Range-over function iterators

Fatih Arslan
Software Engineer @**PlanetScale**

GoKonf 24'
İstanbul

About me

- Software Engineer **@PlanetScale**
  Previously **GitHub, DigitalOcean, Koding**
- Creator of **vim-go**
- Go contributor. Go packages (i.e: **color, structs, hcl, gomodifytags**, etc..)
- **Coffee** and **Design geek**

What are range-over iterators?

For statements with `range` clause

A "for" statement with a "range" clause iterates through all entries of an array, slice, string or map, or values received on a channel. For each entry it assigns *iteration values* to corresponding *iteration variables* if present and then executes the block.

```
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .
```

The expression on the right in the "range" clause is called the *range expression*, its core type must be an array, pointer to an array, slice, string, map, or channel permitting receive operations. As with an assignment, if present the operands on the left must be addressable or map index expressions; they denote the iteration variables. If the range expression is a channel, at most one iteration variable is permitted, otherwise there may be up to two. If the last iteration variable is the blank identifier, the range clause is equivalent to the same clause without that identifier.

The range expression `x` is evaluated once before beginning the loop, with one exception: if at most one iteration variable is present and `len(x)` is constant, the range expression is not evaluated.

Function calls on the left are evaluated once per iteration. For each iteration, iteration values are produced as follows if the respective iteration variables are present:

```
Range expression                          1st value          2nd value

array or slice  a  [n]E, *[n]E, or []E    index    i  int    a[i]       E
string          s  string type            index    i  int    see below  rune
map             m  map[K]V                key      k  K      m[k]       V
channel         c  chan E, <-chan E       element  e  E
```

1. For an array, pointer to array, or slice value `a`, the index iteration values are produced in increasing order, starting at element index 0. If at most one iteration variable is present, the range loop produces iteration values from 0 up to `len(a)-1` and does not index into the array or slice itself. For a `nil` slice, the number of iterations is 0.
2. For a string value, the "range" clause iterates over the Unicode code points in the string starting at byte index 0. On successive iterations, the index value will be the index of the first byte of successive UTF-8-encoded code points in the string, and the second value, of type `rune`, will be the value of the corresponding code point. If the iteration encounters an invalid UTF-8 sequence, the second value will be `0xFFFD`, the Unicode replacement character, and the next iteration will advance a single byte in the string.
3. The iteration order over maps is not specified and is not guaranteed to be the same from one iteration to the next. If a map entry that has not yet been reached is removed during iteration, the corresponding iteration value will not be produced. If a map entry is created during iteration, that entry may be produced during the iteration or may be skipped. The choice may vary for each entry created and from one iteration to the next. If the map is `nil`, the number of iterations is 0.
4. For channels, the iteration values produced are the successive values sent on the channel until the channel is closed. If the channel is `nil`, the range expression blocks forever.

The iteration values are assigned to the respective iteration variables as in an assignment statement.

For statements with `range` clause

A "for" statement with a "range" clause iterates through all entries of an array, slice, string or map, or values received on a channel. For each entry it assigns *iteration values* to corresponding *iteration variables* if present and then executes the block.

```
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .
```

The expression on the right in the "range" clause is called the *range expression*, its core type must be an array, pointer to an array, slice, string, map, or channel permitting receive operations. As with an assignment, if present the operands on the left must be addressable or map index expressions; they denote the iteration variables. If the range expression is a channel, at most one iteration variable is permitted, otherwise there may be up to two. If the last iteration variable is the blank identifier, the range clause is equivalent to the same clause without that identifier.

The range expression `x` is evaluated once before beginning the loop, with one exception: if at most one iteration variable is present and `len(x)` is constant, the range expression is not evaluated.

Function calls on the left are evaluated once per iteration. For each iteration, iteration values are produced as follows if the respective iteration variables are present:

```
Range expression                          1st value          2nd value

array or slice  a  [n]E, *[n]E, or []E    index    i  int    a[i]       E
string          s  string type            index    i  int    see below  rune
map             m  map[K]V                key      k  K      m[k]       V
channel         c  chan E, <-chan E       element  e  E
```

1. For an array, pointer to array, or slice value `a`, the index iteration values are produced in increasing order, starting at element index 0. If at most one iteration variable is present, the range loop produces iteration values from 0 up to `len(a)-1` and does not index into the array or slice itself. For a `nil` slice, the number of iterations is 0.
2. For a string value, the "range" clause iterates over the Unicode code points in the string starting at byte index 0. On successive iterations, the index value will be the index of the first byte of successive UTF-8-encoded code points in the string, and the second value, of type `rune`, will be the value of the corresponding code point. If the iteration encounters an invalid UTF-8 sequence, the second value will be `0xFFFD`, the Unicode replacement character, and the next iteration will advance a single byte in the string.
3. The iteration order over maps is not specified and is not guaranteed to be the same from one iteration to the next. If a map entry that has not yet been reached is removed during iteration, the corresponding iteration value will not be produced. If a map entry is created during iteration, that entry may be produced during the iteration or may be skipped. The choice may vary for each entry created and from one iteration to the next. If the map is `nil`, the number of iterations is 0.
4. For channels, the iteration values produced are the successive values sent on the channel until the channel is closed. If the channel is `nil`, the range expression blocks forever.

The iteration values are assigned to the respective iteration variables as in an assignment statement.

Range expression			1st value			2nd value	
array or **slice**	a	[n]E, *[n]E, or []E	index	i	int	a[i]	E
string	s	string type	index	i	int	see below	rune
map	m	map[K]V	key	k	K	m[k]	V
channel	c	chan E, <-chan E	element	e	E		

Array or Slice

Code:

```
func main() {
 numbers := []int{17, 2, 2024, 2016, 4}

 for i, n := range numbers {
  fmt.Printf("[%d] %d\n", i, n)
 }
}
```

Output:

```
$ go run main.go
[0] 17
[1] 2
[2] 2024
[3] 2016
[4] 4
```

String

Code:

```
func main() {
 str := "GoKonf🚀 2024"

 for i, s := range str {
  fmt.Printf("[%d]\t%c\n", i, s)
 }
}
```

Output:

```
$ go run main.go
[0]     G
[1]     o
[2]     K
[3]     o
[4]     n
[5]     f
[6]     🚀
[10]
[11]    2
[12]    0
[13]    2
[14]    4
```

Maps

Code:

```
func main() {
  persons := map[string]string{
    "Dieter":  "Rams",
    "Richard": "Sapper",
    "Ray":     "Eames",
  }

  for k, v := range persons {
    fmt.Printf("%s, %s\n", k, v)
  }
}
```

Output:

```
$ go run main.go
Ray, Eames
Richard, Sapper
Dieter, Rams
```

Channels

Code:

```
func main() {
  names := make(chan string, 3)
  names <- "Dieter Rams"
  names <- "Charles Eames"
  names <- "Isamu Noguchi"
  close(names)

  for name := range names {
    fmt.Printf("%q\n", name)
  }
}
```

Output:

```
$ go run main.go
"Dieter Rams"
"Charles Eames"
"Isamu Noguchi"
```

Go 1.22

Range expression			1st value			2nd value	
array or slice	a	[n]E, *[n]E, or []E	index	i	int	a[i]	E
string	s	string type	index	i	int	see below	rune
map	m	map[K]V	key	k	K	m[k]	V
channel	c	chan E, <-chan E	element	e	E		
integer	n	integer type	index	i	int		
function, 0 values	f	func(func()bool) bool					
function, 1 value	f	func(func(V)bool) bool	value	v	V		
function, 2 values	f	func(func(K, V)bool) bool	key	k	K	v	V

NEW

Integers

Code:

```
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
```

Output:

```
$ go run main.go
0
1
2
3
4
5
6
7
8
9
```

Integers NEW

Code:

```
func main() {
 for x := range 10 {
  fmt.Println(x)
 }
}
```

Output:

```
$ go run main.go
0
1
2
3
4
5
6
7
8
9
```

Benchmarks

```
func BenchmarkRandInt(b *testing.B) {
    for i := 0; i < b.N; i++ {
        rand.Int()
    }
}
```

Benchmarks
(using range over integers)

```
func BenchmarkRandInt(b *testing.B) {
    for range b.N {
        rand.Int()
    }
}
```

Go 1.23

Range expression			1st value			2nd value	
array or slice	a	[n]E, *[n]E, or []E	index	i	int	a[i]	E
string	s	string type	index	i	int	see below	rune
map	m	map[K]V	key	k	K	m[k]	V
channel	c	chan E, <-chan E	element	e	E		
integer	n	integer type	index	i	int		
function, 0 values	f	func(func()bool) bool					
function, 1 value	f	func(func(V)bool) bool	value	v	V		
function, 2 values	f	func(func(K, V)bool) bool	key	k	K	v	V

NEW

Functions

NEW

(also the topic of my talk)

Code:

```
func main() {
 s := []string{"hello", “world”, “GoKonf”}

 for i, x := range backward(s) {
  fmt.Println(i, x)
 }
}
```

Output:

```
$ go run main.go
2 GoKonf
1 world
0 hello
```

The usual way

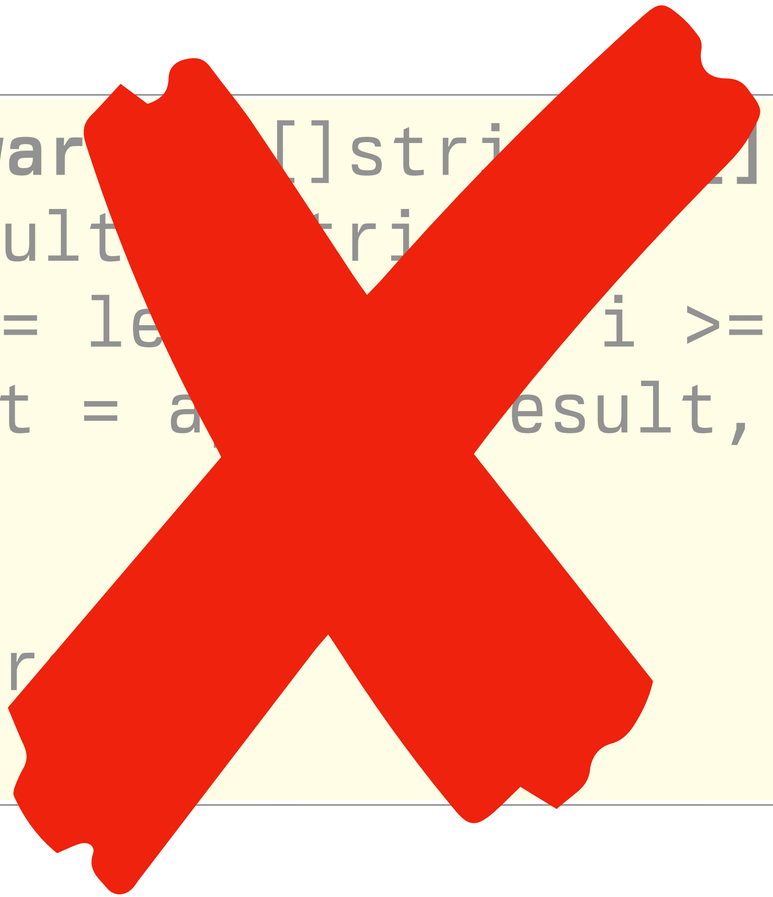
```
func backward(s []string) []string {
   var result []string
   for i := len(s) - 1; i >= 0; i-- {
     result = append(result, s[i])
   }

   return result
}
```

This is *not* ranging over functions

```
func backwar    []stri      ]string {
   var result  tri
   for i := le          i >= 0; i-- {
     result = a        esult, s[i])
   }

   return r
}
```



What exactly are range over functions?

Signature(s) of an iterator

```
func (yield func() bool)
func (yield func(V) bool)
func (yield func(K, V) bool)
```

Iterator

```
func numbers(yield func(int) bool) {
    for i := 0; i < 5; i++ {
        if !yield(i) {
            return
        }
    }
}
```

Run

```
func main() {
    for x := range numbers {
        fmt.Println(x)
    }
}
```

Output

```
0
1
2
3
4
```

Iterator with an argument

```
func numbers(n int) func(func(int) bool) {
    return func(yield func(int) bool) {
        for i := 0; i < n; i++ {
            if !yield(i) {
                return
            }
        }
    }
}
```

Run with argument

```
func main() {
    for x := range numbers(5) {
        fmt.Println(x)
    }
}
```

Output

```
0
1
2
3
4
```

Run with argument (3)

```
func main() {
    for x := range numbers(3) {
        fmt.Println(x)
    }
}
```

Output

```
0
1
2
```

The for-range loop over functions will be automatically **transformed** by the **compiler**.

How is this example converted by the compiler?

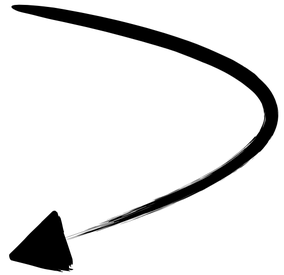
```
func main() {
    for x := range numbers {
        fmt.Println(x)
    }
}
```

Output

```
0
1
2
3
4
```

Conversion by the compiler

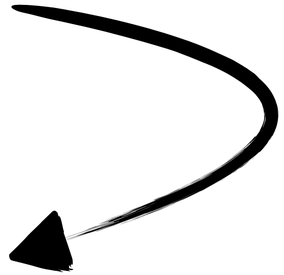
```
for x := range numbers {
  fmt.Println(x)
}
```



```
numbers(func (i int) bool) {
    fmt.Println(x)
    return true
})
```

Returns true if continue is used

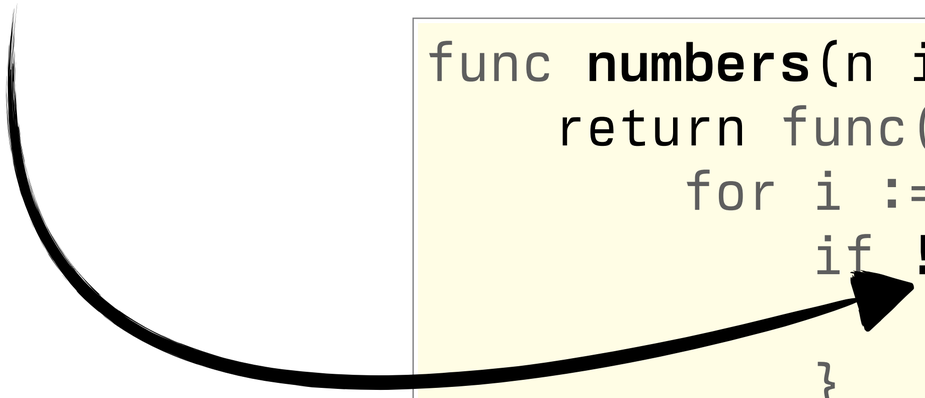
```
for x := range numbers {
  fmt.Println(x)
  // implicit continue
}
```



```
numbers(func (i int) bool) {
    fmt.Println(x)
    return true
})
```

Yield function returns true

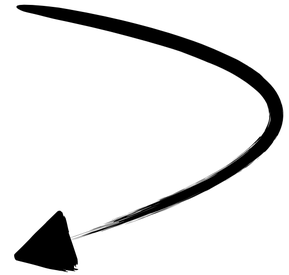
```
numbers(func (i int) bool) {
    fmt.Println(x)
    return true
})
```



```
func numbers(n int) func(func(int) bool) {
    return func(yield func(int) bool) {
        for i := 0; i < n; i++ {
            if !yield(i) {
                return
            }
        }
    }
}
```

Returns false if break is used

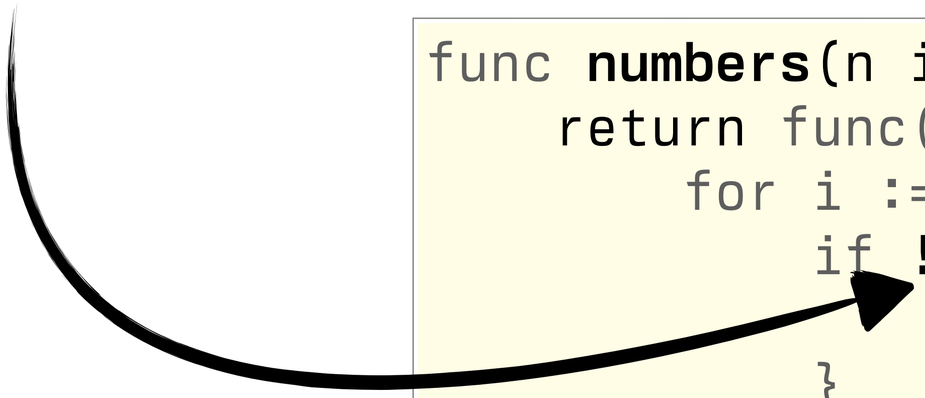
```
for x := range numbers {
  fmt.Println(x)
  break
}
```



```
numbers(func (i int) bool) {
    fmt.Println(x)
    return false
})
```

Yield function returns false

```
numbers(func (i int) bool) {
    fmt.Println(x)
    return false
})
```



```
func numbers(n int) func(func(int) bool) {
    return func(yield func(int) bool) {
        for i := 0; i < n; i++ {
            if !yield(i) {
                return
            }
        }
    }
}
```

cmd/compile/internal/rangefunc/rewrite.go

(implementation details)

iter

package

iter package

(predefined function types)

```
type Seq[V any]      func(yield func(V) bool)
type Seq2[K, V any] func(yield func(K, V) bool)
```

Without the iter package

```
func numbers(n int) func(func(int) bool) {
    return func(yield func(int) bool) {
        for i := 0; i < n; i++ {
            if !yield(i) {
                return
            }
        }
    }
}
```

Using the iter package

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

Always return an iterator

It make things easier

Examples

Backward

```
s := []string{"hello", “world”, “GoKonf”}

for _, x := range backward(s) {
  fmt.Println(x)
}

Output:
GoKonf
world
hello
```

Backward (imply.)

```
func backward(s []string) func(func(int, string) bool) {
    return func(yield func(int, string) bool) {
        for i := len(s)-1; i >= 0; i-- {
            if !yield(i, s[i]) {
                return
            }
        }
    }
}
```

Backward (alternative)

```
func backward(s []string) iter.Seq2[int, string] {
    return func(yield func(int, string) bool) {
        for i := len(s)-1; i >= 0; i-- {
            if !yield(i, s[i]) {
                return
            }
        }
    }
}
```

Let's convert to generic

```
func backward[E any](s []E) func(func(int, E) bool) {
    return func(yield func(int, E) bool) {
        for i := len(s)-1; i >= 0; i-- {
            if !yield(i, s[i]) {
                return
            }
        }
    }
}
```

Backward using generics (alternative)

```
func backward[E any](s []E) iter.Seq2[int, E] {
    return func(yield func(int, E) bool) {
        for i := len(s)-1; i >= 0; i-- {
            if !yield(i, s[i]) {
                return
            }
        }
    }
}
```

Backward

```
s := []int{2, 3, 4, 5}
// s := []string{"hello", “world”, “GoKonf”}

for _, x := range backward(s) {
  fmt.Println(x)
}

Output:
5
4
3
2
```

Benefits of using range-over functions

Traditional slice

```
func backward(s []string) []string {
   var result []string
   for i := len(s) - 1; i >= 0; i-- {
     result = append(result, s[i])
   }

   return result
}
```

Traditional slice (cont.)

```
s := []string{"hello", “world”, “GoKonf”}

for _, x := range backward(s) {
  fmt.Println(x)
}

Output:
GoKonf
world
hello
```

Range over function vs Traditional

- Range over is **one item at a time**, Traditional you have to **wait until the slice is populated**.
- Range over **doesn’t allocate**, Traditional **allocates** a whole slice.
- With Range over we **can early return and bail out**, Traditional you **don’t have that ability**.

Another Example

Time ticker

```
for range every(2 * time.Second) {
  log.Println("Hello!")
}
```

Time ticker (impl.)

```
func every(d time.Duration) func(func() bool) {
    return func(yield func() bool) {
        ticker := time.NewTicker(d)
        defer ticker.Stop()

        for {
            select {
            case <-ticker.C:
                if !yield() {
                    return
                }
            }
        }
    }
}
```

Time ticker (impl.)

```
func every(d time.Duration) func(func() bool) {
    return func(yield func() bool) {
        ticker := time.NewTicker(d)
        defer ticker.Stop()

        for {
            select {
            case <-ticker.C:
                if !yield() {
                    return
                }
            }
        }
    }
}
```

Time ticker (impl.)

```
func every(d time.Duration) func(func() bool) {
    return func(yield func() bool) {
        ticker := time.NewTicker(d)
        defer ticker.Stop()

        for {
            select {
            case <-ticker.C:
                if !yield() {
                    return
                }
            }
        }
    }
}
```

Time ticker with context

```
ctx, cancel := context.WithCancel(ctx)
defer cancel()

for range every(ctx, 2 * time.Second) {
  log.Println("Hello!")
}
```

Time ticker with context (impl.)

```
func every(ctx context.Context, d time.Duration) func(func() bool) {
    return func(yield func() bool) {
        ticker := time.NewTicker(d)
        defer ticker.Stop()

        for {
            select {
            case <-ticker.C:
                if !yield() {
                    return
                }
            case <-ctx.Done:
                return
            }
        }
    }
}
```

Time ticker with context (impl.)

```
func every(ctx context.Context, d time.Duration) func(func() bool) {
    return func(yield func() bool) {
        ticker := time.NewTicker(d)
        defer ticker.Stop()

        for {
            select {
            case <-ticker.C:
                if !yield() {
                    return
                }
            case <-ctx.Done:
                return
            }
        }
    }
}
```

Custom ticker vs time.Tick (*from std*)

- **Doesn’t leak**, because there is no underlying ticker
- **Context** support
- **Customizable**

Real world examples

Typical SQL query

```
rows, _ := db.Query("SELECT * FROM users")
defer rows.Close()

var user []Users
for rows.Next() {
   var user User
   err := rows.Scan(&user.Name, &user.Age, &user.City, &user.BirthDate)
   if err != nil {
    log.Fatal(err)
   }
   users = append(users, user)
}

if err = rows.Err(); err != nil {
   log.Fatal(err)
}
```

Typical SQL query

```
rows, _ := db.Query("SELECT * FROM users")
defer rows.Close()

var user []Users
for rows.Next() {
   var user User
   err := rows.Scan(&user.Name, &user.Age, &user.City, &user.BirthDate)
   if err != nil {
    log.Fatal(err)
   }
   users = append(users, user)
}

if err = rows.Err(); err != nil {
   log.Fatal(err)
}
```

Typical SQL query

```
rows, _ := db.Query("SELECT * FROM users")
defer rows.Close()

var user []Users
for rows.Next() {
   var user User
   err := rows.Scan(&user.Name, &user.Age, &user.City, &user.BirthDate)
   if err != nil {
    log.Fatal(err)
   }
   users = append(users, user)
}

if err = rows.Err(); err != nil {
   log.Fatal(err)
}
```

Typical SQL query

```
rows, _ := db.Query("SELECT * FROM users")
defer rows.Close()

var user []Users
for rows.Next() {
   var user User
   err := rows.Scan(&user.Name, &user.Age, &user.City, &user.BirthDate)
   if err != nil {
    log.Fatal(err)
   }
   users = append(users, user)
}

if err = rows.Err(); err != nil {
   log.Fatal(err)
}
```

Let’s use iterators!

github.com/achille-roussel/sqlrange

Using sqlrange

```
func Query(...) iter.Seq2[Row, error]
```

Using sqlrange

```
var users []User
for user, err := range sqlrange.Query[User](db, `SELECT * FROM users`) {
    if err != nil {
        break
    }

    users = append(users, user)
}
```

Benefit of using sqlrange

- Error **needs** to be handled.

```
var users []User
for user, err := range sqlrange.Query[User](db, `SELECT * FROM users`) {
    if err != nil {
        break
    }

    users = append(users, user)
}
```

Benefit of using sqlrange

- No need to **run** **`defer rows.Close()`**.

```
rows, _ := db.Query("SELECT * FROM users")
defer rows.Close()

var user []Users
for rows.Next() {
  var user User
  err := rows.Scan(&user.Name, &user.Age, &user.City, &user.BirthDate)
  if err != nil {
   log.Fatal(err)
  }
  users = append(users, user)
}
```

Benefit of using sqlrange

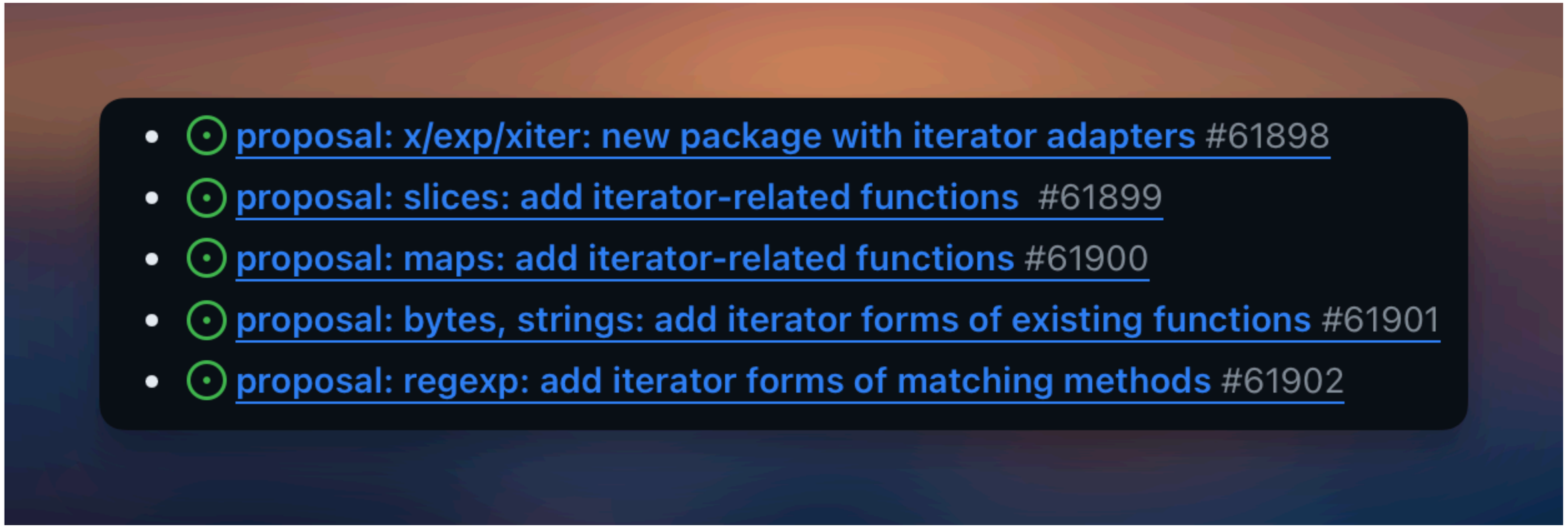
- **Less code**, easier to maintain.

```
var users []User
for user, err := range sqlrange.Query[User](db, `SELECT * FROM users
{
    if err != nil {
        break
    }

    users = append(users, user)
}
```

What’s next

stdlib proposals



maps

```
func All[Map ~map[K]V, K comparable, V any](m Map) iter.Seq2[K, V]
    All returns an iterator over key-value pairs from m.

func Collect[K comparable, V any](seq iter.Seq2[K, V]) map[K]V
    Collect collects key-value pairs from seq into a new map and
    returns it.

func Insert[Map ~map[K]V, K comparable, V any](m Map, seq iter.Seq2[K, V])
    Insert adds the key-value pairs from seq to m.

func Keys[Map ~map[K]V, K comparable, V any](m Map) iter.Seq[K]
    Keys returns an iterator over keys in m.

func Values[Map ~map[K]V, K comparable, V any](m Map) iter.Seq[V]
    Values returns an iterator over values in m.
```

slices

```
func All[Slice ~[]Elem, Elem any](s Slice) iter.Seq2[int, Elem]
func Backward[Slice ~[]Elem, Elem any](s Slice) iter.Seq2[int, Elem]
func Values[Slice ~[]Elem, Elem any](s Slice) iter.Seq[Elem]
func Append[Slice ~[]Elem, Elem any](x Slice, seq iter.Seq[Elem]) Slice
func Collect[Elem any](seq iter.Seq[Elem]) []Elem
func Sorted[Elem comparable](seq iter.Seq[Elem]) []Elem
func SortedFunc[Elem any](seq iter.Seq[Elem], cmp func(Elem, Elem) int) []Elem
func SortedStableFunc[Elem any](seq iter.Seq[Elem], cmp func(Elem, Elem) int) []Elem
```

bytes

```
// Lines returns an iterator over the newline-terminated lines
// in the string s.
func Lines(s string) iter.Seq[string]

// SplitSeq returns an iterator over all substrings of s separated by sep.
func SplitSeq(s, sep string) iter.Seq[string]

// SplitAfterSeq returns an iterator over substrings of s split after
// each instance of sep.
func SplitAfterSeq(s, sep string) iter.Seq[string]

// FieldsSeq returns an iterator over substrings of s split around runs of
func FieldsSeq(s string) iter.Seq[string]

// FieldsFuncSeq returns an iterator over substrings of s split
// around runs of
func FieldsFuncSeq(s string, f func(rune) bool) iter.Seq[string]
```

xiter
(defines **adapters** on **iterators**)

[**Concat**] and [**Concat2**] concatenate sequences.
[**Equal**], [**Equal2**], [**EqualFunc**], and [**EqualFunc2**] check whether two sequences contain equal values.
[**Filter**] and [**Filter2**] filter a sequence according to a function f.
[**Limit**] and [**Limit2**] truncate a sequence after n items.
[**Map**] and [**Map2**] apply a function f to a sequence.
[**Merge**], [**Merge2**], [**MergeFunc**], and [**MergeFunc2**] merge two ordered sequences.
[**Reduce**] and [**Reduce2**] combine the values in a sequence.
[**Zip**] and [**Zip2**] iterate over two sequences in parallel.

How to try it out?

```
$ go version
go version go1.22.0 darwin/arm64

# run your program
$ export GOEXPERIMENT=rangefunc
$ go run main.go
```

```
$ go version
go version go1.22.0 darwin/arm64


# run your program
$ export GOEXPERIMENT=rangefunc
$ go run main.go
```

Recap

- Range-over-function iterations let us **iterate over functions**
- Allows us to write **custom iterators for custom ADT's** (not just builtin slices, maps, channels, etc...)
- **Memory** and **CPU** efficient
- **Minimize** code, improve **maintenance** and **readability**

Thanks!

Fatih Arslan

 arslan.io

 @fatih

 @fatih

GoKonf 24'

İstanbul