Go Code Review Comments

This page collects common comments made during reviews of Go code, so that a single detailed explanation can be referred to by shorthands. This is a laundry list of common mistakes, not a comprehensive style guide.

You can view this as a supplement to Effective Go.

Additional comments related to testing can be found at Go Test Comments

Please discuss changes before editing this page, even *minor* ones. Many people have opinions and this is not the place for edit wars.

- Gofmt
- Comment Sentences
- Contexts
- Copying
- Crypto Rand
- Declaring Empty Slices
- Doc Comments
- Don't Panic
- Error Strings
- Examples
- Goroutine Lifetimes
- Handle Errors
- Imports
- Import Blank
- Import Dot
- In-Band Errors
- Indent Error Flow
- Initialisms
- Interfaces
- Line Length
- Mixed Caps
- Named Result Parameters
- Naked Returns
- Package Comments
- Package Names
- Pass Values
- Receiver Names
- Receiver Type
- Synchronous Functions
- Useful Test Failures
- Variable Names

Gofmt

Run gofmt on your code to automatically fix the majority of mechanical style issues. Almost all Go code in the wild uses `gofmt`. The rest of this document addresses non-mechanical style points.

An alternative is to use goimports, a superset of `gofmt` which additionally adds (and removes) import lines as necessary.

Comment Sentences

See https://go.dev/doc/effective_go#commentary. Comments documenting declarations should be full sentences, even if that seems a little redundant. This approach makes them format well when extracted into godoc documentation. Comments should begin with the name of the thing being described and end in a period:

```
// Request represents a request to run a command.
type Request struct { ...

// Encode writes the JSON encoding of req to w.
func Encode(w io.Writer, req *Request) { ...
```

and so on.

Contexts

Values of the context.Context type carry security credentials, tracing information, deadlines, and cancellation signals across API and process boundaries. Go programs pass Contexts explicitly along the entire function call chain from incoming RPCs and HTTP requests to outgoing requests.

Most functions that use a Context should accept it as their first parameter:

```
func F(ctx context.Context, /* other arguments */) {}
```

A function that is never request-specific may use context.Background(), but err on the side of passing a Context even if you think you don't need to. The default case is to pass a Context; only use context.Background() directly if you have a good reason why the alternative is a mistake.

Don't add a Context member to a struct type; instead add a ctx parameter to each method on that type that needs to pass it along. The one exception is for methods whose signature must match an interface in the standard library or in a third party library.

Don't create custom Context types or use interfaces other than Context in function signatures.

If you have application data to pass around, put it in a parameter, in the receiver, in globals, or, if it truly belongs there, in a Context value.

Contexts are immutable, so it's fine to pass the same ctx to multiple calls that share the same deadline, cancellation signal, credentials, parent trace, etc.

Copying

To avoid unexpected aliasing, be careful when copying a struct from another package. For example, the bytes.Buffer type contains a `[]byte` slice. If you copy a `Buffer`, the slice in the copy may alias the array in the original, causing subsequent method calls to have surprising effects.

In general, do not copy a value of type `T` if its methods are associated with the pointer type, `*T`.

Crypto Rand

Do not use package `math/rand` to generate keys, even throwaway ones. Unseeded, the generator is completely predictable. Seeded with `time.Nanoseconds()`, there are just a few bits of entropy. Instead, use `crypto/rand`'s Reader, and if you need text, print to hexadecimal or base64:

```
import (
    "crypto/rand"
    // "encoding/base64"
    // "encoding/hex"
    "fmt"
)

func Key() string {
    buf := make([]byte, 16)
    _, err := rand.Read(buf)
    if err != nil {
        panic(err)  // out of randomness, should never happen
    }
    return fmt.Sprintf("%x", buf)
    // or hex.EncodeToString(buf)
    // or base64.StdEncoding.EncodeToString(buf)
}
```

Declaring Empty Slices

When declaring an empty slice, prefer

```
var t []string
```

over

```
t := []string{}
```

The former declares a nil slice value, while the latter is non-nil but zero-length. They are functionally equivalent—their `len` and `cap` are both zero—but the nil slice is the preferred style.

Note that there are limited circumstances where a non-nil but zero-length slice is preferred, such as when encoding JSON objects (a `nil` slice encodes to `null`, while `[]string{}` encodes to the JSON array `[]`).

When designing interfaces, avoid making a distinction between a nil slice and a non-nil, zero-length slice, as this can lead to subtle programming errors.

For more discussion about nil in Go see Francesc Campoy's talk Understanding Nil.

Doc Comments

All top-level, exported names should have doc comments, as should non-trivial unexported type or function declarations. See https://go.dev/doc/effective_go#commentary for more information about commentary conventions.

Don't Panic

See https://go.dev/doc/effective_go#errors. Don't use panic for normal error handling. Use error and multiple return values.

Error Strings

Error strings should not be capitalized (unless beginning with proper nouns or acronyms) or end with punctuation, since they are usually printed following other context. That is, use `fmt.Errorf("something bad")` not `fmt.Errorf("Something bad")`, so that `log.Printf("Reading %s: %v", filename, err)` formats without a spurious capital letter mid-message. This does not apply to logging, which is implicitly line-oriented and not combined inside other messages.

Examples

When adding a new package, include examples of intended usage: a runnable Example, or a simple test demonstrating a complete call sequence.

Read more about testable Example() functions.

Goroutine Lifetimes

When you spawn goroutines, make it clear when - or whether - they exit.

Goroutines can leak by blocking on channel sends or receives: the garbage collector will not terminate a goroutine even if the channels it is blocked on are unreachable.

Even when goroutines do not leak, leaving them in-flight when they are no longer needed can cause other subtle and hard-to-diagnose problems. Sends on closed channels panic. Modifying still-in-use inputs "after the result isn't needed" can still lead to data races. And leaving goroutines in-flight for arbitrarily long can lead to unpredictable memory usage.

Try to keep concurrent code simple enough that goroutine lifetimes are obvious. If that just isn't feasible, document when and why the goroutines exit.

Handle Errors

See https://go.dev/doc/effective_go#errors. Do not discard errors using `_` variables. If a function returns an error, check it to make sure the function succeeded. Handle the error, return it, or, in truly exceptional situations, panic.

Imports

Avoid renaming imports except to avoid a name collision; good package names should not require renaming. In the event of collision, prefer to rename the most local or project-specific import.

Imports are organized in groups, with blank lines between them. The standard library packages are always in the first group.

```
package main

import (
    "fmt"
    "hash/adler32"
    "os"

    "appengine/foo"
    "appengine/user"

    "github.com/foo/bar"
    "rsc.io/goversion/version"
)
```

goimports will do this for you.

Import Blank

Packages that are imported only for their side effects (using the syntax `import _ "pkg"`) should only be imported in the main package of a program, or in tests that require them.

Import Dot

The import . form can be useful in tests that, due to circular dependencies, cannot be made part of the package being tested:

```
package foo_test

import (
    "bar/testutil" // also imports "foo"
    . "foo"
)
```

In this case, the test file cannot be in package foo because it uses bar/testutil, which imports foo. So we use the 'import .' form to let the file pretend to be part of package foo even though it is not. Except for this one case, do not use import . in your programs. It makes the programs much harder to read because it is unclear whether a name like Quux is a top-level identifier in the current package or in an imported package.

In-Band Errors

In C and similar languages, it's common for functions to return values like -1 or null to signal errors or missing results:

```
// Lookup returns the value for key or "" if there is no mapping for key.
func Lookup(key string) string

// Failing to check for an in-band error value can lead to bugs:
Parse(Lookup(key))  // returns "parse failure for value" instead of "no value for key"
```

Go's support for multiple return values provides a better solution. Instead of requiring clients to check for an in-band error value, a function should return an additional value to indicate whether its other return values are valid. This return value may be an error, or a boolean when no explanation is needed. It should be the final return value.

```
// Lookup returns the value for key or ok=false if there is no mapping for key.
func Lookup(key string) (value string, ok bool)
```

This prevents the caller from using the result incorrectly:

```
Parse(Lookup(key))  // compile-time error
```

And encourages more robust and readable code:

```
value, ok := Lookup(key)
if !ok {
    return fmt.Errorf("no value for %q", key)
}
return Parse(value)
```

This rule applies to exported functions but is also useful for unexported functions.

Return values like nil, "", 0, and -1 are fine when they are valid results for a function, that is, when the caller need not handle them differently from other values.

Some standard library functions, like those in package "strings", return in-band error values. This greatly simplifies string-manipulation code at the cost of requiring more diligence from the programmer. In general, Go code should return additional values for errors.

Indent Error Flow

Try to keep the normal code path at a minimal indentation, and indent the error handling, dealing with it first. This improves the readability of the code by permitting visually scanning the normal path quickly. For instance, don't write:

```
if err != nil {
    // error handling
} else {
    // normal code
}
```

Instead, write:

```
if err != nil {
    // error handling
    return // or continue, etc.
}
// normal code
```

If the `if` statement has an initialization statement, such as:

```
if x, err := f(); err != nil {
    // error handling
    return
} else {
    // use x
}
```

then this may require moving the short variable declaration to its own line:

```
x, err := f()
if err != nil {
    // error handling
    return
}
// use x
```

Initialisms

Words in names that are initialisms or acronyms (e.g. “URL” or “NATO”) have a consistent case. For example, “URL” should appear as “URL” or “url” (as in “urlPony”, or “URLPony”), never as “Url”. As an example: ServeHTTP not ServeHttp. For identifiers with multiple initialized “words”, use for example “xmlHTTPRequest” or “XMLHTTPRequest”.

This rule also applies to “ID” when it is short for “identifier” (which is pretty much all cases when it’s not the “id” as in “ego”, “superego”), so write “appID” instead of “appId”.

Code generated by the protocol buffer compiler is exempt from this rule. Human-written code is held to a higher standard than machine-written code.

Interfaces

Go interfaces generally belong in the package that uses values of the interface type, not the package that implements those values. The implementing package should return concrete (usually pointer or struct) types: that way, new methods can be added to implementations without requiring extensive refactoring.

Do not define interfaces on the implementor side of an API “for mocking”; instead, design the API so that it can be tested using the public API of the real implementation.

Do not define interfaces before they are used: without a realistic example of usage, it is too difficult to see whether an interface is even necessary, let alone what methods it ought to contain.

```
package consumer  // consumer.go

type Thinger interface { Thing() bool }

func Foo(t Thinger) string { ... }
```

```
package consumer // consumer_test.go

type fakeThinger struct{ ... }
func (t fakeThinger) Thing() bool { ... }
...
if Foo(fakeThinger{...}) == "x" { ... }
```

```
// DO NOT DO IT!!!
package producer

type Thinger interface { Thing() bool }

type defaultThinger struct{ ... }
func (t defaultThinger) Thing() bool { ... }
```

```
func NewThinger() Thinger { return defaultThinger{ ... } }
```

Instead return a concrete type and let the consumer mock the producer implementation.

```
package producer

type Thinger struct{ ... }
func (t Thinger) Thing() bool { ... }

func NewThinger() Thinger { return Thinger{ ... } }
```

Line Length

There is no rigid line length limit in Go code, but avoid uncomfortably long lines. Similarly, don't add line breaks to keep lines short when they are more readable long–for example, if they are repetitive.

Most of the time when people wrap lines "unnaturally" (in the middle of function calls or function declarations, more or less, say, though some exceptions are around), the wrapping would be unnecessary if they had a reasonable number of parameters and reasonably short variable names. Long lines seem to go with long names, and getting rid of the long names helps a lot.

In other words, break lines because of the semantics of what you're writing (as a general rule) and not because of the length of the line. If you find that this produces lines that are too long, then change the names or the semantics and you'll probably get a good result.

This is, actually, exactly the same advice about how long a function should be. There's no rule "never have a function more than N lines long", but there is definitely such a thing as too long of a function, and of too repetitive tiny functions, and the solution is to change where the function boundaries are, not to start counting lines.

Mixed Caps

See https://go.dev/doc/effective_go#mixed-caps. This applies even when it breaks conventions in other languages. For example an unexported constant is `maxLength` not `MaxLength` or `MAX_LENGTH`.

Also see Initialisms.

Named Result Parameters

Consider what it will look like in godoc. Named result parameters like:

```
func (n *Node) Parent1() (node *Node) {}
func (n *Node) Parent2() (node *Node, err error) {}
```

will be repetitive in godoc; better to use:

```
func (n *Node) Parent1() *Node {}
func (n *Node) Parent2() (*Node, error) {}
```

On the other hand, if a function returns two or three parameters of the same type, or if the meaning of a result isn't clear from context, adding names may be useful in some contexts. Don't name result parameters just to avoid declaring a var inside the function; that trades off a minor implementation brevity at the cost of unnecessary API verbosity.

```
func (f *Foo) Location() (float64, float64, error)
```

is less clear than:

```
// Location returns f's latitude and longitude.
// Negative values mean south and west, respectively.
func (f *Foo) Location() (lat, long float64, err error)
```

Naked returns are okay if the function is a handful of lines. Once it's a medium sized function, be explicit with your return values. Corollary: it's not worth it to name result parameters just because it enables you to use naked returns. Clarity of docs is always more important than saving a line or two in your function.

Finally, in some cases you need to name a result parameter in order to change it in a deferred closure. That is always OK.

Naked Returns

A `return` statement without arguments returns the named return values. This is known as a "naked" return.

```
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

See Named Result Parameters.

Package Comments

Package comments, like all comments to be presented by godoc, must appear adjacent to the package clause, with no blank line.

```
// Package math provides basic constants and mathematical functions.
package math
```

```
/*
Package template implements data-driven templates for generating textual
output such as HTML.
```

```
....
*/
package template
```

For "package main" comments, other styles of comment are fine after the binary name (and it may be capitalized if it comes first), For example, for a `package main` in the directory `seedgen` you could write:

```
// Binary seedgen ...
package main
```

or

```
// Command seedgen ...
package main
```

or

```
// Program seedgen ...
package main
```

or

```
// The seedgen command ...
package main
```

or

```
// The seedgen program ...
package main
```

or

```
// Seedgen ..
package main
```

These are examples, and sensible variants of these are acceptable.

Note that starting the sentence with a lower-case word is not among the acceptable options for package comments, as these are publicly-visible and should be written in proper English, including capitalizing the first word of the sentence. When the binary name is the first word, capitalizing it is required even though it does not strictly match the spelling of the command-line invocation.

See https://go.dev/doc/effective_go#commentary for more information about commentary conventions.

Package Names

All references to names in your package will be done using the package name, so you can omit that name from the identifiers. For example, if you are in package chubby, you don't need type ChubbyFile, which clients will write as `chubby.ChubbyFile`. Instead, name the type `File`, which clients will write as

`chubby.File`. Avoid meaningless package names like util, common, misc, api, types, and interfaces. See https://go.dev/doc/effective_go#package-names and https://go.dev/blog/package-names for more.

Pass Values

Don't pass pointers as function arguments just to save a few bytes. If a function refers to its argument `x` only as `*x` throughout, then the argument shouldn't be a pointer. Common instances of this include passing a pointer to a string (`*string`) or a pointer to an interface value (`*io.Reader`). In both cases the value itself is a fixed size and can be passed directly. This advice does not apply to large structs, or even small structs that might grow.

Receiver Names

The name of a method's receiver should be a reflection of its identity; often a one or two letter abbreviation of its type suffices (such as "c" or "cl" for "Client"). Don't use generic names such as "me", "this" or "self", identifiers typical of object-oriented languages that gives the method a special meaning. In Go, the receiver of a method is just another parameter and therefore, should be named accordingly. The name need not be as descriptive as that of a method argument, as its role is obvious and serves no documentary purpose. It can be very short as it will appear on almost every line of every method of the type; familiarity admits brevity. Be consistent, too: if you call the receiver "c" in one method, don't call it "cl" in another.

Receiver Type

Choosing whether to use a value or pointer receiver on methods can be difficult, especially to new Go programmers. If in doubt, use a pointer, but there are times when a value receiver makes sense, usually for reasons of efficiency, such as for small unchanging structs or values of basic type. Some useful guidelines:

- If the receiver is a map, func or chan, don't use a pointer to them. If the receiver is a slice and the method doesn't reslice or reallocate the slice, don't use a pointer to it.
- If the method needs to mutate the receiver, the receiver must be a pointer.
- If the receiver is a struct that contains a sync.Mutex or similar synchronizing field, the receiver must be a pointer to avoid copying.
- If the receiver is a large struct or array, a pointer receiver is more efficient. How large is large? Assume it's equivalent to passing all its elements as arguments to the method. If that feels too large, it's also too large for the receiver.
- Can function or methods, either concurrently or when called from this method, be mutating the receiver? A value type creates a copy of the receiver when the method is invoked, so outside updates will not be applied

to this receiver. If changes must be visible in the original receiver, the receiver must be a pointer.
- If the receiver is a struct, array or slice and any of its elements is a pointer to something that might be mutating, prefer a pointer receiver, as it will make the intention clearer to the reader.
- If the receiver is a small array or struct that is naturally a value type (for instance, something like the time.Time type), with no mutable fields and no pointers, or is just a simple basic type such as int or string, a value receiver makes sense. A value receiver can reduce the amount of garbage that can be generated; if a value is passed to a value method, an on-stack copy can be used instead of allocating on the heap. (The compiler tries to be smart about avoiding this allocation, but it can't always succeed.) Don't choose a value receiver type for this reason without profiling first.
- Don't mix receiver types. Choose either pointers or struct types for all available methods.
- Finally, when in doubt, use a pointer receiver.

Synchronous Functions

Prefer synchronous functions - functions which return their results directly or finish any callbacks or channel ops before returning - over asynchronous ones.

Synchronous functions keep goroutines localized within a call, making it easier to reason about their lifetimes and avoid leaks and data races. They're also easier to test: the caller can pass an input and check the output without the need for polling or synchronization.

If callers need more concurrency, they can add it easily by calling the function from a separate goroutine. But it is quite difficult - sometimes impossible - to remove unnecessary concurrency at the caller side.

Useful Test Failures

Tests should fail with helpful messages saying what was wrong, with what inputs, what was actually got, and what was expected. It may be tempting to write a bunch of assertFoo helpers, but be sure your helpers produce useful error messages. Assume that the person debugging your failing test is not you, and is not your team. A typical Go test fails like:

```
if got != tt.want {
    t.Errorf("Foo(%q) = %d; want %d", tt.in, got, tt.want) // or Fatalf, if test can't test
}
```

Note that the order here is actual != expected, and the message uses that order too. Some test frameworks encourage writing these backwards: 0 != x, "expected 0, got x", and so on. Go does not.

If that seems like a lot of typing, you may want to write a [[table-driven

test|TableDrivenTests]].

Another common technique to disambiguate failing tests when using a test helper with different input is to wrap each caller with a different TestFoo function, so the test fails with that name:

```
func TestSingleValue(t *testing.T) { testHelper(t, []int{80}) }
func TestNoValues(t *testing.T)    { testHelper(t, []int{}) }
```

In any case, the onus is on you to fail with a helpful message to whoever's debugging your code in the future.

Variable Names

Variable names in Go should be short rather than long. This is especially true for local variables with limited scope. Prefer `c` to `lineCount`. Prefer `i` to `sliceIndex`.

The basic rule: the further from its declaration that a name is used, the more descriptive the name must be. For a method receiver, one or two letters is sufficient. Common variables such as loop indices and readers can be a single letter (`i`, `r`). More unusual things and global variables need more descriptive names.