S00

Change the length of the byte slice, from:

```
// TODO: Change a single character in this program so the complete file is read and printed
...
b := make([]byte, 11)
...
```

e.g. to 31:

```
...
b := make([]byte, 31)
...
```

Any larger number will do as well. The length of the byte slice is the space that we allow the read method to fill. If this space is too small, we won't be able to read the whole file.

S01

We can shorten these lines:

```
    // TODO: We don't need to loop manually, there is a helper function for that.
    // TODO: Replace the next 10 lines with 5 that do the same.
    var contents []byte
    for {
        b := make([]byte, 8)
        _, err := file.Read(b)
        if err == io.EOF {
            break
        }
        contents = append(contents, b...)
    }
    fmt.Println(string(contents))
```

by using io.ReadAll:

```
    b, err := ioutil.ReadAll(file)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(string(b))
```

While io.ReadAll is useful, it is sometimes overused. Why is that? Often one wants to read something, process it and then write it somewhere. Imagine a HTTP request body, that we want to read, then preprocess and then maybe write to a file. io.ReadAll would consume the *whole data at once*. But for example in the case of a large file upload, there is seldom a reason, why we would need to load the whole file into memory before writing it to disk. There are other ways to solve this problem, which are both more efficient and elegant.

However, io.ReadAll is in the standard library and has perfectly fine use cases, too.

Noteworthy: EOF will not be reported by `ioutil.ReadAll` as the purpose of the method is to consume the reader as a whole:

> ReadAll reads from r until an error or EOF and returns the data it read. A successful call returns err == nil, not err == EOF. Because ReadAll is defined to read from src until EOF, it does not treat an EOF from Read as an error to be reported.

S02

Use: io.Copy and os.Stdout.

```
// TODO: Write output to Stdout, without using a byte slice (3 lines, including error h
if _, err := io.Copy(os.Stdout, file); err != nil {
    log.Fatal(err)
}
```

The importance of io.Copy can hardly be overstated:

> Copy copies from src to dst until either EOF is reached on src or an error occurs. It returns the number of bytes copied and the first error encountered while copying, if any.

Internally, io.Copy uses a buffer in an essential sense:

> In computer science, a data buffer (or just buffer) is a region of a physical memory storage used to temporarily store data while it is being *moved from one place to another*.

Everywhere, where readers and writers need to connect, io.Copy can be used. As a first example, here we read from a file and write to one of the standard streams.

We will see the helpful io.Copy over and over again.

S03

Use os.Stdout and os.Stdin.

```
// TODO: Read input from standard input and pass it to standard output,
// TODO: without using a byte slice (3 lines).
if _, err := io.Copy(os.Stdout, os.Stdin); err != nil {
```

```
    log.Fatal(err)
}
```

Here, we have the essence of a filter, namely a program, that works with streams, but does not change the stream at all. One can be reminded of the identify function.

S04

Use gzip.Reader.

> A gzip.Reader is an io.Reader that can be read to retrieve uncompressed data from a gzip-format compressed file.

```
// TODO: Read gzip compressed input from standard input and print it to standard output
// TODO: without using a byte slice (7 lines).
r, err := gzip.NewReader(os.Stdin)
if err != nil {
    log.Fatal(err)
}
if _, err := io.Copy(os.Stdout, r); err != nil {
    log.Fatal(err)
}
```

A filter, that decompresses data read from standard input. As soon we get to io.Copy, a decompressed stream has the same *shape* as any other type that implements io.Reader.

S05

Go comes with an image package in the standard library, which implements a basic 2-D image support.

> The fundamental interface is called Image.

There is a Decode method, that takes a reader and turn it into an Image.

In turn, the concrete image subpackages implement an Encode method, which take an io.Writer and an Image as an argument.

```
// TODO: Read the image, encode the image (5 lines).
img, _, err := image.Decode(r)
if err != nil {
    return err
}
return jpeg.Encode(w, img, nil)
```

This snippet takes an arbitrary reader (e.g. standard input) and turns it into an image. The encoding methods are indifferent to the data sink, as long as they implement io.Writer.

S06

The package encoding/json supports handling streams with json.Decoder:

```
// TODO: Unmarshal from standard input into a record struct (4 lines).
var rec record
if err := json.NewDecoder(os.Stdin).Decode(&rec); err != nil {
    log.Fatal(err)
}
```

The decoder takes an io.Reader and decodes the read bytes into the given values. This is useful, if your have a possible large number of values you want to decode, one at a time. The whole stream might not fit into memory at once, but the records, that make up the stream can be processed - one by one.

S07a

Example for a utility reader: A io.LimitReader modifies a reader, so that it returns EOF after (at most) a fixed number of bytes.

```
// TODO: Only read the first 27 bytes from standard input (3/6 lines).
if _, err := io.Copy(os.Stdout, io.LimitReader(os.Stdin, 27)); err != nil {
    log.Fatal(err)
}
```

Where can this be useful? Imagine a HTTP response, where the header specifies the content length and you want to limit the reading of the HTTP response body to the number of bytes indicated in the header.

Alternative implementation with a byte slice:

```
// TODO: Only read the first 27 bytes from standard input (3/6 lines).
p := make([]byte, 27)
_, err := os.Stdin.Read(p)
if err != nil {
    log.Fatal(err)
}
fmt.Printf(string(p))
```

Yet another implementation, using io.CopyN:

```
// TODO: Only read the first 27 bytes from standard input (3/6 lines).
if _, err := io.CopyN(os.Stdout, os.Stdin, 27); err != nil {
```

```
        log.Fatal(err)
    }
```

S07b

A io.SectionReader wraps seek and read operations. We skip 5 bytes, then read 9 bytes, which should yield the desired string.

We also see that strings can be turned into readers, too.

```
    // TODO: Print the string "io.Reader" to stdout (4 lines).
    s := io.NewSectionReader(r, 5, 9)
    if _, err := io.Copy(os.Stdout, s); err != nil {
        log.Fatal(err)
    }
```

Where can this be useful? Imagine a binary file format, that keeps information in various parts of the file and maybe has an index to these sections in a header.

S08

Here, we use io.ReadFull, which will reads the exactly the size of the buffer from the reader.

> ReadFull reads exactly len(buf) bytes from r into buf. It returns the number of bytes copied and an error if fewer bytes were read. The error is EOF only if no bytes were read.

```
    // TODO: Read the first 7 bytes of the string into a byte slice, then print to stdout (
    b := make([]byte, 7)
    if _, err := io.ReadFull(r, b); err != nil {
        log.Fatal(err)
    }
    fmt.Println(string(b))
```

This is a variation of limited reading. Here the limitation is controlled by the size of the byte slice.

S09

We could apply any of the limiting techniques. Here is an example with io.CopyN:

```
    // TODO: Copy 12 byte from random source into the encoder (3 lines).
    if _, err := io.CopyN(encoder, r, 12); err != nil {
        log.Fatal(err)
    }
```

If you vary the random seed from call to call, this snippet can serve as a simple version of a password generator.

S10

Another example for io.Copy. Here, the destination is a writer, that prettifies tabular data.

```go
// TODO: Read tabulated data from standard in and write it to the tabwriter (3 lines).
if _, err := io.Copy(w, os.Stdin); err != nil {
    log.Fatal(err)
}
```

S11

All done.

S12

You can combine any number of readers with io.MultiReader.

```go
// TODO: Read from these four readers and write to standard output (4 lines).
rs := []io.Reader{
    strings.NewReader("Hello\n"),
    strings.NewReader("Gopher\n"),
    strings.NewReader("World\n"),
    strings.NewReader("!\n"),
}
r := io.MultiReader(rs...)
if _, err := io.Copy(os.Stdout, r); err != nil {
    log.Fatal(err)
}
```

S13

The counterpart to io.MultiReader is io.MultiWriter. It is similar to the Unix tee command.

```go
// TODO: Write to both, the file and standard output (4 lines).
w := io.MultiWriter(file, os.Stdout)
if _, err := fmt.Fprintf(w, "SPQR\n"); err != nil {
    log.Fatal(err)
}
```

S14

Fscan belongs to a family of functions, which can be considered the opposite of formatted output: They scan formatted text to yield values.

```
// TODO: Read an int, a float and a string from standard input (3 lines).
if _, err := fmt.Fscan(os.Stdin, &i, &f, &s); err != nil {
    log.Fatal(err)
}
```

S15

Buffers are versatile types. The bytes.Buffer is a variable-sized buffer of bytes with Read and Write methods. You can a read a single byte, bytes, runes or a string from it. Writing is analogue.

```
// TODO: Read one byte at a time from the buffer and print the hex value on stdout (10
for {
    b, err := buf.ReadByte()
    if err == io.EOF {
        break
    }
    if err != nil {
        log.Fatal(err)
    }
    fmt.Fprintf(os.Stdout, "%x\n", b)
}
```

Here, we read one byte after another. We first check for io.EOF, so we can break the loop accordingly. Any other error still needs to be handled. Finally, we use a format verb to format the integer value in base 16, with lower-case letters for a-f.

S16

The exec.Cmd struct contains fields for the standard streams, namely `Stdin` of type io.Reader and `Stdout` and `Stderr` or type io.Writer. Since bytes.Buffer is an io.Writer we can connect the standard output of a command directly with a bytes.Buffer.

```
// TODO: Stream output of command into the buffer (4 lines).
cmd.Stdout = &buf
if err := cmd.Run(); err != nil {
    log.Fatal(err)
}
```

Imagine, you want to wrap a legacy command line application with a nice Go API. By controlling the input, output and error stream of the application you have basic control over the application and you can start parsing and interpreting the command output into Go structures.

S17

A urgent request.

Imagine you get a urgent request to analyze some image data. It's compressed. You need to find the distribution of the "red" values in an image and create report in form of a pretty table.

This example is short, about 20 lines of code and uses readers and writers all over the place:

- first we read from standard input
- we decompress the data on the fly with a gzip
- the image decoding takes a reader
- we use a formatter, that works with a writer
- we use a buffer to temporarily store tab separated values
- we use a tabwriter to prettify the data
- we write the final report to standard output

We see, how we can build more complex filters from simple parts.

S18a

We find another important value that is an io.Reader: the body of an http.Response. It is actually a ReadCloser, a type that can be read from and closed.

```
// TODO: Like curl, print the response body to standard output (4 lines).
defer resp.Body.Close()
if _, err := io.Copy(os.Stdout, resp.Body); err != nil {
    log.Fatal(err)
}
```

With the familiar io.Copy, we have a simple curl-like program.

While this program would work without the line with the defer statement, for serious program you should always close the response body.

From the http.Client documentation:

> Caller should close resp.Body when done reading from it.

S18b

HTTP is a text based protocol. Instead of using the http.Get we craft a similar request ourselves. We write the request string:

```
GET / HTTP/1.0\r\n\r\n
```

to the connection. After that we try to read from the connection and print the result to standard output.

```go
    // TODO: Send a GET request, read the reponse and print it to standard output (6 lines)
    if _, err := io.WriteString(conn, "GET / HTTP/1.0\r\n\r\n"); err != nil {
        log.Fatal(err)
    }
    if _, err := io.Copy(os.Stdout, conn); err != nil {
        log.Fatal(err)
    }
```

S19

No TODO.

Example for a custom type derived from a file: atomicfile. We can write to an atomic file just like any other file:

```go
    if _, err := io.WriteString(file, "Atomic gopher.\n"); err != nil {
        log.Fatal(err)
    }
```

But the semantics are slightly different. Instead of writing to the file directly, all content is written to a temporary file first. The real file is only created (by renaming the temporary file, which is an atomic operation on many operating systems) when the file is closed.

S20

This is our first own io.Reader implementation. It does one thing: it says there is nothing to read. Hence the name *Empty*.

```go
// TODO: Implement io.Reader interface. Always return EOF (3 lines).
func (r *Empty) Read(p []byte) (n int, err error) {
    return 0, io.EOF
}
```

Even this is very limited in functionality, you can use this type anywhere, where you can use an io.Reader, e.g. in io.Copy.

S21

All types implementing io.Reader must implement a Read method with the exact signature. A typical pattern to implement custom readers is to embed another reader inside the type, like this:

```
// UpperReader is an uppercase filter.
type UpperReader struct {
    r io.Reader
}
```

Why embed another reader? If we want to uppercase bytes, we have to read the bytes from somewhere. By embedding the type we make it easy to drop in a filter like this into a processing pipeline. In the example, we can seamlessly connect the UpperReader with the standard input:

```
...
_, err := io.Copy(os.Stdout, &UpperReader{r: os.Stdin})
...
```

Often, you'll see dedicated constructor for this, like bufio.NewReader or gzip.NewReader. For our tiny example, we can create the type in a more ad-hoc fashion - but the idea is the same.

```
// TODO: Implement UpperReader, a reader that converts all Unicode letter mapped to their u
func (r *UpperReader) Read(p []byte) (n int, err error) {
    n, err = r.r.Read(p)
    if err != nil {
        return
    }
    copy(p, bytes.ToUpper(p))
    return
}
```

The Read method implements the core logic. It first reads from the underlying reader. If everything went well, the byte slice `p` will be populated. Now we apply bytes.ToUpper to the read bytes and copy them back into the same slice. This works, because

```
bytes.ToUpper(p)
```

is evalated first, holding the result, which is

> a copy of the byte slice [p] with all Unicode letters mapped to their upper case.

This upper-cased version of the byte slice is then copied into the byte slice, that the Read method got as an argument, basically the space we are allowed to populate. Since the mapping from lowercase to uppercase does not change the number of bytes read, we can reuse `n` as the number of bytes read.

If this looks complicated, please be patient. While this pattern might look unfamiliar at first, it will become familiar the more you are exposed to it and the more you try to implement readers yourself.

S22

```go
// TODO: Implement Discard, that throws away everything that is written. 4 lines.
type Discard struct{}

func (r *Discard) Write(p []byte) (n int, err error) {
    return len(p), nil
}
```

S23

```go
type UpperWriter struct {
    w io.Writer
}

func (w *UpperWriter) Write(p []byte) (n int, err error) {
    return w.w.Write(bytes.ToUpper(p))
}
```

S24a

```go
// TODO: implement a reader that counts the total number of bytes read. 9 lines.
type CountingReader struct {
    r     io.Reader
    count uint64
}

func (r *CountingReader) Read(p []byte) (n int, err error) {
    n, err = r.r.Read(p)
    atomic.AddUint64(&r.count, uint64(n))
    return
}

func (r *CountingReader) Count() uint64 {
    return atomic.LoadUint64(&r.count)
}
```

S24b

All done.

S25

All done.

S26

All done.

S27a

All done.

S27b

```go
// TODO: Implement a reader that times out after a certain a given timeout. 19 lines.
type readResult struct {
    b   []byte
    err error
}

func (r *TimeoutReader) Read(p []byte) (n int, err error) {
    ch := make(chan readResult, 1)

    go func() {
        pp := make([]byte, len(p))
        _, err := r.r.Read(pp)
        ch <- readResult{pp, err}
    }()

    select {
    case <-time.After(r.timeout):
        return 0, ErrTimeout
    case res := <-ch:
        copy(p, res.b)
        return len(p), res.err
    }
}
```

S28

All done.

S29

All done.

S30

All done.

S40

All done.

S41

All done.

S42

All done.

S43

All done.

S44

All done.