Intro

packages - ways to reference files. - package main: entry point file - import math/rand: links “package rand” files

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println("My favorite number is", rand.Intn(10))
}
```

- to export exported vars from packages, MUST be capital

```
fmt.Println(math.Pi)
```

Functions

declaring arguments

```
x int, y int
x, y int
ptr *int
arr [3]int
```

arithmetic

```
func add(x, y int) int {  // short for (x int, y int)
    z := x+y
    return z
}

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

    x, y := 3, 4
    var f = math.Sqrt(float64(x*x + y*y))  //sqrt requires float input
    var z uint = uint(f)  // var zuint = f  returns error, MANUAL convert
    fmt.Println(x, y, z) //3,4,5
}
```

string

```
func swap(x, y string) (string, string) { # returns two str
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}
```

naked return (shorthand)

```
func split(sum int) (x, y int) {
    x = sum * 4 / 9  // not := because already declared above as ret
    y = sum - x
    return  //returns x and y
}

func main() {
    fmt.Println(split(17))
}
```

Functions as values

as variables

```
func main() {
    hypot := func(x, y float64) float64 {
        return math.Sqrt(x*x + y*y)
    }
    fmt.Println(hypot(5, 12))
}
```

as parameters

```
func compute(fn func(float64, float64) float64) float64 {
    return fn(3, 4)
}
func main() {
    fmt.Println(compute(math.Pow))
}
```

Closure Functions

```
func adder() func(int) int { //returns func
    sum := 0
    return func(x int) int { //that remembers sum
        sum += x
        return sum
    }
}

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

Example : fibonacci

```
package main

import "fmt"

// returns func
func fibonacci() func() int {
    f_0 := 0
    f_1 := 1
    return func() int { //that remembers f_0 and f_1
        temp := f_0
        f_0 = f_1
        f_1 = temp + f_1
        return temp
    }
}

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

Variables

```
var c, python, java bool  //all bools, default = false

func main() {
    var i int  //int default = 0
    fmt.Println(i, c, python, java)
}
```

initialized:

```
var i, j int = 1, 2

func main() {
    var c, python, java = true, false, "no!"  //type optional w/ init
    fmt.Println(i, j, c, python, java)
}
```

quick initializing shorthand

```
var i, j int = 1, 2  // := not usable outside functions

func main() {
    k := 3
    c, python, java := true, false, "no!"
    fmt.Println(i, j, k, c, python, java)
}
```

Types

example of () blocks for import and var

```
package main

import (
    "fmt"
    "math/cmplx"
)

var (
    ToBe   bool       = false
    MaxInt uint64     = 1<<64 - 1
    z      complex128 = cmplx.Sqrt(-5 + 12i)
    s string
)

func main() {
    fmt.Printf("Type: %T Value: %v\n", ToBe, ToBe)
    fmt.Printf("Type: %T Value: %v\n", MaxInt, MaxInt)
    fmt.Printf("Type: %T Value: %v\n", z, z)
    fmt.Printf("Type: %T Value: %q\n", s, s)
}
```

```
output:
Type: bool          Value: false
Type: uint64        Value: 18446744073709551615
Type: complex128    Value: (2+3i)
Type: string        Value: ""
```

all types in Go:

```
bool //default=false

string //default=""

int  int8  int16  int32  int64  //int = 64 bit on 64-bit system
uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

rune // alias for int32
     // represents a Unicode code point

float32 float64

complex64 complex128
```

Type conversion

```
func main() {
    i := 42
    f := float64(i)
    u := uint(f)
    fmt.Println(x, y, z)
}
```

Type inference

```
var i int
j := i // j is also int (whatever type i was)

i := 42           // int
f := 3.142        // float64
g := 0.867 + 0.5i // complex128
```

constants

```
const Truth = true  // cannot use :=

var bb = 1 << 300 // error overflow
const bb = 1 << 300  //works because const untyped values
fmt.Println(bb) // error, tries to inference to int64 but too big
fmt.Println(bb*1.0) //works by converting to float64
```

Flow control

Loops

```
sum := 0
for i := 0; i < 10; i++ {  // {} always required
    sum += i // i only visible inside for loop
}

// all 3 for args optional

//while loop:
sum := 1
for sum < 1000 {
    sum += sum
}

// forever loop
for {}
```

If

```
func sqrt(x float64) string {
    if x < 0 { return sqrt(-x) + "i" }
    return fmt.Sprint(math.Sqrt(x))  //fmt Spring formats into str
}

// can declare before evaluating if condition like declaring i in for loop
func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    } else {
        fmt.Printf("%g >= %g\n", v, lim)
    }
    return lim  // v only within if-else statement scope
}
```

Exercise

```
// computing sqr root by looping 10 times
func Sqrt(x float64) float64 {
    z := 1.0
    for i := 1; i <= 10; i++ {
        z -= (z*z - x) / (2*z)
        fmt.Println(z)
    }
    return z
}

// compute sqr root by stopping once new guess changes very little.
func Sqrt(x float64) float64 {
    z := 1.0
    lim := 0.00001
    for {
        v := z - (z*z-x)/(2*z)
        if math.Abs(v-z) < lim { break }
        z = v
        fmt.Println(z)
    }
    return z
}

func main() {
    fmt.Println(Sqrt(1), math.Sqrt(1))
    fmt.Println(Sqrt(2), math.Sqrt(2))
    fmt.Println(Sqrt(3), math.Sqrt(3))
    fmt.Println(Sqrt(253), math.Sqrt(253))
}
```

Switch

built-in “break” (no fall-through cases) cases can be vars not just consts or ints

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("When's Saturday?")
    today := time.Now().Weekday() //time from 2009-11-10 23:00:00 UTC
    switch time.Saturday {
        case today + 0:
            fmt.Println("Today.")
        case today + 1:
            fmt.Println("Tomorrow.")
        case today + 2:
            fmt.Println("In two days.")
        default:
            fmt.Println("Too far away.")
    }
}
```

always true switches == long if/else chains

```
t := time.Now()
switch {
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    default:
        fmt.Println("Good evening.")
}
```

Defer

delays execution until after its parent function exits

```
func main() {
    defer fmt.Println("world")
    fmt.Println("hello")
}

// output: hello world
```

multiple defers execute popping stack (reverse order)

```
func main() {
    fmt.Println("counting")

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

    fmt.Println("done")
}

output: counting, done, 9,8,7,6,5,4,3,2,1,0
```

More types

Pointers

```
var p *int
p := &i         // point to i
fmt.Println(*p) // read i
*p = 21         // set i
```

Structs

```
type Vertex struct {
    X int
    Y int
}

func main() {
    fmt.Println(Vertex{1, 2})
    v := Vertex{1, 2}

    p := &v         //points to struct v
    p.X = 1e9       // sets v.X.   shorthand to (*p).X
    fmt.Println(v)  // output: {1000000000 2}

    v2 := Vertex{X: 1}  // = {1,0}
    v3 = Vertex{}      // = {0,0}
    p  = &Vertex{1, 2} // p of type *Vertex
    fmt.Printf("Type: %T, Value: %v\n", p, p)
    //output: Type: *main.Vertex, Value: &{1 2}
}
```

Arrays

fixed size as size is part of the type

```
package main

import "fmt"

func main() {
    var a [2]string
    a[0] = "Hello"
    a[1] = "World"
    fmt.Println(a[0], a[1]) //Hello World
    fmt.Println(a) //[Hello World]

    primes := [6]int{2, 3, 5, 7, 11, 13}
    fmt.Println(primes) //[2 3 5 7 11 13]
}
```

Slices

indexing same as in Python

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

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

only pts to existing array, so underlying changes reflected in old slices

```
func main() {
    names := [4]string{"John", "Paul", "George", "Ringo"}
    a := names[1:3]      // [Paul George]

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

Slice literals

length = current len = `len(s)` capacity = len of underlying array = `cap(s)` if both are zero: nil slice: `s == nil` - otherwise panic: runtime error: slice bounds out of range with capacity 6 convenient printing:

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow { // can skip i or v:  for _, v :=....
        fmt.Println(i, v)   // index, copied value
    }
}
```

```
package main

import "fmt"

func main() {
    q := []int{2, 3, 5, 7, 11, 13} // creates array len 6, then slice
ontop
    r := []bool{true, false, true, true, false, true}

    s := []struct {i int, b bool}
    {
        {2, true},
        {3, false},
        {5, true},
        {7, true},
        {11, false},
        {13, true},
    }
}
```

Dynamically-Sized Slices

```
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]
}

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

2D

```
board := [][]string
{
    []string{"_", "_", "_"},
    []string{"_", "_", "_"},
    []string{"_", "_", "_"},
}
board[1][2] = "X"

for i := 0; i < len(board); i++ {
    fmt.Printf("%s\n", strings.Join(board[i], " "))
}
```

append dynamically (will auto-move and update memory if exceeds capacity)

```
var s []int         // nil slice
s = append(s, 0)    // [0]
s = append(s, 2, 3) // [0, 2, 3]
```

Example

```
package main

import "golang.org/x/tour/pic"

func Pic(dx, dy int) [][]uint8 {
    image := make([][]uint8, dy) //slice of length dy
    for y := 0; y < dy; y++ {
        row := make([]uint8, dx) //each dx slice of 8-bit unsigned
        for x := 0; x < dx; x++ { //fpr each element in row
            row[x] = uint8((x+y)/2)
        }
        image[y] = row
    }
    return image
}

func main() {
    pic.Show(Pic)
}
```

[[6e0aef73ce6873ce266bbd3f75ff5e5d_MD5.jpeg]] !
[[6e0aef73ce6873ce266bbd3f75ff5e5d_MD5.jpeg]] ## Maps declaring/initializing:

```
type Vertex struct {
    Lat, Long float64
}

// dynamic map
var m map[string]Vertex
m = make(map[string]Vertex)

//static map
var m = map[string]Vertex {
    "Bell Labs": Vertex{ 1,2},
    "Google": {37.42202, -122.08408}, //"Vertex" optional, already
declared
}
```

mutating:

```
//insert key
m["bell"] = Vertex{40.68, -74}

//retrieve copy of value
elem = m[key]

//delete
delete(m, key)

//test key exists
v, ok := m["Answer"]  //v=0, ok=false
m["Answer"] = 2
v, ok = m["Answer"]  //v=2, ok=true
```

Example, WordCounter

```
package main

import (
    "golang.org/x/tour/wc"
    "fmt"
    "strings"
)

func WordCount(s string) map[string]int {
    m := make(map[string]int)
    split_fields := strings.Fields(s)

    for _, v := range split_fields {
        m[v]++
    }
    return m
}

func main() {
    wc.Test(WordCount)
}
```