Introducing Go

Learning Outcomes

After completing this session you should be able to

- Describe **basic features of Go** programming language
- Explain and use the **numeric**, **string**, **boolean data types**
- Store numeric and text data using **variables** and **constants**
- Use **arithmetic**, **boolean**, **comparison**, **logic** and **concatenation operators**
- Write go programs using **for**, **if** and **switch control statements** and **functions**

Learning Outcomes

After completing this session you should be able to

Explain and use **recursion functions**, **variadic functions** and **closures**

Explain and use **Arrays**, **Slices** and **Maps**

Explain and use **Pointers**, **Structs**, and **Interfaces**

Go Language

Go is an **open source** programming language that makes it easy to build **simple**, **reliable**, and **efficient** software

Go was conceived in September 2007 by **Robert Griesemer**, **Rob Pike**, and **Ken Thompson**, all at Google, and was announced in November 2009

The goals of the language and its accompanying tools were to be **expressive**, **efficient** in **both compilation and execution**, and **effective** in **writing reliable and robust programs**

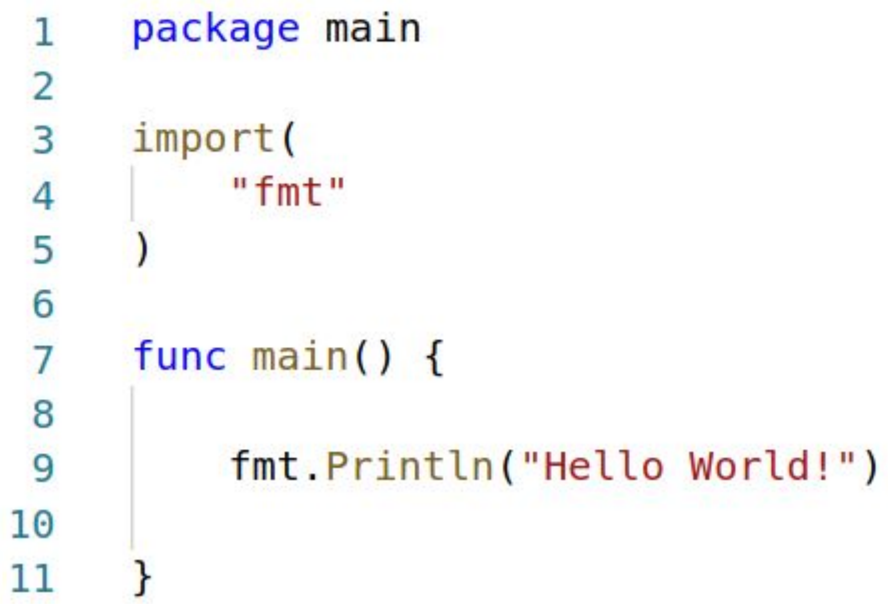
Go Language

Go borrows and adapts good ideas from many other languages, while avoiding features that have led to complexity and unreliable code

Its facilities for **concurrency** are new and efficient, and its approach to data abstraction and object-oriented programming is unusually flexible.

It has automatic memory management or garbage collection

Hello, World



```go
package main

import(
	"fmt"
)

func main() {

	fmt.Println("Hello World!")

}
```

Store the program in example.go file and run the following command

```
go run example.go

OR

go build example.go

./example
```

Source code structure

```go
package main
```

Package declaration

```go
import(
    "fmt"
)
```

Package Import Declaration

```go
func main() {

    fmt.Println("Hello World!")

}
```

Program body

Packages

Go code is organized into packages, which are similar to libraries or modules in other languages

A package consists of one or more **.go** source files in a single directory that define what the package does

The **Go standard library** has over 100 packages for common tasks like input and output, sorting, and text manipulation (example **`fmt`** package)

Package main is special. It defines a stand alone executable program, not a library

Main Function

Function main is special: it's where execution of the program begins.

Comments

Single Line comment

```
//
```

Multiline Comments

```
/*
 */
```

Types

Go is a **statically typed** programming language. This means that variables always have a specific type and that type cannot change

Types

Integers

`uint8` , `uint16` , `uint32` , `uint64` , `int8` , `int16` , `int32` , and `int64`

8, **16**, **32**, and **64** tell us how many bits each of the types use

`uint` means “**unsigned integer**” while `int` means “**signed integer**”

Two Aliases

`byte` which is the same as `uint8` and

`rune` which is the same as `int32`

Types

Integers

Three Machine dependent integer types

`uint`, `int`, and `uintptr`

Their size depends on the type of architecture you are using

They are usually 32 bits wide on 32-bit systems and 64 bits wide on 64-bit systems.

When you need an integer value you should use int unless you have a specific reason to use a sized or unsigned integer type

Types

Floating-Point Numbers

- **Single Precision:** `float32`
- **Double Precision:** `float64`

Complex Numbers

- `complex64` and `complex128`

Types

Strings

A sequence of characters with a definite length used to represent text

Go strings are made up of individual bytes, usually one for each character

Characters from non-english languages, such as Amharic, are represented by more than one byte

Common operations on string

```
len("Hello, World") "Hello, World"[1] "Hello, " + "World"
```

Types

Booleans

A boolean value represents `true` and `false` (or on and off) values

Three logical operators are used with boolean values

`&&`	**and**
`||`	**or**
`!`	**not**

Operators

Arithmetic

`+` `*` `-` `/` `%`

String Concatenation

`+`

Logical

`&&` `||` `!`

Comparison

`<` `>` `==` `!=`

`<=` `>=`

Exercise

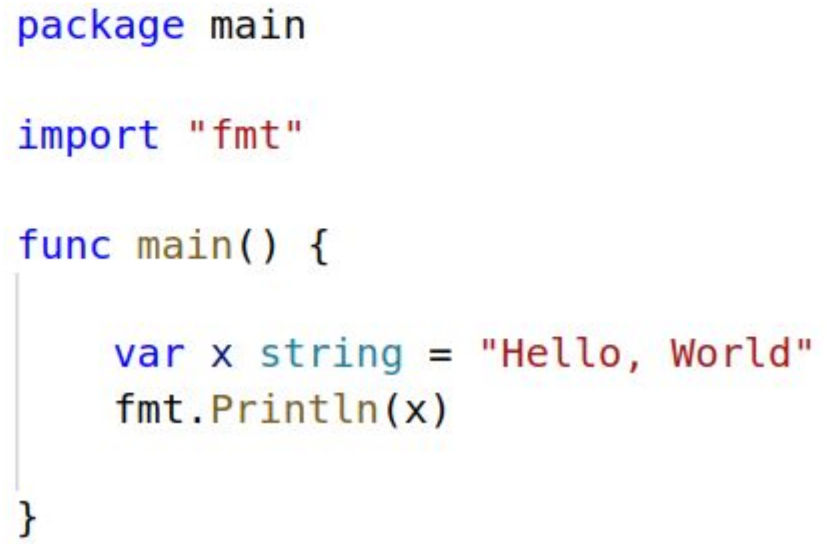
Write a program that

1. Perform arithmetic operations on two integer, real and complex numbers then print the result
2. Concatenate two string literals then print the result
3. Compare two numeric values and print the result

Variables

A **variable** is a storage location, with a specific type and an associated name

Variables are created by **first using the var keyword**, then **specifying the variable name (x)** and the **type (string)**



```
package main

import "fmt"

func main() {

    var x string = "Hello, World"
    fmt.Println(x)

}
```

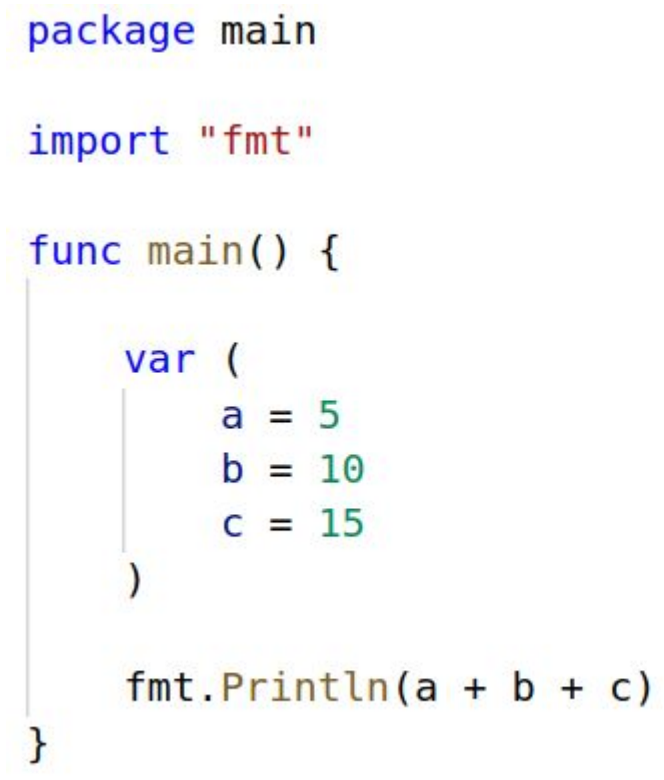
Variables

Variable declaration variants

```
var x string = "hello"  // initialization with type

var x = "hello"      // initialization without type

var x string         // declaration, with “” initial value
    x = "hello"      // assignment

x := "hello"      // no var key work

x, y, z := 2.3, 7, “itsc”  // initi.. multiple variables
```

Defining Multiple Variables/Constants

Use the keyword **`var (`** or **`const )`** followed by parentheses with each variable on its own line



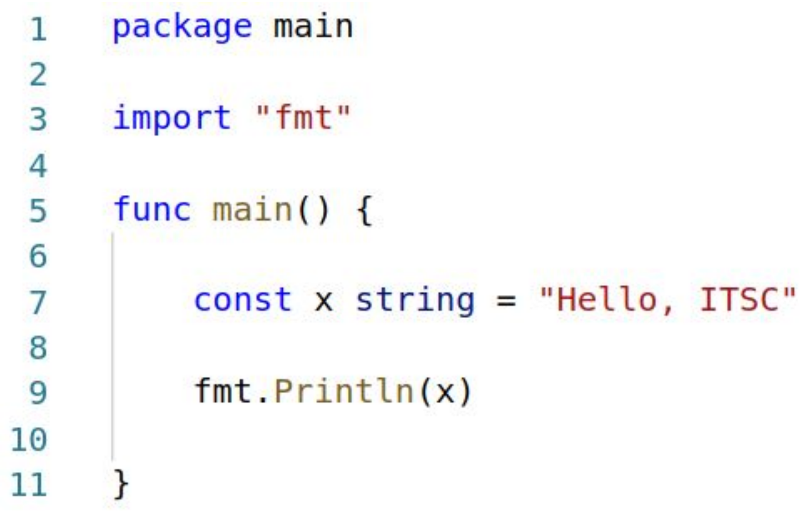
```go
package main

import "fmt"

func main() {

    var (
        a = 5
        b = 10
        c = 15
    )

    fmt.Println(a + b + c)
}
```

Constants

Constants are essentially variables whose values cannot be changed later.

They are created in the same way you create variables, but instead of **using** the var keyword we use the `const` **keyword**:

Constants cannot be declared using the `:=` syntax.



```go
package main

import "fmt"

func main() {
    const x string = "Hello, ITSC"
    
    fmt.Println(x)
    
}
```

Type conversions

The expression **T(v)** converts the value **v** to the type **T**.

*Assignment between items of different type requires an **explicit** conversion*

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

OR

```
i := 42
f := float64(i)
u := uint(f)
```

Exercise

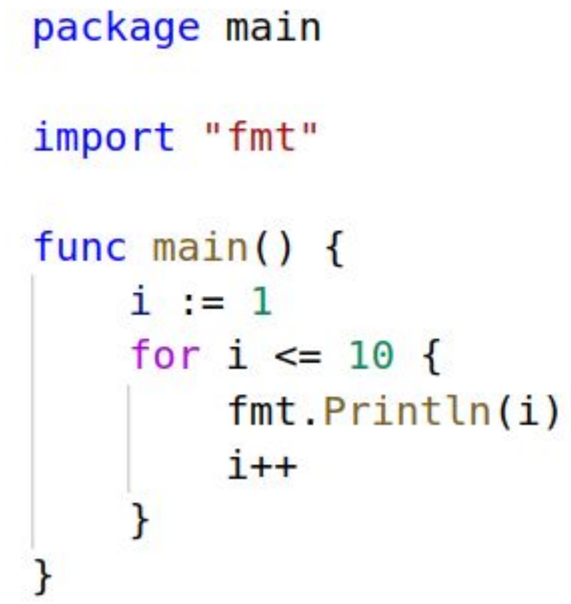
Write a program that compute area of a circle

Write a program that converts from Fahrenheit into Celsius

```
    (C = (F - 32) * 5/9)
```

Control Structures

Looping: The `for` Statement



```go
package main

import "fmt"

func main() {
    i := 1
    for i <= 10 {
        fmt.Println(i)
        i++
    }
}
```

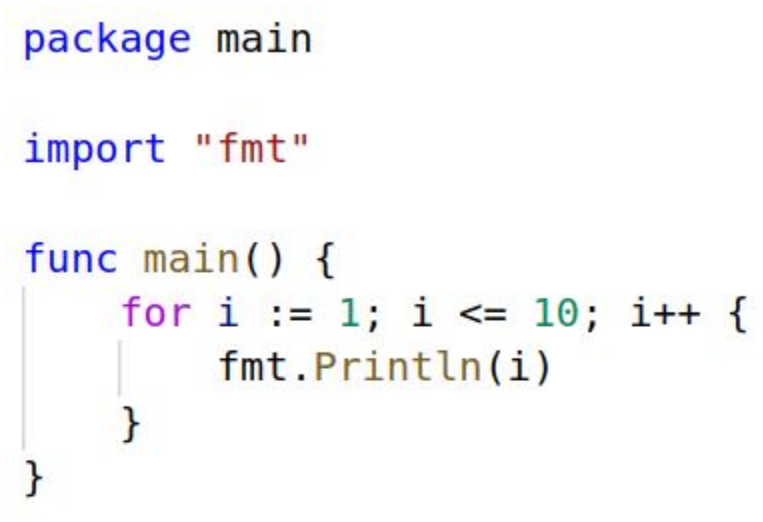
Control Structures

Looping: The `for` Statement

The **init** statement: executed before the first iteration (`i := 1`)

The **condition** expression: evaluated before every iteration (`i <= 10`)

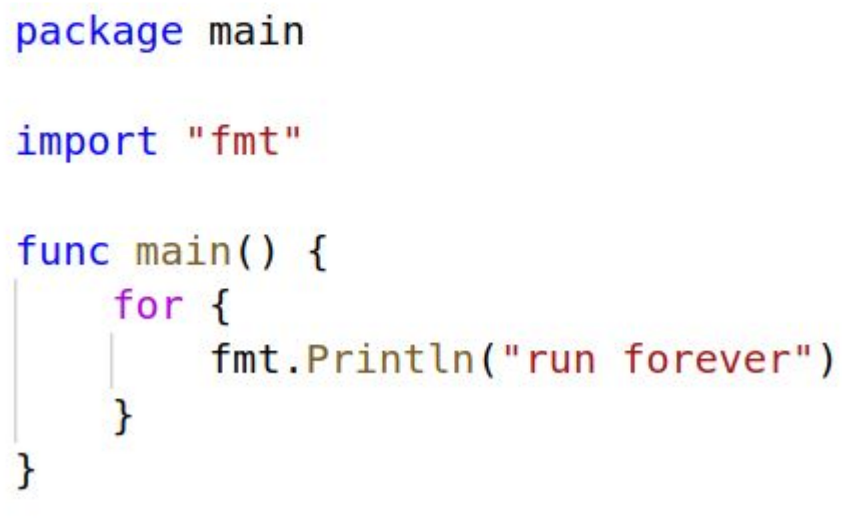
The **post** statement: executed at the end of every iteration (`i++`)



```go
package main

import "fmt"

func main() {
	for i := 1; i <= 10; i++ {
		fmt.Println(i)
	}
}
```

Control Structures

Looping: The `for` Statement

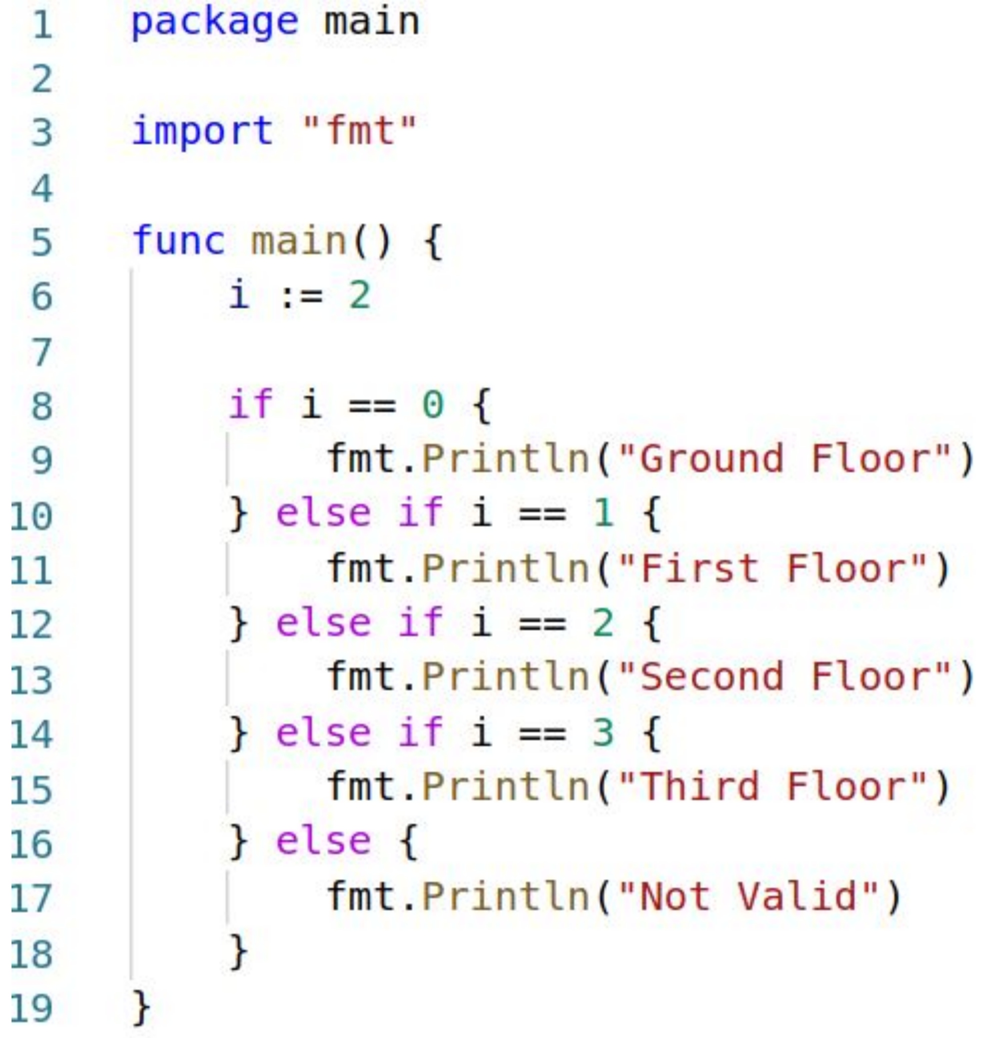


```go
package main

import "fmt"

func main() {
	for {
		fmt.Println("run forever")
	}
}
```

Control Structures

Conditional: The `if` Statement

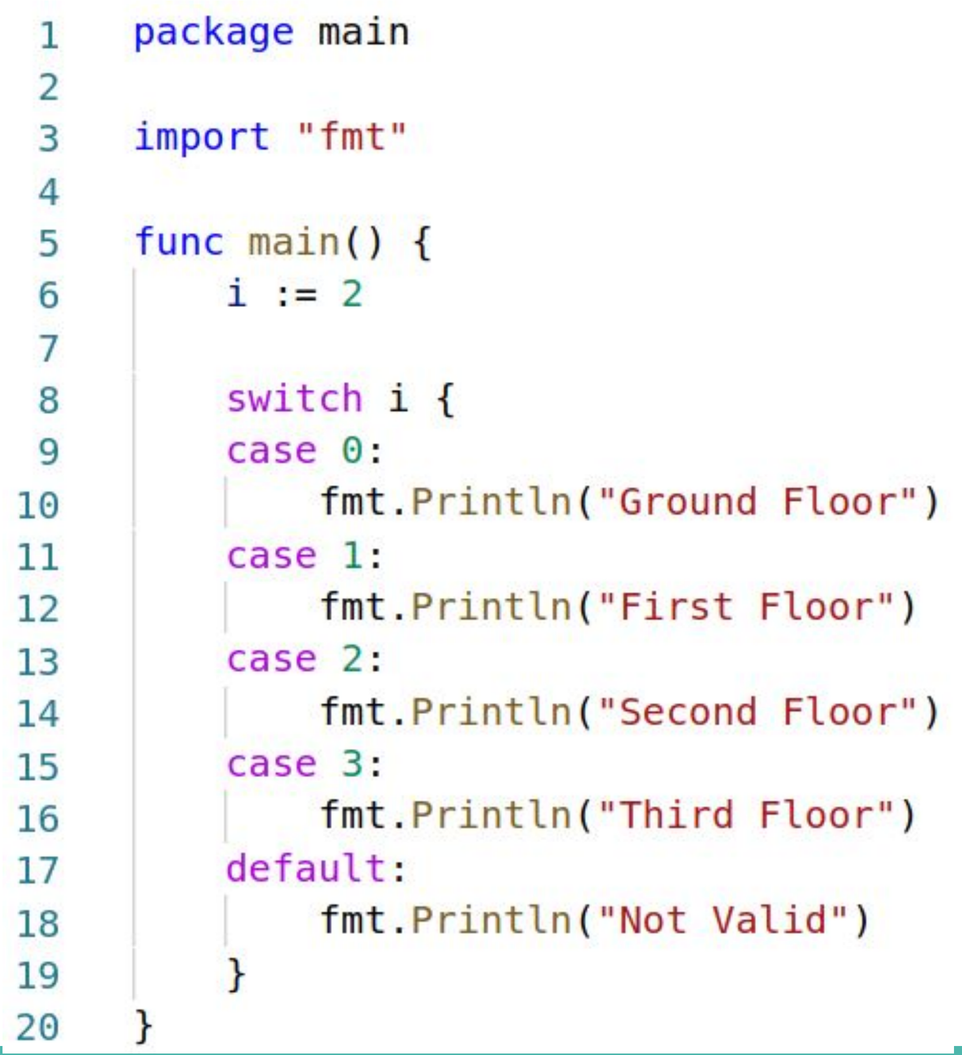
Notice that there the condition is not enclosed in brackets



```go
package main

import "fmt"

func main() {
    i := 2

    if i == 0 {
        fmt.Println("Ground Floor")
    } else if i == 1 {
        fmt.Println("First Floor")
    } else if i == 2 {
        fmt.Println("Second Floor")
    } else if i == 3 {
        fmt.Println("Third Floor")
    } else {
        fmt.Println("Not Valid")
    }
}
```

Control Structures

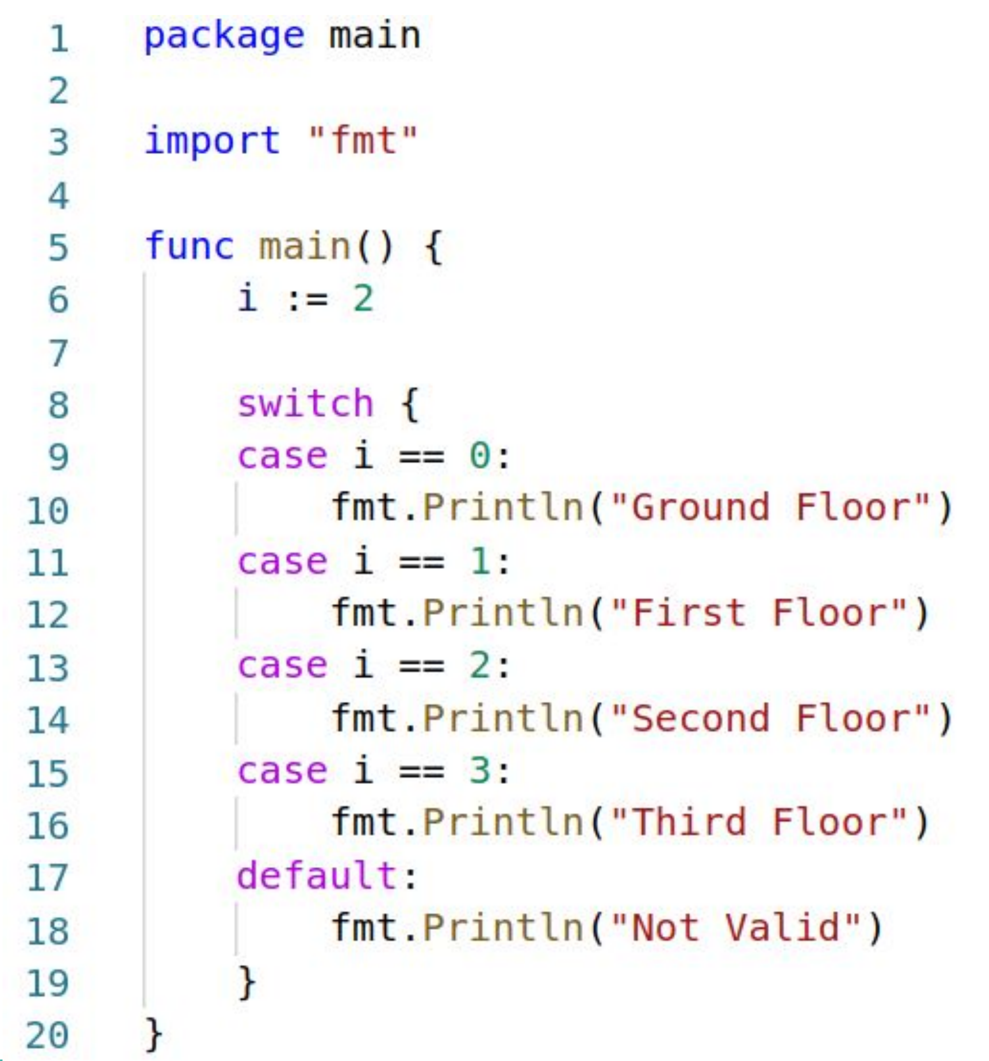
Conditional: The **`switch`** Statement



```go
package main

import "fmt"

func main() {
	i := 2

	switch i {
	case 0:
		fmt.Println("Ground Floor")
	case 1:
		fmt.Println("First Floor")
	case 2:
		fmt.Println("Second Floor")
	case 3:
		fmt.Println("Third Floor")
	default:
		fmt.Println("Not Valid")
	}
}
```

Control Structures

Conditional: The `switch` Statement



```go
package main

import "fmt"

func main() {
	i := 2

	switch {
	case i == 0:
		fmt.Println("Ground Floor")
	case i == 1:
		fmt.Println("First Floor")
	case i == 2:
		fmt.Println("Second Floor")
	case i == 3:
		fmt.Println("Third Floor")
	default:
		fmt.Println("Not Valid")
	}
}
```

Exercise

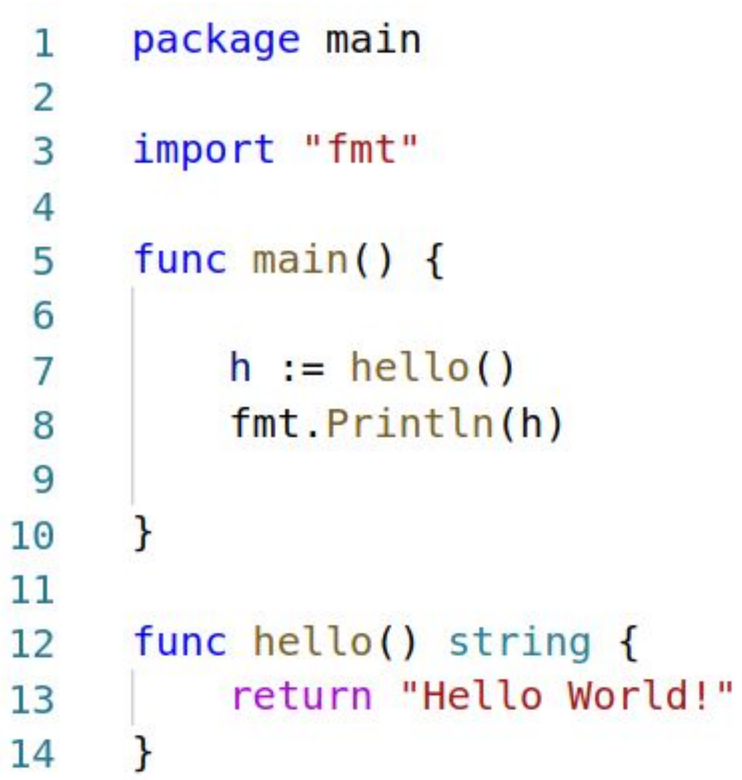
Write a program that prints out all the numbers between 1 and 100 that are evenly divisible by 3 (i.e., 3, 6, 9, etc.)

Functions

Use the key work func to define a function

The function return type is specified after the function parameters

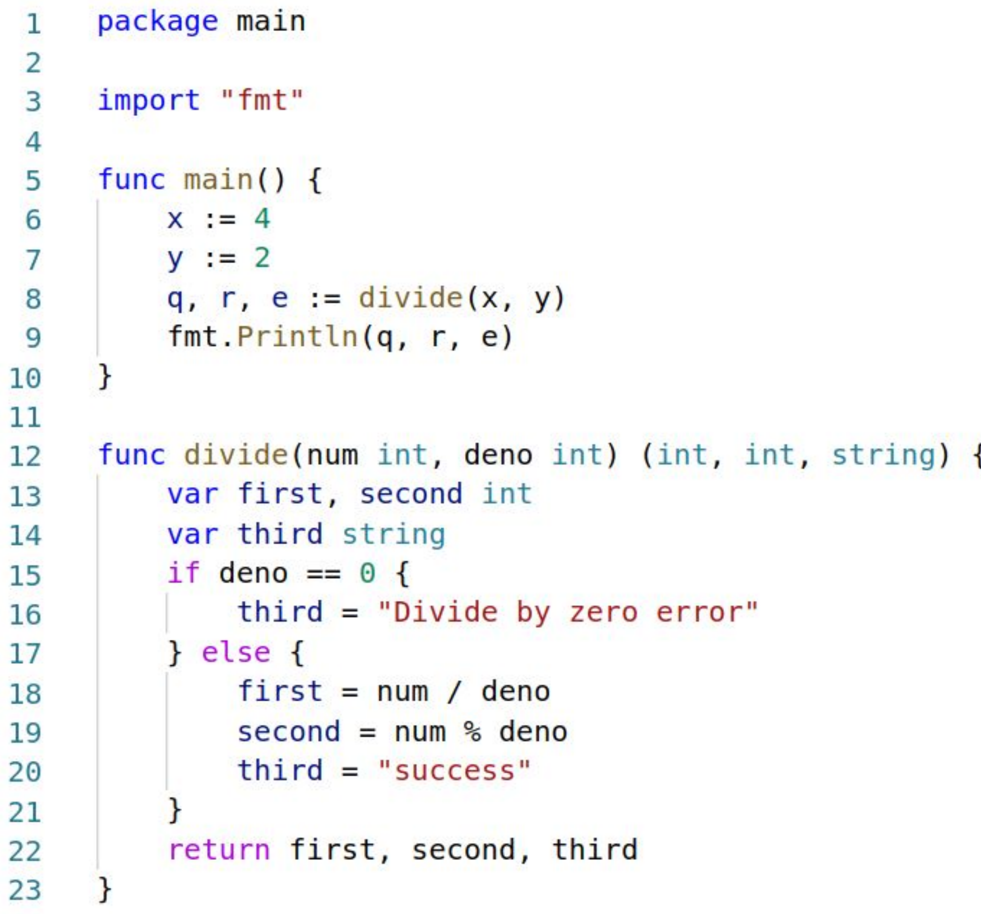
Use the key work return to return some value



```go
package main

import "fmt"

func main() {

	h := hello()
	fmt.Println(h)

}

func hello() string {
	return "Hello World!"
}
```

Functions

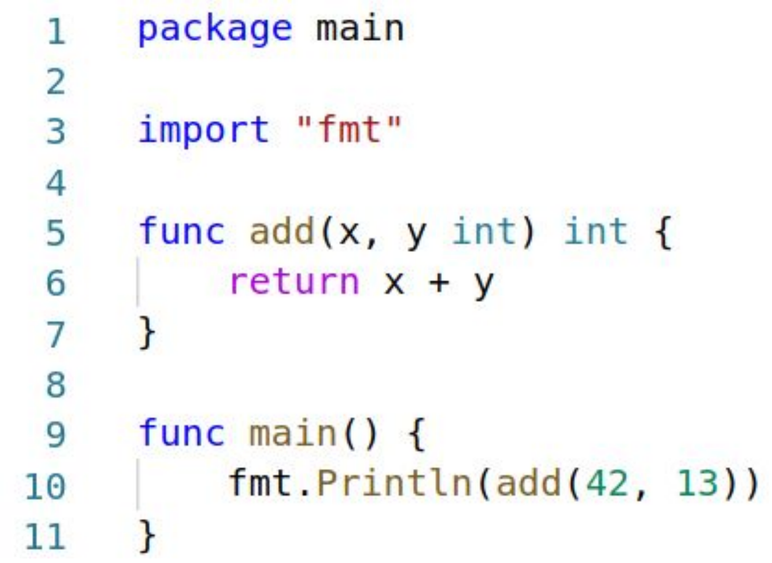
Functions can return multiple values



```go
package main

import "fmt"

func main() {
    x := 4
    y := 2
    q, r, e := divide(x, y)
    fmt.Println(q, r, e)
}

func divide(num int, deno int) (int, int, string) {
    var first, second int
    var third string
    if deno == 0 {
        third = "Divide by zero error"
    } else {
        first = num / deno
        second = num % deno
        third = "success"
    }
    return first, second, third
}
```

Functions

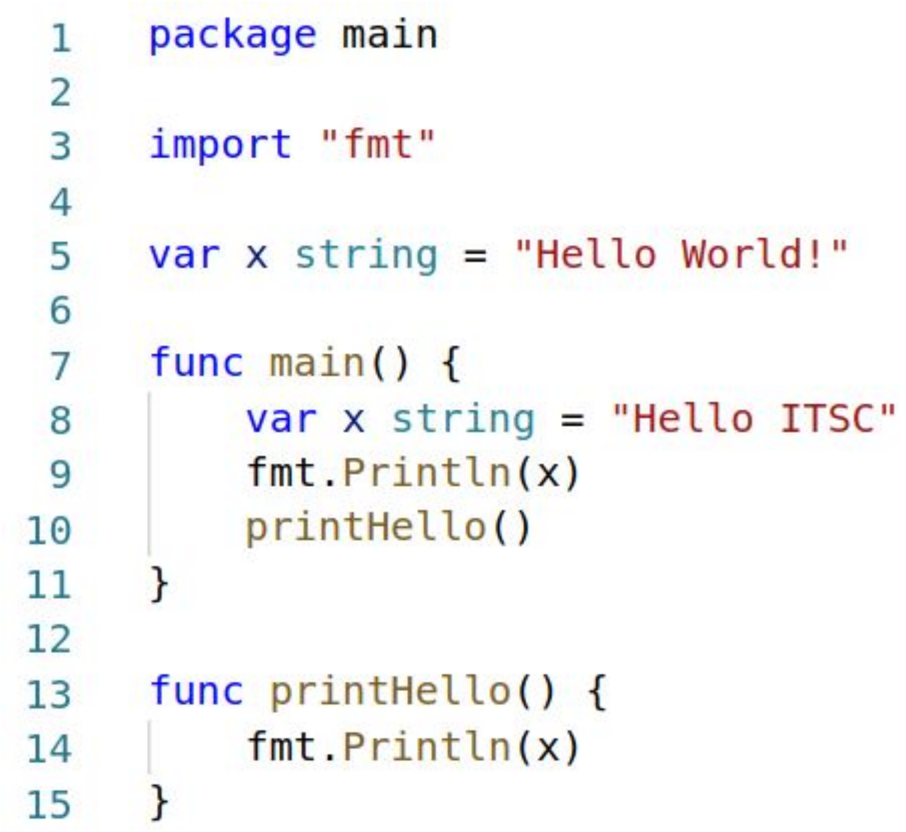
When two or more consecutive named function parameters share a type, you can omit the type from all but the last



```go
package main

import "fmt"

func add(x, y int) int {
	return x + y
}

func main() {
	fmt.Println(add(42, 13))
}
```

Variable Scope

What is the out of this program?



```go
package main

import "fmt"

var x string = "Hello World!"

func main() {
	var x string = "Hello ITSC"
	fmt.Println(x)
	printHello()
}

func printHello() {
	fmt.Println(x)
}
```

Variadic Functions

`add` function can accept any number of arguments

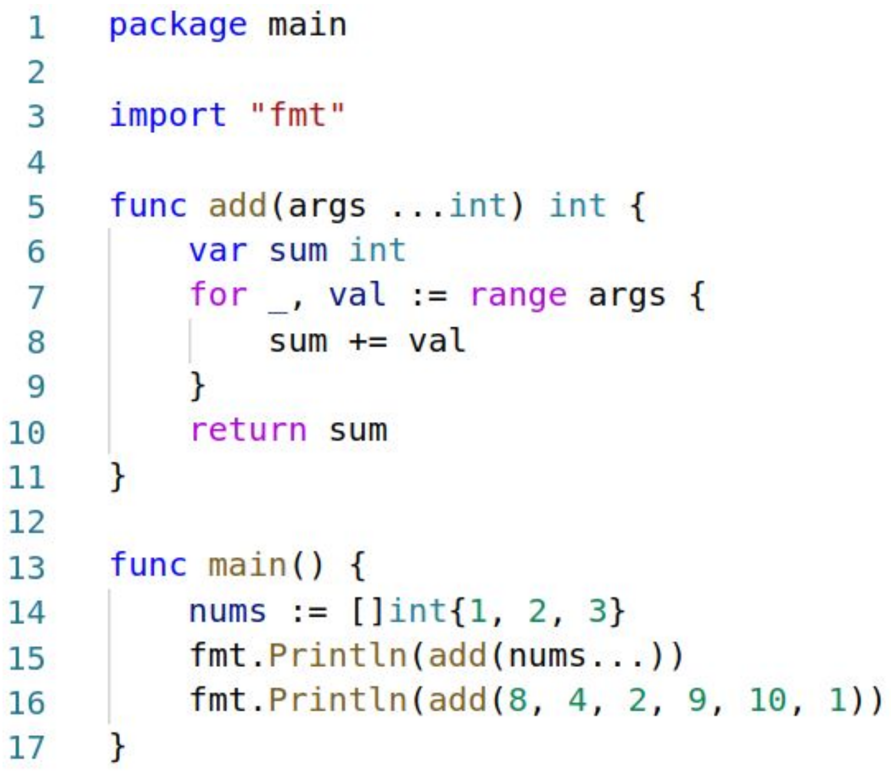
By using an ellipsis (`...`) before the type name of the **last parameter**, you can indicate that it takes zero or more of those parameters (**variadic parameter**)

```go
package main

import "fmt"

func add(args ...int) int {
    var sum int
    for _, val := range args {
        sum += val
    }
    return sum
}

func main() {
    fmt.Println(add(1, 2, 3, 4))
    fmt.Println(add(8, 4, 2, 9, 10, 1))
}
```

Variadic Functions

`add` function can accept any number of arguments

You can use `slices` as an argument to variadic parameter by using ellipsis (`...`) after the argument name



```go
package main

import "fmt"

func add(args ...int) int {
    var sum int
    for _, val := range args {
        sum += val
    }
    return sum
}

func main() {
    nums := []int{1, 2, 3}
    fmt.Println(add(nums...))
    fmt.Println(add(8, 4, 2, 9, 10, 1))
}
```

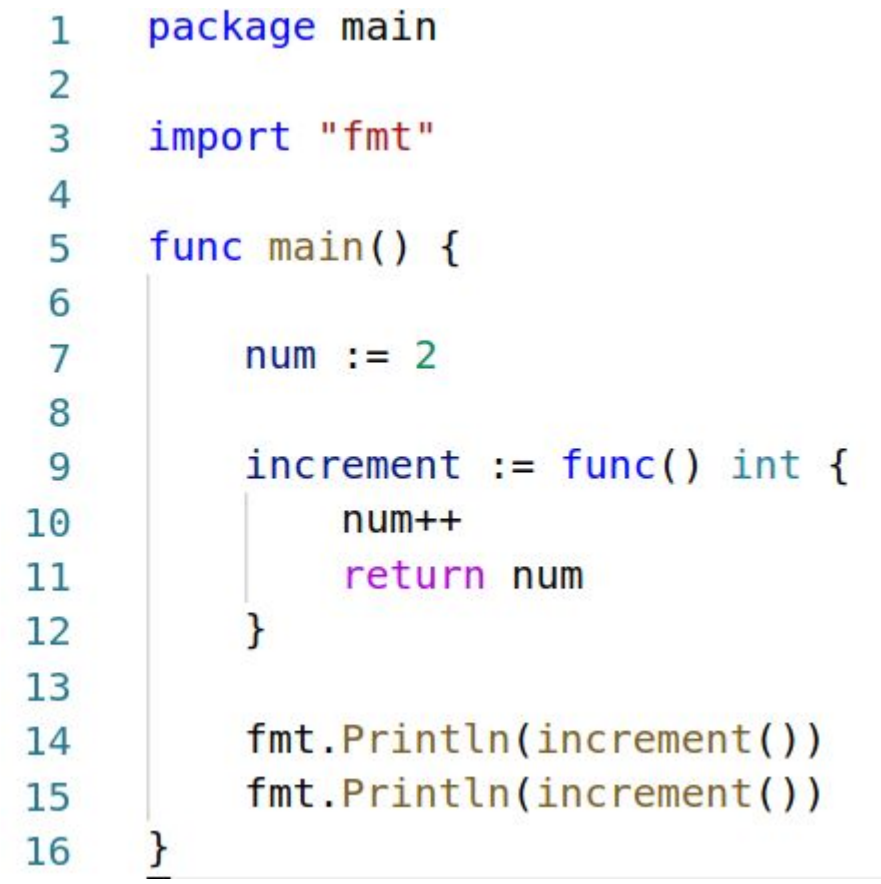
Exercise

Write a function with one variadic parameter that finds the greatest number in a list of numbers

Closure

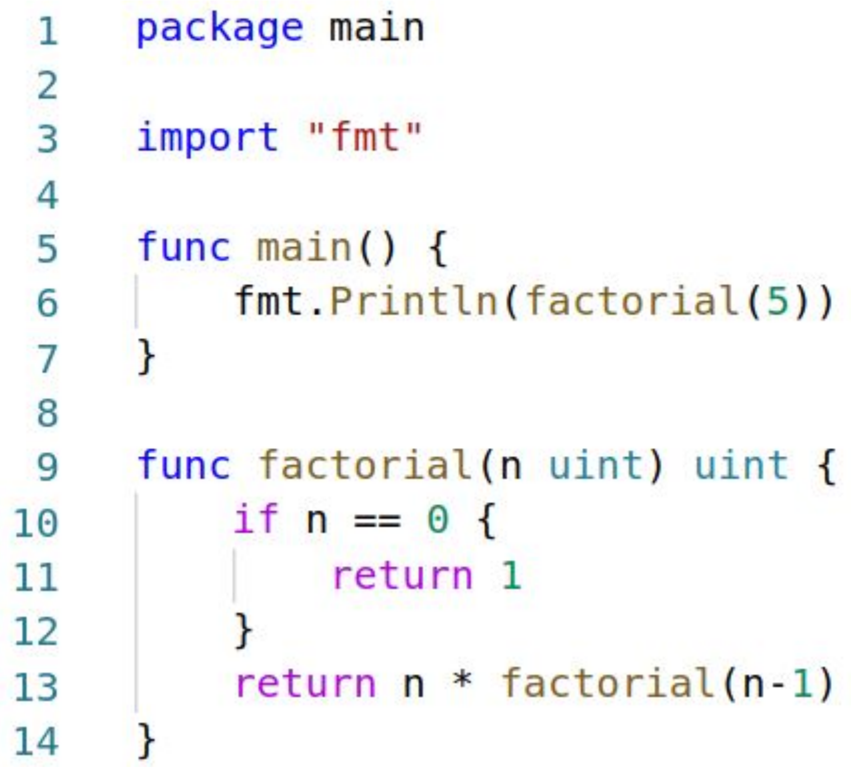
It is possible to create functions inside of functions

Inner function can access local variables of the outer function



```go
package main

import "fmt"

func main() {

    num := 2

    increment := func() int {
        num++
        return num
    }

    fmt.Println(increment())
    fmt.Println(increment())
}
```

Recursion



```go
package main

import "fmt"

func main() {
	fmt.Println(factorial(5))
}

func factorial(n uint) uint {
	if n == 0 {
		return 1
	}
	return n * factorial(n-1)
}
```

Exercise

The **Fibonacci sequence** is defined as:

`fib(0) = 0 , fib(1) = 1 , fib(n) = fib(n-1) + fib(n-2)`

Write a **recursive function** that can find `fib(n)`

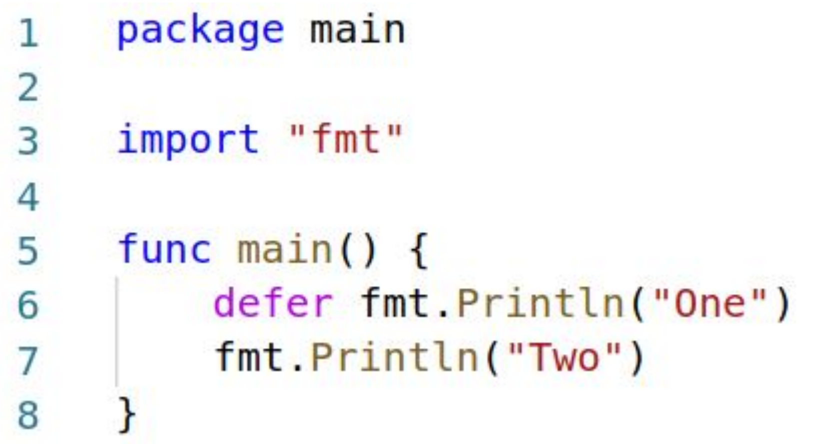
Exercise

Write a swap function by taking advantage the fact that Go functions can return multiple values

Defer

A defer statement **defers the execution of a function until the surrounding function returns**.

The deferred call's arguments are evaluated immediately, **but the function call is not executed until the surrounding function returns**.

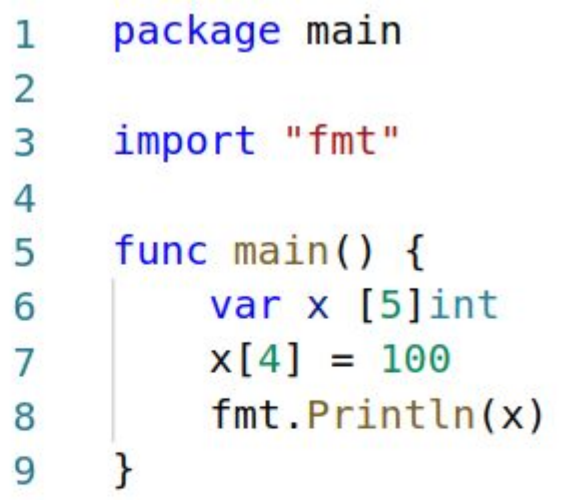


```go
package main

import "fmt"

func main() {
	defer fmt.Println("One")
	fmt.Println("Two")
}
```

Arrays

The type **`[n]T`** is an array of **`n`** values of type **`T`**

An array is a numbered sequence of elements of a single type with a fixed length

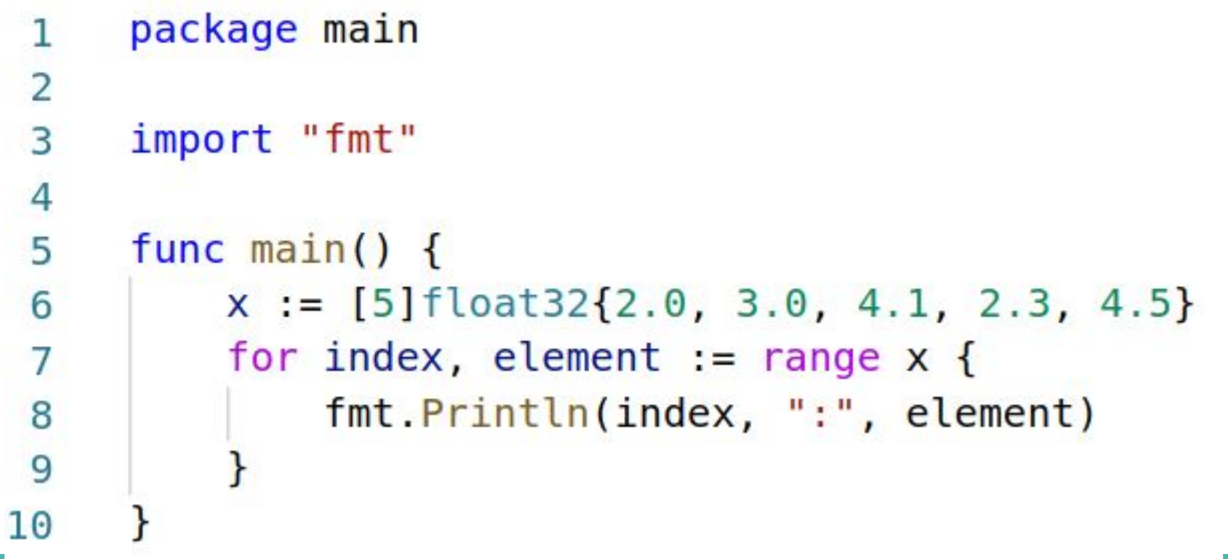


```go
package main

import "fmt"

func main() {
    var x [5]int
    x[4] = 100
    fmt.Println(x)
}
```

Iterating through array elements

You can use the **`range`** construct together with for loop.

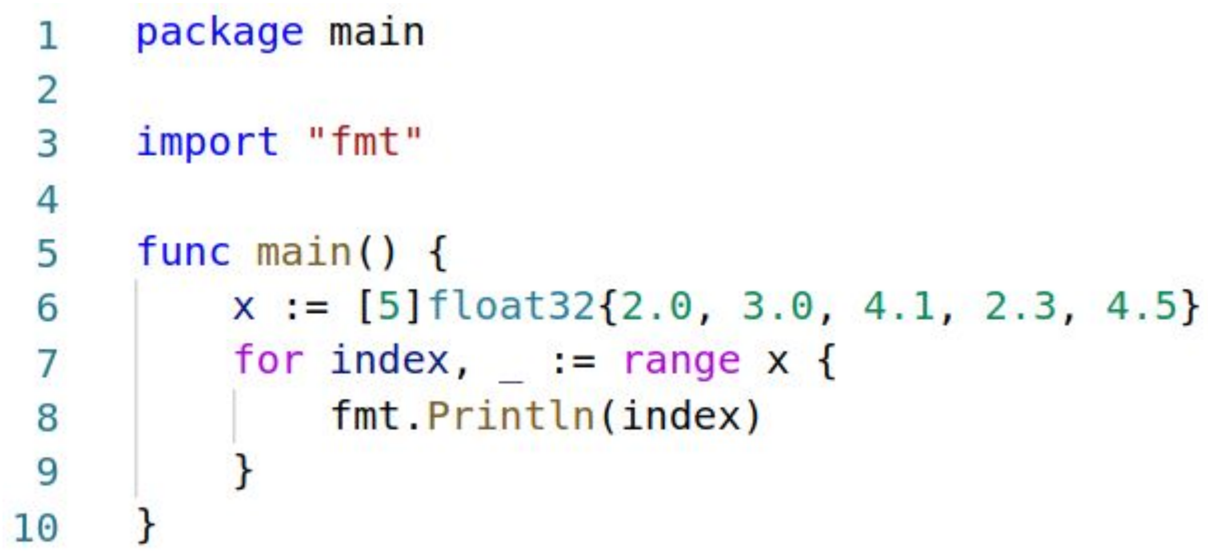
Notice the the number of returned values of **`range`**



```go
package main

import "fmt"

func main() {
	x := [5]float32{2.0, 3.0, 4.1, 2.3, 4.5}
	for index, element := range x {
		fmt.Println(index, ":", element)
	}
}
```

Print first return value (index only)

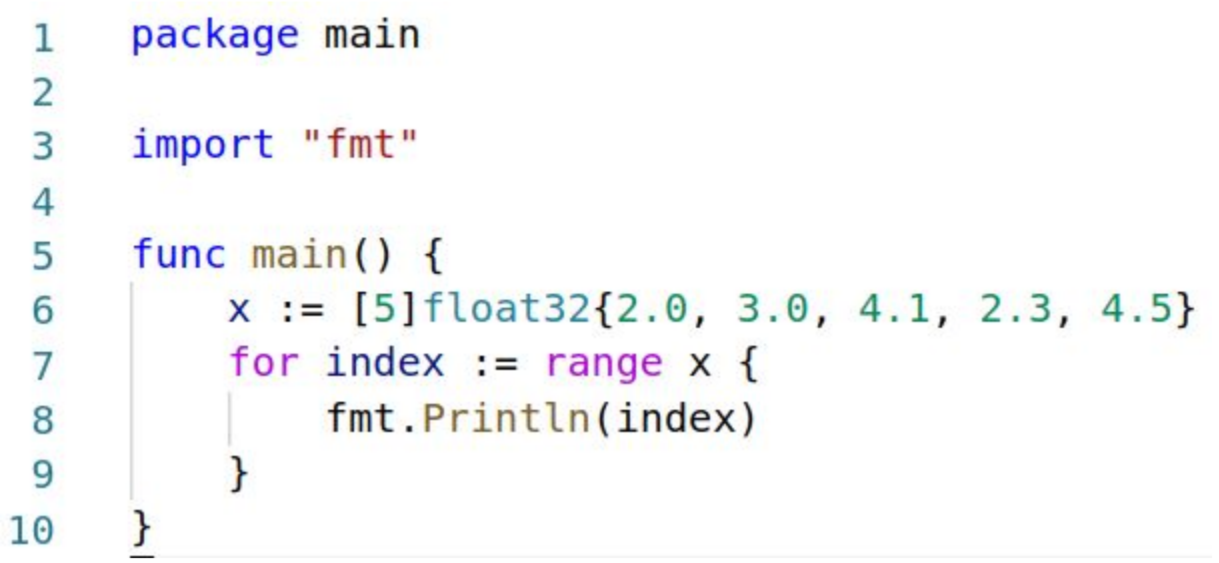
Use _ symbol to throw away the second returned value



```go
package main

import "fmt"

func main() {
	x := [5]float32{2.0, 3.0, 4.1, 2.3, 4.5}
	for index, _ := range x {
		fmt.Println(index)
	}
}
```

Print first return value (index only)

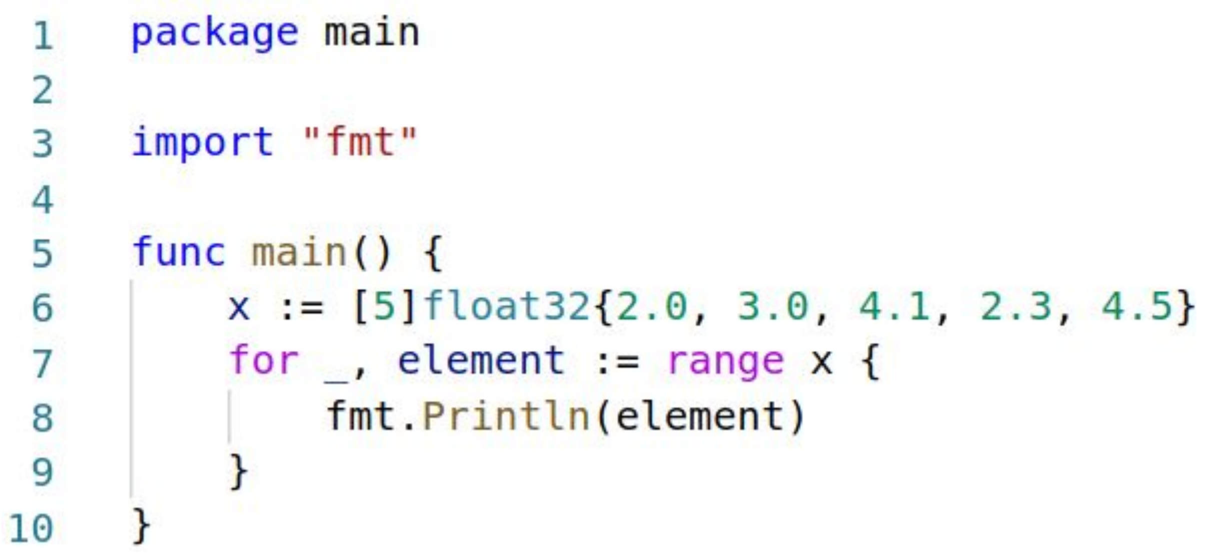
Or you can eliminate the second value altogether



```go
package main

import "fmt"

func main() {
    x := [5]float32{2.0, 3.0, 4.1, 2.3, 4.5}
    for index := range x {
        fmt.Println(index)
    }
}
```

Print second return value (element only)

Note using _ is mandatory here

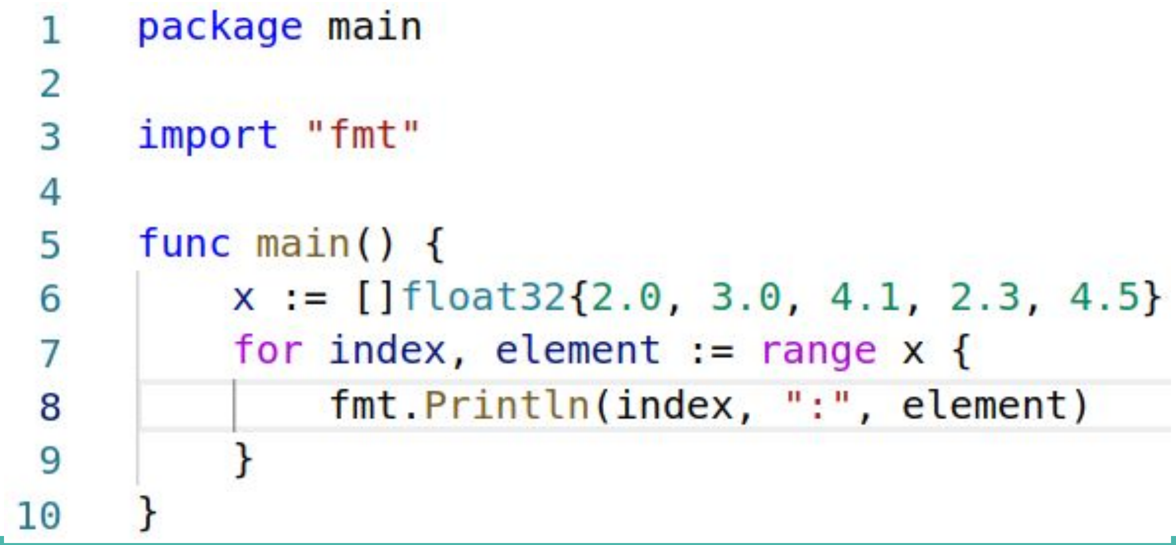


```go
package main

import "fmt"

func main() {
	x := [5]float32{2.0, 3.0, 4.1, 2.3, 4.5}
	for _, element := range x {
		fmt.Println(element)
	}
}
```

Slices

A slice is a segment of an array. Like arrays, slices are indexable and have a length. Unlike arrays, this length is allowed to change

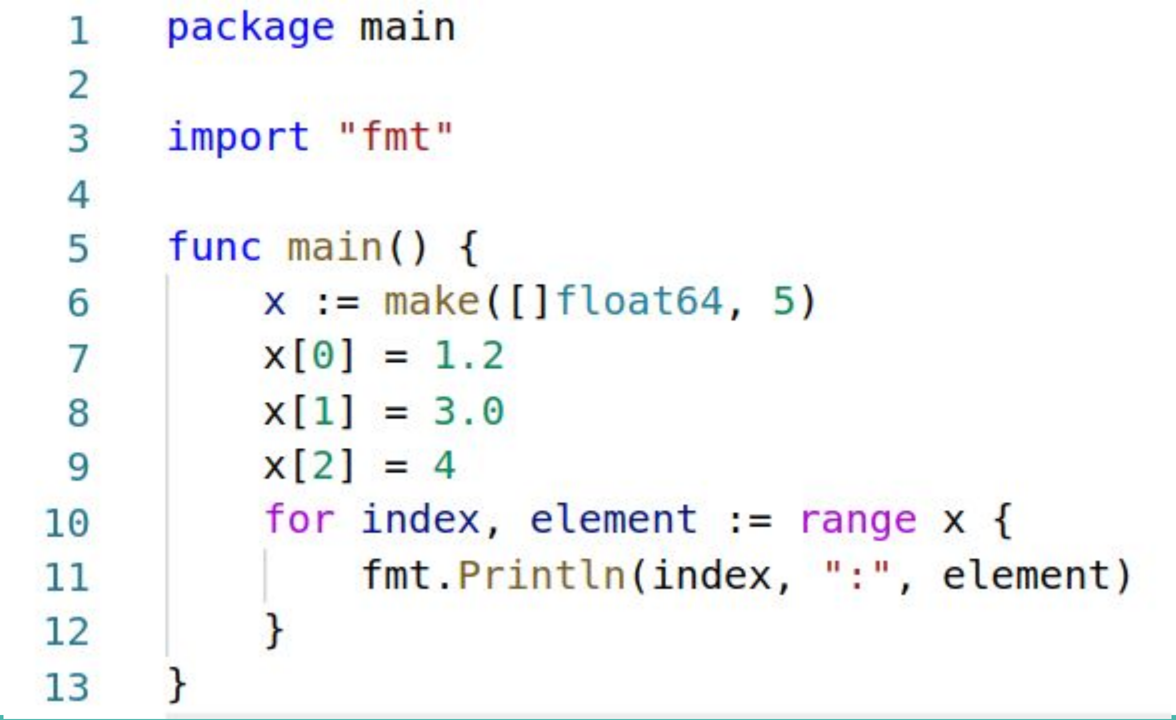
The type `[]T` is a slice with elements of type **T**



```go
package main

import "fmt"

func main() {
	x := []float32{2.0, 3.0, 4.1, 2.3, 4.5}
	for index, element := range x {
		fmt.Println(index, ":", element)
	}
}
```

Slices

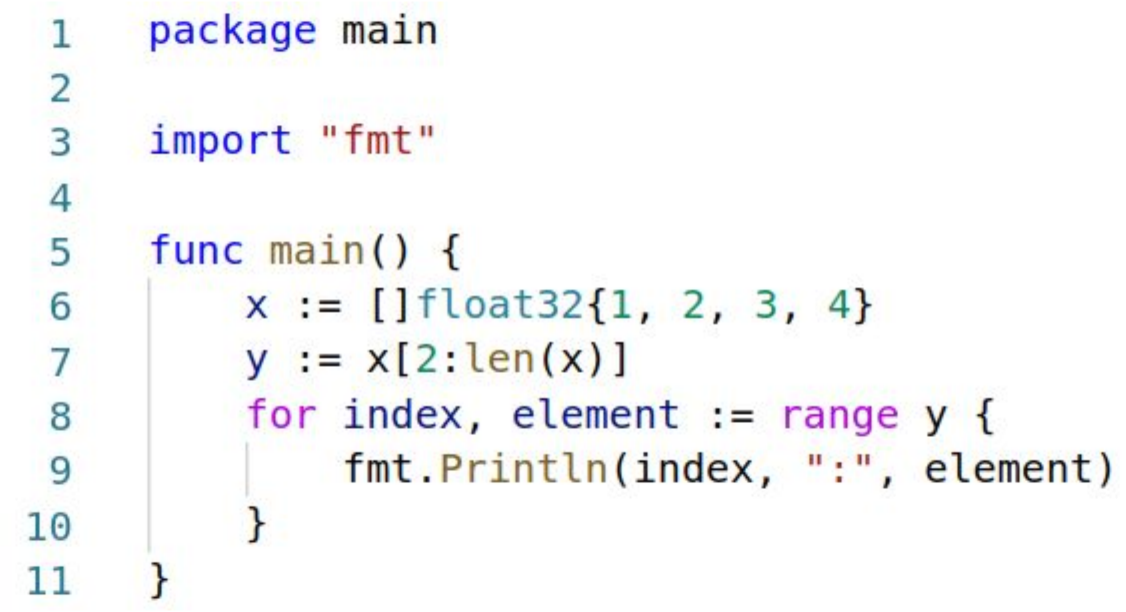
You can create slices by using make function



```go
package main

import "fmt"

func main() {
    x := make([]float64, 5)
    x[0] = 1.2
    x[1] = 3.0
    x[2] = 4
    for index, element := range x {
        fmt.Println(index, ":", element)
    }
}
```

Slices

It is possible to create slices from another slice or array



```go
package main

import "fmt"

func main() {
	x := []float32{1, 2, 3, 4}
	y := x[2:len(x)]
	for index, element := range y {
		fmt.Println(index, ":", element)
	}
}
```

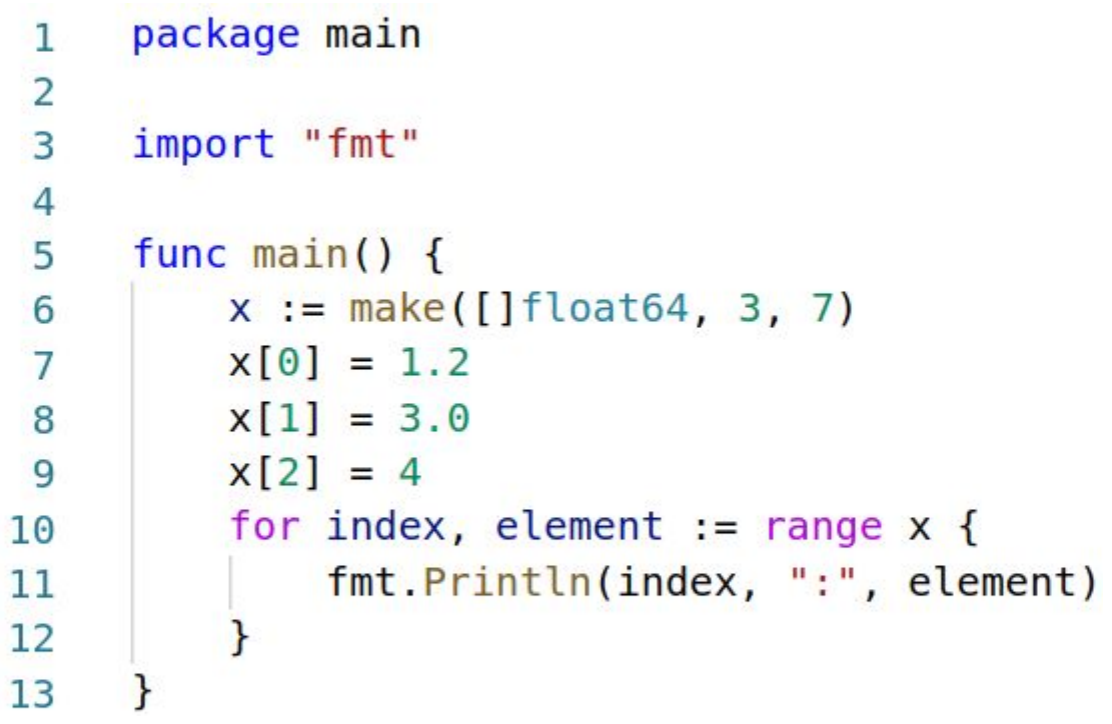
Slices, len and cap functions

Slices are always associated with some array, and although they can never be longer than the array, they can be smaller.

The make function also allows a third parameter:

Length (len): second parameter of make

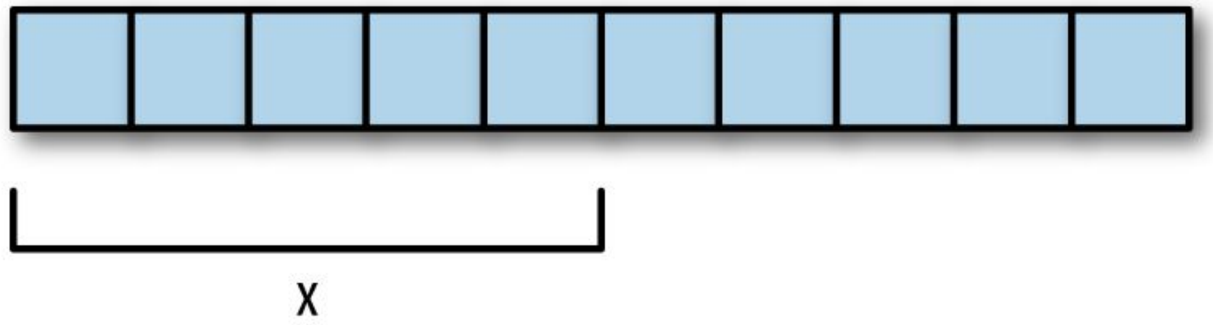
Capacity(cap): third parameter of make



```go
package main

import "fmt"

func main() {
    x := make([]float64, 3, 7)
    x[0] = 1.2
    x[1] = 3.0
    x[2] = 4
    for index, element := range x {
        fmt.Println(index, ":", element)
    }
}
```

Slices, len and cap functions

The following figure shows a slice of length 5 and with the underlying capacity of the array 10



Slice: Append

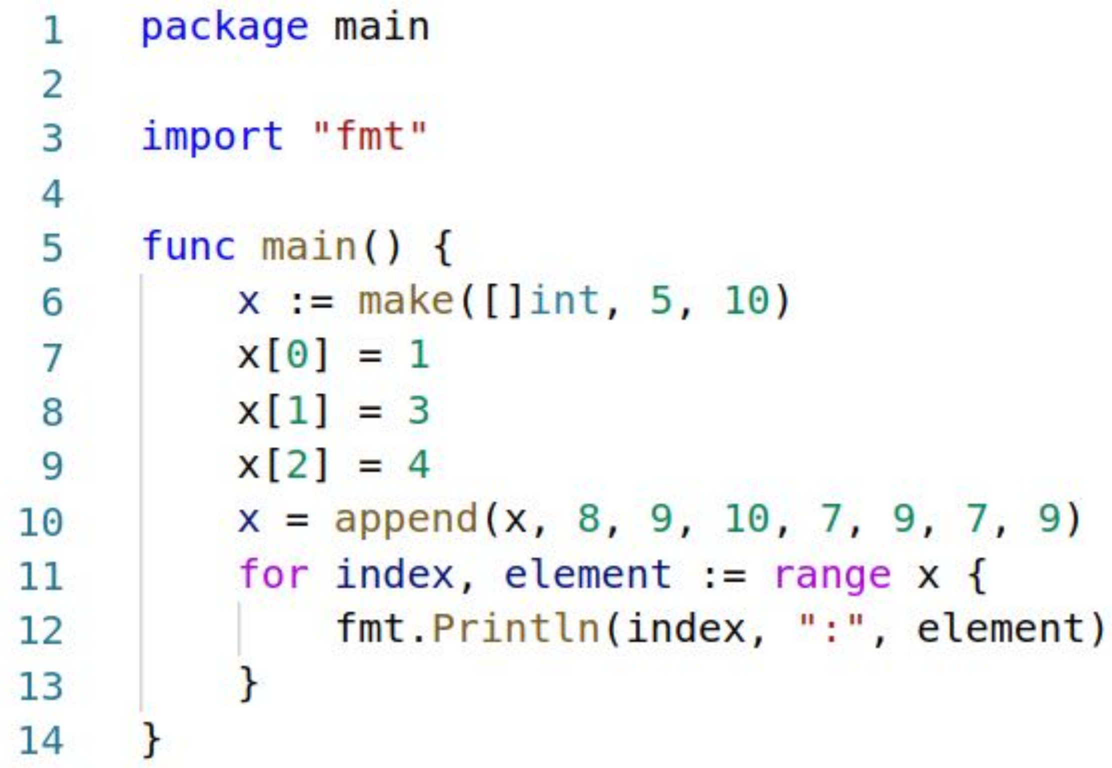
append adds elements onto the end of a slice

If there's sufficient capacity in the underlying array, the element is placed after the last element and the length is incremented.

However, if there is not sufficient capacity, a new array is created, all of the existing elements are copied over, the new element is added onto the end, and the new slice is returned

Slice: Append

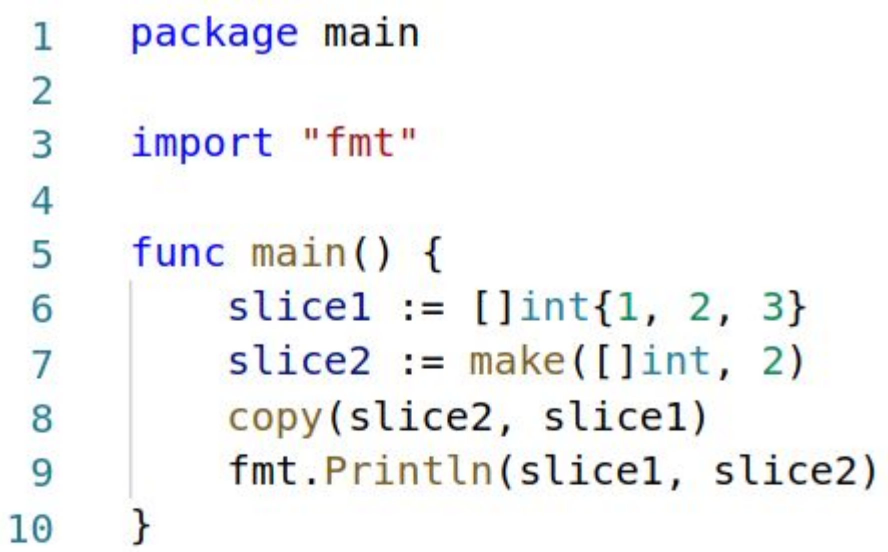
append adds elements onto the end of a slice



```go
package main

import "fmt"

func main() {
    x := make([]int, 5, 10)
    x[0] = 1
    x[1] = 3
    x[2] = 4
    x = append(x, 8, 9, 10, 7, 9, 7, 9)
    for index, element := range x {
        fmt.Println(index, ":", element)
    }
}
```

Slice: Copy

copy takes two arguments: dst and src . All of the entries in src are copied into dst overwriting whatever is there. If the lengths of the two slices are not the same, the smaller of the two will be used.

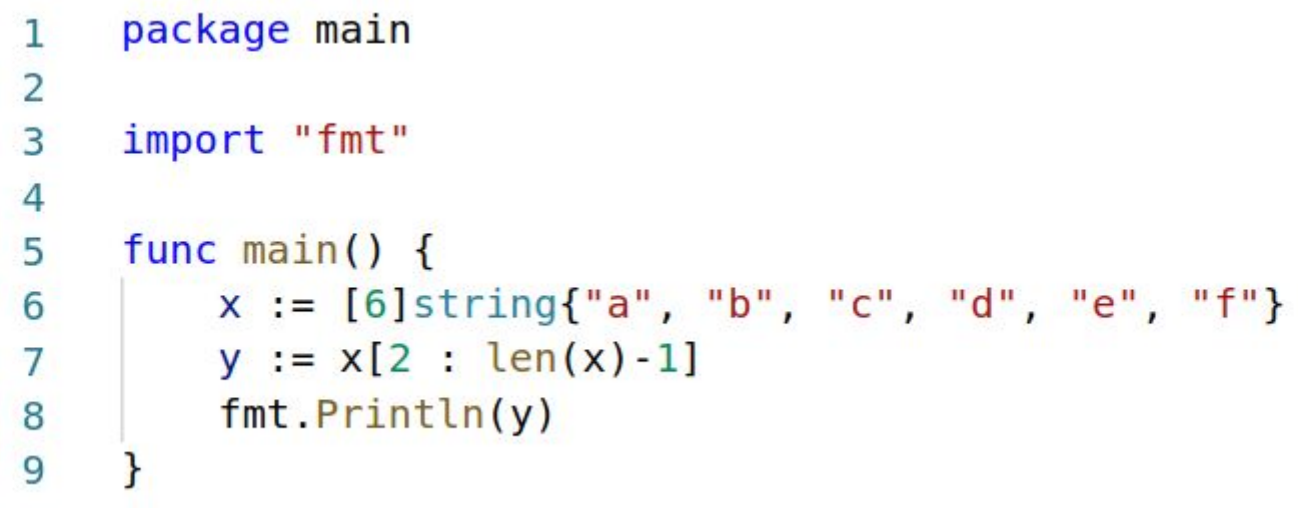


```go
package main

import "fmt"

func main() {
	slice1 := []int{1, 2, 3}
	slice2 := make([]int, 2)
	copy(slice2, slice1)
	fmt.Println(slice1, slice2)
}
```

Exercises

What is the length of a slice created using `make([]int, 3, 9)`?

What would be the output of the following program?



```go
package main

import "fmt"

func main() {
	x := [6]string{"a", "b", "c", "d", "e", "f"}
	y := x[2 : len(x)-1]
	fmt.Println(y)
}
```

Exercise

Write a program that finds the smallest number in this list:

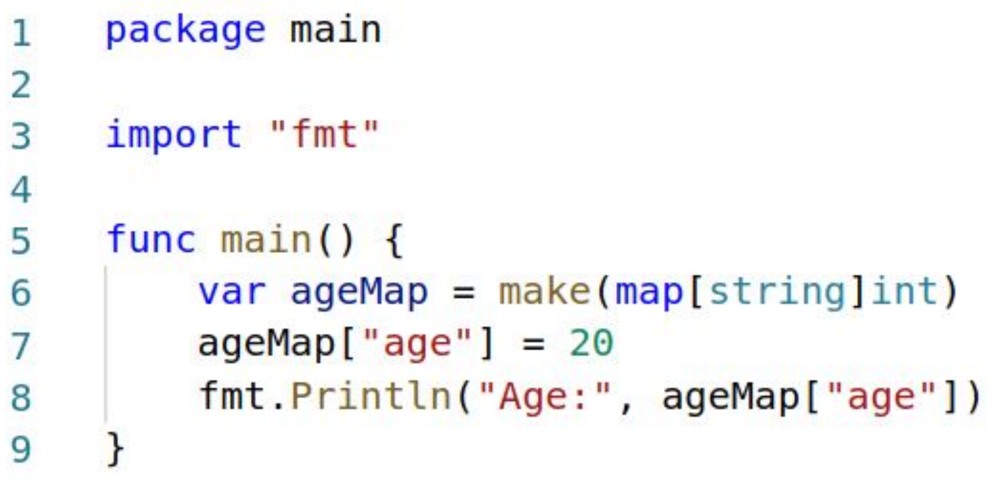
```
    x := []int{48,37,34,83,27,19,97,9,17}
```

Write a function that takes a slice of numbers and returns the sum of the slice elements

Maps

A **map** is an unordered collection of key-value pairs (maps are also sometimes called associative arrays, hash tables, or dictionaries). Maps are used to lookup a value by its associated key

Maps have to be initialized before they can be used

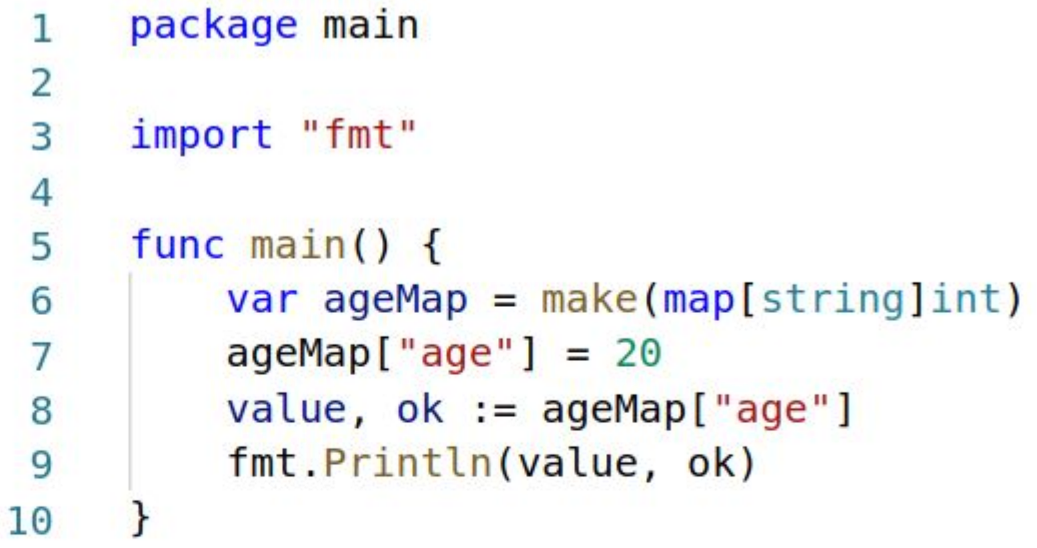


```go
package main

import "fmt"

func main() {
	var ageMap = make(map[string]int)
	ageMap["age"] = 20
	fmt.Println("Age:", ageMap["age"])
}
```

Map

Accessing an element of a map can return two values.

The first value is the **result of the lookup**, the second tells us **whether or not the lookup was successful**



```go
package main

import "fmt"

func main() {
	var ageMap = make(map[string]int)
	ageMap["age"] = 20
	value, ok := ageMap["age"]
	fmt.Println(value, ok)
}
```

Map Operations

Computing length of map

```
len(ageMap)
```

Deleting map element by key

```
delete(ageMap, “age”)
```

Exercise

Write a program, using map, that maps the following course code with the corresponding title

Course Code	Course Title
ITSE-3241	Object-Oriented Programming II
ITSE-3193	Web Programming I
MEng-1052	Engineering Mechanics II (Dynamics)
ITSE-3161	Digital Logic Design
ITSE-3133	System Programming

Pointers

Pointers **reference a location in memory where a value is stored** rather than the value itself

The type `*T` is a pointer to a `T` **value**. Its zero value is `nil`.

A pointer is represented using an **asterisk (*)** followed by the type of the stored value

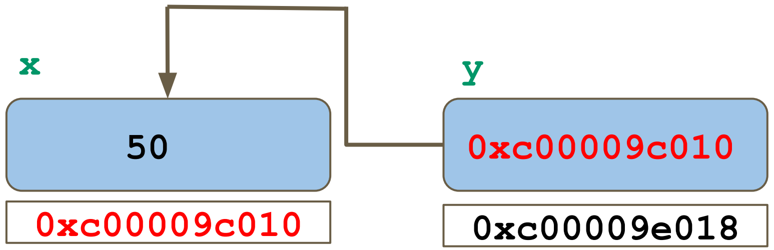
An asterisk is also used to **dereference pointer variables**.

Dereferencing a pointer gives us access to the value the pointer points to

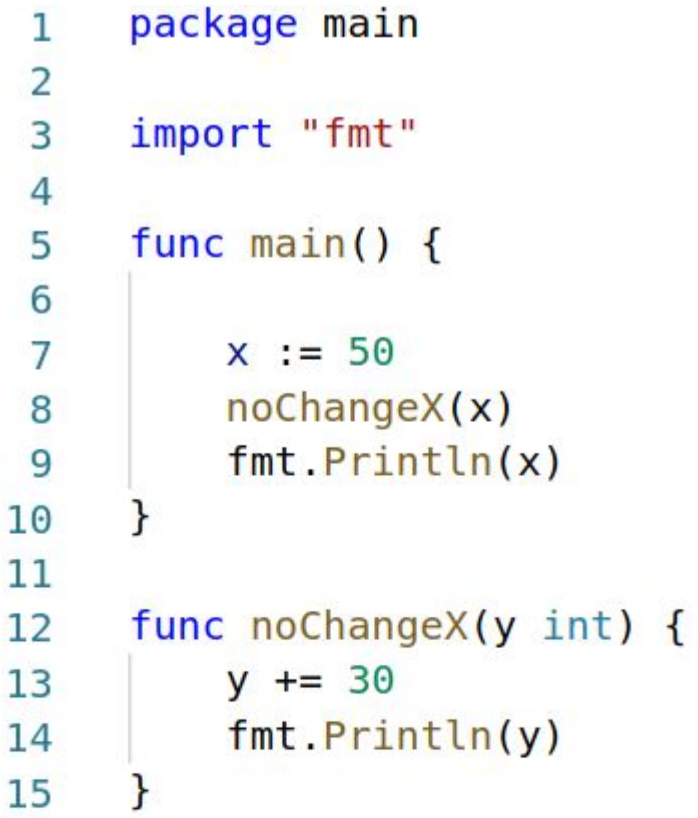
Pointers

We use the **&** operator **to find the address of a variable**

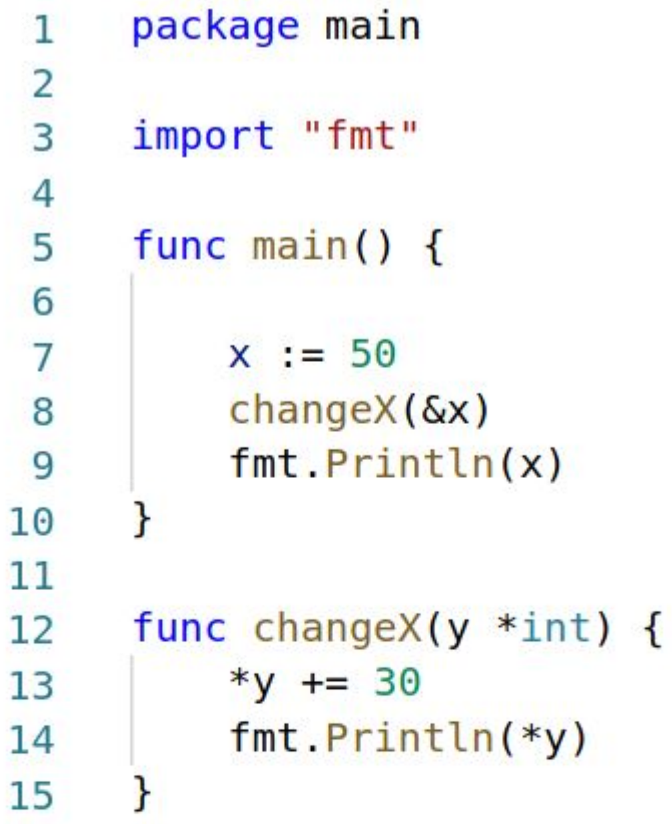
```go
package main

import "fmt"

func main() {
	var x int = 50
	var y *int = &x
	fmt.Println(x, y)
}
```



Pointers



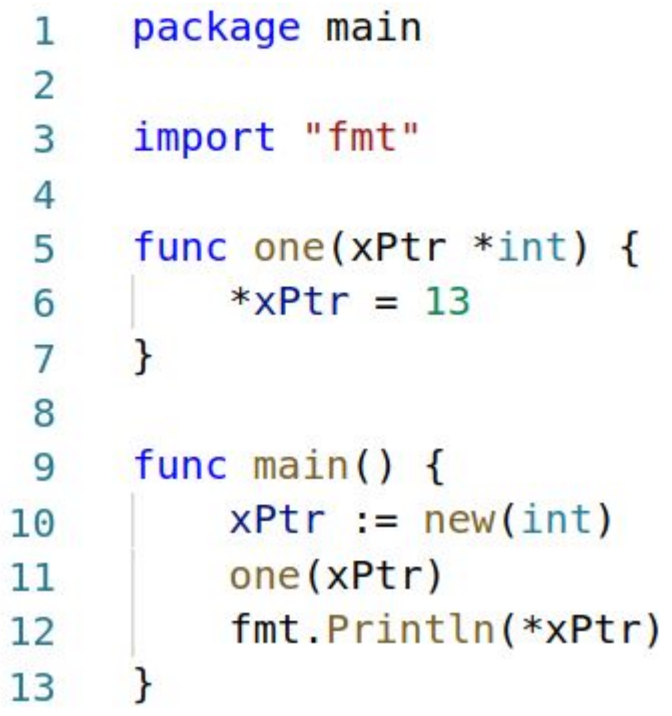
```go
package main

import "fmt"

func main() {

    x := 50
    noChangeX(x)
    fmt.Println(x)
}

func noChangeX(y int) {
    y += 30
    fmt.Println(y)
}
```



```go
package main

import "fmt"

func main() {

    x := 50
    changeX(&x)
    fmt.Println(x)
}

func changeX(y *int) {
    *y += 30
    fmt.Println(*y)
}
```

Pointers

Another way to get a pointer is to use the built-in new function



```go
package main

import "fmt"

func one(xPtr *int) {
	*xPtr = 13
}

func main() {
	xPtr := new(int)
	one(xPtr)
	fmt.Println(*xPtr)
}
```

Exercises

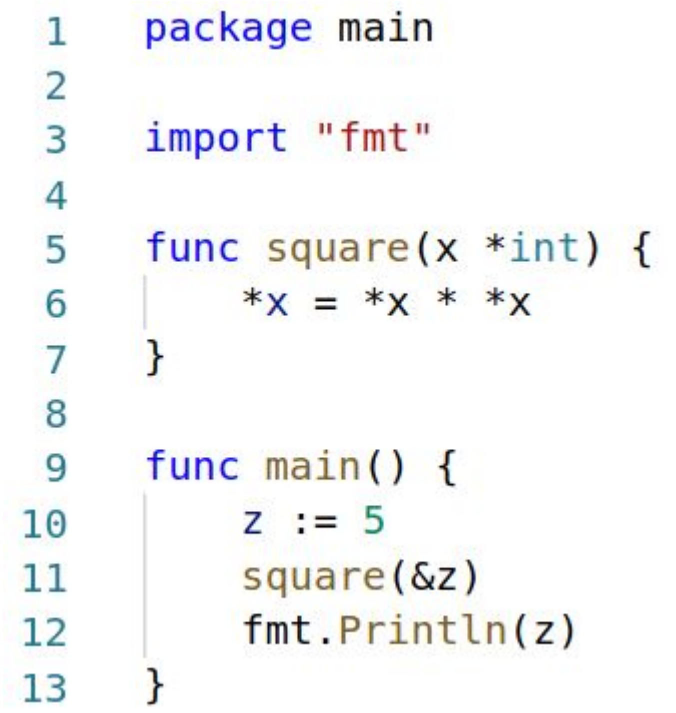
How do you get the memory address of a variable?

How do you assign a value to a pointer?

Write a program that can swap two integers `( x := 1; y := 2; swap(&x, &y)` should give you `x=2` and `y=1`)

Exercises

What is the output of the following program

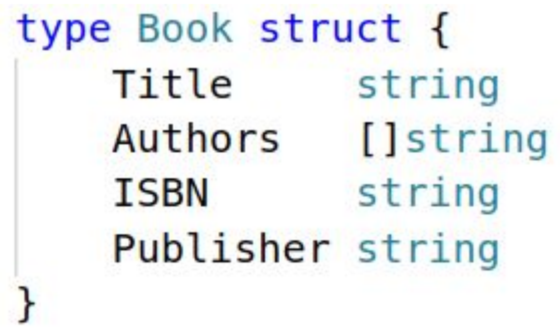


```go
package main

import "fmt"

func square(x *int) {
    *x = *x * *x
}

func main() {
    z := 5
    square(&z)
    fmt.Println(z)
}
```

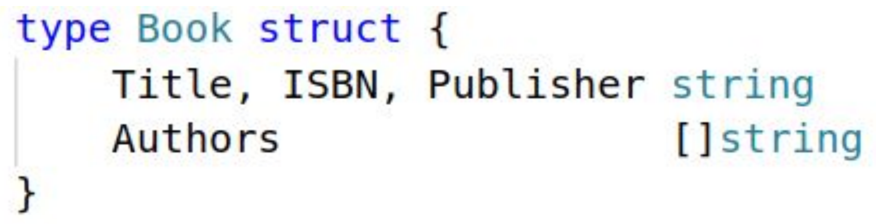
Structs

A struct is a type that contains named fields

The `type` **keyword** introduces a new type. It's followed by the name of the type (`Book`), the **keyword struct** to indicate that we are defining a struct type, and a **list of fields** inside of **curly braces**

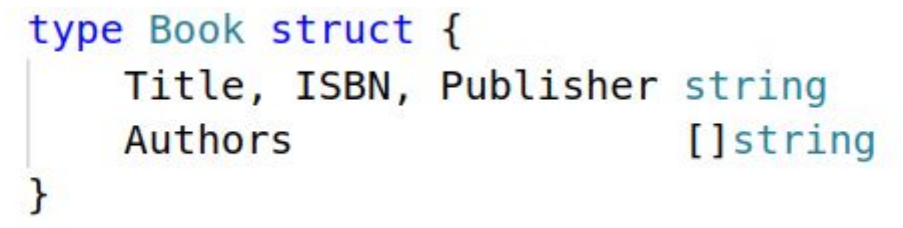


```
type Book struct {
    Title     string
    Authors   []string
    ISBN      string
    Publisher string
}
```



```
type Book struct {
    Title, ISBN, Publisher string
    Authors                []string
}
```

Struct: Initialization



```
type Book struct {
    Title, ISBN, Publisher string
    Authors                []string
}
```

Initialization alternatives

Alternative One

```
var b Book
```

Alternative Two

```
var b = new(Book)
```

This allocates memory for all the fields, sets each of them to their zero value, and returns a pointer to the struct (`*Book`). Pointers are often used with structs so that functions can modify their contents

Struct: Initialization

```go
type Book struct {
    Title, ISBN, Publisher string
    Authors                []string
}
```

Initialization alternatives

Alternative Three

```go
b := Book{
    Title:     "Go Web Programming",
    ISBN:      "9781617292569",
    Publisher: "Manning Publications",
    Authors:   []string{"Sau Sheong Chang"},
}
```

Struct: Initialization

```go
type Book struct {
    Title, ISBN, Publisher string
    Authors                []string
}
```

Initialization alternatives

Alternative Four

```go
b := Book{
    "Go Web Programming",
    "9781617292569",
    "Manning Publications",
    []string{"Sau Sheong Chang"},
}
```

Struct: Accessing Fields

```go
type Book struct {
    Title, ISBN, Publisher string
    Authors                []string
}
```

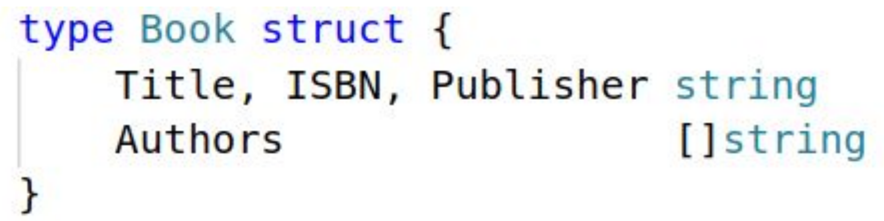
We can access fields using the **. operator**

```go
func main() {

    b := Book{
        "Go Web Programming",
        "9781617292569",
        "Manning Publications",
        []string{"Sau Sheong Chang"},
    }

    fmt.Println(b.Title, b.ISBN, b.Publisher, b.Authors[0])
}
```

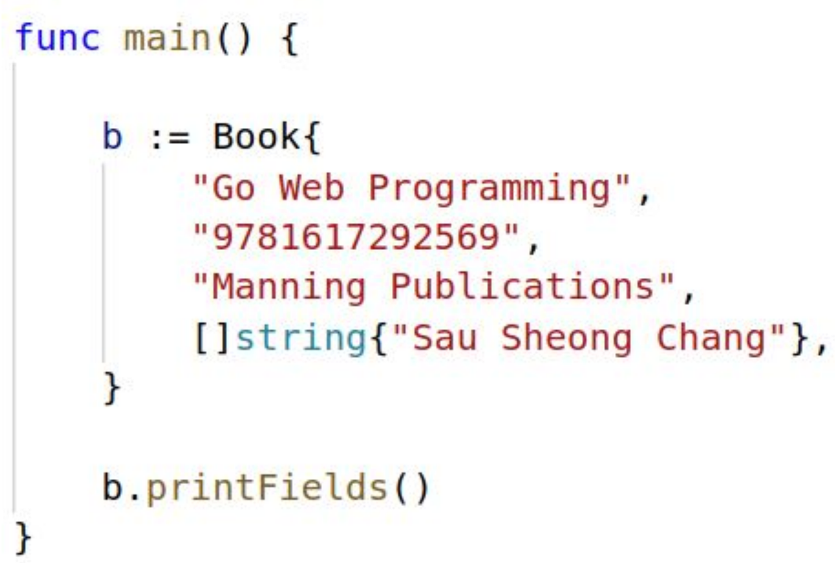
Struct: Methods

You can associate methods to **types**

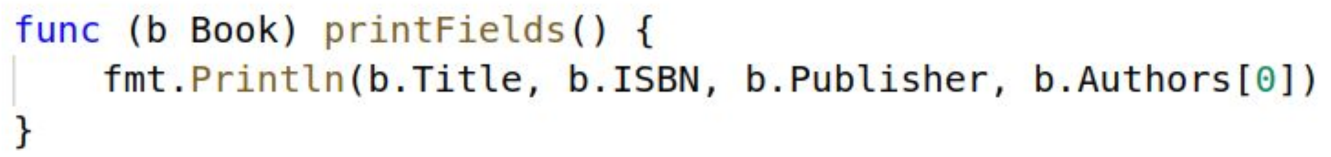
In the function **`printFields()`** b is called **receiver**



```
type Book struct {
    Title, ISBN, Publisher string
    Authors                []string
}
```



```
func main() {

    b := Book{
        "Go Web Programming",
        "9781617292569",
        "Manning Publications",
        []string{"Sau Sheong Chang"},
    }

    b.printFields()
}
```



```
func (b Book) printFields() {
    fmt.Println(b.Title, b.ISBN, b.Publisher, b.Authors[0])
}
```

Struct: Methods

You can associate methods to **types**

In the function **`printFields()`** b is called **receiver**

```
type Book struct {
    Title, ISBN, Publisher string
    Authors                []string
}
```

```
func main() {

    b := Book{
        "Go Web Programming",
        "9781617292569",
        "Manning Publications",
        []string{"Sau Sheong Chang"},
    }

    b.printFields()
}

func (b *Book) printFields() {
    fmt.Println(b.Title, b.ISBN, b.Publisher, b.Authors[0])
}
```

Struct: Methods with Pointer receivers

You can declare methods with pointer receivers

This means the receiver type has the literal syntax `*T` for some type `T`

`T` cannot itself be a pointer such as `*int`

Methods with pointer receivers can modify the value to which the receiver points.

Since methods often need to modify their receiver, pointer receivers are more common than value receivers

Study the program shown in the next slide

Struct: Methods with Pointer receivers

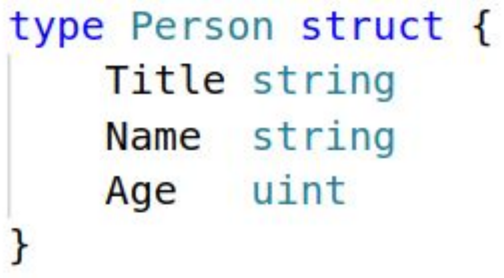
```go
// Rectangle struct
type Rectangle struct {
    height float32
    width  float32
}

func (r Rectangle) scaleOne(size float32) {
    r.height *= size
    r.width *= size
}

func (r *Rectangle) scaleTwo(size float32) {
    (*r).height *= size
    r.width *= size
}
```

```go
func main() {
    r := Rectangle{2.3, 4.6}
    fmt.Println(r)
    r.scaleOne(10)
    fmt.Println(r)
    r.scaleTwo(10)
    fmt.Println(r)
}
```

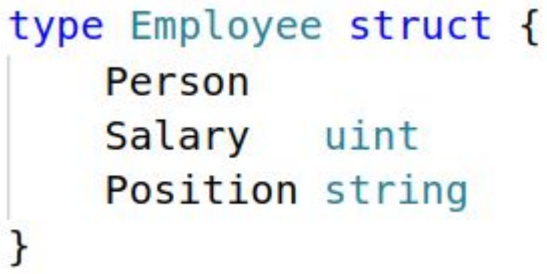
Struct: Embedded Types

A struct's fields usually represent the **has-a** relationship (e.g. Book **has-a** title)

How to represent **is-a** relationship



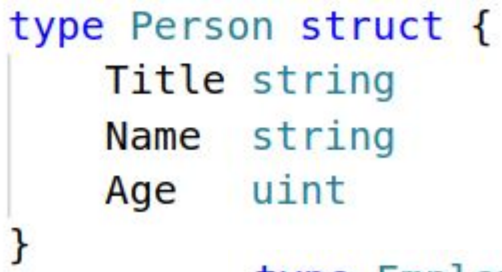
```go
type Person struct {
    Title string
    Name  string
    Age   uint
}
```



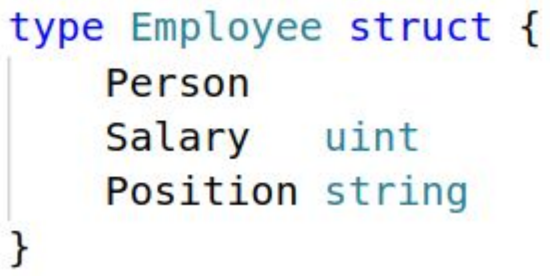
```go
type Employee struct {
    Person
    Salary   uint
    Position string
}
```

Struct: Embedded Types

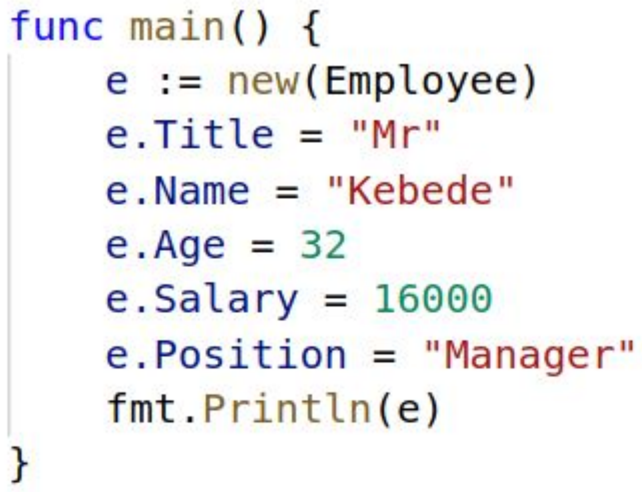
Is-a relationship



```go
type Person struct {
    Title string
    Name  string
    Age   uint
}
```



```go
type Employee struct {
    Person
    Salary   uint
    Position string
}
```



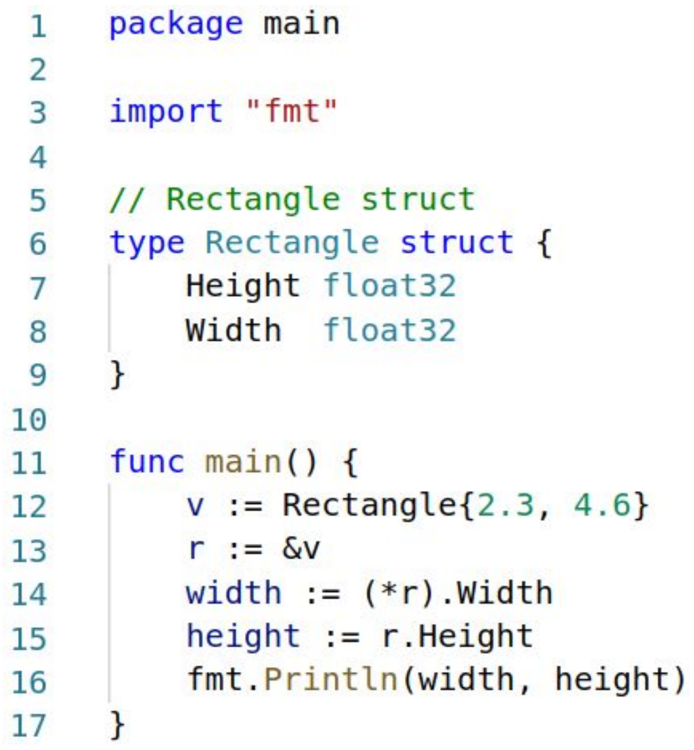
```go
func main() {
    e := new(Employee)
    e.Title = "Mr"
    e.Name = "Kebede"
    e.Age = 32
    e.Salary = 16000
    e.Position = "Manager"
    fmt.Println(e)
}
```

Struct: Pointers to structs

Struct fields can be accessed through a **struct pointer**

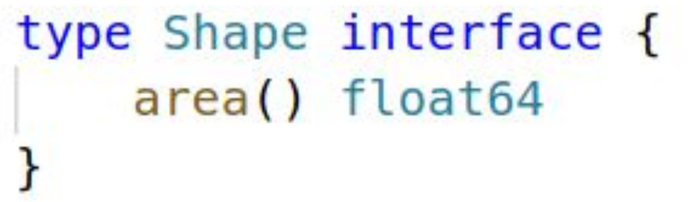
To access the field `X` of a struct when we have the struct pointer `p` we could write `(*p).X`

However, that notation is cumbersome, so the language permits us instead to write just `p.X`, without the explicit dereference



```go
package main

import "fmt"

// Rectangle struct
type Rectangle struct {
    Height float32
    Width  float32
}

func main() {
    v := Rectangle{2.3, 4.6}
    r := &v
    width := (*r).Width
    height := r.Height
    fmt.Println(width, height)
}
```

Interfaces



```
type Shape interface {
    area() float64
}
```

Like a struct, an **interface** is created using the **type** keyword, followed by a **name** and the keyword **interface**

But instead of defining fields, **we define a method set**.

A **method set** is a **list of methods that a type must have in order to implement the interface**

A type implements an interface by implementing its methods

There is no explicit declaration of intent, no "implements" keyword

Interfaces

```go
type Circle struct {
    radius float64
}

type Rectangle struct {
    width  float64
    height float64
}

type Shape interface {
    area() float64
}
```

```go
func (c Circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (r Rectangle) area() float64 {
    return r.width * r.height
}

func totalArea(shp ...Shape) float64 {
    total := 0.0
    for _, s := range shp {
        total += s.area()
    }
    return total
}
```

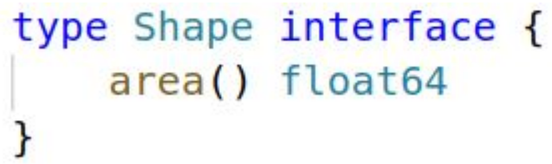
Interfaces

```go
type Circle struct {
    radius float64
}

type Rectangle struct {
    width  float64
    height float64
}

type Shape interface {
    area() float64
}
```

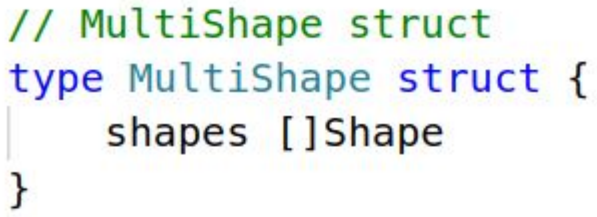
```go
func main() {
    c := Circle{3.0}
    r := Rectangle{3.0, 4.0}
    println("Area of circle:", c.area())
    println("Area of rectangle:", r.area())
    println("Total area:", totalArea(c, r))
}
```

Interfaces

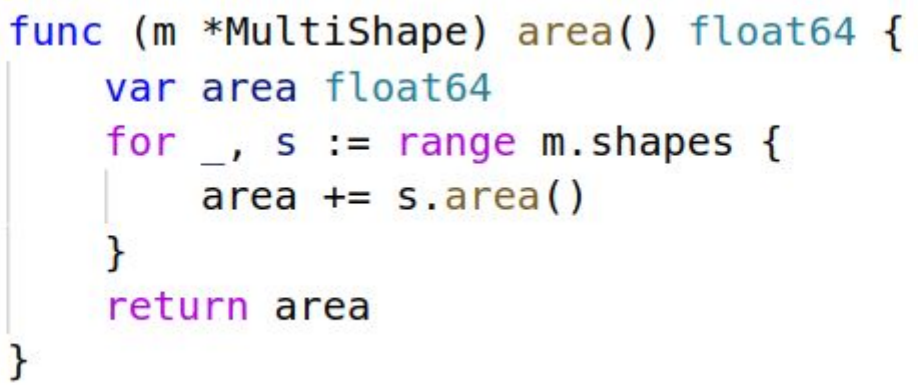
Interfaces can also be used as fields



```go
type Shape interface {
    area() float64
}
```



```go
// MultiShape struct
type MultiShape struct {
    shapes []Shape
}
```



```go
func (m *MultiShape) area() float64 {
    var area float64
    for _, s := range m.shapes {
        area += s.area()
    }
    return area
}
```

Interfaces

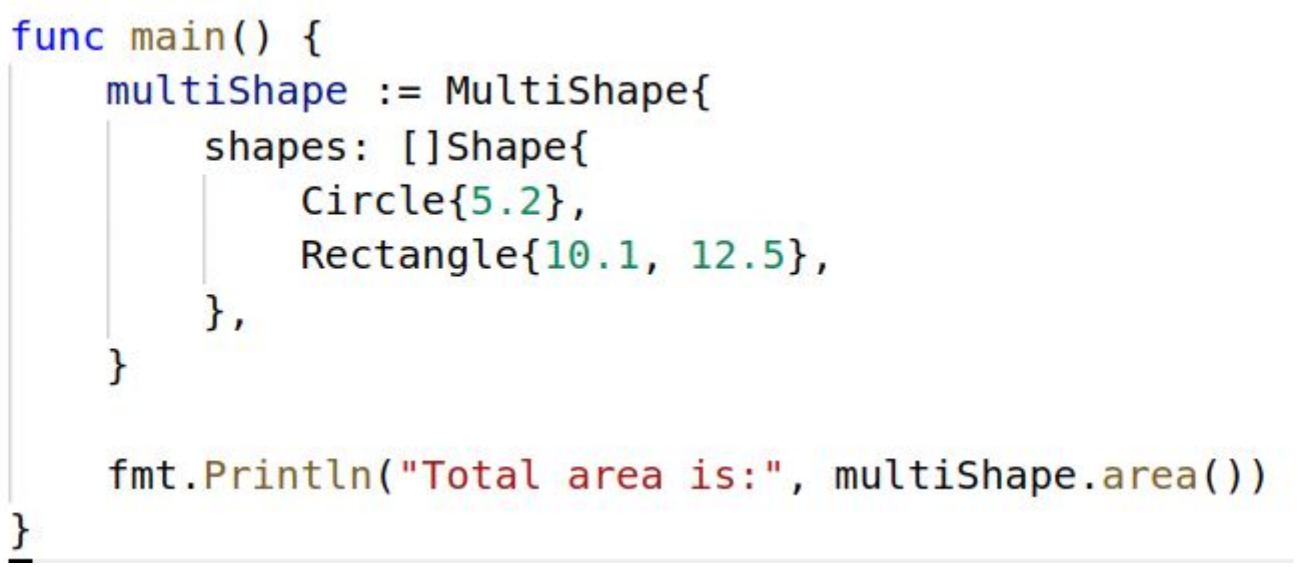
Interfaces can also be used as fields

```go
// Circle struct
type Circle struct {
    radius float64
}

// Rectangle struct
type Rectangle struct {
    width  float64
    height float64
}
```

```go
func (c Circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (r Rectangle) area() float64 {
    return r.width * r.height
}
```

Interfaces

Interfaces can also be used as fields



```go
func main() {
    multiShape := MultiShape{
        shapes: []Shape{
            Circle{5.2},
            Rectangle{10.1, 12.5},
        },
    }

    fmt.Println("Total area is:", multiShape.area())
}
```

Exercise

What’s the difference between a **method** and a **function**?

Why would you use an **embedded anonymous field** instead of a normal named field?

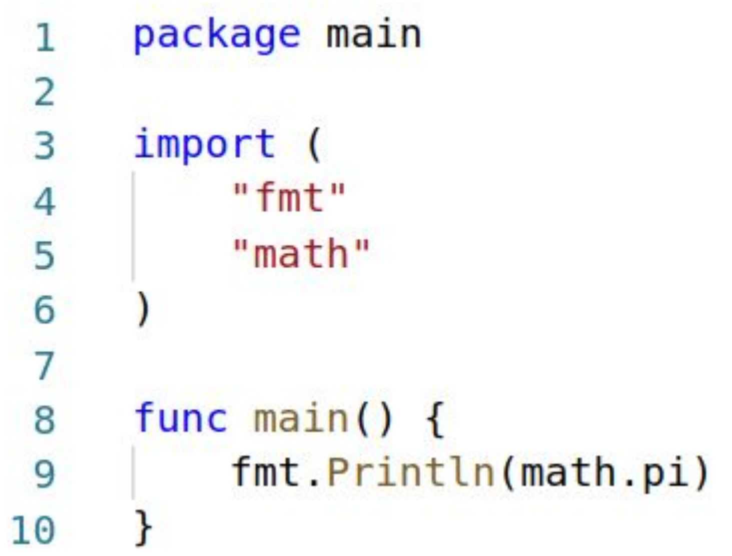
Add a new perimeter method to the Shape interface to calculate the perimeter of a shape. Implement the method for Circle and Rectangle .

Exported Names

In Go, **a name is exported if it begins with a capital letter**

The code shown in the right cannot run as pi is not exported

Try to capitalize pi to Pi and check if it works



```go
package main

import (
	"fmt"
	"math"
)

func main() {
	fmt.Println(math.pi)
}
```