Level 3–3

Following the Trail

Slice Literals & Looping With *range*

ON TRACK with GOLANG

Creating Slices With Initial Values

When we know beforehand which elements will be part of a slice, multiple calls to append will start looking **too verbose**. There's a better way...

```go
package main

import "fmt"

func main() {
  var a []string

  langs = append(langs, "Go")
  langs = append(langs, "Ruby")
  langs = append(langs, "JavaScript")

  fmt.Println(langs)
}
```

```
$ go run main.go
[Go Ruby JavaScript]
```

We can use a shorter syntax to create this slice in one line.

Slice Literals

A **slice literal** is a quick way to create slices with initial elements via type inference. We can pass elements between curly braces `{}`.

```go
package main

import "fmt"

func main() {
  langs := []string{"Go", "Ruby", "JavaScript"}
  fmt.Println(langs)
}
```

```
$ go run main.go
[Go Ruby JavaScript]
```

Element count is inferred from the number of initial elements.

Reading From a Slice by Index

One way to read from a slice is to **access elements by index**, just like an array.

```go
package main

import "fmt"

func main() {
  langs := []string{"Go", "Ruby", "JavaScript"}
  fmt.Println(langs[0])
  fmt.Println(langs[1])
  fmt.Println(langs[2])
}
```

Prints each individual element to the console.

```
$ go run main.go
```

```
Go
Ruby
JavaScript
```

Reading From a Slice With Unknown Size

While reading elements by index works, it doesn't scale well once our slice grows or when the exact number of elements **is not known until the program is run.**

```
...

func main() {
  langs := getLangs()

  fmt.Println(langs[???])

}

func getLangs() []string {
  ...
}
```

Based on the function signature, we can see this returns a slice, but the total number of elements is unknown.

Can't tell which index is valid.

The function signature tells us a slice will be returned from this function call.

Navigating a Slice With for and range

The range clause provides a way to iterate over slices. When only one value is used on the left of range, then this value will be the index for each element on the slice, one at a time.

```
...

func main() {
  langs := getLangs()

  for i := range langs {

  }
}

func getLangs() []string {
  ...
}
```

Loop runs once for each value in langs.

The index for each element is returned on each run of the loop.

Navigating a Slice With for and range

By using range on a slice, we can be sure the indices used are **always valid** for that slice.

```
...

func main() {
  langs := getLangs()

  for i := range langs {
    fmt.Println(langs[i])
  }
}

func getLangs() []string {
  ...
}
```

We can now safely use the index (of type int) to fetch each element from the slice...

```
$ go run main.go

Go
Ruby
JavaScript
```

...and print them to the console.

Unused Variables Produce Errors

Using range we can also read **the index** and **the element associated with it** at the same time. However, if we don't use a variable that's been assigned, then Go will produce an error.

```
...

func main() {
  langs := getLangs()

  for i, element := range langs {
    fmt.Println(element)
  }
}

func getLangs() []string {
  ...
}
```

For each run of the loop, this variable is assigned each actual element from the slice.

Not used anywhere else in the code. This will produce an error!

```
$ go run main.go
```

```
./main.go:8: i declared and not used
```

Ignoring Unused Variables With Underline

The underline character tells Go this value will not be used from anywhere else in the code.

```
...

func main() {
  langs := getLangs()

  for _, element := range langs {
    fmt.Println(element)
  }
}

func getLangs() []string {
  ...
}
```

We use the underline character to indicate variables that will not be used.

```
$ go run main.go
```

```
Go
Ruby
JavaScript
```

When given a single identifier, range assigns index, NOT the element.

```
for element := range langs {
  fmt.Println(element)
}
```

```
0
1
2
```