Chapter 8. Style Guides and Rules

Written by Shaindel Schwartz

Edited by Tom Manshreck

Most engineering organizations have rules governing their codebases—rules about where source files are stored, rules about the formatting of the code, rules about naming and patterns and exceptions and threads. Most software engineers are working within the bounds of a set of policies that control how they operate. At Google, to manage our codebase, we maintain a set of style guides that define our rules.

Rules are laws. They are not just suggestions or recommendations, but strict, mandatory laws. As such, they are universally enforceable—rules may not be disregarded except as approved on a need-to-use basis. In contrast to rules, guidance provides recommendations and best practices. These bits are good to follow, even highly advisable to follow, but unlike rules, they usually have some room for variance.

We collect the rules that we define, the do's and don'ts of writing code that must be followed, in our programming style guides, which are treated as canon. "Style" might be a bit of a misnomer here, implying a collection limited to formatting practices. Our style guides are more than that; they are the full set of conventions that govern our code. That's not to say that our style guides are strictly prescriptive; style guide rules may call for judgement, such as the rule to use names that are "as descriptive as possible, within reason." Rather, our style guides serve as the definitive source for the rules to which our engineers are held accountable.

We maintain separate style guides for each of the programming languages used at Google.[1] At a high level, all of the guides have similar goals, aiming to steer code development with an eye to sustainability. At the same time, there is a lot of variation among them in scope, length, and content. Programming languages have different strengths, different features, different priorities, and different historical paths to adoption within Google's ever-evolving repositories of code. It is far more practical, therefore, to independently tailor each language's guidelines. Some of our style guides are concise, focusing on a few overarching principles like naming and formatting, as demonstrated in our Dart, R, and Shell guides. Other style guides include far more detail, delving into specific language features and stretching into far lengthier documents—notably, our C++, Python, and Java guides. Some style guides put a premium on typical non-Google use of the language—our Go style guide is very short, adding just a few rules to a summary directive to adhere to the practices outlined in the externally recognized conventions. Others include rules that fundamentally differ from external norms; our C++ rules disallow use of exceptions, a language feature widely used outside of Google code.

The wide variance among even our own style guides makes it difficult to pin down the precise description of what a style guide should cover. The decisions guiding the development of Google's style guides stem from the need to keep our codebase sustainable. Other organizations' codebases will inherently have different requirements for sustainability that necessitate a different set of tailored rules. This chapter discusses the principles and processes that steer the development of our rules and guidance, pulling examples primarily from Google's C++, Python, and Java style guides.

Why Have Rules?

So why do we have rules? The goal of having rules in place is to encourage "good" behavior and discourage "bad" behavior. The interpretation of "good" and "bad" varies by organization, depending on what the organization cares about. Such designations are not universal preferences; good versus bad is subjective, and tailored to needs. For some organizations, "good" might promote usage patterns that support a small memory footprint or prioritize potential runtime optimizations. In other organizations, "good" might promote choices that exercise new language features. Sometimes, an organization cares most deeply about consistency, so that anything inconsistent with existing patterns is "bad." We must first recognize what a given organization values; we use rules and guidance to encourage and discourage behavior accordingly.

As an organization grows, the established rules and guidelines shape the common vocabulary of coding. A common vocabulary allows engineers to concentrate on what their code needs to say rather than how they're saying it. By shaping this vocabulary, engineers will tend to do the "good" things by default, even subconsciously. Rules thus give us broad leverage to nudge common development patterns in desired directions.

Creating the Rules

When defining a set of rules, the key question is not, “What rules should we have?” The question to ask is, “What goal are we trying to advance?” When we focus on the goal that the rules will be serving, identifying which rules support this goal makes it easier to distill the set of useful rules. At Google, where the style guide serves as law for coding practices, we do not ask, “What goes into the style guide?” but rather, “Why does something go into the style guide?” What does our organization gain by having a set of rules to regulate writing code?

Guiding Principles

Let’s put things in context: Google’s engineering organization is composed of more than 30,000 engineers. That engineering population exhibits a wild variance in skill and background. About 60,000 submissions are made each day to a codebase of more than two billion lines of code that will likely exist for decades. We’re optimizing for a different set of values than most other organizations need, but to some degree, these concerns are ubiquitous—we need to sustain an engineering environment that is resilient to both scale and time.

In this context, the goal of our rules is to manage the complexity of our development environment, keeping the codebase manageable while still allowing engineers to work productively. We are making a trade-off here: the large body of rules that helps us toward this goal does mean we are restricting choice. We lose some flexibility and we might even offend some people, but the gains of consistency and reduced conflict furnished by an authoritative standard win out.

Given this view, we recognize a number of overarching principles that guide the development of our rules, which must:

- Pull their weight
- Optimize for the reader
- Be consistent

- Avoid error-prone and surprising constructs
- Concede to practicalities when necessary

Rules must pull their weight

Not everything should go into a style guide. There is a nonzero cost in asking all of the engineers in an organization to learn and adapt to any new rule that is set. With too many rules,[2] not only will it become harder for engineers to remember all relevant rules as they write their code, but it also becomes harder for new engineers to learn their way. More rules also make it more challenging and more expensive to maintain the rule set.

To this end, we deliberately chose not to include rules expected to be self-evident. Google's style guide is not intended to be interpreted in a lawyerly fashion; just because something isn't explicitly outlawed does not imply that it is legal. For example, the C++ style guide has no rule against the use of `goto`. C++ programmers already tend to avoid it, so including an explicit rule forbidding it would introduce unnecessary overhead. If just one or two engineers are getting something wrong, adding to everyone's mental load by creating new rules doesn't scale.

Optimize for the reader

Another principle of our rules is to optimize for the reader of the code rather than the author. Given the passage of time, our code will be read far more frequently than it is written. We'd rather the code be tedious to type than difficult to read. In our Python style guide, when discussing conditional expressions, we recognize that they are shorter than `if` statements and therefore more convenient for code authors. However, because they tend to be more difficult for readers to understand than the more verbose `if` statements, we restrict their usage. We value "simple to read" over "simple to write." We're making a trade-off here: it can cost more upfront when engineers must repeatedly type potentially longer, descriptive names for variables and types. We choose to pay this cost for the readability it provides for all future readers.

As part of this prioritization, we also require that engineers leave explicit evidence of intended behavior in their code. We want readers to clearly understand what the code is doing as they read it. For example, our Java, JavaScript, and C++ style guides mandate use of the override annotation or keyword whenever a method overrides a superclass method. Without the explicit in-place evidence of design, readers can likely figure out this intent, though it would take a bit more digging on the part of each reader working through the code.

Evidence of intended behavior becomes even more important when it might be surprising. In C++, it is sometimes difficult to track the ownership of a pointer just by reading a snippet of code. If a pointer is passed to a function, without being familiar with the behavior of the function, we can't be sure what to expect. Does the caller still own the pointer? Did the function take ownership? Can I continue using the pointer after the function returns or might it have been deleted? To avoid this problem, our C++ style guide prefers the use of `std::unique_ptr` when ownership transfer is intended. `unique_ptr` is a construct that manages pointer ownership, ensuring that only one copy of the pointer ever exists. When a function takes a `unique_ptr` as an argument and intends to take ownership of the pointer, callers must explicitly invoke move semantics:

```
// Function that takes a Foo* and may or may not assume
// the passed pointer.
void TakeFoo(Foo* arg);

// Calls to the function don't tell the reader anything
// expect with regard to ownership after the function r
Foo* my_foo(NewFoo());
TakeFoo(my_foo);
```

Compare this to the following:

```
// Function that takes a std::unique_ptr<Foo>.
void TakeFoo(std::unique_ptr<Foo> arg);

// Any call to the function explicitly shows that owner
// yielded and the unique_ptr cannot be used after the
// returns.
std::unique_ptr<Foo> my_foo(FooFactory());
TakeFoo(std::move(my_foo));
```

Given the style guide rule, we guarantee that all call sites will include clear evidence of ownership transfer whenever it applies. With this signal in place, readers of the code don't need to understand the behavior of every function call. We provide enough information in the API to reason about its interactions. This clear documentation of behavior at the call sites ensures that code snippets remain readable and understandable. We aim for local reasoning, where the goal is clear understanding of what's happening at the call site without needing to find and reference other code, including the function's implementation.

Most style guide rules covering comments are also designed to support this goal of in-place evidence for readers. Documentation comments (the block comments prepended to a given file, class, or function) describe the design or intent of the code that follows. Implementation comments (the comments interspersed throughout the code itself) justify or highlight non-obvious choices, explain tricky bits, and underscore important parts of the code. We have style guide rules covering both types of comments, requiring engineers to provide the explanations another engineer might be looking for when reading through the code.

Be consistent

Our view on consistency within our codebase is similar to the philosophy we apply to our Google offices. With a large, distributed engineering population, teams are frequently split among offices, and Googlers often find themselves traveling to other sites. Although each office maintains its unique personality, embracing local flavor and style, for anything necessary to get work done, things are deliberately kept the same. A visiting Googler's badge will work with all local badge readers; any Google devices will always get WiFi; the video conferencing setup in any conference room will have the same interface. A Googler doesn't need to spend time learning how to get this all set up; they know that it will be the same no matter where they are. It's easy to move between offices and still get work done.

That's what we strive for with our source code. Consistency is what enables any engineer to jump into an unfamiliar part of the codebase and get to work fairly quickly. A local project can have its unique personality, but its tools are the same, its techniques are the same, its libraries are the same, and it all Just Works.

Advantages of consistency

Even though it might feel restrictive for an office to be disallowed from customizing a badge reader or video conferencing interface, the consistency benefits far outweigh the creative freedom we lose. It’s the same with code: being consistent may feel constraining at times, but it means more engineers get more work done with less effort:[3]

- When a codebase is internally consistent in its style and norms, engineers writing code and others reading it can focus on what’s getting done rather than how it is presented. To a large degree, this consistency allows for expert chunking.[4] When we solve our problems with the same interfaces and format the code in a consistent way, it’s easier for experts to glance at some code, zero in on what’s important, and understand what it’s doing. It also makes it easier to modularize code and spot duplication. For these reasons, we focus a lot of attention on consistent naming conventions, consistent use of common patterns, and consistent formatting and structure. There are also many rules that put forth a decision on a seemingly small issue solely to guarantee that things are done in only one way. For example, take the choice of the number of spaces to use for indentation or the limit set on line length.[5] It’s the consistency of having one answer rather than the answer itself that is the valuable part here.
- Consistency enables scaling. Tooling is key for an organization to scale, and consistent code makes it easier to build tools that can understand, edit, and generate code. The full benefits of the tools that depend on uniformity can’t be applied if everyone has little pockets of code that differ—if a tool can keep source files updated by adding missing imports or removing unused includes, if different projects are choosing different sorting strategies for their import lists, the tool might not be able to work everywhere. When everyone is using the same components and when everyone’s code follows the same rules for structure and organization, we can invest in tooling that works everywhere, building in automation for many of our maintenance tasks. If each team needed to separately invest in a bespoke version of the same tool, tailored for their unique environment, we would lose that advantage.
- Consistency helps when scaling the human part of an organization, too. As an organization grows, the number of engineers working on the codebase increases. Keeping the code that everyone is working on as consistent as possible enables better mobility across projects, minimizing the ramp-up time for an engineer switching teams and building in the ability for the

organization to flex and adapt as headcount needs fluctuate. A growing organization also means that people in other roles interact with the code—SREs, library engineers, and code janitors, for example. At Google, these roles often span multiple projects, which means engineers unfamiliar with a given team's project might jump in to work on that project's code. A consistent experience across the codebase makes this efficient.

- Consistency also ensures resilience to time. As time passes, engineers leave projects, new people join, ownership shifts, and projects merge or split. Striving for a consistent codebase ensures that these transitions are low cost and allows us nearly unconstrained fluidity for both the code and the engineers working on it, simplifying the processes necessary for long-term maintenance.

AT SCALE

A few years ago, our C++ style guide promised to almost never change style guide rules that would make old code inconsistent: "In some cases, there might be good arguments for changing certain style rules, but we nonetheless keep things as they are in order to preserve consistency."

When the codebase was smaller and there were fewer old, dusty corners, that made sense.

When the codebase grew bigger and older, that stopped being a thing to prioritize. This was (for the arbiters behind our C++ style guide, at least) a conscious change: when striking this bit, we were explicitly stating that the C++ codebase would never again be completely consistent, nor were we even aiming for that.

It would simply be too much of a burden to not only update the rules to current best practices, but to also require that we apply those rules to everything that's ever been written. Our Large Scale Change tooling and processes allow us to update almost all of our code to follow nearly every new pattern or syntax so that most old code exhibits the most recent approved style (see Chapter 22). Such mechanisms aren't perfect, however; when the codebase gets as large as it is, we can't be sure every bit of old code can conform to the new best practices. Requiring perfect consistency has reached the point where there's too much cost for the value.

Setting the standard

When we advocate for consistency, we tend to focus on internal consistency. Sometimes, local conventions spring up before global ones are adopted, and it isn’t reasonable to adjust everything to match. In that case, we advocate a hierarchy of consistency: “Be consistent” starts locally, where the norms within a given file precede those of a given team, which precede those of the larger project, which precede those of the overall codebase. In fact, the style guides contain a number of rules that explicitly defer to local conventions,[6] valuing this local consistency over a scientific technical choice.

However, it is not always enough for an organization to create and stick to a set of internal conventions. Sometimes, the standards adopted by the external community should be taken into account.

COUNTING SPACES

The Python style guide at Google initially mandated two-space indents for all of our Python code. The standard Python style guide, used by the external Python community, uses four-space indents. Most of our early Python development was in direct support of our C++ projects, not for actual Python applications. We therefore chose to use two-space indentation to be consistent with our C++ code, which was already formatted in that manner. As time went by, we saw that this rationale didn’t really hold up. Engineers who write Python code read and write other Python code much more often than they read and write C++ code. We were costing our engineers extra effort every time they needed to look something up or reference external code snippets. We were also going through a lot of pain each time we tried to export pieces of our code into open source, spending time reconciling the differences between our internal code and the external world we wanted to join.

When the time came for Starlark (a Python-based language designed at Google to serve as the build description language) to have its own style guide, we chose to change to using four-space indents to be consistent with the outside world.[7]

If conventions already exist, it is usually a good idea for an organization to be consistent with the outside world. For small, self-contained, and short-lived efforts, it likely won’t make a difference; internal consistency matters more than anything happening outside the project's limited scope. Once the passage of time and potential scaling become factors, the likelihood of your code interacting with outside projects or even ending up in the outside world increase. Looking long-term, adhering to the widely accepted standard will likely pay off.

Avoid error-prone and surprising constructs

Our style guides restrict the use of some of the more surprising, unusual, or tricky constructs in the languages that we use. Complex features often have subtle pitfalls not obvious at first glance. Using these features without thoroughly understanding their complexities makes it easy to misuse them and introduce bugs. Even if a construct is well understood by a project's engineers, future project members and maintainers are not guaranteed to have the same understanding.

This reasoning is behind our Python style guide ruling to avoid using power features such as reflection. The reflective Python functions `hasattr()` and `getattr()` allow a user to access attributes of objects using strings:

```
if hasattr(my_object, 'foo'):
some_var = getattr(my_object, 'foo')
```

Now, with that example, everything might seem fine. But consider this:

some_file.py:

```
A_CONSTANT = [
'foo',
'bar',
'baz',
]
```

other_file.py:

```
values = []
for field in some_file.A_CONSTANT:
values.append(getattr(my_object, field))
```

When searching through code, how do you know that the fields `foo`, `bar`, and `baz` are being accessed here? There's no clear evidence left for the reader. You don't easily see and therefore can't easily validate which strings are used to access attributes of your object. What if, instead of reading those values from `A_CONSTANT`, we read them from a Remote Procedure Call (RPC) request message or from a data store? Such obfuscated code could cause a major security flaw, one that would be very difficult to notice, simply by validating the message incorrectly. It's also difficult to test and verify such code.

Python's dynamic nature allows such behavior, and in very limited circumstances, using `hasattr()` and `getattr()` is valid. In most cases, however, they just cause obfuscation and introduce bugs.

Although these advanced language features might perfectly solve a problem for an expert who knows how to leverage them, power features are often more difficult to understand and are not very widely used. We need all of our engineers able to operate in the codebase, not just the experts. It's not just support for the novice software engineer, but it's also a better environment for SREs—if an SRE is debugging a production outage, they will jump into any bit of suspect code, even code written in a language in which they are not fluent. We place higher value on simplified, straightforward code that is easier to understand and maintain.

Concede to practicalities

In the words of Ralph Waldo Emerson: "A foolish consistency is the hobgoblin of little minds." In our quest for a consistent, simplified codebase, we do not want to blindly ignore all else. We know that some of the rules in our style guides will encounter cases that warrant exceptions, and that's OK. When necessary, we permit concessions to optimizations and practicalities that might otherwise conflict with our rules.

Performance matters. Sometimes, even if it means sacrificing consistency or readability, it just makes sense to accommodate performance optimizations. For example, although our C++ style guide prohibits use of exceptions, it includes a rule that allows the use of `noexcept`, an exception-related language specifier that can trigger compiler optimizations.

Interoperability also matters. Code that is designed to work with specific non-Google pieces might do better if tailored for its target. For example, our C++ style guide includes an exception to the general CamelCase naming guideline that permits use of the standard library's snake_case style for entities that mimic standard library features.[8] The C++ style guide also allows exemptions for Windows programming, where compatibility with platform features requires multiple inheritance, something explicitly forbidden for all other C++ code. Both our Java and JavaScript style guides explicitly state that generated code, which frequently interfaces with or depends on components outside of a project's ownership, is out of scope for the guide's rules.[9] Consistency is vital; adaptation is key.

The Style Guide

So, what does go into a language style guide? There are roughly three categories into which all style guide rules fall:

- Rules to avoid dangers
- Rules to enforce best practices
- Rules to ensure consistency

Avoiding danger

First and foremost, our style guides include rules about language features that either must or must not be done for technical reasons. We have rules about how to use static members and variables; rules about using lambda expressions; rules about handling exceptions; rules about building for threading, access control, and class inheritance. We cover which language features to use and which constructs to avoid. We call out standard vocabulary types that may be used and for what purposes. We specifically include rulings on the hard-to-use and the hard-to-use-correctly—some language features have nuanced usage patterns that might not be intuitive or easy to apply properly, causing subtle bugs to creep in. For each ruling in the guide, we aim to include the pros and cons that were weighed with an explanation of the decision that was reached. Most of these decisions are based on the need for resilience to time, supporting and encouraging maintainable language usage.

Enforcing best practices

Our style guides also include rules enforcing some best practices of writing source code. These rules help keep the codebase healthy and maintainable. For example, we specify where and how code authors must include comments.[10] Our rules for comments cover general conventions for commenting and extend to include specific cases that must include in-code documentation—cases in which intent is not always obvious, such as fall-through in switch statements, empty exception catch blocks, and template metaprogramming. We also have rules detailing the structuring of source files, outlining the organization of expected content. We have rules about naming: naming of packages, of classes, of functions, of variables. All of these rules are intended to guide engineers to practices that support healthier, more sustainable code.

Some of the best practices enforced by our style guides are designed to make source code more readable. Many formatting rules fall under this category. Our style guides specify when and how to use vertical and horizontal whitespace in order to improve readability. They also cover line length limits and brace alignment. For some languages, we cover formatting requirements by deferring to autoformatting tools—gofmt for Go, dartfmt for Dart. Itemizing a detailed list of formatting requirements or naming a tool that must be applied, the goal is the same: we have a consistent set of formatting rules designed to improve readability that we apply to all of our code.

Our style guides also include limitations on new and not-yet-well-understood language features. The goal is to preemptively install safety fences around a feature's potential pitfalls while we all go through the learning process. At the same time, before everyone takes off running, limiting use gives us a chance to watch the usage patterns that develop and extract best practices from the examples we observe. For these new features, at the outset, we are sometimes not sure of the proper guidance to give. As adoption spreads, engineers wanting to use the new features in different ways discuss their examples with the style guide owners, asking for allowances to permit additional use cases beyond those covered by the initial restrictions. Watching the waiver requests that come in, we get a sense of how the feature is getting used and eventually collect enough examples to generalize good practice from bad. After we have that information, we can circle back to the restrictive ruling and amend it to allow wider use.

CASE STUDY: INTRODUCING STD::UNIQUE_PTR

When C++11 introduced `std::unique_ptr`, a smart pointer type that expresses exclusive ownership of a dynamically allocated object and deletes the object when the `unique_ptr` goes out of scope, our style guide initially disallowed usage. The behavior of the `unique_ptr` was unfamiliar to most engineers, and the related move semantics that the language introduced were very new and, to most engineers, very confusing. Preventing the introduction of `std::unique_ptr` in the codebase seemed the safer choice. We updated our tooling to catch references to the disallowed type and kept our existing guidance recommending other types of existing smart pointers.

Time passed. Engineers had a chance to adjust to the implications of move semantics and we became increasingly convinced that using `std::unique_ptr` was directly in line with the goals of our style guidance. The information regarding object ownership that a `std::unique_ptr` facilitates at a function call site makes it far easier for a reader to understand that code. The added complexity of introducing this new type, and the novel move semantics that come with it, was still a strong concern, but the significant improvement in the long-term overall state of the codebase made the adoption of `std::unique_ptr` a worthwhile trade-off.

Building in consistency

Our style guides also contain rules that cover a lot of the smaller stuff. For these rules, we make and document a decision primarily to make and document a decision. Many rules in this category don’t have significant technical impact. Things like naming conventions, indentation spacing, import ordering: there is usually no clear, measurable, technical benefit for one form over another, which might be why the technical community tends to keep debating them.[11] By choosing one, we’ve dropped out of the endless debate cycle and can just move on. Our engineers no longer spend time discussing two spaces versus four. The important bit for this category of rules is not *what* we’ve chosen for a given rule so much as the fact that we *have* chosen.

And for everything else...

With all that, there’s a lot that’s not in our style guides. We try to focus on the things that have the greatest impact on the health of our codebase. There are absolutely best practices left unspecified by these documents, including many fundamental pieces of good engineering advice: don’t be clever, don’t fork the codebase, don’t reinvent the wheel, and so on. Documents like our style guides can’t serve to take a complete novice all the way to a master-level understanding of software engineering—there are some things we assume, and this is intentional.

Changing the Rules

Our style guides aren’t static. As with most things, given the passage of time, the landscape within which a style guide decision was made and the factors that guided a given ruling are likely to change. Sometimes, conditions change enough to warrant reevaluation. If a new language version is released, we might want to update our rules to allow or exclude new features and idioms. If a rule is causing engineers to invest effort to circumvent it, we might need to reexamine the benefits the rule was supposed to provide. If the tools that we use to enforce a rule become overly complex and burdensome to maintain, the rule itself might have decayed and need to be revisited. Noticing when a rule is ready for another look is an important part of the process that keeps our rule set relevant and up to date.

The decisions behind rules captured in our style guides are backed by evidence. When adding a rule, we spend time discussing and analyzing the relevant pros and cons as well as the potential consequences, trying to verify that a given change is appropriate for the scale at which Google operates. Most entries in Google’s style guides include these considerations, laying out the pros and cons that were weighed during the process and giving the reasoning for the final ruling. Ideally, we prioritize this detailed reasoning and include it with every rule.

Documenting the reasoning behind a given decision gives us the advantage of being able to recognize when things need to change. Given the passage of time and changing conditions, a good decision made previously might not be the best current one. With influencing factors clearly noted, we are able to identify when changes related to one or more of these factors warrant reevaluating the rule.

CASE STUDY: CAMELCASE NAMING

At Google, when we defined our initial style guidance for Python code, we chose to use CamelCase naming style instead of snake_case naming style for method names. Although the public Python style guide (PEP 8) and most of the Python community used snake_case naming, most of Google's Python usage at the time was for C++ developers using Python as a scripting layer on top of a C++ codebase. Many of the defined Python types were wrappers for corresponding C++ types, and because Google's C++ naming conventions follow CamelCase style, the cross-language consistency was seen as key.

Later, we reached a point at which we were building and supporting independent Python applications. The engineers most frequently using Python were Python engineers developing Python projects, not C++ engineers pulling together a quick script. We were causing a degree of awkwardness and readability problems for our Python engineers, requiring them to maintain one standard for our internal code but constantly adjust for another standard every time they referenced external code. We were also making it more difficult for new hires who came in with Python experience to adapt to our codebase norms.

As our Python projects grew, our code more frequently interacted with external Python projects. We were incorporating third-party Python libraries for some of our projects, leading to a mix within our codebase of our own CamelCase format with the externally preferred snake_case style. As we started to open source some of our Python projects, maintaining them in an external world where our conventions were nonconformist added both complexity on our part and wariness from a community that found our style surprising and somewhat weird.

Presented with these arguments, after discussing both the costs (losing consistency with other Google code, reeducation for Googlers used to our Python style) and benefits (gaining consistency with most other Python code, allowing what was already leaking in with third-party libraries), the style arbiters for the Python style guide decided to change the rule. With the restriction that it be applied as a file-wide choice, an exemption for existing code, and the latitude for projects to decide what is best for them, the Google Python style guide was updated to permit snake_case naming.

The Process

Recognizing that things will need to change, given the long lifetime and ability to scale that we are aiming for, we created a process for updating our rules. The process for changing our style guide is solution based. Proposals for style guide updates are framed with this view, identifying an existing problem and presenting the proposed change as a way to fix it. "Problems," in this process, are not hypothetical examples of things that could go wrong; problems are proven with patterns found in existing Google code. Given a demonstrated problem, because we have the detailed reasoning behind the existing style guide decision, we can reevaluate, checking whether a different conclusion now makes more sense.

The community of engineers writing code governed by the style guide are often best positioned to notice when a rule might need to be changed. Indeed, here at Google, most changes to our style guides begin with community discussion. Any engineer can ask questions or propose a change, usually by starting with the language-specific mailing lists dedicated to style guide discussions.

Proposals for style guide changes might come fully-formed, with specific, updated wording suggested, or might start as vague questions about the applicability of a given rule. Incoming ideas are discussed by the community, receiving feedback from other language users. Some proposals are rejected by community consensus, gauged to be unnecessary, too ambiguous, or not beneficial. Others receive positive feedback, gauged to have merit either as-is or with some suggested refinement. These proposals, the ones that make it through community review, are subject to final decision-making approval.

The Style Arbiters

At Google, for each language's style guide, final decisions and approvals are made by the style guide's owners—our style arbiters. For each programming language, a group of long-time language experts are the owners of the style guide and the designated decision makers. The style arbiters for a given language are often senior members of the language's library team and other long-time Googlers with relevant language experience.

The actual decision making for any style guide change is a discussion of the engineering trade-offs for the proposed modification. The arbiters make decisions within the context of the agreed-upon goals for which the style guide optimizes. Changes are not made according to personal preference; they're trade-off judgments. In fact, the C++ style arbiter group currently consists of four members. This might seem strange: having an odd number of committee members would prevent tied votes in case of a split decision. However, because of the nature of the decision making approach, where nothing is "because I think it should be this way" and everything is an evaluation of trade-off, decisions are made by consensus rather than by voting. The four-member group is happily functional as-is.

Exceptions

Yes, our rules are law, but yes, some rules warrant exceptions. Our rules are typically designed for the greater, general case. Sometimes, specific situations would benefit from an exemption to a particular rule. When such a scenario arises, the style arbiters are consulted to determine whether there is a valid case for granting a waiver to a particular rule.

Waivers are not granted lightly. In C++ code, if a macro API is introduced, the style guide mandates that it be named using a project-specific prefix. Because of the way C++ handles macros, treating them as members of the global namespace, all macros that are exported from header files must have globally unique names to prevent collisions. The style guide rule regarding macro naming does allow for arbiter-granted exemptions for some utility macros that are genuinely global. However, when the reason behind a waiver request asking to exclude a project-specific prefix comes down to preferences due to macro name length or project consistency, the waiver is rejected. The integrity of the codebase outweighs the consistency of the project here.

Exceptions are allowed for cases in which it is gauged to be more beneficial to permit the rule-breaking than to avoid it. The C++ style guide disallows implicit type conversions, including single-argument constructors. However, for types that are designed to transparently wrap other types, where the underlying data is still accurately and precisely represented, it's perfectly reasonable to allow implicit conversion. In such cases, waivers to the no-implicit-conversion rule are granted. Having such a clear case for valid exemptions might indicate that the rule in question needs to be clarified or amended. However, for this specific rule, enough waiver requests are received that appear to fit the valid case for exemption but in fact do not—either because the specific type in question is not actually a transparent wrapper type or because the type is a wrapper but is not actually needed—that keeping the rule in place as-is is still worthwhile.

Guidance

In addition to rules, we curate programming guidance in various forms, ranging from long, in-depth discussion of complex topics to short, pointed advice on best practices that we endorse.

Guidance represents the collected wisdom of our engineering experience, documenting the best practices that we've extracted from the lessons learned along the way. Guidance tends to focus on things that we've observed people frequently getting wrong or new things that are unfamiliar and therefore subject to confusion. If the rules are the "musts," our guidance is the "shoulds."

One example of a pool of guidance that we cultivate is a set of primers for some of the predominant languages that we use. While our style guides are prescriptive, ruling on which language features are allowed and which are disallowed, the primers are descriptive, explaining the features that the guides endorse. They are quite broad in their coverage, touching on nearly every topic that an engineer new to the language's use at Google would need to reference. They do not delve into every detail of a given topic, but they provide explanations and recommended use. When an engineer needs to figure out how to apply a feature that they want to use, the primers aim to serve as the go-to guiding reference.

A few years ago, we began publishing a series of C++ tips that offered a mix of general language advice and Google-specific tips. We cover hard things—object lifetime, copy and move semantics, argument-dependent lookup; new things—C++ 11 features as they were adopted in the codebase, preadopted C++17 types like `string_view`, `optional`, and `variant`; and things that needed a gentle nudge of correction—reminders not to use `using` directives, warnings to remember to look out for implicit `bool` conversions. The tips grow out of actual problems encountered, addressing real programming issues that are not covered by the style guides. Their advice, unlike the rules in the style guide, are not true canon; they are still in the category of advice rather than rule. However, given the way they grow from observed patterns rather than abstract ideals, their broad and direct applicability set them apart from most other advice as a sort of "canon of the common." Tips are narrowly focused and relatively short, each one no more than a few minutes' read. This "Tip of the Week" series has been extremely successful internally, with frequent citations during code reviews and technical discussions.[12]

Software engineers come in to a new project or codebase with knowledge of the programming language they are going to be using, but lacking the knowledge of how the programming language is used within Google. To bridge this gap, we maintain a series of "<Language>@Google 101" courses for each of the primary programming languages in use. These full-day courses focus on what makes development with that language different in our codebase. They cover the most frequently used libraries and idioms, in-house preferences, and custom tool usage. For a C++ engineer who has just become a Google C++ engineer, the course fills in the missing pieces that make them not just a good engineer, but a good Google codebase engineer.

In addition to teaching courses that aim to get someone completely unfamiliar with our setup up and running quickly, we also cultivate ready references for engineers deep in the codebase to find the information that could help them on the go. These references vary in form and span the languages that we use. Some of the useful references that we maintain internally include the following:

- Language-specific advice for the areas that are generally more difficult to get correct (such as concurrency and hashing).
- Detailed breakdowns of new features that are introduced with a language update and advice on how to use them within the codebase.
- Listings of key abstractions and data structures provided by our libraries. This keeps us from reinventing structures that already exist and provides a response to, "I need a thing, but I don't know what it's called in our libraries."

Applying the Rules

Rules, by their nature, lend greater value when they are enforceable. Rules can be enforced socially, through teaching and training, or technically, with tooling. We have various formal training courses at Google that cover many of the best practices that our rules require. We also invest resources in keeping our documentation up to date to ensure that reference material remains accurate and current. A key part of our overall training approach when it comes to awareness and understanding of our rules is the role that code reviews play. The readability process that we run here at Google—where engineers new to Google's development environment for a given language are mentored through code reviews—is, to a great extent, about cultivating the habits and patterns required by our style guides (see details on the readability process in Chapter 3). The process is an important piece of how we ensure that these practices are learned and applied across project boundaries.

Although some level of training is always necessary—engineers must, after all, learn the rules so that they can write code that follows them—when it comes to checking for compliance, rather than exclusively depending on engineer-based verification, we strongly prefer to automate enforcement with tooling.

Automated rule enforcement ensures that rules are not dropped or forgotten as time passes or as an organization scales up. New people join; they might not yet know all the rules. Rules change over time; even with good communication, not everyone will remember the current state of everything. Projects grow and add new features; rules that had previously not been relevant are suddenly applicable. An engineer checking for rule compliance depends on either memory or documentation, both of which can fail. As long as our tooling stays up to date, in sync with our rule changes, we know that our rules are being applied by all our engineers for all our projects.

Another advantage to automated enforcement is minimization of the variance in how a rule is interpreted and applied. When we write a script or use a tool to check for compliance, we validate all inputs against a single, unchanging definition of the rule. We aren't leaving interpretation up to each individual engineer. Human engineers view everything with a perspective colored by their biases. Unconscious or not, potentially subtle, and even possibly harmless, biases still change the way people view things. Leaving enforcement up to engineers is likely to see inconsistent interpretation and application of the rules, potentially with inconsistent expectations of accountability. The more that we delegate to the tools, the fewer entry points we leave for human biases to enter.

Tooling also makes enforcement scalable. As an organization grows, a single team of experts can write tools that the rest of the company can use. If the company doubles in size, the effort to enforce all rules across the entire organization doesn't double, it costs about the same as it did before.

Even with the advantages we get by incorporating tooling, it might not be possible to automate enforcement for all rules. Some technical rules explicitly call for human judgment. In the C++ style guide, for example: "Avoid complicated template metaprogramming." "Use `auto` to avoid type names that are noisy, obvious, or unimportant—cases where the type doesn't aid in clarity for the reader." "Composition is often more appropriate than inheritance." In the Java style guide: "There's no single correct recipe for how to [order the members and initializers of your class]; different classes may order their contents in different ways." "It is very rarely correct to do nothing in response to a caught exception." "It is extremely rare to override `Object.finalize`." For all of these rules, judgment is required and tooling can't (yet!) take that place.

Other rules are social rather than technical, and it is often unwise to solve social problems with a technical solution. For many of the rules that fall under this category, the details tend to be a bit less well defined and tooling would become complex and expensive. It's often better to leave enforcement of those rules to humans. For example, when it comes to the size of a given code change (i.e., the number of files affected and lines modified) we recommend that engineers favor smaller changes. Small changes are easier for engineers to review, so reviews tend to be faster and more thorough. They're also less likely to introduce bugs because it's easier to reason about the potential impact and effects of a smaller change. The definition of small, however, is somewhat nebulous. A change that propagates the identical one-line update across hundreds of files might actually be easy to review. By contrast, a smaller, 20-line change might introduce complex logic with side effects that are difficult to evaluate. We recognize that there are many different measurements of size, some of which may be subjective—particularly when taking the complexity of a change into account. This is why we do not have any tooling to autoreject a proposed change that exceeds an arbitrary line limit. Reviewers can (and do) push back if they judge a change to be too large. For this and similar rules, enforcement is up to the discretion of the engineers authoring and reviewing the code. When it comes to technical rules, however, whenever it is feasible, we favor technical enforcement.

Error Checkers

Many rules covering language usage can be enforced with static analysis tools. In fact, an informal survey of the C++ style guide by some of our C++ librarians in mid-2018 estimated that roughly 90% of its rules could be automatically verified. Error-checking tools take a set of rules or patterns and verify that a given code sample fully complies. Automated verification removes the burden of remembering all applicable rules from the code author. If an engineer only needs to look for violation warnings—many of which come with suggested fixes—surfaced during code review by an analyzer that has been tightly integrated into the development workflow, we minimize the effort that it takes to comply with the rules. When we began using tools to flag deprecated functions based on source tagging, surfacing both the warning and the suggested fix in-place, the problem of having new usages of deprecated APIs disappeared almost overnight. Keeping the cost of compliance down makes it more likely for engineers to happily follow through.

We use tools like clang-tidy (for C++) and Error Prone (for Java) to automate the process of enforcing rules. See Chapter 20 for an in-depth discussion of our approach.

The tools we use are designed and tailored to support the rules that we define. Most tools in support of rules are absolutes; everybody must comply with the rules, so everybody uses the tools that check them. Sometimes, when tools support best practices where there's a bit more flexibility in conforming to the conventions, there are opt-out mechanisms to allow projects to adjust for their needs.

Code Formatters

At Google, we generally use automated style checkers and formatters to enforce consistent formatting within our code. The question of line lengths has stopped being interesting.[13] Engineers just run the style checkers and keep moving forward. When formatting is done the same way every time, it becomes a non-issue during code review, eliminating the review cycles that are otherwise spent finding, flagging, and fixing minor style nits.

In managing the largest codebase ever, we've had the opportunity to observe the results of formatting done by humans versus formatting done by automated tooling. The robots are better on average than the humans by a significant amount. There are some places where domain expertise matters—formatting a matrix, for example, is something a human can usually do better than a general-purpose formatter. Failing that, formatting code with an automated style checker rarely goes wrong.

We enforce use of these formatters with presubmit checks: before code can be submitted, a service checks whether running the formatter on the code produces any diffs. If it does, the submit is rejected with instructions on how to run the formatter to fix the code. Most code at Google is subject to such a presubmit check. For our code, we use clang-format for C++; an in-house wrapper around yapf for Python; gofmt for Go; dartfmt for Dart; and buildifier for our `BUILD` files.

CASE STUDY: GOFMT

Sameer Ajmani

Google released the Go programming language as open source on November 10, 2009. Since then, Go has grown as a language for developing services, tools, cloud infrastructure, and open source software.[14]

We knew that we needed a standard format for Go code from day one. We also knew that it would be nearly impossible to retrofit a standard format after the open source release. So the initial Go release included gofmt, the standard formatting tool for Go.

MOTIVATIONS

Code reviews are a software engineering best practice, yet too much time was spent in review arguing over formatting. Although a standard format wouldn't be everyone's favorite, it would be good enough to eliminate this wasted time.[15]

By standardizing the format, we laid the foundation for tools that could automatically update Go code without creating spurious diffs: machine-edited code would be indistinguishable from human-edited code.[16]

For example, in the months leading up to Go 1.0 in 2012, the Go team used a tool called gofix to automatically update pre-1.0 Go code to the stable version of the language and libraries. Thanks to gofmt, the diffs gofix produced included only the important bits: changes to uses of the language and APIs. This allowed programmers to more easily review the changes and learn from the changes the tool made.

IMPACT

Go programmers expect that *all* Go code is formatted with gofmt. gofmt has no configuration knobs, and its behavior rarely changes. All major editors and IDEs use gofmt or emulate its behavior, so nearly all Go code in existence is formatted identically. At first, Go users complained about the enforced standard; now, users often cite gofmt as one of the many reasons they like Go. Even when reading unfamiliar Go code, the format is familiar.

Thousands of open source packages read and write Go code.[17] Because all editors and IDEs agree on the Go format, Go tools are portable and easily integrated into new developer environments and workflows via the command line.

RETROFITTING

In 2012, we decided to automatically format all `BUILD` files at Google using a new standard formatter: `buildifier`. `BUILD` files contain the rules for building Google's software with Blaze, Google's build system. A standard `BUILD` format would enable us to create tools that automatically edit `BUILD` files without disrupting their format, just as Go tools do with Go files.

It took six weeks for one engineer to get the reformatting of Google's 200,000 `BUILD` files accepted by the various code owners, during which more than a thousand new `BUILD` files were added each week. Google's nascent infrastructure for making large-scale changes greatly accelerated this effort. (See Chapter 22.)

Conclusion

For any organization, but especially for an organization as large as Google's engineering force, rules help us to manage complexity and build a maintainable codebase. A shared set of rules frames the engineering processes so that they can scale up and keep growing, keeping both the codebase and the organization sustainable for the long term.

TL;DRs

- Rules and guidance should aim to support resilience to time and scaling.
- Know the data so that rules can be adjusted.
- Not everything should be a rule.
- Consistency is key.
- Automate enforcement when possible.

1 Many of our style guides have external versions, which you can find at *https://google.github.io/styleguide*. We cite numerous examples from these guides within this chapter.

2 Tooling matters here. The measure for "too many" is not the raw number of rules in play, but how many an engineer needs to remember. For example, in the bad-old-days pre-clang-format, we needed to remember a ton of formatting rules. Those rules haven't gone away, but with our current tooling, the cost of adherence has fallen

dramatically. We've reached a point at which somebody could add an arbitrary number of formatting rules and nobody would care, because the tool just does it for you.

3 Credit to H. Wright for the real-world comparison, made at the point of having visited around 15 different Google offices.

4 "Chunking" is a cognitive process that groups pieces of information together into meaningful "chunks" rather than keeping note of them individually. Expert chess players, for example, think about configurations of pieces rather than the positions of the individuals.

5 See 4.2 Block indentation: +2 spaces, Spaces vs. Tabs, 4.4 Column limit:100 and Line Length.

6 Use of const, for example.

7 Style formatting for BUILD files implemented with Starlark is applied by buildifier. See *https://github.com/bazelbuild/buildtools*.

8 See Exceptions to Naming Rules. As an example, our open sourced Abseil libraries use snake_case naming for types intended to be replacements for standard types. See the types defined in *https://github.com/abseil/abseil-cpp/blob/master/absl/utility/utility.h*. These are C++11 implementation of C++14 standard types and therefore use the standard's favored snake_case style instead of Google's preferred CamelCase form.

9 See Generated code: mostly exempt.

10 See *https://google.github.io/styleguide/cppguide.html#Comments*, *http://google.github.io/styleguide/pyguide#38-comments-and-docstrings*, and *https://google.github.io/styleguide/javaguide.html#s7-javadoc*, where multiple languages define general comment rules.

11 Such discussions are really just bikeshedding, an illustration of Parkinson's law of triviality.

12 *https://abseil.io/tips* has a selection of some of our most popular tips.

13 When you consider that it takes at least two engineers to have the discussion and multiply that by the number of times this conversation is likely to happen within a collection of more than 30,000 engineers, it turns out that "how many characters" can become a very expensive question.

14 In December 2018, Go was the #4 language on GitHub as measured by pull requests.

15 Robert Griesemer’s 2015 talk, “The Cultural Evolution of gofmt,” provides details on the motivation, design, and impact of gofmt on Go and other languages.

16 Russ Cox explained in 2009 that gofmt was about automated changes: “So we have all the hard parts of a program manipulation tool just sitting waiting to be used. Agreeing to accept ‘gofmt style’ is the piece that makes it doable in a finite amount of code.”

17 The Go AST and format packages each have thousands of importers.