JANUARY 2026 SERIES

# FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

## #05

# STRING VS. RUNE VS. BYTE

MEMORY EFFICIENCY

# What is a String, really?

## A string in Go is a read-only slice of bytes.

In many languages, a string is just an array of characters..
A string in Go is a read-only slice of bytes.
When you pass a string to a function, you aren't copying the whole text—you're passing a tiny 16-byte header.

```
// Internal Go representation
type stringStruct struct {
    str unsafe.Pointer
    len int
}
```

How a string is implemented in Go

# Byte vs. Rune

## The Anatomy of a Character

Byte (uint8): 1 byte. Perfect for standard ASCII (a–z, 0–9).

Rune (int32): 4 bytes. Represents a single Unicode Code Point.

```
var r Rune = 'a'
var b Byte = 'b'
```

*Because Go uses UTF-8, a single "character" can take up anywhere from 1 to 4 bytes.

# The Length Trap

## Why len(s) might lie to you

In Go, using the len() function returns the number of bytes in a string, not number of characters.

In cases where our data might contain ‘special’ characters (emojis, characters encoded with more than 1 byte, etc.), we might get the wrong result when trying to count characters in a string.

```
go run main.go
s =  coûts
len(s) =  6
utf8.RuneCountInString(s) =  5
```

To get the actual number of characters (runes) in a string, you can use the utf8.RuneCountInString() function from the built-in unicode/utf8 package

# Iteration Pitfall (Index)

## The Indexing Bug

Accessing by index s[i] retrieves a single byte.
This will mangle any multi-byte characters like emojis or non-English letters.

```go
s := "Go 🌍"
for i := 0; i < len(s); i++ {
    fmt.Printf("%c ", s[i]) // Prints: G o followed by broken bits
}
```

```
go run main.go
G o   ð
```

# Iteration Success (Range)

## The Idiomatic Way

The range keyword is "Unicode-aware", it automatically decodes UTF-8 on the fly, giving you the full rune and its starting byte position. This is the only safe way to iterate strings.

```go
s := "Go 🌍"
for index, runeVal := range s {
    fmt.Printf("%c starts at byte %d\n", runeVal, index)
}
```

```
go run main.go
G starts at byte 0
o starts at byte 1
  starts at byte 2
🌍 starts at byte 3
```

# Conversions & Allocations

## The Cost of Casting

Converting a string to a slice creates a new copy in memory

```
b := []byte(s) // Allocation!
r := []rune(s) // Even heavier Allocation!
```

In high-performance loops, avoid these conversions. Use the raw string or the strings package directly to save the Garbage Collector (GC) some work

# Optimization: strings.Builder

## Building Strings Efficiently

Strings are immutable, therefore using + in a loop creates a new string every time.
strings.Builder minimizes allocations by reusing a buffer.

```go
import "strings"

var b strings.Builder
for range 10 {
    b.WriteString("Go")
}
result := b.String() // result = "GoGoGo...." concatenated 10 times
```

GOLANG /// BA

# To summarize:

- len() counts bytes.
- range decodes runes.
- strings.Builder saves memory.

**Think in Bytes, Write in Runes.**