# Table of Contents




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

# 6. Files and Directories

As programming languages increase in power, we programmers get further and further from the details of the underlying machine language. When it comes to the operating system, though, even the most modern programming languages live on a level of abstraction that looks a lot like the C and Unix libraries that have been around for decades.

We covered this kind of situation in Chapter 3 with Ruby's `Time` objects, but the issue really shows up when you start to work with files. Ruby provides an elegant object-oriented interface that lets you do basic file access, but the more advanced file libraries tend to look like the C libraries they're based on. To lock a file, change its Unix permissions, or read its metadata, you'll need to remember method names like `mtime`, and the meaning of obscure constants like `File::LOCK_EX` and `0644`. This chapter will show you how to use the simple interfaces, and how to make the more obscure interfaces easier to use.



Looking at Ruby's support for file and directory operations, you'll see four distinct tiers of support. The most common operations tend to show up on the lowernumbered tiers:

1. `File` objects to read and write the contents of files, and `Dir` objects to list the contents of directories. For examples, see Recipes 6.5, 6.7, and 6.17. Also see Recipe 6.13 for a Ruby-idiomatic approach.
2. Class methods of `File` to manipulate files without opening them. For instance, to delete a file, examine its metadata, or change its permissions. For examples, see Recipes 6.1, 6.3, and 6.4.
3. Standard libraries, such as `find` to walk directory trees, and `fileutils` to perform common filesystem operations like copying files and creating directories. For examples, see Recipes 6.8, 6.12, and 6.20.
4. Gems like `file-tail`, `lockfile`, and `rubyzip`, which fill in the gaps left by the standard library. Most of the file-related gems covered in this book deal with specific file formats, and are covered in Chapter 12.

`Kernel#open` is the simplest way to open a file. It returns a `Filel` object that you can read from or write to, depending on the "mode" constant you pass in. I'll introduce read mode and write mode here; there are several others, but I'll talk about most of those as they come up in recipes.

To write data to a file, pass a mode of '`w`' to `open`. You can then write lines to the file with `File#puts`, just like printing to standard output with `Kernel#puts`. For more possibilities, see Recipe 6.7.


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

```
open('beans.txt', "w") do |file|
  file.puts('lima beans')
  file.puts('pinto beans')
  file.puts('human beans')
end
```

To read data from a file, open it for read access by specifying a mode of 'r', or just omitting the mode. You can slurp the entire contents into a string with `File#read`, or process the file line-by-line with `File#each`. For more details, see Recipe 6.6.

```
open('beans.txt') do |file|
   file.each { |l| puts "A line from the file: #{l}" }
end
# A line from the file: lima beans
# A line from the file: pinto beans
# A line from the file: human beans
```

As seen in the examples above, the best way to use the `open` method is with a code block. The `open` method creates a new `File` object, passes it to your code block, and closes the file automatically after your code block runs—even if your code throws an exception. This saves you from having to remember to close the file after you're done with it. You could rely on the Ruby interpreter's garbage collection to close the file once it's no longer being used, but Ruby makes it easy to do things the right way.

To find a file in the first place, you need to specify its disk path. You may specify an absolute path, or one relative to the current directory of your Ruby process (see Recipe 6.21). Relative paths are usually better, because they're more portable across platforms. Relative paths like "beans.txt" or "subdir/beans.txt" will work on any platform, but absolute Unix paths look different from absolute Windows paths:

```
# A stereotypical Unix path.
open('/etc/passwd')

# A stereotypical Windows path; note the drive letter.
open('c:/windows/Documents and Settings/User1/My Documents/ruby.doc')
```

Windows paths in Ruby use forward slashes to separate the parts of a path, even though Windows itself uses backslashes. Ruby will also accept backslashes in a Windows path, so long as you escape them:

```
open('c:\\windows\\Documents and Settings\\User1\\My Documents\\ruby.doc')
```

Although this chapter focuses mainly on disk files, most of the methods of `File` are actually methods of its superclass, IO. You'll encounter many other classes that are also subclasses of `IO`, or just respond to the same methods. This means that most of the tricks described in this chapter are applicable to classes like the `Socket` class for Internet sockets and the infinitely useful `StringIO` (see Recipe 6.15).


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

Your Ruby program's standard input, output, and error (`$stdin`, `$stdout`, and `$stderr`) are also `IO` objects, which means you can treat them like files. This one-line program echoes its input to its output:

```
$stdin.each { |l| puts l }
```

The `Kernel#puts` command just calls `$stdout.puts`, so that one-liner is equivalent to this one:

```
$stdin.each { |l| $stdout.puts l }
```

Not all file-like objects support all the methods of `IO`. See Recipe 6.11 for ways to get around the most common problem with unsupported methods. Also see Recipe 6.16 for more on the default `IO` objects.

Several of the recipes in this chapter (such as Recipes 6.12 and 6.20) create specific directory structures to demonstrate different concepts. Rather than bore you by filling up recipes with the Ruby code to create a certain directory structure, I've written a method that takes a short description of a directory structure, and creates the appropriate files and subdirectories:

```
# create_tree.rb
def create_tree(directories, parent=".")
  directories.each_pair do |dir, files|
    path = File.join(parent, dir)
    Dir.mkdir path unless File.exists? path
    files.each do |filename, contents|
      if filename.respond_to? :each_pair  # It's a subdirectory
        create_tree filename, path
      else # It's a file
        open(File.join(path, filename), 'w') { |f| f << contents || "" }
      end
    end
   end
  end
```

Now I can present th directory structure as a data structure and you can create it with a single method call:

```
require 'create_tree'
create_tree 'test' =>
  [ 'An empty file',
    ['A file with contents', 'Contents of file'],
    { 'Subdirectory' => ['Empty file in subdirectory',
                         ['File in subdirectory', 'Contents of file'] ] },
    { 'Empty subdirectory' => [] }
]
require 'find'
Find.find('test') { |f| puts f }
# test
# test/Empty subdirectory
# test/Subdirectory
# test/Subdirectory/File in subdirectory
# test/Subdirectory/Empty file in subdirectory
```


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

```
# test/A file with contents
# test/An empty file

File.read('test/Subdirectory/File in subdirectory')
# => "Contents of file"
```

# Recipe 6.1. Checking to See If a File Exists

## Problem

Given a filename, you want to see whether the corresponding file exists and is the right kind for your purposes.

## Solution

Most of the time you'll use the `File.file?` predicate, which returns true only if the file is an existing regular file (that is, not a directory, a socket, or some other special file).

```
filename = 'a_file.txt'
File.file? filename                          # => false

require 'fileutils'
FileUtils.touch(filename)
File.file? filename                          # => true
```

Use the `File.exists?` predicate instead if the file might legitimately be a directory or other special file, or if you plan to create a file by that name if it doesn't exist. `File.exists?` will return true if a file of the given name exists, no matter what kind of file it is.

```
directory_name = 'a_directory'
FileUtils.mkdir(directory_name)
File.file? directory_name           # => false
File.exists? directory_name         # => true
```

## Discussion

A true response from `File.exists?` means that the file is present on the filesystem, but says nothing about what type of file it is. If you open up a directory thinking it's a regular file, you're in for an unpleasant surprise. This is why `File.file?` is usually more useful than `File.exists?`.

Ruby provides several other predicates for checking the type of a file: the other commonly useful one is `File.directory?`:

```
File.directory? directory_name         # => true
File.directory? filename               # => false
```


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

The rest of the predicates are designed to work on Unix systems. `File.blockdev?` tests or block-device files (such as hard-drive partitions), `File.chardev?` tests for character-device files (such as TTYs), `File.socket?` tests for socket files, and `File.pipe?` tests for named pipes,

```
File.blockdev? '/dev/hda1'                # => true
File.chardev? '/dev/tty1'                 # => true
File.socket? '/var/run/mysqld/mysqld.sock' # => true
system('mkfifo named_pipe')
File.pipe? 'named_pipe'                   # => true
```

`File.symlink?` tests whether a file is a symbolic link to another file, but you only need to use it when you want to treat symlinks differently from other files. A symlink to a regular file will satisfy `File.file?`, and can be opened and used just like a regular file. In most cases, you don't even have to know it's a symlink. The same goes for symlinks to directories and to other types of files.

```
new_filename = "#{filename}2"
File.symlink(filename, new_filename)

File.symlink? new_filename       # => true
File.file? new_filename          # => true
```

All of Ruby's file predicates return false if the file doesn't exist at all. This means you can test "exists and is a directory" by just testing directory?; it's the same for the other predicates.

## See Also

- Recipe 6.8, "Writing to a Temporary File," and Recipe 6.14, "Backing Up to Versioned Filenames," deal with writing to files that don't currently exist

# Recipe 6.2. Checking Your Access to a File

## Problem

You want to see what you can do with a file: whether you have read, write, or (on Unix systems) execute permission on it.

## Solution

Use the class methods `File.readable?`, `File.writeable?`, and `File.executable?`.

```
File.readable?('/bin/ls')      # => true
```


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

```
File.readable?('/etc/passwd-') # => false

filename = 'test_file'
File.open(filename, 'w') {}

File.writable?(filename) # => true
File.writable?('/bin/ls') # => false

File.executable?('/bin/ls') # => true
File.executable?(filename) # => false
```

## Discussion

Ruby's file permission tests are Unix-centric, but `readable?` and `writable?` work on any platform; the rest fail gracefully when the OS doesn't support them. For instance, Windows doesn't have the Unix notion of execute permission, so `File.executable?` always returns `true` on Windows.

The return value of a Unix permission test depends in part on whether your user owns the file in question, or whether you belong to the Unix group that owns it. Ruby provides convenience tests `File.owned?` and `File.grpowned?` to check this.

```
File.owned? 'test_file'       # => true
File.grpowned? 'test_file'    # => true
File.owned? '/bin/ls'         # => false
```

On Windows, `File.owned?` always returns true (even for a file that belongs to another user) and `File.grpowned?` always returns false.

The `File` methods described above should be enough to answer most permission questions about a file, but you can also see a file's Unix permissions in their native form by looking at the file's *mode*. The mode is a number, each bit of which has a different meaning within the Unix permission system.[1] You can view a file's mode with `File::Lstat#mode`.

[1] If you're not familiar with this, Recipe 6.3 describes the significance of the permission bits in a file's mode.

The result of `mode` contains some extra bits describing things like the type of a file. You probably want to strip that information out by masking those bits. This example demonstrates that the file originally created in the solution has a Unix permission mask of `0644`:

```
File.lstat('test_file').mode & 0777           # Keep only the permission bits.
# => 420                                       # That is, 0644 octal.
```


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

## Setuid and setgid scripts

`readable?`, `writable?`, and `executable?` return answers that depend on the effective user and group ID you are using to run the Ruby interpreter. This may not be your actual user or group ID: the Ruby interpreter might be running setuid or setgid, or you might have changed their effective ID with `Process.euid=` or `Process.egid=`.

Each of the permission checks has a corresponding method that returns answers from the perspective of the process's real user and real group IDs: `executable_real?`, `readable_real?`, and `writable_real?`. If you're running the Ruby interpreter setuid, then `readable_real?` (for instance) will give different answers from readable?. You can use this to disallow users from reading or modifying certain files unless they actually are the root user, not just taking on the root users' privileges through setuid.

For instance, consider the following code, which prints our real and effective user and group IDs, then checks to see what it can do to a system file:

```
def what_can_i_do?
  sys = Process::Sys
  puts "UID=#{sys.getuid}, GID=#{sys.getgid}"
  puts "Effective UID=#{sys.geteuid}, Effective GID=#{sys.getegid}"

  file = '/bin/ls'
  can_do = [:readable?, :writable?, :executable?].inject([]) do |arr, method|
    arr << method if File.send(method, file); arr
  end
  puts "To you, #{file} is: #{can_do.join(', ')}"
end
```

If you run this code as root, you can call this method and get one set of answers, then take on the guise of a less privileged user and get another set of answers:

```
what_can_i_do?
# UID=0, GID=0
# Effective UID=0, Effective GID=0
# To you, /bin/ls is: readable?, writable?, executable?

Process.uid = 1000
what_can_i_do?
# UID=0, GID=0
# Effective UID=1000, Effective GID=0
# To you, /bin/ls is: readable?, executable?
```

## See Also

- Recipe 6.3, "Changing the Permissions on a File"
- Recipe 23.3, "Running Code as Another User," has more on setting the effective user ID


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

# Recipe 6.3. Changing the Permissions on a File

## Problem

You want to control access to a file by modifying its Unix permissions. For instance, you want to make it so that everyone on your system can read a file, but only you can write to it.

## Solution

Unless you've got a lot of Unix experience, it's hard to remember the numeric codes for the nine Unix permission bits. Probably the first thing you should do is define constants for them. Here's one constant for every one of the permission bits. If these names are too concise for you, you can name them `USER_READ, GROUP_WRITE, OTHER_ EXECUTE`, and so on.

```
class File
  U_R = 0400
  U_W = 0200
  U_X = 0100
  G_R = 0040
  G_W = 0020
  G_X = 0010
  O_R = 0004
  O_W = 0002
  O_X = 0001
end
```

You might also want to define these three special constants, which you can use to set the user, group, and world permissions all at once:

```
class File
  A_R = 0444
  A_W = 0222
  A_X = 0111
  end
```

Now you're ready to actually change a file's permissions. Every Unix file has a permission bitmap, or *mode*, which you can change (assuming you have the permissions!) by calling `File.chmod`. You can manipulate the constants defined above to get a new mode, then pass it in along with the filename to `File.chmod`.

The following three `chmod` calls are equivalent: for the file `my_file`, they give readwrite access to to the user who owns the file, and restrict everyone else to read-only access. This is equivalent to the permission bitmap 11001001, the octal number 0644, or the decimal number 420.

```
open("my_file", "w") {}
```


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

```
File.chmod(File::U_R | File::U_W | File::G_R | File::O_R, "my_file")
File.chmod(File::A_R | File::U_W, "my_file")
File.chmod(0644, "my_file")                      # Bitmap: 110001001

File::U_R | File::U_W | File::G_R | File::O_R    # => 420
File::A_R | File::U_W                            # => 420
0644 # => 420
File.lstat("my_file").mode & 0777                # => 420
```

Note how I build a full permission bitmap by combining the permission constants with the `OR` operator (|).

## Discussion

A Unix file has nine associated permission bits that are consulted whenever anyone tries to access the file. They're divided into three sets of three bits. There's one set for the user who owns the file, one set is for the user group who owns the file, and one set is for everyone else.

Each set contains one bit for each of the three basic things you might do to a file in Unix: read it, write it, or execute it as a program. If the appropriate bit is set for you, you can carry out the operation; if not, you're denied access.

When you put these nine bits side by side into a bitmap, they form a number that you can pass into `File.chmod`. These numbers are difficult to construct and read without a lot of practice, which is why I recommend you use the constants defined above. It'll make your code less buggy and more readable.[2]

[2] It's true that it's more macho to use the numbers, but if you really wanted to be macho you'd be writing a shell script, not a Ruby program.

`File.chmod` completely overwrites the file's current permission bitmap with a new one. Usually you just want to change one or two permissions: make sure the file isn't world-writable, for instance. The simplest way to do this is to use `File.lstat#mode` to get the file's current permission bitmap, then modify it with bit operators to add or remove permissions. You can pass the result into `File.chmod`.

Use the XOR operator (^) to remove permissions from a bitmap, and the OR operator, as seen above, to add permissions:

```
# Take away the world's read access.
new_permission = File.lstat("my_file").mode ^ File::O_R
File.chmod(new_permission, "my_file")

File.lstat("my_file").mode & 0777        # => 416 # 0640 octal

# Give everyone access to everything
new_permission = File.lstat("my_file").mode | File::A_R | File::A_W | File::A_X
File.chmod(new_permission, "my_file")

File.lstat("my_file").mode & 0777        # => 511 # 0777 octal

# Take away the world's write and execute access
```


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

```
new_permission = File.lstat("my_file").mode ^ (File::O_W | File::O_X)
File.chmod(new_permission, "my_file")

File.lstat("my_file").mode & 0777        # => 508 # 0774 octal
```

If doing bitwise math with the permission constants is also too complicated for you, you can use code like this to parse a permission string like the one accepted by the Unix `chmod` command:

```
class File
  def File.fancy_chmod(permission_string, file)
  mode = File.lstat(file).mode
    permission_string.scan(/[ugoa][+-=][rwx]+/) do |setting|
    who = setting[0..0]
    setting[2..setting.size].each_byte do |perm|
      perm = perm.chr.upcase
      mask = eval("File::#{who.upcase}_#{perm}")
    (setting[1] == ?+) ? mode |= mask : mode ^= mask
    end
  end
  File.chmod(mode, file)
end
end
# Give the owning user write access
File.fancy_chmod("u+w", "my_file")

File.lstat("my_file").mode & 0777                # => 508 # 0774 octal

# Take away the owning group's execute access
File.fancy_chmod("g-x", "my_file")

File.lstat("my_file").mode & 0777                # => 500 # 0764 octal
# Give everyone access to everything

File.fancy_chmod("a+rwx", "my_file")

File.lstat("my_file").mode & 0777                # => 511 # 0777 octal

# Give the owning user access to everything. Then take away the
# execute access for users who aren't the owning user and aren't in
# the owning group.
File.fancy_chmod("u+rwxo-x", "my_file")
File.lstat("my_file").mode & 0777                # => 510 # 0774 octal
```

Unix-like systems such as Linux and Mac OS X support the full range of Unix permissions. On Windows systems, the only one of these operations that makes sense is adding or subtracting the `U_W` bit of a file—making a file read-only or not. You can use `File.chmod` on Windows, but the only bit you'll be able to change is the user write bit.

## See Also

- Recipe 6.2, "Checking Your Access to a File"
- Recipe 23.9, "Normalizing Ownership and Permissions in User Directories"

# Recipe 6.4. Seeing When a File Was Last Used Problem


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

## Problem

You want to see when a file was last accessed or modified.

## Solution

The result of `File.stat` contains a treasure trove of metadata about a file. Perhaps the most useful of its methods are the two time methods `mtime` (the last time anyone wrote to the file), and `atime` (the last time anyone read from the file).

```
open("output", "w") { |f| f << "Here's some output.\n" }
stat = File.stat("output")
stat.mtime                              # => Thu Mar 23 12:23:54 EST 2006
stat.atime                              # => Thu Mar 23 12:23:54 EST 2006

sleep(2)
open("output", "a") { |f| f << "Here's some more output.\n" }
stat = File.stat("output")
stat.mtime                              # => Thu Mar 23 12:23:56 EST 2006
stat.atime                              # => Thu Mar 23 12:23:54 EST 2006

sleep(2)
open("output") { |f| contents = f.read }
stat = File.stat("output")
stat.mtime                              # => Thu Mar 23 12:23:56 EST 2006
stat.atime                              # => Thu Mar 23 12:23:58 EST 2006
```

## Discussion

A file's `atime` changes whenever data is read from the file, and its `mtime` changes whenever data is written to the file.

There's also a `ctime` method, but it's not as useful as the other two. Contrary to semi-popular belief, `ctime` does not track the creation time of the file (there's no way to track this in Unix). A file's `ctime` is basically a more inclusive version of its `mtime`. The `ctime` changes not only when someone modifies the contents of a file, but when someone changes its permissions or its other metadata.

All three methods are useful for separating the files that actually get used from the ones that just sit there on disk. They can also be used in sanity checks.

Here's code for the part of a game that saves and loads the game state to a file. As a deterrent against cheating, when the game loads a save file it performs a simple check against the file's modification time. If it differs from the timestamp recorded inside the file, the game refuses to load the save file.

The `save_game` method is responsible for recording the timestamp:


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

```
def save_game(file)
  score = 1000
  open(file, "w") do |f|
    f.puts(score)
    f.puts(Time.new.to_i)
  end
end
```

The `load_game` method is responsible for comparing the timestamp within the file to the time the filesystem has associated with the file:

```
def load_game(file)
  open(file) do |f|
    score = f.readline.to_i
    time = Time.at(f.readline.to_i)
    difference = (File.stat(file).mtime - time).abs
    raise "I suspect you of cheating." if difference > 1
    "Your saved score is #{score}."
  end
end
```

This mechanism can detect simple forms of cheating:

```
save_game("game.sav")
sleep(2)
load_game("game.sav")
# => "Your saved score is 1000."

# Now let's cheat by increasing our score to 9000

open("game.sav", "r+b") { |f| f.write("9") }

load_game("game.sav")
# RuntimeError: I suspect you of cheating.
```

Since it's possible to modify a file's times with tools like the Unix `touch` command, you shouldn't depend on these methods to defend you against a skilled attacker actively trying to fool your program.

## See Also

- An example in Recipe 3.12, "Running a Code Block Periodically," monitors a file for changes by checking its `mtime` periodically
- Recipe 6.20, "Finding the Files You Want," shows examples of filesystem searches that make comparisons between the file times

# Recipe 6.5. Listing a Directory

## Problem

You want to list or process the files or subdirectories within a directory.


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

## Solution

If you're starting from a directory name, you can use `Dir.entries` to get an array of the items in the directory, or `Dir.foreach` to iterate over the items. Here's an exampleof each run on a sample directory:

```
# See the chapter intro to get the create_tree library
require 'create_tree'
create_tree 'mydir' =>
  [ {'subdirectory' => [['file_in_subdirectory', 'Just a simple file.']] },
  '.hidden_file', 'ruby_script.rb', 'text_file' ]

Dir.entries('mydir')

# => [".", "..", ".hidden_file", "ruby_script.rb", "subdirectory",
# "text_file"]

Dir.foreach('mydir') { |x| puts x if x != "." && x != ".."}
# .hidden_file
# ruby_script.rb
# subdirectory
# text_file
```

You can also use `Dir[]` to pick up all files matching a certain pattern, using a format similar to the bash shell's glob format (and somewhat less similar to the wildcard format used by the Windows command-line shell):

```
# Find all the "regular" files and subdirectories in mydir. This excludes
# hidden files, and the special directories . and ..
Dir["mydir/*"]
# => ["mydir/ruby_script.rb", "mydir/subdirectory", "mydir/text_file"]

# Find all the .rb files in mydir
Dir["mydir/*.rb"]                # =>  ["mydir/ruby_script.rb"]
```

You can also open a directory handle with `Dir#open`, and treat it like any other Enumerable. Methods like `each`,`each_with_index`, `grep`, and `reject` will all work (but see below if you want to call them more than once). As with `File#open`, you should do your directory processing in a code block so that the directory handle will get closed once you're done with it.

```
Dir.open('mydir') { |d| d.grep /file/ }
# => [".hidden_file", "text_file"]

Dir.open('mydir') { |d| d.each { |x| puts x } }
# .
# ..
# .hidden_file
# ruby_script.rb
# subdirectory
# text_file
```


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

## Discussion

Reading entries from a `Dir` object is more like reading data from a file than iterating over an array. If you call one of the `Dir` instance methods and then want to call another one on the same `Dir` object, you'll need to call `Dir#rewind` first to go back to the beginning of the directory listing:

```
#Get all contents other than ".", "..", and hidden files.

d = Dir.open('mydir')
d.reject { |f| f[0] == '.' }
# => ["subdirectory", "ruby_script.rb", "text_file"]
#Now the Dir object is useless until we call Dir#rewind.
d.entries.size                                    # => 0
d.rewind
d.entries.size                                    # => 6

#Get the names of all files in the directory.
d.rewind
d.reject { |f| !File.file? File.join(d.path, f) }
# => [".hidden_file", "ruby_script.rb", "text_file"]

d.close
```

Methods for listing directories and looking for files return string pathnames instead of `File` and `Dir` objects. This is partly for efficiency, and partly because creating a `File` or `Dir` actually opens up a filehandle on that file or directory.

Even so, it's annoying to have to take the output of these methods and patch together real `File` or `Dir` objects on which you can operate. Here's a simple method that will build a `File` or `Dir`, given a filename and the name or `Dir` of the parent directory:

```
def File.from_dir(dir, name)
  dir = dir.path if dir.is_a? Dir
  path = File.join(dir, name)
  (File.directory?(path) ? Dir : File).open(path) { |f| yield f }
end
```

As with `File#open` and `Dir#open`, the actual processing happens within a code block:

```
File.from_dir("mydir", "subdirectory") do |subdir|
 File.from_dir(subdir, "file_in_subdirectory") do |file|
    puts %{My path is #{file.path} and my contents are "#{file.read}".}
  end
end
# My path is mydir/subdirectory/file_in_subdirectory and my contents are
# "Just a simple file".
```

Globs make excellent shortcuts for finding files in a directory or a directory tree. Especially useful is the ** glob, which matches any number of directories. A glob is the easiest and fastest way to recursively process every file in a directory tree, although it loads all the


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

filenames into an array in memory. For a less memoryintensive solution, see the `find` library, described in Recipe 6.12.

```
Dir["mydir/**/*"]
# => ["mydir/ruby_script.rb", "mydir/subdirectory", "mydir/text_file",
# "mydir/subdirectory/file_in_subdirectory"]

Dir["mydir/**/*file*"]
# => ["mydir/text_file", "mydir/subdirectory/file_in_subdirectory"]
```

A brief tour of the other features of globs:

```
#Regex-style character classes
Dir["mydir/[rs]*"] # => ["mydir/ruby_script.rb", "mydir/subdirectory"]
Dir["mydir/[^s]*"] # => ["mydir/ruby_script.rb", "mydir/text_file"]

# Match any of the given strings
Dir["mydir/{text,ruby}*"]  # => ["mydir/text_file", "mydir/ruby_script.rb"]

# Single-character wildcards
Dir["mydir/?ub*"]          # => ["mydir/ruby_script.rb", "mydir/subdirectory"]
```

Globs will not pick up files or directories whose names start with periods, unless you match them explicitly:

```
Dir["mydir/.*"] # => ["mydir/.", "mydir/..", "mydir/.hidden_file"]
```

## See Also

- Recipe 6.12, "Walking a Directory Tree"
- Recipe 6.20, "Finding the Files You Want"

# Recipe 6.6. Reading the Contents of a File

## Problem

You want to read some or all of a file into memory.

## Solution

Open the file with `Kernel#open`, and pass in a code block that does the actual reading. To read the entire file into a single string, use `IO#read`:

```
#Put some stuff into a file.
open('sample_file', 'w') do |f|
  f.write("This is line one.\nThis is line two.")
end

# Then read it back out.
```


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

```
open('sample_file') { |f| f.read }
# => "This is line one.\nThis is line two."
```

To read the file as an array of lines, use `IO#readlines`:

```
open('sample_file') { |f| f.readlines }
# => ["This is line one.\n", "This is line two."]
```

To iterate over each line in the file, use `IO#each`. This technique loads only one line into memory at a time:

```
open('sample_file').each { |x| p x }
# "This is line one.\n"
# "This is line two."
```

## Discussion

How much of the file do you want to read into memory at once? Reading the entire file in one gulp uses memory equal to the size of the file, but you end up with a string, and you can use any of Ruby's string processing techniques on it.

The alternative is to process the file one chunk at a time. This uses only the memory needed to store one chunk, but it can be more difficult to work with, because any given chunk may be incomplete. To process a chunk, you may end up reading the next chunk, and the next. This code reads the first 50-byte chunk from a file, but it turns out not to be enough:

```
puts open('conclusion') { |f| f.read(50) }
# "I know who killed Mr. Lambert," said Joe. "It was
```

If a certain string always marks the end of a chunk, you can pass that string into `IO#each` to get one chunk at a time, as a series of strings. This lets you process each full chunk as a string, and it uses less memory than reading the entire file.

If a certain string always marks the end of a chunk, you can pass that string into `IO#each` to get one chunk at a time, as a series of strings. This lets you process each full chunk as a string, and it uses less memory than reading the entire file.

```
# Create a file…
open('end_separated_records', 'w') do |f|
  f << %{This is record one.
It spans multiple lines.ENDThis is record two.END}
end

# And read it back in.
open('end_separated_records') { |f| f.each('END') { |record| p record } }
# "This is record one.\nIt spans multiple lines.END"
# "This is record two.END"
```


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

You can also pass a delimiter string into `IO#readlines` to get the entire file split into an array by the delimiter string:

```
# Create a file…
open('pipe_separated_records', 'w') do |f|
  f << "This is record one.|This is record two.|This is record three."
end

# And read it back in.
open('pipe_separated_records') { |f| f.readlines('|') }
# => ["This is record one.|", "This is record two.|",
#     "This is record three."]
```

The newline character usually makes a good delimiter (many scripts process a file one line at a time), so by default, `IO#each` and `IO#readlines` split the file by line:

```
open('newline_separated_records', 'w') do |f|
  f.puts 'This is record one. It cannot span multiple lines.'
  f.puts 'This is record two.'
end

open('newline_separated_records') { |f| f.each { |x| p x } }
# "This is record one. It cannot span multiple lines.\n"
# "This is record two.\n"
```

The trouble with newlines is that different operating systems have different newline formats. Unix newlines look like "\n", while Windows newlines look like "\r\n", and the newlines for old (pre-OS X) Macintosh files look like "\r". A file uploaded to a web application might come from any of those systems, but `IO#each` and `IO#readlines` split files into lines depending on the newline character of the OS that's running the Ruby script (this is kept in the special variable $/). What to do?

By passing "\n" into `IO#each` or `IO#readlines`, you can handle the newlines of files created on any recent operating system. If you need to handle all three types of newlines, the easiest way is to read the entire file at once and then split it up with a regular expression.

```
open('file_from_unknown_os') { |f| f.read.split(/\r?\n|\r(?!\n)/) }
```

`IO#each` and `IO#readlines` don't strip the delimiter strings from the end of the lines. Assuming the delimiter strings aren't useful to you, you'll have to strip them manually.

To strip delimiter characters from the end of a line, use the `String#chomp` or `String#chomp`! methods. By default, these methods will remove the last character or set of characters that can be construed as a newline. However, they can be made to strip any other delimiter string from the end of a line.

```
"This line has a Unix/Mac OS X newline.\n".chomp
# => "This line has a Unix/Mac OS X newline."

"This line has a Windows newline.\r\n".chomp
```


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

```
# => "This line has a Windows newline."

"This line has an old-style Macintosh newline.\r".chomp
# => "This line has an old-style Macintosh newline."

"This string contains two newlines.\n\n".chomp
# "This string contains two newlines.\n"

'This is record two.END'.chomp('END')
# => "This is record two."

'This string contains no newline.'.chomp
# => "This string contains no newline."
```

You can `chomp` the delimiters as `IO#each` yields each record, or you can `chomp` each line returned by `IO#readlines`:

```
open('pipe_separated_records') do |f|
  f.each('|') { |l| puts l.chomp('|') }
end
# This is record one.
# This is record two.
# This is record three.

lines = open('pipe_separated_records') { |f| f.readlines('|') }
# => ["This is record one.|", "This is record two.|",
#     "This is record three."]
lines.each { |l| l.chomp!('|') }
# => ["This is record one.", "This is record two.", "This is record three."]
```

You've got a problem if a file is too big to fit into memory, and there are no known delimiters, or if the records between the delimiters are themselves too big to fit in memory. You've got no choice but to read from the file in chunks of a certain number of bytes. This is also the best way to read binary files; see Recipe 6.17 for more.

Use `IO#read` to read a certain number of bytes, or `IO#each_byte` to iterate over the `File` one byte at a time. The following code uses `IO#read` to continuously read uniformly sized chunks until it reaches end-of-file:

```
class File
  def each_chunk(chunk_size=1024)
   yield read(chunk_size) until eof?
  end
end

open("pipe_separated_records") do |f|
  f.each_chunk(15) { |chunk| puts chunk }
end
# This is record
# one.|This is re
# cord two.|This
# is record three
# .
```

All of these methods are made available by the `IO` class, the superclass of `File`. You can use the same methods on `Socket` objects. You can also use `each` and `each_byte` on


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

`String` objects, which in some cases can save you from having to `create` a `StringIO` object (see Recipe 6.15 for more on those beasts).

## See Also

- Recipe 6.11, "Performing Random Access on "Read-Once" Input Streams"
- Recipe 6.17, "Processing a Binary File," goes into more depth about reading files as chunks of bytes
- Recipe 6.15, "Pretending a String Is a File"

# Recipe 6.7. Writing to a File

## Problem

You want to write some text or Ruby data structures to a file. The file might or might not exist. If it does exist, you might want to overwrite the old contents, or just append new data to the end of the file.

## Solution

Open the file in write mode ('w'). The file will be created if it doesn't exist, and truncated to zero bytes if it does exist. You can then use `IO#write` or the << operator to write strings to the file, as though the file itself were a string and you were appending to it.

You can also use `IO#puts` or `IO#p` to write lines to the file, the same way you can use `Kernel#puts` or `Kernel#p` to write lines to standard output.

Both of the following chunks of code destroy the previous contents of the file `output`, then write a new string to the file:

```
open('output', 'w') { |f| f << "This file contains great truths.\n" }
open('output', 'w') do |f|
  f.puts 'The great truths have been overwritten with an advertisement.'
end

open('output') { |f| f.read }
# => "The great truths have been overwritten with an advertisement.\n"
```

To append to a file without overwriting its old contents, open the file in append mode ('a') instead of write mode:

```
open('output', "a") { |f| f.puts 'Buy Ruby(TM) brand soy sauce!' }

open('output') { |f| puts f.read }
# The great truths have been overwritten with an advertisement.
# Buy Ruby(TM) brand soy sauce!
```


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

## Discussion

Sometimes you'll only need to write a single (possibly very large) string to a file. Usually, though, you'll be getting your strings one at a time from a data structure or some other source, and you'll call puts or the append operator within some kind of loop:

```
open('output', 'w') do |f|
  [1,2,3].each { |i| f << i << ' and a ' }
end
open('output') { |f| f.read }     # => "1 and a 2 and a 3 and a "
```

Since the << operator returns the filehandle it wrote to, you can chain calls to it. As seen above, this feature lets you write multiple strings to a file in a single line of Ruby code.

Because opening a file in write mode destroys the file's existing contents, you should only use it when you don't care about the old contents, or after you've read them into memory for later use. Append mode is nondestructive, making it useful for files like log iles, which need to be updated periodically without destroying their old contents.

## Buffered I/O

There's no guarantee that data will be written to your file as soon as you call << or `puts`. Since disk writes are expensive, Ruby lets changes to a file pile up in a buffer. It occasionally flushes the buffer, sending the data to the operating system so it can be ritten to disk.

You can manually flush Ruby's buffer for a particular file by calling its `IO#flush` method. You can turn off Ruby's buffering altogether by setting `IO.sync` to false. However, your operating system probably does some disk buffering of its own, so doing these things won't neccessarily write your changes directly to disk.

```
open('output', 'w') do |f|
  f << 'This is going into the Ruby buffer.'
  f.flush # Now it's going into the OS buffer.
end

IO.sync = false
open('output', 'w') { |f| f << 'This is going straight into the OS buffer.' }
```

## See Also

- Recipe 1.1, "Building a String from Parts"
- Recipe 6.6, "Reading the Contents of a File"
- Recipe 6.19, "Truncating a File"


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

# Recipe 6.8. Writing to a Temporary File

## Problem

You want to write data to a secure temporary file with a unique name.

## Solution

Create a `Tempfile object`. It has all the methods of a `File` object, and it will be in a location on disk guaranteed to be unique.

```
require 'tempfile'
out = Tempfile.new("tempfile")
out.path # =>              "/tmp/tempfile23786.0"
```

A `Tempfile` object is opened for read-write access (mode w+), so you can write to it and then read from it without having to close and reopen it:

```
out << "Some text."
out.rewind
out.read # => "Some text."
out.close
```

Note that you can't pass a code block into the `Tempfile` constructor: you have to assign the temp file to an object, and call `Tempfile#close` when you're done.

## Discussion

To avoid security problems, use the `Tempfile` class to generate temp file names, instead of writing the code yourself. The `Tempfile` class creates a file on disk guaranteed not to be in use by any other thread or process, and sets that file's permissions so that only you can read or write to it. This eliminates any possibility that a hostile process might inject fake data into the temp file, or read what you write.[3]

[3] Unless the hostile process is running as you or as the root user, but then you've got bigger problems.

The name of a temporary file incorporates the string you pass into the `Tempfile` constructor, the process ID of the current process ($$, or `$PID` if you've done an `include English`), and a unique number. By default, temporary files are created in `Dir::tmpdir` (usually /tmp), but you can pass in a different directory name:

```
out = Tempfile.new("myhome_tempfile", "/home/leonardr/temp/")
```

No matter where you create your temporary files, when your process exits, all of its temporary files are automatically destroyed. If you want the data you wrote to temporary


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

files to live longer than your process, you should copy or move the temporary files to "real" files:

```
require 'fileutils'
FileUtils.mv(out.path, "/home/leonardr/old_tempfile")
```

The `tempfile` assumes that the operating system can atomically open a file and get an exclusive lock on it. This doesn't work on all filesystems. Ara Howard's `lockfile` library (available as a gem of the same name) uses linking, which is atomic everywhere.

# Recipe 6.9. Picking a Random Line from a File

## Problem

You want to choose a random line from a file, without loading the entire file into memory.

## Solution

Iterate over the file, giving each line a chance to be the randomly selected one:

```
module Enumerable
  def random_line
    selected = nil
    each_with_index { |line, lineno| selected = line if rand < 1.0/lineno }
    return selected.chomp if selected
  end
end

#Create a file with 1000 lines
open('random_line_test', 'w') do |f|
  1000.times { |i| f.puts "Line #{i}" }
end

#Pick random lines from the file.
f = open('random_line_test')
f.random_line # => "Line 520"
f.random_line # => nil
f.rewind
f.random_line # => "Line 727"
```

## Discussion

The obvious solution reads the entire file into memory;

```
File.open('random_line_test') do |f|
l = f.readlines
l[rand(l.size)].chomp
end
# => "Line 708"
```

The recommended solution is just as fast, and only reads one line at a time into memory. However, once it's done, the file pointer has been set to the end of the file and you can't


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

access the file anymore without calling `File#rewind`. If you want to pick a lot of random lines from a file, reading the entire file into memory might be preferable to iterating over it multiple times.

This recipe makes for a good command-line tool. The following code uses the special variable $., which holds the number of the line most recently read from a file:

```
$ ruby -e 'rand < 1.0/$. and line = $_ while gets; puts line.chomp if line'
```

The algorithm works because, although lines that come earlier in the file have a better chance of being selected initially, they also have more chances to be replaced by a later line. A proof by induction demonstrates that the algorithm gives equal weight to each line in the file.

The base case is a file of a single line, where it will obviously work: any value of `Kernel#rand` will be less than 1, so the first line will always be chosen.

Now for the inductive step. Assume that the algorithm works for a file of *n* lines: that is, each of the first *n* lines has a *1/n* chance of being chosen. Then, add another line to the file and process the new line. The chance that line *n+1* will become the randomly chosen line is *1/(n+1)*. The remaining probability, *n/n+1*, is the chance that one of the other *n* lines is the randomly chosen one.

Our inductive assumption was that each of the *n* original lines had an equal chance of being chosen, so this remaining *n/n+1* probability must be distributed evenly across the *n* original lines. Given a line in the first *n*, what's it's chance of being the chosen one? It's just *n/n+1* divided by *n*,or *1/n+1*. Line *n+1* and all earlier lines have a *1/n+1* chance of being chosen, so the choice is truly random.

## See Also

- Recipe 2.5, "Generating Random Numbers"
- Recipe 4.10, "Shuffling an Array"
- Recipe 5.11, "Choosing Randomly from a Weighted List"

# Recipe 6.10. Comparing Two Files

## Problem

You want to see if two files contain the same data. If they differ, you might want to represent the differences between them as a string: a patch from one to the other.


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

## Solution

If two files differ, it's likely that their sizes also differ, so you can often solve the problem quickly by comparing sizes. If both files are regular files with the same size, you'll need to look at their contents.

This code does the cheap checks first:

1. If one file exists and the other does not, they're not the same.
2. If neither file exists, say they're the same.
3. If the files are the same file, they're the same.
4. If the files are of different types or sizes, they're not the same.

```
class File
  def File.same_contents(p1, p2)
    return false if File.exists?(p1) != File.exists?(p2)
    return true if !File.exists?(p1)
    return true if File.expand_path(p1) == File.expand_path(p2)
    return false if File.ftype(p1) != File.ftype(p2) ||
       File.size(p1) != File.size(p2)
```

Otherwise, it compares the files contents, a block at a time:

```
    open(p1) do |f1|
      open(p2) do |f2|
        blocksize = f1.lstat.blksize
        same = true
        while same && !f1.eof? && !f2.eof?
          same = f1.read(blocksize) == f2.read(blocksize)
        end
        return same
      end
    end
  end
end
```

To illustrate, I'll create two identical files and compare them. I'll then make them slightly different, and compare them again.

```
1.upto(2) do |i|
  open("output#{i}", 'w') { |f| f << 'x' * 10000 }
end
File.same_contents('output1', 'output2')           # => true
open("output1", 'a') { |f| f << 'x' }
open("output2", 'a') { |f| f << 'y' }
File.same_contents('output1', 'output2')           # => false

File.same_contents('nosuchfile', 'output1')        # => false
File.same_contents('nosuchfile1', 'nosuchfile2')   # => true
```

## Discussion

The code in the Solution works well if you only need to determine whether two files are identical. If you need to see the differences between two files, the most useful tool is is


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

Austin Ziegler's `Diff::LCS` library, available as the `diff-lcs` gem. It implements a sophisticated diff algorithm that can find the differences between any two enumerable objects, not just strings. You can use its `LCS` module to represent the differences between two nested arrays, or other complex data structures.

The downside of such flexibility is a poor interface when you just want to diff two files or strings. A diff is represented by an array of `Change` objects, and though you can traverse this array in helpful ways, there's no simple way to just turn it into a string representation of the sort you might get by running the Unix command `diff`.

Fortunately, the `lcs-diff` gem comes with command-line diff programs `ldiff` and `htmldiff`. If you need to perform a textual diff from within Ruby code, you can do one of the following:

1. Call out to one of those programs: assuming the gem is installed, this is more portable than relying on the Unix `diff` command.
2. Import the program's underlying library, and fake a command-line call to it. You'll have to modify your own program's `ARGV`, at least temporarily.
3. Write Ruby code that copies one of the underlying implementations to do what you want.

Here's some code, adapted from the `ldiff` command-line program, which builds a string representation of the differences between two strings. The result is something you might see by running `ldiff`, or the Unix command `diff`. The most common diff formats are `:unified` and `:context`.

```
require 'rubygems'
require 'diff/lcs/hunk'

def diff_as_string(data_old, data_new, format=:unified, context_lines=3)
```

First we massage the data into shape for the `diff` algorithm:

```
data_old = data_old.split(/\n/).map! { |e| e.chomp }
data_new = data_new.split(/\n/).map! { |e| e.chomp }
```

Then we perform the `diff`, and transform each "hunk" of it into a string:

```
  output = ""
  diffs = Diff::LCS.diff(data_old, data_new)
  return output if diffs.empty?
  oldhunk = hunk = nil
  file_length_difference = 0
  diffs.each do |piece|
    begin
      hunk = Diff::LCS::Hunk.new(data_old, data_new, piece, context_lines,
                                 file_length_difference)
      file_length_difference = hunk.file_length_difference
```


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

```
      next unless oldhunk

      # Hunks may overlap, which is why we need to be careful when our
      # diff includes lines of context. Otherwise, we might print
      # redundant lines.
      if (context_lines > 0) and hunk.overlaps?(oldhunk)
         hunk.unshift(oldhunk)
      else
        output << oldhunk.diff(format)
      end
    ensure
      oldhunk = hunk
      output << "\n"
    end
  end

  #Handle the last remaining hunk
  output << oldhunk.diff(format) << "\n"
end
```

Here it is in action:

```
s1 = "This is line one.\nThis is line two.\nThis is line three.\n"
s2 = "This is line 1.\nThis is line two.\nThis is line three.\n" +
     "This is line 4.\n"
puts diff_as_string(s1, s2)
# @@ -1,4 +1,5 @@
# -This is line one.
# +This is line 1.
# This is line two.
# This is line three.
# +This is line 4.
```

With all that code, on a Unix system you could be forgiven for just calling out to the Unix `diff` program:

```
open('old_file', 'w') { |f| f << s1 }
open('new_file', 'w') { |f| f << s2 }

puts %x{diff old_file new_file}
# 1c1
# < This is line one.
# ---
# > This is line 1.
# 3a4
# > This is line 4.
```

## See Also

- The `algorithm-diff` gem is another implementation of a general diff algorithm; its API is a little simpler than `diff-lcs`, but it has the same basic structure; both gems are descended from Perl's `Algorithm::Diff` module
- It's not available as a gem, but the `diff.rb` package is a little easier to script from Ruby if you need to create a textual diff of two files; look at how the `unixdiff.rb` program creates a `Diff` object and manipulates it (http://users.cybercity.dk/~dsl8950/ruby/diff.html)


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

- The MD5 checksum is often used in file comparisons: I didn't use it in this recipe because when you're only comparing two files, it's faster to compare their contents; in Recipe 23.7, "Finding Duplicate Files," though, the MD5 checksum is used as a convenient shorthand for the contents of many files

# Recipe 6.11. Performing Random Access on "Read-Once" Input Streams

## Problem

You have an `IO` object, probably a socket, that doesn't support random-access methods like `seek`, `pos=`, and `rewind`. You want to treat this object like a file on disk, where you can jump around and reread parts of the file.

## Solution

The simplest solution is to read the entire contents of the socket (or as much as you're going to need) and put it into a `StringIO` object. You can then treat the `StringIO` object exactly like a file:

```
require 'socket'
require 'stringio'

sock = TCPSocket.open("www.example.com", 80)
sock.write("GET /\n")

file = StringIO.new(sock.read)
file.read(10)                                 # => "<HTML>\r\n<H"
file.rewind
file.read(10)                                 # => "<HTML>\r\n<H"
file.pos = 90
file.read(15)                                 # => " this web page "
```

## Discussion

A socket is supposed to work just like a file, but sometimes the illusion breaks down. Since the data is coming from another computer over which you have no control, you can't just go back and reread data you've already read. That data has already been sent over the pipe, and the server doesn't care if you lost it or need to process it again.

If you have enough memory to read the entire contents of a socket, it's easy to put the results into a form that more closely simulates a file on disk. But you might not want to read the entire socket, or the socket may be one that keeps sending data until you close it. In that case you'll need to buffer the data as you read it. Instead of using memory for the entire contents of the socket (which may be infinite), you'll only use memory for the data you've actually read.


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

This code defines a `BufferedIO` class that adds data to an internal `StringIO` as it's read from its source:

```
class BufferedIO
  def initialize(io)
    @buff = StringIO.new
    @source = io
    @pos = 0
  end

  def read(x=nil)
    to_read = x ? to_read = x+@buff.pos-@buff.size : nil
    _append(@source.read(to_read)) if !to_read or to_read > 0
    @buff.read(x)
  end

  def pos=(x)
   read(x-@buff.pos) if x > @buff.size
    @buff.pos = x
  end

  def seek(x, whence=IO::SEEK_SET)
    case whence
      when IO::SEEK_SET then self.pos=(x)
      when IO::SEEK_CUR then self.pos=(@buff.pos+x)
      when IO::SEEK_END then read; self.pos=(@buff.size-x)
      # Note: SEEK END reads all the socket data.
    end
    pos
  end

  # Some methods can simply be delegated to the buffer.
  ["pos", "rewind", "tell"].each do |m|
  module_eval "def #{m}\n@buff.#{m}\nend"
  end

  private

    def _append(s)
  @buff << s
    @buff.pos -= s.size
  end
end
```

Now you can `seek`, `rewind`, and generally move around in an input socket as if it were a disk file. You only have to read as much data as you need:

```
sock = TCPSocket.open("www.example.com", 80)
sock.write("GET /\n")
file = BufferedIO.new(sock)

file.read(10)                               # => "<HTML>\r\n<H"
file.rewind                                 # => 0
file.read(10)                               # => "<HTML>\r\n<H"
file.pos = 90                               # => 90
file.read(15)                               # => " this web page "
file.seek(-10, IO::SEEK_CUR)                # => 95
file.read(10)                               # => " web page "
```

`BufferedIO` doesn't implement all the methods of `IO`, only the ones not implemented by socket-type `IO` objects. If you need the other methods, you should be able to implement


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

the ones you need using the existing methods as guidelines. For instance, you could implement `readline` like this:

```
class BufferedIO
  def readline
    oldpos = @buff.pos
    line = @buff.readline unless @buff.eof?
    if !line or line[-1] != ?\n
       _append(@source.readline) # Finish the line
       @buff.pos = oldpos # Go back to where we were
       line = @buff.readline # Read the line again
    end
    line
  end
end

file.readline               # => "by typing "example.com",\r\n"
```

## See Also

- Recipe 6.17, "Processing a Binary File," for more information on `IO#seek`

# Recipe 6.12. Walking a Directory Tree

## Problem

You want to recursively process every subdirectory and file within a certain directory.

## Solution

Suppose that the directory tree you want to walk looks like this (see this chapter's introduction section for the `create_tree` library that can build this directory tree automatically):

```
require 'create_tree'
create_tree './' =>
  [ 'file1',
    'file2',
         { 'subdir1/' => [ 'file1' ] },
         { 'subdir2/' => [ 'file1',
                     'file2',
                      { 'subsubdir/' => [ 'file1' ] }
                    ]
    }
   ]
```

The simplest solution is to load all the files and directories into memory with a big recursive file glob, and iterate over the resulting array. This uses a lot of memory because all the filenames are loaded into memory at once:

```
Dir['**/**']
# => ["file1", "file2", "subdir1", "subdir2", "subdir1/file1",
```


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

```
#     "subdir2/file1", "subdir2/file2", "subdir2/subsubdir",
#     "subdir2/subsubdir/file1"]
```

A more elegant solution is to use the `find` method in the `Find` module. It performs a depth-first traversal of a directory tree, and calls the given code block on each directory and file. The code block should take as an argument the full path to a directory or file.

This snippet calls `Find.find` with a code block that simply prints out each path it receives. This demonstrates how Ruby performs the traversal:

```
require 'find'
Find.find('./') { |path| puts path }
# ./
# ./subdir2
# ./subdir2/subsubdir
# ./subdir2/subsubdir/file1
# ./subdir2/file2
# ./subdir2/file1
# ./subdir1
# ./subdir1/file1
# ./file2
# ./file1
```

## Discussion

Even if you're not a system administrator, the demands of keeping your own files organized will frequently call for you to process every file in a directory tree. You may want to backup, modify, or delete each file in the directory structure, or you may just want to see what's there.

Normally you'll want to at least look at every file in the tree, but sometimes you'll want to skip certain directories. For instance, you might know that a certain directory is full of a lot of large files you don't want to process. When your block is passed a path to a directory, you can prevent `Find.find` from recursing into a directory by calling `Find.prune`. In this example, I'll prevent `Find.find` from processing the files in the `subdir2` directory.

```
Find.find('./') do |path|
  Find.prune if File.basename(path) == 'subdir2'
  puts path
end
# ./
# ./subdir1
# ./subdir1/file1
# ./file2
# ./file1
```

Calling `Find.prune` when your block has been passed a file will only prevent `Find.find` from processing that one file. It won't halt the processing of the rest of the files in that directory:

```
Find.find('./') do |path|
  if File.basename(path) =~ /file2$/
```


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

```
    puts "PRUNED #{path}"
    Find.prune
  end
  puts path
end
# ./
# ./subdir2
# ./subdir2/subsubdir
# ./subdir2/subsubdir/file1
# PRUNED ./subdir2/file2
# ./subdir2/file1
# ./subdir1
# ./subdir1/file1
# PRUNED ./file2
# ./file1
```

`Find.find` works by keeping a queue of files to process. When it finds a directory, it inserts that directory's files at the beginning of the queue. This gives it the characteristics of a depth-first traversal. Note how all the files in the top-level directory are processed after the subdirectories. The alternative would be a breadth-first traversal, which would process the files in a directory before even touching the subdirectories.

If you want to do a breadth-first traversal instead of a depth-first one, the simplest solution is to use a glob and sort the resulting array. Pathnames sort naturally in a way that simulates a breadth-first traversal:

```
Dir["**/**"].sort.each { |x| puts x }
# file1
# file2
# subdir1
# subdir1/file1
# subdir2
# subdir2/file1
# subdir2/file2
# subdir2/subsubdir
# subdir2/subsubdir/file1
```

## See Also

- Recipe 6.20, "Finding the Files You Want"
- Recipe 23.7, "Finding Duplicate Files"

# Recipe 6.13. Locking a File

## Problem

You want to prevent other threads or processes from modifying a file that you're working on.


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

## Solution

Open the file, then lock it with `File#flock`. There are two kinds of lock; pass in the `File` constant for the kind you want.

- `File::LOCK_EX` gives you an exclusive lock, or write lock. If your thread has an exclusive lock on a file, no other thread or process can get a lock on that file. Use this when you want to write to a file without anyone else being able to write to it.
- `File::LOCK_SH` will give you a shared lock, or read lock. Other threads and processes can get their own shared locks on the file, but no one can get an exclusive lock. Use this when you want to read a file and know that it won't change while you're reading it.

Once you're done using the file, you need to unlock it. Call `File#flock` again, and pass in `File::LOCK_UN` as the lock type. You can skip this step if you're running on Windows.

The best way to handle all this is to enclose the locking and unlocking in a method that takes a block, the way `open` does:

```
  def flock(file, mode)
    success = file.flock(mode)
    if success
      begin
        yield file
      ensure
        file.flock(File::LOCK_UN)
    end
  end
  return success
end
```

This makes it possible to lock a file without having to worry about unlocking it later. Even if your block raises an exception, the file will be unlocked and another thread can use it.

```
open('output', 'w') do |f|
  flock(f, File::LOCK_EX) do |f|
  f << "Kiss me, I've got a write lock on a file!"
  end
end
```

## Discussion

Different operating systems support different ways of locking files. Ruby's `flock` implementation tries to hide the differences behind a common interface that looks like Unix's file locking interface. In general, you can use `flock` as though you were on Unix, and your scripts will work across platforms.

On Unix, both exclusive and shared locks work only if all threads and processes play by the rules. If one thread has an exclusive lock on a file, another thread can still open the file


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

without locking it and wreak havoc by overwriting its contents. That's why it's important to get a lock on any file that might conceivably be used by another thread or another process on the system.

Ruby's block-oriented coding style makes it easy to do the right thing with locking. The following shortcut method works with the `flock` method previously defined. It takes care of opening, locking, unlocking, and closing a file, letting you focus on whatever you want to do with the file's contents.

```
def open_lock(filename, openmode="r", lockmode=nil)
  if openmode == 'r' || openmode == 'rb'
    lockmode ||= File::LOCK_SH
  else
    lockmode ||= File::LOCK_EX
  end
  value = nil
  open(filename, openmode) do |f|
    flock(f, lockmode) do
      begin
        value = yield f
      ensure
        f.flock(File::LOCK_UN) # Comment this line out on Windows.
      end
    end
    return value
  end
end
```

This code creates two threads, each of which want to access the same file. Thanks to locks, we can guarantee that only one thread is accessing the file at a time (see Chapter 20 if you're not comfortable with threads).

```
t1 = Thread.new do
  puts 'Thread 1 is requesting a lock.'
  open_lock('output', 'w') do |f|
    puts 'Thread 1 has acquired a lock.'
    f << "At last we're alone!"
    sleep(5)
  end

  puts 'Thread 1 has released its lock.'
end

t2 = Thread.new do
  puts 'Thread 2 is requesting a lock.'
  open_lock('output', 'r') do |f|
    puts 'Thread 2 has acquired a lock.'
    puts "File contents: #{f.read}"
  end
  puts 'Thread 2 has released its lock.'
  end
  t1.join
  t2.join
  # Thread 1 is requesting a lock.
  # Thread 1 has acquired a lock.
  # Thread 2 is requesting a lock.
  # Thread 1 has released its lock.
  # Thread 2 has acquired a lock.
  # File contents: At last we're alone!
  # Thread 2 has released its lock.
```


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

## Nonblocking locks

If you try to get an exclusive or shared lock on a file, your thread will block until Ruby can lock the file. But you might be left waiting a long time, perhaps forever. The code that has the file locked may be buggy and in an infinite loop; or it may itself be blocking, waiting to lock a file that *you* have locked.

You can avoid deadlock and similar problems by asking for a nonblocking lock. When you do, if Ruby can't lock the file for you, `File#flock` returns false, rather than waiting (possibly forever) for another thread or process to release its lock. If you don't get a lock, you can wait a while and try again, or you can raise an exception and let the user deal with it.

To make a lock into a nonblocking lock, use the OR operator (|) to combine `File::LOCK_NB` with either `File::LOCK_EX` or `File::LOCK_SH`.

The following code will print "I've got a lock!" if it can get an exclusive lock on the file "output"; otherwise it will print "I couldn't get a lock." and continue:

```
def try_lock
  puts "I couldn't get a lock." unless
    open_lock('contested', 'w', File::LOCK_EX | File::LOCK_NB) do
    puts "I've got a lock!"
    true
  end
end

try_lock
# I've got a lock!

open('contested', 'w').flock(File::LOCK_EX) # Get a lock, hold it forever.
try_lock
# I couldn't get a lock.
```

## See Also

- Chapter 20, especially Recipe 20.11, "Avoiding Deadlock," which covers other types of deadlock problems in a multithreaded environment

# Recipe 6.14. Backing Up to Versioned Filenames


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

## Problem

You want to copy a file to a numbered backup before overwriting the original file. More generally: rather than overwriting an existing file, you want to use a new file whose name is based on the original filename.

## Solution

Use `String#succ` to generate versioned suffixes for a filename until you find one that doesn't already exist:

```
class File
  def File.versioned_filename(base, first_suffix='.0')
    suffix = nil
    filename = base
    while File.exists?(filename)
      suffix = (suffix ? suffix.succ : first_suffix)
      filename = base + suffix
    end
    return filename
  end
end

5.times do |i|
  name = File.versioned_filename('filename.txt')
  open(name, 'w') { |f| f << "Contents for run #{i}" }
  puts "Created #{name}"
end
# Created filename.txt
# Created filename.txt.0
# Created filename.txt.1
# Created filename.txt.2
# Created filename.txt.3
```

If you want to copy or move the original file to the versioned filename as a prelude to writing to the original file, include the `ftools` library to add the class methods `File.copy` and `File.move`. Then call `versioned_filename` and use `File.copy` or `File.move` to put the old file in its new place:

```
require 'ftools'
class File
def File.to_backup(filename, move=false)
   new_filename = nil
   if File.exists? filename
     new_filename = File.versioned_filename(filename)
     File.send(move ? :move : :copy, filename, new_filename)
   end
    return new_filename
 end
end
```

Let's back up `filename.txt` a couple of times. Recall from earlier that the files filename.txt.[0-3] already exist.

```
    File.to_backup('filename.txt')                  # => "filename.txt.4"
    File.to_backup('filename.txt')                  # => "filename.txt.5"
```


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

Now let's do a destructive backup:

```
File.to_backup('filename.txt', true)                # => "filename.txt.6"
File.exists? 'filename.txt'                         # => false
```

You can't back up what doesn't exist:

```
File.to_backup('filename.txt')                      # => nil
```

## Discussion

If you anticipate more than 10 versions of a file, you should add additional zeroes to the initial suffix. Otherwise, `filename.txt.10` will sort before `filename.txt.2` in a directory listing. A commonly used suffix is ".000".

```
200.times do |i|
  name = File.versioned_filename('many_versions.txt', '.000')
  open(name, 'w') { |f| f << "Contents for run #{i}" }
  puts "Created #{name}"
end
# Created many_versions.txt
# Created many_versions.txt.000
# Created many_versions.txt.001
# …
# Created many_versions.txt.197
# Created many_versions.txt.198
```

The result of `versioned_filename` won't be trustworthy if other threads or processes on your machine might be trying to write the same file. If this is a concern for you, you shouldn't be satisfied with a negative result from `File.exists?`. In the time it takes to open that file, some other process or thread might open it before you. Once you find a file that doesn't exist, you must get an exclusive lock on the file before you can be totally certain it's okay to use.

Here's how such an implementation might look on a Unix system. The `versioned_filename` methods return the name of a file, but this implementation needs to return the actual file, opened and locked. This is the only way to avoid a race condition between the time the method returns a filename, and the time you open and lock the file.

```
class File
  def File.versioned_file(base, first_suffix='.0', access_mode='w')
    suffix = file = locked = nil
    filename = base
    begin
      suffix = (suffix ? suffix.succ : first_suffix)
      filename = base + suffix
      unless File.exists? filename
        file = open(filename, access_mode)
        locked = file.flock(File::LOCK_EX | File::LOCK_NB)
        file.close unless locked
      end
```


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

```
      end until locked
      return file
    end
end

File.versioned_file('contested_file') # => #<File:contested_file.0>
File.versioned_file('contested_file') # => #<File:contested_file.1>
File.versioned_file('contested_file') # => #<File:contested_file.2>
```

The construct `begin...end until locked` creates a loop that runs at least once, and continues to run until the variable locked becomes true, indicating that a file has been opened and successfully locked.

## See Also

- Recipe 6.13, "Locking a File"

# Recipe 6.15. Pretending a String Is a File

## Problem

You want to call code that expects to read from an open file object, but your source is a string in memory. Alternatively, you want to call code that writes its output to a file, but have it actually write to a string.

## Solution

The `StringIO` class wraps a string in the interface of the `IO` class. You can treat it like a file, then get everything that's been "written" to it by calling its `string` method.

Here's a `StringIO` used as an input source:

```
require 'stringio'
s = StringIO.new %{I am the very model of a modern major general.
I've information vegetable, animal, and mineral.}

s.pos                                 # => 0
s.each_line { |x| puts x }
# I am the very model of a modern major general.
# I've information vegetable, animal, and mineral.
s.eof?                                # => true
s.pos                                 # => 95
s.rewind
s.pos                                 # => 0
s.grep /general/
# => ["I am the very model of a modern major general.\n"]
```

Here are `StringIO` objects used as output sinks:

```
s = StringIO.new
s.write('Treat it like a file.')
```


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

```
s.rewind
s.write("Act like it's")
s.string                              # => "Act like it's a file."

require 'yaml'
s = StringIO.new
YAML.dump(['A list of', 3, :items], s)
puts s.string
# ---
# - A list of
# - 3
# - :items
```

## Discussion

The Adapter is a common design pattern: to make an object acceptable as input to a method, it's wrapped in another object that presents the appropriate interface. The `StringIO` class is an Adapter between `String` and `File` (or `IO`), designed for use with methods that work on `File` or `IO` instances. With a `StringIO`, you can disguise a string as a file and use those methods without them ever knowing they haven't really been given a file.

For instance, if you want to write unit tests for a library that reads from a file, the simplest way is to pass in predefined `StringIO` objects that simulate files with various contents. If you need to modify the output of a method that writes to a file, a `StringIO` can capture the output, making it easy to modify and send on to its final destination.

`StringIO`-type functionality is less necessary in Ruby than in languages like Python, because in Ruby, strings and files implement a lot of the same methods to begin with. Often you can get away with simply using these common methods. For instance, if all you're doing is writing to an output sink, you don't need a `StringIO` object, because `String#<<` and `File#<<` work the same way:

```
def make_more_interesting(io)
  io << "… OF DOOM!"
end

make_more_interesting("Cherry pie")       # => "Cherry pie… OF DOOM!"

open('interesting_things', 'w') do |f|
   f.write("Nightstand")
   make_more_interesting(f)
end
open('interesting_things') { |f| f.read } # => "Nightstand… OF DOOM!"
```

Similarly, `File` and `String` both include the `Enumerable` mixin, so in a lot of cases you can read from an object without caring what type it is. This is a good example of Ruby's duck typing.

Here's a string:

```
poem = %{The boy stood on the burning deck
```


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

```
Whence all but he had fled
He'd stayed above to wash his neck
Before he went to bed}
```

and a file containing that string:

```
output = open("poem", "w")
output.write(poem)
output.close
input = open("poem")
```

will give the same result when you call an `Enumerable` method:

```
poem.grep /ed$/
# => ["Whence all but he had fled\n", "Before he went to bed"]
input.grep /ed$/
# => ["Whence all but he had fled\n", "Before he went to bed"]
```

Just remember that, unlike a string, you can't iterate over a file multiple times without calling rewind:

```
input.grep /ed$/                          # => []
input.rewind
input.grep /ed$/
# => ["Whence all but he had fled\n", "Before he went to bed"]
```

`StringIO` comes in when the `Enumerable` methods and << aren't enough. If a method you're writing needs to use methods specific to `IO`, you can accept a string as input and wrap it in a `StringIO`. The class also comes in handy when you need to call a method someone else wrote, not anticipating that anyone would ever need to call it with anything other than a file:

```
def fifth_byte(file)
  file.seek(5)
  file.read(1)
end

fifth_byte("123456")
# NoMethodError: undefined method `seek' for "123456":String
fifth_byte(StringIO.new("123456"))           # => "6"
```

When you write a method that accepts a file as an argument, you can silently accommodate callers who pass in strings by wrapping in a `StringIO` any string that gets passed in:

```
def file_operation(io)
  io = StringIO(io) if io.respond_to? :to_str && !io.is_a? StringIO
  #Do the file operation…
end
```

A `StringIO` object is always open for both reading and writing:


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

```
s = StringIO.new
s << "A string"
s.read                    # => ""
s << ", and more."
s.rewind
s.read                    # => "A string, and more."
```

Memory access is faster than disk access, but for large amounts of data (more than about 10 kilobytes), `StringIO` objects are slower than disk files. If speed is your aim, your best bet is to write to and read from temp files using the `tempfile` module. Or you can do what the `open-uri` library does: start off by writing to a `StringIO` and, if it gets too big, switch to using a temp file.

## See Also

- Recipe 6.8, "Writing to a Temporary File"
- Recipe 6.11, "Performing Random Access on "Read-Once" Input Streams"

# Recipe 6.16. Redirecting Standard Input or Output

## Problem

You don't want the standard input, output, or error of your process to go to the default `IO` objects set up by the Ruby interpreter. You want them to go to other filetype objects of your own choosing.

## Solution

You can assign any `IO` object (a `File`, a `Socket`, or what have you) to the global variables `$stdin`, `$stdout`, or `$stderr`. You can then read from or write to those objects as though they were the originals.

This short Ruby program demonstrates how to redirect the `Kernel` methods that print to standard output. To avoid confusion, I'm presenting it as a standalone Ruby program rather than an interactive `irb` session.[4]

[4] `irb` prints the result of each Ruby expression to `$stdout`, which tends to clutter the results in this case.

```
#!/usr/bin/ruby -w
# ./redirect_stdout.rb
require 'stringio'
new_stdout = StringIO.new

$stdout = new_stdout
puts "Hello, hello."
puts "I'm writing to standard output."

$stderr.puts "#{new_stdout.size} bytes written to standard ouput so far."
```


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

```
$stderr.puts "You haven't seen anything on the screen yet, but you soon will:"
$stderr.puts new_stdout.string
```

Run this program and you'll see the following:

```
$ ruby redirect_stdout.rb
46 bytes written to standard output so far.
You haven't seen anything on the screen yet, but you soon will:
Hello, hello.
I'm writing to standard output.
```

## Discussion

If you have any Unix experience, you know that when you run a Ruby script from the command line, you can make the shell redirect its standard input, output, and error streams to files or other programs. This technique lets you do the same thing from within a Ruby script.

You can use this as a quick and dirty way to write errors to a file, write output to a `StringIO` object (as seen above), or even read input from a socket. Within a script, you can programatically decide where to send your output, or receive standard input from multiple sources. These things are generally not possible from the command line without a lot of fancy shell scripting.

The redirection technique is especially useful when you've written or inherited a script that prints text to standard output, and you need to make it capable of printing to any file-like object. Rather than changing almost every line of your code, you can just set `$stdout` at the start of your program, and let it run as is. This isn't a perfect solution, but it's often good enough.

The original input and output streams for a process are always available as the constants `STDIN, STDOUT`, and `STDERR`. If you want to temporarily swap one IO stream for another, change back to the "standard" standard output by setting `$stdin = STDIN`. Keep in mind that since the `$std` objects are global variables, even a temporary change affects all threads in your script.

## See Also

Recipe 6.15, "Pretending a String Is a File," has much more information on `StringIO`

# Recipe 6.17. Processing a Binary File

## Problem

You want to read binary data from a file, or write it to one.


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

## Solution

Since Ruby strings make no distinction between binary and text data, processing a binary file needn't be any different than processing a text file. Just make sure you add "b" to your file mode when you open a binary file on Windows.

This code writes 10 bytes of binary data to a file, then reads it back:

```
open('binary', 'wb') do |f|
  (0..100).step(10) { |b| f << b.chr }
end

s = open('binary', 'rb') { |f| f.read }
# => "\000\n\024\036(2<FPZd"
```

If you want to process a binary file one byte at a time, you'll probably enjoy the way `each_byte` returns each byte of the file as a number, rather than as single-character strings:

```
open('binary', 'rb') { |f| f.each_byte { |b| puts b } }
# 0
# 10
# 20
# …
# 90
# 100
```

## Discussion

The methods introduced earlier to deal with text files work just as well for binary files, assuming that your binary files are supposed to be processed from beginning to end, the way text files typically are. If you want random access to the contents of a binary file, you can manipulate your file object's "cursor."

Think of the cursor as a pointer to the first unread byte in the open file. The current position of the cursor is accessed by the method `IO#pos`. When you open the file, it's set to zero, just before the first byte. You can then use `IO#read` to read a number of bytes starting from the current position of the cursor, incrementing the cursor as a side effect.

```
f = open('binary')
f.pos                                   # => 0
f.read(1)                               # => "\000"
f.pos                                   # => 1
```

You can also just set `pos` to jump to a specific byte in the file:

```
f.pos = 4                               # => 4
f.read(2)                               # => "(2"
f.pos                                   # => 6
```


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

You can use `IO#seek` to move the cursor forward or backward relative to its current position (with `File::SEEK_CUR`), or to move to a certain distance from the *end* of a file (with `File::SEEK_END`). Unlike the iterator methods, which go through the entire file once, you can use `seek` or `set pos` to jump anywhere in the file, even to a byte you've already read.

```
f.seek(8)
f.pos                                # => 8

f.seek(-4, File::SEEK_CUR)
f.pos                                # => 4
f.seek(2, File::SEEK_CUR)
f.pos                                # => 6

# Move to the second-to-last byte of the file.
f.seek(-2, File::SEEK_END)
f.pos                                # => 9
```

Attempting to read more bytes than there are in the file returns the rest of the bytes, and set your file's `eof?` flag to true:

```
f.read(500)                 # => "Zd"
f.pos                       # => 11
f.eof?                      # => true
f.close
```

Often you need to read from and write to a binary file simultaneously. You can open any file for simultaneous reading and writing using the "r+" mode (or, in this case, "rb+"):

```
f = open('binary', 'rb+')
f.read                                  # => "\000\n\024\036(2<FPZd"
f.pos = 2
f.write('Hello.')
f.rewind
f.read                                  # => "\000\nHello.PZd"
f << 'Goodbye.'
f.rewind
f.read                                  # => "\000\nHello.PZdGoodbye."

f.close
```

You can append new data to the end of a file you've opened for read-write access, and you can overwrite existing data byte for byte, but you can't insert new data into the middle of a file. This makes the read-write technique useful for binary files, where exact byte offsets are often important, and less useful for text files, where it might make sense to add an extra line in the middle.

Why do you need to append "b" to the file mode when opening a binary file on Windows? Because otherwise Windows will mangle any newline characters that show up in your binary file. The "b" tells Windows to leave the newlines alone, because they're not really newlines: they're binary data. Since it doesn't hurt anything on Unix to put "b" in the file


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

mode, you can make your code cross-platform by appending "b" to the mode whenever you open a file you plan to treat as binary. Note that "b" by itself is not a valid file mode: you probably want "rb".

## An MP3 example

Because every binary format is different, probably the best I can do to help you beyond this point is show you an example. Consider MP3 music files. Many MP3 files have a 128-byte data structure at the end called an *ID3 tag*. These 128 bytes are literally packed with information about the song: its name, the artist, which album it's from, and so on. You can parse this data structure by opening an MP3 file and doing a series of reads from a `pos` near the end of the file.

According to the ID3 standard, if you start from the 128th-to-last byte of an MP3 file and read three bytes, you should get the string "TAG". If you don't, there's no ID3 ag for this MP3 file, and nothing to do. If there is an ID3 tag present, then the 30 bytes after "TAG" contain the name of the song, the 30 bytes after that contain the name of the artist, and so on. Here's some code that parses a file's ID3 tag and puts the results into a hash:

```
def parse_id3(mp3_file)
  fields_and_sizes = [[:track_name, 30], [:artist_name, 30],
                      [:album_name, 30], [:year, 4], [:comment, 30],
                      [:genre, 1]]

  tag = {}
  open(mp3_file) do |f|
    f.seek(-128, File::SEEK_END)
    if f.read(3) == 'TAG' # An ID3 tag is present
      fields_and_sizes.each do |field, size|
        # Read the field and strip off anything after the first null
        # character.
        data = f.read(size).gsub(/\000.*/, '')
        # Convert the genre string to a number.
        data = data[0] if field == :genre
        tag[field] = data
      end
    end
  end
  return tag
end

parse_id3('ID3.mp3')
# => {:year=>"2005", :artist_name=>"The ID Three",
# :album_name=>"Binary Brain Death",
# :comment=>"http://www.example.com/id3/", :genre=>22,
# :track_name=>"ID 3"}

parse_id3('Too Indie For ID3 Tags.mp3')     # => {}
```

Rather than specifying the genre of the music as a string, the `:genre` element of the hash is a single byte, an entry into a lookup table shared by all applications that use ID3. In this table, genre number 22 is "Death metal".


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

It's less code to specify the byte offsets for a binary file is in the format recognized by `String#unpack`, which can parse the bytes of a string according to a given format. It returns an array containing the results of the parsing.

```
#Returns [track, artist, album, year, comment, genre]
def parse_id3(mp3_file)
  format = 'Z30Z30Z30Z4Z30C'
  open(mp3_file) do |f|
    f.seek(-128, File::SEEK_END)
    if f.read(3) == "TAG" # An ID3 tag is present
      return f.read(125).unpack(format)
    end
  end
  return nil
end

parse_id3('ID3.mp3')
# => ["ID 3", "The ID Three", "Binary Brain Death", "2005", "http://www.example.com/
id3/", 22]
```

As you can see, the `unpack` format is obscure but very concise. The string "Z30Z30Z30Z4Z30C" passed into `String#unpack` completely describes the elements of the ID3 format after the "TAG":

- Three strings of 30 bytes, with null characters stripped ("Z30Z30Z30")
- A string of 4 bytes, with null characters stripped ("Z4")
- One more string of 30 bytes, with null characters stripped ("Z30")
- A single character, represented as an unsigned integer ("C")

It doesn't describe what those elements are supposed to be used for, though.

When writing binary data to a file, you can use `Array#pack`, the opposite of `String#unpack`:

```
id3 = ["ID 3", "The ID Three", "Binary Brain Death", "2005",
       "http://www.example.com/id3/", 22]
id3.pack 'Z30Z30Z30Z4Z30C'
# => "ID 3\000\000\000\000\000…http://www.example.com/id3/\000\000\000\026"
```

## See Also

- The ID3 standard, described at http://en.wikipedia.org/wiki/ID3 along with the table of genres; the code in this recipe parses the original ID3v1 standard, which is much simpler than ID3v2
- `ri String#unpack` and `ri Array#pack`

# Recipe 6.18. Deleting a File


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

## Problem

You want to delete a single file, or a whole directory tree.

## Solution

Removing a file is simple, with `File.delete`:

```
import 'fileutils'
FileUtils.touch "doomed_file"
File.exists? "doomed_file"                    # => true
File.delete "doomed_file"
File.exists? "doomed_file"                    # => false
```

Removing a directory tree is also fairly simple. The most confusing thing about it is the number of different methods Ruby provides to do it. The method you want is probably `FileUtils.remove_dir`, which recursively deletes the contents of a directory:

```
Dir.mkdir "doomed_directory"
File.exists? "doomed_directory"            # => true
FileUtils.remove_dir "doomed_directory"
File.exists? "doomed_directory"            # => false
```

## Discussion

Ruby provides several methods for removing directories, but you really only need `remove_dir`. `Dir.delete` and `FileUtils.rmdir` will only work if the directory is already empty. The `rm_r` and `rm_rf` defined in `FileUtils` are similar to `remove_dir`, but if you're a Unix user you may find their names more mneumonic.

You should also know about the `:secure` option to `rm_rf`, because the `remove_dir` method and all its variants are vulnerable to a race condition when you remove a world-writable directory. The risk is that a process owned by another user might create a symlink in that directory while you're deleting it. This would make you delete the symlinked file along with the files you actually meant to delete.

Passing in the `:secure` option to `rm_rf` slows down deletions significantly (it has to change the permissions on the directory before deleting it), but it avoids the race condition. If you're running Ruby 1.8, you'll also need to hack the `FileUtils` module a little bit to work around a bug (the bug is fixed in Ruby 1.9):?

```
# A hack to make a method used by rm_rf actually available
module FileUtils
  module_function :fu_world_writable?
end

Dir.mkdir "/tmp/doomed_directory"
FileUtils.rm_rf("/tmp/doomed_directory", :secure=>true)
File.exists? "/tmp/doomed_directory"            # => false
```


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

Why isn't the `:secure` option the default for rm_rf? Because secure deletion isn't thread-safe: it actually changes the current working directory of the process. You need to choose between thread safety and a possible security hole.

# Recipe 6.19. Truncating a File

## Problem

You want to truncate a file to a certain length, probably zero bytes.

## Solution

Usually, you want to destroy the old contents of a file and start over. Opening a file for write access will automatically truncate it to zero bytes, and let you write new contents to the file:

```
filename = 'truncate.txt'
open(filename, 'w') { |f| f << "All of this will be truncated." }
File.size(filename)                  # => 30

f = open(filename, 'w') {}
File.size(filename)                  # => 0
```

If you just need to truncate the file to zero byt es, and not write any new contents to it, you can open it with an access mode of `File::TRUNC`.

```
open(filename, 'w') { |f| f << "Here are some new contents." }

File.size(filename)                        # => 27

f = open(filename, File::TRUNC) {}
File.size(filename)                        # => 0
```

You can't actually do anything with a `FILE` whose access mode is `File::TRUNC`:

```
open(filename, File::TRUNC) do |f|
  f << "At last, an empty file to write to!"
end
# IOError: not opened for writing
```

## Discussion

Transient files are the most likely candidates for truncation. Log files are often truncated, automatically or by hand, before they grow too large.

The most common type of truncation is truncating a file to zero bytes, but the `File.truncate` method can truncate a file to any number of bytes, not just zero. You


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

can also use the instance method, `File#truncate`, to truncate a file you've opened for writing:

```
f = open(filename, 'w') do |f|
  f << 'These words will remain intact after the file is truncated.'
end
File.size(filename) # => 59

File.truncate(filename, 30)
File.size(filename)               # => 30
open(filename) { |f| f.read }     # => "These words will remain intact"
```

These methods don't always make a file smaller. If the file starts out smaller than the size you give, they append zero-bytes (\000) to the end of file until the file reaches the specified size.

```
f = open(filename, "w") { |f| f << "Brevity is the soul of wit." }
File.size(filename)                 # => 27
File.truncate(filename, 30)
File.size(filename)                 # => 30
open(filename) { |f| f.read }
# => "Brevity is the soul of wit.\000\000\000"
```

`File.truncate` and `File#truncate` act like the bed of Procrustes: they force a file to be a certain number of bytes long, whether that means stretching it or chopping off the end.

# Recipe 6.20. Finding the Files You Want

## Problem

You want to locate all the files in a directory hierarchy that match some criteria. For instance, you might want to find all the empty files, all the MP3 files, or all the files named "README."

## Solution

Use the `Find.find` method to walk the directory structure and accumulate a list of matching files.

Pass in a block to the following method and it'll walk a directory tree, testing each file against the code block you provide. It returns an array of all files for which the value of the block is true.

```
require 'find'
module Find
  def match(*paths)
    matched = []
    find(*paths) { |path| matched << path if yield path }
    return matched
```


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

```
  end
  module_function :match
end
```

Here's what `Find.match` might return if you used it on a typical disorganized home directory:

```
Find.match("./") { |p| File.lstat(p).size == 0 }
# => ["./Music/cancelled_download.MP3", "./tmp/empty2", "./tmp/empty1"]

Find.match("./") { |p| ext = p[-4…p.size]; ext && ext.downcase == ".mp3" }
# => ["./Music/The Snails - Red Rocket.mp3",
# => "./Music/The Snails - Moonfall.mp3", "./Music/cancelled_download.MP3"]

Find.match("./") { |p| File.split(p)[1] == "README" }
# => ["./rubyprog-0.1/README", "./tmp/README"]
```

## Discussion

This is an especially useful chunk of code for system administration tasks. It gives you functionality at least as powerful as the Unix `find` command, but you can write your search criteria in Ruby and you won't have to remember the arcane syntax of `find`.

As with `Find.walk` itself, you can stop `Find.match` from processing a directory by calling `Find.prune`:

```
Find.match("./") do |p|
  Find.prune if p == "./tmp"
  File.split(p)[1] == "README"
end
# => ["./rubyprog-0.1/README"]
```

You can even look inside each file to see whether you want it:

```
# Find all files that start with a particular phrase.
must_start_with = "This Ruby program"
Find.match("./") do |p|
  if File.file? p
    open(p) { |f| f.read(must_start_with.size) == must_start_with }
  else
    false
  end
end
# => ["./rubyprog-0.1/README"]
```

A few other useful things to search for using this function:

```
# Finds files that were probably left behind by emacs sessions.
def emacs_droppings(*paths)
  Find.match(*paths) do |p|
    (p[-1] == ?~ and p[0] != ?~) or (p[0] == ?# and p[-1] == ?#)
  end
end

# Finds all files that are larger than a certain threshold. Use this to find
# the files hogging space on your filesystem.
```


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

```
def bigger_than(bytes, *paths)
  Find.match(*paths) { |p| File.lstat(p).size > bytes }
end

# Finds all files modified more recently than a certain number of seconds ago.
def modified_recently(seconds, *paths)
  time = Time.now - seconds
  Find.match(*paths) { |p| File.lstat(p).mtime > time }
end

# Finds all files that haven't been accessed since they were last modified.
def possibly_abandoned(*paths)
  Find.match(*paths) { |p| f = File.lstat(p); f.mtime == f.atime }
end
```

## See Also

- Recipe 6.12, "Walking a Directory Tree"

# Recipe 6.21. Finding and Changing the Current Working Directory

## Problem

You want to see which directory the Ruby process considers its current working directory, or change that directory.

## Solution

To find the current working directory, use `Dir.getwd`:

```
Dir.getwd                    # => "/home/leonardr"
```

To change the current working directory, use `Dir.chdir`:

```
Dir.chdir("/bin")
Dir.getwd          # => "/bin"
File.exists? "ls"  # => true
```

## Discussion

The current working directory of a Ruby process starts out as the directory you were in when you started the Ruby interpreter. When you refer to a file without providing an absolute pathname, Ruby assumes you want a file by that name in the current working directory. Ruby also checks the current working directory when you `require` a library that can't be found anywhere else.


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

The current working directory is a useful default. If you're writing a Ruby script that operates on a directory tree, you might start from the current working directory if the user doesn't specify one.

However, you shouldn't rely on the current working directory being set to any particular value: this makes scripts brittle, and prone to break when run from a different directory. If your Ruby script comes bundled with libraries, or needs to load additional files from subdirectories of the script directory, you should set the working directory in code.

You can change the working directory as often as necessary, but it's more reliable to use absolute pathnames, even though this can make your code less portable. This is especially true if you're writing multithreaded code.

The current working directory is global to a process. If multiple threads are running code that changes the working directory to different values, you'll never know for sure what the working directory is at any given moment.

## See Also

- Recipe 6.18, "Deleting a File," shows some problems created by a process-global working directory


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