JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#19

ITERATION - FOR & RANGE

ONE LOOP TO RULE THEM ALL

The Three Faces of For

Simplicity by Design

Go keeps it lean. One keyword handles every iteration scenario.

```go
// Standard C-style
for i := 0; i < 10; i++ {
    ...
}

// The "While" equivalent
for condition {
    ...
}

// The Infinite Loop
for {
    ...
}
```

Use the simplest form possible. If you don't need an index, don't use the C-style loop. Clean code is about reducing cognitive load.

The Range Clause

Iterating over Collections

range is the idiomatic way to iterate over slices, maps, strings, and channels.

```go
for index, value := range slice {
    ...
}

for key, value := range map {
    ...
}

for _, char := range "hello" {
    ...
}
```

On a string, *range* iterates over runes (Unicode points), not bytes. This is a critical distinction for internationalization.

The Copy Trap

Range is a Value Copy

When you use *range*, the second variable is a copy of the element, not a reference to it.

```go
type User struct {
    ID int
    Active bool
}
users := []User{
    {
        ID: 1,
        Active: false
    }
}

for _, u := range users {
    u.Active = true // This only updates the COPY
}
// users[0].Active is still false!
```

If you need to modify the original slice, use the index: *users[i].Active = true*.

The Pointer Trap (Pre-Go 1.22)

Reusing the Iteration Variable

Historically, Go reused the same memory address for the *v* in for *i, v := range*.

```go
var out []*int
for i, v := range []int{1, 2, 3} {
    out = append(out, &v) // All elements point to the same address!
}
// Result: [3, 3, 3]
```

* While Go 1.22+ fixed this for loop variables, understanding this "shared memory" behavior is vital for maintaining legacy code and understanding closures in loops.

The Randomness Factor

Map Iteration is Non-Deterministic

Never assume map order. Go intentionally randomizes map iteration to prevent developers from relying on a specific order.

```go
m := map[string]int{
        "Alpha": 1,
        "Beta":  2,
        "Gamma": 3,
}

fmt.Printf("Order:")
for k, v := range m {
    fmt.Printf("%s:%d ", k, v)
}
```

```
go run main.go
Order: Gamma:3 Alpha:1 Beta:2
go run main.go
Order: Beta:2 Gamma:3 Alpha:1
```

Warning: If your logic requires a sorted output (like an API response), you must collect the keys, sort them manually, and then iterate.

Efficient Slicing in Loops

Avoiding Unnecessary Allocations

If you are filtering a slice inside a loop, don't create a new slice unless necessary.

```go
n := 0
// The in-place filter (remember this one? :) )
for _, x := range slice {
    if keep(x) {
        slice[n] = x
        n++
    }
}
slice = slice[:n] // Efficient, 0-allocation filter
```

This pattern reuses the underlying array, drastically reducing GC pressure.

Channel Iteration

Looping Until the Close

In Go, *range* isn't just for collections in memory. You can use it to iterate over a Channel, creating a natural queue. The loop effectively "subscribes" to the channel and stays open as long as there is data being sent.

```go
queue := make(chan string, 3) // Create a channel, will get to that in the future

// The channel is open and receives input async...

// The Iteration Loop
// This loop "awaits" each message and exits gracefully when closed.
for msg := range queue {
        fmt.Println("Processing:", msg)
}
```

The "Close" Contract: The *range* loop only exits when the channel is closed. If you forget to *close(queue)*, your program will deadlock or leak a Goroutine.
Pattern Recognition: Use this when you don't know how many items are coming (e.g., streaming logs, processing webhooks). It's far cleaner than a manually managed *for* loop with if *ok := <-queue*.

Summary:

- Use range for readability, but watch for copies.
- Modify slices via index, not value.
- Beware of map order randomization.
- Check your Go version for loop variable behavior.

Tonight we look at Advanced Patterns: Generators & Streams - using loops and closures to handle data that doesn't fit in memory.