be calculated. The following command gives the third derivative of $x^2 \sin(x)$ with respect to x.

```
D[x^2 Sin[x], {x, 3}]
```

```
6 Cos[x] – x^2 Cos[x] – 6 x Sin[x]
```

Partial derivatives can also be taken by using the prime notation with the **'** symbol. For example, **Sin'[x]** will return the derivative of $\sin(x)$.

```
Sin'[x]
```

```
Cos[x]
```

The prime notation can be useful with user-defined functions whose names are similar to mathematical conventions for naming.

```
f[x_] := x^3 − 2 x^2 − 5 x + 6
```

```
f'[x]
```

```
−5 − 4 x + 3 x^2
```

The use of the **D** command and the **'** shorthand are equivalent.

```
{D[f[x], x], f'[x]}
```

```
{−5 − 4 x + 3 x^2, −5 − 4 x + 3 x^2}
```

> Remember: if you prefer your results to be more traditional looking, including having the exponents listed in decreasing order, then convert your output to **TraditionalForm**.

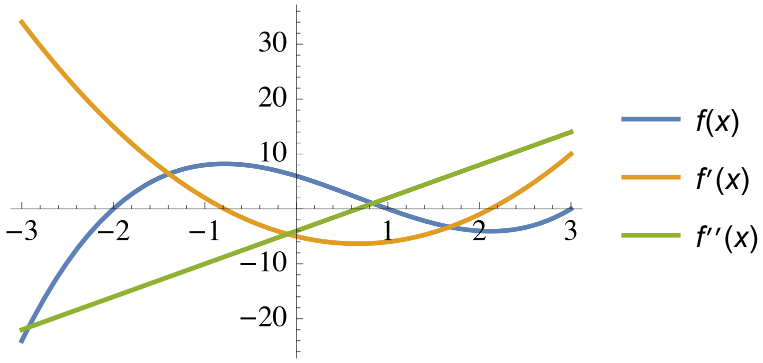
Along the same lines, the double prime notation, with shorthand **''**, is accepted for a second derivative, and so on.

```
{D[f[x], {x, 2}], f''[x]}
```

```
{−4 + 6 x, −4 + 6 x}
```

Since the prime notation is an acceptable form, it can be used with other Wolfram Language commands, like the following example that uses the **Plot** command to visualize a function and its first and second derivatives. (This is a good example of when adding a **PlotLegend** can be useful.)

```
Plot[{f[x], f'[x], f''[x]}, {x, −3, 3}, PlotLegends → "Expressions"]
```



The prime notation can also be useful for constructing a table of values of $f(x)$, $f'(x)$ and $f''(x)$ by using the **Table** command.

```
Table[{x, f[x], f'[x], f''[x]}, {x, 1, 10}] // TableForm
```

```
1     0      −6     2
2     −4     −1     8
3     0      10     14
4     18     27     20
5     56     50     26
6     120    79     32
7     216    114    38
8     350    155    44
9     528    202    50
10    756    255    56
```

As a reminder, the **//** operator allows you to apply a postfix operation to the output, which means that **TableForm[list]** is identical to **list // TableForm**. This is very useful for applying a function that changes the appearance of results, like using **TableForm** to display a dataset in a tabular layout and using **MatrixForm** to display values as a matrix.

D also supports differentiation with respect to multiple variables.

```
D[x^2 Cos[x y] + y^2 Sin[x y], x, y]
```

$-x^3 y \text{Cos}[x y] + 3 y^2 \text{Cos}[x y] - 3 x^2 \text{Sin}[x y] - x y^3 \text{Sin}[x y]$

The results returned by **D** may not be in the simplest possible form, so if a simpler form is suspected, simplification commands like **Simplify** can be used to check for one.

```
D[Sin[x]^10, {x, 4}]
```

```
5040 Cos[x]^4 Sin[x]^6 – 4680 Cos[x]^2 Sin[x]^8 + 280 Sin[x]^10
```

```
Simplify[%]
```

```
10 (141 + 238 Cos[2 x] + 125 Cos[4 x]) Sin[x]^6
```

Derivatives of purely symbolic forms can be taken as well, which can be useful for formal discussions on the topic. Here, the first derivative of the expression $x\,g(x)$ is taken; this is possible even though the symbol **g[x]** is undefined.

```
D[x g[x], x]
```

```
g[x] + x g′[x]
```

Just like previous examples, a multiple derivative can be taken.

```
D[x g[x], {x, 2}]
```

```
2 g′[x] + x g′′[x]
```

Once the formal idea is understood, rule replacement techniques can be used to substitute specific functions into the results.

```
D[g[x^2] g''[x], x] /. g → Sin
```

```
-2 x Cos[x^2] Sin[x] – Cos[x] Sin[x^2]
```

Limits

Limits can be found using the **Limit** command: as its input, it takes an expression and a variable with a value to approach, and it outputs the limiting value. The second argument is entered as a rule of the form **variable → value**. For example, to take the limit of $\frac{1}{x}$ as x approaches 1, use the following command.

```
Limit[1/x, x → 1]
```

```
1
```

Limits are often explored as values approach infinity, and Mathematica supports such computation. There are three ways to compute with the concept of infinity: use the built-in symbol **Infinity**, enter its symbolic form with a palette or use one of its keyboard shortcuts, like the escape sequence `Esc` inf `Esc`, to create the symbolic form **∞**. Both **Infinity** and **∞** are treated the same.

```
Limit[1/x, x → Infinity]
```

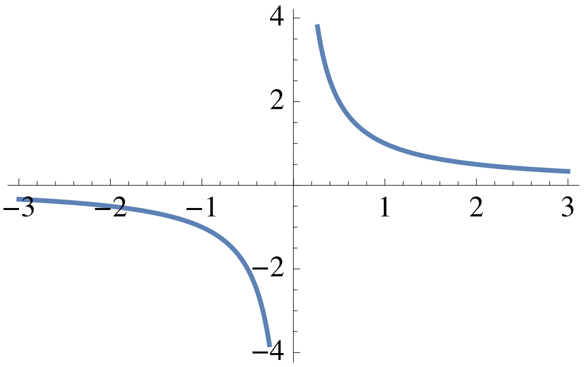
```
0
```

```
Limit[1/x, x → ∞]
```

```
0
```

The **Limit** command can accept an option to specify the direction of the limit. When the option is given as **Direction → 1**, the limit is approached from the left, and when the option is given as **Direction → -1**, the limit is approached from the right. If the direction is not specified, Mathematica will automatically determine which direction to use. Here is a plot of the function $\frac{1}{x}$, which has different limits from the left and right.

```
Plot[1/x, {x, -3, 3}]
```



Add the **Direction → 1** option setting to take the limit from the left.

```
Limit[1/x, x → 0, Direction → 1]
```

```
-∞
```

Add the **Direction → -1** option setting to take the limit from the right.

```
Limit[1/x, x → 0, Direction → -1]
```

```
∞
```

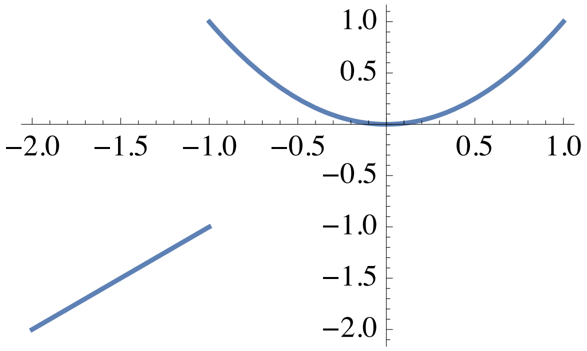
With no direction specified, Mathematica will choose the direction to take the limit. In this example, Mathematica approaches the limit from the right.

```
Limit[1/x, x → 0]
```

```
∞
```

The direction of the limit may be particularly important when used in conjunction with discontinuous piecewise functions.

```
f[x_] := { x     x < -1
         { x^2   x ≥ -1
Plot[f[x], {x, -2, 1}]
```



```
Limit[f[x], x → −1, Direction → −1]
```

```
1
```

```
Limit[f[x], x → −1, Direction → 1]
```

```
−1
```

If the **Limit** command cannot compute a limit, it will return an output that matches the input.

```
Limit[x^a, a → ∞]
```

```
Limit[x^a, a → ∞]
```

For situations where a limit is assumed to exist, assumptions can be passed to the **Limit** command. This will give the command additional information to take into consideration when attempting to compute the limit. Adding the assumption that $x > 1$ allows a limit to be found for x^a as a approaches ∞.

```
Limit[x^a, a → ∞, Assumptions → x > 1]
```

```
∞
```

Different assumptions may change what the limit is or whether it even exists.

```
Limit[x^a, a → ∞, Assumptions → x == 1]
```

```
1
```

```
Limit[x^a, a → ∞, Assumptions → −1 < x < 0]
```

```
0
```

Integration

There are two powerful commands used to take integrals in the Wolfram Language. **Integrate** can be used to compute definite and indefinite integrals, and **NIntegrate** can be used for numerical approximations of integrals.

Indefinite Integration

The **Integrate** command has ∫ as its symbolic form. The symbolic form can be entered with palettes or the Esc int Esc escape sequence. These forms of **Integrate** are interchangeable, so integrating $x^2 + 2\,x + 1$ with respect to x can be achieved by either of the following examples.

$$\textbf{Integrate}[x^2 + 2\,x + 1,\, x]$$

$$x + x^2 + \frac{x^3}{3}$$

$$\int (x^2 + 2\,x + 1)\, \mathbb{d} x$$

$$x + x^2 + \frac{x^3}{3}$$

If you use the symbolic form of **Integrate**, you need to include the 𝕕 operator to indicate the variable for the integration. If you use the palettes to paste in the integral symbol, you will get a template with the 𝕕 included. If you prefer keyboard shortcuts, then you can use Esc int Esc to create the integral symbol and Esc dd Esc to create the 𝕕.

When using the symbolic form of **Integrate**, the right-hand side may need to be enclosed in parentheses in order for Mathematica to correctly interpret the input. This is not necessary when the function being integrated has a single term, like sin(x).

$$\int \textbf{Sin}[x]\, \mathbb{d} x$$

```
-Cos[x]
```

If the function being integrated has multiple terms, however, they need to be placed in a set of parentheses. If they are not, then Mathematica will return an error.

$$\int \text{Sin[x]} + \text{Cos[x]}\, d\!\!\!/\, \text{x}$$

••• Integrate : $\int$Sin[x] cannot be interpreted. Integrals are entered in the form $\int$f $d\!\!\!/$x, $\int_a^b$f $d\!\!\!/$x, or $\int_{\text{vars} \in \text{region}}$ f, where $d\!\!\!/$ is entered as ESC dd ESC.

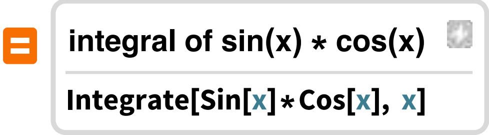
The preceding example failed because the **$d\!\!\!/$x** was only associated with **Cos[x]**. Enclosing the entire expression in a set of parentheses gives Mathematica the necessary information to perform the calculation.

$$\int (\text{Sin[x]} + \text{Cos[x]})\, d\!\!\!/\, \text{x}$$

```
-Cos[x] + Sin[x]
```

A common question new users ask when integrating is, but wait—where is the constant of integration? When Mathematica computes an integral of a function, say **f**, it does not return an entire family of results; instead, it returns an expression whose derivative is mathematically equivalent to **f**. For the sake of simplicity, then, a constant is not returned when Mathematica computes an integral.

Like many other operations in Mathematica, integration can be performed with free-form input.



```
integral of sin(x) * cos(x)
Integrate[Sin[x]*Cos[x], x]
```

$$-\frac{1}{2}\text{Cos[x]}^2$$

Expanding the integration results returned by free-form input may reveal other interesting pieces of information, like series expansions and alternate forms.

Definite Integration

Integrate can also be used to compute definite integrals. This form of the command takes a function as its first argument, and then a second argument in the form of a list containing the variable and the bounds for the integration. These bounds can be given in numeric or symbolic form, or a mixture thereof. For example, the following command will compute the integral of $x^2 e^x$ from 0 to 1.

```
Integrate[x^2 ⅇ^x, {x, 0, 1}]
```

```
-2 + ⅇ
```

You can enter ⅇ with a palette or by using the Esc ee Esc escape sequence.

A list of symbolic bounds could be passed as the second argument instead.

```
Integrate[x^2 ⅇ^x, {x, a, b}]
```

```
-(2 + (-2 + a) a) ⅇ^a + (2 + (-2 + b) b) ⅇ^b
```

If you convert an input containing the **Integrate** command into **TraditionalForm**, the command will display as a typeset integral, but it can still be evaluated to perform a computation.

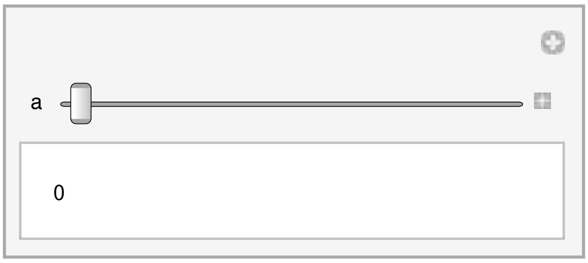
Bounds can contain numeric and symbolic values. Here, $x^2 e^x$ is integrated from 0 to a.

```
Integrate[x^2 ⅇ^x, {x, 0, a}]
```

```
-2 + (2 + (-2 + a) a) ⅇ^a
```

This use of symbols as bounds lends itself to using **Manipulate** to explore the effect of changing the value of a.

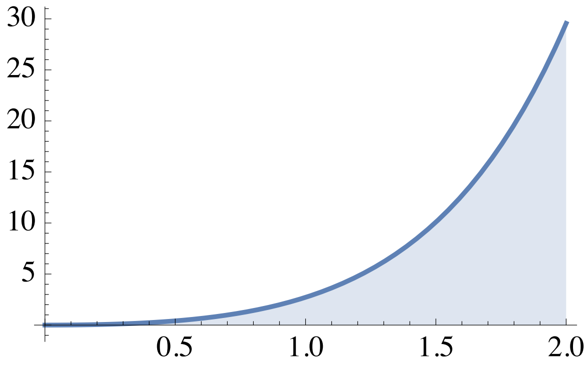
```
Manipulate[
  Integrate[x^2 ⅇ^x, {x, 0, a}],
  {a, 0, 8}]
```



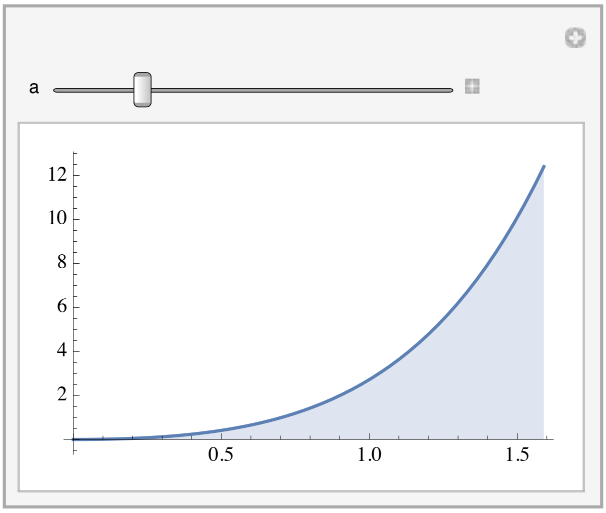
```
0
```

When discussing integration, it can be handy to use the **Filling** option to illustrate the concept of integration as a calculation for the area under a curve.

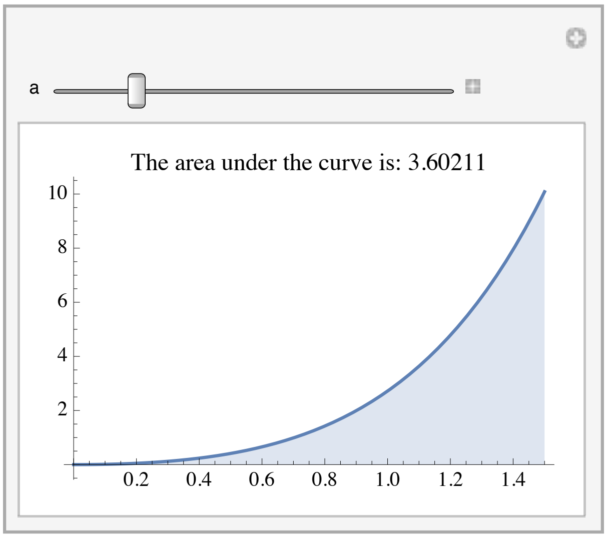
```
Plot[x^2 ⅇ^x, {x, 0, 2}, Filling → Axis]
```



The ideas of the two preceding examples can be combined to create a single **Manipulate** command that both calculates a definite integration and visualizes its representation. First, a **Manipulate** to create an interactive plot is created.



Now, a **PlotLabel** is used that displays the result of the integration. Here, **ToString** is used to create a plot label that prints the result from integrating $x^2\ e^x$ from 0 to a, where a is being manipulated by the slider bar.



Definite integrals have a symbolic form that can be entered with a palette or through the escape sequence Esc dintt Esc, which will print a typeset template that can be filled out quickly and easily.

$$\int_{a}^{b} \mathrm{Sin}[x]\, dx$$

```
Cos[a] – Cos[b]
```

> If you use the keyboard shortcut to create the definite integral template, you will need to add the ⅆ symbol, just like when using the symbolic form for indefinite integration.

The goal of **Integrate** is to provide an exact antiderivative of the function in question. As such, **Integrate** returns exact results, even in the case of definite integrals, as in the following example.

$$\mathrm{Integrate}\left[x^{2}\, e^{x}, \{x, 0, 1\}\right]$$

$$-2 + e$$

If a numerical approximation is needed, one can be obtained with the **N** command.

```
N[%]
```

```
0.718282
```

If a numerical approximation is the desired result, then the **NIntegrate** command can be used instead.

$$\textbf{NIntegrate}\left[x^{2}\, e^{x}, \{x, 0, 1\}\right]$$

```
0.718282
```

> If **N[Integrate[expr]]** and **NIntegrate[expr]** arrive at the same result, why would you use one over the other? **NIntegrate** focuses on purely numerical methods, so it will not spend time trying to find a closed-form solution. But if a closed-form solution is what you are after, then use **Integrate**; you can always use **N** to numerically approximate the results later on.

Integrate can be used to compute multiple integrals by using the command name directly or by using its symbolic form. The integration variables are entered to match how the integral signs appear, but the last variable entered will be the innermost integral and the one that is computed first. In the following example, the expression is first integrated with respect to y from -2 to x and *then* integrated with respect to x from -1 to 1.

$$\textbf{Integrate}\left[x^{3}\, \textbf{Sin}[y] + y^{2}\, \textbf{Cos}\left[x^{2}\right], \{x, -1, 1\}, \{y, -2, x\}\right]$$

$$\frac{8}{3}\sqrt{2\,\pi}\ \text{FresnelC}\left[\sqrt{\frac{2}{\pi}}\right]$$

The following command is equivalent to the preceding one but helps to illustrate this point about the order of the integration variables.

$$\int_{-1}^{1}\int_{-2}^{x}\left(x^3\ \text{Sin}[y]+y^2\ \text{Cos}\left[x^2\right]\right) dy\, dx$$

$$\frac{8}{3}\sqrt{2\,\pi}\ \text{FresnelC}\left[\sqrt{\frac{2}{\pi}}\right]$$

If the inputs for these multiple integration examples are confusing, this tip might help: create a set of nested **Integrate** function calls. The following example is identical to the ones above, but it may help you think through how output from the first **Integrate** is passed as input to the second **Integrate**.

```
Integrate[
 Integrate[x^3 Sin[y] + y^2 Cos[x^2], {y, -2, x}],
 {x, -1, 1}]
```

$$\frac{8}{3}\sqrt{2\,\pi}\ \text{FresnelC}\left[\sqrt{\frac{2}{\pi}}\right]$$

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[f]
```

Conclusion

While this chapter provides a succinct overview of the commands for differentiation, taking limits and integration, there is a great deal more discussion and thousands of examples in the documentation for those who want to explore the topics in more detail. The next chapter will focus on a related topic by introducing readers to the differential equation-solving commands available in the Wolfram Language.

Exercises

1. Use free-form input to find the derivative of $x^3 + 4\,x^2 - x$.

2. Use the Wolfram Language to create a single input that finds the derivatives of $7\,x^3 + \sin(x)$ and $x^2 + \tan(x)$.

3. Use the Wolfram Language to write a compound expression that defines a function $f(x) = x^3 + 5\,x^2 - 4$ and then finds the second derivative of that function by using the prime notation.

4. Use free-form input to find the limit of $\frac{x^2+2\,x}{x^2}$.

5. In Exercise 4, the use of free-form input required Mathematica to make an assumption of what value x should approach. Use the Wolfram Language to find the limit of the same expression when x goes to infinity.

6. Use the Wolfram Language to find the limit of $\frac{\sin\left(\sqrt{x}\right)}{x}$ when x goes to 10.

7. Copy and paste the input and output cells from Exercise 6 and then convert both to **TraditionalForm**.

8. Use free-form input to find the indefinite integral of $\sin(x) + 3\,x^2 - 9$.

9. Use the Wolfram Language to find the numerical approximation of the definite integral of $\cos(x^2) + \sin(x)$, where x goes from 0 to 1.

10. Use the Wolfram Language to find the numerical approximation of the double integral of $5\,x^5 + 3\,\sin(x^3) - 4\,y^4 + 9\,\cos(y^2)$ with respect to y and then with respect to x, where x and y both range from 0 to 1.

CHAPTER 16

Differential Equations

Introduction

Mathematica can symbolically or numerically solve differential equations, including ordinary differential equations (ODEs), partial differential equations (PDEs), differential-algebraic equations (DAEs) and delay differential equations (DDEs).

The differential equation solving commands are excellent examples of the automation that is built into the Wolfram Language. Rather than forcing a user to choose an appropriate algorithm to solve a particular equation or system of equations, these commands just require the user to provide the equations themselves. Then, Mathematica analyzes the system and determines which algorithm to use, sometimes automatically switching mid-operation if a better choice is found.

Solving Symbolically with **DSolve**

The **DSolve** command is used to solve linear and nonlinear ODEs, linear and weakly nonlinear PDEs, and DAEs. The command accepts input to solve for a single equation or system of equations. The following example solves $y'(x) = x^2 \sin(x)$ for the function y, with independent variable x.

```
DSolve[y'[x] == x^2 Sin[x], y[x], x]
```

```
{{y[x] → C[1] - (-2 + x^2) Cos[x] + 2 x Sin[x]}}
```

As a reminder, you need to use the double equal sign **==** when solving for equations. The single equal sign is reserved for assigning values to variables.

Since the preceding example did not contain any initial conditions, the solution returned by **DSolve** includes a notation for **C[1]**, which is a placeholder for a parameter. Substituting a value into this parameter will provide a specific solution, and more detailed examples are outlined later in this chapter.

To give initial conditions for an equation, place the equation and the initial conditions in a list, and then pass that as the first argument to **DSolve**. The following example builds on the previous one but includes an initial condition by using a list for the first argument.

```
DSolve[{y'[x] == x^2 Sin[x], y[1] == 1}, y[x], x]
```

```
{{y[x] → 1 − Cos[1] + 2 Cos[x] − x^2 Cos[x] − 2 Sin[1] + 2 x Sin[x]}}
```

This time, a specific solution corresponding to this initial condition is returned.

Notice that if the value of 1 were substituted for **C[1]** in the first example, the resulting expression would exactly match the output in the second example.

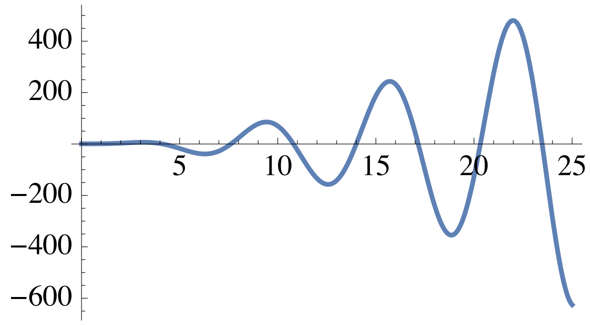
Since results from **DSolve** are returned as rules, the methods detailed in earlier chapters can be used to extract these results in order to use them with other commands. However, there is also a different command, **DSolveValue**, that can be used to extract the value from a solution without the need for additional processing. Many times, this is the appropriate function to use.

```
soln = DSolveValue[{y'[x] == x^2 Sin[x], y[1] == 1}, y[x], x]
```

```
1 − Cos[1] + 2 Cos[x] − x^2 Cos[x] − 2 Sin[1] + 2 x Sin[x]
```

Since the output is returned as an expression, it can be immediately used with other commands, like feeding the result into the **Plot** command to visualize the result.

```
soln = DSolveValue[{y'[x] == x^2 Sin[x], y[1] == 1}, y[x], x];
Plot[soln, {x, 0, 25}]
```



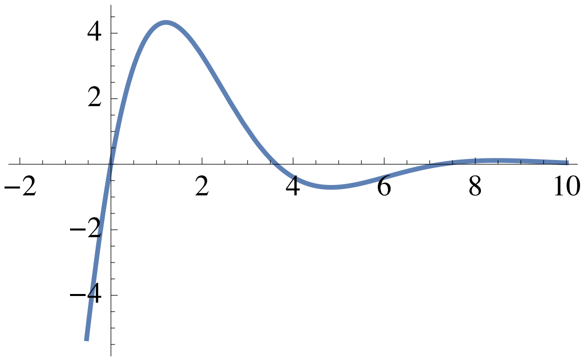
Higher-order equations can be solved in the same manner. In this example, multiple initial conditions are given to identify a particular solution.

```
soln = DSolveValue[{y''[x] + y'[x] + y[x] == 0, y[0] == 0, y'[1] == 1}, y[x], x]
```

$$\frac{2\,e^{1/2-x/2}\,\mathrm{Sin}\left[\frac{\sqrt{3}\,x}{2}\right]}{\sqrt{3}\,\mathrm{Cos}\left[\frac{\sqrt{3}}{2}\right]-\mathrm{Sin}\left[\frac{\sqrt{3}}{2}\right]}$$

And now the result is visualized.

```
Plot[soln, {x, −2, 10}]
```



When initial conditions are not passed to **DSolveValue**, it returns general solutions. For example, removing the initial conditions from the previous example returns two different placeholders for values: **C[1]** and **C[2]**.

```
soln = DSolveValue[{y''[x] + y'[x] + y[x] == 0}, y[x], x]
```

$$e^{-x/2}\,C[2]\,\mathrm{Cos}\left[\frac{\sqrt{3}\,x}{2}\right]+e^{-x/2}\,C[1]\,\mathrm{Sin}\left[\frac{\sqrt{3}\,x}{2}\right]$$

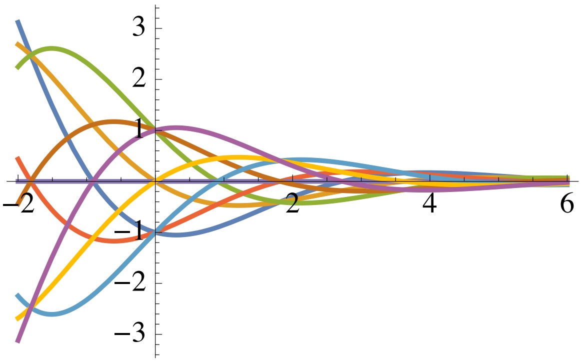
Values need to be given to **C[1]** and **C[2]** to arrive at a specific solution. It might be useful to explore several different solutions at once. In order to do that, the **Table** command can be used to generate a list of solutions by replacing **C[1]** with **i** and **C[2]** with **j** and then iterating over **i** and **j**. The results are stored in a new variable named **solnTable**.

```
solnTable = Table[soln /. {C[1] → i, C[2] → j},
    {i, −1, 1, 1}, {j, −1, 1, 1}]
```

$$\left\{\left\{-e^{-x/2}\,\text{Cos}\left[\frac{\sqrt{3}\,x}{2}\right]-e^{-x/2}\,\text{Sin}\left[\frac{\sqrt{3}\,x}{2}\right],\right.\right.$$
$$-e^{-x/2}\,\text{Sin}\left[\frac{\sqrt{3}\,x}{2}\right],\,e^{-x/2}\,\text{Cos}\left[\frac{\sqrt{3}\,x}{2}\right]-e^{-x/2}\,\text{Sin}\left[\frac{\sqrt{3}\,x}{2}\right]\Big\},$$
$$\Big\{-e^{-x/2}\,\text{Cos}\left[\frac{\sqrt{3}\,x}{2}\right],\,0,\,e^{-x/2}\,\text{Cos}\left[\frac{\sqrt{3}\,x}{2}\right]\Big\},\,\Big\{-e^{-x/2}\,\text{Cos}\left[\frac{\sqrt{3}\,x}{2}\right]+e^{-x/2}\,\text{Sin}\left[\frac{\sqrt{3}\,x}{2}\right],$$
$$\left.\left.e^{-x/2}\,\text{Sin}\left[\frac{\sqrt{3}\,x}{2}\right],\,e^{-x/2}\,\text{Cos}\left[\frac{\sqrt{3}\,x}{2}\right]+e^{-x/2}\,\text{Sin}\left[\frac{\sqrt{3}\,x}{2}\right]\right\}\right\}$$

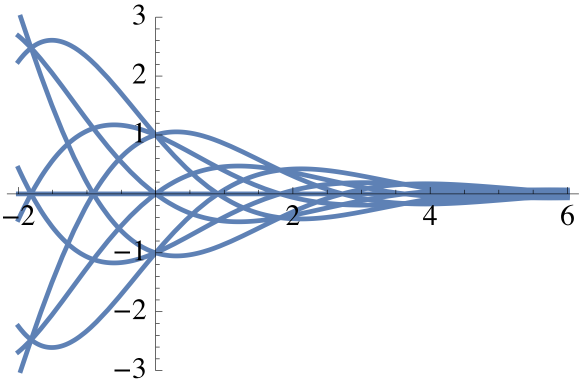
Now that the table of solutions—which are really just a bunch of mathematical expressions—is generated, the solutions can be visualized as a plot.

```
Plot[solnTable, {x, −2, 6}, PlotRange → All]
```



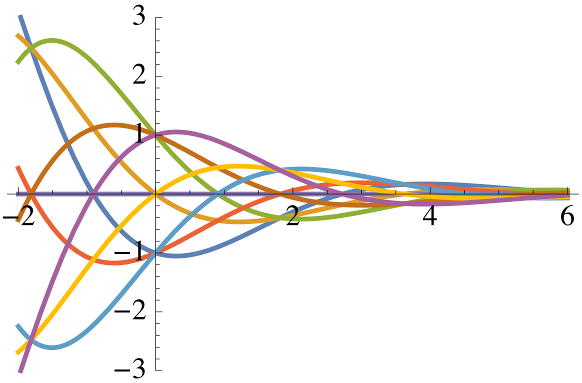
Of course, creating this table of results and then plotting it can be done in a single step, but it exposes a behavior that may seem odd at first.

```
Plot[
  Table[soln /. {C[1] → i, C[2] → j},
    {i, −1, 1, 1}, {j, −1, 1, 1}],
  {x, −2, 6}, PlotRange → {−3, 3}]
```



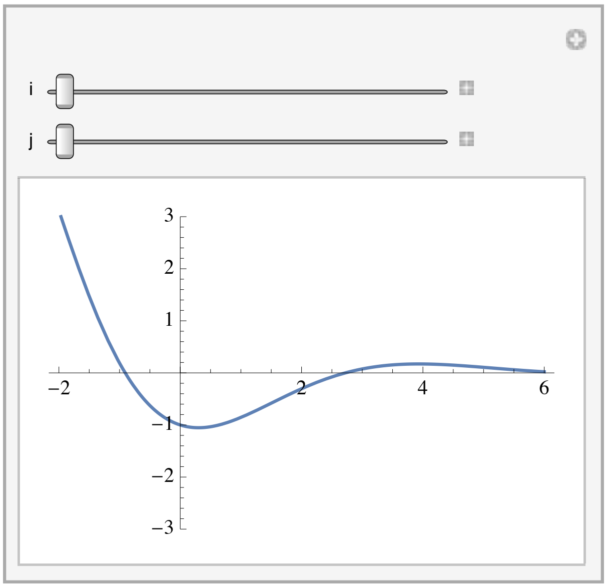
In the preceding example, all the functions are colored the same instead of the different colors that Mathematica uses by default. This behavior has to do with Mathematica's internal order of operations and how its plotting routines are implemented. Instead of getting into too much detail about why this happens, a solution can be pointed to instead: the **Evaluate** command can be used to force evaluation of the table before the arguments are given to **Plot**, which means they will then be treated as individual curves and will automatically be colored differently.

```
Plot[
  Evaluate[
    Table[soln /. {C[1] → i, C[2] → j},
      {i, −1, 1, 1}, {j, −1, 1, 1}]],
  {x, −2, 6}, PlotRange → {−3, 3}]
```



Of course, anything that can be represented with a static table in Mathematica can be made much more interesting by using **Manipulate** instead. The same is true for interactively exploring solutions to differential equations. The following example prints a single solution to the differential equation but allows the user to choose the solution by adjusting the values of **C[1]** and **C[2]** with slider bars.

```
Manipulate[
  Plot[
    soln /. {C[1] → i, C[2] → j} /. {C[1] → i, C[2] → j},
    {x, −2, 6}, PlotRange → {−3, 3}],
  {i, −1, 1, 1},
  {j, −1, 1, 1},
  Initialization :→ (soln = DSolveValue[{y''[x] + y'[x] + y[x] == 0}, y[x], x])]
```



Solving Numerically with **NDSolve**

The **NDSolve** command is used to solve ODEs, PDEs and DDEs. When passing input to this function, adequate initial or boundary conditions must be given for **NDSolve** to be able to completely determine the corresponding solutions. There is also a function, **NDSolveValue**, that can be used to return results without the need for additional processing. Here, the equation $y'(x) = x^2 \sin(x)$ is solved for the function y with the boundary condition $y(1) = 1$ and with independent variable x in the range from 0 to 10.

```
NDSolveValue[{y'[x] == x^2 Sin[x], y[1] == 1}, y, {x, 0, 10}]
```

```
InterpolatingFunction[ Domain: {{0., 10.}}
                       Output: scalar ]
```

NDSolveValue returns results as **InterpolatingFunction** objects. These are approximating functions and can be used like other functions once they are extracted from the rule list.

```
nSoln = NDSolveValue[{y'[x] == x^2 Sin[x], y[1] == 1}, y, {x, 0, 10}]
```

```
InterpolatingFunction[ Domain: {{0., 10.}}
                       Output: scalar ]
```

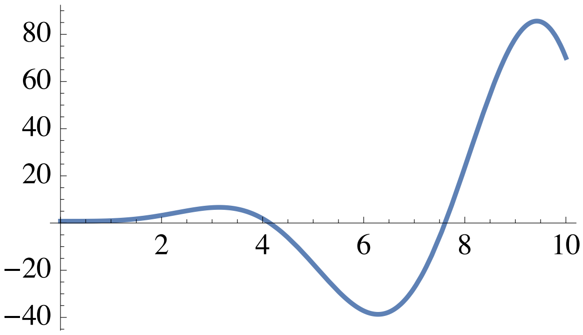
Now the solution is stored in the variable **nSoln** and can be used with other Wolfram Language commands. It can be evaluated at a particular value, like a mathematical function.

```
nSoln[5]
```

```
-17.3367
```

It can be visualized by using a plotting command.

```
Plot[nSoln[x], {x, 0, 10}]
```



It can also be integrated.

```
NIntegrate[nSoln[x], {x, 0, 10}]
```

```
72.4684
```

Like **DSolveValue**, the **NDSolveValue** command can be used for solving systems of differential equations, assuming all necessary initial and boundary conditions are passed as arguments. The following example solves a system of differential equations.

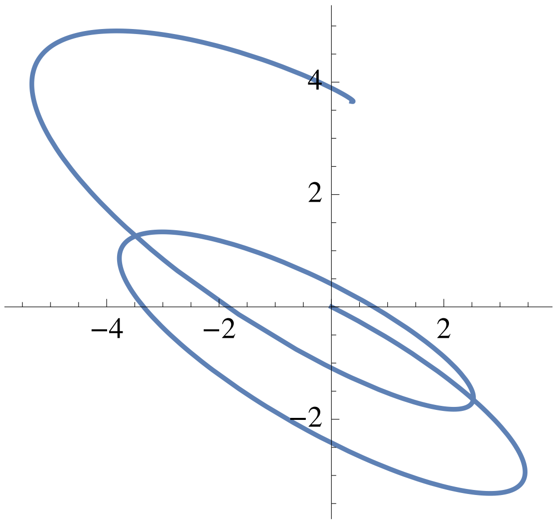
```
sysSoln =
  NDSolveValue[{x'[t] - 4 x[t] + y''[t] == t^2/2, x'[t] + 2 x[t] + 2 y'[t] == 0, x[0] == 0,
    x[1] == 1, y[0] == 0}, {x[t], y[t]}, {t, 0, 5}]
```

```
{InterpolatingFunction[ Domain: {{0., 5.}}
                        Output: scalar ][t],

 InterpolatingFunction[ Domain: {{0., 5.}}
                        Output: scalar ][t]}
```

Two **InterpolatingFunction** objects are returned, which can then be visualized in the phase plane by using **ParametricPlot**.

```
ParametricPlot[sysSoln, {t, 0, 5}]
```



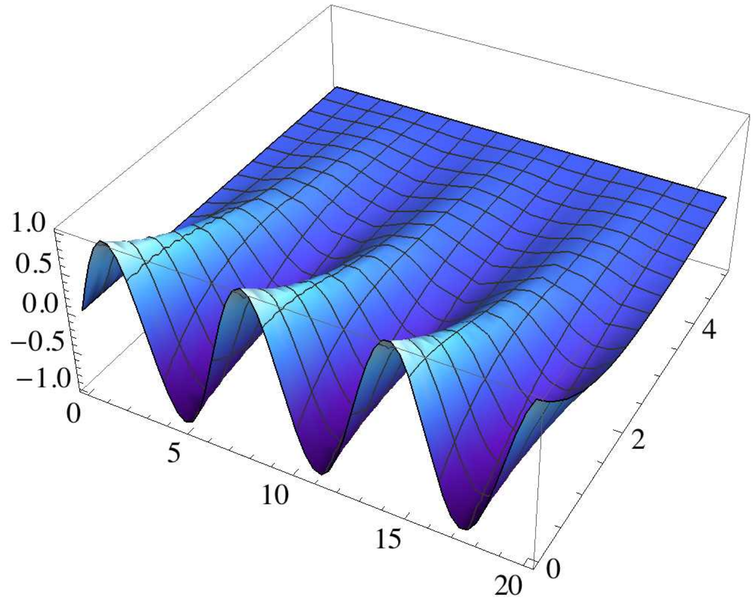
Solving partial differential equations follows a similar approach: multivariate equations are passed to **NDSolveValue** along with the appropriate boundary conditions and regions.

```
pdeSoln =
  NDSolveValue[{D[u[t, x], t] == D[u[t, x], x, x], u[0, x] == 0, u[t, 0] == Sin[t], u[t, 5] == 0},
    u[t, x], {t, 0, 20}, {x, 0, 5}]
```

```
InterpolatingFunction[ Domain: {{0., 20.}, {0., 5.}}
                       Output: scalar ][t, x]
```

As before, results are returned as **InterpolatingFunction** objects, which can become the input for other commands, like **Plot3D**.

```
Plot3D[pdeSoln, {t, 0, 20}, {x, 0, 5}, PlotRange → All, ColorFunction → "DeepSeaColors"]
```



Why did the preceding example use the **ColorFunction** option instead of the default styling? Because this was a *wave* equation. (That was an attempt at a joke, and along those same lines, feel free to use a different setting for that option; there are other appropriate choices, such as **"BeachColors"**, **"IslandColors"** and **"LakeColors"** as available named color gradients.)

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[soln, solnTable, nSoln, sysSoln, pdeSoln]
```

Conclusion

Solving differential equations in Mathematica can help model a wide variety of physical phenomena without addressing the involved mechanics of solving them by hand. The differential equation solving commands are some of the most sophisticated and complex functions in the Wolfram Language, and the documentation on these commands is quite comprehensive, with tutorials that are hundreds of printed pages long. This book should not be seen as a reference but rather a very basic introduction to how to use these commands and work with the results they return.

Exercises

1. Symbolically solve the differential equation $y'(x) = x^3 \cos(x)$.
2. Symbolically solve the same equation from Exercise 1, but include the condition that $y(3) = 6$.
3. Plot the solution from Exercise 2, where x goes from 0 to 10.
4. Symbolically solve the differential equation $y''(x) + y'(x) = x^2 \sqrt{x}$.
5. Extract a specific solution from Exercise 4, where $C[1] = 3$ and $C[2] = 6$.
6. Plot the specific solution from Exercise 5, where x goes from 0 to 1.
7. Use **Table** to create a list of solutions for Exercise 4, where $C[1] \to i$ and $C[2] \to j$, i goes from -1 to 1 in steps of 0.25, and j goes from 0 to 1 in steps of 0.25.
8. Plot the solution set from Exercise 7. (Hint: be sure to use **Evaluate** so that each solution set is treated as an individual curve.)
9. Numerically solve the differential equation $y'(x) = x^3 \cos(x)$ and $y(3) = 6$, where x goes from 0 to 5.
10. Plot the solution to Exercise 9, where x goes from 1 to 3.

CHAPTER 17

Linear Algebra

Introduction

Mathematica includes and uses highly efficient libraries for linear algebra and can work with both numeric and symbolic vectors and matrices. Since both vectors and matrices are represented as lists in Mathematica, the same suite of data-filtering commands can be used on either vectors or matrices. In addition, there are specialized graphics commands that are quite useful for visualizing results. All standard linear algebra operations are supported for both symbolic and numerical work with arbitrary precision, and there is no need for a user to keep track of row vectors differently than column vectors, as is the case in some other software.

Vectors

Vectors in Mathematica are represented by lists. There is no need to specify if a particular vector is a row vector or column vector; this makes things easier for the user and keeps the focus on the result instead of mathematical bookkeeping.

Vectors can be constructed explicitly or programmatically. To define a vector manually, just create a list.

```
vec1 = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
```

```
{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
```

To define a vector programmatically, use **Table**.

```
vec2 = Table[i^2, {i, 1, 10}]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

There is also a function, **Array**, that constructs a vector from a function. **Table** can do this as well, but it requires specification for the iterator, while **Array** assumes the iterator and only requires its bound.

```
myFunction[x_] := x Sin[x]
vec3 = Array[myFunction, 5]
```

```
{Sin[1], 2 Sin[2], 3 Sin[3], 4 Sin[4], 5 Sin[5]}
```

Vectors can contain a mixture of different elements, like exact numbers, approximate numbers and symbols.

```
vec4 = {a, c, π, ⅇ, 1, 2, 3, 4.5, 5.6}
```

```
{a, c, π, ⅇ, 1, 2, 3, 4.5, 5.6}
```

To test whether an object is a vector, the testing function **VectorQ** can be used. All the vectors defined in this chapter—**vec1**, **vec2**, **vec3**, **vec4**—pass this test.

```
{VectorQ[vec1], VectorQ[vec2], VectorQ[vec3], VectorQ[vec4]}
```

```
{True, True, True, True}
```

Mathematical operations like addition, subtraction, multiplication, division and exponentiation can operate on vectors of the same length. The operations are applied element-wise, so the operation is performed on the first two elements of the vector, then the operation is performed on the second two elements of the vector and so on.

```
vec1 + vec2
```

```
{2, 7, 14, 23, 34, 47, 62, 79, 98, 119}
```

```
vec1 * vec2
```

```
{1, 12, 45, 112, 225, 396, 637, 960, 1377, 1900}
```

Sometimes new users expect the multiplication of two vectors to give the dot product instead of performing element-wise multiplication. To compute a dot product, use the **Dot** command or **.**, which is a shorthand form.

```
Dot[{a, f}, {c, d}]
```

```
a c + d f
```

```
{a, f}.{c, d}
```

```
a c + d f
```

Other common vector operations include cross products and norms. Cross products can be computed with **Cross** or **⨯**, which is a shorthand form. Norms are computed with the **Norm** command.

If you are looking for the **Cross** operator, be sure you do not accidentally use the **Times** operator instead. They look very similar, but the cross product (**⨯**) is drawn smaller than **Times** (**×**). You will not find a button for **Cross** in the Assistant palettes, but you can click the **Palettes** menu and choose **Special Characters** to find it on one of the symbol tabs for that palette. You can also use the escape sequence Esc cross Esc to enter the **Cross** operator.

```
Cross[{1, 3, 5}, {π, ⅇ, 0}]
```

```
{−5 ⅇ, 5 π, ⅇ − 3 π}
```

```
{1, 3, 5} ⨯ {π, ⅇ, 0}
```

```
{−5 ⅇ, 5 π, ⅇ − 3 π}
```

```
Norm[vec3]
```

$$\sqrt{\text{Sin}[1]^2 + 4\,\text{Sin}[2]^2 + 9\,\text{Sin}[3]^2 + 16\,\text{Sin}[4]^2 + 25\,\text{Sin}[5]^2}$$

As with any exact result, the **N** command can be used to get a numerical approximation.

```
N[{1, 3, 5} ⨯ {π, ⅇ, 0}]
```

```
{−13.5914, 15.708, −6.7065}
```

```
N[Norm[vec3]]
```

```
6.02885
```

Other commands for working with vector spaces—computing vectors related to angles (**AngleVector** and **VectorAngle**), normalization of vectors (**Normalize**), projection of vectors (**Projection**) and orthogonalization of vectors (**Orthogonalize**)—and computing measures of distance are also available.

Matrices

The Wolfram Language uses nested lists to represent matrices. Like vectors, matrices can be comprised of any sort of expression: symbols, numbers, strings and images, and even mixtures thereof.

Constructing Matrices

Matrices are constructed by creating nested lists. Small matrices can be manually entered by typing, and the simplest method is to create a nested list in one-dimensional format using list notation. In this example, a nested list is created and assigned to the variable **mat1**, and then **MatrixForm** is used to display the result in two-dimensional format.

```
mat1 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
MatrixForm[mat1]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Matrices can also be entered with palettes like the **Basic Math Assistant**. The **Basic Commands** section of the palette has a tab for matrix commands, including a button that will paste an empty 2 × 2 matrix into a notebook. There are buttons to add rows and columns to newly created matrices, providing users with an interactive way to construct a template for a larger matrix. For example, clicking the matrix button to create a 2 × 2 matrix template and then clicking the **Add Row** and **Add Column** buttons once each will create a blank 3 × 3 matrix template as seen in the following example.

$$\begin{pmatrix} \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \end{pmatrix}$$

Templates for larger matrices or those with some specific characteristics, such as being filled with identical entries along the diagonal, can be created more quickly by using a special menu item. Use the **Insert** menu to select **Table/Matrix** and then choose **New**. This menu allows the user to select the number of rows and columns, whether the matrix should be filled with a particular value and whether a different value should be used for filling the diagonal. This menu makes it very easy to create something like a 10 × 10 matrix filled with the value 1 except for the diagonal, which is filled with the value 9.

$$\begin{pmatrix} 9 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 9 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 9 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 9 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 9 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 9 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 9 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 9 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 9 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 9 \end{pmatrix}$$

If you click **Insert**, select **Table/Matrix**, choose **New** and ignore the top section in that menu item, you might get a **Grid** output, which will not have the parentheses that typically surround matrices. If that happens, you can use **MatrixForm** to print the output as a matrix, or you can use the menu to retrace the steps to construct your matrix, making sure that **Matrix (List of lists)** is selected when the dialog window opens.

These same methods can be used to insert matrices into other types of cells, like text cells. This approach comes in quite handy when using Mathematica to write technical documents.

Programmatic creation of matrices can be accomplished using functions like **Table** and **Array**. For example, **Table** can be used to construct a multiplication table matrix, where the $i\,j^{\text{th}}$ entry has the value $i \times j$.

```
mat2 = Table[i*j, {i, 1, 5}, {j, 1, 5}];
MatrixForm[mat2]
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \\ 5 & 10 & 15 & 20 & 25 \end{pmatrix}$$

Commands like **RandomInteger** and **RandomReal** can produce multidimensional output, making them suitable for generating random matrices. Both commands can accept a first argument that gives the command the upper and lower bounds for the random number generation, and a second argument that specifies the dimensions of a matrix. The following command generates a 5 × 5 matrix with integer values from 0 to 10.

```
mat3 = RandomInteger[{0, 10}, {5, 5}];
MatrixForm[mat3]
```

```
⎛ 3   4  8  6   1 ⎞
⎜ 4   5  9  9   8 ⎟
⎜ 6   8  9  4  10 ⎟
⎜10   5  6  3   1 ⎟
⎝ 5   9  6  4   6 ⎠
```

There are special commands like **ConstantArray** and **DiagonalMatrix** that mirror the behavior of the **Insert ▶ Table/Matrix** menu, along with commands to construct identity matrices, matrices with special structures and matrices that correspond to geometric operations like rotation and scaling.

```
MatrixForm[ConstantArray[10, {4, 4}]]
```

```
⎛10  10  10  10⎞
⎜10  10  10  10⎟
⎜10  10  10  10⎟
⎝10  10  10  10⎠
```

```
MatrixForm[DiagonalMatrix[{1, 2, 3, 4}]]
```

```
⎛1  0  0  0⎞
⎜0  2  0  0⎟
⎜0  0  3  0⎟
⎝0  0  0  4⎠
```

Finally, matrices can be constructed by importing multidimensional data, such as data from a spreadsheet. Here, the **Import** command is used to bring in the contents of a CSV file from a URL. The CSV file contains data arranged in a tabular manner and can be visualized as a matrix.

```
mat4 = Import["http://www.handsonstart.com/RandomMatrix.csv"];
```

```
MatrixForm[mat4]
```

```
⎛42   30   3   3  45  44  78   1  16  66⎞
⎜72   54  30  94  60  75  23  92  49  41⎟
⎜69  100  34  51  56  55   7  47  30  20⎟
⎜76    9  20  13  49  78  18  68  38  54⎟
⎜71   51  37  21  68   8  91  51  22  91⎟
⎜99   22  92  78  88  66  69  61  96  96⎟
⎜52   44  18  87  42  60   7  53  93  15⎟
⎜95   79  94  71  75  93  69  95  17  84⎟
⎜11   40  40   4  15  99  79  68  89  52⎟
⎝89   29  55  95   6  33  83  77  86  18⎠
```

We will cover importing files in much greater detail in **Chapter 19: Importing and Exporting Data**.

Working with Parts of Matrices

Lists are important constructs in the Wolfram Language, and there are many commands devoted to extracting parts of lists, including extraction of individual elements. One of the most commonly used commands is **Part** or ⟦ ⟧, which is its shorthand notation. **Part[list, n]** or **list⟦n⟧** can be used to extract the n^{th} element from a list.

```
myList = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
Part[myList, 5]
myList⟦5⟧
```

```
5

5
```

The ⟦ and ⟧ symbols can be entered with Esc[[Esc and Esc]]Esc. You can use [[and]] instead. Entering the command as **Part[list, n]**, **list⟦n⟧**, or **list[[n]]** will all do the same thing. Using the special ⟦ ⟧ symbols makes it easier to distinguish between brackets that refer to extracting elements from a list and brackets that are being used to surround arguments of other functions.

Part can be used with **Span** to extract a range of values. **Span** uses **;;** as a shorthand form, so using **1;;5** will extract the first five elements of a list.

```
Part[myList, 1 ;; 5]
```

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

```
myList〚1 ;; 5〛
```

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

It is very common to need to extract the first *n* elements of a list, so there is a special command, **Take**, that does this. The arguments for **Take** are a list and an index value, and it returns the first *n* elements of the list, up to the index value.

```
Take[myList, 5]
```

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

Take can also be given a range by passing a list as its second argument. The following example takes values at positions 3 through 6.

```
Take[myList, {3, 6}]
```

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

Take can be used on matrices to extract entire columns or rows. The syntax for this form of the command is to give a row or list of rows as the second argument and a column or list of columns as the third argument. **All** can be substituted as a shorthand notation to request all rows or all columns.

The following command returns the result of taking the first two rows of a matrix and all the columns in those rows. This is equivalent to taking the first two rows of the matrix.

$$\text{Take}\left[\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix}, 2\right]$$

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

$$\textbf{Take}\left[\begin{pmatrix}\mathbf{1}&\mathbf{2}&\mathbf{3}\\\mathbf{4}&\mathbf{5}&\mathbf{6}\\\mathbf{7}&\mathbf{8}&\mathbf{9}\end{pmatrix}\textbf{, 2, All}\right]\textbf{ // MatrixForm}$$

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

The order of the arguments is reversed in the following command, which returns the result of taking all the rows for the first two columns of a matrix. This is equivalent to taking the first two columns of the matrix.

$$\textbf{Take}\left[\begin{pmatrix}\mathbf{1}&\mathbf{2}&\mathbf{3}\\\mathbf{4}&\mathbf{5}&\mathbf{6}\\\mathbf{7}&\mathbf{8}&\mathbf{9}\end{pmatrix}\textbf{, All, 2}\right]\textbf{ // MatrixForm}$$

$$\begin{pmatrix}1&2\\4&5\\7&8\end{pmatrix}$$

You can actually get away with not using the **All** for the column specification with **Take**. You can use both **Take[matrix, 2]** and **Take[matrix, 2, All]** to accomplish the same thing, but using both parameters can remove some ambiguity for readers of your code, and it is the approach that we authors recommend.

Some additional commands useful for working with matrices are **Diagonal**, which returns a list of elements along the diagonal, and **Transpose**, which interchanges rows and columns.

$$\textbf{Diagonal}\left[\begin{pmatrix}\mathbf{1}&\mathbf{2}&\mathbf{3}\\\mathbf{4}&\mathbf{5}&\mathbf{6}\\\mathbf{7}&\mathbf{8}&\mathbf{9}\end{pmatrix}\right]\textbf{ // MatrixForm}$$

$$\begin{pmatrix}1\\5\\9\end{pmatrix}$$

$$\textbf{Transpose}\left[\begin{pmatrix}\mathbf{1}&\mathbf{2}&\mathbf{3}\\\mathbf{4}&\mathbf{5}&\mathbf{6}\end{pmatrix}\right]\textbf{ // MatrixForm}$$

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

The **Transpose** command uses ᵀ as a shorthand form. The shorthand form can be entered with the escape sequence Esc tr Esc.

```
(1 2 3)ᵀ // MatrixForm
(4 5 6)
```

```
(1 4)
(2 5)
(3 6)
```

When working with matrices, it can be a common practice to use the **Postfix** operator // with **MatrixForm** to print results in a two-dimensional layout. Care must be taken, however, when assigning matrices to variables, since postfix operations will be applied *before* the variable assignment is made, and this can cause problems; other commands will expect nested lists to be given as arguments, and arguments wrapped in display forms like **MatrixForm** will not be accepted.

The moral of this story is: Keep form-printing functions, like **MatrixForm** and **Table**∵. **Form**, away from variable assignments. Assign the value to the variable, and then apply the printing function to the variable on a separate line or in a separate input cell.

```
mat3 = (1 2 3)ᵀ;
       (4 5 6)
MatrixForm[mat3]
```

```
(1 4)
(2 5)
(3 6)
```

Matrix Operations

The Wolfram Language has commands for standard linear algebra operations like inverses, row reduction, determinants, traces and eigenvalues. **Inverse** takes a matrix as its argument and returns its inverse.

$$\textbf{Inverse}\left[\begin{pmatrix} 9 & 4 & 3 & 2 \\ 4 & 6 & 7 & 0 \\ 3 & 0 & 10 & 0 \\ 2 & 6 & 2 & 6 \end{pmatrix}\right]$$

$$\left\{\left\{\frac{1}{7}, -\frac{1}{21}, 0, -\frac{1}{21}\right\}, \left\{-\frac{19}{420}, \frac{229}{1260}, -\frac{7}{60}, \frac{19}{1260}\right\},\right.$$
$$\left.\left\{-\frac{3}{70}, \frac{1}{70}, \frac{1}{10}, \frac{1}{70}\right\}, \left\{\frac{1}{84}, -\frac{43}{252}, \frac{1}{12}, \frac{41}{252}\right\}\right\}$$

Inverse may give exact output, so **N** can be used to get a numerical approximation. The following example gets a five-digit approximation for an inverse and then uses **MatrixForm** to format the output as a matrix.

$$\textbf{N}\left[\textbf{Inverse}\left[\begin{pmatrix} 9 & 4 & 3 & 2 \\ 4 & 6 & 7 & 0 \\ 3 & 0 & 10 & 0 \\ 2 & 6 & 2 & 6 \end{pmatrix}\right], 5\right] \textbf{// MatrixForm}$$

$$\begin{pmatrix} 0.14286 & -0.047619 & 0 & -0.047619 \\ -0.045238 & 0.18175 & -0.11667 & 0.015079 \\ -0.042857 & 0.014286 & 0.10000 & 0.014286 \\ 0.011905 & -0.17063 & 0.083333 & 0.16270 \end{pmatrix}$$

RowReduce can be used to transform matrices into reduced row echelon form.

$$\textbf{RowReduce}\left[\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}\right] \textbf{// MatrixForm}$$

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

Row reduction is sometimes used to find a solution to a matrix equation, but there is also a built-in command, **LinearSolve**, that can be used for this. **LinearSolve** takes components m and b and returns x such that the equation $m.x = b$ holds true. First, variable assignments are made to store a matrix and a vector.

$$\mathbf{m} = \begin{pmatrix} 1 & 9 & 8 \\ 1 & 2 & 7 \\ 3 & 8 & 4 \end{pmatrix};$$

```
b = {10, 15, 20};
```

LinearSolve is used to find a solution to the equation. The following example stores the solution in a new variable, **x**.

```
x = LinearSolve[m, b]
```

$$\left\{\frac{80}{11}, -\frac{10}{11}, \frac{15}{11}\right\}$$

The solution can be verified with substitution. The dot product of **m** and **x** should equal **b**, and it does.

```
m.x == b
```

```
True
```

> Could **Solve** be used instead of **LinearSolve**? The answer is yes, but **Solve** requires the vector to be given explicitly, which is a little clunky. However, with the variables for **m**, **b** and **x** defined as in the previous examples, you can use the **Solve** command in the form of **Solve[m.{x1,x2,x3}==b,{x1,x2,x3}]** to get the same result as you would from using **LinearSolve**.

LinearSolve can also be used when the right-hand side of the equation is a matrix instead of a vector.

$$\mathbf{m} = \begin{pmatrix} 1 & 9 & 8 \\ 1 & 2 & 7 \\ 3 & 8 & 4 \end{pmatrix};$$

$$\mathbf{b} = \begin{pmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{pmatrix};$$

```
x = LinearSolve[m, b];
MatrixForm[x]
```

$$\begin{pmatrix} -\frac{82}{121} & -\frac{109}{121} \\ \frac{24}{121} & \frac{26}{121} \\ \frac{74}{121} & \frac{60}{121} \end{pmatrix}$$

The Wolfram Language commands **Eigenvalues**, **Eigenvectors** and **Eigensystem** make it easy to work with eigenvalues.

$$\mathbf{Eigenvalues}\left[\begin{pmatrix}\mathbf{1} & \mathbf{2}\\ \mathbf{3} & \mathbf{4}\end{pmatrix}\right]$$

$$\left\{\frac{1}{2}\left(5+\sqrt{33}\right),\ \frac{1}{2}\left(5-\sqrt{33}\right)\right\}$$

$$\mathbf{Eigenvectors}\left[\begin{pmatrix}\mathbf{1} & \mathbf{2}\\ \mathbf{3} & \mathbf{4}\end{pmatrix}\right]$$

$$\left\{\left\{\frac{1}{6}\left(-3+\sqrt{33}\right),\ 1\right\},\ \left\{\frac{1}{6}\left(-3-\sqrt{33}\right),\ 1\right\}\right\}$$

$$\mathbf{Eigensystem}\left[\begin{pmatrix}\mathbf{1} & \mathbf{2}\\ \mathbf{3} & \mathbf{4}\end{pmatrix}\right]$$

$$\left\{\left\{\frac{1}{2}\left(5+\sqrt{33}\right),\ \frac{1}{2}\left(5-\sqrt{33}\right)\right\},\ \left\{\left\{\frac{1}{6}\left(-3+\sqrt{33}\right),\ 1\right\},\ \left\{\frac{1}{6}\left(-3-\sqrt{33}\right),\ 1\right\}\right\}\right\}$$

Mathematica can find exact eigenvalues, and depending on the input, it may output **Root** objects, which are exact representations of roots of equations. These may look strange at first, but they can be converted to numerical approximations with the **N** command.

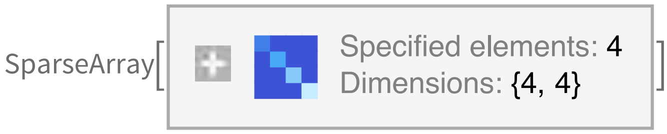
Sparse Arrays

Mathematica supports linear algebra on sparse arrays, allowing computations on matrices of incredible dimensions to be performed when only a fraction of their elements are nonzero. Sparse arrays are represented as special objects, which print as **SparseArray** when they are returned.

SparseArray and **InterpolatingFunction** objects are returned in a similar fashion. (More about the latter can be found in **Chapter 16: Differential Equations**.)

The **SparseArray** command is used to create a sparse array. **SparseArray** is given a list of rules containing positions and values, which are then used to construct the sparse array object. The following command creates a sparse array where position (1, 1) has value 1, position (2, 2) has value 2, position (3, 3) has value 3 and position (4, 4) has value 4. The other elements are not specified, so Mathematica assumes they have a value of 0.

```
sa = SparseArray[{{1, 1} → 1, {2, 2} → 2, {3, 3} → 3, {4, 4} → 4}]
```



```
SparseArray[ Specified elements: 4
             Dimensions: {4, 4} ]
```

The **SparseArray** object was stored in the variable **sa** and can now be used like a regular matrix.

```
MatrixForm[sa]
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

```
Dimensions[sa]
```

```
{4, 4}
```

```
Det[sa]
```

```
24
```

```
b = {40, 30, 20, 10};
LinearSolve[sa, b]
```

$$\left\{40, 15, \frac{20}{3}, \frac{5}{2}\right\}$$

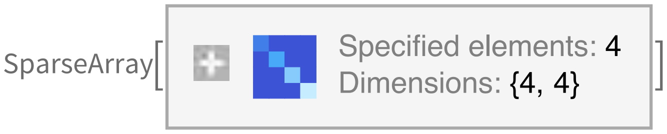
A sparse array can be converted into a regular (dense) matrix by using the **Normal** command.

```
Normal[sa] // MatrixForm
```

$$\begin{pmatrix}1&0&0&0\\0&2&0&0\\0&0&3&0\\0&0&0&4\end{pmatrix}$$

And a dense matrix can be passed to **SparseArray** to create a sparse array representation of the matrix.

$$\mathbf{SparseArray}\left[\begin{pmatrix}\mathbf{1}&\mathbf{0}&\mathbf{0}&\mathbf{0}\\\mathbf{0}&\mathbf{2}&\mathbf{0}&\mathbf{0}\\\mathbf{0}&\mathbf{0}&\mathbf{3}&\mathbf{0}\\\mathbf{0}&\mathbf{0}&\mathbf{0}&\mathbf{4}\end{pmatrix}\right]$$



```
SparseArray[ Specified elements: 4
             Dimensions: {4, 4} ]
```

Sparse arrays are useful when working with large datasets for which only certain values are nonzero. Sparse matrices provide a more efficient way to represent such data than standard dense matrices do. This can be illustrated with an example that compares the computation time needed to solve a linear system with a dense matrix to the computation time needed to solve the same linear system with a sparse array.

The following **Table** command constructs a 10,000 × 10,000 matrix where most of the elements are 0, but the diagonal has values of 9 and the diagonals that flank it have values of 1. This is done by comparing two indices, i and j. If $i = j$, which happens for the diagonal elements, then a 9 is placed in that position. If $|i - j| = 1$, which happens for the flanking diagonals, then a 1 is placed in that position. Otherwise, a 0 is placed in the position. The matrix is stored as the variable **mNew**.

```
mNew = Table[If[i == j, 9, If[Abs[i – j] == 1, 1, 0]], {i, 1, 10 000}, {j, 1, 10 000}];
```

Next, a vector of 10,000 random reals from 0 to 10 is constructed and stored in the variable **bNew**.

```
SeedRandom["CKM"];
bNew = RandomReal[{0, 10}, {10 000}];
```

You may wonder what the **SeedRandom** command is doing. **SeedRandom** is used to reset the random number generator, but seeding a particular value will mean that the random numbers generated by a function will be repeatable for other users who use the same **SeedRandom** command. If you evaluate the input cell above, you will get the same set of pseudorandom numbers that the authors did when writing this book.

The **LinearSolve** command is used to find a vector **xNew**, such that **mNew.xNew=bNew**. Since the output of this command is a vector with 10,000 elements, the **;** operator is used to suppress the output. Finally, the **AbsoluteTiming** command is wrapped around the command to show the amount of computation time for the evaluation.

```
AbsoluteTiming[LinearSolve[mNew, bNew];]
```

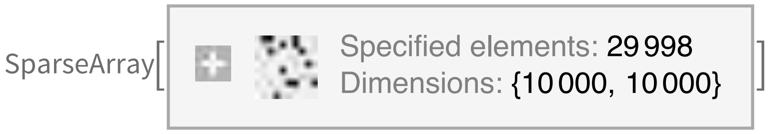
```
{9.71387, Null}
```

The standard matrix representation took 9.71 seconds of computation time.

The **AbsoluteTiming** will be dependent on your computer system, so do not be surprised if you get a result that is different than what is shown in the preceding example.

The same example is now explored with a sparse array representation. First, a sparse array is constructed by passing the dense matrix **mNew** to the **SparseArray** command.

```
saNew = SparseArray[mNew]
```



```
SparseArray[ Specified elements: 29998
             Dimensions: {10000, 10000} ]
```

The vector **bNew** is already constructed, so there is no need to recreate it. The last step is to solve the matrix equation and record how much time it takes.

```
AbsoluteTiming[LinearSolve[saNew, bNew];]
```

```
{0.00522, Null}
```

This took much less time than solving the matrix when using the dense array representation. This example used a relatively small matrix (10,000 × 10,000) but effectively illustrates the differences in speed when working with dense matrices and sparse arrays.

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[vec1, vec2, vec3, vec4, myFunction, mat1, mat2, mat3, mat4, myList, m, b,
  x, sa, mNew, bNew]
```

Conclusion

Mathematica has robust support for linear algebra, from working with exact matrices to optimized functionality for numerical linear algebra. The list structure of matrices allows list manipulation commands to be applied for easy extraction of elements, rows, columns and submatrices. Using the commands and techniques outlined in this chapter should allow users to successfully start doing linear algebra in Mathematica.

Exercises

1. Use free-form input to add the vectors $\{1, 3, 5, \pi, \sin(2)\}$ and $\{2, 4, 6, 3\,\pi, x\}$.
2. Use the Wolfram Language to create a variable named **vector1** and assign to it the output from Exercise 1. Then create a second variable named **vector2** and assign to it a list of values of the form $i^3 + 1$, where i goes from 0 to 4.
3. Create a two-line program that calculates the multiplication and dot product of **vector1** and **vector2**.
4. Define a new variable **vector3**, which is defined as the second element of **vector2**, and also define a new variable **vector4**, which is defined to be the first three elements of **vector1**.
5. Find the numerical approximation of the norm of **vector4** to four digits.
6. Create a variable named **matrix1** and assign to it a list of values defined by $i^2 - j$, and create a variable named **matrix2** and assign to it a list of values defined by $3\,i - j^2$, where i and j both go from 1 to 3.

7. Calculate the determinant of **matrix1**.
8. Calculate the transpose of **matrix1** and format the result to resemble the mathematical typesetting typically found in textbooks.
9. Calculate the dot product of **matrix1** and **matrix2**.
10. Find the value of x that satisfies the matrix equation $m.x = b$, where m is **matrix2** and b is **vector4**.

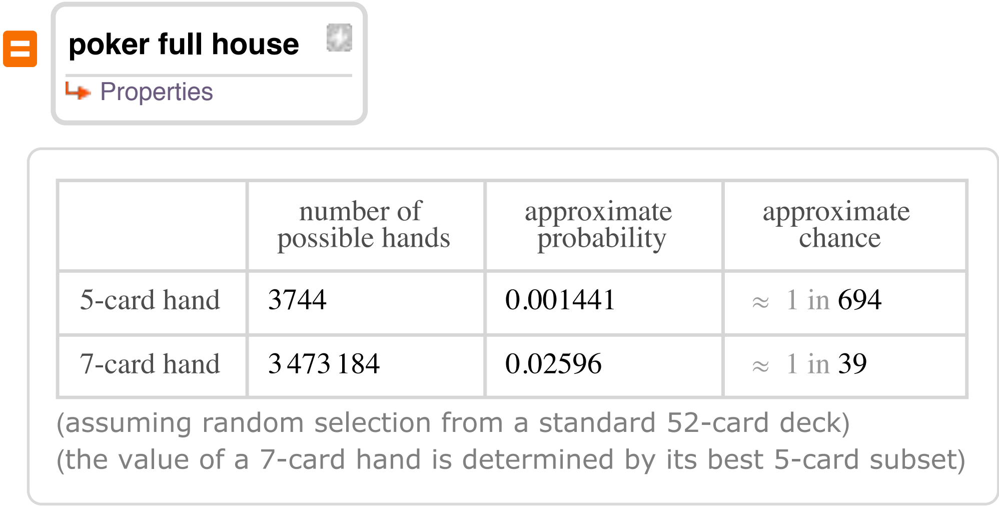
CHAPTER 18
Probability and Statistics

Introduction

Mathematica is an extremely powerful tool for working with probability and statistics calculations. Unlike specialized statistical software that is designed to perform a narrow set of tasks, Mathematica leverages its other capabilities, including exact and numerical computation, visualization, and an extensive list of built-in distributions, to provide a rich environment for all kinds of statistics work.

Probability and Distributions

Free-form input can be used to calculate the probability of many different types of events.



poker full house

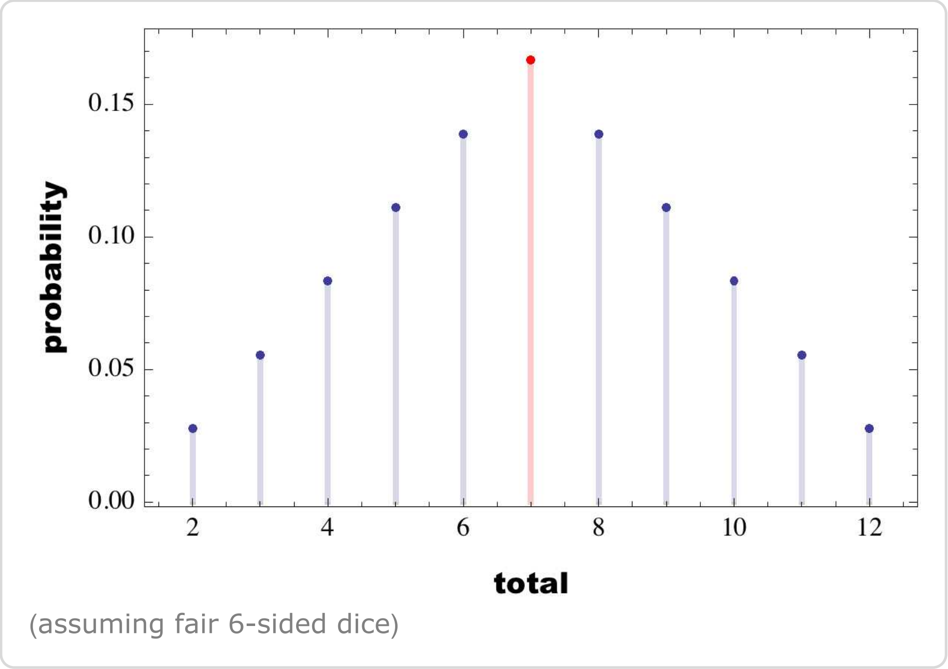
↳ Properties

	number of possible hands	approximate probability	approximate chance
5-card hand	3744	0.001441	≈ 1 in 694
7-card hand	3 473 184	0.02596	≈ 1 in 39

(assuming random selection from a standard 52-card deck)
(the value of a 7-card hand is determined by its best 5-card subset)

rolling a 7 on two 6-sided dice

Distribution of total



(assuming fair 6-sided dice)

For situations that concern behavior governed by particular distributions, the Wolfram Language command **Probability** can be used to calculate the probability of that event. For example, the probability of rolling a three on a fair six-sided die can be described by a discrete uniform distribution, and the **Probability** command can be used to determine the likelihood of this event. The first argument is the expression that describes the event—in this case, **x==3**—and the second argument is the distribution from which *x* is sampled.

```
Probability[x == 3, x ≈ DiscreteUniformDistribution[{1, 6}]]
```

$$\frac{1}{6}$$

The distribution symbol ≈ can be entered as Esc dist Esc. The symbolic form is **Distributed**, so **x ≈ distribution** is equivalent to **Distributed[x, distribution]**.

The preceding example can be read almost as how the problem would be phrased in a textbook: What is the probability that $x = 3$, if x is sampled from a discrete uniform distribution over the integers from 1 to 6?

Probability can be used for compound statements as well, like the probability of rolling a 12 using two fair six-sided dice. In that case, two instances of the **DiscreteUniformDistribution** command are used, with each representing the possible outcomes for each die, and the **&&** operator (shorthand form of **And**) is used to join those two events together.

```
Probability[x + y == 12, x ≈ DiscreteUniformDistribution[{1, 6}] &&
    y ≈ DiscreteUniformDistribution[{1, 6}]]
```

$$\frac{1}{36}$$

Probability can be used with other Wolfram Language commands. For example, **Table** might be used to generate a list of all of the probabilities for the outcomes of rolling two dice. Since there are 11 possible results, from 2 to 12, those become the bounds for the iterator in the **Table** command.

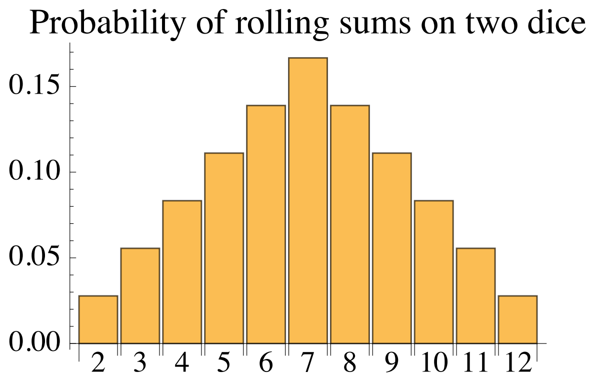
```
probs = Table[
    Probability[x + y == result, x ≈ DiscreteUniformDistribution[{1, 6}] &&
        y ≈ DiscreteUniformDistribution[{1, 6}]],
    {result, 2, 12, 1}]
```

$$\left\{\frac{1}{36}, \frac{1}{18}, \frac{1}{12}, \frac{1}{9}, \frac{5}{36}, \frac{1}{6}, \frac{5}{36}, \frac{1}{9}, \frac{1}{12}, \frac{1}{18}, \frac{1}{36}\right\}$$

The symbol **&&** is shorthand for the **And** function. The statements **expr1 && expr2** and **And[expr1, expr2]** are identical, so use whichever one you like. When joining just two expressions together, the **expr1 && expr2** form is handy, but for long expressions, using the formal command name with the commas separating the arguments might be easier to mentally parse.

It might be useful to visualize this result by using **BarChart**. A **Table** statement is used to generate a list of the results (a roll of 2, 3, 4, ..., 12) for the **ChartLabels** option, and a **PlotLabel** is also used.

```
BarChart[probs, ChartLabels → Table[i, {i, 2, 12, 1}],
  PlotLabel → "Probability of rolling sums on two dice"]
```



Working with Distributions

The preceding examples introduced the **Probability** command and used the **DiscreteUniformDistribution** command to describe a certain type of event. The Wolfram Language gives Mathematica users an incredible number of distributions for working with and modeling all sorts of behaviors. These distributions are univariate, multivariate, continuous and discrete, and there are also specialized distributions for areas like finance.

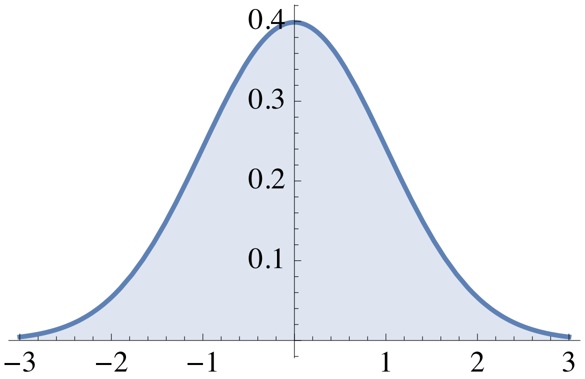
Distributions can be sampled from, as shown earlier when the **Probability** command was used to sample from a discrete uniform distribution to model outcomes from rolling dice. Properties for distributions, like probability density functions (**PDF**), cumulative distribution functions (**CDF**), measures and moments can also be computed. The following example shows the probability density function for the discrete uniform distribution that was used earlier.

```
PDF[DiscreteUniformDistribution[{1, 6}], x]
```

$$\begin{cases} \frac{1}{6} & 1 \le x \le 6 \\ 0 & \text{True} \end{cases}$$

The probability density function can be computed for any distribution in the Wolfram Language, such as the normal distribution with mean 0 and standard deviation 1.

```
Plot[PDF[NormalDistribution[0, 1], x], {x, −3, 3}, Filling → Axis]
```



The output from the **PDF** command can even be used as part of the definition of a user-defined function. The following example creates a function, **myFun**, based on the probability density function of the normal distribution with mean 0 and standard deviation 1. Then **myFun** is used to compute values of the probability density function at certain points.

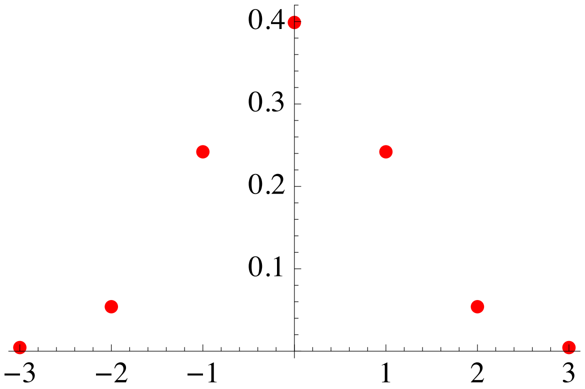
```
myFun[x_] := PDF[NormalDistribution[0, 1], x]
```

```
myFun[0]
```

$$\frac{1}{\sqrt{2\pi}}$$

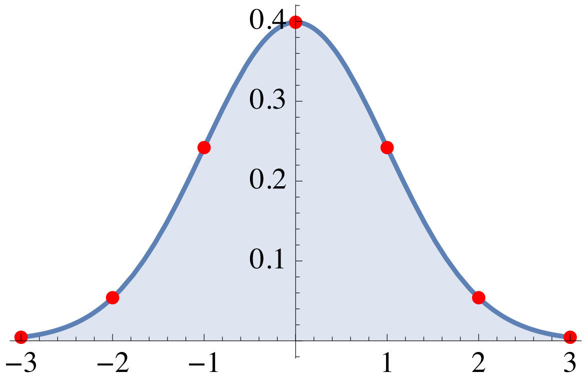
Next **myFun** is used with **Table** to create a list of points.

```
points = Table[{x, myFun[x]}, {x, −3, 3, 1}];
ListPlot[points, PlotStyle → {Red, PointSize[Medium]}]
```



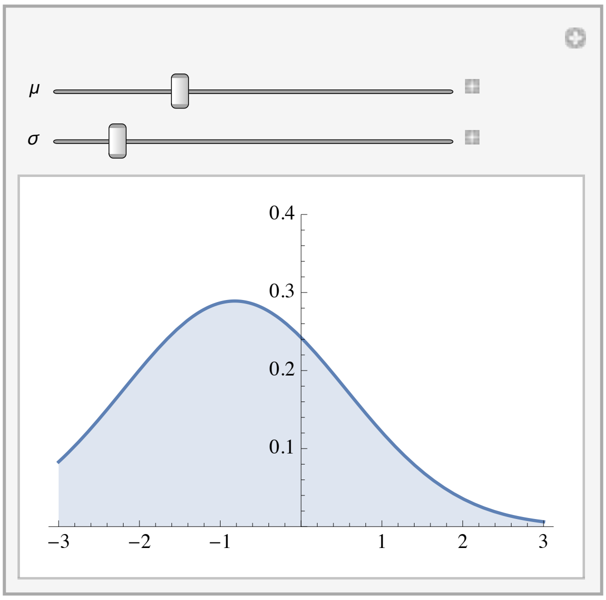
Now the probability density function can be plotted using **myFun** and placed on the same set of axes as the data points by using the **Show** command.

```
Show[
  Plot[myFun[x], {x, −3, 3}, Filling → Axis],
  ListPlot[points, PlotStyle → {Red, PointSize[Medium]}]]
```



If you read the chapter on creating diagrams with graphics primitives, then you know that graphics are generated in order: the first one goes on the bottom, the next one is stacked on top of that and so on. The **Show** command does something similar. In this example, the **ListPlot** is passed as the second argument so that the points appear on top of the curve. If you reverse the order of the **Plot** and **ListPlot** commands when passing these to the **Show** command, you will still see the points, but they will appear underneath (or behind) the curve.

Examples like this also lend themselves to being used with **Manipulate**. The following command creates an interactive model that can be used to explore the effect of changing the mean and standard deviation parameters for a normal distribution.



Statistics

The Wolfram Language has comprehensive coverage for statistics operations, with deep functionality for sophisticated and advanced users. This section will explore some of the commands for calculating basic measures, curve fitting and visualizing statistical information.

Descriptive Statistics

Mathematica can calculate all types of measures of data, including means and medians; variance, standard deviation and interquartile ranges; skewness and kurtosis; and covariance and correlation for multi-datasets. Like other Wolfram Language commands, the statistics commands follow the same literate-naming style, making it easy to guess a command name even if it has not been used before.

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

```
Mean[data]
```

$$\frac{11}{3}$$

```
Median[data]
```

```
4
```

```
Commonest[data]
```

```
{5}
```

> I know what you are thinking—no **Mode** command? A rare sign of deviation (pardon the pun!) from Wolfram Language naming conventions is the command name for finding the mode, or most common value, in a set of data. The command name for this operation in the Wolfram Language is **Commonest**, since **Mode** is a reserved symbol used by the system for other purposes. However, you can define your own function named **mode** with the definition **mode[list_]:=Commonest[list]** if you really want to use that particular name in your programs.

These commands work on arbitrary data, such as datasets containing symbolic values.

```
Mean[{p1, p2, p3, p4, p5}]
```

$$\frac{1}{5}(\text{p1} + \text{p2} + \text{p3} + \text{p4} + \text{p5})$$

$$\text{Commonest}\left[\left\{x, x^2, x^2, x^3, x^3, x^3\right\}\right]$$

$$\{x^3\}$$

There are commands for other measures, like **Variance**, **StandardDeviation**, **InterquartileRange**, **Covariance** and **Correlation**.

```
data = {1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5};
Variance[data]
```

$$\frac{5}{3}$$

```
StandardDeviation[data]
```

$$\sqrt{\frac{5}{3}}$$

```
InterquartileRange[data]
```

```
2
```

Given two lists, **Covariance** can be used to find the covariance coefficient, and **Correlation** can be used to find the correlation coefficient.

```
list1 = {1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5};
list2 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 8, 7, 6, 5, 4, 3};
```

```
Covariance[list1, list2]
```

$$\frac{23}{14}$$

```
Correlation[list1, list2]
```

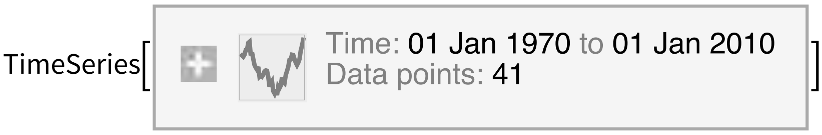
$$\frac{23\sqrt{\frac{3}{7}}}{28}$$

Curve Fitting

The Wolfram Language has several commands for curve fitting and creating linear regression models. Some of the commands are only focused on finding a fitted curve, but other commands, like **LinearModelFit** and **NonlinearModelFit**, construct models. These models can be used to find fits, and they also provide a framework for users to look up other properties of the fitted model. It is easiest to appreciate the power of these commands by seeing them in action.

First, a dataset is created. Here the **CountryData** command is used to import data related to Iceland's GDP from 1970 to 2010.

```
myData = CountryData["Iceland", {{"GDP"}, {1970, 2010}}]
```



```
TimeSeries[ Time: 01 Jan 1970 to 01 Jan 2010
            Data points: 41 ]
```

The data is returned as a **TimeSeries** object. The temporal nature of the dataset is not of interest in this example, so the values can be extracted from the **TimeSeries** object by using the **"Values"** property.

```
myData["Values"]
```

```
{ $5.31005×10^8 per year , $6.75723×10^8 per year ,
   $8.46507×10^8 per year , $1.16386×10^9 per year , $1.52756×10^9 per year ,
   $1.41836×10^9 per year , $1.68312×10^9 per year , $2.22654×10^9 per year ,
   $2.53233×10^9 per year , $2.87673×10^9 per year , $3.40902×10^9 per year ,
   $3.52151×10^9 per year , $3.2328×10^9 per year , $2.78853×10^9 per year ,
   $2.88783×10^9 per year , $3.00841×10^9 per year , $4.02219×10^9 per year ,
   $5.56538×10^9 per year , $6.15649×10^9 per year , $5.71888×10^9 per year ,
   $6.52154×10^9 per year , $6.96614×10^9 per year , $7.13879×10^9 per year ,
   $6.26935×10^9 per year , $6.44162×10^9 per year , $7.18179×10^9 per year ,
   $7.50195×10^9 per year , $7.59613×10^9 per year , $8.46834×10^9 per year ,
   $8.92819×10^9 per year , $8.92474×10^9 per year , $8.12616×10^9 per year ,
   $9.1618×10^9 per year , $1.1297×10^10 per year , $1.37044×10^10 per year ,
   $1.67493×10^10 per year , $1.70413×10^10 per year , $2.12938×10^10 per year ,
   $1.75307×10^10 per year , $1.28553×10^10 per year , $1.32369×10^10 per year }
```

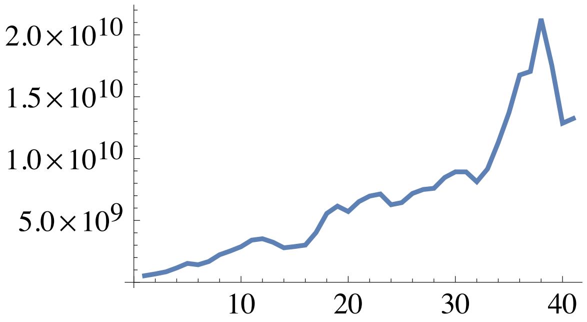
The values are given in unitized form, and only the magnitude is of interest, so the **QuantityMagnitude** function is used to discard the units and return the values.

```
QuantityMagnitude[myData["Values"]]
```

```
{5.31005×10^8, 6.75723×10^8, 8.46507×10^8, 1.16386×10^9, 1.52756×10^9, 1.41836×10^9,
  1.68312×10^9, 2.22654×10^9, 2.53233×10^9, 2.87673×10^9, 3.40902×10^9,
  3.52151×10^9, 3.2328×10^9, 2.78853×10^9, 2.88783×10^9, 3.00841×10^9,
  4.02219×10^9, 5.56538×10^9, 6.15649×10^9, 5.71888×10^9, 6.52154×10^9,
  6.96614×10^9, 7.13879×10^9, 6.26935×10^9, 6.44162×10^9, 7.18179×10^9,
  7.50195×10^9, 7.59613×10^9, 8.46834×10^9, 8.92819×10^9, 8.92474×10^9,
  8.12616×10^9, 9.1618×10^9, 1.1297×10^10, 1.37044×10^10, 1.67493×10^10,
  1.70413×10^10, 2.12938×10^10, 1.75307×10^10, 1.28553×10^10, 1.32369×10^10}
```

The result of this operation is stored back into the **myData** variable and can be visualized with commands like **ListLinePlot**.

```
myData = QuantityMagnitude[myData["Values"]];
ListLinePlot[myData]
```



You could use **ListPlot** instead. Remember, **ListPlot** will plot a dataset as individual points, and **ListLinePlot** will plot the data with a single line that connects all the points.

The **LinearModelFit** command can be used to fit a model to this data. For its arguments, **LinearModelFit** requires a dataset, a list of some functions that will be used to construct a fit and the variable of interest. In this case, an eighth-degree polynomial is tried, so a list of the form $\{1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8\}$ is passed as the second argument. The output from **LinearModelFit** is stored in a variable named **myModel**.

```
myModel = LinearModelFit[myData, {1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8}, x]
```

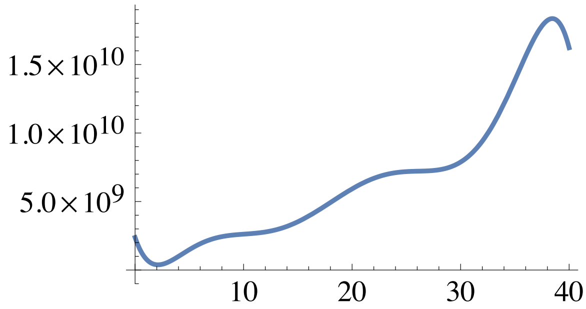
```
FittedModel[ ≪12≫ + 0.316809 x^8 ]
```

myModel works like any other univariate function, so it can be evaluated at certain values or plotted with the **Plot** command.

```
myModel[1]
```

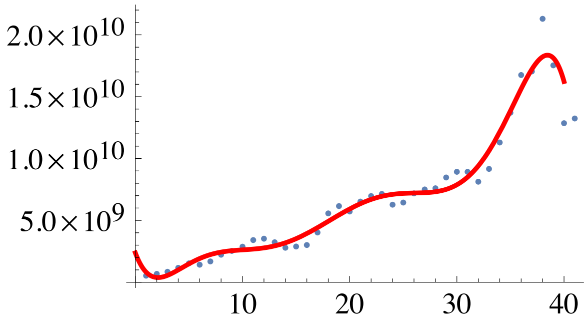
```
8.23834×10^8
```

```
Plot[myModel[x], {x, 0, 40}]
```



The **Show** command can be used to superimpose the **ListPlot** of the original dataset with the **Plot** command to visualize the fitted curve.

```
Show[
  ListPlot[myData],
  Plot[myModel[x], {x, 0, 40}, PlotStyle → Red]
]
```



Other properties for the linear model can be computed, like the expression for the curve of best fit. These properties are accessed by treating **myModel** like a function, but instead of passing it a numerical value to evaluate, a property value is passed as a string. For example, the **"BestFit"** property value will return the expression for the curve of best fit.

```
TraditionalForm[myModel["BestFit"]]
```

$$0.316809\, x^8 - 98.8321\, x^7 + 9684.09\, x^6 - 444163.\, x^5 + 1.07227\times 10^7\, x^4 - 1.38376\times 10^8\, x^3 + 9.03512\times 10^8\, x^2 - 2.32711\times 10^9\, x + 2.37552\times 10^9$$

Many different properties are available. A full list of properties can be listed by passing **"Properties"** as a property value, but for the sake of brevity, a subset of the properties is shown by evaluating the following command.

```
Short[myModel["Properties"], 10]
```

```
{AdjustedRSquared, AIC, AICc, ANOVATable, ANOVATableDegreesOfFreedom,
  ANOVATableEntries, ≪52≫, SinglePredictionConfidenceIntervalTable,
  SinglePredictionConfidenceIntervalTableEntries, SinglePredictionErrors,
  StandardizedResiduals, StudentizedResiduals, VarianceInflationFactors}
```

Any of these properties can be computed immediately by passing the value to **myModel**. For example, the following command computes the adjusted r^2 value for the model.

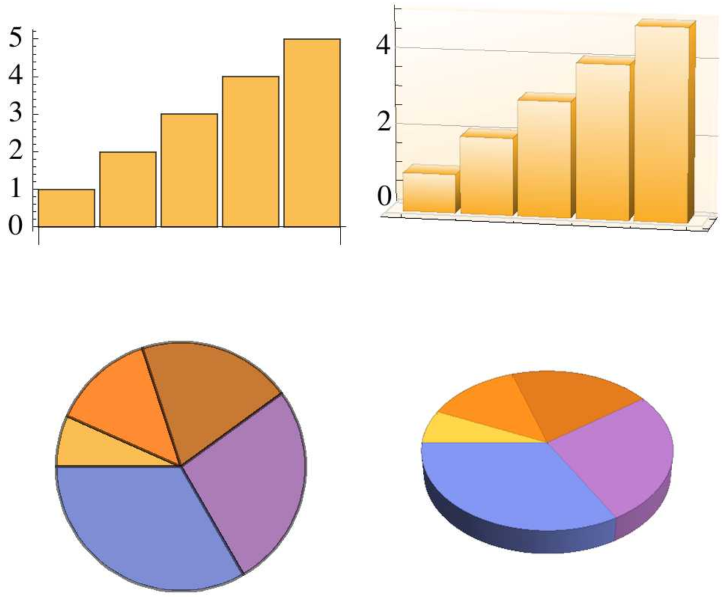
```
myModel["AdjustedRSquared"]
```

```
0.955542
```

Statistics Visualization

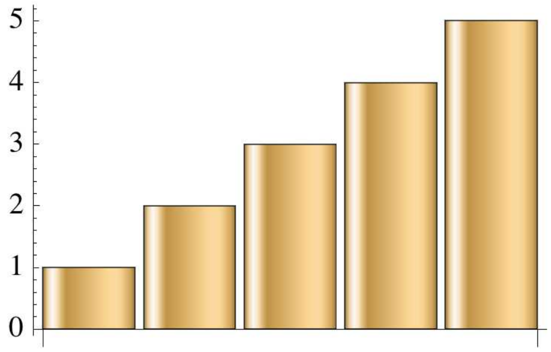
Data visualization has already been discussed in some detail in a previous chapter, but it is worth pointing out that specialized commands are available for statistical visualization. There is a host of charting commands available to visualize data in both 2D and 3D representations.

```
data = {1, 2, 3, 4, 5};
GraphicsGrid[{
    {BarChart[data], BarChart3D[data]},
    {PieChart[data], PieChart3D[data]}
  }]
```



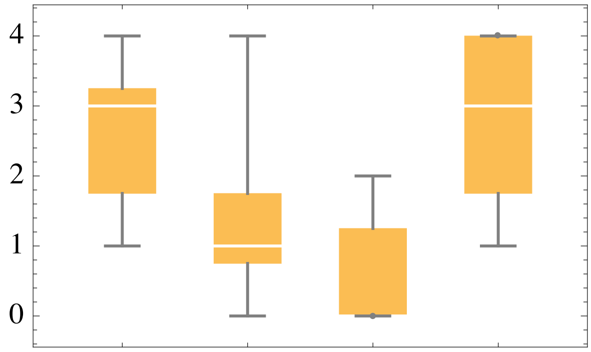
There is also a special palette, found by clicking the **Palettes** menu and choosing **Chart Element Schemes**, that provides an easy way to customize the style of a chart.

```
BarChart[data, ChartElementFunction → "GlassRectangle"]
```



Other useful statistical visualization commands include **PairedBarChart**, **Histogram** and **BoxWhiskerChart**. Charts include interactive elements, like tooltips, which give more information when the pointer hovers over chart elements.

```
boxData = RandomInteger[{0, 4}, {4, 5}];
BoxWhiskerChart[boxData]
```



Clear is used to remove all variable and function definitions from this chapter.

```
Clear[probs, myFun, points, data, list1, list2, myData, myModel, boxData]
```

Conclusion

Mathematica can be used for a wide range of statistical work, from analysis of datasets to probability and expectation computation of distributions. The statistical functionality is quite extensive, so this chapter is only a quick introduction to help users get familiar with some of the most common statistics commands. Other resources, like the documentation, the Wolfram Demonstrations Project and Wolfram Training courses, are recommended for those who want to gain a deeper understanding of using Mathematica for statistics work.

Exercises

1. Use free-form input to calculate the average of the following test scores: 93, 86, 68, 94, 91, 88 and 74.
2. Use the Wolfram Language to numerically approximate the result from Exercise 1 to two digits.
3. Use the Wolfram Language to create a two-statement program, where the variable **steps1** is defined as a list of points of the form $i - 1$, where i goes from 0 to 5, and where the output of defining **steps1** is suppressed. For the second statement, find the numeric approximation of the median of the list of points.
4. Use the Wolfram Language to find the numeric approximation of the probability of rolling three fair six-sided dice and receiving a sum of 11 or 12.
5. The command **BinomialDistribution** can be used for events where there are a certain number of trials n and a specific success probability p. Use this command to find the probability that a basketball player who makes 80% of his free throws makes three out of his four free throws during a game.
6. Use free-form input to find Larry Bird's free throw percentage in his 1985–86 NBA season. Then use the Wolfram Language to compute the probability of Bird making three or four free throws (of an assumed four attempts) in a game during that season.
7. Create a three-statement program where the first statement defines the variable **labresults1** to be a table of pairs of values of the form $(2\,i - 1, i^2 - 5\,i + 1)$, where i goes from 1 to 25. The second statement should find a linear fit of the data and store the result as **linearfit1**. The third statement should find a quadratic fit of the data and store the result as **quadfit1**.
8. Create a three-statement program where a list plot of **labresults1** is stored as **vislabresults1**, a plot of **linearfit1** from 0 to 50 is stored as **vislinearfit1** and a plot of **quadfit1** from 0 to 50 is stored as **visquadfit1**.
9. Use the Wolfram Language to display the plots for Exercise 8 on a single set of axes.
10. Create a two-statement program that creates the variable **rolls** and assigns to it the outcome of 100 random choices from the list {1, 2, 3, 4, 5, 6}, which are the outcomes of rolling a common, fair, six-sided die. The second statement in the program should create a histogram of the results. (Note: since a "random" command is being used, the results will be different each time the program is run.)

CHAPTER 19

Importing and Exporting Data

Introduction

Previous chapters have addressed Mathematica's representation of lists and how they can be used with plotting or charting functions to visualize data. While these examples have used functions like **Table** to generate data to illustrate the functionality of particular commands, in real-world use, it is very common to import data with which to work. Mathematica supports import and export of many file formats, so regardless of where the data comes from—a spreadsheet application, a digital camera, an audio capture card or specialized hardware—Mathematica is likely to support the file format. Having a single platform like Mathematica for data analysis and visualization is very useful, but this utility is further increased by Mathematica's data processing capabilities, which allow it to combine and deconstruct separate datasets for more intricate analyses.

Importing an External File

Mathematica supports many different file types for importing, and evaluating the following symbol will show a list of the supported file formats.

```
$ImportFormats
```

```
{3DS, ACO, Affymetrix, AgilentMicroarray, AIFF, ApacheLog, ArcGRID, AU, AVI,
  Base64, BDF, Binary, Bit, BMP, Byte, BYU, BZIP2, CDED, CDF, Character16,
  Character8, CIF, Complex128, Complex256, Complex64, CSV, CUR, DAE, DBF,
  DICOM, DIF, DIMACS, Directory, DOT, DXF, EDF, EML, EPS, ExpressionJSON,
  ExpressionML, FASTA, FASTQ, FCS, FITS, FLAC, GenBank, GeoTIFF, GIF, GPX,
  Graph6, Graphlet, GraphML, GRIB, GTOPO30, GXL, GZIP, HarwellBoeing,
  HDF, HDF5, HIN, HTML, ICC, ICNS, ICO, ICS, Integer128, Integer16, Integer24,
  Integer32, Integer64, Integer8, JCAMP-DX, JPEG, JPEG2000, JSON, JVX, KML,
  LaTeX, LEDA, List, LWO, MAT, MathML, MBOX, MDB, MESH, MGF, MIDI, MMCIF,
  MOL, MOL2, MP3, MPS, MTP, MTX, MX, NASACDF, NB, NDK, NetCDF, NEXUS,
  NOFF, OBJ, ODS, OFF, OGG, OpenEXR, Package, Pajek, PBM, PCX, PDB, PDF,
  PGM, PLY, PNG, PNM, PPM, PXR, QuickTime, Raw, RawBitmap, RawJSON,
  Real128, Real32, Real64, RIB, RSS, RTF, SCT, SDF, SDTS, SDTSDEM, SFF,
  SHP, SMILES, SND, SP3, Sparse6, STL, String, SurferGrid, SXC, Table, TAR,
  TerminatedString, Text, TGA, TGF, TIFF, TIGER, TLE, TSV, UnsignedInteger128,
  UnsignedInteger16, UnsignedInteger24, UnsignedInteger32, UnsignedInteger64,
  UnsignedInteger8, USGSDEM, UUE, VCF, VCS, VTK, WAV, Wave64, WDX, WebP,
  WLNet, XBM, XHTML, XHTMLMathML, XLS, XLSX, XML, XPORT, XYZ, ZIP}
```

Supported file types range from numeric data to images, to sound, to markup languages, to specialized formats that may include additional metadata. The documentation contains simple examples for each type of file format, and searching for a specific file format is a quick way to find examples of how to work with that type of data.

Instead of having special import commands for different file types, there is a single command, **Import**, that takes a file path as its primary argument. Besides a user's own data, Mathematica also provides example data files as part of its standard installation, and these can be used for testing and exploration. The following command imports a CSV file from this set of example data files by giving its file path to the **Import** command.

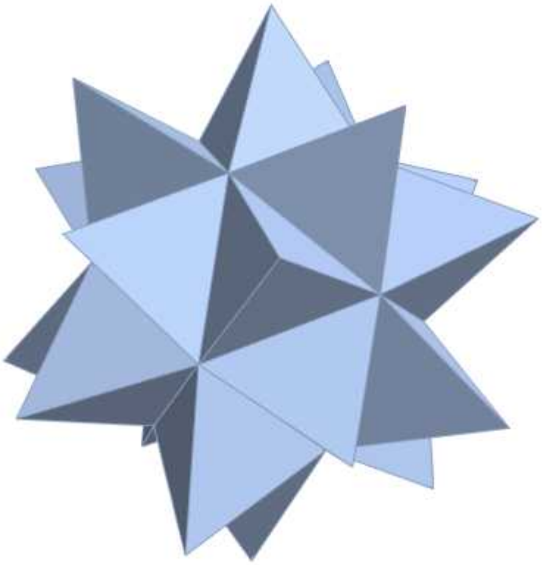
```
Import["ExampleData/numberdata.csv"]
```

```
{{1.2, 4.5, 6.7}, {5.4, 1., 0.}, {0., 2.1, 3.1}}
```

If you search for **ExampleData** in the documentation, you can learn about what other files are included, along with examples that you can try out. You can also see a complete list of available files by evaluating **ExampleData[All]**.

As long as the file type is supported, **Import** takes care of the heavy lifting and brings the data into the notebook in a form that can be used for further operations. For example, the following command imports a 3D geometry file, which is rendered and can be rotated like all other 3D objects.

```
Import["ExampleData/spikey.dxf"]
```



Importing Common File Types

A common starting point when importing external files is to import data containing a list of numbers or an array of values, and such files may also contain textual headers or descriptors. Spreadsheet applications typically display and store numeric approximations, so importing a spreadsheet of numbers will usually create a list with approximated values.

> This means that even if your spreadsheet contained integers like 1, 4 and 10, they might be imported as **1.**, **4.** and **10.** to indicate they are approximations. If the values really are integers and the decimal points bother you, then you can pass the list as an argument to the **Round** command to convert the values to integers.

In the Wolfram Language, blocks of text are represented as strings, and strings are the default representation for text that is imported into the system. Just as it is possible to assign a list of values to a variable directly, a list of values resulting from the **Import** command can also be assigned to a variable. In the following example, the symbol **data1** is assigned a nested list of real numbers and strings that are imported from a spreadsheet file.

```
data1 = Import["ExampleData/population.xls", {"Data", 1}]
```

```
{{1.31397 × 10^9, China}, {1.09535 × 10^9, India},
  {2.98444 × 10^8, United States}, {2.45453 × 10^8, Indonesia},
  {1.88078 × 10^8, Brazil}, {1.65804 × 10^8, Pakistan}, {1.47365 × 10^8, Bangladesh},
  {1.42894 × 10^8, Russia}, {1.3186 × 10^8, Nigeria}, {1.27464 × 10^8, Japan}}
```

> The second argument given to the **Import** command in the preceding example is to specify that only the first worksheet of the spreadsheet should be imported. This topic is covered in much more detail in the following chapter on data manipulation.

The **Length** and **Dimensions** commands can be used to identify the quantity of elements in a nested list. These commands can serve as checks to verify that the imported data is being represented as expected. The **Length** command gives the number of elements of the list, and the **Dimensions** command gives the dimensions.

```
Length[data1]
```

```
10
```

```
Dimensions[data1]
```

```
{10, 2}
```

The data has been verified to have length 10 (since it contains 10 elements, each of which is a sublist) and dimensions {10, 2} (since it contains 10 elements, each of which contains two elements).

> **Dimensions** gives more information than **Length**, but **Length** is perfectly acceptable if you only care about the number of elements in the parent list, or if your list is one-dimensional. If you want the complete picture of how an imported data file is structured, though, you should use **Dimensions**.

When working with large datasets, using the semicolon to suppress the output can be a useful approach to avoid accidentally printing large amounts of data to the display. The **Dimensions** command can still be used to ascertain that the imported data is being represented in a manner consistent with expectations.

```
data2 = Import["ExampleData/population.xls", {"Data", 1}];
Dimensions[data2]
```

```
{10, 2}
```

As mentioned earlier, when textual data is imported into Mathematica, it will be represented as a string or list of strings. Evaluate the following command to import some textual data and store the result in a variable.

```
data3 = Import["ExampleData/USConstitution.txt"];
Dimensions[data3]
```

```
{}
```

The **Dimensions** command returns an empty list because the data that was imported is not stored as a list but rather is stored as a string. This can be verified by using the **Head** command, which takes an argument and returns symbolic representation of that argument. When **Head** is used with the variable **data3**, it indicates that its information is stored as a string.

```
Head[data3]
```

```
String
```

Using **Head** with a list will return the value **List**.

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

```
List
```

This property is true even if the list is a list of strings. After all, at the top level, a list of strings is still a list.

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

```
List
```

This means that the information stored in the variable **data3** is a single string, and commands like **Length** and **Dimensions** are not applicable to this variable. However, a different command, **StringLength**, can be used to determine the number of characters in the single string.

```
StringLength[data3]
```

```
44 808
```

Wondering how to import your own files? This chapter uses sample datasets, since they will work on any machine with Mathematica, but a later section has instructions on how to specify a file path for a file located on your local machine or in the cloud.

Importing Images and Sounds

When Mathematica imports an image, the data is represented as an **Image** expression. The underlying data can be extracted, but the default output is a rendering of the image itself. For example, the following command imports an image that is 200 pixels wide by 200 pixels tall, and the output shows the image instead of returning a list of pixel values.

```
data4 = Import["ExampleData/ocelot.jpg"]
```

Like the previous example with strings, the **Dimensions** command will not work because an image is stored as an **Image** expression, just as a string is stored as a **String** expression.

```
Dimensions[data4]
```

```
{}
```

However, the **ImageDimensions** command can be used to determine the size of the image.

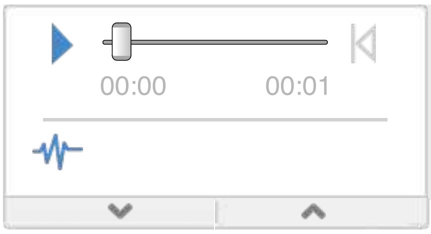
```
ImageDimensions[data4]
```

```
{200, 200}
```

The Wolfram Language contains many commands for image processing, but that topic is not discussed in much detail in this book. The documentation has a guide page for image processing and analysis that comprehensively outlines the functionality available for that type of work.

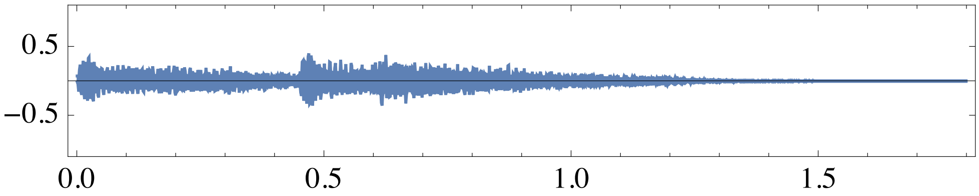
When an audio file is imported into Mathematica, the output generates a built-in audio player that can be used to listen to the file.

```
Import["ExampleData/rule30.wav"]
```



The audio player is an easy and efficient way to verify that the external file was imported correctly. In addition, **AudioPlot** graphically shows the loudness of the audio throughout the duration of the audio file. The same **AudioPlot** is also shown when mousing over the audio data icon in the preceding audio player output.

```
AudioPlot[Import["ExampleData/rule30.wav"]]
```



If the underlying data is of interest or importance, then a second argument for the **Import** command can be used to extract this data. The output indicates that **data5** is a single list with 79,830 elements.

```
data5 = Import["ExampleData/rule30.wav", "Data"];
Dimensions[data5]
```

```
{79 380}
```

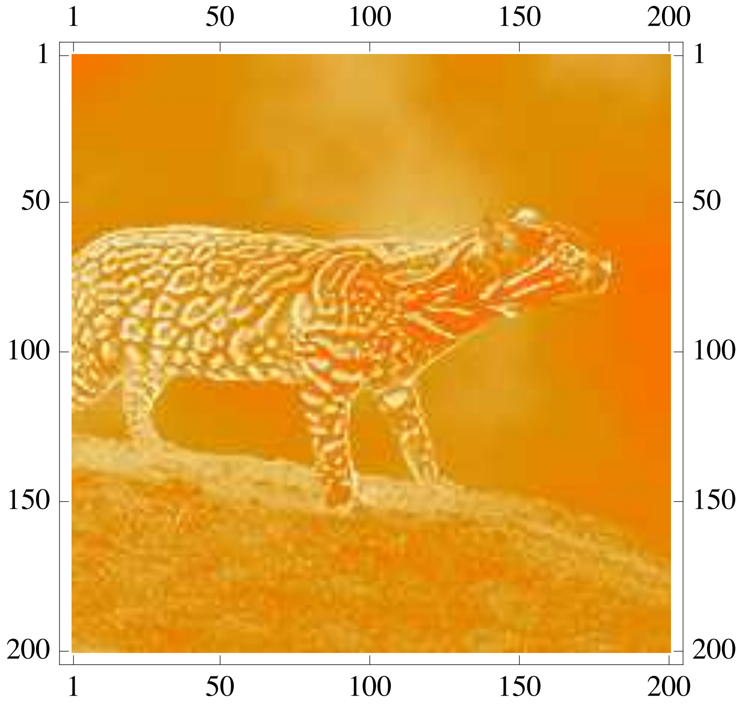
The same approach to extracting underlying data works with other file types, like images. Rather than using **ImageData** on a file that is already imported in order to get access to the underlying data, the underlying data itself can be directly imported.

```
data6 = Import["ExampleData/ocelot.jpg", "Data"];
Dimensions[data6]
```

```
{200, 200}
```

One common theme in Mathematica is that all functionality is nicely interconnected. For example, once an image is represented as a dataset of its pixel values, any plotting functions to visualize data can be used with the dataset. Here, a **MatrixPlot** is used to visualize the pixel values of the ocelot image stored in the variable **data6**.

```
MatrixPlot[data6]
```



Importing Local Files

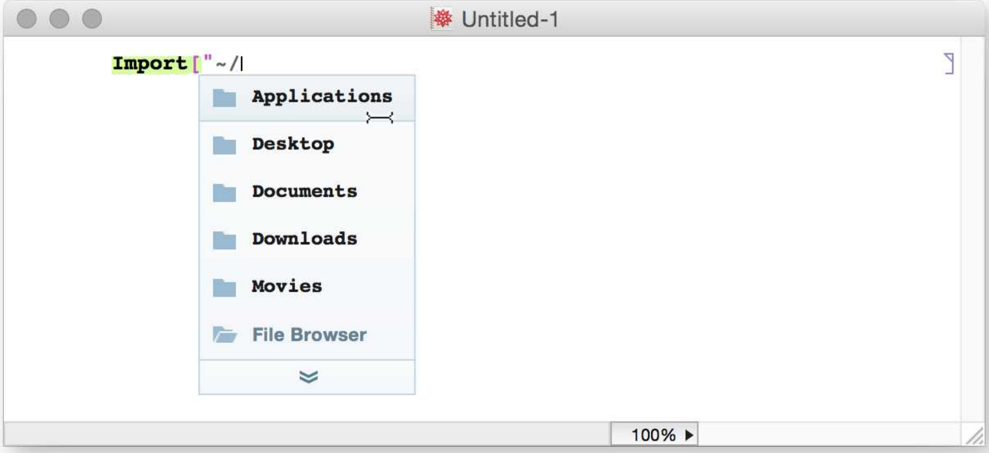
Importing files stored on a local machine is simple, thanks to a menu item that can be used to browse to the file of interest. Place the cursor inside the **Import** command and choose **File Path** from the **Insert** menu to open a dialog window that allows the file to be selected by browsing the computer's file system.

Create an empty **Import[]** command, then place the cursor between the brackets. Choose **File Path** from the **Insert** menu, and browse your filesystem to look for a suitable file to import. Once the file has been selected, evaluate the command to see the result. If the result is not as expected, do not worry—data manipulation and reformatting will be discussed in the next chapter.

```
Import[]
```

The example above is incomplete, since each machine running Mathematica will potentially have a unique file path, but it gives you the starting ground to import your own file using the **Insert** menu.

Alternatively, the Code Assist autocompletion feature extends to browsing local file systems as well. As arguments are given to the **Import** command, the autocompletion popup window will suggest folder and file names to choose from to make the process of selecting the correct file even easier. The popup window includes a link to the same file browser that can be accessed by using the **Insert** menu.



```
Import["~/|
```

If the full file path is not specified for a command like **Import**, Mathematica will look for the file in the current working directory. If the file is found, it will be imported; if the file is not found, an error message will be displayed.

The current working directory can be found by evaluating the **Directory** command, which is an example of a function that does not take any arguments.

```
Directory[]

 /Users/michael/Desktop
```

Many Mathematica users have a deliberate folder structure to organize relevant project files in appropriate directories. If that is the case, then **SetDirectory** can be used to specify the working directory that should be used by default when working with external files. Using a **SetDirectory** command with an organized file structure allows files to be imported and exported by referencing just their local file name instead of the absolute file path, which can save on typing and prevent frustration from having to remember details of an operating system's directory structure.

> Some additional commands can make using **SetDirectory** even easier. Type **SetDirectory[FileNameDrop[]]** in an input cell, place the cursor in the inner set of brackets and choose **File Path** from the **Insert** menu. Browse and select any file located in the directory that you wish to set as your working directory, and evaluate the command. You have now set your working directory without needing to know or type the directory name itself.

Importing Files from the Web

So far, this chapter has outlined how to import files that are stored on the same machine as Mathematica. It is also possible to specify a file path to a shared network drive in order to import a file located on a remote machine or file server. If the file server requires a secure connection, Mathematica will prompt the user for a login and password before importing the file.

The **Import** command also accepts URLs as input through HTTP and FTP. With many files being stored and distributed online, the ability to directly import such files into a Mathematica session makes for a streamlined workflow.

The following example imports a text file from a public website using HTTP.

```
data7 = Import["http://www.handsonstart.com/ExampleDataScores.txt"]
```

```
Joe Smith  94
Jane Smith  85
Bob Example  82
Bill Student  83
Michelle Abacus  98
```

When a file is imported by just passing its file name to the **Import** command, Mathematica will make determinations on how to represent imported data. In this particular example, since no obvious delimiter is used to separate the text from the numeric values, the default representation is that of a single string with 82 characters. This can be verified through the same methods already presented.

```
Dimensions[data7]
```

```
{}
```

```
StringLength[data7]
```

```
82
```

The format of the source file makes it difficult to work with the data in a meaningful way. Luckily, Mathematica provides an easy workaround by allowing the underlying data to be extracted, which is the same approach that was shown earlier.

```
data8 = Import["http://www.handsonstart.com/ExampleDataScores.txt", "Data"]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 85},
  {Bob, Example, 82}, {Bill, Student, 83}, {Michelle, Abacus, 98}}
```

The Mathematica representation of the underlying data is much more useful, separating text into strings and representing the numeric data as actual values.

```
Dimensions[data8]
```

```
{5, 3}
```

Additional strategies and methods for massaging data into workable form will be discussed in the following chapter.

You might wonder why a text-only representation would be useful. Well, sometimes you might want a text representation. You might want to import a file like the United States Constitution and then count the frequency of a certain word or quantity of spaces. In such cases, having one dataset be represented as a string is useful.

Using **SemanticImport**

The tight integration between different parts of the Wolfram Language is a common theme in this book, and this integration is extremely valuable when importing data. Rather than importing only the data that exists in a file, **SemanticImport** can be used to compare the data with curated datasets from the Wolfram Knowledgebase.

Using **SemanticImport** gives much richer datasets by correlating pieces of data with recognized entities. Rather than importing text like "New York City" and storing that data simply as a string, using **SemanticImport** allows that data to be stored as an entity, meaning that all sorts of information about New York City—home values, crime rates, population, sales tax rate and much, much more—is now associated with that data. When the **SemanticImport** command is used, the imported data is automatically compared against entries in the Wolfram Knowledgebase to find any relevant entities for the data to draw these correlations.

The Wolfram Knowledgebase is the collection of curated datasets that powers Mathematica, Wolfram|Alpha and other products.

When the **Import** command is used to import data, the data is typically represented as numerical values, strings, images or sounds. The following example imports a text file containing the names of the 50 states of the United States. The **First** command is also used to return the first element of the dataset.

```
data9 = Import["ExampleData/50states.txt", "Data"];
First[data9]
```

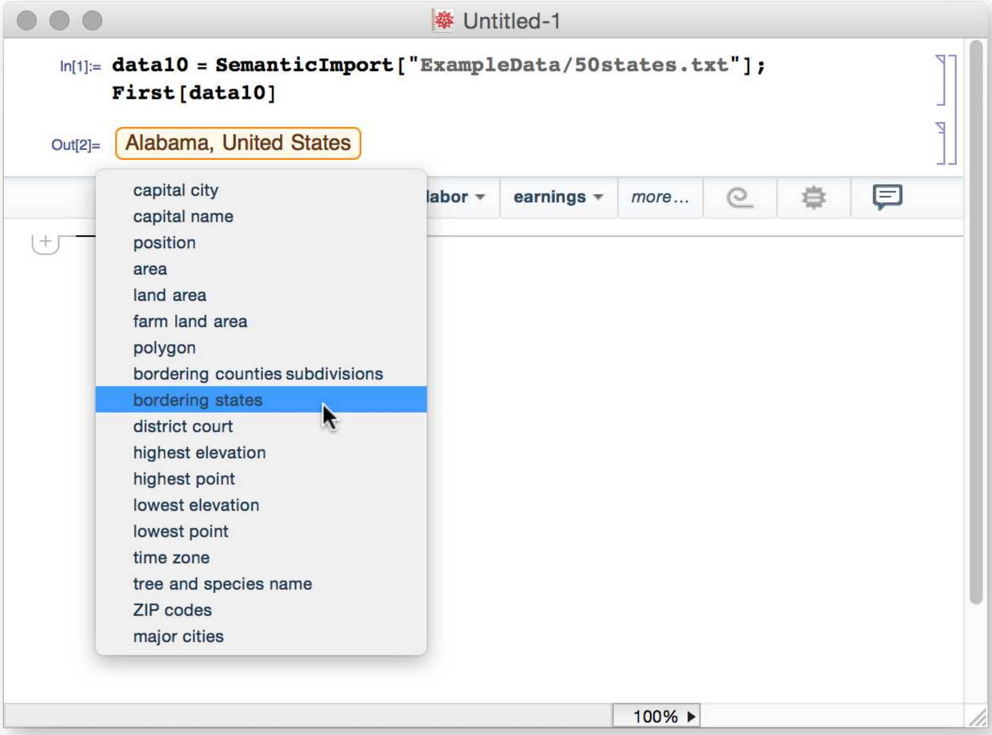
```
Alabama
```

SemanticImport, on the other hand, takes that same text file as input and returns a list of states, each represented by **Entity**. Entity objects are printed with a special form to distinguish them from normal data, and the exact Wolfram Language representation for the **Entity** can be seen by mousing over the box.

```
data10 = SemanticImport["ExampleData/50states.txt"];
First[data10]
```

```
Alabama, United States
```

The Suggestions Bar now provides a list of additional bits of data related to Alabama that can be accessed, since Mathematica now knows that "Alabama" refers to a specific state rather than a generic string of text.



```
In[1]:= data10 = SemanticImport["ExampleData/50states.txt"];
        First[data10]

Out[2]= Alabama, United States
```

Choosing "bordering states" creates a new input cell, just like any other calculation that results from the use of the Suggestions Bar. The input uses the **Entity** function to represent the state of Alabama, and the output also references **Entity** functions related to the bordering states. This makes it easy to continue with calculations related to this new representation of states as entities.

```
AdministrativeDivisionData[ Alabama, United States (administrative division) ,
  "BorderingStates"]

{ Florida, United States , Georgia, United States ,
   Mississippi, United States , Tennessee, United States }
```

Once an entity is identified with **SemanticImport**, it is easy to experiment with the Suggestions Bar to find out what properties are available. A more thorough review of Wolfram Knowledgebase data, however, is covered in another chapter.

Exporting Data from Mathematica

Mathematica can be used for a complete technical workflow, from experimenting with ideas to finding solutions and documenting and presenting results. In fact, using Mathematica for all these tasks often simplifies users' workflows by allowing them to do all their work in a single environment instead of using multiple software applications.

That being said, data can also be exported from Mathematica so that work can be continued elsewhere. Support for exporting data is similarly robust as its import capabilities, with support for many different file types. The complete list of these file types can be seen by evaluating the following command.

```
$ExportFormats

{3DS, ACO, AIFF, AU, AVI, Base64, Binary, Bit, BMP, Byte, BYU, BZIP2, C, CDF,
  Character16, Character8, Complex128, Complex256, Complex64, CSV, CUR,
  DAE, DICOM, DIF, DIMACS, DOT, DXF, EMF, EPS, ExpressionJSON, ExpressionML,
  FASTA, FASTQ, FCS, FITS, FLAC, FLV, GIF, Graph6, Graphlet, GraphML, GXL,
  GZIP, HarwellBoeing, HDF, HDF5, HTML, HTMLFragment, ICNS, ICO, Integer128,
  Integer16, Integer24, Integer32, Integer64, Integer8, JPEG, JPEG2000, JSON,
JVX, KML, LEDA, List, LWO, MAT, MathML, Maya, MGF, MIDI, MOL, MOL2, MP3,
MTX, MX, NASACDF, NB, NetCDF, NEXUS, NOFF, OBJ, OFF, OGG, Package, Pajek,
PBM, PCX, PDB, PDF, PGM, PICT, PLY, PNG, PNM, POV, PPM, PXR, QuickTime,
RawBitmap, RawJSON, Real128, Real32, Real64, RIB, RTF, SCT, SDF, SND,
Sparse6, STL, String, SurferGrid, SVG, SWF, Table, TAR, TerminatedString,
TeX, TeXFragment, Text, TGA, TGF, TIFF, TSV, UnsignedInteger128,
UnsignedInteger16, UnsignedInteger24, UnsignedInteger32, UnsignedInteger64,
UnsignedInteger8, UUE, VideoFrames, VRML, VTK, WAV, Wave64, WDX, WebP,
WLNet, X3D, XBM, XHTML, XHTMLMathML, XLS, XLSX, XML, XYZ, ZIP, ZPR}
```

Exporting Lists of Numbers

The **Export** command can be used to write a new file based on an expression, like a list, graphic or sound file. This single command can be used to export any of the supported file types.

The simplest form of **Export** takes two arguments: a file name to export the data to and the expression. The expression can take the form of a variable.

$$\text{data11} = \begin{pmatrix} 3\pi & \frac{1}{7} & 5 \\ 4.5 & 4.75 & 4.875 \\ e & 5! & \text{N}[\pi, 10] \end{pmatrix};$$

The following **Export** statement creates a new file called myExportedFiles.xls and whose contents are the values stored in the variable **data11**. The output for the **Export** command is the file name. Unless given a complete file path, the file is stored in the current working directory.

```
Export["myExportedFile.xlsx", data11]
```

```
myExportedFile.xlsx
```

Is the variable definition necessary here? No, a list can be entered directly as the second argument to the **Export** command rather than using a variable. The advantage to using a variable is convenience.

Rather than opening this file in an external spreadsheet program, it can be imported back in to Mathematica to view the contents of the file.

```
Import["myExportedFile.xlsx"]
```

```
{{{9.42478, 0.142857, 5.}, {4.5, 4.75, 4.875}, {2.71828, 120., 3.14159}}}
```

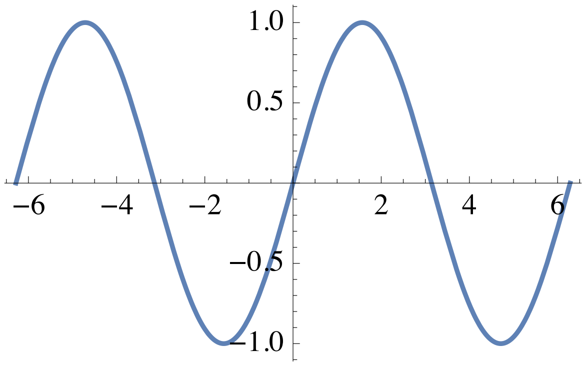
The values are all numeric approximations of the original list that was exported. This is because while Mathematica recognizes and can compute with exact quantities like 3π and $\frac{1}{7}$, when exporting to an external file, the content of the data is based on the conventions and limitations of that file format. For example, π, which in Mathematica represents the exact quantity of the mathematical constant, was converted to a numeric approximation since the XLSX format can only work with a numeric approximation of this quantity.

If the limitations of XLSX or another file format do not work with your project, Mathematica can export as its own notebook file format, where you can use the same useful mix of data structures, along with the ability to store exact values.

Exporting Graphics as Images

Exporting graphics works in a very similar manner to exporting a list, with the **Export** command taking a file name as its first argument and an expression to export as its second argument. The expression to export can be a variable, the output produced by a command or the actual command itself. For example, it may be desirable to export the result from a plot command, like the following.

```
Plot[Sin[x], {x, -2 π, 2 π}]
```

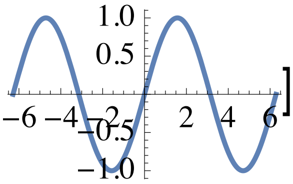


The following three statements all produce identical results.

```
Export["data12–A.png", Plot[Sin[x], {x, –2 π, 2 π}]]
```

```
data12–A.png
```

```
Export["data12–B.png",
```



```
]
```

```
data12–B.png
```

```
myPlot = Plot[Sin[x], {x, –2 π, 2 π}];
Export["data12–C.png", myPlot]
```

```
data12–C.png
```

It is a good idea to pause here to open these files in another software program to see how they look in a program other than Mathematica.

When exporting graphics, finer control can be achieved by specifying options like the **ImageSize** option for plotting commands or by using the **ImageResolution** option for the **Export** command when exporting to a bitmap format. Here, a larger version of the plot is exported because the plot itself is made larger by adding **ImageSize→800** as an option to **Plot**.

```
Export["data12–D.png",
  Plot[Sin[x], {x, –3 π, 3 π}, ImageSize → 800]]
```

```
data12–D.png
```

The following command creates a higher-resolution version of this same image by adding the **ImageResolution→800** option to the **Export** command.

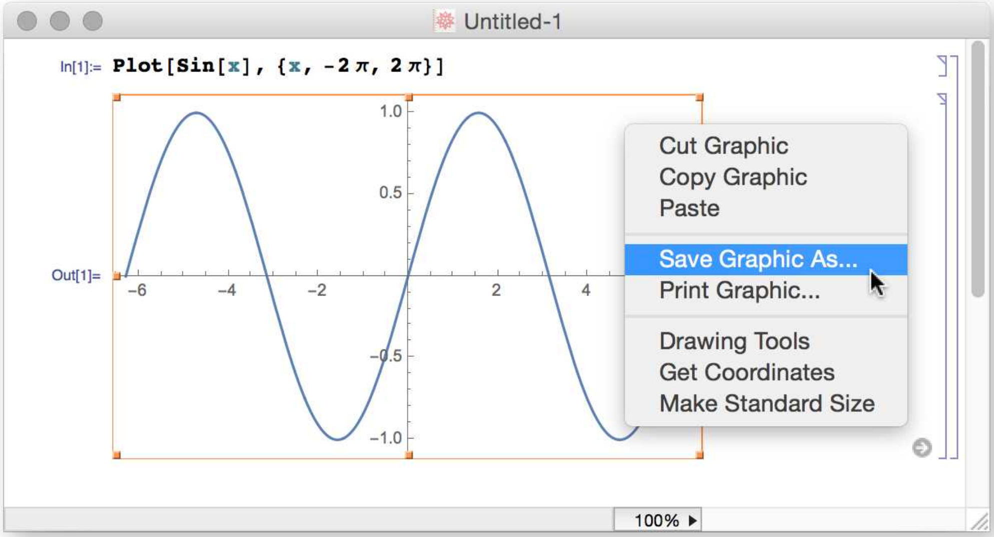
```
Export["data12–E.png",
  Plot[Sin[x], {x, –3 π, 3 π}, ImageSize → 800],
  ImageResolution → 300]
```

```
data12–E.png
```

Open the **data12-D.png** and **data12-E.png** files in a different program to compare the differences between them. You can also compare them to the files exported in the preceding examples.

A Shortcut for Saving Images

The **Export** command gives users the most control over how images are saved, by providing options to control attributes like dimensions and resolution. If such attributes are not important, then a quicker way to save images can be achieved by right-clicking an image and choosing **Save Graphic As**. This will open a dialog box to choose the location to save the file, along with an option for choosing the desired file format.



```
In[1]:= Plot[Sin[x], {x, -2 π, 2 π}]
```

You can also copy and paste graphics into other programs. The results may vary, depending on what program you are pasting into, but simple graphics should be fine in most modern programs that support images as input.

Exporting Graphics for 3D Printing

The previous section shows useful methods for exporting files that can be used in other software environments. 3D printing is also becoming a popular avenue for visualization since it produces a physical object that can be held and examined with a tactile approach.While the **Export** command accepts file formats that are suitable for use with a 3D printer, Mathematica also contains functions specifically designed to create 3D output by eliminating potential errors related to the printing process.

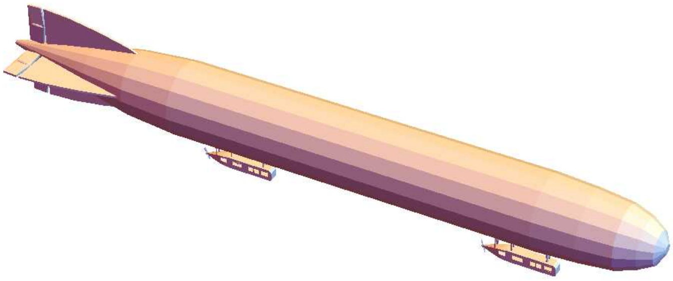
Similar to the built-in datasets for 2D images, Mathematica includes sample datasets for 3D images. These datasets can be discovered by evaluating an appropriate function call to **ExampleData**.

```
ExampleData["Geometry3D"]
```

```
{{Geometry3D, BassGuitar}, {Geometry3D, Beethoven}, {Geometry3D, CastleWall},
  {Geometry3D, Cone}, {Geometry3D, Cow}, {Geometry3D, Deimos},
  {Geometry3D, Galleon}, {Geometry3D, HammerheadShark}, {Geometry3D, Horse},
  {Geometry3D, KleinBottle}, {Geometry3D, MoebiusStrip}, {Geometry3D, Phobos},
  {Geometry3D, PottedPlant}, {Geometry3D, Seashell}, {Geometry3D, SedanCar},
  {Geometry3D, SpaceShuttle}, {Geometry3D, StanfordBunny}, {Geometry3D, Torus},
  {Geometry3D, Tree}, {Geometry3D, Triceratops}, {Geometry3D, Tugboat},
  {Geometry3D, UtahTeapot}, {Geometry3D, UtahVWBug}, {Geometry3D, Vase},
  {Geometry3D, VikingLander}, {Geometry3D, Wrench}, {Geometry3D, Zeppelin}}
```

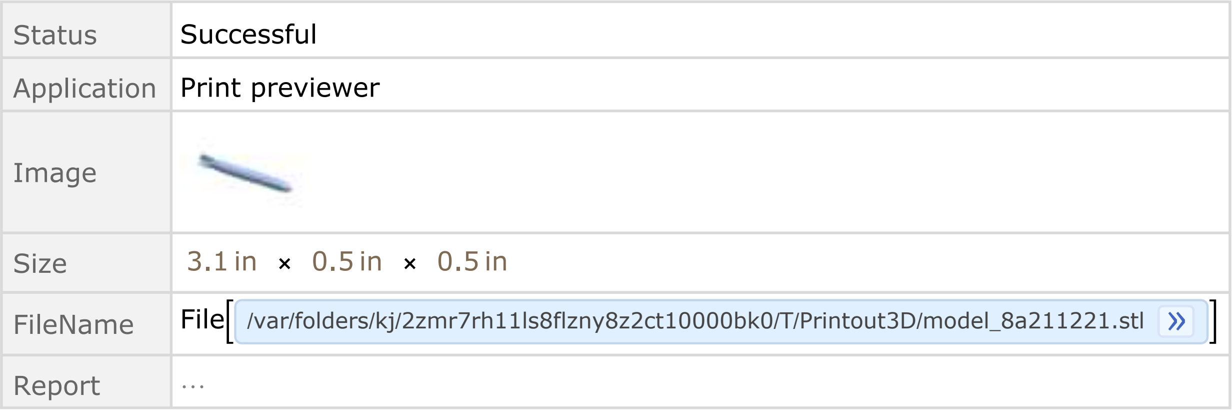
Any of these example models can be passed to the **ExampleData** function to display a 3D rendering of the object. The following input returns a 3D model of a zeppelin, which is stored in a variable named **zeppelinModel**.

```
zeppelinModel = ExampleData[{"Geometry3D", "Zeppelin"}]
```



Instead of using the function **Export**, the **Printout3D** function can be used to create a file optimized for a 3D printer. The output produced by **Printout3D** is a summary of the model file.

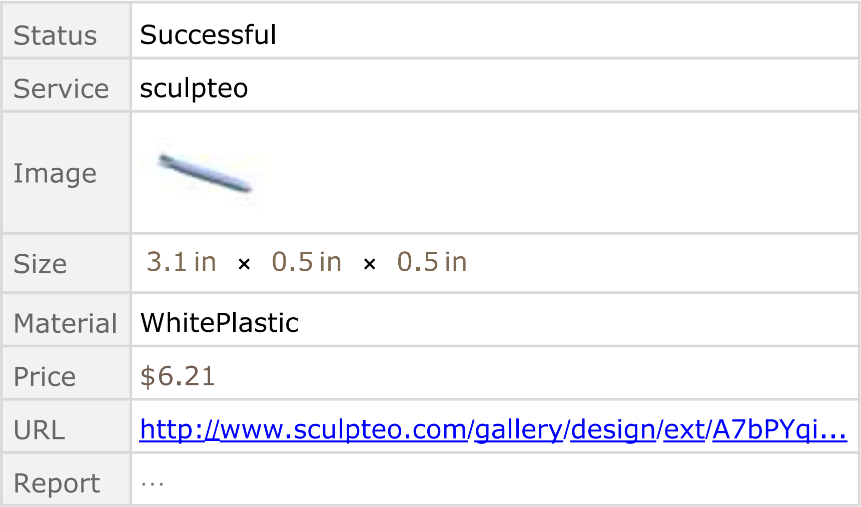
```
Printout3D[zeppelinModel]
```



Status	Successful
Application	Print previewer
Image	
Size	3.1 in × 0.5 in × 0.5 in
FileName	File[/var/folders/kj/2zmr7rh11ls8flzny8z2ct10000bk0/T/Printout3D/model_8a211221.stl »]
Report	⋯

If Mathematica does not have direct access to a 3D printer, a section option for **Printout3D** can specify an external service for 3D printing. Several choices are available for external companies that will print and ship the model for a fee. The following example creates an optimized printing file to be used by the company Sculpteo, and the output includes information on the material and price for the printout, along with a URL to place an order.

```
Printout3D[zeppelinModel, "Sculpteo"]
```



Status	Successful
Service	sculpteo
Image	
Size	3.1 in × 0.5 in × 0.5 in
Material	WhitePlastic
Price	$6.21
URL	http://www.sculpteo.com/gallery/design/ext/A7bPYqi...
Report	⋯

While this example used a built-in dataset to explore the concept of 3D printing, Mathematica's 3D printing capabilities can be used with any output that is amenable to that format. More information can be found on the guide page for 3D printing in the Documentation Center.

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[data1, data2, data3, data4, data5, data6, data7, data8, data9, data10,
  data11, myPlot, zeppelinModel]
```

Conclusion

Mathematica's support for importing and exporting files makes it an excellent environment for data processing. The functionality for importing and exporting is integrated with all the other parts of the system, providing users with a single platform for a complete technical workflow or allowing users to export results to continue their work somewhere else.

Once data is imported into Mathematica, some sorting, reformatting or filtering is often useful to visualize different aspects of the data. The Wolfram Language includes a rich set of functions for data manipulation, which will be outlined in the next chapter.

Exercises

1. Use the Wolfram Language to list the names of the example datasets that are accessible by using **ExampleData**.
2. "1138BUS" is a specific example available within the "Matrix" dataset. Use **ExampleData** to retrieve this dataset.
3. Use the Suggestions Bar with the output from Exercise 2 to create a matrix plot of the data.
4. Import the dataset from www.handsonstart.com/HOS-Chapter19-1.xlsx and assign it to the variable **testdata19**.
5. Find the dimensions of **testdata19**.
6. Import the image from www.handsonstart.com/HOS-Chapter10-2.jpg and assign it to the variable **testimage19**.
7. Find the image dimensions of **testimage19**.
8. Semantically import the dataset from www.handsonstart.com/HOS-Chapter19-3.csv.
9. In the output from Exercise 8, hover over the name of the city in the first row. A tooltip window will show the Wolfram Language representation of the piece of data. Evaluate that Wolfram Language command in a new input cell.
10. Use the Suggestions Bar and the output from Exercise 9 to find the coordinates of that city.

CHAPTER 20

Data Filtering and Manipulation

Introduction

Mathematica has robust capabilities for importing data by supporting many different file types and even applying semantic analysis to recognize data as real-world objects. Data can be imported for all types of analytical work, from computing statistics to creating visualizations. However, the data that is imported might not always be in the ideal structure for immediately working with it: there may be extraneous data that should be discarded, or missing data that should be filled in, or multiple datasets that may need to be combined together, to name just a few examples.

This chapter outlines the scope of Mathematica's data filtering and manipulation capabilities. These techniques can be used to reformat lists, extract values and prepare data for the next part of a workflow, like visualization.

Extracting Parts of Datasets

The previous chapter outlined how Mathematica stores many imported data formats as lists. Once a dataset is defined as a list, parts of the dataset can be extracted for further analysis. The main command to extract part of a dataset is **Part**, which takes a list as its first argument and a position or a range of positions as its second argument. The following example returns the fourth element of the list.

```
Part[{1, 10, 100, 1000}, 4]
```

```
1000
```

The **Part** command is so commonly used that it has its own shortcut notation, **[[]]**. The following example is functionally identical to using the **Part** function, as in the preceding example.

```
{1, 10, 100, 1000}[[4]]
```

```
1000
```

Another version of the **Part** command uses ⟦ ⟧ as a special symbolic notation. This allows the list extraction to be more easily differentiated than **[[]]**, which is especially useful in a compound expression where brackets are also being used to surround the arguments of other commands. The ⟦ ⟧ symbols can be entered with the escape sequence Esc [[Esc for the left bracket and the escape sequence Esc]] Esc for the right bracket. Here is the same command as the preceding example but with the special symbolic form of the **Part** command.

```
{1, 10, 100, 1000}⟦3⟧
```

```
100
```

All three of these notations are functionally identical, so use the notation that makes the most sense to you. Most examples in this book use the ⟦ ⟧ form, since it is more concise than the **Part** function and allows easy distinction between different types of brackets.

The **Part** command also works with nested lists. The following command imports a list of students and test scores and then stores the result in the variable **listOfScores**.

Remember: We suggest that user-defined variables and functions start with lowercase letters to differentiate them from Wolfram Language commands. Using this naming scheme allows for immediate distinction between a built-in Wolfram Language command and one that was defined by a user.

```
listOfScores = Import["http://www.handsonstart.com/ExampleDataScores.txt", "Data"]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 85},
  {Bob, Example, 82}, {Bill, Student, 83}, {Michelle, Abacus, 98}}
```

The **Part** command can now be used to extract elements from this list. When extracting the first element of **listOfScores**, a sublist containing three elements is returned.

```
listOfScores⟦1⟧
```

```
{Joe, Smith, 94}
```

The **Part** command has a syntactical form that uses multiple arguments to extract values from multiple levels of nesting. The following example grabs the first sublist and then extracts the third element from that sublist, which corresponds to the test score for Joe Smith.

```
listOfScores⟦1, 3⟧
```

```
94
```

Sometimes an entire row or column is needed rather than a specific value. In such cases, **All** can be used. To elucidate the discussion, first examine the **listOfScores** variable in tabular form.

```
TableForm[listOfScores]
```

```
Joe        Smith      94
Jane       Smith      85
Bob        Example    82
Bill       Student    83
Michelle   Abacus     98
```

TableForm is a display function used to visualize rows and columns of data. **TraditionalForm** is another option, which will display any list in a matrix format and will line up the rows and columns in a similar style.

While examining the data in that format, think of the two-argument syntax for **Part** as requesting the row and column from the data, but rather than requesting a single value, the **All** descriptor is used to specify all values for the row or column. For example, to extract the third row, the entries from row 3 (**⟦3, …⟧**) and the entries from all the columns of row 3 (**⟦…, All⟧**) are needed.

```
listOfScores⟦3, All⟧
```

```
{Bob, Example, 82}
```

Similarly, to extract the second column, the entries from all the rows and the entries from column 2 are needed. The following statement returns only the last names of the students in the dataset.

```
listOfScores⟦All, 2⟧
```

```
{Smith, Smith, Example, Student, Abacus}
```

Besides extracting specific values, entire rows and entire columns, **Part** can also be used to extract spans or submatrices of data. Spans are specified using the **;;** notation. The following command will extract the values at positions 2 through 4 of the dataset.

```
{2, 10, 100, 1000, 2000}⟦2 ;; 4⟧
```

```
{10, 100, 1000}
```

Combining spans with the other methods outlined previously provides a powerful and concise way to extract ranges of contiguous rows or columns.

```
TableForm[listOfScores]
```

```
Joe         Smith       94
Jane        Smith       85
Bob         Example     82
Bill        Student     83
Michelle    Abacus      98
```

The following extracts only the first and second columns of the dataset, resulting in just the first and last names with the score excluded.

```
listOfScores⟦All, 1 ;; 2⟧
```

```
{{Joe, Smith}, {Jane, Smith}, {Bob, Example}, {Bill, Student}, {Michelle, Abacus}}
```

Part also allows noncontiguous positions to be specified as a list of values. For example, this command will extract the first and third columns of the dataset.

```
listOfScores⟦All, {1, 3}⟧
```

```
{{Joe, 94}, {Jane, 85}, {Bob, 82}, {Bill, 83}, {Michelle, 98}}
```

Negative numbers can be used in a **Part** command as well. Evaluating **list⟦-1⟧** will return the last item from **list**, evaluating **list⟦-2⟧** will return the second-to-last item in **list** and so on.

Besides extraction, the **Part** command can be used to overwrite values in a list. The following example overwrites the fourth entry in the original dataset with a new sublist. The output displayed from this operation is only the replacement values, not the entire list.

```
listOfScores⟦4⟧ = {"New", "Entry", 81}
```

```
{New, Entry, 81}
```

Evaluating the **listOfScores** variable will show all the values currently stored in the variable, including the values that were just updated.

```
listOfScores
```

```
{{Joe, Smith, 94}, {Jane, Smith, 85},
  {Bob, Example, 82}, {New, Entry, 81}, {Michelle, Abacus, 98}}
```

The ordering of a list depends on how the list was first defined. Sorting a list will be covered in detail later in this chapter.

In a nested list, single values can be overwritten through proper specification down to the element in question. Since the **listOfScores** variable is two dimensional, the following will change the value in the second row and third column. In this specific example, the entry for Jane Smith is being changed from a score of 85 to a score of 88.

```
listOfScores⟦2, 3⟧ = 88
```

```
88
```

Evaluating the list shows the updated score for Jane Smith.

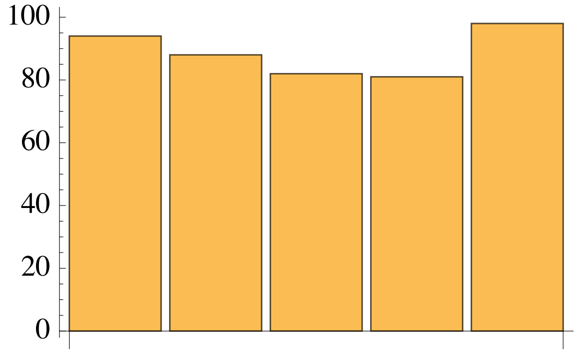
```
listOfScores
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88},
  {Bob, Example, 82}, {New, Entry, 81}, {Michelle, Abacus, 98}}
```

If you have been recreating this notebook in order, you will note that **listOfScores** has been evaluated several times in this chapter, and different outputs are displayed corresponding to the then-current values stored in the variable. When a variable definition is changed, it does not force the previous definitions to change unless some additional commands (e.g. **Dynamic**) are used. However, the **In** and **Out** cell labels that number the order of evaluations for a current session can provide insight about the order in which inputs have been evaluated.

Data extraction is useful when preparing data to be used with other commands, like those used for visualization. For example, the **BarChart** command might be used to visualize the exam scores that are stored in the **listOfScores** variable.

```
BarChart[listOfScores⟦All, 3⟧]
```



What happens if you just pass the **listOfScores** variable, in entirety, to **BarChart**? Well, Mathematica is forgiving, and it will actually output a result, although maybe not the one you really want. **BarChart** will not plot string values, so it will create bars only for the numerical values stored in the third position—but the chart will look rather strange, with the bars spaced far apart due to the empty space where the strings are encountered in positions one (first name) and two (last name) of the list.

The **Part** command is particularly useful when working with imported data. Files created by another application or another person may have unexpected or inconsistent formatting that makes the data difficult to visualize or display.

For example, when a spreadsheet is imported, Mathematica automatically creates a nested list based on values from different worksheets in the spreadsheet. This means that if there was only one worksheet represented in the file, there will be an extra set of braces when the spreadsheet is imported. Note the extra set of curly braces in the following example.

```
Import["ExampleData/elements.xls"]
```

```
{{{AtomicNumber, Abbreviation, Name, AtomicWeight},
    {1., H, Hydrogen, 1.00793}, {2., He, Helium, 4.00259}, {3., Li, Lithium, 6.94141},
    {4., Be, Beryllium, 9.01218}, {5., B, Boron, 10.8086}, {6., C, Carbon, 12.0107},
    {7., N, Nitrogen, 14.0067}, {8., O, Oxygen, 15.9961}, {9., F, Fluorine, 18.9984}}}
```

The structure of the dataset can be verified using the **Dimensions** command. In this case, **Dimensions** verifies that the data contains only 1 worksheet with 10 rows and 4 columns of data.

```
Dimensions[Import["ExampleData/elements.xls"]]
```

```
{1, 10, 4}
```

> If this particular spreadsheet had data on two worksheets, for example, then the output from Dimensions would be **{2, 10, 4}** to indicate that the data contains 2 worksheets, each of which has 10 rows and 4 columns of values.

The preceding example had an extra (and unnecessary) parent list enclosing the data. The **Part** command can be used to extract the first element of data from the first list, which in this case is the top-level list. Using this approach eliminates the extra braces and returns the rest of the data.

```
Import["ExampleData/elements.xls"]⟦1⟧
```

```
{{AtomicNumber, Abbreviation, Name, AtomicWeight},
  {1., H, Hydrogen, 1.00793}, {2., He, Helium, 4.00259}, {3., Li, Lithium, 6.94141},
  {4., Be, Beryllium, 9.01218}, {5., B, Boron, 10.8086}, {6., C, Carbon, 12.0107},
  {7., N, Nitrogen, 14.0067}, {8., O, Oxygen, 15.9961}, {9., F, Fluorine, 18.9984}}
```

Since the extra level of nesting was removed from the list, using **Dimensions** shows that the dataset is now two dimensional, with 10 rows and 4 columns.

```
Dimensions[Import["ExampleData/elements.xls"]〚1〛]
```

```
{10, 4}
```

An optional argument can be passed to the **Import** command when importing a spreadsheet to specify data from a particular worksheet. Using this argument eliminates the situation with the extra braces and the subsequent need to postprocess the imported data. The following example specifies that only the first element of the file—which in this case is the one and only worksheet in the file—should be imported as raw data.

```
Import["ExampleData/elements.xls", {"Data", 1}]
```

```
{{AtomicNumber, Abbreviation, Name, AtomicWeight},
  {1., H, Hydrogen, 1.00793}, {2., He, Helium, 4.00259}, {3., Li, Lithium, 6.94141},
  {4., Be, Beryllium, 9.01218}, {5., B, Boron, 10.8086}, {6., C, Carbon, 12.0107},
  {7., N, Nitrogen, 14.0067}, {8., O, Oxygen, 15.9961}, {9., F, Fluorine, 18.9984}}
```

There are other options besides **"Data"** for the second option of **Import**; instead of importing as raw data, files can be imported as images, graphics and plain text.

Using **Dimensions** shows that the dataset that was imported with this new approach is two dimensional, with 10 sublists of 4 elements each. (This is equivalent to picturing the dataset as a spreadsheet with 10 rows and four columns.)

```
Dimensions[Import["ExampleData/elements.xls", {"Data", 1}]]
```

```
{10, 4}
```

Specifying this second argument in the **Import** statement returns the same result as postprocessing imported data by using the **Part** command. An advantage to specifying a subset of the data—like the first and only worksheet of a spreadsheet—when using the **Import** command is that there is no need to manipulate the data afterward, so you can start using the results immediately. Both methods will work, so it is left for you to choose the one you like.

Changing the Structure of Lists

Lists can be constructed manually through typing or copying and pasting. The following example takes the dataset of student test scores that was stored as **listOfScores** and adds a new sublist to create a larger list. The result is stored as **newScores**.

```
newScores =
  {listOfScores, {{"Michael", "Morrison", 95}, {"Kelvin", "Mischo", 96},
      {"Cliff", "Hastings", 99}}}
```

```
{{{Joe, Smith, 94}, {Jane, Smith, 88},
    {Bob, Example, 82}, {New, Entry, 81}, {Michelle, Abacus, 98}},
  {{Michael, Morrison, 95}, {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}}}
```

Now consider the problem of extracting the last names from **newScores**. Since this list is separated into two different sublists, the previous method of using **Part** to extract all the second elements of the list does not work.

```
newScores⟦All, 2⟧
```

```
{{Jane, Smith, 88}, {Kelvin, Mischo, 96}}
```

You may be wondering why this does not work. When this approach was done previously using **listOfScores**, the list was two-dimensional; there was a single parent list, comprised of sublists, and each sublist was of the form **{firstName, lastName, examScore}**. Evaluating **listOfScores⟦All, 2⟧** extracted all the second elements from the list, which grabbed all the last names. Since **newScores** has three dimensions, though, evaluating **newScores⟦All, 2⟧** extracts the second element of each of the sublists; the second element of each sublist is, in turn, *another* sublist, namely **{{Jane, Smith, 88}}** for the first sublist and **{{Kelvin, Mischo, 96}}** for the second sublist.

The **Part** command can be used on lists of any dimension, so it could help in this situation. For example, **Part** could be used to extract the value at position **{1,1,2}** of **newScores**; this will extract the first sublist, then the first element of that sublist (which is also a sublist: **{Joe, Smith, 94}**) and then the second element of that sublist (**Smith**).

```
newScores⟦1, 1, 2⟧
```

```
Smith
```

Similarly, **Part** could be used to extract the value at position **{2,1,2}** of **newScores**.

```
newScores⟦2, 1, 2⟧
```

```
Morrison
```

With that in mind, the following example is a manual and verbose way to extract all the last names from **newScores**.

```
{newScores⟦1, 1, 2⟧, newScores⟦1, 2, 2⟧, newScores⟦1, 3, 2⟧, newScores⟦1, 4, 2⟧,
  newScores⟦1, 5, 2⟧, newScores⟦2, 1, 2⟧, newScores⟦2, 2, 2⟧, newScores⟦2, 3, 2⟧}
```

```
{Smith, Smith, Example, Entry, Abacus, Morrison, Mischo, Hastings}
```

Now, imagine the list of data were hundreds or thousands of elements long. It would be much more appealing to let Mathematica do the work instead. Luckily, there are many Wolfram Language commands to manipulate the structure of lists.

The **Flatten** command eliminates nested lists to create a single, one-dimensional list with the elements in the same order as they were in the original list. The following examples show the current value of **newScores** along with the result of using **Flatten** on this variable.

```
newScores
```

```
{{{Joe, Smith, 94}, {Jane, Smith, 88},
    {Bob, Example, 82}, {New, Entry, 81}, {Michelle, Abacus, 98}},
  {{Michael, Morrison, 95}, {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}}}
```

```
Flatten[newScores]
```

```
{Joe, Smith, 94, Jane, Smith, 88, Bob, Example, 82, New, Entry, 81, Michelle,
  Abacus, 98, Michael, Morrison, 95, Kelvin, Mischo, 96, Cliff, Hastings, 99}
```

To go the opposite direction and create a two-dimensional list from a one-dimensional list, **Partition** is used to create sublists of a specified length. The following example takes a list of 10 elements and puts them in sublists of length 2.

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

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

> The length specification given to **Partition** does not necessarily need to be exactly divisible by the length of the list being partitioned. If a length is used that results in a remainder, then the last few values of the data will be orphaned. For example, when specifying a length of 3 in the preceding example, the value 10 is discarded and the sublists **{{1, 2, 3},{4, 5, 6},{7, 8, 9}}** are returned as output.

Using **Partition** with **Flatten** is one approach to clean up two similarly structured datasets that have been messily combined. First, **Flatten** is used to create a single list, and then **Partition** is used to create sublists of the appropriate length. Since each of the sublists has three elements—first name, last name and exam score—then 3 becomes the second argument for **Partition**. The result is stored in a new variable, **cleanedNewScores**.

```
cleanedNewScores = Partition[Flatten[newScores], 3]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88},
  {Bob, Example, 82}, {New, Entry, 81}, {Michelle, Abacus, 98},
  {Michael, Morrison, 95}, {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}}
```

```
Partition[Flatten[listOfScores⟦All, 1 ;; 2⟧], 5]
```

```
{{Joe, Smith, Jane, Smith, Bob}, {Example, New, Entry, Michelle, Abacus}}
```

Now that there is a single list, it is easy to extract all the last names (which are in position 2) by using **Part**.

```
cleanedNewScores⟦All, 2⟧
```

```
{Smith, Smith, Example, Entry, Abacus, Morrison, Mischo, Hastings}
```

In addition to commands related to extraction and restructuring, a variety of commands are available to delete parts of lists. In this particular case, the **Drop** function can be used to delete the first two elements from the list.

```
Drop[cleanedNewScores, 2]
```

```
{{Bob, Example, 82}, {New, Entry, 81}, {Michelle, Abacus, 98},
  {Michael, Morrison, 95}, {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}}
```

Drop can accept a second argument to drop columns. In this form, the row argument must also be specified, but **None** can be used to specify that only columns are to be dropped. The following drops none of the rows but does drop the first column. The result is a list of last names and exam scores.

```
Drop[cleanedNewScores, None,  1]
```

```
{{Smith, 94}, {Smith, 88}, {Example, 82}, {Entry, 81},
  {Abacus, 98}, {Morrison, 95}, {Mischo, 96}, {Hastings, 99}}
```

Depending on the dataset, it is sometimes easier to specify what should be extracted instead of specifying what should be dropped from a list. The following takes the list of student grades and returns only the second and third columns, which correspond to the last names and grades.

```
cleanedNewScores⟦All, {2, 3}⟧
```

```
{{Smith, 94}, {Smith, 88}, {Example, 82}, {Entry, 81},
  {Abacus, 98}, {Morrison, 95}, {Mischo, 96}, {Hastings, 99}}
```

New elements can be added to lists with commands like **Append** and **Prepend**. These commands create new lists with the data appended or prepended, respectively, but do not overwrite existing symbols.

```
Append[cleanedNewScores, {"Jill New", 90}]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill New, 90}}
```

This new value is not added to the definition **cleanedNewScores**, so when **cleanedNewScores** is evaluated, it does not contain the sublist of **{Jill, New, 90}**.

```
cleanedNewScores
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88},
  {Bob, Example, 82}, {New, Entry, 81}, {Michelle, Abacus, 98},
  {Michael, Morrison, 95}, {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}}
```

AppendTo and **PrependTo** are different from **Append** and **Prepend** in that they *do* overwrite the original list. Otherwise, the syntax for **AppendTo** and **Append** is the same, and the syntax for **PrependTo** and **Prepend** is the same.

```
AppendTo[cleanedNewScores, {"Jill", "New", 90}]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}}
```

Now evaluating the variable **cleanedNewScores** verifies that this new element has been stored in the list.

```
cleanedNewScores
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}}
```

In addition to manipulating lists, Mathematica can manipulate strings in a variety of ways. It is possible to extract parts of a string, replace parts of a string, find patterns in strings or perform other operations, like combining two strings. The last is achieved with **StringJoin**, which will create a single string from the values passed to it as arguments.

```
StringJoin["Joe", "Smith"]
```

```
JoeSmith
```

In the example dataset used in this chapter, the first name and last name of each person are stored as separate elements. **StringJoin** can be used to combine those two elements into a single string. A whitespace character is introduced between the two names to make them easier to read.

```
StringJoin[cleanedNewScores〚1, 1〛, " ", cleanedNewScores〚1, 2〛]
```

```
Joe Smith
```

Using the **Table** command allows this process to be automated for all the elements of the **cleanedNewScores** list.

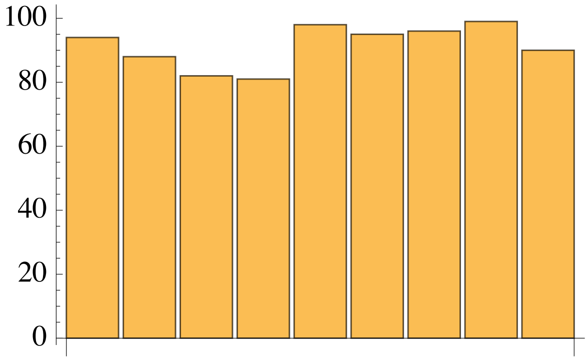
```
Table[StringJoin[cleanedNewScores〚i, 1〛, " ", cleanedNewScores〚i, 2〛],
  {i, 1, Length[cleanedNewScores], 1}]
```

```
{Joe Smith, Jane Smith, Bob Example, New Entry, Michelle Abacus,
  Michael Morrison, Kelvin Mischo, Cliff Hastings, Jill New}
```

Rather than using a literal value for the upper bound of the iterator, the **Length** command is used to determine the number of elements in the list. This allows the same command to be reused in the future if **cleanedNewScores** takes on different values and becomes longer.

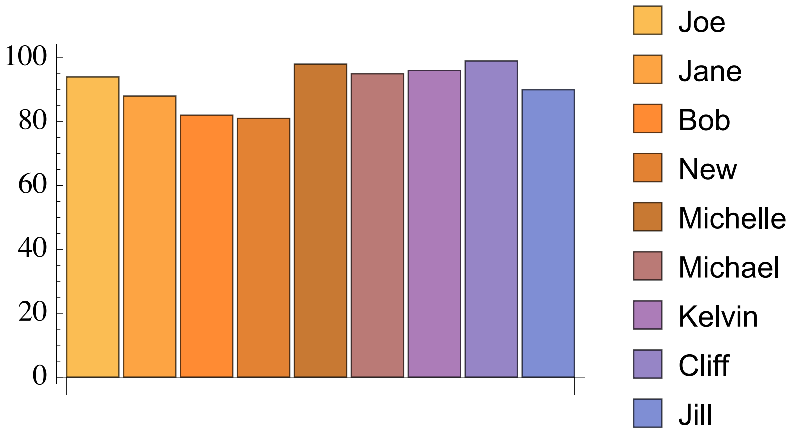
The following is a practical application of using list manipulation commands. The **Part** command can be used to extract just the scores for each student and create a bar chart.

```
BarChart[cleanedNewScores〚All, 3〛]
```



This chart would be more useful if it contained a legend. **ChartLegend** can be used to add this, and the first names can be passed as the option setting once they are extracted from the dataset using **Part**. The dataset is placed inside a parent list; this forces **BarChart** to treat each element as a separate sublist and as a result, each bar is colored differently.

```
BarChart[{cleanedNewScores〚All, 3〛}, ChartLegends → cleanedNewScores〚All, 1〛]
```



Sorting and Pattern Matching

The **Sort** command can be used on either numeric data or strings to order a list. When used on a nested list, the first non-list element is used for determining the sort order. Since the element in the first position of **cleanedNewScores** is a string, the result is ordered by alphabetical order of the strings in the first position.

```
Sort[cleanedNewScores]

 {{Bob, Example, 82}, {Cliff, Hastings, 99}, {Jane, Smith, 88},
   {Jill, New, 90}, {Joe, Smith, 94}, {Kelvin, Mischo, 96},
   {Michael, Morrison, 95}, {Michelle, Abacus, 98}, {New, Entry, 81}}
```

Of course, if a sublist is passed, then it is ordered according to the type of data in that sublist. For example, all the numeric values can be extracted using **Part**, and the results can be sorted with **Sort**.

```
Sort[cleanedNewScores⟦All, 3⟧]

 {81, 82, 88, 90, 94, 95, 96, 98, 99}
```

With a nested list, the **SortBy** command can be used to specify the particular function or pattern to be used when sorting the list. One pattern can be specified by using **Last** as the second argument; this will return the list sorted by the values in the last position of the list. The last element of **cleanedNewScores** is numerical values, so the results are returned in that order.

```
SortBy[cleanedNewScores, Last]

 {{New, Entry, 81}, {Bob, Example, 82}, {Jane, Smith, 88},
   {Jill, New, 90}, {Joe, Smith, 94}, {Michael, Morrison, 95},
   {Kelvin, Mischo, 96}, {Michelle, Abacus, 98}, {Cliff, Hastings, 99}}
```

The second argument passed to **SortBy** can be a user-defined function. In the following example, a function named **sortFun** takes one argument and returns the length of the string in the first position. This function can be tested for a single case before being used in a **SortBy** function. The following demonstrates that Joe, the first name in the list, has three characters.

```
sortFun[x_] := StringLength[x⟦1⟧]
cleanedNewScores
sortFun[cleanedNewScores⟦1⟧]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}}
```

```
3
```

Remember, multiple statements can be placed in a single input cell, either separated by semicolons or each on its own line. When such an input cell is evaluated, an output cell will be created for each command unless the semicolon is used to suppress the output.

When using **sortFun** with **SortBy**, each of the elements is evaluated with this function, and the numeric results—which correspond to the quantity of characters in the first element of each sublist—are used to sort the list. The following example shows how using this function with **SortBy** sorts the list in order from the shortest first name to the longest first name.

```
sortFun[x_] := StringLength[x⟦1⟧]
cleanedNewScores;
SortBy[cleanedNewScores, sortFun]
```

```
{{Bob, Example, 82}, {Joe, Smith, 94}, {New, Entry, 81},
  {Jane, Smith, 88}, {Jill, New, 90}, {Cliff, Hastings, 99},
  {Kelvin, Mischo, 96}, {Michael, Morrison, 95}, {Michelle, Abacus, 98}}
```

Since there are multiple first names with the same length—like Bob and Joe—**SortBy** does an additional task of sorting based on the first letter of each string and returns the result in alphabetical order.

The **Reverse** function returns a list in reverse order. This can be useful when coupled with **Sort**, as it allows for lists to be sorted in descending order rather than ascending order. The following example uses **SortBy** and **Last** to sort by exam scores in ascending order, and then **Reverse** is used to sort those results in descending order.

```
Reverse[SortBy[cleanedNewScores, Last]]

{{Cliff, Hastings, 99}, {Michelle, Abacus, 98},
  {Kelvin, Mischo, 96}, {Michael, Morrison, 95}, {Joe, Smith, 94},
  {Jill, New, 90}, {Jane, Smith, 88}, {Bob, Example, 82}, {New, Entry, 81}}
```

Lists can also be manipulated through pattern matching. Commands like **Cases** provide an easy way to specify on-the-fly filters to extract data that matches particular patterns. Pattern-matching syntax in the Wolfram Language makes use of the underscore character _ to represent any expression, but this is more commonly used to find patterns that match certain heads. A head is the most basic representation of an expression and can be found with the **Head** command.

```
Head[1]

Integer
```

```
Head[1.5]

Real
```

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

List
```

With this idea in mind, **Cases** can be used to pick out elements that match certain types of patterns. For example, the following command finds all the integer values in a list.

```
Cases[{1, 2.5, 3.1, 4, 5.2, 6, 7.7, 8, 9, 10.3}, _Integer]

{1, 4, 6, 8, 9}
```

If a different head is given, such as **Real**, all real values are found.

```
Cases[{1, 2.5, 3.1, 4, 5.2, 6, 7.7, 8, 9, 10.3}, _Real]
```

```
{2.5, 3.1, 5.2, 7.7, 10.3}
```

If a head that is not present in the list is given as the pattern for **Cases**, then an empty list is returned.

```
Cases[{1, 2.5, 3.1, 4, 5.2, 6, 7.7, 8, 9, 10.3}, _String]
```

```
{}
```

Cases can be used to extract the exam scores from the **cleanedNewScores** variable.

```
Cases[Flatten[cleanedNewScores], _Integer]
```

```
{94, 88, 82, 81, 98, 95, 96, 99, 90}
```

The datasets that have been used for examples thus far have been uniform in structure and types of values. The following **AppendTo** statement introduces a new sublist in the list that does not have the same list structure as the other sublists—it contains a single element instead of a list of three elements.

```
AppendTo[cleanedNewScores, {"Missing"}]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}, {Missing}}
```

When working with data with missing values or with data whose elements may contain different structures, the **Cases** command can be used to identify sublists of a desired structure. For example, only sublists of length three are returned through the following pattern specification. This effectively ignores the data that does not match the desired format of three elements.

```
Cases[cleanedNewScores, {_, _, _}]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}}
```

The pattern used with **Cases** can be as specific as needed. The following **AppendTo** statement introduces a new sublist that has the desired structure of three elements, but it does not have a numeric value in the third position.

```
AppendTo[cleanedNewScores, {"Bad", "Data", "Here"}]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82}, {New, Entry, 81},
  {Michelle, Abacus, 98}, {Michael, Morrison, 95}, {Kelvin, Mischo, 96},
  {Cliff, Hastings, 99}, {Jill, New, 90}, {Missing}, {Bad, Data, Here}}
```

A **Cases** statement can be constructed that ignores sublists that do not have three values and sublists that do not have integer values in the third position.

```
Cases[cleanedNewScores, {_, _, _Integer}]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}}
```

You can take this idea as far as you would like. For example, the pattern could be changed to **{_String, _String, _Integer}** to ignore sublists like **{93, 94, 95}** that only contain numerical values.

Instead of using **Cases** to display data that corresponds to a specified pattern, it is also possible to identify individual elements and delete them with a function called **Nothing**. When an element in a list is replaced with **Nothing**, the element is omitted from the list. The following example takes the variable **cleanedNewScores** and replaces all instances of **{"Missing"}** with **Nothing**, which has the net result of deleting that element from the list.

```
cleanedNewScores /. {"Missing"} → Nothing
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}, {Bad, Data, Here}}
```

Select is also useful for extracting elements, but where **Cases** requires a pattern, **Select** requires statements that must evaluate to **True** or **False**. A preceding example used **Cases** with a pattern to return only sublists of three elements. A similar approach can be used with **Select** by creating a function that returns Boolean values depending on the length of the list that is examined. (Recall that the double equal sign (**==**) stands for testing equality, and that the single equal sign (**=**) is used for assigning values to variables.)

```
selectFun[x_] := Length[x] == 3
```

```
Select[cleanedNewScores, selectFun]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}, {Bad, Data, Here}}
```

Boolean operators such as **And**, **Or** and **Not** can be used to create more complex criteria for **Select**, such as to return values only for people with the last name "Smith" who scored at least 90 on the exam.

```
selectFun2[x_] := And[x[[2]] == "Smith", x[[3]] ≥ 90]
Select[cleanedNewScores, selectFun2]
```

```
      ••• Part: Part 2 of {Missing} does not exist.
      ••• Part: Part 3 of {Missing} does not exist.
{{Joe, Smith, 94}}
```

In the preceding example, a warning message is printed to indicate that the tests performed by **Select** failed, since one of the elements of the list did not have any elements in positions 2 or 3. The error message clearly communicates that the sublist **{"Missing"}** does not have elements in positions 2 or 3. Clicking the three red dots that precede the warning message provides an option to request a stack trace, which helps gain insight into the order of calculations to pin down where the operation went awry.

Select could be combined with **Cases** to address this. **Cases** can be used to filter the data so that only lists of length three with strings in position 2 and integers in position 3 are returned. Then, that result can become the argument for a **Select** command that looks for people with the last name "Smith" who scored at least 90 on the exam.

```
selectFun2[x_] := And[x⟦2⟧ == "Smith", x⟦3⟧ ≥ 90]
Select[Cases[cleanedNewScores, {_, _String, _Integer}], selectFun2]
```

```
{{Joe, Smith, 94}}
```

Select works with any expression that returns a **True** or **False** output. Boolean operators are commonly used since they only return **True** or **False**, but other Wolfram Language commands that return **True** or **False** can also be used.

The support for Boolean constructs makes **Select** an extremely flexible way to filter data according to certain specifications. For example, data at the extremes of the spectrum can be extracted through clever use of **Or** to find all scores either below 85 or above 95. (The same use of **Cases** is employed to first find only data of the relevant form.)

```
selectFun3[x_] := Or[x⟦3⟧ < 85, x⟦3⟧ > 95]
Select[Cases[cleanedNewScores, {_, _String, _Integer}], selectFun3]
```

```
{{Bob, Example, 82}, {New, Entry, 81},
  {Michelle, Abacus, 98}, {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}}
```

In the preceding example, **Select** was used in conjunction with a user-defined function to create the pattern test. However, a user-defined function is not required. There are many commands in the Wolfram Language to test whether a statement is true or false. In the following example, the function **NumberQ** is used with **Select** to only return values from **cleanedNewScores** that are numbers. **Flatten** is also used to simplify the list and make it easier to extract the scores.

```
Select[Flatten[cleanedNewScores], NumberQ]
```

```
{94, 88, 82, 81, 98, 95, 96, 99, 90}
```

EvenQ is a similar function that evaluates to **True** when it encounters an even integer.

```
Select[Flatten[cleanedNewScores], EvenQ]
```

```
{94, 88, 82, 98, 96, 90}
```

There are also comparative functions, like **GreaterThan**, that can be used for Boolean tests. Here, **GreaterThan** is used to find all scores greater than 90.

```
Select[Flatten[cleanedNewScores], GreaterThan[90]]
```

```
{94, 98, 95, 96, 99}
```

Using Data Manipulation Functions with **Manipulate**

So far in this chapter, the calculation times have been a fraction of a second. When working with very large datasets, calculation times can be much longer, depending on both the size of the datasets and the operations that are performed.

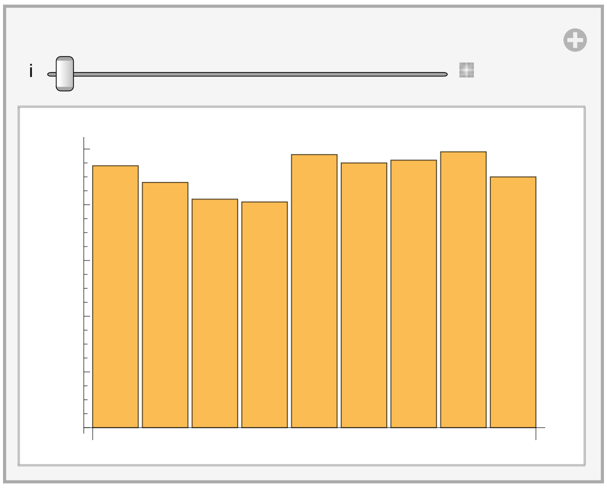
Manipulate uses dynamic variables and recalculates values as its controls, like slider bars, are manipulated. If one of those recalculations takes longer than a fraction of a second, the movement of the slider can feel sluggish, which will defeat the purpose of having an interactive model.

To eliminate this sluggishness, **Table** can be used to precompute a list of values for a calculation, which in turn can be viewed using **Manipulate**. The following example illustrates this. First, a new dataset consisting of only relevant scores is created. Then, **Table** creates a list of student scores above **score**, as **score** ranges from 70 to 100. A semicolon is used to suppress the output.

```
manipulateData = Cases[cleanedNewScores, {_String, _String, _Integer}]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 88}, {Bob, Example, 82},
  {New, Entry, 81}, {Michelle, Abacus, 98}, {Michael, Morrison, 95},
  {Kelvin, Mischo, 96}, {Cliff, Hastings, 99}, {Jill, New, 90}}
```

```
scoreList = Table[(
        selectFun4[val_] := val⟦3⟧ ≥ score;
        Select[manipulateData, selectFun4]),
      {score, 70, 100, 1}
    ];
Manipulate[BarChart[scoreList⟦i⟧⟦All, 3⟧], {i, 1, 30, 1}, SaveDefinitions → True]
```



After the list was generated, the **Manipulate** command was used with **Part** to simply flip through the list to see the outputs. Even if the **Table** command takes some time to evaluate, the **Manipulate** statement will be quick since it is only displaying the precomputed values.

In the definition for **scoreList**, parentheses are used to tell the **Table** command to execute both statements for each new value of **score** as the table is generated. The function definition for **selectFun4** does not produce an output, but it is redefined and used for filtering results with the **Select** statement as the variable **score** takes on new values.

Organizing Data with Associations

The Wolfram Language has a way to work with symbolically indexed lists, and these are referred to as associations. When labels are present in data, use of **Association** and **Dataset** makes data extraction easier than using general purpose functions like **Cases** and **Select**.

Association is basically a set of rules between keys and values in a dataset.

```
myAssoc = Association[{"Cliff" → 99, "Kelvin" → 96, "Michael" → 95}]
```

```
<| Cliff → 99, Kelvin → 96, Michael → 95 |>
```

Once an association is defined, values can be looked up using the relevant key. For example, the following command looks up the score associated with the key "Michael."

```
myAssoc["Michael"]
```

```
95
```

An association can also be entered using the **<||>** notation.

```
myAssoc2 = <|"Tasha" → "Florida", "Kathy" → "Arizona"|>
```

```
<| Tasha → Florida, Kathy → Arizona |>
```

```
myAssoc2["Tasha"]
```

```
Florida
```

There are Wolfram Language commands that help construct associations. First, though, since many variables have been defined in this chapter, **listOfScores** is now redefined to its original values.

```
listOfScores = Import["http://www.handsonstart.com/ExampleDataScores.txt", "Data"]
```

```
{{Joe, Smith, 94}, {Jane, Smith, 85},
  {Bob, Example, 82}, {Bill, Student, 83}, {Michelle, Abacus, 98}}
```

One command to help create associations is **AssociationThread**. This command takes a list of keys and a list of values and creates an association between them. For example, the following input will create an association between the first name of a student and their exam score.

```
scoreAssoc = AssociationThread[listOfScores⟦All, 1⟧, listOfScores⟦All, 3⟧]
```

```
<| Joe → 94, Jane → 85, Bob → 82, Bill → 83, Michelle → 98 |>
```

This provides a nice format to search for names and return the corresponding exam scores.

```
scoreAssoc["Bill"]
```

```
83
```

```
scoreAssoc["Michelle"]
```

```
98
```

Associations are so powerful because order really is not important; the value is found by providing the key without having to know the position of the key. In contrast, commands like **Select** require knowledge of the positions in order to effectively filter and find data.

> In the documentation, **Select** uses a convention called pure functions, which are represented by the symbol **#** in the Wolfram Language. Pure functions are not covered in any detail in this book, but they essentially act as instant function definitions that can be defined on the fly when coding. The examples in this book do not use pure functions in an effort to make the examples easier to learn and understand, but pure functions are extremely powerful and can be useful when writing compact code.

Besides looking up the specific value for a certain key, there are also ways to extract all the keys or values in an association. The **Keys** command gives a list of all the keys, and the **Values** command gives a list of all the values.

```
Sort[Values[scoreAssoc]]
```

```
{82, 83, 85, 94, 98}
```

```
Sort[Keys[scoreAssoc]]
```

```
{Bill, Bob, Jane, Joe, Michelle}
```

Dataset can be used to represent a structured dataset of lists and associations. This representation is especially useful with larger datasets containing dozens or hundreds of labels. **Dataset** prints its output in a special form to indicate the number of levels and elements in the dataset, as in the following example.

```
Dataset[scoreAssoc]
```

Joe	94
Jane	85
Bob	82
Bill	83
Michelle	98

Here, all the keys (the names) are printed as row labels, the values (the exam scores) are shown in another column and the number of levels and the number of elements in the dataset are also displayed.

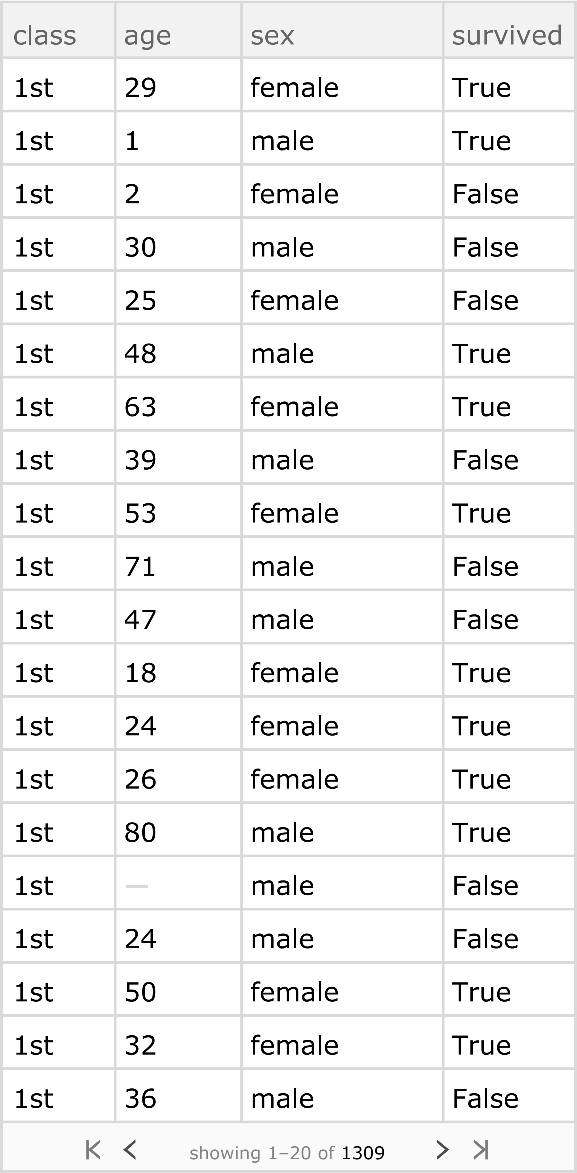
Some Wolfram Language commands return their results in dataset form. In those cases, it can be useful to remove the styling wrapper from the dataset representation to access the underlying data by itself, and the **Normal** command can be used to do that. The following command displays the underlying data for the dataset displayed in the preceding example.

```
Normal[Dataset[scoreAssoc]]
```

```
<| Joe → 94, Jane → 85, Bob → 82, Bill → 83, Michelle → 98 |>
```

Sample datasets can be found with the **ExampleData** command, like this dataset related to statistics for passengers on the *Titanic*.

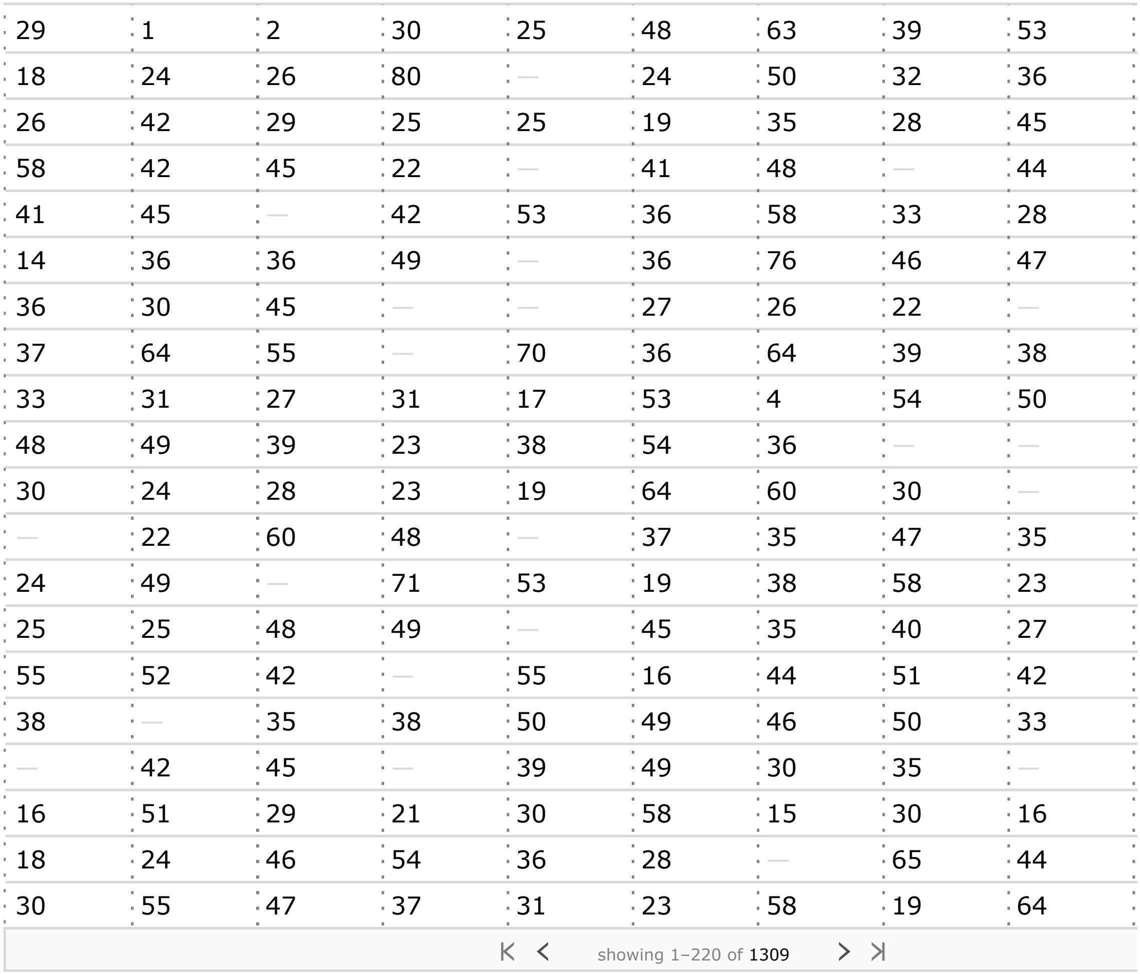
```
titanic = ExampleData[{"Dataset", "Titanic"}]
```



class	age	sex	survived
1st	29	female	True
1st	1	male	True
1st	2	female	False
1st	30	male	False
1st	25	female	False
1st	48	male	True
1st	63	female	True
1st	39	male	False
1st	53	female	True
1st	71	male	False
1st	47	male	False
1st	18	female	True
1st	24	female	True
1st	26	female	True
1st	80	male	True
1st	—	male	False
1st	24	male	False
1st	50	female	True
1st	32	female	True
1st	36	male	False

If a dataset has hundreds of elements with labels, extracting information with position-dependent commands like **Part** can be time consuming, since an intermediate step is required to first locate the position of the column of interest. The **Dataset** command, however, can use a key from an association to look up information without needing to know anything about the position of that information. The following example has syntax that may seem similar to **Part**, but a symbolic key value is used to extract the ages of each person in the dataset instead of extracting those values by referencing the position of that column.

```
titanic[All, "age"]
```

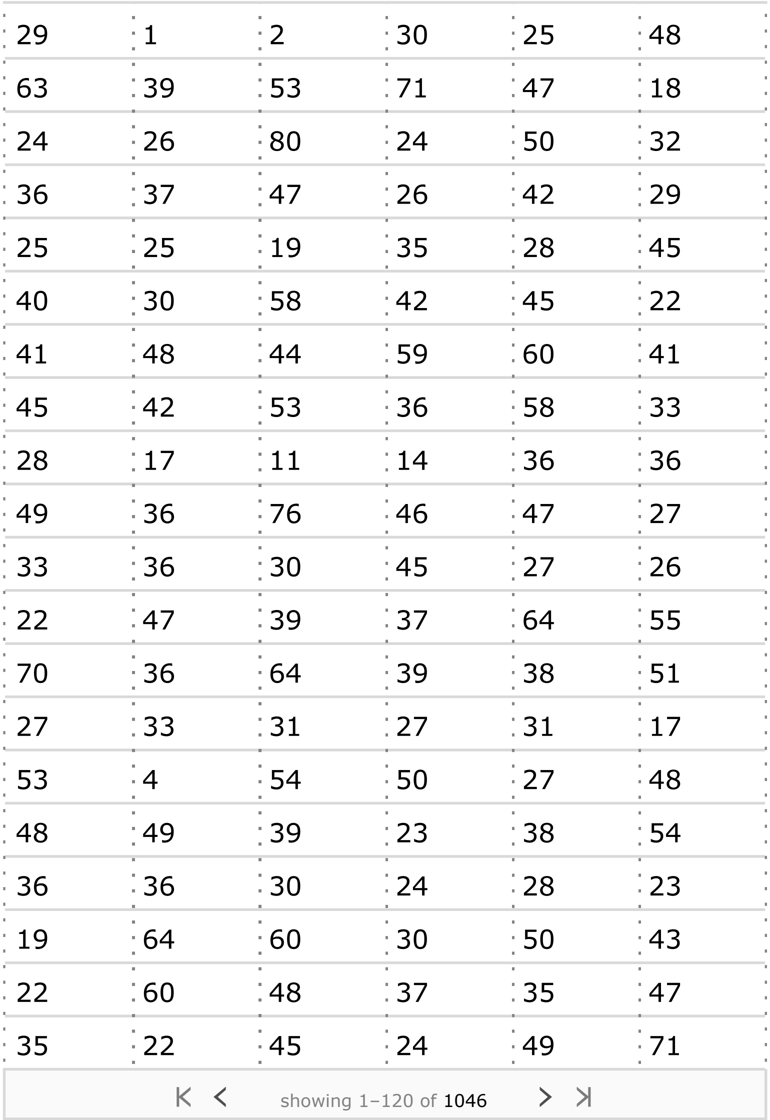


29	1	2	30	25	48	63	39	53
18	24	26	80	—	24	50	32	36
26	42	29	25	25	19	35	28	45
58	42	45	22	—	41	48	—	44
41	45	—	42	53	36	58	33	28
14	36	36	49	—	36	76	46	47
36	30	45	—	—	27	26	22	—
37	64	55	—	70	36	64	39	38
33	31	27	31	17	53	4	54	50
48	49	39	23	38	54	36	—	—
30	24	28	23	19	64	60	30	—
—	22	60	48	—	37	35	47	35
24	49	—	71	53	19	38	58	23
25	25	48	49	—	45	35	40	27
55	52	42	—	55	16	44	51	42
38	—	35	38	50	49	46	50	33
—	42	45	—	39	49	30	35	—
16	51	29	21	30	58	15	30	16
18	24	46	54	36	28	—	65	44
30	55	47	37	31	23	58	19	64

|< < showing 1–220 of 1309 > >|

The output shows that at least one of the values is missing for a person's age. Missing values can be eliminated immediately by using the **DeleteMissing** command.

```
DeleteMissing[titanic[All, "age"]]
```



```
29   1    2    30   25   48
63   39   53   71   47   18
24   26   80   24   50   32
36   37   47   26   42   29
25   25   19   35   28   45
40   30   58   42   45   22
41   48   44   59   60   41
45   42   53   36   58   33
28   17   11   14   36   36
49   36   76   46   47   27
33   36   30   45   27   26
22   47   39   37   64   55
70   36   64   39   38   51
27   33   31   27   31   17
53   4    54   50   27   48
48   49   39   23   38   54
36   36   30   24   28   23
19   64   60   30   50   43
22   60   48   37   35   47
35   22   45   24   49   71
```

Once values are extracted from an association, they can be used with other commands. Here, **Mean** is used to find the average age of passengers on the *Titanic*.

```
N[Mean[DeleteMissing[titanic[All, "age"]]]]
```

```
29.9006
```

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[listOfScores, newScores, cleanedNewScores, sortFun, selectFun, selectFun2,
  selectFun3, manipulateData, selectFun4, myAssoc, myAssoc2, scoreAssoc,
  titanic]
```

Conclusion

The Wolfram Language's symbolic representation of expressions makes Mathematica an excellent platform for data manipulation, cleaning and transformation. Structures like associations and datasets allow incredibly fast and useful ways to look up and display information imported from files or constructed programmatically in Mathematica itself.

The following chapters will take advantage of these data manipulation techniques while outlining ways to access Wolfram Knowledgebase data from within Mathematica.

Exercises

1. Create a list of the first 10 prime numbers raised to the power of the first 10 integers, so 2^1, 3^2 and so on. Assign this list to the variable **data25**.
2. Use the **Part** command to extract the element of **data25** that is closest to the number 100.
3. Write a two-statement program that uses the **Part** command to return the first, third and fourth elements of **data25** and suppress the output from this statement. For the second statement, calculate the total of the three elements.
4. Use the **[[ ]]** notation of **Part** to extract the second through sixth elements of **data25**.
5. Create a list of ordered triples of the form $\{i, i^i, i - i^i\}$, where i goes from 1 to 4. Assign this list to the variable **data29**.
6. Use the **[[ ]]** notation of **Part** to extract the second element of **data29**.
7. Use the **[[ ]]** notation of **Part** to extract the second element of the third ordered triple of **data29**.
8. Extract the second element of each of the ordered triples of **data29**, and assign that resulting list to the variable **data33**.
9. Replace the fourth element of **data33** with the square root of itself.
10. Create a bar chart to visualize the result from Exercise 9.

CHAPTER 21

Working with Curated Data

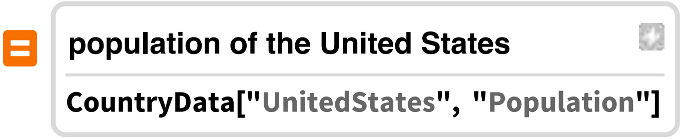
Introduction

The previous chapters outlined how to import external files, as well as strategies to reformat data into desired structures. Besides using data from external files, Mathematica can also access curated data from the Wolfram Knowledgebase, as long as an active internet connection is available. This data is designed to be delivered in a way that lends itself to immediate computation, eliminating much of the need for postprocessing that importing external files can necessitate. Accessing trusted Wolfram Knowledgebase data can eliminate the effort necessary to find a data source and ensure its accuracy, as well as the need to maintain costly subscriptions to expensive data feeds.

Accessing Curated Data

The collection of curated data in the Wolfram Knowledgebase encompasses more than ten trillion pieces of data across a variety of disciplines. The data is accessed through Wolfram Language commands and is returned in list format, making all the previously discussed data filtering and manipulation strategies applicable.

Curated data was introduced in **Chapter 3: Input and Output**, and it can be accessed through free-form input, allowing queries to be stated in plain English.



```
population of the United States
CountryData["UnitedStates", "Population"]
```

```
322 422 965 people
```

In this example, the free-form input is translated into precise Wolfram Language syntax, which uses a curated data function, **CountryData**, to retrieve information about the population of the United States.

Although free-form input is always great for open-ended exploration, using precise syntax with the curated data commands allows for more control when querying for specific pieces of information. This control lends itself well to programmatic generation of lists comprised of curated data, and examples of this will be shown later in this chapter.

Since the previous example exposed the name of the curated data command used to look up information about countries, it is very easy to use this command directly to find the population of a different country.

```
CountryData["Japan", "Population"]
```

```
126 225 259 people
```

Each curated data command has a function page in the documentation that lists the unit used for each property of the data.

Unlike free-form input, which strives to give the best possible result to an unstructured query along with related results that may be of interest, use of the curated data commands directly allows for a more comprehensive exploration of what specific data is available. Queries can be performed to find all the members of a dataset. For example, specifying **All** with the function **CountryData** lists all the countries available via that command. Here the **Short** function is used to abbreviate the list and save screen space.

```
Short[CountryData[All]]
```

```
{Afghanistan, Albania, Algeria, American Samoa, Andorra,
 Angola, Anguilla, Antigua and Barbuda, Argentina,
 Armenia, Aruba, Australia, Austria, Azerbaijan,
 Bahamas, ≪210≫, United Kingdom, United States,
 United States Virgin Islands, Uruguay, Uzbekistan, Vanuatu,
 Vatican City, Venezuela, Vietnam, Wallis and Futuna Islands,
 West Bank, Western Sahara, Yemen, Zambia, Zimbabwe}
```

Removing the **Short** command in the statement above will print the full list of countries to the screen so they can be viewed. If the list is impractical to print to the screen, Mathematica will print a partial list of results along with a warning message that the list is quite long, along with options to see more results. To see an example of this behavior, evaluate **ChemicalData[All]** in an input cell; Mathematica will only show a partial list of results instead of printing a list of the tens of thousands of chemicals that are available.

Along with seeing the list of entities for a curated dataset, queries can also be performed to learn what properties are available for the entities.

```
Short[CountryData["Properties"], 10]
```

```
{AdultPopulation, AgriculturalProducts, AgriculturalValueAdded, Airports,
  AlternateNames, AlternateStandardNames, AMRadioStations, AnnualBirths,
  ≪208≫, UNNumber, UnpavedAirportLengths, UnpavedAirports,
  UnpavedRoadLength, ValueAdded, WaterArea, WaterwayLength}
```

```
Length[CountryData["Properties"]]
```

```
223
```

Now that the elements and their associated properties are known, queries can be constructed to extract the specific data of interest, such as the number of cellular phones in use in Japan.

```
CountryData["Japan", "CellularPhones"]
```

1.10395×10^{8}

A couple of the previous examples can be combined to do a single calculation; this is possible since the curated data is returned in a form (in this case, numerical data) that allows it to be immediately used by other Mathematica commands.

```
CountryData["Japan", "CellularPhones"]
--------------------------------------
  CountryData["Japan", "Population"]
```

```
0.874587 per person
```

Remember, you can create a two-dimensional fraction in Mathematica by using the Ctrl+/ keyboard shortcut or by using the buttons in the Assistant palettes.

This ratio may be surprising (or it might not be!), but it may be interesting to perform this same computation for other countries, both larger and smaller, to see different values around the world.

```
CountryData["UnitedStates", "CellularPhones"]
---------------------------------------------
   CountryData["UnitedStates", "Population"]

  0.83896 per person
```

This type of exploration can be automated with commands like **Table**. The **Table** command has a syntactical form that lets an index iterate over a list of values. In most examples shown in this book, **Table** has been used for a numerical range, like creating a list of pairs of the form (x, x^2), where x goes from 1 to 10 in steps of 1.

```
Table[{x, x^2}, {x, 1, 10, 1}]

 {{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}, {6, 36}, {7, 49}, {8, 64}, {9, 81}, {10, 100}}
```

However, **Table** can also be used to iterate over a specific list of values. In the following example, a list of pairs of the form (x, x^2) is constructed, but only for the values in the list that is given for the iterator.

```
Table[{x, x^2}, {x, {2, 5, 8, 13}}]

 {{2, 4}, {5, 25}, {8, 64}, {13, 169}}
```

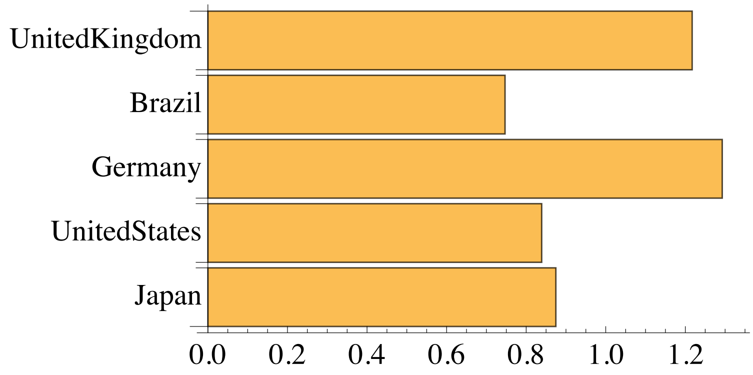
This use of **Table** can be very useful when paired with symbolic values, like those used for looking up curated data. The following example uses that approach to calculate the ratio of cellular phones per person for a list of five countries.

```
results = Table[{country, CountryData[country, "CellularPhones"] / CountryData[country, "Population"]},
   {country, {"Japan", "UnitedStates", "Germany", "Brazil", "UnitedKingdom"}}]
```

```
{{Japan, 0.874587 per person },
  {UnitedStates, 0.83896 per person }, {Germany, 1.29277 per person },
  {Brazil, 0.746857 per person }, {UnitedKingdom, 1.2172 per person }}
```

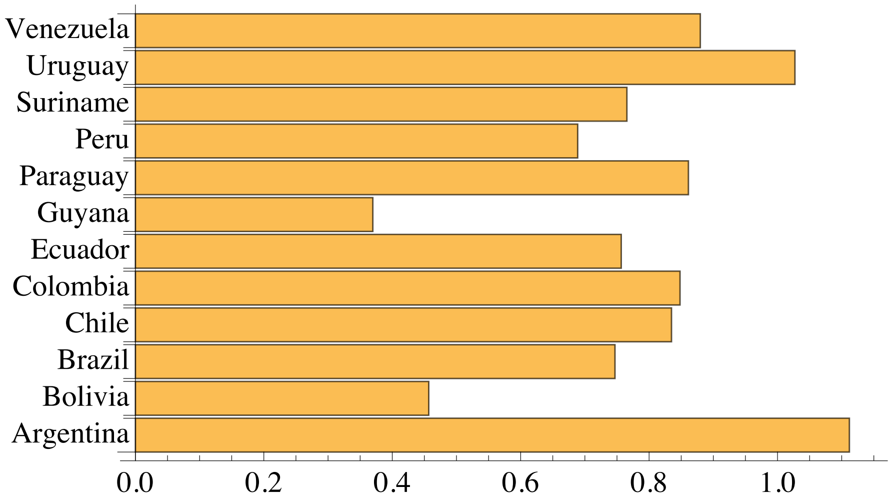
Now the results are plotted with **BarChart**. The **BarOrigin** option is used to place the bars on the left, allowing more space for the labels.

```
BarChart[results[[All, 2]], ChartLabels → results[[All, 1]], BarOrigin → Left]
```



Changing the list and using a few commands to extract values creates an example that can be used to explore cell phone usage among a group of countries, simply by changing the list of countries to the continent **"SouthAmerica"**. Use of the **Cases** command eliminates missing data, which minimizes the risk of errors in future calculations. In this case, all countries with a specific value for either cell phones or population with have a unit of "per person," which is represented with the function **Quantity**.

```
data = Cases[Table[{country, CountryData[country, "CellularPhones"] / CountryData[country, "Population"]},
        {country, CountryData["SouthAmerica"]}], {_, _Quantity}];
BarChart[data⟦All, 2⟧, ChartLabels → data⟦All, 1⟧, BarOrigin → Left]
```



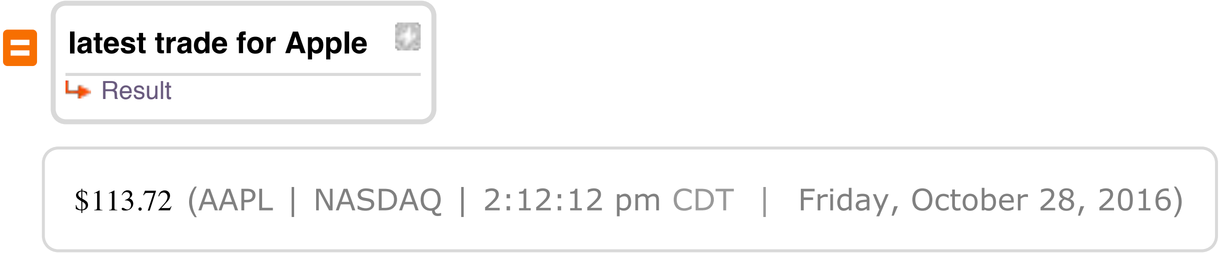
Major Categories of Curated Data

Curated data commands fall into major categories like computational finance data, mathematical data, geographic data, scientific data, technical data and linguistic data. Specific commands have names like **CountryData** and **ChemicalData**.

Remember, you can use the **?** operator to find command names, and the ***** symbol can be used for wild card searches. Evaluating **?*Data** will return a list of curated data commands.

Computational Finance Data

Many curated datasets provide data that does not change or does not change rapidly; some examples are historical information, which does not change, or the population of a country, which may only be assessed on a decennial basis or other schedule when a census is undertaken. Financial data, on the other hand, updates much more frequently.



latest trade for Apple

Result

$113.72 (AAPL | NASDAQ | 2:12:12 pm CDT | Friday, October 28, 2016)

The **FinancialData** command can be used to look up this same information by passing the ticker symbol for a particular stock.

```
FinancialData["AAPL"]
```

```
106.63
```

The output in the published version of this book represents the value at the time this chapter was written. The output will likely be different when this statement is evaluated in Mathematica at different times.

As before, the available properties for a financial entity can be explored.

```
Short[FinancialData["Properties"], 10]
```

```
{Ask, AskSize, Average200Day, Average50Day, AverageVolume3Month,
  Bid, BidSize, BookValuePerShare, Change, ≪56≫, SICCode,
  StandardName, Symbol, Volatility20Day, Volatility50Day,
  Volume, Website, YearEarningsEstimate, YearPERatioEstimate}
```

```
Length[FinancialData["Properties"]]
```

```
74
```

```
FinancialData["AAPL", "LatestTrade"]
```

```
{{2016, 9, 1, 13, 55, 0.}, 106.63}
```

Dates in Mathematica have a format of **{year, month, day, hour, minute, second}**. Mathematica will recognize the format **{year, month, day}** as well, if the last three specifications are not necessary.

FinancialData can be used to find historical data by giving a second argument to specify a date range. If a single date is given, **FinancialData** will return all information from that specified date to the current date. For example, the following command will find historical information related to the closing price of Apple's stock from August 1, 2016 to the present day.

```
FinancialData["AAPL", {2016, 8, 1}]
```

```
{{{2016, 8, 1}, 105.479}, {{2016, 8, 2}, 103.917}, {{2016, 8, 3}, 105.22}, {{2016, 8, 4}, 105.87},
  {{2016, 8, 5}, 107.48}, {{2016, 8, 8}, 108.37}, {{2016, 8, 9}, 108.81}, {{2016, 8, 10}, 108.},
  {{2016, 8, 11}, 107.93}, {{2016, 8, 12}, 108.18}, {{2016, 8, 15}, 109.48},
  {{2016, 8, 16}, 109.38}, {{2016, 8, 17}, 109.22}, {{2016, 8, 18}, 109.08},
  {{2016, 8, 19}, 109.36}, {{2016, 8, 22}, 108.51}, {{2016, 8, 23}, 108.85},
  {{2016, 8, 24}, 108.03}, {{2016, 8, 25}, 107.57}, {{2016, 8, 26}, 106.94},
  {{2016, 8, 29}, 106.82}, {{2016, 8, 30}, 106.}, {{2016, 8, 31}, 106.1}}
```

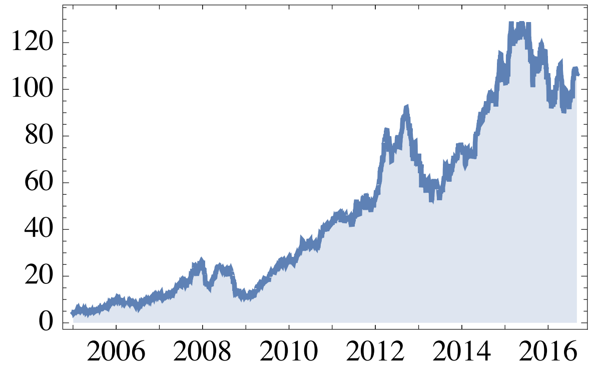
Two dates can be given to find closing stock price information between those dates.

```
FinancialData["AAPL", {{2010, 12, 15}, {2011, 1, 15}}]
```

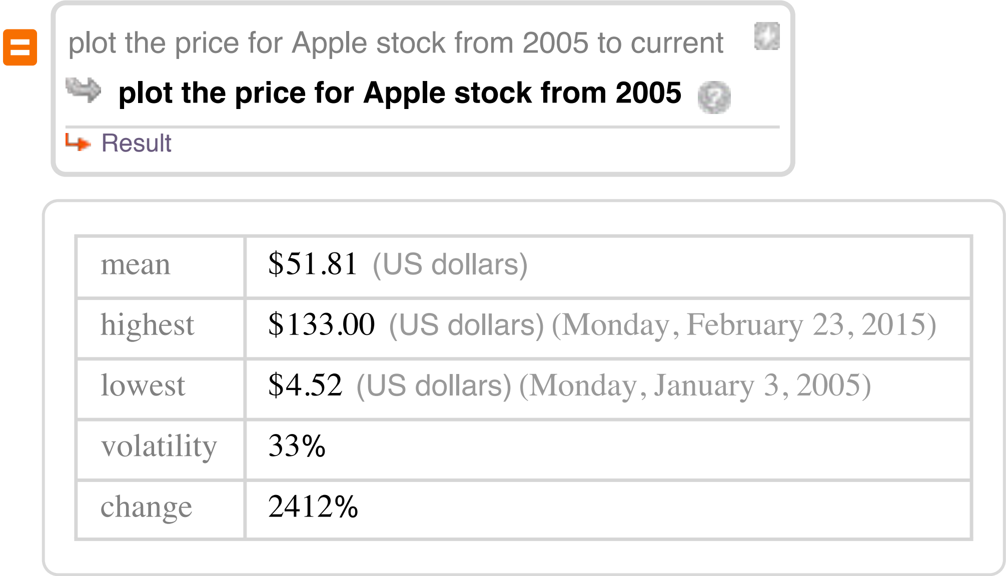
```
{{{2010, 12, 15}, 41.8997}, {{2010, 12, 16}, 42.0161},
  {{2010, 12, 17}, 41.9324}, {{2010, 12, 20}, 42.1416},
  {{2010, 12, 21}, 42.4019}, {{2010, 12, 22}, 42.5275}, {{2010, 12, 23}, 42.3234},
  {{2010, 12, 27}, 42.4647}, {{2010, 12, 28}, 42.568}, {{2010, 12, 29}, 42.5445},
  {{2010, 12, 30}, 42.3313}, {{2010, 12, 31}, 42.1874}, {{2011, 1, 3}, 43.1042},
  {{2011, 1, 4}, 43.3292}, {{2011, 1, 5}, 43.6836}, {{2011, 1, 6}, 43.6483},
  {{2011, 1, 7}, 43.9609}, {{2011, 1, 10}, 44.7888}, {{2011, 1, 11}, 44.6829},
  {{2011, 1, 12}, 45.0465}, {{2011, 1, 13}, 45.2113}, {{2011, 1, 14}, 45.5775}}
```

The data for stock performance is returned as a time series and can be used with **DateListPlot** for an immediate visualization. Some additional options are used to customize the plot.

```
DateListPlot[FinancialData["AAPL", {2005, 1, 1}], Joined → True, Filling → Bottom]
```



Free-form input does not always provide the desired output for multistep calculations or calculations that have extra specifications like the one below. For exact charts and views of the data, the Wolfram Language is the correct choice to ensure that the desired result is achieved. However, for curated datasets, the free-form input pods provide quite a bit of information and might display information that is interesting and unexpected.



plot the price for Apple stock from 2005 to current

plot the price for Apple stock from 2005

Result

mean	$51.81 (US dollars)
highest	$133.00 (US dollars) (Monday, February 23, 2015)
lowest	$4.52 (US dollars) (Monday, January 3, 2005)
volatility	33%
change	2412%

Of course, the data returned by **FinancialData** and other curated data commands can be reformatted using the same techniques as previously discussed. For example, a list of closing prices can be queried with **FinancialData** and then the values can be extracted to separate them from their corresponding dates.

```
FinancialData["AAPL", {{2010, 1, 1}, {2010, 1, 31}}]⟦All, 2⟧
```

```
{27.9902, 28.0386, 27.5926, 27.5416, 27.7247, 27.4801, 27.1676, 27.5508, 27.3912,
  26.9334, 28.1249, 27.692, 27.2133, 25.8636, 26.5594, 26.9348, 27.1885, 26.065, 25.1194}
```

Now that the values are extracted, the results can be used with other commands, like using **Mean** to calculate the average closing price for Apple's stock performance in January 2010.

```
Mean[FinancialData["AAPL", {{2010, 1, 1}, {2010, 1, 31}}]⟦All, 2⟧]
```

```
27.1669
```

The programmatic use of **Table** can be particularly useful in these types of calculations. For example, the following constructs a dataset of the year and the mean closing price for that year for Apple's stock from 2005 to 2011. The part of the **Table** calculation starting with **Mean** is identical to the calculation above; the only difference is the use of the iterator **year** to run the calculation multiple times for a range of years.

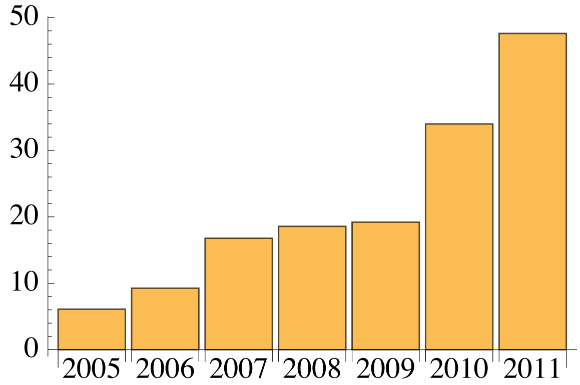
```
closingPrices = Table[
    {year, Mean[FinancialData["AAPL", {{year, 1, 1}, {year, 12, 31}}]⟦All, 2⟧]},
    {year, 2005, 2011, 1}]
```

```
{{2005, 6.10473}, {2006, 9.26128}, {2007, 16.7769},
  {2008, 18.5693}, {2009, 19.2017}, {2010, 33.9846}, {2011, 47.6079}}
```

When creating a calculation that wraps several functions around each other, it is often useful to separate arguments with new lines by pressing the Enter key. Mathematica will automatically indent pieces of code to indicate which command they belong to.

These prices can be plotted as a bar chart for another presentation of the data.

```
BarChart[closingPrices[[All, 2]], ChartLabels → closingPrices[[All, 1]]]
```

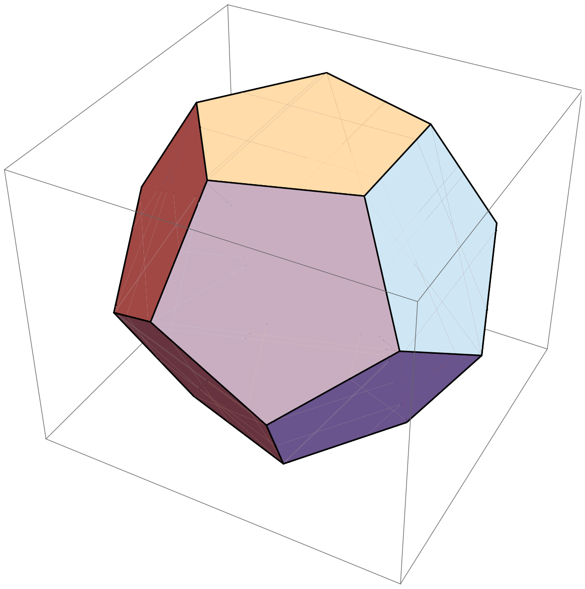


The **ChartLabels** option can be switched to **ChartLegends** to display the years to the right of the bar chart.

Mathematical Data

The examples thus far have focused on outputting numerical data, but other datasets provide graphical output. **PolyhedronData** is an example of a command that can be used to return graphics objects instead of pure data.

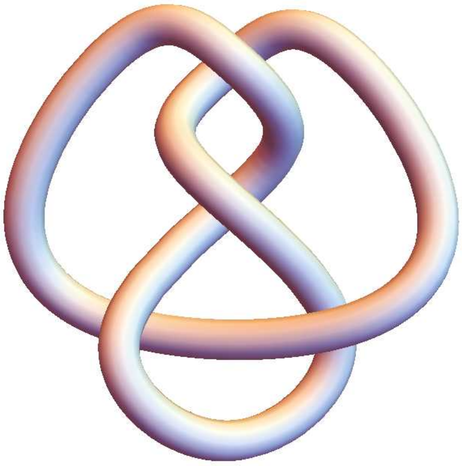
```
PolyhedronData["Dodecahedron"]
```



Note that this object is interactive like other 3D graphics and figures, allowing for rotation, zooming and panning.

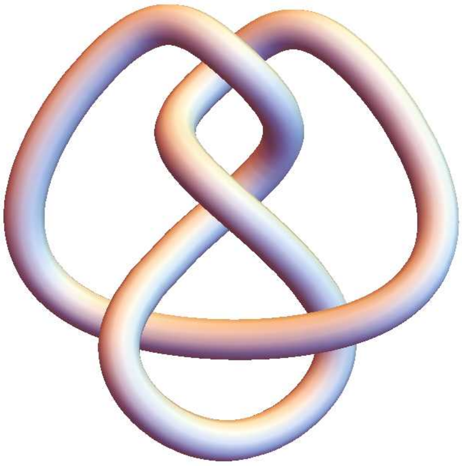
KnotData and **GraphData** are other curated data commands whose default behavior is to return images of the objects in question.

```
KnotData["FigureEight"]
```

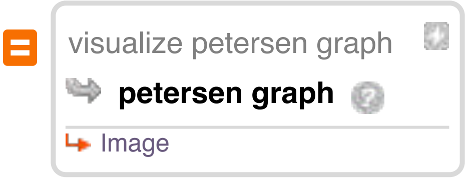


Just like the other curated datasets, free-form input works quite well for visualizing single properties of the data. The previous knot can be created with free-form input.

```
display a figure eight knot
KnotData["FigureEight", "Image"]
```



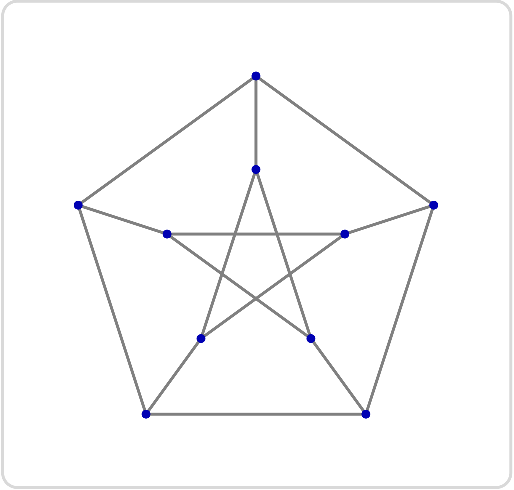
Once free-form input provides a function name, the proper Mathematica function can then be used to explore the scope of the dataset. For example, clicking the plus icon in the following example reveals that **GraphData** was used to create the graph.



visualize petersen graph

petersen graph

Image



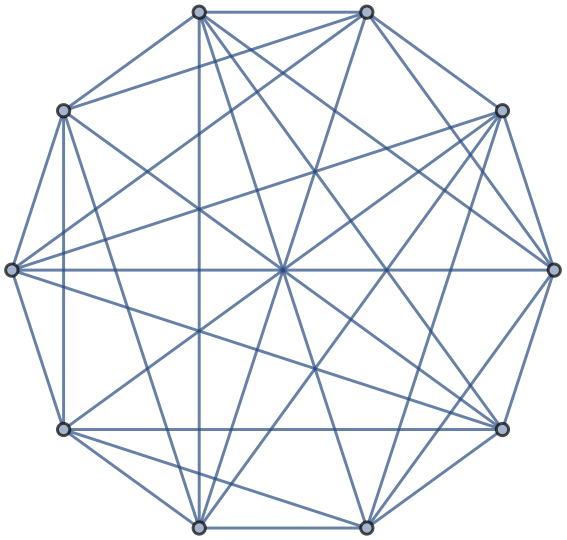
Other properties for these objects (polyhedra, knots and graphs) are available by querying the available classes, similarly to querying properties.

```
GraphData["PetersenGraph", "Classes"]
```

```
{AlmostHamiltonian, ArcTransitive, Biconnected, Bridgeless,
  Cage, ChromaticallyUnique, Class2, Connected, Cubic, Cyclic,
  DeterminedByResistance, DeterminedBySpectrum, DistanceRegular,
  DistanceTransitive, EdgeTransitive, GeneralizedPetersen, Hypohamiltonian,
  IGraph, Imperfect, Integral, Kneser, Local, MaximallyNonhamiltonian, Moore,
  Noncayley, Nonempty, Noneulerian, Nonhamiltonian, Nonplanar, Odd,
  PerfectMatching, Regular, Simple, Snark, SquareFree, StronglyRegular,
  Symmetric, Toroidal, Traceable, TriangleFree, UnitDistance, VertexTransitive}
```

Adding a second argument to the curated data command allows a different, nondefault property to be returned.

```
GraphData["PetersenGraph", "ComplementGraph"]
```



For certain free-form input calculations, there is not a corresponding Wolfram Language function to recreate the query to the dataset. **Chapter 22: Using Wolfram|Alpha Data in Mathematica** will outline how to use the **WolframAlpha** function to more effectively use the Wolfram Language with free-form input queries that do not have a corresponding function.

Geographic Data

CityData is a dataset similar to **CountryData** but is much larger, since it contains data on most cities in the world. Since there is more redundancy possible with this dataset—thanks to the existence of multiple cities with the same name—**CityData** commonly uses a list as the first argument to specify a city name along with a state or country to identify the intended element in the list.

CityData includes heuristics that help balance the need for exact syntax with ease of use. For example, when querying for the population of either London or Paris, Mathematica assumes that the user is interested in the most well-known instances of those city names, which are London, England, and Paris, France. By making the first argument a list, the user can specify a different city named Paris, such as Paris, Illinois.

```
CityData["London", "Population"]

 8 173 941 people
```

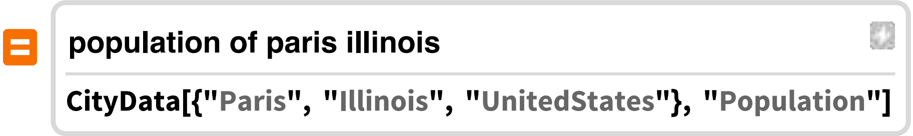
```
CityData["Paris", "Population"]

 2 233 818 people
```

```
CityData[{"Paris", "Illinois"}, "Population"]

 8498 people
```

Use of free-form input can reduce the likelihood that data for an unintended city will be returned. This is a case where free-form input is quite good at accepting additional specifications such as countries or states to differentiate between different bits of data.



```
population of paris illinois
CityData[{"Paris", "Illinois", "UnitedStates"}, "Population"]
```

```
 8498 people
```

CityData includes a variety of interesting properties, like latitude and longitude coordinates and elevation of cities.

```
CityData[{"Chicago", "Illinois", "UnitedStates"}, "Coordinates"]

{41.8376, –87.6818}
```

```
CityData[{"Chicago", "Illinois", "UnitedStates"}, "Elevation"]

 179 m
```

```
CityData[{"Boston", "Massachusetts", "UnitedStates"}, "Elevation"]

 14 m
```

The natural language processing in free-form input can also usually compare two pieces of information. For example, free-form input can be used to compare the elevation for two specific cities.

```
elevation of boston massachusetts versus the elevation of chicago
illinois
{CityData[{"Boston", "Massachusetts", "UnitedStates"}, "Elevation"],
  CityData[{"Chicago", "Illinois", "UnitedStates"}, "Elevation"]}
```

```
{14 m , 179 m}
```

The free-form input parser is very flexible, so it can accept things like "vs." instead of "versus," used in the preceding example.

Some of the data commands have arguments that allow a group of data to be queried. This can be used as a shortcut to fetch only data of interest instead of fetching more than is necessary and then using filtering techniques. An example is the use of **Large** with **CityData** to find all large cities in a region, with large cities being defined as having a population above 100,000.

```
CityData[{Large, "Illinois", "UnitedStates"}]
```

```
{ Chicago , Aurora , Rockford , Joliet ,
  Naperville , Springfield , Peoria , Elgin }
```

Using the approach in the preceding example might be easier than using a command like **Select** to filter the data, although using **Select** does give the user more fine-grained control. For example, **Select** can return all cities with a population above 50,000, or above 500,000, or above 712,684, or whatever number is important for your project.

Scientific and Technical Data

Commands like **ChemicalData**, **ElementData**, **GenomeData** and **ProteinData** provide a wealth of information without needing to interrupt a workflow to find and consult additional sources.

Many chemicals and corresponding properties are available.

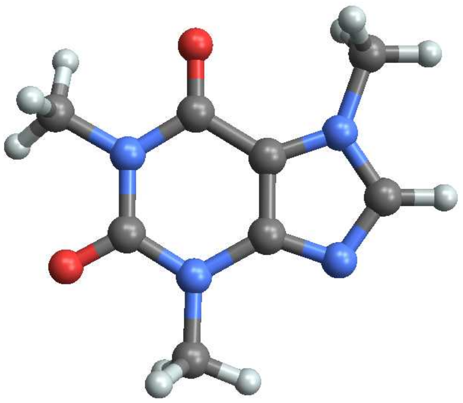
```
Short[ChemicalData[All]]
```

```
{ liquid hydrogen , hydrogen , deuterium hydride , liquid helium , helium ,
  deuterium , tritium deuteride , lithium , lithium hydride , lithium deuteride ,
  beryllium , boron , beryllium hydride , diamond , activated charcoal ,
  graphite , carbon-13 , ≪44056≫, 1-16:0-lysophosphatidylcholine ,
  1-18:0-lysophosphatidylcholine , 2-hydroxylaminobenzoate ,
  1-carboxymethylpyridinium , adenosylcobinamide-gdp ,
  1-18:1-lysophosphatidylcholine , 1-18:2-lysophosphatidylcholine ,
  ubiquinol-6 , o-sinapoylcholine , polymyxin b sulfate ,
  3-hexaprenyl-4-hydroxybenzoate , 1-16:0-2-18:1-phosphatidylcholine ,
  n-methylvaline , isowillardiine , 7-methylinosine , DL-glyceraldehyde dimer }
```

```
Length[ChemicalData["Properties"]]
```

```
95
```

```
ChemicalData["Caffeine", "MoleculePlot"]
```



ElementData can be used to look up information about elements and groups of elements.

```
ElementData["Gold", "MolarVolume"]
```

```
0.0000102 m^3/mol
```

```
ElementData["NobleGas"]
```

```
{ helium , neon , argon ,
  krypton , xenon , radon , ununoctium }
```

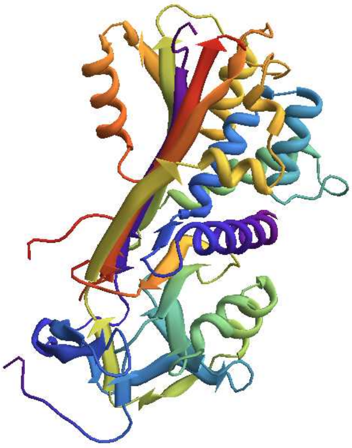
Much like exploring a complete listing of elements that are available in a curated dataset, the element groups that are available can be found by evaluating a similar command.

```
Short[ElementData["Groups"]]
```

```
{ actinides , alkali metals , alkaline earth metals ,
  antiferromagnetic elements , chalcogens , conductors ,
  diamagnetic elements , f-block elements , ferromagnetic elements ,
  gaseous elements , group 1 elements , group 10 elements ,
  group 11 elements , group 12 elements , group 13 elements ,
  group 14 elements , group 15 elements , group 16 elements ,
  ≪21≫, period 1 elements , period 2 elements ,
  period 3 elements , period 4 elements , period 5 elements ,
  period 6 elements , period 7 elements , poor metals ,
  radioactive elements , rare earth metals , s-block elements ,
  semiconductors , solid elements , stable elements ,
  super-heavy elements , synthetic elements , transition metals }
```

Other scientific datasets include **ProteinData**, which provides information for over 27,000 proteins, including their associated molecule plots and biological processes.

```
ProteinData["SERPINA3", "MoleculePlot"]
```



```
ProteinData["SERPINA3", "BiologicalProcesses"]
```

```
{AcutePhaseResponse, InflammatoryResponse, RegulationOfLipidMetabolicProcess}
```

GenomeData includes data on the human genome, from gene sequencing and location properties to functional properties. **GenomeLookup** in particular leverages Mathematica's efficient pattern-matching capabilities by using string patterns to find corresponding chromosomes where given strings occur.

```
GenomeData["ZXDB", "ProteinNames"]
```

```
{zinc finger, X–linked, duplicated B}
```

```
GenomeLookup["GATTACAGATTACAGATTA"]
```

```
{{{Chromosome2, 1}, {87 696 557, 87 696 575}},
  {{Chromosome2, –1}, {131 142 192, 131 142 210}},
  {{Chromosome4, –1}, {178 364 156, 178 364 174}}}
```

Linguistic Data

Curated data functions exist to look up words in 27 dictionaries, as well as properties of words and the relationships that may exist between them. Similarly to **GenomeLookup**, Mathematica's pattern-matching strengths can provide a lot of flexibility when looking for particular words, such as all English words that begin with the letter *e* and end with the letter *u*.

```
DictionaryLookup[{"English", "e" ~~ ___ ~~ "u"}]
```

```
{ecru, ecu, emu}
```

The statement above can be interpreted as a dictionary lookup in the English language for any word that begins with the letter *e* and ends with the letter *u*, with anything in between. Three underscore characters in a row represent any expression, which in this case means any string character(s). It is also possible to specify a pattern of looking for only one character between the starting and ending characters by using a single underscore character.

The French language consists of more words that match this pattern.

```
DictionaryLookup[{"French", "e" ~~ ___ ~~ "u"}]
```

```
{eau, encouru, enjeu, entendu, entr'aperçu, entretenu,
  entrevu, escabeau, esquimau, essieu, eu, exclu, exigu}
```

The function **WordTranslation** accepts a word in one language and attempts to translate it to another language. If multiple translations exist, they are returned in the form of a list.

```
WordTranslation["enjeu", "French" → "English"]
```

```
{stake, wager}
```

In addition to translating a word from one language to another, **WordTranslation** will accept **All** as an argument, allowing it to return translations for all available languages at the same time.

```
WordTranslation["enjeu", "French" → All]
```

```
⟨| English → {stake, wager},
   Spanish → {interés, apuesta, estaca, poste, apostar},
   Portuguese → {participação, parte, aposta, estaca, poste, apostar, arriscar},
   Japanese → {杙, 杭, 棒杙, 棒杭, 火刑, 賭す, 賭ける, 賭する, 利害関係},
   German → {Bedeutung, Anteil, Einsatz, Pfahl, Wette, wetten}, Italian →
    {interesse, quota, posta, palo, picchetto, scommessa, scommettere}, Dutch →
    {belang, aandeel, inzet, paal, staak, weddenschap, wedden, verwedden},
   Swedish → {intresse, andel, insats, stake, påle, stolpe, vad, slå vad},
   Latin → {talea} |⟩
```

A related function, **LanguageIdentify**, can be used to identify the language for a particular word or phrase.

```
LanguageIdentify["enjeu"]
```

```
French
```

Using Curated Data to Visualize Relationships

With a working knowledge of how the curated datasets work in Mathematica, it is possible to use the datasets to create visualizations and investigate relationships in groups of data. An introductory example in this chapter investigated the ratio of the number of cellular phones to the population of certain countries. The **Table** command can be used to construct a dataset of interest by having the iterator run through a list of symbolic values instead of numerical ones.

First, a list of all available countries is constructed.

```
listOfAllCountries = CountryData[All];
```

And now **Table** is used to construct a list of pairs consisting of the country name and its ratio of cellular phones to population.

```
                    CountryData[i, "CellularPhones"]
results = Table[{i, ────────────────────────────────}, {i, listOfAllCountries}];
                      CountryData[i, "Population"]
results⟦40 ;; 50⟧
```

```
{{ Chad , 0.145207 per person },
  { Chile , 0.83491 per person }, { China , 0.401009 per person },

                                         1
  { Christmas Island , Missing[NotAvailable]( ──── per person )},
                                        1513

                                             1
  { Cocos Keeling Islands , Missing[NotAvailable]( ─── per person )},
                                            550

  { Colombia , 0.848155 per person }, { Comoros , 0.0516854 per person },
  { Cook Islands , 0.661141 per person }, { Costa Rica , 0.383169 per person },
  { Croatia , 1.35562 per person }, { Cuba , 0.0295149 per person }}
```

Some countries do not have complete information, and these cases are easily seen, thanks to the **Missing[NotAvailable]** statement returned in the preceding result, which shows a subset of the full result. (The **Part** command is used here to highlight the missing values rather than **Short**, which only prints a sample of the dataset, and which may not indicate that some values are missing.)

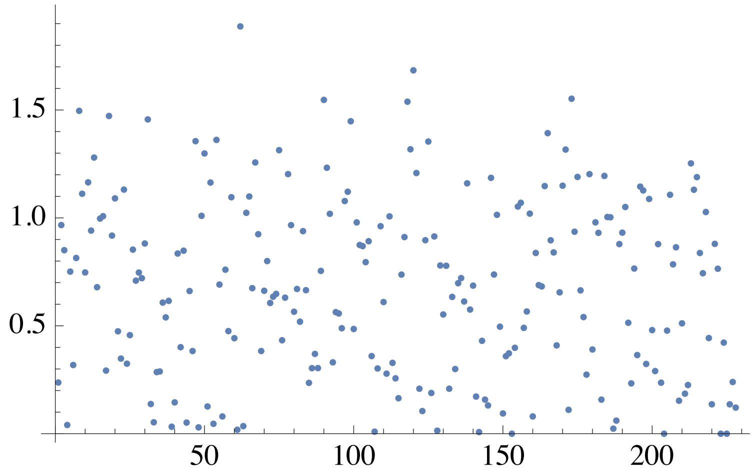
For example, **Missing[NotAvailable]** is a useful way to know when a certain country has not reported the quantity of cellular phone users and should be excluded from any statistical calculations. The data can be filtered to remove these cases by looking for a pattern of two elements, the second of which is a real number with the head **Quantity**; this will remove all cases where the second element is **Missing[NotAvailable]**.

```
results = Cases[Table[{i, CountryData[i, "CellularPhones"] / CountryData[i, "Population"]}, {i, listOfAllCountries}],
      {_, _Quantity}];
results⟦40 ;; 50⟧
```

```
{{Chad, 0.145207 per person}, {Chile, 0.83491 per person},
  {China, 0.401009 per person}, {Colombia, 0.848155 per person},
  {Comoros, 0.0516854 per person}, {Cook Islands, 0.661141 per person},
  {Costa Rica, 0.383169 per person}, {Croatia, 1.35562 per person},
  {Cuba, 0.0295149 per person}, {Cyprus, 1.00963 per person},
  {Czech Republic, 1.29862 per person}}
```

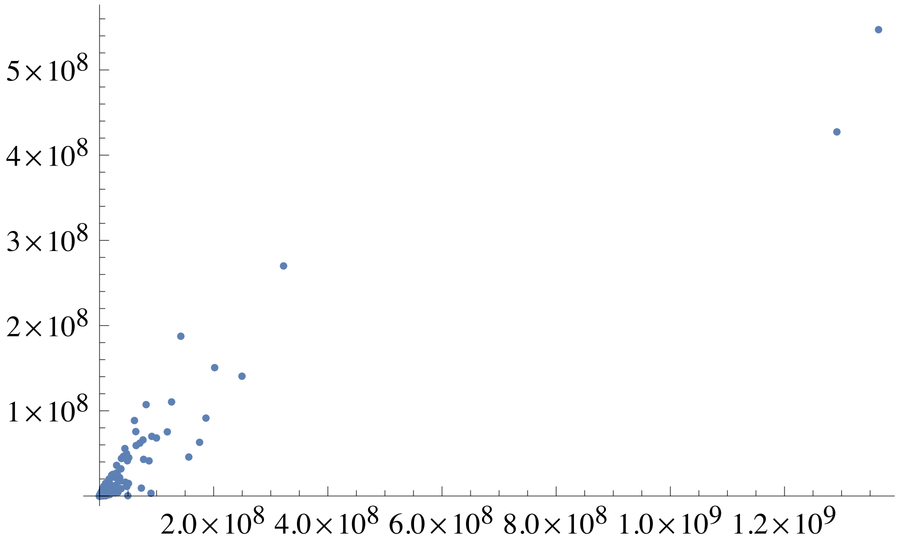
With the dataset constructed, it can now be used with other Mathematica commands, such as **ListPlot**, which visualizes the ratio as a data plot.

```
ListPlot[results⟦All, 2⟧]
```



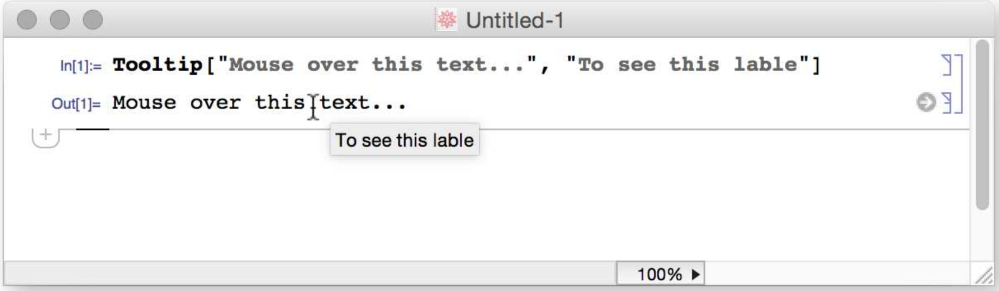
Without an immediate visual correlation between cellular phone prevalence and population, perhaps it would be more interesting to plot the actual population versus the number of cellular phones.

```
popVsPhones = Table[{CountryData[i, "Population"], CountryData[i, "CellularPhones"]},
        {i, listOfAllCountries}];
ListPlot[popVsPhones, PlotRange → All, ImageSize → 300]
```



ListPlot ignores non-numeric values, so while the data could be filtered the same way as it was previously in order to ignore missing values, it is not necessary to do so in this case.

This new plot makes it much easier to view outliers for the dataset, but it would be better if the outliers could be immediately identified. Luckily, there is a command called **Tooltip** that can help with this. **Tooltip** takes two arguments: an expression and a label to display as a popup window when the mouse pointer is over the expression. An example of **Tooltip** is shown in the following screen shot.

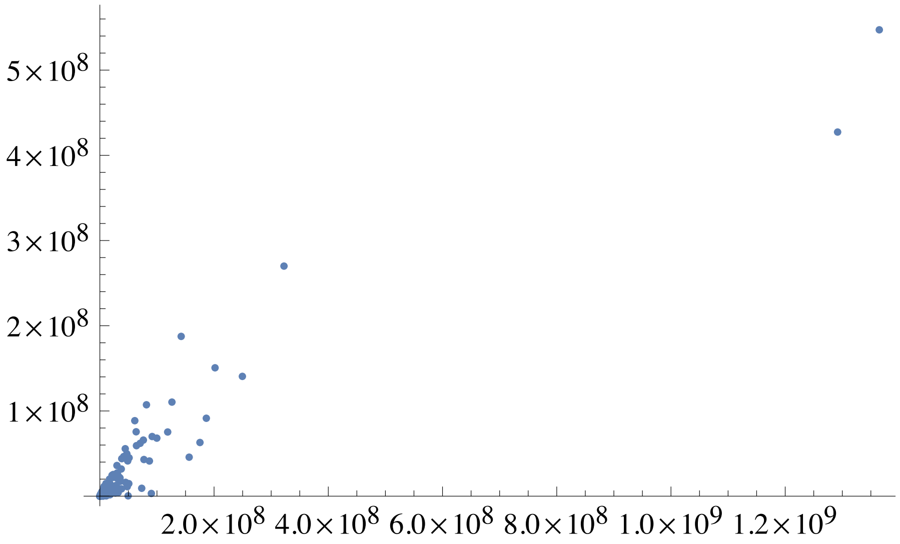


```
In[1]:= Tooltip["Mouse over this text...", "To see this lable"]
Out[1]= Mouse over this text...
```

To see this lable

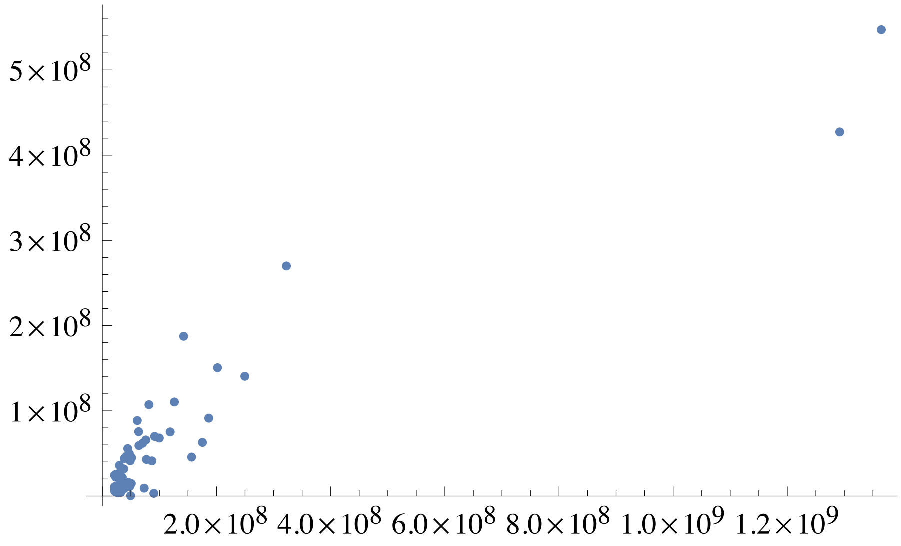
Thanks to the design of the Wolfram Language, is it very simple to wrap **Tooltip** around a set of data, such as the argument passed to **ListPlot** in the following example.

```
popVsPhones = Table[
        Tooltip[{CountryData[i, "Population"], CountryData[i, "CellularPhones"]}, i],
        {i, listOfAllCountries}];
ListPlot[popVsPhones, PlotRange → All, ImageSize → 300]
```



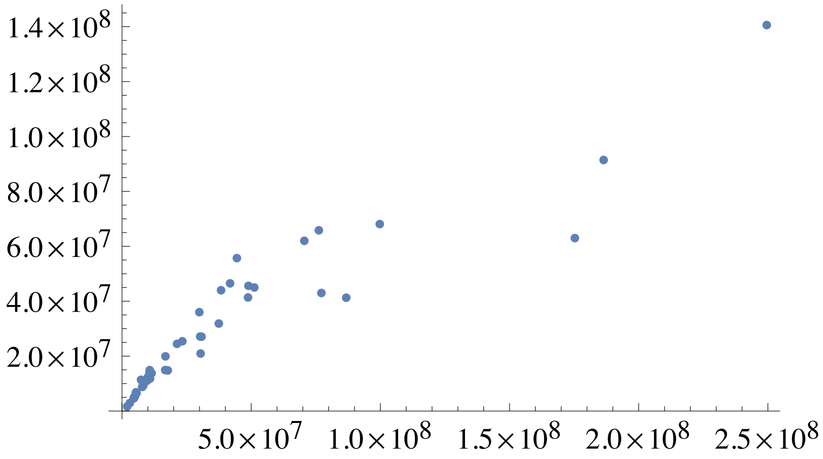
With so much clustering at lower populations, it can be hard to examine the relationship between population and prevalence of cellular phones. The data can be filtered to consider only countries with populations greater than 20,000,000 in order to examine the data more effectively. **QuantityMagnitude** is a function that eliminates units so that a direct comparison can be made with the inequality that is introduced as part of the **filter** command.

```
filter[x_] := QuantityMagnitude[CountryData[x, "Population"]] > 2*10^7
listOfCountries = Select[CountryData[All], filter];
results = Table[
        Tooltip[{CountryData[i, "Population"], CountryData[i, "CellularPhones"]}, i],
        {i, listOfCountries}];
ListPlot[results, PlotRange → All, ImageSize → 300]
```



The data filtering function can be changed to examine a narrower or wider scope of countries. For example, filtering the data so that only countries with a GDP between 10^{11} and 10^{12} US dollars per year are displayed seems to yield a stronger correlation between population and cellular phone prevalence.

```
filter[x_] := 10^11 ≤ QuantityMagnitude[CountryData[x, "GDP"]] ≤ 10^12
listOfCountries = Select[CountryData[All], filter];
results = Table[
        Tooltip[{CountryData[i, "Population"], CountryData[i, "CellularPhones"]}, i],
        {i, listOfCountries}];
ListPlot[results, PlotRange → All, ImageSize → 300]
```



Creating Tables with Curated Data

Mathematica's **TraditionalForm** displays multidimensional lists in a two-dimensional layout automatically, which is usually the desired output.

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

```
⎛  {1, 2, 3}      {4, 5, 6}   ⎞
⎜  {7, 8, 9}    {10, 11, 12}  ⎟
⎝{13, 14, 15}  {16, 17, 18}   ⎠
```

TableForm can be used to visualize the rows and columns of a dataset, but **TraditionalForm** is a better way to visualize lists with more of a nested structure.

In cases where more deliberate formatting is needed, functions like **Row**, **Column** and **Grid** can be used to create output according to specific layout directives. These commands accept arbitrary expressions such as data, strings, lists and graphics.

The **Row** command takes a list of arguments and displays them as a single row.

```
Row[{"item 1", "item 2", "item 3"}]
```

```
item 1item 2item 3
```

Items are displayed directly next to one another with no spaces in between them unless other instructions are given. This behavior can be adjusted through the introduction of explicit spacing, and the **Spacer** command provides an easy way to do this.

```
Row[{"item 1", Spacer[20], "item 2", Spacer[5], "item 3"}]
```

```
item 1      item 2  item 3
```

If the same amount of spacing is desired between elements, the **Spacer** command can be given as a second argument to **Row** instead of needing to be placed between each pair of elements.

```
Row[{"item 1", "item 2", "item 3"}, Spacer[10]]
```

```
item 1    item 2    item 3
```

The **Column** command works similarly by taking a single list and displaying the list as a column, with each element of the list in its own row.

```
Column[{"item 1", "item 2", "item 3"}]
```

```
item 1
item 2
item 3
```

Since each element is placed in a separate row, it is usually not necessary to adjust the amount of spacing for purposes of readability, like it is for **Row**. Nonetheless, **Column** also accepts an optional argument to adjust the spacing between elements.

```
Column[{"item 1", "item 2", "item 3"}, Spacings → 3]
```

```
item 1


item 2


item 3
```

For situations where a multidimensional layout is required, **Grid** can be used. **Grid** takes a nested list as its argument, with each sublist corresponding to a row, and each element in the sublist corresponding to a column. When constructing a complex grid layout, it can be good practice to arrange these arguments in the manner that they will be displayed, in order to help keep track of what is going where.

```
Grid[{
    {"item 1", "item 2"},
    {"item 3", "item 4"}
  }]
```

```
item 1  item 2
item 3  item 4
```

There are many options for **Grid** that can be explored in the documentation. A very common option is **Frame**, which draws a frame around each element in the grid.

```
Grid[{
    {"item 1", "item 2"},
    {"item 3", "item 4"}
  }, Frame → All]
```

```
┌────────┬────────┐
│ item 1 │ item 2 │
├────────┼────────┤
│ item 3 │ item 4 │
└────────┴────────┘
```

The option **Frame → All** will work with **Grid**, **Row** and **Column**.

While **Grid** is a general-purpose function that can be used for arranging all manner of content, **TextGrid** is specifically designed for arranging strings, with default line-wrapping and formatting options that are helpful with laying out text.

```
TextGrid[{
    {"item 1", "item 2"},
    {"item 3", "item 4"}
  }, Frame → All]
```

```
┌────────┬────────┐
│ item 1 │ item 2 │
├────────┼────────┤
│ item 3 │ item 4 │
└────────┴────────┘
```

When constructing a table of values, **Grid** can be especially useful for creating a nicely formatted presentation of these values. Once the table is constructed, table headings can be prepended to display as the first row, and **TraditionalForm** can be used as a final wrapper to generate a more aesthetically pleasing display.

```
f[x_] := x^2
values = Table[{i, f[i]}, {i, 1, 10, 1}];
PrependTo[values, {"x", "x^2"}];
TraditionalForm[Grid[values, Frame → All]]
```

```
┌────┬─────┐
│ x  │ x^2 │
├────┼─────┤
│ 1  │ 1   │
│ 2  │ 4   │
│ 3  │ 9   │
│ 4  │ 16  │
│ 5  │ 25  │
│ 6  │ 36  │
│ 7  │ 49  │
│ 8  │ 64  │
│ 9  │ 81  │
│ 10 │ 100 │
└────┴─────┘
```

When creating a visualization or application, it is often useful to have a mix of numeric values, graphics and strings to act as labels or to provide additional information about the data being presented. In the following example, the **Row** command is used to create an output with a user-defined label and a corresponding numerical value queried from **ChemicalData**.

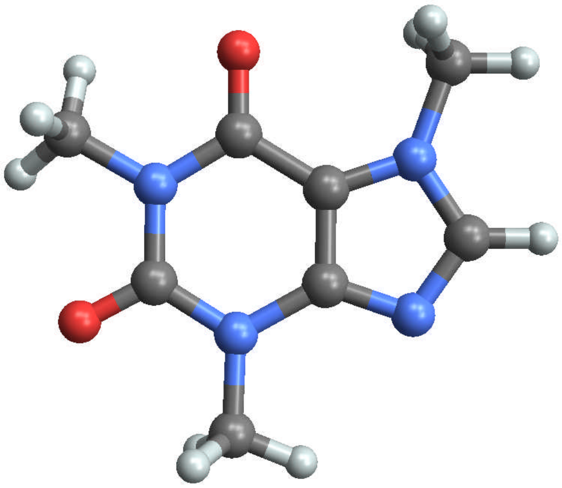
```
Row[{"Molecular weight:", Spacer[10],
   ChemicalData["Caffeine", "MolecularMass"]}]
```

```
Molecular weight:    194.191 u
```

The functions **Row** and **Column** can be used together to create four columns that are a mix of a chemical structure, a single chemical name or a row with a mix of text and data.

```
TraditionalForm[Column[{
      ChemicalData["Caffeine", "Name"],
      Row[{"Molecular weight:", Spacer[10],
           ChemicalData["Caffeine", "MolecularWeight"], Spacer[5],
           ChemicalData["Caffeine", "MolecularWeight", "Units"]}],
      Row[{"Boiling Point:", Spacer[10],
           ChemicalData["Caffeine", "BoilingPoint"], Spacer[5],
           ChemicalData["Caffeine", "BoilingPoint", "Units"]
        }],
      ChemicalData["Caffeine", "MoleculePlot"]
   }, Frame → All]]
```

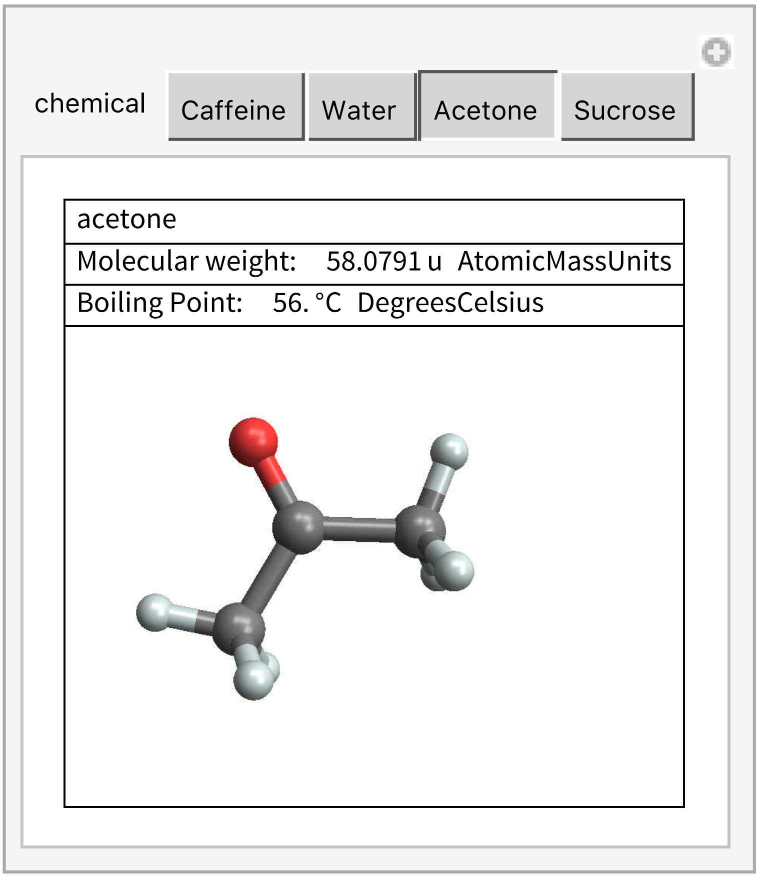
```
caffeine
Molecular weight:   194.191 u  AtomicMassUnits
Boiling Point:   —  DegreesCelsius
```



The use of **TraditionalForm** above displays missing data with dashes, which is the case above for the boiling point of caffeine.

Like other examples shown so far, **Manipulate** can be used to make this model more interesting.

```
Manipulate[
  TraditionalForm[
    Column[{
        ChemicalData[chemical, "Name"],
        Row[{"Molecular weight:", Spacer[10],
            ChemicalData[chemical, "MolecularWeight"], Spacer[5],
            ChemicalData[chemical, "MolecularWeight", "Units"]}],
        Row[{"Boiling Point:", Spacer[10],
            ChemicalData[chemical, "BoilingPoint"], Spacer[5],
            ChemicalData[chemical, "BoilingPoint", "Units"]
          }],
        ChemicalData[chemical, "MoleculePlot"]
      }, Frame → All]],
  {chemical, {"Caffeine", "Water", "Acetone", "Sucrose"}}
]
```



acetone

Molecular weight: 58.0791 u AtomicMassUnits

Boiling Point: 56. °C DegreesCelsius

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[results, data, closingPrices, listOfAllCountries, popVsPhones, filter, f, values]
```

Conclusion

This chapter outlined a general process to access the Wolfram Knowledgebase, which contains trillions of pieces of curated data. Data can be accessed by using free-form input along with specific Wolfram Language commands. Data is delivered in a form suitable for instant computation and can be used for creating visualizations and building programs.

The next chapter will expand on this idea by introducing the use of the **WolframAlpha** command to access even more curated data and specialized functionality.

Exercises

1. Use free-form input to find the molecular weight obtained by adding hydrochloric acid and nitric acid.
2. Use free-form input to find the highest elevation in the United States minus the highest elevation in Canada.
3. Use free-form input to find the temperature in the Windy City at 9am on February 25, 1989.
4. Use free-form input to find the nutritional value of eating a chicken sandwich and drinking a soda.
5. Use the Wolfram Language to find the boiling point of the chemical benzene.
6. Use the Wolfram Language to write a two-statement program, where you first create a variable **bp** that is used to store the boiling points of benzene, ethanol, chloroform, diethylamine and pentane. Suppress the output from the first statement. Use the second statement to sort the boiling points from lowest to highest.
7. Use the Wolfram Language to create a date list plot of the closing stock prices for Microsoft (stock ticker symbol: MSFT), starting from January 1, 2000, and add the necessary option to join the points into a single line and add filling to the bottom.
8. Build on the statement from Exercise 7 to create an interactive model that allows stocks to be chosen from Southwest Airlines (LUV), Delta Air Lines, Inc. (DAL), United Airlines, Inc. (UAL) and American Airlines, Inc. (AAL). Add an option to set the plot range to be from 0 to 80, to make it easy to quickly compare the performance of all four stocks.

9. Use **Manipulate** and **CountryData** to create an interactive model that allows a user to choose a country in Asia to see its corresponding flag.

10. Use **GenomeLookup** to find the chromosome in which the gene sequence "GGGTATAGGGTATAGAT" appears.

CHAPTER 22

Using Wolfram|Alpha Data in Mathematica

Introduction

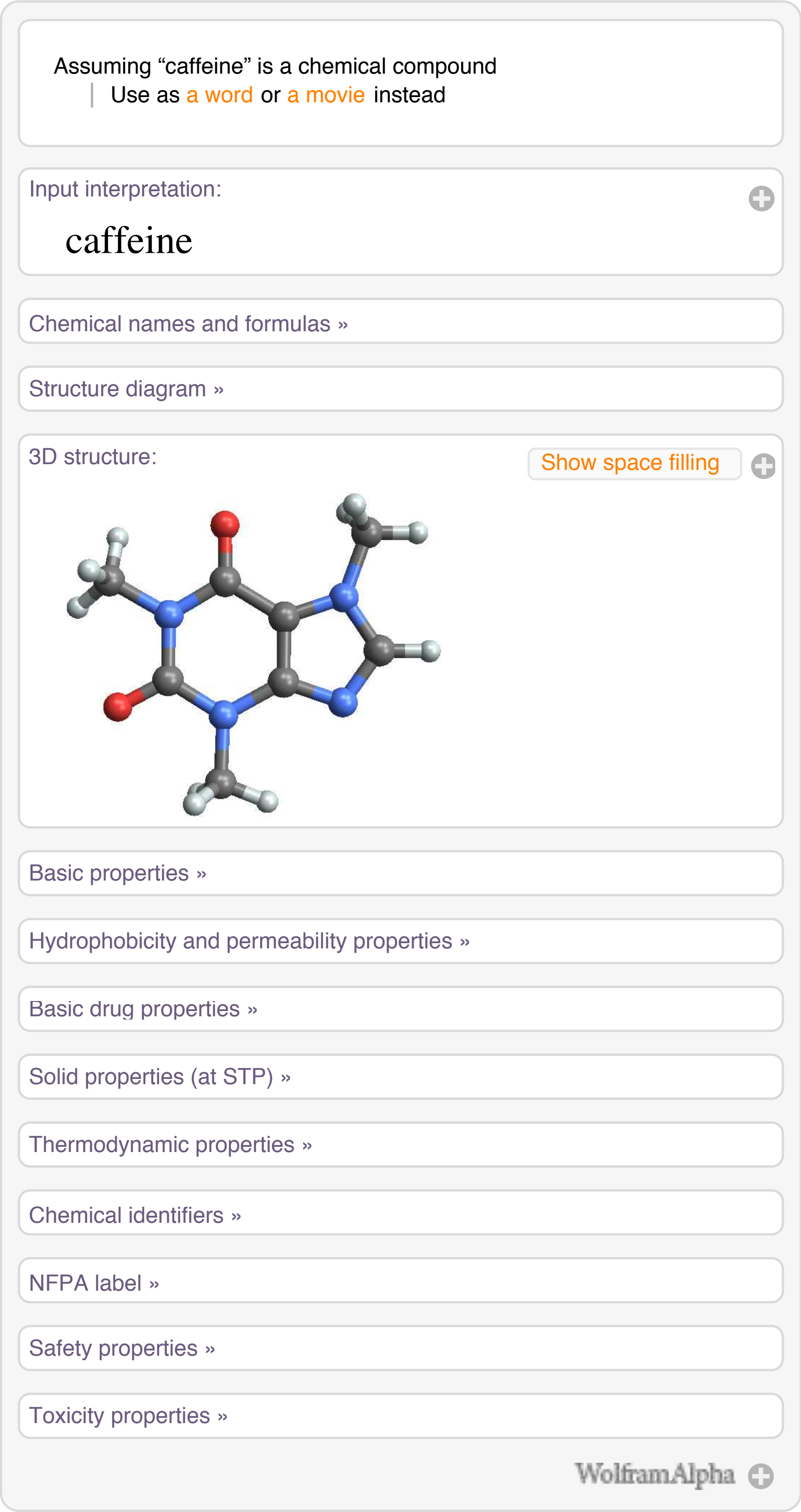
Previous chapters have outlined how data can be queried from the Wolfram Knowledgebase by using free-form input, and how formal curated data commands can be used to extract precise results from the vast amount of available data. A third method can be used to query data, and that is by using the **WolframAlpha** command.

The **WolframAlpha** command can be used to access data when a formal curated data command does not yet exist, and it also delivers data in a structured form that can be used for immediate computation. Like free-form input and the named curated data commands, the **WolframAlpha** command requires internet connectivity in order to work.

The **WolframAlpha** Command

The simplest syntax for the **WolframAlpha** function takes a single argument as a string, which represents the desired query. The results returned by the **WolframAlpha** function are the same as they would appear on the Wolfram|Alpha website. As such, if a general query is given, the result may include a lot of information. A specific query may return a single result, and a variety of related results may be displayed as well. The following is an example of a general query.

```
WolframAlpha["caffeine"]
```



Assuming “caffeine” is a chemical compound
| Use as a word or a movie instead

Input interpretation:

caffeine

Chemical names and formulas »

Structure diagram »

3D structure:

Show space filling

Basic properties »

Hydrophobicity and permeability properties »

Basic drug properties »

Solid properties (at STP) »

Thermodynamic properties »

Chemical identifiers »

NFPA label »

Safety properties »

Toxicity properties »

A second argument can be given to the command to control the desired form of the output or to get more specific information. For example, by learning what pods are returned for a specific query by passing **"PodIDs"** as an argument, the data from a specific pod can be examined.

```
WolframAlpha["caffeine", "PodIDs"]
```

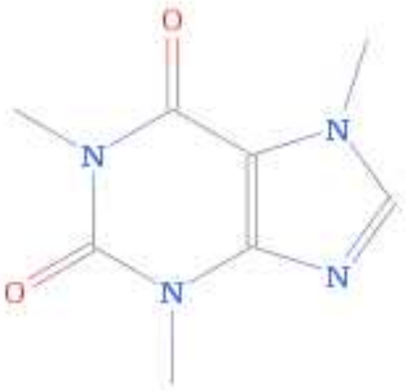
```
{Input, ChemicalNamesFormulas:ChemicalData,
  StructureDiagramPod:ChemicalData, 3DStructure:ChemicalData,
  Basic:ChemicalData, HydrophobicityPermeabilityProperties:ChemicalData,
  DrugNamesProperties:ChemicalData,
  SolidProperties:ChemicalData, Thermodynamics:ChemicalData,
  ChemicalIdentifiers:ChemicalData, NFPALabel:ChemicalData,
  SafetyProperties:ChemicalData, ToxicityProperties:ChemicalData}
```

```
WolframAlpha["caffeine", {"StructureDiagramPod:ChemicalData"}]
```

```
{{{StructureDiagramPod:ChemicalData, 0}, Title},
  {{StructureDiagramPod:ChemicalData, 0}, Scanner},
  {{StructureDiagramPod:ChemicalData, 0}, ID},
  {{StructureDiagramPod:ChemicalData, 0}, Position},
  {{StructureDiagramPod:ChemicalData, 1}, Cell},
  {{StructureDiagramPod:ChemicalData, 1}, Content},
  {{StructureDiagramPod:ChemicalData, 1}, Image},
  {{StructureDiagramPod:ChemicalData, 1}, DataFormats}}
```

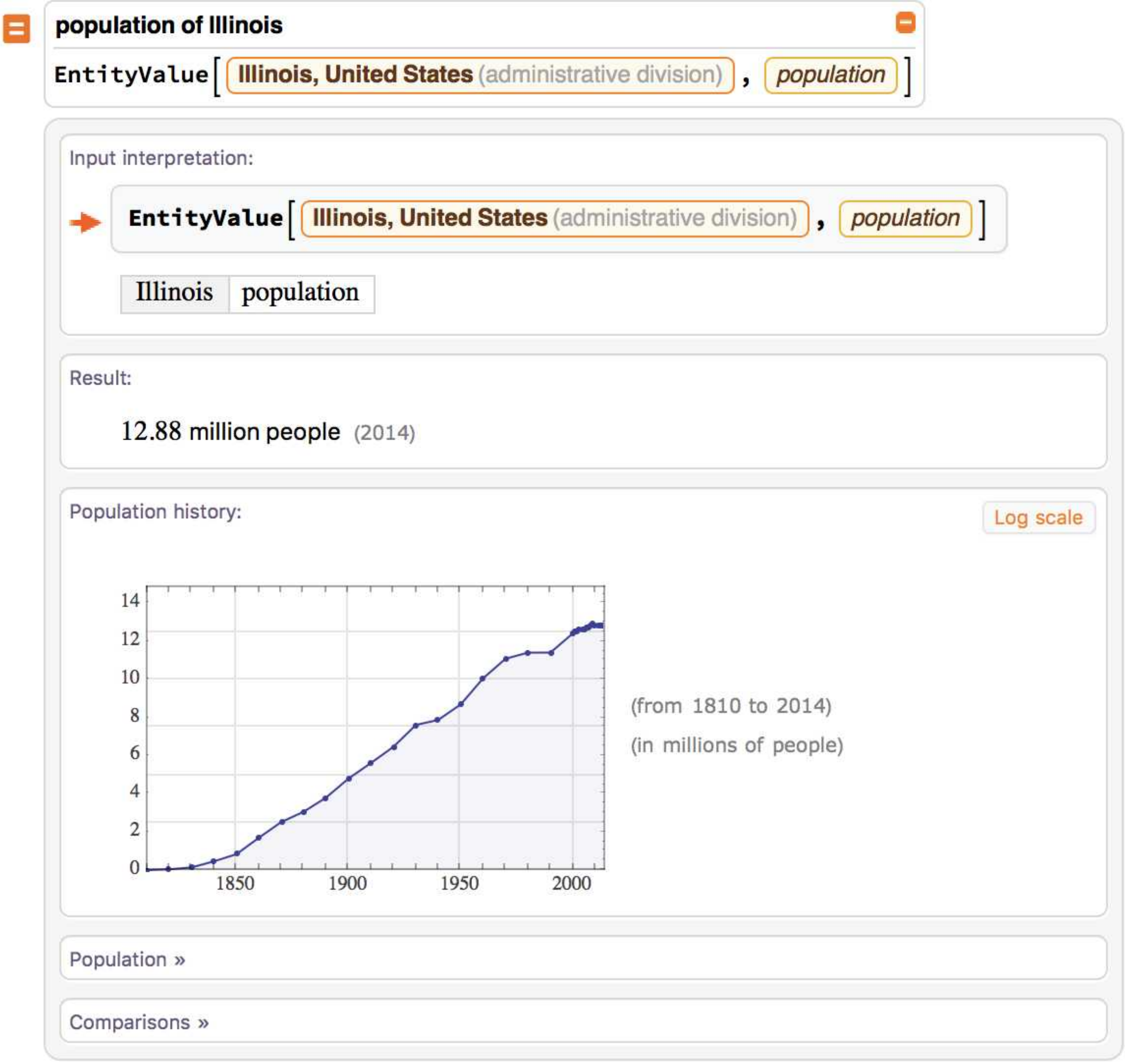
Once the lowest level is reached, a specific piece of data from a specific pod can be retrieved.

```
WolframAlpha["caffeine", {{"StructureDiagramPod:ChemicalData", 1}, "Image"}]
```



> When using formal curated data commands like **ChemicalData**, the function arguments require capitalization. The **WolframAlpha** command, however, accepts free-form input, so it can accept arguments like "caffeine" or even a certain threshold of misspellings.

Specifying pod IDs and content descriptors with the **WolframAlpha** command is the most exact way to extract information from Wolfram|Alpha results, but there are some additional ways to get at the data. For example, free-form input queries will often return results that can be expanded by clicking the orange plus icon.



population of Illinois

```
EntityValue[ Illinois, United States (administrative division) , population ]
```

Input interpretation:

```
EntityValue[ Illinois, United States (administrative division) , population ]
```

Illinois	population

Result:

12.88 million people (2014)

Population history:

(from 1810 to 2014)

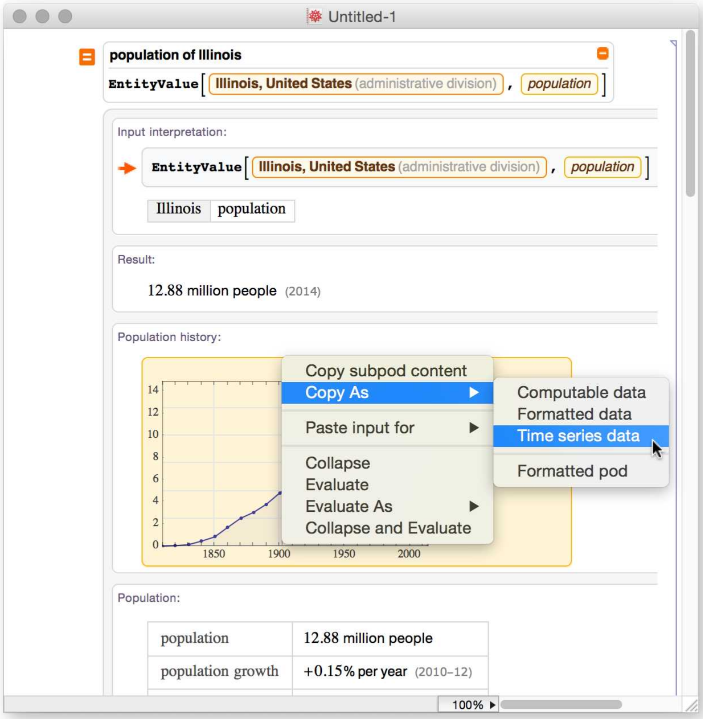
(in millions of people)

Population »

Comparisons »

12 880 580 people

While the **WolframAlpha** command can be used to extract data from any of the pods, right-clicking the pods gives a context menu that allows the data to be copied in a variety of ways; the form of the data is dependent on the type of information displayed in the pod. For example, right-clicking the time series plot of the population allows the data to be copied as time series data, which can be pasted into other commands.



Untitled-1

population of Illinois

```
EntityValue[Illinois, United States (administrative division), population]
```

Input interpretation:

```
EntityValue[Illinois, United States (administrative division), population]
```

Illinois	population

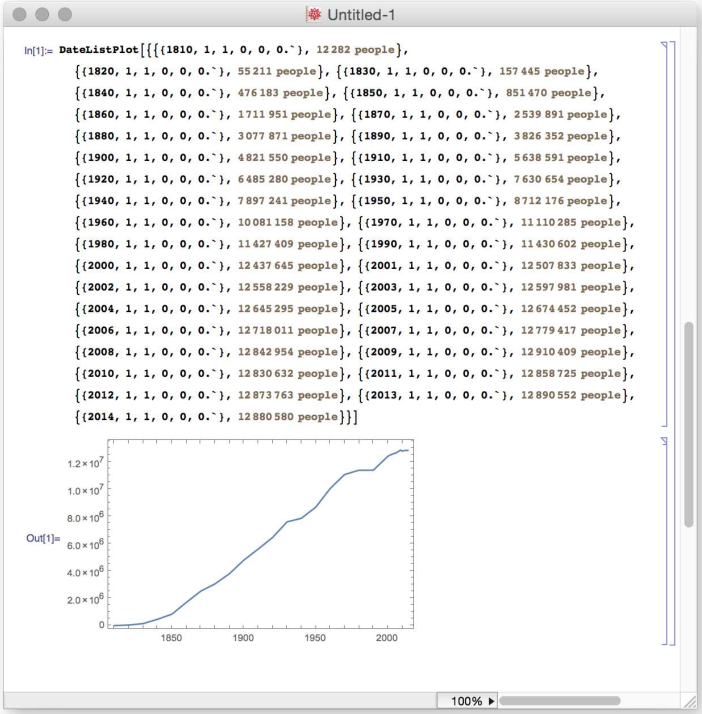
Result:

12.88 million people (2014)

Population history:

Population:

population	12.88 million people
population growth	+0.15% per year (2010–12)

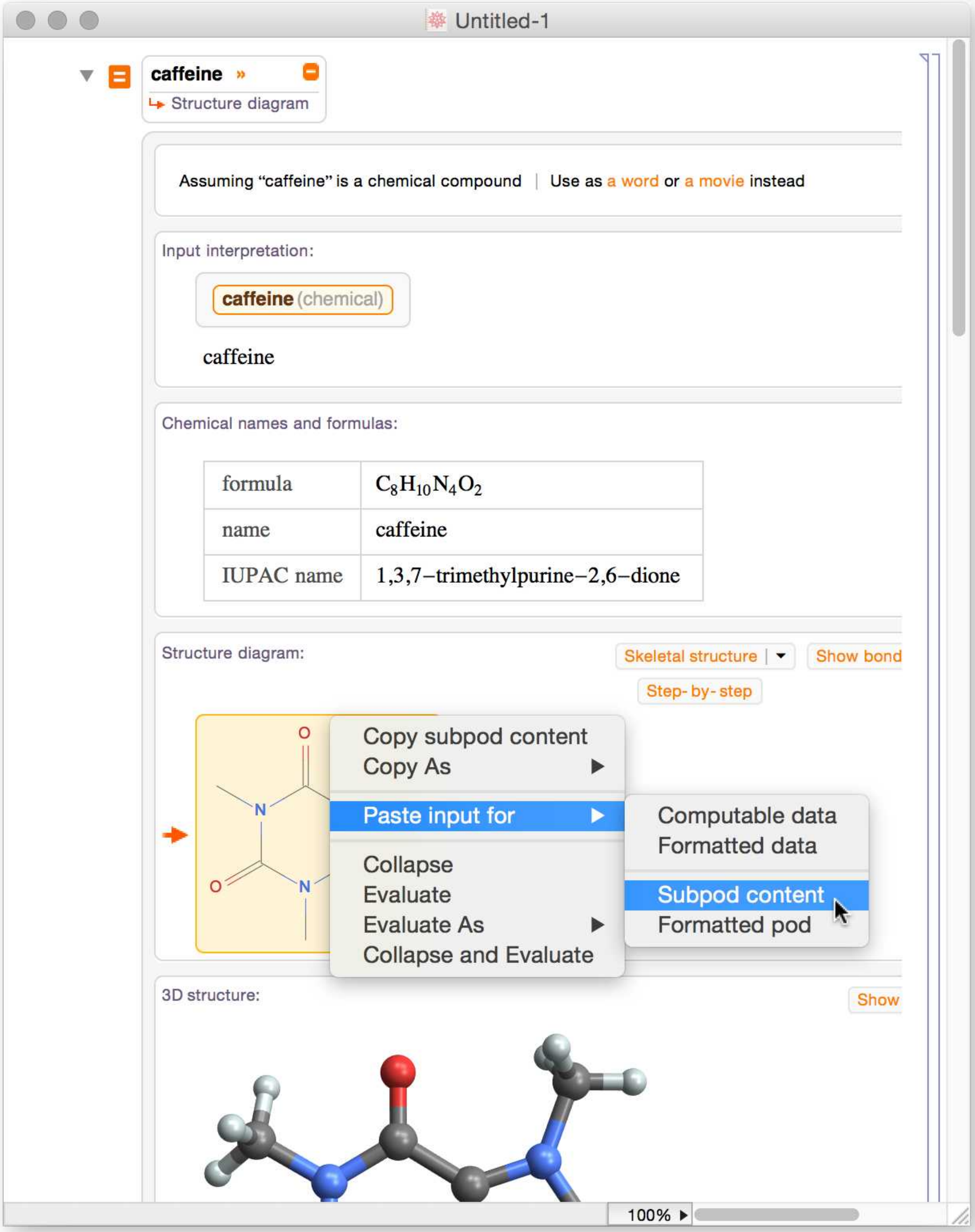


Untitled-1

```
In[1]:= DateListPlot[{{{1810, 1, 1, 0, 0, 0.`}, 12282 people},
        {{1820, 1, 1, 0, 0, 0.`}, 55211 people}, {{1830, 1, 1, 0, 0, 0.`}, 157445 people},
        {{1840, 1, 1, 0, 0, 0.`}, 476183 people}, {{1850, 1, 1, 0, 0, 0.`}, 851470 people},
        {{1860, 1, 1, 0, 0, 0.`}, 1711951 people}, {{1870, 1, 1, 0, 0, 0.`}, 2539891 people},
        {{1880, 1, 1, 0, 0, 0.`}, 3077871 people}, {{1890, 1, 1, 0, 0, 0.`}, 3826352 people},
        {{1900, 1, 1, 0, 0, 0.`}, 4821550 people}, {{1910, 1, 1, 0, 0, 0.`}, 5638591 people},
        {{1920, 1, 1, 0, 0, 0.`}, 6485280 people}, {{1930, 1, 1, 0, 0, 0.`}, 7630654 people},
        {{1940, 1, 1, 0, 0, 0.`}, 7897241 people}, {{1950, 1, 1, 0, 0, 0.`}, 8712176 people},
        {{1960, 1, 1, 0, 0, 0.`}, 10081158 people}, {{1970, 1, 1, 0, 0, 0.`}, 11110285 people},
        {{1980, 1, 1, 0, 0, 0.`}, 11427409 people}, {{1990, 1, 1, 0, 0, 0.`}, 11430602 people},
        {{2000, 1, 1, 0, 0, 0.`}, 12437645 people}, {{2001, 1, 1, 0, 0, 0.`}, 12507833 people},
        {{2002, 1, 1, 0, 0, 0.`}, 12558229 people}, {{2003, 1, 1, 0, 0, 0.`}, 12597981 people},
        {{2004, 1, 1, 0, 0, 0.`}, 12645295 people}, {{2005, 1, 1, 0, 0, 0.`}, 12674452 people},
        {{2006, 1, 1, 0, 0, 0.`}, 12718011 people}, {{2007, 1, 1, 0, 0, 0.`}, 12779417 people},
        {{2008, 1, 1, 0, 0, 0.`}, 12842954 people}, {{2009, 1, 1, 0, 0, 0.`}, 12910409 people},
        {{2010, 1, 1, 0, 0, 0.`}, 12830632 people}, {{2011, 1, 1, 0, 0, 0.`}, 12858725 people},
        {{2012, 1, 1, 0, 0, 0.`}, 12873763 people}, {{2013, 1, 1, 0, 0, 0.`}, 12890552 people},
        {{2014, 1, 1, 0, 0, 0.`}, 12880580 people}}]
```

Out[1]=

Another useful function displayed when right-clicking Wolfram|Alpha results is the **Paste input for** menu. Choosing **Subpod content** from this menu item will paste the syntax for the **WolframAlpha** command that can be evaluated to retrieve that piece of data.



Untitled-1

caffeine »

Structure diagram

Assuming “caffeine” is a chemical compound | Use as a word or a movie instead

Input interpretation:

caffeine (chemical)

caffeine

Chemical names and formulas:

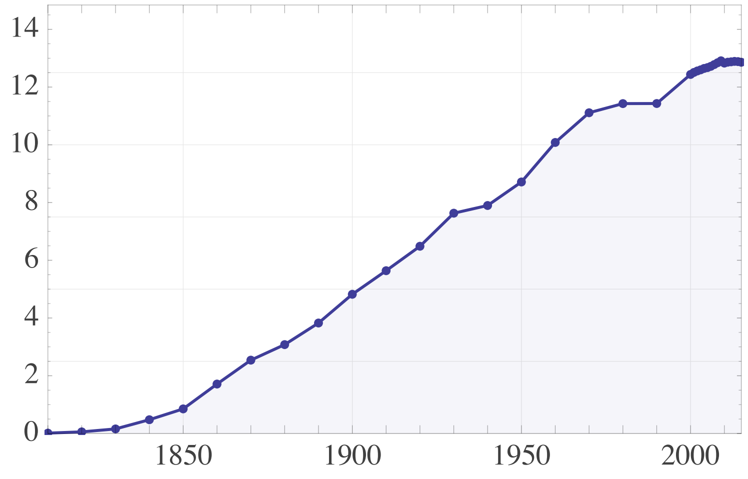
formula	$C_8H_{10}N_4O_2$
name	caffeine
IUPAC name	1,3,7–trimethylpurine–2,6–dione

Structure diagram:

3D structure:

Another choice in the **Paste input for** menu is **Formatted pod**, which will paste the **WolframAlpha** command to summon a standalone version of the formatted pod that is displayed in the full list of results. Formatted pods maintain their interactive features, such as an interactive tracer that may be displayed when mousing over a set of time series data.

```
WolframAlpha["population of Illinois", {{"History:Population:USStateData", 1}, "Content"}]
```



(from 1810 to 2015)

(in millions of people)

The string of text "population of Illinois" works just like it does with free-form input, so it does not require capitalization in order to retrieve a result.

Right-clicking Wolfram|Alpha output provides an easy way to construct a **WolframAlpha** command to programmatically access a particular result. Once the necessary syntax to use the **WolframAlpha** command to retrieve a particular piece of data is identified, it can easily be modified to capture related data. Using the same example for the population of Illinois and the methods just discussed, it is easy to find the **WolframAlpha** command to get the most recent population data directly.

```
WolframAlpha["population of Illinois (US state)", {{"Result", 1}, "NumberData"}]
```

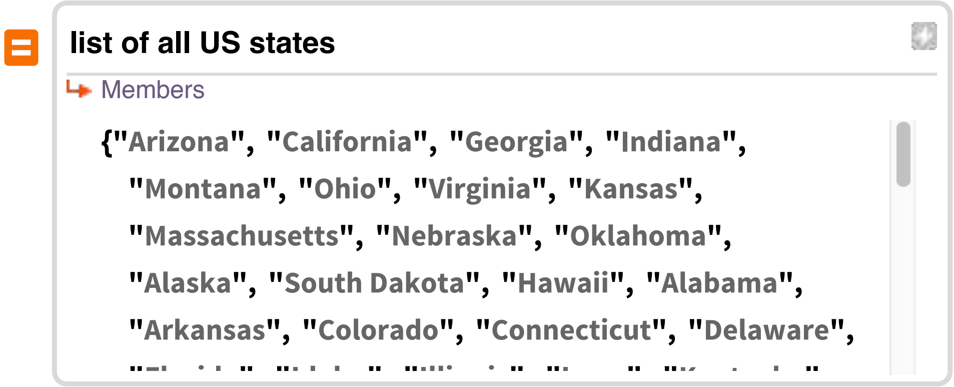
1.286×10^{7}

The same command can be modified to find the population of the state of Missouri.

```
WolframAlpha["population of Missouri (US state)", {{"Result", 1}, "NumberData"}]
```

6.084×10^{6}

The next logical step might be to run this command for the rest of the US states. Rather than typing each state name manually, the data can be queried from Wolfram|Alpha and assigned to a variable.



list of all US states

```
{"Arizona", "California", "Georgia", "Indiana",
  "Montana", "Ohio", "Virginia", "Kansas",
  "Massachusetts", "Nebraska", "Oklahoma",
  "Alaska", "South Dakota", "Hawaii", "Alabama",
  "Arkansas", "Colorado", "Connecticut", "Delaware",
```

```
{Arizona, California, Georgia, Indiana, Montana, Ohio, Virginia, Kansas, Massachusetts,
  Nebraska, Oklahoma, Alaska, South Dakota, Hawaii, Alabama, Arkansas,
  Colorado, Connecticut, Delaware, Florida, Idaho, Illinois, Iowa, Kentucky,
  Louisiana, Maine, Maryland, Michigan, Minnesota, Mississippi, Missouri,
  Nevada, New Hampshire, New Jersey, New Mexico, New York, North Carolina,
  North Dakota, Oregon, Pennsylvania, RhodeIsland, SouthCarolina, Tennessee,
  Texas, Utah, Vermont, Washington, WestVirginia, Wisconsin, Wyoming}
```

```
states = {"Arizona", "California", "Georgia", "Indiana", "Montana", "Ohio",
    "Virginia", "Kansas", "Massachusetts", "Nebraska", "Oklahoma", "Alaska",
    "South Dakota", "Hawaii", "Alabama", "Arkansas", "Colorado", "Connecticut",
    "Delaware", "Florida", "Idaho", "Illinois", "Iowa", "Kentucky", "Louisiana",
    "Maine", "Maryland", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York",
    "North Carolina", "North Dakota", "Oregon", "Pennsylvania", "RhodeIsland",
    "SouthCarolina", "Tennessee", "Texas", "Utah", "Vermont", "Washington",
    "WestVirginia", "Wisconsin", "Wyoming"};
```

It is always possible to copy the output of a calculation and paste it as input. The calculation above simply copies the output from the free-form input and uses it to define a variable. The advantage to hard-coding values into a notebook this way is that since this list is not likely to change, the free-form input cell can now be deleted and the list can be used directly for future calculations that need a list of all 50 US states, without needing to use free-form input again to construct that list.

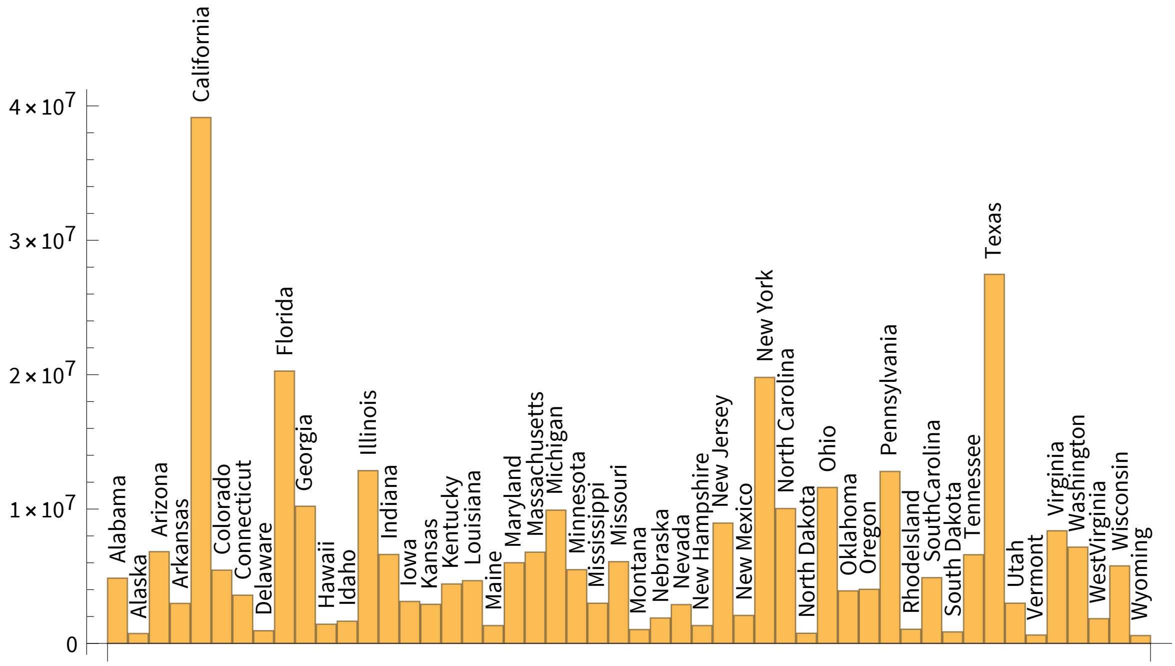
Using the methods discussed earlier in this section, a table of values can now be created consisting of the state's name and population, which will be programmatically queried using the **WolframAlpha** command. The **StringJoin** command, which has a shorthand form of **<>**, will be used to pass arguments to the **WolframAlpha** function call.

```
statePopulations =
   Table[{s, WolframAlpha["population of " <> s <> " (US state)",
         {{"Result", 1}, "NumberData"}]},
     {s, states}];
```

The **Table** statement above can seem like a lengthy calculation, but it is doing many things at once, like sending text to Wolfram|Alpha, interpreting the text and returning results for a list of values. The calculation time will be much faster than typing in all the states as individual queries. The timing is also dependent on the speed of the available internet connection when running the calculation.

Sorting the data to be alphabetical and then plotting it as a bar chart provides a more interesting presentation. Some additional options are used to make the labeling more effective by placing the labels above the bars and rotating the labels by 90°. Notice that Mathematica automatically uses scientific notation for the labeling on the *y* axis for a better presentation of those values.

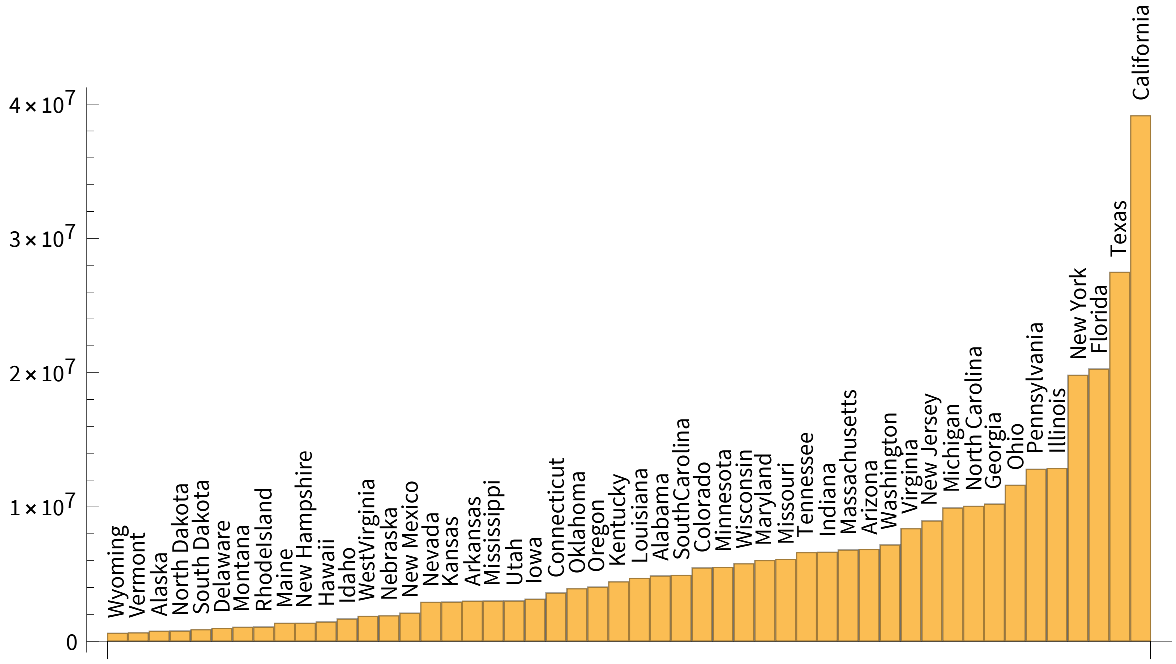
```
statePopulations = SortBy[statePopulations, First];
BarChart[statePopulations⟦All, 2⟧,
  ChartLabels → Placed[statePopulations⟦All, 1⟧, Above, Rotate[#, 90 Degree] &],
  ImageSize → All]
```



We said we were not going to use pure functions, and here is an example of a pure function represented by the **#** symbol. Just take our word when we say this approach is the easiest way to rotate labels for a chart. You can read more about pure functions in the documentation.

The previous two calculations can be combined into one series of calculations that also sorts the data based on the population value before charting it.

```
statePopulations =
    Table[{s, WolframAlpha["population of " <> s <> "(US state)",
          {{"Result", 1}, "NumberData"}]},
      {s, states}];
statePopulations = SortBy[statePopulations, Last];
BarChart[statePopulations⟦All, 2⟧,
  ChartLabels → Placed[statePopulations⟦All, 1⟧, Above, Rotate[#, Pi/2] &],
  ImageSize → All]
```



By now it should be evident that results returned by the **WolframAlpha** command are Wolfram Language expressions like everything else in the system, allowing them to be manipulated in the same way. For example, exploring the full results for a free-form query on the state of New York lists education expenses, and right-clicking the pod can be used to find the command necessary to access the computable data.

```
WolframAlpha["New York (US state)",
  {{"EducationFunding:USStateData", 1}, "ComputableData"}]
```

```
{{total expenditures, $5.735 × 10^10}, {total revenue, $5.568 × 10^10},
  {total debt, $3.053 × 10^10}, {total cash and securities, $9.017 × 10^9}}
```

The **Part** command can be used to extract the numeric value for total education expenditures, which are the elements in the first row and second column. The same **Table** statement that was used to generate the list of population values for each state can be changed to find total education expenditures instead.

```
Table[
  {s,
    WolframAlpha[s <> " (US state)",
      {{"EducationFunding:USStateData", 1}, "ComputableData"}][[1, 2]]},
  {s, states}]
```

```
{{Arizona, $9.58×10^9}, {California, $7.185×10^10}, {Georgia, $1.901×10^10},
  {Indiana, $1.095×10^10}, {Montana, $1.608×10^9}, {Ohio, $2.226×10^10},
  {Virginia, $1.529×10^10}, {Kansas, $5.816×10^9}, {Massachusetts, $1.484×10^10},
  {Nebraska, $3.501×10^9}, {Oklahoma, $5.935×10^9}, {Alaska, $2.396×10^9},
  {South Dakota, $1.262×10^9}, {Hawaii, $2.319×10^9}, {Alabama, $7.806×10^9},
  {Arkansas, $4.981×10^9}, {Colorado, $8.634×10^9}, {Connecticut, $9.107×10^9},
  {Delaware, $1.732×10^9}, {Florida, $2.887×10^10}, {Idaho, $2.078×10^9},
  {Illinois, $2.687×10^10}, {Iowa, $5.514×10^9}, {Kentucky, $6.83×10^9},
  {Louisiana, $7.936×10^9}, {Maine, $2.545×10^9}, {Maryland, $1.248×10^10},
  {Michigan, $1.879×10^10}, {Minnesota, $1.101×10^10},
  {Mississippi, $4.553×10^9}, {Missouri, $1.013×10^10}, {Nevada, $4.574×10^9},
  {New Hampshire, $2.595×10^9}, {New Jersey, $2.539×10^10},
  {New Mexico, $3.941×10^9}, {New York, $5.735×10^10},
  {North Carolina, $1.484×10^10}, {North Dakota, $1.079×10^9},
  {Oregon, $6.7×10^9}, {Pennsylvania, $2.5×10^10}, {RhodeIsland, $2.149×10^9},
  {SouthCarolina, $8.413×10^9}, {Tennessee, $8.582×10^9}, {Texas, $5.342×10^10},
  {Utah, $4.417×10^9}, {Vermont, $1.475×10^9}, {Washington, $1.22×10^10},
  {WestVirginia, $3.12×10^9}, {Wisconsin, $1.068×10^10}, {Wyoming, $1.65×10^9}}
```

The same approach of hard-coding the values is taken so that the data lookup command does not need to be run again at a later date.

```
stateEduSpending = {{"Arizona", $9.58`*^9}, {"California", $7.185`*^10},
    {"Georgia", $1.901`*^10}, {"Indiana", $1.095`*^10}, {"Montana", $1.608`*^9},
    {"Ohio", $2.226`*^10}, {"Virginia", $1.529`*^10}, {"Kansas", $5.816`*^9},
    {"Massachusetts", $1.484`*^10}, {"Nebraska", $3.501`*^9},
    {"Oklahoma", $5.935`*^9}, {"Alaska", $2.396`*^9},
    {"South Dakota", $1.262`*^9}, {"Hawaii", $2.319`*^9},
    {"Alabama", $7.806`*^9}, {"Arkansas", $4.981`*^9}, {"Colorado", $8.634`*^9},
    {"Connecticut", $9.107`*^9}, {"Delaware", $1.732`*^9},
    {"Florida", $2.887`*^10}, {"Idaho", $2.0779999999999998`*^9},
    {"Illinois", $2.687`*^10}, {"Iowa", $5.514`*^9}, {"Kentucky", $6.83`*^9},
    {"Louisiana", $7.936`*^9}, {"Maine", $2.545`*^9}, {"Maryland", $1.248`*^10},
    {"Michigan", $1.879`*^10}, {"Minnesota", $1.101`*^10},
    {"Mississippi", $4.553`*^9}, {"Missouri", $1.0129999999999998`*^10},
    {"Nevada", $4.574`*^9}, {"New Hampshire", $2.595`*^9},
    {"New Jersey", $2.539`*^10}, {"New Mexico", $3.941`*^9},
    {"New York", $5.735`*^10}, {"North Carolina", $1.484`*^10},
    {"North Dakota", $1.079`*^9}, {"Oregon", $6.7`*^9},
    {"Pennsylvania", $2.5`*^10}, {"RhodeIsland", $2.149`*^9},
    {"SouthCarolina", $8.413`*^9}, {"Tennessee", $8.582000000000001`*^9},
    {"Texas", $5.342`*^10}, {"Utah", $4.417`*^9}, {"Vermont", $1.475`*^9},
    {"Washington", $1.22`*^10}, {"WestVirginia", $3.12`*^9},
    {"Wisconsin", $1.068`*^10}, {"Wyoming", $1.65`*^9}};
```

Combining these two datasets—for population and public education expenditures—can be a way to explore if larger states are efficient with education spending, or to see if there are outliers in education spending when comparing states with similar populations. This exploration, in turn, might lead to further investigation of why those outliers exist. Since the component datasets of population and total education expenditures are already created, **Table** can be used to construct a new dataset consisting of the state name, population and education expenditures. First, though, the two component datasets can be compared to make sure they are of equal length and consist of the same states in the same order.

```
Length[statePopulations] == Length[stateEduSpending]
```

```
True
```

```
statePopulations⟦All, 1⟧ == stateEduSpending⟦All, 1⟧
```

```
False
```

Since the two datasets do not have the states in the same order, each variable definition is replaced by its sorted version.

```
statePopulations = Sort[statePopulations];
stateEduSpending = Sort[stateEduSpending];
```

Now it can be verified that both datasets have the states in the same order.

```
statePopulations⟦All, 1⟧ == stateEduSpending⟦All, 1⟧
```

```
True
```

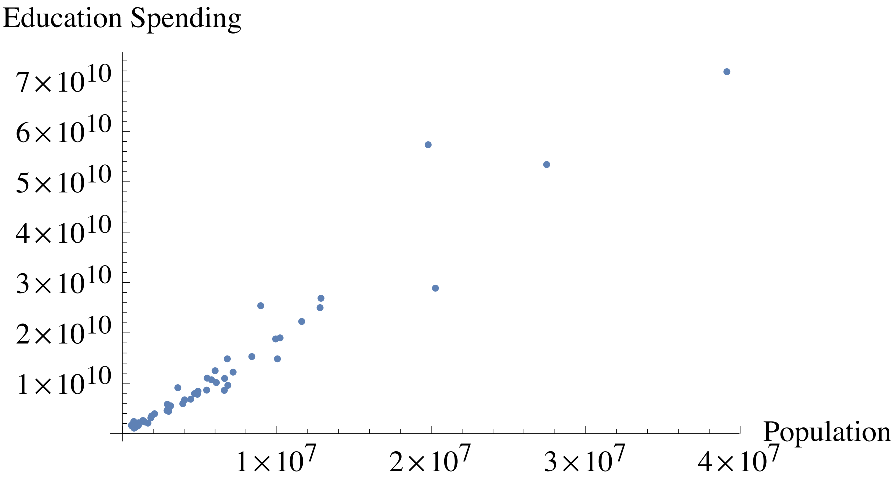
The **⟦All, 1⟧** is used to compare the state names only. If equality were checked for **statePopulations** and **stateEduSpending**, that would fail, since they do not have the same values in the second positions. The first variable (**statePopulations**) has population figures in the second position, and the second variable (**stateEduSpending**) has spending figures in the second position.

Now the **Table** command is used to construct a new dataset that has a population figure as its first entry and a spending figure as its second entry. **Tooltip** is invoked in order to give additional information once the data is plotted. The semicolon is used to suppress the rather long output.

```
statePopAndEduSpendingTooltipped =
    Table[
      Tooltip[{
            statePopulations[[i, 2]],
            stateEduSpending[[i, 2]]},
         statePopulations[[i, 1]]],
      {i, 1, 50, 1}];
```

Now the data can be visualized with **ListPlot**, with **Tooltip** giving popup windows with state names as the pointer hovers over the data points.

```
ListPlot[
   statePopAndEduSpendingTooltipped,
   AxesLabel → {"Population", "Education Spending"}, PlotRange → All,
   ImageSize → 300]
```



There does not appear to be an obvious relationship between population and education spending. But there are certainly outlier states that have higher total education expenditures compared to states with similar populations. Considering other factors can help explore hypotheses regarding the outliers. For example, the **WolframAlpha** command can be used to examine the crime rate per capita for each state. States with higher crime rates might allocate more money to this area, leaving less funding for education.

Similarly to the approach in **Chapter 20: Data Filtering and Manipulation**, a filter is used to find states that have fewer than 3,000 crimes per 100,000 people per year and ignore states that do not meet this criterion.

```
statePopAndEduSpending =
    Table[{
        statePopulations[[i, 1]],
        statePopulations[[i, 2]],
        stateEduSpending[[i, 2]]},
      {i, 1, 50, 1}];
statePopAndEduSpending = SortBy[statePopAndEduSpending, First];

lowCrimesFilter[x_] :=
    WolframAlpha[x[[1]] <> " (US state)",
        {{"StateCrimeInformation:CrimeData", 1}, "ComputableData"},
        TimeConstraint → 300][[1, 2, 1]] < 3000;
lowCrimeStates = Select[statePopAndEduSpending, lowCrimesFilter];
```

The preceding example returns units that make use of the **Quantity** command. The statement of **[[1,2,1]]** returns the first element in the list (which deletes empty data), then the second element in the list (which is the **Quantity** function with a numeric value and a unit of crimes per person) and then the first element (which is the numeric value of the crimes per person but not the unit).

The variable **lowCrimeStates** is used above to store this new list, to eliminate the need for redundant downloads of this dataset. The **Dimensions** command shows that 23 states fall into this low crime rate category.

```
lowCrimeStates[[1 ;; 5]]
```

```
{{Colorado, 5.457×10^6, $8.63400×10^9 },
  {Connecticut, 3.591×10^6, $9.10700×10^9 }, {Idaho, 1.655×10^6, $2.078×10^9 },
  {Illinois, 1.286×10^7, $2.68700×10^10 }, {Iowa, 3.124×10^6, $5.51400×10^9 }}
```

```
Dimensions[lowCrimeStates]
```

```
{23, 3}
```

A similar filter can be used to return only states with a higher crime rate, which for this example is defined as more than 4,000 crimes per 100,000 people per year. This list of 11 states is stored in a variable called **highCrimeStates**.

```
highCrimesFilter[x_] :=
    WolframAlpha[x⟦1⟧ <> " (US state)",
        {{"StateCrimeInformation:CrimeData", 1}, "ComputableData"},
        TimeConstraint → 300]⟦1, 2, 1⟧ > 4000;
highCrimeStates = Select[statePopAndEduSpending, highCrimesFilter];
```

Note that the **TimeConstraint** option was used with the **WolframAlpha** function to give the command more time to collect data before the operation times out. This number can be adjusted up or down depending on whether you want to give an operation more time to complete or you want to limit its evaluation time.

```
Dimensions[highCrimeStates]
```

```
{5, 3}
```

The remaining states can be filtered using a similar approach, or the **Complement** command can be employed to find the remaining states that have not been categorized into either high-crime or low-crime states.

```
mediumCrimeStates = Complement[statePopAndEduSpending, lowCrimeStates,
    highCrimeStates];
```

```
Dimensions[mediumCrimeStates]
```

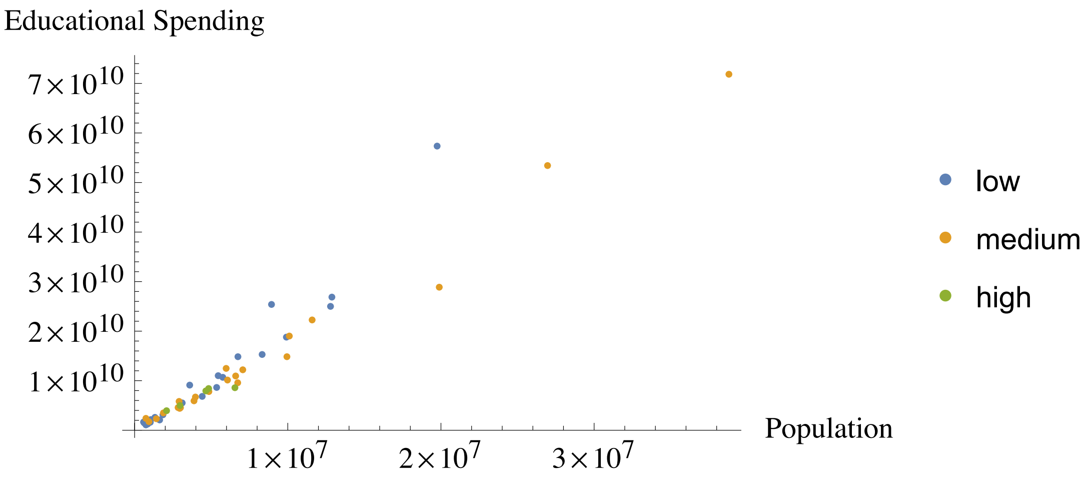
```
{22, 3}
```

Now that the states are categorized, **Table** and **Tooltip** can be used to add labels for each data point. The **Table** statement is used to collect the second and third element of each sublist, which correspond to the population and total education expenditures, respectively, and the first element of each sublist is used as the label for the **Tooltip** command. This process is repeated for each of the three lists so that the datasets will be colored differently when plotted on the same set of axes.

```
lowCrimeStates = Table[Tooltip[i⟦2 ;; 3⟧, i⟦1⟧], {i, lowCrimeStates}];
mediumCrimeStates = Table[Tooltip[i⟦2 ;; 3⟧, i⟦1⟧], {i, mediumCrimeStates}];
highCrimeStates = Table[Tooltip[i⟦2 ;; 3⟧, i⟦1⟧], {i, highCrimeStates}];
```

The overall result mirrors the chart that plotted population against education expenditures, but here the states with low crime rates, medium crime rates and high crime rates are colored differently.

```
ListPlot[
  {lowCrimeStates, mediumCrimeStates, highCrimeStates},
  AxesLabel → {"Population", "Educational Spending"},
  PlotLegends → {"low", "medium", "high"},
  PlotRange → All, ImageSize → 300]
```



Although by no means a uniform relationship, the states with high crime rates do tend to have less total educational expenditures, and the states with low crime rates tend to have higher total educational expenditures.

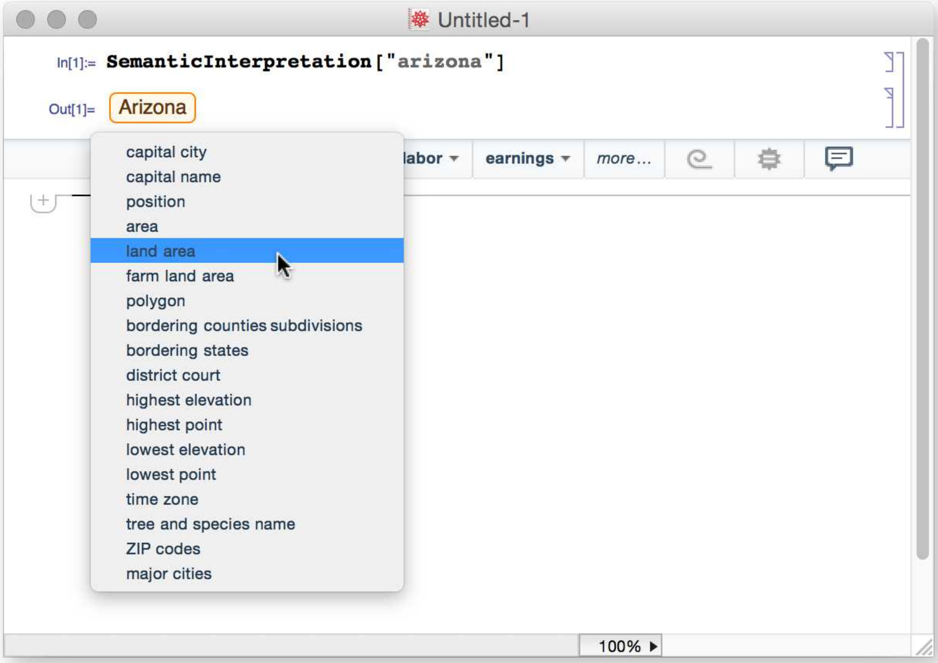
Semantic Import Based on Everyday English

Chapter 19: Importing and Exporting Data introduced **SemanticImport**, which correlates imported data with the data in the Wolfram Knowledgebase. Once a correlation is made, properties related to the identified object can be looked up and used in computations.

A related command, **SemanticInterpretation**, can be applied to user-given input. Rather than taking an entire file and trying to relate its elements to canonical entities in the Wolfram Knowledgebase, **SemanticInterpretation** takes a string and tries to find knowledge about that single entity. For example, the **SemanticInterpretation** command can be used to identify that "Arizona" corresponds to the US state, and then the Suggestions Bar can provide instant access to bits of information related to Arizona.

```
SemanticInterpretation["arizona"]
```

Arizona



Untitled-1

```
In[1]:= SemanticInterpretation["arizona"]

Out[1]= Arizona
```

One advantage to **SemanticInterpretation** is that a single **Entity** is returned as the output, rather than a large collection of pods. This might save you time by eliminating the need to extract a single desired pod from output returned by the **WolframAlpha** command.

When the input is ambiguous, **SemanticInterpretation** uses heuristics to determine an appropriate output. For example, with the following input, Mathematica returns an **Entity** representing the city of Washington, D.C. and not Washington state.

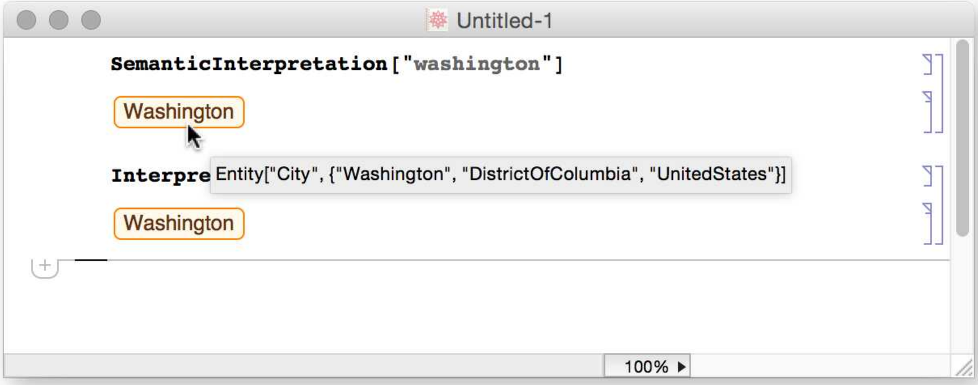
```
SemanticInterpretation["washington"]
```

```
Washington
```

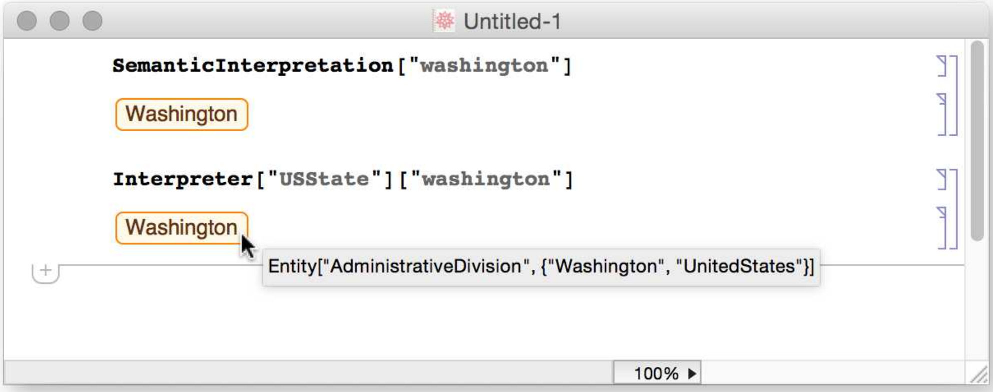
Interpreter is a function that can narrow down the search parameters to a certain type of data. In this case, the **Interpreter** specifies US states. The **Entity** graphical representation looks identical to the previous output, but after mousing over each, it is clear that one entity refers to a city and the other entity refers to a state.

```
Interpreter["USState"]["washington"]
```

```
Washington, United States
```



```
SemanticInterpretation["washington"]
Washington
Interpre Entity["City", {"Washington", "DistrictOfColumbia", "UnitedStates"}]
Washington
```



```
SemanticInterpretation["washington"]
Washington
Interpreter["USState"]["washington"]
Washington
Entity["AdministrativeDivision", {"Washington", "UnitedStates"}]
```

Interpreter can be used to define a function that restricts the output to only US states. This is useful when programmatically looking up information, like the following **Table** statement that queries data on many states. With this interpreter function, the input can be any format that commonly represents a US state, including misspellings.

```
myint = Interpreter["USState"];
Table[myint[state], {state, {"Washingto", "California", "ME", "NE", "NH"}}]
```

```
{ Washington, United States ,
   California, United States , Maine, United States ,
   Nebraska, United States , New Hampshire, United States }
```

With **WolframAlpha** calls earlier in the chapter, strings with "(US state)" were used to make sure the input was interpreted as a US state. **Interpreter** is a more robust way to do this, and the formal syntax makes it easier to specify a certain class of data.

After returning an **Entity**, the **EntityProperties** command can be used to show the types of data that can be extracted for each US state.

```
Short[EntityProperties[Entity["AdministrativeDivision", {"Maine", "UnitedStates"}]]]
```

```
{ accommodation and food services sales , number of aggravated assaults ,
  rate of aggravated assault , aggregate household income ,
  annual births , annual deaths , housing units change rate , area ,
  average commute time , average home sale price , bordering counties ,
  bordering states , number of burglaries , rate of burglary ,
  number of businesses , business ownership by ethnicity , ≪112≫,
  abbreviation , state government tax collections , state sales tax rate ,
  subdivisions , time zones , total voter registration rate ,
  total voting rate , state tree , unweighted sample housing units ,
  unweighted sample population , total rate of violent crime ,
  total number of violent crimes , wholesale sales , employment , ZIP codes }
```

EntityValue is used to extract any of the properties listed in the last output. **EntityValue** is simply wrapped around the **Entity** input corresponding to that specific state. **Entity** can either be typed out in regular Wolfram Language syntax, or the graphical representation can be used as input directly.

```
EntityValue[Entity["AdministrativeDivision", {"Maine", "UnitedStates"}],
  "Slogans"]
```

```
{Worth a visit, worth a lifetime, The way life should be,
  Where America's day begins, Vacationland, It must be Maine}
```

```
EntityValue[ Maine (administrative division) , "Slogans"]
```

```
{Worth a visit, worth a lifetime, The way life should be,
  Where America's day begins, Vacationland, It must be Maine}
```

Rather than querying individual states, a broader input can be given to create a list of all US states. The output displays only the first 10 states in the list to save screen space.

```
EntityList[SemanticInterpretation["All US states"]]⟦1 ;; 10⟧
```

```
{ Alabama, United States , Alaska, United States ,
   Arizona, United States , Arkansas, United States ,
   California, United States , Colorado, United States ,
   Connecticut, United States , Delaware, United States ,
   District of Columbia, United States , Florida, United States }
```

Table can also be used with **Entity** and **EntityValue**. Starting with a list of all US states, a list is created that displays the name and then the total voting rate. This new list can be sorted and stored as a variable; variables can store any expressions, including **Entity** representations of data.

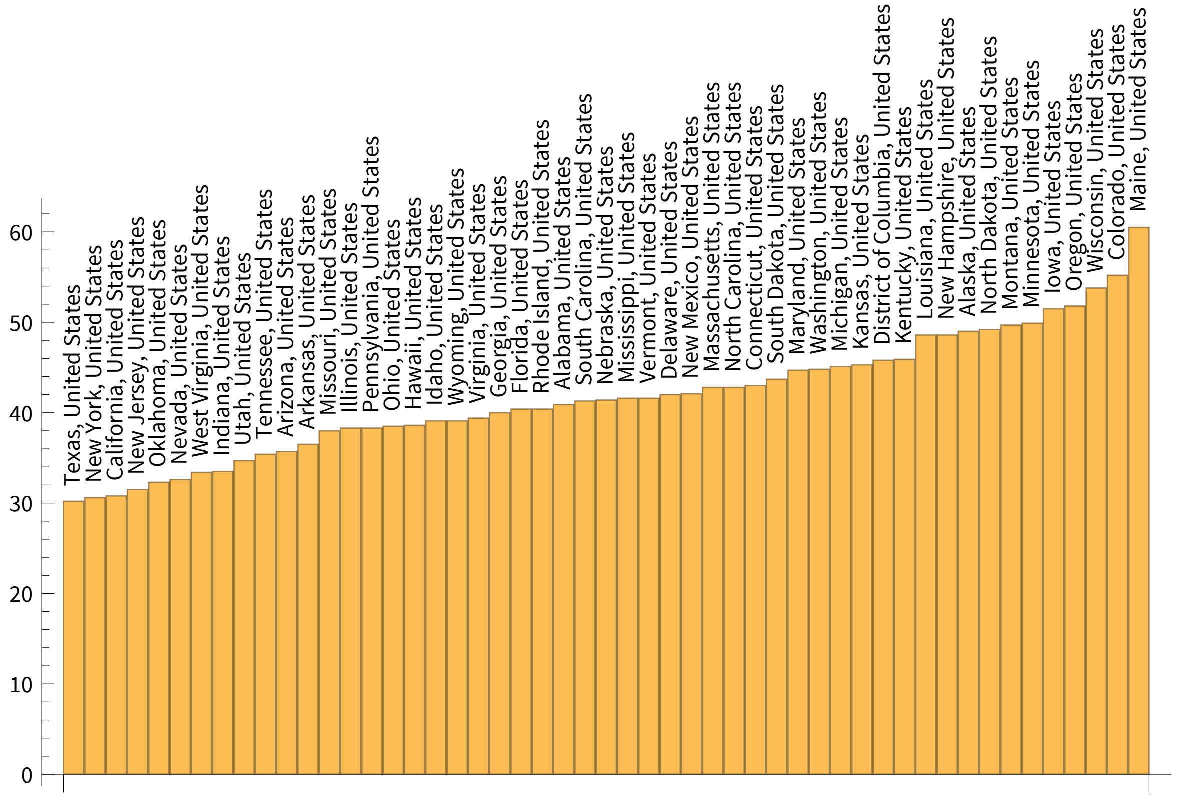
```
votingData =
    SortBy[Table[{state, EntityValue[state, "TotalVotingRate"]},
        {state, EntityList[SemanticInterpretation["All US states"]]}], Last];
votingData⟦1 ;; 10⟧
```

```
{{ Hawaii, United States , 46.8% },{ Texas, United States , 48.8% },
  { Utah, United States , 50.5% },{ California, United States , 51.2% },
  { New York, United States , 51.5% },{ Arkansas, United States , 51.8% },
  { Nevada, United States , 52.8% },{ West Virginia, United States , 53.1% },
  { Arizona, United States , 53.3% },{ Tennessee, United States , 53.6% }}
```

The overall approach for this example is similar to the **Table** statements used earlier in the chapter with the **WolframAlpha** command. The advantages to using **WolframAlpha** instead of **SemanticInterpretation** are partially related to workflow, and partially due to the data that is available through each command. **WolframAlpha** allows easier discovery of the most popular datasets and has more graphics and precomputed sets of information, which can be useful. **SemanticInterpretation** provides only one output and provides access to less common pieces of information, but lends itself extremely well to programmatic interpretation of a specific type of data.

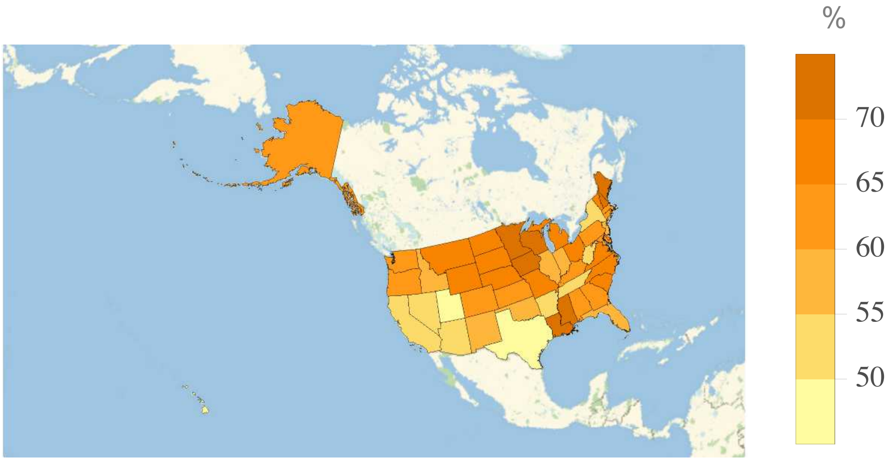
By using the stored variable **votingData**, the percentages can be displayed graphically with labels for each state.

```
BarChart[votingData⟦All, 2⟧,
  ChartLabels → Placed[votingData⟦All, 1⟧, Above, Rotate[#, Pi/2] &],
  ImageSize → All]
```



In addition to viewing the voting data as a bar chart, **GeoRegionValuePlot** can be used to create a geographical heat map of the values. The legend is automatically created to show voting rate differences among US states.

```
GeoRegionValuePlot[votingData]
```



Clear is used to remove all variable and function definitions from this chapter.

```
Clear[states, statePopulations, stateEduSpending, statePopAndEduSpendingTooltipped,
  statePopAndEduSpending, lowCrimesFilter, lowCrimeStates, highCrimesFilter,
  highCrimeStates, mediumCrimeStates, myint, votingData]
```

Conclusion

The **WolframAlpha** and **SemanticInterpretation** commands provide additional ways to access data from the Wolfram Knowledgebase. The **WolframAlpha** command is a useful starting point, especially for existing users of the Wolfram|Alpha website. **SemanticInterpretation** can be a better approach when trying to discover data sources that do not have a dedicated curated data function or are not displayed as **WolframAlpha** pods. Mathematica gives you the flexibility to choose between these methods when using curated data with projects.

Exercises

1. Use **SemanticInterpretation** to find entities for the term "big apple."
2. Use the Suggestions Bar to find the population of the result from Exercise 1.
3. Use free-form input to find the three largest cities in the United States.
4. Copy and paste the input and output from Exercise 3 and use the plus icon to show all results for the evaluation.
5. Right-click the subpod for **Educational attainment**, copy the subpod content and paste the results into a new output cell. Create a new subsubsection cell above this output cell, and add the text "Educational Attainment for the Three Largest US Cities" to the subsubsection cell.
6. Use the **CityData** command to find the populations of the "windy city" and the "gateway to the west." (Hint: use the Ctrl + = keyboard shortcut to enter these free-form input strings as arguments to **CityData**.)
7. Create an interpreter to identify breeds of cats.
8. Use the result from Exercise 7 to find an entity representation of a Singapura cat.
9. Find the available properties for the result from Exercise 8.
10. Use **EntityValue** to display an image of a Singapura cat.

CHAPTER 23

Statistical Functionality for Data Analysis

Introduction

In addition to importing, processing and plotting data, Mathematica can be used for various statistical tests. The syntax for many of the commands requires a list or dataset as the only argument, making it easy to combine the steps of importing data, reformatting data, filtering data and performing statistical tests into a single compound statement.

Basic and Descriptive Statistics

Chapter 21: Working with Curated Data used the curated data command **CountryData** to explore the relationship between a country's population and its number of cellular phones. The following example is almost identical in form but creates a list to explore the relationship between the number of internet users and the number of cellular phones in large countries, defined as those with populations greater than 5×10^6 people.

```
popSelect[x_] := QuantityMagnitude[CountryData[x, "Population"]] > 5*10^6;
largeCountries = Select[CountryData[All], popSelect];
```

Now that the list of large countries is created, the **Table** command is used with **CountryData** to look up the properties of interest, namely the number of cellular phones and the number of internet users.

```
largeCountriesCellAndInternet = Table[
        {CountryData[i, "CellularPhones"], CountryData[i, "InternetUsers"]},
        {i, largeCountries}];
```

And the size of the dataset is examined using **Dimensions**.

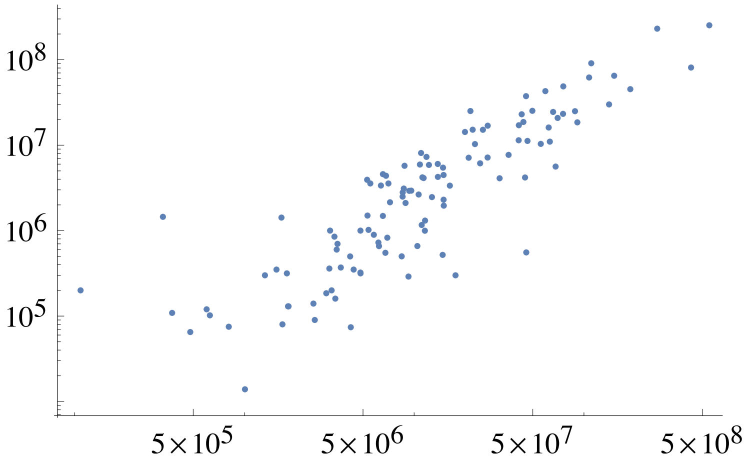
```
Dimensions[largeCountriesCellAndInternet]

 {118, 2}
```

Using the **Dimensions** function to verify that the dataset is of the intended format is a good habit. In this case, **Dimensions** verifies that the dataset includes 118 rows and 2 columns.

An intuitive guess may be that countries that embrace technology should show a relationship between these two properties. **ListLogLogPlot** can be used to visualize the data for these 118 countries while avoiding the dense clustering that a basic **ListPlot** could show in the lower ranges of both properties. There does indeed seem to be a strong relationship between the number of cellular phones in a country and the number of internet users for this group of countries.

```
ListLogLogPlot[
  Table[
    Tooltip[{CountryData[i, "CellularPhones"], CountryData[i, "InternetUsers"]}, i],
    {i, largeCountries}]
]
```



While **ListLogLogPlot** is forgiving and ignores missing values, a better approach is to use **Cases** to remove the non-numerical entries corresponding to missing data.

```
largeCountriesCellAndInternet =
    Cases[Table[
        {CountryData[i, "CellularPhones"], CountryData[i, "InternetUsers"]},
        {i, largeCountries}],
      {_Real, _Quantity}];
```

Dimensions can be used to verify that the incomplete entries were removed. The result verifies that the data has a consistent format of 116 rows and 2 columns.

```
Dimensions[largeCountriesCellAndInternet]
```

```
{116, 2}
```

In situations where **Cases** is being used to identify a pattern, the **Head** function is useful to identify that pattern. **Head** exposes the underlying symbolic function call that is used to represent an expression. In this case, the first element in the list is a **Real** number and the second uses **Quantity**, since the data is unitized.

```
Head[largeCountriesCellAndInternet[[1, 1]]]
```

```
Real
```

```
Head[largeCountriesCellAndInternet[[1, 2]]]
```

```
Quantity
```

> Every Mathematica expression has a head. The head of the expression **26** is **Integer**, the head of **26.1** is **Real**, the head of **{1, 2, 3}** is **List** and the head of **a + b** (assuming **a** and **b** are both undefined) is **Plus**. You may remember the use of the commands **TreeForm** and **FullForm** from **Chapter 6: Fundamentals of the Wolfram Language**; these commands are used to examine the underlying symbolic representation of any expression in Mathematica. **Head** is somewhat similar, only instead of giving the entire symbolic representation, it only returns the outermost command of that symbolic representation.

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

```
List[1, 2, 3]
```

This dataset can be used to show a variety of the descriptive statistics commands available in the Wolfram Language. Many of the functions will thread over multiple dimensions, meaning that if the **Mean** command is given a two-dimensional list, the means of elements in each column will be returned.

```
Mean[largeCountriesCellAndInternet]
```

$\{3.43083\times10^{7},\ 1.3486\times10^{7}\ \text{people}\}$

The **Mean** function also works on one-dimensional data.

```
Mean[largeCountriesCellAndInternet⟦All, 1⟧]
```

3.43083×10^{7}

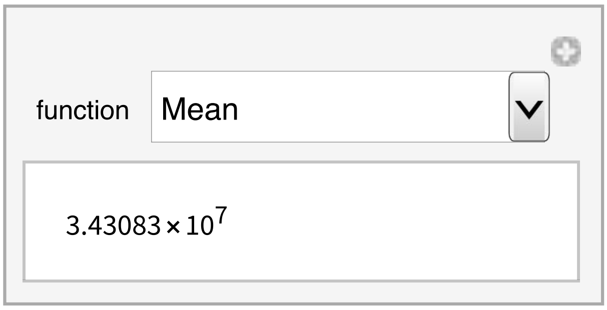
The **Total** command is used to add up each element in a list, so the following calculation returns the total number of cellular phones for the group of large countries in question.

```
Total[largeCountriesCellAndInternet⟦All, 1⟧]
```

3.92207×10^{9}

Since many descriptive statistics commands have the same overall syntax and structure, **Manipulate** can be used to explore a variety of basic statistics applied to the number of cellular phones for the set of large countries. The **function** variable is used to interactively change the desired descriptive statistics command.

```
Manipulate[
  function[largeCountriesCellAndInternet⟦All, 1⟧],
  {function, {Mean, HarmonicMean, GeometricMean, Median, Variance, Total,
      StandardDeviation, InterquartileRange, Max, Min, Accumulate}},
  SaveDefinitions → True
]
```



function: Mean

3.43083×10^{7}

To ensure that the **Manipulate** statement works without requiring evaluation of other cells in the notebook, the definition of the **largeCountriesCellAndInternet** variable is saved by using the **SaveDefinitions → True** option setting.

Measures between datasets can be computed as well, such as the covariance between the number of cellular phones and the number of internet users for large countries.

```
Covariance[largeCountriesCellAndInternet〚All, 1〛, largeCountriesCellAndInternet〚All, 2〛]
```

```
2.54141 × 10^15 people
```

Another example of a measure between datasets is correlation. The **Correlation** command takes two datasets as arguments and shows the correlation between those datasets. In this case, the result for the correlation between the number of cellular phones and the number of internet users for large countries makes sense based on the behavior shown in the log plot earlier in the chapter, which indicated a pattern between the two properties.

```
Correlation[largeCountriesCellAndInternet〚All, 1〛, largeCountriesCellAndInternet〚All, 2〛]
```

```
0.889035
```

Curve Fitting

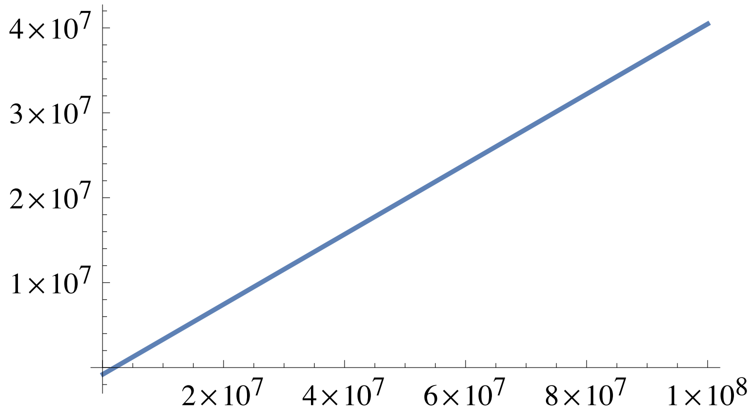
The Wolfram Language has several commands that can be used for curve fitting. The **Fit** command can be used to find a least-squares fit to a list of data, and this command is most commonly used for polynomial fitting. **Fit** takes three arguments: a one- or multidimensional dataset, a list of parameters to specify the form of the fitted equation and the variable in question. In the following case, a linear fit is calculated for the data relating to the number of cellular phones and the number of internet users for each large country. The default output is the fitted equation.

```
fit1 = Fit[QuantityMagnitude[largeCountriesCellAndInternet], {1, x}, x]
```

```
-1.40708 × 10^6 + 0.434096 x
```

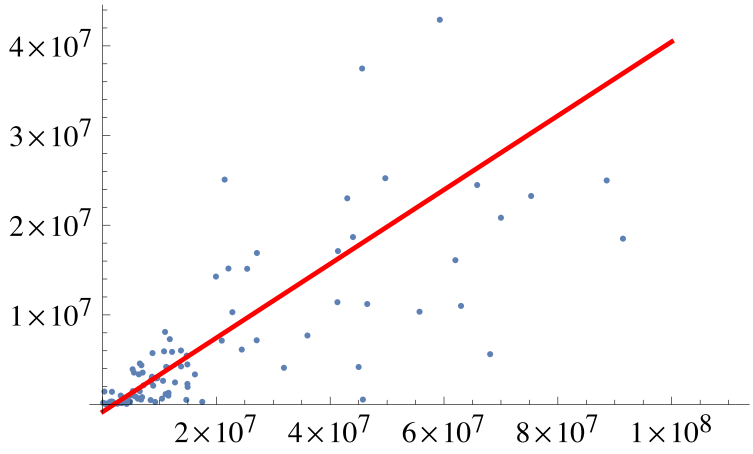
This result is stored in the **fit1** variable. The variable that stores the equation can be used directly in a **Plot** function to visualize the linear fit.

```
Plot[fit1, {x, 10^5, 10^8}]
```



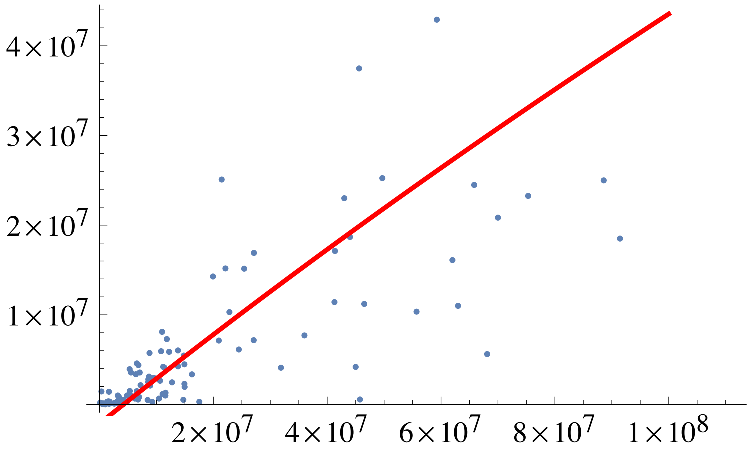
The **Show** command can be used to combine multiple graphics outputs, such as a list plot of the raw data and the curve fitted to the data. **Show** takes as its arguments the graphics objects to be combined. In this case, the first argument is the plot of the raw data and the second is the plot of the fitted equation. A **PlotStyle** option setting is used to differentiate the two types of plots.

```
Show[{
    ListPlot[largeCountriesCellAndInternet],
    Plot[fit1, {x, 10^5, 10^8}, PlotStyle → Red]
  }]
```



The **Fit** command can be used to fit higher-order polynomials by adding values to the second argument to generate a higher-order fitted equation. A third-order polynomial can be found as follows.

```
fit2 = Fit[largeCountriesCellAndInternet, {1, x, x^2, x^3}, x];
Show[{
    ListPlot[largeCountriesCellAndInternet],
    Plot[fit2, {x, 10^5, 10^8}, PlotStyle → Red]
  }]
```



The higher-order polynomial still looks linear in nature, which might mean that the linear fit provides as much information about the dataset as a higher-order polynomial, or that the data is too messy to easily describe with a polynomial.

This series of calculations can be used with **Manipulate** to explore several fits for the dataset by increasing the number of degrees for the polynomial fit. The **Range** command can be used to generate a simple list of integers, but it can also be used to generate a list of variables for the second argument of the **Fit** command.

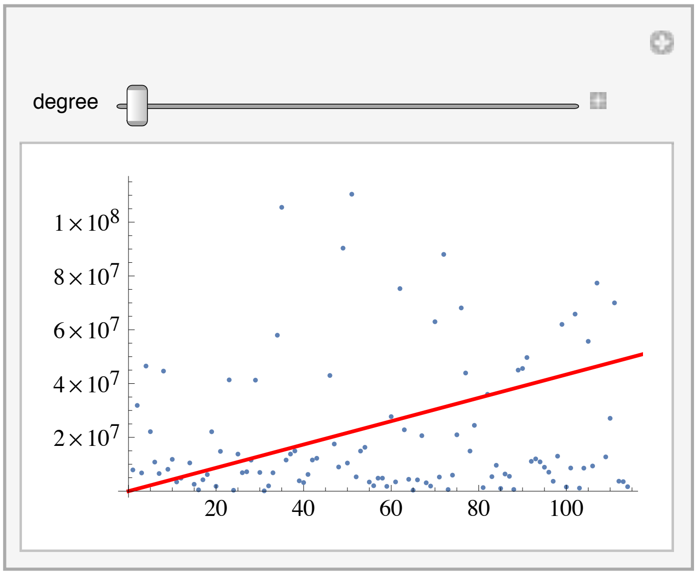
```
x ^ Range[0, 4]
```

$\{1, x, x^2, x^3, x^4\}$

Range is a command that can be used to quickly construct a list of numbers. If only a lower bound and upper bound are given, then a step size of 1 is used. A third argument can be passed to change the step size.

By setting the second argument in the **Range** function call as a manipulable parameter, **Manipulate** will create an interactive model that allows the user to increase or decrease the degree of the polynomial fit. The rest of the syntax is borrowed from the previous evaluations.

```
Manipulate[
  fit3 = Fit[largeCountriesCellAndInternet[[All, 1]], x^Range[degree], x];
  Show[{
      ListPlot[largeCountriesCellAndInternet[[All, 1]]],
      Plot[fit3, {x, 0, 150}, PlotStyle → Red]
    }],
  {degree, 1, 6, 1}]
```



Chapter 20: Data Filtering and Manipulation outlined filtering data with a **Manipulate** example. The two concepts can be combined as needed. For example, the **Manipulate** statement above could contain a second slider that displays only data points that satisfy a certain threshold of internet users or cellular phones, which would in turn recalculate a fitted equation based on this subset of the data.

In cases where a nonlinear fit is desired, **FindFit** can provide more flexibility in the form of the fitted equation. **FindFit** takes four arguments rather than three: the first is the data to be fitted, the second is the expression to fit to, the third is the best-fit parameters and the fourth is the variable in question. **FindFit** returns a list of real values for each of the coefficients as a list of rules.

For the example in question, **FindFit** can be used to find the curve of best-fit between the expression **a Log[x] + b** and the given data.

```
fit4 = FindFit[largeCountriesCellAndInternet⟦All, 1⟧, a Log[x] + b, {a, b}, x]
```

```
{a → 616 892., b → 3.14778 × 10^7}
```

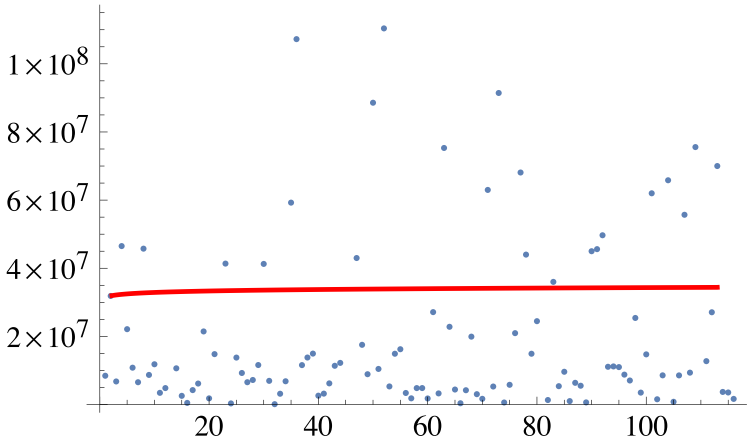
FindFit returns the parameter values in a list of rules. A different chapter discusses how to extract values from lists of rules, but the **ReplaceAll** command can be used to substitute the calculated values for **a** and **b** into the particular format for the fitted equation.

```
ReplaceAll[a Log[x] + b, fit4]
```

```
3.14778 × 10^7 + 616 892. Log[x]
```

As before, **Show** can be leveraged to visualize the raw data and the newly fitted curve.

```
Show[{
    ListPlot[largeCountriesCellAndInternet⟦All, 1⟧],
    Plot[ReplaceAll[a Log[x] + b, fit4], {x, 0, 113}, PlotStyle → Red]
  }]
```



Building Statistical Models

Fit and **FindFit** are designed to return expressions in the form of an equation corresponding to best fits of the data, and each command has parameters to customize the format for the fitted equation. Mathematica also includes commands to build statistical models based on raw data. Rather than giving a single expression or list of parameter values as output, creating a fitted model object gives convenient access to the fit as well as a variety of diagnostic information. An observant reader may notice that the structure returned by model-fitting commands is very similar to Mathematica's curated data commands. For example, any fitted model can be queried for additional properties.

One of the most common functions to build a model is **LinearModelFit**, which has a syntax similar to **Fit**.

```
model = LinearModelFit[QuantityMagnitude[largeCountriesCellAndInternet], {1, x, x^2}, x]
```

```
FittedModel[ -1.39151 × 10^6 + ≪19≫ x - 7.33838 × 10^-11 x^2 ]
```

Rather than returning just an expression as the default output, the output above is an object used to represent the fitted model in Mathematica. By assigning this fitted model to the variable named **model**, additional information about the model can be retrieved.

In many cases, the functional form of the fit is desired, and the **Normal** command can be used to obtain this form.

```
Normal[model]
```

```
-1.39151 × 10^6 + 0.443746 x - 7.33838 × 10^-11 x^2
```

The fitted model can act as a function to calculate the value at a particular point.

```
model[10^7]
```

```
3.03861 × 10^6
```

The fitted model can be used with **Table** to generate a list of values based on the fitted equation.

```
Table[model[i], {i, 10^6, 10^7, 10^6}]
```

```
{-947840., -504314., -60935.4, 382297., 825382., 1.26832×10^6,
  1.71111×10^6, 2.15376×10^6, 2.59626×10^6, 3.03861×10^6}
```

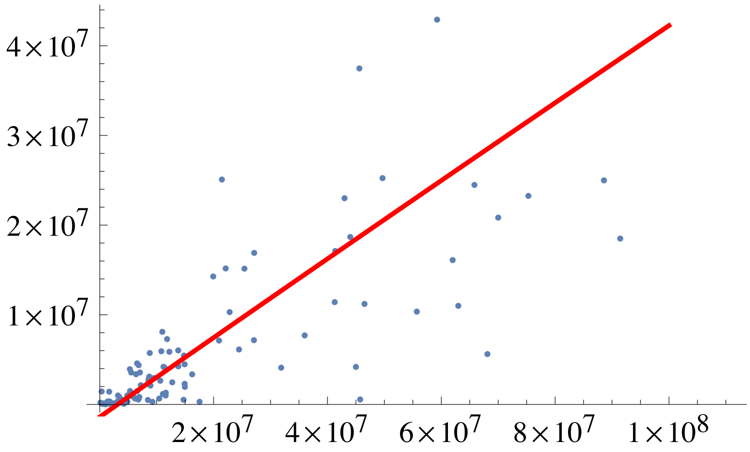
The fitted model can also be used with other commands, like **Integrate**.

```
∫model[x] ⅆx
```

```
-1.39151×10^6 x + 0.221873 x^2 - 2.44613×10^-11 x^3
```

Since the fitted model from **Fit** or **FindFit** is defined as a function, the **Show** command can be used to combine a plot of data with a **Plot** statement that includes the fitted model.

```
Show[ListPlot[largeCountriesCellAndInternet],
  Plot[model[x], {x, 10^5, 10^8}, PlotStyle → Red]]
```



The previous result looks the same as it did when using the **Fit** function earlier in the chapter. So why use **LinearModelFit**? The difference is that **LinearModelFit** provides a framework to quickly look up related information for the fit.

The syntax to display available statistical properties is similar to querying the available properties for a curated data command like **ChemicalData**. The following displays the first 10 statistical tests that can be displayed for the fitted model.

```
Short[model["Properties"], 10]
```

```
{AdjustedRSquared, AIC, AICc, ANOVATable, ANOVATableDegreesOfFreedom,
  ANOVATableEntries, ANOVATableFStatistics, ANOVATableMeanSquares,
  ANOVATablePValues, ANOVATableSumsOfSquares, ≪44≫,
  SequentialSumOfSquares, SingleDeletionVariances, SinglePredictionBands,
  SinglePredictionConfidenceIntervals, SinglePredictionConfidenceIntervalTable,
  SinglePredictionConfidenceIntervalTableEntries, SinglePredictionErrors,
  StandardizedResiduals, StudentizedResiduals, VarianceInflationFactors}
```

Any of these properties can be accessed for the model with a simple function call, like the associated ANOVA table.

```
model["ANOVATable"]
```

	DF	SS	MS	F-Statistic	P-Value
x	1	1.03981×10^{17}	1.03981×10^{17}	363.624	4.11185×10^{-37}
x^2	1	9.72784×10^{13}	9.72784×10^{13}	0.340185	0.560885
Error	113	3.23131×10^{16}	2.85957×10^{14}		
Total	115	1.36391×10^{17}			

Having easy access to all this information makes **LinearModelFit** a very useful command, since once the model is constructed, getting diagnostics like fit residuals and mean prediction errors is instantaneous.

```
Short[model["FitResiduals"], 5]
```

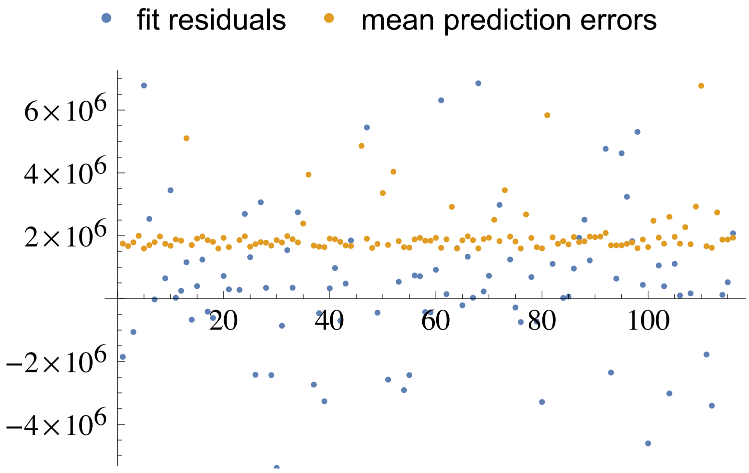
```
{-1.8529 × 10^6, -8.57657 × 10^6, -1.06061 × 10^6,
  -7.87592 × 10^6, ≪109≫, 120 658., 522 015., 2.07831 × 10^6}
```

```
Short[model["MeanPredictionErrors"], 5]
```

```
{1.75122 × 10^6, 1.67257 × 10^6, 1.79142 × 10^6,
  ≪110≫, 1.87564 × 10^6, 1.88041 × 10^6, 1.93877 × 10^6}
```

The previous lists can be plotted together to visualize error analysis and goodness of fit between the raw data and the fitted model.

```
ListPlot[
  {model["FitResiduals"], model["MeanPredictionErrors"]},
  PlotLegends → Placed[{"fit residuals", "mean prediction errors"}, Top]]
```

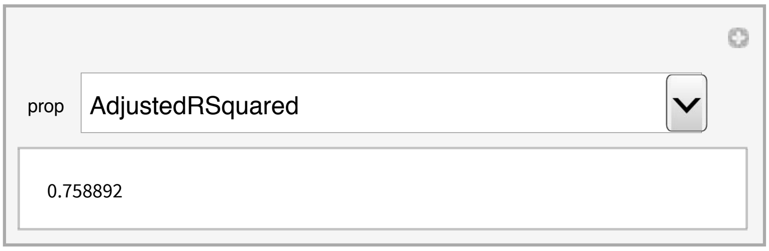


Because the fitted model can be investigated programmatically, it is very easy to create a dynamic model with **Manipulate** to allow interactive exploration of various properties.

```
LinearModelFit[QuantityMagnitude[largeCountriesCellAndInternet], {1, x, x^2},
  x]
```

```
FittedModel[ -1.39151×10^6 + ≪19≫ x − 7.33838×10^-11 x^2 ]
```

```
model = LinearModelFit[QuantityMagnitude[largeCountriesCellAndInternet], {1, x, x^2}, x];
Manipulate[model[prop], {prop, model["Properties"]}, SaveDefinitions → True]
```



prop AdjustedRSquared

0.758892

LinearModelFit is one of several model-fitting functions in Mathematica. The others include **GeneralizedLinearModelFit**, **NonlinearModelFit**, **LogitModelFit** and **ProbitModelFit**. The overall structure for the models returned by each command is similar, allowing additional diagnostics to be computed once the model is constructed.

Random Number Generation

Several commands are available for generating a list of pseudorandom numbers, as well as generating numbers for built-in distributions or sampling from lists.

The **RandomReal** and **RandomInteger** commands generate lists of uniformly distributed pseudorandom reals or integers, respectively. Each command can be used without arguments, which will generate a single random value of the appropriate type between 0 and 1.

```
SeedRandom["CKM"];
RandomReal[]
```

```
0.0229577
```

SeedRandom is used to reset the random number generator, but seeding a particular value will mean that the sequence of random numbers generated by a function will be the same. If you evaluate the input cell above, you will get the same set of pseudorandom numbers that the authors did when writing this book.

```
RandomInteger[]
```

```
0
```

When a single argument is used, each command will generate a pseudorandom number between zero and the specified value. In this case, the output will be a single random real or single random integer between 0 and 5.

```
RandomReal[5]
```

```
1.85405
```

```
RandomInteger[5]
```

```
2
```

If a lower bound other than zero is required, a minimum and maximum can be specified for the random number generation.

```
RandomReal[{3, 5}]
```

```
3.9661
```

```
RandomInteger[{3, 5}]
```

```
4
```

Since generating a list of random numbers is a common operation, a second argument to these commands allows the user to specify where a vector or array of numbers is desired. For example, 10 random real numbers between 1 and 6 can be generated.

```
RandomReal[{1, 6}, 10]
```

```
{1.25542, 4.83004, 1.65316, 4.86, 2.40025, 2.76463, 5.47558, 5.11047, 5.33032, 4.06133}
```

Or a 3 × 5 matrix of random integers between 0 and 10 can be generated.

```
RandomInteger[{0, 10}, {3, 5}]
```

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

It is worth pointing out that it is not possible to use a single random number generation command to create a multidimensional list whose elements follow different specifications. For example, suppose the goal is to construct a 3 × 5 matrix with row 1 containing random reals between 0 and 1, row 2 containing random reals between 1 and 2, and row 3 containing random reals between 2 and 3. For a situation like this, a good strategy is to construct each row independently and then use list manipulation commands to tie them together into the desired form.

```
MatrixForm[{
    RandomReal[{0, 1}, {4}],
    RandomReal[{1, 2}, {4}],
    RandomReal[{2, 3}, {4}]}]
```

```
( 0.801286   0.784834   0.919601   0.168855 )
(  1.91369    1.80112    1.37765    1.76867  )
(  2.23262    2.00886    2.39634    2.72117  )
```

Random sampling with or without replacement can be achieved through the use of **RandomChoice** and **RandomSample**.

```
vals = Range[5]
```

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

```
RandomChoice[vals, 3]
```

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

```
RandomSample[vals, 3]
```

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

RandomChoice does sampling with replacement, allowing more samples to be taken than the number of available choices.

```
RandomChoice[vals, 10]
```

```
{3, 3, 5, 3, 2, 5, 2, 5, 3, 5}
```

RandomSample does sampling without replacement, and the command will fail if a user tries to sample more items than are available.

```
RandomSample[vals, 10]
```

```
    ••• RandomSample: RandomSample cannot generate a sample of length
          10, which is greater than the length of the sample set {1, 2, 3, 4, 5}. If
          you want a choice of possibly repeated elements from the set, use
          RandomChoice.
RandomSample[{1, 2, 3, 4, 5}, 10]
```

Since there are only five elements in the list stored as the variable **vals**, 5 is the largest value that can be specified as a second argument to **RandomSample**.

```
RandomSample[vals, 5]
```

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

The following command creates a list of 10 values, with each value corresponding to the mean of randomly sampling five integers from 1 to 100. As expected, the mean values vary quite a bit.

```
vals = Table[N[Mean[RandomChoice[Range[100], 5]]], {i, 1, 10, 1}]
```

```
{64., 74.6, 47.8, 28.6, 66.6, 37.8, 52.8, 51.4, 56.4, 23.4}
```

But the mean of the means is close to 50.

```
Mean[vals]
```

```
50.34
```

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[popSelect, largeCountries, largeCountriesCellAndInternet, fit1, fit2, fit3,
  fit4, model, vals]
```

Conclusion

Mathematica's statistical capabilities are nicely integrated with the rest of the system, so once a dataset is imported and manipulated, statistics commands are available to perform the required analysis, whether that is computation of measures, curve fitting or random sampling.

Exercises

1. Use free-form input to generate a random integer between 1 and 100.
2. Use the Wolfram Language to create a list of five random integers between 1 and 100.
3. Use the Wolfram Language to write a two-statement program, where the first statement creates a table of integers, in order from 1 to 10. Suppress the output from that statement. The second statement should use the output from the first statement to construct a random sample of 10 choices from the output. The result of this two-statement program will be a random ordering of the integers.
4. It might be theorized that a correlation may exist between GDP per capita and government debt when looking at countries with a large number of television stations. Create a function named **tvSelect** that takes a list of countries as its input and returns any countries that have more than 100 television stations.

5. Create a variable named **manyStations** to hold the names of all of the countries in the world, and use this variable with the function created in Exercise 4 to find a comprehensive list of countries that have more than 100 television stations.
6. Create a table of values of the form {*GDP per capita, government debt*} for all the countries that are stored in **manyStations**. Store this table of values in a new variable named **manyStationsGDPGovDebt**.
7. Since the result in Exercise 6 was relatively short, it was easy to see the number of countries with over 100 television stations. In the event that the list was much longer, a more efficient approach would be to find the dimensions of the dataset. Do this now.
8. Use **ListLogLogPlot** to visualize the data from Exercise 6, and use the **Tooltip** command so that mousing over each data point shows the name of the associated country. (Hint: use the variable definition from Exercise 6 instead of the variable name directly, since that makes it easier to work with **Tooltip** in this example.)
9. Find the average value for household consumption for these countries by using **Part** to extract the first value for each element of the list stored in **manyStationsGDPGovDebt**.
10. The plot from Exercise 8 did not provide compelling visual evidence that the correlation hypothesis was correct. Use the **Correlation** command and apply it to all the first values in the list stored in **manyStationsGDPGovDebt** and all the second values in the list stored in **manyStationsGDPGovDebt**.

CHAPTER 24

Creating Programs

Introduction

Many examples in this book have involved multiple computations that lead to a single result, like a visualization. Mathematica is commonly used in this manner to test hypotheses and to get a feel for whether a project or idea is worth further exploration. Mathematica's interpretive nature allows this type of exploration to be done efficiently and immediately without needing to write a lot of code, compile a program and then execute the program to see the results.

Encapsulating ideas into function definitions lets users access the functionality without dwelling on exactly how it was built with the Wolfram Language, keeping the focus on the application and not on the code. Previous chapters have included examples of commands that allow programmatic generation of lists, and user-defined functions have been used to illustrate several programming concepts. This same idea of function definitions also serves as a basic building block to programming in the Wolfram Language. This chapter will further show the scope of functionality for user-defined functions and will explore looping constructs in more detail, with an eye toward creating functions that can be shared with and used by others who do not need to have any understanding of the underlying code.

Programming Basics

The following example illustrates the use of programming to do a repetitive task. **CountryData** is used to define a list consisting of a country's name, population and total land area. This type of task is well suited to Mathematica, even if the data needs to be massaged a bit, as will be detailed in this chapter.

Remember that a symbol that is displayed in blue *does not* have a stored value, and a symbol or function that is displayed in black *does* have a stored value. As you type in the following example, the variables will turn from blue to black as they are evaluated and receive definitions.

```
listOfAllCountries = CountryData[All];
countryList = Table[{i, CountryData[i, "Population"], CountryData[i, "LandArea"]},
    {i, listOfAllCountries}];
Short[countryList]
```

```
{{ Afghanistan , 35 623 235 people , 652 230. km^2 },
  { Albania , 3 248 655 people , 27 398. km^2 },
  { Algeria , 37 473 690 people , 2.38174 × 10^6 km^2 },
  ≪234≫, { Yemen , 27 162 547 people , 527 968. km^2 },
  { Zambia , 14 767 550 people , 743 398. km^2 },
  { Zimbabwe , 13 665 123 people , 386 847. km^2 }}
```

A third argument can be passed to the **CountryData** command to determine the units of measurement for population and land area.

```
CountryData["Afghanistan", "Population", "Units"]
```

```
People
```

```
CountryData["Afghanistan", "LandArea", "Units"]
```

```
SquareKilometers
```

In this particular example, it is desirable to represent population density in terms of number of people per acre, not number of people per square kilometer. The **UnitConvert** command can be used to find a suitable conversion factor from square kilometers to acres.

```
UnitConvert[Quantity[1, "Kilometers"^2], "Acres"] // N
```

```
247.104 acres
```

The // allows a command to be applied to a result before it is displayed. The preceding example uses // to return a numeric approximation and is equivalent to wrapping the **N** command around the **UnitConvert** statement.

Now that the conversion rate is known, it can be used directly. A new list is created, consisting of the previous information (country name, population and total land area), and the conversion factor is applied to calculate the population density in the desired units of number of people per acre.

```
countryDensity = Table[
        Flatten[{country⟦1 ;; 3⟧, UnitConvert[(country⟦2⟧/country⟦3⟧), ("people/acre")]}],
        {country, countryList}
    ];
Style[
    countryDensity⟦1 ;; 10⟧, FontSize → 10] // TraditionalForm
```

```
⎛        Afghanistan      35 623 235 people     652 230. km²        0.22103 people/acre   ⎞
⎜          Albania         3 248 655 people      27 398. km²       0.479849 people/acre   ⎟
⎜          Algeria        37 473 690 people  2.38174 × 10⁶ km²   0.0636724 people/acre   ⎟
⎜      American Samoa         54 719 people        199. km²        1.11277 people/acre   ⎟
⎜          Andorra            85 458 people        468. km²       0.738969 people/acre   ⎟
⎜          Angola         21 274 503 people   1.2467 × 10⁶ km²   0.0690585 people/acre   ⎟
⎜          Anguilla           16 086 people         91. km²       0.715363 people/acre   ⎟
⎜    Antigua and Barbuda      91 295 people        442.6 km²       0.834747 people/acre   ⎟
⎜         Argentina       41 827 217 people  2.73669 × 10⁶ km²   0.0618519 people/acre   ⎟
⎝          Armenia         3 125 790 people      28 203. km²       0.448522 people/acre   ⎠
```

Recall that ⟦ ⟧ is a special typeset representation of the **Part** command. In the preceding example, **country** is the iterator for the **Table** command and is replaced by a different country from **countryList** at each step. Double semicolons represent a span, which in this example is comprised of the elements in the first through the third positions in the list. Finally, **Style** is used to set the font size in order to fit the output on the page, and **TraditionalForm** is used to clean up the presentation of the results.

To make this a more general-purpose example, this calculation can be defined as a user-defined function rather than a variable definition. The following program takes a list as its input and then returns the first through the third elements of the list, as well as a new element constructed from adding the second and third elements together.

In the preceding variable definition, the program returned population plus land area for each country as a new fourth element in the list. In the user-defined function, the list can be any expression; the function will add the second and third elements regardless of their value. That makes the program useful, since it can be referenced in other examples for this general type of list creation.

```
addSecondThird[x_] :=
  Table[
    Flatten[
      {country⟦1 ;; 3⟧,
        (QuantityMagnitude[country⟦2⟧] + QuantityMagnitude[country⟦3⟧])}],
    {country, x}
  ]
```

Once this program is defined, it can be run just like any other Wolfram Language function. The following example repeats the above calculation but uses the function to run the calculation with a dataset. The output is identical to the preceding in terms of dimensions but obviously not content.

Recall that the original dataset is a list of country name, population and land area for each country in the world. The **Part** command is used to show the first three elements to preview the dataset without printing its entire contents.

```
countryList⟦1 ;; 3⟧
```

```
{{ Afghanistan , 35 623 235 people , 652 230. km^2 },
  { Albania , 3 248 655 people , 27 398. km^2 },
  { Algeria , 37 473 690 people , 2.38174 × 10^6 km^2 }}
```

The user-defined function can now take this dataset as input and will add the second and third columns for each country. As before, the **Part** command is only used to preview the output; the program completes the operation on all the countries in the list.

```
addSecondThird[countryList]⟦1 ;; 3⟧
```

```
{{ Afghanistan , 35 623 235 people , 652 230. km^2 , 3.62755 × 10^7},
  { Albania , 3 248 655 people , 27 398. km^2 , 3.27605 × 10^6},
  { Algeria , 37 473 690 people , 2.38174 × 10^6 km^2 , 3.98554 × 10^7}}
```

A program can have additional commands wrapped around it. For example, **Prepend** can add a sublist to the beginning of the list, and this sublist can contain strings that will serve as labels for each column.

```
Prepend[addSecondThird[countryList]⟦1 ;; 3⟧,
    {"Country", "Pop.", "Land Area", "Pop. + Land Area"}] // TraditionalForm
```

$$\begin{pmatrix} \text{Country} & \text{Pop.} & \text{Land Area} & \text{Pop. + Land Area} \\ \text{Afghanistan} & 35\,623\,235\ \text{people} & 652\,230.\ \text{km}^2 & 3.62755\times 10^7 \\ \text{Albania} & 3\,248\,655\ \text{people} & 27\,398.\ \text{km}^2 & 3.27605\times 10^6 \\ \text{Algeria} & 37\,473\,690\ \text{people} & 2.38174\times 10^6\ \text{km}^2 & 3.98554\times 10^7 \end{pmatrix}$$

Now that this function is defined, it can be used with any dataset with little thought about the **Table** and **Part** commands that do the actual work. The following example builds a list containing a country name, adult population and water area.

```
listOfAllCountries = CountryData[All];
countryList2 =
    Table[{i, CountryData[i, "AdultPopulation"], CountryData[i, "WaterArea"]},
      {i, listOfAllCountries}];
countryList2⟦1 ;; 10⟧ // TraditionalForm
```

$$\begin{pmatrix} \text{Afghanistan} & 1.80536\times 10^7\ \text{people} & 0.\ \text{km}^2 \\ \text{Albania} & 2.22943\times 10^6\ \text{people} & 1350.\ \text{km}^2 \\ \text{Algeria} & 2.54063\times 10^7\ \text{people} & 0.\ \text{km}^2 \\ \text{American Samoa} & 38\,337.\ \text{people} & 0.\ \text{km}^2 \\ \text{Andorra} & 60\,280.\ \text{people} & 0.\ \text{km}^2 \\ \text{Angola} & 1.07557\times 10^7\ \text{people} & 0.\ \text{km}^2 \\ \text{Anguilla} & 10\,765.\ \text{people} & 0.\ \text{km}^2 \\ \text{Antigua and Barbuda} & 61\,414.\ \text{people} & 0.\ \text{km}^2 \\ \text{Argentina} & 2.69058\times 10^7\ \text{people} & 30\,200.\ \text{km}^2 \\ \text{Armenia} & 2.13897\times 10^6\ \text{people} & 1540.\ \text{km}^2 \end{pmatrix}$$

Since printing this book involves a specific page width, several of the tables in this chapter involve extra formatting with the printed page in mind. When you use Mathematica on your own machine, you have the flexibility to resize the notebook to accommodate the display of larger datasets with more columns.

This new list can be passed to the function to create a new fourth column by adding the second and third columns.

```
addSecondThird[countryList2][[1 ;; 5]]
```

```
{{ Afghanistan , 1.80536×10^7 people , 0. km^2 , 1.80536×10^7},
 { Albania , 2.22943×10^6 people , 1350. km^2 , 2.23078×10^6},
 { Algeria , 2.54063×10^7 people , 0. km^2 , 2.54063×10^7},
 { American Samoa , 38337. people , 0. km^2 , 38337.},
 { Andorra , 60280. people , 0. km^2 , 60280.}}
```

This sort of workflow is common when creating the major parts of a project. If one person builds a function or set of functions, others who work on the project only need to take the time to learn what the functions do and what input parameters are required; they do not need to learn precisely how the functions work. This approach also helps a user solve a problem once and create a solution that can be used in the future when the problem arises again.

Local and Global Variables

Many examples in this book have used variables to store values for a variety of expressions, including numbers, strings and lists. Each of these variable assignments has created variables that are considered global by Mathematica. This means that if the variable is used in either the current notebook or another notebook that is linked to the same Mathematica session (i.e. a session using the same kernel), then Mathematica will recognize the value for that symbol.

Global variables are advantageous in many cases, since users can store commonly used variables in one section of a notebook, evaluate that section and then continue their work in a new section. Or users can keep a central notebook with common definitions, evaluate that notebook and then start another notebook to create a new project. There are times, however, when it is preferential to use variables that are only defined for the scope of a particular operation.

The use of the character **x** as a variable is a good example: at one point in a notebook, the defined variable **x** could be a list, and at another point in the notebook, **x** could be used as a symbol in a function argument. To avoid this conflict, **x** can be defined as a local variable so that its definition does not travel outside the intended scope.

The following example shows global variable assignment.

```
x = 5
```

```
5
```

Now that **x** has been defined, any instances of **x** will be immediately replaced by its current value, 5. Sometimes this is desired behavior, like when this variable needs to be referenced by a calculation.

```
2 x
```

```
10
```

However, this can also cause problems. For example, if **x** is used as the variable for an integration, the integration will fail, because **x** is already defined.

```
∫Sin[x] dx
```

```
••• Integrate: Invalid integration variable or limit(s) in 5.
∫Sin[5] d5
```

Global variable definitions can be cleared with the **Clear** command. This command will remove the assigned value from the symbol in question, and this clearing will apply to any Mathematica sessions using the same kernel as the one used to evaluate **Clear**.

```
Clear[x]
```

> This is the second time we have mentioned the kernel that a particular notebook is using. For many users, this is not a topic that needs to be given much thought. When Mathematica is launched, a kernel is started, and that kernel is what is used for all calculations unless Mathematica is given different instructions, like to launch and use a kernel on a remote machine. If you plan on using your own machine for your work, then you can ignore the notes about what kernels a notebook is using.

Once the value of **x** is cleared, it evaluates strictly as a symbol.

```
2 x
```

```
2 x
```

There are several ways to create local variables, and use of the function **Module** is the most common. **Module** takes two arguments: the first is a list of variables that are local to that statement, and the second is a calculation or list of operations. In this case, **x** is given a value of 5 for the multiplication calculation within the **Module** statement.

```
Module[{x = 5}, x*5]
```

```
25
```

However, this assignment of value to **x** does not travel outside of the **Module** command. To the rest of the Mathematica session, **x** appears as just a symbol, so multiplying 2 *x* will return the same value as the input.

```
2 x
```

```
2 x
```

This means that **x** can continue to be used for other purposes, such as the variable for an integration command, without triggering the clashing problems that were seen earlier.

```
∫Sin[x] ⅆx
```

```
−Cos[x]
```

> Using unique variable names is always a good idea, but it is likely that you will use words like data, steps, time or other common words in several programs. In such cases, using **Module** to localize variables can serve as extra protection to prevent two programs from inadvertently interacting with each other and causing undesired results.

The use of **Module** and similar approaches to create local variables is a good habit. One distinction that is important to note is that it is not necessary to define a local variable when defining a user-defined function. When using delayed assignment and pattern matching, the symbol used for pattern matching does not need to be declared as a local variable.

As an example, first set the value of **x** to be 5.

```
x = 5
```

```
5
```

Now define a function that squares its argument.

```
f[x_] := x^2
```

```
f[10]
```

```
100
```

The variable **x**, which is currently defined as having the value 5, is treated differently than the pattern **x_**. In this case there was no need to use **Module** to "protect" the function definition from being corrupted by an existing definition for **x**.

The symbol **f**, on the other hand, now has a specific definition: a delayed assignment that can be used to evaluate the function for a particular value or list of values. This means that the symbol **f** will need to be cleared if **f** is used elsewhere in a notebook or Mathematica session as a global variable.

```
?f
```

```
Global`f
```

```
f[x_] := x^2
```

```
Clear[f, x]
```

Undefined symbols do not have any definitions.

```
?f
```

```
Global`f
```

```
?x
```

```
Global`x
```

Multiparadigm Programming Language

Many examples thus far have used **Table** to create lists of various values and forms. This command is extremely efficient and in many cases is the optimal command to use when creating lists. However, since the Wolfram Language is multiparadigm, it supports constructs and approaches from many different styles of programming. As a result, commands like **Do**, **While** and **For** can also be used for creating lists through looping. For those with procedural backgrounds, these commands may be old friends, but it is strongly encouraged that functional alternatives, like **Table**, be explored and understood as well.

Do can be used to evaluate an expression multiple times, but it does not create any output. As a result, the **Print** command is sometimes used as part of the body of the expression in order to show results when using **Do**.

```
Do[Print[x], {x, 1, 5}]
```

```
1
2
3
4
5
```

Do can also be used to create lists. A typical approach is to define an empty list and then append elements to that list based on iterative evaluations of an expression. **AppendTo** is used so that the same list will have new values added to it each time the expression is looped through the iteration.

```
myList = {};
Do[
  AppendTo[myList, x],
  {x, 1, 5}]
```

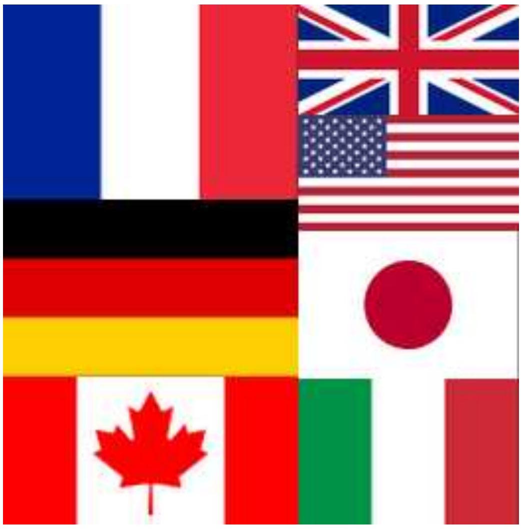
```
myList
```

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

Just like the **Table** command, **Do** can iterate over an arbitrary list of values. For example, if a list of countries is created, then the list of countries can be iterated over.

```
countryList = CountryData["G7"];
myCountryFlagList = {};
Do[
  AppendTo[myCountryFlagList, CountryData[x, "Flag"]],
  {x, countryList}]
```

```
ImageCollage[myCountryFlagList]
```



Another command that can be thought of as a looping construct is **Map**. While **Table** and **Do** have been primarily used to create lists in the examples in this book, **Map** operates on existing lists to modify their elements as desired. **Map** takes two arguments: the first is a function and the second is an expression—which could be a list—on which to apply that function.

For example, define a list as follows.

```
eList = Range[5]
```

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

Now suppose the goal is to compute the factorial of each of these numbers. **Map** provides an easy way to do this.

```
Map[Factorial, eList]
```

```
{1, 2, 6, 24, 120}
```

Because **Map** is so commonly used, it uses **/@** as its shorthand form, and this shorthand may be encountered frequently when browsing Wolfram Language programs.

```
Factorial /@ eList
```

```
{1, 2, 6, 24, 120}
```

Note that **eList** could have just as easily been constructed with a **Table** command.

```
Table[Factorial[i], {i, 5}]
```

```
{1, 2, 6, 24, 120}
```

While commands like **Do** and **Loop** can be used to procedurally create lists of values, commands that take a functional approach, like **Table** and **Map**, are usually more efficient and take less time to evaluate.

Another looping construct is the **For** command, which allows a user to define a test to specify the number of times an expression or series of expressions will evaluate. Following is the basic syntax of **For**.

```
?For
```

For[*start*, *test*, *incr*, *body*] executes *start*, then repeatedly evaluates *body* and *incr* until *test* fails to give True. ≫

```
For[i = 1, i ≤ 5, i++, Print[i]]
```

```
1
2
3
4
5
```

Since the **For** command takes four arguments, when those arguments are long or complicated, it can be good practice to place each of them on a separate line for the sake of readability. This structure makes it very easy to differentiate between the initialization, test, incrementation and loop body parameters of the function.

```
For[i = 1,
  i ≤ 5,
  i++,
  Print[i]]
```

```
1
2
3
4
5
```

The same example to create a list of the flags of the Group of 7 can be created using **For**, although this example compresses the program definition, execution and display of results into a single cell.

```
g7List = CountryData["G7"];
myFlagList = {};
For[i = 1,
  i ≤ Length[g7List],
  i++,
  AppendTo[myFlagList, CountryData[g7List⟦i⟧, "Flag"]]]
ImageCollage[myFlagList]
```

While is another looping function that also uses a test to determine whether evaluation should stop or continue. Unlike **For**, which uses an iterator and is often used to loop through a particular list of values, **While** can use an evaluation test that is more general and is not tied to keeping track of particular values.

```
rn = 1;
While[rn ≤ 3, Print[rn]; rn = rn + 1]

    1
    2
    3
```

For those who are used to updating a variable assignment by using **+=** for adding and **-=** for subtracting, you will be happy to know that those operations are supported, along with ***=** and **/=** for multiplication and division. Using the expression **var += 1** will add 1 to the value stored in **var**.

While could be used to construct the same list of flags for the Group of 7 countries, although the approach is different than before. Rather than iterating over a list of values, an empty list is initialized and used to store the results, and a list of the countries is emptied, one-by-one, as the flag for each country is found. This allows the test for the **While** command to be whether the list of countries has been exhausted, meaning the program is complete and should end.

```
g7List = CountryData["G7"];
myFlagList = {};
While[
  g7List ≠ {},
  AppendTo[myFlagList, CountryData[First[g7List], "Flag"]];
  g7List = Rest[g7List]
]
ImageCollage[myFlagList]
```

Since the preceding examples all have the same output, it might not be clear when to use **Map**, **Table**, **For** or **While**. That is one of the great benefits of having Mathematica at your disposal: use the approach that fits your needs.

Another command that is sometimes used for looping is **Nest**. The **Nest** command is used to return the result of applying a single function or operation to an expression multiple times.

```
f[x_] := x + 1
Nest[f, 1, 5]
```

```
6
```

The difference between **Nest** and the other commands discussed in this chapter is significant. Rather than sending successive elements in a list through a loop for calculation, **Nest** applies the same function to an expression over and over again with a single invocation. For example, to calculate the result of investing $10,000 in an annual CD that gives 2% interest, and reinvesting the result in the same CD, **Nest** proves invaluable.

```
f[x_] := x*1.02
Nest[f, 10 000, 10]
```

```
12 189.9
```

Nest can also be used with lists, allowing the same series of repeated evaluations to be applied to multiple datasets.

```
Nest[f, {500, 10 000, 250 000}, 10]
```

```
{609.497, 12 189.9, 304 749.}
```

A matching command, **NestList**, can be used to see all the intermediate values for the calculations.

```
NestList[f, 10 000, 10]
```

```
{10 000, 10 200., 10 404., 10 612.1, 10 824.3,
  11 040.8, 11 261.6, 11 486.9, 11 716.6, 11 950.9, 12 189.9}
```

In the case of using **NestList** with multiple starting values, the intermediate calculations for each step will be grouped together.

```
NestList[f, {500, 10 000, 250 000}, 10] // TableForm
```

```
500        10 000     250 000
510.       10 200.    255 000.
520.2      10 404.    260 100.
530.604    10 612.1   265 302.
541.216    10 824.3   270 608.
552.04     11 040.8   276 020.
563.081    11 261.6   281 541.
574.343    11 486.9   287 171.
585.83     11 716.6   292 915.
597.546    11 950.9   298 773.
609.497    12 189.9   304 749.
```

Pattern Matching

Other chapters have shown a few specific examples of pattern matching, but this functionality is extremely powerful in Mathematica and can be used for many types of tasks. The **ReplaceAll** command can be used to replace symbols or patterns with other expressions, either by invoking its proper command name or by using **/.** as a shortcut notation. **ReplaceAll** works by substituting expressions based on a list of rules.

```
ReplaceAll[a x^2, a → 4]
```

```
4 x^2
```

```
a x^2 /. a → 4
```

```
4 x^2
```

ReplaceAll can replace multiple values at one time if given a list of transformation rules.

```
a x^2 + b x + c /. {a → 2, b → 3, c → 4}
```

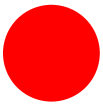
```
4 + 3 x + 2 x^2
```

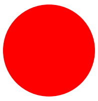
Transformation rules do not have to keep the data in the same structure as it was originally, which allows a lot of flexibility when working with lists. Instead of transforming symbols to values, a more general pattern can be specified and then used for manipulation. For example, a pattern might look for a list of three elements and then change their order, regardless of the type of elements in the list.

```
{a, b, c} /. {x_, y_, z_} → {y, z, x}
```

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

Rather than using the symbols **a**, **b** and **c**, the following list has an image, a string and π, which are reordered in exactly the same manner as the previous example that used only symbols.

{, "This is a string", π} /. {x_, y_, z_} → {y, z, x}

{This is a string, π, }

A pattern might take a list of three elements and add them together rather than reorder the list.

```
{a, b, c} /. {x_, y_, z_} → x + y + z
```

```
a + b + c
```

```
{1, 2, 3} /. {x_, y_, z_} → x + y + z
```

```
6
```

Or a pattern might be more complicated. The following example takes a list of three elements and creates a new list consisting of sublists of each element and its square, as well as a sublist consisting of the sum of the elements and that sum's square.

```
{a, b, c} /. {x_, y_, z_} → {{x, x^2}, {y, y^2}, {z, z^2}, {x + y + z, (x + y + z)^2}}
```

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

```
{1, 2, 3} /. {x_, y_, z_} → {{x, x^2}, {y, y^2}, {z, z^2}, {x + y + z, (x + y + z)^2}}
```

```
{{1, 1}, {2, 4}, {3, 9}, {6, 36}}
```

Patterns allow additional specifications so that only expressions with certain heads are matched. This means a pattern can be defined that only matches when it encounters an integer but not other values. The following replacement rule squares all the integer values in a list.

```
{2, 2.5, π} /. x_Integer → x^2
```

```
{4, 2.5, π}
```

Defining a pattern that matches specific heads allows a single function to have multiple definitions, each of which may correspond to a different type of argument being passed to it.

```
h[x_Integer] := x
h[x_Real] := x^2
h[x_Symbol] := x^3
h[x_String] := "This function does not operate on strings."
```

```
{h[1], h[1.1], h[π], h["test"]}
```

```
{1, 1.21, π^3, This function does not operate on strings.}
```

Conditional Functions

The Wolfram Language has conditional functions that control the flow of a program. The conditional functions specify a test, and additional arguments control what should happen as a result of the test. The following is the basic syntax for **If**, which is one of these conditional commands.

```
?If
```

If[*condition*, *t*, *f*] gives *t* if *condition*
evaluates to True, and *f* if it evaluates to False.

If[*condition*, *t*, *f*, *u*] gives *u* if *condition* evaluates to neither True nor False. >>

The first argument needs to be a test that evaluates to either true or false.

```
If[π > 3, "My test succeeded!", "My test failed."]
```

```
My test succeeded!
```

The test does not need to be a single expression but can be a more complicated, compound Boolean expression.

```
If[π > 3 && π > 5, "My test succeeded!", "My test failed."]
```

```
My test failed.
```

Another flow control command is **Which**. The **Which** command takes multiple tests and values, evaluating each test sequentially and returning the value following the first test that is satisfied. This means that even if multiple tests would have been satisfied, the evaluation terminates once a single successful test is passed.

```
Which[π > 5, "Greater than 5", π > 3, "Greater than 3", π > 1, "Greater than 1"]
```

```
Greater than 3
```

You can introduce line breaks to group tests and the operation that is performed once that test is successfully satisfied; this can help with readability for long or complex **Which** statements.

Since Mathematica's notebook interface allows text and programming in the same document, in many cases the text can serve as the documentation or explanation for programs contained therein. However, sometimes it is convenient to include comments within a block of programming commands to give the reader some explanation of what is happening. Comments can be placed in between the **(*** and ***)** symbols within input cells, and any input between those delimiters will be ignored.

```
myFunction[x_] :=
  If[x < 0, x, (* negative values remain the same *)
    x^2 (* positive values are squared *)]
```

```
{myFunction[-2], myFunction[2]}
```

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

> Comments can also be extremely useful when trying different approaches with a program, or when debugging a program. Rather than deleting a piece of code, you can comment it out to see what happens when it is removed. An easy way to comment a piece of code is to highlight it, click the **Edit** menu, and choose **Un/Comment Selection**. That menu also shows the keyboard shortcut for toggling comments.

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[listOfAllCountries, countryList, countryDensity, addSecondThird, countryList2,
  x, f, myList, eList, g7List, myFlagList, rn, h, myFunction]
```

Conclusion

The Wolfram Language is certainly not limited to the small number of functions outlined in this chapter, and many tasks can be accomplished using different techniques. The language is broad in constructs and supports a multiparadigm style, letting the user choose the approach that matches the style of how they think about a problem.

When a program is repetitive and involves a large dataset or an involved series of calculations, running the program on multiple CPUs or GPUs can speed up computation time. The next chapter outlines how to parallelize programs in Mathematica.

Exercises

1. Calculate the numerical approximation of the square root of 20 to three digits, and assign the result to the variable **b**.
2. Create a function named **f**, which takes a single argument **b** and returns the numerical approximation of the square root of **b** to three digits.
3. Create a table of values by using the function **f** from Exercise 2 with a list of the first 10 integers as inputs. Store this table of values in the variable **tab10**.
4. Define the local variable **g**, assign to it the value of $\mathbf{b}^2$ and then calculate **g + 1**.
5. Evaluate **b** and **g**. If the variable assignments from Exercises 1 and 4 have been performed correctly, then only **b** will return a numeric value.

6. Replace the third element of **tab10** with the number 10.
7. Calculate $x^2 + y - z$, where x is replaced with 5, y is replaced with 3 and z is replaced with 1.
8. Use **If** to create a conditional statement that returns the string "LOWER" if $b^3 < 95$ and "HIGHER" otherwise.
9. Use **Which** to create a conditional statement that examines the first three elements of **tab10** in order and returns the first element position for which the element value is greater than 1.
10. Write a two-statement program that first clears the definition for **b** and then calculates **b**$^{\mathbf{2}}$ to ensure that the variable no longer has a definition.

CHAPTER 25

Creating Parallel and GPU Programs

Introduction

Mathematica can perform many common calculations in a fraction of a second. For complex calculations or a series of calculations, Mathematica can take advantage of multiple CPU cores to decrease its computation time by delegating pieces of a calculation to be worked on by different units. The Mathematica controlling kernel acts as a delegator and automatically splits up a calculation by assigning pieces to computational kernels, which in turn each work on their piece of the calculation simultaneously. Mathematica can identify the quantity of CPU cores on a user's machine and launch the appropriate number of computational kernels as the first step of a parallel calculation, if those kernels have not already been explicitly launched. GPUs can be used in a similar manner to decrease computation time. Parallel computation is only available for desktop licenses of Mathematica, so the majority of the examples in this chapter will not work in Mathematica Online.

> Mathematica can also be used for parallel computation across multiple machines, from dedicated clusters to supercomputers, and it has built-in tools to create ad hoc grids from dormant machines. The examples in this chapter will show parallel computations on a single machine that has multiple CPU cores. You will need to run these calculations on a similarly configured machine, and your results may be a bit different, depending on the configuration of the machine that is used.

Using Multiple CPU Cores for a Single Calculation

While the details of parallel evaluation are usually cumbersome in other programming languages, the Wolfram Language commands automate many aspects of parallel programming. This allows the user to focus on the problem at hand and fluidly explore solutions instead of needing to spend time with the intricacies of partitioning a problem, passing information to CPU cores, collecting the output and organizing it into a cohesive result.

The commands used for parallel evaluation follow the same syntax as their standard Wolfram Language counterparts. This makes it straightforward to run a single evaluation on multiple CPU cores. **Table**, **Map** and **Evaluate** can be used on multicore machines by replacing them with **ParallelTable**, **ParallelMap** and **ParallelEvaluate**.

Evaluate and **ParallelEvaluate** both force an evaluation in the Mathematica kernel. **Evaluate** calculates an expression using a single CPU core, and **ParallelEvaluate** will carry out the calculation on multiple CPU cores, assuming they are available for use, and that the Mathematica license in question provides enough computational kernels for evaluation.

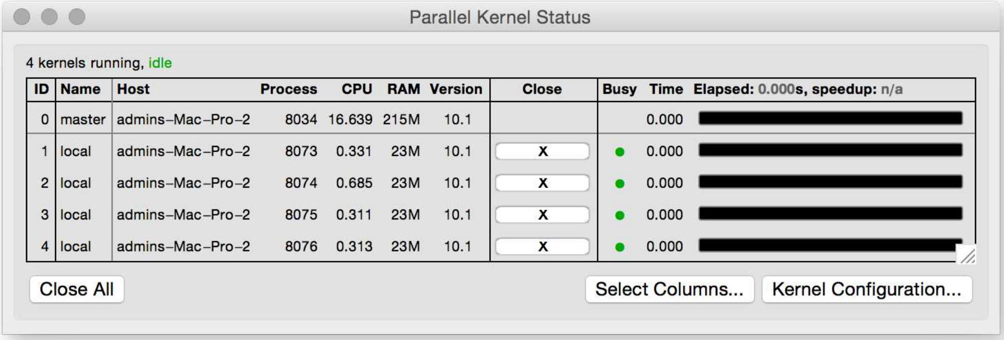
```
Evaluate[5 !]

120
```

```
ParallelEvaluate[5 !]

{120, 120, 120, 120}
```

The previous command automatically launched multiple computational kernels and then evaluated **5!** on each CPU. The result is a list of outputs from each computational kernel. Clicking the **Evaluation** menu and choosing **Parallel Kernel Status** opens a new window that shows the status of the master kernel and the quantity of the computational kernels that have been launched. In this case, there are four computational kernels that are each local to the multicore machine used for this evaluation.



4 kernels running, idle

ID	Name	Host	Process	CPU	RAM	Version	Close	Busy	Time	Elapsed: 0.000s, speedup: n/a
0	master	admins–Mac–Pro–2	8034	16.639	215M	10.1			0.000	
1	local	admins–Mac–Pro–2	8073	0.331	23M	10.1	X	●	0.000	
2	local	admins–Mac–Pro–2	8074	0.685	23M	10.1	X	●	0.000	
3	local	admins–Mac–Pro–2	8075	0.311	23M	10.1	X	●	0.000	
4	local	admins–Mac–Pro–2	8076	0.313	23M	10.1	X	●	0.000	

Mathematica notebooks are cross-platform compatible, so you can write a program on one machine and then run that program on a different machine that has more cores.

Additional information can be passed to the computational kernels to aid in execution. For example, the following command will set the value of the variable **a** in each computational kernel, and then each computational kernel will use its local definition of **a** to calculate $\mathbf{a}^2$.

```
ParallelEvaluate[a = 5; a^2]
```

```
{25, 25, 25, 25}
```

The results from each computational kernel are identical, since each one contains the same definition for the variable **a**. However, different definitions for variables or functions can be given to each computational kernel, as can be seen by evaluating the following expression.

```
ParallelEvaluate[a = RandomReal[]; a^2]
```

```
{0.373261, 0.788036, 0.322687, 0.325716}
```

In this instance, each computational kernel receives a local definition for **a** that is dependent on its evaluation of **RandomReal**, and since **RandomReal** returns different values each time it is evaluated, this has the net effect of assigning a different value of **a** that is local to each computational kernel.

This approach can be advantageous when creating large datasets, because it allows the user to avoid passing large amounts of data between the master kernel and the computational kernels that might reside on different machines. The following example creates a list of five real numbers between 0 and 1 on each computational kernel and then returns the mean of each list to the master kernel.

```
ParallelEvaluate[a = RandomReal[{0, 1}, {5, 1}]; Mean[a]]
```

```
{{0.533235}, {0.59548}, {0.552252}, {0.417788}}
```

A common approach is to run calculations on the master kernel either prior to a parallel evaluation or after the results are returned from a parallel evaluation. The following example calculates the mean using the master kernel, with the input being the list of two means returned from the computational kernels.

```
Mean[
  ParallelEvaluate[a = RandomReal[{0, 1}, {5, 1}]; Mean[a]]
]
```

```
{0.599101}
```

Using Multiple CPU Cores for Simulations

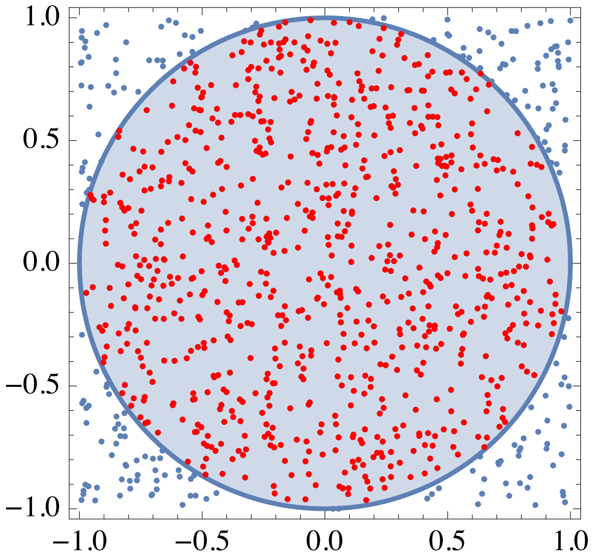
The following short program is a simple simulation that defines a sample size, creates a list and then returns the points in the list that satisfy a certain condition. In this case, the **Select** command is used to return the points that satisfy the equation $x^2 + y^2 < 1$, which can be used to calculate a ratio of the quantity of points inside the circle to the quantity of points that fall outside the circle.

```
sampleSize = 10^3;
myList = RandomReal[{−1, 1}, {sampleSize, 2}];
myPattern[pair_] := pair⟦1⟧^2 + pair⟦2⟧^2 < 1;
myResult = Select[myList, myPattern];
N[Length[myResult]/Length[myList]]

0.8
```

The following graphic shows all the points that were sampled, and the points that satisfied the inequality are colored red.

```
Show[
  RegionPlot[x^2 + y^2 < 1, {x, −1, 1}, {y, −1, 1}],
  ListPlot[myList, PlotStyle → {PointSize[Small]}],
  ListPlot[myResult, PlotStyle → {Red, PointSize[Small]}]
]
```



Problems that use random sampling to generate data like this lend themselves well to parallel calculations. By wrapping the **ParallelEvaluate** function around the series of calculations, the simulation is run with each computational kernel returning a result. In the following example, each result is the ratio of the points that satisfy the conditions to the points that do not satisfy the condition.

```
ParallelEvaluate[
  sampleSize = 10^3;
  myList = RandomReal[{-1, 1}, {sampleSize, 2}];
  myPattern[pair_] := pair〚1〛^2 + pair〚2〛^2 < 1;
  myResult = Select[myList, myPattern];
  N[Length[myResult]/Length[myList]]
]
```

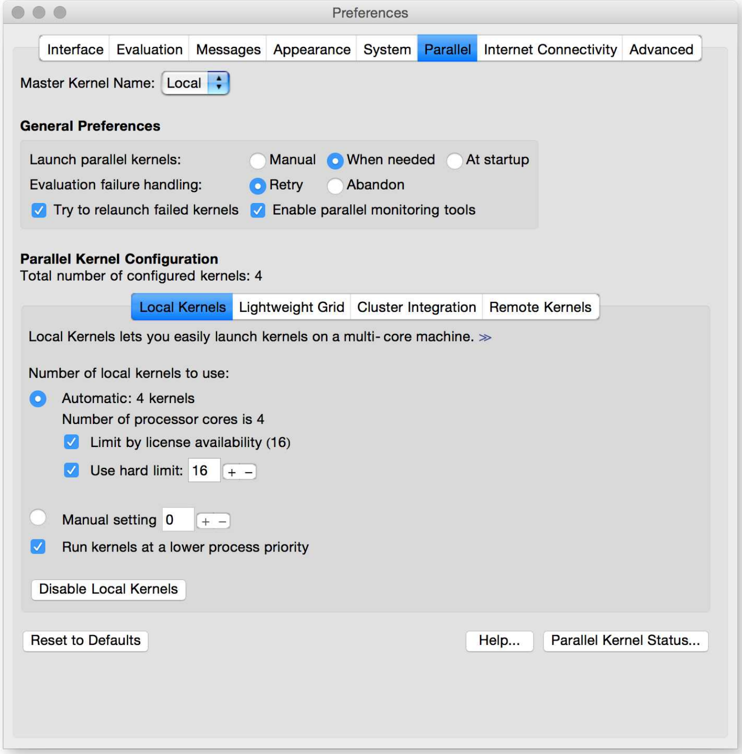
```
{0.785, 0.792, 0.786, 0.782}
```

After each computational kernel computes the ratio based on its unique random sampling, the **Mean** command can be wrapped around the statement above to calculate an average of the ratios. The evaluation of **Mean** in the following example is computed by the master kernel.

```
ParallelEvaluate[
    sampleSize = 10^6;
    myList = RandomReal[{-1, 1}, {sampleSize, 2}];
    myPattern[pair_] := pair〚1〛^2 + pair〚2〛^2 < 1;
    myResult = Select[myList, myPattern];
    N[Length[myResult]/Length[myList]]
  ] // Mean
```

```
0.785004
```

By opening the **Preferences** menu and navigating to the **Parallel** tab, users can adjust the default settings for working with parallel evaluations. For example, the number of local kernels can be set to a specific value if that is desirable for resource management. In the following screen shot, Mathematica is configured to automatically launch four computational kernels for parallel evaluation on the local machine.



Preferences

Master Kernel Name: Local

General Preferences

Launch parallel kernels: ○ Manual ● When needed ○ At startup

Evaluation failure handling: ● Retry ○ Abandon

☑ Try to relaunch failed kernels ☑ Enable parallel monitoring tools

Parallel Kernel Configuration
Total number of configured kernels: 4

Local Kernels lets you easily launch kernels on a multi- core machine. »

Number of local kernels to use:

● Automatic: 4 kernels
Number of processor cores is 4
☑ Limit by license availability (16)
☑ Use hard limit: 16 + −

○ Manual setting 0 + −

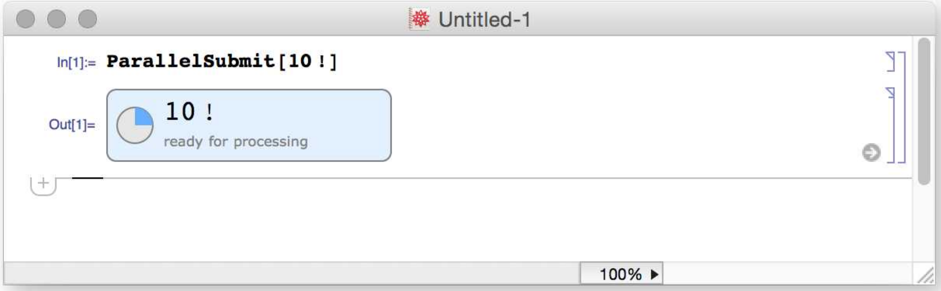
☑ Run kernels at a lower process priority

The **Remote Kernels** tab allows users to enable the launching of kernels on other computers using remote login. Hosts can be added by specifying their host names and the quantity of kernels to launch on that machine. Once configured, remote kernels can be added to the computational kernel pool for parallel evaluations.

The **Cluster Integration** tab can be used to interface with cluster management software, and the **Lightweight Grid** tab can be used to discover available Mathematica kernels on a network (assuming those instances of Mathematica are broadcasting their availability).

Submitting Jobs for Parallel Evaluation

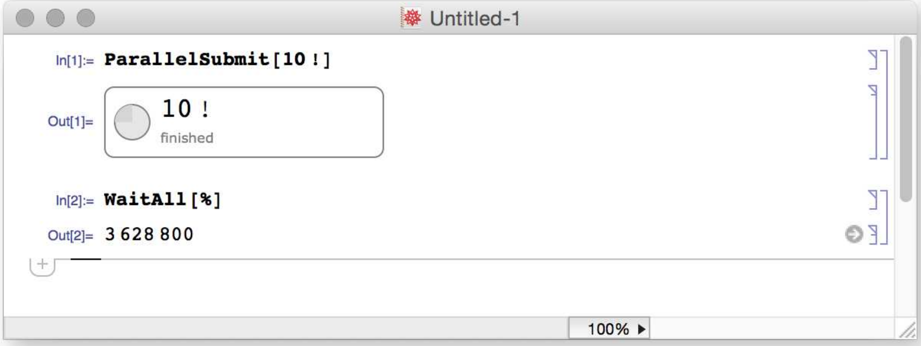
The preceding examples involved commands that were immediately evaluated on computational kernels upon execution. It is also possible to queue a calculation by using the **ParallelSubmit** command to submit a job in an unevaluated form to the computational kernels.



```
In[1]:= ParallelSubmit[10 !]

Out[1]= 10 !
        ready for processing
```

Once a job has been submitted with **ParallelSubmit**, the **WaitAll** command takes the **EvaluationObject** output from the preceding example and tells the computational kernels to perform the calculations, then returns their output to the master kernel. The **EvaluationObject** then updates its display to indicate the job has completed.



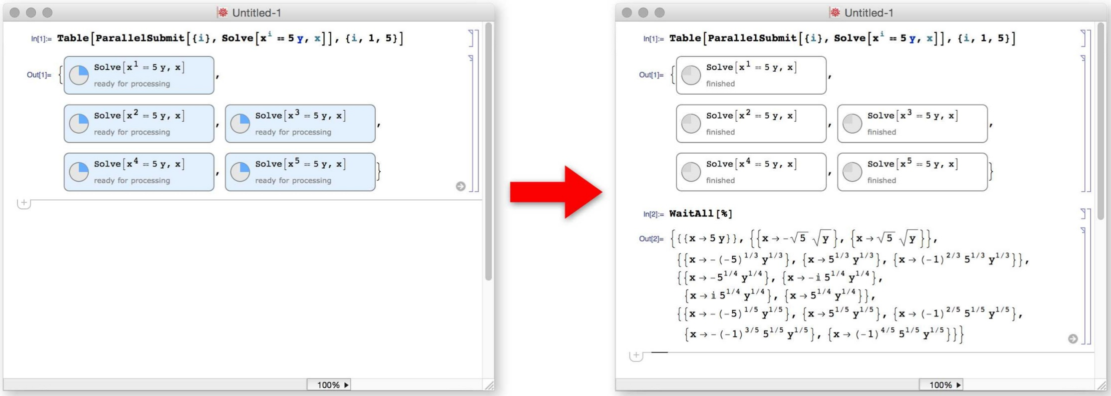
```
In[1]:= ParallelSubmit[10 !]

Out[1]= 10 !
        finished

In[2]:= WaitAll[%]

Out[2]= 3 628 800
```

This approach can also be used to submit multiple jobs simultaneously. The following screen shot shows a **Table** statement that creates a list of equations to solve, but the evaluations are held in a waiting state until **WaitAll** is evaluated.



```
In[1]:= Table[ParallelSubmit[{i}, Solve[x^i == 5 y, x]], {i, 1, 5}]

Out[1]= {Solve[x^1 == 5 y, x]   ,
         ready for processing

         Solve[x^2 == 5 y, x]   , Solve[x^3 == 5 y, x]   ,
         ready for processing     ready for processing

         Solve[x^4 == 5 y, x]   , Solve[x^5 == 5 y, x]   }
         ready for processing     ready for processing
```

```
In[1]:= Table[ParallelSubmit[{i}, Solve[x^i == 5 y, x]], {i, 1, 5}]

Out[1]= {Solve[x^1 == 5 y, x]   ,
         finished

         Solve[x^2 == 5 y, x]   , Solve[x^3 == 5 y, x]   ,
         finished                 finished

         Solve[x^4 == 5 y, x]   , Solve[x^5 == 5 y, x]   }
         finished                 finished

In[2]:= WaitAll[%]

Out[2]= {{{x → 5 y}}, {{x → -√5 √y}, {x → √5 √y}},
         {{x → -(-5)^(1/3) y^(1/3)}, {x → 5^(1/3) y^(1/3)}, {x → (-1)^(2/3) 5^(1/3) y^(1/3)}},
         {{x → -5^(1/4) y^(1/4)}, {x → -i 5^(1/4) y^(1/4)},
          {x → i 5^(1/4) y^(1/4)}, {x → 5^(1/4) y^(1/4)}},
         {{x → -(-5)^(1/5) y^(1/5)}, {x → 5^(1/5) y^(1/5)}, {x → (-1)^(2/5) 5^(1/5) y^(1/5)},
          {x → -(-1)^(3/5) 5^(1/5) y^(1/5)}, {x → (-1)^(4/5) 5^(1/5) y^(1/5)}}}
```

ParallelEvaluate is just one of the built-in commands for parallel evaluation. The **ParallelTable** command is a parallel implementation of **Table**, and the **ParallelMap** command is a parallel implementation of **Map**.

The following example creates a list of solutions by using the same call to the **Solve** command as the preceding example.

```
Table[Solve[x^i == 5 y, x], {i, 1, 5, 1}]
```

$$
\begin{aligned}
&\{\{\{x\to 5\,y\}\},\{\{x\to -\sqrt{5}\,\sqrt{y}\},\{x\to \sqrt{5}\,\sqrt{y}\}\},\\
&\quad\{\{x\to -(-5)^{1/3}\,y^{1/3}\},\{x\to 5^{1/3}\,y^{1/3}\},\{x\to (-1)^{2/3}\,5^{1/3}\,y^{1/3}\}\},\\
&\quad\{\{x\to -5^{1/4}\,y^{1/4}\},\{x\to -i\,5^{1/4}\,y^{1/4}\},\{x\to i\,5^{1/4}\,y^{1/4}\},\{x\to 5^{1/4}\,y^{1/4}\}\},\\
&\quad\{\{x\to -(-5)^{1/5}\,y^{1/5}\},\{x\to 5^{1/5}\,y^{1/5}\},\{x\to (-1)^{2/5}\,5^{1/5}\,y^{1/5}\},\\
&\qquad\{x\to -(-1)^{3/5}\,5^{1/5}\,y^{1/5}\},\{x\to (-1)^{4/5}\,5^{1/5}\,y^{1/5}\}\}\}
\end{aligned}
$$

ParallelTable has the exact same syntax as **Table**, but its execution splits up the list of calculations to be evaluated among the pool of available computational kernels. The result is identical to the previous example. Mathematica's approach of having parallel analogs to common commands makes it easy to create a parallel version of many routines by simply changing a few command names.

```
ParallelTable[Solve[x^i == 5 y, x], {i, 1, 5, 1}]
```

$$
\begin{aligned}
&\{\{\{x\to 5\,y\}\},\{\{x\to -\sqrt{5}\,\sqrt{y}\},\{x\to \sqrt{5}\,\sqrt{y}\}\},\\
&\quad\{\{x\to -(-5)^{1/3}\,y^{1/3}\},\{x\to 5^{1/3}\,y^{1/3}\},\{x\to (-1)^{2/3}\,5^{1/3}\,y^{1/3}\}\},\\
&\quad\{\{x\to -5^{1/4}\,y^{1/4}\},\{x\to -i\,5^{1/4}\,y^{1/4}\},\{x\to i\,5^{1/4}\,y^{1/4}\},\{x\to 5^{1/4}\,y^{1/4}\}\},\\
&\quad\{\{x\to -(-5)^{1/5}\,y^{1/5}\},\{x\to 5^{1/5}\,y^{1/5}\},\{x\to (-1)^{2/5}\,5^{1/5}\,y^{1/5}\},\\
&\qquad\{x\to -(-1)^{3/5}\,5^{1/5}\,y^{1/5}\},\{x\to (-1)^{4/5}\,5^{1/5}\,y^{1/5}\}\}\}
\end{aligned}
$$

Some examples in this chapter have compared results for calculations running serially and in parallel. Although showing that the results are identical is useful to demonstrate the functionality, the calculation time usually holds the greatest practical interest when exploring parallel calculations. A transition from using a single CPU core to multiple CPU cores can decrease calculation time quite a bit, and running the calculation on multiple machines may decrease calculation times even more.

The **AbsoluteTiming** command is useful for measuring the number of seconds that an evaluation takes to complete. This function is used in the following example to compare the serial and parallel evaluations of an expression. The semicolon after the **Table** and **ParallelTable** functions suppresses the output to highlight the evaluation time rather than the full result.

```
AbsoluteTiming[Table[Length[Solve[x^i == 5 y, x]], {i, 1, 500, 1}];]

 {2.991646, Null}

AbsoluteTiming[ParallelTable[Length[Solve[x^i == 5 y, x]], {i, 1, 500, 1}];]

 {0.789843, Null}
```

This particular example shows a measurable decrease in calculation time for the parallel version.

> Note that increasing the number of CPU cores might not result in an exact linear speedup, since some extra time can be spent for scheduling the calculation and putting the results together.

There is also a general command, **Parallelize**, that can be wrapped around an evaluation. If the evaluation can be done in parallel, it will.

```
Parallelize[
    Table[Length[Solve[x^i == 5 y, x]], {i, 1, 500, 1}]] // Short

 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ≪474≫, 488,
   489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500}
```

If this command is wrapped around a command that cannot be run in parallel, like an integral, a warning message prints that the calculation cannot be run in parallel, and Mathematica proceeds with a sequential evaluation.

$$\mathbf{Parallelize}\left[\int \frac{1}{\left(1+x^6\right)} \, dx\right]$$

••• Parallelize: $\int \frac{1}{1+x^6} \, dx$ cannot be parallelized; proceeding with sequential evaluation.

$$\frac{1}{12}\left(-2\,\mathrm{ArcTan}\left[\sqrt{3}-2\,x\right]+4\,\mathrm{ArcTan}[x]+2\,\mathrm{ArcTan}\left[\sqrt{3}+2\,x\right]-\sqrt{3}\,\mathrm{Log}\left[1-\sqrt{3}\,x+x^2\right]+\sqrt{3}\,\mathrm{Log}\left[1+\sqrt{3}\,x+x^2\right]\right)$$

Using Parallel Commands in Larger Programs

Other chapters have outlined how to create custom functions to help with common tasks. The following example is a short program with two inputs: a dataset and an integer that specifies a maximum polynomial degree. The main structure of the program consists of two **Table** statements. The first statement calculates a list of polynomial fits of varying degree for a specified dataset, and the second statement creates a plot of the raw data and the fitted curve, and displays r^2 values for each plot's corresponding polynomial fit.

```
dataFitApp[data_, polydegree_] :=
 (fits = Table[LinearModelFit[data, Table[x^j, {j, 0, i}], x],
      {i, 1, polydegree}];
   Table[
    Show[ListPlot[data, PlotStyle → PointSize[Medium]],
     Plot[fits⟦i⟧["BestFit"], {x, 0, 100}, PlotStyle → Red],
     PlotLabel → "degree " <> ToString[i] <> " polynomial" <> "\n" <> "r^2: " <>
       ToString[fits⟦i⟧["RSquared"]]],
    {i, polydegree}]
 )
```

Next a dataset is created for use with the program. The built-in **CountryData** function is used to return a list of GDP values for a certain country, based on a specific data range. This example collects GDP information for Sweden from 1970 to 2010.

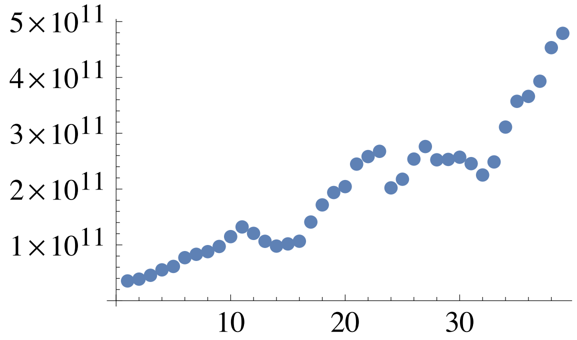
```
data1 =
  QuantityMagnitude[Normal[CountryData["Sweden", {{"GDP"}, {1970, 2010}}]]⟦All, 2⟧];
data1⟦1 ;; 10⟧
```

```
{3.54416×10^10, 3.8674×10^10, 4.55473×10^10, 5.52717×10^10, 6.14191×10^10,
  7.71176×10^10, 8.31435×10^10, 8.78938×10^10, 9.7175×10^10, 1.148×10^11}
```

The purpose of the **Normal** function might not be immediately obvious. Mathematica uses a special object for time series data that automatically summarizes the data. This is very useful for display purposes on the screen, but with variable assignment, **Normal** can be used to extract and store the actual data rather than the summary. **QuantityMagnitude** is used to discard the units and extract the values.

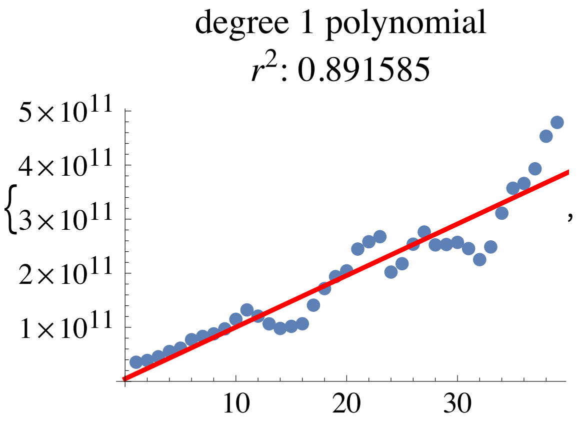
Plotting the GDP data with **ListPlot** shows that the data does not have a simple linear relationship over time but rather has periods of increasing and decreasing.

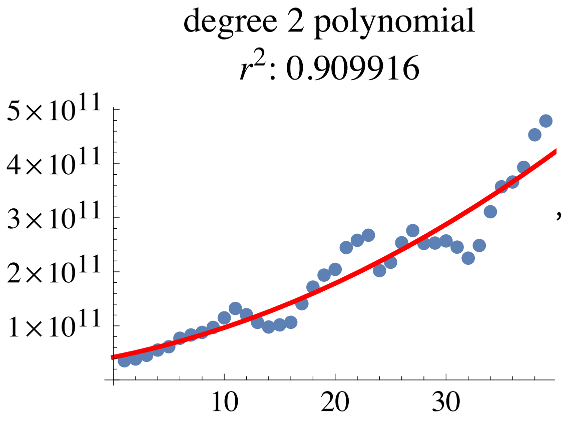
```
ListPlot[data1, PlotStyle → PointSize[Medium]]
```

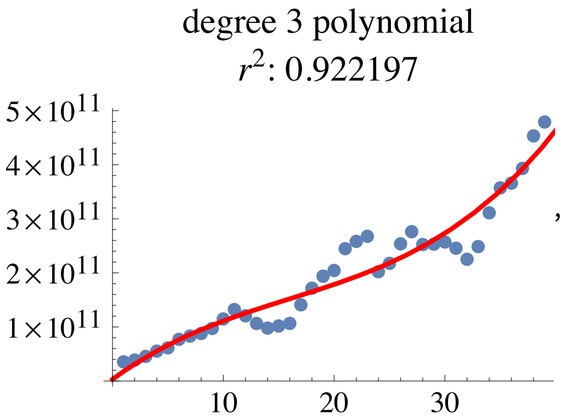


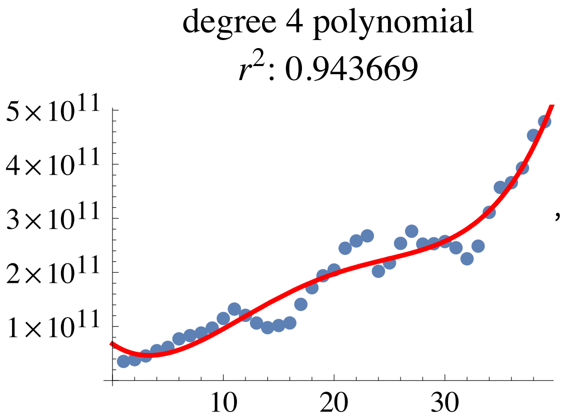
With the program and dataset defined, both can be conveniently used to explore various polynomial fits for this dataset. The program can now be used to try various polynomial fits to the GDP data.

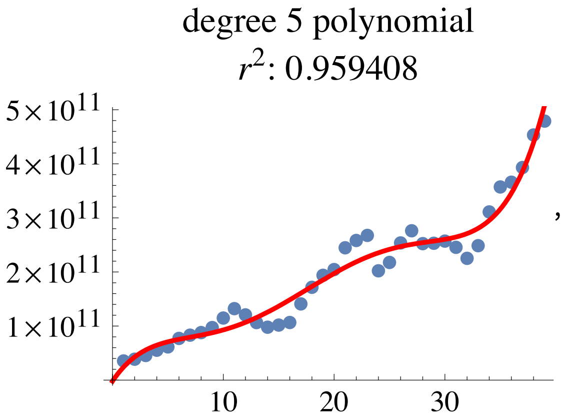
```
dataFitApp[data1, 8]
```

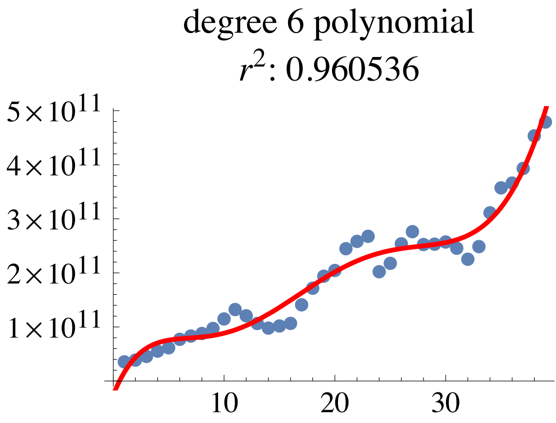




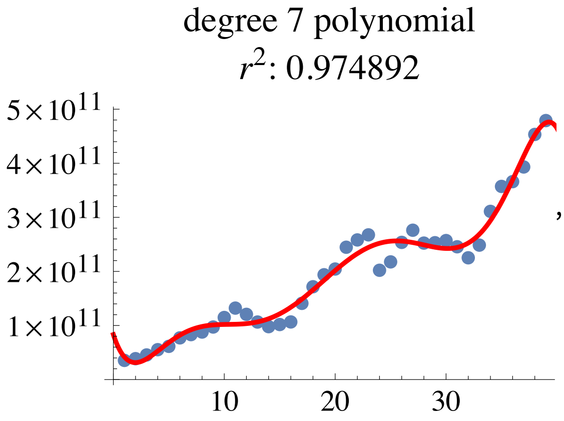

degree 3 polynomial
r^2: 0.922197
5×10^11
4×10^11
3×10^11
2×10^11
1×10^11
10
20
30
,


degree 4 polynomial
r^2: 0.943669
5×10^11
4×10^11
3×10^11
2×10^11
1×10^11
10
20
30
,

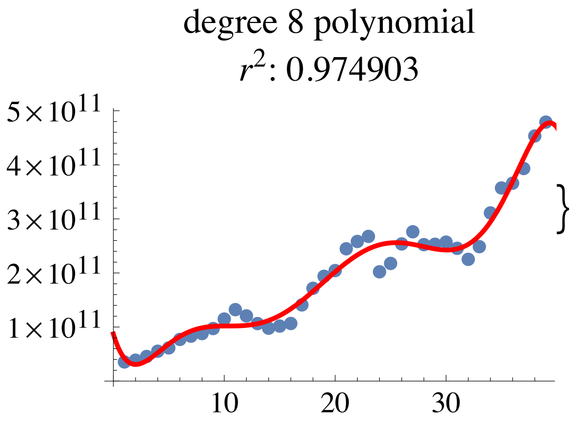

degree 5 polynomial
r^2: 0.959408
5×10^11
4×10^11
3×10^11
2×10^11
1×10^11
10
20
30
,



,



,



}

Since this program uses the **Table** function for iteration, the process to create a parallel version of this program involves changing just two function names. To more fully illustrate this point, the name of the program is changed as well. The structure of the program is now based on two **ParallelTable** statements rather than two **Table** statements, but all other parts of the program remain unchanged.

```
parallelDataFitApp[data_, polydegree_] :=
  (fits = ParallelTable[LinearModelFit[data, Table[x^j, {j, 0, i}], x],
        {i, 1, polydegree}];
    ParallelTable[
      Show[ListPlot[data, PlotStyle → PointSize[Medium]],
        Plot[fits〚i〛["BestFit"], {x, 0, 100}, PlotStyle → Red],
        PlotLabel → "degree " <> ToString[i] <> " polynomial" <> "\n" <> "r^2: " <>
            ToString[fits〚i〛["RSquared"]]],
      {i, polydegree}]
  )
```

AbsoluteTiming can be used to compare computation times between the original program and the parallel version. Using multiple computational kernels to work on the problem in parallel significantly decreases the calculation time.

```
dataFitApp[data1, 8]; // AbsoluteTiming
```

```
{4.182668, Null}
```

```
parallelDataFitApp[data1, 8]; // AbsoluteTiming
```

```
{1.745976, Null}
```

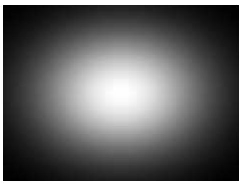
Using GPUs for Computation

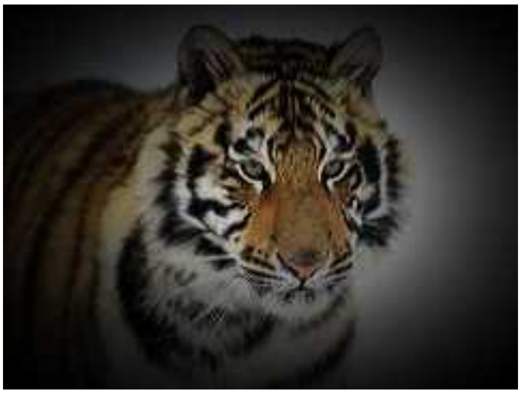
The examples in this chapter so far have focused on splitting up calculations between multiple CPU cores. Mathematica also contains built-in functionality to parallelize calculations across GPUs that support CUDA parallel computing architecture or the OpenCL programming language. To use either technology, first load the relevant linking package using the **Needs** command. The following command shows an example of how to load the package to work with CUDA cards.

```
Needs["CUDALink`"]
```

> Note that when loading a package, the grave accent or back tick character ` is used at the end of the name. This is different from an apostrophe or single quote.

Device-specific commands can be used after the package is loaded. Many of these commands are related to image processing, like the following example that takes two images and uses the GPU to multiply them together. (The documentation for **CUDAImageMultiply** contains these same images, which can be used to directly experiment with this command.)

CUDAImageMultiply[, ]



The quickest way to determine if your machine supports CUDA is to load the CUDALink package using **Needs["CUDALink`"]** and evaluate **CUDAQ[]**. If a supported device is found, it will return **True**, and you can get more information about the device by evaluating **CUDAInformation[]**. A similar set of commands can be performed to determine the presence of **OpenCL** cards on your machine.

However, GPUs can also be used for other computations, like certain linear algebra operations. Like the serial and parallel versions of certain commands, there are also serial and GPU versions of some commands. The syntax for these commands is the same, but using the appropriate function name with the right hardware will leverage that hardware for faster computation. An example is the **CUDADot** command, which computes dot products using CUDA-enabled cards.

$$\textbf{CUDADot}\left[\begin{pmatrix}1&0&0\\0&4&0\\0&0&5\end{pmatrix},\begin{pmatrix}2&0&0\\0&6&0\\0&0&8\end{pmatrix}\right]$$

```
{{2, 0, 0}, {0, 24, 0}, {0, 0, 40}}
```

Another example is **CUDATranspose**, which has the same format as the **Transpose** command but uses a CUDA-enabled GPU to transpose a matrix.

```
               ⎡⎛ 0  1.1315  0 ⎞⎤
CUDATranspose⎢⎜ 0  4.7572  0 ⎟⎥
               ⎣⎝ 0  5.1222  0 ⎠⎦
```

```
{{0., 0., 0.}, {1.1315, 4.7572, 5.1222}, {0., 0., 0.}}
```

More GPU commands can be found in the documentation guide page for GPU computing.

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[myList, dataFitApp, data1, parallelDataFitApp]
```

Conclusion

Not all calculations in Mathematica require a parallel approach, but faster computation time can be the difference between finding an answer in minutes rather than hours. When using Mathematica for serial or parallel programs, its expansive list of commands means a wide variety of problems can be addressed fairly quickly, and sometimes converting a serial program to a parallel one is as simple as swapping in a few versions of parallel commands.

Exercises

1. Use **ParallelEvaluate** to launch a computation that chooses a random integer between 1 and 25. The computation will automatically launch a subkernel for each processor core on your machine, so the number of random numbers that are returned will be dependent on your machine.
2. Define a variable **var25** that contains a list of five random integers between 1 and 25, and then use the appropriate command to compute **var25**^{2} in parallel.
3. Using the answer from Exercise 2 as a starting point, modify the code so that each subkernel creates its own list of five random integers between 1 and 25, and then compute **var25**^{2} in parallel.
4. Use **FindRoot** to locate where $\sin(x) + e^{x}$ passes the x axis near a particular starting point i on the x axis. Pass this **FindRoot** command to **ParallelSubmit**, and wrap everything in a **Table** command to vary i from -3 to 3.
5. Use the appropriate command to evaluate all the calculations from Exercise 4 that are in a waiting state.

6. Using the answer from Exercise 4 as a starting point, rewrite the code to use **ParallelTable** instead of **Table**, such that the parallel evaluation is calculated immediately instead of being placed into a waiting state.

7. Write a statement with **ParallelTable** to create a series of 3D plots of sin(x) cos(a y), where a varies from 1 to 3 in steps of 1.

8. Use **ParallelTable** to create a series of list plots, where each plot contains two datasets. The first dataset is the list of integers from 1 to 100, and the second dataset is the list of integers from 1 to 100 multiplied by i, where i goes from 2 to 5 in steps of 1.

9. Create a function **fun25** that accepts a single argument **data25** and then constructs a series of list plots, where each plot contains two datasets. The first dataset is simply **data25**, and the second list plot is **data25** multiplied by i, where i goes from 2 to 5 in steps of 1.

10. Pass a list of the form $\{j,\ j^2\}$, where j goes from 1 to 100, as the argument to the **fun25** function.

Index