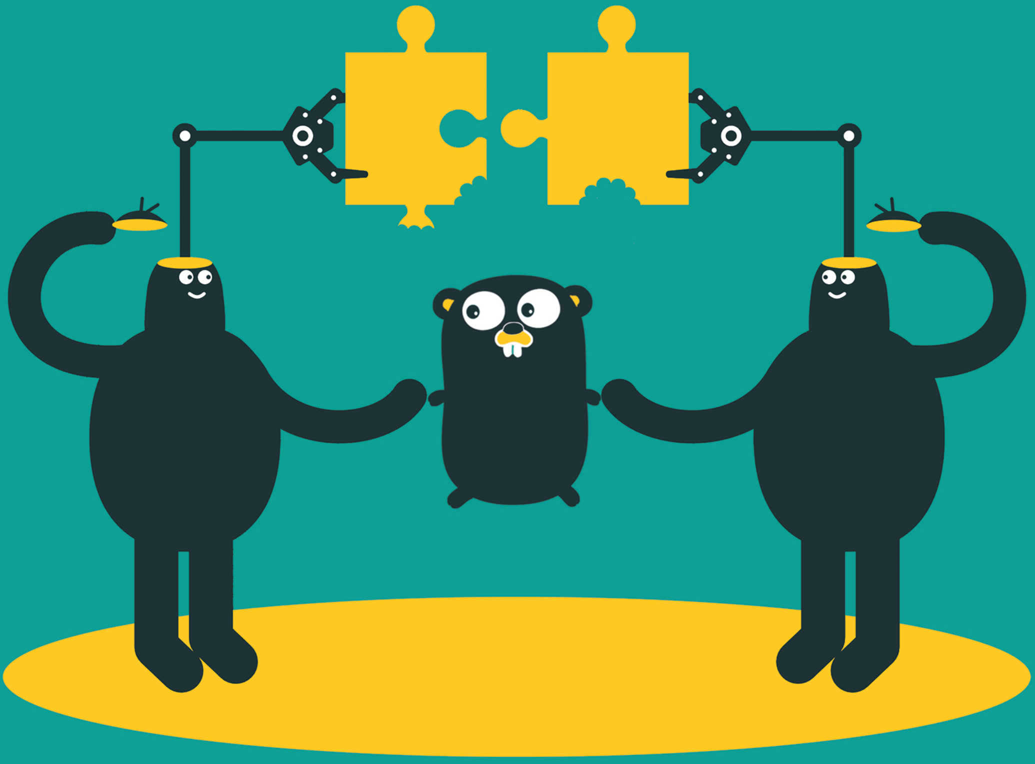
Get Programming with Go



Nathan Youngman
Roger Peppé

MANNING

SAMPLE LESSON

Get Programming with Go

by Nathan Youngman
Roger Peppé

Lesson 17

Contents

Unit 7

LESSON 17

SLICES: WINDOWS INTO ARRAYS

After reading lesson 17, you'll be able to

- Use 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, as shown in figure 17.1. You can focus on the terrestrial ones 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*.

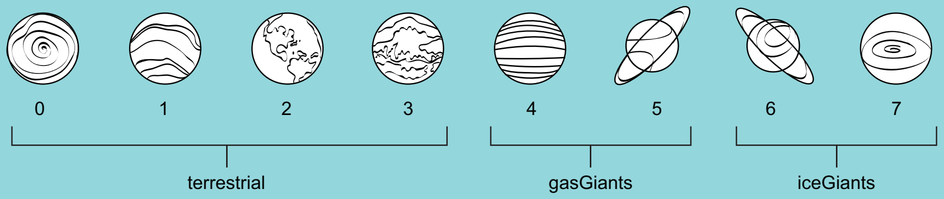


Figure 17.1 Slicing the solar system

Consider this If you have a collection, is it organized in a certain way? The books on a library shelf may be ordered by the last name of the author, for example. This arrangement allows you to focus in on other books they wrote.

You can use slices to zero in on part of a collection in the same way.

17.1 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 17.1 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)
```

Prints [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])
```

Prints 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)
```

Prints [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 an element of a slice modifies the underlying `planets` array. The change will be visible through the other slices:

```
iceGiantsMarkII := iceGiants
iceGiants[1] = "Poseidon"
fmt.Println(planets)
fmt.Println(iceGiants, iceGiantsMarkII, ice)
```

Copies the iceGiants slice (a view of the planets array)

Prints [Mercury Venus Earth Mars Jupiter Saturn Uranus Poseidon]

Prints [Uranus Poseidon] [Uranus Poseidon] [Uranus Poseidon]

Quick check 17.1

1. What does slicing an array produce?
2. When slicing with `planets[4:6]`, how many elements are in the result?

17.1.1 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 from listing 17.1 to be written as shown in the following listing.

Listing 17.2 Default indices: slicing-default.go

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

NOTE Slice indices may not be negative.

QC 17.1 answer

1. A slice.
2. Two.

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)        ← Prints tune
```

The result of slicing a string is another string. However, assigning a new value to `neptune` won't change the value of `tune` or vice versa:

```
neptune = "Poseidon"
fmt.Println(tune)       ← Prints tune
```

Be aware that the indices indicate the number of bytes, not runes:

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

Quick check 17.2 If Earth and Mars were the only colonized planets, how could you derive the slice `colonized` from `terrestrial`?

17.2 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 `[:]`, like this:

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

QC 17.2 answer

```
colonized := terrestrial[2:]
```

Slicing an array is one way to create a slice, but you can also declare a slice directly. A slice of strings has the type `[]string`, with no value between the brackets. This differs from an array declaration, which always specifies a fixed length or ellipsis between the brackets.

In the following listing, `dwarfs` is a slice initialized with the familiar composite literal syntax.

Listing 17.3 Start with a slice: dwarf-slice.go

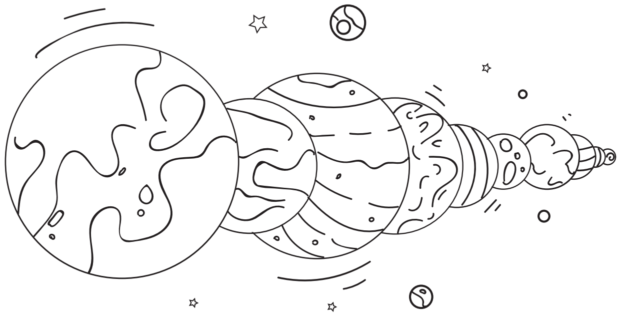
```
dwarfs := []string{"Ceres", "Pluto", "Haumea", "Makemake", "Eris"}
```

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.

Quick check 17.3 Use the `%T` format verb to compare the types of `dwarfArray` and the `dwarfs` slice.

17.3 The power of slices

What if there were a way to fold the fabric of space-time, bringing worlds together for instantaneous travel? Using the Go standard library and some ingenuity, the `hyperspace` function in listing 17.4 modifies a slice of `worlds`, removing the (white) space between them.



QC 17.3 answer

```
fmt.Printf("array %T\n", dwarfArray)    ← Prints array [5]string
fmt.Printf("slice %T\n", dwarfs)    ← Prints slice []string
```

Listing 17.4 Bringing worlds together: hyperspace.go

```go
package main

import (
    "fmt"
    "strings"
)

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

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

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

This argument is a slice, not an array.

Planets surrounded by space

Prints 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. But `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.

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

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

Arrays are rarely used directly. Gophers prefer slices for their versatility, especially when passing arguments to functions.

Quick check 17.4 Look up `TrimSpace` and `Join` in the Go documentation at golang.org/pkg. What functionality do they provide?

17.4 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 the classes of other languages.

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 17.5 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()     ← Sorts planets alphabetically
    fmt.Println(planets)     ← Prints [Earth Jupiter Mars Mercury Neptune Saturn Uranus Venus]
}
```

QC 17.4 answer

1a `TrimSpace` returns a slice with leading and trailing white space removed.
1b `Join` concatenates a slice of elements with a separator placed between them.

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)
```

Quick check 17.5 What does `sort.StringSlice(planets)` do?

Summary

- Slices are windows or views 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.
- You can attach methods to slices.

Let's see if you got this...

Experiment: terraform.go

Write a program to terraform a slice of strings by prepending each planet with `"New "`. Use your program to terraform Mars, Uranus, and Neptune.

Your first iteration can use a terraform function, but your final implementation should introduce a `Planets` type with a terraform method, similar to `sort.StringSlice`.

QC 17.5 answer The `planets` variable is converted from `[]string` to the `StringSlice` type, which is declared in the `sort` package.

SOFTWARE DEVELOPMENT

GET PROGRAMMING WITH GO

Nathan Youngman | Roger Peppé

Go is a small programming language designed by Google to tackle big problems. Large projects mean large teams with people of varying levels of experience. Go offers a small, yet capable, language that can be understood and used by anyone, no matter their experience.

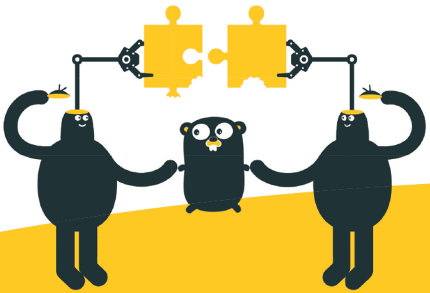
Hobbyists, newcomers, and professionals alike can benefit from a fast, modern language; all you need is the right resource! *Get Programming with Go* provides a hands-on introduction to Go language fundamentals, serving as a solid foundation for your future programming projects. You'll master Go syntax, work with types and functions, and explore bigger ideas like state and concurrency, with plenty of exercises to lock in what you learn.

WHAT'S INSIDE

- **Language concepts like slices, interfaces, pointers, and concurrency**
- **Seven capstone projects featuring spacefaring gophers, Mars rovers, ciphers, and simulations**
- **All examples run in the Go Playground — no installation required!**

This book is for anyone familiar with computer programming, as well as anyone with the desire to learn.

Nathan Youngman *organizes the Edmonton Go meetup and is a mentor with Canada Learning Code.* ***Roger Peppé*** *contributes to Go and runs the Newcastle upon Tyne Go meetup.*



***To download their free eBook** in PDF, ePub, and Kindle formats, owners of this book should visit*
manning.com/books/get-programming-with-go

MANNING

US $34.99 | Can $46.99 [Including eBOOK]

FREE EBOOK
See first page

"Perfectly organized for learning Go quickly; especially useful for inexperienced programmers."
—MARIO CARRION
MEREDITH CORPORATION

"Learn by doing! Plenty of examples will help you learn the core of the language and expose you to common Go idioms."
—ULISES FLYNN, NAV

"A great book about Go. Written for beginners but useful for seasoned developers too."
—MIKAËL DAUTREY, ISITIX

"The first rung on successfully climbing the Go ladder."
—JEFF SMITH, AGILIFY

ISBN-13: 978-1-617-29309-2
ISBN-10: 1-617-29309-1
53499
9 781617 293092