# Working with Files in Ruby

www.vikingcodeschool.com

Ruby makes your life pretty easy with regards to dealing with files. It has the tools necessary to read those long streams of bytes into your program and then allows you to work with them using the objects you're familiar with.

As long as you remember that the files are just a long stream of words/characters/bytes being read in from top to bottom, it should be fairly intuitive. Sometimes you might need to dig a bit deeper than the basics (like if you write to a specific point inside a file), but for the most part it's quite simple.

In this lesson, we'll cover how to open, read, update, and close files using Ruby so you can fully explore the (computing) world around you.

Our goal is to give you a taste of working with files so you'll be comfortable writing your serialized objects to or reading them from a file.

## Opening and Reading Files

To open a file, Ruby needs to know two things -- the path to the file and how you'd like to open it. The path should be obvious and the "open mode" is necessary because some files are only opened as "read-only" while others can be "read-write".

See more about modes here if you're curious. The most common are:

- **"r"** -- read only from the beginning of the file
- **"r+"** -- read/write from the beginning of the file
- **"w"** -- write only, overwriting the entire existing file or creating a new one if none existed
- **"w+"** -- read/write, overwriting the entire existing file or creating a new one if none existed
- **"a"** -- write only, starting at the end of existing file or creating a new one if none existed

- **"a+"** -- read/write, starting at the end of existing file or creating a new one if none existed

Here's an example where we open a file in Read/Write mode and write to it:

```
# Set the mode to Read/Write, starting at the beginning
> mode = "r+"
#=> "r+"

# Open the file with the specified mode
# The file information is saved as the `File` class
> file = File.open("fruits.txt", mode)
#=> #<File:fruits.txt>

# Now output the contents of the file directly
# In this case, the file itself contains several
# lines of fruit types.  The newlines come from
# the file contents, not anything to do with `puts`.
> puts file.read
apple
banana
orange
> file.close
#=> nil
```

Just remember to **Always close your files**. If you forget to close the file, you could have a memory leak on your hands. As you'll see, some of Ruby's file I/O methods do that for you.

For instance... you can just pass the **open** method a block and it will automatically read the file, give you access to it within the block, and then close the file when the block is done. Isn't Ruby helpful?

```
> File.open("fruits.txt", "r+") do |file|
>    puts file.read
> end
apple
banana
orange
#=> nil
```

## Reading the File

The file stream is one long string. You saw the **`read`** method above, which simply lets you grab onto that stream and feed it into some other object like a variable or directly to STDOUT with **`puts`**.

Because we know you're lazy, you can actually read a file in one step by calling **`read`** on the filename itself without bothering to use the full **`open`** method.

**`read`** returns the full string of the file, including any line breaks, in read-only mode from the beginning:

```
> File.read("fruits.txt")
#=> "apple\nbanana\norange"
```

### Reading Line-by-Line to an Array

If you want to read it line-by-line (for instance to save each line to a separate item in an array), you'll want to use Ruby's helpful **`readlines`** method instead of the normal **`read`**. You can use this in any place where you might otherwise use **`read`**:

```
# The long way:
> File.open("fruits.txt", "r+") do |file|
>    arr = file.readlines
>    puts arr.inspect
> end
["apple\n", "banana\n", "orange"]
#=> nil

# The short way:
> arr = File.readlines("fruits.txt")
#=> ["apple\n", "banana\n", "orange"]
> arr.inspect
#=> "[\"apple\\n\", \"banana\\n\", \"orange\"]"
```

Since the **readlines** method returns an array, you can run array methods like **each** on it for convenience:

```
> File.readlines("fruits.txt").each do |line|
>    puts line.upcase
> end
APPLE
BANANA
ORANGE
#=> ["apple\n", "banana\n", "orange"]
```

...of course, you'll probably want to **chomp** or **strip** the lines to remove any newline characters as well if you're handling text:

```
> arr = []
#=> []

> file_lines = File.readlines("fruits.txt")
#=> ["apple\n", "banana\n", "orange"]

> file_lines.each do |line|
>   arr << line.strip
> end
#=> ["apple\n", "banana\n", "orange"]  # each returning the
original array

> arr
#=> ["apple", "banana", "orange"] # your new cleaner array
```

## Writing to a File

Writing to a file has a very similar feel as reading from a file -- it's the same thing but in reverse. Simply use the **`write`** method instead of the **`read`** method and supply the string you'd like to read.

Here's the important part -- **make sure you open the file with the proper mode or you might accidentally overwrite the whole thing!** There aren't any warning dialogs when working with files programmatically.

```
# Let's overwrite/create `test.txt`
# This method returns the number of characters written by
# the last write
File.open("test.txt","w") do |file|
    file.write "I am the test text!!!"
    file.write "...hear me roar."
end
#=> 16

# Now let's double check that the file is as expected
# Note that there are no newlines created just because
# you had a couple different `file.write` lines...
# `file.write` acts more like `p` than `puts`.
> File.read("test.txt")
#=> "I am the test text!!!...hear me roar."
```

## Using File I/O to Save Objects

Great, so you've learned serialization and File I/O. The natural step is combining the two to save your objects to a file or read them back again. Easy!

```
> require 'json'
#=> true

# Our character from the game
> my_hash = {:name => "Dolph", :age => 21}
#=> {:name => "Dolph", :age => 21}

# Write (or overwrite) our character to `saved_hash.txt`
> File.open("saved_hash.txt","w") do |file|
>     file.write my_hash.to_json
> end
#=> 25

# Check that our file looks right
> File.read("saved_hash.txt")
#=> "{\"name\":\"Dolph\",\"age\":21}"

# Bring the object back into Ruby
> my_saved_hash = JSON.parse(File.read("saved_hash.txt"))
# => {"name"=>"Dolph", "age"=>21}
```

Note that you can save any objects, not just hashes... it even works with classes!

### Saving Class Instances

You'll want to be careful when serializing complex objects like class instances. Let's look at what happens with JSON and YAML:

```
> some_instance = SomeClass.new
=> #<SomeClass:0x007f972fc68180 @var1="val1", @var2="val2">

> require 'json'
=> true

# The class instance is just saved as a string...
# of the class name and memory location
> my_json_instance = some_instance.to_json
=> "\"#<SomeClass:0x007f972fc68180>\""

# ...so when we parse it, JSON gets mad
> JSON.parse(my_json_instance)
JSON::ParserError: 757: unexpected token at '"#
<SomeClass:0x007f972fc68180>"'
from /Users/eriktrautman/.rvm/gems/ruby-2.0.0-p481/gems/json-
1.8.1/lib/json/common.rb:155:in `parse'

# Let's try YAML instead
> require 'yaml'
=> true

# Well, it seems to clearly demark it as a Ruby object...
> my_yaml_instance = some_instance.to_yaml
=> "--- !ruby/object:SomeClass\nvar1: val1\nvar2: val2\n"

# Okay, this looks a bit better
> YAML.load(my_yaml_instance)
=> #<SomeClass:0x007f97300ea9d8 @var1="val1", @var2="val2">
```

In this case, JSON broke and YAML worked out fine. Keep that in mind!

If you end up building more robust applications that should be serialized to JSON (e.g. web apps transmitting classes), it makes sense to write your own **`to_json`** and **`JSON.parse`** methods which work within the framework of your app. For instance, part of the Rails framework is the Active Support gem that contains Rails-friendly versions of these common JSON operations so you can blissfully serialize and deserialize ActiveRecord objects.

### What to Serialize and Save

So should you serialize/save/send the entire class instance? Just the essential bits? What if you have a whole array of class instances or hashes?

Generally, serialization is just an intermediate step before storing or sending data. Thus the rule of thumb is to simply save at whatever granularity you're going to need when it's time to receive/open the sent/saved object. If you will need to boot up an entire saved game, how much is necessary to determine the full game state?

When you're trying to do things like saving multiple class objects to a single file, you'll need to get a bit creative, for instance storing them in an array first or deliberately entering a custom delineator like a double-newline. Don't sweat that now, though.

## Code Review

*The important bits of code from this lesson*

```
# Open the file with the specified mode
our_file = File.open("fruits.txt", "r+")

# Now output the contents of the file directly
puts our_file.read

# Close the file after we're done
our_file.close
#=> nil

# Read the file and automatically close it
File.read("fruits.txt")

# Read the file into a new array item for each line
File.readlines("fruits.txt")

# You can run array methods on the resulting array
# and should strip off extraneous newlines to clean it up
File.readlines("fruits.txt").each do |line|
  some_arr << line.strip
end
```

## Wrapping Up

Working with files in Ruby is generally pretty straightforward. Pairing this knowledge with an understanding of how to serialize and de-serialize objects gives you all the tools you need to play effectively with your operating system. Fly free and build some cool stuff!