styleguide

Go Style Decisions

https://google.github.io/styleguide/go/decisions

Overview | Guide | Decisions | Best practices

Note: This is part of a series of documents that outline Go Style at Google. This document is normative **but not** canonical, and is subordinate to the core style guide. See the overview for more information.

About

This document contains style decisions intended to unify and provide standard guidance, explanations, and examples for the advice given by the Go readability mentors.

This document is **not exhaustive** and will grow over time. In cases where the core style guide contradicts the advice given here, **the style guide takes precedence**, and this document should be updated accordingly.

See the Overview for the full set of Go Style documents.

The following sections have moved from style decisions to another part of the guide:

MixedCaps: see guide#mixed-caps

Formatting: see guide#formatting

Line Length: see guide#line-length

Naming

See the naming section within the core style guide for overarching guidance on naming. The following sections provide further clarification on specific areas within naming.

Underscores

Names in Go should in general not contain underscores. There are three exceptions to this principle:

Package names that are only imported by generated code may contain underscores. See package names for more detail around how to choose multi-word package names.

Test, Benchmark and Example function names within `*_test.go` files may include underscores.

Low-level libraries that interoperate with the operating system or cgo may reuse identifiers, as is done in `syscall`. This is expected to be very rare in most codebases.

Package names

Go package names should be short and contain only lowercase letters. A package name composed of multiple words should be left unbroken in all lowercase. For example, the package `tabwriter` is not named `tabWriter`, `TabWriter`, or `tab_writer`.

Avoid selecting package names that are likely to be shadowed by commonly used local variable names. For example, `usercount` is a better package name than `count`, since `count` is a commonly used variable name.

Go package names should not have underscores. If you need to import a package that does have one in its name (usually from generated or third party code), it must be renamed at import time to a name that is suitable for use in Go code.

An exception to this is that package names that are only imported by generated code may contain underscores. Specific examples include:

Using the `_test` suffix for an external test package, for example an integration test

Using the `_test` suffix for package-level documentation examples

Avoid uninformative package names like `util`, `utility`, `common`, `helper`, and so on. See more about so-called "utility packages".

When an imported package is renamed (e.g. `import foopb "path/to/foo_go_proto"`), the local name for the package must comply with the rules above, as the local name dictates how the symbols in the package are referenced in the file. If a given import is renamed in multiple files, particularly in the same or nearby packages, the same local name should be used wherever possible for consistency.

See also: Go blog post about package names.

Receiver names

Receiver variable names must be:

Short (usually one or two letters in length)

Abbreviations for the type itself

Applied consistently to every receiver for that type

Long Name	Better Name
`func (tray Tray)`	`func (t Tray)`
`func (info *ResearchInfo)`	`func (ri *ResearchInfo)`

Long Name	Better Name
`func (this *ReportWriter)`	`func (w *ReportWriter)`
`func (self *Scanner)`	`func (s *Scanner)`

Constant names

Constant names must use MixedCaps like all other names in Go. (Exported constants start with uppercase, while unexported constants start with lowercase.) This applies even when it breaks conventions in other languages. Constant names should not be a derivative of their values and should instead explain what the value denotes.

```
// Good:
const MaxPacketSize = 512

const (
    ExecuteBit = 1 << iota
    WriteBit
    ReadBit
)
```

Do not use non-MixedCaps constant names or constants with a `K` prefix.

```
// Bad:
const MAX_PACKET_SIZE = 512
const kMaxBufferSize = 1024
const KMaxUsersPergroup = 500
```

Name constants based on their role, not their values. If a constant does not have a role apart from its value, then it is unnecessary to define it as a constant.

```
// Bad:
const Twelve = 12

const (
    UserNameColumn = "username"
    GroupColumn    = "group"
)
```

Initialisms

Words in names that are initialisms or acronyms (e.g., `URL` and `NATO`) should have the same case. `URL` should appear as `URL` or `url` (as in `urlPony`, or `URLPony`), never as `Url`. This also applies to `ID` when it is short for "identifier"; write `appID` instead of `appId`.

In names with multiple initialisms (e.g. `XMLAPI` because it contains `XML` and `API`), each letter within a given initialism should have the same case, but each initialism in the name does not need to have the same case.

In names with an initialism containing a lowercase letter (e.g. `DDoS`, `iOS`, `gRPC`), the initialism should appear as it would in standard prose, unless you need to change the first letter for the sake of exportedness. In these cases, the entire initialism should be the same case (e.g. `ddos`, `IOS`, `GRPC`).

Initialism(s)	Scope	Correct	Incorrect
XML API	Exported	`XMLAPI`	`XmlApi`, `XMLApi`, `XmlAPI`, `XMLapi`
XML API	Unexported	`xmlAPI`	`xmlapi`, `xmlApi`
iOS	Exported	`IOS`	`Ios`, `IoS`
iOS	Unexported	`iOS`	`ios`
gRPC	Exported	`GRPC`	`Grpc`
gRPC	Unexported	`gRPC`	`grpc`
DDoS	Exported	`DDoS`	`DDOS`, `Ddos`
DDoS	Unexported	`ddos`	`dDoS`, `dDOS`

Getters

Function and method names should not use a `Get` or `get` prefix, unless the underlying concept uses the word "get" (e.g. an HTTP GET). Prefer starting the name with the noun directly, for example use `Counts` over `GetCounts`.

If the function involves performing a complex computation or executing a remote call, a different word like `Compute` or `Fetch` can be used in place of `Get`, to make it clear to a reader that the function call may take time and could block or fail.

Variable names

The general rule of thumb is that the length of a name should be proportional to the size of its scope and inversely proportional to the number of times that it is used within that scope. A variable created at file scope may require multiple words, whereas a variable scoped to a single inner block may be a single word or even just a character or two, to keep the code clear and avoid extraneous information.

Here is a rough baseline. These numeric guidelines are not strict rules. Apply judgement based on context, clarity, and concision.

A small scope is one in which one or two small operations are performed, say 1-7 lines.

A medium scope is a few small or one large operation, say 8-15 lines.

A large scope is one or a few large operations, say 15-25 lines.

A very large scope is anything that spans more than a page (say, more than 25 lines).

A name that might be perfectly clear (e.g., `c` for a counter) within a small scope could be insufficient in a larger scope and would require clarification to remind the reader of its purpose further along in the code. A scope in which there are many variables, or variables that represent similar values or concepts, may necessitate longer variable names than the scope suggests.

The specificity of the concept can also help to keep a variable's name concise. For example, assuming there is only a single database in use, a short variable name like `db` that might normally be reserved for very small scopes may remain perfectly clear even if the scope is very large. In this case, a single word `database` is likely acceptable based on the size of the scope, but is not required as `db` is a very common shortening for the word with few alternate interpretations.

The name of a local variable should reflect what it contains and how it is being used in the current context, rather than where the value originated. For example, it is often the case that the best local variable name is not the same as the struct or protocol buffer field name.

In general:

Single-word names like `count` or `options` are a good starting point.

Additional words can be added to disambiguate similar names, for example `userCount` and `projectCount`.

Do not simply drop letters to save typing. For example `Sandbox` is preferred over `Sbx`, particularly for exported names.

Omit [types and type-like words] from most variable names.
For a number, `userCount` is a better name than `numUsers` or `usersInt`.

For a slice, `users` is a better name than `userSlice`.

It is acceptable to include a type-like qualifier if there are two versions of a value in scope, for example you might have an input stored in `ageString` and use `age` for the parsed value.

Omit words that are clear from the [surrounding context]. For example, in the implementation of a `UserCount` method, a local variable called `userCount` is probably redundant; `count`, `users`, or even `c` are just as readable.

Single-letter variable names

Single-letter variable names can be a useful tool to minimize repetition, but can also make code needlessly opaque. Limit their use to instances where the full word is obvious and where it would be repetitive for it to appear in place of the single-letter variable.

In general:

For a method receiver variable, a one-letter or two-letter name is preferred.

Using familiar variable names for common types is often helpful:
`r` for an `io.Reader` or `*http.Request`

`w` for an `io.Writer` or `http.ResponseWriter`

Single-letter identifiers are acceptable as integer loop variables, particularly for indices (e.g., `i`) and coordinates (e.g., `x` and `y`).

Abbreviations can be acceptable loop identifiers when the scope is short, for example `for _, n := range nodes { ... }`.

Repetition

A piece of Go source code should avoid unnecessary repetition. One common source of this is repetitive names, which often include unnecessary words or repeat their context or type. Code itself can also be unnecessarily repetitive if the same or a similar code segment appears multiple times in close proximity.

Repetitive naming can come in many forms, including:

Package vs. exported symbol name

When naming exported symbols, the name of the package is always visible outside your package, so redundant information between the two should be reduced or eliminated. If a package exports only one type and it is named after the package itself, the canonical name for the constructor is `New` if one is required.

> **Examples:** Repetitive Name -> Better Name
>
> `widget.NewWidget` -> `widget.New`
>
> `widget.NewWidgetWithName` -> `widget.NewWithName`
>
> `db.LoadFromDatabase` -> `db.Load`
>
> `goatteleportutil.CountGoatsTeleported` -> `gtutil.CountGoatsTeleported` or `goatteleport.Count`

> `myteampb.MyTeamMethodRequest` -> `mtpb.MyTeamMethodRequest` or `myteampb.MethodRequest`

Variable name vs. type

The compiler always knows the type of a variable, and in most cases it is also clear to the reader what type a variable is by how it is used. It is only necessary to clarify the type of a variable if its value appears twice in the same scope.

Repetitive Name	Better Name
`var numUsers int`	`var users int`
`var nameString string`	`var name string`
`var primaryProject *Project`	`var primary *Project`

If the value appears in multiple forms, this can be clarified either with an extra word like `raw` and `parsed` or with the underlying representation:

```
// Good:
limitStr := r.FormValue("limit")
limit, err := strconv.Atoi(limitStr)
```

```
// Good:
limitRaw := r.FormValue("limit")
limit, err := strconv.Atoi(limitRaw)
```

External context vs. local names

Names that include information from their surrounding context often create extra noise without benefit. The package name, method name, type name, function name, import path, and even filename can all provide context that automatically qualifies all names within.

```
// Bad:
// In package "ads/targeting/revenue/reporting"
type AdsTargetingRevenueReport struct{}

func (p *Project) ProjectName() string
```

```
// Good:
// In package "ads/targeting/revenue/reporting"
type Report struct{}
```

```
func (p *Project) Name() string
```

```
// Bad:
// In package "sqldb"
type DBConnection struct{}
```

```
// Good:
// In package "sqldb"
type Connection struct{}
```

```
// Bad:
// In package "ads/targeting"
func Process(in *pb.FooProto) *Report {
    adsTargetingID := in.GetAdsTargetingID()
}
```

```
// Good:
// In package "ads/targeting"
func Process(in *pb.FooProto) *Report {
    id := in.GetAdsTargetingID()
}
```

Repetition should generally be evaluated in the context of the user of the symbol, rather than in isolation. For example, the following code has lots of names that may be fine in some circumstances, but redundant in context:

```
// Bad:
func (db *DB) UserCount() (userCount int, err error) {
    var userCountInt64 int64
    if dbLoadError := db.LoadFromDatabase("count(distinct users)",
&userCountInt64); dbLoadError != nil {
        return 0, fmt.Errorf("failed to load user count: %s", dbLoadError)
    }
    userCount = int(userCountInt64)
    return userCount, nil
}
```

Instead, information about names that are clear from context or usage can often be omitted:

```
// Good:
func (db *DB) UserCount() (int, error) {
    var count int64
```

```
    if err := db.Load("count(distinct users)", &count); err != nil {
        return 0, fmt.Errorf("failed to load user count: %s", err)
    }
    return int(count), nil
}
```

The conventions around commentary (which include what to comment, what style to use, how to provide runnable examples, etc.) are intended to support the experience of reading the documentation of a public API. See Effective Go for more information.

The best practices document's section on documentation conventions discusses this further.

Best Practice: Use doc preview during development and code review to see whether the documentation and runnable examples are useful and are presented the way you expect them to be.

Tip: Godoc uses very little special formatting; lists and code snippets should usually be indented to avoid linewrapping. Apart from indentation, decoration should generally be avoided.

Ensure that commentary is readable from source even on narrow screens.

When a comment gets too long, it is recommended to wrap it into multiple single-line comments. When possible, aim for comments that will read well on an 80-column wide terminal, however this is not a hard cut-off; there is no fixed line length limit for comments in Go. The standard library, for example, often chooses to break a comment based on punctuation, which sometimes leaves the individual lines closer to the 60-70 character mark.

There is plenty of existing code in which comments exceed 80 characters in length. This guidance should not be used as a justification to change such code as part of a readability review (see consistency), though teams are encouraged to opportunistically update comments to follow this guideline as a part of other refactors. The primary goal of this guideline is to ensure that all Go readability mentors make the same recommendation when and if recommendations are made.

See this post from The Go Blog on documentation for more on commentary.

```
# Good:
// This is a comment paragraph.
// The length of individual lines doesn't matter in Godoc;
// but the choice of wrapping makes it easy to read on narrow screens.
//
// Don't worry too much about the long URL:
// https://supercalifragilisticexpialidocious.example.com:8080/Animalia
/Chordata/Mammalia/Rodentia/Geomyoidea/Geomyidae/
```

```
//
// Similarly, if you have other information that is made awkward
// by too many line breaks, use your judgment and include a long line
// if it helps rather than hinders.
```

Avoid comments that will wrap repeatedly on small screens, which is a poor reader experience.

```
# Bad:
// This is a comment paragraph. The length of individual lines doesn't
matter in
Godoc;
// but the choice of wrapping causes jagged lines on narrow screens or in
code
review,
// which can be annoying, especially when in a comment block that will
wrap
repeatedly.
//
// Don't worry too much about the long URL:
// https://supercalifragilisticexpialidocious.example.com:8080/Animalia
/Chordata/Mammalia/Rodentia/Geomyoidea/Geomyidae/
```

All top-level exported names must have doc comments, as should unexported type or function declarations with unobvious behavior or meaning. These comments should be [full sentences](#) that begin with the name of the object being described. An article (“a”, “an”, “the”) can precede the name to make it read more naturally.

```
// Good:
// A 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) { ...
```

Doc comments appear in [Godoc](#) and are surfaced by IDEs, and therefore should be written for anyone using the package.

A documentation comment applies to the following symbol, or the group of fields if it appears in a struct.

```
// Good:
// Options configure the group management service.
type Options struct {
```

```
    // General setup:
    Name  string
    Group *FooGroup

    // Dependencies:
    DB *sql.DB

    // Customization:
    LargeGroupThreshold int // optional; default: 10
    MinimumMembers      int // optional; default: 2
}
```

Best Practice: If you have doc comments for unexported code, follow the same custom as if it were exported (namely, starting the comment with the unexported name). This makes it easy to export it later by simply replacing the unexported name with the newly-exported one across both comments and code.

Comments that are complete sentences should be capitalized and punctuated like standard English sentences. (As an exception, it is okay to begin a sentence with an uncapitalized identifier name if it is otherwise clear. Such cases are probably best done only at the beginning of a paragraph.)

Comments that are sentence fragments have no such requirements for punctuation or capitalization.

Documentation comments should always be complete sentences, and as such should always be capitalized and punctuated. Simple end-of-line comments (especially for struct fields) can be simple phrases that assume the field name is the subject.

```
// Good:
// A Server handles serving quotes from the collected works of
Shakespeare.
type Server struct {
    // BaseDir points to the base directory under which Shakespeare's
works are stored.
    //
    // The directory structure is expected to be the following:
    //   {BaseDir}/manifest.json
    //   {BaseDir}/{name}/{name}-part{number}.txt
    BaseDir string

    WelcomeMessage  string // displayed when user logs in
    ProtocolVersion string // checked against incoming requests
```

```
    PageLength       int     // lines per page when printing (optional;
default: 20)
}
```

Examples

Packages should clearly document their intended usage. Try to provide a runnable example; examples show up in Godoc. Runnable examples belong in the test file, not the production source file. See this example (Godoc, source).

If it isn't feasible to provide a runnable example, example code can be provided within code comments. As with other code and command-line snippets in comments, it should follow standard formatting conventions.

Named result parameters

When naming parameters, consider how function signatures appear in Godoc. The name of the function itself and the type of the result parameters are often sufficiently clear.

```
// Good:
func (n *Node) Parent1() *Node
func (n *Node) Parent2() (*Node, error)
```

If a function returns two or more parameters of the same type, adding names can be useful.

```
// Good:
func (n *Node) Children() (left, right *Node, err error)
```

If the caller must take action on particular result parameters, naming them can help suggest what the action is:

```
// Good:
// WithTimeout returns a context that will be canceled no later than d
duration
// from now.
//
// The caller must arrange for the returned cancel function to be called
when
// the context is no longer needed to prevent a resource leak.
func WithTimeout(parent Context, d time.Duration) (ctx Context, cancel
func())
```

In the code above, cancellation is a particular action a caller must take. However, were the result parameters written as `(Context, func())` alone, it would be unclear what is meant by "cancel function".

Don't use named result parameters when the names produce unnecessary repetition.

```
// Bad:
func (n *Node) Parent1() (node *Node)
func (n *Node) Parent2() (node *Node, err error)
```

Don't name result parameters in order to avoid declaring a variable inside the function. This practice results in unnecessary API verbosity at the cost of minor implementation brevity.

Naked returns are acceptable only in a small function. Once it's a medium-sized function, be explicit with your returned values. Similarly, do not name result parameters just because it enables you to use naked returns. Clarity is always more important than saving a few lines in your function.

It is always acceptable to name a result parameter if its value must be changed in a deferred closure.

> **Tip:** Types can often be clearer than names in function signatures. GoTip #38: Functions as Named Types demonstrates this.
>
> In, `WithTimeout` above, the real code uses a `CancelFunc` instead of a raw `func()` in the result parameter list and requires little effort to document.

Package comments must appear immediately above the package clause with no blank line between the comment and the package name. Example:

```
// Good:
// Package math provides basic constants and mathematical functions.
//
// This package does not guarantee bit-identical results across
architectures.
package math
```

There must be a single package comment per package. If a package is composed of multiple files, exactly one of the files should have a package comment.

Comments for `main` packages have a slightly different form, where the name of the `go_binary` rule in the BUILD file takes the place of the package name.

```
// Good:
// The seed_generator command is a utility that generates a Finch seed
file
// from a set of JSON study configs.
package main
```

Other styles of comment are fine as long as the name of the binary is exactly as written in the BUILD file.

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.

```
// Good:
// Binary seed_generator ...
// Command seed_generator ...
// Program seed_generator ...
// The seed_generator command ...
// The seed_generator program ...
// Seed_generator ...
```

Tips:

Example command-line invocations and API usage can be useful documentation. For Godoc formatting, indent the comment lines containing code.

If there is no obvious primary file or if the package comment is extraordinarily long, it is acceptable to put the doc comment in a file named `doc.go` with only the comment and the package clause.

Multiline comments can be used instead of multiple single-line comments. This is primarily useful if the documentation contains sections which may be useful to copy and paste from the source file, as with sample command-lines (for binaries) and template examples.

```
// Good:
/*
The seed_generator command is a utility that generates a Finch seed file
from a set of JSON study configs.

    seed_generator *.json | base64 > finch-seed.base64
*/
package template
```

Comments intended for maintainers and that apply to the whole file are typically placed after import declarations. These are not surfaced in Godoc and are not subject to the rules above on package comments.

Imports

Import renaming

Imports should only be renamed to avoid a name collision with other imports. (A corollary of this is that [good package names] should not require renaming.) In the event of a name collision, prefer to rename the most local or project-specific import. Local names (aliases) for packages must follow [the guidance around package naming, including the prohibition on the use of underscores and capital letters.

Generated protocol buffer packages must be renamed to remove underscores from their names, and their aliases must have a `pb` suffix. See proto and stub best practices for more information.

```
// Good:
import (
    fspb "path/to/package/foo_service_go_proto"
)
```

Imports that have package names with no useful identifying information (e.g. `package v1`) should be renamed to include the previous path component. The rename must be consistent with other local files importing the same package and may include the version number.

Note: It is preferred to rename the package to conform with good package names, but that is often not feasible for packages in vendored directories.

```
// Good:
import (
    core "github.com/kubernetes/api/core/v1"
    meta "github.com/kubernetes/apimachinery/pkg/apis/meta/v1beta1"
)
```

If you need to import a package whose name collides with a common local variable name that you want to use (e.g. `url`, `ssh`) and you wish to rename the package, the preferred way to do so is with the `pkg` suffix (e.g. `urlpkg`). Note that it is possible to shadow a package with a local variable; this rename is only necessary if the package still needs to be used when such a variable is in scope.

Import grouping

Imports should be organized in two groups:

Standard library packages

Other (project and vendored) packages

```
// Good:
package main

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

```
    "github.com/dsnet/compress/flate"
    "golang.org/x/text/encoding"
    "google.golang.org/protobuf/proto"
    foopb "myproj/foo/proto/proto"
    _ "myproj/rpc/protocols/dial"
    _ "myproj/security/auth/authhooks"
)
```

It is acceptable to split the project packages into multiple groups, for example if you want a separate group for renamed, imported-only-for-side-effects or another special group of imports.

```
// Good:
package main

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


    "github.com/dsnet/compress/flate"
    "golang.org/x/text/encoding"
    "google.golang.org/protobuf/proto"

    foopb "myproj/foo/proto/proto"

    _ "myproj/rpc/protocols/dial"
    _ "myproj/security/auth/authhooks"
)
```

Note: Maintaining optional groups - splitting beyond what is necessary for the mandatory separation between standard library and Google imports - is not supported by the goimports tool. Additional import subgroups require attention on the part of both authors and reviewers to maintain in a conforming state.

Google programs that are also AppEngine apps should have a separate group for AppEngine imports.

Gofmt takes care of sorting each group by import path. However, it does not automatically separate imports into groups. The popular goimports tool combines Gofmt and import management, separating imports into groups based on the decision above. It is permissible to let goimports manage import arrangement entirely, but as a file is revised its import list must remain internally consistent.

Import “blank” (`import _`)

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

Some examples of such packages include:

time/tzdata

image/jpeg in image processing code

Avoid blank imports in library packages, even if the library indirectly depends on them. Constraining side-effect imports to the main package helps control dependencies, and makes it possible to write tests that rely on a different import without conflict or wasted build costs.

The following are the only exceptions to this rule:

You may use a blank import to bypass the check for disallowed imports in the nogo static checker.

You may use a blank import of the embed package in a source file which uses the `//go:embed` compiler directive.

Tip: If you create a library package that indirectly depends on a side-effect import in production, document the intended usage.

Import “dot” (`import .`)

The `import .` form is a language feature that allows bringing identifiers exported from another package to the current package without qualification. See the language spec for more.

Do **not** use this feature in the Google codebase; it makes it harder to tell where the functionality is coming from.

```
// Bad:
package foo_test

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

var myThing = Bar() // Bar defined in package foo; no qualification
needed.
```

```go
// Good:
package foo_test

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

var myThing = foo.Bar()
```

Errors

Returning errors

Use `error` to signal that a function can fail. By convention, `error` is the last result parameter.

```go
// Good:
func Good() error { /* ... */ }
```

Returning a `nil` error is the idiomatic way to signal a successful operation that could otherwise fail. If a function returns an error, callers must treat all non-error return values as unspecified unless explicitly documented otherwise. Commonly, the non-error return values are their zero values, but this cannot be assumed.

```go
// Good:
func GoodLookup() (*Result, error) {
    // ...
    if err != nil {
        return nil, err
    }
    return res, nil
}
```

Exported functions that return errors should return them using the `error` type. Concrete error types are susceptible to subtle bugs: a concrete `nil` pointer can get wrapped into an interface and thus become a non-nil value (see the [Go FAQ entry on the topic]).

```go
// Bad:
func Bad() *os.PathError { /*...*/ }
```

Tip: A function that takes a `context.Context` argument should usually return an `error` so that the caller can determine if the context was cancelled while the function was running.

Error strings

Error strings should not be capitalized (unless beginning with an exported name, a proper noun or an acronym) and should not end with punctuation. This is because error strings usually appear within other context before being printed to the user.

```
// Bad:
err := fmt.Errorf("Something bad happened.")
```

```
// Good:
err := fmt.Errorf("something bad happened")
```

On the other hand, the style for the full displayed message (logging, test failure, API response, or other UI) depends, but should typically be capitalized.

```
// Good:
log.Infof("Operation aborted: %v", err)
log.Errorf("Operation aborted: %v", err)
t.Errorf("Op(%q) failed unexpectedly; err=%v", args, err)
```

Handle errors

Code that encounters an error should make a deliberate choice about how to handle it. It is not usually appropriate to discard errors using _ variables. If a function returns an error, do one of the following:

Handle and address the error immediately.

Return the error to the caller.

In exceptional situations, call `log.Fatal` or (if absolutely necessary) `panic`.

Note: `log.Fatalf` is not the standard library log. See [#logging].

In the rare circumstance where it is appropriate to ignore or discard an error (for example a call to `(*bytes.Buffer).Write` that is documented to never fail), an accompanying comment should explain why this is safe.

```
// Good:
var b *bytes.Buffer

n, _ := b.Write(p) // never returns a non-nil error
```

For more discussion and examples of error handling, see Effective Go and best practices.

In-band errors

In C and similar languages, it is common for functions to return values like -1, null, or the empty string to signal errors or missing results. This is known as in-band error handling.

```
// Bad:
// Lookup returns the value for key or -1 if there is no mapping for key.
func Lookup(key string) int
```

Failing to check for an in-band error value can lead to bugs and can attribute errors to the wrong function.

```
// Bad:
// The following line returns an error that Parse failed for the input
value,
// whereas the failure was that there is no mapping for missingKey.
return Parse(Lookup(missingKey))
```

Go's support for multiple return values provides a better solution (see the [Effective Go section on multiple returns]). 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, and should be the final return value.

```
// Good:
// 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 API prevents the caller from incorrectly writing `Parse(Lookup(key))` which causes a compile-time error, since `Lookup(key)` has 2 outputs.

Returning errors in this way encourages more robust and explicit error handling:

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

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 in the Google codebase should return additional values for errors.

Indent error flow

Handle errors before proceeding with the rest of your code. This improves the readability of the code by enabling the reader to find the normal path quickly. This same logic applies to any block which tests a condition then ends in a terminal condition (e.g., `return`, `panic`, `log.Fatal`).

Code that runs if the terminal condition is not met should appear after the `if` block, and should not be indented in an `else` clause.

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

```
// Bad:
if err != nil {
    // error handling
} else {
    // normal code that looks abnormal due to indentation
}
```

> **Tip:** If you are using a variable for more than a few lines of code, it is generally not worth using the `if`-with-initializer style. In these cases, it is usually better to move the declaration out and use a standard `if` statement:
>
> ```
> // Good:
> x, err := f()
> if err != nil {
>   // error handling
>   return
> }
> // lots of code that uses x
> // across multiple lines
> ```
>
> ```
> // Bad:
> if x, err := f(); err != nil {
>   // error handling
>   return
> } else {
>   // lots of code that uses x
> ```

```
  // across multiple lines
}
```

See Go Tip #1: Line of Sight and TotT: Reduce Code Complexity by Reducing Nesting for more details.

Language

Literal formatting

Go has an exceptionally powerful composite literal syntax, with which it is possible to express deeply-nested, complicated values in a single expression. Where possible, this literal syntax should be used instead of building values field-by-field. The `gofmt` formatting for literals is generally quite good, but there are some additional rules for keeping these literals readable and maintainable.

Field names

Struct literals should usually specify **field names** for types defined outside the current package.

Include field names for types from other packages.

```
// Good:
good := otherpkg.Type{A: 42}
```

The position of fields in a struct and the full set of fields (both of which are necessary to get right when field names are omitted) are not usually considered to be part of a struct's public API; specifying the field name is needed to avoid unnecessary coupling.

```
// Bad:
// https://pkg.go.dev/encoding/csv#Reader
r := csv.Reader{',', '#', 4, false, false, false, false}
```

Field names may be omitted within small, simple structs whose composition and order are documented as being stable.

```
// Good:
okay := image.Point{42, 54}
also := image.Point{X: 42, Y: 54}
```

For package-local types, field names are optional.

```
// Good:
okay := Type{42}
also := internalType{4, 2}
```

Field names should still be used if it makes the code clearer, and it is very common to do so. For example, a struct with a large number of fields should almost always be initialized with field names.

```
// Good:
okay := StructWithLotsOfFields{
  field1: 1,
  field2: "two",
  field3: 3.14,
  field4: true,
}
```

Matching braces

The closing half of a brace pair should always appear on a line with the same amount of indentation as the opening brace. One-line literals necessarily have this property. When the literal spans multiple lines, maintaining this property keeps the brace matching for literals the same as brace matching for common Go syntactic constructs like functions and `if` statements.

The most common mistake in this area is putting the closing brace on the same line as a value in a multi-line struct literal. In these cases, the line should end with a comma and the closing brace should appear on the next line.

```
// Good:
good := []*Type{{Key: "value"}}
```

```
// Good:
good := []*Type{
    {Key: "multi"},
    {Key: "line"},
}
```

```
// Bad:
bad := []*Type{
    {Key: "multi"},
    {Key: "line"}}
```

```
// Bad:
bad := []*Type{
    {
        Key: "value"},
}
```

Cuddled braces

Dropping whitespace between braces (aka “cuddling” them) for slice and array literals is only permitted when both of the following are true.

The indentation matches

The inner values are also literals or proto builders (i.e. not a variable or other expression)

```
// Good:
good := []*Type{
    { // Not cuddled
        Field: "value",
    },
    {
        Field: "value",
    },
}
```

```
// Good:
good := []*Type{{ // Cuddled correctly
    Field: "value",
}, {
    Field: "value",
}}
```

```
// Good:
good := []*Type{
    first, // Can't be cuddled
    {Field: "second"},
}
```

```
// Good:
okay := []*pb.Type{pb.Type_builder{
    Field: "first", // Proto Builders may be cuddled to save vertical
space
}.Build(), pb.Type_builder{
    Field: "second",
}.Build()}
```

```
// Bad:
```

```
bad := []*Type{
    first,
    {
        Field: "second",
    }}
```

Repeated type names

Repeated type names may be omitted from slice and map literals. This can be helpful in reducing clutter. A reasonable occasion for repeating the type names explicitly is when dealing with a complex type that is not common in your project, when the repetitive type names are on lines that are far apart and can remind the reader of the context.

```
// Good:
good := []*Type{
    {A: 42},
    {A: 43},
}
```

```
// Bad:
repetitive := []*Type{
    &Type{A: 42},
    &Type{A: 43},
}
```

```
// Good:
good := map[Type1]*Type2{
    {A: 1}: {B: 2},
    {A: 3}: {B: 4},
}
```

```
// Bad:
repetitive := map[Type1]*Type2{
    Type1{A: 1}: &Type2{B: 2},
    Type1{A: 3}: &Type2{B: 4},
}
```

Tip: If you want to remove repetitive type names in struct literals, you can run `gofmt -s`.

Zero-value fields

Zero-value fields may be omitted from struct literals when clarity is not lost as a result.

Well-designed APIs often employ zero-value construction for enhanced readability. For example, omitting the three zero-value fields from the following struct draws attention to the only option that is being specified.

```
// Bad:
import (
  "github.com/golang/leveldb"
  "github.com/golang/leveldb/db"
)

ldb := leveldb.Open("/my/table", &db.Options{
    BlockSize: 1<<16,
    ErrorIfDBExists: true,

    // These fields all have their zero values.
    BlockRestartInterval: 0,
    Comparer: nil,
    Compression: nil,
    FileSystem: nil,
    FilterPolicy: nil,
    MaxOpenFiles: 0,
    WriteBufferSize: 0,
    VerifyChecksums: false,
})
```

```
// Good:
import (
  "github.com/golang/leveldb"
  "github.com/golang/leveldb/db"
)

ldb := leveldb.Open("/my/table", &db.Options{
    BlockSize: 1<<16,
    ErrorIfDBExists: true,
})
```

Structs within table-driven tests often benefit from explicit field names, especially when the test struct is not trivial. This allows the author to omit the zero-valued fields entirely when the fields in question are not related to the test case. For example, successful test cases should omit any error-related or failure-related fields. In cases where the zero value is necessary to understand the test case, such as testing for zero or `nil` inputs, the field names should be specified.

Concise

```
tests := []struct {
    input      string
    wantPieces []string
    wantErr    error
}{
    {
        input:      "1.2.3.4",
        wantPieces: []string{"1", "2", "3", "4"},
    },
    {
        input:   "hostname",
        wantErr: ErrBadHostname,
    },
}
```

Explicit

```
tests := []struct {
    input    string
    wantIPv4 bool
    wantIPv6 bool
    wantErr  bool
}{
    {
        input:    "1.2.3.4",
        wantIPv4: true,
        wantIPv6: false,
    },
    {
        input:    "1:2::3:4",
        wantIPv4: false,
        wantIPv6: true,
    },
```

```
    {
        input:    "hostname",
        wantIPv4: false,
        wantIPv6: false,
        wantErr:  true,
    },
}
```

Nil slices

For most purposes, there is no functional difference between `nil` and the empty slice. Built-in functions like `len` and `cap` behave as expected on `nil` slices.

```
// Good:
import "fmt"

var s []int         // nil

fmt.Println(s)      // []
fmt.Println(len(s)) // 0
fmt.Println(cap(s)) // 0
for range s {...}   // no-op

s = append(s, 42)
fmt.Println(s)      // [42]
```

If you declare an empty slice as a local variable (especially if it can be the source of a return value), prefer the nil initialization to reduce the risk of bugs by callers.

Do not create APIs that force their clients to make distinctions between nil and the empty slice.

```
// Good:
// Ping pings its targets.
// Returns hosts that successfully responded.
func Ping(hosts []string) ([]string, error) { ... }
```

```
// Bad:
// Ping pings its targets and returns a list of hosts
// that successfully responded. Can be empty if the input was empty.
// nil signifies that a system error occurred.
func Ping(hosts []string) []string { ... }
```

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. This is typically accomplished by using `len` to check for emptiness, rather than `== nil`.

This implementation accepts both `nil` and zero-length slices as “empty”:

```
// Good:
// describeInts describes s with the given prefix, unless s is empty.
func describeInts(prefix string, s []int) {
    if len(s) == 0 {
        return
    }
    fmt.Println(prefix, s)
}
```

Instead of relying on the distinction as a part of the API:

```
// Bad:
func maybeInts() []int { /* ... */ }

// describeInts describes s with the given prefix; pass nil to skip
completely.
func describeInts(prefix string, s []int) {
  // The behavior of this function unintentionally changes depending on
what
  // maybeInts() returns in 'empty' cases (nil or []int{}).
  if s == nil {
    return
  }
  fmt.Println(prefix, s)
}

describeInts("Here are some ints:", maybeInts())
```

See in-band errors for further discussion.

Indentation confusion

Avoid introducing a line break if it would align the rest of the line with an indented code block. If this is unavoidable, leave a space to separate the code in the block from the wrapped line.

```
// Bad:
if longCondition1 && longCondition2 &&
    // Conditions 3 and 4 have the same indentation as the code within the
if.
    longCondition3 && longCondition4 {
    log.Info("all conditions met")
}
```

See the following sections for specific guidelines and examples:

Function formatting

Conditionals and loops

Literal formatting

Function formatting

The signature of a function or method declaration should remain on a single line to avoid indentation confusion.

Function argument lists can make some of the longest lines in a Go source file. However, they precede a change in indentation, and therefore it is difficult to break the line in a way that does not make subsequent lines look like part of the function body in a confusing way:

```
// Bad:
func (r *SomeType) SomeLongFunctionName(foo1, foo2, foo3 string,
    foo4, foo5, foo6 int) {
    foo7 := bar(foo1)
    // ...
}
```

See best practices for a few options for shortening the call sites of functions that would otherwise have many arguments.

```
// Good:
good := foo.Call(long, CallOptions{
    Names:   list,
    Of:      of,
    The:     parameters,
    Func:    all,
    Args:    on,
    Now:     separate,
```

```
    Visible: lines,
})
```

```
// Bad:
bad := foo.Call(
    long,
    list,
    of,
    parameters,
    all,
    on,
    separate,
    lines,
)
```

Lines can often be shortened by factoring out local variables.

```
// Good:
local := helper(some, parameters, here)
good := foo.Call(list, of, parameters, local)
```

Similarly, function and method calls should not be separated based solely on line length.

```
// Good:
good := foo.Call(long, list, of, parameters, all, on, one, line)
```

```
// Bad:
bad := foo.Call(long, list, of, parameters,
    with, arbitrary, line, breaks)
```

Do not add comments to specific function parameters. Instead, use an option struct or add more detail to the function documentation.

```
// Good:
good := server.New(ctx, server.Options{Port: 42})
```

```
// Bad:
bad := server.New(
    ctx,
    42, // Port
)
```

If call-sites are uncomfortably long, consider refactoring:

```
// Good:
// Sometimes variadic arguments can be factored out
replacements := []string{
    "from", "to", // related values can be formatted adjacent to one
another
    "source", "dest",
    "original", "new",
}

// Use the replacement struct as inputs to NewReplacer.
replacer := strings.NewReplacer(replacements...)
```

If the API cannot be changed or if the local call is unusual (whether or not the call is too long), it is always permissible to add line breaks if it aids in understanding the call.

```
// Good:
canvas.RenderCube(cube,
    x0, y0, z0,
    x0, y0, z1,
    x0, y1, z0,
    x0, y1, z1,
    x1, y0, z0,
    x1, y0, z1,
    x1, y1, z0,
    x1, y1, z1,
)
```

Note that the lines in the above example are not wrapped at a specific column boundary but are grouped based on co-ordinate triples.

Long string literals within functions should not be broken for the sake of line length. For functions that include such strings, a line break can be added after the string format, and the arguments can be provided on the next or subsequent lines. The decision about where the line breaks should go is best made based on semantic groupings of inputs, rather than based purely on line length.

```
// Good:
log.Warningf("Database key (%q, %d, %q) incompatible in transaction
started by (%q, %d, %q)",
    currentCustomer, currentOffset, currentKey,
    txCustomer, txOffset, txKey)
```

```
// Bad:
log.Warningf("Database key (%q, %d, %q) incompatible in"+
    " transaction started by (%q, %d, %q)",
    currentCustomer, currentOffset, currentKey, txCustomer,
    txOffset, txKey)
```

Conditionals and loops

An `if` statement should not be line broken; multi-line `if` clauses can lead to [indentation confusion].

```
// Bad:
// The second if statement is aligned with the code within the if block,
causing
// indentation confusion.
if db.CurrentStatusIs(db.InTransaction) &&
    db.ValuesEqual(db.TransactionKey(), row.Key()) {
    return db.Errorf(db.TransactionError, "query failed: row (%v): key
does not match transaction key", row)
}
```

If the short-circuit behavior is not required, the boolean operands can be extracted directly:

```
// Good:
inTransaction := db.CurrentStatusIs(db.InTransaction)
keysMatch := db.ValuesEqual(db.TransactionKey(), row.Key())
if inTransaction && keysMatch {
    return db.Error(db.TransactionError, "query failed: row (%v): key does
not match transaction key", row)
}
```

There may also be other locals that can be extracted, especially if the conditional is already repetitive:

```
// Good:
uid := user.GetUniqueUserID()
if db.UserIsAdmin(uid) || db.UserHasPermission(uid,
perms.ViewServerConfig) || db.UserHasPermission(uid, perms.CreateGroup) {
    // ...
}
```

```
// Bad:
if db.UserIsAdmin(user.GetUniqueUserID()) ||
```

```
db.UserHasPermission(user.GetUniqueUserID(), perms.ViewServerConfig) ||
db.UserHasPermission(user.GetUniqueUserID(), perms.CreateGroup) {
    // ...
}
```

`if` statements that contain closures or multi-line struct literals should ensure that the braces match to avoid indentation confusion.

```
// Good:
if err := db.RunInTransaction(func(tx *db.TX) error {
    return tx.Execute(userUpdate, x, y, z)
}); err != nil {
    return fmt.Errorf("user update failed: %s", err)
}
```

```
// Good:
if _, err := client.Update(ctx, &upb.UserUpdateRequest{
    ID:   userID,
    User: user,
}); err != nil {
    return fmt.Errorf("user update failed: %s", err)
}
```

Similarly, don't try inserting artificial linebreaks into `for` statements. You can always let the line simply be long if there is no elegant way to refactor it:

```
// Good:
for i, max := 0, collection.Size(); i < max &&
!collection.HasPendingWriters(); i++ {
    // ...
}
```

Often, though, there is:

```
// Good:
for i, max := 0, collection.Size(); i < max; i++ {
    if collection.HasPendingWriters() {
        break
    }
    // ...
}
```

`switch` and `case` statements should also remain on a single line.

```
// Good:
switch good := db.TransactionStatus(); good {
case db.TransactionStarting, db.TransactionActive, db.TransactionWaiting:
    // ...
case db.TransactionCommitted, db.NoTransaction:
    // ...
default:
    // ...
}
```

```
// Bad:
switch bad := db.TransactionStatus(); bad {
case db.TransactionStarting,
    db.TransactionActive,
    db.TransactionWaiting:
    // ...
case db.TransactionCommitted,
    db.NoTransaction:
    // ...
default:
    // ...
}
```

If the line is excessively long, indent all cases and separate them with a blank line to avoid [indentation confusion](#):

```
// Good:
switch db.TransactionStatus() {
case
    db.TransactionStarting,
    db.TransactionActive,
    db.TransactionWaiting,
    db.TransactionCommitted:

    // ...
case db.NoTransaction:
    // ...
```

```
default:
    // ...
}
```

In conditionals comparing a variable to a constant, place the variable value on the left hand side of the equality operator:

```
// Good:
if result == "foo" {
  // ...
}
```

Instead of the less clear phrasing where the constant comes first ("Yoda style conditionals"):

```
// Bad:
if "foo" == result {
  // ...
}
```

Copying

To avoid unexpected aliasing and similar bugs, be careful when copying a struct from another package. For example, synchronization objects such as `sync.Mutex` must not be copied.

The `bytes.Buffer` type contains a `[]byte` slice and, as an optimization for small strings, a small byte array to which the slice may refer. 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`.

```
// Bad:
b1 := bytes.Buffer{}
b2 := b1
```

Invoking a method that takes a value receiver can hide the copy. When you author an API, you should generally take and return pointer types if your structs contain fields that should not be copied.

These are acceptable:

```
// Good:
type Record struct {
  buf bytes.Buffer
  // other fields omitted
}
```

```
func New() *Record {...}

func (r *Record) Process(...) {...}

func Consumer(r *Record) {...}
```

But these are usually wrong:

```
// Bad:
type Record struct {
  buf bytes.Buffer
  // other fields omitted
}


func (r Record) Process(...) {...} // Makes a copy of r.buf

func Consumer(r Record) {...} // Makes a copy of r.buf
```

This guidance also applies to copying `sync.Mutex`.

Don't panic

Do not use `panic` for normal error handling. Instead, use `error` and multiple return values. See the [Effective Go section on errors](#).

Within `package main` and initialization code, consider [`log.Exit`](#) for errors that should terminate the program (e.g., invalid configuration), as in many of these cases a stack trace will not help the reader. Please note that [`log.Exit`](#) calls [`os.Exit`](#) and any deferred functions will not be run.

For errors that indicate "impossible" conditions, namely bugs that should always be caught during code review and/or testing, a function may reasonably return an error or call [`log.Fatal`](#).

Note: `log.Fatalf` is not the standard library log. See [#logging].

Must functions

Setup helper functions that stop the program on failure follow the naming convention `MustXYZ` (or `mustXYZ`). In general, they should only be called early on program startup, not on things like user input where normal Go error handling is preferred.

This often comes up for functions called to initialize package-level variables exclusively at [package](#)

initialization time (e.g. template.Must and regexp.MustCompile).

```
// Good:
func MustParse(version string) *Version {
    v, err := Parse(version)
    if err != nil {
        log.Fatalf("MustParse(%q) = _, %v", version, err)
    }
    return v
}

// Package level "constant". If we wanted to use `Parse`, we would have
had to
// set the value in `init`.
var DefaultVersion = MustParse("1.2.3")
```

Note: `log.Fatalf` is not the standard library log. See [#logging].

The same convention may be used in test helpers that only stop the current test (using `t.Fatal`). Such helpers are often convenient in creating test values, for example in struct fields of table driven tests, as functions that return errors cannot be directly assigned to a struct field.

```
// Good:
func mustMarshalAny(t *testing.T, m proto.Message) *anypb.Any {
  t.Helper()
  any, err := anypb.New(m)
  if err != nil {
    t.Fatalf("MustMarshalAny(t, m) = %v; want %v", err, nil)
  }
  return any
}

func TestCreateObject(t *testing.T) {
  tests := []struct{
    desc string
    data *anypb.Any
  }{
    {
      desc: "my test case",
```

```
        // Creating values directly within table driven test cases.
        data: mustMarshalAny(t, mypb.Object{}),
      },
      // ...
    }
    // ...
}
```

In both of these cases, the value of this pattern is that the helpers can be called in a "value" context. These helpers should not be called in places where it's difficult to ensure an error would be caught or in a context where an error should be checked (e.g., in many request handlers). For constant inputs, this allows tests to easily ensure that the `Must` arguments are well-formed, and for non-constant inputs it permits tests to validate that errors are properly handled or propagated.

Where `Must` functions are used in a test, they should generally be marked as a test helper and call `t.Fatal` on error (see error handling in test helpers for more considerations of using that).

They should not be used when ordinary error handling is possible (including with some refactoring):

```
// Bad:
func Version(o *servicepb.Object) (*version.Version, error) {
    // Return error instead of using Must functions.
    v := version.MustParse(o.GetVersionString())
    return dealiasVersion(v)
}
```

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. Sending on a channel that has been closed causes a panic.

```
// Bad:
ch := make(chan int)
ch <- 42
close(ch)
ch <- 13 // panic
```

Modifying still-in-use inputs "after the result isn't needed" can lead to data races. Leaving goroutines in-flight for arbitrarily long can lead to unpredictable memory usage.

Concurrent code should be written such that the goroutine lifetimes are obvious. Typically this will mean keeping synchronization-related code constrained within the scope of a function and factoring out the logic into [synchronous functions](). If the concurrency is still not obvious, it is important to document when and why the goroutines exit.

Code that follows best practices around context usage often helps make this clear. It is conventionally managed with a `context.Context`:

```
// Good:
func (w *Worker) Run(ctx context.Context) error {
    // ...
    for item := range w.q {
        // process returns at latest when the context is cancelled.
        go process(ctx, item)
    }
    // ...
}
```

There are other variants of the above that use raw signal channels like `chan struct{}`, synchronized variables, [condition variables](), and more. The important part is that the goroutine's end is evident for subsequent maintainers.

In contrast, the following code is careless about when its spawned goroutines finish:

```
// Bad:
func (w *Worker) Run() {
    // ...
    for item := range w.q {
        // process returns when it finishes, if ever, possibly not cleanly
        // handling a state transition or termination of the Go program
itself.
        go process(item)
    }
    // ...
}
```

This code may look OK, but there are several underlying problems:

The code probably has undefined behavior in production, and the program may not terminate cleanly, even if the operating system releases the resources.

The code is difficult to test meaningfully due to the code's indeterminate lifecycle.

The code may leak resources as described above.

See also:

Never start a goroutine without knowing how it will stop

Rethinking Classical Concurrency Patterns: slides, video

When Go programs end

Interfaces

Go interfaces generally belong in the package that *consumes* values of the interface type, not a package that *implements* the interface type. The implementing package should return concrete (usually pointer or struct) types. That way, new methods can be added to implementations without requiring extensive refactoring. See GoTip #49: Accept Interfaces, Return Concrete Types for more details.

Do not export a test double implementation of an interface from an API that consumes it. Instead, design the API so that it can be tested using the public API of the real implementation. See GoTip #42: Authoring a Stub for Testing for more details. Even when it is not feasible to use the real implementation, it may not be necessary to introduce an interface fully covering all methods in the real type; the consumer can create an interface containing only the methods it needs, as demonstrated in GoTip #78: Minimal Viable Interfaces.

To test packages that use Stubby RPC clients, use a real client connection. If a real server cannot be run in the test, Google's internal practice is to obtain a real client connection to a local test double using the internal rpctest package (coming soon!).

Do not define interfaces before they are used (see TotT: Code Health: Eliminate YAGNI Smells). Without a realistic example of usage, it is too difficult to see whether an interface is even necessary, let alone what methods it should contain.

Do not use interface-typed parameters if the users of the package do not need to pass different types for them.

Do not export interfaces that the users of the package do not need.

TODO: Write a more in-depth doc on interfaces and link to it here.

```
// Good:
package consumer // consumer.go

type Thinger interface { Thing() bool }

```

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

```
// Good:
package consumer // consumer_test.go

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

```
// Bad:
package producer

type Thinger interface { Thing() bool }

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

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

```
// Good:
package producer

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

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

Generics

Generics (formally called "Type Parameters") are allowed where they fulfill your business requirements. In many applications, a conventional approach using existing language features (slices, maps, interfaces, and so on) works just as well without the added complexity, so be wary of premature use. See the discussion on least mechanism.

When introducing an exported API that uses generics, make sure it is suitably documented. It's highly encouraged to include motivating runnable examples.

Do not use generics just because you are implementing an algorithm or data structure that does not care about the type of its member elements. If there is only one type being instantiated in practice, start by making your code work on that type without using generics at all. Adding polymorphism later will be straightforward compared to removing abstraction that is found to be unnecessary.

Do not use generics to invent domain-specific languages (DSLs). In particular, refrain from introducing error-handling frameworks that might put a significant burden on readers. Instead prefer established error handling practices. For testing, be especially wary of introducing assertion libraries or frameworks that result in less useful test failures.

In general:

Write code, don't design types. From a GopherCon talk by Robert Griesemer and Ian Lance Taylor.

If you have several types that share a useful unifying interface, consider modeling the solution using that interface. Generics may not be needed.

Otherwise, instead of relying on the `any` type and excessive type switching, consider generics.

See also:

Using Generics in Go, talk by Ian Lance Taylor

Generics tutorial on Go's webpage

Pass values

Do not pass pointers as function arguments just to save a few bytes. If a function reads 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 may increase in size. In particular, protocol buffer messages should generally be handled by pointer rather than by value. The pointer type satisfies the `proto.Message` interface (accepted by `proto.Marshal`, `protocmp.Transform`, etc.), and protocol buffer messages can be quite large and often grow larger over time.

Receiver type

A method receiver can be passed either as a value or a pointer, just as if it were a regular function parameter. The choice between the two is based on which method set(s) the method should be a part of.

Correctness wins over speed or simplicity. There are cases where you must use a pointer value. In other cases, pick pointers for large types or as future-proofing if you don't have a good sense of how the code will grow, and use values for simple plain old data.

The list below spells out each case in further detail:

If the receiver is a slice and the method doesn't reslice or reallocate the slice, use a value rather than a pointer.

```
// Good:
type Buffer []byte

func (b Buffer) Len() int { return len(b) }
```

If the method needs to mutate the receiver, the receiver must be a pointer.

```
// Good:
type Counter int

func (c *Counter) Inc() { *c++ }

// See https://pkg.go.dev/container/heap.
type Queue []Item

func (q *Queue) Push(x Item) { *q = append([]Item{x}, *q...) }
```

If the receiver is a struct containing fields that cannot safely be copied, use a pointer receiver. Common examples are `sync.Mutex` and other synchronization types.

```
// Good:
type Counter struct {
    mu    sync.Mutex
    total int
}

func (c *Counter) Inc() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.total++
}
```

Tip: Check the type's Godoc for information about whether it is safe or unsafe to copy.

If the receiver is a "large" struct or array, a pointer receiver may be more efficient. Passing a struct is equivalent to passing all of its fields or elements as arguments to the method. If that seems too large to pass by value, a pointer is a good choice.

For methods that will call or run concurrently with other functions that modify the receiver, use a value if those modifications should not be visible to your method; otherwise use a pointer.

If the receiver is a struct or array, any of whose elements is a pointer to something that may be mutated, prefer a pointer receiver to make the intention of mutability clear to the reader.

```
// Good:
type Counter struct {
    m *Metric
}

func (c *Counter) Inc() {
    c.m.Add(1)
}
```

If the receiver is a [built-in type](), such as an integer or a string, that does not need to be modified, use a value.

```
// Good:
type User string

func (u User) String() { return string(u) }
```

If the receiver is a map, function, or channel, use a value rather than a pointer.

```
// Good:
// See https://pkg.go.dev/net/http#Header.
type Header map[string][]string

func (h Header) Add(key, value string) { /* omitted */ }
```

If the receiver is a "small" array or struct that is naturally a value type with no mutable fields and no pointers, a value receiver is usually the right choice.

```
// Good:
// See https://pkg.go.dev/time#Time.
type Time struct { /* omitted */ }

func (t Time) Add(d Duration) Time { /* omitted */ }
```

When in doubt, use a pointer receiver.

As a general guideline, prefer to make the methods for a type either all pointer methods or all value methods.

Note: There is a lot of misinformation about whether passing a value or a pointer to a function can affect performance. The compiler can choose to pass pointers to values on the stack as well as copying values on the stack, but these considerations should not outweigh the readability and correctness of the code in most circumstances. When the performance does matter, it is important to profile both approaches with a realistic benchmark before deciding that one approach outperforms the other.

`switch` and `break`

Do not use `break` statements without target labels at the ends of `switch` clauses; they are redundant. Unlike in C and Java, `switch` clauses in Go automatically break, and a `fallthrough` statement is needed to achieve the C-style behavior. Use a comment rather than `break` if you want to clarify the purpose of an empty clause.

```
// Good:
switch x {
case "A", "B":
    buf.WriteString(x)
case "C":
    // handled outside of the switch statement
default:
    return fmt.Errorf("unknown value: %q", x)
}
```

```
// Bad:
switch x {
case "A", "B":
    buf.WriteString(x)
    break // this break is redundant
case "C":
    break // this break is redundant
default:
    return fmt.Errorf("unknown value: %q", x)
}
```

> **Note:** If a `switch` clause is within a `for` loop, using `break` within `switch` does not exit the enclosing `for` loop.
>
> ```
> for {
>   switch x {
>   case "A":
>      break // exits the switch, not the loop
>   }
> }
> ```

To escape the enclosing loop, use a label on the `for` statement:

```
loop:
  for {
    switch x {
    case "A":
       break loop // exits the loop
    }
  }
```

Synchronous functions

Synchronous functions return their results directly and finish any callbacks or channel operations before returning. Prefer synchronous functions over asynchronous functions.

Synchronous functions keep goroutines localized within a call. This helps to reason about their lifetimes, and avoid leaks and data races. Synchronous functions are also easier to test, since the caller can pass an input and check the output without the need for polling or synchronization.

If necessary, the caller can add concurrency by calling the function in a separate goroutine. However, it is quite difficult (sometimes impossible) to remove unnecessary concurrency at the caller side.

See also:

“Rethinking Classical Concurrency Patterns”, talk by Bryan Mills: slides, video

Type aliases

Use a *type definition*, `type T1 T2`, to define a new type. Use a *type alias*, `type T1 = T2`, to refer to an existing type without defining a new type. Type aliases are rare; their primary use is to aid migrating packages to new source code locations. Don’t use type aliasing when it is not needed.

Use %q

Go’s format functions (`fmt.Printf` etc.) have a `%q` verb which prints strings inside double-quotation marks.

```
// Good:
fmt.Printf("value %q looks like English text", someText)
```

Prefer using `%q` over doing the equivalent manually, using `%s`:

```
// Bad:
fmt.Printf("value \"%s\" looks like English text", someText)
// Avoid manually wrapping strings with single-quotes too:
```

```
fmt.Printf("value '%s' looks like English text", someText)
```

Using `%q` is recommended in output intended for humans where the input value could possibly be empty or contain control characters. It can be very hard to notice a silent empty string, but `""` stands out clearly as such.

Use any

Go 1.18 introduces an `any` type as an alias to `interface{}`. Because it is an alias, `any` is equivalent to `interface{}` in many situations and in others it is easily interchangeable via an explicit conversion. Prefer to use `any` in new code.

Common libraries

Flags

Go programs in the Google codebase use an internal variant of the standard `flag` package. It has a similar interface but interoperates well with internal Google systems. Flag names in Go binaries should prefer to use underscores to separate words, though the variables that hold a flag's value should follow the standard Go name style (mixed caps). Specifically, the flag name should be in snake case, and the variable name should be the equivalent name in camel case.

```
// Good:
var (
    pollInterval = flag.Duration("poll_interval", time.Minute, "Interval
to use for polling.")
)
```

```
// Bad:
var (
    poll_interval = flag.Int("pollIntervalSeconds", 60, "Interval to use
for polling in seconds.")
)
```

Flags must only be defined in `package main` or equivalent.

General-purpose packages should be configured using Go APIs, not by punching through to the command-line interface; don't let importing a library export new flags as a side effect. That is, prefer explicit function arguments or struct field assignment or much less frequently and under the strictest of scrutiny exported global variables. In the extremely rare case that it is necessary to break this rule, the flag name must clearly indicate the package that it configures.

If your flags are global variables, place them in their own `var` group, following the imports section.

There is additional discussion around best practices for creating complex CLIs with subcommands.

See also:

Tip of the Week #45: Avoid Flags, Especially in Library Code

Go Tip #10: Configuration Structs and Flags

Go Tip #80: Dependency Injection Principles

Logging

Go programs in the Google codebase use a variant of the standard `log` package. It has a similar but more powerful interface and interoperates well with internal Google systems. An open source version of this library is available as package `glog`, and open source Google projects may use that, but this guide refers to it as `log` throughout.

Note: For abnormal program exits, this library uses `log.Fatal` to abort with a stacktrace, and `log.Exit` to stop without one. There is no `log.Panic` function as in the standard library.

Tip: `log.Info(v)` is equivalent `log.Infof("%v", v)`, and the same goes for other logging levels. Prefer the non-formatting version when you have no formatting to do.

See also:

Best practices on logging errors and custom verbosily levels

When and how to use the log package to stop the program

Contexts

Values of the `context.Context` type carry security credentials, tracing information, deadlines, and cancellation signals across API and process boundaries. Unlike C++ and Java, which in the Google codebase use thread-local storage, Go programs pass contexts explicitly along the entire function call chain from incoming RPCs and HTTP requests to outgoing requests.

When passed to a function or method, `context.Context` is always the first parameter.

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

Exceptions are:

In an HTTP handler, where the context comes from `req.Context()`.

In streaming RPC methods, where the context comes from the stream.

Code using gRPC streaming accesses a context from a `Context()` method in the generated server type, which implements `grpc.ServerStream`. See [gRPC Generated Code documentation](#).

In entrypoint functions (see below for examples of such functions), use [`context.Background()`](#).

In binary targets: `main`

In general purpose code and libraries: `init`

In tests: `TestXXX`, `BenchmarkXXX`, `FuzzXXX`

> **Note**: It is very rare for code in the middle of a callchain to require creating a base context of its own using `context.Background()`. Always prefer taking a context from your caller, unless it's the wrong context.
>
> You may come across server libraries (the implementation of Stubby, gRPC, or HTTP in Google's server framework for Go) that construct a fresh context object per request. These contexts are immediately filled with information from the incoming request, so that when passed to the request handler, the context's attached values have been propagated to it across the network boundary from the client caller. Moreover, these contexts' lifetimes are scoped to that of the request: when the request is finished, the context is cancelled.
>
> Unless you are implementing a server framework, you shouldn't create contexts with `context.Background()` in library code. Instead, prefer using context detachment, which is mentioned below, if there is an existing context available. If you think you do need `context.Background()` outside of entrypoint functions, discuss it with the Google Go style mailing list before committing to an implementation.

The convention that `context.Context` comes first in functions also applies to test helpers.

```
// Good:
func readTestFile(ctx context.Context, t *testing.T, path string) string
{}
```

Do not add a context member to a struct type. Instead, add a context parameter to each method on the 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 outside Google's control. Such cases are very rare, and should be discussed with the Google Go style mailing list before implementation and readability review.

Code in the Google codebase that must spawn background operations which can run after the parent context has been cancelled can use an internal package for detachment. Follow [issue #40221](#) for discussions on an open source alternative.

Since contexts are immutable, it is fine to pass the same context to multiple calls that share the same deadline, cancellation signal, credentials, parent trace, and so on.

See also:

[Contexts and structs]

Custom contexts

Do not create custom context types or use interfaces other than `context.Context` in function signatures. There are no exceptions to this rule.

Imagine if every team had a custom context. Every function call from package `p` to package `q` would have to determine how to convert a `p.Context` to a `q.Context`, for all pairs of packages `p` and `q`. This is impractical and error-prone for humans, and it makes automated refactorings that add context parameters nearly impossible.

If you have application data to pass around, put it in a parameter, in the receiver, in globals, or in a `Context` value if it truly belongs there. Creating your own context type is not acceptable since it undermines the ability of the Go team to make Go programs work properly in production.

crypto/rand

Do not use package `math/rand` to generate keys, even throwaway ones. If 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.

```
// Good:
import (
    "crypto/rand"
    // "encoding/base64"
    // "encoding/hex"
    "fmt"

    // ...
)

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

```
}
```

Note: `log.Fatalf` is not the standard library log. See [#logging].

Useful test failures

It should be possible to diagnose a test's failure without reading the test's source. Tests should fail with helpful messages detailing:

What caused the failure

What inputs resulted in an error

The actual result

What was expected

Specific conventions for achieving this goal are outlined below.

Assertion libraries

Do not create "assertion libraries" as helpers for testing.

Assertion libraries are libraries that attempt to combine the validation and production of failure messages within a test (though the same pitfalls can apply to other test helpers as well). For more on the distinction between test helpers and assertion libraries, see [best practices].

```
// Bad:
var obj BlogPost

assert.IsNotNil(t, "obj", obj)
assert.StringEq(t, "obj.Type", obj.Type, "blogPost")
assert.IntEq(t, "obj.Comments", obj.Comments, 2)
assert.StringNotEq(t, "obj.Body", obj.Body, "")
```

Assertion libraries tend to either stop the test early (if `assert` calls `t.Fatalf` or `panic`) or omit relevant information about what the test got right:

```
// Bad:
package assert

func IsNotNil(t *testing.T, name string, val interface{}) {
    if val == nil {
        t.Fatalf("data %s = nil, want not nil", name)
```

```
    }
}

func StringEq(t *testing.T, name, got, want string) {
    if got != want {
        t.Fatalf("data %s = %q, want %q", name, got, want)
    }
}
```

Complex assertion functions often do not provide [useful failure messages]() and context that exists within the test function. Too many assertion functions and libraries lead to a fragmented developer experience: which assertion library should I use, what style of output format should it emit, etc.? Fragmentation produces unnecessary confusion, especially for library maintainers and authors of large-scale changes, who are responsible for fixing potential downstream breakages. Instead of creating a domain-specific language for testing, use Go itself.

Assertion libraries often factor out comparisons and equality checks. Prefer using standard libraries such as `cmp` and `fmt` instead:

```
// Good:
var got BlogPost

want := BlogPost{
    Comments: 2,
    Body:     "Hello, world!",
}

if !cmp.Equal(got, want) {
    t.Errorf("blog post = %v, want = %v", got, want)
}
```

For more domain-specific comparison helpers, prefer returning a value or an error that can be used in the test's failure message instead of passing `*testing.T` and calling its error reporting methods:

```
// Good:
func postLength(p BlogPost) int { return len(p.Body) }

func TestBlogPost_VeritableRant(t *testing.T) {
    post := BlogPost{Body: "I am Gunnery Sergeant Hartman, your senior
drill instructor."}
```

```
    if got, want := postLength(post), 60; got != want {
        t.Errorf("length of post = %v, want %v", got, want)
    }
}
```

Best Practice: Were `postLength` non-trivial, it would make sense to test it directly, independently of any tests that use it.

See also:

[Equality comparison and diffs]

[Print diffs]

For more on the distinction between test helpers and assertion helpers, see [best practices]

Identify the function

In most tests, failure messages should include the name of the function that failed, even though it seems obvious from the name of the test function. Specifically, your failure message should be `YourFunc(%v) = %v, want %v` instead of just `got %v, want %v`.

Identify the input

In most tests, failure messages should include the function inputs if they are short. If the relevant properties of the inputs are not obvious (for example, because the inputs are large or opaque), you should name your test cases with a description of what's being tested and print the description as part of your error message.

Got before want

Test outputs should include the actual value that the function returned before printing the value that was expected. A standard format for printing test outputs is `YourFunc(%v) = %v, want %v`. Where you would write "actual" and "expected", prefer using the words "got" and "want", respectively.

For diffs, directionality is less apparent, and as such it is important to include a key to aid in interpreting the failure. See the [section on printing diffs]. Whichever diff order you use in your failure messages, you should explicitly indicate it as a part of the failure message, because existing code is inconsistent about the ordering.

Full structure comparisons

If your function returns a struct (or any data type with multiple fields such as slices, arrays, and maps), avoid writing test code that performs a hand-coded field-by-field comparison of the struct. Instead, construct the data that you're expecting your function to return, and compare directly using a [deep comparison].

Note: This does not apply if your data contains irrelevant fields that obscure the intention of the test.

If your struct needs to be compared for approximate (or equivalent kind of semantic) equality or it contains fields that cannot be compared for equality (e.g., if one of the fields is an `io.Reader`), tweaking a [`cmp.Diff`]() or [`cmp.Equal`]() comparison with [`cmpopts`]() options such as [`cmpopts.IgnoreInterfaces`]() may meet your needs ([example]()).

If your function returns multiple return values, you don't need to wrap those in a struct before comparing them. Just compare the return values individually and print them.

```
// Good:
val, multi, tail, err := strconv.UnquoteChar(`\"Fran & Freddie's Diner\"`,
'"')
if err != nil {
  t.Fatalf(...)
}
if val != `"` {
  t.Errorf(...)
}
if multi {
  t.Errorf(...)
}
if tail != `Fran & Freddie's Diner"` {
  t.Errorf(...)
}
```

Compare stable results

Avoid comparing results that may depend on output stability of a package that you do not own. Instead, the test should compare on semantically relevant information that is stable and resistant to changes in dependencies. For functionality that returns a formatted string or serialized bytes, it is generally not safe to assume that the output is stable.

For example, [`json.Marshal`]() can change (and has changed in the past) the specific bytes that it emits. Tests that perform string equality on the JSON string may break if the `json` package changes how it serializes the bytes. Instead, a more robust test would parse the contents of the JSON string and ensure that it is semantically equivalent to some expected data structure.

Keep going

Tests should keep going for as long as possible, even after a failure, in order to print out all of the failed checks in a single run. This way, a developer who is fixing the failing test doesn't have to re-run the test after fixing each bug to find the next bug.

Prefer calling `t.Error` over `t.Fatal` for reporting a mismatch. When comparing several different properties of a function's output, use `t.Error` for each of those comparisons.

Calling `t.Fatal` is primarily useful for reporting an unexpected error condition, when subsequent comparison failures are not going to be meaningful.

For table-driven test, consider using subtests and use `t.Fatal` rather than `t.Error` and `continue`. See also [GoTip #25: Subtests: Making Your Tests Lean](#).

Best practice: For more discussion about when `t.Fatal` should be used, see [best practices](#).

Equality comparison and diffs

The == operator evaluates equality using [language-defined comparisons](#). Scalar values (numbers, booleans, etc) are compared based on their values, but only some structs and interfaces can be compared in this way. Pointers are compared based on whether they point to the same variable, rather than based on the equality of the values to which they point.

The [`cmp`](#) package can compare more complex data structures not appropriately handled by `==`, such as slices. Use [`cmp.Equal`](#) for equality comparison and [`cmp.Diff`](#) to obtain a human-readable diff between objects.

```
// Good:
want := &Doc{
    Type:     "blogPost",
    Comments: 2,
    Body:     "This is the post body.",
    Authors:  []string{"isaac", "albert", "emmy"},
}
if !cmp.Equal(got, want) {
    t.Errorf("AddPost() = %+v, want %+v", got, want)
}
```

As a general-purpose comparison library, `cmp` may not know how to compare certain types. For example, it can only compare protocol buffer messages if passed the [`protocmp.Transform`](#) option.

```
// Good:
if diff := cmp.Diff(want, got, protocmp.Transform()); diff != "" {
    t.Errorf("Foo() returned unexpected difference in protobuf messages (-want +got):\n%s", diff)
```

```
}
```

Although the `cmp` package is not part of the Go standard library, it is maintained by the Go team and should produce stable equality results over time. It is user-configurable and should serve most comparison needs.

Existing code may make use of the following older libraries, and may continue using them for consistency:

[`pretty`] produces aesthetically pleasing difference reports. However, it quite deliberately considers values that have the same visual representation as equal. In particular, `pretty` does not catch differences between nil slices and empty ones, is not sensitive to different interface implementations with identical fields, and it is possible to use a nested map as the basis for comparison with a struct value. It also serializes the entire value into a string before producing a diff, and as such is not a good choice for comparing large values. By default, it compares unexported fields, which makes it sensitive to changes in implementation details in your dependencies. For this reason, it is not appropriate to use `pretty` on protobuf messages.

Prefer using `cmp` for new code, and it is worth considering updating older code to use `cmp` where and when it is practical to do so.

Older code may use the standard library `reflect.DeepEqual` function to compare complex structures. `reflect.DeepEqual` should not be used for checking equality, as it is sensitive to changes in unexported fields and other implementation details. Code that is using `reflect.DeepEqual` should be updated to one of the above libraries.

Note: The `cmp` package is designed for testing, rather than production use. As such, it may panic when it suspects that a comparison is performed incorrectly to provide instruction to users on how to improve the test to be less brittle. Given cmp's propensity towards panicking, it makes it unsuitable for code that is used in production as a spurious panic may be fatal.

Level of detail

The conventional failure message, which is suitable for most Go tests, is `YourFunc(%v) = %v, want %v`. However, there are cases that may call for more or less detail:

Tests performing complex interactions should describe the interactions too. For example, if the same `YourFunc` is called several times, identify which call failed the test. If it's important to know any extra state of the system, include that in the failure output (or at least in the logs).

If the data is a complex struct with significant boilerplate, it is acceptable to describe only the important parts in the message, but do not overly obscure the data.

Setup failures do not require the same level of detail. If a test helper populates a Spanner table but Spanner was down, you probably don't need to include which test input you were going to store in the database. `t.Fatalf("Setup: Failed to set up test database: %s", err)` is usually helpful enough to resolve the issue.

Tip: Make your failure mode trigger during development. Review what the failure message looks like and whether a maintainer can effectively deal with the failure.

There are some techniques for reproducing test inputs and outputs clearly:

When printing string data, `%q` is often useful to emphasize that the value is important and to more easily spot bad values.

When printing (small) structs, `%+v` can be more useful than `%v`.

When validation of larger values fails, printing a diff can make it easier to understand the failure.

Print diffs

If your function returns large output then it can be hard for someone reading the failure message to find the differences when your test fails. Instead of printing both the returned value and the wanted value, make a diff.

To compute diffs for such values, `cmp.Diff` is preferred, particularly for new tests and new code, but other tools may be used. See types of equality for guidance regarding the strengths and weaknesses of each function.

`cmp.Diff`

`pretty.Compare`

You can use the `diff` package to compare multi-line strings or lists of strings. You can use this as a building block for other kinds of diffs.

Add some text to your failure message explaining the direction of the diff.

Something like `diff (-want +got)` is good when you're using the `cmp`, `pretty`, and `diff` packages (if you pass `(want, got)` to the function), because the `-` and `+` that you add to your format string will match the `-` and `+` that actually appear at the beginning of the diff lines. If you pass `(got, want)` to your function, the correct key would be `(-got +want)` instead.

The `messagediff` package uses a different output format, so the message `diff (want -> got)` is appropriate when you're using it (if you pass `(want, got)` to the function), because the direction of the arrow will match the direction of the arrow in the "modified" lines.

The diff will span multiple lines, so you should print a newline before you print the diff.

Test error semantics

When a unit test performs string comparisons or uses a vanilla `cmp` to check that particular kinds of errors are returned for particular inputs, you may find that your tests are brittle if any of those error messages are reworded in the future. Since this has the potential to turn your unit test into a change detector (see TotT: Change-Detector Tests Considered Harmful), don't use string comparison to check what type of error your function returns. However, it is permissible to use string comparisons to check that error messages coming from the package under test satisfy certain properties, for example, that it includes the parameter name.

Error values in Go typically have a component intended for human eyes and a component intended for semantic control flow. Tests should seek to only test semantic information that can be reliably observed, rather than display information that is intended for human debugging, as this is often subject to future changes. For guidance on constructing errors with semantic meaning see best-practices regarding errors. If an error with insufficient semantic information is coming from a dependency outside your control, consider filing a bug against the owner to help improve the API, rather than relying on parsing the error message.

Within unit tests, it is common to only care whether an error occurred or not. If so, then it is sufficient to only test whether the error was non-nil when you expected an error. If you would like to test that the error semantically matches some other error, then consider using `cmp` with `cmpopts.EquateErrors`.

> **Note:** If a test uses `cmpopts.EquateErrors` but all of its `wantErr` values are either `nil` or `cmpopts.AnyError`, then using `cmp` is unnecessary mechanism. Simplify the code by making the want field a `bool`. You can then use a simple comparison with `!=`.
>
> ```
> // Good:
> gotErr := f(test.input) != nil
> if gotErr != test.wantErr {
>     t.Errorf("f(%q) returned err = %v, want error presence = %v",
> test.input, gotErr, test.wantErr)
> }
> ```

See also GoTip #13: Designing Errors for Checking.

Test structure

Subtests

The standard Go testing library offers a facility to define subtests. This allows flexibility in setup and cleanup, controlling parallelism, and test filtering. Subtests can be useful (particularly for table-driven tests), but using them is not mandatory. See also the Go blog post about subtests.

Subtests should not depend on the execution of other cases for success or initial state, because subtests are expected to be able to be run individually with using `go test -run` flags or with Bazel test filter expressions.

Subtest names

Name your subtest such that it is readable in test output and useful on the command line for users of test filtering. When you use `t.Run` to create a subtest, the first argument is used as a descriptive name for the test. To ensure that test results are legible to humans reading the logs, choose subtest names that will remain useful and readable after escaping. Think of subtest names more like a function identifier than a prose description. The test runner replaces spaces with underscores, and escapes non-printing characters. If your test data benefits from a longer description, consider putting the description in a separate field (perhaps to be printed using `t.Log` or alongside failure messages).

Subtests may be run individually using flags to the [Go test runner](#) or Bazel [test filter](#), so choose descriptive names that are also easy to type.

> **Warning:** Slash characters are particularly unfriendly in subtest names, since they have [special meaning for test filters](#).
>
> ```
> # Bad:
> # Assuming TestTime and t.Run("America/New_York", ...)
> bazel test :mytest --test_filter="Time/New_York"    # Runs nothing!
> bazel test :mytest --test_filter="Time//New_York"   # Correct, but
> awkward.
> ```

To [identify the inputs](#) of the function, include them in the test's failure messages, where they won't be escaped by the test runner.

```
// Good:
func TestTranslate(t *testing.T) {
    data := []struct {
        name, desc, srcLang, dstLang, srcText, wantDstText string
    }{
        {
            name:        "hu=en_bug-1234",
            desc:        "regression test following bug 1234. contact:
cleese",
            srcLang:     "hu",
            srcText:     "cigarettát és egy öngyújtót kérek",
            dstLang:     "en",
            wantDstText: "cigarettes and a lighter please",
        }, // ...
    }
    for _, d := range data {
        t.Run(d.name, func(t *testing.T) {
```

```
            got := Translate(d.srcLang, d.dstLang, d.srcText)
            if got != d.wantDstText {
                t.Errorf("%s\nTranslate(%q, %q, %q) = %q, want %q",
                    d.desc, d.srcLang, d.dstLang, d.srcText, got,
d.wantDstText)
            }
        })
    }
}
```

Here are a few examples of things to avoid:

```
// Bad:
// Too wordy.
t.Run("check that there is no mention of scratched records or
hovercrafts", ...)
// Slashes cause problems on the command line.
t.Run("AM/PM confusion", ...)
```

Table-driven tests

Use table-driven tests when many different test cases can be tested using similar testing logic.

When testing whether the actual output of a function is equal to the expected output. For example, the many [tests of `fmt.Sprintf`] or the minimal snippet below.

When testing whether the outputs of a function always conform to the same set of invariants. For example, [tests for `net.Dial`].

Here is the minimal structure of a table-driven test, copied from the standard `strings` library. If needed, you may use different names, move the test slice into the test function, or add extra facilities such as subtests or setup and cleanup functions. Always keep [useful test failures] in mind.

```
// Good:
var compareTests = []struct {
    a, b string
    i    int
}{
    {"", "", 0},
    {"a", "", 1},
    {"", "a", -1},
```

```
    {"abc", "abc", 0},
    {"ab", "abc", -1},
    {"abc", "ab", 1},
    {"x", "ab", 1},
    {"ab", "x", -1},
    {"x", "a", 1},
    {"b", "x", -1},
    // test runtime·memeq's chunked implementation
    {"abcdefgh", "abcdefgh", 0},
    {"abcdefghi", "abcdefghi", 0},
    {"abcdefghi", "abcdefghj", -1},
}

func TestCompare(t *testing.T) {
    for _, tt := range compareTests {
        cmp := Compare(tt.a, tt.b)
        if cmp != tt.i {
            t.Errorf(`Compare(%q, %q) = %v`, tt.a, tt.b, cmp)
        }
    }
}
```

Note: The failure messages in this example above fulfill the guidance to identify the function and identify the input. There's no need to identify the row numerically.

When some test cases need to be checked using different logic from other test cases, it is more appropriate to write multiple test functions, as explained in GoTip #50: Disjoint Table Tests. The logic of your test code can get difficult to understand when each entry in a table has its own different conditional logic to check each output for its inputs. If test cases have different logic but identical setup, a sequence of subtests within a single test function might make sense.

You can combine table-driven tests with multiple test functions. For example, when testing that a function's output exactly matches the expected output and that the function returns a non-nil error for an invalid input, then writing two separate table-driven test functions is the best approach: one for normal non-error outputs, and one for error outputs.

Data-driven test cases

Table test rows can sometimes become complicated, with the row values dictating conditional behavior inside the test case. The extra clarity from the duplication between the test cases is necessary for readability.

```
// Good:
type decodeCase struct {
    name   string
    input  string
    output string
    err    error
}

func TestDecode(t *testing.T) {
    // setupCodex is slow as it creates a real Codex for the test.
    codex := setupCodex(t)

    var tests []decodeCase // rows omitted for brevity

    for _, test := range tests {
        t.Run(test.name, func(t *testing.T) {
            output, err := Decode(test.input, codex)
            if got, want := output, test.output; got != want {
                t.Errorf("Decode(%q) = %v, want %v", test.input, got,
want)
            }
            if got, want := err, test.err; !cmp.Equal(got, want) {
                t.Errorf("Decode(%q) err %q, want %q", test.input, got,
want)
            }
        })
    }
}

func TestDecodeWithFake(t *testing.T) {
    // A fakeCodex is a fast approximation of a real Codex.
    codex := newFakeCodex()

    var tests []decodeCase // rows omitted for brevity

    for _, test := range tests {
```

```
        t.Run(test.name, func(t *testing.T) {
            output, err := Decode(test.input, codex)
            if got, want := output, test.output; got != want {
                t.Errorf("Decode(%q) = %v, want %v", test.input, got,
want)
            }
            if got, want := err, test.err; !cmp.Equal(got, want) {
                t.Errorf("Decode(%q) err %q, want %q", test.input, got,
want)
            }
        })
    }
}
```

In the counterexample below, note how hard it is to distinguish between which type of `Codex` is used per test case in the case setup. (The highlighted parts run afoul of the advice from [TotT: Data Driven Traps!](#) .)

```
// Bad:
type decodeCase struct {
  name   string
  input  string
  codex  testCodex
  output string
  err    error
}

type testCodex int

const (
  fake testCodex = iota
  prod
)

func TestDecode(t *testing.T) {
  var tests []decodeCase // rows omitted for brevity

  for _, test := range tests {
    t.Run(test.name, func(t *testing.T) {
```

```
      var codex Codex
      switch test.codex {
      case fake:
        codex = newFakeCodex()
      case prod:
        codex = setupCodex(t)
      default:
        t.Fatalf("unknown codex type: %v", codex)
      }
      output, err := Decode(test.input, codex)
      if got, want := output, test.output; got != want {
        t.Errorf("Decode(%q) = %q, want %q", test.input, got, want)
      }
      if got, want := err, test.err; !cmp.Equal(got, want) {
        t.Errorf("Decode(%q) err %q, want %q", test.input, got, want)
      }
    })
  }
}
```

Identifying the row

Do not use the index of the test in the test table as a substitute for naming your tests or printing the inputs. Nobody wants to go through your test table and count the entries in order to figure out which test case is failing.

```
// Bad:
tests := []struct {
    input, want string
}{
    {"hello", "HELLO"},
    {"wORld", "WORLD"},
}
for i, d := range tests {
    if strings.ToUpper(d.input) != d.want {
        t.Errorf("failed on case #%d", i)
    }
}
```

Add a test description to your test struct and print it along failure messages. When using subtests, your subtest name should be effective in identifying the row.

Important: Even though `t.Run` scopes the output and execution, you must always identify the input. The table test row names must follow the subtest naming guidance.

Test helpers

A test helper is a function that performs a setup or cleanup task. All failures that occur in test helpers are expected to be failures of the environment (not from the code under test) — for example when a test database cannot be started because there are no more free ports on this machine.

If you pass a `*testing.T`, call `t.Helper` to attribute failures in the test helper to the line where the helper is called. This parameter should come after a context parameter, if present, and before any remaining parameters.

```
// Good:
func TestSomeFunction(t *testing.T) {
    golden := readFile(t, "testdata/golden-result.txt")
    // ... tests against golden ...
}

// readFile returns the contents of a data file.
// It must only be called from the same goroutine as started the test.
func readFile(t *testing.T, filename string) string {
    t.Helper()
    contents, err := runfiles.ReadFile(filename)
    if err != nil {
        t.Fatal(err)
    }
    return string(contents)
}
```

Do not use this pattern when it obscures the connection between a test failure and the conditions that led to it. Specifically, the guidance about assert libraries still applies, and `t.Helper` should not be used to implement such libraries.

Tip: For more on the distinction between test helpers and assertion helpers, see best practices.

Although the above refers to `*testing.T`, much of the advice stays the same for benchmark and fuzz helpers.

Test package

Tests in the same package

Tests may be defined in the same package as the code being tested.

To write a test in the same package:

Place the tests in a `foo_test.go` file

Use `package foo` for the test file

Do not explicitly import the package to be tested

```
# Good:
go_library(
    name = "foo",
    srcs = ["foo.go"],
    deps = [
        ...
    ],
)

go_test(
    name = "foo_test",
    size = "small",
    srcs = ["foo_test.go"],
    library = ":foo",
    deps = [
        ...
    ],
)
```

A test in the same package can access unexported identifiers in the package. This may enable better test coverage and more concise tests. Be aware that any examples declared in the test will not have the package names that a user will need in their code.

Tests in a different package

It is not always appropriate or even possible to define a test in the same package as the code being tested. In these cases, use a package name with the `_test` suffix. This is an exception to the "no underscores" rule to package names. For example:

If an integration test does not have an obvious library that it belongs to

```
// Good:
package gmailintegration_test

import "testing"
```

If defining the tests in the same package results in circular dependencies

```
// Good:
package fireworks_test

import (
  "fireworks"
  "fireworkstestutil" // fireworkstestutil also imports fireworks
)
```

Use package `testing`

The Go standard library provides the `testing` package. This is the only testing framework permitted for Go code in the Google codebase. In particular, assertion libraries and third-party testing frameworks are not allowed.

The `testing` package provides a minimal but complete set of functionality for writing good tests:

Top-level tests

Benchmarks

Runnable examples

Subtests

Logging

Failures and fatal failures

These are designed to work cohesively with core language features like composite literal and if-with-initializer syntax to enable test authors to write [clear, readable, and maintainable tests].

Non-decisions

A style guide cannot enumerate positive prescriptions for all matters, nor can it enumerate all matters about which it does not offer an opinion. That said, here are a few things where the readability community has previously debated and has not achieved consensus about.

Local variable initialization with zero value. `var i int` and `i := 0` are equivalent. See also initialization best practices.

Empty composite literal vs. `new` or `make`. `&File{}` and `new(File)` are equivalent. So are `map[string]bool{}` and `make(map[string]bool)`. See also composite declaration best practices.

got, want argument ordering in cmp.Diff calls. Be locally consistent, and include a legend in your failure message.

`errors.New` vs `fmt.Errorf` on non-formatted strings. `errors.New("foo")` and `fmt.Errorf("foo")` may be used interchangeably.

If there are special circumstances where they come up again, the readability mentor might make an optional comment, but in general the author is free to pick the style they prefer in the given situation.

Naturally, if anything not covered by the style guide does need more discussion, authors are welcome to ask – either in the specific review, or on internal message boards.