Go for Rubyists

Introduction

It was 2012, and I was exploring a Linux forum. I was really hoping for a *trollful* topic about something like *Which name is correct? Linux or GNU/Linux?* or something like *Why macOS sucks? a free software advocate's perspective*. But instead, I found a **really useful** topic. One of members, just posted a PDF, written by him about *Go programming language*. It made me happy of course. I was really excited to see a brand new programming language which is created by Ken Thompson and Rob Pike and has the support from Google. Unfortunately, back then I didn't have time to read books or documents and I even haven't been this good in English. So I decided when I got better at English and also had the enough free time, I will learn that programming language. I've got much better in reading, writing, speaking and listening English, I had free times, but I didn't learn Go. So you may ask what did happen to me? Well, I learned Ruby instead.

In 2014, I have become the student of computer hardware engineering. I also found that I love backend programming. In an IRC channel, one of my friends sent me this link and told me *take a look at this!*. Then my story began. I learned ruby, I learned Sinatra (a micro framework with ruby) and I also learned Rails. But now, 8 years after those days, I'm gonna learn Go as a ruby programmer. And I wanna share the whole process with you, my friend. You're reading this because I have decided to learn Go step-by-step and just tell you what I've learned. In this introduction, I will tell you something that you **must** know before starting reading the book. So I hope the book is useful enough for you.

Should I be a rubyist to understand this book?

It'd be better if you have a little bit of experience in ruby. But if you have a background in similar languages, I mean interpreted scripting languages such as *python* or *javascript* or even *perl*, you'll understand what I am going to say in the book.

Should I take a look at *Go* documentations before starting to read this book?

Of course you should. But, please do not fo too deep. Otherwise I won't have so much to offer. Just kidding, it's always good to read the official documents. In this book, I am going to give you my story of a being a *Go developer* who was a rubyist for a long time. You know how do I feel? Like a lawyer in a court who created a communist dictatorship in the Latin America. Fortunately, we're not going to form any dictatorships here. We're going to become great programmers!

Let's start

Yes, let's start. *Table of Contents* is this file and you will be able to find other chapters/sections of the book there. After I finish the book, I'll create a *PDF* version available and it will be freely available to you. So do not worry :)

Table of Contents

- ~~Introduction~~
- Chapter 1 : A Journey to the land of Gophers
  - Smoking kills, so does the trend!
  - Go in a first glance
  - Advantages of using Go
  - What's next?
- Chapter 2 : Setting up the environment
  - A deep look at RVM
  - Go installation
  - Go environment
  - Hello World
  - What's next?
- Chapter 3 : Flashback to Ruby
  - Let's cut our Rubies!
  - What's next?
- Chapter 4 : FizzBuzz
  - Ruby
  - Before we write the code in Go
  - Finally we'll Fizz the Buzz
  - What's next?
- Chapter 5 : Functions
  - A simple function : Area of a rectangle
  - Data Types in Go
  - Factorial Function
  - What's next?

Chapter 1 : A Journey to the land of Gophers

When Go became a somehow famous language among programmers, I asked myself *Why should I learn it?*. I still ask this, but for now I have some strong answers and the strongest one is that *it helps me find a job easier*. Let's get more realistic about the language and its properties.

Go, has everything you need at the same time. Primarily it was created to become an alternative to C (but I doubt the whole idea, as most of system programs are still being written in C or C++, and also Rust has been born in pretty much the same time as Go.) but shortly, Go became a replacement for *every single backend language*. Isn't it beautiful?

Personally, looked for a way to write a whole new operating system in Go. In this wiki I found some *bare bones* and to be honest I even couldn't compile the codes (in my defense though, I was only 16 when I was doing that and I didn't know S*it about Go). After that, one of my friends just made this which is a tmux-like tool with Go and I wanted to learn Go, again! Years passed, I just learned how Go works and how does its syntax look like, but never tried to code *seriously* in that language. I never tried to make a *microservice* using Go, never tried to make a simple shell or something like that.

Everything was fine, until I watched this video. I don't know are you like me or not, but if you're like me, you obviously like to make everything yourself. I always wanted to understand how docker, lxc or any other containerization system works. I personally thought *there's no better way than implementing one myself!* and of course this idea always worked. For example, I made an operating system kernel when I was 21, just to understand how it works. Or even earlier in my teenage years, I was almost 16, I made an Ubuntu based linux distribution just to understand how Linux distributions are made and how they work. When I watched the video, I realized how cool is Docker, and also, how cool is Go! It was almost a year ago. And then, I had first sparks in my head for seriously learning Go!

Fine, I told you a story just in four paragraphs, and I'm not going to continue, because this is not a work of fiction. This book is going to help you guys migrate from Ruby to Go. So, the rest of the chapter, although it's going to be storytelling, I try to keep it as technical as I can.

Smoking kills, so does the trend!

It is really good to know that smoking kills. For the love of God, everytime you buy a packet of cigarettes, it's printed in a bold font on the packet! But do you know what also kills? the trend. Let me give you an example. K-POP music (which is short for Korean pop) existed for a long time, but in the past few years, it became *a trend*. So, when you criticize the music or the taste of the fans, you're somehow a Nazi to eyes of the others.

So why do I tell you this? Let's just take a deep look at the community I'm coming from. I come from Iran. In Iran, for a long long time, *PHP* was the best! Why? probably because most of the internet was written in PHP back then, facebook used it and there was a lot of resources to learn the language. So, why not? I don't know it was 2004 or 2005, but Blogfa became a trendy Iranian blogging service. It was written in *C#* and *ASP*. This was enough for people to just make a new trend. For years, you couldn't find a job without knowing *C#*. It wasn't good, at least to me.

Around 2011, Python became trendy here. I know python existed long before. But in 2011, I witnessed some organizations, companies and startups here used Python to create their apps, backends, etc. At least it was something good. For finding a job, you could learn python and django and then you could print money.

After a few years, when *Go* became the trend of the world, most of those companies who used *PHP*, did a refactor to *Go*. Now, a new age has been started. You really *need* to learn Go when this happens. Why? because Go is a new language. It has a lot of potentials to become the next *PHP* and this means we'll have a web written in Go in a few years.

Although what I told you in the previous section, wasn't the whole story and it's not the most accurate story about trends, but it gives you the idea about how trends can be killer, too!

Go in a first glance

Although I didn't want to put any code here, but hey, why not? This is how *Hello World* looks like in Go :

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

How does it look like? To me, it looks like *C* or *C++*. And this is one of the reasons I love the language. Let's write a more complex code. Why not using some functions? Fine, I'll write a code that takes width and height of a rectangle and then gives you the area :

```go
package main

import (
    "fmt"
    "math"
)

func rectangle_area(width, height float64) float64 {
    return width * height
}

func main() {
    width := 10
    height := 22

    fmt.Printf("Area is %d\n", rectangle_area(width, height))
}
```

So, this means this language is a bit harder than Ruby. Is it a bad thing? So if you don't know the fact that codes written in Go can be 1500 times faster than ruby, YES! But if you know the fact, No. You spend more time coding, but you spend less time running your codes. And does it mean Ruby is bad? HELL NO! Ruby is awesome, but you know there's something more awesome.

Advantages of using Go

So, I think I told you everything about Go and its advantages, right? But in this section I just list all of them :

- Modern language, with the strength of *C* and *C++*.
- Supported by Google (I know if you're a *FLOSS* advocate this will piss you off, but do you know having support from a big corporate means what?)
- Learning resources
- Models they introduced in the language for concurrent programming, etc.
- Packages! (Fine, I think I'm out of words for this section, because having a package manager is pretty much the definition of a *modern* language.)

And if you still look for more to know, I highly suggest reading this article.

What's next?

In the next chapter, we're going to install Go, and take a look at the differences between installation of Go and Ruby, and we also try to set up a development environment. It won't be a long chapter, but I recommend reading that to not miss my jokes!

Chapter 2 : Setting up the environment

So now we're going to set our environment up. There are so many different ways to set up a Ruby environment, right? I personally use RVM but there are more ways such as `rbenv` or even installing ruby from repositories (in case you use Linux) or installing from `brew` (If you have a Mac) and something like that. In this chapter, I just refer to `rvm` and do not use any other Ruby installation system. The Only reason for this is that I just want to show you differences and show you how to *not* get confused while working with Go.

A deep look at RVM

Fine, RVM stands for *Ruby Version Manager*. It's somehow like `npm` (if you used javascript, you know what I mean) but not completely. It's a tool that installs and manages different versions of Ruby on your computer. For example, if you want Ruby version 2.6.6, you just do this :

```
rvm install 2.6.6
```

And if you want to use that version, you just do this :

```
rvm use 2.6.6
```

For some reasons, I mostly use Ruby 2.6.5, and I want it to be my default. So what should I do? it's easy :

```
rvm use 2.6.5 --default
```

Isn't it beautiful? By the way I believe this way of installing a language isn't cool. But do you know why we have to keep using this way - specially when we use ruby - and not get over it? The answer is most of apps we created, Use different versions of ruby and ruby *gems* (I think gems are more like `npm` by the way.) and we need to test them out with different versions on our own machine, then we have to install *the right version* on the host machine. Do we need the same thing in Go? not really. Why? You'll find out later in the book yourself :)

Go installation

There are a lot of ways to install Go on your machine. If you're on Windows, website offers an installation file (`msi` or `exe`) which installs everything for you. If you're on macOS, that's the same, but this time the whole thing is just made for *Darwin* kernel. And if you use Linux, this is how you should install the damn thing. Isn't it too easy then? It is. And this is why I kinda like Go, even more than Ruby.

After installation, just run this :

```
go version
```

and you'll have this response :

```
go version go1.15.6 darwin/amd64
```

And remember, I use macOS, so that `darwin` thing may differ on your machine.

Go environment

Fine, finally we arrived here. Now, I just run this in my terminal :

```
go env
```

What does it do? It just lists a bunch of environment variables usef by Go. Something like this :

```
GO111MODULE=""
GOARCH="amd64"
GOBIN=""
GOCACHE="/Users/prp-e/Library/Caches/go-build"
GOENV="/Users/prp-e/Library/Application Support/go/env"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOINSECURE=""
GOMODCACHE="/Users/prp-e/go/pkg/mod"
GONOPROXY=""
GONOSUMDB=""
GOOS="darwin"
GOPATH="/Users/prp-e/go"
GOPRIVATE=""
GOPROXY="https://proxy.golang.org,direct"
GOROOT="/usr/local/go"
GOSUMDB="sum.golang.org"
GOTMPDIR=""
GOTOOLDIR="/usr/local/go/pkg/tool/darwin_amd64"
GCCGO="gccgo"
AR="ar"
CC="clang"
CXX="clang++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-arguments
```

Let's see what we've got here. The most important one is `GOPATH` for now. As you can see it tells us `/Users/prp-e/go` is my `GOPATH`. What does it mean? It means everything we've done is here. So, let's take a look at my own `GOPATH`. It looks like this :

```
/Users/prp-e/go
└── src
    └── github.com
        └── prp-e
```

Inside my `GOPATH`, I have a `src` directory, and as you know it's short for *source*. And I also have `github.com` directory. Which means I host my projects on github. Depending on the services you use, this can be different. GitLab? your own instance of gitea? doesn't matter, just create a folder for each one!

> Something you should know - specailly if you're a newbie programmer - is that most of companies, startups or organizations may have their own *internal* git server as well. So you may need a directory for that as well.

Inside that github folder, I have a `prp-e` directory. That's my github username. So if I have another user or organization, it's recommended to have another folder for that.

Great! We have our development environment now. Why not creating a *Hello World* project?

Hello World

Let's do a flashback to Ruby. In Ruby, *Hello World* was this simple :

```
puts "Hello, World"
```

but here, it's a bit more coding work, but the result is really cool. The code we have here looks like this :

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

but, let's do it more classy. I did these :

```
cd ~/go
```

which navigates to my `GOPATH`, then :

```
cd src/github.com/prp-e/
mkdir go-for-rubyists-projects
```

and yes, there will be a repository for projects we're making here. After that, I did these :

```
cd go-for-rubyists-projects
mkdir 01_hello
cd 01_hello
touch main.go
```

And copied the code I gave you earlier to `main.go` file. After that, I just ran this :

```
go run main.go
```

and this was the result :

```
 ~/go/src/github.com/prp-e/go-for-rubyists-projects/01_hello/ [master] go
Hello World!
```

Yes, we ran our very first code written in Go!

What's next?

In the next chapter, we just try to take a look at our knowledge of Ruby programming language. We just do a quick review on Ruby and in future chapters of this book, we're going to re-learn everything we knew from Ruby, but this time in Go!

Chapter 3 : Flashback to Ruby

In the last section of the previous chapter, I've mentioned that in this chapter, we just review our knowledge of Ruby programming. I don't think it takes so long, specially if you're a rubyist or you have ideas about Ruby. You also may use this chapter as a *quick tutorial* of Ruby, specially if you have experience in programming and your primary language is not ruby.

As I mentioned before, Ruby is a little bit too easy to understand, so this chapter obviously is a bunch of long and boring codes. But, in the next chapter we'll see all of them in Go! So, why not just get bored before we see the true magic?

Let's cut our Rubies!

In this section, I am going to show you how ruby codes work. Ruby is a scripting language and developers even have provided an online interpreter which lets you run these codes. To be honest, we won't do so fancy ruby here. That `repl.it` thing is enough to just see the idea.

Enough talk, let's write the code.

Hello World

```
puts "Hello, World!"
```

Variable declaration

```
1  a = 1
2  b = "Hello"
```

Conditionals

```
a = 5
b = 10

if a > b
    puts "#{a} is bigger than #{b}"
else
    puts "#{b} is bigger than #{a}"
end
```

Arrays and Hashes

```
array = [1, 2, 3, 4]
hash = {:a => "a", :b => "b" }
```

Loops

Fine, here I have to explain different scenarios. All of them pretty much do the same thing, except some of them are directly from the entities like arrays, strings or everything that has a similar behavior.

While loops

```
counter = 1

while counter < 10 do
    puts counter
    counter += 1
```

For loops

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
2
3  for i in a do
4      puts i
5  end
```

But as I mentioned, we can iterate through the whole thing much easier. Pretty much like this :

Iteration through the array

```
1  a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2
3  a.each do |i|
4      puts i
5  end
```

And this concept applies to hashes and strings as well.

Functions and classes

For functions and classes, we'll have a separate chapter. I prefer to explain functions and classes of Ruby in the chapter we're coding Go. Why? because Ruby is a *mostly object oriented* language. But Go is not even object oriented. So I need you to feel the difference.

What's next?

I think after doing a quick flashback to ruby, it's time to write a simple code, which is also a commonly asked programming interview question. In the next chapter, we'll be writing the famous *FizzBuzz* first in Ruby, then in Go. It's a very very good way to learn a whole new language, specially when you have a background in another language.

Chapter 4 : FizzBuzz

FizzBuzz, is a commonly asked programming question. It doesn't matter if it's a simple programming class exam or a coding interview in a big corporate. It's simple, but needs a lot of focus and attention. In this chapter, I write it in ruby, then we try to re-write it in Go. So, why should we do this, when we don't know shit about Go? because we're going to learn different parts of a Go application in a *real world* application and not useless examples. If you want a lot of useless examples, you can just search over the web, and you'll witness how many times people repeated pretty much the same thing, over and over again!

Fine, instead of being angry, let's do our job. First, we need to know what is FizzBuzz (I assume you know, but I also explain it here.) After that, We will see how we can implement the algorithm in Ruby, and then we re-write that in Go.

What's FizzBuzz?

FizzBuzz is a gimmicky algorithm and determines if a number is divisble by 3, 5 or 15 in a very fun way. We just give the algorithm a bunch of numbers (For example every number from 1 to 100) and then if it's divisible by 3, it outputs *Fizz*. If it's divisible by 5 it outputs *Buzz*. Finally, if it's divisible by 15, it outputs *FizzBuzz*. In most of coding interviews, you're asked to code this. Why? because although it's simple, you can make a world of mistakes while writing the code.

Ruby

```ruby
counter = 1

while counter < 100 do
    if counter % 15 == 0
        puts "FizzBuzz"
    elsif counter % 3 == 0
        puts "Fizz"
    elsif counter % 5 == 0
```

```
        puts "Buzz"
    else
        puts counter
    end
end
```

Well, this is our ruby code. But we should understand Go is not that easy. But it's still easy enough to have fun with! Before writing our version of *FizzBuzz*, we need to learn alternatives in Go. I really didn't want to make another disaster like previous chapter, so I kept if fun and cool in this chapter. Well, let's find out what we need to learn about Go!

Before we write the code in Go

General code structure

Remember this :

```
package main

import "fmt"

func main(){
    fmt.Println("Hello, World")
}
```

So, what's this? Imagine we have a file called `main.go` . The *pacakge* keyword here determines the name of the file, and what it does in the whole application. The next thing is `import "fmt"` . If you did code in ruby, you know we use `require` to import a library. This is the same thing. I also think if you're from a python backgorund, you're more familiar, because it's the same keyword in python. `"fmt"` is *formatter*. If you want to print something, you need that. And of course, it does more than printing but for now, we just need the print function.

The part we said `func main(){}`, we declare *main function*. This *main function* is inherited from *C* and *C++*, which were the main inspirations for Go. Of course, we need to mind that the one and only Ken Thompson had a great roll in the creation of Go. So, it means we have to deal with a lot of C inheritance here.

> Although I've always been a Ruby developer, but I also used to work with AVR microcontrollers and Arduino boards. I am really familiar with *C* and I highly recommend to learn a bit of *C*. It will make learning programming much easier.

Fine, when we pretty much know what everything does here, we need to just know what `fmt.Println("Hello World")` does. It simply prints out the input text with a `\n` appended at the ent.

Please, remember this *general code structure*, because it's what we're using all the time in this book and pretty much our career as a Go programmer!

Conditions and statements

Again, these structures are pretty much similar to *C*. Let's make a simple program that checks three things (I hate this example) :

1. A is greater than B
2. A is equal to B
3. A is less than B

This is written like this in Go:

```
package main

import "fmt"

func main(){
    a := 10
    b := 5

    if a > b {
        fmt.Printf("%d is greater than %d", a, b)
    } else if a == b {
        fmt.Printf("%d is equal to %d", a, b)
```

```
    } else {
        fmt.Printf("%d is less than %d", a, b)
    }
}
```

Are we cool now? Let's take a deeper look. First, I used `fmt.Printf` because I'm more familiar with it's structure (remember my hardware programming background?) and of course there are better ways to print out integers. But the focus here is on `if {}` statements. The structure is :

```
if CONDITION {
    STATEMENR
}
```

and other stuff like `else if` and `else` are pretty much the same. Remember, Go is more like *C* than Ruby, so it looks (trust me, it just looks like this) a bit more difficult to understand.

Now only loops remain...

Loops

For some reason, there's only `for` loop in Go. I think it is a wise move. Why? Fine. Let's look at ruby. We have `while` and `for` . These are commonly used structures in Ruby. We also have `until` which is some how like reverse of `while` (and although it looks cool to have something like that, I never could understand it!) and each iterable entity also has `each` method which helps us iterate. Enough. It's too much syntax! Let's just learn one strong enough to destroy a planet.

`for` loops in Go mostly look like the loops in *C*, but they're a bit more flexible. So here, I'm showing the simplest to the most advanced form.

```
func main(){
    i := 1
```

```
    for i < 10 {
        fmt.Println(i)
        i++
    }
}
```

This is somehow similar to `while`. We also can write the same concept like this (Which looks like *C* for loops):

```
func main() {
    for i := 1; i < 10; i++ {
        fmt.Println(i)
    }
}
```

Now, we almost learned everything we needed, so why not making the whole *FizzBuzz* thing?!

Finally we'll Fizz the Buzz!

With the knowledge we have now, we can write *FizzBuzz* in Go as well! So I created `02_fizzbuzz` project as well and let's open `main.go` file and write these inside the file:

```
package main

import "fmt"

func main() {
    for i:= 1; i < 10; i++ {
        if i % 15 == 0 {
            fmt.Println("FizzBuzz")
        } else if i % 3 == 0 {
            fmt.Println("Fizz")
        } else if i % 5 == 0 {
            fmt.Println("Buzz")
        } else {
            println(i)
```

```
15            }
16        }
17
18        fmt.Println("Done :)")
19    }
```

I hope you enjoyed this coding experience. It's amazing, isn't it? It's all for now :)

What's next?

In the next chapter, we'll take a look at functions and how we can write different functions in Go and we'll do a comparison with Ruby. For now, I think you've learned pretty much everything an *intermediate* Go programmer must know.

Chapter 5 : Functions

In the Chapter 3 I told you that I planned to discuss functions in a different chapter, right? This is it. First of all, I have to say, writing a function in a language like *C* or *Go* is a bit difficult. Why? Because you need to difine pretty much everything you want as the input and the output. You'll find out what I say later. In the other hand, this way of writing and declaring functions has something people like to call *type* safety. I have examples of when Ruby can go mad when you do not provide type safety.

Does this mean Ruby is bad? of course not. Ruby is the definition of the word love for me, but let's be honest, something like type safety is much easier to achieve in Go.

In this chapter, we'll write two simple functions both in Ruby and Go. Then we run the code. In this chapter, you really experience writing codes which are similar to the real world applications!

A simple function : Area of a rectangle

I know this is a simple mathematical operation, but we just need to see how a function works, right? And I tried to keep it as simple as possible. Here, we need an algorithm. The algorithm will become a function and we can add our functions to our codes. Enough said, let's see what algorithm we have.

The whole algorithm we have is `f(x, y) = x * y`. I know this is not how scientific notation of functions work, but let's take it easy. Here, we assume x is our height and y is our width. So the result is x * y. The function in Ruby will be :

```ruby
def rectangle_area(x, y)
    return x * y
end
```

And this is very simple, isn't it? In Go we have a more complex but *more accurate* and *type safe* structure for writing our functions. The same function in Go will be:

```go
func rectangleArea(x, y float64) float64 {
    return x * y
}
```

Fine, so I think you need explanations for this code. The keyword `func` acts like `def` in Ruby. It's short for *function* and when you see that, you notice this is a function. Then, we have to name our function (and I called it `rectangleArea`, because Go suggests camel case naming.) and after that, we need to declare *arguments* in parantheses. As we don't always have integers as widths and heights, we need to declare those variables as *float64*. And here our *type safety* comes. After parantheses, we have *float64* again. What's that? That is our output type! We need that in order to get our output correctly.

Now, we know how it works, let's see how our whole code looks like in Go :

```go
package main

import "fmt"

func rectangleArea(x, y float64) float64 {
    return x * y
}

func main() {
    x, y := 5.0, 10.0
    fmt.Println(rectangleArea(x, y))
}
```

I think we all know how it works, so we just need to move forward and write the second function. But before that, we need to learn something.

Data Types in Go

We all know even in languages like Python, Ruby, PHP and pretty much every scripting language, we have *data types*. The simple way to find out the type of data is using methods

provided. For example in Python, if we define `a = 2` and then run `type(a)`, we'll find it's an integer, although we don't tell the interpreter we have an integer. In Ruby we just run `a.class` and so on.

In Go, although we can declare variables without type as easy as typing `a := 5` we still need to define types. As these types are literally a lot, I highly suggest taking a look at this link. We'll need this link in future, specially when we try to do web stuff using Go!

Factorial function

Remember the scene from the movie Doctor Strange, when he returned over and over to kick Dormammu's ass? We're going to do the same thing here. But remember that recursive functions are not usually the wisest choice you can make. They easily can ruin your memory. I included this one here, because I like the factorial function. It's simple and it still can show us the whole concept of recursive functions.

A recursive function, is a function that calls itself. One of the most famous recursive functions is *Fibonacci*. And it's up to you to learn and code that function in Ruby and Go. Here, we're just going to write codes for Factorial.

Factorial algorithm is very easy. We assume 0 and 1 both have the factorial of 1. You know when you multiply something to 1, result is pretty much the same. So 1 has no effects on the results. Now, we have the algorithm :

```
n! = n * factorial(n-1)
```

This cycle continues until we reach 1 or 0 (depending on how you code the algorithm) and the final number, is the `n!` we've been looking for.

The code in Ruby is :

```
1  def factorial(n)
```

```
    if n == 0
        return 1
    else
        return n * factorial(n)
    end
end
```

It's simple and it works. Now, it's time to write the whole thing in Go. This time, I only write the function. Printing and other stuff is up to you.

```
func factorial(n int) int {
    if n == 0 {
        return 1
    } else {
        return n * factorial(n -1)
    }
}
```

It got much simpler when you keep practicing coding in Go. Functions are very important here, because we're dealing with a language which is not *object oriented* and it's mostly functional. In the future, we'll learn how these functions can be usueful to us.

What's next?

To be honest, it took more than three hours of thinking to find out what should I write here. Go is a very big language (in terms of features, frameworks, community, etc.) and we covered most of it in these five chapters. I am almost out of content for the rest of the book.

Do not worry, in the next chapter, we'll take a look at `slice` and `map` data structures in Go. Why? I think most used structures in Ruby are arrays and hash tables. So, why not learning the equivalent in Go? Stay tuned for the next chapter. In the mean time, go ahead and write code for different algorithms with the knowledge you have.