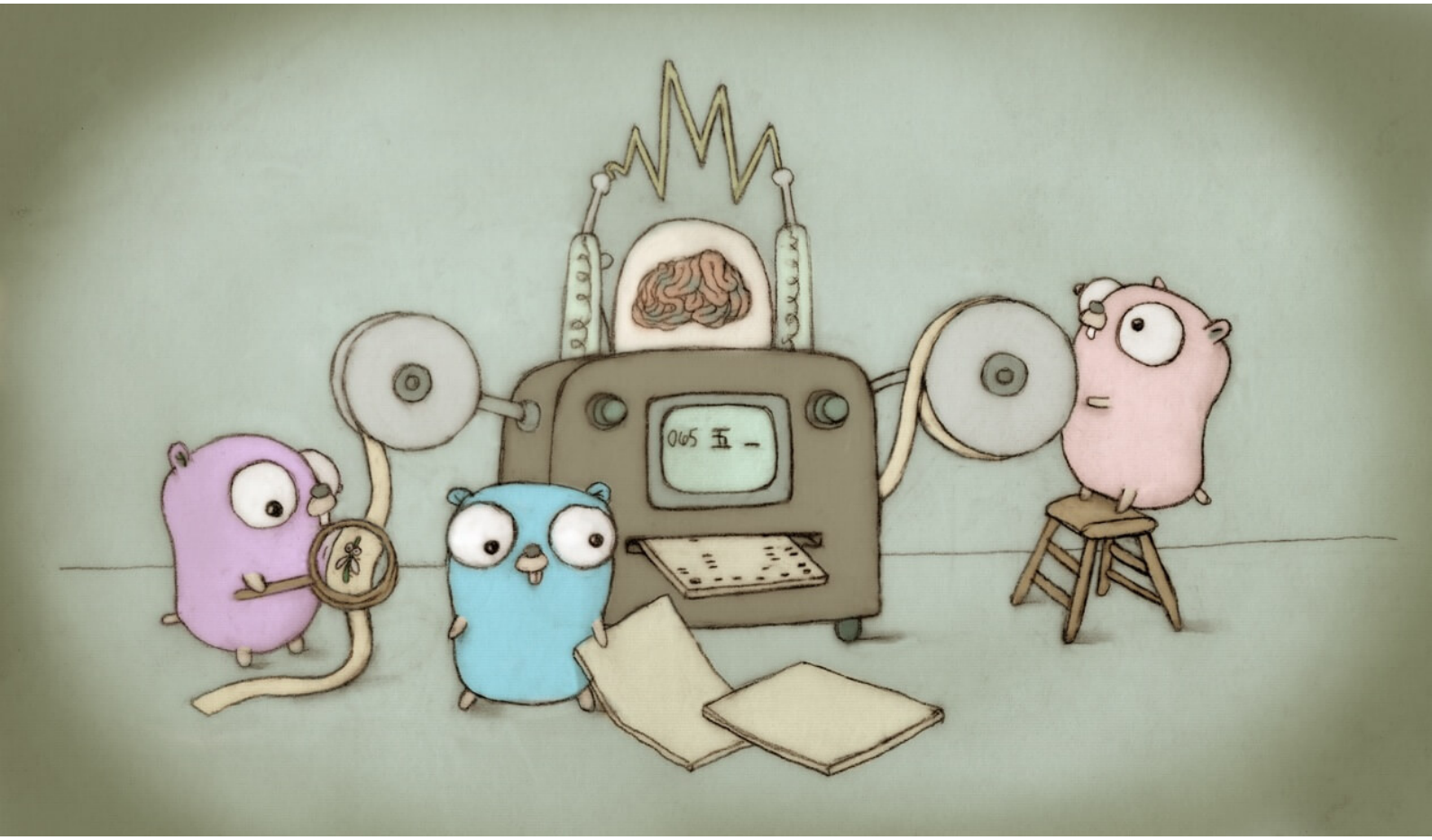

Coding from the Get Go

Robert S. Muhlestein (rwxrob)

Version v0.0.1, 2025-01-09 15:16:05: Under development

Table of Contents

Learn practical computer science and programming with Go as a first language

https://linktr.ee/rwxrob	Download PDF	Download EPUB

This content is under active development and based on original content from SKILSTAK Coding Arts developed in Python and JavaScript back in 2014. It will likely be several months before it is complete entirely. Many other great alternatives exist for learning computer science and Go programming if you prefer to wait. I make zero promises.

You can join me for livestream sessions if you don't mind a bit of informal exploration as we work through these projects together.

Copyright

Preface

Who should read this?

This book is for anyone who wants to **learn to code and learn enough computer science as is practical**. I assume you have never coded anything and might not know what the term *coding* even means. That's okay. This is where we begin. If you are looking for more, you're in the wrong place. Plenty of other resources exist for people who already know how do code and just want to learn Go programming.

> There are presently zero books or resources for absolute beginners who have never coded before that *begin* with the Go programming language. That's the reason I created this one. I find this interesting since Go is a far better language with which to demonstrate *modern* software development practices than traditional Java as is used in the US AP Computer Science program. Just Go's handling of implied *interfaces* alone is reason enough. Go contains all that is good from object-oriented programming without all that is bad from class-oriented programming.

Why the Go language?

The language we learn is **Go**. My artist wife is obsessed with traditional fables, which might explain why I like to call Go the *Goldilocks* of modern languages. (Yes, I know, it's pretty cheesy—but you'll remember it, right?) Go was explicitly designed and developed to do real work, originally at Google and now for the world of "cloud-native" computing upon which all enterprise depends.

The Goldilocks comparison comes from the fact that Go is low-level and "serious" enough to withstand the weight of huge enterprise software development codebases, while remaining light and agile enough to feel like a softly typed, "approachable" language such as Python. Go fills this space between them perfectly, which is probably why it is the fastest-growing language among Systems Development Engineers modernizing monstrous codebases of technical debt from other, less agile, and more error-prone languages.

What do I need?

You really don't need a lot to get going with this book. It's designed to have the least amount of setup and dependency:

- A decent computer that can run reasonable performant games
- A reliable Internet connection for installing and saving
- An email account that you control
- The ability to type 30 words per minute from home row
- Administrator permission to allow the installation of software
- Enough snacks and drinks to keep you from getting hangry

Take your coding to the next level by first reading *Terminal Velocity*.

Strongly consider an AI assistant

In my experience ChatGPT is an absolute necessity for anyone wanting to truly take their learning to the next level. I'm not getting paid to say that (but I would happily accept money for saying so). You really should consider subscribing if you are serious about adopting an autodidactic lifestyle. ChatGPT is specifically designed for general learning and creativity and runs on anything with full contextual history. Mine has helped me keep my conversational Russian and French skills up, planned bike trips, helped me code the mundane stuff, reminded me of things I forgot about, offered up random ideas, and so much more.

Claude, another AI assistant, is better for `mods` terminal API integration and for coding.

Still skeptical? I know I was.

Here's the thing. On demand learning exponentially increases when an AI is involved. Nothing breaks through frustration and loneliness better when taking on learning challenges with a supportive AI companion—even when a helpful human mentor is also available.

We are quickly approaching a time when the digital divide will no longer be just between those who have computers and Internet access and those who do not, but between those who have learned to leverage a personal AI assistant loaded with contextual history and those who have not. Don't get left behind.

We are already seeing this difference around us every day. I'm remarkably faster at the same job now with AI doing the same thing I've been doing for four years, and that's not even just the coding part. Querying an AI is exponentially better for research than a Google search, provided I verify the results, as always.

How long will it take?

Anticipate about 55 hours of time with a mentor going over the material and know that you will need another 110 on your own studying to master just these concepts and mini-projects. That's 165 total hours (give or take) equivalent to a 5 credit hour (11 hours per week) course if crammed over 15 weeks.

Remember also that these mini-projects are just to introduce you to the many possibilities. None of these projects stand on their own as worthy of an addition to a portfolio, but they give you a great head start to make your own. You will still need at least another 165 working on real projects—your own or others—after you have learned these before you will be ready to do any of this on your own professionally.

Learning to code isn't easy, it's just worth it.

How do I use this book?

Join a learning community and/or find a mentor, then do these *mnemonic mini-projects* in order to learn everything the Go programming language has to offer as well as the core concepts of computer science. Each project gradually adds new concepts and skills while repeating the stuff you have already done. This scaffold learning approach ensures you retain what you have learned by repetition rather than just moving on to the next thing.

You really should not attempt to learn all of this on your own without support from someone who has already mastered these specific concepts and skills. It doesn't have to be a "teacher" per se. Anyone who has successfully completed them can serve as a mentor. This means you may find such people in a learning community focused on the same things. You are always welcome to join mine if you cannot find another.

Associate name with concepts

These challenges are designed to be memorable so that you associate concepts with whatever silly mnemonic devices were associated with the challenge. Each project is named for the primary artifact (usually a runnable program) that you will create by the time it is done. This short name should be used whenever talking about these projects with others to help everyone know exactly which project you are talking about.

Start writing your own book

Each challenge is designed for you to research on your own and then solve and document in your own way. The idea is that *you* are writing your own codebook repo while you work through these summarizing the key concepts and code in your own way. You'll remember everything a lot better this way and can continue to expand your own codebook as you add your own challenges and learning.

If you use AsciiDoc to write it, you can even publish your own booklet for reference or create your own EPUB or PDF you can carry with you on your phone of tablet.

Learn the ~~hard~~ right way

The "hard way" is really the *only* way to truly learn something in a sustainable way. It just means you try to do a thing and figure out how to learn to do that thing on your own before depending on someone who knows. The scaffold approach means you have someone there to spot you and support you if you really screw up so you don't break your neck but by doing what you can on your own first—knowing your limitations without fearing them—leads you to greater confidence in your own ability to learn and assess your own learning without help. You grow to have no dependence on a book, teacher, guru, or app. Developing this autodidactic, just-in-time learning independence is *mandatory* for any tech professional or knowledge worker today. AI just supplements this approach for those who approach learning the *right* way. It's "hard" because you have to do your own research rather than just slurping it off the page.

Run screaming from "banker style education" where the teacher thinks you should be injected with learning by sitting there listening to them instead of asking questions and interactively coming up with solutions. This concept aligns with Freire's critique of traditional education models, often referred to as the "banking" model of education. (Freire, *Pedagogy of the Oppressed*, p. 71).

I've spent a lot of time separating the list of requirements and the associated tasks from the code and explanation of how to do it. This means you can read those first without spoiling the solution—if you want. For each project, consider following these steps:

1. Grok the requirements

When you first open a project, read the description and the list of requirements and think hard about what you would do without ever reading the rest. If you don't understand a term, look it up with your AI assistant or the Internet. Practice doing the research just as you would on the job.

Copy and paste the requirements as-is into your own notes so you can check them off as you complete them. Just add an `x` between the square brackets. This is supported in both AsciiDoc and GitHub flavored Markdown and is a good habit to get into because these are regularly used in pull requests and other developer operations.

It is *more* important to be fast at looking things up—doing research—than it ever will be to memorize algorithms. Memorization is good to for sure, but fast research is at least as important. This is why I love moving at *Terminal Velocity* because I can do it all from the terminal. I can find thing on the Internet in a fraction of the time that I could with a GUI browser and then open up a GUI browser if needed with one keystroke, no images, no cookies, no tracking, just lightning fast browsing and search history that is saved in my command line history as well.

2. Imagine a solution

This immediately gets your creative neurons working in your brain. You want that. Coding is *mostly* a creative endeavor. This gets you in the habit of approach problems creatively in the real world the same way.

It is *more* important to learn how to approach problems and deconstruct them than it is learn to code. People hiring you want *solution providers* not programmers. There's a big difference—especially now that AI can already program better than most programmers alive today.

3. Write down your plan

After you have really imagined how you would go about fulfilling the requirements *write you plan down* in your own notes. Take a break and come back. Really digest your strategy and think it through. Consider it mental training because that is exactly what it is.

4. Commit some code

Only after you have written down at least a sentence or two of what it is that you are going to do—as specifically as you like—start coding what you can from your memory and notes so far. This is important because a lot of each new project repeats stuff from previous projects on purpose to reinforce the learning.

You might end up with the solution without continuing. But go ahead and proceed.

Even if you don't think your code is ready to share, go ahead and commit it so that you have a backup and something to share.

5. Get some peer review

Once your code is committed you can allow others to look at it. Don't be ashamed. Make sure you get an experienced mentor to look at it as well.

> Here is where AI really shines. You can submit your code and the challenge with the description to any AI and it will critique it for you in addition or in the absence of a more experienced mentor.

6. Study sample solution

The solution samples in this book are exactly that: samples. There might be several different ways to accomplish the requirements or perform the tasks. Each project solution is just one possibility. This is why getting peer review is so important as well. Others will bring contributions to your idea that they might never have thought of.

7. Revise and repeat

Make changes and comments about things you need for the next time. Then commit them. Remember, these projects are designed to be repeated like martial arts so you can refresh the skills over time. In fact, now that you have committed (saved) your latest solution, why not delete everything and see if you can do it entirely from scratch without looking at your notes. Consider repeating this process until you can.

8. Share your learning

The act of helping another learn what you just learned is the best way to assess your own learning while reinforcing it at the same time. This is why a math lab or hacker space full of others learning the same thing is always such a huge success, because people are in the room to do dynamic peer review while working on the same stuff. (Incidentally, this is the flawed justification for open cubicle workspaces.)

Don't forget to write down what you learned in your notes in a way that is as if you were writing a lesson for another. You could even polish up your lesson a bit and publish it for others who might respond to your method of learning.

> The problem with *conformity* in education is that people are

> not standardized to begin with. (Robinson, *Creative Schools*)
>
> Learning is dramatically different for every person. Sir Ken Robinson regularly shared these distinct differences in learning style. Never become depressed because something was hard for you to learn while someone else learned it right away. While aptitude and intelligence are very real, so is the *method* of learning that is hard-coded into your specific brain. Half the battle is figuring out how *you* learn and developing learning strategies based on that.

Dear parents and pedagogues

Are you a parent or a teacher? If so, thank you for being *awesome*! It takes a lot to bring kids into this world and even more to dedicate your life to helping kids that aren't even your own to find their way and learn something even if they don't want to. Deep respect. But let's get real. You're reading this right now wondering if I'm going to corrupt your children.

The answer is yes. Yes I am. In fact, my goal is to fully corrupt them much the same as Socrates. If I am successful your children will become *better* that the status quo, they'll think critically, ask uncomfortable questions, and use their new powers and tools to ethically disrupt this broken world transforming it into something better. They'll become the most annoying people you've ever known—in the best way possible.

Why me? Because getting people to discover this potential and realize it through their own learning is my super power. I've been mentored by the best and I'm *really* good at it. Since the first time my scouts begged "Mr. Rob, teach us to code. Mr. Rob, teach us to hack. Mr. Rob, teach us Russian." I have been bound and determined to make this a reality. In fact, I started SKILSTAK Coding Arts in 2013 with my own retirement money to address the *real* needs of the tech industry and those who might consider it as a career, raising a generation of highly skilled knowledge warriors.

Let me tell you about some of these amazing people:

- One of them quit his grocery clerk job to write code for a solar energy company and was paired with essentially his own intern at 16.
- Another couldn't get a job with a psychology degree so he learned this stuff and got a starting salary over $100,000.
- A few people got coveted invitations to special FBI cybersecurity camps.
- One drove 40 minutes both ways just to attend our sessions for *four years* and went on to create an entire coding club at her rural school.
- Some got cybersecurity degrees from accredited colleges and invites to work with professional hacking crews.

It's been a blast being with these people. My favorite memories are all the times we hacked companies with permission and helped them patch their security, or shut down hackers for panicked parents whose businesses had been compromised.

The point is, these success stories don't have to be about someone else. They can be about someone *you know and love*. Terminal tech skills really do change lives and I'm here to help in whatever way

I can.

Prepare to code

I've noticed that coding in Go is something that people with all backgrounds enjoy. This is probably because Go can be compiled on anything for anything else there is zero limitation as to the computer you use for development. I regularly use Go programs I've written on macOS or Linux on my Windows machine as well.

Install a Go editor

The editor you use really doesn't matter—for now. Some prefer a graphic user interface and others a terminal. They are all fine. The original Go development team used Vim for everything and that is my personal preference.

- Notepad - yes, you can code with Notepad if you want
- Nano - ultra simple terminal editor, already installed
- Vim - traditional and ubiquitous, already installed
- Neovim - powerful but requires installation
- Emacs - preferred terminal editor by some
- VS Code - favorite light-weight graphic editor
- Goland - professional proprietary graphic editor

Consider a container

If you know about containers this might be the fastest way to get going—especially if you use the official Go container image.

Install the Go compiler

Mac: `brew install golang`
Linux: `sudo apt get golang`
Windows: `winget install --id golang.go`

Upgrade to latest Go version

```
go get go
```

Install Go binary tools

`:GoInstallBinaries`, VS Code extension

Create a GitHub account

You must be 13 years old or older to use GitHub legally. Check with your instructor

> for alternatives. Or, just skip the `git` and `gh` stuff and just remember to copy your files wherever you want to back them up.

Install git and gh

- `brew install git gh`
- `sudo apt install git gh`
- `winget install --id git.git`, `winget install --id github.cli`

Authenticate to GitHub from CLI

`gh auth login`

Create a GitHub repo to contain your Go coding work and notes.

```
mkdir -p ~/Repos/github.com/youraccount
cd ~/Repos/github.com/youraccount
gh repo create learn-golang --private --clone
git branch -m main
git push -u origin main
```

Projects you'll remember

Before you continue, make sure you have read the *Learn the hard way* description of how to proceed.

This part of the book contains nothing but projects that build on one another to help you master everything you can do with Go and Go's most substantial standard libraries—in a fun way so you remember it better. Each project is separated into different parts allowing you to hold back on more help than you might need:

Section	Purpose
Introduction	The intro has no heading. It's just a fun introduction to the activity with some things that will hopefully help you remember it better. Remember, the sillier the more memorable.
Requirements	This section contains a checklist that you can cut and paste into your own notes if you are reading this in digital form. It is designed to read like something you might receive from a customer or see on a job description. This should make searching for them easier later on—especially if you add them to your own code repo or knowledge base.
Bonus	Contains further requirements that are above and beyond what is expected in this project. In a group setting, this gives people more to do who finish early. It also gives you more to attempt when you delete it and try again.
Sample	The sample solution has this title because it is not the *only* solution. There are almost always variations that will work. This is one reason that automating the checks of such work is a completely futile use of time. You are your own best assessor when given the tools and contacts to do so.
Ideas	This is a list of random ideas for things you might want to create on your own that have more demonstrable value in a GitHub portfolio, for example. Your creation of the code for the idea is 100% original and owned by you so you can use it to build trust with potential employers or project leaders.
Tags	Contains tags and keywords related to this project. This is useful when looking for all projects that might relate to the same content or task.

Sometimes I get the sense that some people reading this part of the book regret that it isn't a recipe book of exact applications they would create on the job. I do love those sorts of books, but all of them are completely obsolete now that AI is here to stay. Any tech professional or knowledge worker *without* an AI assistant at this point is practically guilty of malfeasance.

No, this book *deliberately* creates silly, memorable learning activities so that they stick better in your brain and are repeatable. Most of them can easily provide the building blocks to "serious" applications that you and/or your AI could create to show off your skills. But keep this in mind: ***employers no longer trust GitHub repos as much as before as an indication of skill***. This was true at one point,

> but no more. AI has permanently destroyed that assumption.

hello

Requirements

Sample

- Add code to git: `git add .`
- Commit code to git: `git commit -m 'some comment'`
- Push code to GitHub: `git push`
- Check code on GitHub: `gh repo view --web`

module, package, main, func, print, println, fmt.Print, fmt.Println, fmt.Printf, block, parameter, argument, variable, stdout, stderr, go-run, %v, %b, string, number, boolean, type, semicolon-insertion, go-mod-init, import, double-quote, backtick-quote, single-quote, module, package, file, go.mod, pkg, cmd, README.adoc, comment, markdown, asciidoc, godoc, adoc

```
package main

import "fmt"

func decrement(x int) int      { return x - 1 }
func increment(x int) int      { return x + 1 }
func halve(x int) int          { return x / 2 }
func double(x int) int         { return x * 2 }
func greet(name string) string { return "Hello " + name + "." }

func main() {
    fmt.Println(greet("Doris"))
    fmt.Println(greet("Rob"))
}
```

- Package is the current directory name
- All `.go` files in directory as as if they were all in a single file
- Unit test files have `_test.go` (but more on that later)
- By convention, `main.go` contains `func main()` (also `package main`)
- All packages are either *commands* or *libraries*
- Put library packages into `pkg`
- Put command packages into `cmd`
- A *module* contains one or more packages
- Keep `go.mod` (which identifies the top of the *module*) at the root

- Regularly use `go mod tidy` to remove unused module dependencies
- Commit `go.mod` and `go.sum` in your GitHub repos
- Make sure command packages have good names
- Build binaries in same directory with `go build`
- Build binary with specific name with `go build -o hello`
- Install into `GOBIN` (in `PATH`) with `go install`
- Use `type hello` to see what is being run
- Use `hash -r` to reset to latest install
- Note that `go install` required full path to file `go install ./cmd/hello`
- Go is an *explicitly* compiled language (like C)
- Comment a single line with `//`
- Comment multiple lines with `/* */`
- Create same-named file with `_test` for unit tests
- Add `// Output:` comment followed by expected output as `//` comment
- Run `go test` to check example-based unit test
- Learn basic Markdown
- Consider learning AsciiDoc

world

Under development ...

bincount

Under development ...

arrrgs

Under development ...

greet

In development...

- Block - anything within curly brackets
- Subroutine - any collection of steps or instructions within a block
- Procedure - specifically a collection of instructions in a subroutine
- Function - internal calculation with a specific output return value (like math)
- Scope - how visible a thing is to other things

- Caller - the thing that called the other thing
- Paradigm - how to look at the world
- Operator - performs some action on one or more operands
- Operand - involved in the the action of an operator

---

ansi

Under development ...

Requirements

```
- [ ] Create high-level function library: `ansi`
- [ ] Create callable command: `ansi`
- [ ] If one argument beginning with zero assume 256
- [ ] If one argument matching CSS regex assume CSS
- [ ] If three arguments assume RGB
- [ ] If one argument `reset` or `x` print reset
- [ ] Must not take any dashed options
```

Bonus

```
- [ ] Implement as Bonzai branch
- [ ] Add named CSS colors
- [ ] Add named CSS color completer
- [ ] Submit CSS color completer to Bonzai project
- [ ] Submit CSS color library to Bonzai project
- [ ] Add custom completer that uses CSS completer
```

Sample

In development...

Ideas

In development...

Tags

`color, ANSI, escapes`

nyan

Under development ...

waffles

Under development …

eightball

Under development …

diceroll

Under development …

lights

Under development …

pre

Under development …

toemoji

Under development …

bridge

Under development …

fizzbuzz

The fizzbuzz project is one of the most common filter challenges give to any new computer programmer being interviewed. Apparently, 95% of candidates cannot do it even after completing a computer science degree.

Under development …

fields

Under development …

simon

Under development …

quiz

Under development …

madforms

Under development …

vimtutor

Under development …

story

Under development …

pokeapi

Under development …

serve

Under development …

wschatserve

Under development …

wschatclient

Under development …

calc

Under development …

resume

Under development …

clip

Under development …

completer

Under development …

persister

Under development …

Planning what's next

Hard to believe you made it through all those projects. Congratulations! The few who do inevitably go through a cathartic emotional period right about now. I have had a front-row seat for hundreds going through this experience. It's almost like completing your first marathon. But its what happens next that really matters most. What are you going to *do* with your new skills?

Get a job

In development…

Join a project

In development…

Find a meetup

In development…

Change your stars

In development…

Save the world

In development…

Gotchas

This part of the book contains all the many ways I've been burned over more than a decade of Go programming as a SysDE and how to avoid being burned yourself.

Input string argument comes last

Go's `text/template` package is Rob Pike's pet package (he has a great video talking about it) and it shows. It is the single most powerful, extensible template framework I've ever seen. No wonder it is the basis of every Helm chart, Hugo, and so much more but here is a big gotcha that you might not know about when coding normal everyday string transformation utility functions. Don't be me, do these right from the beginning.

Problem

You might think that the following is absolutely fine. I certainly did:

```
package to

import (
    "fmt"
    "strings"
)

// WRONG! DO NOT DO THIS. This unnecessarily prevents a perfectly good
// function from being used within text/template pipelines.
func Indented(in string, indent int) string {
    var buf string
    for _, line := range Lines(in) {
        buf += fmt.Sprintln(strings.Repeat(" ", indent) + line)
    }
    return buf
}
```

The problem is subtle, but important. As the comment points out, the order of the arguments unnecessarily prevents this function from being used in `text/template` pipelines because the string is passed as the first argument and not the last.

> A pipeline may be "chained" by separating a sequence of commands with pipeline characters '|'. In a chained pipeline, the result of each command is passed as the last argument of the following command. The output of the final command in the pipeline is the value of the pipeline.

Solution

Here's the better way along with a comment explaining it for the next poor soul who might not

know:

```
package to

import (
    "fmt"
    "strings"
)

// Indented returns a string with each line indented by the specified
// number of spaces. Carriage returns are stripped (if found) as
// a side-effect. Per [pkg/text/template] rules, the string input
// argument is last so that this function can be used as-is within
// template pipe constructs:
//
//  {{ .Some | indent 4 }}
func Indented(indent int, in string) string {
    var buf string
    for _, line := range Lines(in) {
        buf += fmt.Sprintln(strings.Repeat(" ", indent) + line)
    }
    return buf
}
```

String is best last anyway

By making the string value always last as a general rule you also allow for much clearer testing because the value arguments are not obfuscated behind a long, inline string of text:

```
it := to.Indented(4,`
line one
line two
line three
`)
fmt.Print(it)
```

Moment

Even though I knew about `text/template` when I wrote my `github.com/rwxrob/bonzai/to` package I did not really intend for it to be used in a template—until I did. While creating my templates for Bonzai command help I went to do an indentation so that I didn't have to force it on everyone. That's the first time I really focused on pipelines. Even though I had read about the very obscure rule about passing the string last, I forgot and had to choose to make a bunch of template `FuncMap` commands that corrected the argument order or deprecate my entire `to` package and fix them all. You can probably guess which one I did.

References

- Coplien, James. *The Dehumanisation of Agile and Objects*. Presented at GOTO Berlin 2017. YouTube video, 1:00:25. Published November 2017. https://youtu.be/ZrBQmIDdls4
- Freire, Paulo. *Pedagogy of the Oppressed*. Translated by Myra Bergman Ramos. 30th Anniversary Edition. Continuum, 2000.
- Muhlestein, Rob. *Awesome Go*. GitHub repository. https://github.com/rwxrob/awesome-go
- Muhlestein, Rob. *rwxrob*. Linktree. Accessed July 2024. https://linktr.ee/rwxrob.
- Muhlestein, Rob. *Terminal Velocity*. GitHub repository. Accessed July 2024. https://github.com/rwxrob/terminal-velocity.
- Robinson, Ken, and Lou Aronica. *Creative Schools: The Grassroots Revolution That's Transforming Education*. Penguin Books, 2016.
- Robinson, Ken. *Out of Our Minds: Learning to Be Creative*. 3rd ed., Capstone, 2017.
- Robinson, Ken. *Do Schools Kill Creativity?* Filmed at TED February 2006, posted June 2006. YouTube video, 20:03. https://youtu.be/iG9CE55wbtY
- Silva, Elena, Taylor White, and Thomas Toch. *The Carnegie Unit: A Century-Old Standard in a Changing Education Landscape*. Carnegie Foundation for the Advancement of Teaching, 2015. https://www.carnegiefoundation.org/wp-content/uploads/2015/01/Carnegie_Unit_Report.pdf.