**Strings, Runes, and Bytes**

**1. Strings in Go**

* A **string** in Go is a **read-only slice of bytes** ([]byte).
* Go uses **UTF-8 encoding** for strings, which means:
  + Characters (Unicode code points) are encoded using **1 to 4 bytes**.
  + Regular **ASCII characters** (like a, b, c) take **1 byte**.
  + Extended characters (like è, ù) take **2+ bytes**.

**2. Indexing Strings: Bytes, Not Characters**

var myString = "rèsumè"

var indexed = myString[1]

* Indexing a string gives you **a byte**, not a character.
* indexed is of type uint8 (a byte).
* In "rèsumè", the character è is encoded using **two bytes**, so indexing may give you only **part of a character**.

fmt.Printf("%v %T\n", indexed, indexed) //195 uint8

* The number 195 is the **first byte** of the UTF-8 encoded è.

**3. Iterating Strings Using range**

for ind, val := range myString {

    fmt.Println(ind, val)

}

* When you iterate over a string with range, Go decodes **each complete character (rune)**.
* ind is the **starting byte index** of each rune.
* val is the **rune** (i.e., the Unicode code point).

Output:

0 114 // 'r'

1 232 // 'è' (Unicode: 232)

* 1. 115 // 's'

...

Notice how the index **jumps** (e.g., from 1 to 3) because multi-byte characters like è take up **two bytes**, so Go skips over them.

**4. String Length: Bytes, Not Characters**

* fmt.Printf("Length: %v\n", len(myString))
* len(myString) gives the number of **bytes**, not characters (runes).
* "rèsumè" has 6 characters, but more than 6 bytes ( 8 bytes).

**5. Runes: Proper Way to Handle Unicode Characters**

var myString2 = []rune("rèsumè")

* Converting a string to a slice of **runes** gives you the actual **Unicode code points** (i.e., complete characters).
* Now indexing will return a full character.

var indexed2 = myString2[1]

fmt.Println(indexed2) // 232

fmt.Printf("%v %T\n", indexed2, indexed2) // 232 int32

* Runes are of type **int32**.
* You get correct indexing, and the length will now reflect the **character count**, not byte count.

fmt.Printf("Length: %v\n", len(myString2)) // Output: 6

**6. Declaring Runes**

var rune1 = 'a'

fmt.Printf("\nrune1= %v", rune1) // Output: rune1 = 97

* A rune can be declared using **single quotes**.
* It's stored as the **Unicode code point** (int32).

**7. String Concatenation: Immutable Strings**

var strSlice = []string{"a", "n", "a", "c", "o", "n", "d", "a"}

    var catstr = ""

    for i := range strSlice {

        catstr += strSlice[i]

    }

    fmt.Printf("\n%v", catstr)

* Strings are **immutable** in Go.
* Every time you concatenate (+=), Go creates a **new string** in memory.
* This is **inefficient** for large operations.

// This line won't work:

catstr[5]='a'

// Error: cannot assign to catstr[0]

**8. Efficient String Building with strings.Builder**

    var strSlice2 = []string{"p", "y", "t", "h", "o", "n"}

    var strBuilder strings.Builder

    for i := range strSlice2 {

        strBuilder.WriteString(strSlice2[i])

    }

    var catstr2 = strBuilder.String()

    fmt.Printf("\n%v", catstr2)

* strings.Builder provides a way to **build strings efficiently**.
* Internally, it uses a **growable byte buffer**.
* Much faster and memory-friendly compared to repeated concatenation.

**Summary of Key Takeaways:**

| **Concept** | **Behavior** |
| --- | --- |
| string[index] | Returns a **byte** (uint8) |
| range string | Iterates over **runes (Unicode points)** |
| len(string) | Returns **byte length**, not character count |
| []rune(string) | Converts string into **slice of runes** (int32) |
| Strings are immutable | You **cannot modify** them in place |
| strings.Builder | Use it for **efficient** string construction |