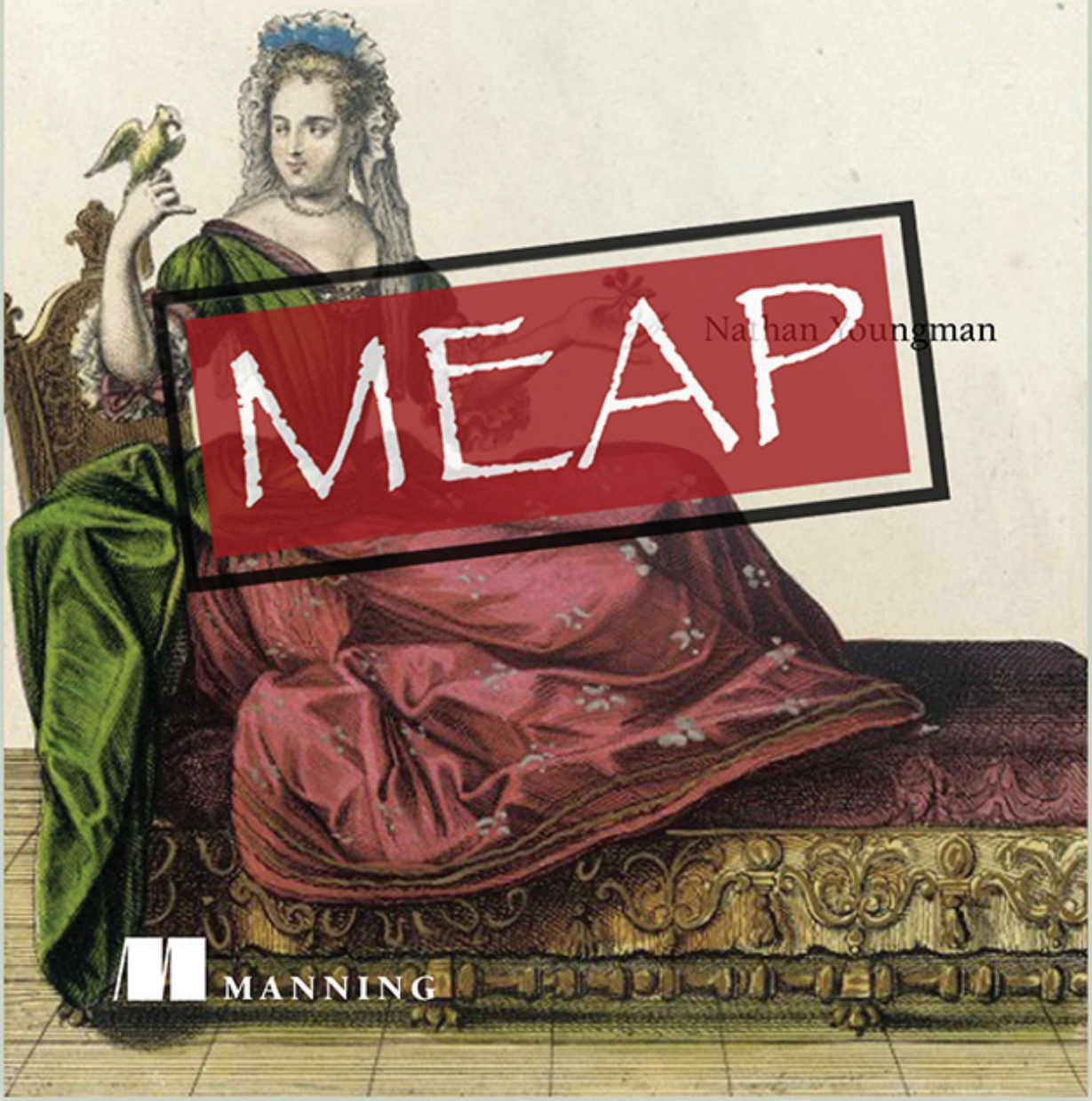
Learn Go



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

Acknowledgements

About this book

PART 1: IMPERATIVE PROGRAMMING

1 *Get ready, get set, Go*

2 *A glorified calculator*

3 *Loops and branches*

4 *Variable scope*

Challenge 1: ticket to Mars

PART 2: TYPES

5 *Real numbers*

6 *Whole numbers*

7 *Big numbers*

8 *Multilingual text*

9 *Converting between types*

Challenge 2: the Vigenère cipher

PART 3: BUILDING BLOCKS

10 *Functions*

11 *First-class functions*

12 *Methods*

Challenge 3: to be determined

PART 4: COLLECTIONS

13 Arrayed in splendor

14 Slices: a window into an array

15 A bigger slice

16 The ever versatile map

Challenge 4: a slice of life

PART 5: STATE AND BEHAVIOR

17 A little structure

18 Go's got no class

19 Interfaces

20 Composition and forwarding

Challenge 5: to be determined

PART 6: DOWN THE GOPHER HOLE

21 Pointers

22 Handling errors

23 Concurrency

24 Writing and consuming packages

Where to Go from here

A: Answers

B: Beyond the playground

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.

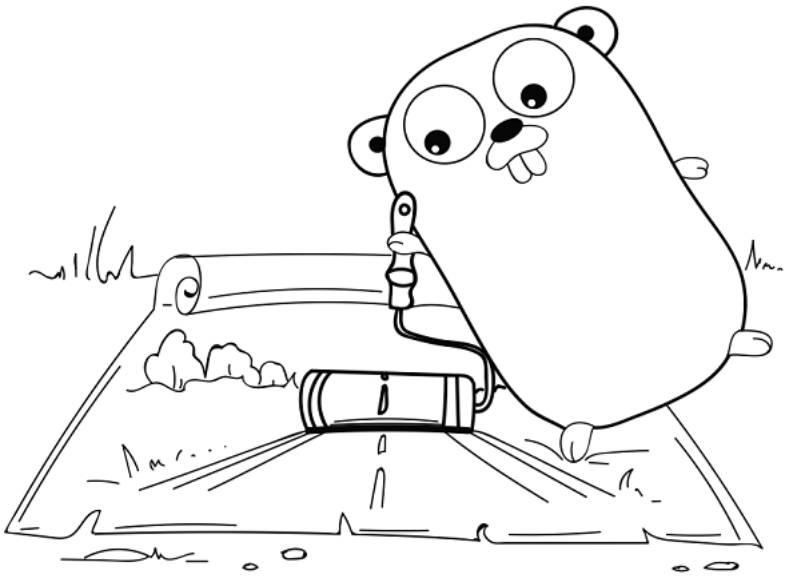
This is a beginner's guide to Go. It isn't a complete specification [2] of every language feature. Learn Go touches on several advanced topics but it is only the beginning.

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.

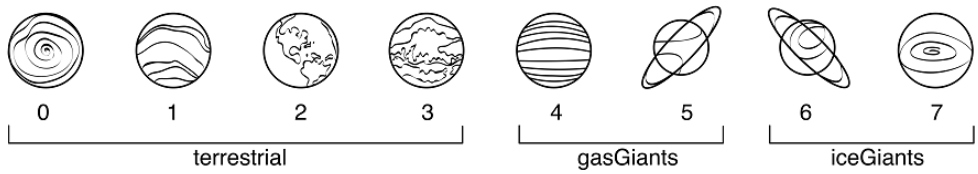
14 Slices: a window into an array

In this chapter:

- Uses slices to view the solar system through a window
- Alphabetize slices with the standard library

The planets in our solar system are classified as terrestrial, gas giants, and ice giants. You can focus on the terrestrial by *slicing* the first four elements of the planets array with `planets[0:4]`. Slicing doesn't alter the `planets` array, it just creates a window or view into the array. This view is a type called a *slice*.

Figure 14.1 Slicing the solar system



14.1 Slicing an array

Slicing is expressed with a *half-open range*. For example, in the following listing, `planets[0:4]` begins with the planet at index 0 and continues up to, but not including, the planet at index 4.

Listing 14.1 Slicing an array: slicing.go

```
planets := [...]string{
    "Mercury",
    "Venus",
    "Earth",
    "Mars",
    "Jupiter",
    "Saturn",
    "Uranus",
    "Neptune",
}

terrestrial := planets[0:4]
gasGiants := planets[4:6]
iceGiants := planets[6:8]

fmt.Println(terrestrial, gasGiants, iceGiants) ❶
```

❶ Print **[Mercury Venus Earth Mars] [Jupiter Saturn] [Uranus Neptune]**

Though `terrestrial`, `gasGiants`, and `iceGiants` are slices, you can still index into slices like arrays.

```
fmt.Println(gasGiants[0]) ❶
```

❶ Print **Jupiter**

You can also slice an array, and then slice the resulting slice.

```
giants := planets[4:8]
gas := giants[0:2]
ice := giants[2:4]
fmt.Println(giants, gas, ice) ❶
```

❶ Print **[Jupiter Saturn Uranus Neptune] [Jupiter Saturn] [Uranus Neptune]**

The `terrestrial`, `gasGiants`, `iceGiants`, `giants`, `gas`, and `ice` slices are all views of the same `planets` array. Assigning a new value to a element of a slice actually modifies the underlying `planets` array. The change will be visible through the other slices.

```
iceGiantsMarkII := iceGiants                ❶
iceGiants[1] = "🤮"
fmt.Println(planets)                        ❷
fmt.Println(iceGiants, iceGiantsMarkII, ice) ❸
```

❶ Copy the iceGiants slice (a view of the planets array)

❷ Print **[Mercury Venus Earth Mars Jupiter Saturn Uranus 🤮]**

❸ Print [Uranus 🤏] [Uranus 🤏] [Uranus 🤏]

Question

Q14.1 What does slicing an array produce?

Q14.2 When slicing with `planets[4:6]`, how many elements are in the result?

14.1.1 Default indices for slicing

When slicing an array, omitting the first index defaults to the beginning of the array. Omitting the last index defaults to the length of the array. This allows the slicing in Listing 14.1 to be written as follows.

Listing 14.2 Default indices: slicing-default.go

```
terrestrial := planets[:4]
gasGiants := planets[4:6]
iceGiants := planets[6:]
```

Note

Slice indices may not be negative.

You can probably guess what omitting both indices does. The `allPlanets` variable is a slice containing all eight planets.

```
allPlanets := planets[:]
```

Slicing strings

The slicing syntax for arrays also works on strings.

```
neptune := "Neptune"
tune := neptune[3:]

fmt.Println(tune) ❶
```

❶ **Print tune**

The result of slicing a string is another string. Assigning a new value to `neptune` won't change the value of `tune` or visa-versa.

```
neptune = "🤏"
fmt.Println(tune) ❶
```

❶ **Print tune**

Be aware that the indices are in number of bytes, not runes.

```
question := "¿Cómo estás?"
fmt.Println(question[:6]) ❶
```

❶ **Print ¿Cóm**

14.2 Composite literals for slices

Many functions in Go operate on slices rather than arrays. If you need a slice that reveals every element of the underlying array, one option is to declare an array and then slice it with `[:]`.

```
dwarfArray := [...]string{"Ceres", "Pluto", "Haumea", "Makemake", "Eris"}
dwarfSlice := dwarfArray[:]
```

Alternatively, Go lets you declare a slice in a single step. In the following listing, `dwarfs` is a slice, as indicated by the empty brackets `[]`. An array declaration always specifies a fixed length or ellipsis between the brackets.

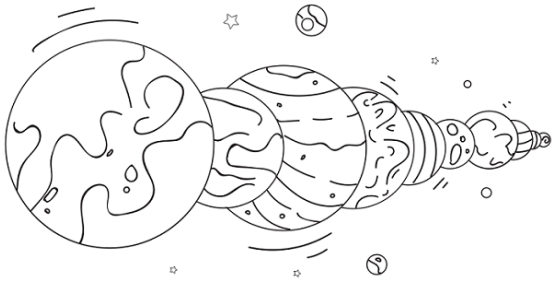
Listing 14.3 Start with a slice: dwarf-slice.go

```
dwarfs := []string{"Ceres", "Pluto", "Haumea", "Makemake", "Eris"}
```

There is still an underlying array. Behind the scenes, Go declares a five element array, and then makes a slice that views all of its elements.

14.3 The power of slices

What if there was a way to fold the fabric of space-time, bringing worlds together for instantaneous travel? Using the Go standard library and some ingenuity, the following `hyperspace` function modifies a slice of `worlds`, removing the space between them.



Listing 14.4 Bringing worlds together: hyperspace.go

```
package main

import (
        "fmt"
        "strings"
)

// hyperspace removes the space surrounding worlds
func hyperspace(worlds []string) {                              ❶
        for i := range worlds {
                worlds[i] = strings.TrimSpace(worlds[i])
        }
}

func main() {
        planets := []string{" Venus   ", "Earth  ", " Mars"} ❷
        hyperspace(planets)

        fmt.Println(strings.Join(planets, ""))                  ❸
}
```

❶ This argument is a slice, not an array
❷ Planets surrounded by space
❸ Print **VenusEarthMars**

Both `worlds` and `planets` are slices, and though `worlds` is a copy, they both point to the same underlying array.

If `hyperspace` were to change where the `worlds` slice points, begins, or ends, those changes would have no impact on the `planets` slice. However, `hyperspace` is able to reach into the underlying array that `worlds` points to, and change its elements. Those changes are visible by other slices (views) of the array.

Question

Q14.3 Lookup TrimSpace and Join in the Go documentation golang.org/pkg/. What functionality do they provide?

Slices are more versatile than arrays in other ways too. Slices have a length, but unlike arrays, the length is not part of the type. You can pass a slice of any size to the `hyperspace` function.

```
dwarfs := []string{" Ceres  ", " Pluto"}
hyperspace(dwarfs)
```

Arrays are rarely used directly. Gophers prefer slices for their versatility,

especially when passing arguments to functions.

14.4 Slices with methods

In Go you can define a type with an underlying slice or array. Once you have a type, you can attach methods to it. Go's ability to declare methods on types proves more versatile than classes.

The `sort` package in the standard library declares a `StringSlice` type:

```
type StringSlice []string
```

Attached to `StringSlice` is a `Sort` method:

```
func (p StringSlice) Sort()
```

To alphabetize the planets, the following listing converts `planets` to the `sort.StringSlice` type and then calls the `Sort` method.

Listing 14.5 Sorting a slice of strings: sort.go

```
package main

import (
        "fmt"
        "sort"
)

func main() {
        planets := []string{
                "Mercury", "Venus", "Earth", "Mars",
                "Jupiter", "Saturn", "Uranus", "Neptune",
        }

        sort.StringSlice(planets).Sort() ❶
        fmt.Println(planets)             ❷
}
```

❶ Sort planets alphabetically
❷ Print **[Earth Jupiter Mars Mercury Neptune Saturn Uranus Venus]**

To make it even simpler, the `sort` package has a `Strings` helper function that performs the type conversion and calls the `Sort` method for you.

```
sort.Strings(planets)
```

14.5 Summary

- Slices are a window or view into an array.

- The `range` keyword can iterate over slices.
- Slices share the same underlying data when assigned or passed to functions.
- Composite literals provide a convenient means to initialize slices.
- Methods can be attached to slices.