# rangefuncs – iterators in Go

Bärner Go Meetup – 11.09.2024
Lucas Bremgartner

## Hello!
## I’m Lucas

lucas@bremis.ch | github.com/breml | linkedin.com/in/lucas-bremgartner/

## What are Iterators?

**Wikipedia:**

*In computer programming, an iterator is an object that progressively provides access to each item of a collection, in order.*

**ChatGPT (for a 5 year old):**

**Imagine you have a box full of colorful marbles. If you want to look at each marble one by one, you could ask a friend to hand you each marble, one at a time. Your friend is like an "iterator".**

**Examples from Go Stdlib:**

- `bufio.Reader.ReadByte`
- `bufio.Scanner.Scan`
- `database/sql.Rows`
- `path/filepath.Walk`

## Situation pre Go 1.23

- Standardized way to iterate over collections ❌
- State tracking hidden from the caller ❌
- Automatic cleanup after use (e.g. close a file or a DB result set) ❌

Additionally:

- Foundation laid for more custom containers (generics) with iterators ❌
- Foundation laid to replace “collect and return slice” with iterators ❌
- Future addition of iterators to Go Stdlib unblocked ❌

## Refresher – Loops pre Go 1.23

**C style:**

```
for i := 0; i < 10; i++ {…
```

**For-range-loop for *array, slice, string, map, channel*:**

```
for i, v := range []int{2, 4, 5} {…
```

**And since Go 1.22: range over int:**

```
for v := range 3 {…
```

**same as**

```
for i := 0; i < 3; i++ {…
```

## Enter range-over-func aka Iterators in Go*

**With Go 1.23, the `for-range` loop allows to “range over a function”.**

**3 forms are supported, differing only in the number of return values:**

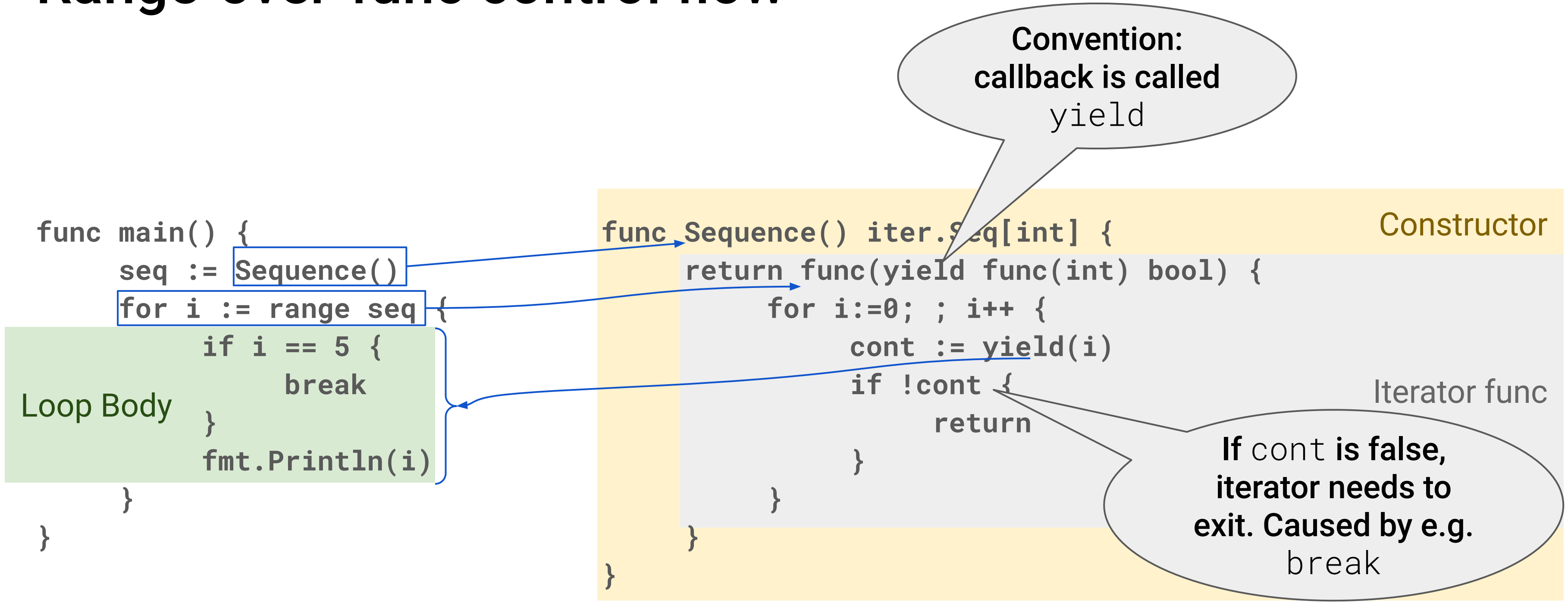
- **0 values:** `for range myIterator { … }`
- **1 value:** `for v := range myIterator { … }`
- **2 values:** `for k, v := range myIterator { … }`

**Iterator signatures:**

- **0 values:** `func(func() bool)`
- **1 value:** `func(func(V) bool)` **or** `iter.Seq[V any]`
- **2 values:** `func(func(K, V) bool)` **or** `iter.Seq2[K, V any]`

* available since Go 1.23, released 13.08.2024 and as experiment in Go 1.22

# Range-over-func control flow


Convention: callback is called `yield`
Constructor
Iterator func
Loop Body
If `cont` is false, iterator needs to exit. Caused by e.g. `break`

```go
func main() {
	seq := Sequence()
	for i := range seq {
		if i == 5 {
			break
		}
		fmt.Println(i)
	}
}
```

```go
func Sequence() iter.Seq[int] {
	return func(yield func(int) bool) {
		for i:=0; ; i++ {
			cont := yield(i)
			if !cont {
				return
			}
		}
	}
}
```

## Show me some code

```
package main

import "fmt"

func main() {
 a := make([]int, 5)
 printSlice("a", a)
 b := make([]int, 0, 5)
```

## Range-over-func stages

```
func Sequence(ctx context.Context) func(yield func() bool) {
	// ① Global initialization of iterator

	context.AfterFunc(ctx, func() {
		// ⑧ Global cleanup of iterator
	})

	return func(yield func() bool) {
		// ② Setup, before iterations

		defer func() {
			// ⑦ Teardown, cleanup after iterations
		}()

		for {
			// ③ before each iteration

			// ④ call loop body
			cont := yield()

			// ⑤ after each iteration

			if !cont {
				// ⑥ before cancel of iterator (external)
				return
			}
		}

		// ⑥ before end of iterator (internal)
	}
}
```

## More code

```
package main

import "fmt"

func main() {
 a := make([]int, 5)
 printSlice("a", a)
b := make([]int, 0, 5)
```

## Situation with Go 1.23

- Standardized way to iterate over collections ✅
- State tracking hidden from the caller ✅
- Automatic cleanup after use (e.g. close a file or a DB result set) ✅

Additionally:

- Foundation laid for more custom containers (generics) with iterators ✅
- Foundation laid to replace “collect and return slice” with iterators ✅
- Future addition of iterators to Go Stdlib unblocked ✅

## Push versus Pull Iterators

- **Push iterators push the value to the loop body (all Examples so far).**
- **Convert with** `iter.Pull(iter.Seq)` **and** `iter.Pull2(iter.Seq2)` **to “pull-style” iterators.**
- **“Pull-style” iterators are accessed by two functions:** `next()` **and** `stop()`

**Example:**

```
func Contains[T any](seq iter.Seq[T], needle T) bool {
  next, stop := iter.Pull(seq)
  defer stop()

  for v, ok := next(); ok; v, ok = next() {
    if v == needle {
      return true
    }
  }
  return false
}
```

**Stop iterator when done**

**Pull next value**

## Thank you

# Resources

Official Resources

- Release Notes: https://tip.golang.org/doc/go1.23#language,
- Official Blog Post: https://go.dev/blog/range-functions, Package iter: https://pkg.go.dev/iter
- Range func experiment: https://go.dev/wiki/RangefuncExperiment, https://github.com/golang/go/issues/61405, https://github.com/golang/go/discussions/56413
- Range func loop rewrite implementation: https://go.googlesource.com/go/+/refs/changes/41/510541/7/src/cmd/compile/internal/rangefunc/rewrite.go

Selected Blog Posts

- https://bitfieldconsulting.com/posts/iterators
- https://www.ardanlabs.com/blog/2024/04/range-over-functions-in-go.html
- https://medium.com/eureka-engineering/a-look-at-iterators-in-go-f8e86062937c

Critique

- https://itnext.io/go-evolves-in-the-wrong-direction-7dfda8a1a620
- https://www.gingerbill.org/article/2024/06/17/go-iterator-design/