Chapter 2
Working with 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

A computer science professor in the early 1980s 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," and 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 therefore strings) enables communication between humans and computers. Every kind of information we can imagine, including natural language text, can be encoded in character strings.

A *string* is simply a sequence of characters. Like most entities in Ruby, strings are first-class objects. In everyday programming, we need to manipulate strings in many ways. We want to concatenate strings, tokenize them, analyze them, perform searches and substitutions, and more. Ruby makes most of these tasks easy.

For much of the history of Ruby, a single byte was considered a character. That is not true of special characters, emoji, and most non-Latin scripts. For a more detailed discussion of the ways that bytes and characters are often not the same, refer to Chapter 4, "Internationalization in Ruby."

2.1 Representing Ordinary Strings

A string in Ruby is composed simply of a sequence of 8-bit bytes. It is not null terminated as in C, so it may contain null characters. Strings containing bytes above 0xFF are always legal, but are only meaningful in non-ASCII encodings. Strings are assumed to use the UTF-8 encoding. Before Ruby 2.0, they were assumed to be simple ASCII. (For more information on encodings, refer to Chapter 4.)

The simplest string in Ruby is single quoted. Such a string is taken absolutely literally; the only escape sequences recognized are the single quote (`\'`) and the escaped backslash itself (`\\`). Here are some examples:

```
s1 = 'This is a string'    # This is a string
s2 = 'Mrs. O\'Leary'       # Mrs. O'Leary
s3 = 'Look in C:\\TEMP'    # Look in C:\TEMP
```

A double-quoted string is more versatile. It allows many more escape sequences, such as backspace, tab, carriage return, and linefeed. It allows control characters to be embedded as octal numbers, and Unicode code points to be embedded via their hexadecimal reference number. Consider these examples:

```
s1 = "This is a tab: (\t)"
s2 = "Some backspaces: xyz\b\b\b"
s3 = "This is also a tab: \011"
s4 = "And these are both bells: \a \007"
s5 = "This is the unicode snowman: \u2603"
```

Non-ASCII characters will be shown "backslash escaped" when their string is inspected, but will print normally. Double-quoted strings also allow expressions to be embedded inside them. See Section 2.21, "Embedding Expressions within Strings."

2.2 Representing Strings with Alternate Notations

Sometimes we want to represent strings that are rich in metacharacters, such as single quotes, double quotes, and more. For these situations, we have the `%q` and `%Q` notations. Following either of these is a string within a pair of delimiters; I personally favor square brackets (`[]`).

The difference between the `%q` and `%Q` variants is that the former acts like a single-quoted string, and the latter like a double-quoted string:

```
S1 = %q[As Magritte said, "Ceci n'est pas une pipe."]
s2 = %q[This is not a tab: (\t)]  # same as: 'This is not a tab: \t'
s3 = %Q[This IS a tab: (\t)]      # same as: "This IS a tab: \t"
```

Both kinds of notation can be used with different delimiters. Besides brackets, there are other paired delimiters (parentheses, braces, and angle brackets):

```
s1 = %q(Bill said, "Bob said, 'This is a string.'")
s2 = %q{Another string.}
s3 = %q<Special characters '"[](){} in this string.>
```

There are also "nonpaired" delimiters. Basically any character may be used that is printable, but not alphanumeric, not whitespace, and not a paired character:

```
s1 = %q:"I think Mrs. O'Leary's cow did it," he said.:
s2 = %q*\r is a control-M and \n is a control-J.*
```

2.3 Using Here-Documents

If you want to represent a long string spanning multiple lines, you can certainly use a regular quoted string:

```
str = "Once upon a midnight dreary,
       While I pondered, weak and weary..."
```

However, the indentation will be part of the string.

Another way is to use a *here-document,* a string that is inherently multiline. (This concept and term are borrowed from older languages and contexts.) The syntax is the << symbol, followed by an end marker, then zero or more lines of text, and finally the same end marker on a line by itself:

```
str = <<EOF
Once upon a midnight dreary,
While I pondered weak and weary,...
EOF
```

Be careful about things such as trailing spaces on the final end marker line. Current versions of Ruby will fail to recognize the end marker in those situations.

Note that here-documents may be "stacked"; for example, here is a method call with three such strings passed to it:

```
some_method(<<STR1, <<STR2, <<STR3)
first piece
of text...
STR1
second piece...
STR2
third piece
of text.
STR3
```

By default, a here-document is like a double-quoted string—that is, its contents are subject to interpretation of escape sequences and interpolation of embedded expressions. But if the end marker is single-quoted, the here-document behaves like a single-quoted string:

```
str = <<'EOF'
This isn't a tab: \t
and this isn't a newline: \n
EOF
```

If a here-document's end marker is preceded by a hyphen, the end marker may be indented. *Only* the spaces before the end marker are deleted from the string, not those on previous lines:

```
str = <<-EOF
  Each of these lines
  starts with a pair
  of blank spaces.
  EOF
```

To delete the spaces from the beginning of each line, we need another method. The `ActiveSupport` gem (included in Rails) defines a `strip_heredoc` method that works similarly to this one:

```
class String
  def strip_heredoc
    # Find the margin whitespace on the first line
    margin = self[/\A\s*/]
    # Remove margin-sized whitespace from each line
    gsub(/\s{#{margin.size}}/,"")
  end
end
```

The amount of whitespace before the start of the first line is detected, and that amount of whitespace is then stripped off of each line. It's used in this way:

```
str = <<end.strip_heredoc
  This here-document has a "left margin"
  set by the whitespace on the first line.

  We can do inset quotations here,
  hanging indentions, and so on.
end
```

The word *end* is used naturally enough as an end marker. (This, of course, is a matter of taste. It looks like the reserved word `end` but is really just an arbitrary marker.) Many text editors use the end marker as a hint for syntax highlighting. As a result, using `<<SQL` or `<<RUBY` can make it dramatically easier to read blocks of code inside here-docs in those editors.

2.4 Finding the Length of a String

The method `length` can be used to find a string's length. A synonym is `size`:

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

2.5 Processing a Line at a Time

A Ruby string can contain newlines. For example, a file can be read into memory and stored in a single string. Strings provide an iterator, `each_line`, to process a string one line at a time:

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

The preceding code produces three lines of output:

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

Iterators (such as `each_line`) can be chained together with other iterators (such as `with_index`). Connecting function outputs and inputs in a line like this is a technique sometimes called *function composition* or *method chaining*. Instead of tracking the line number manually, `with_index` can be composed with `each_line` to produce the exact same output:

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

2.6 Processing a Character or Byte at a Time

Ruby used to treat each byte as a character, but that is no longer the case. The bytes in a string are available as an array via the `bytes` method. To process the bytes, one at a time, use the `each_byte` iterator:

```
str = "ABC"
str.each_byte {|byte| print byte, " " }
puts
# Produces output: 65 66 67
```

A character is essentially the same as a one-character string. In multibyte encodings, a one-character string may be more than one byte:

```
str = "ABC"
str.each_char {|char| print char, " " }
puts
# Produces output: A B C
```

In any version of Ruby, you can break a string into an array of one-character strings by using `scan` with a simple wildcard regular expression matching a single character:

```
str = "ABC"
chars = str.scan(/./)
chars.each {|char| print char, " " }
puts
# Produces output: A B C
```

2.7 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.

For 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 >=). Listing 2.1 shows how we do this.

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 does not 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.

Suppose that we wanted to do case-insensitive string comparisons. The built-in method `casecmp` will do this; we just have to make sure that it is used instead of the usual comparison.

Here is one way:

```
class String
  def <=>(other)
    casecmp(other)
  end
end
```

But there is a slightly easier way:

```
class String
  alias <=> casecmp
end
```

However, we haven't finished. We need to redefine == so that it will behave in the same way:

```
class String
  def ==(other)
    casecmp(other) == 0
  end
end
```

Now all string comparisons will be strictly case insensitive. Any sorting operation that depends on <=> will likewise be case insensitive.

2.8 Tokenizing a String

The `split` method parses a string and returns an array of tokenized strings. 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:

- If it is omitted, trailing null entries are suppressed.
- 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.

- 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", "", ""]
```

Similarly, the `scan` method can be used to match regular expressions or strings against a target string:

```
str = "I am a leaf on the wind..."

# A string is interpreted literally, not as a regex
arr = str.scan("a")     # ["a","a","a"]

# A regex will return all matches
arr = str.scan(/\w+/)
# ["I", "am", "a", "leaf", "on", "the", "wind"]

# A block will be passed each match, one at a time
str.scan(/\w+/) {|x| puts x }
```

The `StringScanner` class, from the standard library, is different in that it maintains state for the scan rather than doing it all at once:

```
require 'strscan'
str = "Watch how I soar!"
ss = StringScanner.new(str)
loop do
  word = ss.scan(/\w+/)    # Grab a word at a time
  break if word.nil?
  puts word
  sep = ss.scan(/\W+/)     # Grab next non-word piece
  break if sep.nil?
end
```

2.9 Formatting a String

Formatting a string 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 available with C's `sprintf` (or `printf`).

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

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 the method `%`, which does 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"
```

If a second parameter is specified, it is used as the pad string (which may possibly be truncated as needed):

```
str = "Captain Ahab"
s1 = str.ljust(20,"+")       # "Captain Ahab++++++++"
s2 = str.center(20,"-")      # "----Captain Ahab----"
s3 = str.rjust(20,"123")     # "12312312Captain Ahab"
```

2.10 Using Strings as IO Objects

Besides `sprintf` and `scanf`, there is another way to fake input/output to a string—the `StringIO` class.

Because this is a very IO-like object, we cover it in a later chapter. See Section 10.1.24, "Treating a String as a File."

2.11 Controlling Uppercase and Lowercase

Ruby's `String` class offers a rich set of methods for controlling case. This section offers an overview of these.

The `downcase` method converts a string to all lowercase. Likewise, `upcase` converts it to all uppercase. Here is an example each:

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

The `capitalize` method capitalizes the first character of a string while forcing all the remaining characters to lowercase:

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

The `swapcase` method exchanges the case of each letter in a string:

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

There is also the `casecmp` method, which acts like the <=> method but ignores case:

```
n1 = "abc".casecmp("xyz")    # -1
n2 = "abc".casecmp("XYZ")    # -1
n3 = "ABC".casecmp("xyz")    # -1
n4 = "ABC".casecmp("abc")    # 0
n5 = "xyz".casecmp("abc")    # 1
```

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

There are no built-in methods for detecting case, but this is easy to do with regular expressions, as shown in the following example:

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

if string =~ /[A-Z]/
  puts "string contains uppercase characters"
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
```

Regular expressions of this sort will only match ASCII characters. To match Unicode uppercase or lowercase characters, use a named character class, as shown here:

```
if string =~ /\p{Upper}/
  puts "string contains uppercase Unicode characters like Ü"
end
```

For more information about regular expressions, see Chapter 3, "Working with Regular Expressions."

2.12 Accessing and Assigning Substrings

In Ruby, substrings may be accessed in several different ways. Normally the bracket notation is used, as for an array, but the brackets may contain a pair of Fixnums, 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 and the length is still added in the forward direction:

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

A range may be specified. In this case, the range is taken as a range of indices into the string. Ranges may 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, as shown here:

```
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 does not):

```
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, using a `Fixnum` as the index will yield a single character (or `nil` if out of range):

```
str = "Aaron Burr"
ch1 = str[0]     # "A"
ch1 = str[1]     # "a"
ch3 = str[99]    # nil
```

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

```
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] = "B"            # "Baron Burr"
```

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

2.13 Substituting in Strings

We'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!`, their in-place counterparts.

The `sub` method substitutes 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 may be used in a substitute string. However, special variables (such as `$&` or its English equivalent `$MATCH`) may not.

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

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`.

2.14 Searching a String

Besides the techniques for accessing substrings, there are other ways of searching within strings. The `index` method returns the starting location of the specified substring, character, or regex. If the item is not 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) starts 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, shown next, 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
```

The `scan` method repeatedly scans for occurrences of a pattern. If called without a block, it returns 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 passes the successive values to the block, as shown here:

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

This code produces the following output:

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

2.15 Converting Between Characters and ASCII Codes

Single characters in Ruby are returned as one-character strings. Here is an example:

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

The `Integer` class has a method called `chr` that will convert an integer to a character. By default, integers will be interpreted as ASCII, but other encodings may be specified for values greater than 127. The `String` class has an `ord` method that is in effect an inverse:

```
str = 77.chr            # "M"
s2  = 233.chr("UTF-8")  # "é"
num = "M".ord           # 77
```

2.16 Implicit and Explicit Conversion

At first glance, the `to_s` and `to_str` methods seem confusing. They both convert an object into a string representation, don't they?

There are several differences, however. First, *any* object can in principle be converted to some kind of string representation; that is why nearly every core class has a `to_s` method. But the `to_str` method is never implemented in the core.

As a rule, `to_str` is for objects that are really very much like strings—that can "masquerade" as strings. Better yet, think of the short name `to_s` as being *explicit conversion* and the longer name `to_str` as being *implicit conversion.*

You see, the core does not *define* any `to_str` methods. But core methods do *call* `to_str` sometimes (if it exists for a given class).

The first case we might think of is a *subclass* of `String`; but, in reality, any object of a subclass of `String` already "is a" `String`, so `to_str` is unnecessary there.

In real life, `to_s` and `to_str` usually return the same value, but they don't have to do so. The implicit conversion should result in the "real string value" of the object; the explicit conversion can be thought of as a "forced" conversion.

The `puts` method calls an object's `to_s` method in order to find a string representation. This behavior might be thought of as an implicit call of an explicit conversion. The same is true for string interpolation. Here's a crude example:

```
class Helium
  def to_s
    "He"
  end

  def to_str
    "helium"
  end
end

e = Helium.new
print "Element is "
puts e                    # Element is He
puts "Element is " + e    # Element is helium
puts "Element is #{e}"    # Element is He
```

So you can see how defining these appropriately in your own classes can give you a little extra flexibility. But what about honoring the definitions of the objects passed into your methods?

For example, suppose that you have a method that is "supposed" to take a `String` as a parameter. Despite our "duck typing" philosophy, this is frequently done and is often completely appropriate. For example, the first parameter of `File.new` is "expected" to be a string.

The way to handle this is simple. When you expect a string, check for the existence of `to_str` and call it as needed:

```
def set_title(title)
  if title.respond_to? :to_str
    title = title.to_str
  end
  # ...
end
```

Now, what if an object *doesn't* respond to `to_str`? We could do several things. We could force a call to `to_str`, we could check the class to see whether it is a `String` or a subclass thereof, or we could simply keep going, knowing that if we apply some mean-ingless operation to this object, we will eventually get an `ArgumentError` anyway. A shorter way to do this is:

```
title = title.to_str if title.respond_to?(:to_str)
```

which replaces the value of `title` only if it has a `to_str` method.

Double-quoted string interpolation will implicitly call `to_s`, and is usually the easiest way to turn multiple objects into strings at once:

```
e = Helium.new
str = "Pi #{3.14} and element #{e}
# str is now "3.14 and element He"
```

Implicit conversion *would* allow you to make strings and numbers essentially equivalent. You could, for example, do this:

```
class Fixnum
  def to_str
    self.to_s
  end
end

str = "The number is " + 345     # The number is 345
```

However, I don't recommend this sort of thing. There is such a thing as "too much magic"; Ruby, like most languages, considers strings and numbers to be different, and I believe that most conversions should be explicit for the sake of clarity.

There is nothing *magical* about the `to_str` method. It is intended to return a string, but if you code your own, it is your responsibility to see that it does.

2.17 Appending an Item onto a String

The append operator (<<) can be used to append a string 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"
```

2.18 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 removes 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' conceptions of what a newline is. On systems such as UNIX, the newline character is represented internally as a linefeed (`\n`). On others, such as 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 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 may not always be present, the `chomp` method may be a better alternative:

```
str = "abcxyz"
str2 = "123\n"
str3 = "123\r"
str4 = "123\r\n"
s1 = str.chomp          # "abcxyz"
s2 = str2.chomp         # "123"
# With the default record separator, \r and \r\n are removed
# as well as \n
s3 = str3.chomp         # "123"
s4 = str4.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, as shown here:

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

2.19 Trimming Whitespace from a String

The `strip` method removes whitespace from the beginning and end of a string, whereas its counterpart, `strip!`, modifies 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 or end of a string, we can use the `lstrip` and `rstrip` methods:

```
str = "  abc  "
s2 = str.lstrip       # "abc  "
s3 = str.rstrip       # "  abc"
```

There are in-place variants (`rstrip!` and `lstrip!`) also.

2.20 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. Here is an example:

```
etc = "Etc. "*3                        # "Etc. Etc. Etc. "
ruler = "+" + ("."*4+"5"+"."*4+"+")*3
# "+....5....+....5....+....5....+"
```

2.21 Embedding Expressions within Strings

The `#{}` notation makes embedding expressions within strings easy. We need not worry about converting, appending, and concatenating; we 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."
```

Bear in mind that full statements can also be used inside the braces. The last evaluated expression will be the one returned.

```
str = "The answer is #{ def factorial(n)
                          n==0 ? 1 : n*factorial(n-1)
                        end

                        answer = factorial(3) * 7}, of course."
# The answer is 42, of course.
```

There are some shortcuts for global, class, and instance variables, in which case the braces can be dispensed with:

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

Note that this technique is not applicable for single-quoted strings (because their contents are not expanded), but it does work for double-quoted here-documents and regular expressions.

2.22 Delayed Interpolation of Strings

Sometimes we might want to delay the interpolation of values into a string. There is no perfect way to do this.

A naive approach is to store a single-quoted string and then evaluate it:

```
str = '#{name} is my name, and #{nation} is my nation.'
name, nation = "Stephen Dedalus", "Ireland"
s1  = eval('"' + str + '"')
# Stephen Dedalus is my name, and Ireland is my nation.
```

However, using `eval` is almost always the worst option. Any time you use `eval`, you are opening yourself up to many problems, including extremely slow execution and unexpected security vulnerabilities, so it should be avoided if at all possible.

A much less dangerous way is to use a block:

```
str = proc do |name, nation|
  "#{name} is my name, and #{nation} is my nation."
end
s2 = str.call("Gulliver Foyle", "Terra")
# Gulliver Foyle is my name, and Terra is my nation.
```

2.23 Parsing Comma-Separated Data

The use of comma-delimited data is common in computing. It is a kind of "lowest common denominator" of data interchange 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 that all strings are enclosed in quotes. We further assume that all characters are escaped as necessary (commas and quotes inside strings, for example).

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 produces the following output:

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

For a more heavy-duty solution, refer to the CSV library (which is a standard library) .

2.24 Converting Strings to Numbers (Decimal and Otherwise)

Basically there are two ways to convert strings to numbers: the `Kernel` method `Integer` and `Float` and the `to_i` and `to_f` methods of `String`. (Capitalized method names such as `Integer` are usually reserved for special data conversion functions like this.)

The simple case is trivial, and these are equivalent:

```
x = "123".to_i          # 123
y = Integer("123")      # 123
```

When a string is not a valid number, however, their behaviors differ:

```
x = "junk".to_i         # silently returns 0
y = Integer("junk")     # error
```

`to_i` stops converting when it reaches a non-numeric character, but `Integer` raises an error:

```
x = "123junk".to_i      # 123
y = Integer("123junk")  # error
```

Both allow leading and trailing whitespace:

```
x = " 123 ".to_i        # 123
y = Integer(" 123 ")    # 123
```

Floating point conversion works much the same way:

```
x = "3.1416".to_f      # 3.1416
y = Float("2.718")     # 2.718
```

Both conversion methods honor scientific notation:

```
x = Float("6.02e23")   # 6.02e23
y = "2.9979246e5".to_f # 299792.46
```

`to_i` and `Integer` also differ in how they handle different bases. The default, of course, is decimal or base ten; but we can work in other bases also. (The same is not true for floating point.)

When talking about converting between numeric bases, strings always are involved. After all, an integer is an integer, and they are all stored in binary.

Base conversion, therefore, always means converting to or from some kind of string. Here, we're looking at converting *from* a string. (For the reverse, see Section 5.18, "Performing Base Conversions," and Section 5.5, "Formatting Numbers for Output.")

When a number appears in program text as a literal numeric constant, it may have a "tag" in front of it to indicate base. These tags are 0b for binary, a simple 0 for octal, and 0x for hexadecimal.

These tags are honored by the `Integer` method but *not* by the `to_i` method, as demonstrated here:

```
x = Integer("0b111")     # binary      - returns 7
y = Integer("0111")      # octal       - returns 73
z = Integer("0x111")     # hexadecimal - returns 291

x = "0b111".to_i         # 0
y = "0111".to_i          # 0
z = "0x111".to_i         # 0
```

`to_i`, however, allows an optional parameter to indicate the base. Typically, the only meaningful values are 2, 8, 10 (the default), and 16. However, tags are not recognized even with the base parameter:

```
x = "111".to_i(2)        # 7
y = "111".to_i(8)        # octal       - returns 73
z = "111".to_i(16)       # hexadecimal - returns 291

x = "0b111".to_i         # 0
y = "0111".to_i          # 0
z = "0x111".to_i         # 0
```

Because of the "standard" behavior of these methods, a digit that is inappropriate for the given base will be treated differently:

```
x = "12389".to_i(8)      # 123    (8 is ignored)
y = Integer("012389")    # error  (8 is illegal)
```

Although it might be of limited usefulness, `to_i` handles bases up to 36, using all letters of the alphabet. (This may remind you of the Base64 encoding; for information on that, see Section 2.37, "Encoding and Decoding Base64 Strings.")

```
x = "123".to_i(5)        # 66
y = "ruby".to_i(36)      # 1299022
```

It's also possible to use the `scanf` standard library to convert character strings to numbers. This library adds a `scanf` method to `Kernel`, to `IO`, and to `String`:

```
str = "234 234 234"
x, y, z = str.scanf("%d %o %x")    # 234, 156, 564
```

The `scanf` methods implement all the meaningful functionality of their C counterparts: `scanf`, `sscanf`, and `fscanf`. However, `scanf` does not handle binary.

2.25 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 was commonly seen in Usenet posts; 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 12* on the day before the premiere.

The encoding method consists simply of "rotating" a string through the alphabet, so that *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 example 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: Naanxva qbrfa'g trg xvyyrq."
puts episode2.rot13
```

2.26 Encrypting Strings

There are times when we don't want strings to be immediately legible. For example, passwords should not be stored in plaintext, no matter how tight the file permissions are.

The standard method `crypt` uses the standard function of the same name to DES-encrypt a string. It takes a "salt" value as a parameter (similar to the seed value for a random number generator). On non-UNIX platforms, this parameter may be different.

A trivial application for this follows, 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
  puts "What are you, an orc?"
end
```

It is worth noting that you should never use encryption to store passwords. Instead, employ *password hashing* using a hashing algorithm designed specifically for passwords, such as bcrypt. Additionally, never rely on encryption of this nature for communications with a server-side web application. To secure web applications, use the HTTPS protocol and Secure Sockets Layer (SSL) to encrypt all traffic. Of course, you could still use encryption on the server side, but for a different reason—to protect the data as it is stored rather than during transmission.

2.27 Compressing Strings

The `Zlib` library provides a way of compressing and decompressing strings and files.

Why might we want to compress strings in this way? Possibly to make database I/O faster, to optimize network usage, or even to obscure stored strings so that they are not easily read.

The `Deflate` and `Inflate` classes have class methods named `deflate` and `inflate`, respectively. The `deflate` method (which obviously compresses) has an extra parameter to specify the style of compression. The styles show a typical trade-off between compression quality and speed; `BEST_COMPRESSION` results in a smaller compressed string, but compression is relatively slow; `BEST_SPEED` compresses faster but does not compress as much. The default (`DEFAULT_COMPRESSION`) is typically somewhere in between in both size and speed.

```
require 'zlib'
include Zlib

long_string = ("abcde"*71 + "defghi"*79 + "ghijkl"*113)*371
# long_string has 559097 characters

s1 = Deflate.deflate(long_string,BEST_SPEED)        # 4188 chars
s2 = Deflate.deflate(long_string)                   # 3568 chars
s3 = Deflate.deflate(long_string,BEST_COMPRESSION)  # 2120 chars
```

Informal experiments suggest that the speeds vary by a factor of two, and the compression amounts vary inversely by the same amount. Speed and compression are greatly dependent on the contents of the string. Speed, of course, also is affected by hardware.

Be aware that there is a "break-even" point below which it is essentially useless to compress a string (unless you are trying to make the string unreadable). Below this point, the overhead of compression may actually result in a *longer* string.

2.28 Counting Characters in Strings

The `count` method counts the number of occurrences of any of a set of specified characters:

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

The string parameter is like a 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
```

2.29 Reversing a String

A string may be reversed simply by using the `reverse` method (or its in-place counterpart `reverse!`):

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

Suppose that you want to reverse the word order (rather than character order). You can use `String#split`, which gives you an array of words. The `Array` class also has a `reverse` method, so you can then reverse the array and join to make a new string:

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

2.30 Removing Duplicate Characters

Runs of duplicate characters may be removed using the `squeeze` method. If a parameter is specified, only those characters will be squeezed.

```
s1 = "bookkeeper"
s2 = s1.squeeze         # "bokeper"
s3 = "Hello..."
s4 = s3.squeeze         # "Helo."
s5 = s3.squeeze(".")    # "Hello."
```

This parameter follows the same rules as the one for the `count` method (see Section 2.28, "Counting Characters in Strings," earlier in this chapter); that is, it understands the hyphen and the caret.

There is also a `squeeze!` method.

2.31 Removing Specific Characters

The `delete` method removes 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"
```

This parameter follows the same rules as the one for the `count` method (see Section 2.28, "Counting Characters in Strings," earlier in this chapter); that is, it understands the hyphen and the caret.

There is also a `delete!` method.

2.32 Printing Special Characters

The `dump` method (like `inspect`) provides explicit printable representations of characters that may ordinarily be invisible or print differently. Here is an example:

```
s1 = "Listen" << “\007\007\007” # 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: \"
```

2.33 Generating Successive Strings

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

Ruby provides the method `succ` (successor) 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 applies `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 is not used frequently, and you use it at your own risk. In addition, there is no corresponding "predecessor" function.

2.34 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).

The `zlib` library, created by Ueno Katsuhiro, enables you to do this.

The method `crc32` computes a CRC given a string as a parameter:

```
require 'zlib'
include Zlib
crc = crc32("Hello")            # 4157704578
crc = crc32(" world!",crc)      # 461707669
crc = crc32("Hello world!")     # 461707669 (same as above)
```

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.

2.35 Calculating the SHA-256 Hash of a String

The `Digest::SHA256` class produces a 256-bit *hash* or *message digest* of a string of arbitrary length. This hashing function is one-way, and does not allow for the discovery of the original message from the digest. There are also `MD5`, `SHA384`, and `SHA512` classes inside `Digest` for each of those algorithms.

The most commonly used class method is `hexdigest`, but there are also `digest` and `base64digest`. They all accept a string containing the message and return the digest as a string, as shown here:

```
require 'digest'
Digest::SHA256.hexdigest("foo")[0..20]    # "2c26b46b68f"
Digest::SHA256.base64digest("foo")[0..20] # "LCa0a2j/xo/"
Digest::SHA256.digest(“foo")[0..5]        # ",&\xB4kh\xFF"
```

Although the `digest` method provides a 64-byte string containing the 512-bit digest, the `hexdigest` method is actually the most useful. It provides the digest as an ASCII string of 64 hex characters representing the 64 bytes.

Instances and the `update` method allow the hash to be built incrementally, perhaps because the data is coming from a streaming source:

```
secret = Digest::SHA256.new
source.each { |chunk| secret.update(chunk) }
```

Repeated calls are equivalent to a single call with concatenated arguments:

```
# These two statements...
cryptic.update("Data...")
cryptic.update(" and more data.")

# ...are equivalent to this one.
cryptic.update("Data... and more data.”)

cryptic.hexdigest[0..20] # "50605ba0a90"
```

2.36 Calculating the Levenshtein Distance Between Two Strings

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

The Levenshtein distance is the minimum number of modifications needed to change one string into another, using three basic modification operations: *del*(-etion), *ins*(-ertion), and *sub*(-stitution). 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 (shown in Listing 2.2) 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).

Listing 2.2 The Levenshtein distance

```
class String

  def levenshtein(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
    # Levenshtein distance between the strings
    dm[other.length][self.length]
  end

end


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


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


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

Now that we have the Levenshtein distance defined, it's conceivable that we could define a `similar?` method, giving it a threshold for similarity. Here is an example:

```
class String

  def similar?(other, thresh=2)
    self.levenshtein(other) < thresh
  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 `levenshtein` method. We have omitted these for simplicity.

2.37 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, images and fonts stored inline inside CSS files are encoded with Base64.

The easiest way to do a Base64 encode/decode is to use the built-in `Base64` module. The `Base64` class has an `encode64` method that returns a Base64 string (with a newline appended). It also has the method `decode64`, which changes the string back to its original bytes, as shown here:

```
require "base64"
str = "\xAB\xBA\x02abdce"
encoded  = Base64.encode64(str)     # "q7oCYWJkY2U=\n"
original = Base64.decode64(encoded) # "\xAB\xBA\x02abdce"
```

2.38 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 here do these operations:

```
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 is not smart enough to handle backspaces.

2.39 Wrapping Lines of Text

Occasionally we may want to take long text lines and print them within margins of our own choosing. The code fragment shown here accomplishes this, splitting only on word boundaries and honoring tabs (but not honoring backspaces or preserving tabs):

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

max = 20

line = 0
out = [""]
```

```
input = str.gsub(/\n/," ")
words = input.split(" ")

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

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

The `ActiveSupport` gem includes similar functionality in a method named `word_wrap`, along with many other string manipulation helpers. Search for it online.

2.40 Conclusion

In this chapter, we have seen the basics of representing strings (both single-quoted strings and double-quoted strings). We've seen how to interpolate expressions into double-quoted strings, and how the double quotes also allow certain special characters to be inserted with escape sequences. We've seen the `%q` and `%Q` forms, which permit us to choose our own delimiters for convenience. Finally, we've seen the here-document syntax, carried over from older contexts such as UNIX shells.

This chapter has demonstrated all the important operations a programmer wants to perform on a string, including concatenation, searching, extracting substrings, tokenizing, and much more. We have seen how to iterate over a string by line or by byte. We have seen how to transform a string to and from a coded form such as Base64 or compressed form.

It's time now to move on to a related topic—regular expressions. Regular expressions are a powerful tool for detecting patterns in strings. We'll cover this topic in the next chapter.