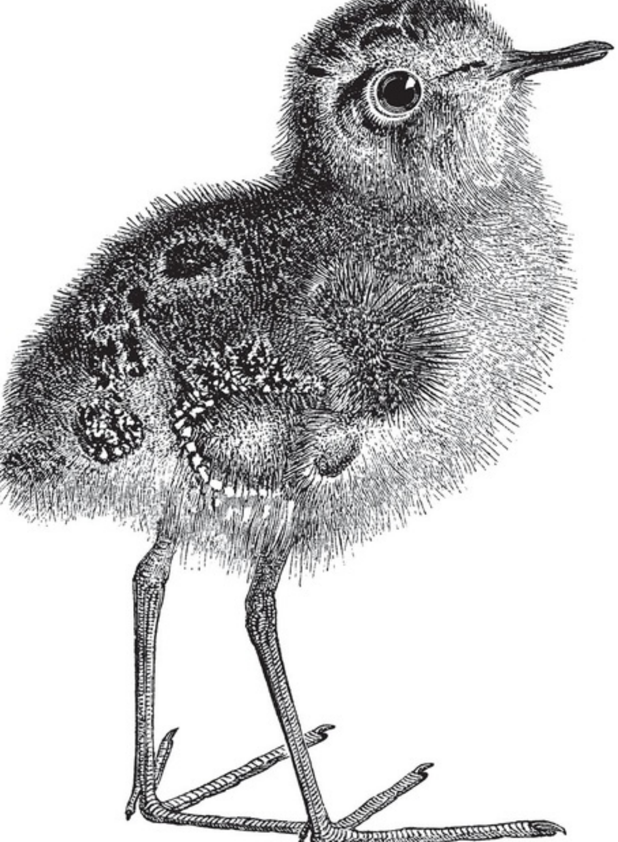
O'REILLY®

Go Cookbook

Expert Solutions for Commonly Needed Go Tasks



Early Release
RAW & UNEDITED

Sau Sheong Chang

Go Cookbook

Expert Solutions for Commonly Needed Go Tasks

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Sau Sheong Chang

Go Cookbook

by Sau Sheong Chang

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Suzanne McQuade
- Development Editor: Shira Evans
- Production Editor: Elizabeth Faerm
- Copyeditor: TO COME
- Proofreader: TO COME
- Indexer: TO COME
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- July 2023: First Edition

Revision History for the Early Release

- 2022-02-01: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098122119 for release details.

978-1-098-12205-8

[TO COME]

Chapter 1. General Input/Output Recipes

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 8th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

1.0 Introduction

Input and output (or more popularly known as I/O) is how a computer communicates with the external world. I/O is a key part of developing software and therefore most programming languages, including Go, has standard libraries that can read from input and write to output. Typical input into a computer refers to the keystrokes from a keyboard or clicks or movement from a mouse, but can also refer to other external sources like a camera or a microphone, or gaming joystick and so on. Output in many cases refer to whatever is shown on the screen (or on the terminal) or printed out on a printer. I/O can also refer to network connections and often also to files.

In this chapter, we’ll be exploring some common Go recipes for managing I/O. We’ll warm up with with some basic I/O recipes, then talk about files in general. In the next few chapters we’ll move on to CSV, followed by JSON and also binary files.

The `io` package is the base package for input and output in Go. It contains the main interfaces for I/O and a few convenient functions. The main and the most commonly used interfaces are `Reader` and `Writer` but there are a number of variants of these like the `ReadWriter`, `TeeReader`, `WriterTo` and many more.

Generally these interfaces are nothing more than a descriptor for functions, for example a struct that is a `Reader` is one that has a `Read` function. A struct that is a `WriterTo` is one with a `WriteTo` function. Some interfaces combine more two or more interfaces for example, the `ReadWriter` combines the `Reader` and `Writer` interfaces and has both the `Read` and `Write` functions.

This chapter explains a bit more about how these interfaces are used.

1.1 Reading from an Input

Problem

You want to read from an input.

Solution

Use the `io.Reader` interface to read from an input.

Discussion

Go uses the `io.Reader` interface to represent the ability to read from an input stream of data. Many packages in the Go standard library as well as 3rd party packages use the `Reader` interface to allow data to be read from it.

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

Any struct that implement the `Read` function is a `Reader`. Let's say you have a reader (a struct that implements the Reader interface). To read data from the reader, you make a slice of bytes and you pass that slice to the `Read` method.

```
bytes = make([]byte, 1024)
reader.Read(bytes)
```

It might look counterintuitive and seems like you would want to read data from `bytes` into the reader, but you're actually reading the data from the reader into `bytes`. Just think of it as the data flowing from left to right, from the reader into `bytes`.

`Read` will only fill up the bytes to its capacity. If you want to read everything from the reader, you can use the `io.ReadAll` function.

```
bytes, err := os.ReadAll(reader)
```

This looks more intuitive because the `ReadAll` reads from the reader passed into the parameter and returns the data into `bytes`. In this case, the data flows from the reader on the right, into the bytes on the lft.

You will also often find functions that expect a reader as an input parameter. Let's say you have a string and you want to pass the string to the function, what can you do? You can create a reader from the string using the `strings.NewReader` function then pass it into the function.

```
str := "My String Data"
reader := strings.NewReader(str)
```

You can now pass `reader` into functions that expect a reader.

1.2 Writing to an Output

Problem

You want to write to an output.

Solution

Use the `io.Writer` interface to write to an output.

Discussion

The interface `io.Writer` works the same way as `io.Reader`.

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

When you call `Write` on an `io.Writer` you are writing the bytes to the underlying data stream.

```
bytes = []byte("Hello World")
writer.Write(bytes)
```

You might notice that this method calling pattern is the reverse of `io.Reader` in recipe 8.1. In `Reader` you call the `Read` method to read from the struct into the `bytes` variable, whereas here you call the `Write` method to write from the `bytes` variable into the struct. In this case, the data flows from right to left, from `bytes` into the writer.

A common pattern in Go is for a function to take in a writer as a parameter. The function then calls the `Write` function on the writer, and later you can extract the data from writer. Let's take a look at an example of this.

```
var buf bytes.Buffer
fmt.Fprintf(&buf, "Hello %s", "World")
s := buf.String() // s == "Hello World"
```

The `bytes.Buffer` struct is a `Writer` (it implements the `Write` function) so you can easily create one, and pass it to the `fmt.Fprintf` function, which takes in an `io.Writer` as its first parameter. The `fmt.Fprintf` function writes data on to the buffer and you can extract the data out from it later.

This pattern of using a writer to pass data around by writing to it, then extracting it out later is quite common in Go. Another example is in the HTTP handlers with the `http.ResponseWriter`.

```go
func myHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]bytes("Hello World"))
}
```

Here, we write to the `ResponseWriter` and the data will be taken as input to be sent back to the browser.

1.3 Copying from a Reader to a Writer

Problem

You want to copy from a reader to a writer.

Solution

Use the `io.Copy` function to copy from a reader to a writer.

Discussion

Sometimes we read from a reader because we want to write it to a writer. The process can take a few steps to read everything from a reader into buffer then write it out to the writer again. Instead of doing this, we can use the `io.Copy` function instead. The `io.Copy` function takes from a reader and writes to a writer all in one function.

Let's see how `io.Copy` can be used. We want to download a file, so we use `http.Get` to get a reader, which we read and then we use `os.WriteFile` to write to a file.

```go
// using a random 1MB test file
var url string =
"http://speedtest.ftp.otenet.gr/files/test1Mb.db"

func readWrite() {
        r, err := http.Get(url)
        if err != nil {
                log.Println("Cannot get from URL", err)
        }
        defer r.Body.Close()
```

```
        data, _ := os.ReadAll(r.Body)
        os.WriteFile("rw.data", data, 0755)
}
```

When we use `http.Get` to download a file we get a `http.Response` struct back. The content of the file is in the `Body` variable of the `http.Response` struct, which is a `io.ReadCloser`. A `ReadCloser` is just an interface that groups a `Reader` and a `Closer` so we here we can treat it just like a reader. We use the `os.ReadAll` function to read the data from `Body` and then `os.WriteFile` to write it to file.

That's simple enough but let's take a look at the performance of the function. We use the benchmarking capabilities that's part of the standard Go tools to do this. First we create a test file, just like any other test files.

```
package main

import "testing"

func BenchmarkReadWrite(b *testing.B) {
        readWrite()
}
```

In this test file instead of a function that starts with `Testxxx` we create a function that starts with `Benchmarkxxx` which takes in a parameter `b` that is a reference to `testing.B`.

The benchmark function is very simple, we just call our `readWrite` function. Let's run it from the command line and see how we perform.

```
$ go test -bench=. -benchmem
```

We use the `-bench=.` flag telling Go to run all the benchmark tests and `-benchmem` flag to show memory benchmarks. This is what you should see.

```
goos: darwin
goarch: amd64
pkg: github.com/sausheong/go-cookbook/io/copy
cpu: Intel(R) Core(TM) i7-7920HQ CPU @ 3.10GHz
BenchmarkReadWrite-8          1       1998957055 ns/op
5892440 B/op
```

```
219 allocs/op
PASS
ok      github.com/sausheong/go-cookbook/io/copy        2.327s
```

We ran a benchmark test for a function that downloaded a 1 MB file. The test only ran one time, and it took 1.99 seconds. It also took 5.89 MB of memory and 219 distinct memory allocations.

As you can see, it's quite an expensive operation just to download a 1 MB file. After all, it takes almost 6 MB of memory to download a 1 MB file. Alternatively we can use `io.Copy` to do pretty much the same thing for a lot less memory.

```
func copy() {
	r, err := http.Get(url)
	if err != nil {
		log.Println("Cannot get from URL", err)
	}
	defer r.Body.Close()
	file, _ := os.Create("copy.data")
	defer file.Close()
	writer := bufio.NewWriter(file)
	io.Copy(writer, r.Body)
	writer.Flush()
}
```

First, we need to create a file for the data, here using `os.Create`. Next we create a buffered writer using `bufio.NewWriter`, wrapping around the file. This will be used in the `Copy` function, copying the contents of the response `Body` into the buffered writer. Finally we flush the writer's buffers and make the underlying writer write to the file.

If you run this, `copy` function it works the same way, but how does the performance compare? Let's go back to our benchmark and add another benchmark function for this `copy` function.

```
package main

import "testing"

func BenchmarkReadWrite(b *testing.B) {
	readWrite()
```

```
}

func BenchmarkCopy(b *testing.B) {
        copy()
}
```

We run the benchmark again and this is what you should see.

```
goos: darwin
goarch: amd64
pkg: github.com/sausheong/go-cookbook/io/copy
cpu: Intel(R) Core(TM) i7-7920HQ CPU @ 3.10GHz
BenchmarkReadWrite-8          1       2543665782 ns/op
5895360 B/op
227 allocs/op
BenchmarkCopy-8               1       1426656774 ns/op
42592 B/op
61 allocs/op
PASS
ok      github.com/sausheong/go-cookbook/io/copy        4.103s
```

This time the `readWrite` function took 2.54 seconds, used 5.89 MB of memory and did 227 memory allocations. The `copy` function however only too 1.43 seconds, used 42.6 kB of memory and did 61 memory allocations.

The `copy` function is around 80% faster, but uses only a fraction (less than 1%) of the memory. With really large files, if you're using the `os.ReadAll` and `os.WriteFile` you might run out of memory quickly.

1.4 Reading from a Text File

Problem

You want to read a text file into memory.

Solution

You can usethe `os.Open` function to open the file, followed by `Read` on the file. Alternatively you can also use the simpler `os.ReadFile` function to do it in a single function call.

Discussion

Reading and writing to the filesystem are one of the basic things a programming language needs to do. Of course you can always store in memory but sooner or later if you need to persist the data beyond a shutdown you need to store it somewhere. There are a number of ways that data can be persistent but the most commonly accessible is probably to the local filesystem.

Read everything at one go

The easiest way to read a text file is to use `os.ReadFile`. Let's say we want to read from a text file named `data.txt`.

```
hello world!
```

To read the file, just give the name of the file as a parameter to `os.ReadFile` and you're done!

```
data, err := os.ReadFile("data.txt")
if err != nil {
  log.Println("Cannot read file:", err)
}
fmt.Println(string(data))
```

This will print out `hello world!`.

Opening a file and reading from it

Reading a file by opening it and then doing a read on it is more flexible but takes a few more steps. First, you need to open the file.

```
// open the file
file, err := os.Open("data.txt")
if err != nil {
  log.Println("Cannot open file:", err)
}
// close the file when we are done with it
defer file.Close()
```

This can be done using `os.Open`, which returns a `File` struct in read-only. If you want to open it in different modes, you can use `os.OpenFile`. It's good practice to set up the file for closing using the `defer` keyword, which will close the file just before the function returns.

Next, we need to create a byte array to store the data.

```
// get some info from the file
stat, err := file.Stat()
if err != nil {
  log.Println("Cannot read file stats:", err)
}
// create the byte array to store the read data
data := make([]byte, stat.Size())
```

To do this, we need to know how large the byte array should be, and that should be the size of the file. We use the `Stat` method on the file to get a `FileInfo` struct, which we can call the `Size` method to get the size of the file.

Once we have the byte array, we can pass it as a parameter to the `Read` method on the file struct.

```
// read the file
bytes, err := file.Read(data)
if err != nil {
  log.Println("Cannot read file:", err)
}
fmt.Printf("Read %d bytes from file\n", bytes)
fmt.Println(string(data))
```

This will store the read data into the byte array and return the number of bytes read. If all goes well you should see something like this from the output.

```
Read 13 bytes from file
Hello World!
```

There are a few more steps, but you have the flexibility of reading parts of the whole document and you can also do other stuff in between opening the file and reading it.

1.5 Writing to a Text File

Problem

You want to write data a text file.

Solution

You can use the `os.Open` function to open the file, followed by `Write` on the file. Alternatively you can use the `os.WriteFile` function to do it in a single function call.

Discussion

Just as in reading a file, there are a couple of ways of writing to a file.

Writing to a file at one go

Given the data, you can write to a file at one go using `os.WriteFile`.

```
data := []byte("Hello World!\n")

err := os.WriteFile("data.txt", data, 0644)
if err != nil {
  log.Println("Cannot write to file:", err)
}
```

The first parameter is the name of the file, the data is in a byte array and the final parameter is the Unix file permissions you want to give to the file. If the file doesn’t exist, this will create a new file. If it exists, it will remove all the data in the file and write the new data into it, but without changing the permissions.

Creating a file and writing to it

Writing to a file by creating the file and then writing to it is a bit more involved but it's also more flexible. First, you need to create or open a file using the `os.Create` function.

```
data := []byte("Hello World!\n")
// write to file and read from file using the File struct
file, err := os.Create("data.txt")
if err != nil {
  log.Println("Cannot create file:", err)
}
defer file.Close()
```

This will create a new file with the given name and mode `0666` if the file doesn't exist. If the file exists, this will remove all the data in it. As before you would want to set up the file to be closed at the end of the function, using the `defer` keyword.

Once you have the file you can write to it directly using the `Write` method and passing it the byte array with the data.

```
bytes, err := file.Write(data)
if err != nil {
  log.Println("Cannot write to file:", err)
}
fmt.Printf("Wrote %d bytes to file\n", bytes)
```

This will return the number of bytes that was written to the file. As before while it takes a few more steps, breaking up the steps between creating a file and writing to it gives you more flexibility to write in smaller chunks instead of everything at once.

1.6 Using a Temporary File

Problem

You want to create a temporary file for use and depose of it afterwards.

Solution

Use the `os.CreateTemp` function to create a temporary file, and then remove it once you don't need it anymore.

Discussion

A temporary file is a file that's created to store data temporarily while the program is doing something. It's meant to be deleted or copied to permanent storage once the task is done. In Go, we can use `os.CreateTemp` function to create a temporary file. Then after that we can remove it.

Different operating systems store their temporary files in different places. Regardless where it is, Go will let you know where it is using the `os.TempDir` function.

```
fmt.Println(os.TempDir())
```

We need to know because the temp files created by `os.CreateTemp` will be created there. Normally we wouldn't care, but because we're trying to analyse step by step how the temp files get created, we want to know exactly where it is. When we execute this statement, we should see something like this.

```
/var/folders/nj/2xd4ssp94zz41gnvsyvth38m0000gn/T/
```

This is the directory that your computer tells Go (and some other programs) to use as a temporary directory. We can use this directly or we can create our own directory here using the `os.MkdirTemp` function.

```
tmpdir, err := os.MkdirTemp(os.TempDir(), "mytmpdir_*")
if err != nil {
        log.Println("Cannot create temp directory:", err)
}
defer os.RemoveAll(tmpdir)
```

The first parameter to `os.MkdirTemp` is the temporary directory and the second parameter is a pattern string. The function will apply a random string to replace the `*` in the pattern string. It is also a good practise to defer the cleaning up of the temporary directory by remove it using `os.RemoveAll`.

Next we're creating the actual temporary file using `os.CreateTemp`, passing it the temporary directory we just created and also a pattern string for the file name, which works the same as as the temporary directory.

```
tmpfile, err := os.CreateTemp(tmpdir, "mytmp_*")
if err != nil {
        log.Println("Cannot create temp file:", err)
}
```

With that, we have a file and everything else works the same way as any other file.

```
data := []byte("Some random stuff for the temporary file")
_, err = tmpfile.Write(data)
if err != nil {
        log.Println("Cannot write to temp file:", err)
}
err = tmpfile.Close()
if err != nil {
        log.Println("Cannot close temp file:", err)
}
```

If you didn't choose to put your temporary files into a separate directory (which you delete and also everything in it when you're done), you can use `os.Remove` with the temporary file name like this.

```
defer os.Remove(tmpfile.Name())
```

Chapter 2. CSV Recipes

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

2.0 Introduction

The CSV format is a file format in which tabular data (numbers and text) can be easily written and read in a text editor. CSV is widely supported, and most spreadsheet programs, such as Microsoft Excel and Apple Numbers, support CSV. Consequently, many programming languages, including Go, come with libraries that produce and consume the data in CSV files.

It might come as a surprise to you that CSV has been around for almost 50 years. The IBM Fortran compiler supported it in OS/360 back in 1972. If you're not quite sure that that is, OS/360 is the batch processing operating system developed by IBM for their System/360 mainframe computer. So yes, one of the first uses for CSV was for Fortran in an IBM mainframe computer.

CSV (comma separated values) is not very well standardized and not all CSV formats are separated by commas either. Sometimes it can be a tab or a semicolon or other delimiters. However, there is a RFC specification for CSV — the RFC 4180 though not everyone follows that standard.

The Go standard library has an `encoding/csv` package that supports RFC 4180 and helps us to read and write CSV.

2.1 Reading a CSV File

Problem

You want to read a CSV file into memory for use.

Solution

Use the `encoding/csv` package and `csv.ReadAll` to read all data in the CSV file into a 2 dimensional array of strings.

Discussion

Let's say you have a file like this:

```
id,first_name,last_name,email
1,Sausheong,Chang,sausheong@email.com
2,John,Doe,john@email.com
```

The first row is the header, the next 2 rows are data for the user. Here's the code to open the file and read it into the 2 dimensional array of strings.

```
file, err := os.Open("users.csv")
if err != nil {
 log.Println("Cannot open CSV file:", err)
}
defer file.Close()
reader := csv.NewReader(file)
rows, err := reader.ReadAll()
if err != nil {
 log.Println("Cannot read CSV file:", err)
}
```

First, we open the file using `os.Open`. This creates an `os.File` struct (which is an `io.Reader`) that we can use as a parameter to `csv.NewReader`. The `csv.NewReader` creates a new `csv.Reader` struct that can be used to read data from the CSV file. With this CSV reader, we can use `ReadAll` to read all the data in the file and return a 2D array of strings [][]string.

A 2 dimensional array of strings? You might be surprised, what if the CSV row item is an integer? Or a boolean or any other types? You should remember CSV files are text files, so there is really no way for you to differentiate if a value is anything else other than a string. In other words, all values are assumed to be string, and if you think otherwise you need to cast it to something else.

Unmarshalling CSV data into structs

Problem

You want to unmarshal CSV data into structs instead of a 2-dimensional array of strings.

Solution

First read the CSV into a 2-dimensional array of strings then store it into structs.

Discussion

For some other formats like JSON or XML, it's common to unmarshal the data read from file (or anywhere) into structs. You can also do this in CSV though you need to do a bit more work.

Let's say you want to put the data into a User struct.

```
type User struct {
   Id          int
   firstName   string
   lastName    string
   email       string
}
```

If you want to unmarshal the data in the 2D array of strings to the User struct, you need to convert each item yourself.

```
var users []User
for _, row := range rows {
 id, _ := strconv.ParseInt(row[0], 0, 0)
```

```
    user := User{Id: int(id),
      firstName: row[1],
      lastName:  row[2],
      email:     row[3],
    }
    users = append(users, user)
  }
```

In the example above, because the user ID is an integer, I used `strconv.ParseInt` to convert the string into integer before using it to create the User struct.

You see that at the end of the for loop you will have an array of User structs. If you print that out, this is what you should see.

```
{0  first_name  last_name  email}
{1  Sausheong  Chang  sausheong@email.com}
{2  John  Doe  john@email.com}
```

2.2 Removing the Header Line

Problem

If your CSV file has a line of headers that are column labels, you will get that as well in your returned 2 dimensional array of strings or array of structs. You want to remove it.

Solution

Read the first line using `Read` and then continue reading the rest.

Discussion

When you use `Read` on the reader, you will read the first line and then move the cursor to the next line. If you use `ReadAll` afterwards, you can read the rest of the file into the rows that you want.

```
file, err := os.Open("users.csv")
if err != nil {
```

```
  log.Println("Cannot open CSV file:", err)
 }
 defer file.Close()
 reader := csv.NewReader(file)
 reader.Read() // use Read to remove the first line
 rows, err := reader.ReadAll()
 if err != nil {
  log.Println("Cannot read CSV file:", err)
 }
```

This will give us something like this:

```
{1   Sausheong   Chang   sausheong@email.com}
{2   John   Doe   john@email.com}
```

2.3 Using Different Delimiters

Problem

CSV doesn't necessarily need to use commas as delimiters. You want to read a CSV file which has delimiter that is not a comma.

Solution

Set the `Comma` variable in the `csv.Reader` struct to the delimiter used in the file and read as before.

Discussion

Let's say the file we want to read has semi-colons as delimiters.

```
id;first_name;last_name;email
1;Sausheong;Chang;sausheong@email.com
2;John;Doe;john@email.com
```

What we just need to do is the set the `Comma` in the `csv.Reader` struct we created earlier and you read the file as before.

```go
file, err := os.Open("users2.csv")
if err != nil {
  log.Println("Cannot open CSV file:", err)
}
defer file.Close()
reader := csv.NewReader(file)
reader.Comma = ';' // change Comma to the delimiter in the file
rows, err := reader.ReadAll()
if err != nil {
  log.Println("Cannot read CSV file:", err)
}
```

2.4 Ignoring Rows

Problem

You want to ignore certain rows when reading the CSV file.

Solution

Use comments in the file to indicate the rows to be ignored. Then enable coding in the `csv.Reader` and read the file as before.

Discussion

Let's say you want to ignore certain rows; what you'd like to do is simply comment those rows out. Well, in CSV you can't because comments are not in the standard. However with the Go `encoding/csv` package you can specify a comment rune, which if you place at the beginning of the row, ignores the entire row.

So say you have this CSV file.

```
id,first_name,last_name,email
1,Sausheong,Chang,sausheong@email.com
# 2,John,Doe,john@email.com
```

To enable commenting, just set the Comment variable in the csv.Reader struct that we got from csv.NewReader.

```
file, err := os.Open("users.csv")
if err != nil {
  log.Println("Cannot open CSV file:", err)
}
defer file.Close()
reader := csv.NewReader(file)
reader.Comment = '#' // lines that start with this will be
ignored
rows, err := reader.ReadAll()
if err != nil {
  log.Println("Cannot read CSV file:", err)
}
```

When you run this, you'll see:

```
{0 first_name last_name email}
{1 Sausheong Chang sausheong@email.com}
```

2.5 Writing CSV Files

Problem

You want to write data from memory into a CSV file.

Solution

Use the `encoding/csv` package and `csv.Writer` to write to file.

Discussion

We had fun reading CSV files, now we have to write one. Writing is quite simliar to reading. First you need to create a file (an `io.Writer`).

```
file, err := os.Create("new_users.csv")
if err != nil {
  log.Println("Cannot create CSV file:", err)
}
defer file.Close()
```

The data to write to the file needs to be in a 2 dimensional array of strings. Remember, if you don't have the data as a string, just convert it into a string before you do this. Create a `csv.Writer` struct with the file. After that you can call `WriteAll` on the writer and the file will be created. This writes all the data in your 2 dimensional string array into the file.

```
data := [][]string{
 {"id", "first_name", "last_name", "email"},
 {"1", "Sausheong", "Chang", "sausheong@email.com"},
 {"2", "John", "Doe", "john@email.com"},
}
writer := csv.NewWriter(file)
err = writer.WriteAll(data)
if err != nil {
 log.Println("Cannot write to CSV file:", err)
}
```

2.6 Writing to File One Row at a Time

Problem

Instead of writing everything in our 2 dimensional string, we want to write to the file one row at a time.

Solution

Use the `Write` method on `csv.Writer` to write a single row/

Discussion

Writing to file one row at a time is pretty much the same, except you will want to iterate the 2 dimensional array of strings to get each row and then call `Write`, passing that row. You will also need to call `Flush` whenever you want to write the buffered data to the `Writer` (the file). In the example above I called `Flush` after I have written all the data to the writer, but that's because I don't have a lot of data. If you have a lot of rows, you would probably want to flush the data to the file once in a while. To check if there's any problems with writing or flushing, you can call `Error`.

```
writer := csv.NewWriter(file)
for _, row := range data {
 err = writer.Write(row)
 if err != nil {
  log.Println("Cannot write to CSV file:", err)
 }
}
writer.Flush()
```