



Programming in Go

Lesson 0: The Very Basics

Matt Holiday
24 February 2019
Cardinal Peak



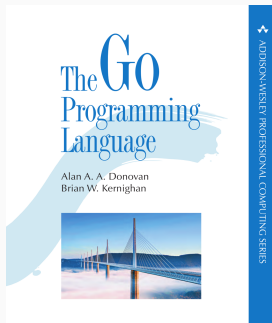
The Book

Hereinafter referred to as *GOPL*

I will be taking exercise material from this book

Amazon paper: \$28

informit.com PDF: \$19
(with coupon IUGD45)



“Anything with Brian Kernighan’s name on it is worth reading.”
— Matt Holiday

Hello, world!

What the simplest program looks like:

```
package main

import "fmt"

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

Hello, world!

What the simplest program looks like:

```
package main

import "fmt"

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

The program must have a `main` function to get started

The main function must live in package `main` because everything must be in some package

Anything we use from another package must be *imported*

Hello, world!

What the simplest program looks like:

```
package main

import "fmt"

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

The body of `main` is in curly braces `{}` with just a *function call* to print output to the console

Package `fmt` is where we find utilities for formatted output

`Println` prints its *arguments* and terminates the line

Hello, playground!

Simple programs run at the [Go playground](https://play.golang.org)



The screenshot shows a web browser window at play.golang.org. The page has a light blue header with the title "The Go Playground" and four buttons: "Run", "Format", "Imports", and "Share". On the right side of the header is an "About" button. The main area has a yellow background for the code editor. It contains a Go program with the following code:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello, playground!")
9 }
10
11
12
13
14
15
16
```

Below the code editor, the output of the program is displayed on a white background:

```
Hello, playground!
Program exited.
```

Hello, world!

A little more info

```
package main

import "fmt"

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

Notice that semicolons ‘;’ aren’t normally used in Go

But everything else here is needed, such as the parentheses () for the function call

Anything you leave out or misspell will cause an error

Hello, world!

Still more info

```
package main

import "fmt"

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

To call a function from a package, we must write the package name and then a dot '.' and then the function name

The function must have the correct number of arguments in order

Literal strings are text sequences in double quotes ""

More strings

Strings may have *escaped* characters

```
package main

import "fmt"

func main() {
    fmt.Println("A string with \"quotes\" inside")
}
```

The backslash `\` is used to *escape* a special character

A `\n` prints a new line, while `\t` prints a tab

Use two backslashes `\\` to print an actual backslash character

Declaring variables

This code example declares a variable and then assigns to it later

```
var x int  
  
x = len("a string")
```

But often it's more convenient to use a *short declaration* with the `:=` operator

```
x := len("a string")
```

In both cases `x` is an *integer* (it holds whole numbers)

The first example gave the *type* explicitly; in the second, the type is determined by what `len()` returns

Variable names

Names may contain letters & digits, and must begin with a letter

```
short  
shortName           // capitalize inner words
```

```
short_name          // not Go style  
SHORT_NAME          // ditto  
SHORTNAME           // ditto
```

```
name12              // OK  
12name             // not allowed
```

Integer types

Go has a variety of hardware-related integer types

- “unsized” (defaults to the machine’s natural wordsize):

`int`, `uint`

- on my Core i7 laptop, these are 64 bits in size
- on my Raspberry Pi, these are 32 bits in size

`int` is the default type for integers in Go, even lengths

- sized, signed:

`int8` `int16` `int32` `int64`

- sized, unsigned:

`uint8` `uint16` `uint32` `uint64` `uintptr`

Other types

The `bool` (boolean) type has two values: `false`, `true`

These values are **not** convertible to/from integers!

Integer values can't be combined unless they have the same type

```
var b bool
var x int16
var y int32
```

```
z := x + y           // compile error, type mismatch
```

```
z := int32(x) + y    // this is OK
```

```
z := int32(b) + y    // this is not allowed
```

Declarations are forever

Once a variable is declared, its type can't change

```
var x int = 12
```

```
x = "a string"    // compile error, type mismatch  
x = uint(12)      // ditto
```

This is also true when short declarations are used

```
y := 12           // defaults to type "int"  
y = 13           // OK  
  
y = uint(12)      // not OK, type mismatch  
y := 13           // not OK, already declared
```

The `:=` operator should not be confused with assignment (`=`)

String-related types

Types related to strings:

- `byte`: a synonym for `uint8`
- `rune`: a synonym for `int32` for characters
- `string`: an immutable sequence of “characters”
 - physically a sequence of `byte`
 - logically a sequence of `rune`

Runes (characters) are enclosed in single quotes: `'a'`

“Raw” strings use backtick quotes: ``string with "quotes" ``

They also don't evaluate escape characters such as `\n`

String-related types

Let's see rune vs byte in a string:

```
package main
import "fmt"

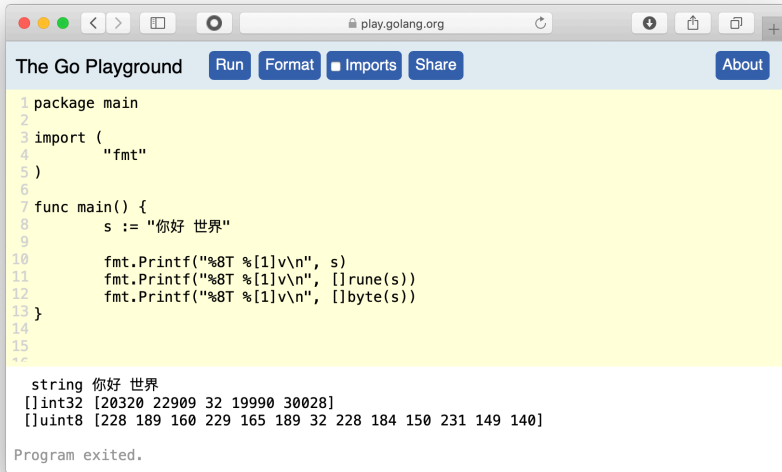
func main() {
    s := "élite"
    fmt.Printf("%8T %[1]v\n", s)
    fmt.Printf("%8T %[1]v\n", []rune(s))
    fmt.Printf("%8T %[1]v\n", []byte(s))
}
```

é is one rune (character) but two bytes in UTF-8 encoding:

```
string élite
[]int32 [233 108 105 116 101]
[]uint8 [195 169 108 105 116 101]
```


String-related types, in Chinese

I can't do this in the slides:



The screenshot shows a web browser window with the address bar displaying "play.golang.org". The page title is "The Go Playground". There are buttons for "Run", "Format", "Imports", "Share", and "About". The main area contains a Go program that prints a string and its underlying byte and rune representations.

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     s := "你好 世界"
9
10    fmt.Printf("%8T %[1]v\n", s)
11    fmt.Printf("%8T %[1]v\n", []rune(s))
12    fmt.Printf("%8T %[1]v\n", []byte(s))
13 }
14
15
```

The output of the program is:

```
string 你好 世界
[]int32 [20320 22909 32 19990 30028]
[]uint8 [228 189 160 229 165 189 32 228 184 150 231 149 140]
```

Program exited.

Slices

Wait — what where the `[]` things?

A *slice* `[]T` is a *sequence* of zero or more `T`

A slice of n objects has *indexes* 0 up to $n - 1$; in math, $[0, n)$

A slice `x` can be indexed `x[0]` to yield a single item

Or it can be indexed `x[m:n]` to yield a slice of $n - m$ items
(with index values m up to $n - 1$)

Strings can also use the index brackets to select bytes

```
package main
import "fmt"

func main() {
    t := []byte("string")    // 0:s 1:t 2:r 3:i 4:n 5:g

    fmt.Println(len(t), t)   // 6 bytes in t
    fmt.Println(t[2])        // 1 item
    fmt.Println(t[:2])       // 2 items
    fmt.Println(t[2:])       // 6-2 items
    fmt.Println(t[3:5])      // 5-3 items
}
```

```
6 [115 116 114 105 110 103]
114
[115 116]
[114 105 110 103]
[105 110]
```

String operations

Strings are immutable but can be combined

```
func main() {  
    s1 := "a string"  
    s2 := "another string"  
  
    fmt.Println(s1 + ", " + s2)  
  
    s3 := s2 + string('?') // can't add a rune  
  
    fmt.Println(s3)  
}
```

which produces

a string, another string
another string?

String functions

Package `strings` has many functions on strings

```
s := "a string"

x := len(s)           // built-in, = 8

strings.Contains(s, "g") // returns true
strings.Contains(s, "x") // returns false

strings.HasPrefix(s, "a") // returns true
strings.ToUpper(s)       // returns "A STRING"
```

Indexes in strings are numbered from 0 up to `len(s) - 1`

```
strings.Index(s, "string") // returns 2
```

Program Logic

The logic of a program

Programs don't just have variables, they need logic

There are three main kinds of logic structures:

making choices: if-then-else

repeating things: loops

separating out pieces of logic: functions

In the absence of other structure, execution proceeds from top to bottom:

```
func main() {  
    doThisFirst()  
    thenDoThis()  
}
```

Alternation

“If it’s sunny, we’ll go to the ballgame, otherwise a movie”

That same type of logic is applied using the “if-then-else” structure

```
func main() {  
    weather := os.Args[1]    // command-line arguments  
  
    if weather == "sunny" {  
        doGame()  
    } else {  
        doMovie()  
    }  
}
```

The if-condition is a *boolean* expression which must evaluate to either false or true

Boolean and conditional operators

Conditions include the notion of “equal to” for most types:

`==` equal to

`!=` not equal to

Only strings and numbers normally have order:

`<` less than

`<=` less than or equal to

`>` greater than

`>=` greater than or equal to

`=` means assignment, not equality, while `:=` is a short declaration

Boolean and conditional operators

Conditions can be combined with boolean operators:

`&&` and

`||` or

`!` not (as a prefix)

```
x < y || x < z // x must be less than one of y, z  
n > 0 && !b    // n must be positive and b not true
```

`&&` and `||` are *shortcut* operators; they evaluate the right side only if necessary

For example, if `x` is 0, we know this expression is false just from its left side:

```
x != 0 && y/x >= 1
```

Repetition

“Print the line ten times”

The “for” loop is called that because most often it looks like

```
func main() {  
    s := "I will not talk in class"  
  
    for i := 0; i < 10; i++ {  
        fmt.Println(s)  
    }  
}
```

The parts separated by ‘;’ are

- a short declaration of a *loop index*
- a repeating condition (do the *loop body* while true)
- an expression changing the loop index (after the body)

Repetition

“Roll the dough until it is smooth”

The loop control can be just a boolean expression:

```
func main() {  
    d := makeDough()  
  
    for !d.smooth() {  
        d.roll()           // must set smooth() true sometime  
    }  
  
    bake(d)  
}
```

Go only has one loop structure (there's no “while” or “until”), but it does have some additional options

Repetition

“For all values in the sequence, do this”

The `range` operator ranges over a sequence, returning an index and a value:

```
func main() {  
    s := "abc"  
  
    for i, r := range s {  
        fmt.Println(i, r, string(r))  
    }  
}
```

```
0 97 a  
1 98 b  
2 99 c
```

Repetition

“Do this forever”

This loop will run until `break` makes it stop

```
func main() {  
    scanner := bufio.NewScanner(os.Stdin)  
  
    for {  
        if !scanner.Scan() {  
            break  
        }  
  
        fmt.Println(scanner.Text())  
    }  
}
```

Here `Scan()` will return `false` only if there's no more input

A complete repeating program

```
package main
import ("fmt"; "bufio"; "os")    // legal, bad style

func main() {
    scanner := bufio.NewScanner(os.Stdin)

    for {
        fmt.Print("Enter your text: ")
        ok, text := scanner.Scan(), scanner.Text()

        if !ok || text == "q" {
            fmt.Println()
            break
        }

        fmt.Println("Your text was: ", text)
    }
}
```

Functions

We've already created one function: `main`

Often we have some code we don't want to retype over & over

If we put it into a function, it's reusable

Breaking up big programs into functions make them easier to read

And think about

Functions

Most functions take *parameters* (or “arguments”) as input

And possibly give back a *return value*

```
func twice(x int) int {    // twice(10) will be 20
    return 2 * x
}
```

OK, that was lame; how about

```
func quadratic(a, b, c float64) (float, float) {
    d := math.Sqrt(b*b - 4*a*c)
    return (-b + d)/(2*a), (-b - d)/(2*a)
}
```

Yes, a function can return more than one value

Functions

We've been calling a bunch of functions so far

```
s := "a string"
```

```
strings.ToUpper(s)    // returns "A STRING"
```

```
strings.Contains(s, "a") // returns true
```

```
x := 16.0
```

```
math.Sqrt(x)          // returns 4.0
```

Note that we must write 16.0 or x will be an int

Why does that matter?

Function parameters

Every function parameter has a name and a type

To call a function, the *actual* parameters must match the all the *formal* parameters in the correct order

```
func friz(a int, b string)
```

```
friz(2, "a")           // OK
```

```
frize(2.0, "x")        // wrong type for "a"
```

```
friz(2, 1)             // wrong type for "b"
```

```
friz(2)                // not enough parameters
```

math.Sqrt() takes a floating point number, not an integer

```
func Sqrt(x float64) float64
```

Methods

Some functions are *methods* because we call them on an object

They're just a special type of function call

```
package main

import ("fmt"; "bufio"; "os")

func main() {
    scanner := bufio.NewScanner(os.Stdin)    // function

    scanner.Scan()                            // method
    fmt.Println("That's ", scanner.Text())    // one of each
}
```

We qualify the function with the object name, not a package name

Tools and Techniques

Installation

Start from the Go language page: <https://golang.org>

Mac: run `brew install go` (or use the installer package)

Homebrew installation: <https://brew.sh>

Windows: open the installer (MSI) file and follow the prompts to install the Go tools

(otherwise you can download a ZIP file, but you have to set some environment stuff)

Linux: download the archive and extract it into `/usr/local`, creating a Go tree in `/usr/local/go`

```
sudo tar -C /usr/local -xzf go1.11.5.linux-amd64.tar.gz
```

and don't forget to add `/usr/local/go/bin` to `$PATH` (Linux)

Go command-line tools

The go program can run code or make a binary

go run compiles to a temporary directory, then deletes the result

```
$ go run hello.go      ## temporary compile  
Hello, world!
```

```
$ gofmt -w hello.go   ## standard format
```

```
$ go build hello.go   ## produces a binary "hello"
```

```
$ ./hello  
Hello, world!
```

gofmt reformats code, use the -w option to overwrite the file

Printing 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)
}
```

We'll skip `os.Args[0]` because that's the program name:

```
/var/folders/6y/q8z5w4xn1dzb0_qz680mcs6h0000gn/T/go-build451715571/b001/exe/hello
```


Running a program with a bug

Here's a short program:

```
func main() {  
    fmt.Println(os.Args[1])  
}
```

Running it on the command line:

```
$ go run badargs.go    ## no argument given  
panic: runtime error: index out of range  
goroutine 1 [running]:  
main.main()  
/Users/mholiday/go/src/badargs/badargs.go:55 +0x202  
exit status 2
```

What went wrong? We read past the end of `os.Args`!

Reading a file

Here's another program that checks a file's size

```
package main

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

func main() {
    fname := os.Args[1]
    if f, err := os.Open(fname); err != nil {
        fmt.Fprintln(os.Stderr, "bad file:", err)
    } else if d, err := ioutil.ReadAll(f); err != nil {
        fmt.Fprintln(os.Stderr, "can't read:", err)
    } else {
        fmt.Printf("The file has %d bytes\n", len(d))
    }
}
```

If run on itself (the source file), it prints “The file has 333 bytes”

Reading a file

Wait, what's going on here?

```
if f, err := os.Open(fname); err != nil {  
    fmt.Fprintln(os.Stderr, "bad file:", err)  
} . . .
```

An if-statement can have a short declaration as its first part

We often call functions whose 2nd return value is a possible error

```
func Open(name string) (*File, error)
```

where the error can be compared to `nil`, meaning no error

Always check the error — the file might not really be open!

Homework

Homework # 1

Write a program to take a file name from the command line and count the lines in the file (a line ends with "\n").

You can test your first program with `wc`:

```
$ wc testing.txt
    9      35    180 testing.txt
```

which prints out the number of lines, words, and characters

You may want to use `scanner.Scan()` and `scanner.Text()`