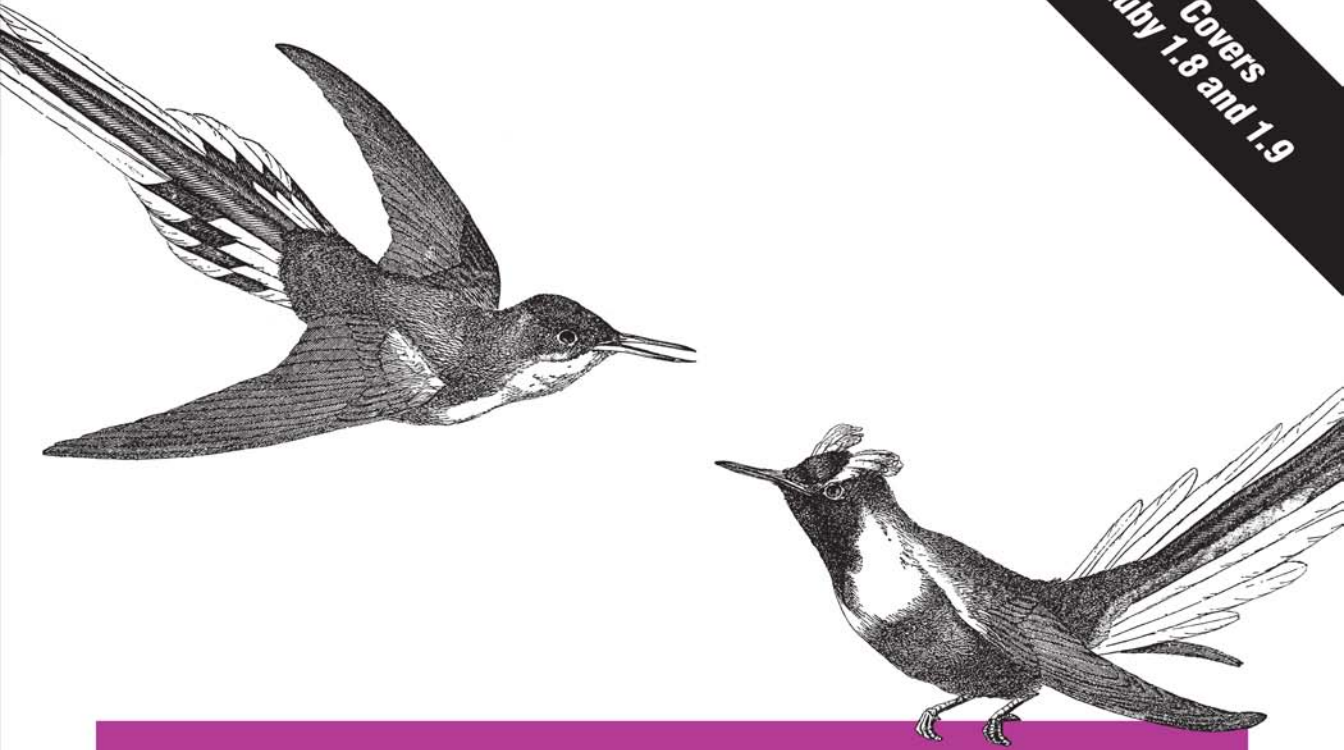


Everything You Need to Know

**Covers
Ruby 1.8 and 1.9**



The Ruby Programming Language

O'REILLY®

*David Flanagan & Yukihiro Matsumoto
with drawings by why the lucky stiff*

The Ruby Programming Language



The Ruby Programming Language is *the* authoritative guide to Ruby, with comprehensive coverage of versions 1.8 and 1.9 of the language. Written for experienced programmers new to Ruby, and for current Ruby programmers who want to challenge their understanding and increase their mastery of the language, this book documents Ruby definitively but without the formality of a language specification.

This guide begins with a quick-start tutorial to the language, and then explains the language in detail from the bottom up, covering:

- Lexical and syntactic structure of Ruby programs
- Datatypes and objects
- Expressions and operators
- Statements and control structures
- Methods, procs, lambdas, and closures
- Classes and modules
- Reflection and metaprogramming
- The Ruby platform

The Ruby Programming Language includes a long and thorough introduction to the rich API of the Ruby platform, demonstrating—with heavily commented example code—Ruby's facilities for text processing, numeric manipulation, collections, input/output, networking, and concurrency.

If you really want to understand Ruby, this is the one book you need to have.

David Flanagan is a programmer and author whose bestselling books *Java in a Nutshell* and *JavaScript: The Definitive Guide* (both O'Reilly) are considered the standard references for Java and JavaScript.

Yukihiro "Matz" Matsumoto is the creator, designer, and lead developer of Ruby.

"Flanagan has assembled a comprehensive reference manual for Ruby, spelunking its depths and then putting it all together in a treasure map that everyone will want to use."

—Evan Phoenix, creator of the Rubinius implementation of Ruby

www.oreilly.com

US \$39.99

CAN \$39.99

ISBN-10: 0-596-51617-7

ISBN-13: 978-0-596-51617-8



Safari®
Books Online

Free online edition
with purchase of this book.
Details on last page.

The Ruby Programming Language

The Ruby Programming Language

David Flanagan and Yukihiro Matsumoto

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

The Ruby Programming Language

by David Flanagan and Yukihiro Matsumoto
with drawings by *why the lucky stiff*

Copyright © 2008 David Flanagan and Yukihiro Matsumoto. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Mike Loukides
Production Editor: Sarah Schneider
Proofreader: Sarah Schneider

Indexer: Joe Wizda
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Rob Romano and *why the lucky stiff*

Printing History:

January 2008: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *The Ruby Programming Language*, the image of Horned Sungem hummingbirds, and related trade dress are trademarks of O'Reilly Media, Inc.

Java™ and all Java-based trademarks are registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly Media, Inc. is independent of Sun Microsystems.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. The drawings on the chapter title pages were drawn by *why the lucky stiff* and are licensed under the Creative Commons Attribution-ShareAlike 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/legalcode> or send a letter to Creative Commons, 171 2nd Street, Suite 300, San Francisco, California, 94105, USA.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-13: 978-0-596-51617-8

[M]

[12/08]

1264438633

Table of Contents

Preface	ix
1. Introduction	1
1.1 A Tour of Ruby	2
1.2 Try Ruby	11
1.3 About This Book	15
1.4 A Sudoku Solver in Ruby	17
2. The Structure and Execution of Ruby Programs	25
2.1 Lexical Structure	26
2.2 Syntactic Structure	33
2.3 File Structure	35
2.4 Program Encoding	36
2.5 Program Execution	39
3. Datatypes and Objects	41
3.1 Numbers	42
3.2 Text	46
3.3 Arrays	64
3.4 Hashes	67
3.5 Ranges	68
3.6 Symbols	70
3.7 True, False, and Nil	72
3.8 Objects	72
4. Expressions and Operators	85
4.1 Literals and Keyword Literals	86
4.2 Variable References	87
4.3 Constant References	88
4.4 Method Invocations	89
4.5 Assignments	92
4.6 Operators	100

5.	Statements and Control Structures	117
5.1	Conditionals	118
5.2	Loops	127
5.3	Iterators and Enumerable Objects	130
5.4	Blocks	140
5.5	Altering Control Flow	146
5.6	Exceptions and Exception Handling	154
5.7	BEGIN and END	165
5.8	Threads, Fibers, and Continuations	166
6.	Methods, Procs, Lambdas, and Closures	175
6.1	Defining Simple Methods	177
6.2	Method Names	180
6.3	Methods and Parentheses	183
6.4	Method Arguments	185
6.5	Procs and Lambdas	192
6.6	Closures	200
6.7	Method Objects	203
6.8	Functional Programming	205
7.	Classes and Modules	213
7.1	Defining a Simple Class	214
7.2	Method Visibility: Public, Protected, Private	232
7.3	Subclassing and Inheritance	234
7.4	Object Creation and Initialization	241
7.5	Modules	247
7.6	Loading and Requiring Modules	252
7.7	Singleton Methods and the Eigenclass	257
7.8	Method Lookup	258
7.9	Constant Lookup	261
8.	Reflection and Metaprogramming	265
8.1	Types, Classes, and Modules	266
8.2	Evaluating Strings and Blocks	268
8.3	Variables and Constants	271
8.4	Methods	272
8.5	Hooks	277
8.6	Tracing	279
8.7	ObjectSpace and GC	281
8.8	Custom Control Structures	281
8.9	Missing Methods and Missing Constants	284
8.10	Dynamically Creating Methods	287
8.11	Alias Chaining	290

8.12 Domain-Specific Languages	296
9. The Ruby Platform	303
9.1 Strings	304
9.2 Regular Expressions	310
9.3 Numbers and Math	321
9.4 Dates and Times	325
9.5 Collections	328
9.6 Files and Directories	350
9.7 Input/Output	356
9.8 Networking	366
9.9 Threads and Concurrency	373
10. The Ruby Environment	389
10.1 Invoking the Ruby Interpreter	390
10.2 The Top-Level Environment	395
10.3 Practical Extraction and Reporting Shortcuts	403
10.4 Calling the OS	405
10.5 Security	409
Index	415

Preface

This book is an updated and expanded version of *Ruby in a Nutshell* (O'Reilly) by Yukihiro Matsumoto, who is better known as Matz. It is loosely modeled after the classic *The C Programming Language* (Prentice Hall) by Brian Kernighan and Dennis Ritchie, and aims to document the Ruby language comprehensively but without the formality of a language specification. It is written for experienced programmers who are new to Ruby, and for current Ruby programmers who want to take their understanding and mastery of the language to the next level.

You'll find a guide to the structure and organization of this book in Chapter 1.

Acknowledgments

David Flanagan

Before anything else, I must thank Matz for the beautiful language he has designed, for his help understanding that language, and for the *Nutshell* that this book grew out of.

Thanks also to:

- *why the lucky stiff* for the delightful drawings that grace these pages (you'll find them on the chapter title pages) and, of course, for his own book on Ruby, *why's (poignant) guide to Ruby*, which you can find online at <http://poignantguide.net/ruby/>.
- My technical reviewers: David A. Black, director of Ruby Power and Light, LLC (<http://www.rubypal.com>); Charles Oliver Nutter of the JRuby team (<http://www.jruby.org>) at Sun Microsystems; Shyouhei Urabe, the maintainer of the Ruby 1.8.6 branch; and Ken Cooper. Their comments helped improve the quality and clarity of the book. Any errors that remain are, of course, my own.
- My editor, Mike Loukides, for asking and persistently encouraging me to write this book, and for his patience while I did so.

Finally, of course, my love and thanks to my family.

—David Flanagan

<http://www.davidflanagan.com>

January 2008

Yukihiro Matsumoto

In addition to the people listed by David (except myself), I appreciate the help from community members all around the world, especially from Japan: Koichi Sasada, Nobuyoshi Nakada, Akira Tanaka, Shugo Maeda, Usaku Nakamura, and Shyouhei Urabe to name a few (not in any particular order).

And finally, I thank my family, who hopefully forgive their husband and father for dedicating time to Ruby development.

—Yukihiro Matsumoto

January 2008

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, datatypes, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example

code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*The Ruby Programming Language* by David Flanagan and Yukihiro Matsumoto. Copyright 2008 David Flanagan and Yukihiro Matsumoto, 978-0-596-51617-8.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

<http://www.oreilly.com/catalog/9780596516178>


To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

<http://www.oreilly.com>

Safari® Enabled

 When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at <http://safari.oreilly.com>.

Introduction



Ruby is a dynamic programming language with a complex but expressive grammar and a core class library with a rich and powerful API. Ruby draws inspiration from Lisp, Smalltalk, and Perl, but uses a grammar that is easy for C and Java™ programmers to learn. Ruby is a pure object-oriented language, but it is also suitable for procedural and functional programming styles. It includes powerful metaprogramming capabilities and can be used to create domain-specific languages or DSLs.

Matz on Ruby

Yukihiro Matsumoto, known as Matz to the English-speaking Ruby community, is the creator of Ruby and the author of *Ruby in a Nutshell* (O'Reilly) (which has been updated and expanded into the present book). He says:

I knew many languages before I created Ruby, but I was never fully satisfied with them. They were uglier, tougher, more complex, or more simple than I expected. I wanted to create my own language that satisfied me, as a programmer. I knew a lot about the language's target audience: myself. To my surprise, many programmers all over the world feel very much like I do. They feel happy when they discover and program in Ruby.

Throughout the development of the Ruby language, I've focused my energies on making programming faster and easier. All features in Ruby, including object-oriented features, are designed to work as ordinary programmers (e.g., me) expect them to work. Most programmers feel it is elegant, easy to use, and a pleasure to program.

Matz's guiding philosophy for the design of Ruby is summarized in an oft-quoted remark of his:

Ruby is designed to make programmers happy.

1.1 A Tour of Ruby

This section is a guided, but meandering, tour through some of the most interesting features of Ruby. Everything discussed here will be documented in detail later in the book, but this first look will give you the flavor of the language.

1.1.1 Ruby Is Object-Oriented

We'll begin with the fact that Ruby is a *completely* object-oriented language. Every value is an object, even simple numeric literals and the values `true`, `false`, and `nil` (`nil` is a special value that indicates the absence of value; it is Ruby's version of `null`). Here we invoke a method named `class` on these values. Comments begin with `#` in Ruby, and the `=>` arrows in the comments indicate the value returned by the commented code (this is a convention used throughout this book):

```
1.class # => Fixnum: the number 1 is a Fixnum
0.0.class # => Float: floating-point numbers have class Float
```



```

true.class # => TrueClass: true is a the singleton instance of TrueClass
false.class # => FalseClass
nil.class # => NilClass

```

In many languages, function and method invocations require parentheses, but there are no parentheses in any of the code above. In Ruby, parentheses are usually optional and they are commonly omitted, especially when the method being invoked takes no arguments. The fact that the parentheses are omitted in the method invocations here makes them look like references to named fields or named variables of the object. This is intentional, but the fact is, Ruby is very strict about encapsulation of its objects; there is no access to the internal state of an object from outside the object. Any such access must be mediated by an accessor method, such as the `class` method shown above.

1.1.2 Blocks and Iterators

The fact that we can invoke methods on integers isn't just an esoteric aspect of Ruby. It is actually something that Ruby programmers do with some frequency:

```

3.times { print "Ruby! " } # Prints "Ruby! Ruby! Ruby! "
1.upto(9) {|x| print x } # Prints "123456789"

```

`times` and `upto` are methods implemented by integer objects. They are a special kind of method known as an *iterator*, and they behave like loops. The code within curly braces—known as a *block*—is associated with the method invocation and serves as the body of the loop. The use of iterators and blocks is another notable feature of Ruby; although the language does support an ordinary `while` loop, it is more common to perform loops with constructs that are actually method calls.

Integers are not the only values that have iterator methods. Arrays (and similar “enumerable” objects) define an iterator named `each`, which invokes the associated block once for each element in the array. Each invocation of the block is passed a single element from the array:

```

a = [3, 2, 1] # This is an array literal
a[3] = a[2] - 1 # Use square brackets to query and set array elements
a.each do |elt| # each is an iterator. The block has a parameter elt
  print elt+1 # Prints "4321"
end # This block was delimited with do/end instead of {}

```

Various other useful iterators are defined on top of each:

```

a = [1,2,3,4] # Start with an array
b = a.map {|x| x*x } # Square elements: b is [1,4,9,16]
c = a.select {|x| x%2==0 } # Select even elements: c is [2,4]
a.inject do |sum,x| # Compute the sum of the elements => 10
  sum + x
end

```

Hashes, like arrays, are a fundamental data structure in Ruby. As their name implies, they are based on the hashtable data structure and serve to map arbitrary key objects to value objects. (To put this another way, we can say that a hash associates arbitrary

value objects with key objects.) Hashes use square brackets, like arrays do, to query and set values in the hash. Instead of using an integer index, they expect key objects within the square brackets. Like the `Array` class, the `Hash` class also defines an `each` iterator method. This method invokes the associated block of code once for each key/value pair in the hash, and (this is where it differs from `Array`) passes both the key and the value as parameters to the block:

```
h = {
  :one => 1,      # A hash that maps number names to digits
  :two => 2      # The "arrows" show mappings: key=>value
}               # The colons indicate Symbol literals
h[:one]         # => 1. Access a value by key
h[:three] = 3   # Add a new key/value pair to the hash
h.each do |key,value|
  print "#{value}:#{key}; " # Iterate through the key/value pairs
end             # Note variables substituted into string
               # Prints "1:one; 2:two; 3:three; "
```

Ruby's hashes can use any object as a key, but `Symbol` objects are the most commonly used. Symbols are immutable, interned strings. They can be compared by identity rather than by textual content (because two distinct `Symbol` objects will never have the same content).

The ability to associate a block of code with a method invocation is a fundamental and very powerful feature of Ruby. Although its most obvious use is for loop-like constructs, it is also useful for methods that only invoke the block once. For example:

```
File.open("data.txt") do |f| # Open named file and pass stream to block
  line = f.readline         # Use the stream to read from the file
end                         # Stream automatically closed at block end

t = Thread.new do          # Run this block in a new thread
  File.read("data.txt")    # Read a file in the background
end                        # File contents available as thread value
```

As an aside, notice that the `Hash.each` example previously included this interesting line of code:

```
print "#{value}:#{key}; " # Note variables substituted into string
```

Double-quoted strings can include arbitrary Ruby expressions delimited by `#{` and `}`. The value of the expression within these delimiters is converted to a string (by calling its `to_s` method, which is supported by all objects). The resulting string is then used to replace the expression text and its delimiters in the string literal. This substitution of expression values into strings is usually called *string interpolation*.

1.1.3 Expressions and Operators in Ruby

Ruby's syntax is expression-oriented. Control structures such as `if` that would be called statements in other languages are actually expressions in Ruby. They have values like other simpler expressions do, and we can write code like this:

```
minimum = if x < y then x else y end
```

Although all “statements” in Ruby are actually expressions, they do not all return meaningful values. `while` loops and method definitions, for example, are expressions that normally return the value `nil`.

As in most languages, expressions in Ruby are usually built out of values and operators. For the most part, Ruby’s operators will be familiar to anyone who knows C, Java, JavaScript, or any similar programming language. Here are examples of some commonplace and some more unusual Ruby operators:

```
1 + 2           # => 3: addition
1 * 2           # => 2: multiplication
1 + 2 == 3      # => true: == tests equality
2 ** 1024       # 2 to the power 1024: Ruby has arbitrary size ints
"Ruby" + " rocks!" # => "Ruby rocks!": string concatenation
"Ruby! " * 3     # => "Ruby! Ruby! Ruby! ": string repetition
"%d %s" % [3, "rubies"] # => "3 rubies": Python-style, printf formatting
max = x > y ? x : y # The conditional operator
```

Many of Ruby’s operators are implemented as methods, and classes can define (or redefine) these methods however they want. (They can’t define completely new operators, however; there is only a fixed set of recognized operators.) As examples, notice that the `+` and `*` operators behave differently for integers and strings. And you can define these operators any way you want in your own classes. The `<<` operator is another good example. The integer classes `Fixnum` and `Bignum` use this operator for the bitwise left-shift operation, following the C programming language. At the same time (following C++), other classes—such as strings, arrays, and streams—use this operator for an append operation. If you create a new class that can have values appended to it in some way, it is a very good idea to define `<<`.

One of the most powerful operators to override is `[]`. The `Array` and `Hash` classes use this operator to access array elements by index and hash values by key. But you can define `[]` in your classes for any purpose you want. You can even define it as a method that expects multiple arguments, comma-separated between the square brackets. (The `Array` class accepts an index and a length between the square brackets to indicate a subarray or “slice” of the array.) And if you want to allow square brackets to be used on the lefthand side of an assignment expression, you can define the corresponding `[]=` operator. The value on the righthand side of the assignment will be passed as the final argument to the method that implements this operator.

1.1.4 Methods

Methods are defined with the `def` keyword. The return value of a method is the value of the last expression evaluated in its body:

```
def square(x) # Define a method named square with one parameter x
  x*x        # Return x squared
end          # End of the method
```

When a method, like the one above, is defined outside of a class or a module, it is effectively a global function rather than a method to be invoked on an object. (Technically, however, a method like this becomes a private method of the `Object` class.) Methods can also be defined on individual objects by prefixing the name of the method with the object on which it is defined. Methods like these are known as *singleton methods*, and they are how Ruby defines class methods:

```
def Math.square(x) # Define a class method of the Math module
  x*x
end
```

The `Math` module is part of the core Ruby library, and this code adds a new method to it. This is a key feature of Ruby—classes and modules are “open” and can be modified and extended at runtime.

Method parameters may have default values specified, and methods may accept arbitrary numbers of arguments.

1.1.5 Assignment

The (nonoverridable) `=` operator in Ruby assigns a value to a variable:

```
x = 1
```

Assignment can be combined with other operators such as `+` and `-`:

```
x += 1      # Increment x: note Ruby does not have ++.
y -= 1      # Decrement y: no -- operator, either.
```

Ruby supports parallel assignment, allowing more than one value and more than one variable in assignment expressions:

```
x, y = 1, 2 # Same as x = 1; y = 2
a, b = b, a # Swap the value of two variables
x,y,z = [1,2,3] # Array elements automatically assigned to variables
```

Methods in Ruby are allowed to return more than one value, and parallel assignment is helpful in conjunction with such methods. For example:

```
# Define a method to convert Cartesian (x,y) coordinates to Polar
def polar(x,y)
  theta = Math.atan2(y,x) # Compute the angle
  r = Math.hypot(x,y)     # Compute the distance
  [r, theta]              # The last expression is the return value
end

# Here's how we use this method with parallel assignment
distance, angle = polar(2,2)
```

Methods that end with an equals sign (`=`) are special because Ruby allows them to be invoked using assignment syntax. If an object `o` has a method named `x=`, then the following two lines of code do the very same thing:

```
o.x=(1)      # Normal method invocation syntax
o.x = 1      # Method invocation through assignment
```

1.1.6 Punctuation Suffixes and Prefixes

We saw previously that methods whose names end with `=` can be invoked by assignment expressions. Ruby methods can also end with a question mark or an exclamation point. A question mark is used to mark predicates—methods that return a Boolean value. For example, the `Array` and `Hash` classes both define methods named `empty?` that test whether the data structure has any elements. An exclamation mark at the end of a method name is used to indicate that caution is required with the use of the method. A number of core Ruby classes define pairs of methods with the same name, except that one ends with an exclamation mark and one does not. Usually, the method without the exclamation mark returns a modified copy of the object it is invoked on, and the one with the exclamation mark is a mutator method that alters the object in place. The `Array` class, for example, defines methods `sort` and `sort!`.

In addition to these punctuation characters at the end of method names, you'll notice punctuation characters at the start of Ruby variable names: global variables are prefixed with `$`, instance variables are prefixed with `@`, and class variables are prefixed with `@@`. These prefixes can take a little getting used to, but after a while you may come to appreciate the fact that the prefix tells you the scope of the variable. The prefixes are required in order to disambiguate Ruby's very flexible grammar. One way to think of variable prefixes is that they are one price we pay for being able to omit parentheses around method invocations.

1.1.7 Regexp and Range

We mentioned arrays and hashes earlier as fundamental data structures in Ruby. We demonstrated the use of numbers and strings as well. Two other datatypes are worth mentioning here. A `Regexp` (regular expression) object describes a textual pattern and has methods for determining whether a given string matches that pattern or not. And a `Range` represents the values (usually integers) between two endpoints. Regular expressions and ranges have a literal syntax in Ruby:

```
/[Rr]uby/    # Matches "Ruby" or "ruby"
/\d{5}/      # Matches 5 consecutive digits
1..3         # All x where 1 <= x <= 3
1...3        # All x where 1 <= x < 3
```

`Regexp` and `Range` objects define the normal `==` operator for testing equality. In addition, they also define the `===` operator for testing matching and membership. Ruby's `case` statement (like the `switch` statement of C or Java) matches its expression against each of the possible cases using `===`, so this operator is often called the *case equality operator*. It leads to conditional tests like these:

```

# Determine US generation name based on birth year
# Case expression tests ranges with ===
generation = case birthyear
  when 1946..1963: "Baby Boomer"
  when 1964..1976: "Generation X"
  when 1978..2000: "Generation Y"
  else nil
end

# A method to ask the user to confirm something
def are_you_sure? # Define a method. Note question mark!
  while true # Loop until we explicitly return
    print "Are you sure? [y/n]: " # Ask the user a question
    response = gets # Get her answer
    case response # Begin case conditional
    when /^[yY]/ # If response begins with y or Y
      return true # Return true from the method
    when /^[nN]/, /^$/ # If response begins with n,N or is empty
      return false # Return false
    end
  end
end
end

```

1.1.8 Classes and Modules

A class is a collection of related methods that operate on the state of an object. An object's state is held by its instance variables: variables whose names begin with @ and whose values are specific to that particular object. The following code defines an example class named `Sequence` and demonstrates how to write iterator methods and define operators:

```

#
# This class represents a sequence of numbers characterized by the three
# parameters from, to, and by. The numbers x in the sequence obey the
# following two constraints:
#
#   from <= x <= to
#   x = from + n*by, where n is an integer
#
class Sequence
  # This is an enumerable class; it defines an each iterator below.
  include Enumerable # Include the methods of this module in this class

  # The initialize method is special; it is automatically invoked to
  # initialize newly created instances of the class
  def initialize(from, to, by)
    # Just save our parameters into instance variables for later use
    @from, @to, @by = from, to, by # Note parallel assignment and @ prefix
  end

  # This is the iterator required by the Enumerable module
  def each
    x = @from # Start at the starting point
    while x <= @to # While we haven't reached the end

```

```

        yield x      # Pass x to the block associated with the iterator
        x += @by    # Increment x
    end
end

# Define the length method (following arrays) to return the number of
# values in the sequence
def length
    return 0 if @from > @to      # Note if used as a statement modifier
    Integer((@to-@from)/@by) + 1 # Compute and return length of sequence
end

# Define another name for the same method.
# It is common for methods to have multiple names in Ruby
alias size length # size is now a synonym for length

# Override the array-access operator to give random access to the sequence
def [](index)
    return nil if index < 0 # Return nil for negative indexes
    v = @from + index*@by   # Compute the value
    if v <= @to             # If it is part of the sequence
        v                   # Return it
    else                    # Otherwise...
        nil                 # Return nil
    end
end

# Override arithmetic operators to return new Sequence objects
def *(factor)
    Sequence.new(@from*factor, @to*factor, @by*factor)
end

def +(offset)
    Sequence.new(@from+offset, @to+offset, @by)
end
end

```

Here is some code that uses this Sequence class:

```

s = Sequence.new(1, 10, 2) # From 1 to 10 by 2's
s.each {|x| print x }     # Prints "13579"
print s[s.size-1]        # Prints 9
t = (s+1)*2              # From 4 to 22 by 4's

```

The key feature of our Sequence class is its each iterator. If we are only interested in the iterator method, there is no need to define the whole class. Instead, we can simply write an iterator method that accepts the `from`, `to`, and `by` parameters. Instead of making this a global function, let's define it in a module of its own:

```

module Sequences          # Begin a new module
    def self.fromtooby(from, to, by) # A singleton method of the module
        x = from
        while x <= to
            yield x
            x += by
        end
    end
end

```

```
end
end
```

With the iterator defined this way, we write code like this:

```
Sequences.fromtoby(1, 10, 2) {|x| print x } # Prints "13579"
```

An iterator like this makes it unnecessary to create a `Sequence` object to iterate a sequence of numbers. But the name of the method is quite long, and its invocation syntax is unsatisfying. What we really want is a way to iterate numeric `Range` objects by steps other than 1. One of the amazing features of Ruby is that its classes, even the built-in core classes, are *open*: any program can add methods to them. So we really can define a new iterator method for ranges:

```
class Range
  def by(step)
    x = self.begin
    if exclude_end?
      while x < self.end
        yield x
        x += step
      end
    else
      while x <= self.end
        yield x
        x += step
      end
    end
  end
end

# Examples
(0..10).by(2) {|x| print x} # Prints "0246810"
(0...10).by(2) {|x| print x} # Prints "02468"
```

This `by` method is convenient but unnecessary; the `Range` class already defines an iterator named `step` that serves the same purpose. The core Ruby API is a rich one, and it is worth taking the time to study the platform (see Chapter 9) so you don't end up spending time writing methods that have already been implemented for you!

1.1.9 Ruby Surprises

Every language has features that trip up programmers who are new to the language. Here we describe two of Ruby's surprising features.

Ruby's strings are mutable, which may be surprising to Java programmers in particular. The `[]=` operator allows you to alter the characters of a string or to insert, delete, and replace substrings. The `<<` operator allows you to append to a string, and the `String` class defines various other methods that alter strings in place. Because strings are mutable, string literals in a program are not unique objects. If you include a string literal within a loop, it evaluates to a new object on each iteration of the loop. Call the

`freeze` method on a string (or on any object) to prevent any future modifications to that object.

Ruby's conditionals and loops (such as `if` and `while`) evaluate conditional expressions to determine which branch to evaluate or whether to continue looping. Conditional expressions often evaluate to `true` or `false`, but this is not required. The value of `nil` is treated the same as `false`, and *any other value is the same as true*. This is likely to surprise C programmers who expect `0` to work like `false`, and JavaScript programmers who expect the empty string `""` to be the same as `false`.

1.2 Try Ruby

We hope our tour of Ruby's key features has piqued your interest and you are eager to try Ruby out. To do that, you'll need a Ruby interpreter, and you'll also want to know how to use three tools—*irb*, *ri*, and *gem*—that are bundled with the interpreter. This section explains how to get and use them.

1.2.1 The Ruby Interpreter

The official web site for Ruby is <http://www.ruby-lang.org>. If Ruby is not already installed on your computer, you can follow the download link on the [ruby-lang.org](http://www.ruby-lang.org) (<http://ruby-lang.org>) home page for instructions on downloading and installing the standard C-based reference implementation of Ruby.

Once you have Ruby installed, you can invoke the Ruby interpreter with the `ruby` command:

```
% ruby -e 'puts "hello world!'"  
hello world!
```

The `-e` command-line option causes the interpreter to execute a single specified line of Ruby code. More commonly, you'd place your Ruby program in a file and tell the interpreter to invoke it:

```
% ruby hello.rb  
hello world!
```

Other Ruby Implementations

In the absence of a formal specification for the Ruby language, the Ruby interpreter from [ruby-lang.org](http://www.ruby-lang.org) (<http://ruby-lang.org>) is the reference implementation that defines the language. It is sometimes known as MRI, or “Matz's Ruby Implementation.” For Ruby 1.9, the original MRI interpreter was merged with YARV (“Yet Another Ruby Virtual machine”) to produce a new reference implementation that performs internal compilation to bytecode and then executes that bytecode on a virtual machine.

The reference implementation is not the only one available, however. At the time of this writing, there is one alternative implementation (JRuby) released and several other implementations under development:

JRuby

JRuby is a Java-based implementation of Ruby, available from <http://jruby.org>. At the time of this writing, the current release is JRuby 1.1, which is compatible with Ruby 1.8. A 1.9-compatible release of JRuby may be available by the time you read this. JRuby is open source software, developed primarily at Sun Microsystems.

IronRuby

IronRuby is Microsoft's implementation of Ruby for their .NET framework and DLR (Dynamic Language Runtime). The source code for IronRuby is available under the Microsoft Permissive License. At the time of this writing, IronRuby is not yet at a 1.0 release level. The project home page is <http://www.ironruby.net>.

Rubinius

Rubinius is an open source project that describes itself as “an alternative Ruby implementation written largely in Ruby. The Rubinius virtual machine, named shotgun, is based loosely on the Smalltalk-80 VM architecture.” At the time of this writing, Rubinius is not at version 1.0. The home page for the Rubinius project is <http://rubini.us>.

Cardinal

Cardinal is a Ruby implementation intended to run on the Parrot VM (which aims to power Perl 6 and a number of other dynamic languages). At the time of this writing, neither Parrot nor Cardinal have released a 1.0 version. Cardinal does not have its own home page; it is hosted as part of the open source Parrot project at <http://www.parrotcode.org>.

1.2.2 Displaying Output

In order to try out Ruby features, you need a way to display output so that your test programs can print their results. The `puts` function—used in the “hello world” code earlier—is one way to do this. Loosely speaking, `puts` prints a string of text to the console and appends a newline (unless the string already ends with one). If passed an object that is not a string, `puts` calls the `to_s` method of that object and prints the string returned by that method. `print` does more or less the same thing, but it does not append a newline. For example, type the following two-line program in a text editor and save it in a file named `count.rb`:

```
9.downto(1) {|n| print n } # No newline between numbers
puts " blastoff!"         # End with a newline
```

Now run the program with your Ruby interpreter:

```
% ruby count.rb
```

It should produce the following output:

```
987654321 blastoff!
```

You may find the function `p` to be a useful alternative to `puts`. Not only is it shorter to type, but it converts objects to strings with the `inspect` method, which sometimes

returns more programmer-friendly representations than `to_s` does. When printing an array, for example, `p` outputs it using array literal notation, whereas `puts` simply prints each element of the array on a line by itself.

1.2.3 Interactive Ruby with `irb`

`irb` (short for “interactive Ruby”) is a Ruby shell. Type any Ruby expression at its prompt and it will evaluate it and display its value for you. This is often the easiest way to try out the language features you read about in this book. Here is an example `irb` session, with annotations:

```
$ irb --simple-prompt      # Start irb from the terminal
>> 2**3                  # Try exponentiation
=> 8                      # This is the result
>> "Ruby! " * 3          # Try string repetition
=> "Ruby! Ruby! Ruby! "  # The result
>> 1.upto(3){|x| puts x } # Try an iterator
1                        # Three lines of output
2                        # Because we called puts 3 times
3
=> 1                      # The return value of 1.upto(3)
>> quit                  # Exit irb
$                        # Back to the terminal prompt
```

This example session shows you all you need to know about `irb` to make productive use of it while exploring Ruby. It does have a number of other important features, however, including subshells (type “`irb`” at the prompt to start a subshell) and configurability.

1.2.4 Viewing Ruby Documentation with `ri`

Another critical Ruby tool is the `ri`* documentation viewer. Invoke `ri` on the command line followed by the name of a Ruby class, module, or method, and `ri` will display documentation for you. You may specify a method name without a qualifying class or module name, but this will just show you a list of all methods by that name (unless the method is unique). Normally, you can separate a class or module name from a method name with a period. If a class defines a class method and an instance method by the same name, you must instead use `::` to refer to the class method or `#` to refer to the instance method. Here are some example invocations of `ri`:

```
ri Array
ri Array.sort
ri Hash#each
ri Math::sqrt
```

* Opinions differ as to what “`ri`” stands for. It has been called “Ruby Index,” “Ruby Information,” and “Ruby Interactive.”

This documentation displayed by *ri* is extracted from specially formatted comments in Ruby source code. See §2.1.1.2 for details.

1.2.5 Ruby Package Management with *gem*

Ruby’s package management system is known as RubyGems, and packages or modules distributed using RubyGems are called “gems.” RubyGems makes it easy to install Ruby software and can automatically manage complex dependencies between packages.

The frontend script for RubyGems is *gem*, and it’s distributed with Ruby 1.9 just as *irb* and *ri* are. In Ruby 1.8, you must install it separately—see <http://rubygems.org>. Once the *gem* program is installed, you might use it like this:

```
# gem install rails
Successfully installed activesupport-1.4.4
Successfully installed activerecord-1.15.5
Successfully installed actionpack-1.13.5
Successfully installed actionmailer-1.3.5
Successfully installed actionwebservice-1.2.5
Successfully installed rails-1.2.5
6 gems installed
Installing ri documentation for activesupport-1.4.4...
Installing ri documentation for activerecord-1.15.5...
...etc...
```

As you can see, the *gem install* command installs the most recent version of the gem you request and also installs any gems that the requested gem requires. *gem* has other useful subcommands as well. Some examples:

```
gem list           # List installed gems
gem environment   # Display RubyGems configuration information
gem update rails  # Update a named gem
gem update        # Update all installed gems
gem update --system # Update RubyGems itself
gem uninstall rails # Remove an installed gem
```

In Ruby 1.8, the gems you install cannot be automatically loaded by Ruby’s `require` method. (See §7.6 for more about loading modules of Ruby code with the `require` method.) If you’re writing a program that will be using modules installed as gems, you must first require the `rubygems` module. Some Ruby 1.8 distributions are preconfigured with the `RubyGems` library, but you may need to download and install this manually. Loading this `rubygems` module alters the `require` method itself so that it searches the set of installed gems before it searches the standard library. You can also automatically enable RubyGems support by running Ruby with the `-rubygems` command-line option. And if you add `-rubygems` to the `RUBYOPT` environment variable, then the `RubyGems` library will be loaded on every invocation of Ruby.

The `rubygems` module is part of the standard library in Ruby 1.9, but it is no longer required to load gems. Ruby 1.9 knows how to find installed gems on its own, and you do not have to put `require 'rubygems'` in your programs that use gems.

When you load a gem with `require` (in either 1.8 or 1.9), it loads the most recent installed version of the gem you specify. If you have more specific version requirements, you can use the `gem` method before calling `require`. This finds a version of the gem matching the version constraints you specify and “activates” it, so that a subsequent `require` will load that version:

```
require 'rubygems'           # Not necessary in Ruby 1.9
gem 'RedCloth', '> 2.0', '< 4.0' # Activate RedCloth version 2.x or 3.x
require 'RedCloth'          # And now load it
```

You’ll find more about `require` and gems in §7.6.1. Complete coverage of RubyGems, the `gem` program, and the `rubygems` module are beyond the scope of this book. The `gem` command is self-documenting—start by running `gem help`. For details on the `gem` method, try `ri gem`. And for complete details, see the documentation at <http://rubygems.org>.

1.2.6 More Ruby Tutorials

This chapter began with a tutorial introduction to the Ruby language. You can try out the code snippets of that tutorial using `irb`. If you want more tutorials before diving into the language more formally, there are two good ones available by following links on the <http://www.ruby-lang.org> home page. One `irb`-based tutorial is called “Ruby in Twenty Minutes.”* Another tutorial, called “Try Ruby!”, is interesting because it works in your web browser and does not require you to have Ruby or `irb` installed on your system.†

1.2.7 Ruby Resources

The Ruby web site (<http://www.ruby-lang.org>) is the place to find links to other Ruby resources, such as online documentation, libraries, mailing lists, blogs, IRC channels, user groups, and conferences. Try the “Documentation,” “Libraries,” and “Community” links on the home page.

1.3 About This Book

As its title implies, this book covers the Ruby programming language and aspires to do so comprehensively and accessibly. This edition of the book covers language versions 1.8 and 1.9. Ruby blurs the distinction between language and platform, and so our coverage of the language includes a detailed overview of the core Ruby API. But this book is not an API reference and does not cover the core classes comprehensively. Also,

* At the time of this writing, the direct URL for this tutorial is <http://www.ruby-lang.org/en/documentation/quickstart/>.

† If you can’t find the “Try Ruby!” link on the Ruby home page, try this URL: <http://tryruby.hobix.com>.

this is not a book about Ruby frameworks (like Rails), nor a book about Ruby tools (like *rake* and *gem*).

This chapter concludes with a heavily commented extended example demonstrating a nontrivial Ruby program. The chapters that follow cover Ruby from the bottom up:

- Chapter 2 covers the lexical and syntactic structure of Ruby, including basic issues like character set, case sensitivity, and reserved words.
- Chapter 3 explains the kinds of data—numbers, strings, ranges, arrays, and so on—that Ruby programs can manipulate, and it covers the basic features of all Ruby objects.
- Chapter 4 covers primary expressions in Ruby—literals, variable references, *method* invocations, and assignments—and it explains the operators used to combine primary expressions into compound expressions.
- Chapter 5 explains conditionals, loops (including blocks and iterator methods), exceptions, and the other Ruby expressions that would be called statements or control structures in other languages.
- Chapter 6 formally documents Ruby’s method definition and invocation syntax, and it also covers the invocable objects known as procs and lambdas. This chapter includes an explanation of closures and an exploration of functional programming techniques in Ruby.
- Chapter 7 explains how to define classes and modules in Ruby. Classes are fundamental to object-oriented programming, and this chapter also covers topics such as inheritance, method visibility, mixin modules, and the method name resolution algorithm.
- Chapter 8 covers Ruby’s APIs that allow a program to inspect and manipulate itself, and then demonstrates metaprogramming techniques that use those APIs to make programming easier. The chapter includes an example of domain-specific language.
- Chapter 9 demonstrates the most important classes and methods of the core Ruby platform with simple code fragments. This is not a reference but a detailed overview of the core classes. Topics include text processing, numeric computation, collections (such as arrays and hashes), input/output, networking, and threads. After reading this chapter, you’ll understand the breadth of the Ruby platform, and you’ll be able to use the *ri* tool or an online reference to explore the platform in depth.
- Chapter 10 covers the top-level Ruby programming environment, including global variables and global functions, command-line arguments supported by the Ruby interpreter, and Ruby’s security mechanism.

1.3.1 How to Read This Book

It is easy to program in Ruby, but Ruby is not a simple language. Because this book documents Ruby comprehensively, it is not a simple book (though we hope that you find it easy to read and understand). It is intended for experienced programmers who want to master Ruby and are willing to read carefully and thoughtfully to achieve that goal.

Like all similar programming books, this book contains forward and backward references throughout. Programming languages are not linear systems, and it is impossible to document them linearly. As you can see from the chapter outline, this book takes a bottom-up approach to Ruby: it starts with the simplest elements of Ruby's grammar and moves on to document successively higher-level syntactic structures—from tokens to values to expressions and control structures to methods and classes. This is a classic approach to documenting programming languages, but it does not avoid the problem of forward references.

The book is intended to be read in the order it is written, but some advanced topics are best skimmed or skipped on the first reading; they will make much more sense when you come back to them after having read the chapters that follow. On the other hand, don't let every forward reference scare you off. Many of them are simply informative, letting you know that more details will be presented later. The reference does not necessarily imply that those future details are required to understand the current material.

1.4 A Sudoku Solver in Ruby

This chapter concludes with a nontrivial Ruby application to give you a better idea of what Ruby programs actually look like. We've chosen a Sudoku* solver as a good short to medium-length program that demonstrates a number of features of Ruby. Don't expect to understand every detail of Example 1-1, but do read through the code; it is very thoroughly commented, and you should have little difficulty following along.

Example 1-1. A Sudoku solver in Ruby

```
#
# This module defines a Sudoku::Puzzle class to represent a 9x9
# Sudoku puzzle and also defines exception classes raised for
# invalid input and over-constrained puzzles. This module also defines
# the method Sudoku.solve to solve a puzzle. The solve method uses
# the Sudoku.scan method, which is also defined here.
#
# Use this module to solve Sudoku puzzles with code like this:
```

* Sudoku is a logic puzzle that takes the form of a 9×9 grid of numbers and blank squares. The task is to fill each blank with a digit 1 to 9 so that no row or column or 3×3 subgrid includes the same digit twice. Sudoku has been popular in Japan for some time, but it gained sudden popularity in the English-speaking world in 2004 and 2005. If you are unfamiliar with Sudoku, try reading the Wikipedia entry (<http://en.wikipedia.org/wiki/Sudoku>) and try an online puzzle (<http://websudoku.com/>).

```

#
# require 'sudoku'
# puts Sudoku.solve(Sudoku::Puzzle.new(ARGF.readlines))
#
module Sudoku

  #
  # The Sudoku::Puzzle class represents the state of a 9x9 Sudoku puzzle.
  #
  # Some definitions and terminology used in this implementation:
  #
  # - Each element of a puzzle is called a "cell".
  # - Rows and columns are numbered from 0 to 8, and the coordinates [0,0]
  #   refer to the cell in the upper-left corner of the puzzle.
  # - The nine 3x3 subgrids are known as "boxes" and are also numbered from
  #   0 to 8, ordered from left to right and top to bottom. The box in
  #   the upper-left is box 0. The box in the upper-right is box 2. The
  #   box in the middle is box 4. The box in the lower-right is box 8.
  #
  # Create a new puzzle with Sudoku::Puzzle.new, specifying the initial
  # state as a string or as an array of strings. The string(s) should use
  # the characters 1 through 9 for the given values, and '.' for cells
  # whose value is unspecified. Whitespace in the input is ignored.
  #
  # Read and write access to individual cells of the puzzle is through the
  # [] and []= operators, which expect two-dimensional [row,column] indexing.
  # These methods use numbers (not characters) 0 to 9 for cell contents.
  # 0 represents an unknown value.
  #
  # The has_duplicates? predicate returns true if the puzzle is invalid
  # because any row, column, or box includes the same digit twice.
  #
  # The each_unknown method is an iterator that loops through the cells of
  # the puzzle and invokes the associated block once for each cell whose
  # value is unknown.
  #
  # The possible method returns an array of integers in the range 1..9.
  # The elements of the array are the only values allowed in the specified
  # cell. If this array is empty, then the puzzle is over-specified and
  # cannot be solved. If the array has only one element, then that element
  # must be the value for that cell of the puzzle.
  #
  class Puzzle

    # These constants are used for translating between the external
    # string representation of a puzzle and the internal representation.
    ASCII = ".123456789"
    BIN = "\000\001\002\003\004\005\006\007\010\011"

    # This is the initialization method for the class. It is automatically
    # invoked on new Puzzle instances created with Puzzle.new. Pass the input
    # puzzle as an array of lines or as a single string. Use ASCII digits 1
    # to 9 and use the '.' character for unknown cells. Whitespace,
    # including newlines, will be stripped.
    def initialize(lines)

```



```

if (lines.respond_to? :join) # If argument looks like an array of lines
  s = lines.join           # Then join them into a single string
else                       # Otherwise, assume we have a string
  s = lines.dup           # And make a private copy of it
end

# Remove whitespace (including newlines) from the data
# The '!' in gsub! indicates that this is a mutator method that
# alters the string directly rather than making a copy.
s.gsub!(/\s/, "") # /\s/ is a Regexp that matches any whitespace

# Raise an exception if the input is the wrong size.
# Note that we use unless instead of if, and use it in modifier form.
raise Invalid, "Grid is the wrong size" unless s.size == 81

# Check for invalid characters, and save the location of the first.
# Note that we assign and test the value assigned at the same time.
if i = s.index(/[^\d3456789\.]/)
  # Include the invalid character in the error message.
  # Note the Ruby expression inside #{ } in string literal.
  raise Invalid, "Illegal character #{s[i,1]} in puzzle"
end

# The following two lines convert our string of ASCII characters
# to an array of integers, using two powerful String methods.
# The resulting array is stored in the instance variable @grid
# The number 0 is used to represent an unknown value.
s.tr!(ASCII, BIN) # Translate ASCII characters into bytes
@grid = s.unpack('c*') # Now unpack the bytes into an array of numbers

# Make sure that the rows, columns, and boxes have no duplicates.
raise Invalid, "Initial puzzle has duplicates" if has_duplicates?
end

# Return the state of the puzzle as a string of 9 lines with 9
# characters (plus newline) each.
def to_s
  # This method is implemented with a single line of Ruby magic that
  # reverses the steps in the initialize() method. Writing dense code
  # like this is probably not good coding style, but it demonstrates
  # the power and expressiveness of the language.
  #
  # Broken down, the line below works like this:
  # (0..8).collect invokes the code in curly braces 9 times--once
  # for each row--and collects the return value of that code into an
  # array. The code in curly braces takes a subarray of the grid
  # representing a single row and packs its numbers into a string.
  # The join() method joins the elements of the array into a single
  # string with newlines between them. Finally, the tr() method
  # translates the binary string representation into ASCII digits.
  (0..8).collect{|r| @grid[r*9,9].pack('c9')}.join("\n").tr(BIN,ASCII)
end

# Return a duplicate of this Puzzle object.
# This method overrides Object.dup to copy the @grid array.

```

```

def dup
  copy = super      # Make a shallow copy by calling Object.dup
  @grid = @grid.dup # Make a new copy of the internal data
  copy             # Return the copied object
end

# We override the array access operator to allow access to the
# individual cells of a puzzle. Puzzles are two-dimensional,
# and must be indexed with row and column coordinates.
def [](row, col)
  # Convert two-dimensional (row,col) coordinates into a one-dimensional
  # array index and get and return the cell value at that index
  @grid[row*9 + col]
end

# This method allows the array access operator to be used on the
# lefthand side of an assignment operation. It sets the value of
# the cell at (row, col) to newvalue.
def []=(row, col, newvalue)
  # Raise an exception unless the new value is in the range 0 to 9.
  unless (0..9).include? newvalue
    raise Invalid, "illegal cell value"
  end
  # Set the appropriate element of the internal array to the value.
  @grid[row*9 + col] = newvalue
end

# This array maps from one-dimensional grid index to box number.
# It is used in the method below. The name BoxOfIndex begins with a
# capital letter, so this is a constant. Also, the array has been
# frozen, so it cannot be modified.
BoxOfIndex = [
  0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,
  3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,
  6,6,6,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8
].freeze

# This method defines a custom looping construct (an "iterator") for
# Sudoku puzzles. For each cell whose value is unknown, this method
# passes ("yields") the row number, column number, and box number to the
# block associated with this iterator.
def each_unknown
  0.upto 8 do |row|      # For each row
    0.upto 8 do |col|   # For each column
      index = row*9+col # Cell index for (row,col)
      next if @grid[index] != 0 # Move on if we know the cell's value
      box = BoxOfIndex[index] # Figure out the box for this cell
      yield row, col, box    # Invoke the associated block
    end
  end
end

# Returns true if any row, column, or box has duplicates.
# Otherwise returns false. Duplicates in rows, columns, or boxes are not
# allowed in Sudoku, so a return value of true means an invalid puzzle.

```

```

def has_duplicates?
  # uniq! returns nil if all the elements in an array are unique.
  # So if uniq! returns something then the board has duplicates.
  0.upto(8) {|row| return true if rowdigits(row).uniq! }
  0.upto(8) {|col| return true if coldigits(col).uniq! }
  0.upto(8) {|box| return true if boxdigits(box).uniq! }

  false # If all the tests have passed, then the board has no duplicates
end

# This array holds a set of all Sudoku digits. Used below.
AllDigits = [1, 2, 3, 4, 5, 6, 7, 8, 9].freeze

# Return an array of all values that could be placed in the cell
# at (row,col) without creating a duplicate in the row, column, or box.
# Note that the + operator on arrays does concatenation but that the -
# operator performs a set difference operation.
def possible(row, col, box)
  AllDigits - (rowdigits(row) + coldigits(col) + boxdigits(box))
end

private # All methods after this line are private to the class

# Return an array of all known values in the specified row.
def rowdigits(row)
  # Extract the subarray that represents the row and remove all zeros.
  # Array subtraction is set difference, with duplicate removal.
  @grid[row*9,9] - [0]
end

# Return an array of all known values in the specified column.
def coldigits(col)
  result = [] # Start with an empty array
  col.step(80, 9) {|i| # Loop from col by nines up to 80
    v = @grid[i] # Get value of cell at that index
    result << v if (v != 0) # Add it to the array if non-zero
  }
  result # Return the array
end

# Map box number to the index of the upper-left corner of the box.
BoxToIndex = [0, 3, 6, 27, 30, 33, 54, 57, 60].freeze

# Return an array of all the known values in the specified box.
def boxdigits(b)
  # Convert box number to index of upper-left corner of the box.
  i = BoxToIndex[b]
  # Return an array of values, with 0 elements removed.
  [
    @grid[i], @grid[i+1], @grid[i+2],
    @grid[i+9], @grid[i+10], @grid[i+11],
    @grid[i+18], @grid[i+19], @grid[i+20]
  ] - [0]
end

end # This is the end of the Puzzle class

```

```

# An exception of this class indicates invalid input,
class Invalid < StandardError
end

# An exception of this class indicates that a puzzle is over-constrained
# and that no solution is possible.
class Impossible < StandardError
end

#
# This method scans a Puzzle, looking for unknown cells that have only
# a single possible value. If it finds any, it sets their value. Since
# setting a cell alters the possible values for other cells, it
# continues scanning until it has scanned the entire puzzle without
# finding any cells whose value it can set.
#
# This method returns three values. If it solves the puzzle, all three
# values are nil. Otherwise, the first two values returned are the row and
# column of a cell whose value is still unknown. The third value is the
# set of values possible at that row and column. This is a minimal set of
# possible values: there is no unknown cell in the puzzle that has fewer
# possible values. This complex return value enables a useful heuristic
# in the solve() method: that method can guess at values for cells where
# the guess is most likely to be correct.
#
# This method raises Impossible if it finds a cell for which there are
# no possible values. This can happen if the puzzle is over-constrained,
# or if the solve() method below has made an incorrect guess.
#
# This method mutates the specified Puzzle object in place.
# If has_duplicates? is false on entry, then it will be false on exit.
#
def Sudoku.scan(puzzle)
  unchanged = false # This is our loop variable

  # Loop until we've scanned the whole board without making a change.
  until unchanged
    unchanged = true # Assume no cells will be changed this time
    rmin,cmin,pmin = nil # Track cell with minimal possible set
    min = 10 # More than the maximal number of possibilities

    # Loop through cells whose value is unknown.
    puzzle.each_unknown do |row, col, box|
      # Find the set of values that could go in this cell
      p = puzzle.possible(row, col, box)

      # Branch based on the size of the set p.
      # We care about 3 cases: p.size==0, p.size==1, and p.size > 1.
      case p.size
      when 0 # No possible values means the puzzle is over-constrained
        raise Impossible
      when 1 # We've found a unique value, so set it in the grid
        puzzle[row,col] = p[0] # Set that position on the grid to the value
        unchanged = false # Note that we've made a change
      end
    end
  end
end

```

```

    else # For any other number of possibilities
      # Keep track of the smallest set of possibilities.
      # But don't bother if we're going to repeat this loop.
      if unchanged && p.size < min
        min = p.size # Current smallest size
        rmin, cmin, pmin = row, col, p # Note parallel assignment
      end
    end
  end
end

# Return the cell with the minimal set of possibilities.
# Note multiple return values.
return rmin, cmin, pmin
end

# Solve a Sudoku puzzle using simple logic, if possible, but fall back
# on brute-force when necessary. This is a recursive method. It either
# returns a solution or raises an exception. The solution is returned
# as a new Puzzle object with no unknown cells. This method does not
# modify the Puzzle it is passed. Note that this method cannot detect
# an under-constrained puzzle.
def Sudoku.solve(puzzle)
  # Make a private copy of the puzzle that we can modify.
  puzzle = puzzle.dup

  # Use logic to fill in as much of the puzzle as we can.
  # This method mutates the puzzle we give it, but always leaves it valid.
  # It returns a row, a column, and set of possible values at that cell.
  # Note parallel assignment of these return values to three variables.
  r,c,p = scan(puzzle)

  # If we solved it with logic, return the solved puzzle.
  return puzzle if r == nil

  # Otherwise, try each of the values in p for cell [r,c].
  # Since we're picking from a set of possible values, the guess leaves
  # the puzzle in a valid state. The guess will either lead to a solution
  # or to an impossible puzzle. We'll know we have an impossible
  # puzzle if a recursive call to scan throws an exception. If this happens
  # we need to try another guess, or re-raise an exception if we've tried
  # all the options we've got.
  p.each do |guess| # For each value in the set of possible values
    puzzle[r,c] = guess # Guess the value

    begin
      # Now try (recursively) to solve the modified puzzle.
      # This recursive invocation will call scan() again to apply logic
      # to the modified board, and will then guess another cell if needed.
      # Remember that solve() will either return a valid solution or
      # raise an exception.
      return solve(puzzle) # If it returns, we just return the solution
    rescue Impossible
      next # If it raises an exception, try the next guess
    end
  end
end

```

```
end

  # If we get here, then none of our guesses worked out
  # so we must have guessed wrong sometime earlier.
  raise Impossible
end
end
```

Example 1-1 is 345 lines long. Because the example was written for this introductory chapter, it has particularly verbose comments. Strip away the comments and the blank lines and you're left with just 129 lines of code, which is pretty good for an object-oriented Sudoku solver that does not rely on a simple brute-force algorithm. We hope that this example demonstrates the power and expressiveness of Ruby.

The Structure and Execution of Ruby Programs



This chapter explains the structure of Ruby programs. It starts with the lexical structure, covering tokens and the characters that comprise them. Next, it covers the syntactic structure of a Ruby program, explaining how expressions, control structures, methods, classes, and so on are written as a series of tokens. Finally, the chapter describes files of Ruby code, explaining how Ruby programs can be split across multiple files and how the Ruby interpreter executes a file of Ruby code.

2.1 Lexical Structure

The Ruby interpreter parses a program as a sequence of *tokens*. Tokens include comments, literals, punctuation, identifiers, and keywords. This section introduces these types of tokens and also includes important information about the characters that comprise the tokens and the whitespace that separates the tokens.

2.1.1 Comments

Comments in Ruby begin with a # character and continue to the end of the line. The Ruby interpreter ignores the # character and any text that follows it (but does not ignore the newline character, which is meaningful whitespace and may serve as a statement terminator). If a # character appears within a string or regular expression literal (see Chapter 3), then it is simply part of the string or regular expression and does not introduce a comment:

```
# This entire line is a comment
x = "#This is a string"      # And this is a comment
y = /#This is a regular expression/ # Here's another comment
```

Multiline comments are usually written simply by beginning each line with a separate # character:

```
#
# This class represents a Complex number
# Despite its name, it is not complex at all.
#
```

Note that Ruby has no equivalent of the C-style `/*...*/` comment. There is no way to embed a comment in the middle of a line of code.

2.1.1.1 Embedded documents

Ruby supports another style of multiline comment known as an *embedded document*. These start on a line that begins `=begin` and continue until (and include) a line that begins `=end`. Any text that appears after `=begin` or `=end` is part of the comment and is also ignored, but that extra text must be separated from the `=begin` and `=end` by at least one space.

Embedded documents are a convenient way to comment out long blocks of code without prefixing each line with a # character:


```
=begin Someone needs to fix the broken code below!  
  Any code here is commented out  
=end
```

Note that embedded documents only work if the = signs are the first characters of each line:

```
# =begin This used to begin a comment. Now it is itself commented out!  
  The code that goes here is no longer commented out  
# =end
```

As their name implies, embedded documents can be used to include long blocks of documentation within a program, or to embed source code of another language (such as HTML or SQL) within a Ruby program. Embedded documents are usually intended to be used by some kind of postprocessing tool that is run over the Ruby source code, and it is typical to follow `=begin` with an identifier that indicates which tool the comment is intended for.

2.1.1.2 Documentation comments

Ruby programs can include embedded API documentation as specially formatted comments that precede method, class, and module definitions. You can browse this documentation using the *ri* tool described earlier in §1.2.4. The *rdoc* tool extracts documentation comments from Ruby source and formats them as HTML or prepares them for display by *ri*. Documentation of the *rdoc* tool is beyond the scope of this book; see the file *lib/rdoc/README* in the Ruby source code for details.

Documentation comments must come immediately before the module, class, or method whose API they document. They are usually written as multiline comments where each line begins with #, but they can also be written as embedded documents that start `=begin rdoc`. (The *rdoc* tool will not process these comments if you leave out the “*rdoc*”.)

The following example comment demonstrates the most important formatting elements of the markup grammar used in Ruby’s documentation comments; a detailed description of the grammar is available in the *README* file mentioned previously:

```
#  
# Rdoc comments use a simple markup grammar like those used in wikis.  
#  
# Separate paragraphs with a blank line.  
#  
# = Headings  
#  
# Headings begin with an equals sign  
#  
# == Sub-Headings  
# The line above produces a subheading.  
# === Sub-Sub-Heading  
# And so on.  
#  
# = Examples
```

```

#
# Indented lines are displayed verbatim in code font.
# Be careful not to indent your headings and lists, though.
#
# = Lists and Fonts
#
# List items begin with * or -. Indicate fonts with punctuation or HTML:
# * _italic_ or <i>multi-word italic</i>
# * *bold* or <b>multi-word bold</b>
# * +code+ or <tt>multi-word code</tt>
#
# 1. Numbered lists begin with numbers.
# 99. Any number will do; they don't have to be sequential.
# 1. There is no way to do nested lists.
#
# The terms of a description list are bracketed:
# [item 1] This is a description of item 1
# [item 2] This is a description of item 2
#

```

2.1.2 Literals

Literals are values that appear directly in Ruby source code. They include numbers, strings of text, and regular expressions. (Other literals, such as array and hash values, are not individual tokens but are more complex expressions.) Ruby number and string literal syntax is actually quite complicated, and is covered in detail in Chapter 3. For now, an example suffices to illustrate what Ruby literals look like:

```

1           # An integer literal
1.0        # A floating-point literal
'one'      # A string literal
"two"      # Another string literal
/three/    # A regular expression literal

```

2.1.3 Punctuation

Ruby uses punctuation characters for a number of purposes. Most Ruby operators are written using punctuation characters, such as + for addition, * for multiplication, and || for the Boolean OR operation. See §4.6 for a complete list of Ruby operators. Punctuation characters also serve to delimit string, regular expression, array, and hash literals, and to group and separate expressions, method arguments, and array indexes. We'll see miscellaneous other uses of punctuation scattered throughout Ruby syntax.

2.1.4 Identifiers

An *identifier* is simply a name. Ruby uses identifiers to name variables, methods, classes, and so forth. Ruby identifiers consist of letters, numbers, and underscore characters, but they may not begin with a number. Identifiers may not include whitespace or

nonprinting characters, and they may not include punctuation characters except as described here.

Identifiers that begin with a capital letter A–Z are constants, and the Ruby interpreter will issue a warning (but not an error) if you alter the value of such an identifier. Class and module names must begin with initial capital letters. The following are identifiers:

```
i
x2
old_value
_internal    # Identifiers may begin with underscores
PI           # Constant
```

By convention, multiword identifiers that are not constants are written with underscores like `_this`, whereas multiword constants are written `LikeThis` or `LIKE_THIS`.

2.1.4.1 Case sensitivity

Ruby is a case-sensitive language. Lowercase letters and uppercase letters are distinct. The keyword `end`, for example, is completely different from the keyword `END`.

2.1.4.2 Unicode characters in identifiers

Ruby's rules for forming identifiers are defined in terms of ASCII characters that are not allowed. In general, all characters outside of the ASCII character set are valid in identifiers, including characters that appear to be punctuation. In a UTF-8 encoded file, for example, the following Ruby code is valid:

```
def ×(x,y) # The name of this method is the Unicode multiplication sign
  x*y     # The body of this method multiplies its arguments
end
```

Similarly, a Japanese programmer writing a program encoded in SJIS or EUC can include Kanji characters in her identifiers. See §2.4.1 for more about writing Ruby programs using encodings other than ASCII.

The special rules about forming identifiers are based on ASCII characters and are not enforced for characters outside of that set. An identifier may not begin with an ASCII digit, for example, but it may begin with a digit from a non-Latin alphabet. Similarly, an identifier must begin with an ASCII capital letter in order to be considered a constant. The identifier `Ã`, for example, is not a constant.

Two identifiers are the same only if they are represented by the same sequence of bytes. Some character sets, such as Unicode, have more than one codepoint that represents the same character. No Unicode normalization is performed in Ruby, and two distinct codepoints are treated as distinct characters, even if they have the same meaning or are represented by the same font glyph.

2.1.4.3 Punctuation in identifiers

Punctuation characters may appear at the start and end of Ruby identifiers. They have the following meanings:

- \$ Global variables are prefixed with a dollar sign. Following Perl's example, Ruby defines a number of global variables that include other punctuation characters, such as `$_` and `$-K`. See Chapter 10 for a list of these special globals.
- @ Instance variables are prefixed with a single at sign, and class variables are prefixed with two at signs. Instance variables and class variables are explained in Chapter 7.
- ? As a helpful convention, methods that return Boolean values often have names that end with a question mark.
- ! Method names may end with an exclamation point to indicate that they should be used cautiously. This naming convention is often to distinguish mutator methods that alter the object on which they are invoked from variants that return a modified copy of the original object.
- = Methods whose names end with an equals sign can be invoked by placing the method name, without the equals sign, on the left side of an assignment operator. (You can read more about this in §4.5.3 and §7.1.5.)

Here are some example identifiers that contain leading or trailing punctuation characters:

```
$files      # A global variable
@data      # An instance variable
@@counter  # A class variable
empty?     # A Boolean-valued method or predicate
sort!      # An in-place alternative to the regular sort method
timeout=   # A method invoked by assignment
```

A number of Ruby's operators are implemented as methods, so that classes can redefine them for their own purposes. It is therefore possible to use certain operators as method names as well. In this context, the punctuation character or characters of the operator are treated as identifiers rather than operators. See §4.6 for more about Ruby's operators.

2.1.5 Keywords

The following keywords have special meaning in Ruby and are treated specially by the Ruby parser:

```
__LINE__   case      ensure   not       then
__ENCODING__ class    false    or        true
__FILE__   def      for      redo      undef
BEGIN      defined? if        rescue   unless
END        do       in        retry    until
alias      else     module   return   when
and        elsif   next     self     while
begin      end      nil      super    yield
break
```

In addition to those keywords, there are three keyword-like tokens that are treated specially by the Ruby parser when they appear at the beginning of a line:

```
=begin    =end    __END__
```

As we've seen, `=begin` and `=end` at the beginning of a line delimit multiline comments. And the token `__END__` marks the end of the program (and the beginning of a data section) if it appears on a line by itself with no leading or trailing whitespace.

In most languages, these words would be called “reserved words” and they would be never allowed as identifiers. The Ruby parser is flexible and does not complain if you prefix these keywords with `@`, `@@`, or `$` prefixes and use them as instance, class, or global variable names. Also, you can use these keywords as method names, with the caveat that the method must always be explicitly invoked through an object. Note, however, that using these keywords in identifiers will result in confusing code. The best practice is to treat these keywords as reserved.

Many important features of the Ruby language are actually implemented as methods of the `Kernel`, `Module`, `Class`, and `Object` classes. It is good practice, therefore, to treat the following identifiers as reserved words as well:

```
# These are methods that appear to be statements or keywords
at_exit      catch      private      require
attr         include     proc         throw
attr_accessor lambda     protected
attr_reader  load       public
attr_writer  loop       raise

# These are commonly used global functions
Array        chomp!     gsub!       select
Float        chop       iterator?   sleep
Integer      chop!     load        split
String       eval       open        sprintf
URI          exec       p           srand
abort        exit       print       sub
autoload     exit!     printf      sub!
autoload?    fail      puts        syscall
binding      fork      puts        system
block_given? format     rand        test
callcc       getc      readline    trap
caller       gets      readlines   warn
chomp        gsub     scan

# These are commonly used object methods
allocate     freeze     kind_of?    superclass
clone        frozen?    method      taint
display      hash       methods     tainted?
dup          id         new         to_a
enum_for     inherited  nil?        to_enum
eq?          inspect   object_id   to_s
equal?       instance_of? respond_to? untaint
extend       is_a?     send
```