Table of Contents

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Chapter 2. Simple Data Tasks

IN THIS CHAPTER

Theory attracts practice as the magnet attracts iron.

—Karl Friedrich Gauss

One measure of the sophistication of a programming language is: What kinds of data will it directly support? The earliest computers were programmed strictly in machine language with purely numeric data. Soon after, the concept of character data and strings of characters was invented, which was crucial to the development of general-purpose languages.

As time goes by, we find ourselves dealing with data of increasing complexity. Modern languages frequently include support for many kinds of data. Note that we don't say types here because the usual notion of a type might be somewhat different. For example, a regular expression might be stored essentially in the form of a character string, but we don't really consider them to be strings because of their special uses.

We could easily add things like arrays and hashes to this list because these are, for Ruby, fairly low-level entities. In fact, there are some incidental uses of these in this chapter. But arrays and hashes (and more complex data structures) deserve a chapter of their own.

This chapter, then, is devoted to four of the most common kinds of data in Ruby. These are strings, regular expressions, numbers, and times and dates.

A *string,* as in other languages, is simply a sequence of characters. Similar to most entities in Ruby, strings are first-class objects.

Regular expressions form a very condensed notation for describing patterns within text. These have been around for decades and have become even more commonly used in the last 10 years.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Numbers need little explanation; they comprise both integers and floating-point numbers. In Ruby, integers can be of class `Fixnum` or `Bignum`, depending on their magnitude.

Times and dates are problematic in any language. Ruby strives to sort through the confusion with an object-oriented interface to the traditional time and date routines.

The alert reader might notice that we don't include the `Range` class in this discussion. This isn't because ranges aren't useful, but because they aren't that complex; that class is far less rich and interesting than the others covered here. But ranges are certainly covered in incidental code throughout the entire book.

Let's look at some sample code now. We'll begin with strings.

Strings

> *Atoms were once thought to be fundamental, elementary building blocks of nature; protons were then thought to be fundamental, then quarks. Now we say the string is fundamental.*
>
> —David Gross, professor of theoretical physics, Princeton University

We offer an anecdote here. In the early 1980s, a computer science professor started out his data structures class with a single question. He didn't introduce himself or state the name of the course; he didn't hand out a syllabus or give the name of the textbook. He walked to the front of the class and asked, "What is the most important data type?"

There were one or two guesses. Someone guessed, "Pointers." He brightened, but said no, that wasn't it. Then he offered his opinion: The most important data type was *character data*.

He had a valid point. Computers are supposed to be our servants, not our masters, and character data has the distinction of being human readable. (Some humans can read binary data easily, but we will ignore them.) The existence of characters (and thus strings) enables communication between humans and computers. Every kind of information we can imagine, including natural language text, can be encoded in character strings.

What do we find ourselves wanting to do with strings? We want to concatenate them, tokenize them, analyze them, perform searches and substitutions, and more. Ruby makes most of these tasks easy.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Performing Specialized String Comparisons

Ruby has built-in ideas about comparing strings; comparisons are done lexicographically as we have come to expect (that is, based on character set order). But if we want, we can introduce rules of our own for string comparisons, and these can be of arbitrary complexity.

As an example, suppose that we want to ignore the English articles *a, an,* and *the* at the front of a string, and we also want to ignore most common punctuation marks. We can do this by overriding the built-in method <=>, which is called for <, <=, >, and >= (see Listing 2.1).

Listing 2.1. Specialized String Comparisons

```
class String

    alias old_compare <=>

    def <=>(other)
      a = self.dup
    b = other.dup
    # Remove punctuation
    a.gsub!(/[\,\.\?\!\:\;]/, "")
    b.gsub!(/[\,\.\?\!\:\;]/, "")
    # Remove initial articles
    a.gsub!(/^(a |an |the )/i, "")
    b.gsub!(/^(a |an |the )/i, "")
    # Remove leading/trailing whitespace
    a.strip!
    b.strip!
    # Use the old <=>
    a.old_compare(b)
  end

end


title1 = "Calling All Cars"
title2 = "The Call of the Wild"

# Ordinarily this would print "yes"

if title1 < title2
  puts "yes"
else
  puts "no"         # But now it prints "no"
end
```

Note that we save the old <=> with an alias and then call it at the end. This is because if we tried to use the < method, it would call the new <=> rather than the old one, resulting in infinite recursion and a program crash.

Note also that the == operator doesn't call the <=> method (mixed in from comparable). This means that if we need to check equality in some specialized way, we will have to override the == method separately. But in this case, == works as we want it to anyhow.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Tokenizing a String

The `split` method will parse a string and return an array of tokens. It accepts two parameters, a delimiter, and a field limit, which is an integer.

The delimiter defaults to whitespace. Actually, it uses `$;` or the English equivalent `$FIELD_SEPARATOR`. If the delimiter is a string, the explicit value of that string is used as a token separator.

```
s1 = "It was a dark and stormy night."
words = s1.split          # ["It", "was", "a", "dark", "and",
                          #  "stormy", "night"]

s2 = "apples, pears, and peaches"
list = s2.split(", ")     # ["apples", "pears", "and peaches"]

s3 = "lions and tigers and bears"
zoo = s3.split(/ and /)   # ["lions", "tigers", "bears"]
```

The limit parameter places an upper limit on the number of fields returned, according to these rules:

1. If it is omitted, trailing null entries are suppressed.
2. If it is a positive number, the number of entries will be limited to that number (stuffing the rest of the string into the last field as needed). Trailing null entries are retained.
3. If it is a negative number, there is no limit to the number of fields, and trailing null entries are retained.

These three rules are illustrated here:

```
str = "alpha,beta,gamma,,"
list1 = str.split(",")     # ["alpha","beta","gamma"]
list2 = str.split(",",2)   # ["alpha", "beta,gamma,,"]
list3 = str.split(",",4)   # ["alpha", "beta", "gamma", ","]
list4 = str.split(",",8)   # ["alpha", "beta", "gamma", "", ""]
list5 = str.split(",",-1)  # ["alpha", "beta", "gamma", "", ""]
```

Formatting a String

String formatting is done in Ruby as it is in C—with the `sprintf` method. It takes a string and a list of expressions as parameters and returns a string. The format string contains essentially the same set of specifiers that are available with C's `sprintf` (or `printf`).

```
name = "Bob"
age = 28
str = sprintf("Hi, %s... I see you're %d years old.", name, age)
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

You might ask why we would use this instead of simply interpolating values into a string using the `#{ expr}` notation. The answer is that `sprintf` makes it possible to do extra formatting such as specifying a maximum width, specifying a maximum number of decimal places, adding or suppressing leading zeroes, left-justifying, right-justifying, and more.

```
str = sprintf("%-20s  %3d", name, age)
```

The `String` class has a method `%`, which will do much the same thing. It takes a single value or an array of values of any type.

```
str = "%-20s  %3d" % [name, age]  # Same as previous example
```

We also have the methods `ljust`, `rjust`, and `center`; these take a length for the destination string and pad with spaces as needed.

```
str = "Moby-Dick"
s1 = str.ljust(13)           # "Moby-Dick    "
s2 = str.center(13)          # "  Moby-Dick  "
s3 = str.rjust(13)           # "    Moby-Dick"
```

For more information, see any reference.

Controlling Uppercase and Lowercase

Ruby's `String` class offers a rich set of methods for controlling case. We offer an overview of them here.

The `downcase` method will convert a string to all lowercase. Likewise `upcase` will convert it to all uppercase.

```
s1 = "Boston Tea Party"
s2 = s1.downcase             # "boston tea party"
s3 = s2.upcase               # "BOSTON TEA PARTY"
```

The `capitalize` method will capitalize the first character of a string while forcing all the remaining characters to be lowercase.

```
s4 = s1.capitalize           # "Boston tea party"
s5 = s2.capitalize           # "Boston tea party"
s6 = s3.capitalize           # "Boston tea party"
```

The `swapcase` method will exchange the case of each letter in a string.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
s7 = "THIS IS AN ex-parrot."
s8 = s7.swapcase             # "this is an EX-PARROT."
```

Each of these has its in-place equivalent (`upcase!`, `downcase!`, `capitalize!`, `swapcase!`).

There are no built-in methods for detecting case, but this is easy to do with regular expressions.

```
if string =~ /[a-z]/
  puts "string contains lowercase charcters"
end

if string =~ /[A-Z]/
  puts "string contains uppercase charcters"
end

if string =~ /[A-Z]/ and string =~ /a-z/
  puts "string contains mixed case"
end

if string[0..0] =~ /[A-Z]/
  puts "string starts with a capital letter"
end
```

Note that all these methods ignore locale.

Accessing and Assigning Substrings

In Ruby, substrings can be accessed in several different ways. Normally the bracket notation is used, as for an array; but the brackets can contain a pair of `Fixnum`s, a range, a regex, or a string. Each case is discussed in turn.

If a pair of `Fixnum` values is specified, they are treated as an offset and a length, and the corresponding substring is returned:

```
str = "Humpty Dumpty"
sub1 = str[7,4]          # "Dump"
sub2 = str[7,99]         # "Dumpty" (overrunning is OK)
sub3 = str[10,-4]        # nil (length is negative)
```

It is important to remember that these are an offset and a length (number of characters), not beginning and ending offsets.

A negative index counts backward from the end of the string. In this case, the index is one-based, not zero-based. The length is still added in the forward direction.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
str1 = "Alice"
sub1 = str1[-3,3]   # "ice"
str2 = "Through the Looking-Glass"
sub3 = str2[-13,4]  # "Look"
```

A range can be specified. In this case, the range is taken as a range of indices into the string. Ranges can have negative numbers, but the numerically lower number must still be first in the range. If the range is backward or if the initial value is outside the string, `nil` is returned.

```
str = "Winston Churchill"
sub1 = str[8..13]    # "Church"
sub2 = str[-4..-1]   # "hill"
sub3 = str[-1..-4]   # nil
sub4 = str[25..30]   # nil
```

If a regular expression is specified, the string matching that pattern will be returned. If there is no match, `nil` will be returned.

```
str = "Alistair Cooke"
sub1 = str[/l..t/]   # "list"
sub2 = str[/s.*r/]   # "stair"
sub3 = str[/foo/]    # nil
```

If a string is specified, that string will be returned if it appears as a substring (or `nil` if it doesn't).

```
str = "theater"
sub1 = str["heat"]  # "heat"
sub2 = str["eat"]   # "eat"
sub3 = str["ate"]   # "ate"
sub4 = str["beat"]  # nil
sub5 = str["cheat"] # nil
```

Finally, in the trivial case, a single `Fixnum` as index will yield an ASCII code (or `nil` if out of range).

```
str = "Aaron Burr"
ch1 = str[0]     # 65
ch1 = str[1]     # 97
ch3 = str[99]    # nil
```

It is important to realize that the notations we have described here will serve for assigning values as well as for accessing them.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
str1 = "Humpty Dumpty"
str1[7,4] = "Moriar"     # "Humpty Moriarty"

str2 = "Alice"
str2[-3,3] = "exandra"   # "Alexandra"

str3 = "Through the Looking-Glass"
str3[-13,13]  = "Mirror" # "Through the Mirror"

str4 = "Winston Churchill"
str4[8..13] = "H"        # "Winston Hill"

str5 = "Alistair Cooke"
str5[/e$/] ="ie Monster" # "Alistair Cookie Monster"

str6 = "theater"
str6["er"] = "re"        # "theatre"
str7 = "Aaron Burr"
str7[0] = 66             # "Baron Burr"
```

Assigning to an expression evaluating to `nil` will have no effect.

Substituting in Strings

You've already seen how to perform simple substitutions in strings. The `sub` and `gsub` methods provide more advanced pattern-based capabilities. There are also `sub!` and `gsub!`, which are their in-place counterparts.

The `sub` method will substitute the first occurrence of a pattern with the given substitute string or the given block.

```
s1 = "spam, spam, and eggs"
s2 = s1.sub(/spam/,"bacon")               # "bacon, spam,
and eggs"

s3 = s2.sub(/(\w+), (\w+),/,'\2, \1,')  # "spam, bacon, and eggs"

s4 = "Don't forget the spam."
s5 = s4.sub(/spam/) {  |m| m.reverse }      # "Don't forget the maps."

s4.sub!(/spam/) {  |m| m.reverse }
# s4 is now "Don't forget the maps."
```

As this example shows, the special symbols `\1`, `\2`, and so on can be used in a substitute string. However, special variables such as `$&` (or the English version `$MATCH`) might not.

If the block form is used, the special variables can be used. However, if all you need is the matched string, it will be passed into the block as a parameter. If it isn't needed at all, the parameter can of course be omitted.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The `gsub` method (global substitution) is essentially the same except that all matches are substituted rather than just the first.

```
s5 = "alfalfa abracadabra"
s6 = s5.gsub(/a[bl]/,"xx")     # "xxfxxfa xxracadxxra"
s5.gsub!(/[lfdbr]/) {  |m| m.upcase + "-" }
# s5 is now "aL-F-aL-F-a aB-R-acaD-aB-R-a"
```

The method `Regexp.last_match` is essentially identical to `$&` or `$MATCH`.

Searching a String

Besides the techniques for accessing substrings, there are other ways of searching within strings. The `index` method will return the starting location of the specified substring, character, or regex. If the item isn't found, the result is `nil`.

```
str = "Albert Einstein"
pos1 = str.index(?E)     # 7
pos2 = str.index("bert") # 2
pos3 = str.index(/in/)   # 8
pos4 = str.index(?W)     # nil
pos5 = str.index("bart") # nil
pos6 = str.index(/wein/) # nil
```

The method `rindex` (right index) will start from the right side of the string (that is, from the end). The numbering, however, proceeds from the beginning as usual.

```
str = "Albert Einstein"
pos1 = str.rindex(?E)     # 7
pos2 = str.rindex("bert") # 2
pos3 = str.rindex(/in/)   # 13 (finds rightmost match)
pos4 = str.rindex(?W)     # nil
pos5 = str.rindex("bart") # nil
pos6 = str.rindex(/wein/) # nil
```

The `include?` method simply tells whether the specified substring or character occurs within the string.

```
str1 = "mathematics"
flag1 = str1.include? ?e         # true
flag2 = str1.include? "math"     # true
str2 = "Daylight Saving Time"
flag3 = str2.include? ?s         # false
flag4 = str2.include? "Savings"  # false
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The `scan` method will repeatedly scan for occurrences of a pattern. If called without a block, it will return an array. If the pattern has more than one (parenthesized) group, the array will be nested.

```
str1 = "abracadabra"
sub1 = str1.scan(/a./)
# sub1 now is ["ab","ac","ad","ab"]

str2 = "Acapulco, Mexico"
sub2 = str2.scan(/(.)(c.)/)
# sub2 now is [ ["A","ca"], ["l","co"], ["i","co"] ]
```

If a block is specified, the method will pass the successive values to the block:

```
str3 = "Kobayashi"
str3.scan(/[^aeiou]+[aeiou]/) do |x|
  print "Syllable: #{ x} \n"
end
```

This code will produce the following output:

```
Syllable: Ko
Syllable: ba
Syllable: ya
Syllable: shi
```

Converting Between Characters and ASCII Codes

In Ruby, a character is already an integer.

```
str = "Martin"
print str[0]        # 77
```

If a `Fixnum` is appended directly onto a string, it is converted to a character.

```
str2 = str << 111   # "Martino"
```

The method `length` can be used for finding the length of a string. A synonym is size.

```
str1 = "Carl"
x = str1.length     # 4
str2 = "Doyle"
x = str2.size       # 5
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Processing a Line at a Time

A Ruby string can contain newlines. For example, a small enough file can be read into memory and stored in a single string. The default iterator `each` will process such a string one line at a time.

```
str = "Once upon\na time...\nThe End\n"
num = 0
str.each do |line|
  num += 1
  print "Line #{ num} : #{ line} "
end
```

This code produces three lines of output:

```
Line 1: Once upon
Line 2: a time...
Line 3: The End
```

The method `each_with_index` could also be used in this case.

Processing a Byte at a Time

Because Ruby isn't fully internationalized at the time of this writing, a character is essentially the same as a byte. To process these in sequence, use the `each_byte` iterator.

```
str = "ABC"
str.each_byte do |char|
  print char, " "
end
# Produces output: 65 66 67
```

Appending an Item onto a String

The append operator << can be used to append strings onto another string. It is *stackable* in that multiple operations can be performed in sequence on a given receiver.

```
str = "A"
str << [1,2,3].to_s << " " << (3.14).to_s
# str is now "A123 3.14"
```

If a `Fixnum` in the range 0–255 is specified, it will be converted to a character.

```
str = "Marlow"
str << 101 << ", Christopher"
# str is now "Marlowe, Christopher"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Removing Trailing Newlines and Other Characters

Often we want to remove extraneous characters from the end of a string. The prime example is a newline on a string read from input.

The `chop` method will remove the last character of the string (typically, a trailing newline character). If the character before the newline is a carriage return (`\r`), it will be removed also. The reason for this behavior is the discrepancy between different systems' concept of what a newline is. On some systems such as UNIX, the newline character is represented internally as a linefeed (`\n`). On others, such as DOS and Windows, it is stored as a carriage return followed by a linefeed (`\r\n`).

```
str = gets.chop          # Read string, remove newline
s2 = "Some string\n"     # "Some string" (no newline)
s3 = s2.chop!            # s2 is now "Some string" also
s4 = "Other string\r\n"
s4.chop!                 # "Other string" (again no newline)
```

Note that the in-place version of the method (`chop!`) will modify its receiver.

It is also very important to note that in the absence of a trailing newline, the last character will be removed anyway.

```
str = "abcxyz"
s1 = str.chop            # "abcxy"
```

Because a newline might not always be present, the `chomp` method might be a better alternative.

```
str = "abcxyz"
str2 = "123\n"
s1 = str.chomp           # "abcxyz"
s2 = str2.chomp          # "123"
```

There is also a `chomp!` method as we would expect.

If a parameter is specified for `chomp`, it will remove the set of characters specified from the end of the string rather than the default record separator. Note that if the record separator appears in the middle of the string, it is ignored.

```
str1 = "abcxyz"
str2 = "abcxyz"
s1 = str1.chomp("yz")    # "abcx"
s2 = str2.chomp("x")     # "abcxyz"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Trimming Whitespace from a String

The `strip` method will remove whitespace from the beginning and end of a string. Its counterpart `strip!` will modify the receiver in place.

```
str1 = "\t  \nabc  \t\n"
str2 = str1.strip         # "abc"
str3 = str1.strip!        # "abc"
# str1 is now "abc" also
```

Whitespace, of course, consists mostly of blanks, tabs, and end-of-line characters.

If we want to remove whitespace only from the beginning of a string, it is better to do it another way. Here we do substitution with the `sub` method. (Here `\s` matches a whitespace character.)

```
str1 = "\t  \nabc  \t\n"
# Remove from beginning of string
str2 = str1.sub(/^\s*/,"")   # "abc  \t\n"
```

However, note that removing whitespace from the end of a string is problematic. If we only remove spaces and tabs, we are fine; but if we try to remove a newline, we run into difficulties. This is because a newline is considered to mark the end of a string; the dollar sign (`$`) will match the earliest newline even if multiline mode is being used. So the naive method of using `$` won't work. Here we show a technique that will work even for newlines; it is unconventional but effective.

```
str3 = str2.reverse.sub(/^[ \t\n]*/,"").reverse
# Reverse the string; remove the whitespace; reverse it again
# Result is "\t  \nabc"
```

Repeating Strings

In Ruby, the multiplication operator (or method) is overloaded to enable repetition of strings. If a string is multiplied by *n*, the result is *n* copies of the original string concatenated together.

```
yell = "Fight! "*3    # "Fight! Fight! Fight! "
ruler = "+" + ("."*4+"5"+"."*4+"+")*3
# "+....5....+....5....+....5....+"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Expanding Variables in Input

This is a case of the *use-mention* distinction that is so common in computer science: Am I *using* this entity or only *mentioning* it? Suppose that a piece of data from outside the program (for example, from user input) is to be treated as containing a variable name or expression. How can we evaluate that expression?

The `eval` method comes to our rescue. Suppose that we want to read in a variable name and tell the user what its value is. The following fragment demonstrates this idea:

```
alpha=10
beta=20
gamma=30
print "Enter a variable name: "
str = gets.chop!
result = eval(str)

puts "#{ str}  = #{ result} "
```

If the user enters alpha, for instance, the program will respond with `alpha = 10`.

However, we will point out a potential danger here. It is conceivable that a malicious user could enter a string specifically designed to run an external program and produce side effects that the programmer never intended or dreamed of. For example, on an UNIX system, one might enter `%x[rm -rf *]` as an input string. When the program evaluated that string, it would recursively remove all the files under the current directory!

For this reason, you must exercise caution when doing an `eval` of a string you didn't build yourselves. (This is particularly true in the case of Web-based software that is accessible by anyone on the Internet.) For example, you could scan the string and verify that it didn't contain backticks, the `%x` notation, the method name `system`, and so on.

Embedding Expressions Within Strings

The `#{ }` notation makes embedding expressions within strings easy. You need not worry about converting, appending, and concatenating; you can interpolate a variable value or other expression at any point in a string.

```
puts "#{ temp_f}  Fahrenheit is #{ temp_c}  Celsius"
puts "The discriminant has the value #{ b*b - 4*a*c} ."
puts "#{ word}  is #{ word.reverse}  spelled backward."
```

Some shortcuts for global, class, and instance variables can be used so that the braces can be dispensed with.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
print "$gvar = #$gvar and ivar = #@ivar."
```

Note that this technique isn't applicable for single-quoted strings (because their contents aren't expanded), but it does work for double-quoted documents and regular expressions.

Parsing Comma-separated Data

Comma-delimited data is common in computing. It is a kind of lowest common denominator of data interchange, which is used (for example) to transfer information between incompatible databases or applications that know no other common format.

We assume here that we have a mixture of strings and numbers and all strings are enclosed in quotes. We further assume that all characters are escaped as necessary (commas and quotes inside strings, and so on).

The problem becomes simple because this data format looks suspiciously like a Ruby array of mixed types. In fact, we can simply add brackets to enclose the whole expression, and we have an array of items.

```
string = gets.chop!
# Suppose we read in a string like this one:
# "Doe, John", 35, 225, "5'10\"", "555-0123"
data = eval("[" + string + "]")   # Convert to array
data.each { |x| puts "Value = #{ x} "}
```

This fragment will produce the following output:

```
sValue = Doe, John
Value = 35
Value = 225
Value = 5' 10"
Value = 555-0123
```

Converting Strings to Numbers (Decimal and Otherwise)

Frequently, we need to capture a number that is embedded in a string. For the simple cases, we can use `to_f` and `to_i` to convert to floating point numbers and integers, respectively. Each will ignore extraneous characters at the end of the string, and each will return a zero if no number is found.

```
num1 = "237".to_i                    # 237
num2 = "50 ways to leave...".to_i    # 50
num3 = "You are number 6".to_i       # 0
num4 = "no number here at all".to_i  # 0

num5 = "3.1416".to_f                 # 3.1416
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
num6 = "0.6931 is ln 2".to_f          # 0.6931
num7 = "ln 2 is 0.6931".to_f          # 0.0
num8 = "nothing to see here".to_f     # 0.0
```

Octal and hexadecimal can similarly be converted with the `oct` and `hex` methods as shown in the following. Signs are optional as with decimal numbers.

```
oct1 = "245".oct                      # 165
oct2 = "245 Days".oct                 # 165
# Leading zeroes are irrelevant.
oct3 = "0245".oct                     # 165
oct4 = "-123".oct                     # -83
# Non-octal digits cause a halt
oct4 = "23789".oct                    # 237

hex1 = "dead".hex                     # 57005
# Uppercase is irrelevant
hex2 = "BEEF".hex                     # 48879
# Non-hex letter/digit causes a halt
hex3 = "beefsteak".hex                # 48879
hex4 = "0x212a".hex                   # 8490
hex5 = "unhexed".hex                  # 0
```

There is no `bin` method to convert from binary, but you can write your own (see Listing 2.2). Notice that it follows all the same rules of behavior as `oct` and `hex`.

Listing 2.2. Converting from Binary

```
class String

  def bin
    val = self.strip
    pattern = /^([+-]?)(0b)?([01]+)(.*)$/
        parts = pattern.match(val)
    return 0 if not parts
    sign = parts[1]
    num  = parts[3]
    eval(sign+"0b"+num)
  end

end

a = "10011001".bin       # 153
b = "0b10011001".bin     # 153
c = "0B1001001".bin      # 0
d = "nothing".bin        # 0
e = "0b100121001".bin    # 9
```

Encoding and Decoding `rot13` Text

The `rot13` method is perhaps the weakest form of encryption known to humankind. Its historical use is simply to prevent people from accidentally reading a piece of text. It is

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

commonly seen in Usenet; for example, a joke that might be considered offensive might be encoded in `rot13`, or you could post the entire plot of *Star Wars: Episode II* the day before the premiere.

The encoding method consists simply of rotating a string through the alphabet, so *A* becomes *N, B* becomes *O,* and so on. Lowercase letters are rotated in the same way; digits, punctuation, and other characters are ignored. Because 13 is half of 26 (the size of our alphabet), the function is its own inverse; applying it a second time will decrypt it.

The following is an implementation as a method added to the `String` class. We present it without further comment.

```
class String

  def rot13
    self.tr("A-Ma-mN-Zn-z","N-Zn-zA-Ma-m")
  end

end

joke = "Y2K bug"
joke13 = joke.rot13    # "L2X oht"

episode2 = "Fcbvyre: Nanxva qbrfa'g trg xvyyrq."
puts episode2.rot13
```

Obscuring Strings

Sometimes we don't want strings to be immediately legible. For example, passwords shouldn't be stored in plain text, no matter how tight the file permissions are.

The standard method `crypt` uses the standard function of the same name in order to DES-encrypt a string. It takes a "salt" value as a parameter (similar to the seed value for a random number generator).

A trivial application for this is shown in the following, where we ask for a password that Tolkien fans should know.

```
coded = "hfCghHIE5LAM."

puts "Speak, friend, and enter!"

print "Password: "
password = gets.chop

if password.crypt("hf") == coded
  puts "Welcome!"
else
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
  puts "What are you, an orc?"
end
```

There are other conceivable uses for hiding strings. For example, we sometimes want to hide strings inside a file so that they aren't easily read. Even a binary file can have readable portions easily extracted by the UNIX strings utility or the equivalent, but a DES encryption will stop all but the most determined crackers.

It is worth noting that you should never rely on encryption of this nature for a server-side Web application. That is because a password entered on a Web form is still transmitted over the Internet in plaintext. In a case like this, the easiest security measure is the *Secure Sockets Layer (SSL)*. Of course, you could still use encryption on the server side, but for a different reason—to protect the password as it is stored rather than during transmission.

Counting Characters in Strings

The `count` method will count the number of occurrences of any set of specified characters.

```
s1 = "abracadabra"
a  = s1.count("c")      # 1
b  = s1.count("bdr")    # 5
```

The `string` parameter is similar to a very simple regular expression. If it starts with a caret, the list is negated.

```
c = s1.count("^a")      # 6
d = s1.count("^bdr")    # 6
```

A hyphen indicates a range of characters.

```
e = s1.count("a-d")     # 9
f = s1.count("^a-d")    # 2
```

Reversing a String

A string can be reversed very simply with the `reverse` method (or its in-place counterpart `reverse!`).

```
s1 = "Star Trek"
s2 = s1.reverse         # "kerT ratS"
s1.reverse!             # s1 is now "kerT ratS"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Suppose that you have a sentence and need to reverse the word order (rather than character order). Use the `%w` operator to make it an array of words, reverse the array, and then use join to rejoin them.

```
words = %w( how now brown cow )
# ["how", "now", "brown", "cow"]
words.reverse.join(" ")
# "cow brown now how"
```

This can be generalized with `String#split`, which allows you to divide the words based on your own pattern.

```
phrase = "Now here's a sentence"
phrase.split(" ").reverse.join(" ")
# "sentence a here's Now"
```

Removing Duplicate Characters

Runs of duplicate characters can be removed using the `squeeze` method.

```
s1 = "bookkeeper"
s2 = s1.squeeze         # "bokeper"
s3 = "Hello..."
s4 = s3.squeeze         # "Helo."
```

If a parameter is specified, only those characters will be squeezed.

```
s5 = s3.squeeze(".")    # "Hello."
```

This parameter follows the same rules as the one for the count method (see "Counting Characters in Strings"); that is, it understands the hyphen and the caret.

There is also a `squeeze!` method.

Removing Specific Characters from Within a String

The `delete` method will remove characters from a string if they appear in the list of characters passed as a parameter.

```
s1 = "To be, or not to be"
s2 = s1.delete("b")            # "To e, or not to e"
s3 = "Veni, vidi, vici!"
s4 = s3.delete(",!")           # "Veni vidi vici"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

This parameter follows the same rules as the one for the `count` method (see "Counting Characters in Strings"); that is, it understands the hyphen and the caret.

There is also a `delete!` method.

Printing Special Characters

The `dump` method will provide explicit printable representations of characters that might ordinarily be invisible or print differently.

```
s1 = "Listen" << 7 << 7 << 7   # Add three ASCII BEL characters
puts s1.dump                   # Prints: Listen\007\007\007
s2 = "abc\t\tdef\tghi\n\n"
puts s2.dump                   # Prints: abc\t\tdef\tghi\n\n
s3 = "Double quote: \""
puts s3.dump                   # Prints: Double quote: \"
```

Generating Successive Strings

On rare occasions we might want to find the successor value for a string; for example, the successor for `"aaa"` is `"aab"` (then `"aac"`, `"aad"`, and so on).

Ruby provides the method `succ` for this purpose.

```
droid = "R2D2"
improved  = droid.succ         # "R2D3"
pill  = "Vitamin B"
pill2 = pill.succ              # "Vitamin C"
```

We don't recommend the use of this feature unless the values are predictable and reasonable. If you start with a string that is esoteric enough, you will eventually get strange and surprising results.

There is also an `upto` method that will apply `succ` repeatedly in a loop until the desired final value is reached.

```
"Files, A".upto "Files, X" do |letter|
  puts "Opening: #{ letter} "
end

# Produces 24 lines of output
```

Again, we stress that this isn't used very frequently, and you use it at your own risk. Also we want to point out that there is no corresponding predecessor function at the time of this writing.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Calculate the Levenstein Distance Between Two Strings

The concept of distance between strings is important in inductive learning (AI), cryptography, proteins research, and in other areas.

The Levenstein distance (see Listing 2.3) is the minimum number of modifications needed to change one string into another, using three basic modification operations: `del` (deletion), `ins` (insertion), and `sub` (substitution). A substitution is also considered to be a combination of a deletion and insertion (`indel`). There are various approaches to this, but we will avoid getting too technical. Suffice it to say that this Ruby implementation allows you to provide optional parameters to set the cost for the three types of modification operations, and defaults to a single `indel` cost basis (cost of insertion=cost of deletion).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 2.3. Levenstein Distance

```
class String

  def levenstein(other, ins=2, del=2, sub=1)
    # ins, del, sub are weighted costs
    return nil if self.nil?
    return nil if other.nil?
    dm = []        # distance matrix

    # Initialize first row values
    dm[0] = (0..self.length).collect {  |i| i * ins }
    fill = [0] * (self.length - 1)

    # Initialize first column values
    for i in 1..other.length
      dm[i] = [i * del, fill.flatten]
    end

    # populate matrix
    for i in 1..other.length
      for j in 1..self.length
        # critical comparison
        dm[i][j] = [
             dm[i-1][j-1] +
               (self[j-1] == other[i-1] ? 0 : sub),
             dm[i][j-1] + ins,
             dm[i-1][j] + del
           ].min
      end
    end


    # The last value in matrix is the
    # Levenstein distance between the strings
    dm[other.length][self.length]
  end

end


s1 = "ACUGAUGUGA"
s2 = "AUGGAA"
d1 = s1.levenstein(s2)    # 9


s3 = "pennsylvania"
s4 = "pencilvaneya"
d2 = s3.levenstein(s4)    # 7


s5 = "abcd"
s6 = "abcd"
d3 = s5.levenstein(s6)    # 0
```

Now that we have the Levenstein distance defined, it's conceivable that we could define a `similar?` method, giving it a threshold for similarity.

```
class String

  def similar?(other, thresh=2)
    if self.levenstein(other) < thresh
      true
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
      else
        false
        end
      end

end


if "polarity".similar?("hilarity")
  puts "Electricity is funny!"
end
```

Of course, it would also be possible to pass in the three weighted costs to the `similar?` method so that they could in turn be passed into the Levenstein method. We have omitted these for simplicity.

Using Strings as Stacks and Queues

These routines make it possible to treat a string as a stack or a queue (see Listing 2.4), adding the operations `shift`, `unshift`, `push`, `pop`, `rotate_left`, and `rotate_right`. The operations are implemented both at the character and the word level. These have proved useful in one or two programs that we have written, and they might be useful to you also. Use your imagination.

There might be some confusion as to what is returned by each method. In the case of a retrieving operation such as `pop` or `shift`, the return value is the item that was retrieved. In a storing operation such as `push` or `unshift`, the return value is the new string. All rotate operations return the value of the new string. And we will state the obvious: Every one of these operations modifies its receiver, although none of them is marked with an exclamation point as suffix.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 2.4. String as Queues

```
class String

  def shift
    # Removes first character from self and
    #   returns it, changing self
    return nil if self.empty?
    item=self[0]
    self.sub!(/^./,"")
    return nil if item.nil?
    item.chr
  end

  def unshift(other)
    # Adds last character of provided string to
    #   front of self
    newself = other.to_s.dup.pop.to_s + self
    self.replace(newself)
  end

  def pop
    # Pops last character off self and
    #   returns it, changing self
    return nil if self.empty?
    item=self[-1]
    self.chop!
    return nil if item.nil?
    item.chr
  end

  def push(other)
    # Pushes first character of provided
    #   string onto end of self
    newself = self + other.to_s.dup.shift.to_s
    self.replace(newself)
  end

  def rotate_left(n=1)
    n=1 unless n.kind_of? Integer
    n.times do
      char = self.shift
      self.push(char)
    end
    self
  end

  def rotate_right(n=1)
    n=1 unless n.kind_of? Integer
    n.times do
      char = self.pop
      self.unshift(char)
    end
    self
  end

  @@first_word_re = /^(\w+\W*)/
  @@last_word_re = /(\w+\W*)$/
  def shift_word
    # Shifts first word off of self
    #   and returns; changes self
    return nil if self.empty?
    self=~@@first_word_re
    newself= $' || ""       # $' is POSTMATCH
    self.replace(newself) unless $'.nil?
    $1
  end
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
    def unshift_word(other)
      # Adds provided string to front of self
      newself = other.to_s + self
      self.replace(newself)
    end

    def pop_word
      # Pops and returns last word off
      # self; changes self
      return nil if self.empty?
      self=~@@last_word_re
      newself= $` || ""       # $` is PREMATCH
      self.replace(newself) unless $`.nil?
      $1
    end

    def push_word(other)
      # Pushes provided string onto end of self
      newself = self + other.to_s
      self.replace(newself)
    end

    def rotate_word_left
      word = self.shift_word
      self.push_word(word)
    end

    def rotate_word_right
      word = self.pop_word
      self.unshift_word(word)
    end

    alias rotate_Left rotate_word_left
    alias rotate_Right rotate_word_right
  end

  # ------------

  str = "Hello there"
  puts str.rotate_left                # "ello thereH"
  puts str.pop                        # "H"
  puts str.shift                     # "e"
  puts str.rotate_right                # "ello ther"
  puts str.unshift("H")                # "Hello ther"
  puts str.push("e")                 # "Hello there"

  puts str.push_word(", pal!")        # "Hello there, pal!"
  puts str.rotate_Left                # "there, pal!Hello "

  puts str.pop_word                 # str is "there, pal!"
                                      # result is "Hello "

  puts str.shift_word                # str is "pal!"
                                      # result is "there, "

  puts str.unshift_word("Hi there, ") # "Hi there, pal!"
  puts str.rotate_Right                # "pal!Hi there, "
  puts str.rotate_left(4)             # "Hi there, pal!"

  puts "Trying again..."
  str = "pal! Hi there, "
  puts str.rotate_left(5)             # "Hi there, pal!"
```

Note that the `[]` operator with a range might be used to gain a window onto a string that is being rotated.

```
str = ".....duck....*...*..*..........*......*..."
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
loop do
  print str.rotate_left[0..7],"\r"}
end

# speed reading
string="See Bill run. Run Bill run! See Jane sit. Jane sees Bill."
loop{ print string.rotate_Left[0..4],"\r"}
```

Creating an Abbreviation or Acronym

Suppose that we have a string and we want to create an abbreviation from the initial letters of each word in it. The code fragment shown in Listing 2.5 accomplishes that. We have added a threshold value such that any word fewer than that number of letters will be ignored. The threshold value defaults to zero, including all words.

Note that this uses the `shift_word` function, defined in "Using Strings as Stacks and Queues."

Listing 2.5. Acronym Creator

```
class String

  def acronym(thresh=0)
    acro=""
    str=self.dup.strip
    while !str.nil? && !str.empty?
      word = str.shift_word
      if word.length >= thresh
        acro += word.strip[0,1].to_s.upcase
      end
    end
    acro
  end

end


s1 = "Same old, same old"
puts s1.acronym       #  "SOSO"

s2 = "three-letter abbreviation"
puts s2.acronym       #  "TLA"

s3 = "light amplification by stimulated emission of radiation"
puts s3.acronym       #  "LABSEOR"
puts s3.acronym(3)    #  "LASER"
```

Here is a less readable but perhaps more instructive version of the same method.

```
def acro(thresh=0)
  self.split.find_all { |w| w.length > thresh } .
       collect { |w| w[0,1].upcase} .join
end
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Don't fail to notice the trailing dot on the `find_all` call.

Encoding and Decoding Base64 Strings

Base64 is frequently used to convert machine-readable data into a text form with no special characters in it. For example, newsgroups that handle binary files, such as program executables, frequently will use base64.

The easiest way to do a base64 encode/decode is to use the built-in features of Ruby. The `Array` class has a `pack` method that will return a base64 string (given the parameter `"m"`). The `String` class has a method `unpack` that will likewise unpack the string (decoding the base64).

```
str = "\007\007\002\abdce"

new_string = [str].pack("m")          # "BwcCB2JkY2U="
original   =  new_string.unpack("m") # ["\a\a\002\abdce"]
```

Note that an array is returned by `unpack`.

Encoding and Decoding Strings (uuencode/uudecode)

The *uu* in these names means UNIX-to-UNIX. The `uuencode` and `uudecode` utilities are a time-honored way of exchanging data in text form (similar to the way base64 is used).

```
str = "\007\007\002\abdce" new_string = [str].pack("u") #
'(!P<"!V)D8V4`' original = new_string.unpack("u") # ["\a\a\002\abdce"]
```

Note that an array is returned by `unpack`.

Expanding and Compressing Tab Characters

Occasionally we have a string with tabs in it and we want to convert them to spaces (or vice versa). The two methods shown in Listing 2.6 will do these operations.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 2.6. Convert Tabs to Spaces

```
class String

  def detab(ts=8)
    str = self.dup
    while (leftmost = str.index("\t")) != nil
      space = " "*(ts-(leftmost%ts))
      str[leftmost]=space
    end
    str
  end

  def entab(ts=8)
    str = self.detab
    areas = str.length/ts
    newstr = ""
    for a in 0..areas
      temp = str[a*ts..a*ts+ts-1]
      if temp.size==ts
        if temp =~ / +/
          match=Regexp.last_match[0]
          endmatch = Regexp.new(match+"$")
          if match.length>1
           temp.sub!(endmatch,"\t")
          end
        end
      end
      newstr += temp
    end
    newstr
  end

end

foo = "This       is        only   a     test.          "

puts foo
puts foo.entab(4)
puts foo.entab(4).dump
```

Note that this code isn't smart enough to handle backspaces.

Wrapping Lines of Text

Occasionally, we might want to take long text lines and print them within margins of our own choosing. Listing 2.7 accomplishes this, splitting only on word boundaries and honoring tabs (but not honoring backspaces or preserving tabs).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 2.7. Line Wrap

```
str = "When in the Course of human events it becomes necessary\n
for one people to dissolve the political bands which have\n
connected them with another, and to assume among the powers\n
of the earth the separate and equal station to which the Laws\n
of Nature and of Nature's God entitle them, a decent respect\n
for the opinions of mankind requires that they should declare\n
the causes which impel them to the separation."
max = 20

line = 0
out = [""]

input = str.gsub("\n"," ")
# input.detab!

while input != ""
   word = input.shift_word
   break if not word
   if out[line].length + word.length > max
     out[line].squeeze!(" ")
     line += 1
     out[line] = ""
   end
   out[line].push_word(word)
end

out.each { |line| puts line}   # Prints 24 very short lines
```

Regular Expressions

I would choose

To lead him in a maze along the patterned paths...

—Amy Lowell, "Patterns"

The power of regular expressions as a computing tool has often been underestimated. From their earliest theoretical beginnings in the 1940s, they found their way onto computer systems in the 1960s and thence into various tools in the UNIX operating system. In the 1990s, the popularity of Perl helped make regular expressions a household item rather than the esoteric domain of bearded gurus.

The beauty of regular expressions is that everything in our experience can be understood in terms of patterns. Where there are patterns that we can describe, we can detect matches; we can find the bits of reality that correspond to those matches; and we can replace those bits with others of our own choosing.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Escaping Special Characters

The class method `Regexp.escape` will escape any special characters that are used in regular expressions. Such characters include the asterisk, question mark, and brackets.

```
str1 = "[*?]"
str2 = Regexp.escape(str1)  # "\[\*\?\]"
```

The method `Regexp.quote` is an alias.

Compiling Regular Expressions

Regular expressions can be compiled using the class method `Regexp.compile`, which is really only a synonym for `Regexp.new`. The first parameter is required and can be a string or a regex. (Note that if the parameter is a regex with options, the options won't carry over into the newly compiled regex.)

```
pat1 = Regexp.compile("^foo.*")  # /^foo.*/
pat2 = Regexp.compile(/bar$/i)   # /bar/ (i not propagated)
```

The second parameter, if present, is normally a bitwise `OR` of any of the following constants: `Regexp::EXTENDED`, `Regexp::IGNORECASE`, and `Regexp::MULTILINE`. Additionally, any non-`nil` value will have the result of making the regex not case sensitive; we don't recommend this practice.

```
options = Regexp::MULTILINE || Regexp::IGNORECASE
pat3 = Regexp.compile("^foo", options)
pat4 = Regexp.compile(/bar/, Regexp::IGNORECASE)
```

The third parameter, if specified, is the language parameter, which enables multibyte character support. It can take any of the following four string values:

```
"N" or "n" means None
"E" or "e" means EUC
"S" or "s" means Shift-JIS
"U" or "u" means UTF-8
```

Accessing Backreferences

The class method `Regexp.last_match` will return an object of class `MatchData` (as will the instance method `match`). This object has instance methods that enable the programmer to access backreferences.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The `MatchData` object is manipulated with a bracket notation as though it were an array of matches. The special element `0` contains the text of the entire matched string. Thereafter, element `n` refers to the nth match.

```
pat = /(.+[aiu])(.+[aiu])(.+[aiu])(.+[aiu])/i
# Four identical groups in this pattern
refs = pat.match("Fujiyama")
# refs is now: ["Fujiyama","Fu","ji","ya","ma"]
x = refs[1]
y = refs[2..3]
refs.to_a.each { |x| print "#{ x} \n"}
```

Note that the object `refs` isn't a true array. Thus, when we want to treat it as one by using the iterator `each`, we must use `to_a` (as shown previously) to convert it to an array.

We can use more than one technique to locate a matched substring within the original string. The methods `begin` and `end` will return the beginning and ending offsets of a match. (It is important to realize that the ending offset is really the index of the next character after the match.)

```
str = "alpha beta gamma delta epsilon"
#      0....5....0....5....0....5....
#      (for  your counting convenience)

pat = /(b[^ ]+ )(g[^ ]+ )(d[^ ]+ )/
# Three words, each one a single match
refs = pat.match(str)


# "beta "
p1 = refs.begin(1)          # 6
p2 = refs.end(1)            # 11
# "gamma "
p3 = refs.begin(2)          # 11
p4 = refs.end(2)            # 17
# "delta "
p5 = refs.begin(3)          # 17
p6 = refs.end(3)            # 23
# "beta gamma delta"
p7 = refs.begin(0)          # 6
p8 = refs.end(0)            # 23
```

Similarly, the `offset` method will return an array of two numbers, which are the beginning and ending offsets of that match. To continue the preceding example:

```
range0 = refs.offset(0)     # [6,23]
range1 = refs.offset(1)     # [6,11]
range2 = refs.offset(2)     # [11,17]
range3 = refs.offset(3)     # [17,23]
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The portions of the string `before` and `after` the matched substring can be retrieved by the instance methods `pre_match` and `post_match`, respectively. To continue the preceding example:

```
before = refs.pre_match    # "alpha "
after  = refs.post_match   # "epsilon"
```

Using Character Classes

Ruby regular expression might contain references to character classes, which are basically named patterns (of the form `[[:name:]]`). For example, `[[:digit:]]` means the same as `[0-9]` in a pattern. In most cases, this turns out to be shorthand.

Some others are `[[:print:]]` (printable characters) and `[[:alpha:]]` (alphabetic characters).

```
s1 = "abc\007def"
/[[:print:]]*/.match(s1)
m1 = Regexp::last_match[0]              # "abc"

s2 = "1234def"
/[[:digit:]]*/.match(s2)
m2 = Regexp::last_match[0]              # "1234"

/[[:digit:]]+[[:alpha:]]/.match(s2)
m3 = Regexp::last_match[0]              # "1234d"
```

Treating Newline as a Character

Ordinarily a dot will match any character except a newline. When the `m` (multiline) modifier is used, a newline will be matched by a dot. The same is true when the `Regexp::MULTILINE` option is used in creating a regex.

```
str = "Rubies are red\nAnd violets are blue.\n"
pat1 = /red./
pat2 = /red./m

str =~ pat1       # false
str =~ pat2       # true
```

Matching an IP Address

Suppose that we want to determine whether a string is a valid IPv4 address. The standard form of such an address is a *dotted quad* or *dotted decimal* string. This is, in other words, four decimal numbers separated by periods, each number ranging from 0 to 255.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The pattern given here will do the trick (with a few exceptions such as `127.1`). We break the pattern up a little just for readability. Note that the `\d` symbol is double escaped so that the slash in the string will get passed on to the regex.

```
num = "(\\d|[01]?\ \ d\ \ d|2[0-4]\\d|25[0-5])"
pat = "^#{ num} \.#{ num} \.#{ num} \.#{ num} $"
ip_pat = Regexp.new(pat)

ip1 = "9.53.97.102"

if ip1 =~ ip_pat                # Prints "yes"
  puts "yes"
else
  puts "no"
end
```

IPv6 addresses aren't in widespread use yet, but we include them for completeness. These consist of eight colon-separated 16-bit hex numbers with zeroes suppressed.

```
num = "[0-9A-Fa-f]{ 0,4} "
pat = "^" + "#{ num} :"*7 + "#{ num} $"
ipv6_pat = Regexp.new(pat)

v6ip = "abcd::1324:ea54::dead::beef"

if v6ip =~ ipv6_pat             # Prints "yes"
  puts "yes"
else
  puts "no"
end
```

Matching a Keyword-Value Pair

Occasionally, we want to work with strings of the form *"attribute=value"* (for example, when we parse some kind of configuration file for an application).

This code fragment will extract the keyword and the value. The assumptions are that the keyword or attribute is a single word; the value extends to the end of the line; and the equal sign can be surrounded by whitespace.

```
pat = /(\w+)\s*=\s*(.*?)$/
str = "color = blue"

matches = pat.match(str)

puts matches[1]             # "color"
puts matches[2]             # "blue"
```

For additional information see the section "Adding a Keyword-Value String to a Hash."

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Matching Roman Numerals

Here we match against a complex pattern to determine whether a string is a valid Roman number (up to decimal 3999). As before, the pattern is broken up into parts for readability.

```
rom1 = "m{ 0,3} "
rom2 = "(d?c{ 0,3} |c[dm])"
rom3 = "(l?x{ 0,3} |x[lc])"
rom4 = "(v?i{ 0,3} |i[vx])"
rom_pat = "^#{ rom1} #{ rom2} #{ rom3} #{ rom4} $"

roman = Regexp.new(rom_pat, Regexp::IGNORECASE)

year1985 = "MCMLXXXV"

if year1985 =~ roman      # Prints "yes"
  puts "yes"
else
  puts "no"
end
```

Matching Numeric Constants

A simple decimal integer is the easiest number to match. It has an optional sign and consists thereafter of digits (except that Ruby allows an underscore as a digit separator). Note that the first digit shouldn't be a zero; then it would be interpreted as an octal constant.

```
int_pat = /^[+-]?[1-9][\d_]*/
```

Integer constants in other bases are similar. Note that the hex and binary patterns are not case sensitive because they contain at least one letter.

```
hex_pat = /^[+-]?0x[\da-f_]+$/i
oct_pat = /^[+-]?0[0-7_]+$/
bin_pat = /^[+-]?0b[01_]+/i
```

A normal floating-point constant is a little tricky; the number sequences on each side of the decimal point are optional, but one or the other must be included.

```
float_pat = /^(\d[\d_]*)*\.[\d_]*$/
```

Finally, scientific notation builds on the ordinary floating-point pattern.

```
sci_pat = /^(\d[\d_]*)?\.[\d_]*(e[+-]?)?(_*\d[\d_]*)$/i
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

These patterns can be useful if, for instance, you have a string and you want to verify its validity as a number before trying to convert it.

Matching a Date/Time String

Suppose that we want to match a date/time in the form mm/dd/yy hh:mm:ss. This pattern is a good first attempt: `datetime_re=/(\d\d)\/(\d\d)\/(\d\d) (\d\d):(\d\d):(\d\d)/`.

However, that will also match invalid date/times, and miss valid ones. A pickier pattern is shown in Listing 2.8.

Listing 2.8. Matching Date/Time Strings

```
class String

  def scan_datetime(flag=2)
    datetime_re=/((\d\d)\/(\d\d)\/(\d\d) (\d\d):(\d\d):(\d\d))/

    month_re=/(0?[1-9]|1[0-2])/
    # 01 to 09 or 1 to 9 or 10-12
    day_re=/([0-2]?[1-9]|[1-3][01])/
    # 1-9 or 01-09 or 11-19 or 21-29 or 10,11,20,21,30,31
     year_re=/(\d\d)/
     # 00-99
     hour_re=/([01]?[1-9]|[12][0-4])/
     # 1-9 or 00-09 or 11-19 or 10-14 or 20-24
     minute_re=/([0-5]\d)/
     # 00-59, both digit required
     second_re=/(:[0-6]\d)?/
     # leap seconds ;-) both digits required if present

     date_re=/(#{ month_re.source} \/#{ day_re.source} \
   /#{ year_re.source} )/
     time_re=/(#{ hour_re.source}
   :#{ minute_re.source} #{ second_re.source} )/

     datetime_re2 = /(#{ date_re.source}  #{ time_re.source} )/

     if flag==2
       self.scan(datetime_re2)    # returns arrays
     else
       self.scan(datetime_re)
     end
   end
end


str="Recorded on 11/18/00 20:31:00, viewed 11/18/00 8:31 PM "
str.scan_datetime
# [ ["11/18/00 20:31:00", "11", "18", "00", "20", "31", "00"] ]
str.scan_datetime(2)
# [ ["11/18/00 20:31:00", "11/18/00", "11", "18", "00",
#  "20:31:00", "20", "31", ":00"],
# ["11/18/00 8:31", "11/18/00", "11", "18", "00", "8:31",
#  "8", "31", nil]  ]
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Detecting Doubled Words in Text

Here, we implement the famous double-word detector. Typing the same word twice in succession is one of the most common typing errors. The code we show here will detect instances of that occurrence.

```
double_re = /\b(['A-Z]+) +\1\b/i

str="There's there's the the pattern."
str.scan(double_re)  #  [["There's"],["the"]]
```

Note that the trailing `i` in the regex is for not case sensitive matching. There is an array for each grouping, hence the resulting array of arrays.

Matching All-caps Words

This one is simple if we assume no numerics, underscores, and so on.

```
allcaps = /\b[A-Z]+\b/

string = "This is ALL CAPS"
string[allcaps]                #  "ALL"
```

Suppose that you want to simply extract every word in all-caps.

```
string.scan(allcaps)   #  ["ALL", "CAPS"]
```

If we wanted, we could extend this concept to include Ruby identifiers and similar items.

Numbers

> *On two occasions I have been asked [by members of Parliament], 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.*
>
> —Charles Babbage

Numeric data is the original data type, the native language of the computer. We would be hard-pressed to find areas of our experience in which numbers aren't applicable. It doesn't matter whether you're an accountant or an aeronautical engineer; you can't survive without numbers. We present here a few ways to process, manipulate, convert, and analyze numeric data.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Performing Bit-level Operations on Numbers

Occasionally, we might need to operate on a `Fixnum` as a binary entity. This is less common in application level programming, but the need still arises.

Ruby has a relatively full set of capabilities in this area. For convenience, numeric constants can be expressed in binary, octal, or hexadecimal. The usual operators `AND`, `OR`, `XOR`, and `NOT` are expressed by the Ruby operators `&`, `|`, `^`, and `~`, respectively.

```
x = 0377           # Octal  (decimal 255)
y = 0b00100110     # Binary (decimal  38)
z = 0xBEEF         # Hex    (decimal 48879)

a = x | z          # 48895 (bitwise OR)
b = x & z          #   239 (bitwise AND)
c = x ^ z          # 48656 (bitwise XOR)
d = ~ y            #   -39 (negation or 1's complement)
```

The instance method `size` can be used to determine the wordsize of the specific architecture on which the program is running.

```
bytes = 1.size     # Returns 4 for one particular machine
```

There are left-shift and right-shift operators (<< and >>, respectively). These are logical shift operations; they don't disturb the sign bit (although >> does propagate it).

```
x = 8
y = -8

a = x >> 2         # 2
b = y >> 2         # -2
c = x << 2         # 32
d = y << 2         # -32
```

Of course, anything shifted far enough to result in a zero value will lose the sign bit because `-0` is merely `0`.

Brackets can be used to treat numbers as arrays of bits. The 0th bit is the least significant bit regardless of the bit order (endianness) of the architecture.

```
x = 5              # Same as 0b0101
a = x[0]           # 1
b = x[1]           # 0
c = x[2]           # 1
d = x[3]           # 0
# Etc.             # 0
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

It isn't possible to assign bits using this notation (because a `Fixnum` is stored as an immediate value rather than an object reference). However, you can always fake it by left-shifting a `1` to the specified bit position and then doing an `OR` or `AND` operation.

```
# We can't do x[3] = 1
# but we can do:
x |= (1<<3)
# We can't do x[4] = 0
# but we can do:
x &= ~(1<<4)
```

Finding Cube Roots, Fourth Roots, and So On

Ruby has a built-in square root function (`Math.sqrt`) because that function is so commonly used. But what if you need higher-level roots? If you remember your math, this is easy.

One way is to use logarithms. Recall that *e* to the *x* is the inverse of the natural log of *x*. When multiplying numbers, that is equivalent to adding their logarithms.

```
x = 531441
cuberoot = Math.exp(Math.log(x)/3.0)      # 81.0
fourthroot = Math.exp(Math.log(x)/4.0)    # 27.0
```

However, it is just as easy and perhaps clearer to use fractions with an exponentiation operator (which can take any integer or floating-point value).

```
y = 4096
cuberoot = y**(1.0/3.0)       # 16.0
fourthroot = y**(1.0/4.0)     # 8.0
fourthroot = sqrt(sqrt(y))    # 8.0 (same thing)
twelfthroot = y**(1.0/12.0)   # 2.0
```

Note that in all of these examples, we have used floating-point numbers when dividing (to avoid truncation to an integer).

Rounding Floating-Point Values

If you want to round a floating-point value to an integer, the method `round` will do the trick.

```
pi = 3.14159
new_pi = pi.round   # 3
temp = -47.6
temp2 = temp.round  # -48
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Sometimes we want to round not to an integer, but to a specific number of decimal places. In this case, we could use `sprintf` (which knows how to round) and `eval` to do this.

```
pi = 3.1415926535
pi6 = eval(sprintf("%8.6f",pi))  # 3.141593
pi5 = eval(sprintf("%8.5f",pi))  # 3.14159
pi4 = eval(sprintf("%8.4f",pi))  # 3.1416
```

Of course, this is somewhat ugly. Let's encapsulate this behavior in a method that we'll add to `Float`.

```
class Float

  def roundf(places)
    temp = self.to_s.length
    sprintf("%#{ temp} .#{ places} f",self).to_f
  end

end
```

Occasionally, we follow a different rule in rounding to integers. The tradition of rounding *n*+0.5 upward results in slight inaccuracies at times; after all, *n*+0.5 is no closer to *n*+1 than it is to *n*. So there is an alternative tradition that rounds to the nearest even number in the case of 0.5 as a fractional part. If we wanted to do this, we might extend the `Float` class with a method of our own called `round2`, as shown here.

```
class Float

  def round2
    whole = self.floor
    fraction = self - whole
    if fraction == 0.5
      if (whole % 2) == 0
        whole
      else
        whole+1
      end
    else
      self.round
    end
  end

end

a = (33.4).round2  # 33
b = (33.5).round2  # 34
c = (33.6).round2  # 34
d = (34.4).round2  # 34
e = (34.5).round2  # 34
f = (34.6).round2  # 35
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Obviously `round2` differs from `round` only when the fractional part is exactly 0.5; note that 0.5 can be represented perfectly in binary, by the way. What is less obvious is that this method will work fine for negative numbers also. (Try it.) Also note that the parentheses used here aren't actually necessary, but rather they are used for readability.

Now, what if we wanted to round to a number of decimal places, but we wanted to use the even rounding method? In this case, we could add a method called `roundf2` to `Float`.

```
class Float

  # round2 definition as before

  def roundf2(places)
    shift = 10**places
    (self * shift).round2 / shift.to_f
  end

end

a = 6.125
b = 6.135
x = a.roundf(2)    # 6.12
y = b.roundf(2)    # 6.12
```

The code shown here (`roundf` and `roundf2`) has certain limitations, in that a large floating-point number will naturally cause problems when it is multiplied by a large power of ten. For these occurrences, error-checking should be added.

Formatting Numbers for Output

To output numbers in a specific format, you can use the `printf` method in the Kernel module. It is virtually identical to its C counterpart. For more information, see the documentation for the `printf` method.

```
x = 345.6789
i = 123
printf("x = %6.2f\n", x)     # x = 345.68
printf("x = %9.2e\n", x)     # x = 3.457e+02
printf("i = %5d\n", i)       # i =   123
printf("i = %05d\n", i)      # i = 00123
printf("i = %-5d\n", i)      # i = 123
```

To store a result in a string rather than printing it immediately, `sprintf` can be used in much the same way. This method returns a string.

```
str = sprintf("%5.1f",x)     # "345.7"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Finally, the String class has a `%` method that performs this same task. The `%` method has a format string as a receiver; it takes a single argument (or an array of values) and returns a string.

```
# Usage is 'format % value'
str = "%5.1" % x            # "345.7"
str = "%6.2, %05d" % [x,i]  # "345.68, 00123"
```

Working with Large Integers

> *The control of large numbers is possible, and like unto that of small numbers, if we subdivide them.*
>
> —Sun Tzu

In the event it becomes necessary, Ruby programmers can work with integers of arbitrary size. The transition from a `Fixnum` to a `Bignum` is handled automatically and transparently.

```
num1 = 1000000           # One million (10**6)
num2 = num1*num1         # One trillion (10**12)
puts num1                # 1000000
puts num1.type           # Fixnum
puts num2                # 1000000000000
puts num2.type           # Bignum
```

The size of a `Fixnum` will vary from one architecture to another.

Swapping Two Values

This item isn't strictly concerned with numeric data, but we did want to mention it somewhere. In many languages, swapping or exchanging two values requires a temporary variable; in Ruby (as in Perl and some others), this isn't needed. The following statement

```
x, y = y, x
```

will exchange `x` and `y` using multiple assignment. Note, of course, that in the case of numbers, we are exchanging the actual values. In the case of most other objects, we are only swapping what amounts to a pointer or reference—that is, changing which variable refers to which object.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Determining the Architecture's Byte Order

It is an interesting fact of the computing world that we cannot all agree on the order in which binary data ought to be stored. Is the most significant bit stored at the higher-numbered address or the lower? When we shove a message over a wire, do we send the most significant bit first, or the least significant?

Believe it or not, it's not entirely arbitrary. There are good arguments on both sides, which we won't delve into here.

For at least 20 years, the terms "little-endian" and "big-endian" have been applied to the two extreme opposites. These apparently were first used by Danny Cohen; refer to his classic article "On Holy Wars and a Plea for Peace" (*IEEE Computer*, October 1981). The actual terms are derived from the novel *Gulliver's Travels* by Jonathan Swift.

Most of the time, we don't care what byte order our architecture uses. But what if we do need to know?

Here's one little method that will determine this for us. It will return a string that is `LITTLE`, `BIG`, or `OTHER`. It depends on the fact that the `l` directive packs in native mode and the `N` directive unpacks in network order (or big-endian).

```
def endianness
  num=0x12345678
  little = "78563412"
  big    = "12345678"
  native = [num].pack('l')
  netunpack = native.unpack('N')[0]
  str = "%8x" % netunpack
  case str
    when little
      "LITTLE"
    when big
      "BIG"
    else
      "OTHER"
  end
end

puts endianness   # In this case, prints "LITTLE"
```

This technique might come in handy if, for example, you are working with binary data (such as scanned image data) imported from another system.

Calculating the MD5 Hash of a String

The MD5 message-digest algorithm produces a 128-bit fingerprint or *message digest* of a message of arbitrary length. This is in the form of a hash, so the encryption is one way and

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

doesn't allow for the discovery of the original message from the digest. Ruby has an extension for a class to implement MD5; for those interested in the source code, it's in the ext/md5 directory of the standard Ruby distribution.

There are two class methods, `new` and `md5`, to create a new MD5 object. There is really no difference in them.

```
require 'md5'
cryptic = MD5.md5
password = MD5.new
```

There are four instance methods: `clone`, `digest`, `hexdigest`, and `update`. The `clone` method simply copies the object; `update` is used to add content to the object as follows:

```
cryptic.update("Can you keep a secret?")
```

You can also create the object and add to the message at the same time:

```
secret = MD5.new("Sensitive data")
```

If a string argument is given, it is added to the object using `update`. Repeated calls are equivalent to a single call with concatenated arguments.

```
# These two statements...
cryptic.update("Shhh! ")
cryptic.update("Be very, very quiet!")

# ...are equivalent to this one.
cryptic.update("Shhh! Be very, very quiet!").
```

The `digest` method provides a 16-byte binary string containing the 128-bit digest.

The `hexdigest` method is what we actually find most useful. It provides the digest as an ASCII string of 32 hex characters representing the 16 bytes. This method is equivalent to the following:

```
def hexdigest
  ret = ''
  digest.each_byte { |i| ret << sprintf('%02x', i) }
  ret
end

secret.hexdigest  #  "b30e77a94604b78bd7a7e64ad500f3c2"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

In short, you can get an `md5` hash as follows:

```
require 'md5'
m = MD5.new("sensitive data").hexdigest
```

Calculating a 32-bit CRC

The *Cyclic Redundancy Checksum (CRC)* is a well-known way of obtaining a signature for a file or other collection of bytes. The CRC has the property that the chance of data being changed and keeping the same CRC is 1 in 2**N, where *N* is the number of bits in the result (most often 32 bits).

We refer you to the Zlib library for this. This library, created by Ueno Katsuhiro, isn't part of the standard distribution, but is still well known.

The method `crc32` will compute a CRC given a string as a parameter:

```
crc = Zlib::crc32("hello")  # 907060870
```

A previous CRC can be specified as an optional second parameter; the result will be as if the strings were concatenated and a single CRC was computed. This can be used, for example, to compute the checksum of a file so large that we can only read it in chunks.

Numerical Computation of a Definite Integral

> *I'm very good at integral and differential calculus...*
>
> —W. S. Gilbert, *The Pirates of Penzance*, Act I

If you want to estimate the value of a definite integral, there is a time-tested technique for doing so. Essentially we are performing what the calculus student will remember as a Riemann sum.

The `integrate` method shown here will take beginning and ending values for the dependent variable as well as an increment. The fourth parameter (which isn't really a parameter) is a block. This block should evaluate a function based on the value of the dependent variable passed into that block. (Here we are using *variable* in the mathematical sense, not in the computing sense.) It isn't necessary to define a function to call in this block, but we do so here for clarity.

```
def integrate(x0, x1, dx=(x1-x0)/1000.0)
  x = x0
  sum = 0
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
    loop do
      y = yield(x)
      sum += dx * y
      x += dx
      break if x > x1
    end
    sum
end

def f(x)
  x**2
end
z = integrate(0.0,5.0) { |x| f(x) }

puts z, "\n"           # 41.7291875
```

Note that in the preceding example, we are relying on the fact that a block returns a value that `yield` can retrieve. We also make certain assumptions here. First, we assume that x0 is less than x1 (otherwise an infinite loop will result); you can easily improve the code in details such as this one. Second, we assume that the function can be evaluated at arbitrary points in the specified domain. If at any time we try to evaluate the function at any other point, chaos will ensue. (Such functions generally aren't integrable anyway, at least over that set of x values. Consider the function f(*x*)=*x*/(*x*-3), when *x* is 3.)

Drawing on our faded memories of calculus, we might compute the result here to be 41.666 or thereabout (5 cubed divided by 3). Why is the answer not as exact as we might like? It is because of the size of the slice in the Riemann sum; a smaller value for dx will result in greater accuracy (at the expense of an increase in runtime).

Finally, we will point out that a function like this is more useful when we have a variety of functions of arbitrary complexity, not just a simple function such as f(*x*) = *x***2.

Trigonometry in Degrees, Radians, and Grads

When it comes to measuring arc, the mathematical or natural unit is the radian, defined in such a way that an angle of one radian will correspond to an arclength equal to the radius of the circle. A little thought will show that there are 2π radians in a circle.

The degree of arc that we use in everyday life is a holdover from ancient Babylonian base-60 units; this system divides the circle into 360 degrees. The less-familiar *grad* is a pseudo-metric unit defined in such a way that there are 100 grads in a right angle (or 400 in a circle).

Programming languages often default to the radian when calculating trigonometric functions, and Ruby is no exception. But we show here how to do these calculations in degrees or grads, in the event that any of you are engineers.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Because the number of units in a circle is a simple constant, it follows that there are simple conversion factors between all these units. We will define these here and simply use the constant names in subsequent code. As a matter of convenience, we'll stick them in the Math module.

```
module Math

  RAD2DEG  = 360.0/(2.0*PI)  # Radians to degrees
  RAD2GRAD = 400.0/(2.0*PI)  # Radians to grads

end
```

Now we can define new trig functions if we want. Because we are converting to radians in each case, we will divide by the conversion factor we calculated previously. We could place these in the Math module if we wanted, although we don't show it here.

```
def sin_d(theta)
  Math.sin (theta/Math::RAD2DEG)
end

def sin_g(theta)
  Math.sin (theta/Math::RAD2GRAD)
end
```

Of course, the corresponding `cos` and `tan` functions can be similarly defined.

The `atan2` function is a little different. It takes two arguments (the opposite and adjacent legs of a right triangle) and returns an angle. Thus we convert the result, not the argument, handling it this way:

```
def atan2_d(y,x)
  Math.atan2(y,x)/Math::RAD2DEG
end

def atan2_g(y,x)
  Math.atan2(y,x)/Math::RAD2GRAD
end
```

More Advanced Trig: Arcsin, Arccos, and Hyperbolic Functions

Ruby's Math module doesn't provide arcsin and arccos functions, but you can always define your own. Here we don't provide the theory but only the code.

```
def arcsin(x)
  Math.atan2(x, Math.sqrt(1.0-x*x))
end

def arccos(x)
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
  Math.atan2(Math.sqrt(1.0-x*x), x)
end
```

Note that because we used `atan2`, we don't have to worry about dividing by zero. This is a compelling reason to use `atan2`, by the way, along with other issues regarding floating-point error and the speed of floating-point division.

Of course, if you prefer the traditional arctan function that is so familiar to mathematicians, you can define it this way.

```
def arctan(x)
  Math.atan2(x,1.0)
end
```

All the preceding functions could be modified (as you have already seen) to use degrees or grads rather than radians.

The hyperbolic trig functions aren't defined in `Math`, but they can be defined as follows. We assume here that you're working with real (not complex) numbers.

```
def sinh(x)
  (Math.exp(x)-Math.exp(-x))/2.0
end

def cosh(x)
  (Math.exp(x)+Math.exp(-x))/2.0
end

def tanh(x)
  sinh(x)/cosh(x)
end
```

The inverses of these functions can also be defined.

```
def asinh(x)
  Math.log(x + Math.sqrt(1.0+x**2))
end

def acosh(x)
  2.0 * Math.log(Math.sqrt((x+1.0)/2.0)+Math.sqrt((x-1)/2.0))
end

def atanh(x)
  (Math.log (1.0+x) - Math.log(1.0-x)) / 2.0
end
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Finding Logarithms with Arbitrary Bases

When working with logarithms, we frequently use the *natural* logarithms (or base e, which is sometimes written `ln`); we can also use the *common* or base 10 logarithms. These are defined in Ruby as `Math.log` and `Math.log10`, respectively.

In computer science, specifically in such areas as coding and information theory, a base 2 log isn't unusual. For example, this tells the minimum number of bits needed to store a number. We define this function here as `log2`:

```
def log2(x)
  Math.log(x)/Math.log(2)
end
```

The inverse is obviously 2**x just as the inverse of log x is Math::E**x or Math.exp(x).

Furthermore, this same technique can be extended to any base. In the unlikely event that you ever need a base 7 logarithm, this will do the trick.

```
def log7(x)
  Math.log(x)/Math.log(7)
end
```

In practice, the denominator should be calculated once and kept around as a constant.

Comparing Floating-Point Numbers

It is a sad fact of life that computers don't represent floating-point values exactly. The following code fragment, in a perfect world, would print `"yes"`; on every architecture we have tried, it prints `"no"` instead.

```
x = 1000001.0/0.003
y = 0.003*x
if y == 1000001.0
  puts "yes"
else
  puts "no"
end
```

The reason, of course, is that a floating-point number is stored in some finite number of bits; and no finite number of bits is adequate to store a repeating decimal with an infinite number of digits.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Because of this inherent inaccuracy in floating-point comparisons, we might find ourselves in situations (like the one we just saw) in which the values we are comparing are the same for all practical purposes, but the hardware stubbornly thinks they are different.

Here is a simple way to ensure that floating-point comparisons are done with a fudge factor; that is, the comparisons will be done to within any tolerance specified by the programmer.

```
class Float

  EPSILON = 1e-6   # 0.000001

  def ==(x)
    (self-x).abs < EPSILON
  end

end
x = 1000001.0/0.003
y = 0.003*x
if y == 1.0          # Using the new ==
  puts "yes"         # Now we output "yes"
else
  puts "no"
end
```

We might find that we want different tolerances for different situations. For this case, we define a new method `equals?` as a member of `Float`. (We name it this in order to avoid confusion with the standard methods `equal?` and `eql?`; the latter in particular shouldn't be overridden.)

```
class Float

  EPSILON = 1e-6

  def equals?(x, tolerance=EPSILON)
    (self-x).abs < tolerance
  end

end

flag1 = (3.1416).equals? Math::PI          # false
flag2 = (3.1416).equals?(Math::PI, 0.001)  # true
```

We could also use a different operator entirely to represent approximate equality; the =~ operator might be a good choice.

We'll also mention here that there is a `BigFloat` class (created by Shigeo Kobayashi) that isn't part of the standard Ruby distribution; this extension allows essentially infinite-precision floating-point math. The library can be found in the Ruby Application Archive.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Finding the Mean, Median, and Mode of a Data Set

Given an array *x*, let's find the mean of all the values in that array. Actually, there are three common kinds of mean. The ordinary or *arithmetic* mean is what we call the average in everyday life. The *harmonic* mean is the number of terms divided by the sum of all their reciprocals. And finally, the *geometric* mean is the *n*th root of the product of the *n* values. Each of these is shown as follows:

```
def mean(x)
  sum=0
  x.each { |v| sum += v}
  sum/x.size
end
def hmean(x)
  sum=0
  x.each { |v| sum += (1.0/v)}
  x.size/sum
end

def gmean(x)
  prod=1.0
  x.each { |v| prod *= v}
  prod**(1.0/x.size)
end

data = [1.1, 2.3, 3.3, 1.2, 4.5, 2.1, 6.6]

am = mean(data)    # 3.014285714
hm = hmean(data)   # 2.101997946
gm = gmean(data)   # 2.508411474
```

The `median` value of a data set is the value that occurs approximately in the middle of the set. For this value, half the numbers in the set should be less and half should be greater. Obviously, this statistic will be more appropriate and meaningful for some data sets than others.

```
def median(x)
  sorted = x.sort
  mid = x.size/2
  sorted[mid]
end

data = [7,7,7,4,4,5,4,5,7,2,2,3,3,7,3,4]
puts median(data)          # 4
```

The mode of a data set is the value that occurs most frequently. If there is only one such value, the set is unimodal; otherwise it is multimodal. A multimodal data set is a more complex case that we don't consider here. If interested, you can extend and improve the code shown here.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
def mode(x)
  sorted = x.sort
  a = Array.new
  b = Array.new

  sorted.each do |x|
    if a.index(x) == nil
      a << x               # Add to list of values
      b << 1               # Add to list of frequencies
    else
      b[a.index(x)] += 1   # Increment existing counter
    end
  end
  maxval = b.max           # Find highest count
  where  = b.index(maxval) # Find index of highest count
  a[where]                 # Find corresponding data value
end

data = [7,7,7,4,4,5,4,5,7,2,2,3,3,7,3,4]
puts mode(data)            # 7
```

Variance and Standard Deviation

The variance of a set of data is a measure of how spread out the values are. (Here we don't distinguish between biased and unbiased estimates.) The standard deviation, usually represented by a sigma (ε) is simply the square root of the variance.

```
data = [2, 3, 2, 2, 3, 4, 5, 5, 4, 3, 4, 1, 2]

def variance(x)
  m = mean(x)
  sum = 0.0
  x.each { |v| sum += (v-m)**2 }
  sum/x.size
end

def sigma(x)
  Math.sqrt(variance(x))
end

puts variance(data)   # 1.461538462
puts sigma(data)      # 1.20894105
```

Note that the preceding variance function makes use of the mean function defined earlier.

Finding a Correlation Coefficient

The correlation coefficient is one of the simplest and most universal of statistical measures. It is a measure of the "linearity" of a set of x-y pairs, ranging from -1.0 (complete negative correlation) to +1.0 (complete positive correlation).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

We compute this using the mean and sigma (standard deviation) functions that we defined previously. For an explanation of this tool, consult any statistics text.

The first version we show assumes two arrays of numbers (of the same size).

```
def correlate(x,y)
  sum = 0.0
  x.each_index do |i|
    sum += x[i]*y[i]
  end
  xymean = sum/x.size.to_f
  xmean  = mean(x)
  ymean  = mean(y)
  sx = sigma(x)
  sy = sigma(y)
  (xymean-(xmean*ymean))/(sx*sy)
end

a = [3, 6, 9, 12, 15, 18, 21]
b = [1.1, 2.1, 3.4, 4.8, 5.6]
c = [1.9, 1.0, 3.9, 3.1, 6.9]

c1 = correlate(a,a)          # 1.0
c2 = correlate(a,a.reverse)  # -1.0
c3 = correlate(b,c)          # 0.8221970228
```

The next version is similar, but it operates on a single array, each element of which is an array containing an x-y pair.

```
def correlate2(v)
  sum = 0.0
  v.each do |a|
    sum += a[0]*a[1]
  end
  xymean = sum/v.size.to_f
  x = v.collect { |a| a[0]}
  y = v.collect { |a| a[1]}
  xmean  = mean(x)
  ymean  = mean(y)
  sx = sigma(x)
  sy = sigma(y)
  (xymean-(xmean*ymean))/(sx*sy)
end

d = [[1,6.1], [2.1,3.1], [3.9,5.0], [4.8,6.2]]

c4 = correlate2(d)           # 0.2277822492
```

Finally we show a version that assumes the x-y pairs are stored in a hash. It simply builds on the previous example.

```
def correlate_h(h)
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
  correlate2(h.to_a)
end

e = {  1 => 6.1, 2.1 => 3.1, 3.9 => 5.0, 4.8 => 6.2}

c5 = correlate_h(e)          # 0.2277822492
```

Performing Base Conversions

Obviously any integer can be represented in any base because they are all stored internally in binary. Further, we know that Ruby can deal with integer constants in any of the four commonly-used bases. This means that if we are concerned about base conversions, we must be concerned with strings in some fashion.

If you are concerned with converting a string to an integer, that is covered in "Converting Strings to Numbers (Decimal and Otherwise)."

If you are concerned with converting numbers to strings, that is another matter. The best way to do it is with the `%` method of the String class. This method formats its argument according to the `printf` directive found in the string.

```
hex = "%x" % 1234      # "4d2"
oct = "%o" % 1234      # "2322"
bin = "%b" % 1234      # "10011010010"
```

Converting from one nondecimal base to another can be done with a combination of these techniques.

```
oct = "2322"
hex = "%x" % oct.oct   # "4d2"
```

Converting to and from oddball bases such as 5 or 11 is unsupported by Ruby. This is rare enough that we will leave it as an exercise for you.

Generating Random Numbers

If a pseudorandom number is good enough for you, you're in luck. This is what most language implementations supply you with, and Ruby is no exception.

The Kernel method `rand` will return a pseudorandom floating-point number *x* such that *x*>=0.0 and *x*<1.0.

```
a = rand      # 0.6279091137
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

If it is called with an integer parameter `max`, it will return an integer in the range `0...max` (noninclusive of the upper bound).

```
n = rand(10)  # 7
```

If we want to seed the random number generator, we can do so with the Kernel method `srand`, which takes a single numeric parameter. If we pass no value to it, it will construct its own using (among other things) the time of day. If we pass a number to it, it will use that number as the seed. This can be useful in testing, when we want a repeatable sequence of pseudorandom numbers from one script invocation to the next.

```
srand(5)
i, j, k = rand(100), rand(100), rand(100)
# 26, 45, 56

srand(5)
l, m, n = rand(100), rand(100), rand(100)
# 26, 45, 56
```

Caching Functions for Speed

Suppose that you have a computationally expensive mathematical function that will be called repeatedly in the course of execution. If speed is critical and you can afford to sacrifice a little memory and accuracy, it might be effective to store values in a table and look them up.

In this example, we define an arbitrary function called `zeta`, which we want to call over a domain of 0.0 to 90.0. The function `zeta` is defined to be `2 sin x cos x` (in degrees). Let's assume that our parameters will be no more accurate than a tenth of a degree. This means that if we want to store these values, we will need a table of about 900 elements.

Let's look at a code fragment.

```
def zeta(x)
  r2d  = 360.0/(2.0*Math::PI)  # Radians to degrees
  2.0*(Math.sin (x/r2d))*(Math.cos (x/r2d))
end

$fast_zeta = []

 (0..900).each { |x| $fast_zeta[x]=zeta(x/10.0)}

def fast_zeta(x)
  $fast_zeta[(x*10).round]
end

y1 = zeta(37.5)                    # Slow
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
y2 = fast_zeta(37.5)              # Somewhat faster

y3 = $fast_zeta[(37.5*10).round]  # Still faster
```

We define an array called `$fast_zeta`, using a global variable; we then populate it with all the values from `zeta(0.0)` to `zeta(90.0)`. We define a function called `fast_zeta`, which will take a parameter, convert it to an index, and find the appropriate entry in the array. As an alternative, we also access the array directly (in the computation of y3).

In our tests, we put each calculation in a tight loop that ran for millions of iterations. We found that, compared with the calculation of y1, the calculation of y2 was about 66% faster. In addition, the calculation of y3 (which avoided the method call overhead) was about 72.5% faster.

At times, this method won't be practical at all. But we present it to you as a simple demonstration of what can be done to increase speed without dropping into C code.

Matrix Manipulation

There is a standard library `matrix.rb` for this purpose. It is fairly full-featured, with class methods to create matrices in various forms (including identity and zero matrices) and an accessor method to get at the elements in standard *x[i,j]* form. There are methods to find a determinant, to transpose a matrix, to multiply by another matrix or by a scalar, and so on.

This is a standard library. It is too elaborate to document here in detail.

Complex Numbers

The standard library `complex.rb` provides most of the functionality anyone would need for working with numbers in the complex plane. Be warned that some of the methods are named with exclamation points when there isn't necessarily a compelling reason to do so.

This is a well-known standard library. We won't document it here because it is too complex (no pun intended).

Formatting Numbers with Commas

There might be better ways to do it, but this one works. We reverse the string to make it easier to do global substitution, and then reverse it again at the end.

```
def commas(x)
  str = x.to_s.reverse
  str.gsub!("([0-9]{ 3} )","\\1,")
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
  str.gsub(",$","").reverse
end

puts commas(123)         # "123"
puts commas(1234)        # "1,234"
puts commas(12345)       # "12,435"
puts commas(123456)      # "123,456"
puts commas(1234567)     # "1,234,567"
```

Times and Dates

> *Does anybody really know what time it is?*
>
> —Chicago, *Chicago IV*

One of the most complex and confusing areas of human life is that of measuring time. To come to a complete understanding of the subject, you would need to study physics, astronomy, history, law, business, and religion. Astronomers know (as most of us don't!) that solar time and sidereal time aren't quite the same thing, and why a leap second is occasionally added to the end of the year. Historians know that the calendar skipped several days in October 1582, when Italy converted from the Julian calendar to the Gregorian. Very few people know the difference between astronomical Easter and ecclesiastical Easter (which are almost always the same). Many people don't know that century years not divisible by 400 (such as the year 1900) aren't leap years.

Performing calculations with times and dates is common in computing, but it has traditionally been somewhat tedious in most programming languages. It is tedious in Ruby, too, because of the nature of the data. However, Ruby has taken some incremental steps toward making these operations easier.

As a courtesy to you, we'll go over a few terms that might not be familiar to you. Most of these come from standard usage or from other programming languages.

Greenwich Mean Time (GMT) is an old term not really in official use anymore. The new global standard is *Coordinated Universal Time (or UTC),* which is from the French version of the name. GMT and UTC are virtually the same thing; over a period of years, the difference will be on the order of seconds. Much of the software in the industry doesn't distinguish between the two at all (nor does Ruby).

Daylight Saving Time is a semiannual shift in the official time, amounting to a difference of one hour. Thus the U.S. time zones usually end in "ST" (standard time) or "DT" (daylight savings time). This annoying trick is used in most (although not all) of the U.S.A. Other countries need not worry about it.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The epoch is a term borrowed from UNIX lore. In this realm, a time is typically stored internally as a number of seconds from a specific point in time (called the *epoch*), which was midnight January 1, 1970 GMT. (Note that in U.S. time zones, this will actually be the preceding December 31.) The same term is used to denote not only the point of origin, but also the distance in time from that point.

The `Time` class is used for most operations. The Date and ParseDate libraries extend its capabilities somewhat. We will look at the basic techniques and the problems they enable us to solve.

Determining the Current Time

The most fundamental problem in time/date manipulation is to answer the question: What is the time and date right now? In Ruby, when we create a `Time` object with no parameters, it is set to the current date and time.

```
t0 = Time.new
```

`Time.now` is a synonym.

```
t0 = Time.now
```

Note that the resolution of system clocks varies from one architecture to another. It might include microseconds; in which case, two `Time` objects created in succession might actually record different times.

Working with Specific Times (Post-epoch)

Most software only needs to work with dates in the future or in the recent past. For these circumstances, the `Time` class is adequate. The relevant class methods are `mktime`, `local`, `gm`, and `utc`.

The `mktime` method will create a new `Time` object based on the parameters passed to it. These time units are given in reverse from longest to shortest: year, month, day, hours, minutes, seconds, microseconds. All but the year are optional; they default to the lowest possible value. The microseconds can be ignored on many architectures. The hours must be between 0 and 23 (that is, a 24-hour clock).

```
t1 = Time.mktime(2001)                # January 1, 2001 at 0:00:00
t2 = Time.mktime(2001,3)
t3 = Time.mktime(2001,3,15)
t4 = Time.mktime(2001,3,15,21)
t5 = Time.mktime(2001,3,15,21,30)
t6 = Time.mktime(2001,3,15,21,30,15)  # March 15, 2001 9:30:15 pm
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Note that `mktime` assumes the local time zone. In fact, `Time.local` is a synonym for it.

```
t7 = Time.local(2001,3,15,21,30,15)    # March 15, 2001 9:30:15 pm
```

The `Time.gm` method is basically the same, except that it assumes GMT (or UTC). Because the authors are in the U.S. Central time zone, we would see an eight-hour difference here.

```
t8 = Time.gm(2001,3,15,21,30,15)       # March 15, 2001 9:30:15 pm
# This is only 1:30:15 pm in Central time!
```

The `Time.utc` method is a synonym.

```
t9 = Time.utc(2001,3,15,21,30,15)      # March 15, 2001 9:30:15 pm
# Again, 1:30:15 pm Central time.
```

There is one more important item to note. All these methods can take an alternative set of parameters. The instance method `to_a` (which converts a time to an array of relevant values) returns a set of values in this order: seconds, minutes, hours, day, month, year, day of week (0..6), day of year (1..366), daylight saving (`true` or `false`), and time zone (as a string).

Thus, these are also valid calls:

```
t0 = Time.local(0,15,3,20,11,1979,2,324,false,"GMT-8:00")
t1 = Time.gm(*Time.now.to_a)
```

However, in the first example, don't fall into the trap of thinking that you can change the computable parameters such as the day of the week (in this case, `2` meaning Tuesday). A change like this simply contradicts the way our calendar works, and it will have no effect on the time object created. November 20, 1979, was a Tuesday regardless of how we might write our code.

Finally, note that there are obviously many ways to attempt coding incorrect times, such as a thirteenth month or a 35th day of the month. In cases like these, an `ArgumentError` will be raised.

Determining Day of the Week

There are several ways to do this. First of all, the instance method `to_a` will return an array of time information. You can access the seventh element, which is a number from `0` to `6` (`0` meaning Sunday and `6` meaning Saturday).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
time = Time.now
day = time.to_a[6]            # 2 (meaning Tuesday)
```

It's better to use the instance method `wday` as shown here:

```
day = time.wday               # 2 (meaning Tuesday)
```

But both these techniques are a little cumbersome. Sometimes we want the value coded as a number, but more often we don't. To get the actual name of the weekday as a string, we can use the `strftime` method. This name will be familiar to C programmers. There are nearly two dozen different specifiers that it recognizes, enabling us to format dates and times more or less as we please. (See the section "Formatting and Printing Date/Time Values.")

```
day = time.strftime("%a")     # "Tuesday"
```

It's also possible to obtain an abbreviated name.

```
long  = time.strftime("%A")   # "Tuesday"
```

Determining the Date of Easter

Traditionally, this holiday is one of the hardest to compute because it is tied to the lunar cycle. The lunar month doesn't go evenly into the solar year, and thus anything based on the moon can be expected to vary from year to year.

The algorithm we present here is a well-known one that has made the rounds. We have seen it coded in BASIC, Pascal, and C. We now present it to you in Ruby.

```
def easter(year)
  c = year/100
  n = year - 19*(year/19)
  k = (c-17)/25
  i = c - c/4 - (c-k)/3 + 19*n + 15
  i = i - 30*(i/30)
  i = i - (i/28)*(1 -(i/28)*(29/(i+1))*((21-n)/11))
  j = year + year/4 + i + 2 - c + c/4
  j = j - 7*(j/7)
  l = i - j
  month = 3 + (l+40)/44
  day = l + 28 - 31*(month/4)
  [month, day]
end

date = easter 2001      # Find month/day for 2001
date = [2001] + date    # Tack year on front
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
t = Time.local *date     # Pass parameters to Time.local
puts t                   # Sun Apr 15 01:00:00 GMT-8:00 2001
```

We would love to explain this algorithm to you, but we don't understand it ourselves. Some things must be taken on faith, and in the case of Easter, this might be especially appropriate.

Finding the *N*th Weekday in a Month

Sometimes for a given month and year, we want to find the date of the third Monday in the month, or the second Tuesday, and so on. Listing 2.9 makes that calculation simple.

If we are looking for the *nth* occurrence of a certain weekday, we pass *n* as the first parameter. The second parameter is the number of that weekday (*0* meaning Sunday, *1* meaning Monday, and so on). The third and fourth parameters are the month and year, respectively.

Listing 2.9. Finding the Nth Weekday

```
def nth_wday(n, wday, month, year)
  if (!n.between? 1,5) or
     (!wday.between? 0,6) or
     (!month.between? 1,12)
    raise ArgumentError
  end
  t = Time.local year, month, 1
  first = t.wday
  if first == wday
     fwd = 1
   elsif first < wday
     fwd = wday - first + 1
   elsif first > wday
     fwd = (wday+7) - first + 1
   end
   target = fwd + (n-1)*7
   begin
     t2 = Time.local year, month, target
   rescue ArgumentError
     return nil
   end
   if t2.mday == target
     t2
   else
     nil
   end
end

puts nth_wday(ARGV[0].to_i, ARGV[1].to_i, ARGV[2].to_i, ARGV[3].to_i)
```

The peculiar-looking code near the end of the method is put there to counteract a long-standing tradition in the underlying time-handling routines. You might expect that trying to create a date of November 31 would result in an error of some kind. You would be mistaken. Most systems would happily (and silently) convert this to December 1. If you

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

are an old-time UNIX hacker, you might think this is a feature; otherwise, you might consider it a bug.

We won't venture an opinion here as to what the underlying library code ought to do or whether Ruby ought to change that behavior. But we don't want to have this routine perpetuate the tradition. If you are looking for the date of, say, the fifth Friday in November 2000, you will get a `nil` value back (rather than December 1, 2000).

Converting Between Seconds and Larger Units

Sometimes we want to take a number of seconds and convert to days, hours, minutes, and seconds. This little routine will do just that.

```
def sec2dhms(secs)
  time = secs.round          # Get rid of microseconds
  sec = time % 60            # Extract seconds
  time /= 60                 # Get rid of seconds
  mins = time % 60           # Extract minutes
  time /= 60                 # Get rid of minutes
  hrs = time % 24            # Extract hours
  time /= 24                 # Get rid of hours
  days = time                # Days (final remainder)
  [days, hrs, mins, sec]     # Return array [d,h,m,s]
end

t = sec2dhms(1000000)        # A million seconds is...

puts "#{ t[0]}  days,"         # 11 days,
puts "#{ t[1]}  hours,"        # 13 hours,
puts "#{ t[2]}  minutes,"      # 46 minutes,
puts " and #{ t[3]}  seconds." # and 40 seconds.
```

We could, of course, go up to higher units. But a week isn't an overly useful unit, a month isn't a well-defined term, and a year is far from being an integral number of days.

We also present here the inverse of that function.

```
def dhms2sec(days,hrs=0,min=0,sec=0)
  days*86400 + hrs*3600 + min*60 + sec
end
```

Converting to and from the Epoch

For various reasons, we might want to convert back and forth between the internal (or traditional) measure and the standard date form. Internally, dates are stored as a number of seconds since the epoch.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The `Time.at` class method will create a new time given the number of seconds since the epoch.

```
epoch = Time.at(0)          # Find the epoch (1 Jan 1970 GMT)
newmil = Time.at(978307200) # Happy New Millennium! (1 Jan 2001)
```

The inverse is the instance method `to_i`, which converts to an integer.

```
now = Time.now              # 16 Nov 2000 17:24:28
sec = now.to_i              # 974424268
```

If you need microseconds and your system supports that resolution, you can use `to_f` to convert to a floating-point number.

Working with Leap Seconds: Don't!

> *Ah, but my calculations, people say,*
>
> *Reduced the year to better reckoning? Nay,*
>
> *'Twas only striking from the calendar*
>
> *Unborn Tomorrow and dead Yesterday.*
>
> —Omar Khayyam, *The Rubaiyat* (translation by Fitzgerald)

You want to work with leap seconds? Our advice is: Don't do it.

Although leap seconds are very real and for years the library routines have for years allowed for the possibility of a 61-second minute, our experience has been that most systems don't keep track of leap seconds. By most, we mean all the ones we've ever checked.

For example, a leap second is known to have been inserted at the end of the last day of 1998. Immediately following 23:59:59 came that rare event 23:59:60. But the underlying C libraries on which Ruby is built are ignorant of this.

```
t0 = Time.gm(1998, 12, 31, 23, 59, 59)
t1 = t0 + 1
puts t1      # Fri Jan 01 00:00:00 GMT 1999
```

It is (barely) conceivable that Ruby could add a layer of intelligence to correct for this. At the time of this writing, however, there are no plans to add such functionality.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Finding the Day of the Year

The day number within the year is sometimes called the Julian date, which isn't directly related to the Julian calendar that has gone out of style. Other people insist that this usage isn't correct, so we won't use it from here on.

No matter what you call it, there will be times you want to know what day of the year it is, from 1 to 366. This is easy in Ruby; we use the `yday` method.

```
t = Time.now
day = t.yday      # 315
```

Validating a Date/Time

As you saw in "Finding the Nth Weekday in a Month," the standard date/time functions don't check the validity of a date, but roll it over as needed. For example, November 31 will be translated to December 1.

At times, this might be the behavior you want. If it isn't, you will be happy to know that the standard library date doesn't regard such a date as valid. We can use this fact to perform validation of a date as we instantiate it.

```
class Time

  def Time.validate(year, month=1, day=1,
                    hour=0, min=0, sec=0, usec=0)
    require "date"

    begin
      d = Date.new(year,month,day)
    rescue
      return nil
    end
    Time.local(year,month,day,hour,min,sec,usec)
  end

end

t1 = Time.validate(2000,11,30)  # Instantiates a valid object
t2 = Time.validate(2000,11,31)  # Returns nil
```

Here we have taken the lazy way out; we simply set the return value to `nil` if the parameters passed in don't form a valid date (as determined by the `Date` class). We have made this method a class method of `Time` by analogy with the other methods that instantiate objects.

Note that the `Date` class can work with dates prior to the epoch. This means that passing in a date such as 31 May 1961 will succeed as far as the `Date` class is concerned. But when

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

these values are passed into the `Time` class, an `ArgumentError` will result. We don't attempt to catch that exception here because we think it's appropriate to let it be caught at the same level as (for example) `Time.local`, in the user code.

Speaking of `Time.local`, we used that method in the preceding; but if we wanted GMT instead, we could have called the `gmt` method. It would be a good idea to implement both flavors.

Finding the Week of the Year

The definition of week number isn't absolute and fixed. Various businesses, coalitions, government agencies, and standards bodies have differing concepts of what it means. This stems, of course, from the fact that the year can start on any day of the week; we might or might not want to count partial weeks, and we might start on Sunday or Monday.

We offer only three alternatives here. The first two are made available by the Time method `strftime`. The `%U` specifier numbers the weeks starting from Sunday, and the `%W` specifier starts with Monday.

The third possibility comes from the `Date` class. It has an accessor called `cweek`, which returns the week number based on the ISO 8601 definition (which says that week 1 is the week containing the first Thursday of the year).

If none of these three suits you, you might have to roll your own. We present these three in a single code fragment.

```
require "date"

# Let's look at May 1 in the years
# 2002 and 2005.

t1 = Time.local(2002,5,1)
d1 = Date.new(2002,5,1)

week1a = t1.strftime("%U").to_i   # 17
week1b = t1.strftime("%W").to_i   # 17
week1c = d1.cweek                 # 18

t2 = Time.local(2005,5,1)
d2 = Date.new(2005,5,1)

week2a = t2.strftime("%U").to_i   # 18
week2b = t2.strftime("%W").to_i   # 18
week2c = d2.cweek                 # 17
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Detecting Leap Years

The `Date` class has two class methods `julian_leap?` and `gregorian_leap?`; only the latter is of use in recent years. It also has a `method leap?`, which is an alias for the `gregorian_leap?` method.

```
require "date"
flag1 = Date.julian_leap? 1700     # true
flag2 = Date.gregorian_leap? 1700  # false
flag3 = Date.leap? 1700            # false
```

Every child knows the first rule for leap years: The year number must be divisible by four. Fewer people know the second rule, that the year number must not be divisible by 100; and fewer still know the exception, that the year can be divisible by 400. In other words: A century year is a leap year only if it is divisible by 400, so 1900 wasn't a leap year, but 2000 was.

The `Time` class doesn't have a method like this, but if we needed one, it would be simple to create.

```
class Time

  def Time.leap? year
    if year % 400 == 0
      true
    elsif year % 100 == 0
      false
    elsif year % 4 == 0
      true
    else
      false
  end

end
```

We implement this as a class method by analogy with the `Date` class methods. It could also be implemented as an instance method.

Obtaining the Time Zone

The accessor zone in the `Time` class will return a String representation of the time zone name.

```
z1 = Time.gm(2000,11,10,22,5,0).zone     # "GMT-6:00"
z2 = Time.local(2000,11,10,22,5,0).zone  # "GMT-6:00"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Unfortunately, times are stored relative to the current time zone, not the one with which the object was created.

Working with Hours and Minutes Only

We might want to work with times of day as strings. Once again, `strftime` comes to our aid.

We can print the time with hours, minutes, and seconds if we want.

```
t = Time.now
puts t.strftime("%H:%M:%S")     # Prints 22:07:45
```

We can print hours and minutes only (and, using the trick of adding 30 seconds to the time, we can even round to the nearest minute).

```
puts t.strftime("%H:%M")        # Prints 22:07
puts (t+30).strftime("%H:%M")   # Prints 22:08
```

Finally, if we don't like the standard 24-hour (or military) clock, we can switch to the 12-hour clock. It's appropriate to add a meridian indicator then (`AM`/`PM`).

```
puts t.strftime("%I:%M %p")     # Prints 10:07 PM
```

There are other possibilities, of course. Use your imagination.

Comparing Date/Time Values

The `Time` class conveniently mixes in the Comparable module, so dates and times might be compared in a straightforward way.

```
t0 = Time.local(2000,11,10,22,15)      # 10 Nov 2000 22:15
t1 = Time.local(2000,11,9,23,45)       #  9 Nov 2000 23:45
t2 = Time.local(2000,11,12,8,10)       # 12 Nov 2000  8:10
t3 = Time.local(2000,11,11,10,25)      # 11 Nov 2000 10:25

if t0 < t1 then puts "t0 < t1" end
if t1 != t2 then puts "t1 != t2" end
if t1 <= t2 then puts "t1 <= t2" end
if t3.between?(t1,t2)
  puts "t3 is between t1 and t2"
end

# All four if statements test true
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Adding Intervals to Date/Time Values

We can obtain a new time by adding an interval to a specified time. The number is interpreted as a number of seconds.

```
t0 = Time.now
t1 = t0 + 60          # Exactly one minute past t0
t2 = t0 + 3600        # Exactly one hour past t0
t3 = t0 + 86400       # Exactly one day past t0
```

The function `dhms2sec` (defined in "Converting Between Seconds and Larger Units") might be helpful here. Recall that the hours, minutes, and seconds all default to `0`.

```
t4 = t0 + dhms2sec(5,10)        # Ahead 5 days, 10 hours
t5 = t0 + dhms2sec(22,18,15)    # Ahead 22 days, 18 hrs, 15 min
t6 = t0 - dhms2sec(7)           # Exactly one week ago
```

Don't forget that we can move backward in time by subtracting. This is shown in the preceding calculation of `t6`.

Computing the Difference in Two Date/Time Values

We can find the interval of time between two points in time. Subtracting one `Time` object from another gives us a number of seconds.

```
today = Time.local(2000,11,10)
yesterday = Time.local(2000,11,9)
diff = today - yesterday           # 86400 seconds
```

Once again, the function `sec2dhms` comes in handy. (This is defined in "Converting Between Seconds and Larger Units.")

```
past = Time.local(1998,9,13,4,15)
now = Time.local(2000,11,10,22,42)
diff = now - past
unit = sec2dhms(diff)
puts "#{ unit[0]}  days,"            # 789 days,
puts "#{ unit[1]}  hours,"           # 18 hours,
puts "#{ unit[2]}  minutes,"         # 27 minutes,
puts "and #{ unit[3]}  seconds."     # and 0 seconds.
```

Working with Specific Dates (Pre-epoch)

The standard library Date provides a class of the same name for working with dates that precede midnight GMT, January 1, 1970.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Although there is some overlap in functionality with the `Time` class, there are significant differences. Most notably, the `Date` class doesn't handle the time of day at all. Its resolution is a single day. Also, the `Date` class performs more rigorous error-checking than the `Time` class; if you attempt to refer to a date such as June 31 (or even February 29 in a nonleap year), you will get an error. The code is smart enough to know the different cutoff dates for Italy and England switching to the Gregorian calendar (in 1582 and 1752, respectively), and it can detect nonexistent dates that are a result of this switchover. This standard library is a tangle of interesting and arcane code. We don't have space to document it further here.

Retrieving a Date/Time Value from a String

A date and time can be formatted as a string in many different ways because of abbreviations, varying punctuation, different orderings, and so on. Because of the various ways of formatting, writing code to decipher such a character string can be daunting. Consider these examples:

```
s1 = "9/13/98 2:15am"
s2 = "1961-05-31"
s3 = "11 July 1924"
s4 = "April 17, 1929"
s5 = "20 July 1969 16:17 EDT"  # That's one small step...
s6 = "Mon Nov 13 2000"
s7 = "August 24, 79"           # Destruction of Pompeii
s8 = "8/24/79"
```

Fortunately, much of the work has already been done for us. The `ParseDate` module has a single class of the same name, which has a single method called `parsedate`. This method returns an array of elements in this order: year, month, day, hour, minute, second, time zone, day of week. Any fields that cannot be determined are returned as `nil` values.

```
require "parsedate.rb"
include ParseDate

p parsedate(s1)      # [98, 9, 13, 2, 15, nil, nil, nil]
p parsedate(s2)      # [1961, 5, 31, nil, nil, nil, nil, nil]
p parsedate(s3)      # [1924, 7, 11, nil, nil, nil, nil, nil]
p parsedate(s4)      # [1929, 4, 17, nil, nil, nil, nil, nil]
p parsedate(s5)      # [1969, 7, 20, 16, 17, nil, "EDT", nil]
p parsedate(s6)      # [2000, 11, 13, nil, nil, nil, nil, 1]
p parsedate(s7)      # [79, 8, 24, nil, nil, nil, nil, nil]
p parsedate(s8,true) # [1979, 8, 24, nil, nil, nil, nil, nil]
```

The last two strings illustrate the purpose of `parsedate`'s second parameter `guess_year`; because of our cultural habit of representing a year as two digits, ambiguity can result. Thus the last two strings are interpreted differently because we parse `s8` with

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

`guess_year` set to `true`, resulting in its conversion to a four-digit year. On the other hand, `s7` refers to the eruption of Vesuvius in 79 A.D., so we definitely want a two-digit year there.

The rule for `guess_year` is this: If the year is less than 100 and `guess_year` is `true`, convert to a four-digit year. The conversion will be done as follows: If the year is 70 or greater, add 1900 to it; otherwise add 2000. Thus 75 will translate to 1975, but 65 will translate to 2065. This rule isn't uncommon in the computing world.

What about `s1`, where we probably intended 1998 as the year? All is not lost as long as we pass this number to some other piece of code that interprets it as 1998.

Note that `parsedate` does virtually no error checking. For example, if you feed it a date with a weekday and a date that don't correspond correctly, it won't detect this discrepancy. It is only a parser, and it does this job pretty well, but no other.

Also note an American bias in this code. An American writing 3/4/2001 usually means March 4, 2001; in Europe and most other places, this would mean April 3 instead. But if all the data is consistent, this isn't a huge problem. Because the return value is simply an array, you can mentally switch the meaning of elements 1 and 2. Be aware also that this bias happens even with a date such as 15/3/2000, where it is clear (to us) that 15 is the day. The `parsedate` method will happily return 15 as the month value.

Although this method is very flexible, it is far from perfect. We have observed that it tends not to capture the time zone if it follows a meridian indicator such as p.m. We have also noted that it doesn't recognize a year such as '79 (with a leading apostrophe).

Formatting and Printing Date/Time Values

You can obtain the canonical representation of the date and time by calling the `asctime` method (ASCII time); it has an alias called *ctime,* for those who already know it by that name.

You can obtain a similar result by calling the `to_s` method. This is the same as the result you would get if doing a simple `puts` of a date/time value.

The `strftime` method of class `Time` will format a date and time in almost any form you can think of. Other examples in this chapter have shown the use of the directives `%a`, `%A`, `%U`, `%W`, `%H`, `%M`, `%S`, `%I`, and `%p`; we list here all the remaining directives that `strftime` recognizes.

```
%b    Abbreviated month name ("Jan")
%B    Full month name ("January")
%c    Preferred local date/time representation
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
%d    Day of the month (1..31)
%j    Day of the year (1..366); so-called "Julian date"
%m    Month as a number (1..12)
%w    Day of the week as a number (0..6)
%x    Preferred representation for date (no time)
%y    Two-digit year (no century)
%Y    Four-digit year
%Z    Time zone name
%%    A literal "%" character
```

For more information, consult a Ruby reference.

Time Zone Conversions

It is only convenient to work with two time zones: GMT (or UTC) is one, and the other is whatever time zone you happen to be in.

The `gmtime` method will convert a time to GMT (changing the receiver in place). There is an alias named `utc`.

You might expect that it would be possible to convert a time to an array, tweak the time zone, and convert it back. The trouble with this is that all the class methods such as `local` and `gm` (or their aliases `mktime` and `utc`) want to create a `Time` object using either your local time zone or GMT.

It is possible to fake time zone conversions. This does require that you know the time difference in advance.

```
mississippi = Time.local(2000,11,13,9,35) # 9:35 am CST
california  = mississippi - 2*3600        # Minus two hours

time1 = mississippi.strftime("%X CST")    # 09:35:00 CST
time2 = california.strftime("%X PST")     # 07:35:00 PST
```

The `%X` directive to `strftime` that we see here simply uses the hh:mm:ss format as shown.

Finding the Internet Time (@nnn)

> *Time is an illusion created by the Swiss to sell watches.*
>
> —Douglas Adams

We offer this next item mostly as a curiosity. The Swiss watch manufacturer Swatch has created a trendy way of measuring time in cyberspace, a metric-like time that they call Internet Time. This time standard has no time zones and is thus usable, for example, by people meeting each other in chat rooms when they are physically thousands of miles apart.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

It was inaugurated on October 23, 1998, in the presence of Nicholas Negroponte, founder and director of MIT's Media Lab.

Not surprisingly, Internet Time is based on the meridian of Biel, Switzerland. The day is divided into 1000 beats, each 86.4 seconds long (or 1 minute, 26.4 seconds). The three digit number representing the time of day is prefixed by an at (`@`) sign.

This method will find the current time in Internet Time. It returns a number, unless `true` is passed in; then it returns a string, with an `@` and any leading zeroes. Mimicking the behavior of the applications we have seen, we truncate any fractional part rather than rounding up.

```
def internet_time(str=false)
  t = Time.now.gmtime + 3600   # Biel, Switzerland
  midnight = Time.gm(t.year, t.month, t.day)
  secs = t - midnight
  beats = (secs/86.4).to_i
  if str
    "@%03d" % beats
  else
    beats
  end
end

time_now = internet_time     # 27
now = internet_time(true)    # "@027"
```

Summary

That ends our discussion of the simpler tasks we might perform with numbers, strings, and so on. Now it's time to go on to bigger and better data structures and the algorithms that support them.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17