Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

1. Strings

Ruby is a programmer-friendly language. If you are already familiar with object oriented programming, Ruby should quickly become second nature. If you've struggled with learning object-oriented programming or are not familiar with it, Ruby should make more sense to you than other object-oriented languages because Ruby's methods are consistently named, concise, and generally act the way you expect.

Throughout this book, we demonstrate concepts through interactive Ruby sessions. Strings are a good place to start because not only are they a useful data type, they're easy to create and use. They provide a simple introduction to Ruby, a point of comparison between Ruby and other languages you might know, and an approachable way to introduce important Ruby concepts like duck typing (see Recipe 1.12), open classes (demonstrated in Recipe 1.10), symbols (Recipe 1.7), and even Ruby gems (Recipe 1.20).

If you use Mac OS X or a Unix environment with Ruby installed, go to your command line right now and type `irb`. If you're using Windows, you can download and install the One-Click Installer from http://rubyforge.org/projects/rubyinstaller/, and do the same from a command prompt (you can also run the `fxri` program, if that's more comfortable for you). You've now entered an interactive Ruby shell, and you can follow along with the code samples in most of this book's recipes.

Strings in Ruby are much like strings in other dynamic languages like Perl, Python and PHP. They're not too much different from strings in Java and C. Ruby strings are dynamic, mutable, and flexible. Get started with strings by typing this line into your interactive Ruby session:

```
string = "My first string"
```

You should see some output that looks like this:

```
=> "My first string"
```

You typed in a Ruby expression that created a string "My first string", and assigned it to the variable `string`. The value of that expression is just the new value of `string`, which is what your interactive Ruby session printed out on the right side of the arrow. Throughout this book, we'll represent this kind of interaction in the following form:[1]

[1] Yes, this was covered in the Preface, but not everyone reads the Preface.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
string = "My first string"                # => "My first string"
```

In Ruby, everything that can be assigned to a variable is an object. Here, the variable `string` points to an object of class `String`. That class defines over a hundred built-in methods: named pieces of code that examine and manipulate the string. We'll explore some of these throughout the chapter, and indeed the entire book. Let's try out one now: `String#length`, which returns the number of bytes in a string. Here's a Ruby method call:

```
string.length                             # => 15
```

Many programming languages make you put parentheses after a method call:

```
string.length()                           # => 15
```

In Ruby, parentheses are almost always optional. They're especially optional in this case, since we're not passing any arguments into `String#length`. If you're passing arguments into a method, it's often more readable to enclose the argument list in parentheses:

```
string.count 'i'                          # => 2 # "i" occurs twice.
string.count('i')                         # => 2
```

The return value of a method call is itself an object. In the case of `String#length`, the return value is the number 15, an instance of the `Fixnum` class. We can call a method on this object as well:

```
string.length.next                        # => 16
```

Let's take a more complicated case: a string that contains non-ASCII characters. This string contains the French phrase "il était une fois," encoded as UTF-8:[2]

[2] "\xc3\xa9" is a Ruby string representation of the UTF-8 encoding of the Unicode character é.

```
french_string = "il \xc3\xa9tait une fois"   # => "il \303\251tait une fois"
```

Many programming languages (notably Java) treat a string as a series of characters. Ruby treats a string as a series of bytes. The French string contains 14 letters and 3 spaces, so you might think Ruby would say the length of the string is 17. But one of the letters (the e with acute accent) is represented as two bytes, and that's what Ruby counts:

```
french_string.length                      # => 18
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

For more on handling different encodings, see Recipe 1.14 and Recipe 11.12. For more on this specific problem, see Recipe 1.8

You can represent special characters in strings (like the binary data in the French string) with string escaping. Ruby does different types of string escaping depending on how you create the string. When you enclose a string in double quotes, you can encode binary data into the string (as in the French example above), and you can encode newlines with the code "\n", as in other programming languages:

```
puts "This string\ncontains a newline"
# This string
# contains a newline
```

When you enclose a string in single quotes, the only special codes you can use are "\'" to get a literal single quote, and "\\" to get a literal backslash:

```
puts 'it may look like this string contains a newline\nbut it doesn\'t'
# it may look like this string contains a newline\nbut it doesn't

puts 'Here is a backslash: \\'
# Here is a backslash: \
```

This is covered in more detail in Recipe 1.5. Also see Recipes 1.2 and 1.3 for more examples of the more spectacular substitutions double-quoted strings can do.

Another useful way to initialize strings is with the "here documents" style:

```
long_string = <<EOF
Here is a long string
With many paragraphs
EOF
# => "Here is a long string\nWith many paragraphs\n"

puts long_string
# Here is a long string
# With many paragraphs
```

Like most of Ruby's built-in classes, Ruby's strings define the same functionality in several different ways, so that you can use the idiom you prefer. Say you want to get a substring of a larger string (as in Recipe 1.13). If you're an object-oriented programming purist, you can use the `String#slice` method:

```
string                                  # => "My first string"
string.slice(3, 5)                      # => "first"
```

But if you're coming from C, and you think of a string as an array of bytes, Ruby can accommodate you. Selecting a single byte from a string returns that byte as a number.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
string.chr + string.chr + string.chr + string.chr + string.chr
# => "first"
```

And if you come from Python, and you like that language's slice notation, you can just as easily chop up the string that way:

```
string[3, 5]                            # => "first"
```

Unlike in most programming languages, Ruby strings are mutable: you can change them after they are declared. Below we see the difference between the methods `String#upcase` and `String#upcase!`:

```
string.upcase                           # => "MY FIRST STRING"
string                                  # => "My first string"
string.upcase!                          # => "MY FIRST STRING"
string                                  # => "MY FIRST STRING"
```

This is one of Ruby's syntactical conventions. "Dangerous" methods (generally those that modify their object in place) usually have an exclamation mark at the end of their name. Another syntactical convention is that *predicates*, methods that return a true/false value, have a question mark at the end of their name (as in some varieties of Lisp):

```
string.empty?                           # => false
string.include? 'MY'                    # => true
```

This use of English punctuation to provide the programmer with information is an example of Matz's design philosophy: that Ruby is a language primarily for humans to read and write, and secondarily for computers to interpret.

An interactive Ruby session is an indispensable tool for learning and experimenting with these methods. Again, we encourage you to type the sample code shown in these recipes into an `irb` or `fxri` session, and try to build upon the examples as your knowledge of Ruby grows.

Here are some extra resources for using strings in Ruby:

- You can get information about any built-in Ruby method with the `ri` command; for instance, to see more about the `String#upcase!` method, issue the command `ri "String#upcase!"` from the command line.
- "why the lucky stiff" has written an excellent introduction to installing Ruby, and using `irb` and `ri`: http://poignantguide.net/ruby/expansion-pak-1.html
- For more information about the design philosophy behind Ruby, read an interview with Yukihiro "Matz" Matsumoto, creator of Ruby: http://www.artima.com/intv/ruby.html

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 1.1. Building a String from Parts

Problem

You want to iterate over a data structure, building a string from it as you do.

Solution

There are two efficient solutions. The simplest solution is to start with an empty string, and repeatedly append substrings onto it with the << operator:

```
hash = { "key1" => "val1", "key2" => "val2" }
string = ""
hash.each { |k,v| string << "#{k} is #{v}\n" }
puts string
# key1 is val1
# key2 is val2
```

This variant of the simple solution is slightly more efficient, but harder to read:

```
string = ""
hash.each { |k,v| string << k << " is " << v << "\n" }
```

If your data structure is an array, or easily transformed into an array, it's usually more efficient to use `Array#join`:

```
puts hash.keys.join("\n") + "\n"
# key1
# key2
```

Discussion

In languages like Python and Java, it's very inefficient to build a string by starting with an empty string and adding each substring onto the end. In those languages, strings are immutable, so adding one string to another builds an entirely new string. Doing this multiple times creates a huge number of intermediary strings, each of which is only used as a stepping stone to the next string. This wastes time and memory.

In those languages, the most efficient way to build a string is always to put the substrings into an array or another mutable data structure, one that expands dynamically rather than by implicitly creating entirely new objects. Once you're done processing the substrings, you get a single string with the equivalent of Ruby's `Array#join`. In Java, this is the purpose of the `StringBuffer` class.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

In Ruby, though, strings are just as mutable as arrays. Just like arrays, they can expand as needed, without using much time or memory. The fastest solution to this problem in Ruby is usually to forgo a holding array and tack the substrings directly onto a base string. Sometimes using `Array#join` is faster, but it's usually pretty close, and the `<<` construction is generally easier to understand.

If efficiency is important to you, don't build a new string when you can append items onto an existing string. Constructs like `str << 'a' + 'b'` or `str << "#{var1} #{var2}"` create new strings that are immediately subsumed into the larger string. This is exactly what you're trying to avoid. Use `str << var1 <<''<< var2` instead.

On the other hand, you shouldn't modify strings that aren't yours. Sometimes safety requires that you create a new string. When you define a method that takes a string as an argument, you shouldn't modify that string by appending other strings onto it, unless that's really the point of the method (and unless the method's name ends in an exclamation point, so that callers know it modifies objects in place).

Another caveat: `Array#join` does not work precisely the same way as repeated appends to a string. `Array#join` accepts a separator string that it inserts *between* every two elements of the array. Unlike a simple string-building iteration over an array, it will not insert the separator string after the last element in the array. This example illustrates the difference:

```
data = ['1', '2', '3']
s = ''
data.each { |x| s << x << ' and a '}
s                                       # => "1 and a 2 and a 3 and a "
data.join(' and a ')                    # => "1 and a 2 and a 3"
```

To simulate the behavior of `Array#join` across an iteration, you can use `Enumerable#each_with_index` and omit the separator on the last index. This only works if you know how long the `Enumerable` is going to be:

```
s = ""
data.each_with_index { |x, i| s << x; s << "|" if i < data.length-1 }
s                                       # => "1|2|3"
```

Recipe 1.2. Substituting Variables into Strings

Problem

You want to create a string that contains a representation of a Ruby variable or expression.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Within the string, enclose the variable or expression in curly brackets and prefix it with a hash character.

```
number = 5
"The number is #{number}."                      # => "The number is 5."
"The number is #{5}."                           # => "The number is 5."
"The number after #{number} is #{number.next}."
# => "The number after 5 is 6."
"The number prior to #{number} is #{number-1}."
# => "The number prior to 5 is 4."
"We're ##{number}!"                             # => "We're #5!"
```

Discussion

When you define a string by putting it in double quotes, Ruby scans it for special substitution codes. The most common case, so common that you might not even think about it, is that Ruby substitutes a single newline character every time a string contains slash followed by the letter n ("\n").

Ruby supports more complex string substitutions as well. Any text kept within the brackets of the special marker *#{}* (that is, #{text in here}) is interpreted as a Ruby expression. The result of that expression is substituted into the string that gets created. If the result of the expression is not a string, Ruby calls its `to_s` method and uses that instead.

Once such a string is created, it is indistinguishable from a string created without using the string interpolation feature:

```
"#{number}" == '5'                              # => true
```

You can use string interpolation to run even large chunks of Ruby code inside a string. This extreme example defines a class within a string; its result is the return value of a method defined in the class. You should never have any reason to do this, but it shows the power of this feature.

```
%{Here is #{class InstantClass
   def bar
      "some text"
    end
 end
 InstantClass.new.bar
}.}
# => "Here is some text."
```

The code run in string interpolations runs in the same context as any other Ruby code in the same location. To take the example above, the `InstantClass` class has now been defined like any other class, and can be used outside the string that defines it.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If a string interpolation calls a method that has side effects, the side effects are triggered. If a string definition sets a variable, that variable is accessible afterwards. It's bad form to rely on this behavior, but you should be aware of it:

```
"I've set x to #{x = 5; x += 1}."      # => "I've set x to 6."
x                                      # => 6
```

To avoid triggering string interpolation, escape the hash characters or put the string in single quotes.

```
"\#{foo}"                          # => "\#{foo}"
'#{foo}'                           # => "\#{foo}"
```

The "here document" construct is an alternative to the `%{}` construct, which is sometimes more readable. It lets you define a multiline string that only ends when the Ruby parser encounters a certain string on a line by iteself:

```
name = "Mr. Lorum"
email = <<END
Dear #{name},

Unfortunately we cannot process your insurance claim at this
time. This is because we are a bakery, not an insurance company.

Signed,
 Nil, Null, and None
 Bakers to Her Majesty the Singleton
END
```

Ruby is pretty flexible about the string you can use to end the "here document":

```
<<end_of_poem
There once was a man from Peru
Whose limericks stopped on line two
end_of_poem
# => "There once was a man from Peru\nWhose limericks stopped on line two\n"
```

See Also

- You can use the technique described in Recipe 1.3, "Substituting Variables into an Existing String," to define a template string or object, and substitute in variables later

Recipe 1.3. Substituting Variables into an Existing String

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to create a string that contains Ruby expressions or variable substitutions, without actually performing the substitutions. You plan to substitute values into the string later, possibly multiple times with different values each time.

Solution

There are two good solutions: `printf`-style strings, and ERB templates.

Ruby supports a `printf`-style string format like C's and Python's. Put `printf` directives into a string and it becomes a template. You can interpolate values into it later using the modulus operator:

```
template = 'Oceania has always been at war with %s.'
template % 'Eurasia'  # => "Oceania has always been at war with Eurasia."
template % 'Eastasia' # => "Oceania has always been at war with Eastasia."

'To 2 decimal places: %.2f' % Math::PI       # => "To 2 decimal places: 3.14"
'Zero-padded: %.5d' % Math::PI               # => "Zero-padded: 00003"
```

An ERB template looks something like JSP or PHP code. Most of it is treated as a normal string, but certain control sequences are executed as Ruby code. The control sequence is replaced with either the output of the Ruby code, or the value of its last expression:

```
require 'erb'

template = ERB.new %q{Chunky <%= food %>!}
food = "bacon"
template.result(binding)                     # => "Chunky bacon!"
food = "peanut butter"
template.result(binding)                     # => "Chunky peanut butter!"
```

You can omit the call to `Kernel#binding` if you're not in an `irb` session:

```
puts template.result
# Chunky peanut butter!
```

You may recognize this format from the `.rhtml` files used by Rails views: they use ERB behind the scenes.

Discussion

An ERB template can reference variables like `food` before they're defined. When you call `ERB#result`, or `ERB#run`, the template is executed according to the current values of those variables.

Like JSP and PHP code, ERB templates can contain loops and conditionals. Here's a more sophisticated template:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
template = %q{
<% if problems.empty? %>
  Looks like your code is clean!
<% else %>
  I found the following possible problems with your code:
  <% problems.each do |problem, line| %>
    * <%= problem %> on line <%= line %>
  <% end %>
<% end %>}.gsub(/^\s+/, '')
template = ERB.new(template, nil, '<>')

problems = [["Use of is_a? instead of duck typing", 23],
            ["eval() is usually dangerous", 44]]
template.run(binding)
# I found the following possible problems with your code:
# * Use of is_a? instead of duck typing on line 23
# * eval() is usually dangerous on line 44

problems = []
template.run(binding)
# Looks like your code is clean!
```

ERB is sophisticated, but neither it nor the `printf`-style strings look like the simple Ruby string substitutions described in Recipe 1.2. There's an alternative. If you use single quotes instead of double quotes to define a string with substitutions, the substitutions won't be activated. You can then use this string as a template with `eval`:

```
class String
  def substitute(binding=TOPLEVEL_BINDING)
    eval(%{"#{self}"}, binding)
  end
end

template = %q{Chunky #{food}!}                # => "Chunky \#{food}!"

food = 'bacon'
template.substitute(binding)                  # => "Chunky bacon!"
food = 'peanut butter'
template.substitute(binding)                  # => "Chunky peanut butter!"
```

You must be very careful when using `eval`: if you use a variable in the wrong way, you could give an attacker the ability to run arbitrary Ruby code in your `eval` statement. That won't happen in this example since any possible value of `food` gets stuck into a string definition before it's interpolated:

```
food = '#{system("dir")}'
puts template.substitute(binding)
# Chunky #{system("dir")}!
```

See Also

- This recipe gives basic examples of ERB templates; for more complex examples, see the documentation of the ERB class (http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/classes/ERB.html)
- Recipe 1.2, "Substituting Variables into Strings"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

- Recipe 10.12, "Evaluating Code in an Earlier Context," has more about `Binding` objects

Recipe 1.4. Reversing a String by Words or Characters

Problem

The letters (or words) of your string are in the wrong order.

Solution

To create a new string that contains a reversed version of your original string, use the `reverse` method. To reverse a string in place, use the `reverse!` method.

```
s = ".sdrawkcab si gnirts sihT"
s.reverse                          # => "This string is backwards."
s                                  # => ".sdrawkcab si gnirts sihT"

s.reverse!                         # => "This string is backwards."
s                                  # => "This string is backwards."
```

To reverse the order of the words in a string, split the string into a list of whitespaceseparated words, then join the list back into a string.

```
s = "order. wrong the in are words These"
s.split(/(\s+)/).reverse!.join('')   # => "These words are in the wrong order."
s.split(/\b/).reverse!.join('')      # => "These words are in the wrong. order"
```

Discussion

The `String#split` method takes a regular expression to use as a separator. Each time the separator matches part of the string, the portion of the string before the separator goes into a list. `split` then resumes scanning the rest of the string. The result is a list of strings found between instances of the separator. The regular expression `/(\s+)/` matches one or more whitespace characters; this splits the string on word boundaries, which works for us because we want to reverse the order of the words.

The regular expression `\b` matches a word boundary. This is not the same as matching whitespace, because it also matches punctuation. Note the difference in punctuation between the two final examples in the Solution.

Because the regular expression `/(\s+)/` includes a set of parentheses, the separator strings themselves are included in the returned list. Therefore, when we join the strings back together, we've preserved whitespace. This example shows the difference between including the parentheses and omitting them:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
"Three little words".split(/\s+/)   # => ["Three", "little", "words"]
"Three little words".split(/(\s+)/)
# => ["Three", " ", "little", " ", "words"]
```

See Also

- Recipe 1.9, "Processing a String One Word at a Time," has some regular expressions for alternative definitions of "word"
- Recipe 1.11, "Managing Whitespace"
- Recipe 1.17, "Matching Strings with Regular Expressions"

Recipe 1.5. Representing Unprintable Characters

Problem

You need to make reference to a control character, a strange UTF-8 character, or some other character that's not on your keyboard.

Solution

Ruby gives you a number of escaping mechanisms to refer to unprintable characters. By using one of these mechanisms within a double-quoted string, you can put any binary character into the string.

You can reference any any binary character by encoding its octal representation into the format "\ooo", or its hexadecimal representation into the format "\xoo".

```
octal = "\000\001\010\020"
octal.each_byte { |x| puts x }
# 0
# 1
# 8
# 16

hexadecimal = "\x00\x01\x10\x20"
hexadecimal.each_byte { |x| puts x }
# 0
# 1
# 16
# 32
```

This makes it possible to represent UTF-8 characters even when you can't type them or display them in your terminal. Try running this program, and then opening the generated file *smiley.html* in your web browser:

```
open('smiley.html', 'wb') do |f|
  f << '<meta http-equiv="Content-Type" content="text/html;charset=UTF-8">'
  f << "\xe2\x98\xBA"
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The most common unprintable characters (such as newline) have special mneumonic aliases consisting of a backslash and a letter.

```
"\a" == "\x07" # => true # ASCII 0x07 = BEL (Sound system bell)
"\b" == "\x08" # => true # ASCII 0x08 = BS (Backspace)
"\e" == "\x1b" # => true # ASCII 0x1B = ESC (Escape)
"\f" == "\x0c" # => true # ASCII 0x0C = FF (Form feed)
"\n" == "\x0a" # => true # ASCII 0x0A = LF (Newline/line feed)
"\r" == "\x0d" # => true # ASCII 0x0D = CR (Carriage return)
"\t" == "\x09" # => true # ASCII 0x09 = HT (Tab/horizontal tab)
"\v" == "\x0b" # => true # ASCII 0x0B = VT (Vertical tab)
```

Discussion

Ruby stores a string as a sequence of bytes. It makes no difference whether those bytes are printable ASCII characters, binary characters, or a mix of the two.

When Ruby prints out a human-readable string representation of a binary character, it uses the character's `\xxx` octal representation. Characters with special `\x` mneumonics are printed as the mneumonic. Printable characters are output as their printable representation, even if another representation was used to create the string.

```
"\x10\x11\xfe\xff"            # => "\020\021\376\377"
"\x48\145\x6c\x6c\157\x0a"    # => "Hello\n"
```

To avoid confusion with the mneumonic characters, a literal backslash in a string is represented by two backslashes. For instance, the two-character string consisting of a backslash and the 14th letter of the alphabet is represented as "\\n".

```
"\\".size                     # => 1
"\\" == "\x5c"                # => true
"\\n"[0] == ?\\               # => true
"\\n"[1] == ?n                # => true
"\\n" =~ /\n/                 # => nil
```

Ruby also provides special shortcuts for representing keyboard sequences like Control-C. `"\C-_x_"` represents the sequence you get by holding down the control key and hitting the x key, and `"\M-_x_"` represents the sequence you get by holding down the Alt (or Meta) key and hitting the *x* key:

```
"\C-a\C-b\C-c" #              => "\001\002\003"
"\M-a\M-b\M-c" #              => "\341\342\343"
```

Shorthand representations of binary characters can be used whenever Ruby expects a character. For instance, you can get the decimal byte number of a special character by prefixing it with ?, and you can use shorthand representations in regular expression character ranges.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
?\C-a                                  # => 1
?\M-z                                  # => 250

contains_control_chars = /[\C-a-\C-^]/
'Foobar' =~ contains_control_chars     # => nil
"Foo\C-zbar" =~ contains_control_chars # => 3

contains_upper_chars = /[\x80-\xff]/
'Foobar' =~ contains_upper_chars       # => nil
"Foo\212bar" =~ contains_upper_chars   # => 3
```

Here's a sinister application that scans logged keystrokes for special characters:

```
def snoop_on_keylog(input)
  input.each_byte do |b|
    case b
      when ?\C-c; puts 'Control-C: stopped a process?'
      when ?\C-z; puts 'Control-Z: suspended a process?'
      when ?\n; puts 'Newline.'
      when ?\M-x; puts 'Meta-x: using Emacs?'
    end
  end
end

snoop_on_keylog("ls -ltR\003emacsHello\012\370rot13-other-window\012\032")
# Control-C: stopped a process?
# Newline.
# Meta-x: using Emacs?
# Newline.
# Control-Z: suspended a process?
```

Special characters are only interpreted in strings delimited by double quotes, or strings created with `%{}` or `%Q{}`. They are not interpreted in strings delimited by single quotes, or strings created with `%q{}`. You can take advantage of this feature when you need to display special characters to the end-user, or create a string containing a lot of backslashes.

```
puts "foo\tbar"
# foo     bar
puts %{foo\tbar}
# foo     bar
puts %Q{foo\tbar}
# foo     bar

puts 'foo\tbar'
# foo\tbar
puts %q{foo\tbar}
# foo\tbar
```

If you come to Ruby from Python, this feature can take advantage of you, making you wonder why the special characters in your single-quoted strings aren't treated as special. If you need to create a string with special characters and a lot of embedded double quotes, use the `%{}` construct.

Recipe 1.6. Converting Between Characters and Values

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to see the ASCII code for a character, or transform an ASCII code into a string.

Solution

To see the ASCII code for a specific character as an integer, use the `?` operator:

```
?a                # => 97
?!                # => 33
?\n               # => 10
```

To see the integer value of a particular in a string, access it as though it were an element of an array:

```
'a'[0]            # => 97
'bad sound'[1]    # => 97
```

To see the ASCII character corresponding to a given number, call its `#chr` method. This returns a string containing only one character:

```
97.chr             # => "a"
33.chr             # => "!"
10.chr             # => "\n"
0.chr              # => "\000"
256.chr            # RangeError: 256 out of char range
```

Discussion

Though not technically an array, a string acts a lot like like an array of `Fixnum` objects: one `Fixnum` for each byte in the string. Accessing a single element of the "array" yields a `Fixnum` for the corresponding byte: for textual strings, this is an ASCII code. Calling `String#each_byte` lets you iterate over the `Fixnum` objects that make up a string.

See Also

- Recipe 1.8, "Processing a String One Character at a Time"

Recipe 1.7. Converting Between Strings and Symbols

Problem

You want to get a string containing the label of a Ruby symbol, or get the Ruby symbol that corresponds to a given string.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

To turn a symbol into a string, use `Symbol#to_s`, or `Symbol#id2name`, for which `to_s` is an alias.

```
:a_symbol.to_s                      # => "a_symbol"
:AnotherSymbol.id2name              # => "AnotherSymbol"
:"Yet another symbol!".to_s         # => "Yet another symbol!"
```

You usually reference a symbol by just typing its name. If you're given a string in code and need to get the corresponding symbol, you can use `String.intern`:

```
:dodecahedron.object_id             # => 4565262
symbol_name = "dodecahedron"
symbol_name.intern                  # => :dodecahedron
symbol_name.intern.object_id        # => 4565262
```

Discussion

A `Symbol` is about the most basic Ruby object you can create. It's just a name and an internal ID. Symbols are useful becase a given symbol name refers to the same object throughout a Ruby program.

Symbols are often more efficient than strings. Two strings with the same contents are two different objects (one of the strings might be modified later on, and become different), but for any given name there is only one `Symbol` object. This can save both time and memory.

```
"string".object_id          # => 1503030
"string".object_id          # => 1500330
:symbol.object_id           # => 4569358
:symbol.object_id           # => 4569358
```

If you have *n* references to a name, you can keep all those references with only one symbol, using only one object's worth of memory. With strings, the same code would use *n* different objects, all containing the same data. It's also faster to compare two symbols than to compare two strings, because Ruby only has to check the object IDs.

```
"string1" == "string2"      # => false
:symbol1 == :symbol2        # => false
```

Finally, to quote Ruby hacker Jim Weirich on when to use a string versus a symbol:

- If the contents (the sequence of characters) of the object are important, use a string.
- If the identity of the object is important, use a symbol.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- See Recipe 5.1, "Using Symbols as Hash Keys" for one use of symbols
- Recipe 8.12, "Simulating Keyword Arguments," has another
- Chapter 10, especially Recipe 10.4, "Getting a Reference to a Method" and Recipe 10.10, "Avoiding Boilerplate Code with Metaprogramming"
- See http://glu.ttono.us/articles/2005/08/19/understanding-ruby-symbols for a symbol primer

Recipe 1.8. Processing a String One Character at a Time

Problem

You want to process each character of a string individually.

Solution

If you're processing an ASCII document, then each byte corresponds to one character. Use `String#each_byte` to yield each byte of a string as a number, which you can turn into a one-character string:

```
'foobar'.each_byte { |x| puts "#{x} = #{x.chr}" }
# 102 = f
# 111 = o
# 111 = o
# 98 = b
# 97 = a
# 114 = r
```

Use `String#scan` to yield each character of a string as a new one-character string:

```
'foobar'.scan( /./ ) { |c| puts c }
# f
# o
# o
# b
# a
# r
```

Discussion

Since a string is a sequence of bytes, you might think that the `String#each` method would iterate over the sequence, the way `Array#each` does. But `String#each` is actually used to split a string on a given record separator (by default, the newline):

```
"foo\nbar".each { |x| puts x }
# foo
# bar
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The string equivalent of `Array#each` method is actually `each_byte`. A string stores its characters as a sequence of Fixnum objects, and `each_bytes` yields that sequence.

`String#each_byte` is faster than `String#scan`, so if you're processing an ASCII file, you might want to use `String#each_byte` and convert to a string every number passed into the code block (as seen in the Solution).

`String#scan` works by applying a given regular expression to a string, and yielding each match to the code block you provide. The regular expression `/./` matches every character in the string, in turn.

If you have the `$KCODE` variable set correctly, then the `scan` technique will work on UTF-8 strings as well. This is the simplest way to sneak a notion of "character" into Ruby's byte-based strings.

Here's a Ruby string containing the UTF-8 encoding of the French phrase "ça va":

```
french = "\xc3\xa7a va"
```

Even if your terminal can't properly display the character "ç", you can see how the behavior of `String#scan` changes when you make the regular expression Unicodeaware, or set `$KCODE` so that Ruby handles all strings as UTF-8:

```
french.scan(/./) { |c| puts c }
#
#
# a
#
# v
# a

french.scan(/./u) { |c| puts c }
# ç
# a
#
# v
# a

$KCODE = 'u'
french.scan(/./) { |c| puts c }
# ç
# a
#
# v
# a
```

Once Ruby knows to treat strings as UTF-8 instead of ASCII, it starts treating the two bytes representing the "ç" as a single character. Even if you can't see UTF-8, you can write programs that handle it correctly.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 11.12, "Converting from One Encoding to Another"

Recipe 1.9. Processing a String One Word at a Time

Problem

You want to split a piece of text into words, and operate on each word.

Solution

First decide what you mean by "word." What separates one word from another? Only whitespace? Whitespace or punctuation? Is "johnny-come-lately" one word or three? Build a regular expression that matches a single word according to whatever definition you need (there are some samples are in the Discussion).

Then pass that regular expression into `String#scan`. Every word it finds, it will yield to a code block. The `word_count` method defined below takes a piece of text and creates a histogram of word frequencies. Its regular expression considers a "word" to be a string of Ruby identifier characters: letters, numbers, and underscores.

```
class String
  def word_count
    frequencies = Hash.new(0)
    downcase.scan(/\w+/) { |word| frequencies[word] += 1 }
   return frequencies
  end
end

%{Dogs dogs dog dog dogs.}.word_count
# => {"dogs"=>3, "dog"=>2}
%{"I have no shame," I said.}.word_count
# => {"no"=>1, "shame"=>1, "have"=>1, "said"=>1, "i"=>2}
```

Discussion

The regular expression `/\w+/` is nice and simple, but you can probably do better for your application's definition of "word." You probably don't consider two words separated by an underscore to be a single word. Some English words, like "pan-fried" and "fo'c'sle", contain embedded punctuation. Here are a few more definitions of "word" in regular expression form:

```
# Just like /\w+/, but doesn't consider underscore part of a word.
/[0-9A-Za-z]/

# Anything that's not whitespace is a word.
/[^\S]+/
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Accept dashes and apostrophes as parts of words.
/[-'\w]+/

# A pretty good heuristic for matching English words.
/(\w+([-'.]\w+)*/
```

The last one deserves some explanation. It matches embedded punctuation within a word, but not at the edges. "Work-in-progress" is recognized as a single word, and "—-never—-" is recognized as the word "never" surrounded by punctuation. This regular expression can even pick out abbreviations and acronyms such as "Ph.D" and "U.N.C.L.E.", though it can't distinguish between the final period of an acronym and the period that ends a sentence. This means that "E.F.F." will be recognized as the word "E.F.F" and then a nonword period.

Let's rewrite our `word_count` method to use that regular expression. We can't use the original implementation, because its code block takes only one argument. `String#scan` passes its code block one argument for each match group in the regular expression, and our improved regular expression has two match groups. The first match group is the one that actually contains the word. So we must rewrite `word_count` so that its code block takes two arguments, and ignores the second one:

```
class String
  def word_count
    frequencies = Hash.new(0)
    downcase.scan(/(\w+([-'.]\w+)*)/) { |word, ignore| frequencies[word] += 1 }
    return frequencies
  end
end

%{"That F.B.I. fella--he's quite the man-about-town."}.word_count
# => {"quite"=>1, "f.b.i"=>1, "the"=>1, "fella"=>1, "that"=>1,
#         "man-about-town"=>1, "he's"=>1}
```

Note that the "\w" character set matches different things depending on the value of $KCODE. By default, "\w" matches only characters that are part of ASCII words:

```
french = "il \xc3\xa9tait une fois"
french.word_count
# => {"fois"=>1, "une"=>1, "tait"=>1, "il"=>1}
```

If you turn on Ruby's UTF-8 support, the "\w" character set matches more characters:

```
$KCODE='u'
french.word_count
# => {"fois"=>1, "une"=>1, "était"=>1, "il"=>1}
```

The regular expression group \b matches a word *boundary*: that is, the last part of a word before a piece of whitespace or punctuation. This is useful for `String#split` (see Recipe 1.4), but not so useful for `String#scan`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 1.4, "Reversing a String by Words or Characters"
- The Facets core library defines a `String#each_word` method, using the regular expression `/([-'\w]+)/`

Recipe 1.10. Changing the Case of a String

Problem

Your string is in the wrong case, or no particular case at all.

Solution

The String class provides a variety of case-shifting methods:

```
s = 'HELLO, I am not here. I WENT to tHe MaRKEt.'
s.upcase           # => "HELLO, I AM NOT HERE. I WENT TO THE MARKET."
s.downcase         # => "hello, i am not here. i went to the market."
s.swapcase         # => "hello, i AM NOT HERE. i went TO ThE mArkeT."
s.capitalize       # => "Hello, i am not here. i went to the market."
```

Discussion

The `upcase` and `downcase` methods force all letters in the string to upper-or lowercase, respectively. The `swapcase` method transforms uppercase letters into lowercase letters and vice versa. The `capitalize` method makes the first character of the string uppercase, if it's a letter, and makes all other letters in the string lowercase.

All four methods have corresponding methods that modify a string in place rather than creating a new one: `upcase!`, `downcase!`, `swapcase!`, and `capitalize!`. Assuming you don't need the original string, these methods will save memory, especially if the string is large.

```
un_banged = 'Hello world.'
un_banged.upcase     # => "HELLO WORLD."
un_banged            # => "Hello world."

banged = 'Hello world.'
banged.upcase!       # => "HELLO WORLD."
banged               # => "HELLO WORLD."
```

To capitalize a string without lowercasing the rest of the string (for instance, because the string contains proper nouns), you can modify the first character of the string in place. This corresponds to the `capitalize!` method. If you want something more like `capitalize`, you can create a new string out of the old one.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
class String
  def capitalize_first_letter
    self[0].chr.capitalize + self[1, size]
  end

  def capitalize_first_letter!
    unless self[0] == (c = self[0,1].upcase[0])
      self[0] = c
      self
    end
    # Return nil if no change was made, like upcase! et al.
  end
end

s = 'i told Alice. She remembers now.'
s.capitalize_first_letter        # => "I told Alice. She remembers now."
s                                # => "i told Alice. She remembers now."
s.capitalize_first_letter!
s                                # => "I told Alice. She remembers now."
```

To change the case of specific letters while leaving the rest alone, you can use the `tr` or `tr!` methods, which translate one character into another:

```
'LOWERCASE ALL VOWELS'.tr('AEIOU', 'aeiou')
# => "LoWeRCaSe aLL VoWeLS"

'Swap case of ALL VOWELS'.tr('AEIOUaeiou', 'aeiouAEIOU')
# => "SwAp cAsE Of aLL VoWeLS"
```

See Also

- Recipe 1.18, "Replacing Multiple Patterns in a Single Pass"
- The Facets Core library adds a `String#camelcase` method; it also defines the case predicates `String#lowercase?` and `String#uppercase?`

Recipe 1.11. Managing Whitespace

Problem

Your string contains too much whitespace, not enough whitespace, or the wrong kind of whitespace.

Solution

Use `strip` to remove whitespace from the beginning and end of a string:

```
" \tWhitespace at beginning and end. \t\n\n".strip
```

Add whitespace to one or both ends of a string with `ljust, rjust`, and `center`:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
s = "Some text."
s.center(15)
s.ljust(15)
s.rjust(15)
```

Use the `gsub` method with a string or regular expression to make more complex changes, such as to replace one type of whitespace with another.

```
#Normalize Ruby source code by replacing tabs with spaces
rubyCode.gsub("\t", "     ")

#Transform Windows-style newlines to Unix-style newlines
"Line one\n\rLine two\n\r".gsub(\n\r", "\n")
# => "Line one\nLine two\n"

#Transform all runs of whitespace into a single space character
"\n\rThis string\t\t\tuses\n all\tsorts\nof whitespace.".gsub(/\s+/," ")
# => " This string uses all sorts of whitespace."
```

Discussion

What counts as whitespace? Any of these five characters: space, tab (`\t`), newline (`\n`), linefeed (`\r`), and form feed (`\f`). The regular expression `/\s/` matches any one character from that set. The `strip` method strips any combination of those characters from the beginning or end of a string.

In rare cases you may need to handle oddball "space" characters like backspace (`\b` or `\010`) and vertical tab (`\v` or `\012`). These are not part of the `\s` character group in a regular expression, so use a custom character group to catch these characters.

```
" \bIt's whitespace, Jim,\vbut not as we know it.\n".gsub(/[\s\b\v]+/, " ")
# => "It's whitespace, Jim, but not as we know it."
```

To remove whitespace from only one end of a string, use the `lstrip` or `rstrip` method:

```
s = "   Whitespace madness! "
s.lstrip                        # => "Whitespace madness! "
s.rstrip                        # => "   Whitespace madness!"
```

The methods for adding whitespace to a string (`center, ljust, and rjust`) take a single argument: the total length of the string they should return, counting the original string and any added whitespace. If `center` can't center a string perfectly, it'll put one extra space on the right:

```
"four".center(5)                # => "four "
"four".center(6)                # => " four "
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Like most string-modifying methods, `strip, gsub, lstrip`, and `rstrip` have counterparts `strip!, gsub!, lstrip!`, and `rstrip!`, which modify the string in place.

Recipe 1.12. Testing Whether an Object Is String-Like

Problem

You want to see whether you can treat an object as a string.

Solution

Check whether the object defines the `to_str` method.

```
'A string'.respond_to? :to_str        # => true
Exception.new.respond_to? :to_str     # => true
4.respond_to? :to_str                 # => false
```

More generally, check whether the object defines the specific method of `String` you're thinking about calling. If the object defines that method, the right thing to do is usually to go ahead and call the method. This will make your code work in more places:

```
def join_to_successor(s)
  raise ArgumentError, 'No successor method!' unless s.respond_to? :succ
  return "#{s}#{s.succ}"
end

join_to_successor('a')           # => "ab"
join_to_successor(4)             # => "45"
join_to_successor(4.01)
# ArgumentError: No successor method!
```

If I'd checked `s.is_a? String` instead of `s.respond_to? :succ`, then I wouldn't have been able to call `join_to_successor` on an integer.

Discussion

This is the simplest example of Ruby's philosophy of "duck typing:" if an object quacks like a duck (or acts like a string), just go ahead and treat it as a duck (or a string). Whenever possible, you should treat objects according to the methods they define rather than the classes from which they inherit or the modules they include.

Calling `obj.is_a? String` will tell you whether an object derives from the `String` class, but it will overlook objects that, though intended to be used as strings, don't inherit from `String`.

`Exception`s, for instance, are essentially strings that have extra information associated with them. But they don't subclass class name `"String"`. Code that uses `is_a?`

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`String` to check for stringness will overlook the essential stringness of `Exception`s. Many add-on Ruby modules define other classes that can act as strings: code that calls `is_a? String` will break when given an instance of one of those classes.

The idea to take to heart here is the general rule of duck typing: to see whether provided data implements a certain method, use `respond_to?` instead of checking the class. This lets a future user (possibly yourself!) create new classes that offer the same capability, without being tied down to the preexisting class structure. All you have to do is make the method names match up.

See Also

- Chapter 8, especially the chapter introduction and Recipe 8.3, "Checking Class or Module Membership"

Recipe 1.13. Getting the Parts of a String You Want

Problem

You want only certain pieces of a string.

Solution

To get a substring of a string, call its `slice` method, or use the array index operator (that is, call the `[]` method). Either method accepts a `Range` describing which characters to retrieve, or two `Fixnum` arguments: the index at which to start, and the length of the substring to be extracted.

```
s = 'My kingdom for a string!'
s.slice(3,7)                      # => "kingdom"
s[3,7]                            # => "kingdom"
s[0,3]                            # => "My "
s[11, 5]                          # => "for a"
s[11, 17]                         # => "for a string!"
```

To get the first portion of a string that matches a regular expression, pass the regular expression into `slice` or `[]`:

```
s[/.ing/]                         # => "king"
s[/str.*/]                        # => "string!"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

To access a specific byte of a string as a `Fixnum`, pass only one argument (the zerobased index of the character) into `String#slice` or `[]` method. To access a specific byte as a single-character string, pass in its index and the number 1.

```
s.slice(3)                    # => 107
s[3]                          # => 107
107.chr                       # => "k"
s.slice(3,1)                  # => "k"
s[3,1]                        # => "k"
```

To count from the end of the string instead of the beginning, use negative indexes:

```
s.slice(-7,3)                 # => "str"
s[-7,6]                       # => "string"
```

If the length of your proposed substring exceeds the length of the string, `slice` or `[]` will return the entire string after that point. This leads to a simple shortcut for getting the rightmost portion of a string:

```
s[15…s.length]                # => "a string!"
```

See Also

- Recipe 1.9, "Processing a String One Word at a Time"
- Recipe 1.17, "Matching Strings with Regular Expressions"

Recipe 1.14. Handling International Encodings

Problem

You need to handle strings that contain nonASCII characters: probably Unicode characters encoded in UTF-8.

Solution

To use Unicode in Ruby, simply add the following to the beginning of code.

```
$KCODE='u'
require 'jcode'
```

You can also invoke the Ruby interpreter with arguments that do the same thing:

```
$ ruby -Ku -rjcode
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If you use a Unix environment, you can add the arguments to the shebang line of your Ruby application:

```
#!/usr/bin/ruby -Ku -rjcode
```

The `jcode` library overrides most of the methods of `String` and makes them capable of handling multibyte text. The exceptions are `String#length`, `String#count`, and `String#size`, which are not overridden. Instead `jcode` defines three new methods: `String#jlength`, `string#jcount`, and `String#jsize`.

Discussion

Consider a UTF-8 string that encodes six Unicode characters: `efbca1` (A), `efbca2` (B), and so on up to UTF-8 `efbca6` (F):

```
string = "\xef\xbc\xa1" + "\xef\xbc\xa2" + "\xef\xbc\xa3" +
         "\xef\xbc\xa4" + "\xef\xbc\xa5" + "\xef\xbc\xa6"
```

The string contains 18 bytes that encode 6 characters:

```
string.size                                    # => 18
string.jsize                                   # => 6
```

`String#count` is a method that takes a strong of bytes, and counts how many times those bytes occurs in the string. `String#jcount` takes a string of *characters* and counts how many times those characters occur in the string:

```
string.count "\xef\xbc\xa2"                    # => 13
string.jcount "\xef\xbc\xa2"                   # => 1
```

String#count treats `"\xef\xbc\xa2"` as three separate bytes, and counts the number of times each of those bytes shows up in the string. `String#jcount` treats the same string as a single character, and looks for that character in the string, finding it only once.

```
"\xef\xbc\xa2".length                          # => 3
"\xef\xbc\xa2".jlength                         # => 1
```

Apart from these differences, Ruby handles most Unicode behind the scenes. Once you have your data in UTF-8 format, you really don't have to worry. Given that Ruby's creator Yukihiro Matsumoto is Japanese, it is no wonder that Ruby handles Unicode so elegantly.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- If you have text in some other encoding and need to convert it to UTF-8, use the `iconv` library, as described in Recipe 11.2, "Extracting Data from a Document's Tree Structure"
- There are several online search engines for Unicode characters; two good ones are at http://isthisthingon.org/unicode/ and http://www.fileformat.info/info/unicode/char/search.htm

Recipe 1.15. Word-Wrapping Lines of Text

Problem

You want to turn a string full of miscellaneous whitespace into a string formatted with linebreaks at appropriate intervals, so that the text can be displayed in a window or sent as an email.

Solution

The simplest way to add newlines to a piece of text is to use a regular expression like the following.

```
def wrap(s, width=78)
  s.gsub(/(.{1,#{width}})(\s+|\Z)/, "\\1\n")
end

wrap("This text is too short to be wrapped.")
# => "This text is too short to be wrapped.\n"

puts wrap("This text is not too short to be wrapped.", 20)
# This text is not too
# short to be wrapped.

puts wrap("These ten-character columns are stifling my creativity!", 10)
# These
# ten-character
# columns
# are
# stifling
# my
# creativity!
```

Discussion

The code given in the Solution preserves the original formatting of the string, inserting additional line breaks where necessary. This works well when you want to preserve the existing formatting while squishing everything into a smaller space:

```
poetry = %q{It is an ancient Mariner,
And he stoppeth one of three.
"By thy long beard and glittering eye,
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Now wherefore stopp'st thou me?}

puts wrap(poetry, 20)
# It is an ancient
# Mariner,
# And he stoppeth one
# of three.
# "By thy long beard
# and glittering eye,
# Now wherefore
# stopp'st thou me?
```

But sometimes the existing whitespace isn't important, and preserving it makes the result look bad:

```
prose = %q{I find myself alone these days, more often than not,
watching the rain run down nearby windows. How long has it been
raining? The newspapers now print the total, but no one reads them
anymore.}

puts wrap(prose, 60)
# I find myself alone these days, more often than not,
# watching the rain run down nearby windows. How long has it
# been
# raining? The newspapers now print the total, but no one
# reads them
# anymore.
```

Looks pretty ragged. In this case, we want to get replace the original newlines with new ones. The simplest way to do this is to preprocess the string with another regular expression:

```
def reformat_wrapped(s, width=78)
 s.gsub(/\s+/, " ").gsub(/(.{1,#{width}})( |\Z)/, "\\1\n")
end
```

But regular expressions are relatively slow; it's much more efficient to tear the string apart into words and rebuild it:

```
def reformat_wrapped(s, width=78)
  lines = []
  line = ""
  s.split(/\s+/).each do |word|
    if line.size + word.size >= width
      lines << line
      line = word
    elsif line.empty?
     line = word
    else
     line << " " << word
   end
   end
   lines << line if line
  return lines.join "\n"
end

puts reformat_wrapped(prose, 60)
# I find myself alone these days, more often than not,
# watching the rain run down nearby windows. How long has it
# been raining? The newspapers now print the total, but no one
# reads them anymore.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- The Facets Core library defines `String#word_wrap` and `String#word_wrap!` methods

Recipe 1.16. Generating a Succession of Strings

Problem

You want to iterate over a series of alphabetically-increasing strings as you would over a series of numbers.

Solution

If you know both the start and end points of your succession, you can simply create a range and use `Range#each`, as you would for numbers:

```
('aa'..'ag').each { |x| puts x }
# aa
# ab
# ac
# ad
# ae
# af
# ag
```

The method that generates the successor of a given string is `String#succ`. If you don't know the end point of your succession, you can define a generator that uses `succ`, and break from the generator when you're done.

```
def endless_string_succession(start)
  while true
    yield start
    start = start.succ
  end
end
```

This code iterates over an endless succession of strings, stopping when the last two letters are the same:

```
endless_string_succession('fol') do |x|
  puts x
  break if x[-1] == x[-2]
end
# fol
# fom
# fon
# foo
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

Imagine a string as an odometer. Each character position of the string has a separate dial, and the current odometer reading is your string. Each dial always shows the same kind of character. A dial that starts out showing a number will always show a number. A character that starts out showing an uppercase letter will always show an uppercase letter.

The string succession operation increments the odometer. It moves the rightmost dial forward one space. This might make the rightmost dial wrap around to the beginning: if that happens, the dial directly to its left is also moved forward one space. This might make *that* dial wrap around to the beginning, and so on:

```
'89999'.succ                  # => "90000"
'nzzzz'.succ                  # => "oaaaa"
```

When the leftmost dial wraps around, a new dial is added to the left of the odometer. The new dial is always of the same type as the old leftmost dial. If the old leftmost dial showed capital letters, then so will the new leftmost dial:

```
'Zzz'.succ                    # => "AAaa"
```

Lowercase letters wrap around from "z" to "a". If the first character is a lowercase letter, then when it wraps around, an "a" is added on to the beginning of the string:

```
'z'.succ                      # => "aa"
'aa'.succ                     # =>  "ab"
'zz'.succ                     # => "aaa"
```

Uppercase letters work in the same way: "Z" becomes "A". Lowercase and uppercase letters never mix.

```
'AA'.succ     # => "AB"
'AZ'.succ                     # =>  "BA"
'ZZ'.succ # => "AAA"
'aZ'.succ                     # =>  "bA"
'Zz'.succ                     # => "AAa"
```

Digits in a string are treated as numbers, and wrap around from 9 to 0, just like a car odometer.

```
'foo19'.succ           # => "foo20"
'foo99'.succ                  # => "fop00"
'99'.succ                     # => "100"
'9Z99'.succ  # => "10A00"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Characters other than alphanumerics are not incremented unless they are the only characters in the string. They are simply ignored when calculating the succession, and reproduced in the same positions in the new string. This lets you build formatting into the strings you want to increment.

```
'10-99'.succ                # => "11-00"
```

When nonalphanumerics are the only characters in the string, they are incremented according to ASCII order. Eventually an alphanumeric will show up, and the rules for strings containing alphanumerics will take over.

```
'a-a'.succ                  # => "a-b"
'z-z'.succ                  # => "aa-a"
'Hello!'.succ # => "Hellp!"
%q{'zz'}.succ               # => "'aaa'"
%q{z'zz'}.succ              # => "aa'aa'"
'$$$$'.succ                 # => "$$$%"
s = '!@-'
13.times { puts s = s.succ }
                            # !@.
                            # !@/
                            # !@0
                            # !@1
                            # !@2
                            # …
                            # !@8
                            # !@9
                            # !@10
```

There's no reverse version of `String#succ`. Matz, and the community as a whole, think there's not enough demand for such a method to justify the work necessary to handle all the edge cases. If you need to iterate over a succession of strings in reverse, your best bet is to transform the range into an array and iterate over that in reverse:

```
("a".."e").to_a.reverse_each { |x| puts x }
                                    # e
                                    # d
                                    # c
                                    # b
                                    # a
```

See Also

- Recipe 2.15, "Generating a Sequence of Numbers"
- Recipe 3.4, "Iterating Over Dates"

Recipe 1.17. Matching Strings with Regular Expressions

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to know whether or not a string matches a certain pattern.

Solution

You can usually describe the pattern as a regular expression. The =~ operator tests a string against a regular expression:

```
string = 'This is a 30-character string.'

if string =~ /([0-9]+)-character/ and $1.to_i == string.length
  "Yes, there are #$1 characters in that string."
end
# => "Yes, there are 30 characters in that string."
```

You can also use `Regexp#match`:

```
match = Regexp.compile('([0-9]+)-character').match(string)
if match && match[1].to_i == string.length
  "Yes, there are #{match[1]} characters in that string."
end
# => "Yes, there are 30 characters in that string."
```

You can check a string against a series of regular expressions with a `case` statement:

```
string = "123"

case string
when /^[a-zA-Z]+$/
  "Letters"
when /^[0-9]+$/
  "Numbers"
else
  "Mixed"
end
# => "Numbers"
```

Discussion

Regular expressions are a cryptic but powerful minilanguage for string matching and substring extraction. They've been around for a long time in Unix utilities like `sed`, but Perl was the first general-purpose programming language to include them. Now almost all modern languages have support for Perl-style regular expression.

Ruby provides several ways of initializing regular expressions. The following are all equivalent and create equivalent `Regexp` objects:

```
/something/
Regexp.new("something")
Regexp.compile("something")
%r{something}
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The following modifiers are also of note.

Table 1-1.

Regexp::IGNORECASE	i	Makes matches case-insensitive.
Regexp::MULTILINE	m	Normally, a regexp matches against a single line of a string. This will cause a regexp to treat line breaks like any other character.
Regexp::EXTENDED	x	This modifier lets you space out your regular expressions with whitespace and comments, making them more legible.

Here's how to use these modifiers to create regular expressions:

```
/something/mxi
Regexp.new('something',
           Regexp::EXTENDED + Regexp::IGNORECASE + Regexp::MULTILINE)
%r{something}mxi
```

Here's how the modifiers work:

```
case_insensitive = /mangy/i
case_insensitive =~ "I'm mangy!"                       # => 4
case_insensitive =~ "Mangy Jones, at your service."    # => 0

multiline = /a.b/m
multiline =~ "banana\nbanana"                          # => 5
/a.b/ =~ "banana\nbanana"                              # => nil
# But note:
/a\nb/ =~ "banana\nbanana"                             # => 5

extended = %r{ \ was    # Match " was"
               \s       # Match one whitespace character
                a       # Match "a" }xi
extended =~ "What was Alfred doing here?" # => 4
extended =~ "My, that was a yummy mango." # => 8
extended =~ "It was\n\n\na fool's errand" # => nil
```

See Also

- *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly) gives a concise introduction to regular expressions, with many real-world examples
- RegExLib.com provides a searchable database of regular expressions (http://regexlib.com/default.aspx)
- A Ruby-centric regular expression tutorial (http://www.regular-expressions.info/ruby.html)
- `ri Regexp`
- Recipe 1.19, "Validating an Email Address"

Recipe 1.18. Replacing Multiple Patterns in a Single Pass

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to perform multiple, simultaneous search-and-replace operations on a string.

Solution

Use the `Regexp.union` method to aggregate the regular expressions you want to match into one big regular expression that matches any of them. Pass the big regular expression into `String#gsub`, along with a code block that takes a `MatchData` object. You can detect which of your search terms actually triggered the regexp match, and choose the appropriate replacement term:

```
class String
  def mgsub(key_value_pairs=[].freeze)
    regexp_fragments = key_value_pairs.collect { |k,v| k }
    gsub(Regexp.union(*regexp_fragments)) do |match|
      key_value_pairs.detect{|k,v| k =~ match}[1]
    end
  end
end
```

Here's a simple example:

```
"GO HOME!".mgsub([[/.*GO/i, 'Home'], [/home/i, 'is where the heart is']])
# => "Home is where the heart is!"
```

This example replaces all letters with pound signs, and all pound signs with the letter P:

```
"Here is number #123".mgsub([[/[a-z]/i, '#'], [/#/, 'P']])
# => "#### ## ###### P123"
```

Discussion

The naive solution is to simply string together multiple `gsub` calls. The following examples, copied from the solution, show why this is often a bad idea:

```
"GO HOME!".gsub(/.*GO/i, 'Home').gsub(/home/i, 'is where the heart is')
# => "is where the heart is is where the heart is!"

"Here is number #123".gsub(/[a-z]/i, "#").gsub(/#/, "P")
# => "PPPP PP PPPPPP P123"
```

In both cases, our replacement strings turned out to match the search term of a later `gsub` call. Our replacement strings were themselves subject to search-and-replace. In the first example, the conflict can be fixed by reversing the order of the substitutions. The second example shows a case where reversing the order won't help. You need to do all your replacements in a single pass over the string.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `mgsub` method will take a hash, but it's safer to pass in an array of key-value pairs. This is because elements in a hash come out in no particular order, so you can't control the order of substution. Here's a demonstration of the problem:

```
"between".mgsub(/ee/ => 'AA', /e/ => 'E') # Bad code
# => "bEtwEEn"

"between".mgsub([[/ee/, 'AA'], [/e/, 'E']]) # Good code
# => "bEtwAAn"
```

In the second example, the first substitution runs first. In the first example, it runs second (and doesn't find anything to replace) because of a quirk of Ruby's `Hash` implementation.

If performance is important, you may want to rethink how you implement `mgsub`. The more search and replace terms you add to the array of key-value pairs, the longer it will take, because the `detect` method performs a set of regular expression checks for every match found in the string.

See Also

- Recipe 1.17, "Matching Strings with Regular Expressions"
- Confused by the `*regexp_fragments` syntax in the call to `Regexp.union`? Take a look at Recipe 8.11, "Accepting or Passing a Variable Number of Arguments"

Recipe 1.19. Validating an Email Address

Problem

You need to see whether an email address is valid.

Solution

Here's a sampling of valid email addresses you might encounter:

```
test_addresses = [ #The following are valid addresses according to RFC822.
                   'joe@example.com', 'joe.bloggs@mail.example.com',
                   'joe+ruby-mail@example.com', 'joe(and-mary)@example.museum',
                   'joe@localhost',
```

Here are some invalid email addresses you might encounter:

```
                   # Complete the list with some invalid addresses
                   'joe', 'joe@', '@example.com',
                   'joe@example@example.com',
                   'joe and mary@example.com' ]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

And here are some regular expressions that do an okay job of filtering out bad email addresses. The first one does very basic checking for ill-formed addresses:

```
valid = '[^ @]+' # Exclude characters always invalid in email addresses
username_and_machine = /^#{valid}@#{valid}$/

test_addresses.collect { |i| i =~ username_and_machine }
# => [0, 0, 0, 0, 0, nil, nil, nil, nil, nil]
```

The second one prohibits the use of local-network addresses like "joe@localhost". Most applications should prohibit such addresses.

```
username_and_machine_with_tld = /^#{valid}@#{valid}\.#{valid}$/

test_addresses.collect { |i| i =~ username_and_machine_with_tld }
# => [0, 0, 0, 0, nil, nil, nil, nil, nil, nil]
```

However, the odds are good that you're solving the wrong problem.

Discussion

Most email address validation is done with naive regular expressions like the ones given above. Unfortunately, these regular expressions are usually written too strictly, and reject many email addresses. This is a common source of frustration for people with unusual email addresses like joe(and-mary)@example.museum, or people taking advantage of special features of email, as in joe+ruby-mail@example.com. The regular expressions given above err on the opposite side: they'll accept some syntactically invalid email addresses, but they won't reject valid addresses.

Why not give a simple regular expression that always works? Because there's no such thing. The definition of the syntax is anything but simple. Perl hacker Paul Warren defined an 6343-character regular expression for Perl's Mail::RFC822::Address module, and even it needs some preprocessing to accept absolutely every allowable email address. Warren's regular expression will work unaltered in Ruby, but if you really want it, you should go online and find it, because it would be foolish to try to type it in.

Check validity, not correctness

Even given a regular expression or other tool that infallibly separates the RFC822 compliant email addresses from the others, you can't check the *validity* of an email address just by looking at it; you can only check its syntactic correctness.

It's easy to mistype your username or domain name, giving out a perfectly valid email address that belongs to someone else. It's trivial for a malicious user to make up a valid email address that doesn't work at all—I did it earlier with the joe@example.com nonsense. !@ is a valid email address according to the regexp test, but no one in this universe uses it. You can't even compare the top-level domain of an address against a static

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

list, because new top-level domains are always being added. Syntactic validation of email addresses is an enormous amount of work that only solves a small portion of the problem.

The only way to be certain that an email address is valid is to successfully send email to it. The only way to be certain that an email address is the *right* one is to send email to it and get the recipient to respond. You need to weigh this additional work (yours and the user's) against the real value of a verified email address.

It used to be that a user's email address was closely associated with their online identity: most people had only the email address their ISP gave them. Thanks to today's free web-based email, that's no longer true. Email verification no longer works to prevent duplicate accounts or to stop antisocial behavior online—if it ever did.

This is not to say that it's never useful to have a user's working email address, or that there's no problem if people mistype their email addresses. To improve the quality of the addresses your users enter, without rejecting valid addresses, you can do three things beyond verifying with the permissive regular expressions given above:

1. Use a second naive regular expression, more restrictive than the ones given above, but don't prohibit addresses that don't match. Only use the second regular expression to advise the user that they may have mistyped their email address. This is not as useful as it seems, because most typos involve changing one letter for another, rather than introducing nonalphanumerics where they don't belong.

```
def probably_valid?(email)
 valid = '[A-Za-z\d.+-]+' #Commonly encountered email address characters
 (email =~ /#{valid}@#{valid}\.#{valid}/) == 0
end

#These give the correct result.
probably_valid? 'joe@example.com'                # => true
probably_valid? 'joe+ruby-mail@example.com'      # => true
probably_valid? 'joe.bloggs@mail.example.com'    # => true
probably_valid? 'joe@examplecom'                 # => false
probably_valid? 'joe+ruby-mail@example.com'      # => true
probably_valid? 'joe@localhost'                  # => false

# This address is valid, but probably_valid thinks it's not.
probably_valid? 'joe(and-mary)@example.museum'   # => false

# This address is valid, but certainly wrong.
probably_valid? 'joe@example.cpm'                # => true
```

2. Extract from the alleged email address the hostname (the "example.com" of joe@example.com), and do a DNS lookup to see if that hostname accepts email. A hostname that has an MX DNS record is set up to receive mail. The following code will catch most domain name misspellings, but it won't catch any username

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

misspellings. It's also not guaranteed to parse the hostname correctly, again because of the complexity of RFC822.

```
require 'resolv'
def valid_email_host?(email)
  hostname = email[(email =~ /@/)+1..email.length]
  valid = true
  begin
    Resolv::DNS.new.getresource(hostname, Resolv::DNS::Resource::IN::MX)
  rescue Resolv::ResolvError
    valid = false
  end
  return valid
end

#example.com is a real domain, but it won't accept mail
valid_email_host?('joe@example.com')         # => false

#lcqkxjvoem.mil is not a real domain.
valid_email_host?('joe@lcqkxjvoem.mil')      # => false

#oreilly.com exists and accepts mail, though there might not be a 'joe' there.
valid_email_host?('joe@oreilly.com')         # => true
```

3. Send email to the address the user input, and ask the user to verify receipt. For instance, the email might contain a verification URL for the user to click on. This is the only way to guarantee that the user entered a valid email address that they control. See Recipes 14.5 and 15.19 for this.

   This is overkill much of the time. It requires that you add special workflow to your application, it significantly raises the barriers to use of your application, and it won't always work. Some users have spam filters that will treat your test mail as junk, or whitelist email systems that reject all email from unknown sources. Unless you really need a user's working email address for your application to work, very simple email validation should suffice.

See Also

- Recipe 14.5, "Sending Mail"
- Recipe 15.19, "Sending Mail with Rails"
- See the amazing colossal regular expression for email addresses at http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

Recipe 1.20. Classifying Text with a Bayesian Analyzer

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to classify chunks of text by example: an email message is either spam or not spam, a joke is either funny or not funny, and so on.

Solution

Use Lucas Carlson's `Classifier` library, available as the `classifier` gem. It provides a naive Bayesian classifier, and one that implements Latent Semantic Indexing, a more advanced technique.

The interface for the naive Bayesian classifier is very straightforward. You create a `Classifier::Bayes` object with some classifications, and train it on text chunks whose classification is known:

```
require 'rubygems'
require 'classifier'

classifier = Classifier::Bayes.new('Spam', 'Not spam')

classifier.train_spam 'are you in the market for viagra? we sell viagra'
classifier.train_not_spam 'hi there, are we still on for lunch?'
```

You can then feed the classifier text chunks whose classification is unknown, and have it guess:

```
classifier.classify "we sell the cheapest viagra on the market"
# => "Spam"
classifier.classify "lunch sounds great"
# => "Not spam"
```

Discussion

Bayesian analysis is based on probablities. When you train the classifier, you are giving it a set of words and the classifier keeps track of how often words show up in each category. In the simple spam filter built in the Solution, the frequency hash looks like the `@categories` variable below:

```
classifier
# => #<Classifier::Bayes:0xb7cec7c8
#       @categories={:"Not spam"=>
#                      { :lunch=>1, :for=>1, :there=>1,
#                        :"?"=>1, :still=>1, :","=>1 },
#                    :Spam=>
#                      { :market=>1, :for=>1, :viagra=>2, :"?"=>1, :sell=>1 }
#                  },
#       @total_words=12>
```

These hashes are used to build probability calculations. Note that since we mentioned the word "viagra" twice in spam messages, there is a 2 in the "Spam" frequency hash for that

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

word. That makes it more spam-like than other words like "for" (which also shows up in nonspam) or "sell" (which only shows up once in spam). The classifier can apply these probabilities to previously unseen text and guess at a classification for it.

The more text you use to train the classifier, the better it becomes at guessing. If you can verify the classifier's guesses (for instance, by asking the user whether a message really was spam), you should use that information to train the classifier with new data as it comes in.

To save the state of the classifier for later use, you can use Madeleine persistence (Recipe 13.3), which writes the state of your classifier to your hard drive.

A few more notes about this type of classifier. A Bayesian classifier supports as many categories as you want. "Spam" and "Not spam" are the most common, but you are not limited to two. You can also use the generic `train` method instead of calling `train_[category_name]`. Here's a classifier that has three categories and uses the generic `train` method:

```
classifier = Classifier::Bayes.new('Interesting', 'Funny', 'Dramatic')

classifier.train 'Interesting', "Leaving reminds us of what we can part
  with and what we can't, then offers us something new to look forward
  to, to dream about."
classifier.train 'Funny', "Knock knock. Who's there? Boo boo. Boo boo
  who? Don't cry, it is only a joke."
classifier.train 'Dramatic', 'I love you! I hate you! Get out right
  now.'

classifier.classify 'what!'
# => "Dramatic"
classifier.classify "who's on first?"
# => "Funny"
classifier.classify 'perchance to dream'
# => "Interesting"
```

It's also possible to "untrain" a category if you make a mistake or change your mind later.

```
classifier.untrain_funny "boo"
classifier.untrain "Dramatic", "out"
```

See Also

- Recipe 13.3, "Persisting Objects with Madeleine"
- The README file for the Classifier library has an example of an LSI classifier
- Bishop (http://bishop.rubyforge.org/) is another Bayesian classifier, a port of Python's Reverend; it's available as the `bishop` gem
- http://en.wikipedia.org/wiki/Naive_Bayes_classifier
- http://en.wikipedia.org/wiki/Latent_Semantic_Analysis

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01