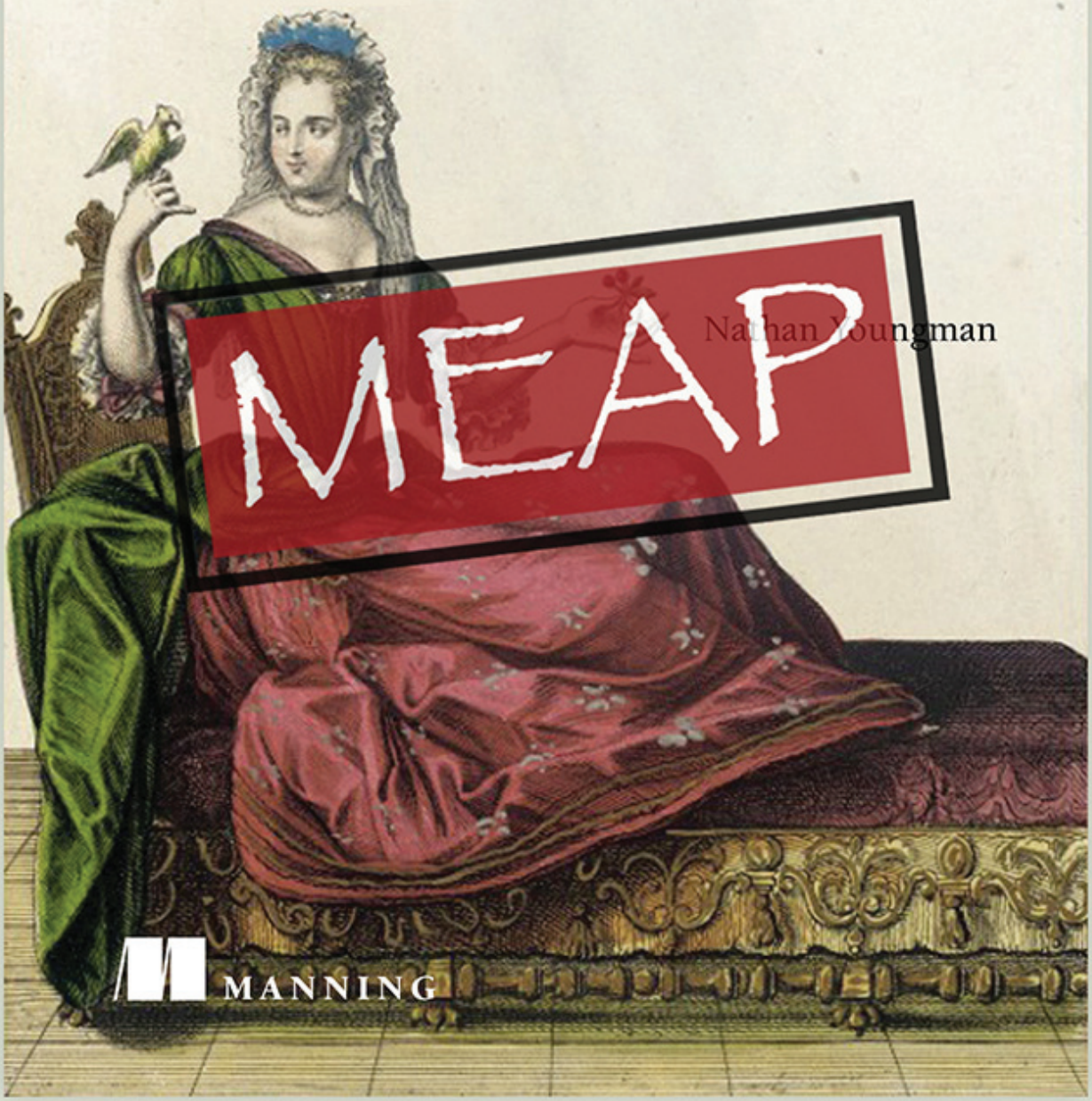
Learn Go



Nathan Youngman

MEAP Edition
Manning Early Access Program
Learn Go
Version 7

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thank you for taking a look at *Learn Go*. I'm excited to be able to share this early release with you. As you're reading, I hope you'll take advantage of the Author Online forum to pose questions and leave feedback. This is your opportunity to influence Learn Go, helping to make it the best book it can be. I plan to update *Learn Go* every four weeks, whether with new chapters, or revisions to existing chapters.

Learn Go is a beginner's guide to the Go programming language. I've strived to make the content approachable and lighthearted, especially accessible to hobbyists and newcomers to compiled programming languages. It's chock-full of exercises and challenges to get you coding in The Go Playground right away. In fact, you can work through most of *Learn Go* with just a web browser.

Chapters are no more than ten pages each. You can read a chapter and do the exercises in an evening and get through the whole book within a month.

After that, it's up to you. Build something truly wonderful and change the world.

—Nathan Youngman

brief contents

preface

"Everything changes and nothing remains still."

-- Heraclitus

While traveling Europe in 2005, I heard rumblings of a new web framework called Rails. I returned to the muddy roads of Edmonton, Alberta just in time to celebrate Christmas. In those days, there was a computer book store downtown, where I found a copy of *Agile Web Development with Rails*. Over the next two years, I transitioned my career from ColdFusion to Ruby.

Ever since, I've kept an eye open for the next language to further my career. When Clojure and Scala came out, I picked up a book and tried them out. While each language has interesting ideas, nothing hit the spot for me.

November 2009 came and went. I watched Rob Pike's tech talk announcing Go, installed it, and ran "Hello World." At the time, I was too enticed by functional programming to recognize what made Go special.

Then one of my co-workers decided to try Go for a side project. He spoke highly of the language, convincing me to take a second look. Over another Christmas break, I read through the Rough Cut of *The Go Programming Language Phrasebook*. I liked it.

In May 2013, my blog post *Why Go?* blew up on Hacker News, bringing in 20,000 readers. I was also preparing a talk for a local meetup and put my practice session on Vimeo. An acquisitions editor saw it and suggested that I write a book. I was intrigued, but ultimately turned down the opportunity. I was no Go expert.

"Write when you're filled with wonder… If you wait until you're an expert, it's too late."

-- Mark Pilgrim

Having become a champion for Go, January 2014 marks the first Edmonton Go meetup. By November, I was taking on freelance work using Go. When I was finally ready to move away from Rails, a call came from Michael Stephens to discuss what would become Learn Go.

about this book

Go is suitable for programmers with a wide range of skill levels; a necessity for any large project.

Unfortunately, many resources for learning Go presume a working knowledge of the C programming language. Learn Go exists to fill the gap for scripters, hobbyists, and newcomers looking for a direct path to Go. If you've used a scripting language like JavaScript, Lua, PHP, Python, or Ruby, you're ready to *Learn Go.*

If you've used Scratch or Excel formulas, or written HTML, you're not alone in choosing Go as your first "real" programming language. [1] It will take patience and effort, but I hope Learn Go is a helpful resource in your quest.

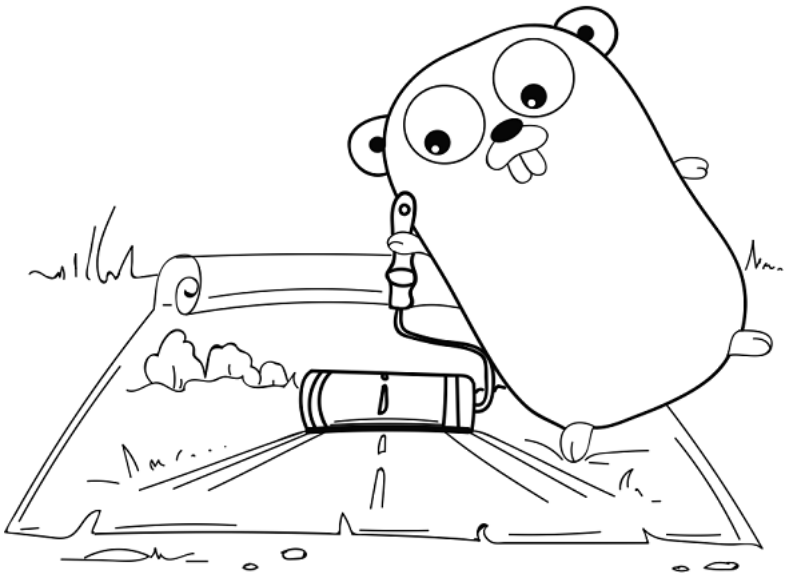
This is a beginner's guide to Go. It isn't a complete specification [2] of every language feature. Learn Go touches on several advanced topics but it is only the beginning.

If you are a polyglot programmer who understands the merits of static typing and composition over inheritance, if pointers and mutexes are second nature, if optimizing memory allocations and CPU caches are an old hat, you will find plenty of books [3] and resources that ramp up quickly and cover advanced topics more thoroughly.

Roadmap

Learn Go gradually explains the concepts needed to use Go effectively and provides a plethora of exercises as you follow along. Whether you go on to write massively *concurrent* web services or small scripts and simple tools, Learn Go helps you establish a solid foundation.

[1] A Beginner's Mind by Audrey Lim www.youtube.com/watch?v=fZh8uCInEfw
[2] The Go Programming Language Specification golang.org/ref/spec
[3] Go books golang.org/wiki/Books



Part 1 brings together *variables*, *loops*, and *branches* to build apps for Mars tourists.

Part 2 explores *types* for both text and numbers. Decode messages from space with ROT13, investigate the destruction of the Arianne 5 rocket, and use big numbers to calculate the time light takes to reach Andromeda.

Part 3 uses *functions* and *methods* to build a fictional weather station on Mars with sensor readouts and temperature conversions.

Part 4 demonstrates how to use *arrays* and *maps* while terraforming the solar system, tallying up temperatures, and simulating the Game of Life.

Part 5 introduces concepts from *object-oriented* languages in a distinctly non-object oriented language. Use *structures* and methods to navigate the surface of Mars, satisfy *interfaces* to improve output, and embed structures in one another.

Part 6 digs into the nitty-gritty. Use *pointers* to share memory, learn how to handle errors appropriately, communicate between thousands of running tasks with Go's *concurrency* primitives, and divide projects into manageable pieces with *packages*.

Appendix B helps you get the Go compiler and a text editor setup on your computer, and introduces the command line tools.

Code conventions and downloads

All source code in listings or in text is in a `fixed-width font` to separate it from ordinary text. Code annotations accompany many of the listings, highlighting important concepts.

You can download the source code for all listings from the Manning website, www.manning.com/books/learn-go. The download also includes solutions for all the exercises in this book. If you prefer to browse the source code online, you can find it in the GitHub repository at github.com/gopherbook/learn-go-code.

While you could copy and paste code from GitHub, I encourage you to type in the examples yourself. You'll get more out of the book by typing the examples, fixing typos, and experimenting with the code than you will just reading alone.

Author Online

The purchase of Learn Go includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, share your solutions to exercises, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to forums.manning.com/forums/learn-go.

The author online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

Nathan Youngman began to code in his preteens with a BASIC interpreter and a manual. Hobby turned career, through an era of ColdFusion and Ruby on Rails before arriving at Go. He is a contributor to Go's open source ecosystem, organizer of the Edmonton Go meetup, and paparazzi of VIP gopher plushies.

8 Multilingual text

In this chapter:

- Access and manipulate individual letters
- Cipher and decipher secret messages
- Write your programs for a multilingual world

From `"Hello, playground"` at the beginning, you have been using text in your programs. The individual letters, digits, and symbols are called *characters*. When you *string* together characters and place them between quotes, it is called a *string*.

Figure 8.1 A string of characters

8.1 Declaring string variables

Literal values wrapped in quotes are inferred to be of the type `string`, so the following three lines are equivalent:

```
peace := "peace"
var peace = "peace"
var peace string = "peace"
```

String literals may contain *escape sequences* such as `\n`. To avoid substituting `\n` for a new line, you can wrap text in backticks (`` ` ``) instead of quotes (`"`). Backticks indicate a *raw* string literal.

Listing 8.1 Raw string literals: raw.go

```
fmt.Println("peace be upon you\nupon you be peace")
fmt.Println(`strings can span multiple lines with the \n escape sequence`)
```

The previous listing displays this output:

```
peace be upon you
upon you be peace
strings can span multiple lines with the \n escape sequence
```

If you declare a variable without providing a value it will be initialized with the zero value for its type. The zero value for the `string` type is an empty string (`""`).

```
var cheese string
```

8.2 Characters, code points, runes, and bytes

The Unicode Consortium assigns numeric values, called *code points*, to over one million unique characters. For example, 65 is the code point for the capital letter A and the number 128515 is the code point for a smiley face □.

To represent a single Unicode code point, Go provides `rune`, which is an alias for the `int32` type.

A `byte` is an alias for the `uint8` type. It is intended for binary data, though `byte` can be used for English characters defined by ASCII, an older 128-character subset of Unicode.

Both `byte` and `rune` behave like the integer types they are aliases for, as shown in the following listing.

Listing 8.2 Runes and bytes: rune.go

```
var pi rune = 960
var alpha rune = 940
var omega rune = 969
var bang byte = 33

fmt.Printf("%v %v %v %v\n", pi, alpha, omega, bang) ❶
```

❶ Print **960 940 969 33**

To display the characters rather than their numeric values, the `%c` format verb can be used with `Printf`:

```
fmt.Printf("%c%c%c%c\n", pi, alpha, omega, bang) ❶
```

❶ Print **πάω!**

Tip

Actually, any integer type will work with `%c`, but the `rune` alias indicates that the number 960 represents a character.

Question

Q8.1 **How many characters does ASCII encode?**

Q8.2 **What type is `byte` an alias for? What about `rune`?**

Rather than memorize Unicode code points, Go provides a character literal. Just enclose a character in single quotes `' '`. If no type is specified, Go will infer

a `rune`, so the following three lines are equivalent:

```
grade := 'A'
var grade = 'A'
var grade rune = 'A'
```

The `grade` variable still contains a numeric value, in this case 65, the code point for a capital `'A'`. Character literals can also be used with the `byte` alias:

```
var star byte = '*'
```

Experiement: code-point.go

Try other English and international characters to see their code points.

8.3 Pulling the strings

A puppeteer manipulates a marionette by pulling on strings, but strings in Go aren't susceptible to manipulation. A variable can be assigned to a different string, but strings themselves cannot be altered.

```
peace := "shalom"
peace = "salām"
```

Your program can access individual characters, but it cannot alter the characters of a string. The following listing use square brackets `[]` to specify an index into a string, allowing it to access a single byte. The index starts from zero, but some characters consume multiple bytes (more on that later).

Listing 8.3 Indexing into a string: index.go

```
message := "salām"
c := message[5]
fmt.Printf("%c\n", c) ❶
```

❶ Print **m**

Strings in Go are *immutable*, as they are in Python, Java, and JavaScript. Unlike strings in Ruby and character arrays in C, you cannot modify a string:

```
message[5] = 'd' ❶
```

❶ cannot assign to message[5]

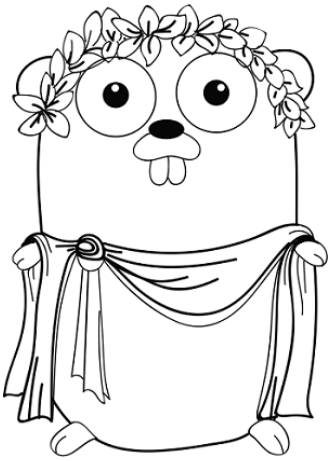
8.4 Manipulating characters with Caesar cipher

An effective method of sending secret messages in the second century was to shift every letter, so `'a'` becomes `'d'`, `'b'` becomes `'e'`, etc. The result

might pass for a foreign language:

L fdph, L vdz, L frqtxhuhg.

-- Julius Caesar



It turns out that manipulating characters as numeric values is really easy with computers.

Listing 8.4 Manipulate a single character: caesar.go

```
c := 'a'
c = c + 3
fmt.Printf("%c", c) ❶
```

❶ Print **d**

Listing 8.4 has one problem though. It doesn't account for all the messages about xylophones, yaks, and zebras. To address this need, the original Caesar cipher wraps around, so `'x'` becomes `'a'`, `'y'` becomes `'b'`, and `'z'` becomes `'c'`. With 26 characters in the English alphabet, it's a simple matter:

```
if c > 'z' {
    c = c - 26
}
```

To decipher Caesar cipher, subtract 3 instead of adding 3. But then you need to account for `c < 'a'` by adding 26. What a pain.

8.4.1 A modern variant

ROT13 (rotate 13) is a 20th century variant of Caesar cipher. It has one difference: it adds 13 instead of 3. With ROT13, ciphering and deciphering are the same convenient operation.

Let's suppose the SETI Institute received a transmission with the following message:

```
message := "uv vagreangvbany fcnpr fgngvba"
```

I suspect this `message` is actually English text that was ciphered with ROT13. Call it a hunch. Before you can crack the code, there is one more things you need to know. This `message` is 30 characters long, which can be determined with the built-in `len` function:

```
fmt.Println(len(message)) ❶
```

❶ Print **30**

Note

Go has a handful of built-in functions that don't require an import statement. The `len` function can determine the length for a variety of types. In this case, `len` returns the length of a `string` in bytes.

Listing 8.5 ROT13 cipher: rot13.go

```
message := "uv vagreangvbany fcnpr fgngvba"

for i := 0; i < len(message); i++ { ❶
    c := message[i]
    if c >= 'a' && c <= 'z' { ❷
        c = c + 13
        if c > 'z' {
            c = c - 26
        }
    }
    fmt.Printf("%c", c)
}
```

❶ Iterate through each ASCII character
❷ Leave spaces and punctuation as is

Experiement: caesar.go

Type Listing 8.5 into the Go Playground to see what the message says.

- Modify `message` to cipher and decipher other messages.
- Add support for shifting upper case characters.
- Handle negative shifts when `c < 'a'` by adding 26. Do the same for upper case characters.
- Decipher the quote from Julius Caesar at the beginning of this section by shifting characters by -3.

Experiement: international.go

It is important to note that the ROT13 implementation in Listing 8.5 is only intended for ASCII characters. Try to cipher these Spanish or Russian message to see what happens to non-ASCII characters:

- Hola Estación Espacial Internacional
- привет Международная космическая станция

The next section will look at a solution for this issue.

Question

Q8.3 What does the built-in `len` function do when passed a string?

8.5 Decoding strings into runes

Strings in Go are encoded with UTF-8, one of several encodings for Unicode code points. UTF-8 is an efficient variable length encoding where a single code point may use 8-bits, 16-bits, or 32-bits. By using a variable length encoding, UTF-8 makes the transition from ASCII straightforward, as ASCII characters are identical to their UTF-8 encoded counterparts.

Note

UTF-8 is the dominant character encoding for the World Wide Web. It was invented in 1992 by Ken Thompson, one of the designers of Go.

The ROT13 program in Listing 8.5 accessed the individual bytes (8-bit) of the `message` string without accounting for characters that are multiple bytes long (16-bit or 32-bit). This is why it worked fine for English characters (ASCII), but produced garbled results for Russian and Spanish. You can do better, amigo.

The first step to support other languages is to decode characters to the `rune` type before manipulating them. Fortunately Go has functions and language features for decoding UTF-8 encoded strings.

The following listing the `utf8` package to determine the length of a string in runes rather than bytes, and to decode the first character of a string. The `DecodeRuneInString` function returns the first character and the number of bytes the character consumed.

Note

Unlike many programming languages, functions in Go can return multiple values. Multiple return values will be discussed in Chapter 10.

Listing 8.6 The utf8 package: spanish.go

```
package main

import (
        "fmt"
        "unicode/utf8"
)

func main() {
        question := "¿Cómo estás?"

        fmt.Println(len(question), "bytes")                    ❶
        fmt.Println(utf8.RuneCountInString(question), "runes") ❷

        c, size := utf8.DecodeRuneInString(question)
```

```
        fmt.Printf("First rune: %c %v bytes", c, size)          ❸
}
```

❶ Print **15 bytes**
❷ Print **12 runes**
❸ Print **First rune: ¿ 2 bytes**

The Go language provides the `range` keyword to iterate over a variety of collections (covered in part 4). It can also decode UTF-8 encoded strings as follows:

Listing 8.7 Decoding runes: spanish-range.go

```
question := "¿Cómo estás?"

for i, c := range question {
    fmt.Printf("%v %c\n", i, c)
}
```

On each iteration, the variables `i` and `c` are assigned to an index into the string and the code point (`rune`) at that position.

If you don't need the index, the blank identifier (an underscore) allows you to ignore it.

```
for _, c := range question {
    fmt.Printf("%c ", c) ❶
}
```

❶ Print **¿ C ó m o e s t á s ?**

Experiement: rot13.go

Modify Listing 8.5 to use the range keyword. Now when you use ROT13 on Spanish text, characters with accents are left as is.

Question

Q8.4 How many runes are in the English alphabet `"abcdefghijklmnopqrstuvwxyz"`? How many bytes?

Q8.5 How many bytes are in the rune `'¿'`?

8.6 Summary

- Escape sequences like `\n` are ignored in raw string literals (`` ` ``).
- Strings are immutable. Individual characters can be accessed but not altered.
- Strings use a variable length encoding called UTF-8, where each character

consumes 1-4 bytes.

- A `byte` is an alias for the `uint8` type and `rune` is an alias for the `int32` type.
- The `range` keyword can decode a UTF-8 encoded string into runes.