When working with bytes we can do two things , reading and writing.

Reading bytes

Reader Interface

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

- The interface is implemented throughout the standard library by everything from network connections to files to wrappers for in-memory slices.
- Buffer p / byte slice p is passed as argument and not retutned. If Read() returned a byte slice instead of accepting one as an argument then the reader would have to allocate a new byte slice on every Read() call. That would wreak havoc on the garbage collector.
- your buffer isn't guaranteed to be filled. If you pass an 8-byte slice you could receive anywhere between 0 and 8 bytes back.
- It returns an io.EOF error as a normal part of usage when the stream is done. It may return the (non-nil) error from the same call or return the error (and n == 0) from a subsequent call. An instance of this general case is that a Reader returning a non-zero number of bytes at the end of the input stream may return either err == EOF or err == nil. The next Read should return 0, EOF.
- Callers should always process the n > 0 bytes returned before considering the error err. Doing so correctly handles I/O errors that happen after reading some bytes and also both of the allowed EOF behaviors.

Imporving Reading guarantees

- `Read(buf []byte) (int, error)`