Extracted from:

Build Awesome Command-Line Applications in Ruby

Control Your Computer, Simplify Your Life

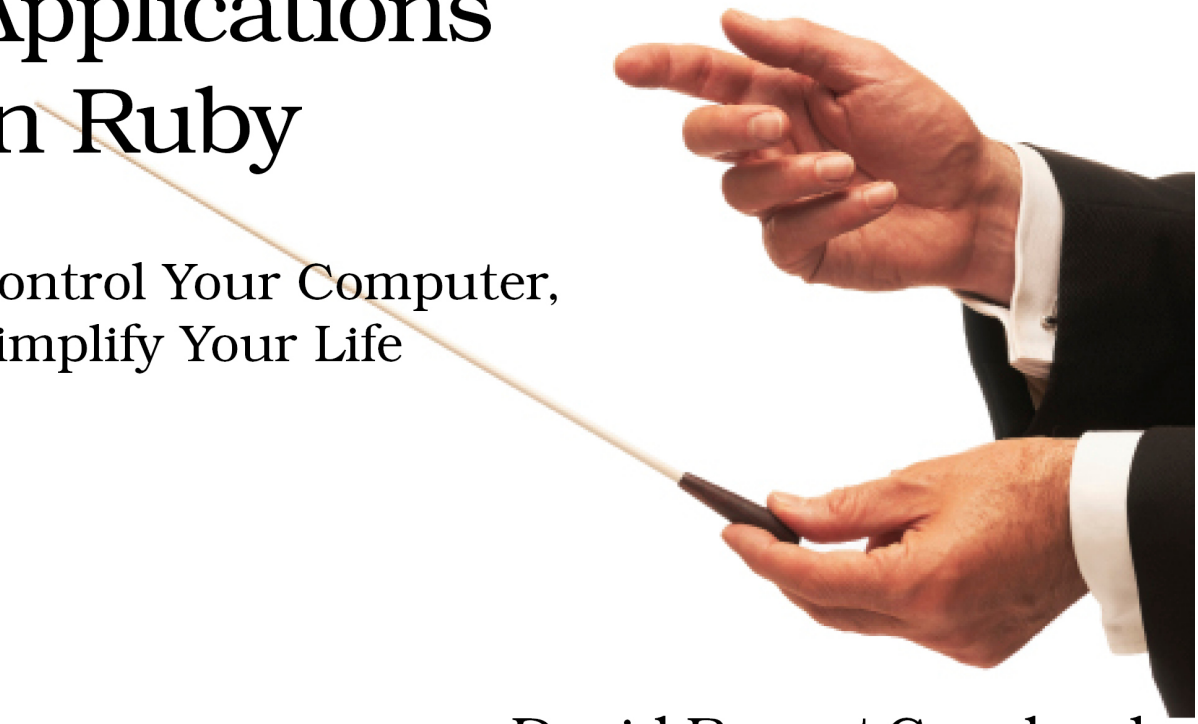
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Build Awesome Command-Line Applications in Ruby

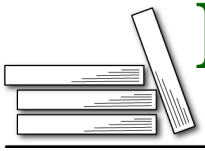
Control Your Computer, Simplify Your Life



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

5.1 Choosing Names for Options and Commands

In Chapter 2, *Be Easy to Use*, on page ?, we learned that OptionParser allows us to create multiple options that mean the same thing. We used this feature in our database backup script, db_backup.rb, by allowing both -i and --iteration to signify an "end-of-iteration" backup. Why does OptionParser have this feature, and why did we use it?

Naming Options

This question is better posed in two parts: "Why did we provide a short-form option?" and "Why did we provide a long-form option?" Short-form options allow frequent users who use the app on the command line to quickly specify things without a lot of typing. Long-form options allow maintainers of systems that use our app to easily understand what the options do without having to go to the documentation. Let's look at an example.

Suppose we've set up db_backup.rb to run nightly at 2 a.m. We've also set up our "end-of-iteration" backup to run on the first of the month at 2:30 a.m. We accomplish this by using cron, which is a common UNIX utility for running regularly scheduled commands. Suppose that Bob, a sysadmin who maintains the servers where we run our backups, wants to configure the system to perform automated maintenance on the first of the month. The first thing he'll do is look at cron's configuration to see what else is going on at the first of the month. He'll need to get a complete picture of what's been configured so he can decide how to get his job done. He'll see something like this:

```
00 02 * * 1-5  db_backup.rb -u dave.c -p P455w0rd small_client
30 02 1 * *    db_backup.rb -i -u dave.c -p P455w0rd small_client
```

(If you aren't familiar with cron, the format for the earlier crontab is as follows: the first five values tell cron when to run the command. The numbers represent, in order, minute, hour, day of month, month, and day of week. An asterisk is the symbol for "all," so the first line tells db_backup.rb to run every weekday (1-5 as the fifth value) at 2 a.m. (the 00 and 02 as the first and second values, respectively). The second line tells cronto run our "end-of-iteration" backup at 2:30 a.m. on the first of the month.

Bob has never run db_backup.rb, and while he does understand that our dev team runs two types of backups (daily and "end of iteration"), the -i isn't going to mean anything to him. He'll have to find the documentation for db_backup.rb or go to the command line and run db_backup.rb --help. While we could have added a comment to the crontab entry, it's actually much clearer to use the long-form option:

```
00 02 * * 1-5  db_backup.rb -u dave.c -p P455w0rd small_client
30 02 1 * *    db_backup.rb --iteration -u dave.c -p P455w0rd small_client
```

Now Bob knows exactly what the second line is doing and why it's there. We could be even more conscientious and turn the long-form option into --end-of-iteration. Of course, we wouldn't change -i to -e; *i* is a good mnemonic for "iteration," which makes it a good name for the short-form version of the option.

This example illustrates the importance of good naming as well as the form of those names. This leads us to the following rules of thumb regarding naming your options:

- For short-form options, use a mnemonic that frequent users will easily remember. Mnemonics are a well-known learning technique that is common in command-line application user interfaces.
- Always provide a long-form option and use it in configuration or other scripts. This allows us to create very specific and readable command-line invocations inside configuration files or other apps. We saw how it helped Bob understand what was going on in cron's configuration; we want everyone to have this experience maintaining systems that use our apps.
- Name long-form options explicitly and completely; they are designed to be *read* more so than written. Users aren't going to frequently type out the long-form options, so it's best to err on the side of clarity.

Let's follow these guidelines and enhance db_backup.rb by renaming --iteration and adding long-form options for -u and -p:

make_easy_possible/db_backup/bin/db_backup.rb

```
opts.on("-i",
        "--end-of-iteration",
        'Indicate that this backup is an "iteration" backup') do
  options[:iteration] = true
end
opts.on("-u USER",
        "--username",
        "Database username, in first.last format") do |user|
  options[:user] = user
end

opts.on("-p PASSWORD",
        "--password",
       "Database password") do |password|
  options[:password] = password
end
```

Now our crontab is easy to read by just about anyone who sees it:

```
00 02 * * 1-5  db_backup.rb --username=dave.c \
                            --password=P455w0rd small_client
30 02 1 * *    db_backup.rb --end-of-iteration \
                            --username=dave.c \
                            --password=P455w0rd small_client
```

While we should always provide a long-form option, the converse isn't true; some options should have long-form names only and *not* short-form versions. The reason for this is to support our second guiding principle: while we want uncommon tasks or features to be possible, we don't want to make them easy.

The reason for this is twofold. First, there's the practical limitation of having only twenty-six letters and ten digits available for short-form option names (or fifty-two if you include uppercase, although using short-form options as mnemonics makes it hard to have both an `-a` and an `-A` that the user will remember). Any new short-form option "uses up" one of these characters. Since we want our short-form options to be mnemonics, we have to ask ourselves, "Is this new option worthy of using one of those letters?"

Second, there is a usability concern with using short-form options. The existence of a short-form option signals to the user that that option is common and encouraged. The absence of a short-form option signals the opposite—that using it is unusual and possibly dangerous. You might think that unusual or dangerous options should simply be omitted, but we want our application to be as flexible as is reasonable. We want to guide our users to do things safely and correctly, but we also want to respect that they know what they're doing if they want to do something unusual or dangerous.

Let's put this to use. `db_backup.rb` compresses the database backup file, but suppose a user didn't want to perform the compression? Currently, they have no way to do that. We're happy to add this feature, but it's not something we want to encourage; database backup files are quite large and can quickly fill the disk. So we allow this feature to be enabled with a long-form option only.

Let's add a new switch, using only a long-form name, and see how the app's help output affects the user experience:

make_easy_possible/db_backup/bin/db_backup.rb

```
options = {
  :gzip => true
}
option_parser = OptionParser.new do |opts|
  # ...
  opts.on("--no-gzip","Do not compress the backup file") do
    options[:gzip] = false
  end
end
```

```
$ ./db_backup.rb --help
Backup one or more MySQL databases

Usage: db_backup.rb [options] database_name

  -i, --end-of-iteration   Indicate that this backup is an "iteration" backup
  -u, --username USER      Database username, in first.last format
  -p, --password PASSWORD  Database password
      --no-gzip            Do not compress the backup file
```

Notice how the documentation for `--no-gzip` is set apart visually from the other options? This is a subtle clue to the user that this option is not to be frequently used. For apps with a lot of options, this visual distinction is a great way for users to quickly scan the output of `--help` to see which common options they might need: those with a short-form name.

Naming Commands in a Command Suite

For command suites, the names of commands should follow the same guidelines: all commands should have a clear, concise name. Common commands can have shorter mnemonics if that makes sense. For example, many command-line users are familiar with the `ls` command, and it is a mnemonic of sorts for "list." We can take advantage of this in our task-management app `todo` and provide `ls` as an alias for the `list` command. Since `todo` is a GLI-based app, we simply pass an `Array` of `Symbol` to `command` instead of just a `Symbol`:

make_easy_possible/todo/bin/todo

```
command [:list,:ls] do |c|

  # ...

end
```

Now frequent users can do `todo ls`:

```
$ todo help
usage: todo [global options] command [command options]

Version: 0.0.1

Global Options:
    -f, --filename=todo_file - Path to the todo file (default: ~/.todo.txt)
    --force-tty              -

Commands:
    done     - Complete a task
    help     - Shows list of commands or help for one command
    list, ls - List tasks
    new      - Create a new task in the task list
```

Naming can be difficult, but our guidelines around mnemonics, descriptive long-form options, and judicious use of short-form names can help. Now it's time to go one level deeper into a command-line app and talk about the default values for flags and arguments. An example of what we mean is the `--filename` global option to our to-do list management app, `todo`. Why did we choose `~/.todo.txt` as a default; *should* we have chosen a default, and is that the best default value we could've chosen?