styleguide

Go Style Guide

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

Overview | Guide | Decisions | Best practices

Note: This is part of a series of documents that outline Go Style at Google. This document is normative **and** canonical. See the overview for more information.

Style principles

There are a few overarching principles that summarize how to think about writing readable Go code. The following are attributes of readable code, in order of importance:

Clarity: The code's purpose and rationale is clear to the reader.

Simplicity: The code accomplishes its goal in the simplest way possible.

Concision: The code has a high signal-to-noise ratio.

Maintainability: The code is written such that it can be easily maintained.

Consistency: The code is consistent with the broader Google codebase.

Clarity

The core goal of readability is to produce code that is clear to the reader.

Clarity is primarily achieved with effective naming, helpful commentary, and efficient code organization.

Clarity is to be viewed through the lens of the reader, not the author of the code. It is more important that code be easy to read than easy to write. Clarity in code has two distinct facets:

What is the code actually doing?

Why is the code doing what it does?

What is the code actually doing?

Go is designed such that it should be relatively straightforward to see what the code is doing. In cases of uncertainty or where a reader may require prior knowledge in order to understand the code, it is worth investing time in order to make the code's purpose clearer for future readers. For example, it may help to:

Use more descriptive variable names

Add additional commentary

Break up the code with whitespace and comments

Refactor the code into separate functions/methods to make it more modular

There is no one-size-fits-all approach here, but it is important to prioritize clarity when developing Go code.

Why is the code doing what it does?

The code's rationale is often sufficiently communicated by the names of variables, functions, methods, or packages. Where it is not, it is important to add commentary. The "Why?" is especially important when the code contains nuances that a reader may not be familiar with, such as:

A nuance in the language, e.g., a closure will be capturing a loop variable, but the closure is many lines away

A nuance of the business logic, e.g., an access control check that needs to distinguish between the actual user and someone impersonating a user

An API might require care to use correctly. For example, a piece of code may be intricate and difficult to follow for performance reasons, or a complex sequence of mathematical operations may use type conversions in an unexpected way. In these cases and many more, it is important that accompanying commentary and documentation explain these aspects so that future maintainers don't make a mistake and so that readers can understand the code without needing to reverse-engineer it.

It is also important to be aware that some attempts to provide clarity (such as adding extra commentary) can actually obscure the code's purpose by adding clutter, restating what the code already says, contradicting the code, or adding maintenance burden to keep the comments up-to-date. Allow the code to speak for itself (e.g., by making the symbol names themselves self-describing) rather than adding redundant comments. It is often better for comments to explain why something is done, not what the code is doing.

The Google codebase is largely uniform and consistent. It is often the case that code that stands out (e.g., by using an unfamiliar pattern) is doing so for a good reason, typically for performance. Maintaining this property is important to make it clear to readers where they should focus their attention when reading a new piece of code.

The standard library contains many examples of this principle in action. Among them:

Maintainer comments in `package sort`.

Good runnable examples in the same package, which benefit both users (they show up in godoc) and maintainers (they run as part of tests).

`strings.Cut` is only four lines of code, but they improve the clarity and correctness of callsites.

Simplicity

Your Go code should be simple for those using, reading, and maintaining it.

Go code should be written in the simplest way that accomplishes its goals, both in terms of behavior and performance. Within the Google Go codebase, simple code:

Is easy to read from top to bottom

Does not assume that you already know what it is doing

Does not assume that you can memorize all of the preceding code

Does not have unnecessary levels of abstraction

Does not have names that call attention to something mundane

Makes the propagation of values and decisions clear to the reader

Has comments that explain why, not what, the code is doing to avoid future deviation

Has documentation that stands on its own

Has useful errors and useful test failures

May often be mutually exclusive with “clever” code

Tradeoffs can arise between code simplicity and API usage simplicity. For example, it may be worthwhile to have the code be more complex so that the end user of the API may more easily call the API correctly. In contrast, it may also be worthwhile to leave a bit of extra work to the end user of the API so that the code remains simple and easy to understand.

When code needs complexity, the complexity should be added deliberately. This is typically necessary if additional performance is required or where there are multiple disparate customers of a particular library or service. Complexity may be justified, but it should come with accompanying documentation so that clients and future maintainers are able to understand and navigate the complexity. This should be supplemented with tests and examples that demonstrate its correct usage, especially if there is both a “simple” and a “complex” way to use the code.

This principle does not imply that complex code cannot or should not be written in Go or that Go code is not allowed to be complex. We strive for a codebase that avoids unnecessary complexity so that when complexity does appear, it indicates that the code in question requires care to understand and maintain. Ideally, there should be accompanying commentary that explains the rationale and identifies the care that should be taken. This often arises when optimizing code for performance; doing so often requires a more complex approach, like

preallocating a buffer and reusing it throughout a goroutine lifetime. When a maintainer sees this, it should be a clue that the code in question is performance-critical, and that should influence the care that is taken when making future changes. If employed unnecessarily, on the other hand, this complexity is a burden on those who need to read or change the code in the future.

If code turns out to be very complex when its purpose should be simple, this is often a signal to revisit the implementation to see if there is a simpler way to accomplish the same thing.

Least mechanism

Where there are several ways to express the same idea, prefer the one that uses the most standard tools. Sophisticated machinery often exists, but should not be employed without reason. It is easy to add complexity to code as needed, whereas it is much harder to remove existing complexity after it has been found to be unnecessary.

Aim to use a core language construct (for example a channel, slice, map, loop, or struct) when sufficient for your use case.

If there isn't one, look for a tool within the standard library (like an HTTP client or a template engine).

Finally, consider whether there is a core library in the Google codebase that is sufficient before introducing a new dependency or creating your own.

As an example, consider production code that contains a flag bound to a variable with a default value which must be overridden in tests. Unless intending to test the program's command-line interface itself (say, with `os/exec`), it is simpler and therefore preferable to override the bound value directly rather than by using `flag.Set`.

Similarly, if a piece of code requires a set membership check, a boolean-valued map (e.g., `map[string]bool`) often suffices. Libraries that provide set-like types and functionality should only be used if more complicated operations are required that are impossible or overly complicated with a map.

Concision

Concise Go code has a high signal-to-noise ratio. It is easy to discern the relevant details, and the naming and structure guide the reader through these details.

There are many things that can get in the way of surfacing the most salient details at any given time:

Repetitive code

Extraneous syntax

Opaque names

Unnecessary abstraction

Whitespace

Repetitive code especially obscures the differences between each nearly-identical section, and requires a reader to visually compare similar lines of code to find the changes. Table-driven testing is a good example of a mechanism that can concisely factor out the common code from the important details of each repetition, but the choice of which pieces to include in the table will have an impact on how easy the table is to understand.

When considering multiple ways to structure code, it is worth considering which way makes important details the most apparent.

Understanding and using common code constructions and idioms are also important for maintaining a high signal-to-noise ratio. For example, the following code block is very common in error handling, and the reader can quickly understand the purpose of this block.

```
// Good:
if err := doSomething(); err != nil {
    // ...
}
```

If code looks very similar to this but is subtly different, a reader may not notice the change. In cases like this, it is worth intentionally "boosting" the signal of the error check by adding a comment to call attention to it.

```
// Good:
if err := doSomething(); err == nil { // if NO error
    // ...
}
```

Maintainability

Code is edited many more times than it is written. Readable code not only makes sense to a reader who is trying to understand how it works, but also to the programmer who needs to change it. Clarity is key.

Maintainable code:

Is easy for a future programmer to modify correctly

Has APIs that are structured so that they can grow gracefully

Is clear about the assumptions that it makes and chooses abstractions that map to the structure of the problem, not to the structure of the code

Avoids unnecessary coupling and doesn't include features that are not used

Has a comprehensive test suite to ensure promised behaviors are maintained and important logic is correct, and the tests provide clear, actionable diagnostics in case of failure

When using abstractions like interfaces and types which by definition remove information from the context in which they are used, it is important to ensure that they provide sufficient benefit. Editors and IDEs can connect directly to a method definition and show the corresponding documentation when a concrete type is used, but can only refer to an interface definition otherwise. Interfaces are a powerful tool, but come with a cost, since the maintainer may need to understand the specifics of the underlying implementation in order to correctly use the interface, which must be explained within the interface documentation or at the call-site.

Maintainable code also avoids hiding important details in places that are easy to overlook. For example, in each of the following lines of code, the presence or lack of a single character is critical to understand the line:

```
// Bad:
// The use of = instead of := can change this line completely.
if user, err = db.UserByID(userID); err != nil {
    // ...
}
```

```
// Bad:
// The ! in the middle of this line is very easy to miss.
leap := (year%4 == 0) && (!(year%100 == 0) || (year%400 == 0))
```

Neither of these are incorrect, but both could be written in a more explicit fashion, or could have an accompanying comment that calls attention to the important behavior:

```
// Good:
u, err := db.UserByID(userID)
if err != nil {
    return fmt.Errorf("invalid origin user: %s", err)
}
user = u
```

```
// Good:
// Gregorian leap years aren't just year%4 == 0.
// See https://en.wikipedia.org/wiki/Leap_year#Algorithm.
var (
    leap4   = year%4 == 0
    leap100 = year%100 == 0
    leap400 = year%400 == 0
)
```

```
leap := leap4 && (!leap100 || leap400)
```

In the same way, a helper function that hides critical logic or an important edge-case could make it easy for a future change to fail to account for it properly.

Predictable names are another feature of maintainable code. A user of a package or a maintainer of a piece of code should be able to predict the name of a variable, method, or function in a given context. Function parameters and receiver names for identical concepts should typically share the same name, both to keep documentation understandable and to facilitate refactoring code with minimal overhead.

Maintainable code minimizes its dependencies (both implicit and explicit). Depending on fewer packages means fewer lines of code that can affect behavior. Avoiding dependencies on internal or undocumented behavior makes code less likely to impose a maintenance burden when those behaviors change in the future.

When considering how to structure or write code, it is worth taking the time to think through ways in which the code may evolve over time. If a given approach is more conducive to easier and safer future changes, that is often a good trade-off, even if it means a slightly more complicated design.

Consistency

Consistent code is code that looks, feels, and behaves like similar code throughout the broader codebase, within the context of a team or package, and even within a single file.

Consistency concerns do not override any of the principles above, but if a tie must be broken, it is often beneficial to break it in favor of consistency.

Consistency within a package is often the most immediately important level of consistency. It can be very jarring if the same problem is approached in multiple ways throughout a package, or if the same concept has many names within a file. However, even this should not override documented style principles or global consistency.

Core guidelines

These guidelines collect the most important aspects of Go style that all Go code is expected to follow. We expect that these principles be learned and followed by the time readability is granted. These are not expected to change frequently, and new additions will have to clear a high bar.

The guidelines below expand on the recommendations in Effective Go, which provide a common baseline for Go code across the entire community.

Formatting

All Go source files must conform to the format outputted by the `gofmt` tool. This format is enforced by a

presubmit check in the Google codebase. Generated code should generally also be formatted (e.g., by using `format.Source`), as it is also browsable in Code Search.

MixedCaps

Go source code uses `MixedCaps` or `mixedCaps` (camel case) rather than underscores (snake case) when writing multi-word names.

This applies even when it breaks conventions in other languages. For example, a constant is `MaxLength` (not `MAX_LENGTH`) if exported and `maxLength` (not `max_length`) if unexported.

Local variables are considered unexported for the purpose of choosing the initial capitalization.

Line length

There is no fixed line length for Go source code. If a line feels too long, it should be refactored instead of broken. If it is already as short as it is practical for it to be, the line should be allowed to remain long.

Do not split a line:

Before an indentation change (e.g., function declaration, conditional)

To make a long string (e.g., a URL) fit into multiple shorter lines

Naming

Naming is more art than science. In Go, names tend to be somewhat shorter than in many other languages, but the same general guidelines apply. Names should:

Not feel repetitive when they are used

Take the context into consideration

Not repeat concepts that are already clear

You can find more specific guidance on naming in decisions.

Local consistency

Where the style guide has nothing to say about a particular point of style, authors are free to choose the style that they prefer, unless the code in close proximity (usually within the same file or package, but sometimes within a team or project directory) has taken a consistent stance on the issue.

Examples of **valid** local style considerations:

Use of `%s` or `%v` for formatted printing of errors

Usage of buffered channels in lieu of mutexes

Examples of **invalid** local style considerations:

Line length restrictions for code

Use of assertion-based testing libraries

If the local style disagrees with the style guide but the readability impact is limited to one file, it will generally be surfaced in a code review for which a consistent fix would be outside the scope of the CL in question. At that point, it is appropriate to file a bug to track the fix.

If a change would worsen an existing style deviation, expose it in more API surfaces, expand the number of files in which the deviation is present, or introduce an actual bug, then local consistency is no longer a valid justification for violating the style guide for new code. In these cases, it is appropriate for the author to clean up the existing codebase in the same CL, perform a refactor in advance of the current CL, or find an alternative that at least does not make the local problem worse.