Go Basics

# Resources

|  |  |
| --- | --- |
| <https://go.dev/doc/effective_go> | <https://go.dev/doc/> |
| <https://gobyexample.com/> | <https://gowebexamples.com/> |
| <https://go.dev/ref/spec> | <https://go.dev/ref/mod> |
| <https://go.dev/blog/> | <https://go.dev/blog/using-go-modules> |
| <https://pkg.go.dev/std> | <https://go.dev/ref/mem> |

# Basics

Let’s jump straight to some code:

package main

import "fmt"

func main() {

fmt.Println("Hello, Chinese letter")

}

* 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.**

$ go run **helloworld.go**

Hello, Chinese letter

* 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, Chinese letter**

* 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. We must tell the compiler what packages are needed by this source file.

You must import exactly the packages you need. A prog ram will not compile if there are missing imports or if there are unnecessary ones.

* + 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**.

**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.**

After the import section, a program consists of the **declarations of functions**, **variables**, **constants**, and **types** (introduce d by the key words **func, var, const, and type**); for the most part, the order of declarations does not matter.

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**.

you should get into the habit of running your code through gofmt. 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.

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

## 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**.

gopl.io/ch1/echo1

// Echo1 prints its commandline 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.
* When applied to strings, the + operator concatenates the values
* 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**;
* **i++ and i-- 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.**

Page 6 The for loop is the only loop…