DWARVES
FOUNDATION

# Data Structures and Pointers

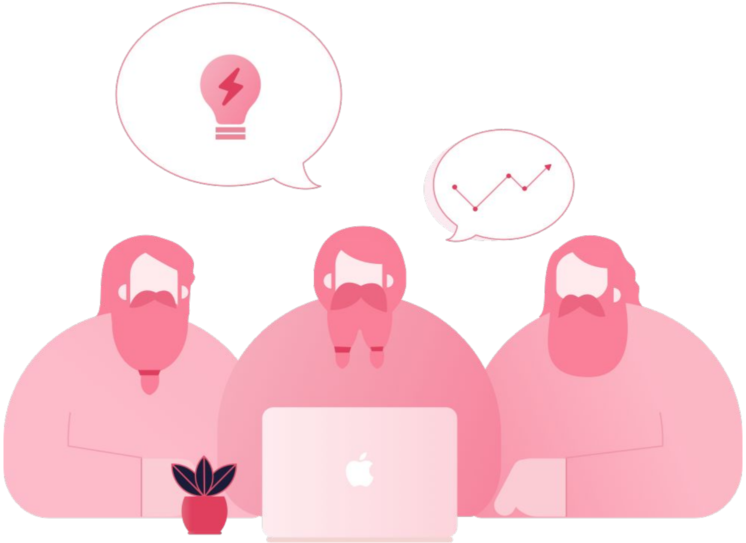
Pointers
Arrays, Slices and Maps

# Author Name

Hieu Phan

@hieupq

andy@dwarvesv.com



# Agenda

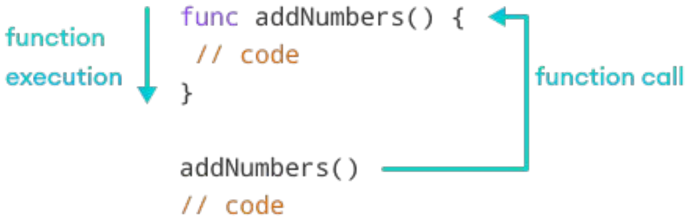
1. Introduction
2. Interface, struct
3. Arrays and Slices
4. Maps
5. Demo

# Introduction

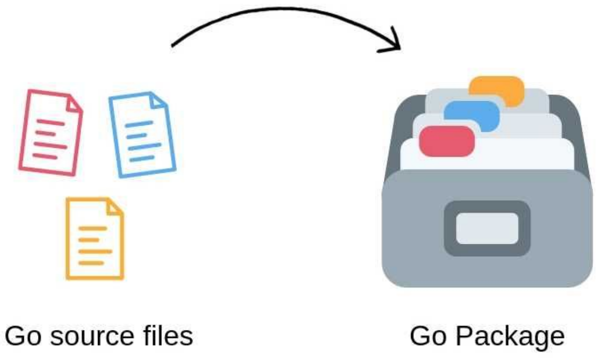
Recap of Day 2

# Day 2

- Functions as a fundamental building block of Go programs.
- Packages as a mechanism for organizing and reusing code.



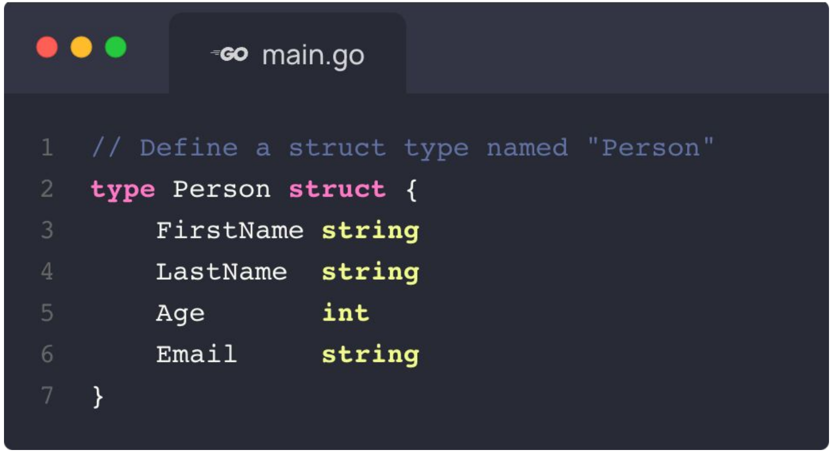
```
func addNumbers() {
 // code
}

addNumbers()
// code
```



# Interface & struct

# Struct

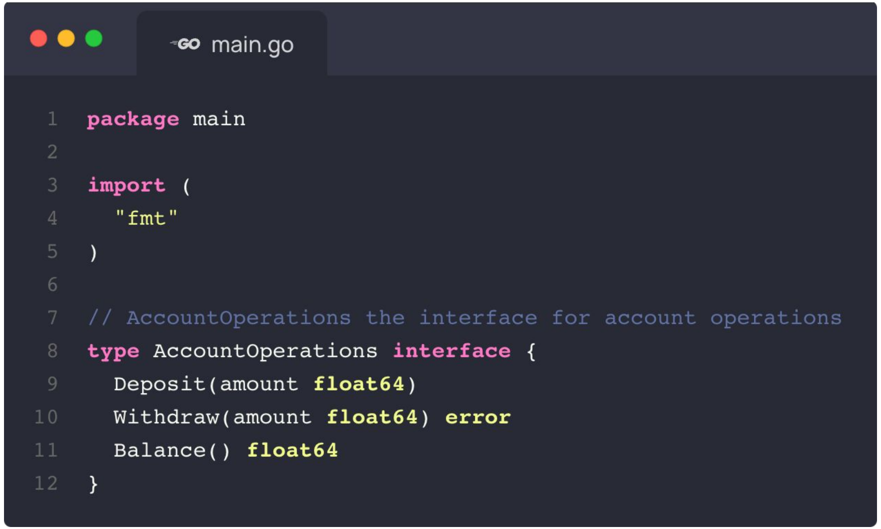
A struct is a composite data type that groups together variables with different data types under one name.



```go
// Define a struct type named "Person"
type Person struct {
    FirstName string
    LastName  string
    Age       int
    Email     string
}
```

# Interface

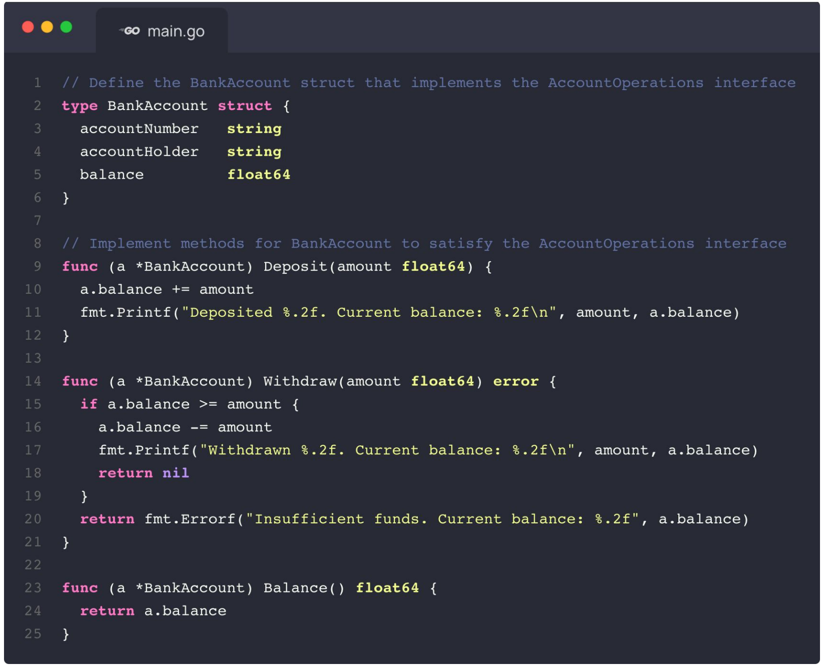
An interface defines a set of methods that a concrete type must implement to be considered as implementing that interface.



```go
package main

import (
  "fmt"
)

// AccountOperations the interface for account operations
type AccountOperations interface {
  Deposit(amount float64)
  Withdraw(amount float64) error
  Balance() float64
}
```

# Implementing Interfaces

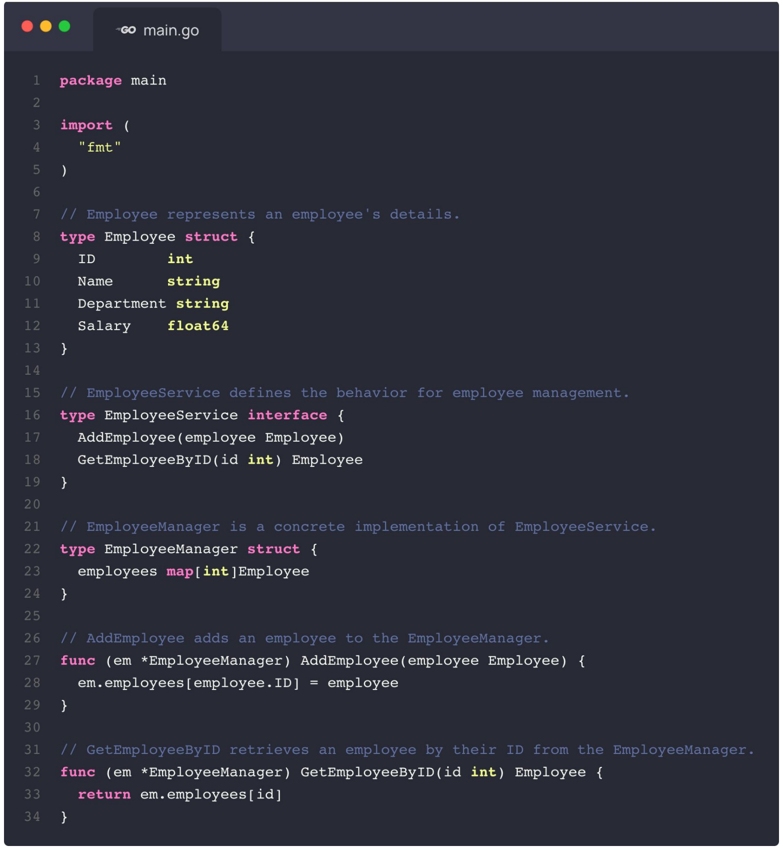
A struct automatically satisfies an interface if it implements all the required methods.



```go
// Define the BankAccount struct that implements the AccountOperations interface
type BankAccount struct {
  accountNumber   string
  accountHolder   string
  balance         float64
}

// Implement methods for BankAccount to satisfy the AccountOperations interface
func (a *BankAccount) Deposit(amount float64) {
  a.balance += amount
  fmt.Printf("Deposited %.2f. Current balance: %.2f\n", amount, a.balance)
}

func (a *BankAccount) Withdraw(amount float64) error {
  if a.balance >= amount {
    a.balance -= amount
    fmt.Printf("Withdrawn %.2f. Current balance: %.2f\n", amount, a.balance)
    return nil
  }
  return fmt.Errorf("Insufficient funds. Current balance: %.2f", a.balance)
}

func (a *BankAccount) Balance() float64 {
  return a.balance
}
```

# Best practices

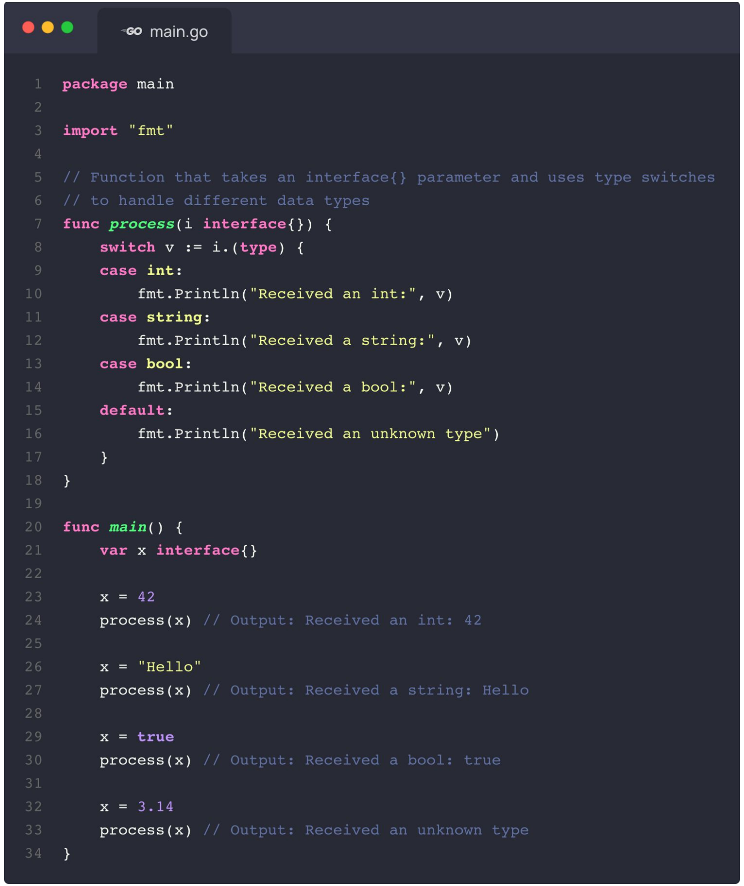
- Use meaningful field names
- Use pointers for large structs or when mutation is necessary
- Name interfaces with "-er" suffix: Reader, Writer, Logger



```go
package main

import (
  "fmt"
)

// Employee represents an employee's details.
type Employee struct {
  ID         int
  Name       string
  Department string
  Salary     float64
}

// EmployeeService defines the behavior for employee management.
type EmployeeService interface {
  AddEmployee(employee Employee)
  GetEmployeeByID(id int) Employee
}

// EmployeeManager is a concrete implementation of EmployeeService.
type EmployeeManager struct {
  employees map[int]Employee
}

// AddEmployee adds an employee to the EmployeeManager.
func (em *EmployeeManager) AddEmployee(employee Employee) {
  em.employees[employee.ID] = employee
}

// GetEmployeeByID retrieves an employee by their ID from the EmployeeManager.
func (em *EmployeeManager) GetEmployeeByID(id int) Employee {
  return em.employees[id]
}
```

# Interface{}

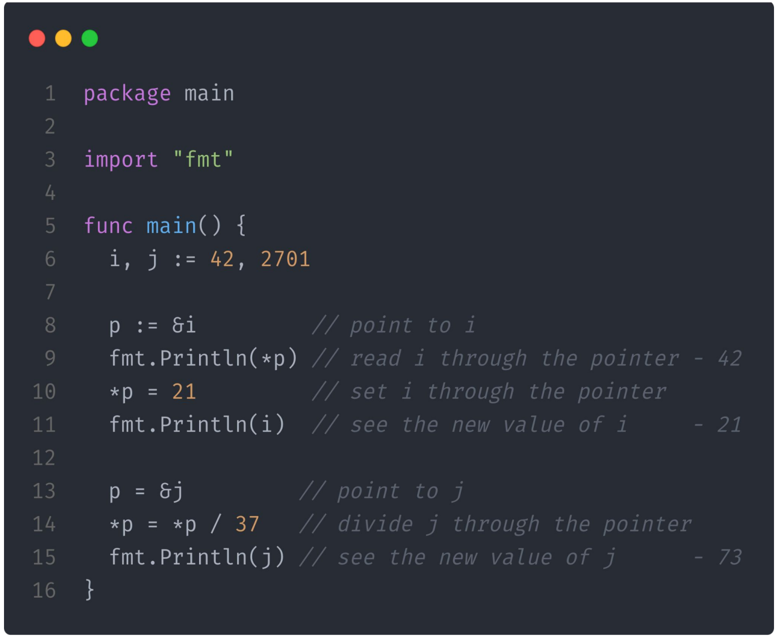
- Hold values of any data type
- Provides flexibility and allows you to handle unknown or mixed data types



```go
package main

import "fmt"

// Function that takes an interface{} parameter and uses type switches
// to handle different data types
func process(i interface{}) {
    switch v := i.(type) {
    case int:
        fmt.Println("Received an int:", v)
    case string:
        fmt.Println("Received a string:", v)
    case bool:
        fmt.Println("Received a bool:", v)
    default:
        fmt.Println("Received an unknown type")
    }
}

func main() {
    var x interface{}

    x = 42
    process(x) // Output: Received an int: 42

    x = "Hello"
    process(x) // Output: Received a string: Hello

    x = true
    process(x) // Output: Received a bool: true

    x = 3.14
    process(x) // Output: Received an unknown type
}
```

# Pointers

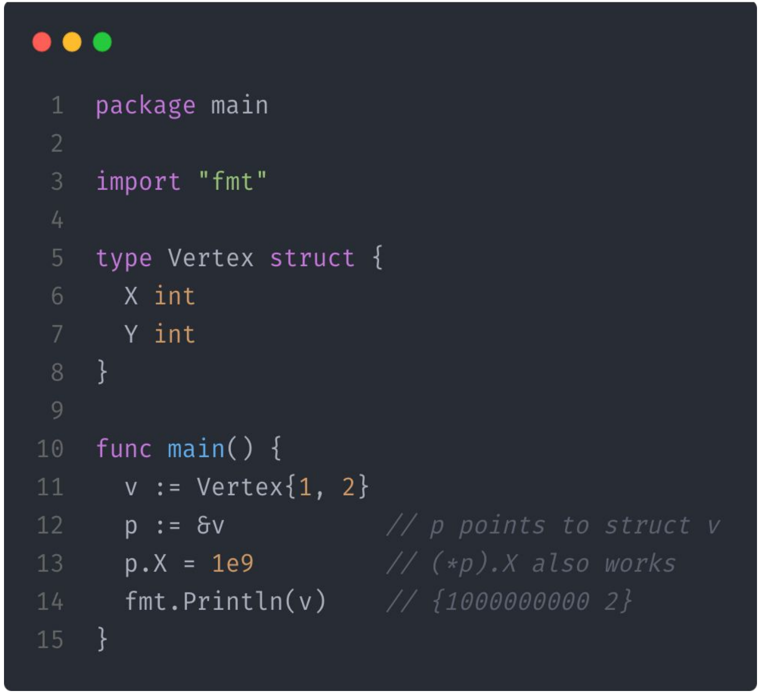
- A pointer holds a memory address to a value.
- Unlike C, Go has no pointer arithmetic (no ++, --, -, +, >, >=, <, <=, == !=)



```go
package main

import "fmt"

func main() {
  i, j := 42, 2701

  p := &i         // point to i
  fmt.Println(*p) // read i through the pointer - 42
  *p = 21         // set i through the pointer
  fmt.Println(i)  // see the new value of i     - 21

  p = &j         // point to j
  *p = *p / 37   // divide j through the pointer
  fmt.Println(j) // see the new value of j      - 73
}
```

# Pointers to structs

- Struct fields can be accessed through a struct pointer.
- Go permits us to right p.X instead of (*p).X for simplicity.

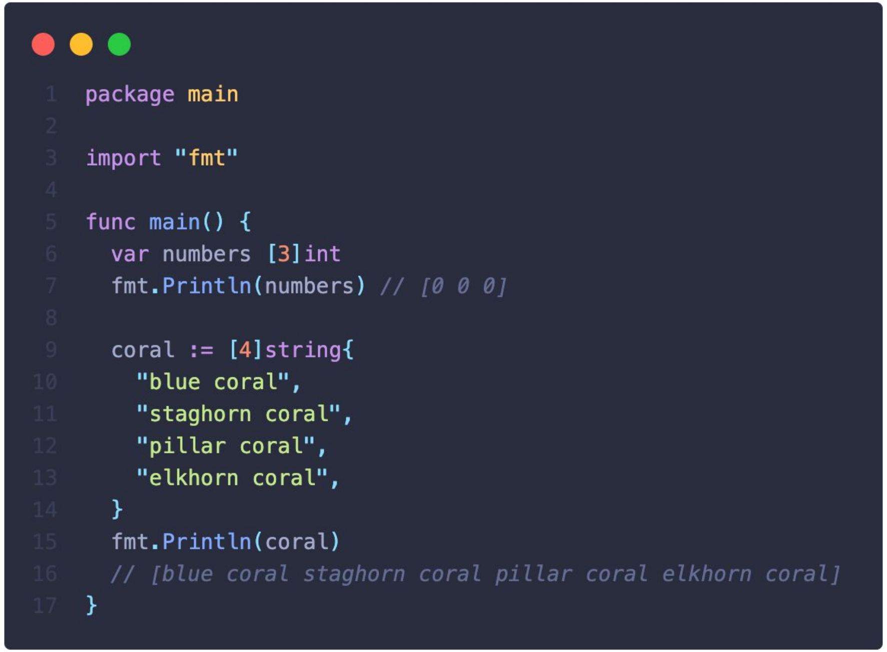


```go
package main

import "fmt"

type Vertex struct {
  X int
  Y int
}

func main() {
  v := Vertex{1, 2}
  p := &v           // p points to struct v
  p.X = 1e9         // (*p).X also works
  fmt.Println(v)    // {1000000000 2}
}
```

# Arrays

`[capacity]data_type{element_values}`

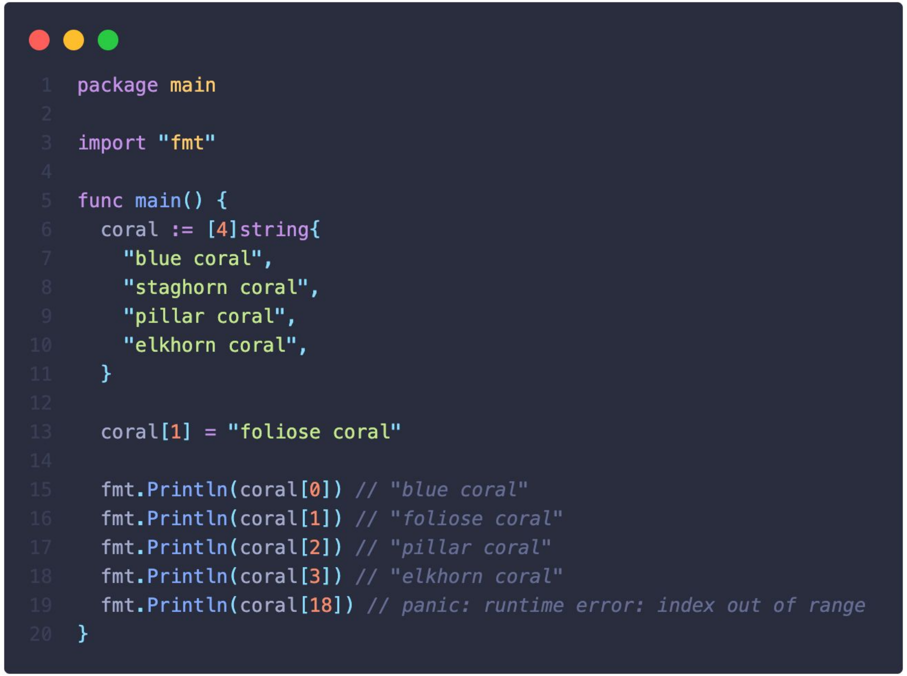
- Arrays are defined by declaring the fixed size of the array in brackets [ ], followed by the data type of the elements.
- An array in Go must have all its elements be the same data type.



```go
package main

import "fmt"

func main() {
  var numbers [3]int
  fmt.Println(numbers) // [0 0 0]

  coral := [4]string{
    "blue coral",
    "staghorn coral",
    "pillar coral",
    "elkhorn coral",
  }
  fmt.Println(coral)
  // [blue coral staghorn coral pillar coral elkhorn coral]
}
```

# Arrays

- You can access array items through its’ discrete index
- We can update individual elements in the array
- Indexes above the capacity of the array will be out of range

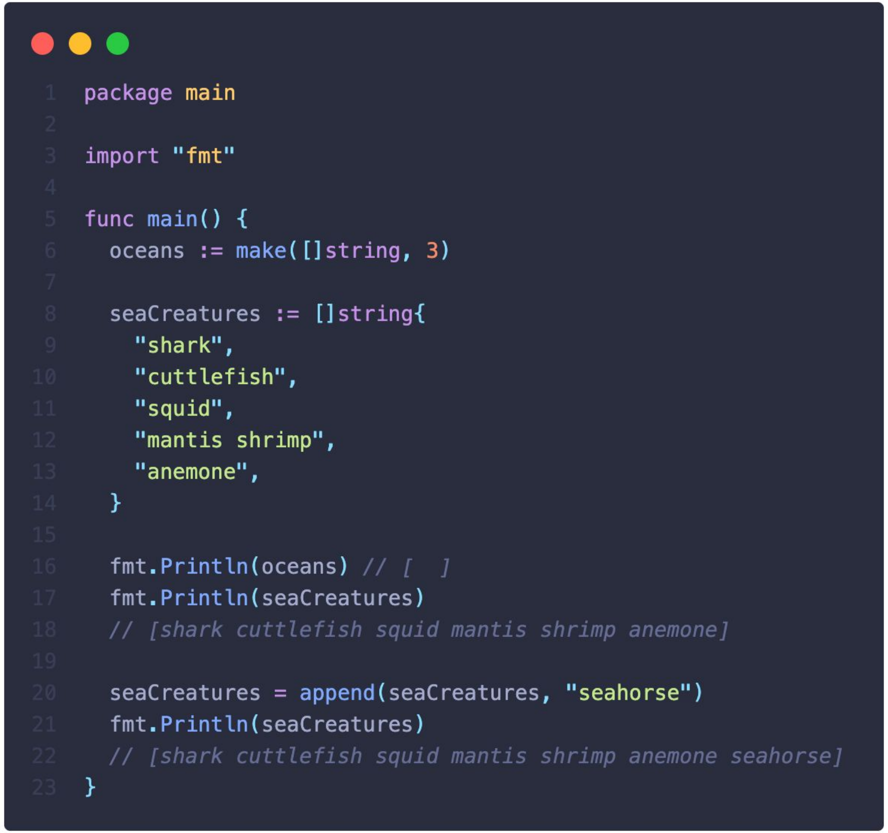


```go
package main

import "fmt"

func main() {
  coral := [4]string{
    "blue coral",
    "staghorn coral",
    "pillar coral",
    "elkhorn coral",
  }

  coral[1] = "foliose coral"

  fmt.Println(coral[0]) // "blue coral"
  fmt.Println(coral[1]) // "foliose coral"
  fmt.Println(coral[2]) // "pillar coral"
  fmt.Println(coral[3]) // "elkhorn coral"
  fmt.Println(coral[18]) // panic: runtime error: index out of range
}
```

# Slices

```
[]data_type{element_values}
```

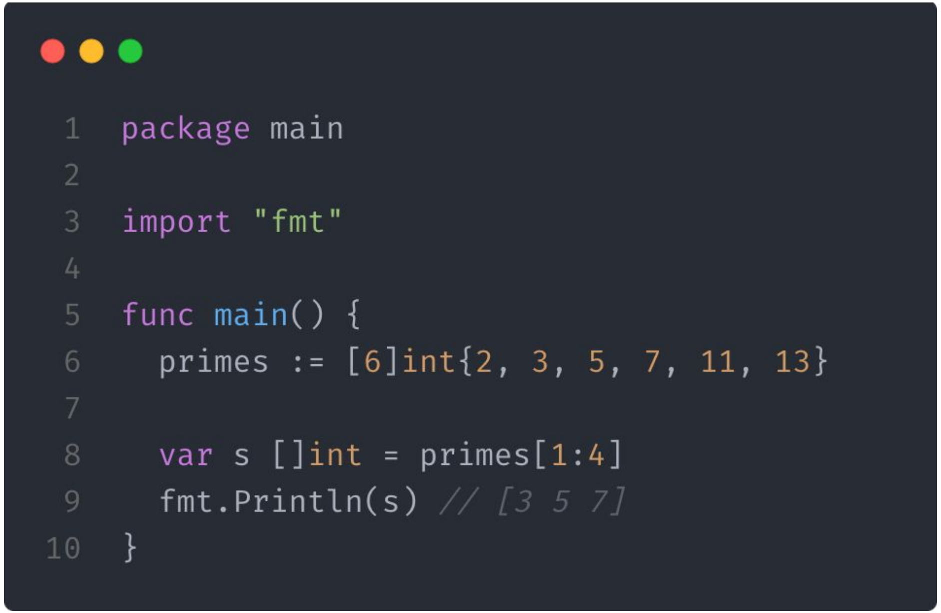
- Slices are like arrays, but do not require a capacity as they are variable in length
- You can instantiate empty slices of a default length
- You can append elements to slices



```go
package main

import "fmt"

func main() {
  oceans := make([]string, 3)

  seaCreatures := []string{
    "shark",
    "cuttlefish",
    "squid",
    "mantis shrimp",
    "anemone",
  }

  fmt.Println(oceans) // [   ]
  fmt.Println(seaCreatures)
  // [shark cuttlefish squid mantis shrimp anemone]

  seaCreatures = append(seaCreatures, "seahorse")
  fmt.Println(seaCreatures)
  // [shark cuttlefish squid mantis shrimp anemone seahorse]
}
```

# Slices

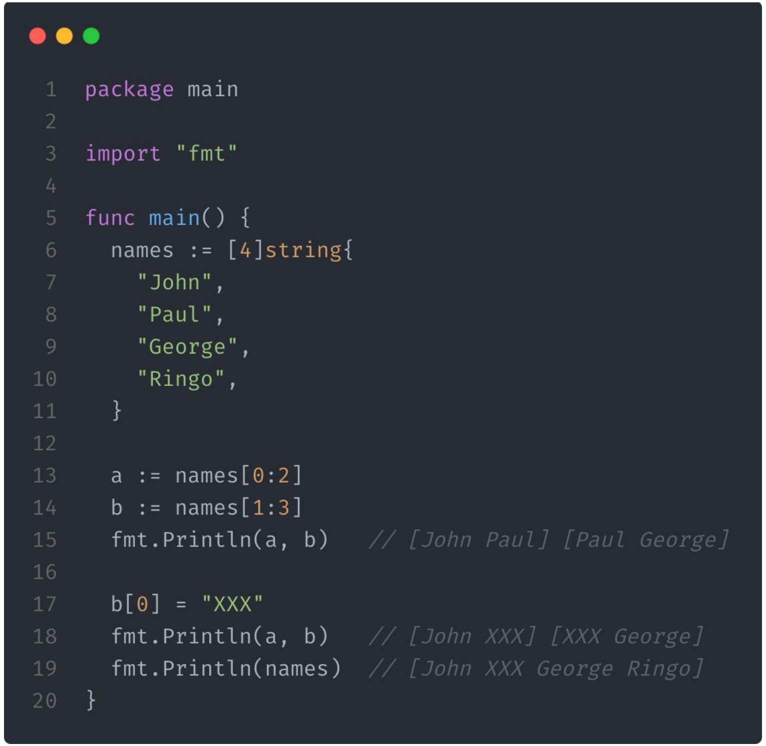
- Another way to initialize a slice is to specify two indices, a low and a high bound, separated by a colon (from an underlying array).



```go
package main

import "fmt"

func main() {
  primes := [6]int{2, 3, 5, 7, 11, 13}

  var s []int = primes[1:4]
  fmt.Println(s) // [3 5 7]
}
```

# Slices are like references to arrays

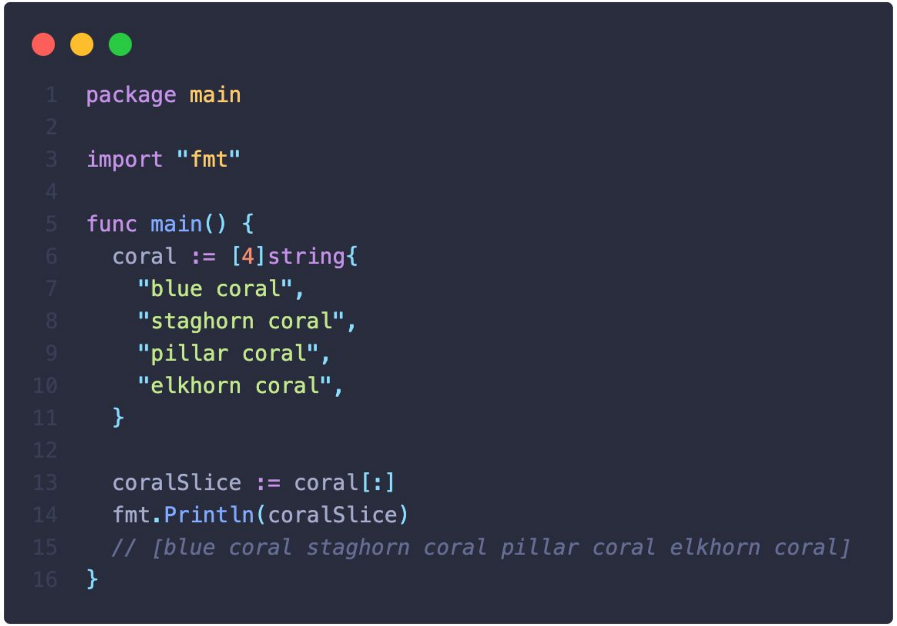
- Changing the elements of a slice modifies the corresponding of its underlying array.



```go
package main

import "fmt"

func main() {
  names := [4]string{
    "John",
    "Paul",
    "George",
    "Ringo",
  }

  a := names[0:2]
  b := names[1:3]
  fmt.Println(a, b)   // [John Paul] [Paul George]

  b[0] = "XXX"
  fmt.Println(a, b)   // [John XXX] [XXX George]
  fmt.Println(names)  // [John XXX George Ringo]
}
```

# Converting Arrays to Slices

- Arrays can be converted to slices for when you decide that you need it to have a variable length
- Use the “:” shorthand to convert the array into a slice
- Operations such as append will work with `coralSlice`

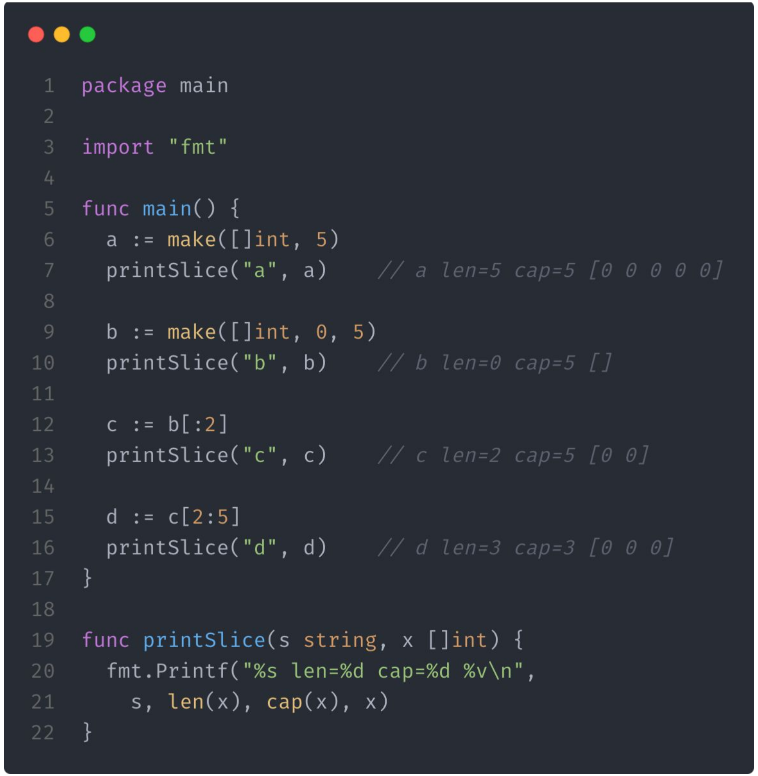


```go
package main

import "fmt"

func main() {
  coral := [4]string{
    "blue coral",
    "staghorn coral",
    "pillar coral",
    "elkhorn coral",
  }

  coralSlice := coral[:]
  fmt.Println(coralSlice)
  // [blue coral staghorn coral pillar coral elkhorn coral]
}
```

# Using make

`make([]T, <len>, (<cap>))`

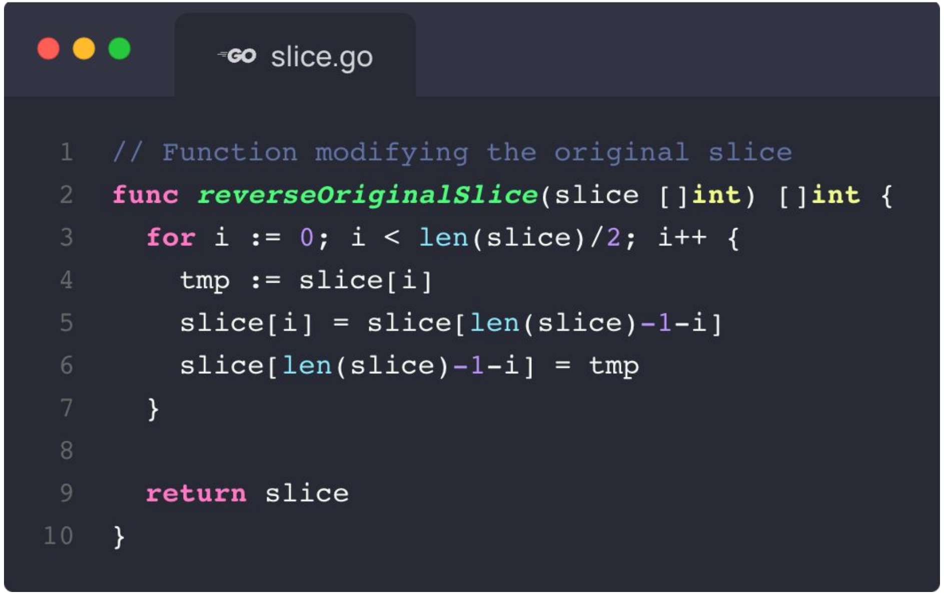
- The make function allocates a zeroed array and returns a slice that refers to that array.
- To specify a capacity, pass a third argument to make



```go
package main

import "fmt"

func main() {
  a := make([]int, 5)
  printSlice("a", a)    // a len=5 cap=5 [0 0 0 0 0]

  b := make([]int, 0, 5)
  printSlice("b", b)    // b len=0 cap=5 []

  c := b[:2]
  printSlice("c", c)    // c len=2 cap=5 [0 0]

  d := c[2:5]
  printSlice("d", d)    // d len=3 cap=3 [0 0 0]
}

func printSlice(s string, x []int) {
  fmt.Printf("%s len=%d cap=%d %v\n",
    s, len(x), cap(x), x)
}
```

# Slices and Memory Management - 1

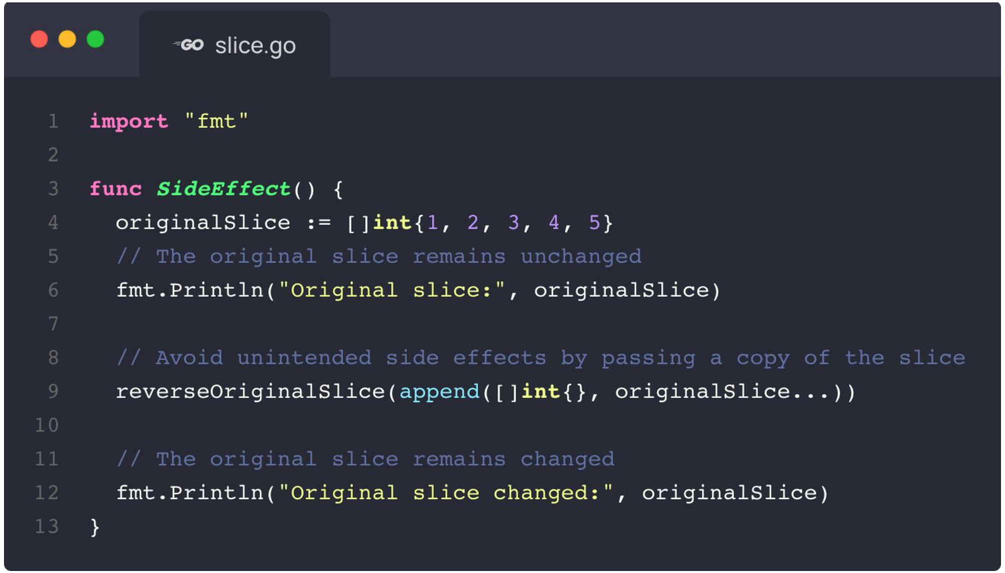
- Slices use a shared underlying array. If multiple slices share the same array, modifying one slice will affect others.
- Care should be taken while passing slices as arguments to functions to avoid unintended **side effects**.



```go
// Function modifying the original slice
func reverseOriginalSlice(slice []int) []int {
  for i := 0; i < len(slice)/2; i++ {
    tmp := slice[i]
    slice[i] = slice[len(slice)-1-i]
    slice[len(slice)-1-i] = tmp
  }

  return slice
}
```

# Slices and Memory Management - 2

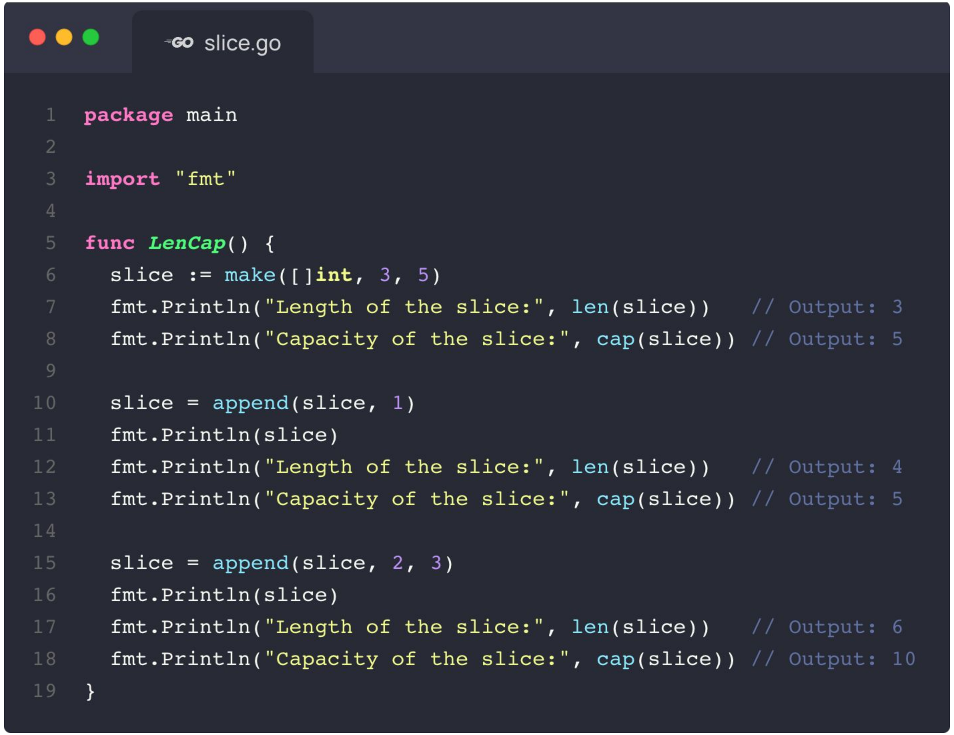
- Avoid unintended side effects by passing a copy of the slice



```go
import "fmt"

func SideEffect() {
  originalSlice := []int{1, 2, 3, 4, 5}
  // The original slice remains unchanged
  fmt.Println("Original slice:", originalSlice)

  // Avoid unintended side effects by passing a copy of the slice
  reverseOriginalSlice(append([]int{}, originalSlice...))

  // The original slice remains changed
  fmt.Println("Original slice changed:", originalSlice)
}
```

# len() and cap()

- The len() function returns the number of elements in the slice.
- The cap() function returns the capacity of the underlying array, starting from the first element in the slice.

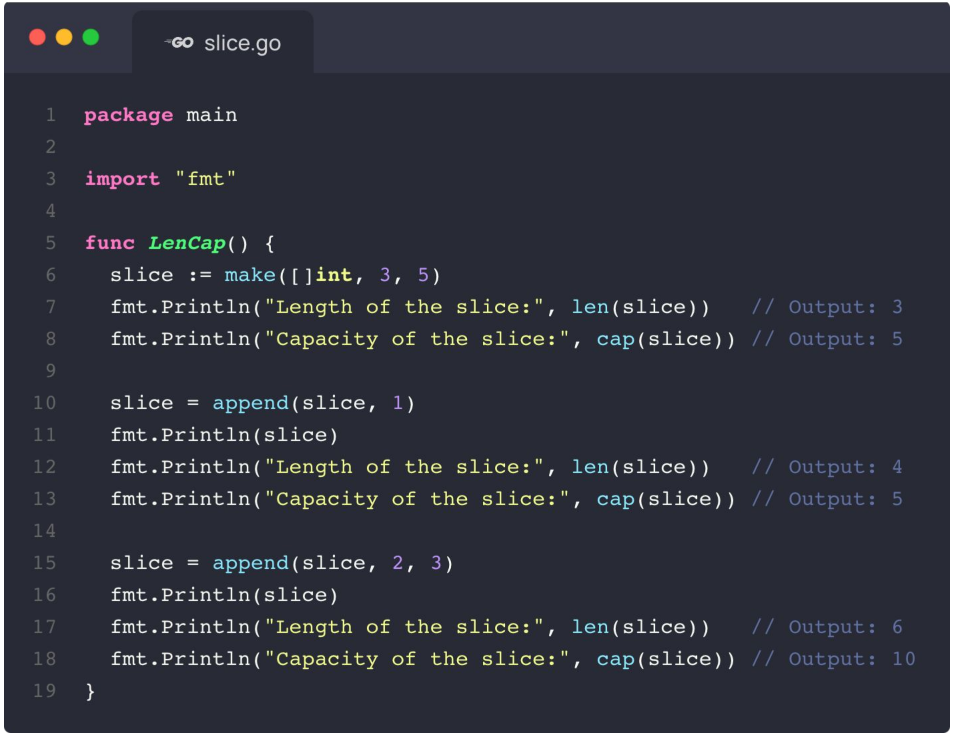


```go
package main

import "fmt"

func LenCap() {
  slice := make([]int, 3, 5)
  fmt.Println("Length of the slice:", len(slice))   // Output: 3
  fmt.Println("Capacity of the slice:", cap(slice)) // Output: 5

  slice = append(slice, 1)
  fmt.Println(slice)
  fmt.Println("Length of the slice:", len(slice))   // Output: 4
  fmt.Println("Capacity of the slice:", cap(slice)) // Output: 5

  slice = append(slice, 2, 3)
  fmt.Println(slice)
  fmt.Println("Length of the slice:", len(slice))   // Output: 6
  fmt.Println("Capacity of the slice:", cap(slice)) // Output: 10
}
```

DWARVES FOUNDATION

# Dynamic Growth of Slices

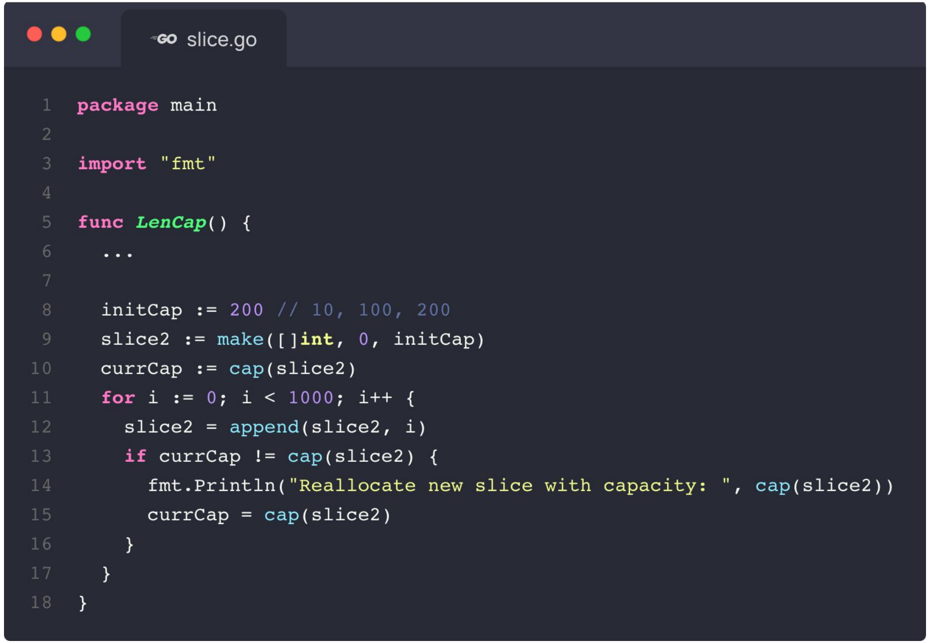
If the underlying array is full, Go will create a new array with increased capacity, copy the elements to the new array, and update the slice reference to the new array.



```go
package main

import "fmt"

func LenCap() {
  slice := make([]int, 3, 5)
  fmt.Println("Length of the slice:", len(slice))   // Output: 3
  fmt.Println("Capacity of the slice:", cap(slice)) // Output: 5

  slice = append(slice, 1)
  fmt.Println(slice)
  fmt.Println("Length of the slice:", len(slice))   // Output: 4
  fmt.Println("Capacity of the slice:", cap(slice)) // Output: 5

  slice = append(slice, 2, 3)
  fmt.Println(slice)
  fmt.Println("Length of the slice:", len(slice))   // Output: 6
  fmt.Println("Capacity of the slice:", cap(slice)) // Output: 10
}
```

# Dynamic Growth of Slices - 2

Understanding capacity is essential for memory optimization, as it helps to minimize the number of array reallocations when appending elements to a slice.

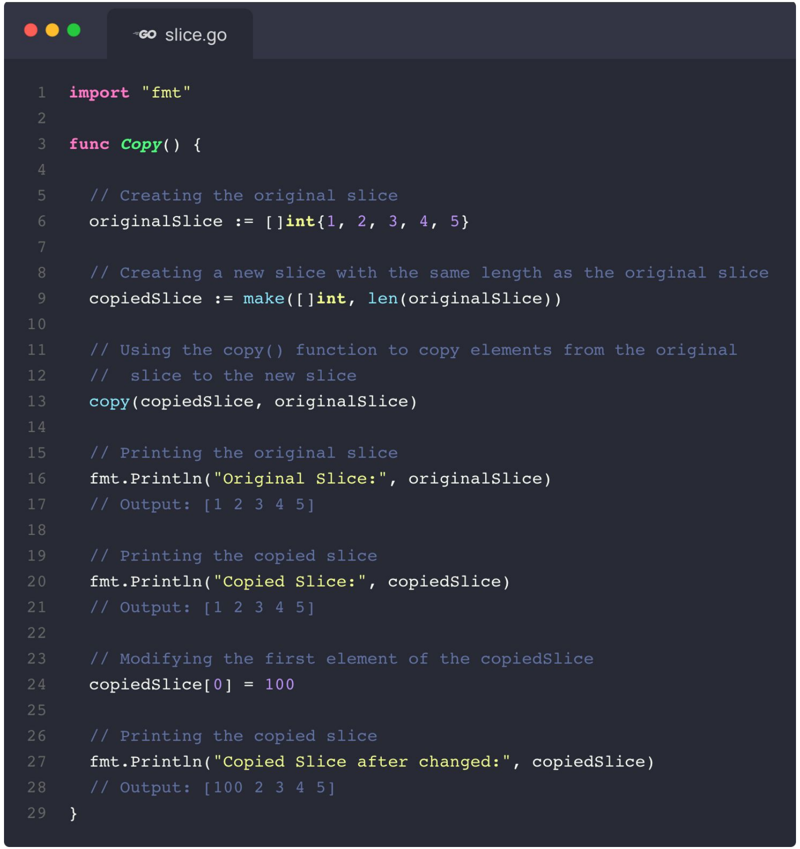


```go
package main

import "fmt"

func LenCap() {
  ...

  initCap := 200 // 10, 100, 200
  slice2 := make([]int, 0, initCap)
  currCap := cap(slice2)
  for i := 0; i < 1000; i++ {
    slice2 = append(slice2, i)
    if currCap != cap(slice2) {
      fmt.Println("Reallocate new slice with capacity: ", cap(slice2))
      currCap = cap(slice2)
    }
  }
}
```

# Using copy

`copy(destSlice, srcSlice)`

- The copy function is used to copy elements from one slice to another
- The length of the destination slice should be equal to or greater than the length of the source slice.
- If the cap(dest) < cap(src), it will allocate additional memory to accommodate the copied elements.

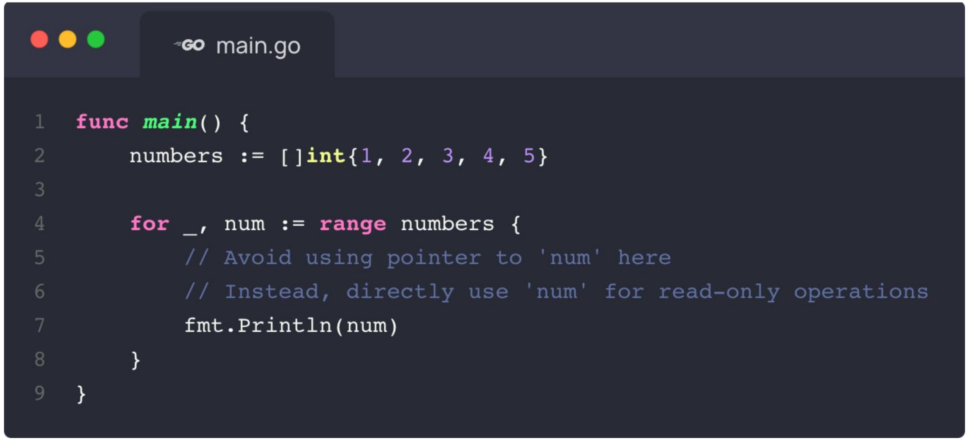


```go
import "fmt"

func Copy() {

  // Creating the original slice
  originalSlice := []int{1, 2, 3, 4, 5}

  // Creating a new slice with the same length as the original slice
  copiedSlice := make([]int, len(originalSlice))

  // Using the copy() function to copy elements from the original
  //  slice to the new slice
  copy(copiedSlice, originalSlice)

  // Printing the original slice
  fmt.Println("Original Slice:", originalSlice)
  // Output: [1 2 3 4 5]

  // Printing the copied slice
  fmt.Println("Copied Slice:", copiedSlice)
  // Output: [1 2 3 4 5]

  // Modifying the first element of the copiedSlice
  copiedSlice[0] = 100

  // Printing the copied slice
  fmt.Println("Copied Slice after changed:", copiedSlice)
  // Output: [100 2 3 4 5]
}
```

# Practices

**`Access item by index`**

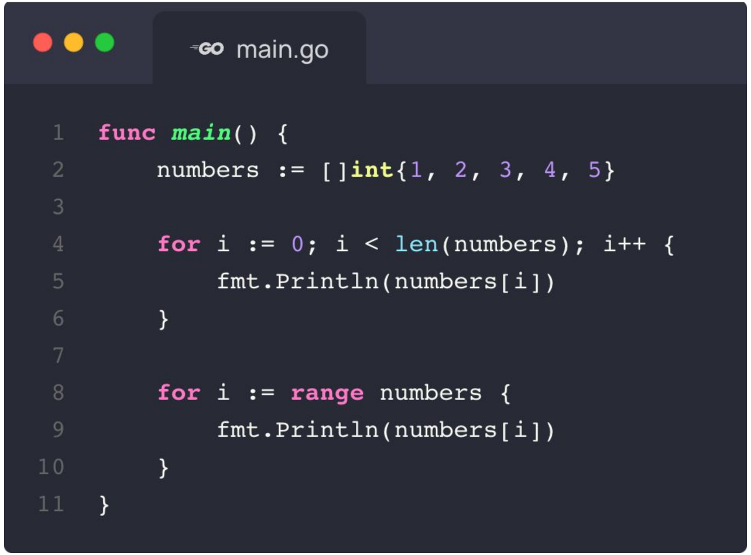
- Loop variables hold the value of the elements
- Pointer to Temporary Copies



```go
func main() {
    numbers := []int{1, 2, 3, 4, 5}

    for _, num := range numbers {
        // Avoid using pointer to 'num' here
        // Instead, directly use 'num' for read-only operations
        fmt.Println(num)
    }
}
```

# Practices

- Using the make()
- Use copy() for Slice Duplication
- Prefer range loop for Iteration
- Avoid pointers from range in Loops
- Append with capacity pre-allocation
- Pass slices by VALUE if Possible
- Use append for removing elements



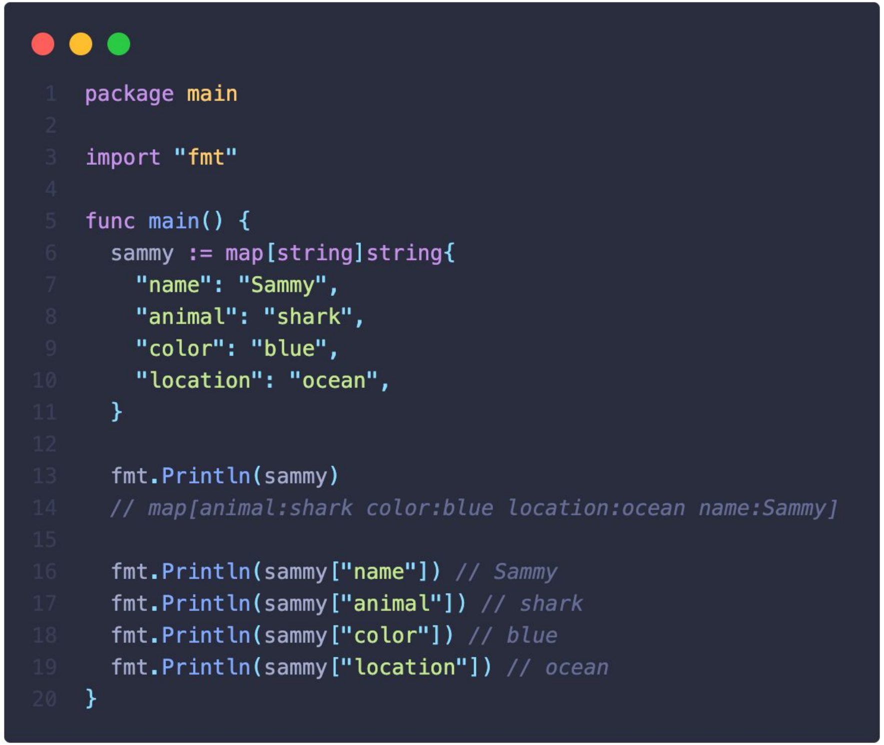
```go
func main() {
    numbers := []int{1, 2, 3, 4, 5}

    for i := 0; i < len(numbers); i++ {
        fmt.Println(numbers[i])
    }

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

# Maps

# Maps

`map[key]value{element_values}`

- Maps are defined by their key data type and value data type
- A map in Go must have all its elements be the same data type.
- Map elements are accessed through their key

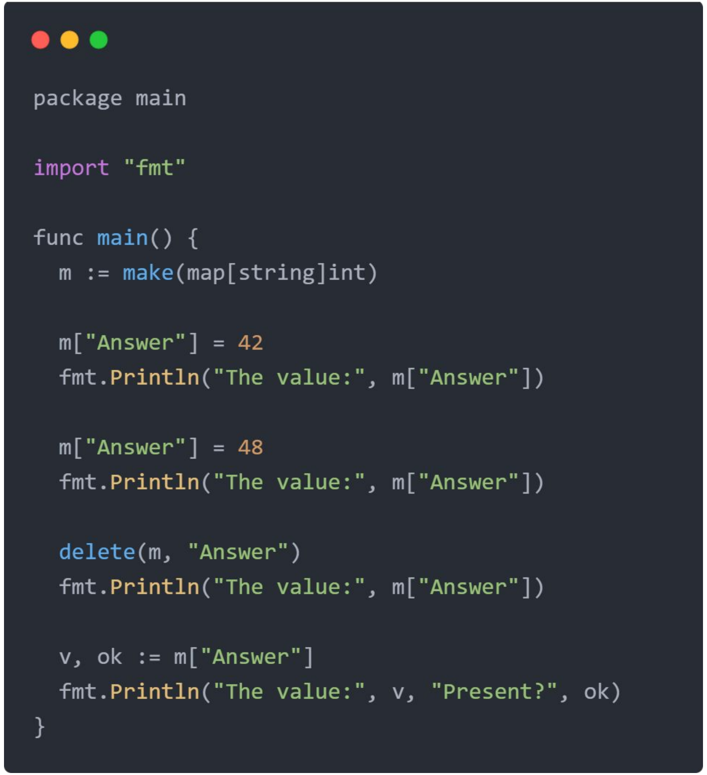


```go
package main

import "fmt"

func main() {
  sammy := map[string]string{
    "name": "Sammy",
    "animal": "shark",
    "color": "blue",
    "location": "ocean",
  }

  fmt.Println(sammy)
  // map[animal:shark color:blue location:ocean name:Sammy]

  fmt.Println(sammy["name"]) // Sammy
  fmt.Println(sammy["animal"]) // shark
  fmt.Println(sammy["color"]) // blue
  fmt.Println(sammy["location"]) // ocean
}
```

# Using make

**`make(map[T]T, (<cap>))`**

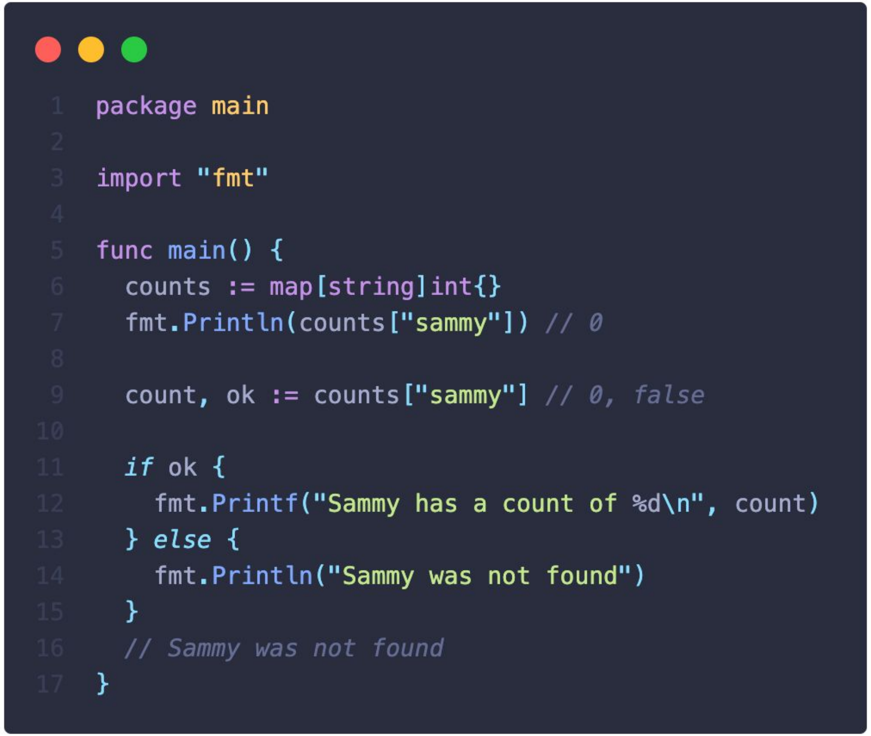
- The make function initializes an empty map that’s ready to use.
- To specify a capacity, pass a second argument to make



```go
package main

import "fmt"

func main() {
  m := make(map[string]int)

  m["Answer"] = 42
  fmt.Println("The value:", m["Answer"])

  m["Answer"] = 48
  fmt.Println("The value:", m["Answer"])

  delete(m, "Answer")
  fmt.Println("The value:", m["Answer"])

  v, ok := m["Answer"]
  fmt.Println("The value:", v, "Present?", ok)
}
```

# Checking if element exists in a map

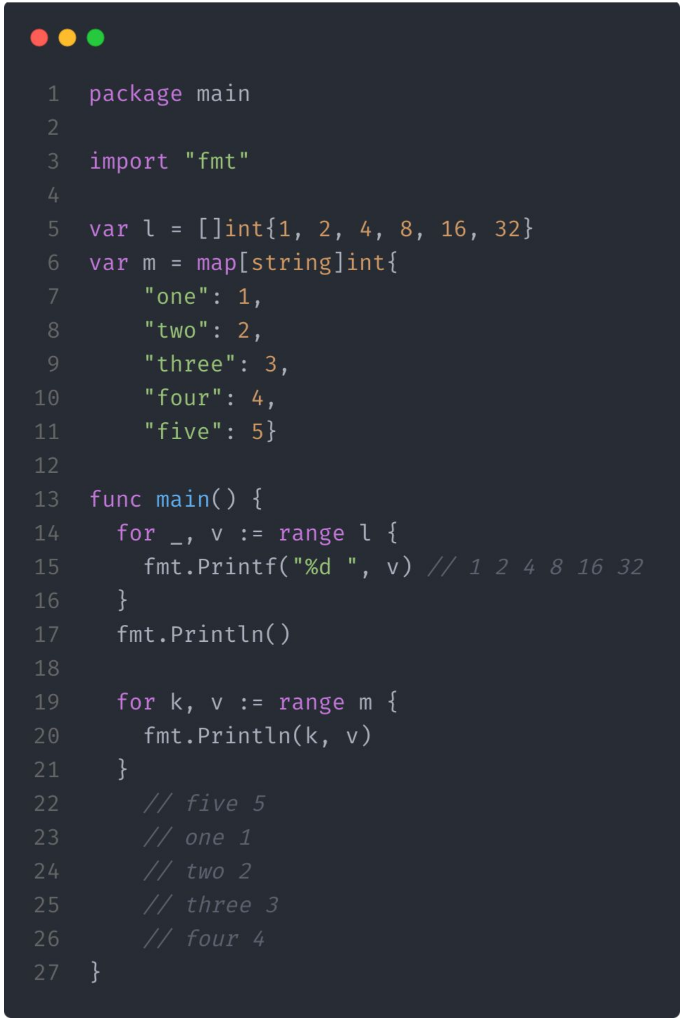
- Maps return a tuple to check the value from the key
- If the value exists, the value will return the element and the boolean will return true
- If the value doesn’t exist, the value will return a default value from the data type and the boolean will return false



```go
package main

import "fmt"

func main() {
  counts := map[string]int{}
  fmt.Println(counts["sammy"]) // 0

  count, ok := counts["sammy"] // 0, false

  if ok {
    fmt.Printf("Sammy has a count of %d\n", count)
  } else {
    fmt.Println("Sammy was not found")
  }
  // Sammy was not found
}
```

# Range

- Use range form of for loop to iterate over a slice or map
- Loop through a map does NOT guarantee the order of it

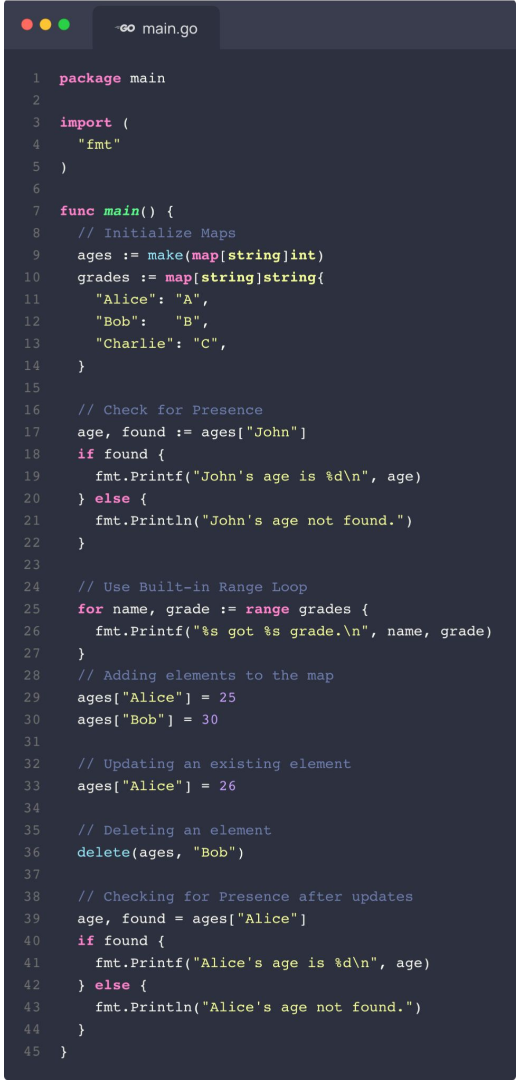


```go
package main

import "fmt"

var l = []int{1, 2, 4, 8, 16, 32}
var m = map[string]int{
    "one": 1,
    "two": 2,
    "three": 3,
    "four": 4,
    "five": 5}

func main() {
  for _, v := range l {
    fmt.Printf("%d ", v) // 1 2 4 8 16 32
  }
  fmt.Println()

  for k, v := range m {
    fmt.Println(k, v)
  }
    // five 5
    // one 1
    // two 2
    // three 3
    // four 4
}
```

DWARVES FOUNDATION

# Practices

- Using the make()
- Don't Assume Order
- Use Built-in Range Loop
- Avoid Pointers to Maps

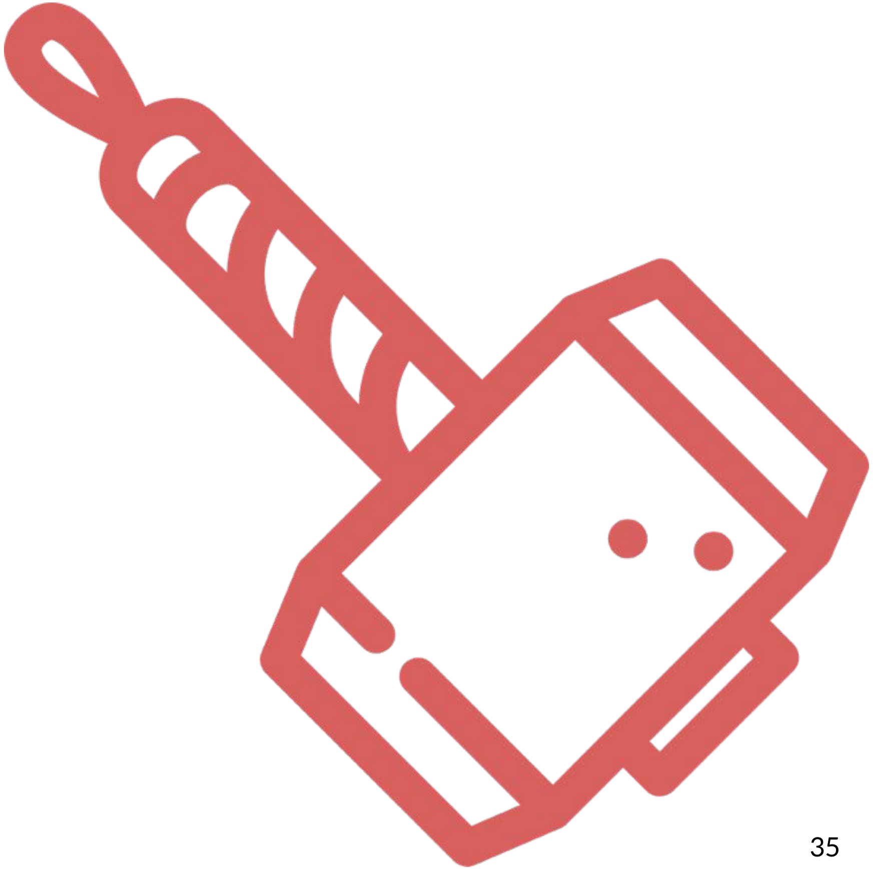


```go
package main

import (
  "fmt"
)

func main() {
  // Initialize Maps
  ages := make(map[string]int)
  grades := map[string]string{
    "Alice": "A",
    "Bob":    "B",
    "Charlie": "C",
  }

  // Check for Presence
  age, found := ages["John"]
  if found {
    fmt.Printf("John's age is %d\n", age)
  } else {
    fmt.Println("John's age not found.")
  }

  // Use Built-in Range Loop
  for name, grade := range grades {
    fmt.Printf("%s got %s grade.\n", name, grade)
  }
  // Adding elements to the map
  ages["Alice"] = 25
  ages["Bob"] = 30

  // Updating an existing element
  ages["Alice"] = 26

  // Deleting an element
  delete(ages, "Bob")

  // Checking for Presence after updates
  age, found = ages["Alice"]
  if found {
    fmt.Printf("Alice's age is %d\n", age)
  } else {
    fmt.Println("Alice's age not found.")
  }
}
```

DWARVES
FOUNDATION

# Demo - Zer0 to Hero

- New a server using gin
- Implement auth service



# Assignments

# Assignment 1

Goal: Find and count number of rectangles in a 2D array.

Inputs: An array filled with 0s and 1s.

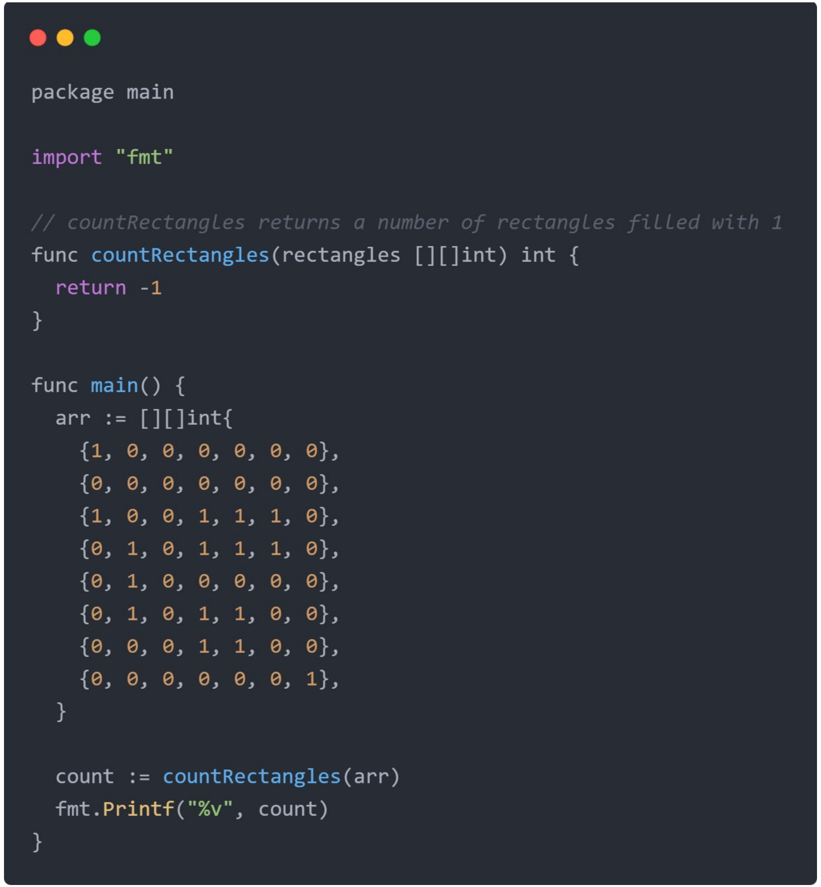
Outputs: Number of rectangles filled with 1s

Given that rectangles are separated and do not touch each other but they can touch the boundary of the array. A single element rectangle counts.

# Assignment 1

Implement countRectangles to return a number of rectangles filled with 1s.

- Each cell is a rectangle '1' or empty '0', return the number of the rectangles on board.
- Each rectangle can be made in the shape of `1*1`, `1*n`, `m*1`, or `m*n` (m rows, n columns)
- There are no adjacent rectangles



```go
package main

import "fmt"

// countRectangles returns a number of rectangles filled with 1
func countRectangles(rectangles [][]int) int {
  return -1
}

func main() {
  arr := [][]int{
    {1, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 1, 1, 1, 0},
    {0, 1, 0, 1, 1, 1, 0},
    {0, 1, 0, 0, 0, 0, 0},
    {0, 1, 0, 1, 1, 0, 0},
    {0, 0, 0, 1, 1, 0, 0},
    {0, 0, 0, 0, 0, 0, 1},
  }

  count := countRectangles(arr)
  fmt.Printf("%v", count)
}
```

DWARVES FOUNDATION

# Assignment 2

Goal: Count the number of different integers in a String.

Outputs: Number of different integers

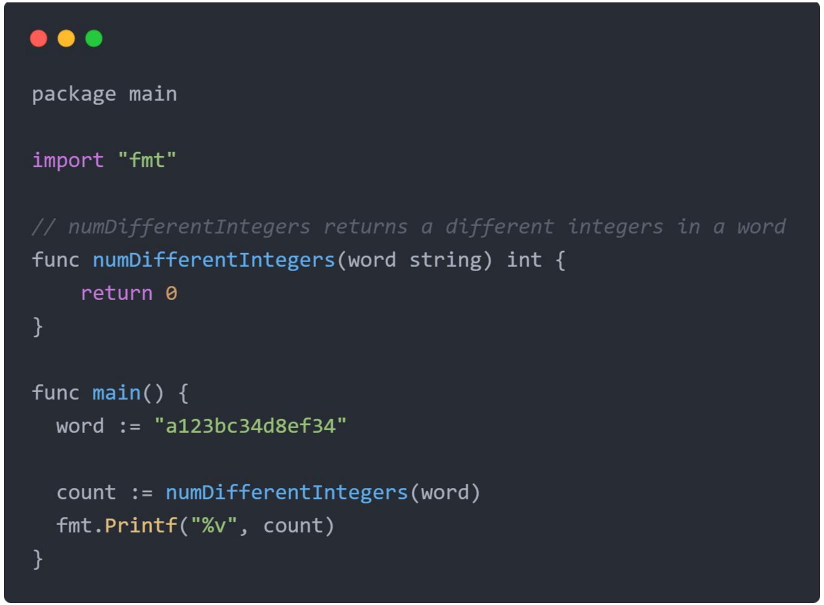
Given that a string word consists of digits and lowercase English letters, 2 integers are considered different if their decimal representation without any leading zeros are different.

E.g: “a123bc34d8ef34” => 3 (123, 34, 8)

"A1b01c001" => 1 (1)

# Assignment 2

Implement numDifferentIntegers to return a number of different integers in a word.



```go
package main

import "fmt"

// numDifferentIntegers returns a different integers in a word
func numDifferentIntegers(word string) int {
    return 0
}

func main() {
  word := "a123bc34d8ef34"

  count := numDifferentIntegers(word)
  fmt.Printf("%v", count)
}
```

# Recap

**Interface & struct**

**Pointers**: hold a memory address of a value. Just like C, except no arithmetic.

**Arrays**: a numbered sequence of elements of a single type.

**Slices**: reference to underlying arrays.

**Maps**: an unordered group of elements of one type, indexed by a set of unique keys of another type.

**Ranges**: use range form of for loop to iterate through a array/slice/map

# Reference

Resources & Reference links

- https://www.digitalocean.com/community/tutorials/understanding-arrays-and-slices-in-go
- https://www.digitalocean.com/community/tutorials/understanding-maps-in-go
- https://www.go.dev/tour

DWARVES
FOUNDATION

# Thank You

DWARVES
FOUNDATION

# Q&A