# The for range Construct

This lesson discusses another variation for running loops in Go, i.e., the for range construct

WE'LL COVER THE FOLLOWING



## Infinite loop

The condition can be absent like in:

```
for i:=0; ; i++ or for { }
```

This is the same as `for ;; { }` but the ; ; is removed by *gofmt*. These are **infinite loops**. The latter could also be written as:

```
for true { }
```

But the normal format is:

```
for { }
```

If a condition check is missing in a for-header, the condition for looping is and remains always true. Therefore, in the loop-body, something has to happen in order for the loop to be exited after a number of iterations. Always check that the exit-condition will evaluate to true at a certain moment in order to avoid an endless loop! This kind of loop is exited via a **break** statement, which we'll study in the next lesson, or a **return** statement. But, there is a difference. *Break* only exits from the loop while *return* exits from the function in which the loop is coded!

## Use of `for range`

The `for range` is the iterator construct in Go, and you will find it useful in a lot of contexts. It is a very elegant variation used to make a loop over every item in a collection. The general format is:

```
for ix, val := range coll { }
```

The `range` keyword is applied to an indexed collection `coll`. Each time `range` is called, it returns an index `ix` and the copy of value `val` at that index in the collection `coll`. So the first time `range` is called on `coll`, it returns the **first** element; that is `ix==0` and `val==coll[0]==coll[ix]`. This statement is similar to a `foreach` statement in other languages, but we still have the index `ix` at each iteration in the loop.

Be careful! Here, `val` is a copy of the value at that index `ix` in the collection `coll`, so it can be used only for read-purposes. The real value in the collection cannot be modified through `val`.

Now, let's solve the issue we encountered when printing characters of a Unicode string.

```
package main
import "fmt"

func main() {
    str := "Go is a beautiful language!"

    // for range
    for pos, char := range str {
        fmt.Printf("Character on position %d is: %c \n", pos, char)
    }
    fmt.Println()
    str2 := "Chinese: 日本語"

    // for range
    for pos, char := range str2 {
        fmt.Printf("Character %c starts at byte position %d\n", char, pos)
    }
}
```

Unicode String with For Range

As you can see at a **line 5**, we declare a string `str` and initialize it with **"Go is a beautiful language!"**. Then at **line 8**, we made a `for range` loop: `for pos, char := range str`. Here, `pos` is the position of a character `char` in string `str`. The loop will run `len(str)` times. Inside its body, at **line 9**, we are printing every instance of the string along with its position line by line. Similarly, we declare another string at **line 12** as `str2` and initialize it with **"Chinese: 日本語"**. At **line 15**, we made a `for range` loop: `for pos, char := range str2`. Here, `pos` is the position of a character `char` in string `str2`. The loop will run until the whole `str2` isn't traversed index by index. Inside the loop's body at **line 16**, we are printing every instance of the string line by line along with its position. The output will show characters of each string line by line. We see that the normal English characters are represented by **1 byte**, and the Chinese characters by **3 bytes**.

---

That's it about how control is transferred using the `for range` construct. The next lesson describes another control construct in Go, also known as the *break and continue* construct.