Extracted from:

Build Awesome Command-Line Applications in Ruby

Control Your Computer, Simplify Your Life

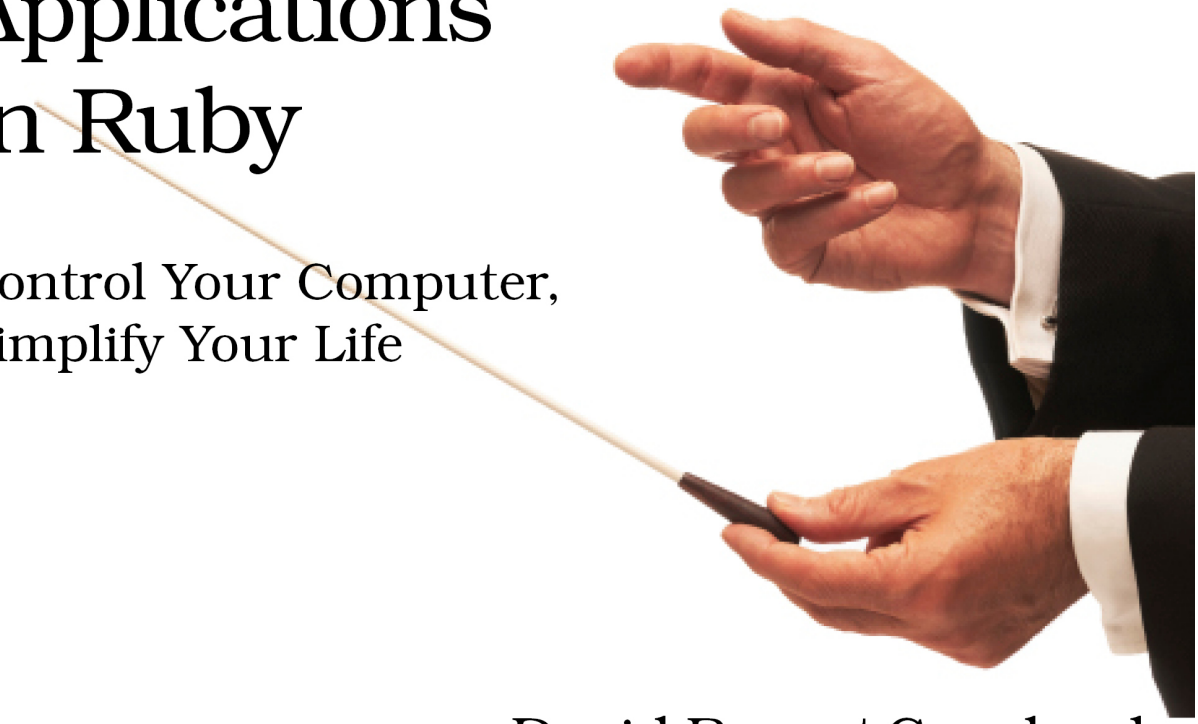
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Build Awesome Command-Line Applications in Ruby

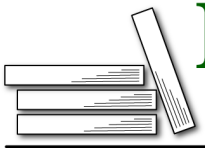
Control Your Computer, Simplify Your Life



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

2.1 Understanding the Command Line: Options, Arguments, and Commands

To tell a command-line application how to do its work, you typically need to enter more than just the name of its executable. For example, we must tell grep which files we want it to search. The database backup app, db_backup.rb, that we introduced in the previous chapter needs a username and password and a database name in order to do its work. The primary way to give an app the information it needs is via *options* and *arguments*, as depicted in Figure 1, *Basic parts of a command-line app invocation*, on page 6. Note that this format isn't imposed by the operating system but is based on the GNU standard for command-line apps.[1] Before we learn how to make a command-line interface that can parse and accept options and arguments, we need to delve a bit deeper into their idioms and conventions. We'll start with options and move on to arguments. After that, we'll discuss *commands*, which are a distinguishing feature of command suites.

Options

Options are the way in which a user modifies the behavior of your app. Consider the two invocations of ls shown here. In the first, we omit options and see the default behavior. In the second, we use the -l option to modify the listing format.

```
$ ls
one.jpg    two.jpg     three.jpg
$ ls -l
-rw-r--r--   1 davec  staff   14005 Jul 13 19:06 one.jpg
-rw-r--r--   1 davec  staff   14005 Jul 11 13:06 two.jpg
-rw-r--r--   1 davec  staff   14005 Jun 10 09:45 three.jpg
```

Options come in two forms: long and short.

Short-form options

Short-form options are preceded by a dash and are only one character long, for example -l. Short-form options can be combined after a single dash, as in the following example. For example, the following two lines of code produce exactly the same result:

```
ls -l -a -t
```

```
ls -lat
```

1. http://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html

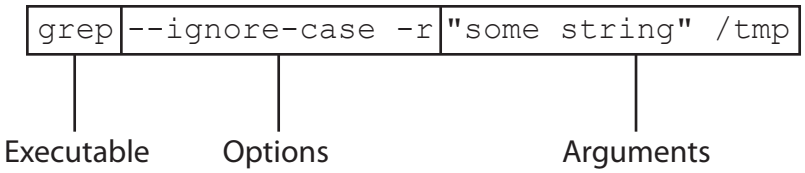


Figure 1—Basic parts of a command-line app invocation

Long-form options

Long-form options are preceded by two dashes and, strictly speaking, consist of two or more characters. However, long-form options are usually complete words (or even several words, separated by dashes). The reason for this is to be explicit about what the option means; with a short-form option, the single letter is often a mnemonic. With long-form options, the convention is to spell the word for what the option does. In the command `curl --basic http://www.google.com`, for example, `--basic` is a single, long-form option. Unlike short options, long options cannot be combined; each must be entered separately, separated by spaces on the command line.

Command-line options can be one of two types: *switches*, which are used to turn options on and off and do not take arguments, and *flags*, which take arguments, as shown in Figure 2, *A command-line invocation with switches and flags*, on page 7. Flags typically require arguments but, strictly speaking, don't need to do so. They just need to accept them. We'll talk more about this in Chapter 5, *Delight Casual Users*, on page ?.

Typically, if a switch is in the long-form (for example `--foo`), which turns "on" some behavior, there is also another switch preceded with `no-` (for example `--no-foo`) that turns "off" the behavior.

Finally, long-form flags take their argument via an equal sign, whereas in the short form of a flag, an equal sign is typically not used. For example, the `curl` command, which makes HTTP requests, provides both short-form and long-form flags to specify an HTTP request method: `-X` and `--request`, respectively. The following example invocations show how to properly pass arguments to those flags:

```
curl -X POST http://www.google.com

curl --request=POST http://www.google.com
```

Although some apps do not require an equal sign between a long-form flag and its argument, your apps should always accept an equal sign, because

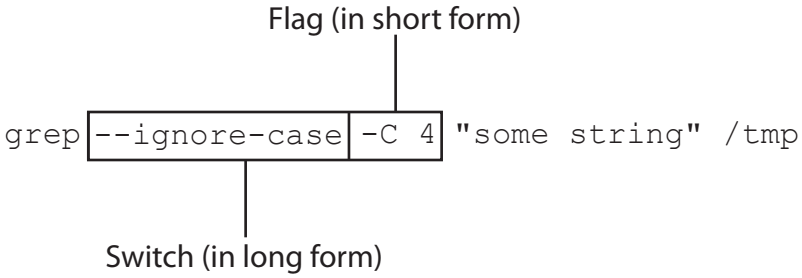


Figure 2—A command-line invocation with switches and flags

this is the idiomatic way of giving a flag its argument. We'll see later in this chapter that the tools provided by Ruby and its open source ecosystem make it easy to ensure your app follows this convention.

Arguments

As shown in Figure 1, *Basic parts of a command-line app invocation*, on page 6, arguments are the elements of a command line that aren't options. Rather, arguments represent the objects that the command-line app will operate on. Typically, these objects are file or directory names, but this depends on the app. We might design our database backup app to treat the arguments as the names of the databases to back up.

Not all command-line apps take arguments, while others take an arbitrary number of them. Typically, if your app operates on a file, it's customary to accept any number of filenames as arguments and to operate on them one at a time.

Commands

Figure 1, *Basic parts of a command-line app invocation*, on page 6 shows a diagram of a basic command-line invocation with the main elements of the command line labeled.

For simple command-line applications, options and arguments are all you need to create an interface that users will find easy to use. Some apps, however, are a bit more complicated. Consider git, the popular distributed version control system. git packs a lot of functionality. It can add files to a repository, send them to a remote repository, examine a repository, or fetch changes from another user's repository. Originally, git was packaged as a collection of individual command-line apps. For example, to commit changes, you would execute the git-commit application. To fetch files from a remote repository, you

would execute git-fetch. While each command provided its own options and arguments, there was some overlap.

For example, almost every git command provided a --no-pager option, which told git *not* to send output through a pager like more. Under the covers, there was a lot of shared code as well. Eventually, git was repackaged as a single executable that operated as a *command suite*. Instead of running git-commit, you run git commit. The single-purpose command-line app git-commit now becomes a *command* to the new command-suite app, git.

A command in a command-line invocation isn't like an option or an argument; it has a more specific meaning. A command is how you specify the action to take from among a potentially large or complex set of available actions. If you look around the Ruby ecosystem, you'll see that the use of command suites is quite common. gem, rails, and bundler are all types of command suites.

Figure 3, *Basic parts of a command-suite invocation*, on page 9 shows a command-suite invocation, with the command's position on the command line highlighted.

You won't always design your app as a command suite; only if your app is complex enough that different behaviors are warranted will you use this style of interface. Further, if you *do* decide to design your app as a command suite, your app should *require* a command (we'll talk about how your app should behave when the command is omitted in Chapter 3, *Be Helpful*, on page ?).

The command names in your command suite should be short but expressive, with short forms available for commonly used or lengthier commands. For example, Subversion, the version control system used by many developers, accepts the short-form co in place of its checkout command.

A command suite can still accept options; however, their position on the command line affects how they are interpreted.

Global options
: Options that you enter before the command are known as *global options*. Global options affect the global behavior of an app and can be used with any command in the suite. Recall our discussion of the --no-pager option for git? This option affects all of git's commands. We know this because it comes before the command on the command line, as shown in Figure 3, *Basic parts of a command-suite invocation*, on page 9.

Command options
: Options that follow a command are known as *command-specific options* or simply command options. These options have meaning only in the

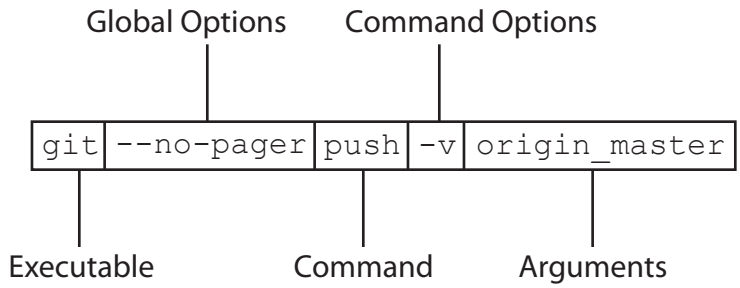


Figure 3—Basic parts of a command-suite invocation

context of their command. Note that they can also have the same names as global options. For example, if our to-do list app took a global option `-f` to indicate where to find the to-do list's file, the `list` command might also take an `-f` to indicate a "full" listing.

The command-line invocation would be `todo -f ~/my_todos.txt list -f`. Since the first `-f` comes before the command and is a global option, we won't confuse it for the second `-f`, which is a command option.

Most command-line apps follow the conventions we've just discussed. If your app follows them as well, users will have an easier time learning and using your app's interface. For example, if your app accepts long-form flags but doesn't allow the use of an equal sign to separate the flag from its argument, users will be frustrated.

The good news is that it's very easy to create a Ruby app that follows all of the conventions we've discussed in this section. We'll start by enhancing our Chapter 1 database backup app from Chapter 1, *Have a Clear and Concise Purpose*, on page ? to demonstrate how to make an easy-to-use, conventional command-line application using `OptionParser`. After that, we'll use GLI to enhance our to-do list app, creating an idiomatic command suite that's easy for our users to use and easy for us to implement.