The Zen of Go | Dave Cheney

by Dave Cheney

This article was derived from my [GopherCon Israel 2020]() presentation. It's also quite long. If you'd prefer a shorter version, head over to [the-zen-of-go.netlify.com]().

A recording of the presentation is available on [YouTube]().

How should I write good code?

Something that I've been thinking about a lot recently, when reflecting on the body of my own work, is a common subtitle, *how should I write good code?* Given nobody actively seeks to write *bad* code, this leads to the question; *how do you know when you've written good Go code?*

If there's a continuum between good and bad, how to do we know what the good parts are? What are its properties, its attributes, its hallmarks, its patterns, and its idioms?

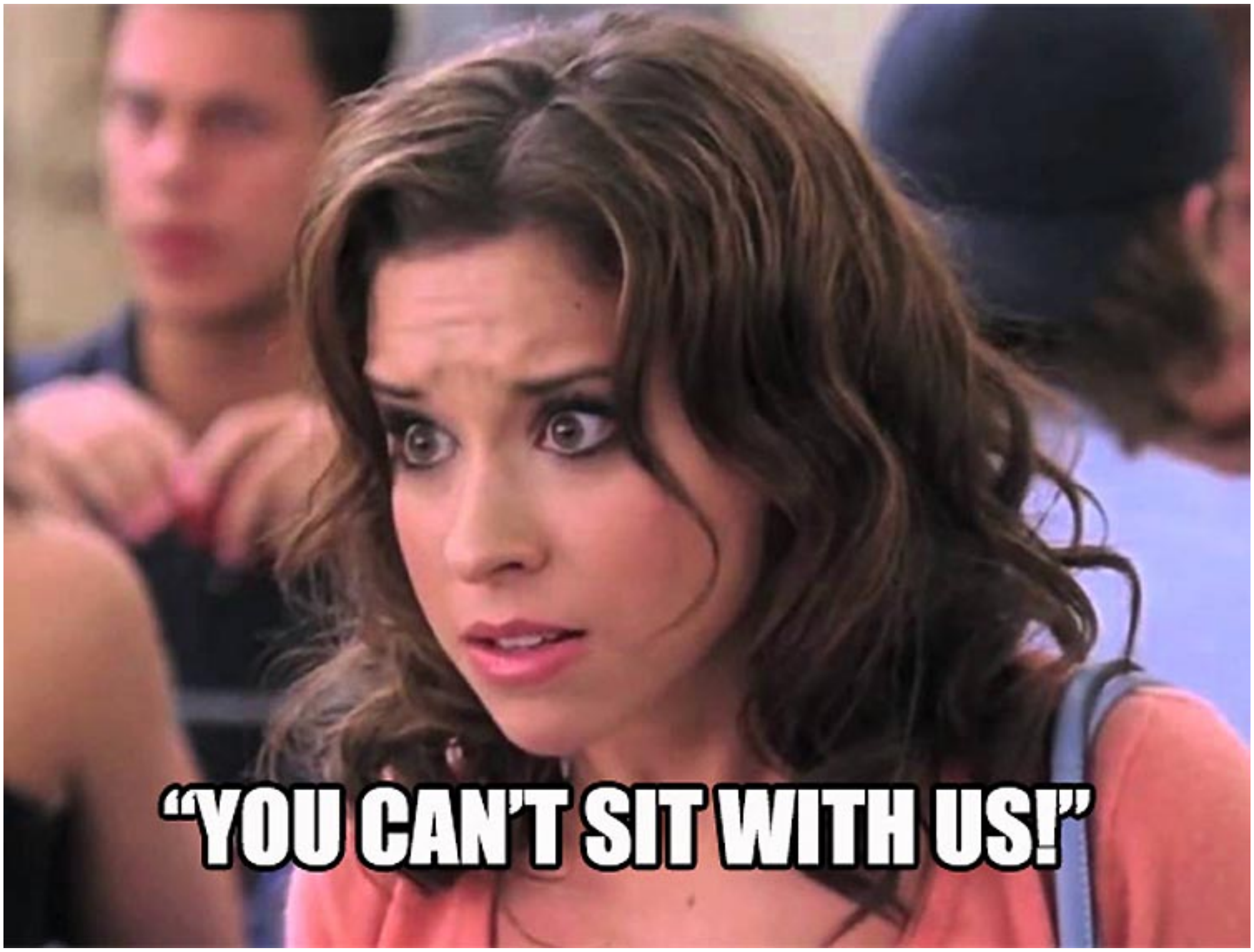
Idiomatic Go

Which brings me to idiomatic Go. To say that something is idiomatic is to say that it follows the style of the time. If something is not idiomatic, it is not following the prevailing style. It is unfashionable.

More importantly, to say to someone that their code is not idiomatic does not explain *why* it's not idiomatic. Why is this? Like all truths, the answer is found in the dictionary.

> idiom (noun): a group of words established by usage as having a meaning not deducible from those of the individual words.

Idioms are hallmarks of shared values. Idiomatic Go is not something you learn from a book, it's something that you acquire by being part of a community.



My concern with the mantra of idiomatic Go is, in many ways, it can be exclusionary. It's saying "you can't sit with us." After all, isn't that what we mean when critique of someone's work as non-idiomatic? They didn't do It right. It doesn't look right. It doesn't follow the style of time.

I offer that idiomatic Go is not a suitable mechanism for teaching how to write good Go code because it is defined, fundamentally, by telling someone they did it wrong. Wouldn't it be better if the advice we gave didn't alienate the author right at the point they were most willing to accept it?

Proverbs

Stepping away problematic idioms, what other cultural artefacts do Gophers have? Perhaps we can turn to Rob Pike's wonderful Go Proverbs. Are these suitable teaching tools? Will these tell newcomers how to write good Go code?

In general, I don't think so. This is not to dismiss Pike's work, it is just that the Go Proverbs, like Segoe Kensaku's original, are observations, not statements of value. Again, the dictionary comes to the rescue:

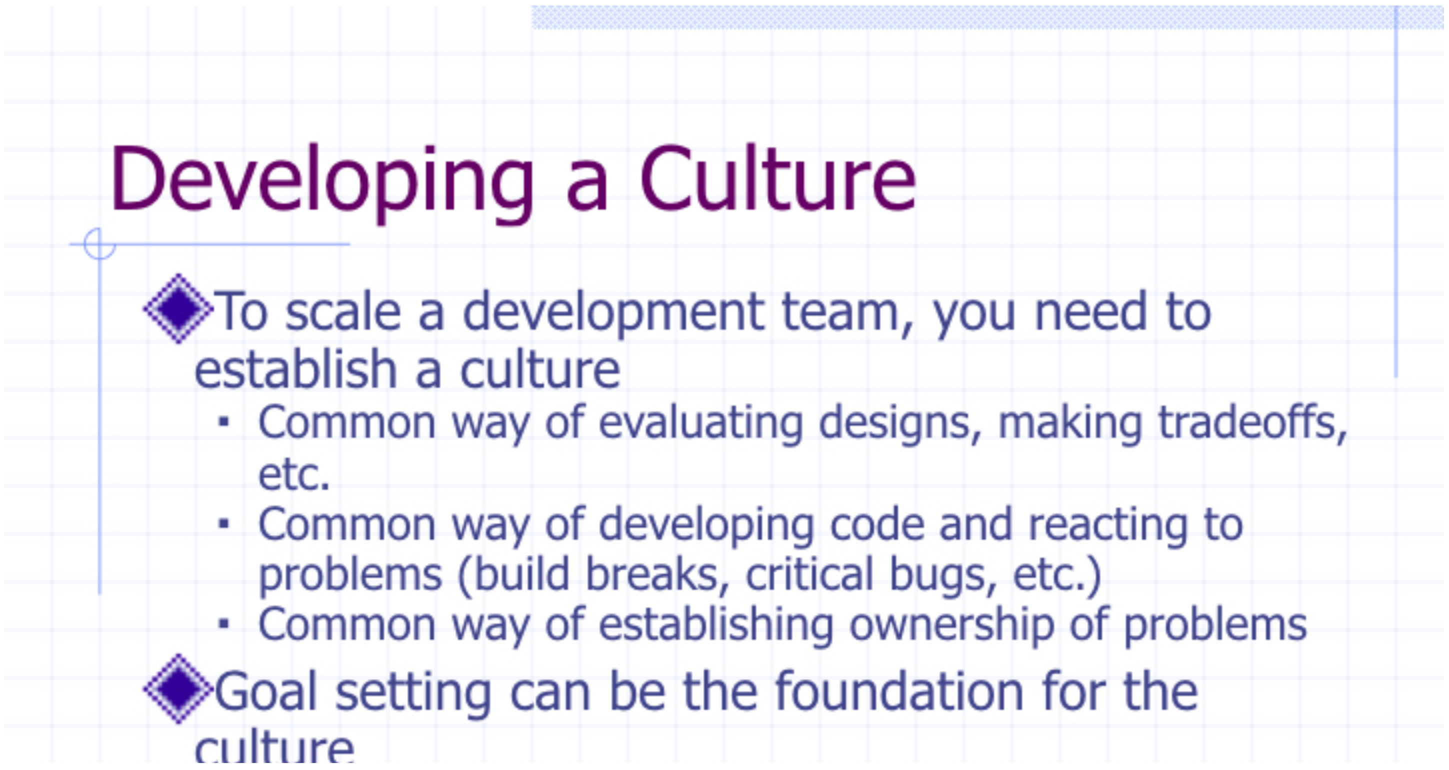
proverb (noun): a short, well-known pithy saying, stating a general truth or piece of advice.

The goal of the Go Proverbs are to reveal a deeper truth about the design of the language, but how useful is advice like the *empty interface says nothing* to a novice from a language that doesn't have structural typing?

It's important to recognise that, in a growing community, at any time the people learning Go far outnumber those who claim to have mastered the language. Thus proverbs are perhaps not the best teaching tool in this scenario.

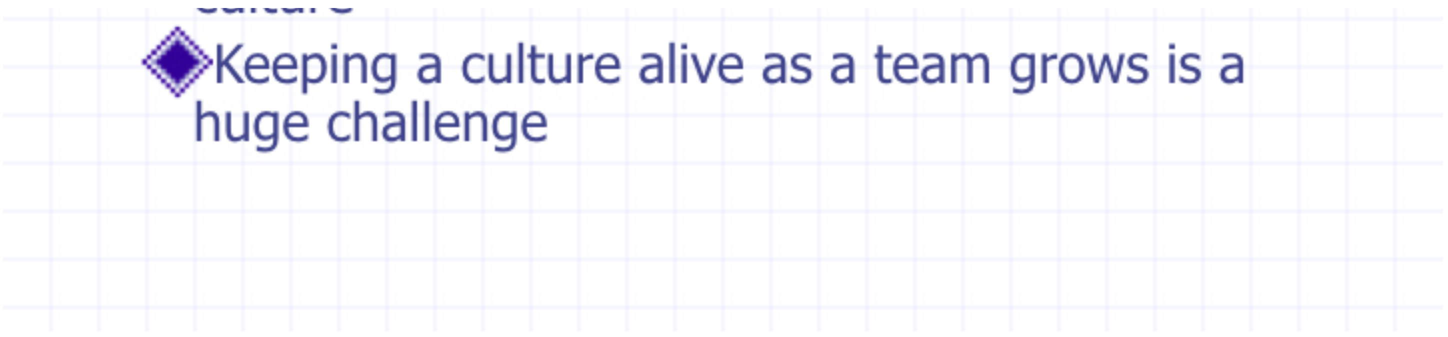
Engineering Values

Dan Luu found an old presentation by Mark Lucovsky about the engineering culture of the windows team around the windows NT-windows 2000 timeframe. The reason I mention it is Lukovsky's description of a culture as a common way of evaluating designs and making tradeoffs.



Developing a Culture

- To scale a development team, you need to establish a culture
  - Common way of evaluating designs, making tradeoffs, etc.
  - Common way of developing code and reacting to problems (build breaks, critical bugs, etc.)
  - Common way of establishing ownership of problems
- Goal setting can be the foundation for the culture



◆Keeping a culture alive as a team grows is a huge challenge

There are many ways of discussing culture, but with respect to an engineering culture Lucovsky’s description is apt. The central idea is *values guide decisions in an unknown design space*. The values of the NT team were; portability, reliability, security, and extensibility. Engineering values are, crudely translated, the way things are done around here.

Go’s values

What are the explicit values of Go? What are the core beliefs or philosophy that define the way a Go programmer interprets the world? How are they promulgated? How are they taught? How are they enforced? How do they change over time?

How will you, as a newly minted Go programmer, inculcate the engineering values of Go? Or, how will you, a seasoned Go professional promulgate your values to a future generations? And just so we’re clear, this process of knowledge transfer is not optional. Without new blood and new ideas, our community become myopic and wither.

The values of other languages

To set the scene for what I’m getting at we can look to other languages we see examples of their engineering values.

For example, C++ (and by extension Rust) believe that a programmer *should not have to pay for a feature they do not use*. If a program does not use some computationally expensive feature of the language, then it shouldn’t be forced to shoulder the cost of that feature. This value extends from the language, to its standard library, and is used as a yardstick for judging the design of all code written in C++.

In Java, and Ruby, and Smalltalk, the core value that *everything is an object* drives the design of programs around message passing, information hiding, and polymorphism. Designs that shoehorn a procedural style, or even a functional style, into these languages are considered to be wrong–or as Gophers would say, non idiomatic.

Turning to our own community, what are the engineering values that bind Go programmers? Discourse in our community is often fractious, so deriving a set of values from first principles would be a formidable challenge. Consensus is critical, but exponentially more difficult as the number of contributors to the discussion increases.

But what if someone had done the hard work for us.

The Zen of ~~Python~~ Go

Several decades ago Tim Peters sat down and penned *PEP-20*, the Zen of Python. Peters' attempted to document the engineering values that he saw Guido van Rossum apply in his role as BDFL for Python.

For the remainder of this article, I'm going to look towards the Zen of Python and ask, is there anything that can inform the engineering values of Go programmers?

A good package starts with a good name

Let's start with something spicy,

> "Namespaces are one honking great idea–let's do more of those!"
>
> *The Zen of Python, Item 19*

This is pretty unequivocal, Python programmers should use namespaces. Lots of them.

In Go parlance a namespace is a package. I doubt there is any question that grouping things into packages is good for design and potentially reuse. But there might be some confusion, especially if you're coming with a decade of experience in another language, about the right way to do this.

In Go each package should have a purpose, and the best way to know a package's purpose is by its name—a noun. A package's name describes what it provides. So too reinterpret Peters' words, every Go package should have a single purpose.

This is not a new idea, I've been saying this a while, but why should you do this rather than approach where packages are used for fine grained taxonomy? Why, because change.

> "Design is the art of arranging code to work today, and be changeable forever."
>
> *Sandi Metz*

Change is the name of the game we're in. What we do as programmers is manage change. When we do that well we call it design, or architecture. When we do it badly we call it technical debt, or legacy code.

If you are writing a program that works perfectly, one time, for one fixed set of inputs then nobody cares if the code is good or bad because ultimately the output of the program is all the business cares about.

But this is *never* true. Software has bugs, requirements change, inputs change, and very few programs are written solely to be executed once, thus your program *will* change over time. Maybe it's you who'll be tasked with this, more likely it will be someone else, but someone has to change that code. Someone has to maintain that code.

So, how can we make it easy to for programs to change? Interfaces everywhere? Make everything mockable? Pernicious dependency injection? Well, maybe, for some classes of programs, but not many, those techniques will be useful. However, for the majority of programs, designing something to be flexible up front is over engineering.

What if, instead, we take a position that rather than enhancing components, we replace them. Then the best way to know when something needs to be replaced, is when it doesn't do what it says on the tin.

A good package starts with choosing a good name. Think of your package's name as an elevator pitch, using just one word, to describe what it provides. When the name no longer matches the requirement, find a replacement.

Simplicity matters

> "Simple is better than complex."
>
> *The Zen of Python, Item 3*

PEP-20 says simple is better than complex, I couldn't agree more. A couple of years ago I made this tweet;

https://twitter.com/davecheney/status/539576755254611968

My observation, at least at the time, was that I couldn't think of a language introduced in my life time that didn't purport to be simple. Each new language offered as a justification, and an enticement, their inherent simplicity. But as I researched, I found that simplicity was not a core value of the many of the languages considered Go's contemporaries. [1] Maybe this is just a cheap shot, but could it be that either these languages aren't simple, or they don't *think* of themselves as being simple. They don't consider simplicity to be a core value.

Call me old fashioned, but when did being simple fall out of style? Why does the commercial software development industry continually, gleefully, forget this fundamental truth?

> "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."
>
> *C. A. R. Hoare, The Emperor's Old Clothes, 1980 Turing Award Lecture*

Simple does not mean easy, we know that. Often it is more work to make something simple to use, than easy to build.

> "Simplicity is prerequisite for reliability."
>
> *Edsger W Dijkstra, EWD498, 18 June 1975*

Why should we strive for simplicity? Why is important that Go programs be simple? Simple doesn't mean crude, it means readable and maintainable. Simple doesn't mean unsophisticated, it means reliable, relatable, and understandable.

> "Controlling complexity is the essence of computer programming."
>
> *Brian W. Kernighan, Software Tools (1976)*

Whether Python abides by its mantra of simplicity is a matter for debate, but Go holds simplicity as a core value. I think that we can all agree that when it comes to Go, simple code is preferable to clever code.

Avoid package level state

> "Explicit is better than implicit."
>
> *The Zen of Python, Item 2*

This is a place where I think Peters' was more aspirational than factual. Many things in Python are not explicit; decorators, dunder methods, and so on. Without doubt they are powerful, there's a reason those features exists. Each feature is something someone cared enough about to do the work to implement it, especially the complicated ones. But heavy use of those features makes is harder for the reader to predict the cost of an operation.

The good news is we have a choice, as Go programmers, to choose to make our code explicit. Explicit could mean many things, perhaps you may be thinking explicit is just a nice way of saying bureaucratic and long winded, but that's a superficial interpretation. It's a misnomer to focus only on the syntax on the page, to fret about line lengths and DRYing up expressions. The more valuable, in my opinon, place to be explicit are to do with coupling and with state.

Coupling is a measure of the amount one thing depends on another. If two things are tightly coupled, they move together. An action that affects one is directly reflected in another. Imagine a train, each carriage joined–ironically the correct word is coupled–together; where the engine goes, the carriages follow.

Another way to describe coupling is the word cohesion. Cohesion measures how well two things naturally belong together. We talk about a cohesive argument, or a cohesive team; all their parts fit together as if they were designed that way.

Why does coupling matter? Because just like trains, when you need to change a piece of code, all the code that is tightly coupled to it must change. A prime example, someone release a new version of their API and now your code doesn't compile.

APIs are an unavoidable source of coupling but there are more insidious forms of coupling. Clearly everyone knows that if an API's signature changes the data passing into and out of that call changes. It's right there in the signature of the function; I take values of these types and return values of other types. But what if the API passed data another way? What if every time you called this API the result was based on the previous time you called that API even though you didn’t change your parameters.

This is state, and management of state is *the* problem in computer science.

```
package counter

var count int

func Increment(n int) int {
        count += n
        return count
}
```

Suppose we have this simple `counter` package. You can call `Increment` to increment the counter, you can even get the value back if you `Increment` with a value of zero.

Suppose you had to test this code, how would you reset the counter after each test? Suppose you wanted to run those tests in parallel, could you do it? Now suppose that you wanted to count more than one thing per program, could you do it?

No, of course not. Clearly the answer is to encapsulate the `count` variable in a type.

```
package counter

type Counter struct {
        count int
}

func (c *Counter) Increment(n int) int {
        c.count += n
        return c.count
}
```

Now imagine that this problem isn’t restricted to just counters, but your applications main business logic. Can you test it in isolation? Can you test it in parallel? Can you use more than one instance at a time? If the answer those question is *no*, the reason is package level state.

Avoid package level state. Reduce coupling and spooky action at a distance by providing the dependencies a type needs as fields on that type rather than using package variables.

Plan for failure, not success

> “Errors should never pass silently.”
>
> *The Zen of Python, Item 10*

It’s been said of languages that favour exception handling follow the Samurai principle; *return victorious or not at all*. In exception based languages functions only return valid results. If they don’t succeed then control flow takes an entirely different path.

Unchecked exceptions are clearly an unsafe model to program in. How can you possibly write code that is robust in the presence of errors when you don’t know which statements could throw an exception? Java tried to make exceptions safer by introducing the notion of a checked exception which, to the best of my knowledge, has not been repeated in another mainstream language. There are plenty of languages which use exceptions but they all, with the singular exception of Java, do so in the unchecked variety.

Obviously Go chose a different path. Go programmers believe that robust programs are composed from pieces that handle the failure cases *before* they handle the happy path. In the space that Go was designed for; server programs, multi threaded programs, programs that handle input over the network, dealing with unexpected data, timeouts, connection failures and corrupted data must be front and centre of the programmer’s mind if they are to produce robust programs.

> “I think that error handling should be explicit, this should be a core value of the language.”
>
> *Peter Bourgon, GoTime #91*

I want to echo Peter’s assertion, as it was the impetus for this article. I think so much of the success of Go is due to the explicit way errors are handled. Go programmers thinks about the failure case first. We solve the “what if…” case first. This leads to programs where failures are handled at the point of writing, rather than the point they occur in production.

The verbosity of

```
if err != nil {
    return err
}
```

is outweighed by the value of deliberately handling each failure condition at the point at which they occur. Key to this is the cultural value of handling each and every error explicitly.

Return early rather than nesting deeply

> “Flat is better than nested.”
>
> *The Zen of Python, Item 5*

This is sage advice coming from a language where indentation is the primary form of control flow. How can we interpret this advice in terms of Go? `gofmt` controls the overall whitespace of a Go program so there's not thing doing there.

I wrote earlier about package names, and there is probably some advice here about avoiding a complicated package hierarchy. In my experience the more a programmer tries to subdivide and taxonimise their Go codebase the more they risk hitting the dead end that is package import loops.

I think the best application of item 5's advice is the control flow *within* a function. Simply put, avoid control flow that requires deep indentation.

> "Line of sight is a straight line along which an observer has unobstructed vision."
>
> *May Ryer, Code: Align the happy path to the left edge*

Mat Ryer describes this idea as line of sight coding. Light of sight coding means things like:

Using guard clauses to return early if a precondition is not met.

Placing the successful return statement at the end of the function rather than inside a conditional block.

Reducing the overall indentation level of the function by extracting functions and methods.

Key to this advice is the thing that you care about, the thing that the function does, is never in danger of sliding out of sight to the right of your screen. This style has a bonus side effect that you'll avoid pointless arguments about line lengths on your team.

Every time you indent you add another precondition to the programmers stack, consuming one of their 7 ±2 short term memory slots. Rather than nesting deeply, keep the successful path of the function close to the left hand side of your screen.

If you think it's slow, prove it with a benchmark

> "In the face of ambiguity, refuse the temptation to guess."
>
> *The Zen of Python, Item 12*

Programming is based on mathematics and logic, two concepts which rarely involve the element of chance. But there are many things we, as programmers, guess about every day. What does this variable do? What does this parameter do? What happens if I pass `nil` here? What happens if I call `Register` twice? There's actually a lot of guesswork in modern programming, especially when it comes to using libraries you didn't write.

> "APIs should be easy to use and hard to misuse."
>
> *Josh Bloch*

One of the best ways I know to help a programmer avoid having to guess is to, when building an API, focus on the default use case. Make it as easy as you can for the caller to do the most common thing. However, I've written and talked a lot about API design in the past, so instead my interpretation of item 12 is; *don't guess about performance*.

Despite how you may feel about Knuth's advice, one of the drivers of Go's success is its efficient execution. You can write efficient programs in Go and thus people *will* choose Go because of this. There are a lot of misconceptions about performance, so my request is, when you're looking to performance tune your code or you're facing some dogmatic advice like defer is slow, CGO is expensive, or always use atomics not mutexes, don't guess.

Don't complicate your code because of outdated dogma, and, if you think something is slow, first prove it with a benchmark. Go has excellent benchmarking and profiling tools that come in the distribution for free. Use them to find your bottlenecks.

Before you launch a goroutine, know when it will stop

At this point I think I think I've mined the valuable points from PEP-20 and possibly stretched its reinterpretation beyond the point of good taste. I think that's fine, because although this was a useful rhetorical device, ultimately we are talking about two different languages.

> "You type g o, a space, and then a function call. Three keystrokes, you can't make it much shorter than that. Three keystrokes and you've just started a sub process."
>
> *Rob Pike, Simplicity is Complicated, dotGo 2015*

The next two suggestions I'll dedicate to goroutines. Goroutines are the signature feature of the language, our answer for first class concurrency. They are so easy to use, just put the word `go` in front of the statement and you've launched that function asynchronously. It's so simple, no threads, no stack sizes, no thread pool executors, no ID's, no tracking completion status.

Goroutines are cheap. Because of the runtime's ability to multiplex goroutines onto a small pool of threads (which you don't have to manage), hundreds of thousands, millions of goroutines are easily accommodated. This opens up designs that would be not be practical under competing concurrency models like threads or evented callbacks.

But as cheap as goroutines are, they're not free. At a minimum there's a few kilobytes for their stack, which, when you're getting up into the 10^6 goroutines, does start to add up. This is not to say you shouldn't use millions of goroutines if that is what the design calls for, but when you do, it's critical that you keep track of them because 10^6 of anything can consume a non trivial amount of resources in aggregate.

Goroutines are the key to resource ownership in Go. To be useful a goroutine has to do something, and that means it almost always holds reference to, or ownership of, a resource; a lock, a network connection, a buffer with data, the sending end of a channel. While that goroutine is alive, the lock is held, the network connection remains open, the buffer retained and the receivers of the channel will continue to wait for more data.

The simplest way to free those resources is to tie them to the lifetime of the goroutine–when the goroutine exits, the resource has been freed. So while it's near trivial to start a goroutine, before you write those three letters, g o and a space, make sure you have an answer to these questions:

Under what condition will a goroutine stop? Go doesn't have a way to tell a goroutine to exit. There is no stop or kill function, for good reason. If we cannot command a goroutine to stop, we must instead ask it, politely. Almost always this comes down to a channel operation. Range loops over a channel exit when the channel is closed. A channel will become selectable if it is closed. The signal from one goroutine to another is best expressed as a closed channel.

What is required for that condition to arise? If channels are both the vehicle to communicate between goroutines and the mechanism for them to signal completion, the next question to the programmer becomes, who will close the channel, when will that happen?

What signal will you use to know the goroutine has stopped? When you signal a goroutine to stop, that stopping will happen at some time in the future relative to the goroutine's frame of reference. It might happen quickly in terms of human perception, but computers execute billions of instructions every second, and from the point of view of each goroutine, their execution of instructions is unsynchronised. The solution is often to use a channel to signal back or a waitgroup where a fan in approach is needed.

Leave concurrency to the caller

It is likely that in any serious Go program you write there will be concurrency involved. This raises the problem, many of the libraries and code that we write fall into this a one goroutine per connection, or worker pattern. How will you manage the lifetime of those goroutines?

`net/http` is a prime example. Shutting down the server owning the listening socket is relatively straight forward, but what about a goroutines spawned from that accepting socket? `net/http` does provide a context object inside the request object which can be used to signal–to code that is listening–that the request should be canceled, thereby terminating the goroutine, however it is less clear how to know when all of these things have been done. It's one thing to call `context.Cancel`, its another to know that the cancellation has completed.[2]

The point I want to make about `net/http` is that its a counter example to good practice. Because each connection is handled by a goroutine spawned inside the `net/http.Server` type, the program, living outside the `net/http` package, does not have an ability to control the goroutines spawned for the accepting socket.

This is an area of design that is still evolving, with efforts like go-kit's `run.Group` and the Go team's `ErrGroup` which provide a framework to execute, cancel and wait on functions run asynchronously.

The bigger design maxim here is for library writers, or anyone writing code that could be run asynchronously, leave the responsibility of starting to goroutine to your caller. Let the caller choose how they want to start, track, and wait on your functions execution.

Write tests to lock in the behaviour of your package’s API

Perhaps you were hoping to read an article from me where I didn’t rant about testing. Sadly, today is not that day.

Your tests are the contract about what your software does and does not do. Unit tests at the package level should lock in the behaviour of the package’s API. They describe, in code, what the package promises to do. If there is a unit test for each input permutation, you have defined the contract for what the code will do *in code*, not documentation.

This is a contract you can assert as simply as typing `go test`. At any stage, you can *know* with a high degree of confidence, that the behaviour people relied on before your change continues to function after your change.

Tests lock in api behaviour. Any change that adds, modifies or removes a public api must include changes to its tests.

Moderation is a virtue

Go is a simple language, only 25 keywords. In some ways this makes the features that are built into the language stand out. Equally these are the features that the language sells itself on, lightweight concurrency, structural typing.

I think all of us have experienced the confusion that comes from trying to use all of Go’s features at once. Who was so excited to use channels that they used them as much as they could, as often as they could? Personally for me I found the result was hard to test, fragile, and ultimately overcomplicated. Am I alone?

I had the same experience with goroutines, attempting to break the work into tiny units I created a hard to manage hurd of Goroutines and ultimately missed the observation that most of my goroutines were always blocked waiting for their predecessor– the code was ultimately sequential and I had added a lot of complexity for little real world benefit. Who has experienced something like this?

I had the same experience with embedding. Initially I mistook it for inheritance. Then later I recreated the fragile base class problem by composing complicated types, which already had several responsibilities, into more complicated mega types.

This is potentially the least actionable piece of advice, but one I think is important enough to mention. The advice is always the same, all things in moderation, and Go’s features are no exception. If you can, don’t reach for a goroutine, or a channel, or embed a struct, anonymous functions, going overboard with packages, interfaces for everything, instead prefer simpler approach rather than the clever approach.

Maintainability counts

I want to close with one final item from PEP-20,

> “Readability Counts.”
>
> *The Zen of Python, Item 7*

So much has been said, about the importance of readability, not just in Go, but all programming languages. People like me who stand on stages advocating for Go use words like simplicity, readability, clarity, productivity, but ultimately they are all synonyms for one word–*maintainability*.

The real goal is to write maintainable code. Code that can live on after the original author. Code that can exist not just as a point in time investment, but as a foundation for future value. It’s not that readability doesn’t matter, maintainability matters *more*.

Go is not a language that optimises for clever one liners. Go is not a language which optimises for the least number of lines in a program. We’re not optimising for the size of the source code on disk, nor how long it takes to type the program into an editor. Rather, we want to optimise our code to be clear to the reader. Because its the reader who’s going to have to maintain this code.

If you’re writing a program for yourself, maybe it only has to run once, or you’re the only person who’ll ever see it, then do what ever works for you. But if this is a piece of software that more than one person will contribute to, or that will be used by people over a long enough time that requirements, features, or the environment it runs in may change, then your goal must be for your program to be maintainable. If software cannot be maintained, then it will be rewritten; and that could be the last time your company will invest in Go.

Can the thing you worked hard to build be maintained after you’re gone? What can you do today to make it easier for someone to maintain your code tomorrow?

the-zen-of-go.netlify.com

This part of the talk had several screenshots of the landing pages for the websites for Ruby, Swift, Elm, Go, NodeJS, Python, Rust, highlighting how the language described itself.

I tend to pick on `net/http` a lot, and this is not because it is bad, in fact it is the opposite, it is the most successful, oldest, most used API in the Go codebase. And because of that its design, evolution, and shortcoming have been thoroughly picked over. Think of this as flattery, not criticism.