Introduction to Programming

Learn programming algorithms and data structures using Golang

Youri Ackx

Introduction to Programming

Introduction to Programming with Go

Learn data structures and algorithms with Go

Youri Ackx

4 Apr. 2023 (preview)

Learn programming algorithms and data structures using Golang

Youri Ackx

This book is about computer programming and algorithms, without the formal academic approach. Learn about recursion, complexity, data structures, and solve classical computer science problems like the Towers of Hanoi, the Eight Queens and Conway's Game Of Life.

Contents

1 Introduction

1.1 It's about programming

This book is first about **programming, algorithms and data structures**. Of course, Go will be our reference language, and for sure, you will learn Go along the way. But the techniques presented in this book will also be transferable to a large extent to other programming languages like Python, Java or C.

We will cover one single programming paradigm: **imperative programming**. Other paradigms such as object oriented programming are not in the scope of this book.

We will take the time to understand what is going on under the hood. A tutorial will usually give you a recipe to solve a specific problem, without necessarily discuss the underlying algorithm. For instance, you can be given instructions on how to sort a list using a library or a built-in function, but it will probably not discuss how sorting a list actually works.

This book covers the equivalent of one semester or more of first year computer science class. It is **designed for beginners** who want to acquire a good grasp on algorithms and data structures. **Teachers** on the other hand can use it as a classroom support and leverage its numerous exercises, while focusing on teaching the material.

Go may seem a peculiar choice to learn programming. We will discuss the reasons of this choice and its merits, with background information on the language.

1.2 Approach

We will take a **non-formal, intuitive and practical approach** to programming, heavily based on exercises of increasing complexity, adding the minimum amount of theory necessary as we progress to solve them.

For **beginners**, the first part lays the foundation of programming. Variables, loops, conditional statements and functions found in most languages will be presented. At this stage, you will be able to solve simple problems, like checking if a paper fits in an enveloppe.

We will continue with more advanced data structures. Slices and maps are the bread and butter of Go programs. We will read data from files and manipulate them. From there, you can already go a long way and write useful programs. At that point, implementing games like Blackjack or Hangman is in reach.

We will extend these data structures to form lists, linked lists, queues, stack and trees to mention the most common ones. At this **intermediate** level, we will talk about recursion, backtracking, space and time complexity, finite automatons. We will solve both fun and classical academic problems like the eight queens problem, the towers of Hanoi, or implement the classical Conway's game of life;, and more.

1.3 Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

1.4 Vocabulary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

1.5 Abstraction

In order to write programs, we need **abstractions**.

Consider a hard disk drive (HDD). It is an enclosure containing rapidly rotating disks or platters, each of which is coated with magnetic material in order to store data. Arms with magnetic heads move above the platters in order to read (and write) data that can represent an image, a song or a poem, or more generally anything we call a *file*.

Figure 1: A hard drive needs an abstraction

You could read the poem stored on the hard drive and display it on screen by sending commands to move the heads above the right platter on the hard drive, communicating with the CPU directly. But the task would be incredibly difficult to achieve. Every program you or someone else writes would have to repeat the same operations, for every single data you would want to read, and for every possible disk geometries. This would be at best a tedious task. Above all, it would be pointless.

Maybe the hard disk drive can be seen as *cyclinders* and *sectors* that form some logical organization. Which it does in reality. This is a first level of **abstraction**. But this is still not the abstraction we are looking for — unless we are writing specific parts of an operating system, or a HDD controller.

For sure, we have a more casual purpose. Say we want to read the poem from the disk with as little knowledge about the underlying hardware as possible. We are interessted in retrieving the poem, not about the low-level hardware details.

Now consider the following program fragment:

```
poem, err := ioutil.ReadFile("poem.txt")
defer poem.Close()
if e != nil {
    panic(e)
}
fmt.Print(string(poem))
```

There are several things that require an explanation. What is `err` or `panic` for instance. All in due time. For now, using a high-level programming language, we have *abstracted* the process of retrieving the information on the disk magnetic surface. In fact, the abstraction is layered: the Go compiler provides you with a first abstraction from the operating system, which in turn abstracts you for the processor, the memory and the hard disk. This is what we were looking for: a mean to **express ideas and concepts while hiding the underlying complexity and details**. It is still nice to know how a hard drive or an operating system work though. But suffice to say it is much more efficient and confortable to rely on the work done by others to access a storage medium, and focus on our program.

> *Note*: Nowadays hard disk drives (HDD) are getting replaced by solid state drives (SDD) that have no mechaninal moving parts. Moving parts or not, you still want to be abstracted from the bare medium. By virtue of abstraction, our program above will still work, no matter the actual underlying physical media, HDD, SSD or other.

1.6 First program

To get acquainted with a programming language, it is customary to write a program that displays a friendly message. For the first chapters, you do not necessarily need to have Go installed on your computer. You can simply enter an execute your program in the Go Playground:

```
package main

import "fmt"

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

This simple program already raises several questions. What is a `package`, why is there an `import` with `"fmt"` in it? One thing at the time. For now, remember that `Println` prints a line on the screen. The term “print” and its variations are found in many programming languages, more often than “display” and comes from the ancient times where the primary interface with the computer was a printer rather than a screen. `main` is the program entry point. It is where the first instruction will be executed.

This program is found by default when you open the the playground, so you don’t even have to copy-paste it. Mark this page, as this program will be the skeleton for many exercises we will solve in the next chapters.

1.7 A paper and a pen

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

1.8 About Go

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

2 Basic data types

2.1 Definition

A data type is a classification that specifies which type of value a data has and what type of operations can be applied to it.

In our first program, `"Hello, world!"` data type is a string of characters, or simply a *string*. `42` is a number.

There are many ways to categorize and group data types. For instance, *numbers* can be further subdived in *integers* (whole numbers like `1`, `42`) and *floating points* or *floats* (like `1.234`). But you may encounter the more mathematical term *Real* to designate them. It depends on the context and… the language.

Every programming languages has **primitives**. They are the data types from which all other data types are constructed. To make matters more confusing, some primitive data types may be considered *derived* primitive data types.

The Go language specification[1] should settle the debate.

In this chapter, we shall have a look at some basic datatypes.

2.2 Bits and bytes

It is a well-known fact that computers only understand "ones and zeros". The most basic unit of information that the computer in store is called a **bit**. A bit can hold two possible values: `1` or `0`. Actually, one and zero are merely conventions. Instead we could use `on` and `off`, or `true` and `false`. We'll stick to the long standing convention though.

Dealing with `1` or `0` exclusively would be extermely cumbersome, even for a computer processor. Bits are usually[2] grouped by **8** to form a **byte**.

[1]https://go.dev/ref/spec#Types

[2]It is actually more complex than that. Abate can technically be of any size. But the common definition is to use eight bits.

1	1	1	0	0	1	0	1
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

It gives us 2^8 or 256 possible combinations.

What does the above byte represents? Typically, an positive number (unsigned integer). To convert it to a decimal value, multiply each bit by 2^k, with k equal to the bit position. In our example:

1	1	1	0	0	1	0	1

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

$$1.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0$$

$$= 1.128 + 1.64 + 1.32 + 0.16 + 0.8 + 1.4 + 0.2 + 1.1$$

$$= 128 + 64 + 32 + 4 + 1$$

$$= 229$$

2.3 Numeric

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

2.4 Strings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

2.5 Overflow

If you attempt to create a numeric value that is outside of the range that can be represented with a given number of digits, an **overflow** will occur. Typically, the result will "wrap around" the maximum.

```
func add(a, b uint8) uint8 {
    return a + b
}

func main() {
    fmt.Println(add(250, 10))
}
```

```
4
```

(From now on, we shall omit the obvious `package main` and `import "fmt"` from our examples.)

In this example, we are trying to add 10 to 250. The resulting 260 exceeds the capacity of `uint8`. When computing, Go reaches 255 and wraps back to 0, hence a result of 4. This can be shown with the following binary addition. It works like a the decimal addition you know, with carry, only with 2 digits. Bits are grouped by 4 for readability. As you can see, the result has a 9th bit on the left that won't fit in `uint8`. It gets dropped, resutling in 00000100 or 4.

```
    --uint8--
    1111 1010 (250)
+   0000 1010 (5)
= 1 0000 0100 (260)
    0000 0100 (4)
```

It is up to the program author to make provisions to avoid such errors. For instance, by modifying the return type so that it can always hold the result.

> Exercise: modify the program to return a larger integer, and test it.

That condition will be most likely unanticipated, leading to an incorrect or undefined behavior of your program. This can have dire consequences.

On June 4th, 1996, an Ariane 5 rocket bursted into flames 39 seconds after liftoff[3]. The explosion was caused by a **buffer overflow** when a program tried to stuff a 64-bit number into a 16-bit space. Sounds familiar? It is estimated that the explosion resulted in a loss of US$ 370m. Fortunatelly there was no crew on board.

Figure 2: Ariane explosion was caused by buffer overflow

Sometimes, the result is less harmful. In 2014, a popular video clip caused the Youtube view counter to overflow[4], forcing Google to switch from 32 to 64 bits integer. A number of views greater than 2 billions had not been anticipated.

The consequences were far greater in the former case than in the latter. Depending on your context, you may need to be extremelly wary, or you can afford to remain relatively casual while designing and testing your program.

[3] https://en.wikipedia.org/wiki/Cluster_(spacecraft)#Launch_failure

[4] https://arstechnica.com/information-technology/2014/12/gangnam-style-overflows-int_max-forces-youtube-to-go-64-bit/

We can reason about a the safety of a datatype (or of a data structure) based on its purpose.

An application dealing with "small" amounts may be confortable with `int32`. The numbers supported by `uint64` seem to go even beyond a banker's wildest dreams, although today sky-high numbers in the financial world cast a doubt on the safest assumptions. Imagine you manipulate cents rather than units in order to avoid dealing with decimal numbers. You would multiply every number by 100. Or even by 10000 to safely manipulate 4 decimals. And suppose you do computations on a currency like japanese yen currently at 1 JPY for 0,008 EUR, leading to further multiply values by about 1000. Say you have to deal with consolidated results in the billions of euros, converted to yens, counting in cents.

How safe is your initial assumption now?

To safelhy manipulate large numbers, as Go has dedicated implementations for big numbers[5]. But they come at the cost of convenience, readability and performance. That is why they are not your go-to solution in all contexts.

2.6 Abstraction vs low level

Why not simply manipulate "integers"? Why "floating point arithmetic" and different integers? After all, we mentionned the importance to abstract ourselves from the underlyting platform. Some languages only expose "integers" and "decimals", but it comes with a substantial performance cost. Go integers types are closer to the hardware architecture. That is a trade-off the languages authors decided to do, based on the intent and purpose of the language, where high performance is key.

From a teaching perspective, this design choice gets a bit into our way as it clutters the explanations, at least at the begining. On the bright side, as far as learning goes, you are exposed to technical underlying details that would otherwise remain hidden, and you can already get a grasp at them.

[5]https://golang.org/pkg/math/big/

3 Programming blocks

3.1 Variables and constants

A **variable** is a storage place in the computer memory used by a program. It has a name that identifies it. For instance, we could perform the following sequence of instructions in pseudo-code:

```
h <- 'hello'
a <- 1
b <- 2
score <- a + b + 4
```

Which would result in:

- The variable `h` contains the string of characters `'hello'`
- The variable `a` contains the value `1`
- The variable `b` contains the value `2`
- The variable `score` is the sum of `a`, `b` and `4`, that is `7`.

Or visually:

$$\begin{array}{cccc} h & \boxed{\text{'hello'}} & b & \boxed{2} \\ a & \boxed{1} & b & \boxed{7} \end{array}$$

The equivalent in Go is to declare the variables, and to assign them a value.

```go
func main() {
    h := "hello"
    a := 1
    b := 2
    score := a + b + 4
}
```

Notice the `:=` operator to assign a value to a variable.

Assigning variables does not do much. If you were to enter this code snippet in the program main method, the program would execute but nothing would be displayed. Go ahead and try it in the Go Playground.

Play with this small program and try to add `"hello"` and `2`. You will get an error message, as one can of course only perform additions on numbers.

Of course, we can print the results. We can modify our program so that it displays the sum of three numbers.

```
fmt.Println(sum)
```

Our modified program would unsurprisingly produce the following output:

```
7.5
```

As its name implies, a variable can *vary*. Or more precisely, the value it holds can vary.

```
a := 7
a = a + 2
```

What is going on?

- After the first instruction, `a` holds the value `7`
- After the second instruction, `a` holds the value `a+2`, that is `7+2`, which evaluates to `9`.

As opposed to variables, we can define **constants** whose values cannot change once they are set.

```
const a := 5
a = 6   // won't compile, already set
```

3.2 Conditional statements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

3.3 Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

3.4 Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

3.5 Arithmetic operators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

3.6 Expressions

3.6.1 Examples

An intuitive way to start programming is to have a look at expressions. `"Hello, world!"` we used in our first program is an expression. `42` is another one. `3+4` a third. The basic arithmetic operations can be performed with `+`, `-`, `\*` and `/`. You can perform complex operations, with parenthesis if you need. The `*` and `/` operators take precedence over `+` and `-`.

We can go further. All of the following are expressions:

```
sumOfSquare(2, 3)
sumOfSquare(2 * 4, 3 * 2)
sumOfSquare(sumOfSquare(2, 3), 3 * 2)
```

```
7 + 2
42
```

3.6.2 Definitions

Expressions in Go are formally defined in the language specification. In the **expression** `7+2`, `+` is the **operand**. `7` and `2` are its **operators**. The operator's **arity** is 2.

Expression:

> An expression specifies the computation of a value by applying operators and functions to operands.

Operand:

> Operands denote the elementary values in an expression.

Arity:

> The number of arguments or operands that a function takes.

Binary operator:

> An operator that applies to two operands. Its **artity** is 2.

Unary operator:

> An operator that applies to one operand. The expression `-7` has an operator `-` (negate) with one operand `7`. Its **arity** is 1. In this case, the operator `-` (of arity 1) is not to be confused with the minus mathematical operator (of arity 2).

3.6.3 Evaluation

Let's take a non trivial yet simple case and evaluate `3+4`. The expression is evaluated to `7` after the following steps:

```
3 + 4
7
```

Our `sumOfSquare` with two integers will be evaluated as follows:

```
sumOfSquare(2, 3)
(2 * 2) + (3 * 3)
4 + (3 * 3)
4 + 9
13
```

NOTE: Notice the use of parenthesis. Although mathematically not mandatory in this case, as multiplation takes precendence over addition, they denote a group to be evaluated regardless of arithmetic precedence considerations.

In order to evaluate `sumOfSquare(2 * 4, 3 * 2)`, `2*4` and `3*2` must be evaluated first. Then we fall back on the case of `sumOfSquare` with two integers as parameters we already know.

```
sumOfSquare(2 * 4, 3 * 2)
sumOfSquare(8, 3 * 2)
sumOfSquare(8, 6)
(8 * 8) + (6 * 6)
64 + 36
100
```

Expression evaluation will come in handy at a later stage, when we examine the complexity of an algorithm.

3.6.4 Boolean expressions

A Boolean expression is **a logical statement that is either true or false**.

The expression `3 < 5` is evaluated as `true` while `2 < 0` is evaluated as `false`. You can assign a boolean expression to a variable, for instance `a = 3 < 5`. In that case, `a` will be evaluated to `true`.

You can test two values for equality with `==` as in `3 == 3` which of course is `true`. The double equal is used to avoid confusion with a variable assignment, as in `a = 3`. Which can be confusing in itself. Actually, most languages use `==` to perform equality comparisons, for historical reasons. If you want to check that two expressions are different, as in "≠", you would use `!=`, like so: `3 != 5` which evaluates to `true` as 3 is not equal to 5.

Math	Go	Meaning
>	`>`	Greater than
≥	`>=`	Greater or equal
<	`<`	Less than
≤	`<=`	Less than or equal
=	`==`	Equal
≠	`!=`	Not equal

Although it would be pretty useless, for illustration purpose, we can write a function `isGreaterThan` that checks if its first argument is greater than its second.

```go
func isGreaterThan(a int, b int) bool {
  return a > b
}

a := 3
```

With for instance the following evaluation steps:

```
isGreaterThan(5, 3)
5 > 3
true
```

4 Lists, Arrays and Slices

4.1 Arrays

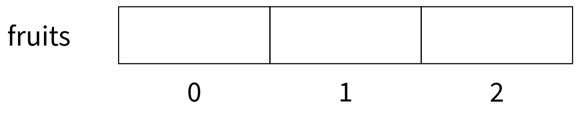
Suppose you want to manipulate several strings of characters. For instance, a shopping list, where each string is an item. So far we have used variables to contain numeric (integer or floating point) and string values. They were single values. You can declare several variables to hold several values, but this will quickly become cumbersome if the number of items on our list is not known in advance, or if there are a large amount of items to purchase.

All high-level languages offer some data structure to that effect. In many instances, the basic building block is called an **array**, defined as a *collection of elements* (values or variables), each identified by an *index* (plurals *indices*).

The following expression declares a variable `fruits` as an array of 3 strings:

```
var fruits [3]string
```

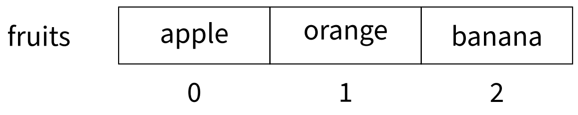
There are no fruits in the array yet, or more precisely, each fruit is the empty string.



You can assign actual values to an index:

```
fruits[0] = "apple"
fruits[2] = "banana"
fruits[1] = "orange"
```

Which can be represented as:



The first element is at position 0. Using 0 rather than 1 as the index of the first element is a widely spread convention in computer programming. There is no obligation to fill all the slots, nor to fill them in any particular order, as shown in the example above. You can also declare the array with its elements:

```
fruits := [3]string{"apple", "banana", "orange"}
```

The compiler can count the elements for you, eliminating the need to explicitly declare how many of them are present by using `...`, like so:

```
fruits := [...]string{"apple", "banana", "organge"}
```

The array has a length, accessible with `len`

```
fmt.Println(len(fruits))
```

```
3
```

An array cannot be resized. Not to worry, in Go there is a more potent data structure at our disposal called *slice*.

4.2 Slices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

4.3 Filtering

A common operation on lists is to filter values from a slice that match a certain criteria. For instance, we have a list of scores, ranging from 0 to 20. We want to keep all scores equal or above 12. Putting together what we already know about loops, slices, and conditional statements, we can write the following program:

```go
scores := []int{12, 14, 20, 3, 10, 16}
success := make([]int, 0)
for _, score := range scores {
    if score >= 12 {
        success = append(success, score)
    }
}
fmt.Println(success)
```

```
[12 14 20 16]
```

Back to the kitchen. Let's say we want to skip every other fruits, and store the result in another slice called `skimmed`. To skip fruits in our iteration, instead of incrementing with `i++`, we will increase `i` by `2`. For the sake of simplicity, we will do something inefficient by declaring `skimmed` with a length of `0` and relying only on **`append`** to expand the slice.

```go
fruits := []string{"apple", "banana", "orange", "grapefruit"}
skimmed := make([]string, 0)
for i := 0; i < len(fruits); i = i + 2 {
    skimmed = append(skimmed, fruits[i])
}
fmt.Printf("%d fruits in %v\n", len(skimmed), skimmed)
```

A more efficient technique would be to declare `skimmed` with the proper length, that is, half of `fruits` length. On top of that, we would need a second index to remember where we stand in `skimmed`, and that index would be different than the index in `fruits`. We could also have leveraged the slice **capacity**, but we have left out that characterstic of slices in order to focus on algorithms. Hopefully our naive and simple approach does the trick.

```
2 fruits in [apple orange]
```

4.4 Min and max

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

4.5 Generics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo. ## Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

5 Complex data types

5.1 Maps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

5.2 struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

5.3 Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

5.4 Sets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

6 Go techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

6.1 Pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

6.2 Concurrency (goroutines)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

6.3 Channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

6.4 Producer-consumer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

7 Programming techniques

7.1 Recusrsion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

8 Classic computer problems

8.1 Fibonacci numbers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

8.2 Hangman

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

8.3 Eight Queens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

8.4 Conway's Game Of Life

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

8.5 Tower of Hanoi

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

8.6 Blackjack (guided exercise)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ipgo.

9 Credits

- Hard drive, Wikipedia, licensed under CC BY-SA 3.0.
- Gopher mascot by Takuya Ueda, licensed under the Creative Commons 3.0 Attributions license.
- Ariane 501, Copyright ESA
- Thinking monkey, photo by Juan Rumimpunu, licensed under Unsplah license. https://unsplash.com/photos/nLXOatvTaLo
- Chessboard, Lichess.
- Tower of Hanoi, Wikipedia, licensed under CC BY-SA 3.0.
- Fibonnaci tiles, Wikipedia, licensed under Creative Commons Attribution-Share Alike 4.0 International
- Blackjack table, Wikipedia, licensed under Creative Commons CC0 1.0 Universal Public Domain Dedication
- Pointer meme, u/tuunraq, Reddit, all rights reserved.