Table of Contents

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

12. Graphics and Other File Formats

Hundreds of standards exist for storing structured data in text or binary files. Some of these are so popular that we've devoted entire chapters to them (Chapters 11 and 13). Some are so simple that you can process them with the ad hoc techniques listed in Chapters 1 and 6. This chapter is a grab bag that tries to cover the rest of the field.

We focus especially on graphics, probably the most common binary files. Ruby lacks a mature image manipulation library like the Python Imaging Library, but it does have bindings to ImageMagick and GraphicsMagick, popular and stable C libraries. The RMagick library provides the same interface against ImageMagick and GraphicsMagick, so it doesn't matter which one you use.

You can get RMagick by installing the `RMagick` or `Rmagick-win32` gem. Unfortunately, the C libraries themselves are difficult to install: they have a lot of dependencies, especially if you want to process image formats like GIF and PostScript. The installation FAQ can help (http://rmagick.rubyforge.org/install-faq.html). On Debian GNU/Linux, you can just install the `imagemagick` package and then the `RMagick` gem.

The first recipes in this chapter show how to use RMagick to manipulate and convert images (on the question of *finding* images, see Recipe 16.2). Then it gets miscellaneous: we cover encryption, archive formats, Excel spreadsheets, and music files. We don't have space to cover every popular file format, but this chapter should give you an idea of what's out there. If this chapter lacks a recipe on your file format of choice, you may be able to find a Ruby library for it on the RAA, or by doing a web search for `ruby[file format name]`.

Recipe 12.1. Thumbnailing Images

Credit: Antonio Cangiano

Problem

Given an image, you want to create a smaller image to serve as a thumbnail.

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

Solution

Use `RMagick`, available from the `rmagick` or `rmagick-win32` gems. Its `Magick` module gives you a simple but versatile way to manipulate images. The class `Magick::Image` lets you resize images four different ways: with `resize`, `scale`, `sample`, or `thumbnail`.

All four methods accept a pair integer values, corresponding to the width and height in pixels of the thumbnail you want. Here's an example that uses `resize`: it takes the file `myimage.jpg` and makes a thumbnail of it 100 pixels wide by 100 pixels tall:

```
require 'rubygems'
require 'RMagick'

img = Magick::Image.read('myimage.jpg').first
width, height = 100, 100
thumb = img.resize(width, height)
thumb.write('mythumbnail.jpg')
```

Discussion

The class method `Image.read`, used in the Solution, receives an image filename as an argument and returns an array of `Image` objects.[1] You obtain the first (and, usually, only) element through `Array#first`.

[1] Why an array? Because you can pass in an animated GIF or a multilayered image file to `Image.read`. If you do, the array will contain an `Image` object for each image in the animated GIF, or for each layer in the multilayered file.

The code given in the Solution produces a thumbnail that is 100 pixels by 100, no matter what dimensions the original image had. If the original image was a square, its proportions will be maintained. But if the initial image was a rectangle, squishing it into a 100 x 100 box will distort it.

If all your thumbnails need to be the same size, you might be willing to live with this distortion. But to maintain the proportions between the longest and shortest dimensions, you should define your thumbnail's width and height in terms of the original image's aspect ratio. You can get the image's original width and height by using its accessor methods, `Magick::Image#columns` and `Magick::Image#rows`.

A simpler solution is to pass `resize` a floating-point number as a scaling factor. This changes the image's size without altering the aspect ratio. Here's how to generate an image that is 15% the size of the original:

```
scale_factor = 0.15
thumb = img.resize(scale_factor)
thumb.write("mythumbnail.jpg")
```


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

To impose a maximum size on an image without altering its aspect ratio, use `change_geometry`:

```
def thumb_no_bigger_than(img, width, height)
  img.change_geometry("#{width}x#{height}") do |cols, rows, img|
    img.resize(cols, rows)
  end
end

img.rows                                    # => 470
img.columns                                 # => 892
thumb = thumb_no_bigger_than(img, 100, 100)
thumb.rows                                  # => 53
thumb.columns                               # => 100
```

There are other ways of getting a thumbnail besides using `resize`. All of the following lines give you some kind of thumbnail. The methods used below also have equivalent methods (like `scale`!) that modify an `Image` object in place:

```
thumb = img.scale(width, height)
thumb = img.scale(scale_factor)
thumb = img.sample(width, height)
thumb = img.sample(scale_factor)
thumb = img.thumbnail(width, height)
thumb = img.thumbnail(scale_factor)
```

You might also want to generate a thumbnail by cropping an image, rather than resizing it. The following code extracts an 80 x 100 pixel rectangle taken from the center of the image:

```
thumb = img.crop(Magick::CenterGravity, 80, 100)
```

Which of these methods should you use? `Magick::Image#resize` is the most advanced method, because it accepts two optional arguments: `filter` and `blur`. When you specify a filter, you alter the resizing algorithm's tradeoff between speed and quality. Refer to the `RMagick` guide for a complete list of available filters.

The second optional argument, `blur`, is a floating-point number that can be used to blur (values greater than 1) or sharpen (values less than 1) your image as it's resized. Blurring an image is a way to hide visual artifacts created by the thumbnailing process.

The `scale` method is simpler than `resize`, because it accepts only a width and height pair, or a scale factor. When you want to generate a thumbnail that's 10% the size of your original image or smaller, `thumbnail` is faster than `resize`.

Finally, `sample` scales images with pixel sampling. Unlike the other methods, it doesn't introduce any new colors through interpolation.

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

The best advice is to try these methods out with your images. Through trial and error, you can determine what works best for your application.

Using `crop` means approaching the problem in a different way. `crop` only includes a portion of the original image in the thumbnail. `crop` has several signatures, each of which requires the output image's width and height:

```
# With an x, y offset relative to the upper-left corner:
thumb = img.crop(x, y, width, height)
# With a GravityType and the x, y offset:
thumb = img.crop(Magick::WestGravity, x, y, width, height)

# With a GravityType:
thumb = img.crop(Magick::EastGravity, width, height)
```

`GravityType` is a constant that lets you specify the position of the region that needs to be cropped. The available options are quite self-explanatory.

Be aware that the x and y offset passed to the method `crop(gravity, x, y, width, height)` are not always calculated from the upper-left corner, but that they depend on the `GravityType` being used. Refer to the crop documentation for specific details.

You may also want to enforce rules on your list of images so that they all match. For example, you may require all your thumbnails to be smaller than 80 x 100 pixels, or you might want them to all have an equal width of 120 pixels. You may even decide that all images smaller than a certain limit should not be resized at all. For details on techniques for this, see the `RMagick` documentation of the `Image#change_geometry` method.

See Also

- This chapter's introduction discusses installing `RMagick`

Recipe 12.2. Adding Text to an Image

Credit: Antonio Cangiano

Problem

You want to add some text to an image—perhaps a caption or a copyright statement.

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

Solution

Create an RMagick `Draw` object and call its `annotate` method, passing in your image and the text.

The following code adds the copyright string '© NPS' to the bottom-right corner of the `canyon.png` image. It also specifies the font, the text color and size, and other features of the text:

```
require 'rubygems'
require 'RMagick'

img = Magick::Image.read('canyon.png').first
my_text = "\251 NPS"

copyright = Magick::Draw.new
copyright.annotate(img, 0, 0, 3, 18, my_text) do
 self.font = 'Helvetica'
 self.pointsize = 12
 self.font_weight = Magick::BoldWeight
 self.fill = 'white'
 self.gravity = Magick::SouthEastGravity
end
img.write('canyoncopyrighted.png')
```

The resulting image looks like Figure 12-1.

Figure 12-1. With a copyright message in the bottom-right corner

Discussion

The `annotate` method takes a code block that sets properties on the `Magick::Draw` object, describing how the annotation should be done. You can also set the properties on

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

the `Draw` object before calling `annotate`. This code works the same as the code given in the Solution:

```
require 'rubygems'
require 'RMagick'

img = Magick::Image.read("canyon.png").first
my_text = '\251 NPS'

copyright = Magick::Draw.new
copyright.font = 'Helvetica'
copyright.pointsize = 12
copyright.font_weight = Magick::BoldWeight
copyright.fill = 'white'
copyright.gravity = Magick::SouthEastGravity
copyright.annotate(img, 0, 0, 3, 18, my_text)
img.write('canyoncopyrighted.png')
```

What do these attributes do?

- The `font` attribute selects the font type from among those installed on your system. You can also specify the path to a specific font that is in a nonstandard location (e.g., "/home/antonio/Arial.ttf").
- `pointsize` is the font size in points (the default is 12). By default, there is one pixel per point, so you can just specify the font size in pixels.
- `font_weight` accepts a `WeightType` constant. This can be a number (100, 200, 300,...900), `BoldWeight` (equivalent to 700), or the default of `NormalWeight` (equivalent to 400).
- If you need your text to be italicized, you can set the `font_style` attribute to `Magick::ItalicStyle`.
- `fill` defines the text color. The default is "black". You can use X or SVG color names (such as "white", "red", "gray85", and "salmon"), or you can express the color in terms of RGB values (such as `"#fff"` or `"#ffffff"`—two of the most common formats)
- `gravity` controls which part of the image will contain the annotated text, subject to the arguments passed in to `annotate`. `SouthEastGravity` means that offsets will be calculated from the bottom-right corner of the image.

`Draw#annotate` itself takes six arguments:

- The `Image` object, or else an `ImageList` containing the images you want to annotate.
- The `width` and `height` of the rectangle in which the text is to be positioned.
- The `x` and `y` offsets of the text, relative to that rectangle and to the `gravity` of the `Draw` object.
- The text to be written.

In the Solution I wrote:

```
copyright.annotate(img, 0, 0, 3, 15, my_text)
```


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

The `width` and `height` are zeros, which indicates that `annotate` should use the whole image as its annotation rectangle. Earlier I gave the `Draw` object a `gravity` attribute of `SouthEastGravity`. This means that `annotate` will position the text at the bottom-right corner of the rectangle: that is, at the bottom-right corner of the image itself. The offsets of 3 and 18 indicate that the text should start vertically 18 pixels from the bottom of the box, and end horizontally 3 pixels from the right border of the box.

To position the text in the center of the image, I just change the gravity:

```
copyright.gravity = Magick::CenterGravity
copyright.annotate(img, 0, 0, 0, 0, my_text)
```

Note that I didn't have to specify any offsets: `CenterGravity` orients the text to be is in the exact center of the image (Figure 12-2). Specifying offsets would only move the text off-center.

The `Magick` library does substitutions for various special characters: for instance, the string `"%t"` will be replaced with the filename of the image. For more information about special characters, `GravityType` constants, and other `annotate` attributes that can let you fully customize the text appearance, refer to the `RMagick` documentation.

Figure 12-2. With a copyright message in the center of the image

See Also

- `RMagick` Documentation (http://studio.imagemagick.org/RMagick/doc/)

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

- On converting points to pixels (http://redux.imagemagick.org/RMagick/doc/draw.html#get_type_metrics)
- SVG color keywords list (http://www.w3.org/TR/SVG/types.html#ColorKeywords)
- This chapter's introduction gives instructions on installing `RMagick`

Recipe 12.3. Converting One Image Format to Another

Credit: Antonio Cangiano

Problem

You want to convert an image to a different format.

Solution

With `RMagick`, you can just read in the file and write it out with a different extension. This code converts a PNG file to JPEG format:

```
require 'rubygems'
require 'RMagick'

img = Magick::Image.read('myimage.png').first
img.write('myimage.jpg')
```

Discussion

As seen in the previous two recipes, `Magick::Image.read` receives the PNG image and returns an array of `Image` objects, from which we select the first and only image.

`RMagick` lets us convert the file into a JPEG by simply changing the filename's extension when we call the `write` method.

The underlying C library, ImageMagick or GraphicsMagick, has three ways of determining the format of image files:

- Checking an explicitly specified format prefix: for example, "GIF:myimage.jpg" indicates that the file `myimage` contains a GIF image, even though the file extension says otherwise.
- Looking inside the file for a "magic number", a set of bytes that indicates the format.
- Checking the file extension: for example, "myphoto.gif" is presumably a GIF file.

Although the format prefix takes precedence over the magic number, `RMagick` won't be fooled by an incorrect prefix. Eventually it will have to parse the image file, and the format mismatch will be revealed:

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

```
Magick::Image.read("JPG:myimage.png")
# Magick::ImageMagickError: Not a JPEG file: starts with 0x89 0x50 `myimage.png':
```

When you write an image to an output file, you can choose the output format by specifying a file extension or a prefix.

```
img = Magick::Image.read("myimage.png").first
img.write("myimage.jpg")                # Writes a JPEG
img.write("myimage.gif")                # Writes a GIF
img.write("JPG:myimage")                # Writes a JPEG
img.write("JPG:myimage.gif")            # Writes a JPEG
```

You can also get or set the file format of an image by calling the `Image#format` or `Image#format=` methods:

```
img.format                              # => "PNG"
img.format = "GIF"
img.format                              # => "GIF"
```

Of course, RMagick can't read to and write from every graphical file format in existence. How can you tell whether your version of RMagick knows how to write a particular file format?

You can query RMagick's capabilities by calling `Magick.formats`. This method returns a hash that maps an image format to a four-character code:

```
Magick.formats["GIF"]                   # => "*rw+"
Magick.formats["JPG"]                   # => "*rw-"
Magick.formats["AVI"]                   # => "*r--"
Magick.formats["PS"]                    # => " rw+"
```

The code represents the things that `RMagick` can do with that file format:

- The first character is an asterisk if RMagick has native blob support for that format. If not, the first character is a space. RMagick can convert most image formats into a generic string format (with `Image#to_blob`)that can be stored in the database as a BLOB and converted back into an `Image` object with `Image.from_blob`.

  The second character is "r" if RMagick knows how to read files in that format. Otherwise, it's a minus sign.
- The third character is "w" if RMagick knows how to write files in that format. Otherwise, it's a minus sign.
- The final character is "+" if RMagick knows how to cram multiple images into a single file (as in an animated GIF).

Here's a little bit of metaprogramming that adds four predicate methods to `Magick`, one for each element of the four-character code. You can use these methods instead of parsing the code string:

```
module Magick
 [["native_blob?", ?*], ["readable?", ?r],
  ["writable?", ?w], ["multi_image?", ?+]].each_with_index do |m, i|
   define_method(m[0]) do |format|
     code = formats[format]
     return code && code[i] == m[1]
   end
   module_function(m[0])
 end
end
```

This code demonstrates that the GIF file format supports multi-image files, but the JPG format doesn't:

```
Magick.multi_image? 'GIF'               # => true
Magick.multi_image? 'JPG'               # => false
```

`ImageMagick` and `GraphicsMagick` support the most common image formats (over 90 in total). However, they delegate support for many of these formats to external libraries or programs, which you may need to install separately. For instance, to read or write Postscript files, you'll need to have the Ghostscript program installed.

See Also

- `RMagick` Documentation (http://studio.imagemagick.org/RMagick/doc/)
- List of supported `ImageMagick` formats (http://www.imagemagick.org/script/formats.php)

Recipe 12.4. Graphing Data

Problem

You want to convert a bunch of data into a graph; usually a line chart, bar chart, or pie chart.

Solution

Use the Gruff library, written by Geoffrey Grosenbach. Install the `gruff` gem and build a `Gruff` object corresponding to the type of graph you want (for instance, `Gruff::Line`, `Gruff::Bar`, or `Gruff::Pie`). Add a dataset to the graph by passing `data` a label and an array of data points.

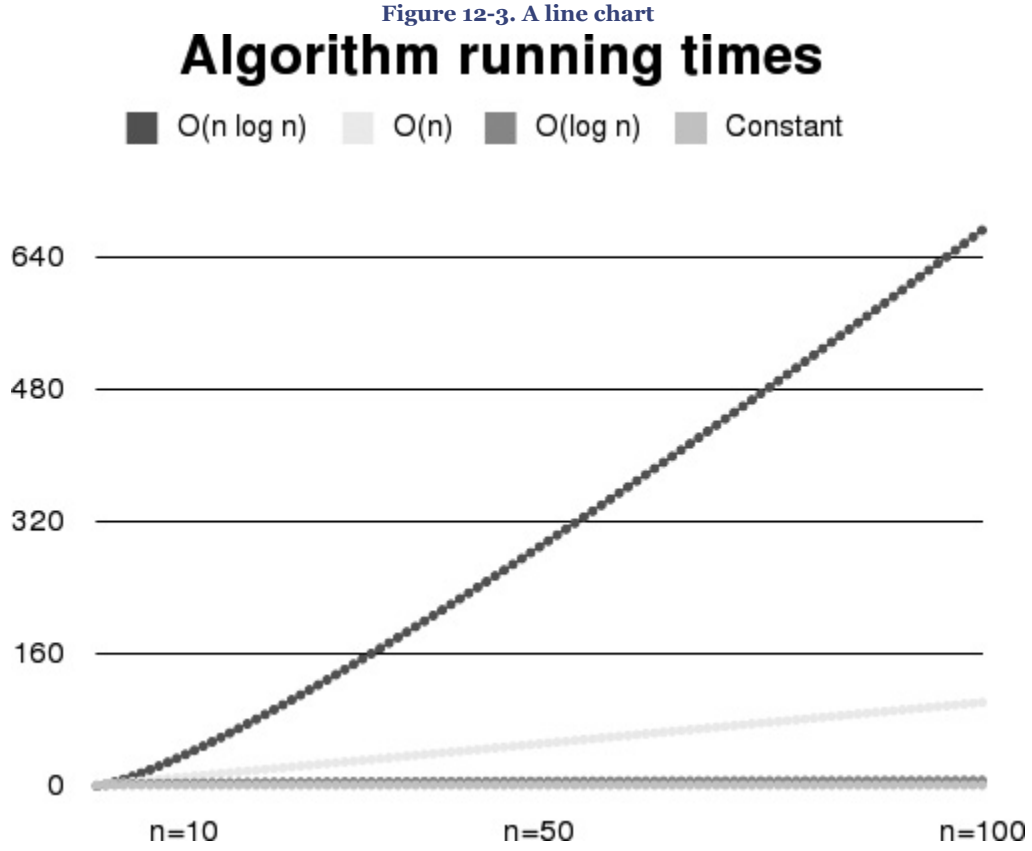
Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's code to create a graph that compares the running times of different sorts of algorithms:

```
require 'rubygems'
require 'gruff'

g = Gruff::Line.new(600)                # The graph will be 600 pixels wide.
g.title = 'Algorithm running times'
g.theme_37signals                       # The best-looking theme, in my opinion.

range = (1..101)
g.data('Constant', range.collect { 1 })
g.data('O(log n)', range.collect { |x| Math::log(x) / Math::log(2) })
g.data('O(n)', range.collect { |x| x })
g.data('O(n log n)', range.collect { |x| x * Math::log(x) / Math::log(2) })

g.labels = {10 => 'n=10', 50 => 'n=50', 100 => 'n=100' }
g.write('algorithms.png')
```

Figure 12-3 shows the graph it produces.

Figure 12-3. A line chart

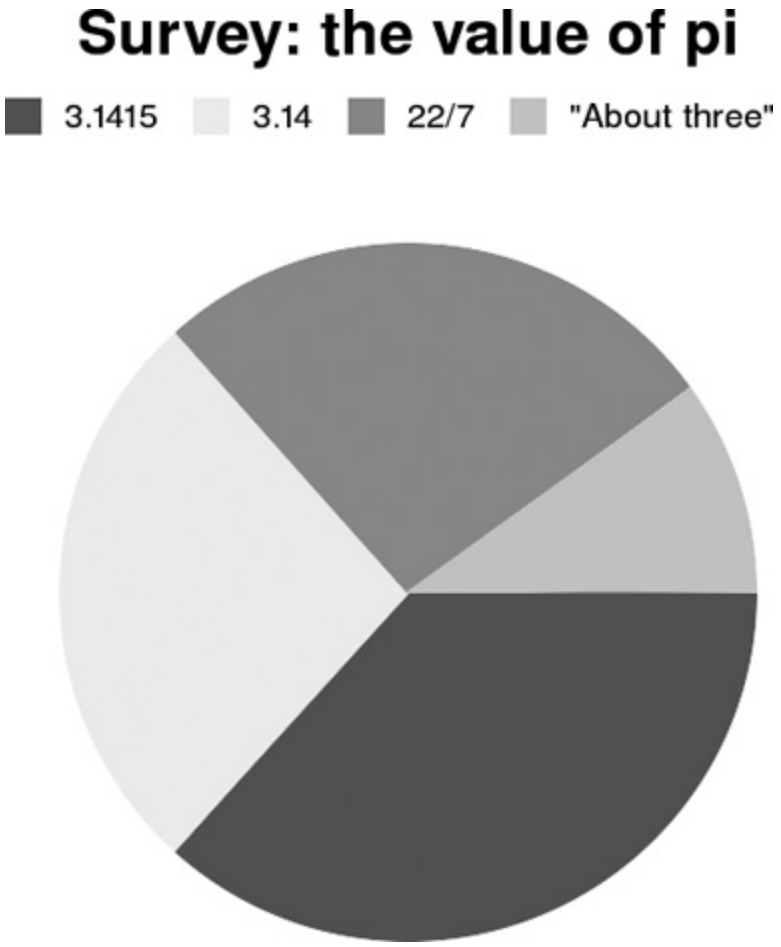


Here's code to create a pie chart (shown in Figure 12-4). Note that the numbers given for the datasets don't have to add up to 100. Gruff automatically scales the the pie chart to display the right proportions.

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

```
p = Gruff::Pie.new
p.theme_monochrome
p.title = "Survey: the value of pi"
p.data('"About three"', [3])
p.data('3.14', [8])
p.data('3.1415', [11])
p.data('22/7', [8])

p.write('pipie.png')
```

Figure 12-4. A pi chart



Discussion

Most of the time, programmers who need a graphing library need a *simple* graphing library: one that lets them easily produce a quick pie, line, or bar graph. Gruff works well for graphing simple datasets, but it doesn't have the functionality of a fullfledged math program.

Gruff's interface for customizing the display of datasets also leaves something to be desired. Instead of letting you tweak the colors individually, it provides a number of themes that

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

package together a background image, a text color, and a number of colors used in the graphs. Unfortunately, most of the provided themes are ugly (`theme_37signals` is pretty nice, though).

Here's a custom theme that makes monochrome graphs whose "colors" can be fairly easily distinguished. It takes advantage of the fact that it's easy to distinguish dark shades of gray from light shades, and that lighter shades are more easily distinguishable from one another. The graphs in this recipe were actually created with this `theme_monochrome`, so that the "colors" would be more easily distinguishable in a printed book.

```
class Gruff::Base
  def theme_monochrome
    reset_themes
    @colors = "6E9C7ADB".scan(/./).collect { |c| "##{c * 6}"}
    @marker_color = 'black'
    @base_image = render_gradiated_background('white', 'white')
  end
end
```

This code adds writer methods for the various colors, letting you modify the current theme on an ad hoc basis. `colors` sets the colors used to differentiate datasets from each other. `marker_color` method sets the color of the title and axis labels. `background` sets the background to a solid color, or to a gradient between two colors.

```
class Gruff::Base
  def colors=(colors)
    @colors = colors
  end

  def marker_color=(color)
    @marker_color = color
  end

  def background=(color1, color2=nil)
    color2 ||= color1
    @base_image = render_gradiated_background(color1, color2)
  end
end
```

See Also

- The Gruff homepage (http://nubyonrails.topfunky.com/pages/gruff)
- A couple of other Ruby graphing libraries deserve mention:
  - MRPlot is useful for plotting mathematical functions; its default implementation works on top of RMagick (http://harderware.bleedingmind.com/index.php?l=en&p=mrplot)
  - The SVG::Graph library doesn't need any external libraries and produces beautiful SVG graphs; unfortunately, not many programs have support for SVG graphics, although newer versions of Firefox do (http://www.germane-software.com/software/SVG/SVG::Graph/)

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

Recipe 12.5. Adding Graphical Context with Sparklines

Problem

You want to display a small bit of statistical context—a trend or a set of percentages—in the middle of a piece of text, without breaking up the flow of the text.

Solution

Install the `sparklines` gem (written by Geoffrey Grosenbach)and create a sparkline: a tiny embedded graphic that can go next to a piece of text without being too intrusive. If you're creating an HTML page, the image doesn't even need to have its own file: it can be embedded directly in the HTML.

This code creates a sparkline for a company's stock price, and embeds it in HTML after the company's stock symbol:

```
require 'rubygems'
require 'sparklines'
require 'base64'

def embedded_sparkline
  %{<img src="#{Base64.encode64(yield)}">}
end

# This method scales data so that the smallest item becomes 0 and the
# largest becomes 100.
def scale(data)
  min, max = data.min, data.max
  data.collect { |x| (x - min) / (max - min) * 100}
end

# Randomly generate closing prices for the past month.
prices = [rand(10)]
30.times { prices << prices.last + (rand - 0.5) }

# Generate HTML containing a stock graph as an embedded sparkline.
sparkline = embedded_sparkline { Sparklines.plot(scale(prices)) }
open('stock.html', 'w') do |f|
  f << "Is EvilCorp (NASDAQ:EVIL #{sparkline}) poised for a comeback?"
end
```

This code generates HTML that renders as shown in Figure 12-5.

Figure 12-5. A stock price history sparkline

Since it has no labels, the meaning of the sparkline must be determined from context. In this case, the graphic follows a stock symbol, so you can guess that it graphs the stock price.

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

In a different context, the sparkline for EvilCorp might be the company's reported earnings over time, or the results of a poll that tracks public opinion of the company.

Embedded sparklines won't show up in Internet Explorer, but if you're using Rails you can use the `sparklines_generator` gem to put cross-browser sparklines in your views.

Discussion

Sparklines are a way of graphically conveying information that would take lots of text to explain. They were invented by interface expert Edward Tufte, who describes them as "intense, simple, word-sized graphics." As implemented in the Ruby Sparklines library, a sparkline displays a small graph that shows a set of related numbers or a single percentage.

Sparklines are especially useful for annotating text with statistical summaries. We humans are visual creatures: when we read a text with sparklines, we come away with a better feel for the underlying numbers because we can visualize them as we read.

Sparklines are good at showing trends and making anomalies obvious. With sparklines, you can distinguish a winning sports team from a losing one at a glance, or notice an abnormally large expense report. Since neither the sparklines nor their axes are labelled, sparklines are not so good at displaying multifaceted information or absolute quantities.

Because sparklines show trends better than absolute values, it's often useful to scale your data so that it takes up the entire width of the sparkline (as in the stock price examples). But if you want to compare two sparklines to each other (for instance, to compare the stock prices of two companies), you shouldn't scale the data.

The Sparklines library can create several types of graph. Here's some code that annotates a politician's stump speech with small pie charts representing polling data. Only two colors are allowed in a sparklines pie chart: we'll choose a dark color to represent the percentage of people who agree with a statement, and a light color to represent the percentage who disagree. At a glance, the politician can see which parts of the speech are working and which need to be retooled.

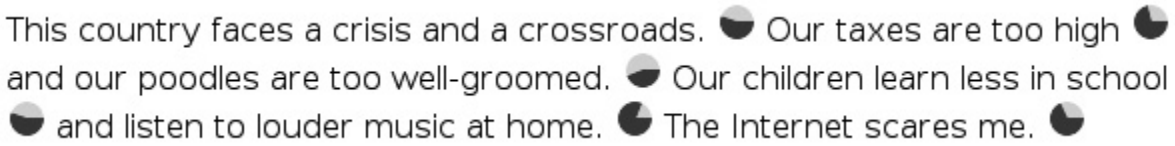
```
agree_percentages = [ 55, 71, 44, 55, 81, 68 ]

speech = %{This country faces a crisis and a crossroads. %s Our taxes
are too high %s and our poodles are too well-groomed. %s Our children
learn less in school %s and listen to louder music at home. %s The
Internet scares me. %s}

open('speech.html', 'w') do |f|
  sparklines = agree_percentages.collect do |p|
    embedded_sparkline do
      Sparklines.plot([p], :type => 'pie', :remain_color => 'pink',
                           :share_color=>'blue',
                           :background_color=>'transparent')
    end
  end
```


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

```
  f << speech % sparklines
end
```

The resulting HTML file renders as shown in Figure 12-6.

Figure 12-6. A speech, annotated with poll result sparklines



The result of `Sparklines.plot` is a binary string containing an image in PNG format. The string can be written to a PNG file on disk, or it can be encoded with the `Base64` library and embedded into a web page. The total size of `speech.html`, with six embedded sparklines, is about six kilobytes. Unfortunately, the Internet Explorer browser doesn't support the trick that lets you embed small images into a web page.

Sparklines in Rails Views

If you're using Rails, you can install the `sparklines_generator` gem on top of `sparklines`. This gem provides a controller and a helper that let you incorporate sparklines into your views, without having to worry about encoding the files or being incompatible with IE.

To add sparklines to your application, run this command to give yourself a sparklines controller:

```
$ ./script/generate sparklines
      create app/controllers/sparklines_controller.rb
      create app/helpers/sparklines_helper.rb
```

Add a `require 'sparklines'` statement to your `config/environment.rb` file, and call `helper :sparklines` from any controllers in which you want to use sparklines. You can then call the `sparkline_tag` method from within your views.

A view that renders part of an annotated speech might look like this:

```
This country faces a crisis and a crossroads.

<%= sparkline_tag [55, 10, 10, 20, 30], :type => "pie", :remain_color=>"pink",
:share_color => "blue", :background_color => "transparent" %>
```

That view generates HTML that looks like this:

```
This country faces a crisis and a crossroads.
```


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

```
<img
src="/sparklines?share_color=blue&remain_color=pink&results=55&type=pie&background_
color=transparent"
class="sparkline" alt="Sparkline Graph" />
```

Instead of embedding the sparkline within the HTML page (which won't work in IE), we call out to the sparklines controller, whose only purpose is to generate image files of sparklines on demand. This image is displayed like any other external image fetched through HTTP.

See Also

- The home page for the Sparklines library, which includes a tutorial on installation and use within Rails (http://nubyonrails.com/articles/2005/07/28/sparklines-graph-library-for-ruby)
- The `sparklines` gem requires RMagick; a pure Ruby implementation with fewer features is available (http://redhanded.hobix.com/inspect/sparklinesForMinimalists.html)
- Sparklines are described in Edward Tufte's book, *Beautiful Evidence* (Graphics Pr); you can see a version of the sparklines chapter from that book online (http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=0001OR&topic_id=1)

Recipe 12.6. Strongly Encrypting Data

Problem

You want to encrypt some data: to keep it private, or to keep it safe when sent through an insecure medium like email.

Solution

There are at least two good symmetric-key cryptography libraries for Ruby: Pelle Braendgaard's EzCrypto (available as the `ezcrypto` gem)and Richard Kernahan's Crypt (a third-party download).

EzCrypto is a user-friendly Ruby wrapper around the OpenSSL library, which you may need to install separately. Here's how to encrypt and decrypt a string with EzCrypto:

```
require 'rubygems'
require 'ezcrypto'

plaintext = '24.9195N 17.821E'

ezcrypto_key = EzCrypto::Key.with_password 'My secret key', 'salt string'
ezcrypto_ciphertext = ezcrypto_key.encrypt(plaintext)
# => "F\262\260\273\217\tR\351\362-\021-a\336\324Qc…"
```


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

```
ezcrypto_key.decrypt(ezcrypto_ciphertext)
# => "24.9195N 17.821E"
```

The Crypt library gives each encryption algorithm its own class, so you need to decide which you want to use. I'll use the AES/Rijndael algorithm: all the other algorithms have the same interface.[2]

[2] The `Crypt::IDEA` class works a little differently, but that algorithm is patented, so you shouldn't use it anyway.

```
require 'crypt/rijndael'

aes_key = Crypt::Rijndael.new('My secret key')
aes_cyphertext = aes_key.encrypt_string(plaintext)
# => "\e\003\203\030]\203\t\346…"

aes_key.decrypt_string(aes_cyphertext)
# => "24.9195N 17.821E"
```

Discussion

EzCrypto is available as a gem (`ezcrypto`), and it's fast because the actual encryption and decryption happens in the C OpenSSL libraries. Crypt is a pure Ruby implementation, so it's slower, but you don't have to worry about OpenSSL being installed.

EzCrypto and Crypt both implement several symmetric key algorithms. With EzCrypto, you can also specify the algorithm to use when you create an EzCrypto key. With Crypt, you need to instantiate the appropriate algorithm's class:

```
# EzCrypto example
blowfish_key = EzCrypto::Key.with_password('My secret password', 'salt string',
                                           :algorithm=>'blowfish')
# Crypt example
require 'crypt/blowfish'
blowfish_key = Crypt::Blowfish.new('My secret password')
```

The Crypt classes provide some convenience methods for encrypting and decrypting files and streams. The `encrypt_file` method takes two filenames: it reads from one file, encrypts the data, and writes ciphertext to the other. The `encrypt_stream` method is a little more general: it reads plaintext from one `IO` object and writes ciphertext to the other.

All the algorithms supported by Crypt and EzCrypto are symmetric-key algorithms: you must use the same key to encrypt and decrypt the data. This is simple when you're only encrypting data so that you can decrypt it later, but it's not so simple when you're sending encrypted data to someone else. You need to securely share the key with the other person ahead of time, or you need to use public-key algorithms like the ones provided by the Ruby PKCS implementation.

There was some controversy about whether this recipe should even be included in this Cookbook. A little knowledge is a dangerous thing, and a little is all we can impart in the

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

space we have for a recipe. Simply using an encryption algorithm won't automatically make your data secure. It won't be secure if you use a lousy password (like, say, "My secret password", as in the examples above).

Further, your data won't be secure if you store your keys on disk the wrong way. It won't be secure if your computer doesn't have a reliable enough source of random numbers. When you prompt the user for their password, the operating system might pick that moment to swap to disk the chunk of memory that contains the password, where an attacker could find it. Even experts frequently make mistakes when they're writing cryptography code.

That said, a strong encryption algorithm is better than a weak one, and trying to write your own algorithm is just about the worst mistake you can make. All we ask that you be careful. Instead of worrying about writing an algorithm to encrypt your data, get a book on security and focus your efforts on making sure you use the existing algorithms correctly.

See Also

- Download the Crypt library from http://crypt.rubyforge.org/, and install it by running `ruby install.rb`
- The EzCrypto documentation (http://ezcrypto.rubyforge.org/)
- The Ruby OpenSSL project (http://www.nongnu.org/rubypki/)
- The Ruby PKCS project homepage (http://dev.ctor.org/pkcs1)

Recipe 12.7. Parsing Comma-Separated Data

Problem

You have a plain-text string in a comma-delimited format. You need to parse this string, either to build a data structure or to perform some operation on the data and write it back out.

Solution

The built-in `csv` library can parse most common character-delimited formats. The FasterCSV library, available as the `fastercsv` gem, improves on `csv`'s performance and interface. I'll show you both, but I recommend `fastercsv` unless you can't use any software at all outside the standard library.

`CSV::Reader.parse` and `FasterCSV.parse` work the same way: they accept a string or an open file as an argument, and yield each parsed row of the comma-delimited file as

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

an array. The `csv` yields a `Row` object that acts like an array full of `Column` objects. FasterCSV just yields an array of strings.

```
require 'csv'
primary_colors = "red,green,blue\nred,yellow,blue"

CSV::Reader.parse(primary_colors) { |row| row.each { |cell| puts cell }}
# red
# green
# blue
# red
# yellow
# blue

require 'rubygems'
require 'faster_csv'
shakespeare = %{Sweet are the uses of adversity,As You Like It
"We few, we happy few",Henry V
"Seems, madam! nay it is; I know not ""seems.""",Hamlet}

FasterCSV.parse(shakespeare) { |row| puts "'#{row[0]}' -- #{row[1]}"}
# 'Sweet are the uses of adversity' -- As You Like It
# 'We few, we happy few' -- Henry V
# 'Seems, madam! nay it is; I know not "seems."' -- Hamlet
```

Discussion

Comma-delimited formats are among the most basic portable file formats. Unfortunately, they're also among the least standardized. There are many different formats, and some are internally inconsistent.

FasterCSV and the `csv` library can't parse every comma-delimited format, but they will parse common formats like the one used by Microsoft Excel, and they're your best tool for making sense of the myriad.

FasterCSV and `csv` both model a comma-delimited file as a nested array of strings. The `csv` library's `CSV` class uses `Row` objects and `Column` objects instead of arrays and strings, but it's the same idea. The terminology is from the spreadsheet world—understand-ably, since a CSV file is a common way of portably storing spreadsheet data.

The complications begin when the spreadsheet cells themselves contain commas or newlines. The standard way to handle this when exporting to comma-delimited format is to surround those cells with double quotes. Then the question becomes what to do with cells that contain double-quote characters. Both Ruby CSV libraries assume that double-quote characters are escaped by doubling, turning each " into "", as in the Hamlet quotation:

```
%{"Seems, madam! nay it is; I know not ""seems.""",Hamlet}
```


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

If you're certain that there are no commas or newlines embedded in your data, and thus no need for quote handling, you can use `String#split` to parse delimited records more quickly than `csv`. To output to this format, you can use `Array#join`:

```
def parse_delimited_naive(input, fieldsep=',', rowsep="\n")
  input.split(rowsep).inject([]) do |arr, line|
    arr << line.split(fieldsep)
  end
end

def join_delimited_naive(structure, fieldsep=',', rowsep="\n")
  rows = structure.inject([]) do |arr, parsed_line|
    arr << parsed_line.join(fieldsep)
  end
  rows.join(rowsep)
end

parse_delimited_naive("1,2,3,4\n5,6,7,8")
# => [["1", "2", "3", "4"], ["5", "6", "7", "8"]]

join_delimited_naive(parse_delimited_naive("1,2,3,4\n5,6,7,8"))
# => "1,2,3,4\n5,6,7,8"

parse_delimited_naive('1;2;3;4|5;6;7;8', ';', '|')
# => [["1", "2", "3", "4"], ["5", "6", "7", "8"]]

parse_delimited_naive('1,"2,3",4')
# => [["1", ""2", "3"", "4"]]
```

This is not recommended unless you wrote all the relevant code yourself, or can manually inspect the code as well as the dataset. Just because you haven't seen any quoted cells yet doesn't mean there won't be any in the future. When in doubt, use `csv` or `fastercsv`. Handwritten CSV generators and parsers are a leading cause of bad data.

To create a comma-delimited file, open an output file with `CSV.open` or `FasterCSV.open`, and append a series of arrays to the resulting file-like object. Every array you append will be converted to a comma-delimited row in the destination file.

```
data = [[1,2,3],['A','B','C'],['do','re','mi']]

writer = FasterCSV.open('first3.csv', 'w')
data.each { |x| writer << x }
writer.close
puts open('first3.csv').read()
# 1,2,3
# A,B,C
# do,re,mi

data = []
FasterCSV.foreach('first3.csv') { |row| data << row }
data
# => [["1", "2", "3"], ["A", "B", "C"], ["do", "re", "mi"]]
```

See Also

- The FasterCSV documentation (http://fastercsv.rubyforge.org/)
- Chapter 11

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

Recipe 12.8. Parsing Not-Quite-Comma-Separated Data

Problem

You need to parse a plain-text string or file that's in a format similar to commadelimited format, but its delimiters are some strings other than commas and newlines.

Solution

When you call a `CSV::Reader` method, you can specify strings to act as a row separator (the string between each `Row`) and a field separator (the string between each `Column`). You can do the same with simulated keyword arguments passed into `FasterCSV.parse`. This should let you parse most formats similar to the comma-delimited format:

```
require 'csv'

pipe_separated="1|2ENDa|bEND"

CSV::Reader.parse(pipe_separated, '|', 'END') { |r| r.each { |c| puts c } }
# 1
# 2
# a
# b

require 'rubygems'
require 'faster_csv'
FasterCSV.parse(pipe_separated, :col_sep=>'|', :row_sep=>'END') do |r|
  r.each { |c| puts c }
end
# 1
# 2
# a
# b
```

Discussion

Value-delimited formats tend to differ along three axes:

- The field separator (usually a single comma)
- The row separator (usually a single newline)
- The quote character (usually a double quote)

Like `Reader` methods, `Writer` methods accept custom values for the field and row separators.

```
data = [[1,2,3],['A','B','C'],['do','re','mi']]

open('first3.csv', 'w') do |output|
  CSV::Writer.generate(output, ':', '-END-') do |writer|
    data.each { |x| writer << x }
  end
```


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

```
end
open('first3.csv') { |input| input.read() }
# => "1:2:3-END-A:B:C-END-do:re:mi-END-"

FasterCSV.open('first3.csv', 'w', :col_sep=>':', :row_sep=>'-END-') do |output|
  data.each { |x| output << x }
end
open('first3.csv') { |input| input.read() }
# => "1:2:3-END-A:B:C-END-do:re:mi-END-"
```

It's rare that you'll need to override the quote character, and neither `csv` nor `fastercsv` will let you do it. Both libraries' quote characters are hardcoded to the double-quote character. If you need to parse a format that has different quote character, the simplest thing to do is subclass `FasterCSV` and override its `init_parsers` method.

Change the regular expression assigned to `@parsers[:csv_row]`, replacing all double quotes with the quote character you want. The most common alternate quote character is the single quote: to get that, you'd have an `init_parsers` method like this:

```
class MyFasterCSV < FasterCSV
  def init_parsers(options)
    super
    @parsers[:csv_row] =
         / \G(?:^|#{Regexp.escape(@col_sep)})         # anchor the match
           (?: '((?>[^']*)(?>''[^']*)*)'              # find quoted fields
               |                                      # … or …
               ([^'#{Regexp.escape(@col_sep)}]*)      # unquoted fields
               )/x
  end
end
MyFasterCSV.parse("1,'2,3',4") { |r| puts r }
# 1
# 2,3
# 4
```

Some value-delimited files are simply corrupt: they were generated by programs that didn't think to escape quote marks or to quote cells with embedded delimiters. Neither `csv` nor `fastercsv` can parse these files, because they're ambiguous or invalid.

```
missing_quotes=%{20051002, Alice says, "I saw that!"}
CSV::Reader.parse(missing_quotes) { |r| r.each { |c| puts c } }
# CSV::IllegalFormatError: CSV::IllegalFormatError

unescaped_quotes=%{20051002, "Alice says, "I saw that!""}
FasterCSV.parse(unescaped_quotes) { |r| r.each { |c| puts c } }
# FasterCSV::MalformedCSVError: Unclosed quoted field.
```

Your best strategy for dealing with this kind of file is to use regular expressions to massage the data into a form that `fastercsv` can parse, or to parse it with `String#split` and deal with any quoting problems afterwards. In either case, your code will have to work with the particular quirks of the data you're trying to parse.

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

See Also

- Recipe 12.7, "Parsing Comma-Separated Data"

Recipe 12.9. Generating and Parsing Excel Spreadsheets

Problem

Your program needs to parse data from Excel spreadsheets, or generate new Excel spreadsheets.

Solution

To generate Excel files, use the `spreadsheet` library, available as a third-party gem (see the See Also section below for where to get it). With it you can create simple Excel spreadsheets. As of this writing, `spreadsheet` does not support formulas or large spreadsheets (seven megabytes is the limit).

This code creates an Excel spreadsheet containing some random numbers with a total, and saves it to disk:

```
require 'rubygems'
require 'spreadsheet/excel'

SUM_SPREADSHEET = 'sum.xls'
workbook = Spreadsheet::Excel.new(SUM_SPREADSHEET)
worksheet = workbook.add_worksheet('Random numbers and their sum.')
sum = 0
random_numbers = (0..9).collect { rand(100) }
worksheet.write_column(0, 0, random_numbers)

format = workbook.add_format(:bold => true)
worksheet.write(10, 0, "Sum:", format)
worksheet.write(10, 1, random_numbers.inject(0) { |sum, x| sum + x })
workbook.close
```

To parse an Excel file, use the `parseexcel` library, also available as a third-party download. It can parse simple data out of the Excel file format. This code parses the Excel file generated by the previous code:

```
require 'parseexcel/parser'
workbook = Spreadsheet::ParseExcel::Parser.new.parse(SUM_SPREADSHEET)

worksheet = workbook.worksheet(0)
sum = (0..9).inject(0) do |sum, row|

  sum + worksheet.cell(row, 0).value.to_i
end

worksheet.cell(10, 0).value                              # => "Sum:"
worksheet.cell(10, 1).value                              # => 602.0
sum                                                      # => 602
```


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

Like `spreadsheet`, `parseexcel` doesn't recognize spreadsheet formulas.

Discussion

The comma-separated file is the *lingua franca* for spreadsheet data, but sometimes you must deal with real spreadsheet files. You can save other people's time by accepting their Excel spreadsheets as input, instead of insisting they convert everything to CSV for you. And nothing impresses manager types like an automatically generated spreadsheet file they can poke at.

The `spreadsheet` and `parseexcel` libraries are only suitable for creating or parsing simple spreadsheets: more or less the ones that export well to comma-delimited format. If you want to handle more complex Excel files from Ruby, you have a couple options. The POI Java library can write various Microsoft Office files, and it has Ruby bindings. If you're running on a Windows computer that has Excel installed, you can use Ruby's built-in `win32ole` library to communicate with the Excel installation.

Hopefully this will be fixed by the time you read this, but just in case: spreadsheets generated with `spreadsheet` may show up as black-on-black in some spreadsheet programs (Gnumeric is one). This is because `spreadsheet` generates workbooks with a default format that specifies no background color. So each spreadsheet program uses *its* default color, and some of them make unfortunate choices. Here's a subclass of `Workbook` that specifies default text and background colors, so that you don't end up with a black-on-black spreadsheet:

```
class ExcelWithBackground < Spreadsheet::Excel
  def initialize(*args)
    super(*args)
    @format = Format.new(:bg_color => 'white', :fg_color => 'black')
  end
end

workbook = ExcelWithBackground.new(SUM_SPREADSHEET)
# …
```

See Also

- You can download `parseexcel` from http://download.ywesee.com/parseexcel/
- The `spreadsheet` homepage is at http://rubyspreadsheet.sourceforge.net/; it's available as a gem (http://prdownloads.sourceforge.net/rubyspreadsheet/), but since it's not hosted on RubyForge, you can't just install it with `gem install spreadsheet-excel`: you must download the gem and run `gem install` on the local gem file
- POI (http://jakarta.apache.org/poi/index.html) and its Ruby bindings (http://jakarta.apache.org/poi/poi-ruby.html)

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

- Information on scripting Excel in Ruby (http://www.rubygarden.org/ruby?ScriptingExcel)
- The "Ruby and Microsoft Windows" chapter in the Pickaxe Book—*Programming Ruby* by Dave Thomas, with Chad Fowler and Andy Hunt (Pragmatic Bookshelf)

Recipe 12.10. Compressing and Archiving Files with Gzip and Tar

Problem

You want to write compressed data to a file to save space, or uncompress the contents of a compressed file. If you're compressing data, you might want to compress multiple files into a single archive file.

Solution

The most common compression format on Unix systems is gzip. Ruby's `zlib` library lets you read to and write from gzipped I/O streams as though they were normal files. The most useful classes in this library are `GzipWriter` and `GzipReader`.[3]

[3] The compressed strings in these examples are actually larger than the originals. This is because I used very short strings to save space in the book, and short strings don't compress well. Any compression technique introduces some overhead; with gzip, you don't actually save any space by compressing a text string of less than about 100 bytes.

Here's `GzipWriter` being used to create a compressed file, and `GzipReader` decompressing the same file:

```
require 'zlib'

file = 'compressed.gz'
Zlib::GzipWriter.open(file) do |gzip|
  gzip << "For my next trick, I'll be written to a compressed file."
  gzip.close
end

open(file, 'rb') { |f| f.read(10) }
# => "\037\213\010\000\201\2766D\000\003"

Zlib::GzipReader.open(file) { |gzip| gzip.read }
# => "For my next trick, I'll be written to a compressed file."
```

Discussion

`GzipWriter` and `GzipReader` are most commonly used to write to files on disk, but you can wrap any file-like object in the appropriate class and automatically compress everything you write to it, or decompress everything you read from it.

The following code works the same way as the compression code in the Solution, but it's more flexible: the `File` object that's passed into the `Zlib::GzipWriter` constructor could just as easily be a `Socket` or other file-like object.

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

```
open('compressed.gz', 'wb') do |file|
  gzip = Zlib::GzipWriter.new(file)
  gzip << "For my next trick, I'll be written to a compressed file."
  gzip.close
end
```

If you need to compress or decompress a string, use the `Zlib::Deflate` or `Zlib::Inflate` classes rather than constructing a `StringIO` object:

```
deflated = Zlib::Deflate.deflate("I'm a compressed string.")
# => "x\234\363T\317UHTH…"
Zlib::Inflate.inflate(deflated)
# => "I'm a compressed string."
```

Tar files

Gzip compresses a single file. What if you want to smash multiple files together into a single archive file? The standard archive format for Unix is `tar`, and tar files are sometimes called tarballs. A tarball might also be compressed with gzip to save space, but on Unix the archiving and the compression are separate steps (unlike on Windows, where a ZIP file both archives multiple files and compresses them).

The Minitar library is the simplest way to create tarballs in pure Ruby. It's available as the `archive-tar-minitar` gem.[4]

[4] The RubyGems package defines the `Gem::Package::TarWriter` and `Gem::Package::TarReader` classes, which expose an interface similar to Minitar's. You can use these classes if you're fanatical about minimizing your dependencies, but I don't recommend it. These classes only implement the bare-bones functionality necessary to pack and unpack gem-like tarballs, and they also make your code look like it has something to do with RubyGems.

Here's some code that creates a tarball containing two files and a directory. Note the Unix permission modes (0644, 0755, and 0600). These are the permissions the files will have when they're extracted, perhaps by the Unix tar command.

```
require 'rubygems'
require 'archive/tar/minitar'

open('tarball.tar', 'wb') do |f|
  Archive::Tar::Minitar::Writer.open(f) do |w|



w.add_file('file1', :mode => 0644, :mtime => Time.now) do |stream, io|
      stream.write('This is file 1.')
    end

w.mkdir('subdirectory', :mode => 0755, :mtime => Time.now)

w.add_file('subdirectory/file2', :mode => 0600,
               :mtime => Time.now) do |stream, io|
      stream.write('This is file 2.')
    end
  end
end
```


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

Here's a method that reads a tarball and print out its contents:

```
def browse_tarball(filename)
  open(filename, 'rb') do |f|
    Archive::Tar::Minitar::Reader.open(f).each do |entry|
      puts %{I see a file "#{entry.name}" that's #{entry.size} bytes long.}
    end
  end
end

browse_tarball('tarball.tar')
# I see a file "file1" that's 15 bytes long.
# I see a file "subdirectory" that's 0 bytes long.
# I see a file "subdirectory/file2" that's 15 bytes long.
```

And here's a simple method for archiving a number of disk files into a compressed tarball. Note how the Minitar `Writer` is wrapped within a `GzipWriter`, which automatically compresses the data as it's written. Minitar doesn't have to know about the `GzipWriter`, because all file-like objects look more or less the same.

```
def make_tarball(destination, *paths)
  Zlib::GzipWriter.open(destination) do |gzip|
    out = Archive::Tar::Minitar::Output.new(gzip)
    paths.each do |file|
      puts "Packing #{file}"
      Archive::Tar::Minitar.pack_file(file, out)
    end
    out.close
  end
end
```

This code creates some files and tars them up:

```
Dir.mkdir('colors')
paths = ['colors/burgundy', 'colors/beige', 'colors/clear']
paths.each do |path|
  open(path, 'w') do |f|
    f.puts %{This is a dummy file.}
  end
end

make_tarball('new_tarball.tgz', *paths)

# Packing colors/burgundy
# Packing colors/beige
# Packing colors/clear
# => #<File:new_tarball.tgz (closed)>
```

See Also

- On Windows, both compression and archiving are usually handled with ZIP files; see the next recipe, Recipe 12.11, "Reading and Writing ZIP Files," for details
- Recipe 14.3, "Customizing HTTP Request Headers," uses zlib to decompress the gzipped body of a response from a web server

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

Recipe 12.11. Reading and Writing ZIP Files

Problem

You want to create or examine a ZIP archive from within Ruby code.

Solution

Use the `rubyzip` gem. Its Zip module gives you several ways of putting files into ZIP archives, and taking them out again. The simplest interface is the `Zip::ZipFileSystem`, which duplicates most of the `File` and `Dir` operations within the context of a ZIP file. You can use this to create ZIP files:

```
require 'rubygems'
require 'zip/zipfilesystem'

Zip::ZipFile.open('zipfile.zip', Zip::ZipFile::CREATE) do |zip|
  zip.file.open('file1', 'w') { |f1| f1 << 'This is file 1.' }
  zip.dir.mkdir('subdirectory')
  zip.file.open('subdirectory/file2', 'w') { |f1| f1 << 'This is file 2.' }
end
```

You can use the same interface to read a ZIP file. Here's a method that uses the equivalent of `Dir#foreach` to recursively print out the contents of a ZIP file:

```
def process_zipfile(zip, path='')
  if zip.file.file? path
    puts %{#{path}: "#{zip.read(path)}"}
  else
    unless path.empty?
      path += '/'
      puts path
    end
    zip.dir.foreach(path) do |filename|
      process_zipfile(zip, path + filename)
    end
  end
end
```

And here it is running against the ZIP file I just created:

```
Zip::ZipFile.open('zipfile.zip') do |zip|
  process_zipfile(zip)
end
# subdirectory/
# subdirectory/file2: "This is file 2."
# file1: "This is file 1."
```

Discussion

ZIP, or PKZip, is the most popular compression format on Windows. As seen in the previous recipe, Unix separates the tasks of stuffing several files into a single archive

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

(tar), and compressing the resulting file *(gzip)*. On Windows, ZIP files perform both tasks. If you want to compress a single file, you need to put it into a ZIP file all by itself.

The `rubyzip` library provides several interfaces for creating and reading ZIP files. `Zip::ZipFileSystem` is the easiest for most programmers: in the example above, `zip.file` has about the same interface as the `File` class, and `zip.dir` is similar to the `Dir` class. The analogy holds because a ZIP file actually contains a tiny filesystem inside it. [5]

[5] This is how Windows XP's Explorer can let you browse a ZIP file as though it were a directory tree.

If you're porting Java code, or you're already familiar with Java's `java.util.zip` library, you might prefer the `Zip::ZipFile` class. It more or less duplicates Java's `ZipFile` class in a Ruby idiom. Here it is being used to create the same ZIP file I created in the Solution:

```
Zip::ZipFile.open('zipfile2.zip', Zip::ZipFile::CREATE) do |zip|
  zip.get_output_stream('file1') { |f| f << 'This is file 1.' }
  zip.mkdir('subdirectory')
  zip.get_output_stream('subdirectory/file2') { |f| f << 'This is file 2.' }
end
```

See Also

- The RDoc for the `rubyzip` gem (http://rubyzip.sourceforge.net/)

Recipe 12.12. Reading and Writing Configuration Files

Problem

You want to store your application's configuration on disk, in a format parseable by Ruby but easily editable by someone with a text editor.

Solution

Put your configuration into a data structure, and write the data structure to disk as YAML. So long as you only use built-in Ruby data types (strings, numbers, arrays, hashes, and so on), the YAML file will be human-readable and -editable.

```
require 'yaml'
configuration = { 'color' => 'blue',
                  'font' => 'Septimus',
                  'font-size' => 7 }
open('text.cfg', 'w') { |f| YAML.dump(configuration, f) }

open('text.cfg') { |f| puts f.read }
# --
# font-size: 7
# color: blue
```


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

```
# font: Septimus

open('text.cfg') { |f| YAML.load(f) }
# => {"font-size"=>7, "color"=>"blue", "font"=>"Septimus"}
```

It's easy for a user to edit this: it's just a colon-separated, line-delimited set of key names and values. Not a problem, even for a relatively unsophisticated user.

Discussion

YAML is a serialization format, designed to store data structures to disk and read them back later. But there's no reason why the data structures can't be modified by other programs while they're on disk. Since simple YAML files are human-editable, they make good configuration files.

A YAML file typically contains a single data structure. The most common structures for configuration data are a hash (seen in the Solution) and an array of hashes.

```
configuration = [ { 'name' => 'Alice', 'donation' => 50 },
                  { 'name' => 'Bob', 'donation' => 15, 'currency' => "EUR" } ]
open('donors.cfg', 'w') { |f| YAML.dump(configuration, f) }
open('donors.cfg') { |f| puts f.read }
# ---
# - name: Alice
#   donation: 50
# - name: Bob
#   donation: 15
#   currency: EUR
```

In Recipe 5.1 we advise saving memory by using symbols as hash keys instead of strings. If your hash is going to be converted into human-editable YAML, you should always use strings. Otherwise, people editing the YAML may become confused. Compare the following two bits of YAML:

```
puts { 'measurements' => 'metric' }.to_yaml
# ---
# measurements: metric
puts { :measurements => :metric }.to_yaml
# ---
# :measurements: :metric
```

Outside the context of a Ruby program, the symbol `:measurements` is too easy to confuse with the string ":measurements".

See Also

- Recipe 13.1, "Serializing Data with YAML"

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

Recipe 12.13. Generating PDF Files

Problem

You want to create a text or graphical document as a PDF, where you have complete control over the layout.

Solution

Use Austin Zeigler's `PDF::Writer` library, available as the `pdf-writer` gem. Its API gives you fine-grained control over the placement of text, images, and shapes.

This code uses `PDF::Writer` to produce a simple flyer with an image and a border (Figure 12-7). It assumes you've got a graphic called `sue.png` to insert into the document:

Figure 12-7. The flyer


```
require 'rubygems'
require 'pdf/writer'                                    # => false

# Putting "false" on the next line suppresses a huge output dump when
# you run this code in irb.
pdf = PDF::Writer.new; false

pdf.text("LOST\nDINOSAUR", :justification => :center, :font_size => 42,
         :left => 50, :right => 50)
pdf.image("sue.png", :left=> 100, :justification => :center, :resize => 0.75)
pdf.text(%{Three-year-old <i>Tyrannosaurus rex</i>\nSpayed\nResponds to "Sue"},
         :left => 80, :font_size => 20, :justification => :left)
pdf.text("(555) 010-7829", :justification => :center, :font_size => 36)
```


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

```
pdf.rectangle(pdf.left_margin + 25, pdf.y-25,
              pdf.margin_width-50, pdf.margin_height-pdf.y+50).stroke; false

pdf.save_as('flyer.pdf')
```

Discussion

So long as you're only calling `Writer#text` and `Writer#image`, PDF generation is easy. PDF automatically adds new text and images to the bottom of the current text, creating new pages as needed.

It gets tricky when you want to do something more complex, like draw shapes. Then you need to specify the placement and dimensions in coordinates.

Take as an example the `Writer#rectangle` call in the Solution:

```
pdf.rectangle(pdf.left_margin, pdf.y-25,
            pdf.margin_width, pdf.margin_height-pdf.y+25).stroke
```

The first two arguments are coordinates: the left edge of the rectangle and the bottom edge of the rectangle. The second two arguments are the width and height of the rectangle.

The width is simple enough: my box starts at the left margin and its width is `pdf.margin_width`*user space units*.[6] That is, my box takes up the entire width of the page except for the margin. The height is a little more tricky, because I do my own margins (25 user space units above and below the text), and because PDF coordinates start from the bottom-left of the page, not the top-left. Think of a Cartesian plane: the point (0,0)is below the point (0,1)and left of the point (1,0). That's how it is on a PDF page.

[6] A PDF user space unit is 1/72 of an inch.

`Writer#y` gives you the current position of the `PDF::Writer` "cursor:" the y-coordinate of the space directly under the most recently added text or image. I use this to place the bottom of the box just under the text.

If you want to generate many PDF documents from a template, you don't need to generate the whole document from scratch each time. You can create a `PDF::Writer` containing the skeleton of a document (say, just the corporate letterhead), then use `Marshal.dump` to save it to a binary string. You can then use `Marshal.load` as many times as necessary to get new documents, and fill in the blanks separately for each document.[7]

[7] Yes, this is kind of hacky. The best we can say is that the author of `PDF::Writer` himself recommends it (see "Creating Printable Documents with Ruby," cited in the following See Also section).

Here's a Ruby class that generates personalized certificates of achievement. We generate the PDF ahead of time with `generate_pdf`, leaving a blank space for the name. We can

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

then fill in names by calling `award_to`. Instead of rerunning the PDF generation code every time, `award_to` copies the predefined PDF over and over again by loading it from its marshalled format.

```
require 'rubygems'
require 'pdf/writer'

class Certificate

  def initialize(achievement)
   @without_name = Marshal.dump(generate_pdf(achievement))
  end

  def award_to(name)
    pdf = Marshal.load(@without_name)
    pdf.move_pointer(-225)
    pdf.text("<i>#{name}</i>", :font_size => 64,
             :justification => :center)
    return pdf
  end

  private
  def generate_pdf(achievement)
    pdf = PDF::Writer.new( :orientation => :landscape )
    pdf.info.title = "Certificate of Achievement"
    draw_border(pdf, 10, 12, 16, 18)
    draw_text(pdf, achievement)
    return pdf
  end

  def draw_border(pdf, *px_pos)
    px_pos.each do |px|
      pdf.rectangle(px, px, pdf.page_width - (px * 2),
                    pdf.page_height - (px * 2)).stroke
    end
  end

  def draw_text(pdf, achievement)
  pdf.select_font "Times-Roman"
    pdf.text("\n", :font_size => 52)
    pdf.text("Certificate of Achievement\n", :justification => :center)
    pdf.text("\n", :font_size => 18)
    pdf.text("hereby granted to\n", :justification => :center)
    pdf.text("\n\n", :font_size => 64)
    pdf.text("in recognition of achieving the status of",
             :font_size => 18, :justification => :center)
    pdf.text(achievement, :font_size => 64, :justification => :center)
  end
end
```

Now we can create a certificate and award it to many different people:

```
certificate = Certificate.new('Ruby Hacker'); false
['Tricia Ball', 'Marty Wise', 'Dung Nguyen'].each do |name|
  certificate.award_to(name).save_as("#{name}.pdf")
end
```

Figure 12-8 shows what `Tricia Ball.pdf` looks like.

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

Figure 12-8. Congratulations!

This recipe only scratches the surface of what you can do with the `PDF::Writer` library. Fortunately, there's an excellent manual and RDoc documentation. Although the library provides a lot of classes, most of the methods you want will be in `PDF::Writer` and the mixin `PDF::Writer::Graphics`.

See Also

- The PDF::Writer homepage (http://ruby-pdf.rubyforge.org/pdf-writer/)
- Generated RDoc (http://ruby-pdf.rubyforge.org/pdf-writer/doc/index.html)
- "Creating Printable Documents with Ruby," published in artima's *Ruby Code & Style*, provides a helpful overview of the library as well as many links to PDF releated resources (http://www.artima.com/rubycs/articles/pdf_writerP.html)
- The `pdf-writer` gem includes the source for the manual (`manual.pwd`)and a script (`bin/techbook`)that turns it into PDF format; the manual is also available online (http://ruby-pdf.rubyforge.org/pdf-writer/manual/index.html)
- If you want to *read* a PDF file and extract its text, try Hannes Wyss's `rpdf2txt` library (http://raa.ruby-lang.org/project/rpdf2txt/)
- Recipe 8.16 for more about the `Marshal` technique for copying an object
- The `Certificate` class is used again in Recipe 14.19, "Running Servlets with WEBrick"

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

Recipe 12.14. Representing Data as MIDI Music

Problem

You want to represent a series of data points as a musical piece, or just create music algorithmically.

Solution

Jim Menard's midilib library makes it easy to generate MIDI music files from Ruby. It's available as the `midilib` gem.

Here's a simple method for visualizing a list of numbers as a piano piece. The largest number in the list is mapped to the highest note on the piano keyboard (MIDI note 108), and the smallest number to the lowest note (MIDI note 21).

```
require 'rubygems'
require 'midilib'                                          # => false

class Array
  def to_midi(file, note_length='eighth')

    midi_max = 108.0
    midi_min = 21.0

    low, high = min, max
    song = MIDI::Sequence.new

    # Create a new track to hold the melody, running at 120 beats per minute.
    song.tracks << (melody = MIDI::Track.new(song))
    melody.events << MIDI::Tempo.new(MIDI::Tempo.bpm_to_mpq(120))

    # Tell channel zero to use the "piano" sound.
    melody.events << MIDI::ProgramChange.new(0, 0)

    # Create a series of note events that play on channel zero.
    each do |number|
      midi_note = (midi_min + ((number-midi_min) * (midi_max-low)/high)).to_i
      melody.events << MIDI::NoteOnEvent.new(0, midi_note, 127, 0)
      melody.events << MIDI::NoteOffEvent.new(0, midi_note, 127,
                                              song.note_to_delta(note_length))
    end

    open(file, 'w') { |f| song.write(f) }
  end
end
```

Now you can get an audible representation of any list of numbers:

```
((1..100).collect { |x| x ** 2 }).to_midi('squares.mid')
```


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

Discussion

The `midilib` library provides a set of classes for modeling a MIDI file: you can parse a MIDI file, modify it with Ruby code, and write it back to disk.

A MIDI file is modeled by a `Sequence` object, which contains `Track` objects. A track is a mainly a series of `Event` objects: for instance, each note in the piece has a `NoteOnEvent` and a `NoteOffEvent`.

`Array#to_midi` works by transforming each number in the array into a corresponding MIDI note. A standard piano keyboard can produce notes ranging from MIDI note 21 to MIDI note 108, with middle C being at MIDI note 60. `Array#to_midi` scales the values of the array to fit into this range as closely as possible, using the same formula you'd use to convert between two temperature scales.

Working directly with the MIDI classes is difficult, especially if you want to compose music instead of just transfering a data stream into MIDI note events. Here's a subclass of `MIDI::Track` that provides some simplifying assumptions and some higher-level musical functions, making it easy to compose simple multitrack tunes. Each `TimedTrack` uses its own MIDI channel and makes sounds from only one instrument. A `TimedTrack` can sound chords (this is very difficult with stock `midilib`), and instead of having to remember the MIDI note range, you can refer to notes in terms of half-steps away from middle C.

```
class TimedTrack < MIDI::Track
  MIDDLE_C = 60
  @@channel_counter=0

  def initialize(number, song)
    super(number)
    @sequence = song
    @time = 0
    @channel = @@channel_counter
    @@channel_counter += 1
  end

  # Tell this track's channel to use the given instrument, and
  # also set the track's instrument display name.
  def instrument=(instrument)
    @events << MIDI::ProgramChange.new(@channel, instrument)
    super(MIDI::GM_PATCH_NAMES[instrument])
  end

  # Add one or more notes to sound simultaneously. Increments the per-track
  # timer so that subsequent notes will sound after this one finishes.
  def add_notes(offsets, velocity=127, duration='quarter')
    offsets = [offsets] unless offsets.respond_to? :each
    offsets.each do |offset|
      event(MIDI::NoteOnEvent.new(@channel, MIDDLE_C + offset, velocity))
    end
    @time += @sequence.note_to_delta(duration)
    offsets.each do |offset|
      event(MIDI::NoteOffEvent.new(@channel, MIDDLE_C + offset, velocity))
    end
    recalc_delta_from_times
```


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

```
  end

  # Uses add_notes to sound a chord (a major triad in root position), using the
  # given note as the low note. Like add_notes, increments the per-track timer.
  def add_major_triad(low_note, velocity=127, duration='quarter')
    add_notes([0, 4, 7].collect { |x| x + low_note }, velocity, duration)
  end

  private

  def event(event)
    @events << event
   event.time_from_start = @time
  end
end
```

Here's a script to write a randomly generated composition with two tracks. The melody track (a trumpet)takes a random walk around the musical scale, and the harmony track (an organ) plays a matching chord at the beginning of each measure.

```
song = MIDI::Sequence.new
song.tracks << (melody = TimedTrack.new(0, song))
song.tracks << (background = TimedTrack.new(1, song))

melody.instrument = 56 # Trumpet
background.instrument = 19 # Church organ

melody.events << MIDI::Tempo.new(MIDI::Tempo.bpm_to_mpq(120))
melody.events << MIDI::MetaEvent.new(MIDI::META_SEQ_NAME,
                                     'A random Ruby composition')

# Some musically pleasing intervals: thirds and fifths.
intervals = [-5, -1, 0, 4, 7]

# Start at middle C.
note = 0
# Create 8 measures of music in 4/4 time
(8*4).times do |i|
  note += intervals[rand(intervals.size)]

  #Reset to middle C if we go out of the MIDI range
  note = 0 if note < -39 or note > 48

  # Add a quarter note on every beat.
  melody.add_notes(note, 127, 'quarter')

  # Add a chord of whole notes at the beginning of each measure.
  background.add_major_triad(note, 50, 'whole') if i % 4 == 0
end

open('random.mid', 'w') { |f| song.write(f) }
```

See Also

- `midilib` has a comprehensive set of RDoc, available online at http://midilib.rubyforge.org/
- The library's `examples/` directory has several good programs that demonstrate how to create and "play" MIDI files
- The `TimedTrack` class presented takes several ideas from Emanuel Borsboom's Midi Scripter application; the Midi Scripter generates MIDI files from Ruby code that

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

incorporates musical notation—it's not really designed for use as a library, but it would make a good one (http://www.epiphyte.ca/downloads/midi_scripter/README.html)

- The names of the standard MIDI instrument and drum sounds are kept in the arrays `MIDI::GM_PATCH_NAMES` and `MIDI::GM_DRUM_NOTE_NAMES`; this isn't as useful as it could be, because you'll usually end up referring to instruments by their numeric IDs; the Wikipedia has a good mapping of numbers to names (http://en.wikipedia.org/wiki/General_MIDI#Program_change_events)

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