3

Composite Data Types

Go offers support for maps and structures, which are composite data types and the main subject of this chapter. The reason that we present them separately from arrays and slices is that both maps and structures are more flexible and powerful than arrays and slices. Each map can use keys of a given predefined data type, whereas structures can group multiple data types and create new data types.

Maps and slices are used for completely different reasons. Arrays and slices are used to store contiguous data and benefit from memory locality and indexing. Maps are useful when you do not need the locality of data but still need a way to reference it in constant time.

The general idea is that if an array or a slice cannot do the job, you might need to look at maps. If a map cannot help you store your data the way you want, then you should consider creating and using a structure—you can also group structures of the same type using arrays or slices. Keep in mind that maps and structures are distinct in their use case. You can easily have a map of structures, as well as an array or slice of structures. However, a structure is useful when you need to combine multiple pieces of logically grouped data and/or variables.

Additionally, the knowledge of this chapter will allow us to read and save data in the CSV format using structures.

Also, we are going to improve the statistics application we originally developed in Chapter 1, A Quick Introduction to Go. The new version of the utility is going to be able to load data from disk, which means that it is no longer needed to hardcode your data or generate random numbers.

This chapter covers:

* Maps
* Structures
* Regular expressions and pattern matching
* Improving the statistics application

Without further ado, let us begin by presenting maps.

Maps

Both arrays and slices limit you to using positive integers as indexes, which start from 0 and cannot have gaps in them—this means that even if you want to put data in the slice element at index 99 only, the slice is still going to occupy 100 elements in memory. Maps are more powerful data structures because they allow you to use indexes of various data types as keys to look up your data, as long as these keys are comparable. Comparable means that Go should be able to tell if two values are equal or which value is bigger (or smaller) than the other.

Although Boolean variables are comparable, it makes no sense to use a bool variable as the key to a map because it only allows for two distinct values. Additionally, although floating point values are comparable, precision issues caused by the internal representation of such values might create bugs and crashes, so you might want to avoid using floating point values as keys to maps.

You might ask, why do we need maps, and what are their advantages? The following list will help clarify things:

* Maps are very versatile. You can even create a database index using a map, which allows you to search and access elements based on a given key or, in more advanced situations, a combination of keys.
* Although this is not always the case, working with maps in Go is fast, as you can access all elements of a map in constant time. Inserting and retrieving elements from a map is a constant time operation.
* Maps are easy to understand, which often leads to clear designs.

You can create a new map variable using either make() or a map literal. Creating a new map with string keys and int values using make() is as simple as writing make(map[string]int) and assigning its return value to a variable. On the other hand, if you decide to create a map using a map literal, you need to write something like the following:

m := map[string]int {

"key1": -1

"key2": 123

}

The map literal version is faster when you want to add data to a map at the time of creation. The previous map literal contains two keys and two values—two pairs in total.

You should make no assumptions about the order of the elements inside a map. Go randomizes keys when iterating over a map—his is done on purpose and is an intentional part of the language design.

You can find the length of a map, which is the number of keys in the map, using the len() function, which also works with arrays and slices; also, you can delete a key and value pair from a map using the delete() function, which accepts two arguments: the name of the map and the name of the key, in that order.

How to tell whether a key exists on a map

You can tell whether a key k exists on a map named aMap by the second return value of the v, ok := aMap[k] statement. If ok is set to true, then k exists, and its value is v. If it does not exist, v will be set to the zero value of its data type, which depends on the definition of the map.

Now, a very important detail: If you try to get the value of a key that does not exist in a map, Go will not complain about it and return the zero value of the data type of the value.

Now, let us discuss a special case where a map variable has the nil value.

Storing to a nil map

You are allowed to assign a map variable to nil. In that case, you will not be able to use that variable until you assign it to a new map variable. Put simply, if you try to store data on a nil map, your program will crash. This is illustrated in the next bit of code, which is the implementation of the main() function of the nilMap.go source file that can be found in the ch03 directory of the GitHub repository of this book:

func main() {

aMap := map[string]int{}

aMap["test"] = 1

This works because aMap points to an existing map, which is the return value of map[string]int{}.

fmt.Println("aMap:", aMap)

aMap = nil

At this point, aMap points to nil, which in Go is a synonym for nothing.

fmt.Println("aMap:", aMap)

if aMap == nil {

fmt.Println("nil map!")

aMap = map[string]int{}

}

Testing whether a map points to nil before using it is a good practice. In this case, if aMap == nil allows us to determine whether we can store a key/value pair to aMap or not—we cannot, and if we try it, the program will crash. We correct that by issuing the aMap = map[string]int{} statement.

aMap["test"] = 1

// This will crash!

aMap = nil

aMap["test"] = 1

}

In this last part of the program, we illustrate how your program will crash if you try to store on a nil map—never use such code in production!

In real-world applications, if a function accepts a map argument, then it should check that the map is not nil before working with it.

Running nilMap.go produces this output:

$ go run nilMap.go

aMap: map[test:1]

aMap: map[]

nil map!

panic: assignment to entry in nil map

goroutine 1 [running]:

main.main()

/Users/mtsouk/Desktop/mGo4th/code/ch03/nilMap.go:21 +0x17c

exit status 2

The reason the program crashed is shown in the program output: panic: assignment to entry in nil map.

Iterating over maps

When for is combined with range, it implements the functionality of foreach loops found in other programming languages and allows you to iterate over all the elements of a map without knowing its size or its keys. In that case, range returns key and value pairs, in that order.

Type the following code and save it as forMaps.go:

package main

import "fmt"

func main() {

aMap := make(map[string]string)

aMap["123"] = "456"

aMap["key"] = "A value"

for key, v := range aMap {

fmt.Println("key:", key, "value:", v)

}

In this case, we use both the key and the value that returned from range.

for \_, v := range aMap {

fmt.Print(" # ", v)

}

fmt.Println()

}

In this case, as we are only interested in the values returned by the map, we ignore the keys.

As you already know, you should make no assumptions about the order that the key and value pairs of a map will be returned in from a for/range loop.

Running forMaps.go produces this output:

$ go run forMaps.go

key: key value: A value

key: 123 value: 456

# 456 # A value

Having covered maps, it is time to learn about Go structures.

Structures

Structures in Go are both very powerful and very popular and are used for organizing and grouping various types of data under the same name. Structures are the more versatile data type in Go—they can even have associated functions, which are called methods.

Structures, as well as other user-defined data types, are usually defined outside the main() function or any other package function so that they can have a global scope and be available to the entire Go package. Therefore, unless you want to make clear that a type is only useful within the current local scope and is not expected to be used elsewhere, you should write the definitions of new data types outside functions.

The type keyword

The type keyword allows you to define new data types or create aliases for existing ones. Therefore, you are allowed to say type myInt int and define a new data type called myInt, which is an alias for int. However, Go considers myInt and int as totally different data types that you cannot compare directly, even though they store the same kind of values. Each structure defines a new data type, hence the use of type.

Defining new structures

When you define a new structure, which is called a struct in the Go documentation, you group a set of values into a single data type, which allows you to pass and receive this set of values as a single entity. A structure has fields, and each field has its own data type, which can even be another structure or a slice of structures. Additionally, as a structure is a new data type, it is defined using the type keyword, followed by the name of the structure, and ending with the struct keyword, which signifies that we are defining a new structure.

The following code defines a new structure named Entry:

type Entry struct {

Name string

Surname string

Year int

}

Although you can embed a structure definition into another structure, it is generally a bad idea and should be avoided. If you even think about doing so, you might need to think about your design decisions. However, it is perfectly acceptable to have existing structs as types inside a struct.

For reasons that will become evident in Chapter 6, Go Packages and Functions, the fields of a structure usually begin with an uppercase letter—this depends on what you want to do with the fields and how their visibility outside of the current package might affect that. The Entry structure has three fields, named Name, Surname, and Year. The first two fields are of the string data type, whereas the last field holds an int value.

These three fields can be accessed with the dot notation as V.Name, V.Surname, and V.Year, where V is the name of the variable holding an instance of the Entry structure. A structure literal named p1 can be defined as p1 := Entry{"Joe", "D.", 2012}.

Two ways exist to work with structure variables. The first one is as regular variables, and the second one is as pointer variables that point to the memory address of a structure. Both ways are equally good and are usually embedded into separate functions, because they allow you to initialize some or all of the fields of structure variables properly and/or do any other tasks you want before using the structure variable.

As a result, there exist two main ways to create a new structure variable using a function. The first one returns a regular structure variable whereas the second one returns a pointer to a structure. Each one of these two ways has two variations. The first variation returns a structure instance that is initialized by the Go compiler, whereas the second variation returns a structure instance that is initialized by the developer.

Last, keep in mind that the order in which you put the fields in the definition of a structure type is significant for the type identity of the defined structure. Put simply, two structures with the same fields will not be considered identical in Go if their fields are not in the same order. This mainly has to do with exchanging data between server and client software because variables of different structures cannot be compared, even if they have the exact same list of fields with the exact data types in the exact same order, as they belong to different data types.

Using the new keyword

Additionally, you can create new structure instances using the new() keyword with statements such as pS := new(Entry). The new() keyword has the following properties:

* It allocates the proper memory space, which depends on the data type, and then it zeroes it.
* It always returns a pointer to the allocated memory.
* It works for all data types except channels and maps.

All these techniques are illustrated in the code that follows. Type the following code in your favorite text editor and save it as structures.go:

package main

import "fmt"

type Entry struct {

Name string

Surname string

Year int

}

// Initialized by Go

func zeroS() Entry {

return Entry{}

}

Now is a good time to remind you of an important Go rule: if no initial value is given to a variable, the Go compiler automatically initializes that variable to the zero value of its data type. For structures, this means that a structure variable without an initial value is initialized to the zero values of the data type of each one of its fields. Therefore, the zeroS() function returns a zero-initialized Entry structure.

// Initialized by the user

func initS(N, S string, Y int) Entry {

if Y < 2000 {

return Entry{Name: N, Surname: S, Year: 2000}

}

return Entry{Name: N, Surname: S, Year: Y}

}

In this case, the user initializes the new structure variable. Additionally, the initS() function checks whether the value of the Year field is smaller than 2000 and acts; if it is smaller than 2000, then the value of the Year field becomes 2000. This condition is specific to the requirements of the application you are developing—what this shows is that the place where you initialize a structure is also good for checking your input.

// Initialized by Go - returns pointer

func zeroPtoS() \*Entry {

t := &Entry{}

return t

}

The zeroPtoS() function returns a pointer to a zero-initialized structure.

// Initialized by the user - returns pointer

func initPtoS(N, S string, Y int) \*Entry {

if len(S) == 0 {

return &Entry{Name: N, Surname: "Unknown", Year: Y}

}

return &Entry{Name: N, Surname: S, Year: Y}

}

The initPtoS() function also returns a pointer to a structure but also checks the length of the user input. Again, this kind of checking is application specific.

func main() {

s1 := zeroS()

p1 := zeroPtoS()

fmt.Println("s1:", s1, "p1:", \*p1)

s2 := initS("Mihalis", "Tsoukalos", 2024)

p2 := initPtoS("Mihalis", "Tsoukalos", 2024)

fmt.Println("s2:", s2, "p2:", \*p2)

fmt.Println("Year:", s1.Year, s2.Year, p1.Year, p2.Year)

pS := new(Entry)

fmt.Println("pS:", pS)

}

The new(Entry) call returns a pointer to an Entry structure. As a rule of thumb, when you have to initialize lots of structure variables, it is considered good practice to create a function for doing so, as this is less error-prone.

Running structures.go creates the following output:

s1: { 0} p1: { 0}

s2: {Mihalis Tsoukalos 2024} p2: {Mihalis Tsoukalos 2024}

Year: 0 2024 0 2024

pS: &{ 0}

As the zero value of a string is the empty string, s1, p1, and pS do not show any data for the Name and Surname fields.

The next subsection shows how to group structures of the same data type and use them as the elements of a slice.

Slices of structures

You can create slices of structures to group and handle multiple structures under a single variable name. However, accessing a field of a given structure requires knowing the exact place of the structure in the slice.

Have a look at the following figure to better understand how a slice of structures works and how you can access the fields of a specific slice element.

A black background with a black square

Description automatically generated

Figure 3.1: A slice of structures

Insert Image B21003\_03\_01

So each slice element is a structure that is accessed using a slice index. Once we select the slice element we want, we can select any one of its fields.

As the whole process can be a little perplexing, the code that follows sheds some light and clarifies things. Type the following code and save it as sliceStruct.go. You can also find it by the same name in the ch03 directory in the GitHub repository of the book.

package main

import (

"fmt"

"strconv"

)

type record struct {

Field1 int

Field2 string

}

func main() {

s := []record{}

for i := 0; i < 10; i++ {

text := "text" + strconv.Itoa(i)

temp := record{Field1: i, Field2: text}

s = append(s, temp)

}

You still need append() to add a new structure to a slice.

// Accessing the fields of the first element

fmt.Println("Index 0:", s[0].Field1, s[0].Field2)

fmt.Println("Number of structures:", len(s))

sum := 0

for \_, k := range s {

sum += k.Field1

}

fmt.Println("Sum:", sum)

}

Running sliceStruct.go produces the following output:

Index 0: 0 text0

Number of structures: 10

Sum: 45

We revisit structures in Chapter 5, where we discuss reflection, as well as Chapter 7, Telling a UNIX System What to Do, where we learn how to work with JSON data using structures. For now, let us discuss regular expressions and pattern matching.

Regular expressions and pattern matching

You might wonder why we are talking about regular expressions and pattern matching in this chapter. The reason is simple. In a while, you will learn how to store and read CSV data from plain text files, and you should be able to tell whether the data is valid or not.

Pattern matching is a technique for searching a string for some set of characters, based on a specific search pattern that is based on regular expressions and grammars.

A regular expression is a sequence of characters that defines a search pattern. Every regular expression is compiled into a recognizer by building a generalized transition diagram called a finite automaton. A finite automaton can be either deterministic or nondeterministic. Nondeterministic means that more than one transition out of a state can be possible for the same input. A recognizer is a program that takes a string x as input and can tell whether x is a sentence of a given language or not.

A grammar is a set of production rules for strings in a formal language—the production rules describe how to create strings from the alphabet of the language which are valid according to the syntax of the language. A grammar does not describe the meaning of a string or what can be done with it in whatever context—it only describes its form. What is important here is to realize that grammars are at the heart of regular expressions because, without a grammar, you cannot define and therefore use a regular expression.

About regexp.Compile and regexp.MustCompile

The Go package responsible for defining regular expressions and performing pattern matching is called regexp. Inside that package exists regexp.Compile() and regexp.MustCompile(), which have similar capabilities.

Both the regexp.MustCompile() and regexp.Compile() functions parse the given regular expression and return a pointer to a regexp.Regexp variable that can be used for matching—regexp.Regexp is the representation of a compiled regular expression. The re.Match() method returns true if the given byte slice matches the re regular expression, which is a regexp.Regexp variable, and false otherwise.

The main and crucial difference between regexp.Compile() and regexp.MustCompile() is that the former returns a \*regexp.Regexp pointer and an error variable, whereas the latter returns a \*regexp.Regexp pointer only. As a result, if there is some kind of error in the parsing of the regular expression, regexp.MustCompile() is going to panic and therefore crash your program!

However, regexp.MustCompile() panicking is not necessarily a bad thing because if a regular expression cannot be parsed, you will know that your expression is invalid early in the process. At the end of the day, it is the developer that decides the overall policy regarding regular expression parsing.

There are times when we want to find only those matches for a pattern that are followed or preceded by another given pattern. These kinds of operations are called lookahead and lookbehind, respectively. Go offers no support for either lookahead or lookbehind and will throw an error message when used. The general syntax of lookahead is X(?=Y), which means, match X only if it is followed by Y. The difference between regexp.Compile() and regexp.MustCompile() is illustrated in the main() function of diffRegExp.go, which is going to be presented in two parts.

func main() {

// This is a raw string literal

var re string = `^.\*(?=.{7,})(?=.\*\d)$`

What is wrong with that particular regular expression? The problem is that it is using lookahead, which is not supported in Go.

The second part is next:

exp1, err := regexp.Compile(re)

if err != nil {

fmt.Println("Error:", err)

}

fmt.Println("RegExp:", exp1)

exp2 := regexp.MustCompile(re)

fmt.Println(exp2)

}

In this second code segment, the use of regexp.Compile() and regexp.MustCompile() is illustrated.

Running diffRegExp.go produces the next output:

$ go run diffRegExp.go

Error: error parsing regexp: invalid or unsupported Perl syntax: `(?=`

RegExp: <nil>

panic: regexp: Compile(`^.\*(?=.{7,})(?=.\*\d)$`): error parsing regexp: invalid or unsupported Perl syntax: `(?=`

goroutine 1 [running]:

regexp.MustCompile({0x100a0c681, 0x15})

/opt/homebrew/Cellar/go/1.20.6/libexec/src/regexp/regexp.go:319 +0xac

main.main()

/Users/mtsouk/Desktop/mGo4th/code/ch03/diffRegExp.go:20 +0xf8

exit status 2

So, in the first case, we know that there is an error in the regular expression because of the return value of regexp.Compile(), whereas when using regexp.MustCompile() with an erroneous regular expression, the program panics and automatically terminates.

The next subsection shows how to define regular expressions.

Go regular expressions

We begin this subsection by presenting some common match patterns used to construct regular expressions.

|  |  |
| --- | --- |
| Expression | Description |
| . | Matches any character |
| \* | Means any number of times—cannot be used on its own |
| ? | Zero or once—cannot be used on its own |
| + | Means one or more times—cannot be used on its own |
| ^ | This denotes the beginning of the line |
| $ | This denotes the end of the line |
| [] | [] is for grouping characters |
| [A-Z] | This means all characters from capital A to capital Z |
| \d | Any digit in 0-9 |
| \D | A non-digit |
| \w | Any word character: [0-9A-Za-z\_] |
| \W | Any non-word character |
| \s | A whitespace character |
| \S | A non-whitespace character |

The characters presented in the previous table are used for constructing and defining the grammar of a regular expression.

Creating separate functions for pattern matching can be handy because it allows you to reuse the functions without worrying about the context of the program.

Keep in mind that although regular expressions and pattern matching look convenient at first, they are the root of lots of bugs. My advice is to use the simplest regular expression that can solve your problem. However, avoiding using regular expressions while still doing your job would be much better in the long run!

About raw string and interpreted string literals

Although we discussed strings in the previous chapter, it is in the definition of the regular expression in diffRegExp.go that we have used a raw string literal for the first time, so let us talk a little bit more about raw string literals, which are included in back quotes instead of double quotes. The advantages of raw string literals are the following:

* They can keep huge amounts of text inside them without the need for control characters, such as \n, for changing lines.
* They are handy when defining regular expressions because you do not need to use backquotes (\) to escape special characters.
* They are used in structure tags, which are explained in Chapter 11, Working with REST APIs.

So, in summary, raw string literals are for storing strings without any escape processing, whereas interpreted string literals are processed when the string is created.

The next subsection presents regular expressions for matching names and surnames.

Matching names and surnames

The presented utility matches names and surnames—according to our definition, these are strings that begin with an uppercase letter and continue with lowercase letters. The input should not contain any numbers or other characters.

The source code of the utility can be found in nameSurRE.go, which is in the ch03 folder. The function that supports the desired functionality is named matchNameSur() and is implemented as follows:

func matchNameSur(s string) bool {

t := []byte(s)

re := regexp.MustCompile(`^[A-Z][a-z]\*$`)

return re.Match(t)

}

The logic of the function is in the `^[A-Z][a-z]\*$` regular expression, where ^ denotes the beginning of a line and $ denotes the end of a line. What the regular expression does is to match anything that begins with an uppercase letter ([A-Z]) and continue with any number of lowercase letters ([a-z]\*). This means that Z is a match, but ZA is not a match because the second letter is in uppercase. Similarly, Jo+ is not a match because it contains the + character.

Running nameSurRE.go with various types of input produces the following output:

$ go run nameSurRE.go Z

true

$ go run nameSurRE.go ZA

false

$ go run nameSurRE.go Mihalis

True

This technique can help you check the validity of user input. The next subsection is about matching integers.

Matching integers

The presented utility matches both signed and unsigned integers—this is implemented in the way we define the regular expression. If we want to match unsigned integers only, then we should replace [-+]? from the regular expression with [+]?.

A better alternative than using a regular expression for matching integer values would have been the use of strconv.Atoi(). As a piece of advice, if you can avoid using regular expressions, prefer the alternative method. However, regular expressions are invaluable when you do not know in advance the kind or the amount of data to expect in the input. In general, regular expressions are invaluable for separating the various parts of the input. Keep in mind that regular expressions are always matching strings, and you can also find digits in strings.

The source code of the utility can be found in intRE.go, which is in the ch03 directory. The matchInt() function that supports the desired functionality is implemented as follows:

func matchInt(s string) bool {

t := []byte(s)

re := regexp.MustCompile(`^[-+]?\d+$`)

return re.Match(t)

}

As before, the logic of the function is found in the regular expression that is used for matching integers, which is `^[-+]?\d+$`. In plain English, what we mean here is that we want to match something that begins with – or +, which is optional (?), and ends with any number of digits (\d+)—it is required that we have at least one digit before the end of the string that is examined ($).

Running intRE.go with various types of input produces the following output:

$ go run intRE.go 123

true

$ go run intRE.go /123

false

$ go run intRE.go +123.2

false

$ go run intRE.go +

false

$ go run intRE.go -123.2

false

Later in this book, you will learn how to test Go code by writing testing functions—for now, we will do most of the testing manually.

Improving the statistics application

It is time to update the statistics application. The new version of the statistics utility has the following improvements:

* It uses functions to simplify the main() function and improve the overall design.
* It can read CSV files that contain the numeric input.

But first, we need to learn how to work with CVS files in Go, which is the subject of the next subsection.

Working with CSV files

Most of the time, you do not want to lose your data or have to begin without any data every time you execute your application. There exist many techniques for doing so—the easiest one is by saving your data locally. A very easy-to-work-with plain text file format is CSV, which is what is explained here and used in the statistics application later on.

The good thing is that Go provides a dedicated package for working with CSV data, named encoding/csv (https://pkg.go.dev/encoding/csv). For the presented utility, both the input and output files are given as command line arguments.

When reading or writing CSV data from disk, everything is considered a string. Therefore, if you have numeric data that you want to treat as such during the reading phase, you might need to convert it to the proper data type on your own.

There exist two very popular Go interfaces, named io.Reader and io.Write, that relate to reading and writing files, respectively. Almost all reading and writing operations in Go use these two interfaces. The use of the same interface for all readers allows readers to share some common characteristics, but most importantly, it allows you to create your own readers and use them anywhere that Go expects an io.Reader reader. The same applies to writers that satisfy the io.Write interface. You are going to learn more about interfaces in Chapter 5, Reflection and Interfaces.

The main tasks that need to be implemented are the following:

* Loading CSV data from disk and putting it into a slice of structures
* Saving data to disk using the CSV format

The encoding/csv package contains functions that can help you read and write CSV files. As we are dealing with small CSV files, we use csv.NewReader(f).ReadAll() to read the entire input file at once. For bigger data files, or if we wanted to check the input or make any changes to the input as we read it, it would have been better to read it line by line using Read() instead of ReadAll().

Go assumes that the CSV file uses the comma character (,) for separating the different fields of each line. Should we wish to change that behavior, we should change the value of the Comma variable of the CSV reader or the writer, depending on the task we want to perform. We change that behavior in the output CSV file, which separates its fields using the tab character.

For reasons of compatibility, it is better if the input and output CSV files use the same field delimiter. We are just using the tab character as the field delimiter in the output file to illustrate the use of the Comma variable.

As working with CSV files is a new topic, there is a separate utility named csvData.go in the ch03 directory of the GitHub repository of this book that illustrates the techniques for reading and writing CSV files. The source code of csvData.go is presented in chunks. First, we present the preamble of csvData.go that contains the import section as well as the Record structure and the myData global variable, which is a slice of Record.

package main

import (

"encoding/csv"

"log"

"os"

)

type Record struct {

Name string

Surname string

Number string

LastAccess string

}

var myData = []Record{}

Then, we present the readCSVFile() function, which reads the plain text file with the CSV data.

func readCSVFile(filepath string) ([][]string, error) {

\_, err := os.Stat(filepath)

if err != nil {

return nil, err

}

f, err := os.Open(filepath)

if err != nil {

return nil, err

}

defer f.Close()

// CSV file read all at once

// lines data type is [][]string

lines, err := csv.NewReader(f).ReadAll()

if err != nil {

return [][]string{}, err

}

return lines, nil

}

Note that we check whether the given file path exists and is associated with a regular file inside the function. There is no right or wrong decision about where to perform that checking—you just have to be consistent. The readCSVFile() function returns a [][]string slice that contains all the lines we have read. Additionally, keep in mind that csv.NewReader() does separate the fields of each input line, which is the main reason for needing a slice with two dimensions to store the input.

After that, we illustrate the writing to a CSV file technique with the help of the saveCSVFile() function.

func saveCSVFile(filepath string) error {

csvfile, err := os.Create(filepath)

if err != nil {

return err

}

defer csvfile.Close()

csvwriter := csv.NewWriter(csvfile)

// Changing the default field delimiter to tab

csvwriter.Comma = '\t'

for \_, row := range myData {

temp := []string{row.Name, row.Surname, row.Number, row.LastAccess}

err = csvwriter.Write(temp)

if err != nil {

return err

}

}

csvwriter.Flush()

return nil

}

Note the change in the default value of csvwriter.Comma to match our needs.

Lastly, we can see the implementation of the main() function.

func main() {

if len(os.Args) != 3 {

log.Println("csvData input output!")

os.Exit(1)

}

input := os.Args[1]

output := os.Args[2]

lines, err := readCSVFile(input)

if err != nil {

log.Println(err)

os.Exit(1)

}

// CSV data is read in columns - each line is a slice

for \_, line := range lines {

temp := Record{

Name: line[0],

Surname: line[1],

Number: line[2],

LastAccess: line[3],

}

myData = append(myData, temp)

log.Println(temp)

}

err = saveCSVFile(output)

if err != nil {

log.Println(err)

os.Exit(1)

}

}

The main() function puts what you have read with readCSVFile() in the myData slice—remember that lines is a slice with two dimensions and that each row in lines is already separated into fields. In this case, each line of input contains four fields. So we process that [][]string slice and put the desired information in the slice of structures (myData).

The contents of the CSV data file used as input are as follows:

$ cat ~/csv.data

Dimitris,Tsoukalos,2101112223,1600665563

Mihalis,Tsoukalos,2109416471,1600665563

Jane,Doe,0800123456,1608559903

Running csvData.go produces the following kind of output:

$ go run csvData.go ~/csv.data /tmp/output.data

{Dimitris Tsoukalos 2101112223 1600665563}

{Mihalis Tsoukalos 2109416471 1600665563}

{Jane Doe 0800123456 1608559903}

The contents of the output CSV file are the following:

$ cat /tmp/output.data

Dimitris Tsoukalos 2101112223 1600665563

Mihalis Tsoukalos 2109416471 1600665563

Jane Doe 0800123456 1608559903

The output.data file uses tab characters to separate the different fields of each record, hence the generated output. The csvData.go utility can be handy for performing conversions between different types of CSV files.

The updated version of the statistics application

In this subsection, we are going to show the updated code of the statistics application. The normalized() function has not changed, so it is not presented again.

The first code excerpt from stats.go is the implementation of the function that reads the CSV file as text and converts it into a slice of float64 values.

func readFile(filepath string) ([]float64, error) {

\_, err := os.Stat(filepath)

if err != nil {

return nil, err

}

f, err := os.Open(filepath)

if err != nil {

return nil, err

}

defer f.Close()

lines, err := csv.NewReader(f).ReadAll()

if err != nil {

return nil, err

}

values := make([]float64, 0)

for \_, line := range lines {

tmp, err := strconv.ParseFloat(line[0], 64)

if err != nil {

log.Println("Error reading:", line[0], err)

continue

}

values = append(values, tmp)

}

return values, nil

}

Once the specified CSV file is read, its data is put into the lines variable. Keep in mind that, in our case, each line in the CSV file has a single field. Nevertheless, lines has two dimensions.

As we want to return a slice of float64 values, we have to convert the [][]string variable into a []float64 variable, which is the purpose of the last for loop. The most important task of the for loop is to make sure that all strings are valid float64 values, in order to put them in the values slice—this is the purpose of the strconv.ParseFloat(line[0], 64) call.

Next, we have the implementation of the function that computes the standard deviation:

func stdDev(x []float64) (float64, float64) {

sum := 0.0

for \_, val := range x {

sum = sum + val

}

meanValue := sum / float64(len(x))

fmt.Printf("Mean value: %.5f\n", meanValue)

// Standard deviation

var squared float64

for i := 0; i < len(x); i++ {

squared = squared + math.Pow((x[i]-meanValue), 2)

}

standardDeviation := math.Sqrt(squared / float64(len(x)))

return meanValue, standardDeviation

}

First, stdDev() computes the sum of all given values and, after that, the mean value of the data. Last, the standard deviation is computed. You can remove the fmt.Printf() call inside the stdDev() function when you are sure that everything works as expected.

Lastly, here is the implementation of main():

func main() {

if len(os.Args) == 1 {

log.Println("Need one argument!")

return

}

file := os.Args[1]

values, err := readFile(file)

if err != nil {

log.Println("Error reading:", file, err)

os.Exit(0)

}

sort.Float64s(values)

fmt.Println("Number of values:", len(values))

fmt.Println("Min:", values[0])

fmt.Println("Max:", values[len(values)-1])

meanValue, standardDeviation := stdDev(values)

fmt.Printf("Standard deviation: %.5f\n", standardDeviation)

normalized := normalize(values, meanValue, standardDeviation)

fmt.Println("Normalized:", normalized)

}

Although the core functionality of the updated version of stats.go is the same as the version developed in the previous chapter, the use of functions simplifies the implementation of main().

Running stats.go produces the following output:

$ go run stats.go csvData.txt

Error reading: a strconv.ParseFloat: parsing "a": invalid syntax

Number of values: 6

Min: -1.2

Max: 3

Mean value: 0.66667

Standard deviation: 1.54883

Normalized: [-1.2053 -1.0761 -0.4305 0.2797 0.9254 1.5065

The previous output shows that csvData.txt contains an invalid line—the contents of csvData.txt are the following:

$ cat csvData.txt

1.1

2.1

-1.2

-1

0

a

3

Despite being much better than the previous version, the new version of the statistics utility is still not perfect. Here is a list of things that can be improved:

* The ability to process multiple CSV data files.
* The ability to sort its output based on a predefined statistics property, such as the mean value when dealing with multiple CSV data files.
* The ability to use JSON records and JSON slices for the data instead of CSV files.

The statistics application will keep improving, starting from Chapter 5, Reflection and Interfaces, where sorting slices with structure elements is implemented.

Summary

In this chapter, we discussed the composite data types of Go, which are maps and structures. Additionally, we talked about working with CSV files as well as using regular expressions and pattern matching. We can now keep our data in proper structures, validate it using regular expressions, and store it in CSV files to achieve data persistency.

Always keep in mind that if you try to get the value of a key that does not exist in a map, Go will not complain about it and return the zero value of the data type of the value.

The next chapter is about Go generics, which is a relatively new Go feature.

Exercises

* Write a Go program that converts an existing array into a map.
* Write a Go program that converts an existing map into two slices—the first slice containing the keys of the map whereas the second one containing the values. The values at index n of the two slices should correspond to a key and value pair that can be found in the original map.
* Make the necessary changes to nameSurRE.go to be able to process multiple command line arguments.
* Change the code of intRE.go to process multiple command line arguments and display totals of true and false results at the end.
* Make changes to csvData.go to separate the fields of a record based on the # character.
* To understand how difficult regular expressions might end up, look on the internet for a regular expression to match email addresses.
* The regexp package includes the MatchString() method. Try to understand its main difference from the Match method and create a working example.
* Write a Go utility that converts os.Args into a slice of structures, with fields for storing the index and the value of each command line argument—you should define the structure that is going to be used.
* Make changes to csvData.go to separate the fields of a record, using a single character that is given as a command line argument.
* Modify the stdDev() function of stats.go, in order to save the mean value of the sample into a global variable and delete the fmt.Printf() call from it.

Additional resources

* The encoding/csv documentation: https://pkg.go.dev/encoding/csv
* The runtime package documentation: https://pkg.go.dev/runtime
* The regexp package documentation: https://pkg.go.dev/regexp