I Wrote a Book—To Teach the Wolfram Language

December 8, 2015

An Elementary Introduction to the Wolfram Language is available in print, free on the web, etc.

I wasn't sure if I was ever going to write another book. My last book—*A New Kind of Science*—took me more than a decade of intensely focused work, and is the largest personal project I've ever done.

But a little while ago, I realized there was another book I had to write: a book that would introduce people with no knowledge of programming to the Wolfram Language and the kind of computational thinking it allows.

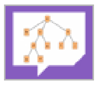
The result is *An Elementary Introduction to the Wolfram Language*, published today in print, free on the web, etc.

Search Blog

RECENT POSTS

Roaring into 2018 with Another Big Release: Launching Version 11.3 of the Wolfram Language & Mathematica
March 8, 2018

Showing Off to the Universe: Beacons for the Afterlife of Our Civilization
January 25, 2018


What Do I Do All Day? Livestreamed Technology CEOing
December 11, 2017

What Is a Computational Essay?
November 14, 2017

Are All Fish the Same Shape if You Stretch Them? The Victorian Tale of *On Growth and Form*
October 25, 2017

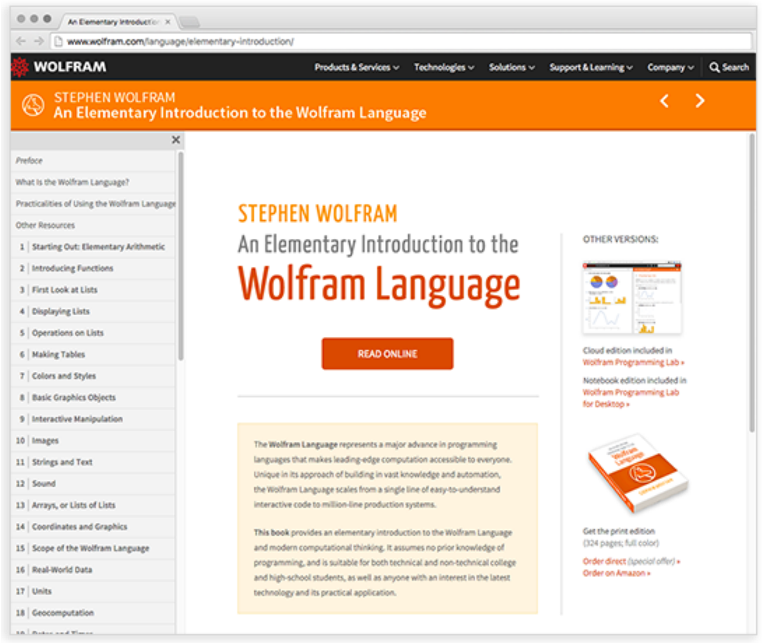
More ↓

POPULAR CATEGORIES

Artificial Intelligence
Big Picture
Companies and Business
Computational Science
Computational Thinking
Data Science
Education
Future Perspectives
Historical Perspectives
Life and Times
Life Science
Mathematica
Mathematics
New Kind of Science
New Technology
Personal Analytics
Physical Science
Software Design
Wolfram|Alpha
Wolfram|One
Wolfram Language
Other

ARCHIVE

2018 | 2017 | 2016 | 2015 | 2014 | 2013 | 2012
2011 | 2010 | 2009 | 2008 | 2007 | 2006 | 2004
2003



An Elementary Introductio ×
www.wolfram.com/language/elementary-introduction/
WOLFRAM
Products & Services
Technologies
Solutions
Support & Learning
Company
Search
STEPHEN WOLFRAM
An Elementary Introduction to the Wolfram Language
READ ONLINE

Preface
What Is the Wolfram Language?
Practicalities of Using the Wolfram Language
Other Resources
1 Starting Out: Elementary Arithmetic
2 Introducing Functions
3 First Look at Lists
4 Displaying Lists
5 Operations on Lists
6 Making Tables
7 Colors and Styles
8 Basic Graphics Objects
9 Interactive Manipulation
10 Images
11 Strings and Text
12 Sound
13 Arrays, or Lists of Lists
14 Coordinates and Graphics
15 Scope of the Wolfram Language
16 Real-World Data
17 Units
18 Geocomputation

STEPHEN WOLFRAM
An Elementary Introduction to the
Wolfram Language

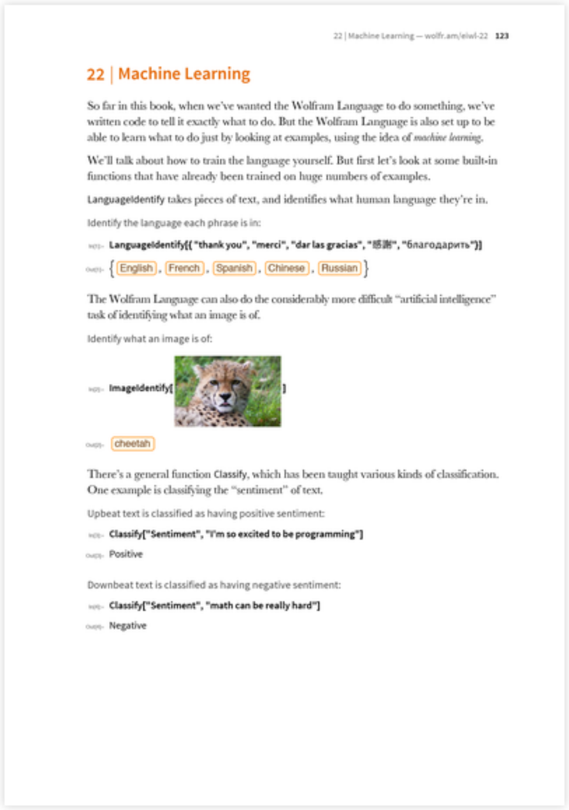
The **Wolfram Language** represents a major advance in programming languages that makes leading-edge computation accessible to everyone. Unique in its approach of building in vast knowledge and automation, the Wolfram Language scales from a single line of easy-to-understand interactive code to million-line production systems.

This book provides an elementary introduction to the Wolfram Language and modern computational thinking. It assumes no prior knowledge of programming, and is suitable for both technical and non-technical college and high-school students, as well as anyone with an interest in the latest technology and its practical application.

OTHER VERSIONS:
Cloud edition included in Wolfram Programming Lab »
Notebook edition included in Wolfram Programming Lab for Desktop »
Get the print edition
(324 pages; full color)
Order direct (special offer) »
Order on Amazon »

The goal of the book is to take people from zero to the point where they know enough about the Wolfram Language that they can routinely use it to create programs for things they want to do. And when I say "zero", I really mean "zero". This is a book for everyone. It doesn't assume any knowledge of programming, or math (beyond basic arithmetic), or anything else. It just starts from scratch and explains things. I've tried to make it appropriate for both adults and kids. I think it'll work for typical kids aged about 12 and up.

In the past, a book like this would have been inconceivable. The necessary underlying technology just didn't exist. Serious programming was always difficult, and there wasn't a good way to connect with real-world concepts. But now we have the Wolfram Language. It's taken three decades. But now we've built in enough knowledge and automated enough of the process of programming that it's actually realistic to take almost anyone from zero to the frontiers of what can be achieved with computation.



22 | Machine Learning — wolfr.am/eiwl-22 123

22 | Machine Learning

So far in this book, when we've wanted the Wolfram Language to do something, we've written code to tell it exactly what to do. But the Wolfram Language is also set up to be able to learn what to do just by looking at examples, using the idea of *machine learning*.

We'll talk about how to train the language yourself. But first let's look at some built-in functions that have already been trained on huge numbers of examples.

LanguageIdentify takes pieces of text, and identifies what human language they're in.

Identify the language each phrase is in:

```
In[1]:= LanguageIdentify[{"thank you", "merci", "dar las gracias", "感谢", "благодарить"}]
Out[1]= {English, French, Spanish, Chinese, Russian}
```

The Wolfram Language can also do the considerably more difficult "artificial intelligence" task of identifying what an image is of.

Identify what an image is of:

```
In[2]:= ImageIdentify[ ]
Out[2]= cheetah
```

There's a general function Classify, which has been taught various kinds of classification. One example is classifying the "sentiment" of text.

Upbeat text is classified as having positive sentiment:

```
In[3]:= Classify["Sentiment", "I'm so excited to be programming"]
Out[3]= Positive
```

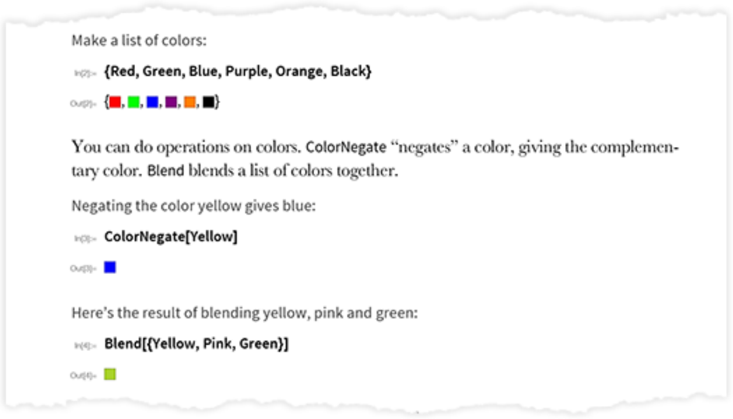
Downbeat text is classified as having negative sentiment:

```
In[4]:= Classify["Sentiment", "math can be really hard"]
Out[4]= Negative
```

But how should one actually do it? What should one explain, in what order? Those were challenges I had to address to write this book. I'd written a Fast Introduction for Programmers that in 30 pages or so introduces people who already know about modern programming to the core concepts of the Wolfram Language. But what about people who don't start off knowing anything about programming?

For many years I've found various opportunities to show what's now the Wolfram Language to people like that. And now I've used my experience to figure out what to do in the book.

It's a Conversation In essence, the book brings the reader into a conversation with the computer. There are two great things about the Wolfram Language that make this really work. First, that the language is symbolic, so that anything one's dealing with—a color, an image, a graph, whatever—can be right there in the dialog. And second, that the language can be purely functional, so that everything is stateless, and every input can be self contained.



Make a list of colors:

```
In[2]:= {Red, Green, Blue, Purple, Orange, Black}
Out[2]= {🟥, 🟩, 🟦, 🟪, 🟧, ⬛}
```

You can do operations on colors. ColorNegate "negates" a color, giving the complementary color. Blend blends a list of colors together.

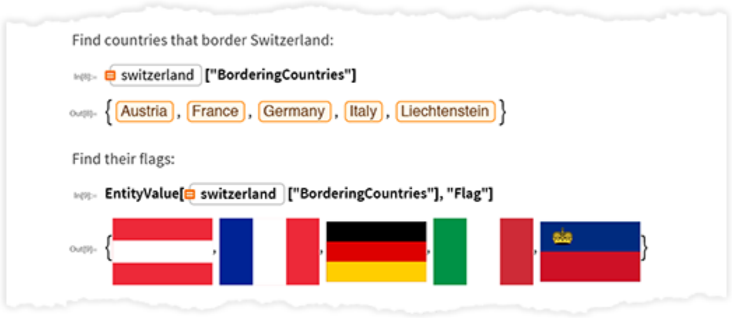
Negating the color yellow gives blue:

```
In[3]:= ColorNegate[Yellow]
Out[3]= 🟦
```

Here's the result of blending yellow, pink and green:

```
In[4]:= Blend[{Yellow, Pink, Green}]
Out[4]= 🟩
```

It's also very important that the Wolfram Language has built-in knowledge that lets one immediately compute with real-world things.



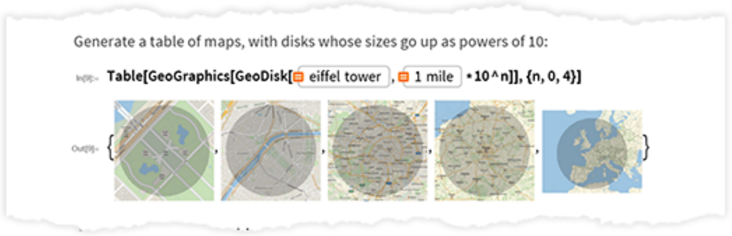
Find countries that border Switzerland:

```
switzerland ["BorderingCountries"]
```

```
{Austria, France, Germany, Italy, Liechtenstein}
```

Find their flags:

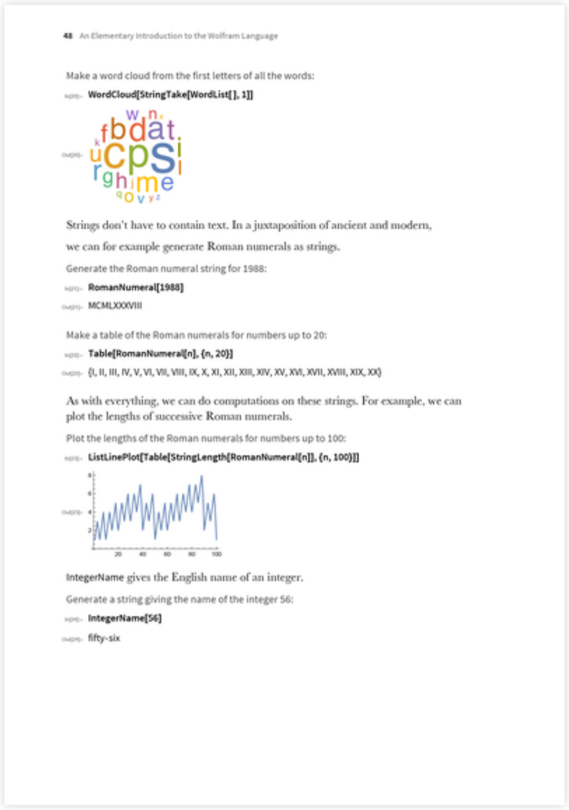
```
EntityValue[switzerland ["BorderingCountries"], "Flag"]
```



Generate a table of maps, with disks whose sizes go up as powers of 10:

```
Table[GeoGraphics[GeoDisk[eiffel tower, 1 mile *10^n]], {n, 0, 4}]
```

Oh, and visualization is really important too—so it's easy to see what one's computing.



48 An Elementary Introduction to the Wolfram Language

Make a word cloud from the first letters of all the words:

```
WordCloud[StringTake[WordList[], 1]]
```

Strings don't have to contain text. In a juxtaposition of ancient and modern, we can for example generate Roman numerals as strings.

Generate the Roman numeral string for 1988:

```
RomanNumeral[1988]
```

```
MCMLXXXVIII
```

Make a table of the Roman numerals for numbers up to 20:

```
Table[RomanNumeral[n], {n, 20}]
```

```
{I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII, XIII, XIV, XV, XVI, XVII, XVIII, XIX, XX}
```

As with everything, we can do computations on these strings. For example, we can plot the lengths of successive Roman numerals.

Plot the lengths of the Roman numerals for numbers up to 100:

```
ListLinePlot[Table[StringLength[RomanNumeral[n]], {n, 100}]]
```

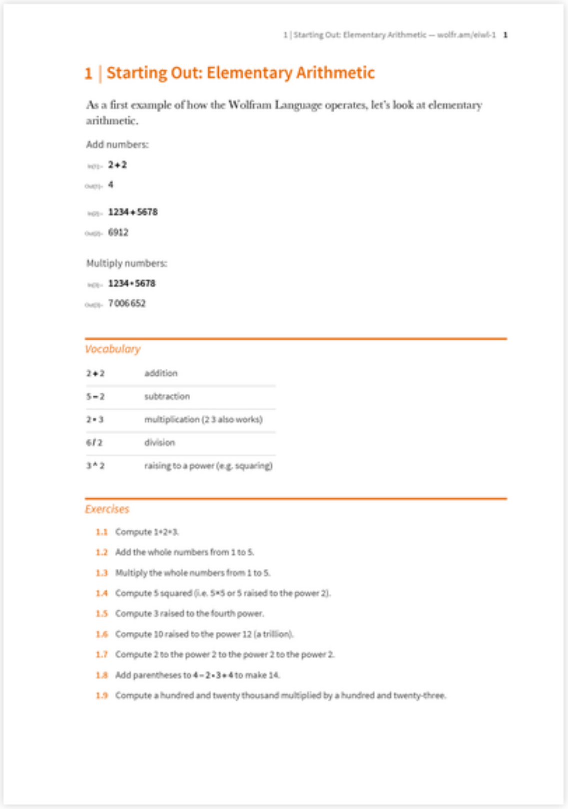
IntegerName gives the English name of an integer.

Generate a string giving the name of the integer 56:

```
IntegerName[56]
```

```
fifty-six
```

Where to Start? OK, but where should one start? The very first page is about arithmetic—just because that's a place where everyone can see that a computation is actually happening:



1 | Starting Out: Elementary Arithmetic — wolfr.am/eiwl-1 1

1 | Starting Out: Elementary Arithmetic

As a first example of how the Wolfram Language operates, let's look at elementary arithmetic.

Add numbers:

```
In[1]:= 2+2
Out[1]= 4

In[2]:= 1234+5678
Out[2]= 6912
```

Multiply numbers:

```
In[3]:= 1234*5678
Out[3]= 7006652
```

Vocabulary

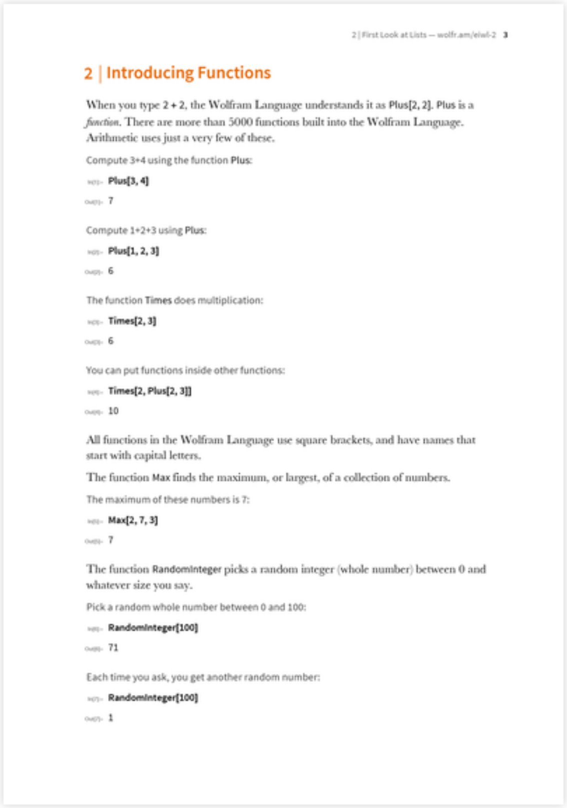
2+2	addition
5-2	subtraction
2*3	multiplication (2 3 also works)
6/2	division
3^2	raising to a power (e.g. squaring)

Exercises

1.1 Compute 1+2+3.
1.2 Add the whole numbers from 1 to 5.
1.3 Multiply the whole numbers from 1 to 5.
1.4 Compute 5 squared (i.e. 5×5 or 5 raised to the power 2).
1.5 Compute 3 raised to the fourth power.
1.6 Compute 10 raised to the power 12 (a trillion).
1.7 Compute 2 to the power 2 to the power 2 to the power 2.
1.8 Add parentheses to 4-2*3+4 to make 14.
1.9 Compute a hundred and twenty thousand multiplied by a hundred and twenty-three.

There's a section called Vocabulary because that's what it is: one's learning some "words" in the Wolfram Language. Then there are exercises, which I'll talk about soon.

OK, but once one's done arithmetic, where should one go next? What I decided to do was to go immediately to the idea of functions—and to first introduce them in terms of arithmetic. The advantage of this is that while the concept of a function may be new, the operation it's doing (namely arithmetic) is familiar.



2 | First Look at Lists — wolfr.am/eiwl-2 3

2 | Introducing Functions

When you type **2 + 2**, the Wolfram Language understands it as **Plus[2, 2]**. **Plus** is a *function*. There are more than 5000 functions built into the Wolfram Language. Arithmetic uses just a very few of these.

Compute 3+4 using the function **Plus**:

```
In[1]:= Plus[3, 4]
Out[1]= 7
```

Compute 1+2+3 using **Plus**:

```
In[2]:= Plus[1, 2, 3]
Out[2]= 6
```

The function **Times** does multiplication:

```
In[3]:= Times[2, 3]
Out[3]= 6
```

You can put functions inside other functions:

```
In[4]:= Times[2, Plus[2, 3]]
Out[4]= 10
```

All functions in the Wolfram Language use square brackets, and have names that start with capital letters.

The function **Max** finds the maximum, or largest, of a collection of numbers.

The maximum of these numbers is 7:

```
In[5]:= Max[2, 7, 3]
Out[5]= 7
```

The function **RandomInteger** picks a random integer (whole number) between 0 and whatever size you say.

Pick a random whole number between 0 and 100:

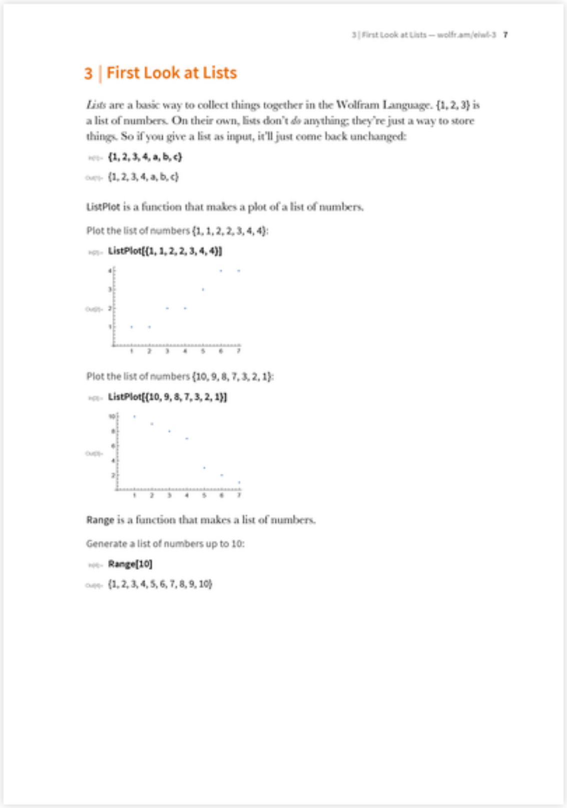
```
In[6]:= RandomInteger[100]
Out[6]= 71
```

Each time you ask, you get another random number:

```
In[7]:= RandomInteger[100]
Out[7]= 1
```

And once one’s understood the function Plus, one can immediately go to functions like Max that don’t have special input forms. What Max does isn’t that exciting, though. So as a slightly more exciting function, what I introduce next is RandomInteger—which people often like to run over and over again, to see what it produces.

OK, so what next? The obvious answer is that we have to introduce lists. But what should one do with lists? Doing something like picking elements out of them isn’t terribly exciting, and it’s hard immediately to see why it’s important. So instead what I decided was to make the very first function I show for lists be ListPlot. It’s nice to start getting in the idea of visualization—and it’s also a good example of how one can type in a tiny piece of code, and get something bigger and more interesting out.



3 | First Look at Lists — wolfr.am/eiwl-3 7

3 | First Look at Lists

Lists are a basic way to collect things together in the Wolfram Language. {1, 2, 3} is a list of numbers. On their own, lists don't *do* anything; they're just a way to store things. So if you give a list as input, it'll just come back unchanged:

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

ListPlot is a function that makes a plot of a list of numbers.

Plot the list of numbers {1, 1, 2, 2, 3, 4, 4}:

```
ListPlot[{1, 1, 2, 2, 3, 4, 4}]
```

Plot the list of numbers {10, 9, 8, 7, 3, 2, 1}:

```
ListPlot[{10, 9, 8, 7, 3, 2, 1}]
```

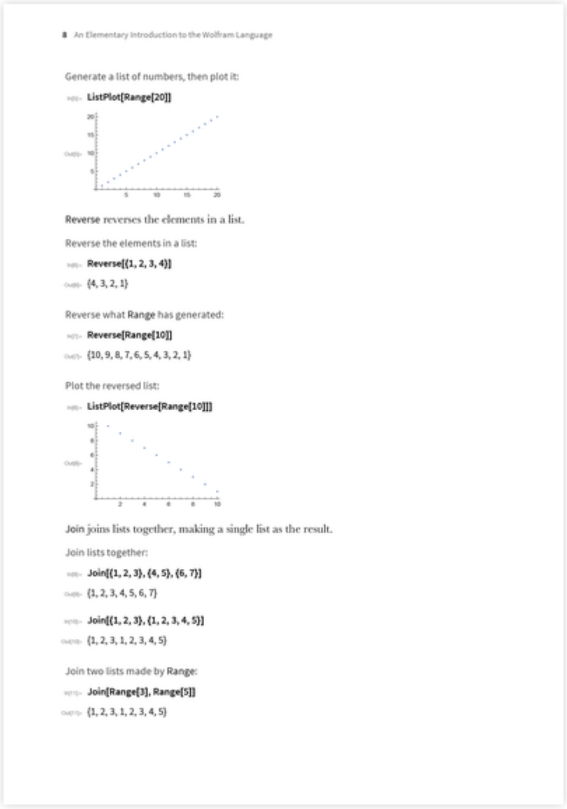
Range is a function that makes a list of numbers.

Generate a list of numbers up to 10:

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

Actually, the best extremely simple example of that is Range, which I also show at this point. Range is a great way to show the computer actually computing something, with a result that's easy to understand.

But OK, so now we want to reinforce the idea of functions, and functions working together. The function Reverse isn't incredibly common in practice, but it's very easy to understand, so I introduce it next, followed by Join.



8 An Elementary Introduction to the Wolfram Language

Generate a list of numbers, then plot it:

```
In[5]:= ListPlot[Range[20]]
```

Reverse reverses the elements in a list.

Reverse the elements in a list:

```
In[6]:= Reverse[{1, 2, 3, 4}]
Out[6]= {4, 3, 2, 1}
```

Reverse what Range has generated:

```
In[7]:= Reverse[Range[10]]
Out[7]= {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

Plot the reversed list:

```
In[8]:= ListPlot[Reverse[Range[10]]]
```

Join joins lists together, making a single list as the result.

Join lists together:

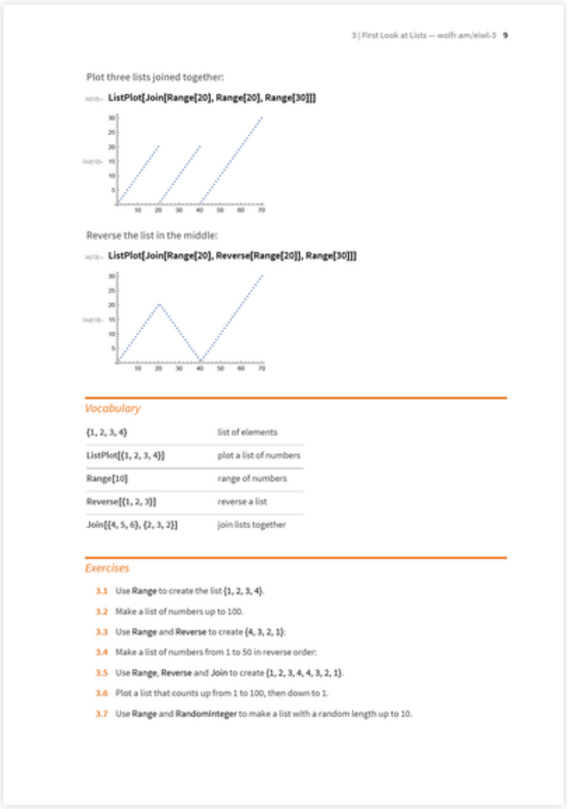
```
In[9]:= Join[{1, 2, 3}, {4, 5}, {6, 7}]
Out[9]= {1, 2, 3, 4, 5, 6, 7}
```

```
In[10]:= Join[{1, 2, 3}, {1, 2, 3, 4, 5}]
Out[10]= {1, 2, 3, 1, 2, 3, 4, 5}
```

Join two lists made by Range:

```
In[11]:= Join[Range[3], Range[5]]
Out[11]= {1, 2, 3, 1, 2, 3, 4, 5}
```

What's nice then is that between Reverse, Range and Join we have a little microlanguage that's completely self-contained, but lets us do a variety of computations. And, of course, whatever computations one does, one can immediately see the results, either symbolically or visually.



3 | First Look at Lists — wolfr.am/eiwl-3 9

Plot three lists joined together:

```
In[12]:= ListPlot[Join[Range[20], Range[20], Range[30]]]
```

Reverse the list in the middle:

```
In[13]:= ListPlot[Join[Range[20], Reverse[Range[20]], Range[30]]]
```

Vocabulary

{1, 2, 3, 4}	list of elements
ListPlot[{1, 2, 3, 4}]	plot a list of numbers
Range[10]	range of numbers
Reverse[{1, 2, 3}]	reverse a list
Join[{4, 5, 6}, {2, 3, 2}]	join lists together

Exercises

3.1 Use **Range** to create the list {1, 2, 3, 4}.

3.2 Make a list of numbers up to 100.

3.3 Use **Range** and **Reverse** to create {4, 3, 2, 1}.

3.4 Make a list of numbers from 1 to 50 in reverse order.

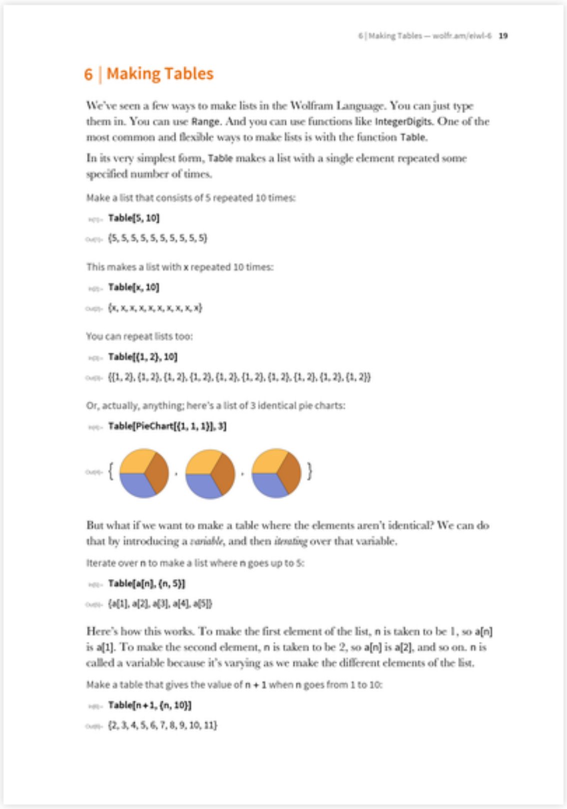
3.5 Use **Range**, **Reverse** and **Join** to create {1, 2, 3, 4, 4, 3, 2, 1}.

3.6 Plot a list that counts up from 1 to 100, then down to 1.

3.7 Use **Range** and **RandomInteger** to make a list with a random length up to 10.

The next couple of sections talk about displaying and operating on lists, reinforcing what's already been said, and introducing a variety of functions that are useful in practice. Then it's on to Table—a very common and powerful function, that in effect packages up a lot of what might otherwise need explicit loops and so on.

I start with trivial versions of Table, without any iteration variable. I take it for granted (as people who don't know "better" do!) that Table can produce a list of graphics just like it can produce a list of numbers. (Of course, the fact that it can do this is a consequence of the fundamentally symbolic character of the Wolfram Language.)



6 | Making Tables

We've seen a few ways to make lists in the Wolfram Language. You can just type them in. You can use **Range**. And you can use functions like **IntegerDigits**. One of the most common and flexible ways to make lists is with the function **Table**.

In its very simplest form, **Table** makes a list with a single element repeated some specified number of times.

Make a list that consists of 5 repeated 10 times:

```
Table[5, 10]
{5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
```

This makes a list with x repeated 10 times:

```
Table[x, 10]
{x, x, x, x, x, x, x, x, x, x}
```

You can repeat lists too:

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

Or, actually, anything; here's a list of 3 identical pie charts:

```
Table[PieChart[{1, 1, 1}], 3]
```

But what if we want to make a table where the elements aren't identical? We can do that by introducing a *variable*, and then *iterating* over that variable.

Iterate over n to make a list where n goes up to 5:

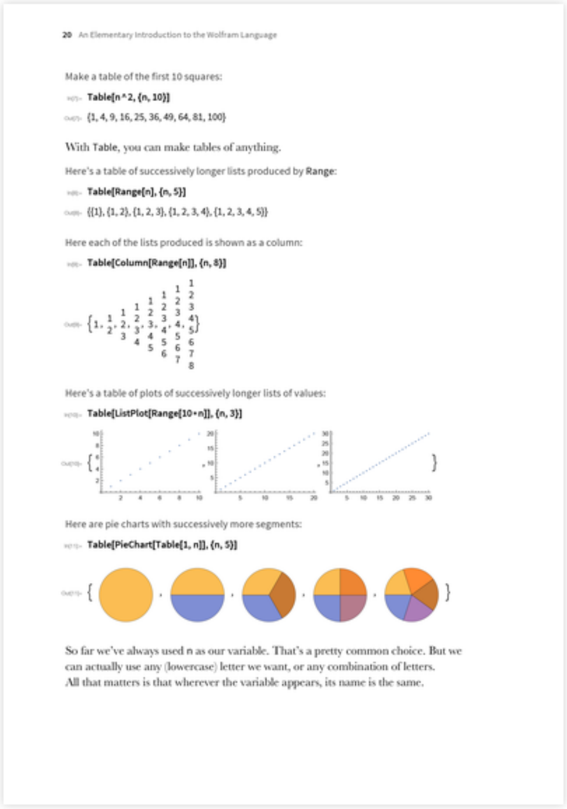
```
Table[a[n], {n, 5}]
{a[1], a[2], a[3], a[4], a[5]}
```

Here's how this works. To make the first element of the list, n is taken to be 1, so a[n] is a[1]. To make the second element, n is taken to be 2, so a[n] is a[2], and so on. n is called a variable because it's varying as we make the different elements of the list.

Make a table that gives the value of n + 1 when n goes from 1 to 10:

```
Table[n + 1, {n, 10}]
{2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

The next big step is to introduce a variable into Table. I thought a lot about how to do this, and decided that the best thing to show first is the purely symbolic version. After all, we've already introduced functions, and with the symbolic version, one can immediately see where the variable goes. But now that we've got Table with variables, we can really go to town and start doing what people will think of as "real computations".



Make a table of the first 10 squares:

```
Table[n^2, {n, 10}]
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

With Table, you can make tables of anything.

Here's a table of successively longer lists produced by **Range**:

```
Table[Range[n], {n, 5}]
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}}
```

Here each of the lists produced is shown as a column:

```
Table[Column[Range[n]], {n, 8}]
                    1
                  1 2
                1 2 3
          1   1 2 3 4
      1   2 1 2 3 4 5
{1, 1, 2, 3, 3, 4, 5}
    2 3   4 4 5 6 6
          4 5 5 6 7
              6 7 8
                  8
```

Here's a table of plots of successively longer lists of values:

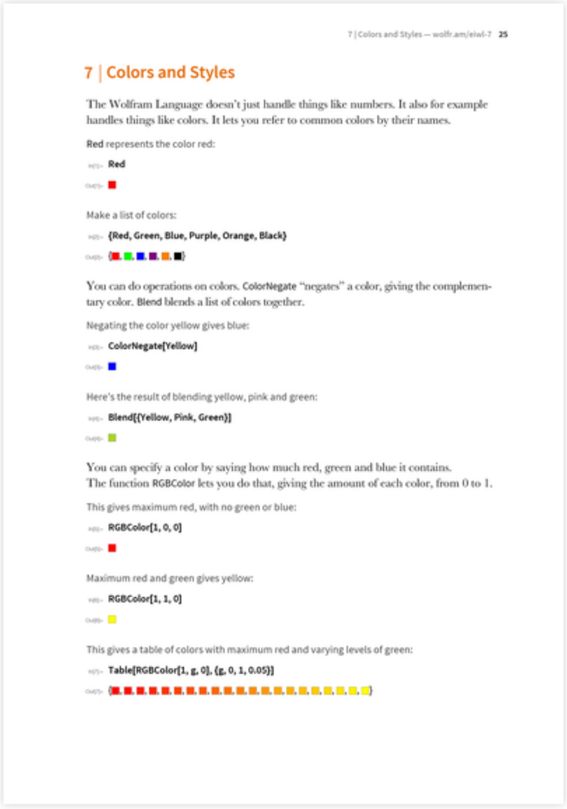
```
Table[ListPlot[Range[10 n]], {n, 3}]
```

Here are pie charts with successively more segments:

```
Table[PieChart[Table[1, n]], {n, 5}]
```

So far we've always used n as our variable. That's a pretty common choice. But we can actually use any (lowercase) letter we want, or any combination of letters. All that matters is that wherever the variable appears, its name is the same.

The Arc of the Book In the first few sections of the book, the raw material for our computations is basically numbers and lists. What I wanted to do next was to show that there are other things to compute with. I chose colors as the first example. Colors are good because (a) everyone knows what they are, (b) you can actually compute with them and (c) they make colorful output (!).



7 | Colors and Styles

The Wolfram Language doesn't just handle things like numbers. It also for example handles things like colors. It lets you refer to common colors by their names.

Red represents the color red:

```
Red
🟥
```

Make a list of colors:

```
{Red, Green, Blue, Purple, Orange, Black}
{🟥, 🟩, 🟦, 🟪, 🟧, ⬛}
```

You can do operations on colors. ColorNegate "negates" a color, giving the complementary color. Blend blends a list of colors together.

Negating the color yellow gives blue:

```
ColorNegate[Yellow]
🟦
```

Here's the result of blending yellow, pink and green:

```
Blend[{Yellow, Pink, Green}]
🟩
```

You can specify a color by saying how much red, green and blue it contains. The function RGBColor lets you do that, giving the amount of each color, from 0 to 1.

This gives maximum red, with no green or blue:

```
RGBColor[1, 0, 0]
🟥
```

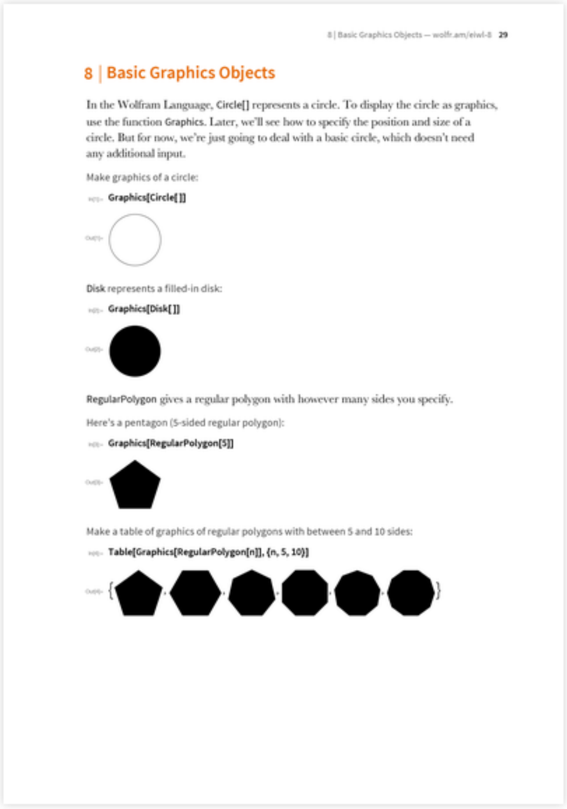
Maximum red and green gives yellow:

```
RGBColor[1, 1, 0]
🟨
```

This gives a table of colors with maximum red and varying levels of green:

```
Table[RGBColor[1, g, 0], {g, 0, 1, 0.05}]
{🟥, 🟥, 🟥, 🟥, 🟥, 🟥, 🟧, 🟧, 🟧, 🟧, 🟧, 🟧, 🟧, 🟧, 🟧, 🟨, 🟨, 🟨, 🟨, 🟨, 🟨}
```

After colors we're ready for some graphics. I haven't talked about coordinates yet, so I can only show individual graphical objects, without placement information.



8 | Basic Graphics Objects — wolfr.am/eiwl-8 29

8 | Basic Graphics Objects

In the Wolfram Language, Circle[] represents a circle. To display the circle as graphics, use the function Graphics. Later, we'll see how to specify the position and size of a circle. But for now, we're just going to deal with a basic circle, which doesn't need any additional input.

Make graphics of a circle:

```
Graphics[Circle[]]
```

Disk represents a filled-in disk:

```
Graphics[Disk[]]
```

RegularPolygon gives a regular polygon with however many sides you specify.

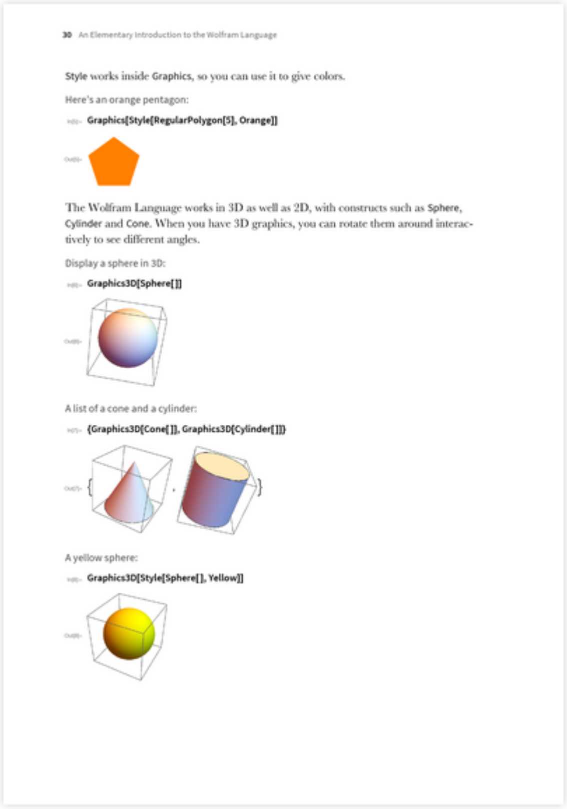
Here's a pentagon (5-sided regular polygon):

```
Graphics[RegularPolygon[5]]
```

Make a table of graphics of regular polygons with between 5 and 10 sides:

```
Table[Graphics[RegularPolygon[n]], {n, 5, 10}]
```

There's absolutely no reason not to go to 3D, and I do.



30 An Elementary Introduction to the Wolfram Language

Style works inside Graphics, so you can use it to give colors.

Here's an orange pentagon:

```
Graphics[Style[RegularPolygon[5], Orange]]
```

The Wolfram Language works in 3D as well as 2D, with constructs such as Sphere, Cylinder and Cone. When you have 3D graphics, you can rotate them around interactively to see different angles.

Display a sphere in 3D:

```
Graphics3D[Sphere[]]
```

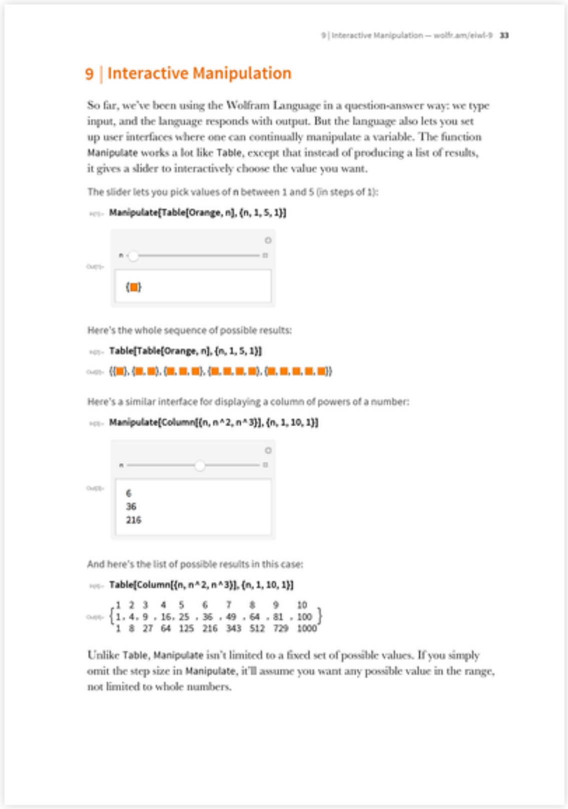
A list of a cone and a cylinder:

```
{Graphics3D[Cone[]], Graphics3D[Cylinder[]]}
```

A yellow sphere:

```
Graphics3D[Style[Sphere[], Yellow]]
```

Now we're all set up for something "advanced": interactive manipulation. It's pretty much like Table, except that one gets out a complete interactive user interface. And since we've introduced graphics, those can be part of the interface. People have seen interactive interfaces in lots of consumer software. My experience is that they're pretty excited to be able to create them from scratch themselves.


9 | Interactive Manipulation — wolfr.am/eiwl-9 33

9 | Interactive Manipulation

So far, we've been using the Wolfram Language in a question-answer way: we type input, and the language responds with output. But the language also lets you set up user interfaces where one can continually manipulate a variable. The function Manipulate works a lot like Table, except that instead of producing a list of results, it gives a slider to interactively choose the value you want.

The slider lets you pick values of n between 1 and 5 (in steps of 1):

```
Manipulate[Table[Orange, n], {n, 1, 5, 1}]
```

```
{■}
```

Here's the whole sequence of possible results:

```
Table[Table[Orange, n], {n, 1, 5, 1}]
```

```
{{■}, {■, ■}, {■, ■, ■}, {■, ■, ■, ■}, {■, ■, ■, ■, ■}}
```

Here's a similar interface for displaying a column of powers of a number:

```
Manipulate[Column[{n, n^2, n^3}], {n, 1, 10, 1}]
```

```
6
36
216
```

And here's the list of possible results in this case:

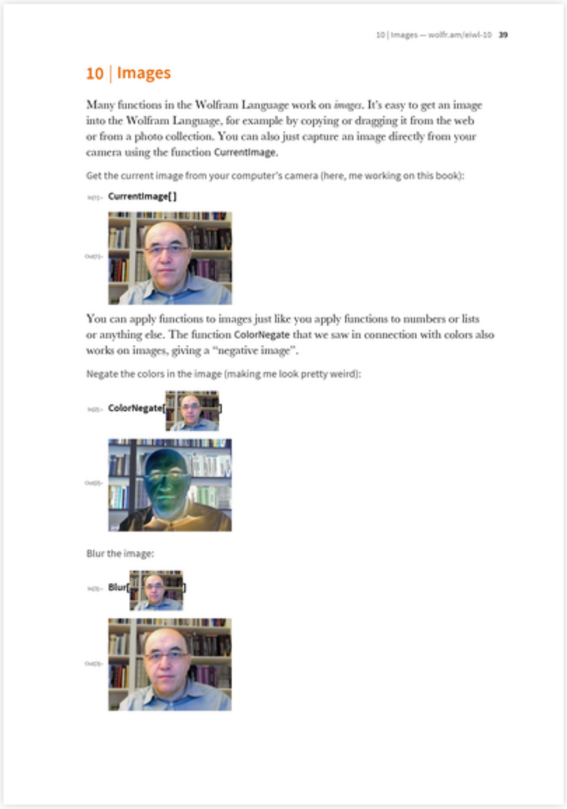
```
Table[Column[{n, n^2, n^3}], {n, 1, 10, 1}]
```

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

Unlike Table, Manipulate isn't limited to a fixed set of possible values. If you simply omit the step size in Manipulate, it'll assume you want any possible value in the range, not limited to whole numbers.

The next, perhaps surprising thing I introduce in the book is image processing. Yes, there's a lot of sophisticated computation behind image processing. But in the Wolfram Language that's all internal. And what people see are just functions—like Blur and ColorNegate—whose purposes are easy to understand.

It's also nice that people—especially kids—can compute with images they take, or drag in. And this is actually the first example in the book where there's rich data coming into a computation from outside. (I needed a sample image for the section, so, yes, I just snapped one right there—of me working on the book.)



10 | Images — wolfr.am/eiwl-10 39

10 | Images

Many functions in the Wolfram Language work on *images*. It's easy to get an image into the Wolfram Language, for example by copying or dragging it from the web or from a photo collection. You can also just capture an image directly from your camera using the function CurrentImage.

Get the current image from your computer's camera (here, me working on this book):

```
CurrentImage[]
```

You can apply functions to images just like you apply functions to numbers or lists or anything else. The function ColorNegate that we saw in connection with colors also works on images, giving a "negative image".

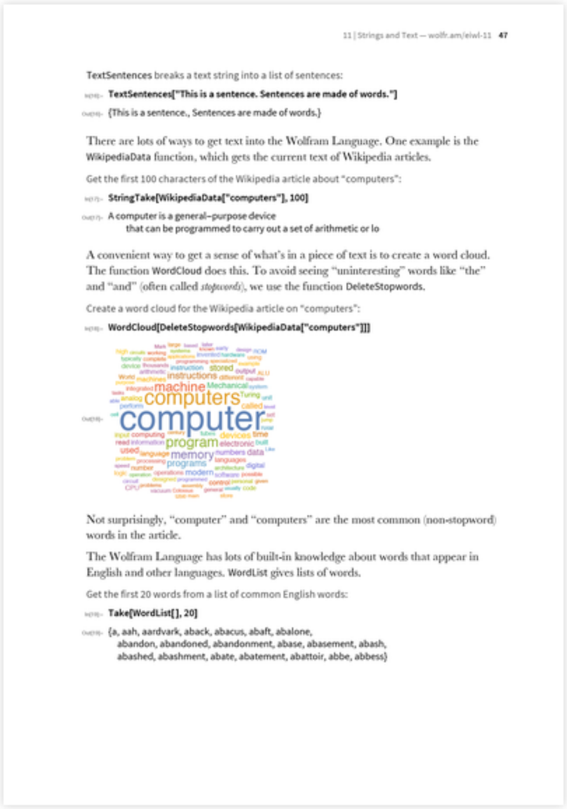
Negate the colors in the image (making me look pretty weird):

```
ColorNegate[ ]
```

Blur the image:

```
Blur[ ]
```

Next I talk about strings and text. String operations on their own are pretty dry. But in the Wolfram Language there's lots of interesting stuff that's easy to do with them—like visualizing word clouds from Wikipedia, or looking at common words in different languages.



11 | Strings and Text — wolfr.am/eiwl-11 47

TextSentences breaks a text string into a list of sentences:

```
TextSentences["This is a sentence. Sentences are made of words."]
{This is a sentence., Sentences are made of words.}
```

There are lots of ways to get text into the Wolfram Language. One example is the WikipediaData function, which gets the current text of Wikipedia articles.

Get the first 100 characters of the Wikipedia article about "computers":

```
StringTake[WikipediaData["computers"], 100]
A computer is a general-purpose device
    that can be programmed to carry out a set of arithmetic or lo
```

A convenient way to get a sense of what's in a piece of text is to create a word cloud. The function WordCloud does this. To avoid seeing "uninteresting" words like "the" and "and" (often called *stopwords*), we use the function DeleteStopwords.

Create a word cloud for the Wikipedia article on "computers":

```
WordCloud[DeleteStopwords[WikipediaData["computers"]]]
```

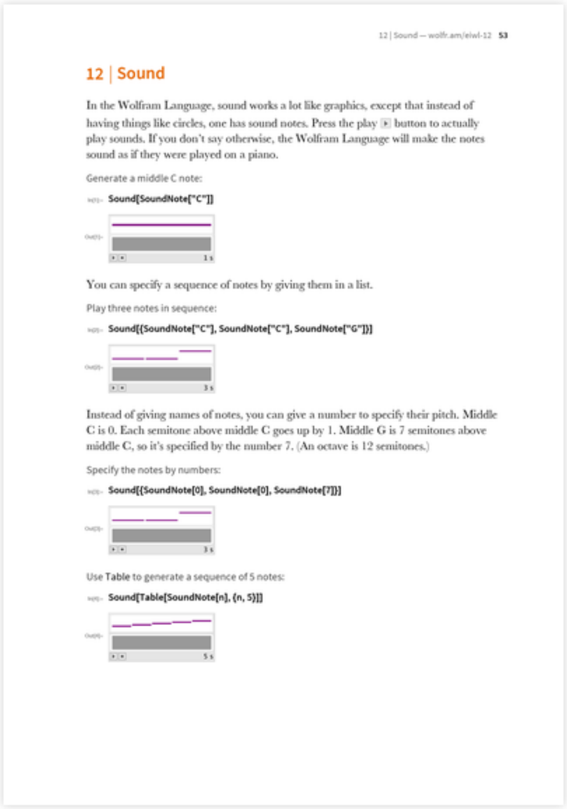
Not surprisingly, "computer" and "computers" are the most common (non-stopword) words in the article.

The Wolfram Language has lots of built-in knowledge about words that appear in English and other languages. WordList gives lists of words.

Get the first 20 words from a list of common English words:

```
Take[WordList[], 20]
{a, aah, aardvark, aback, abacus, abaft, abalone,
  abandon, abandoned, abandonment, abase, abasement, abash,
  abashed, abashment, abate, abatement, abattoir, abbe, abbess}
```

Next I cover sound, and talk about how to generate sequences of musical notes. In the printed book you can't hear them, of course, though the little score icons give some sense of what's there.



12 | Sound — wolfr.am/eiwl-12 53

12 | Sound

In the Wolfram Language, sound works a lot like graphics, except that instead of having things like circles, one has sound notes. Press the play ▸ button to actually play sounds. If you don't say otherwise, the Wolfram Language will make the notes sound as if they were played on a piano.

Generate a middle C note:

```
Sound[SoundNote["C"]]
```

You can specify a sequence of notes by giving them in a list.

Play three notes in sequence:

```
Sound[{SoundNote["C"], SoundNote["C"], SoundNote["G"]}]
```

Instead of giving names of notes, you can give a number to specify their pitch. Middle C is 0. Each semitone above middle C goes up by 1. Middle G is 7 semitones above middle C, so it's specified by the number 7. (An octave is 12 semitones.)

Specify the notes by numbers:

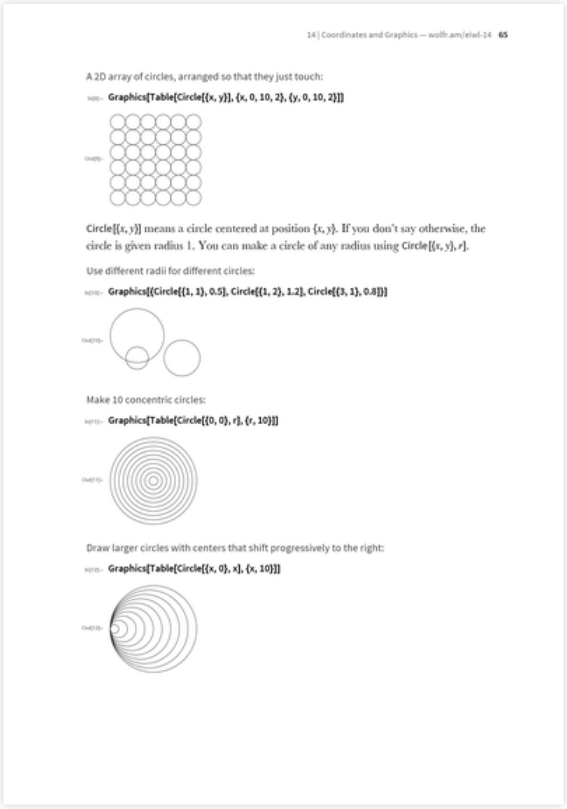
```
Sound[{SoundNote[0], SoundNote[0], SoundNote[7]}]
```

Use Table to generate a sequence of 5 notes:

```
Sound[Table[SoundNote[n], {n, 5}]]
```

One might wonder, "Why not talk about sound right after graphics?" Well, first of all, I thought it wasn't bad to mix things up a bit, to help keep the flow interesting. But more than that, there's a certain chain of dependencies between different areas. For example, the names of musical notes are specified as strings—so one has to have talked about strings before musical notes.

Next it's "Arrays, or Lists of Lists". Then it's "Coordinates and Graphics". At first, I worried that coordinates were too "mathy". But particularly after one's seen arrays, it's not so difficult to understand coordinates. And once one's got the idea of 2D coordinates, it's easy to go to 3D.



14 | Coordinates and Graphics — wolfr.am/eiwl-14 **65**

A 2D array of circles, arranged so that they just touch:

```
In[9]:= Graphics[Table[Circle[{x, y}], {x, 0, 10, 2}, {y, 0, 10, 2}]]
```

Out[9]=

Circle[{*x*, *y*}] means a circle centered at position {*x*, *y*}. If you don't say otherwise, the circle is given radius 1. You can make a circle of any radius using Circle[{*x*, *y*}, *r*].

Use different radii for different circles:

```
In[10]:= Graphics[{Circle[{1, 1}, 0.5], Circle[{1, 2}, 1.2], Circle[{3, 1}, 0.8]}]
```

Out[10]=

Make 10 concentric circles:

```
In[11]:= Graphics[Table[Circle[{0, 0}, r], {r, 10}]]
```

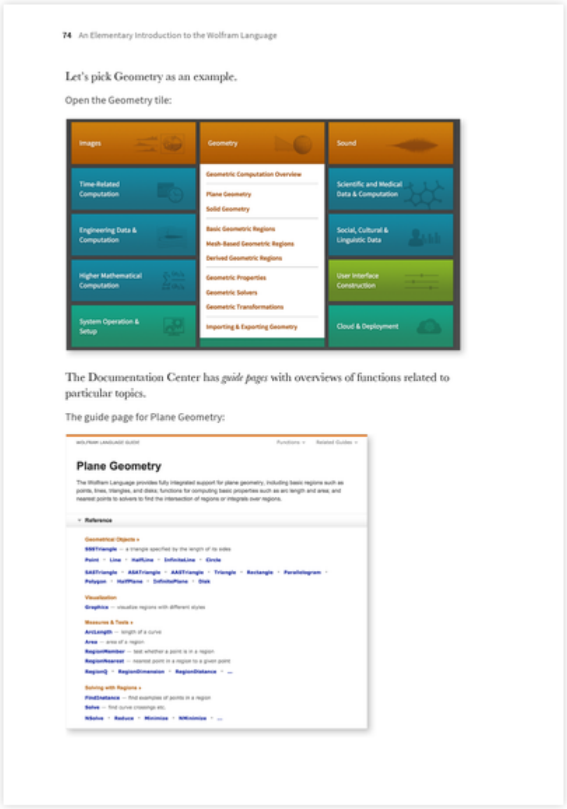
Out[11]=

Draw larger circles with centers that shift progressively to the right:

```
In[12]:= Graphics[Table[Circle[{x, 0}, x], {x, 10}]]
```

Out[12]=

By this point in the book, people already know how to do some useful and real things with the Wolfram Language. So I made the next section a kind of interlude—a meta-section that gives a sense of the overall scope of the Wolfram Language, and also shows how to find information on specific topics and functions.



74 An Elementary Introduction to the Wolfram Language

Let's pick Geometry as an example.

Open the Geometry tile:

Images
Geometry
Sound
Time-Related Computation
Geometric Computation Overview
Plane Geometry
Solid Geometry
Scientific and Medical Data & Computation
Engineering Data & Computation
Basic Geometric Regions
Mesh-Based Geometric Regions
Derived Geometric Regions
Social, Cultural & Linguistic Data
Higher Mathematical Computation
Geometric Properties
Geometric Solvers
Geometric Transformations
User Interface Construction
System Operation & Setup
Importing & Exporting Geometry
Cloud & Deployment

The Documentation Center has *guide pages* with overviews of functions related to particular topics.

The guide page for Plane Geometry:

WOLFRAM LANGUAGE GUIDE Functions ▾ Related Guides ▾

Plane Geometry

The Wolfram Language provides fully integrated support for plane geometry, including basic regions such as points, lines, triangles, and disks; functions for computing basic properties such as arc length and area; and nearest points to solvers to find the intersection of regions or integrals over regions.

▾ **Reference**

Geometrical Objects »

SSSTriangle — a triangle specified by the length of its sides

Point · **Line** · **HalfLine** · **InfiniteLine** · **Circle**

SASTriangle · **ASATriangle** · **AASTriangle** · **Triangle** · **Rectangle** · **Parallelogram** · **Polygon** · **HalfPlane** · **InfinitePlane** · **Disk**

Visualization

Graphics — visualize regions with different styles

Measures & Tools »

ArcLength — length of a curve

Area — area of a region

RegionMember — test whether a point is in a region

RegionNearest — nearest point in a region to a given point

RegionQ · **RegionDimension** · **RegionDistance** · ...

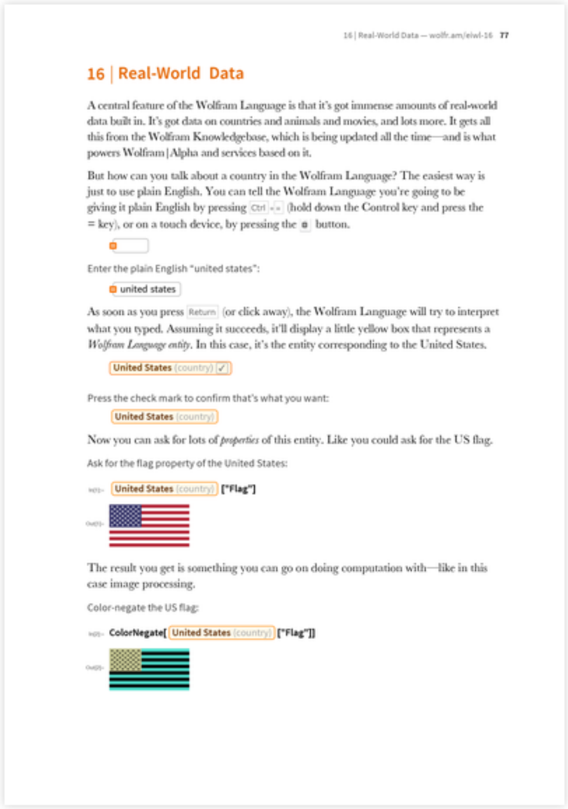
Solving with Regions »

FindInstance — find examples of points in a region

Solve — find curve crossings etc.

NSolve · **Reduce** · **Minimize** · **NMinimize** · ...

Now that people have seen a bit about abstract computation, it's time to talk about real-world data, and to show how to access the vast amount of data that the Wolfram Language shares with Wolfram|Alpha.



16 | Real-World Data — wolfr.am/eiwl-16 77

16 | Real-World Data

A central feature of the Wolfram Language is that it's got immense amounts of real-world data built in. It's got data on countries and animals and movies, and lots more. It gets all this from the Wolfram Knowledgebase, which is being updated all the time—and is what powers Wolfram|Alpha and services based on it.

But how can you talk about a country in the Wolfram Language? The easiest way is just to use plain English. You can tell the Wolfram Language you're going to be giving it plain English by pressing Ctrl = (hold down the Control key and press the = key), or on a touch device, by pressing the button.

Enter the plain English "united states":

```
united states
```

As soon as you press Return (or click away), the Wolfram Language will try to interpret what you typed. Assuming it succeeds, it'll display a little yellow box that represents a *Wolfram Language entity*. In this case, it's the entity corresponding to the United States.

```
United States (country) ✓
```

Press the check mark to confirm that's what you want:

```
United States (country)
```

Now you can ask for lots of *properties* of this entity. Like you could ask for the US flag.

Ask for the flag property of the United States:

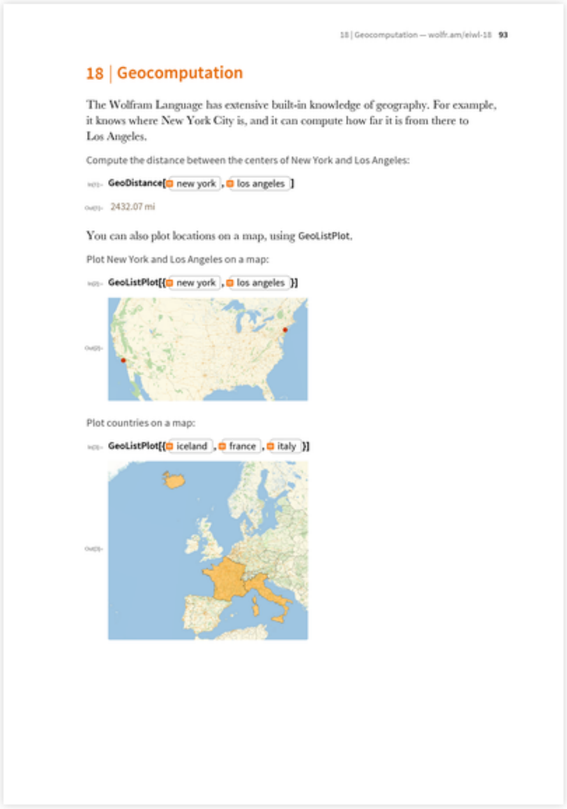
```
In[1]:= United States (country) ["Flag"]
```

The result you get is something you can go on doing computation with—like in this case image processing.

Color-negate the US flag:

```
In[2]:= ColorNegate[ United States (country) ["Flag"]]
```

Lots of real-world data involves units—so the next section is devoted to working with units. Once that's done, we can talk about geocomputation: things like finding distances on the Earth, and drawing maps.



18 | Geocomputation

The Wolfram Language has extensive built-in knowledge of geography. For example, it knows where New York City is, and it can compute how far it is from there to Los Angeles.

Compute the distance between the centers of New York and Los Angeles:

```
GeoDistance[new york, los angeles]
```

```
2432.07 mi
```

You can also plot locations on a map, using **GeoListPlot**.

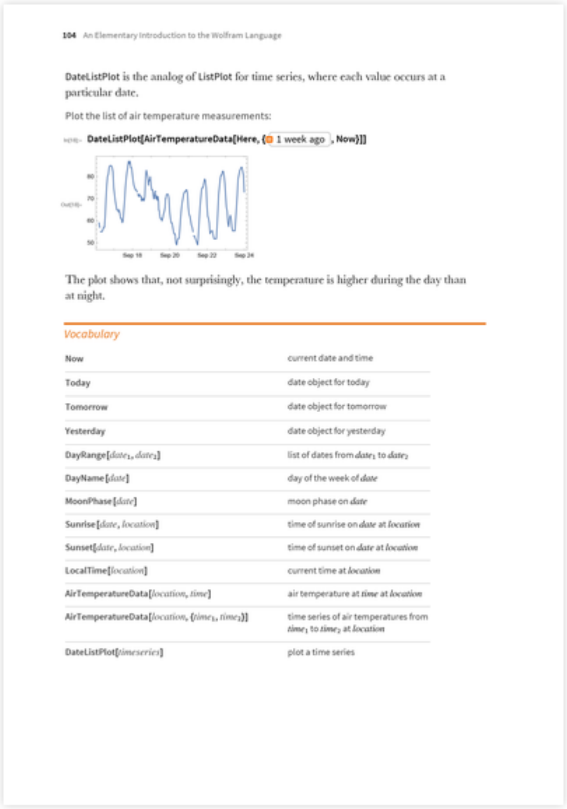
Plot New York and Los Angeles on a map:

```
GeoListPlot[{new york, los angeles}]
```

Plot countries on a map:

```
GeoListPlot[{iceland, france, italy}]
```

After that I talk about dates and times. One might think this wouldn't be an interesting or useful topic. But it's actually a really good example of real-world computation, and it's also something one uses all over the place.



DateListPlot is the analog of **ListPlot** for time series, where each value occurs at a particular date.

Plot the list of air temperature measurements:

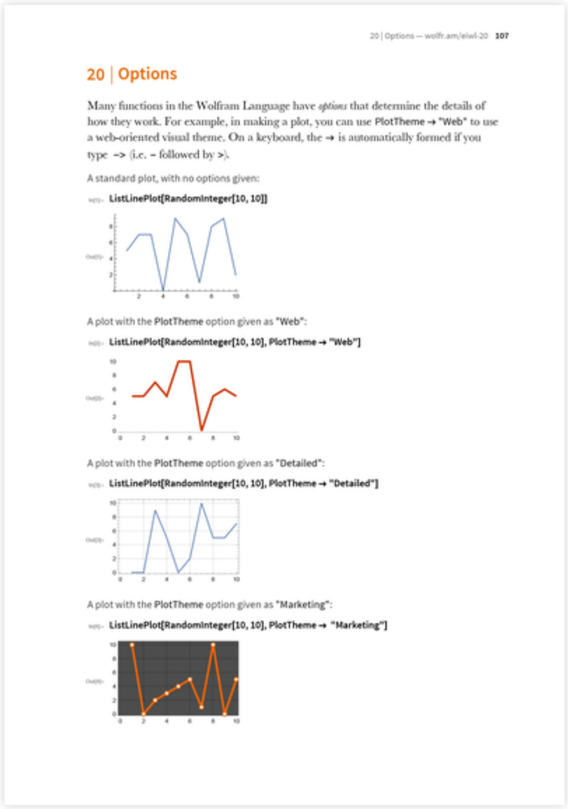
```
DateListPlot[AirTemperatureData[Here, {1 week ago, Now}]]
```

The plot shows that, not surprisingly, the temperature is higher during the day than at night.

Vocabulary

Now	current date and time
Today	date object for today
Tomorrow	date object for tomorrow
Yesterday	date object for yesterday
DayRange[*date*$_1$, *date*$_2$]	list of dates from *date*$_1$ to *date*$_2$
DayName[*date*]	day of the week of *date*
MoonPhase[*date*]	moon phase on *date*
Sunrise[*date*, *location*]	time of sunrise on *date* at *location*
Sunset[*date*, *location*]	time of sunset on *date* at *location*
LocalTime[*location*]	current time at *location*
AirTemperatureData[*location*, *time*]	air temperature at *time* at *location*
AirTemperatureData[*location*, {*time*$_1$, *time*$_2$}]	time series of air temperatures from *time*$_1$ to *time*$_2$ at *location*
DateListPlot[*timeseries*]	plot a time series

The Wolfram Language is big. But it's based on a small number of ideas that are consistently used over and over again. One of the important objectives in the book is to cover these ideas. And the next section—on options—covers one such simple idea that's widely used in practice.



20 | Options

Many functions in the Wolfram Language have *options* that determine the details of how they work. For example, in making a plot, you can use **PlotTheme → "Web"** to use a web-oriented visual theme. On a keyboard, the → is automatically formed if you type -> (i.e. – followed by >).

A standard plot, with no options given:

```
ListLinePlot[RandomInteger[10, 10]]
```

A plot with the **PlotTheme** option given as "Web":

```
ListLinePlot[RandomInteger[10, 10], PlotTheme → "Web"]
```

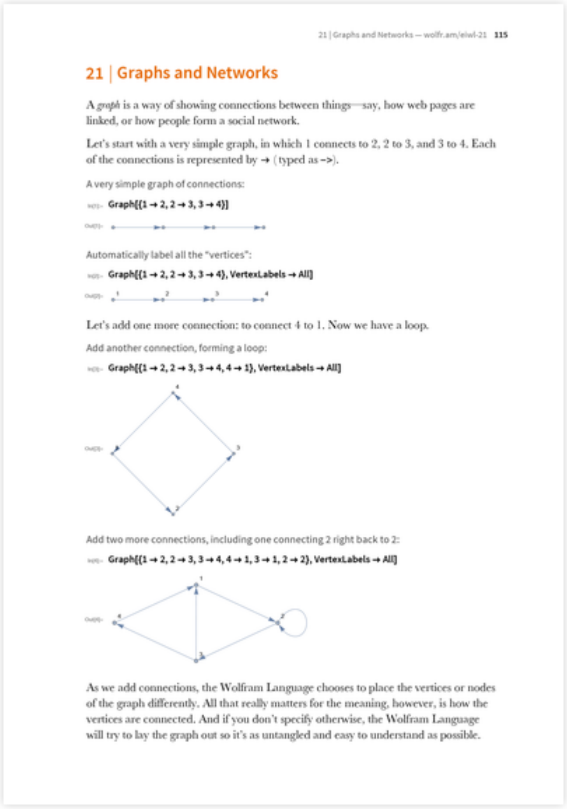
A plot with the **PlotTheme** option given as "Detailed":

```
ListLinePlot[RandomInteger[10, 10], PlotTheme → "Detailed"]
```

A plot with the **PlotTheme** option given as "Marketing":

```
ListLinePlot[RandomInteger[10, 10], PlotTheme → "Marketing"]
```

After covering options, we're set to talk about something that's often viewed as a quite advanced topic: graphs and networks. But my experience is that in modern times, people have seen enough graphs and networks in their everyday life that they don't have much trouble understanding them in the Wolfram Language. Of course, it helps a lot that the language can manipulate them directly, as just another example of symbolic objects.



21 | Graphs and Networks

A *graph* is a way of showing connections between things—say, how web pages are linked, or how people form a social network.

Let's start with a very simple graph, in which 1 connects to 2, 2 to 3, and 3 to 4. Each of the connections is represented by → (typed as ->).

A very simple graph of connections:

```
Graph[{1 → 2, 2 → 3, 3 → 4}]
```

Automatically label all the "vertices":

```
Graph[{1 → 2, 2 → 3, 3 → 4}, VertexLabels → All]
```

Let's add one more connection: to connect 4 to 1. Now we have a loop.

Add another connection, forming a loop:

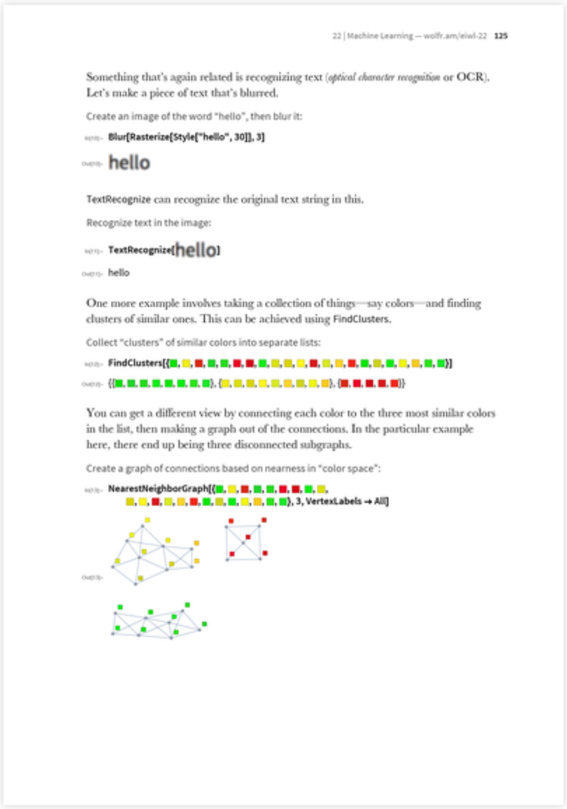
```
Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 1}, VertexLabels → All]
```

Add two more connections, including one connecting 2 right back to 2:

```
Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 1, 3 → 1, 2 → 2}, VertexLabels → All]
```

As we add connections, the Wolfram Language chooses to place the vertices or nodes of the graph differently. All that really matters for the meaning, however, is how the vertices are connected. And if you don't specify otherwise, the Wolfram Language will try to lay the graph out so it's as untangled and easy to understand as possible.

After graphs and networks, we're ready for another seemingly very advanced topic: machine learning. But even though the internal algorithms for machine learning are complicated, the actual functions that do it in the Wolfram Language are perfectly easy to understand. And what's nice is that by doing a bunch of examples with them, one can start to get pretty good high-level intuition about the core ideas of machine learning.



Something that's again related is recognizing text (*optical character recognition* or OCR). Let's make a piece of text that's blurred.

Create an image of the word "hello", then blur it:

```
Blur[Rasterize[Style["hello", 30]], 3]
```

hello

TextRecognize can recognize the original text string in this.

Recognize text in the image:

```
TextRecognize[hello]
```

```
hello
```

One more example involves taking a collection of things—say colors—and finding clusters of similar ones. This can be achieved using FindClusters.

Collect "clusters" of similar colors into separate lists:

```
FindClusters[{🟩, 🟨, 🟥, 🟩, 🟩, 🟥, 🟥, 🟩, 🟨, 🟨, 🟨, 🟥, 🟨, 🟨, 🟥, 🟩, 🟨, 🟩, 🟨, 🟨, 🟩, 🟩}]
```

```
{{🟩, 🟩, 🟩, 🟩, 🟩, 🟩, 🟩, 🟩}, {🟨, 🟨, 🟨, 🟨, 🟨, 🟨, 🟨, 🟨, 🟨}, {🟥, 🟥, 🟥, 🟥, 🟥}}
```

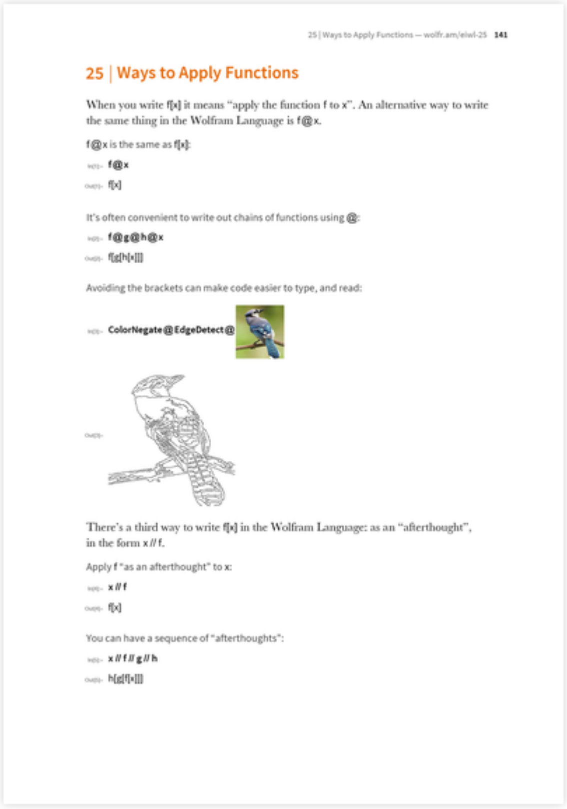
You can get a different view by connecting each color to the three most similar colors in the list, then making a graph out of the connections. In the particular example here, there end up being three disconnected subgraphs.

Create a graph of connections based on nearness in "color space":

```
NearestNeighborGraph[{🟩, 🟨, 🟥, 🟩, 🟩, 🟥, 🟥, 🟩, 🟨,
    🟨, 🟨, 🟥, 🟨, 🟨, 🟥, 🟩, 🟨, 🟩, 🟨, 🟨, 🟩, 🟩}, 3, VertexLabels → All]
```

Throughout the book, I try to keep things as simple as possible. But sometimes that means I have to go back for a deeper view of a topic I've already covered. "More about Numbers" and "More Forms of Visualization" are two examples of doing this—covering things that would have gotten in the way when numbers and visualization were first introduced, but that need to be said to get a full understanding of these areas.

Functional Programming The next few sections tackle the important and incredibly powerful topic of functional programming. In the past, functional programming tended to be viewed as a sophisticated topic—and certainly not something to teach people who are first learning about programming. But I think in the Wolfram Language the picture has changed—and it's now possible to explain functional programming in a way that people will find easy to understand. I start by just talking more abstractly about the process of applying a function.



25 | Ways to Apply Functions — wolfr.am/eiwl-25 141

25 | Ways to Apply Functions

When you write f[x] it means "apply the function f to x". An alternative way to write the same thing in the Wolfram Language is f@x.

f@x is the same as f[x]:

```
f@x
f[x]
```

It's often convenient to write out chains of functions using @:

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

Avoiding the brackets can make code easier to type, and read:

```
ColorNegate@EdgeDetect@
```

There's a third way to write f[x] in the Wolfram Language: as an "afterthought", in the form x // f.

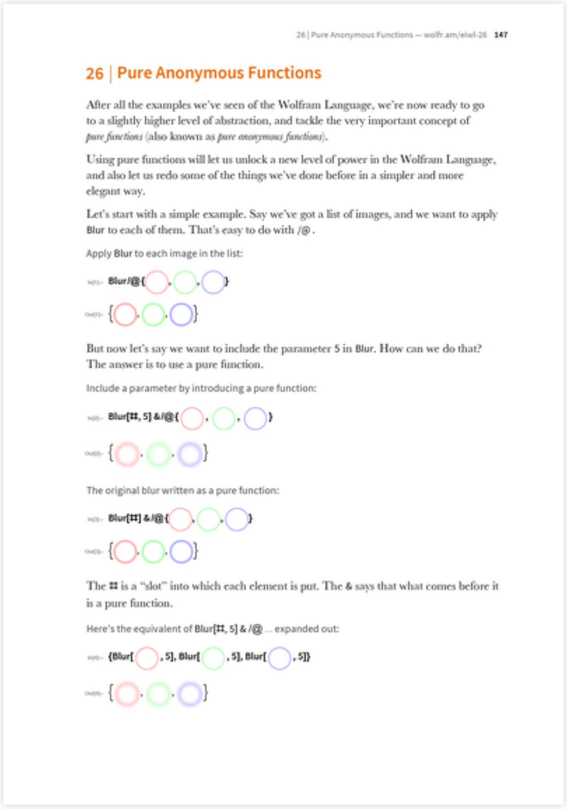
Apply f "as an afterthought" to x:

```
x // f
f[x]
```

You can have a sequence of "afterthoughts":

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

The big thing this does is set me up to talk about pure anonymous functions. In principle I could have talked about these much sooner, but I think it's important for people to have seen many different kinds of examples of how functions are used in general—because that's what's needed to motivate pure functions.



26 | Pure Anonymous Functions

After all the examples we've seen of the Wolfram Language, we're now ready to go to a slightly higher level of abstraction, and tackle the very important concept of *pure functions* (also known as *pure anonymous functions*).

Using pure functions will let us unlock a new level of power in the Wolfram Language, and also let us redo some of the things we've done before in a simpler and more elegant way.

Let's start with a simple example. Say we've got a list of images, and we want to apply **Blur** to each of them. That's easy to do with /@ .

Apply **Blur** to each image in the list:

```
Blur/@{ , , }
```

But now let's say we want to include the parameter 5 in **Blur**. How can we do that? The answer is to use a pure function.

Include a parameter by introducing a pure function:

```
Blur[#, 5] &/@{ , , }
```

The original blur written as a pure function:

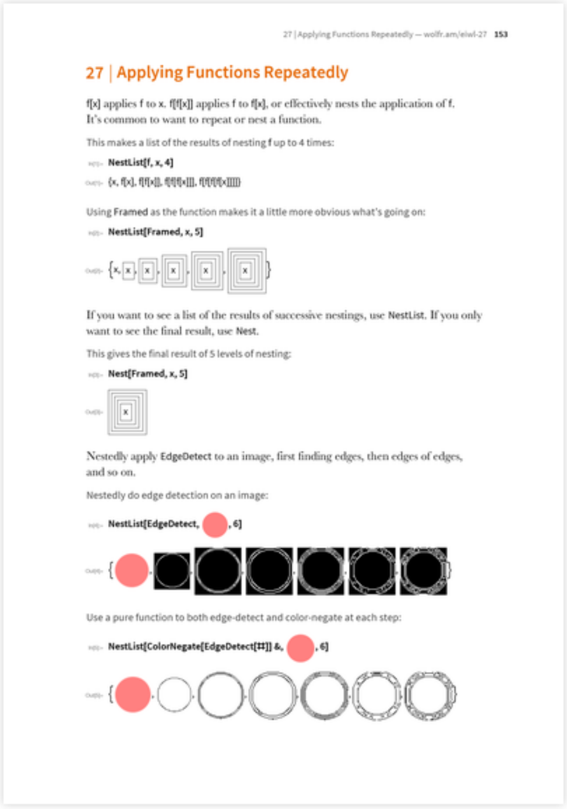
```
Blur[#] &/@{ , , }
```

The # is a "slot" into which each element is put. The & says that what comes before it is a pure function.

Here's the equivalent of **Blur[#, 5] & /@** ... expanded out:

```
{Blur[ , 5], Blur[ , 5], Blur[ , 5]}
```

The next section is where some of the real power of functional programming starts to shine through. In the abstract, functions like NestList and NestGraph sound pretty complicated and abstract. But by this point in the book, we've covered enough of the Wolfram Language that there are plenty of concrete examples to give—that are quite easy to understand.



27 | Applying Functions Repeatedly

f[x] applies f to x. f[f[x]] applies f to f[x], or effectively nests the application of f. It's common to want to repeat or nest a function.

This makes a list of the results of nesting f up to 4 times:

```
NestList[f, x, 4]
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}
```

Using **Framed** as the function makes it a little more obvious what's going on:

```
NestList[Framed, x, 5]
```

If you want to see a list of the results of successive nestings, use **NestList**. If you only want to see the final result, use **Nest**.

This gives the final result of 5 levels of nesting:

```
Nest[Framed, x, 5]
```

Nestedly apply **EdgeDetect** to an image, first finding edges, then edges of edges, and so on.

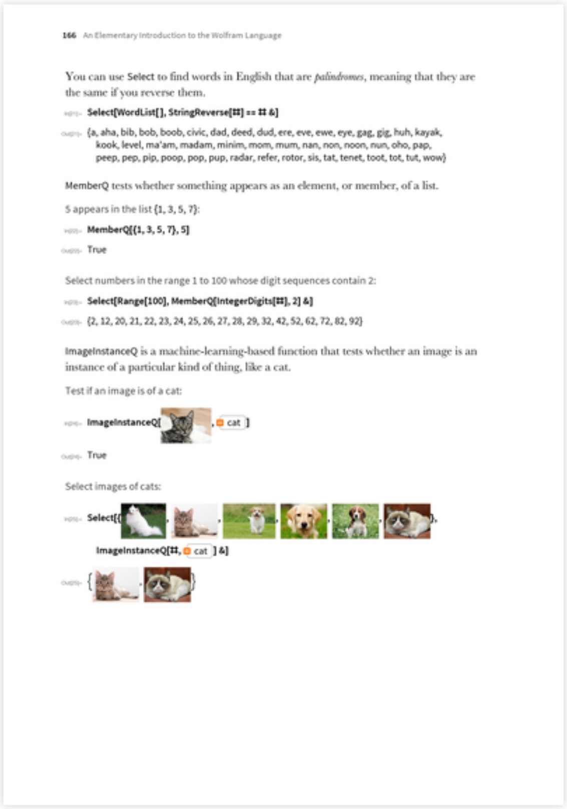
Nestedly do edge detection on an image:

```
NestList[EdgeDetect, , 6]
```

Use a pure function to both edge-detect and color-negate at each step:

```
NestList[ColorNegate[EdgeDetect[#]] &, , 6]
```

The next several sections cover areas of the language that are unlocked as soon as one understands pure functions. There are lots of powerful programming techniques that emerge from a smaller number of ideas.



166 An Elementary Introduction to the Wolfram Language

You can use Select to find words in English that are *palindromes*, meaning that they are the same if you reverse them.

```
Select[WordList[], StringReverse[#] == # &]
{a, aha, bib, bob, boob, civic, dad, deed, dud, ere, eve, ewe, eye, gag, gig, huh, kayak,
  kook, level, ma'am, madam, minim, mom, mum, nan, non, noon, nun, oho, pap,
  peep, pep, pip, poop, pop, pup, radar, refer, rotor, sis, tat, tenet, toot, tot, tut, wow}
```

MemberQ tests whether something appears as an element, or member, of a list.

5 appears in the list {1, 3, 5, 7}:

```
MemberQ[{1, 3, 5, 7}, 5]
True
```

Select numbers in the range 1 to 100 whose digit sequences contain 2:

```
Select[Range[100], MemberQ[IntegerDigits[#], 2] &]
{2, 12, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 32, 42, 52, 62, 72, 82, 92}
```

ImageInstanceQ is a machine-learning-based function that tests whether an image is an instance of a particular kind of thing, like a cat.

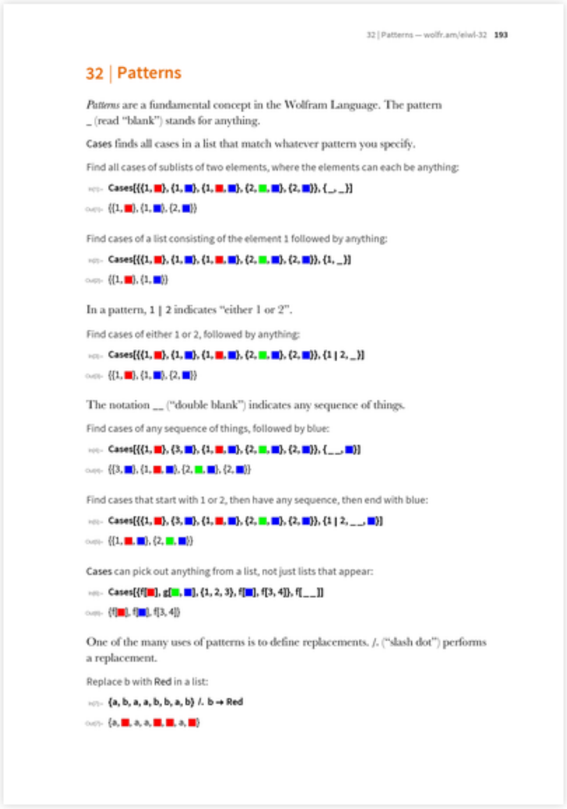
Test if an image is of a cat:

```
ImageInstanceQ[ , cat ]
True
```

Select images of cats:

```
Select[{ , , , , , },
  ImageInstanceQ[#, cat ] &]
{ , }
```

After functional programming, the next big topics are patterns and pattern-based programming. I could have chosen to talk about patterns earlier in the book, but they weren't really needed until now.



32 | Patterns

Patterns are a fundamental concept in the Wolfram Language. The pattern _ (read "blank") stands for anything.

Cases finds all cases in a list that match whatever pattern you specify.

Find all cases of sublists of two elements, where the elements can each be anything:

```
Cases[{{1, 🟥}, {1, 🟦}, {1, 🟥, 🟦}, {2, 🟩, 🟦}, {2, 🟦}}, {_, _}]
{{1, 🟥}, {1, 🟦}, {2, 🟦}}
```

Find cases of a list consisting of the element 1 followed by anything:

```
Cases[{{1, 🟥}, {1, 🟦}, {1, 🟥, 🟦}, {2, 🟩, 🟦}, {2, 🟦}}, {1, _}]
{{1, 🟥}, {1, 🟦}}
```

In a pattern, **1 | 2** indicates "either 1 or 2".

Find cases of either 1 or 2, followed by anything:

```
Cases[{{1, 🟥}, {1, 🟦}, {1, 🟥, 🟦}, {2, 🟩, 🟦}, {2, 🟦}}, {1 | 2, _}]
{{1, 🟥}, {1, 🟦}, {2, 🟦}}
```

The notation __ ("double blank") indicates any sequence of things.

Find cases of any sequence of things, followed by blue:

```
Cases[{{1, 🟥}, {3, 🟦}, {1, 🟥, 🟦}, {2, 🟩, 🟦}, {2, 🟦}}, {__, 🟦}]
{{3, 🟦}, {1, 🟥, 🟦}, {2, 🟩, 🟦}, {2, 🟦}}
```

Find cases that start with 1 or 2, then have any sequence, then end with blue:

```
Cases[{{1, 🟥}, {3, 🟦}, {1, 🟥, 🟦}, {2, 🟩, 🟦}, {2, 🟦}}, {1 | 2, __, 🟦}]
{{1, 🟥, 🟦}, {2, 🟩, 🟦}}
```

Cases can pick out anything from a list, not just lists that appear:

```
Cases[{f[🟥], g[🟩, 🟦], {1, 2, 3}, f[🟦], f[3, 4]}, f[__]]
{f[🟥], f[🟦], f[3, 4]}
```

One of the many uses of patterns is to define replacements. /. ("slash dot") performs a replacement.

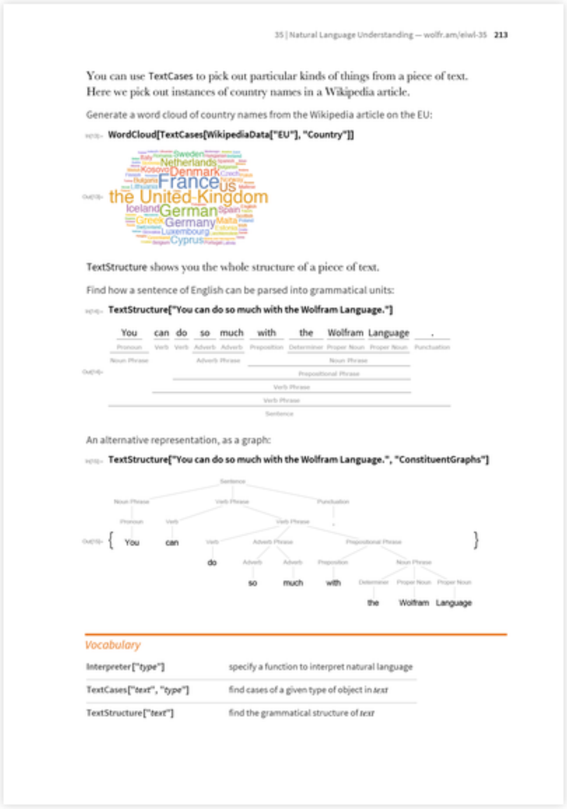
Replace b with **Red** in a list:

```
{a, b, a, a, b, b, a, b} /. b → Red
{a, 🟥, a, a, 🟥, 🟥, a, 🟥}
```

What makes patterns so powerful in the Wolfram Language is something much more fundamental: the uniform structure of everything in the language, based on symbolic expressions. If I were writing a formal specification of the Wolfram Language, I would start with symbolic expressions. And I might do the same if I were writing a book for theoretical computer scientists or pure mathematicians.

It's not that symbolic expressions are a difficult concept to understand. It's just that without seeing how things actually work in practice in the Wolfram Language, it's difficult to motivate abstractly studying them. But now it makes sense to talk about them, not least because they let one see the full power of what's possible with patterns.

The Whole Stack At this point in the book, we're getting ready to see how to actually deploy things like web apps. There are a few more pieces to put in place to get there. I talk about associations—and then I talk about natural language understanding. Internally, the way natural language understanding works is complex. But at the level of the Wolfram Language, it's easy to use—though to see how to connect it into things, it's helpful to know about pure functions.



35 | Natural Language Understanding — wolfr.am/eiwl-35 213

You can use TextCases to pick out particular kinds of things from a piece of text. Here we pick out instances of country names in a Wikipedia article.

Generate a word cloud of country names from the Wikipedia article on the EU:

```
WordCloud[TextCases[WikipediaData["EU"], "Country"]]
```

TextStructure shows you the whole structure of a piece of text.

Find how a sentence of English can be parsed into grammatical units:

```
TextStructure["You can do so much with the Wolfram Language."]
```

```
You      can   do    so      much    with         the          Wolfram       Language      .
Pronoun  Verb  Verb  Adverb  Adverb  Preposition  Determiner   Proper Noun   Proper Noun   Punctuation
Noun Phrase          Adverb Phrase                Noun Phrase
                                     Prepositional Phrase
                     Verb Phrase
               Verb Phrase
Sentence
```

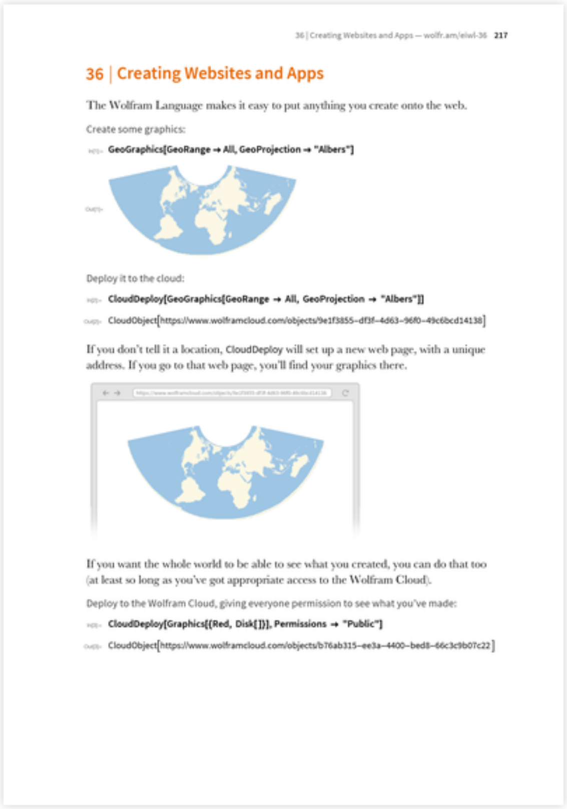
An alternative representation, as a graph:

```
TextStructure["You can do so much with the Wolfram Language.", "ConstituentGraphs"]
```

Vocabulary

Interpreter["*type*"]	specify a function to interpret natural language
TextCases["*text*", "*type*"]	find cases of a given type of object in *text*
TextStructure["*text*"]	find the grammatical structure of *text*

OK, so now everything is ready to talk about deploying things to the web. And at this point, people will be able to start creating useful, practical pieces of software that they can share with the world.



36 | Creating Websites and Apps — wolfr.am/eiwl-36 217

36 | Creating Websites and Apps

The Wolfram Language makes it easy to put anything you create onto the web.

Create some graphics:

```
GeoGraphics[GeoRange → All, GeoProjection → "Albers"]
```

Deploy it to the cloud:

```
CloudDeploy[GeoGraphics[GeoRange → All, GeoProjection → "Albers"]]
CloudObject[https://www.wolframcloud.com/objects/9e1f3855-df3f-4d63-96f0-49c6bcd14138]
```

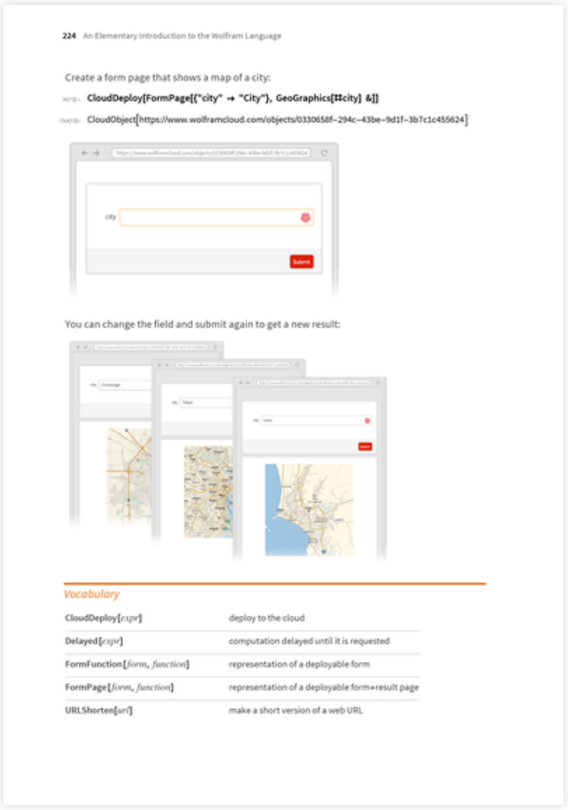
If you don't tell it a location, CloudDeploy will set up a new web page, with a unique address. If you go to that web page, you'll find your graphics there.

If you want the whole world to be able to see what you created, you can do that too (at least so long as you've got appropriate access to the Wolfram Cloud).

Deploy to the Wolfram Cloud, giving everyone permission to see what you've made:

```
CloudDeploy[Graphics[{Red, Disk[]}], Permissions → "Public"]
CloudObject[https://www.wolframcloud.com/objects/b76ab315-ee3a-4400-bed8-66c3c9b07c22]
```

It's taken 220 pages or so. But to me that's an amazingly small number of pages to go from zero to what are essentially professional-grade web apps. If we'd just been talking about some very specific kind of app, it wouldn't be so impressive. But we're talking about extremely general kinds of apps, that do pretty much any kind of computation.



224 An Elementary Introduction to the Wolfram Language

Create a form page that shows a map of a city:

```
In[13]:= CloudDeploy[FormPage[{"city" → "City"}, GeoGraphics[#city] &]]
Out[13]= CloudObject[https://www.wolframcloud.com/objects/0330658f-294c-43be-9d1f-3b7c1c455624]
```

You can change the field and submit again to get a new result:

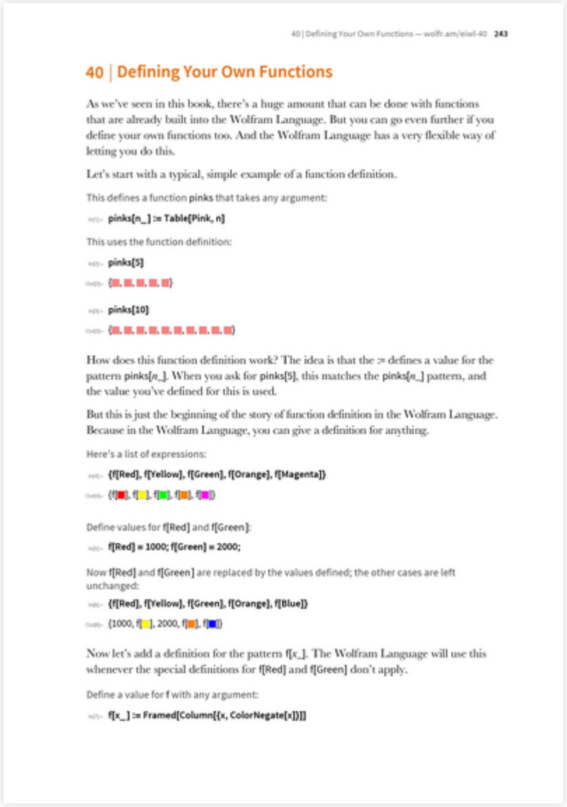
Vocabulary

CloudDeploy[*expr*]	deploy to the cloud
Delayed[*expr*]	computation delayed until it is requested
FormFunction[*form*, *function*]	representation of a deployable form
FormPage[*form*, *function*]	representation of a deployable form+result page
URLShorten[*url*]	make a short version of a web URL

Assigning Values to Variables If you open a book about a traditional programming language like C++ or Java, one of the first things you're likely to see is a discussion of assigning values to variables. But in my book I don't do this until Section 38. At some level, this might seem bizarre—but it really isn't. Because in the Wolfram Language you can do an amazing amount—including for example deploying a complete web app—without ever needing to assign a value to a variable.

And this is actually one of the reasons why it's so easy to learn the Wolfram Language. Because if you don't assign values to variables, every piece of code in the language stands alone, and will do the same thing whenever it's run. But as soon as you're assigning values to variables, there's hidden state, and your code will do different things depending on what values variables happen to have.

Still, having talked about assigning values to variables—as well as about patterns—we're ready to talk about defining your own functions, which is the way to build up more and more sophisticated functionality in the Wolfram Language.



40 | Defining Your Own Functions

As we've seen in this book, there's a huge amount that can be done with functions that are already built into the Wolfram Language. But you can go even further if you define your own functions too. And the Wolfram Language has a very flexible way of letting you do this.

Let's start with a typical, simple example of a function definition.

This defines a function **pinks** that takes any argument:

```
pinks[n_] := Table[Pink, n]
```

This uses the function definition:

```
pinks[5]
```

```
{🟥, 🟥, 🟥, 🟥, 🟥}
```

```
pinks[10]
```

```
{🟥, 🟥, 🟥, 🟥, 🟥, 🟥, 🟥, 🟥, 🟥, 🟥}
```

How does this function definition work? The idea is that the := defines a value for the pattern **pinks[*n*_]**. When you ask for **pinks[5]**, this matches the **pinks[*n*_]** pattern, and the value you've defined for this is used.

But this is just the beginning of the story of function definition in the Wolfram Language. Because in the Wolfram Language, you can give a definition for anything.

Here's a list of expressions:

```
{f[Red], f[Yellow], f[Green], f[Orange], f[Magenta]}
```

```
{f[🟥], f[🟨], f[🟩], f[🟧], f[🟪]}
```

Define values for **f[Red]** and **f[Green]**:

```
f[Red] = 1000; f[Green] = 2000;
```

Now **f[Red]** and **f[Green]** are replaced by the values defined; the other cases are left unchanged:

```
{f[Red], f[Yellow], f[Green], f[Orange], f[Blue]}
```

```
{1000, f[🟨], 2000, f[🟧], f[🟦]}
```

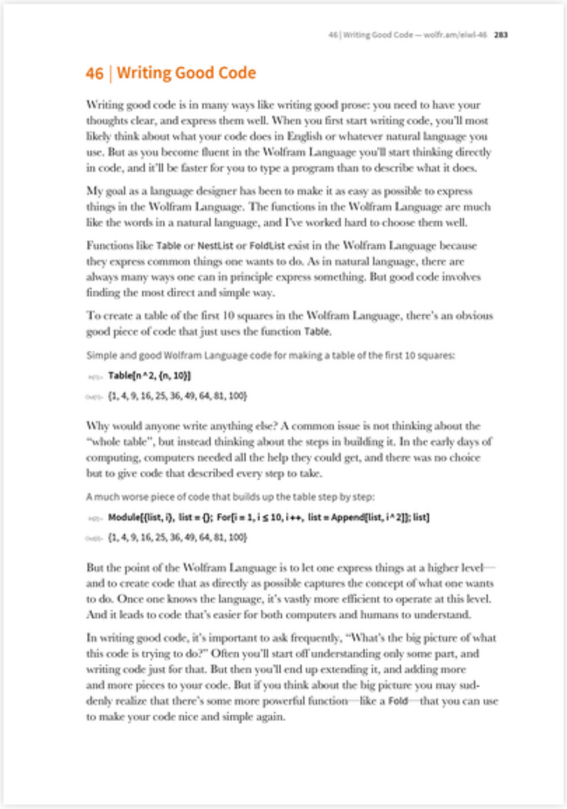
Now let's add a definition for the pattern **f[*x*_]**. The Wolfram Language will use this whenever the special definitions for **f[Red]** and **f[Green]** don't apply.

Define a value for **f** with any argument:

```
f[x_] := Framed[Column[{x, ColorNegate[x]}]]
```

At this point, you're pretty much set in terms of the basic concepts of the Wolfram Language. But the last few sections of the book cover some important practical extensions. There's a section on string patterns and templates. There's a section on storing things, locally and in the cloud. There's a section on importing and exporting. And there's a section on datasets. Not everyone who uses the Wolfram Language will ever need datasets, but when you're dealing with large amounts of structured data they're very useful. And they provide an interesting example that makes use of many different ideas from the Wolfram Language.

Essay Sections At the end of the book, I have what are basically essay sections: about writing good code, about debugging and about being a programmer. My goal in these sections is to build on the way of thinking that I hope people have developed from reading the rest of the book, and then to communicate some more abstract principles.



46 | Writing Good Code

Writing good code is in many ways like writing good prose: you need to have your thoughts clear, and express them well. When you first start writing code, you'll most likely think about what your code does in English or whatever natural language you use. But as you become fluent in the Wolfram Language you'll start thinking directly in code, and it'll be faster for you to type a program than to describe what it does.

My goal as a language designer has been to make it as easy as possible to express things in the Wolfram Language. The functions in the Wolfram Language are much like the words in a natural language, and I've worked hard to choose them well.

Functions like **Table** or **NestList** or **FoldList** exist in the Wolfram Language because they express common things one wants to do. As in natural language, there are always many ways one can in principle express something. But good code involves finding the most direct and simple way.

To create a table of the first 10 squares in the Wolfram Language, there's an obvious good piece of code that just uses the function **Table**.

Simple and good Wolfram Language code for making a table of the first 10 squares:

```
In[1]:= Table[n^2, {n, 10}]
Out[1]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Why would anyone write anything else? A common issue is not thinking about the "whole table", but instead thinking about the steps in building it. In the early days of computing, computers needed all the help they could get, and there was no choice but to give code that described every step to take.

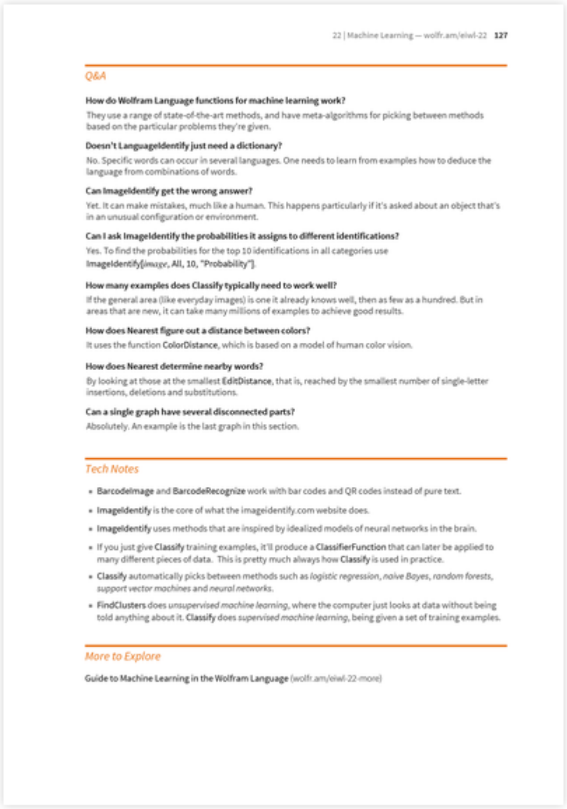
A much worse piece of code that builds up the table step by step:

```
In[2]:= Module[{list, i}, list = {}; For[i = 1, i ≤ 10, i++, list = Append[list, i^2]]; list]
Out[2]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

But the point of the Wolfram Language is to let one express things at a higher level—and to create code that as directly as possible captures the concept of what one wants to do. Once one knows the language, it's vastly more efficient to operate at this level. And it leads to code that's easier for both computers and humans to understand.

In writing good code, it's important to ask frequently, "What's the big picture of what this code is trying to do?" Often you'll start off understanding only some part, and writing code just for that. But then you'll end up extending it, and adding more and more pieces to your code. But if you think about the big picture you may suddenly realize that there's some more powerful function—like a **Fold**—that you can use to make your code nice and simple again.

Structuring the Presentation I said at the beginning of this post that the book is essentially written as a conversation. In almost every section, I found it convenient to add two additional parts: Q&A and Tech Notes. The goal of Q&A is to have a place to answer obvious questions people might have, without distracting from the main narrative.



Q&A

How do Wolfram Language functions for machine learning work?
They use a range of state-of-the-art methods, and have meta-algorithms for picking between methods based on the particular problems they're given.

Doesn't LanguageIdentify just need a dictionary?
No. Specific words can occur in several languages. One needs to learn from examples how to deduce the language from combinations of words.

Can ImageIdentify get the wrong answer?
Yet. It can make mistakes, much like a human. This happens particularly if it's asked about an object that's in an unusual configuration or environment.

Can I ask ImageIdentify the probabilities it assigns to different identifications?
Yes. To find the probabilities for the top 10 identifications in all categories use **ImageIdentify[*image*, All, 10, "Probability"]**.

How many examples does Classify typically need to work well?
If the general area (like everyday images) is one it already knows well, then as few as a hundred. But in areas that are new, it can take many millions of examples to achieve good results.

How does Nearest figure out a distance between colors?
It uses the function **ColorDistance**, which is based on a model of human color vision.

How does Nearest determine nearby words?
By looking at those at the smallest **EditDistance**, that is, reached by the smallest number of single-letter insertions, deletions and substitutions.

Can a single graph have several disconnected parts?
Absolutely. An example is the last graph in this section.

Tech Notes

- **BarcodeImage** and **BarcodeRecognize** work with bar codes and QR codes instead of pure text.
- **ImageIdentify** is the core of what the imageidentify.com website does.
- **ImageIdentify** uses methods that are inspired by idealized models of neural networks in the brain.
- If you just give **Classify** training examples, it'll produce a **ClassifierFunction** that can later be applied to many different pieces of data. This is pretty much always how **Classify** is used in practice.
- **Classify** automatically picks between methods such as *logistic regression*, *naive Bayes*, *random forests*, *support vector machines* and *neural networks*.
- **FindClusters** does *unsupervised machine learning*, where the computer just looks at data without being told anything about it. **Classify** does *supervised machine learning*, being given a set of training examples.

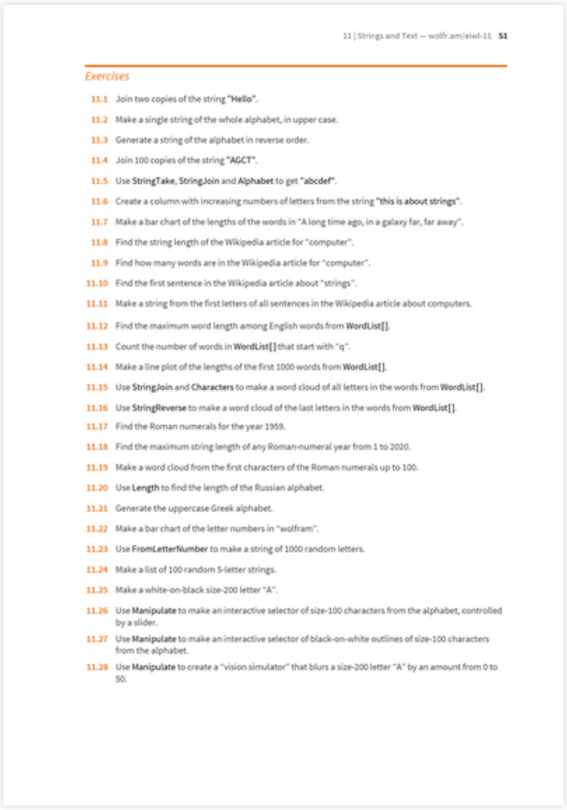
More to Explore

Guide to Machine Learning in the Wolfram Language (wolfr.am/eiwl-22-more)

There are several different types of questions. Some are about extensions to the functionality that's been discussed. Some are about the background to it. And some are questions ("What does 'raised to the power' mean?") that will be trivial to some readers but not to others.

In addition to Q&A, I found it useful to include what I call Tech Notes. Their goal is to add technical information—and to help people who already have sophisticated technical knowledge in some particular area to connect it to what they're reading in this book.

Exercises Another part of most sections is a collection of exercises. The vast majority are basically of the form "write a piece of code to do X"—though a few are instead "find a simpler version of this piece of code".



11 | Strings and Text — wolfr.am/eiwl-11 **51**

Exercises

11.1 Join two copies of the string **"Hello"**.

11.2 Make a single string of the whole alphabet, in upper case.

11.3 Generate a string of the alphabet in reverse order.

11.4 Join 100 copies of the string **"AGCT"**.

11.5 Use **StringTake**, **StringJoin** and **Alphabet** to get **"abcdef"**.

11.6 Create a column with increasing numbers of letters from the string **"this is about strings"**.

11.7 Make a bar chart of the lengths of the words in "A long time ago, in a galaxy far, far away".

11.8 Find the string length of the Wikipedia article for "computer".

11.9 Find how many words are in the Wikipedia article for "computer".

11.10 Find the first sentence in the Wikipedia article about "strings".

11.11 Make a string from the first letters of all sentences in the Wikipedia article about computers.

11.12 Find the maximum word length among English words from **WordList[]**.

11.13 Count the number of words in **WordList[]** that start with "q".

11.14 Make a line plot of the lengths of the first 1000 words from **WordList[]**.

11.15 Use **StringJoin** and **Characters** to make a word cloud of all letters in the words from **WordList[]**.

11.16 Use **StringReverse** to make a word cloud of the last letters in the words from **WordList[]**.

11.17 Find the Roman numerals for the year 1959.

11.18 Find the maximum string length of any Roman-numeral year from 1 to 2020.

11.19 Make a word cloud from the first characters of the Roman numerals up to 100.

11.20 Use **Length** to find the length of the Russian alphabet.

11.21 Generate the uppercase Greek alphabet.

11.22 Make a bar chart of the letter numbers in "wolfram".

11.23 Use **FromLetterNumber** to make a string of 1000 random letters.

11.24 Make a list of 100 random 5-letter strings.

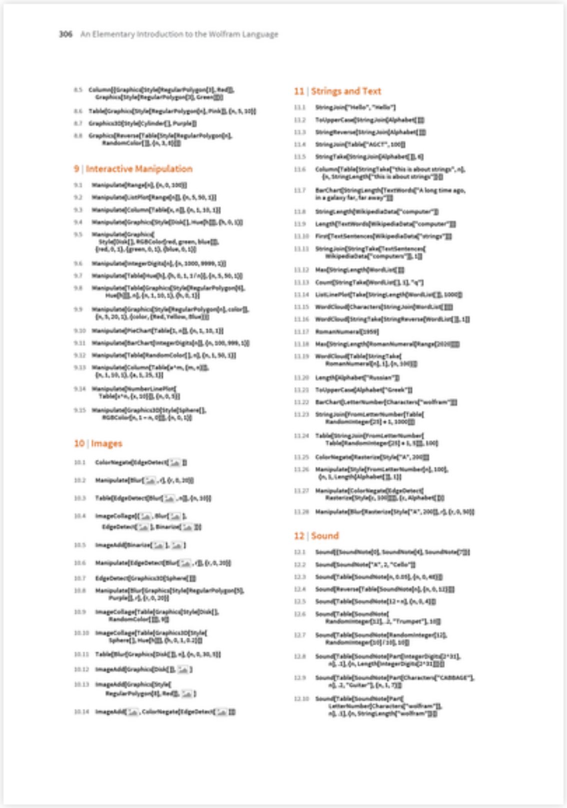
11.25 Make a white-on-black size-200 letter "A".

11.26 Use **Manipulate** to make an interactive selector of size-100 characters from the alphabet, controlled by a slider.

11.27 Use **Manipulate** to make an interactive selector of black-on-white outlines of size-100 characters from the alphabet.

11.28 Use **Manipulate** to create a "vision simulator" that blurs a size-200 letter "A" by an amount from 0 to 50.

There are answers to all the exercises in the printed book at the back—and in the web version there are additional exercises. Of course, the answers that are given are just possible answers—and they're almost never the only possible answers.



306 An Elementary Introduction to the Wolfram Language

```
8.5  Column[{Graphics[Style[RegularPolygon[3], Red]],
       Graphics[Style[RegularPolygon[3], Green]]}]
8.6  Table[Graphics[Style[RegularPolygon[n], Pink]], {n, 5, 10}]
8.7  Graphics3D[Style[Cylinder[], Purple]]
8.8  Graphics[Reverse[Table[Style[RegularPolygon[n],
         RandomColor[]], {n, 3, 8}]]]
```

9 | Interactive Manipulation

```
9.1  Manipulate[Range[n], {n, 0, 100}]
9.2  Manipulate[ListPlot[Range[n]], {n, 5, 50, 1}]
9.3  Manipulate[Column[Table[x, n]], {n, 1, 10, 1}]
9.4  Manipulate[Graphics[Style[Disk[], Hue[h]]], {h, 0, 1}]
9.5  Manipulate[Graphics[
       Style[Disk[], RGBColor[red, green, blue]]],
      {red, 0, 1}, {green, 0, 1}, {blue, 0, 1}]
9.6  Manipulate[IntegerDigits[n], {n, 1000, 9999, 1}]
9.7  Manipulate[Table[Hue[h], {h, 0, 1, 1/n}], {n, 5, 50, 1}]
9.8  Manipulate[Table[Graphics[Style[RegularPolygon[6],
         Hue[h]]], n], {n, 1, 10, 1}, {h, 0, 1}]
9.9  Manipulate[Graphics[Style[RegularPolygon[n], color]],
      {n, 5, 20, 1}, {color, {Red, Yellow, Blue}}]
9.10 Manipulate[PieChart[Table[1, n]], {n, 1, 10, 1}]
9.11 Manipulate[BarChart[IntegerDigits[n]], {n, 100, 999, 1}]
9.12 Manipulate[Table[RandomColor[], n], {n, 1, 50, 1}]
9.13 Manipulate[Column[Table[a^m, {m, n}]],
      {n, 1, 10, 1}, {a, 1, 25, 1}]
9.14 Manipulate[NumberLinePlot[
       Table[x^n, {x, 10}]], {n, 0, 5}]
9.15 Manipulate[Graphics3D[Style[Sphere[],
         RGBColor[n, 1 - n, 0]]], {n, 0, 1}]
```

10 | Images

```
10.1  ColorNegate[EdgeDetect[ ]]
10.2  Manipulate[Blur[ , r], {r, 0, 20}]
10.3  Table[EdgeDetect[Blur[ , n]], {n, 10}]
10.4  ImageCollage[{ , Blur[ ],
        EdgeDetect[ ], Binarize[ ]}]
10.5  ImageAdd[Binarize[ ], ]
10.6  Manipulate[EdgeDetect[Blur[ , r]], {r, 0, 20}]
10.7  EdgeDetect[Graphics3D[Sphere[]]]
10.8  Manipulate[Blur[Graphics[Style[RegularPolygon[5],
          Purple]], r], {r, 0, 20}]
10.9  ImageCollage[Table[Graphics[Style[Disk[],
          RandomColor[]]], 9]]
10.10 ImageCollage[Table[Graphics3D[Style[
          Sphere[], Hue[h]]], {h, 0, 1, 0.2}]]
10.11 Table[Blur[Graphics[Disk[]], n], {n, 0, 30, 5}]
10.12 ImageAdd[Graphics[Disk[]], ]
10.13 ImageAdd[Graphics[Style[
         RegularPolygon[3], Red]], ]
10.14 ImageAdd[ , ColorNegate[EdgeDetect[ ]]]
```

11 | Strings and Text

```
11.1  StringJoin["Hello", "Hello"]
11.2  ToUpperCase[StringJoin[Alphabet[]]]
11.3  StringReverse[StringJoin[Alphabet[]]]
11.4  StringJoin[Table["AGCT", 100]]
11.5  StringTake[StringJoin[Alphabet[]], 6]
11.6  Column[Table[StringTake["this is about strings", n],
       {n, StringLength["this is about strings"]}]]
11.7  BarChart[StringLength[TextWords["A long time ago,
       in a galaxy far, far away"]]]
11.8  StringLength[WikipediaData["computer"]]
11.9  Length[TextWords[WikipediaData["computer"]]]
11.10 First[TextSentences[WikipediaData["strings"]]]
11.11 StringJoin[StringTake[TextSentences[
         WikipediaData["computers"]], 1]]
11.12 Max[StringLength[WordList[]]]
11.13 Count[StringTake[WordList[], 1], "q"]
11.14 ListLinePlot[Take[StringLength[WordList[]], 1000]]
11.15 WordCloud[Characters[StringJoin[WordList[]]]]
11.16 WordCloud[StringTake[StringReverse[WordList[]], 1]]
11.17 RomanNumeral[1959]
11.18 Max[StringLength[RomanNumeral[Range[2020]]]]
11.19 WordCloud[Table[StringTake[
         RomanNumeral[n], 1], {n, 100}]]
11.20 Length[Alphabet["Russian"]]
11.21 ToUpperCase[Alphabet["Greek"]]
11.22 BarChart[LetterNumber[Characters["wolfram"]]]
11.23 StringJoin[FromLetterNumber[Table[
         RandomInteger[25] + 1, 1000]]]
11.24 Table[StringJoin[FromLetterNumber[
         Table[RandomInteger[25] + 1, 5]]], 100]
11.25 ColorNegate[Rasterize[Style["A", 200]]]
11.26 Manipulate[Style[FromLetterNumber[n], 100],
       {n, 1, Length[Alphabet[]], 1}]
11.27 Manipulate[ColorNegate[EdgeDetect[
         Rasterize[Style[c, 100]]]], {c, Alphabet[]}]
11.28 Manipulate[Blur[Rasterize[Style["A", 200]], r], {r, 0, 50}]
```

12 | Sound

```
12.1  Sound[{SoundNote[0], SoundNote[4], SoundNote[7]}]
12.2  Sound[SoundNote["A", 2, "Cello"]]
12.3  Sound[Table[SoundNote[n, 0.05], {n, 0, 48}]]
12.4  Sound[Reverse[Table[SoundNote[n], {n, 0, 12}]]]
12.5  Sound[Table[SoundNote[12 * n], {n, 0, 4}]]
12.6  Sound[Table[SoundNote[
         RandomInteger[12], .2, "Trumpet"], 10]]
12.7  Sound[Table[SoundNote[RandomInteger[12],
         RandomInteger[10] / 10], 10]]
12.8  Sound[Table[SoundNote[Part[IntegerDigits[2^31],
          n], .1], {n, Length[IntegerDigits[2^31]]}]]
12.9  Sound[Table[SoundNote[Part[Characters["CABBAGE"],
          n], .2, "Guitar"], {n, 1, 7}]]
12.10 Sound[Table[SoundNote[Part[
          LetterNumber[Characters["wolfram"]],
          n], .1], {n, StringLength["wolfram"]}]]
```

Writing the exercises was an interesting experience for me, that was actually quite important in my thinking about topics like how to talk to AIs. Because what most of the exercises effectively say is, "Take this description that's written in English, and turn it into Wolfram Language code." If what one's doing is simple enough, then English works quite well as a description language. But when what one's doing gets more complicated, English doesn't do so well. And by later in the book, I was often finding it much easier to write the Wolfram Language answer for an exercise than to create the actual exercise in English.

In a sense this is very satisfying, because it means we really need the Wolfram Language to be able to express ideas. Some things we can express easily in English—and eventually expect Wolfram|Alpha to be able to understand. But there's plenty that requires the greater structure and precision of the Wolfram Language.

A Book? At some level it might seem odd in this day and age to be writing a book that can actually be printed on paper, rather than creating some more flexible online structure. But what I've found is that the concept of a book is very useful. Yes, one can have a website where one can reach lots of information by following links. But when people are trying to systematically learn a subject, I think it's good to have a definite, finite container of information, where there's an expectation of digesting it sequentially, and where one can readily see the overall structure.

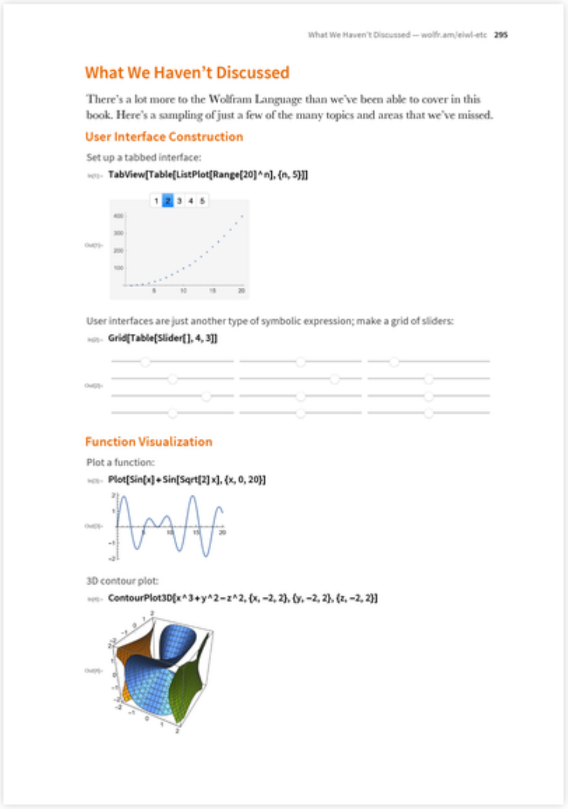
That's not to say that it's not useful to have the book online. Right now the book is available as a website, and for many purposes this web version works very well. But somewhat to my surprise, I still find the physical book, with its definite pagination and browsable pages, better for many things.

Of course, if you're going to learn the Wolfram Language, you actually need to run it. So even if you're using a physical book, it's best to to have a computer (or tablet) by your side—so you can try the examples, do the exercises, etc. You can do this immediately if you're reading the book on the web or in the cloud. But some people have told me that they actually find it helpful to retype the examples: they internalize them better that way, and with all the autocompletion and other features in the Wolfram Language, it's very fast to type in code.

What's Not in the Book I call the book an "elementary introduction". And that's what it is. It's not a complete book about the Wolfram Language—far from it. It's intended to be a basic introduction that gets people to the point where they can start writing useful programs. It covers a lot of the core principles of the language—but only a small fraction of the very large number of specific areas of functionality.

Generally I tried to include areas that are either very commonly encountered in practice, or easy for people to understand without external knowledge—and good for illuminating principles. I'm very happy with the sequence of areas I was able to cover—but another book could certainly pick quite different ones.

Of course, I was a little disappointed to have to leave out all sorts of amazing things that the Wolfram Language can do. And at the end of the book I decided to include a short section that gives a taste of what I wasn't able to talk about.



What We Haven't Discussed — wolfr.am/eiwl-etc 295

What We Haven't Discussed

There's a lot more to the Wolfram Language than we've been able to cover in this book. Here's a sampling of just a few of the many topics and areas that we've missed.

User Interface Construction

Set up a tabbed interface:

```
TabView[Table[ListPlot[Range[20]^n], {n, 5}]]
```

User interfaces are just another type of symbolic expression; make a grid of sliders:

```
Grid[Table[Slider[], 4, 3]]
```

Function Visualization

Plot a function:

```
Plot[Sin[x] + Sin[Sqrt[2] x], {x, 0, 20}]
```

3D contour plot:

```
ContourPlot3D[x^3 + y^2 - z^2, {x, -2, 2}, {y, -2, 2}, {z, -2, 2}]
```

Some Backstory I see my new book as part of the effort to launch the Wolfram Language. And back in 1988, when we first launched Mathematica, I wrote a book for that, too. But it was a different kind of book: it was a book that was intended to provide a complete tutorial introduction and reference guide to the whole system. The first edition was 767 pages. But by the 5th edition a decade later, the book had grown to 1488 pages. And at that point we decided a book just wasn't the correct way to deliver the information—and we built a whole online system instead.

It's just as well we did that, because it allowed us to greatly expand the depth of coverage, particularly in terms of examples. And of course the actual software system grew a lot—with the result that today the full Documentation Center contains more than 50,000 pages of content.

Many people have told me that they liked the original Mathematica book—and particularly the fact that it was short enough to realistically read from cover to cover. My goal with *An Elementary Introduction to the Wolfram Language* was again to have a book that's short enough that people can actually read all of it.

Looking at the book, it's interesting to see how much of it is about things that simply didn't exist in the Wolfram Language—or Mathematica—until very recently. Of course it's satisfying to me to see that things we're adding now are important enough to make it into an elementary introduction. But it also means that even people who've known parts of the Wolfram Language through Mathematica for many years should find the book interesting to read.

Why Me? I've thought for a while that there should be a book like the one I've now written. And obviously there are plenty of people who know the Wolfram Language well and could in principle have written an introduction to it. But I'm happy that I've been the one to write this book. It's reduced my productivity on other things—like writing blogs—for a little while. But it's been a fascinating experience.

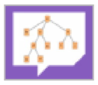
It's been a bit like being back hundreds of years and asking, "How should one approach explaining math to people?" And working out that first one should talk about arithmetic, then algebra, and so on. Well, now we have to do the same kind of thing for computational thinking. And I see the book as a first effort at communicating the tools of computational thinking to a broad range of people.

It's been fun to write. I hope people find it fun to read—and that they use what they learn from it to create amazing things with the Wolfram Language.

Posted in: Education, Wolfram Language
13 Comments More

RELATED POSTS

Roaring into 2018 with Another Big Release: Launching Version 11.3 of the Wolfram Language & Mathematica
March 8, 2018



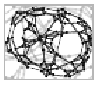
What Do I Do All Day? Livestreamed Technology CEOing
December 11, 2017

What Is a Computational Essay?
November 14, 2017

NEXT POST

Untangling the Tale of Ada Lovelace
December 10, 2015

PREVIOUS POST



What Is Spacetime, Really?
December 2, 2015

Join the discussion

13 comments. Show all »