Extracted from:

Powerful Command-Line Applications in Go

Build Fast and Maintainable Tools

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Powerful Command-Line Applications in Go

Build Fast and Maintainable Tools

Ricardo Gerardi
edited by Brian P. Hogan

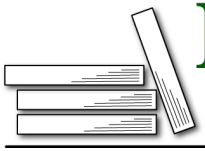
Powerful Command-Line Applications in Go

Build Fast and Maintainable Tools

Ricardo Gerardi

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Brian P. Hogan
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-696-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2021

To my beloved wife Kassia. My best friend, my greatest supporter. Sadly she passed away due to cancer before seeing this book completed.

To my four incredible daughters Gisele, Livia, Elena, and Alice. The reason for everything.

CHAPTER 1

Your First Command-Line Program in Go

Whether you're looking to automate a task, analyze data, parse logs, talk to network services, or address other requirements, writing your own command-line tool may be the fastest—and perhaps the most fun—way to achieve your goal. Go is a modern programming language that combines the reliability of compiled languages with the ease of use and speed of dynamically typed languages. It makes writing cross-platform command-line applications more approachable while providing the features required to ensure these tools are well designed and tested.

Before you dive into more complex programs that read and write files, parse data files, and communicate over networks, you'll create a word counter program that will give you an idea of how to build and test a command-line application using Go. You'll start with a basic implementation, add some features, and explore test-driven development along the way. When you're done, you'll have a functional word counter program and a better understanding of how to build more complex apps.

Throughout the book you'll develop other CLI applications to explore more advanced concepts.

Building the Basic Word Counter

Let's create a tool that counts the number of words or lines provided as input using the *standard input* (STDIN) connection. By default, this tool will count the number of words, unless it receives the `-l` flag, in which case it'll count the number of lines instead.

We'll start by creating the basic implementation. This version reads data from STDIN and displays the number of words. We'll eventually add more features,

but this initial version will let you get comfortable with the code for a Go-based command-line application.

Before you dive into writing code for the word counter, let's set up a project directory. In your home directory, create the subdirectory pragprog.com/rggo/firstProgram/wc and switch to it:

```
$ mkdir -p $HOME/pragprog.com/rggo/firstProgram/wc
$ cd $HOME/pragprog.com/rggo/firstProgram/wc
```

Go programs are composed of *packages*. A package consists of one or more Go source code files with code that can be combined into executable programs or libraries.

Starting with Go 1.11, you can combine one or more packages into Go modules. Modules are a new Go standard for grouping related packages into a single unit that can be versioned together. Modules enable consistent dependency management for your Go applications. For more information about Go modules, consult the official wiki page.[1]

Initialize a new Go module for your project:

```
$ go mod init pragprog.com/rggo/firstProgram/wc
go: creating new go.mod: module pragprog.com/rggo/firstProgram/wc
```

You create an executable program in Go by defining a package named main that contains a function called main(). This function takes no arguments and returns no values. It serves as the entry point for your program.

```
package main

func main() {
 «main contents»
}
```

Although not a requirement, by convention, the main package is usually defined in a file named main.go. You'll use this convention throughout this book.

Code Example File Path

For brevity, the code example path omits the root directory $HOME/pragprog.com/rggo. For example, in the following code sample, the code path starts at firstProgram/wc.

Create the file main.go using your favorite text editor. Add the package main definition to the top of the file like this:

1. github.com/golang/go/wiki/Modules

firstProgram/wc/main.go
```
package main
```

Next, add the import section to bring in the libraries you'll use to read data from STDIN and print results out.

firstProgram/wc/main.go
```
import (
  "bufio"
  "fmt"
  "io"
  "os"
)
```

For this tool, you import the bufio package to read text, the fmt package to print formatted output, the io package which provides the io.Reader interface, and the os package so you can use operating system resources.

Your word counter will have two functions: main() and count(). The main() function is the starting point of the program. All Go programs that will be compiled into executable files require this function. Create this function by adding the following code into your main.go file. This function will call the count() function and print out that function's return value using the fmt.Println() function:

firstProgram/wc/main.go
```
func main() {
  // Calling the count function to count the number of words
  // received from the Standard Input and printing it out
  fmt.Println(count(os.Stdin))
}
```

Next, define the count() function, which will perform the actual counting of the words. This function receives a single input argument: an io.Reader *interface*. You'll learn more about Go interfaces in Chapter 2, Interacting with Your Users, on page ?. For now, think of an io.Reader as any Go type from which you can read data. In this case, the function will receive the contents of the STDIN to process.

firstProgram/wc/main.go
```
func count(r io.Reader) int {
  // A scanner is used to read text from a Reader (such as files)
  scanner := bufio.NewScanner(r)

  // Define the scanner split type to words (default is split by lines)
  scanner.Split(bufio.ScanWords)

  // Defining a counter
  wc := 0
```

```
  // For every word scanned, increment the counter
  for scanner.Scan() {
    wc++
  }

  // Return the total
  return wc
}
```

The count() function uses the NewScanner() function from the bufio package to create a new scanner. A scanner is a convenient way of reading data delimited by spaces or new lines. By default, a scanner reads lines of data, so we instruct the scanner to read words instead by setting the Split() function of the scanner to bufio.ScanWords(). We then define a variable, wc, to hold the word count and increment it by looping through each token using the scanner.Scan() function and adding 1 to the counter each time. We then return the word count.

In this example, for simplicity's sake, we are ignoring the error that may be generated during the scanning. In your code, always check for errors. You'll learn more about dealing with errors in the context of a command-line tool in Creating the Initial To-Do Command-Line Tool, on page ?.

You've completed the basic implementation of the word count tool. Save the file main.go with your changes. Next, you'll write tests to ensure this implementation works the way you expect it to.

Testing the Basic Word Counter

Go lets you test your code automatically without requiring external tools or frameworks. You'll learn more about how to test your command-line applications throughout the book. Right now, let's write a basic test for the word counter to ensure that it correctly counts the words in the given input.

Create a file called main_test.go in the same directory as your main.go file. Include the following content, which defines a testing function that tests the count() function you've already defined in the main program:

firstProgram/wc/main_test.go

```
package main

import (
  "bytes"
  "testing"
)

// TestCountWords tests the count function set to count words
func TestCountWords(t *testing.T) {
  b := bytes.NewBufferString("word1 word2 word3 word4\n")
```

```
  exp := 4

  res := count(b)

  if res != exp {
    t.Errorf("Expected %d, got %d instead.\n", exp, res)
  }
}
```

This test file contains a single test called TestCountWords(). In this test, we create a new buffer of bytes from a string containing four words and pass the buffer into the count() function. If this function returns anything other than 4, the test doesn't pass and we raise an error that shows what we expected and what we actually got instead.

To execute the test, use the go test tool like this:

```
$ ls
go.mod  main.go  main_test.go
$ go test -v
=== RUN   TestCountWords
--- PASS: TestCountWords (0.00s)
PASS
ok      pragprog.com/rggo/firstProgram/wc        0.002s
```

The test passes, so you can compile the program with go build. You'll learn more about the different options you can use to build Go programs in Chapter 11, Distributing Your Tool, on page ?. For now, build your command-line tool like this:

```
$ go build
```

This creates the wc executable in the current directory:

```
$ ls
go.mod  main.go  main_test.go  wc
```

Test the program out by passing it an input string:

```
$ echo "My first command line tool with Go" | ./wc
7
```

The program works as expected. Let's add the ability to count lines to this tool.