The Go Programming Language

Alan A. A. Donovan
Brian W. Kernighan

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Contents

1

Tutorial

This chapter is a tour of the basic components of Go. We hope to provide enough information and examples to get you off the ground and doing useful things as quickly as possible. The examples here, and indeed in the whole book, are aimed at tasks that you might have to do in the real world. In this chapter we'll try to give you a taste of the diversity of programs that one might write in Go, ranging from simple file processing and a bit of graphics to concurrent Internet clients and servers. We certainly won't explain everything in the first chapter, but studying such programs in a new language can be an effective way to get started.

When you're learning a new language, there's a natural tendency to write code as you would have written it in a language you already know. Be aware of this bias as you learn Go and try to avoid it. We've tried to illustrate and explain how to write good Go, so use the code here as a guide when you're writing your own.

1.1. Hello, World

We'll start with the now-traditional "hello, world" example, which appears at the beginning of *The C Programming Language*, published in 1978. C is one of the most direct influences on Go, and "hello, world" illustrates a number of central ideas.

gopl.io/ch1/helloworld

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

Go is a compiled language. The Go toolchain converts a source program and the things it depends on into instructions in the native machine language of a computer. These tools are accessed through a single command called `go` that has a number of subcommands. The simplest of these subcommands is `run`, which compiles the source code from one or more source files whose names end in `.go`, links it with libraries, then runs the resulting executable file. (We will use `$` as the command prompt throughout the book.)

```
$ go run helloworld.go
```

Not surprisingly, this prints

```
Hello, 世界
```

Go natively handles Unicode, so it can process text in all the world's languages.

If the program is more than a one-shot experiment, it's likely that you would want to compile it once and save the compiled result for later use. That is done with `go build`:

```
$ go build helloworld.go
```

This creates an executable binary file called `helloworld` that can be run any time without further processing:

```
$ ./helloworld
Hello, 世界
```

We have labeled each significant example as a reminder that you can obtain the code from the book's source code repository at `gopl.io`:

gopl.io/ch1/helloworld

If you run `go get gopl.io/ch1/helloworld`, it will fetch the source code and place it in the corresponding directory. There's more about this topic in Section 2.6 and Section 10.7.

Let's now talk about the program itself. Go code is organized into packages, which are similar to libraries or modules in other languages. A package consists of one or more `.go` source files in a single directory that define what the package does. Each source file begins with a `package` declaration, here `package main`, that states which package the file belongs to, followed by a list of other packages that it imports, and then the declarations of the program that are stored in that file.

The Go standard library has over 100 packages for common tasks like input and output, sorting, and text manipulation. For instance, the `fmt` package contains functions for printing formatted output and scanning input. `Println` is one of the basic output functions in `fmt`; it prints one or more values, separated by spaces, with a newline character at the end so that the values appear as a single line of output.

Package `main` is special. It defines a standalone executable program, not a library. Within package `main` the *function* `main` is also special—it's where execution of the program begins. Whatever `main` does is what the program does. Of course, `main` will normally call upon functions in other packages to do much of the work, such as the function `fmt.Println`.

We must tell the compiler what packages are needed by this source file; that's the role of the `import` declaration that follows the `package` declaration. The "hello, world" program uses only one function from one other package, but most programs will import more packages.

You must import exactly the packages you need. A program will not compile if there are missing imports or if there are unnecessary ones. This strict requirement prevents references to unused packages from accumulating as programs evolve.

The `import` declarations must follow the `package` declaration. After that, a program consists of the declarations of functions, variables, constants, and types (introduced by the keywords `func`, `var`, `const`, and `type`); for the most part, the order of declarations does not matter. This program is about as short as possible since it declares only one function, which in turn calls only one other function. To save space, we will sometimes not show the `package` and `import` declarations when presenting examples, but they are in the source file and must be there to compile the code.

A function declaration consists of the keyword `func`, the name of the function, a parameter list (empty for `main`), a result list (also empty here), and the body of the function—the statements that define what it does—enclosed in braces. We'll take a closer look at functions in Chapter 5.

Go does not require semicolons at the ends of statements or declarations, except where two or more appear on the same line. In effect, newlines following certain tokens are converted into semicolons, so where newlines are placed matters to proper parsing of Go code. For instance, the opening brace `{` of the function must be on the same line as the end of the `func` declaration, not on a line by itself, and in the expression `x + y`, a newline is permitted after but not before the `+` operator.

Go takes a strong stance on code formatting. The `gofmt` tool rewrites code into the standard format, and the `go` tool's `fmt` subcommand applies `gofmt` to all the files in the specified package, or the ones in the current directory by default. All Go source files in the book have been run through `gofmt`, and you should get into the habit of doing the same for your own code. Declaring a standard format by fiat eliminates a lot of pointless debate about trivia and, more importantly, enables a variety of automated source code transformations that would be infeasible if arbitrary formatting were allowed.

Many text editors can be configured to run `gofmt` each time you save a file, so that your source code is always properly formatted. A related tool, `goimports`, additionally manages the insertion and removal of import declarations as needed. It is not part of the standard distribution but you can obtain it with this command:

```
$ go get golang.org/x/tools/cmd/goimports
```

For most users, the usual way to download and build packages, run their tests, show their documentation, and so on, is with the `go` tool, which we'll look at in Section 10.7.

1.2. Command-Line Arguments

Most programs process some input to produce some output; that's pretty much the definition of computing. But how does a program get input data on which to operate? Some programs generate their own data, but more often, input comes from an external source: a file, a network connection, the output of another program, a user at a keyboard, command-line arguments, or the like. The next few examples will discuss some of these alternatives, starting with command-line arguments.

The `os` package provides functions and other values for dealing with the operating system in a platform-independent fashion. Command-line arguments are available to a program in a variable named `Args` that is part of the `os` package; thus its name anywhere outside the `os` package is `os.Args`.

The variable `os.Args` is a *slice* of strings. Slices are a fundamental notion in Go, and we'll talk a lot more about them soon. For now, think of a slice as a dynamically sized sequence `s` of array elements where individual elements can be accessed as `s[i]` and a contiguous subsequence as `s[m:n]`. The number of elements is given by `len(s)`. As in most other programming languages, all indexing in Go uses *half-open* intervals that include the first index but exclude the last, because it simplifies logic. For example, the slice `s[m:n]`, where 0 ≤ `m` ≤ `n` ≤ `len(s)`, contains `n-m` elements.

The first element of `os.Args`, `os.Args[0]`, is the name of the command itself; the other elements are the arguments that were presented to the program when it started execution. A slice expression of the form `s[m:n]` yields a slice that refers to elements `m` through `n-1`, so the elements we need for our next example are those in the slice `os.Args[1:len(os.Args)]`. If `m` or `n` is omitted, it defaults to 0 or `len(s)` respectively, so we can abbreviate the desired slice as `os.Args[1:]`.

Here's an implementation of the Unix `echo` command, which prints its command-line arguments on a single line. It imports two packages, which are given as a parenthesized list rather than as individual `import` declarations. Either form is legal, but conventionally the list form is used. The order of imports doesn't matter; the `gofmt` tool sorts the package names into alphabetical order. (When there are several versions of an example, we will often number them so you can be sure of which one we're talking about.)

gopl.io/ch1/echo1

```
// Echo1 prints its command-line arguments.
package main

import (
    "fmt"
    "os"
)
```

```
func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

Comments begin with `//`. All text from a `//` to the end of the line is commentary for programmers and is ignored by the compiler. By convention, we describe each package in a comment immediately preceding its package declaration; for a `main` package, this comment is one or more complete sentences that describe the program as a whole.

The `var` declaration declares two variables `s` and `sep`, of type `string`. A variable can be initialized as part of its declaration. If it is not explicitly initialized, it is implicitly initialized to the *zero value* for its type, which is `0` for numeric types and the empty string `""` for strings. Thus in this example, the declaration implicitly initializes `s` and `sep` to empty strings. We'll have more to say about variables and declarations in Chapter 2.

For numbers, Go provides the usual arithmetic and logical operators. When applied to strings, however, the `+` operator *concatenates* the values, so the expression

```
sep + os.Args[i]
```

represents the concatenation of the strings `sep` and `os.Args[i]`. The statement we used in the program,

```
s += sep + os.Args[i]
```

is an *assignment statement* that concatenates the old value of `s` with `sep` and `os.Args[i]` and assigns it back to `s`; it is equivalent to

```
s = s + sep + os.Args[i]
```

The operator `+=` is an *assignment operator*. Each arithmetic and logical operator like `+` or `*` has a corresponding assignment operator.

The `echo` program could have printed its output in a loop one piece at a time, but this version instead builds up a string by repeatedly appending new text to the end. The string `s` starts life empty, that is, with value `""`, and each trip through the loop adds some text to it; after the first iteration, a space is also inserted so that when the loop is finished, there is one space between each argument. This is a quadratic process that could be costly if the number of arguments is large, but for `echo`, that's unlikely. We'll show a number of improved versions of `echo` in this chapter and the next that will deal with any real inefficiency.

The loop index variable `i` is declared in the first part of the `for` loop. The `:=` symbol is part of a *short variable declaration*, a statement that declares one or more variables and gives them appropriate types based on the initializer values; there's more about this in the next chapter.

The increment statement `i++` adds 1 to `i`; it's equivalent to `i += 1` which is in turn equivalent to `i = i + 1`. There's a corresponding decrement statement `i--` that subtracts 1. These are statements, not expressions as they are in most languages in the C family, so `j = i++` is illegal, and they are postfix only, so `--i` is not legal either.

The `for` loop is the only loop statement in Go. It has a number of forms, one of which is illustrated here:

```
for initialization; condition; post {
    // zero or more statements
}
```

Parentheses are never used around the three components of a `for` loop. The braces are mandatory, however, and the opening brace must be on the same line as the *`post`* statement.

The optional *`initialization`* statement is executed before the loop starts. If it is present, it must be a *simple statement*, that is, a short variable declaration, an increment or assignment statement, or a function call. The *`condition`* is a boolean expression that is evaluated at the beginning of each iteration of the loop; if it evaluates to `true`, the statements controlled by the loop are executed. The *`post`* statement is executed after the body of the loop, then the condition is evaluated again. The loop ends when the condition becomes false.

Any of these parts may be omitted. If there is no *`initialization`* and no *`post`*, the semicolons may also be omitted:

```
// a traditional "while" loop
for condition {
    // ...
}
```

If the condition is omitted entirely in any of these forms, for example in

```
// a traditional infinite loop
for {
    // ...
}
```

the loop is infinite, though loops of this form may be terminated in some other way, like a `break` or `return` statement.

Another form of the `for` loop iterates over a *range* of values from a data type like a string or a slice. To illustrate, here's a second version of `echo`:

gopl.io/ch1/echo2

```
// Echo2 prints its command-line arguments.
package main

import (
    "fmt"
    "os"
)
```

```
func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

In each iteration of the loop, `range` produces a pair of values: the index and the value of the element at that index. In this example, we don't need the index, but the syntax of a `range` loop requires that if we deal with the element, we must deal with the index too. One idea would be to assign the index to an obviously temporary variable like `temp` and ignore its value, but Go does not permit unused local variables, so this would result in a compilation error.

The solution is to use the *blank identifier*, whose name is _ (that is, an underscore). The blank identifier may be used whenever syntax requires a variable name but program logic does not, for instance to discard an unwanted loop index when we require only the element value. Most Go programmers would likely use `range` and _ to write the `echo` program as above, since the indexing over `os.Args` is implicit, not explicit, and thus easier to get right.

This version of the program uses a short variable declaration to declare and initialize `s` and `sep`, but we could equally well have declared the variables separately. There are several ways to declare a string variable; these are all equivalent:

```
s := ""
var s string
var s = ""
var s string = ""
```

Why should you prefer one form to another? The first form, a short variable declaration, is the most compact, but it may be used only within a function, not for package-level variables. The second form relies on default initialization to the zero value for strings, which is `""`. The third form is rarely used except when declaring multiple variables. The fourth form is explicit about the variable's type, which is redundant when it is the same as that of the initial value but necessary in other cases where they are not of the same type. In practice, you should generally use one of the first two forms, with explicit initialization to say that the initial value is important and implicit initialization to say that the initial value doesn't matter.

As noted above, each time around the loop, the string `s` gets completely new contents. The `+=` statement makes a new string by concatenating the old string, a space character, and the next argument, then assigns the new string to `s`. The old contents of `s` are no longer in use, so they will be garbage-collected in due course.

If the amount of data involved is large, this could be costly. A simpler and more efficient solution would be to use the `Join` function from the `strings` package:

gopl.io/ch1/echo3

```
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

Finally, if we don't care about format but just want to see the values, perhaps for debugging, we can let `Println` format the results for us:

```
fmt.Println(os.Args[1:])
```

The output of this statement is like what we would get from `strings.Join`, but with surrounding brackets. Any slice may be printed this way.

Exercise 1.1: Modify the `echo` program to also print `os.Args[0]`, the name of the command that invoked it.

Exercise 1.2: Modify the `echo` program to print the index and value of each of its arguments, one per line.

Exercise 1.3: Experiment to measure the difference in running time between our potentially inefficient versions and the one that uses `strings.Join`. (Section 1.6 illustrates part of the `time` package, and Section 11.4 shows how to write benchmark tests for systematic performance evaluation.)

1.3. Finding Duplicate Lines

Programs for file copying, printing, searching, sorting, counting, and the like all have a similar structure: a loop over the input, some computation on each element, and generation of output on the fly or at the end. We'll show three variants of a program called `dup`; it is partly inspired by the Unix `uniq` command, which looks for adjacent duplicate lines. The structures and packages used are models that can be easily adapted.

The first version of `dup` prints each line that appears more than once in the standard input, preceded by its count. This program introduces the `if` statement, the `map` data type, and the `bufio` package.

gopl.io/ch1/dup1

```
// Dup1 prints the text of each line that appears more than
// once in the standard input, preceded by its count.
package main

import (
    "bufio"
    "fmt"
    "os"
)
```

```
func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTE: ignoring potential errors from input.Err()
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

As with `for`, parentheses are never used around the condition in an `if` statement, but braces are required for the body. There can be an optional `else` part that is executed if the condition is false.

A *map* holds a set of key/value pairs and provides constant-time operations to store, retrieve, or test for an item in the set. The key may be of any type whose values can compared with `==`, strings being the most common example; the value may be of any type at all. In this example, the keys are strings and the values are `int`s. The built-in function `make` creates a new empty map; it has other uses too. Maps are discussed at length in Section 4.3.

Each time `dup` reads a line of input, the line is used as a key into the map and the corresponding value is incremented. The statement `counts[input.Text()]++` is equivalent to these two statements:

```
line := input.Text()
counts[line] = counts[line] + 1
```

It's not a problem if the map doesn't yet contain that key. The first time a new line is seen, the expression `counts[line]` on the right-hand side evaluates to the zero value for its type, which is `0` for `int`.

To print the results, we use another `range`-based `for` loop, this time over the `counts` map. As before, each iteration produces two results, a key and the value of the map element for that key. The order of map iteration is not specified, but in practice it is random, varying from one run to another. This design is intentional, since it prevents programs from relying on any particular ordering where none is guaranteed.

Onward to the `bufio` package, which helps make input and output efficient and convenient. One of its most useful features is a type called `Scanner` that reads input and breaks it into lines or words; it's often the easiest way to process input that comes naturally in lines.

The program uses a short variable declaration to create a new variable `input` that refers to a `bufio.Scanner`:

```
input := bufio.NewScanner(os.Stdin)
```

The scanner reads from the program's standard input. Each call to `input.Scan()` reads the next line and removes the newline character from the end; the result can be retrieved by calling `input.Text()`. The `Scan` function returns `true` if there is a line and `false` when there is no more input.

The function `fmt.Printf`, like `printf` in C and other languages, produces formatted output from a list of expressions. Its first argument is a format string that specifies how subsequent arguments should be formatted. The format of each argument is determined by a conversion character, a letter following a percent sign. For example, `%d` formats an integer operand using decimal notation, and `%s` expands to the value of a string operand.

`Printf` has over a dozen such conversions, which Go programmers call *verbs*. This table is far from a complete specification but illustrates many of the features that are available:

`%d`	decimal integer
`%x, %o, %b`	integer in hexadecimal, octal, binary
`%f, %g, %e`	floating-point number: `3.141593 3.141592653589793 3.141593e+00`
`%t`	boolean: `true` or `false`
`%c`	rune (Unicode code point)
`%s`	string
`%q`	quoted string `"abc"` or rune `'c'`
`%v`	any value in a natural format
`%T`	type of any value
`%%`	literal percent sign (no operand)

The format string in `dup1` also contains a tab `\t` and a newline `\n`. String literals may contain such *escape sequences* for representing otherwise invisible characters. `Printf` does not write a newline by default. By convention, formatting functions whose names end in `f`, such as `log.Printf` and `fmt.Errorf`, use the formatting rules of `fmt.Printf`, whereas those whose names end in `ln` follow `Println`, formatting their arguments as if by `%v`, followed by a newline.

Many programs read either from their standard input, as above, or from a sequence of named files. The next version of `dup` can read from the standard input or handle a list of file names, using `os.Open` to open each one:

gopl.io/ch1/dup2

```
// Dup2 prints the count and text of lines that appear more than once
// in the input.  It reads from stdin or from a list of named files.
package main

import (
    "bufio"
    "fmt"
    "os"
)
```

```
func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}

func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTE: ignoring potential errors from input.Err()
}
```


The function `os.Open` returns two values. The first is an open file (`*os.File`) that is used in subsequent reads by the `Scanner`.

The second result of `os.Open` is a value of the built-in `error` type. If `err` equals the special built-in value `nil`, the file was opened successfully. The file is read, and when the end of the input is reached, `Close` closes the file and releases any resources. On the other hand, if `err` is not `nil`, something went wrong. In that case, the error value describes the problem. Our simple-minded error handling prints a message on the standard error stream using `Fprintf` and the verb `%v`, which displays a value of any type in a default format, and `dup` then carries on with the next file; the `continue` statement goes to the next iteration of the enclosing `for` loop.

In the interests of keeping code samples to a reasonable size, our early examples are intentionally somewhat cavalier about error handling. Clearly we must check for an error from `os.Open`; however, we are ignoring the less likely possibility that an error could occur while reading the file with `input.Scan`. We will note places where we've skipped error checking, and we will go into the details of error handling in Section 5.4.

Notice that the call to `countLines` precedes its declaration. Functions and other package-level entities may be declared in any order.

A map is a *reference* to the data structure created by make. When a map is passed to a function, the function receives a copy of the reference, so any changes the called function makes to the underlying data structure will be visible through the caller's map reference too. In our example, the values inserted into the counts map by countLines are seen by main.

The versions of dup above operate in a "streaming" mode in which input is read and broken into lines as needed, so in principle these programs can handle an arbitrary amount of input. An alternative approach is to read the entire input into memory in one big gulp, split it into lines all at once, then process the lines. The following version, dup3, operates in that fashion. It introduces the function ReadFile (from the io/ioutil package), which reads the entire contents of a named file, and strings.Split, which splits a string into a slice of substrings. (Split is the opposite of strings.Join, which we saw earlier.)

We've simplified dup3 somewhat. First, it only reads named files, not the standard input, since ReadFile requires a file name argument. Second, we moved the counting of the lines back into main, since it is now needed in only one place.

gopl.io/ch1/dup3

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

ReadFile returns a byte slice that must be converted into a string so it can be split by strings.Split. We will discuss strings and byte slices at length in Section 3.5.4.

Under the covers, `bufio.Scanner`, `ioutil.ReadFile`, and `ioutil.WriteFile` use the `Read` and `Write` methods of `*os.File`, but it's rare that most programmers need to access those lower-level routines directly. The higher-level functions like those from `bufio` and `io/ioutil` are easier to use.

Exercise 1.4: Modify `dup2` to print the names of all files in which each duplicated line occurs.

1.4. Animated GIFs

The next program demonstrates basic usage of Go's standard image packages, which we'll use to create a sequence of bit-mapped images and then encode the sequence as a GIF animation. The images, called *Lissajous figures*, were a staple visual effect in sci-fi films of the 1960s. They are the parametric curves produced by harmonic oscillation in two dimensions, such as two sine waves fed into the *x* and *y* inputs of an oscilloscope. Figure 1.1 shows some examples.

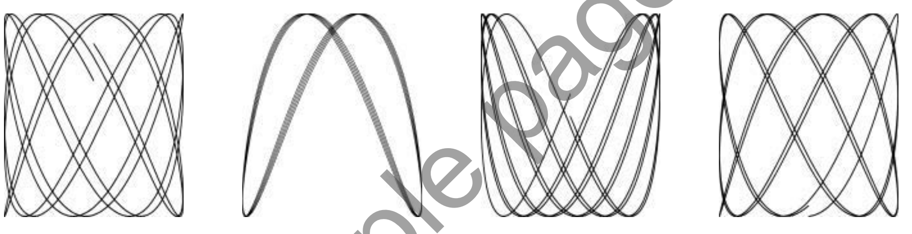

Figure 1.1. Four Lissajous figures.

There are several new constructs in this code, including `const` declarations, struct types, and composite literals. Unlike most of our examples, this one also involves floating-point computations. We'll discuss these topics only briefly here, pushing most details off to later chapters, since the primary goal right now is to give you an idea of what Go looks like and the kinds of things that can be done easily with the language and its libraries.

gopl.io/ch1/lissajous

```
// Lissajous generates GIF animations of random Lissajous figures.
package main

import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
)
```