CHAPTER 4

Text Processing and File Management

Ruby fills a lot of the same roles that languages such as Perl and Python do. Because of this, you can expect to find first-rate support for text processing and file management. Whether it's parsing a text file with some regular expressions or building some *nix-style filter applications, Ruby can help make life easier.

However, much of Ruby's I/O facilities are tersely documented at best. It is also relatively hard to find good resources that show you general strategies for attacking common text-processing tasks. This chapter aims to expose you to some good tricks that you can use to simplify your text-processing needs as well as sharpen your skills when it comes to interacting with and managing files on your system.

As in other chapters, we'll start off by looking at some real open source code—this time, a simple parser for an Adobe Font Metrics (AFM) file. This example will expose you to text processing in its setting. We'll then follow up with a number of detailed sections that look at different practices that will help you master basic I/O skills. Armed with these techniques, you'll be able to take on all sorts of text-processing and file-management tasks with ease.

Line-Based File Processing with State Tracking

Processing a text document line by line does not mean that we're limited to extracting content in a uniform way, treating each line identically. Some files have more structure than that, but can still benefit from being processed linearly. We're now going to look over a small parser that illustrates this general idea by selecting different ways to extract our data based on what section of a file we are in.

The code in this section was written by James Edward Gray II as part of Prawn's AFM support. Though the example itself is domain-specific, we won't get hung up in the particular details of this parser. Instead, we'll be taking a look at the general approach for how to build a state-aware parser that operates on an efficient line-by-line basis. Along the way, you'll pick up some basic I/O tips and tricks as well as see the important role that regular expressions often play in this sort of task.

Before we look at the actual parser, we can take a glance at the sort of data we're dealing with. AFM files are essentially font glyph measurements and specifications, so they tend to look a bit like a configuration file of sorts. Some of these things are simply straight key/value pairs, such as:

```
CapHeight 718
XHeight 523
Ascender 718
Descender -207
```

Others are organized sets of values within a section, as in the following example:

```
StartCharMetrics 315
C 32 ; WX 278 ; N space ; B 0 0 0 0 ;
C 33 ; WX 278 ; N exclam ; B 90 0 187 718 ;
C 34 ; WX 355 ; N quotedbl ; B 70 463 285 718 ;
C 35 ; WX 556 ; N numbersign ; B 28 0 529 688 ;
C 36 ; WX 556 ; N dollar ; B 32 -115 520 775 ;
....
EndCharMetrics
```

Sections can be nested within each other, making things more interesting. The data across the file does not fit a uniform format, as each section represents a different sort of thing. However, we can come up with patterns to parse data in each section that we're interested in, because they are consistent within their sections. We also are interested in only a subset of the sections, so we can safely ignore some of them. This is the essence of the task we needed to accomplish, but as you may have noticed, it's a fairly abstract pattern that we can reuse. Many documents with a simple section-based structure can be worked with using the approach shown here.

The code that follows is essentially a simple finite state machine that keeps track of what section the current line appears in. It attempts to parse the opening or closing of a section first, and then it uses this information to determine a parsing strategy for the current line. We simply skip the sections that we're not interested in parsing.

We end up with a very straightforward solution. The whole parser is reduced to a simple iteration over each line of the file, which manages a stack of nested sections, while determining whether and how to parse the current line.

We'll see the parts in more detail in just a moment, but here is the whole AFM parser that extracts all the information we need to properly render Adobe fonts in Prawn:

```
def parse_afm(file_name)
  section = []

  File.foreach(file_name) do |line|
    case line
    when /^Start(\w+)/
      section.push $1
      next
    when /^End(\w+)/
      section.pop
      next
```

```
    end

    case section
    when ["FontMetrics", "CharMetrics"]
      next unless line =~ /^CH?\s/

      name                  = line[/\bN\s+(\.?\w+)\s*;/, 1]
      @glyph_widths[name]   = line[/\bWX\s+(\d+)\s*;/, 1].to_i
      @bounding_boxes[name] = line[/\bB\s+([^;]+);/, 1].to_s.rstrip
    when ["FontMetrics", "KernData", "KernPairs"]
      next unless line =~ /^KPX\s+(\.?\w+)\s+(\.?\w+)\s+(-?\d+)/
      @kern_pairs[[$1, $2]] = $3.to_i
    when ["FontMetrics", "KernData", "TrackKern"], ["FontMetrics", "Composites"]
      next
    else
      parse_generic_afm_attribute(line)
    end
  end
end
```

You could try to understand the particular details if you'd like, but it's also fine to black-box the expressions used here so that you can get a sense of the overall structure of the parser. Here's what the code looks like if we do that for all but the patterns that determine the section nesting:

```
def parse_afm(file_name)
  section = []

  File.foreach(file_name) do |line|
    case line
    when /^Start(\w+)/
      section.push $1
      next
    when /^End(\w+)/
      section.pop
      next
    end

    case section
    when ["FontMetrics", "CharMetrics"]
      parse_char_metrics(line)
    when ["FontMetrics", "KernData", "KernPairs"]
      parse_kern_pairs(line)
    when ["FontMetrics", "KernData", "TrackKern"], ["FontMetrics", "Composites"]
      next
    else
      parse_generic_afm_attribute(line)
    end
  end
end
```

With these simplifications, it's very clear that we're looking at an ordinary finite state machine that is acting upon the lines of the file. It also makes it easier to notice what's actually going on.

The first `case` statement is just a simple way to check which section we're currently looking at, updating the stack as necessary as we move in and out of sections:

```
case line
when /^Start(\w+)/
  section.push $1
  next
when /^End(\w+)/
  section.pop
  next
end
```

If we find a section beginning or end, we skip to the next line, as we know there is nothing else to parse. Otherwise, we know that we have to do some real work, which is done in the second `case` statement:

```
case section
when ["FontMetrics", "CharMetrics"]
  next unless line =~ /^CH?\s/

  name                  = line[/\bN\s+(\.?\w+)\s*;/, 1]
  @glyph_widths[name]   = line[/\bWX\s+(\d+)\s*;/, 1].to_i
  @bounding_boxes[name] = line[/\bB\s+([^;]+);/, 1].to_s.rstrip
when ["FontMetrics", "KernData", "KernPairs"]
  next unless line =~ /^KPX\s+(\.?\w+)\s+(\.?\w+)\s+(-?\d+)/
  @kern_pairs[[$1, $2]] = $3.to_i
when ["FontMetrics", "KernData", "TrackKern"], ["FontMetrics", "Composites"]
  next
else
  parse_generic_afm_attribute(line)
end
```

Here, we've got four different ways to handle our line of text. In the first two cases, we process the lines that we need to as we walk through the section, extracting the bits of information we need and ignoring the information we're not interested in.

In the third case, we identify certain sections to skip and simply resume processing the next line if we are currently within that section.

Finally, if the other cases fail to match, our last `case` scenario assumes that we're dealing with a simple key/value pair, which is handled by a private helper method in Prawn. Because it does not provide anything different to look at than the first two sections of this `case` statement, we can safely ignore how it works without missing anything important.

However, the interesting thing that you might have noticed is that the first case and the second case use two different ways of extracting values. The code that processes `CharMetrics` uses `String#[]`, whereas the code handling `KernPairs` uses Perl-style global match variables. The reason for this is largely convenience. The following two lines of code are equivalent:

```
name = line[/\bN\s+(\.?\w+)\s*;/, 1]
name = line =~ /\bN\s+(\.?\w+)\s*;/ && $1
```

There are still other ways to handle your captured matches (such as `MatchData` via `String#match`), but we'll get into those later. For now, it's simply worth knowing that when you're trying to extract a single matched capture, `String#[]` does the job well, but if you need to deal with more than one, you need to use another approach. We see this clearly in the second case:

```
next unless line =~ /^KPX\s+(\.?\w+)\s+(\.?\w+)\s+(-?\d+)/
@kern_pairs[[$1, $2]] = $3.to_i
```

This code is a bit clever, as the line that assigns the values to `@kern_pairs` gets executed only when there is a successful match. When the match fails, it will return `nil`, causing the parser to skip to the next line for processing.

We could continue studying this example, but we'd then be delving into the specifics, and those details aren't important for remembering this simple general pattern.

When dealing with a structured document that can be processed by discrete rules for each section, the general approach is simple and does not typically require pulling the entire document into memory or doing multiple passes through the data.

Instead, you can do the following:

- Identify the beginning and end markers of sections with a pattern.
- If sections are nested, maintain a stack that you update before further processing of each line.
- Break up your extraction code into different cases and select the right one based on the current section you are in.
- When a line cannot be processed, skip to the next one as soon as possible, using the `next` keyword.
- Maintain state as you normally would, processing whatever data you need.

By following these basic guidelines, you can avoid overthinking your problem, while still saving clock cycles and keeping your memory footprint low. Although the code here solves a particular problem, it can easily be adapted to fit a wide range of basic document processing needs.

This introduction has hopefully provided a taste of what text processing in Ruby is all about. The rest of the chapter will provide many more tips and tricks, with a greater focus on the particular topics. Feel free to jump around to the things that interest you most, but I'm hoping all of the sections have something interesting to offer—even to seasoned Rubyists.

Regular Expressions

At the time of writing this chapter, I was spending some time watching the Dow Jones Industrial Average, as the world was in the middle of a major financial meltdown. If

you're wondering what this has to do with Ruby or regular expressions, take a quick look at the following code:

```
require "open-uri"
loop do
  puts( open("http://finance.google.com/finance?cid=983582").read[
  /<span class="\w+" id="ref_983582_c">([+-]?\d+\.\d+)/m, 1] )
  sleep(30)
end
```

In just a couple of lines, I was able to throw together a script that would poll Google Finance and pull down the current average price of the Dow. This sort of "find a needle in the haystack" extraction is what regular expressions are all about.

Of course, the art of constructing regular expressions is often veiled in mystery. Even simple patterns such as this one might make some folks feel a bit uneasy:

```
/<span class="\w+" id="ref_983582_c">([+-]?\d+\.\d+)/m
```

This expression is simple by comparison to some other examples we can show, but it still makes use of a number of regular expression concepts. All in one line, we can see the use of character classes (both general and special), escapes, quantifiers, groups, and a switch that enables multiline matching.

Patterns are dense because they are written in a special syntax, which acts as a sort of domain language for matching and extracting text. The reason that it may be considered daunting is that this language is made up of so few special characters:

```
\ [ ] . ^ $ ? * + { } | ( )
```

At its heart, regular expressions are nothing more than a facility to do find and replace operations. This concept is so familiar that anyone who has used a word processor has a strong grasp on it. Using a regex, you can easily replace all instances of the word "Mitten" with "Kitten", just like your favorite text editor or word processor can:

```
some_string.gsub(/\bMitten\b/,"Kitten")
```

Many programmers get this far and stop. They learn to use regex as if it were a necessary evil rather than an essential technique. We can do better than that. In this section, we'll look at a few guidelines for how to write effective patterns that do what they're supposed to without getting too convoluted. I'm assuming you've done your homework and are at least familiar with regex basics as well as Ruby's pattern syntax. If that's not the case, pick up your favorite language reference and take a few minutes to review the fundamentals.

As long as you can comfortably read the first example in this section, you're ready to move on. If you can convince yourself that writing regular expressions is actually much easier than people tend to think it is, the tips and tricks to follow shouldn't cause you to break a sweat.

Don't Work Too Hard

Despite being such a compact format, it's relatively easy to write bloated patterns if you don't consciously remember to keep things clean and tight. We'll now take a look at a couple sources of extra fat and see how to trim them down.

Alternation is a very powerful regex tool. It allows you to match one of a series of potential sequences. For example, if you want to match the name "James Gray" but also match "James gray", "james Gray", and "james gray", the following code will do the trick:

```
>> ["James Gray", "James gray", "james gray", "james Gray"].all? { |e|
?>   e.match(/James|james Gray|gray/) }
=> true
```

However, you don't need to work so hard. You're really talking about possible alternations of simply two characters, not two full words. You could write this far more efficiently using a character class:

```
>> ["James Gray", "James gray", "james gray", "james Gray"].all? { |e|
?>   e.match(/[Jj]ames [Gg]ray/) }
=> true
```

This makes your pattern clearer and also will result in a much better optimization in Ruby's regex engine. So in addition to looking better, this code is actually faster.

In a similar vein, it is unnecessary to use explicit character classes when a shortcut will do. To match a four-digit number, we could write:

```
/[0-9][0-9][0-9][0-9]/
```

which can of course be cleaned up a bit using repetitions:

```
/[0-9]{4}/
```

However, we can do even better by using the special class built in for this:

```
/\d{4}/
```

It pays to learn what shortcuts are available to you. Here's a quick list for further study, in case you're not already familiar with them:

```
. \s \S \w \W \d \D
```

Each one of these shortcuts corresponds to a literal character class that is more verbose when written out. Using shortcuts increases clarity and decreases the chance of bugs creeping in via ill-defined patterns. Though it may seem a bit terse at first, you'll be able to sight-read them with ease over time.

Anchors Are Your Friends

One way to match my name in a string is to write the following simple pattern:

```
string =~ /Gregory Brown/
```

However, consider the following:

```
>> "matched" if "Mr. Gregory Browne".match(/Gregory Brown/)
=> "matched"
```

Oftentimes we mean "match this phrase," but we write "match this sequence of characters." The solution is to make use of anchors to clarify what we mean.

Sometimes we want to match only if a string starts with a phrase:

```
>> phrases = ["Mr. Gregory Browne", "Mr. Gregory Brown is cool",
              "Gregory Brown is cool", "Gregory Brown"]

>> phrases.grep /\AGregory Brown\b/
=> ["Gregory Brown is cool", "Gregory Brown"]
```

Other times we want to ensure that the string contains the phrase:

```
>> phrases.grep /\bGregory Brown\b/
=> ["Mr. Gregory Brown is cool", "Gregory Brown is cool", "Gregory Brown"]
```

And finally, sometimes we want to ensure that the string matches an exact phrase:

```
>> phrases.grep /\AGregory Brown\z/
=> ["Gregory Brown"]
```

Although I am using English names and phrases here for simplicity, this can of course be generalized to encompass any sort of matching pattern. You could be verifying that a sequence of numbers fits a certain form, or something equally abstract. The key thing to take away from this is that when you use anchors, you're being much more explicit about how you expect your pattern to match, which in most cases means that you'll have a better chance of catching problems faster, and an easier time remembering what your pattern was supposed to do.

An interesting thing to note about anchors is that they don't actually match characters. Instead, they match between characters to allow you to assert certain expectations about your strings. So when you use something like `\b`, you are actually matching between one of `\w\W` , `\W\w` , `\A` , `\z`. In English, that means that you're transitioning from a word character to a nonword character, or from a nonword character to a word character, or you're matching the beginning or end of the string. If you review the use of `\b` in the previous examples, it should now be very clear how anchors work.

The full list of available anchors in Ruby is `\A`, `\Z`, `\z`, `^`, `$`, and `\b`. Each has its own merits, so be sure to read up on them.

Use Caution When Working with Quantifiers

One of the most common antipatterns I picked up when first learning regular expressions was to make use of `.*` everywhere. Though this practice may seem innocent, it is similar to my bad habit of using `rm -Rf` on the command line all the time instead of just `rm`. Both can result in catastrophe when used incorrectly.

But maybe you're not as crazy as I am. Instead, maybe you've been writing innocent things like `/(\d*)Foo/` to match any number of digits prepended to the word "Foo":

For some cases, this works great:

```
>> "1234Foo"[/(\d*)Foo/,1]
=> "1234"
```

But does this surprise you?

```
>> "xFoo"[/(\d*)Foo/,1]
=> ""
```

It may not, but then again, it may. It's relatively common to forget that * always matches. At first glance, the following code seems fine:

```
if num = string[/(\d*)Foo/,1]
  Integer(num)
end
```

However, because the match will capture an empty string in its failure case, this code will break. The solution is simple. If you really mean "at least one," use + instead:

```
if num = string[/(\d+)Foo/,1]
  Integer(num)
end
```

Though more experienced folks might not easily be trapped by something so simple, there are more subtle variants. For example, if we intend to match only "Greg" or "Gregory", the following code doesn't quite work:

```
>> "Gregory"[/Greg(ory)?/]
=> "Gregory"
>> "Greg"[/Greg(ory)?/]
=> "Greg"
>> "Gregor"[/Greg(ory)?/]
=> "Greg"
```

Even if the pattern looks close to what we want, we can see the results don't fit. The following modifications remedy the issue:

```
>> "Gregory"[/\bGreg(ory)?\b/]
=> "Gregory"
>> "Greg"[/\bGreg(ory)?\b/]
=> "Greg"
>> "Gregor"[/\bGreg(ory)?\b/]
=> nil
```

Notice that the pattern now properly matches Greg or Gregory, but no other words. The key thing to take away here is that unbounded zero-matching quantifiers are tautologies. They can never fail to match, so you need to be sure to account for that.

A final gotcha about quantifiers is that they are greedy by default. This means they'll try to consume as much of the string as possible before matching. The following is an example of a greedy match:

```
>> "# x # y # z #"[/#(.*)#/,1]
=> " x # y # z "
```

As you can see, this code matches everything between the first and last # character. But sometimes, we want processing to happen from the left and end as soon as we have a match. To do this, append a ? to the repetition:

```
>> "# x # y # z #"[/#(.*?)#/,1]
=> " x "
```

All quantifiers can be made nongreedy this way. Remembering this will save a lot of headaches in the long run.

Though our treatment of regular expressions has been by no means comprehensive, these few basic tips will really carry you a long way. The key things to remember are:

- Regular expressions are nothing more than a special language for find-and-replace operations, built on simple logical constructs.
- There are lots of shortcuts built in for common regular expression operations, so be sure to make use of special character classes and other simplifications when you can.
- Anchors provide a way to set up some expectation about where in a string you want to look for a match. These help with both optimization and pattern correctness.
- Quantifiers such as * and ? will always match, so they should not be used without sufficient boundaries.
- Quantifiers are greedy by default, and can be made nongreedy via ?.

By following these guidelines, you'll write clearer, more accurate, and faster regular expressions. As a result, it'll be a whole lot easier to revisit them when you run into them in your own old code a few months down the line.

A final note on regular expressions is that sometimes we are seduced by their power and overlook other solutions that may be more robust for certain needs. In both the stock ticker and AFM parsing examples, we were working within the realm where regular expressions are a quick, easy, and fine way to go.

However, as documents take on more complex structures, and your needs move from extracting some values to attempting to fully parse a document, you will probably need to look to other techniques that involve full-blown parsers such as Treetop, Ghost Wheel, or Racc. These libraries can solve problems that regular expressions can't solve, and if you find yourself with data that's hard to map a regex to, it's worth looking at these alternative solutions.

Of course, your mileage will vary based on the problem at hand, so don't be afraid of trying a regex-based solution first before pulling out the big guns.

Working with Files

There are a whole slew of options for doing various file management tasks in Ruby. Because of this, it can be difficult to determine what the best approach for a given task might be. In this section, we'll cover two key tasks while looking at three of Ruby's standard libraries.

First, you'll learn how to use the *pathname* and *fileutils* libraries to traverse your filesystem using a clean cross-platform approach that rivals the power of popular *nix shells without sacrificing compatibility. We'll then move on to how to use *tempfile* to automate handling of temporary file resources within your scripts. These practical tips will help you write platform-agnostic Ruby code that'll work out of the box on more systems, while still managing to make your job easier.

Using Pathname and FileUtils

If you are using Ruby to write administrative scripts, it's nearly inevitable that you've needed to do some file management along the way. It may be quite tempting to drop down into the shell to do things like move and rename directories, search for files in a complex directory structure, and other common tasks that involve ferrying files around from one place to the other. However, Ruby provides some great tools to avoid this sort of thing.

The *pathname* and *fileutils* standard libraries provide virtually everything you need for file management. The best way to demonstrate their capabilities is by example, so we'll now take a look at some code and then break it down piece by piece.

To illustrate `Pathname`, we can take a look at a small tool I've built for doing local installations of libraries found on GitHub. This script, called *mooch*, essentially looks up and clones a git repository, puts it in a convenient place within your project (a *vendor/* directory), and optionally sets up a stub file that will include your vendored packages into the loadpath upon requiring it. Sample usage looks something like this:

```
$ mooch init lib/my_project
$ mooch sandal/prawn  0.2.3
$ mooch ruport/ruport 1.6.1
```

We can see the following will work without loading RubyGems:

```
>> require "lib/my_project/dependencies"
=> true
>> require "prawn"
=> true
>> require "ruport"
=> true
>> Prawn::VERSION
=> "0.2.3"
>> Ruport::VERSION
=> "1.6.1"
```

Although this script is pretty useful, that's not what we're here to talk about. Instead, let's focus on how this sort of thing is built, as it shows a practical example of using `Pathname` to manipulate files and folders. I'll start by showing you the whole script, and then we'll walk through it part by part:

```
#!/usr/bin/env ruby
require "pathname"

WORKING_DIR = Pathname.getwd
LOADER = %Q{
  require "pathname"

  Pathname.glob("#{WORKING_DIR}/vendor/*/*/") do |dir|
   lib = dir + "lib"
   $LOAD_PATH.push(lib.directory? ? lib : dir)
  end
}

if ARGV[0] == "init"
  lib = Pathname.new(ARGV[1])
  lib.mkpath
  (lib + 'dependencies.rb').open("w") do |file|
    file.write LOADER
  end
else
  vendor = Pathname.new("vendor")
  vendor.mkpath
  Dir.chdir(vendor.realpath)
  system("git clone git://github.com/#{ARGV[0]}.git #{ARGV[0]}")
  if ARGV[1]
    Dir.chdir(ARGV[0])
    system("git checkout #{ARGV[1]}")
  end
end
```

As you can see, it's not a ton of code, even though it does a lot. Let's shine the spotlight on the interesting `Pathname` bits:

```
WORKING_DIR = Pathname.getwd
```

Here we are simply assigning the initial working directory to a constant. We use this to build up the code for the *dependencies.rb* stub script that can be generated via `mooch init`. Here we're just doing quick-and-dirty code generation, and you can see the full stub as stored in `LOADER`:

```
LOADER = %Q{
  require "pathname"

  Pathname.glob("#{WORKING_DIR}/vendor/*/*/") do |dir|
    lib = dir + "lib"
    $LOAD_PATH.push(lib.directory? ? lib : dir)
  end
}
```

This script does something fun. It looks in the working directory that `mooch init` was run in for a folder called *vendor*, and then looks for folders two levels deep fitting the GitHub convention of *username/project*. We then use a `glob` to traverse the directory structure, in search of folders to add to the loadpath. The code will check to see whether each project has a *lib* folder within it (as is the common Ruby convention), but will add the project folder itself to the loadpath if it is not present.

Here we notice a few of `Pathname`'s niceties. You can see we can construct new paths by just adding new strings to them, as shown here:

```
lib = dir + "lib"
```

In addition to this, we can check to see whether the path we've created actually points to a directory on the filesystem, via a simple `Pathname#directory?` call. This makes traversal downright easy, as you can see in the preceding code.

This simple stub may be a bit dense, but once you get the hang of `Pathname`, you can see that it's quite powerful. Let's look at a couple more tricks, focusing this time on the code that actually writes this snippet to file:

```
lib = Pathname.new(ARGV[1])
lib.mkpath
(lib + 'dependencies.rb').open("w") do |file|
  file.write LOADER
end
```

Before, the invocation looked like this:

```
$ mooch init lib/my_project
```

Here, `ARGV[1]` is *lib/my_project*. So, in the preceding code, you can see we're building up a relative path to our current working directory and then creating a folder structure. A very cool thing about `Pathname` is that it works in a similar way to `mkdir -p` on *nix, so `Pathname#mkpath` will actually create any necessary nesting directories as needed, and won't complain if the structure already exists, which are both results that we want here.

Once we build up the directories, we need to create our *dependencies.rb* file and populate it with the string in `LOADER`. We can see here that `Pathname` provides shortcuts that work in a similar fashion to `File.open()`.

In the code that actually downloads and vendors libraries from GitHub, we see the same techniques in use yet again, this time mixed in with some shell commands and `Dir.chdir`. As this doesn't introduce anything new, we can skip over the details.

Before we move on to discussing temporary files, we'll take a quick look at `FileUtils`. The purpose of this module is to provide a Unix-like interface to file manipulation tasks, and a quick look at its method list will show that it does a good job of this:

```
cd(dir, options)
cd(dir, options) {|dir| .... }
pwd()
mkdir(dir, options)
mkdir(list, options)
```

```
mkdir_p(dir, options)
mkdir_p(list, options)
rmdir(dir, options)
rmdir(list, options)
ln(old, new, options)
ln(list, destdir, options)
ln_s(old, new, options)
ln_s(list, destdir, options)
ln_sf(src, dest, options)
cp(src, dest, options)
cp(list, dir, options)
cp_r(src, dest, options)
cp_r(list, dir, options)
mv(src, dest, options)
mv(list, dir, options)
rm(list, options)
rm_r(list, options)
rm_rf(list, options)
install(src, dest, mode = <src's>, options)
chmod(mode, list, options)
chmod_R(mode, list, options)
chown(user, group, list, options)
chown_R(user, group, list, options)
touch(list, options)
```

You'll see a bit more of `FileUtils` later on in the chapter when we talk about atomic saves. But before we jump into advanced file management techniques, let's review another important foundational tool: the *tempfile* standard library.

The tempfile Standard Library

Producing temporary files is a common need in many applications. Whether you need to store something on disk to keep it out of memory until it is needed again, or you want to serve up a file but don't need to keep it lurking around after your process has terminated, odds are you'll run into this problem sooner or later.

It's quite tempting to roll our own `Tempfile` support, which might look something like the following code:

```
File.open("/tmp/foo.txt","w") do |file|
  file << some_data
end

# Then in some later code

File.foreach("/tmp/foo.txt") do |line|
  # do something with data
end

# Then finally
require "fileutils"
FileUtils.rm("/tmp/foo.txt")
```

This code works, but it has some drawbacks. The first is that it assumes that you're on a *nix system with a */tmp* directory. Secondly, we don't do anything to avoid file collisions, so if another application is using */tmp/foo.txt*, this will overwrite it. Finally, we need to explicitly remove the file, or risk leaving a bunch of trash around.

Luckily, Ruby has a standard library that helps us get around these issues. Using it, our example then looks like this:

```
require "tempfile"
temp = Tempfile.new("foo.txt")
temp << some_data

# then in some later code
temp.rewind
temp.each do |line|
  # do something with data
end

# Then finally
temp.close
```

Let's take a look at what's going on in a little more detail, to really get a sense of what the *tempfile* library is doing for us.

Automatic Temporary Directory Handling

The code looks somewhat similar to our original example, as we're still essentially working with an `IO` object. However, the approach is different. `Tempfile` opens up a file handle for us to a file that is stored in whatever your system's *tempdir* is. We can inspect this value, and even change it if we need to. Here's what it looks like on two of my systems:

```
>> Dir.tmpdir
=> "/var/folders/yH/yHvUeP-oFYamIyTmRPPoKE+++TI/-Tmp-"

>> Dir.tmpdir
=> "/tmp"
```

Usually, it's best to go with whatever this value is, because it is where Ruby thinks your temp files should go. However, in the cases where we want to control this ourselves, it is simple to do so, as shown in the following:

```
temp = Tempfile.new("foo.txt", "path/to/my/tmpdir")
```

Collision Avoidance

When you create a temporary file with `Tempfile.new`, you aren't actually specifying an exact filename. Instead, the filename you specify is used as a base name that gets a unique identifier appended to it. This prevents one temp file from accidentally overwriting another. Here's a trivial example that shows what's going on under the hood:

```
>> a = Tempfile.new("foo.txt")
=> #<File:/tmp/foo.txt.2021.0>
>> b = Tempfile.new("foo.txt")
=> #<File:/tmp/foo.txt.2021.1>
>> a.path
=> "/tmp/foo.txt.2021.0"
>> b.path
=> "/tmp/foo.txt.2021.1"
```

Allowing Ruby to handle collision avoidance is generally a good thing, especially if you don't normally care about the exact names of your temp files. Of course, we can always rename the file if we need to store it somewhere permanently.

Same Old I/O Operations

Because we're dealing with an object that delegates most of its functionality directly to `File`, we can use normal `File` methods, as shown in our example. For this reason, we can write to our file handle as expected:

```
temp << some_data
```

and read from it in a similar fashion:

```
# then in some later code
temp.rewind
temp.each do |line|
  # do something with data
end
```

Because we leave the file handle open, we need to rewind it to point to the beginning of the file rather than the end. Beyond that, the behavior is exactly the same as `File#each`.

Automatic Unlinking

`Tempfile` cleans up after itself. There are two main ways of unlinking a file; which one is correct depends on your needs. Simply closing the file handle is good enough, and it is what we use in our example:

```
temp.close
```

In this case, Ruby doesn't remove the temporary file right away. Instead, it will keep it around until all references to `temp` have been garbage-collected. For this reason, if keeping lots of open file handles around is a problem for you, you can actually close your handles without fear of losing your temp file, as long as you keep a reference to it handy.

However, in other situations, you may want to purge the file as soon as it has been closed. The change to make this happen is trivial:

```
temp.close!
```

Finally, if you need to explicitly delete a file that has already been closed, you can just use the following:

```
temp.unlink
```

In practice, you don't need to think about this in most cases. Instead, *tempfile* works as you might expect, keeping your files around while you need them and cleaning up after itself when it needs to. If you forget to close a temporary file explicitly, it'll be unlinked when the process exits. For these reasons, using the *tempfile* library is often a better choice than rolling your own solution.

There is more to be said about this very cool library, but what we've already discussed covers most of what you'll need day to day, so now is a fine time to go over what's been said and move on to the next thing.

We've gone over some of the tools Ruby provides for working with your filesystem in a platform-agnostic way, and we're about to get into some more advanced strategies for managing, processing, and manipulating your files and their contents. However, before we do that, let's review the key points about working with your filesystem and with temp files:

- There are a whole slew of options for file management in Ruby, including `FileUtils`, `Dir`, and `Pathname`, with some overlap between them.
- `Pathname` provides a high-level, modern Ruby interface to managing files and traversing your filesystem.
- `FileUtils` provides a *nix-style API to file management tools, but works just fine on any system, making it quite useful for porting shell scripts to Ruby.
- The *tempfile* standard library provides a convenient IO-like class for dealing with temp files in a system-independent way.
- The *tempfile* library also helps make things easier through things like name collision avoidance, automatic file unlinking, and other niceties.

With these things in mind, we'll see more of the techniques shown in this section later on in the chapter. But if you're bored with the basics, now is the time to look at higher-level strategies for doing common I/O tasks.

Text-Processing Strategies

Ruby makes basic I/O operations dead simple, but this doesn't mean it's a bad idea to pick up and apply some general approaches to text processing. Here we'll talk about two techniques that most programmers doing file processing will want to know about, and you'll see what they look like in Ruby.

Advanced Line Processing

The case study for this chapter showed the most common use of `File.foreach()`, but there is more to be said about this approach. This section will highlight a couple of tricks worth knowing about when doing line-by-line processing.

Using Enumerator

The following example shows code that extracts and sums the totals found in a file that has entries similar to these:

```
some
lines
of
text
total: 12

other
lines
of
text
total: 16

more
text
total: 3
```

The following code shows how to do this without loading the whole file into memory:

```
sum = 0
File.foreach("data.txt") { |line| sum += line[/total: (\d+)/,1].to_f }
```

Here, we are using `File.foreach` as a direct iterator, and building up our sum as we go. However, because `foreach()` returns an `Enumerator`, we can actually write this in a cleaner way without sacrificing efficiency:

```
enum = File.foreach("data.txt")
sum = enum.inject(0) { |s,r| s + r[/total: (\d+)/,1].to_f }
```

The primary difference between the two approaches is that when you use `File.foreach` directly with a block, you are simply iterating line by line over the file, whereas `Enumerator` gives you some more powerful ways of processing your data.

When we work with arrays, we don't usually write code like this:

```
sum = 0
arr.each { |e| sum += e }
```

Instead, we typically let Ruby do more of the work for us:

```
sum = arr.inject(0) { |s,e| s + e }
```

For this reason, we should do the same thing with files. If we have an `Enumerable` method we want to use to transform or process a file, we should use the enumerator provided by `File.foreach()` rather than try to do our processing within the block. This will allow

us to leverage the power behind Ruby's `Enumerable` module rather than doing the heavy lifting ourselves.

Tracking line numbers

If you're interested in certain line numbers, there is no need to maintain a manual counter. You simply need to create a file handle to work with, and then make use of the `File#lineno` method. To illustrate this, we can very easily implement the Unix command `head`:

```
def head(file_name,max_lines = 10)
  File.open(file_name) do |file|
    file.each do |line|
      puts line
      break if file.lineno == max_lines
    end
  end
end
```

For a more interesting use case, we can consider a file that is formatted in two line pairs, the first line a key, the second a value:

```
first name
gregory
last name
brown
email
gregory.t.brown@gmail.com
```

Using `File#lineno`, this is trivial to process:

```
keys   = []
values = []

File.open("foo.txt") do |file|
  file.each do |line|
    (file.lineno.odd? ? keys : values) << line.chomp
  end
end

Hash[*keys.zip(values).flatten]
```

The result of this code is a simple hash, as you might expect:

```
{ "first name" => "gregory",
  "last name"  => "brown",
  "email"      => "gregory.t.brown@gmail.com" }
```

Though there is probably more we can say about iterating over files line by line, this should get you well on your way. For now, there are other important I/O strategies to investigate, so we'll keep moving.

Atomic Saves

Although many file processing scripts can happily read in one file as input and produce another as output, sometimes we want to be able to do transformations directly on a single file. This isn't hard in practice, but it's a little bit less obvious than you might think.

It is technically possible to rewrite parts of a file using the `"r+"` file mode, but in practice, this can be unwieldy in most cases. An alternative approach is to load the entire contents of a file into memory, manipulate the string, and then overwrite the original file. However, this approach is wasteful, and is not the best way to go in most cases.

As it turns out, there is a simple solution to this problem, and that is simply to work around it. Rather than trying to make direct changes to a file, or store a string in memory and then write it back out to the same file after manipulation, we can instead make use of a temporary file and do line-by-line processing as normal. When we finish the job, we can rename our temp file so as to replace the original. Using this approach, we can easily make a backup of the original file if necessary, and also roll back changes upon error.

Let's take a quick look at an example that demonstrates this general strategy. We'll build a script that strips comments from Ruby files, allowing us to take source code such as this:

```
# The best class ever
# Anywhere in the world
class Foo

  # A useless comment
  def a
     true
  end

  #Another Useless comment
  def b
    false
  end

end
```

and turn it into comment-free code such as this:

```
class Foo

  def a
     true
  end

  def b
    false
  end

end
```

With the help of Ruby's *tempfile* and *fileutils* standard libraries, this task is trivial:

```
require "tempfile"
require "fileutils"
temp = Tempfile.new("working")
File.foreach(ARGV[0]) do |line|
  temp << line unless line =~ /^\s*#/
end

temp.close
FileUtils.mv(temp.path,ARGV[0])
```

We initialize a new `Tempfile` object and then iterate over the file specified on the command line. We append each line to the `Tempfile`, as long as it is not a comment line. This is the first part of our task:

```
temp = Tempfile.new("working")
File.foreach(ARGV[0]) do |line|
  temp << line unless line =~ /^\s*#/
end

temp.close
```

Once we've written our `Tempfile` and closed the file handle, we then use `FileUtils` to rename it and replace the original file we were working on:

```
FileUtils.mv(temp.path,ARGV[0])
```

In two steps, we've efficiently modified a file without loading it entirely into memory or dealing with the complexities of using the `r+` file mode. In many cases, the simple approach shown here will be enough.

Of course, because you are modifying a file in place, a poorly coded script could risk destroying your input file. For this reason, you might want to make a backup of your file. This can be done trivially with `FileUtils.cp`, as shown in the following reworked version of our example:

```
require "tempfile"
require "fileutils"

temp = Tempfile.new("working")
File.foreach(ARGV[0]) do |line|
  temp << line unless line =~ /^\s*#/
end

temp.close
FileUtils.cp(ARGV[0],"#{ARGV[0]}.bak")
FileUtils.mv(temp.path,ARGV[0])
```

This code makes a backup of the original file only if the temp file is successfully populated, which prevents it from producing garbage during testing.

Sometimes it will make sense to do backups; other times, it won't be essential. Of course, it's better to be safe than sorry, so if you're in doubt, just add the extra line of code for a bit more peace of mind.

The two strategies shown in this section will come up in practice again and again for those doing frequent text processing. They can even be used in combination when needed.

We're about to close our discussion on this topic, but before we do that, it's worth mentioning the following reminders:

- When doing line-based file processing, `File.foreach` can be used as an `Enumerator`, unlocking the power of `Enumerable`. This provides an extremely handy way to search, traverse, and manipulate files without sacrificing efficiency.
- If you need to keep track of which line of a file you are on while you are iterating over it, you can use `File#lineno` rather than incrementing your own counter.
- When doing atomic saves, the *tempfile* standard library can be used to avoid unnecessary clutter.
- Be sure to test any code that does atomic saves thoroughly, as there is real risk of destroying your original source files if backups are not made.

Conclusions

When dealing with text processing and file management in Ruby, there are a few things to keep in mind. Most of the pitfalls you can run into while doing this sort of work tend to have to do with performance, platform dependence, or code that doesn't clean up after itself.

In this chapter, we talked about a couple of standard libraries that can help keep things clean and platform-independent. Though Ruby is a fine language to write shell scripts in, there is often no need to resort to code that will run only on certain machines when a pure Ruby solution is just as clean. For this reason, using libraries such as *tempfile*, *pathname*, and *fileutils* will go a seriously long way toward keeping your code portable and maintainable down the line.

For issues of performance, you can almost always squeeze out extra speed and minimize your memory footprint by processing your data line by line rather than slurping everything into a single string. You can also much more effectively find a needle in the haystack if you form well-crafted regular expressions that don't make Ruby work too hard. The techniques we've shown here serve as reminders about common mistakes that even seasoned Rubyists tend to make, and provide good ways around them.

Text processing and file management can quickly become complex, but with a solid grasp of the fundamental strategies, you can use Ruby as an extremely powerful tool that works faster and more effectively than you might imagine.