Extracted from:

Build Awesome Command-Line Applications in Ruby

Control Your Computer, Simplify Your Life

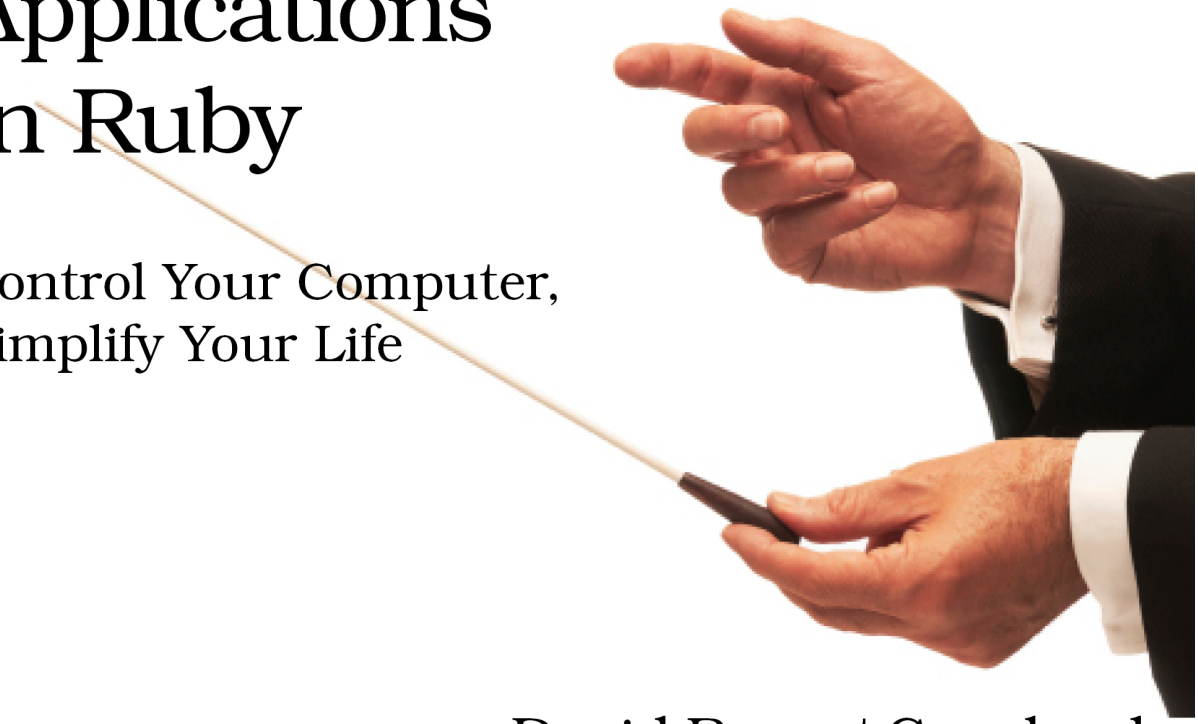
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Build Awesome Command-Line Applications in Ruby

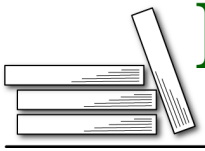
Control Your Computer, Simplify Your Life



David Bryant Copeland

Edited by John Osborn

The Facets of Ruby Series



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

John Osborn (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-91-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

4.2 Using the Standard Output and Error Streams Appropriately

In addition to the ability to return a single value to the calling program, all programs have the ability to provide output. The puts method is the primary way of creating output that we've seen thus far. We've used it to send messages to the terminal. A command line's output mechanism is actually more sophisticated than this; it's possible to send output to either of two standard *output streams*.

By convention, the default stream is called the *standard output* and is intended to receive whatever normal output comes out of your program. This is where puts sends its argument and where, for example, mysqldump sends the SQL statements that make up the database backup.[1]

The second output stream is called the *standard error* stream and is intended for error messages. The reason there are two different streams is so that the calling program can easily differentiate normal output from error messages. Consider how we use mysqldump in db_backup.rb:

play_well/db_backup/bin/db_backup.rb

```
command = "mysqldump #{auth}#{database_name} > #{output_file}"
system(command)
unless $CHILD_STATUS.exitstatus == 0
  puts "There was a problem running '#{command}'"
  exit 1
end
```

Currently, when our app exits with a nonzero status, it outputs a generic error message. This message doesn't tell the user the nature of the problem, only that something went wrong. mysqldump actually produces a specific message on its standard error stream. We can see this by using the UNIX redirect operator (>) to send mysqldump's standard output to a file, leaving the standard error as the only output to our terminal:

```
$ mysqldump some_nonexistent_database > backup.sql
mysqldump: Got error: 1049: Unknown database 'some_nonexistent_database' \
when selecting the database
```

backup.sql contains the standard output that mysqldump generated, and we see the standard error in our terminal; it's the message about an unknown database. If we could access this message and pass it along to the user, the user would know the actual problem.

1. A "database backup" produced by mysqldump is a set of SQL statements that, when executed, re-create the backed-up database.

Using Open3 to Access the Standard Output and Error Streams Separately

The combination of system and $CHILD_STATUS that we've used so far provides access only to the exit status of the application. We can get access to the standard output by using the built-in backtick operator (`) or the %x[] construct, as in stdout = %x[ls -l]. Unfortunately, neither of these constructs provides access to the standard error stream. To get access to both the standard output and the standard error independently, we need to use a module from the standard library called Open3.

Open3 has several useful methods, but the most straightforward is capture3. It's so-named because it "captures" the standard output and error streams (each as a String), as well as the status of the process (as a Process::Status, the same type of variable as $CHILD_STATUS). We can use this method's results to augment our generic error message with the contents of the standard error stream like so:

play_well/db_backup/bin/db_backup_2.rb

```
require 'open3'

puts "Running '#{command}'"
stdout_str, stderr_str, status = Open3.capture3(command)

unless status.exitstatus == 0
  puts "There was a problem running '#{command}'"
  puts stderr_str
  exit -1
end
```

The logic is exactly the same, except that we have much more information to give the user when something goes wrong. Since the standard error from mysqldump contains a useful error message, we're now in a position to pass it along to the user:

```
$ db_backup.rb -u dave.c -p P@ss5word some_nonexistent_database
There was a problem running 'mysqldump -udavec -pP@55word \
some_nonexistent_database > some_nonexistent_database.sql'
mysqldump: Got error: 1049: Unknown database 'some_nonexistent_database' \
when selecting the database
```

Our use of the standard error stream allows us to "handle" any error from mysqldump, such as bad login credentials, which also generates a useful error message:

```
$ db_backup.rb -u dave.c -p password some_nonexistent_database
There was a problem running 'mysqldump -udavec -ppassword \
some_nonexistent_database > some_nonexistent_database.sql'
mysqldump: Got error: 1044: Access denied for user 'dave.c'@'localhost'\
```

```
➤ to database 'some_nonexistent_database' when selecting the database
```

It's always good practice to capture the output of the commands you run and either send it to your app's output or store it in a log file for later reference (we'll see later why you might not want to just send such output to your app's output directly). Note that the version of Open3 that is included in Ruby 1.8 is not sufficient for this purpose; it hides the exit code from us. See *Open3 and Ruby 1.8*, on page 8 for a workaround if you're stuck using Ruby 1.8.

Now that we can read these output streams from programs we execute, we need to start writing to them as well. We just added new code to output an error message, but we used puts, which sends output to the standard output stream. We need to send our error messages to the right place.

Use STDOUT and STDERR to Send Output to the Correct Stream

Under the covers, puts sends output to STDOUT, which is a constant provided by Ruby that allows access to the standard output stream. It's an instance of IO, and essentially the code puts "hello world" is equivalent to STDOUT.puts "hello world". Ruby sets another constant, STDERR, to allow output to the standard error stream (see *STDOUT and STDERR vs. $stdout and $stderr*, on page 9 for another way to access these streams). Changing our app to use STDERR to send error messages to the standard error stream is trivial:

play_well/db_backup/bin/db_backup_3.rb

```
stdout_str, stderr_str, status = Open3.capture3(command)

unless status.success?
➤  STDERR.puts "There was a problem running '#{command}'"
➤  STDERR.puts stderr_str.gsub(/^mysqldump: /,'')
  exit 1
end
```

You could also use the method warn (provided by Kernel) to output messages to the standard error stream. Messages sent with warn can be disabled by the user, using the -W0 flag to ruby (or putting that in the environment variable RUBYOPTS, which is read by Ruby before running any Ruby app). If you want to be sure the user sees the message, however, use STDERR.puts.

Users of our app can now use our standard error stream to get any error messages we might generate. In general, the standard error of apps we call should be sent to our standard error stream.

We now know how to read output from and write output to the appropriate error stream, and we've started to get a sense of what messages go where. Error messages go to the standard error stream, and "everything else" goes

Open3 and Ruby 1.8

Open3 in Ruby 1.8 is far less useful than the version that ships with Ruby 1.9.2. Its main failing is that it doesn't provide access to the exit code of our process. $CHILD_STATUS *is* set; however, the exitstatus method always returns zero, even if the underlying process exited nonzero.

If you can't use Ruby 1.9.2 but still want the benefits of 1.9.2's much-improved Open3 class, there is a library called Open4[a] that works with Ruby 1.8 to do exactly what we need. We could use it like so:

```
$ gem install open4

require 'open4'

pid, stdin, stdout, stderr = Open4::popen4(command)
_, status = Process::waitpid2(pid)
unless status.exitstatus == 0
  puts "There was a problem running '#{command}'"
  puts stderr
end
```

Open4 has the advantage of working on versions of Ruby 1.8 and newer. If you don't need 1.8 compatibility, using the built-in Open3 is preferred, since it's included with Ruby. However, it's nice to know that a third-party gem can do most of what we want on 1.8.

a. https://github.com/ahoward/open4

to the standard output stream. How do we know what's an "error message" and what's not? And for our "normal" output, what format should we use to be most interoperable with other applications?

Use the standard error stream for any message that isn't the proper, expected output of your application. We can take a cue from mysqldump here; mysqldump produces the database backup, as SQL, to its standard output. Everything else it produces goes to the standard error. It's also important to produce *something* to the standard error if your app is going to exit nonzero; this is the only way to tell the user what went wrong.

The standard output, however, is a bit more complicated. You'll notice that mysqldump produces a very specific format of output to the standard output (SQL). There's a reason for this. Its output is designed to be handed off, directly, as input to another app. Achieving this is not nearly as straightforward as producing a human-readable error message, as we'll see in the next section.

STDOUT and STDERR vs. $stdout and $stderr

In addition to assigning the constants `STDOUT` and `STDERR` to the standard output and error streams, respectively, Ruby also assigns the global variables `$stdout` and `$stderr` to these two streams (in fact, `puts` uses `$stdout` internally).

Deciding which one to use is a mostly a matter of taste, but it's worth noting that by using the variable forms, you can easily reassign the streams each represents. Although reassigning the value of a constant is possible in Ruby, it's more straightforward to reassign the value of a variable. For example, you might want to reassign your input and output during testing to capture what's going to the standard error or output streams.

We'll use the constant forms in this book, because we want to think of the standard output and error streams as immutable. The caller of our app should decide whether these streams should be redirected elsewhere, and if we ever need to send output to one of the streams *or* another `IO` instance, we would abstract that out, rather than reassign `$stdin`.