210224 A Tour of Go - Basics

Packages, variables and Functions

Packages

math.rand.Intn

```
package main

import (
    "fmt"
    "math/rand"
)

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

math.Sqrt

```
package main

import (
    "fmt"
    "math"
)

func main() {
fmt.Printf("Now you have %g problems.\n", math.Sqrt(7))
}
```

Exported names

A name is exported if it begins with a capital letter.

```
package main

import (
    "fmt"
    "math"
)

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

Functions

```
package main

import "fmt"

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

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

when two or more consecutive named function share a tpe, we can omit the type from all but the last.

```
package main

import "fmt"

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

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

Mutliple Results

```
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}
```

Named return values

```
package main

import "fmt"

func split(sum int)(x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return          // "naked" return
}

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

Variables

The `var` statement declares a list of variables; as in function argument lists, the type is last.

```
var c, python, java bool
```

Variables with initializers

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

Short Variable Declarations

```
k := 3
```

Basic types

```
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

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

float32 float64

complex64 complex128
```

```
var (
    ToBe Bool = false
    MatInt unint64 = 1<<64 - 1
)
```

Zero values

- `0` for numeric types
- `false` for boolean type
- `""` (the empty string) for strings

Type Conversions

```
var i int = 42
var f float64 = float64(i) // must do this explicitly
var u uint = uint(f)

// or simply
i := 42
f := float64(i)
u := unit(f)
```

Type Inference

When the right hand side of the declaration is typed, the new variable is of that same type:

```
var i int
j := i // j is an int
```

But when the right hand side contains an untyped numeric constant, the new variable may be an `int`, `float64`, or `complex128` depending on the precision of the constant:

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

Constants

```
const Pi = 3.14
```

Numeric Constants

An untyped constant takes the type needed by its context.

```
const (
    Big = 1 << 100
    Small = Big >> 99
)

func needInt(x int) int { return x*10 + 1 }
func needFloat(x float64) float64 {
    return x * 0.1
}

func main() {
    fmt.Println(needInt(Small)) // 21
    fmt.Println(needFloat(Small))  // 0.2
    fmt.Println(needFloat(Big)) // 1.2676506002282295e+29
}
```

Flow Control Statements

For

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

for is while

```
sum := 1
for sum < 1000 {
    sum += sum
}
fmt.Println(sum)
```

Loop forever

```
func main() {
    for {
    }
}
```

If

```
func sqrt(x float64) string {
    if x < 0 {
    return sqrt(-x) + "i"
    }
}
```

If with a short statement

```
func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    }
    return lim
}
```

Excercise: Loops and Functions

```
package main

import (
    "fmt"
    "math"
)

const eps = 1e-6

func abs(x float64) float64 {
    if x < 0 {
        return -x;
    }
    return x;
}

func sqrt(x float64) float64 {
    z := float64(1)
    prev_z := float64(0)
    for abs(prev_z - z) > eps {
        prev_z = z
        z -= (z * z - x) / (2 * z)
    }
    return z
}

func main() {
    fmt.Println(sqrt(2) - math.Sqrt(2)) // prints 0
}
```

Switch

```
fmt.Println("Go runs on ")
switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd,
        // plan 9, windows ...
        fmt.Printf("%s.\n", os)
}
```

Switch with no condition

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

Defer

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

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

Stacking defers

The deferred function calls are pushed onto a stack.

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

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

    fmt.Println("donw")
}
```

output:

```
counting
done
9
8
7
6
5
4
3
2
```

```
1
0
```

More Types: Structs, slices and maps

Pointer

A pointer holds the memory address of a value.

The type `*T` is a pointer to a `T` value.

```
var p *int
```

The `&` operator generates a pointer to its operand

```
i := 42
p = &i
```

The `*` operator denotes the pointer's underlying value.

```
fmt.Println(*p) // read i through the pointer p
*p = 21      // set i throught pointer p
```

```
i, j := 42, 2701

p := &i // p is a pointer to i
*p = 21 // set i through the pointer p
p = &j // pointer p to j
*p = *p / 37 // divide j by 37, accessed vai the pointer p
```

Struct

A `strcut` is a collection of fields

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

func main() {
    fmt.Println(Vertext{1,2})
}
```

Struct Fields

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

func main() {
    v := Vertex{1, 2}
    v.X = 4
    fmt.Println(v.X)
}
```

Pointers to structs

No explicit dereference is required

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

func main() {
    v := Vertex{1, 2}
    p := &v
    p.X = 1e9
    fmt.Println(v)
}
```

Struct Literals

- A newly allocated struct value by listing the values of its fields.

```
type Vertext struct {
    X, Y int
}

var (
    v1 = Vertex{1, 2} // has type Vertex
    v2 = Vertex{X: 1} // Y:0 is implicit
    v3 = Vertex{}   // X:0 and Y:0
    p = &Vertex{1, 2} // has type *Vertex
)

fun main() {
    fmt.Println(v1, p, v2, v3)
}
```

Arrays

The length is part of its type, so arrays cannot be resized

```
var a [10]int
```

```
var a[2] string
a[0] = "Hello"
a[1] = "World"
fmt.Println(a[0], a[1])
fmt.Println(a)

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

Slices

The type []T is a slice with elements of type T .

```
a[low: high]
```

```
[6]int{2,3,5,7,7,11,13}

var s []int = primes[1:4]

fmt.Println(s) // [3 5 7]
```

Slices are references

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

    fmt.Println(names) // [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]
}
```

Slice Literals

```
[]bool{true, false, false}
```

```
// [2 3 5 7 11 13]
q := [] int {2, 3, 5, 7, 11, 13}
// [true false true true false true]
r := []bool{true, false, true, true, false, true}

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

Slice defaults

For the array

```
var a [10]int
```

these slice expressions are equivalent:

```
a[0:10]
a[:10]
a[0:]
a[:]
```

Slice length and capacity

We can extend a slices' length, provided it has sufficient capacity.

```
package main

import "fmt"

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

    // Slice the slice to give it zero length.
    s = s[:0]
    printSlice(s)

    // Extend its length.
    s = s[:4]
    printSlice(s)

    // Drop its first two values.
    s = s[2:]
    printSlice(s)
}

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

```
len=6 cap=6 [2 3 5 7 11 13]
len=0 cap=6 []
len=4 cap=6 [2 3 5 7]
len=2 cap=4 [5 7]
```

Nil Slices

The zero value of a slice is nil .

Creating a slice with make

The `make` function allocates a zeroed array and returns a slice that refers to that array:

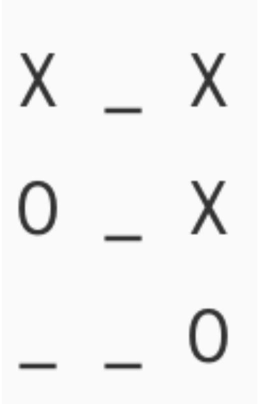
```
a := make([]int, 5)  // len(a)=5
```

To specify a capacity, pass a third argument to `make`:

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

b = b[:cap(b)] // len(b)=5, cap(b)=5
b = b[1:]      // len(b)=4, cap(b)=4
```

Slices of slices

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    // Create a tic-tac-toe board.
    board := [][]string{
        []string{"_", "_", "_"},
        []string{"_", "_", "_"},
        []string{"_", "_", "_"},
    }

    // The players take turns.
    board[0][0] = "X"
    board[2][2] = "O"
    board[1][2] = "X"
    board[1][0] = "O"
    board[0][2] = "X"

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



```
X _ X
O _ X
_ _ O
```

append

```
var s []int
s = append(s, 0)
s = append(s, 1)
s = append(s, 2, 3, 4)
s = append(s, 5, 6)
```

```
len=0 cap=0 []
len=1 cap=1 [0]
```

```
len=2 cap=2 [0 1]
len=5 cap=6 [0 1 2 3 4]
len=7 cap=12 [0 1 2 3 4 5 6]
```

range

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

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

```
2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
2**6 = 64
2**7 = 128
```

variations of range

You can skip the index or value by assigning to `_`.

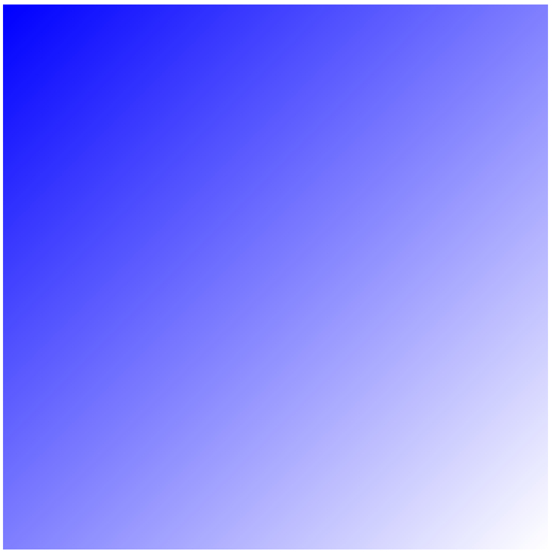
```
for i, _ := range pow
for _, value := range pow
```

If you only want the index, you can omit the second variable.

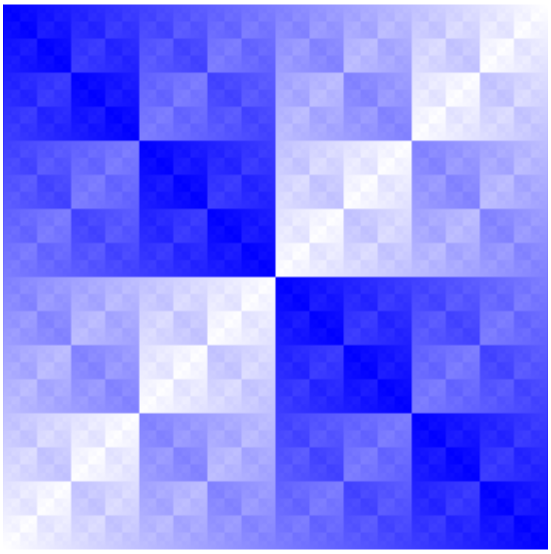
```
for i := range pow
```

Excercise

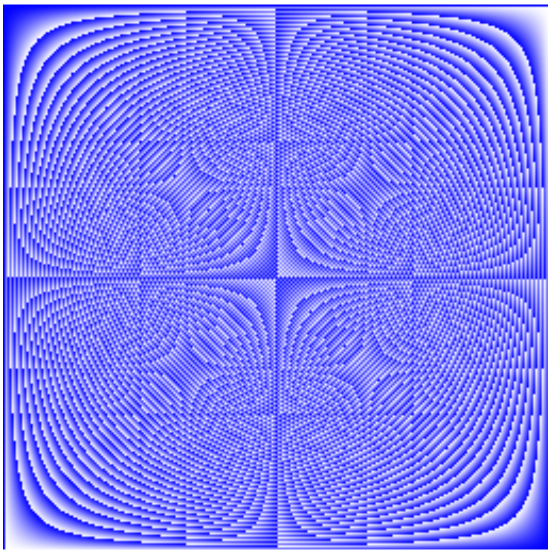
```
package main

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

func Pic(dx, dy int) [][]uint8 {
    var rets [][]uint8;

    for x := 0; x != dx; x++ {
        row := make([]uint8, dy)
        for y := 0; y != dy; y++ {
            row[y] = uint8((x + y) / 2) // key line
        }
        rets = append(rets, row)
    }
    return rets
}

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



```
row[y] = uint8(x^y)
```



```
row[y] = uint8(x*y)
```



map

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

var m map[string] Vertex
```

```
func main() {
    m = make(map[string] Vertex)
    m["Bell Labs"] = Vertex{
        40.68433, -74.39967,
    }
    fmt.Println(m["Bell Labs"]) // {40.68433 -74.39967}
}
```

Map literals

```
package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m = map[string]Vertex{
    "Bell Labs": Vertex{
        40.68433, -74.39967,
    },
    "Google": Vertex{
        37.42202, -122.08408,
    },
}

func main() {
    fmt.Println(m)
}
```

If the top-level type is just a type name, you can omit it from the elements of the literal.

```
package main

import "fmt"
```

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

var m = map[string]Vertex{
    "Bell Labs": {40.68433, -74.39967},
    "Google":    {37.42202, -122.08408},
}

func main() {
    fmt.Println(m)
}
```

Mutating Maps

Insert or update an element in map `m`:

```
m[key] = elem
```

Retrieve an element:

```
elem = m[key]
```

Delete an element:

```
delete(m, key)
```

Test that a key is present with a two-value assignment:

```
elem, ok = m[key]
```

If `key` is in `m`, `ok` is `true`. If not, `ok` is `false`.

If `key` is not in the map, then `elem` is the zero value for the map's element type.

Note: If `elem` or `ok` have not yet been declared you could use a short declaration form:

```
elem, ok := m[key]
```

```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)
}
```

Exercise: Maps

Implement `WordCount`. It should return a map of the counts of each “word” in the string `s`. The `wc.Test` function runs a test suite against the provided function and prints success or failure.

```go
package main

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

func WordCount(s string) map[string]int {
    m := make(map[string]int)
    words := strings.Fields(s)
    for _, word := range words {
        m[word]++
    }
    return m
}

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

Function values

```
package main

import (
    "fmt"
    "math"
)

func compute(fn func(float64, float64) float64) float64 {
    return fn(3, 4)
}

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

    fmt.Println(compute(hypot))
    fmt.Println(compute(math.Pow))
}
```

```
13
5
81
```

Function Closures

A closure is a function value that references variables from outside its body.

```go
package main

import "fmt"

func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    pos, neg := adder(), adder()
    for i := 0; i < 10; i++ {
        fmt.Println(
            pos(i),
            neg(-2*i),
        )
    }
}
```

Exercise: Fibonacci closure

```go
package main

import "fmt"

// fibonacci is a function that returns
// a function that returns an int.
func fibonacci() func() int {
    ptr := 0
    mem := make([]int, 11)
    mem[0] = 0
    mem[1] = 1
    return func() int {
        if ptr >= 2 {
            mem[ptr] = mem[ptr-2] + mem[ptr-1]
        }
        ptr++
        return mem[ptr-1]
    }
}

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

```
0
1
1
2
3
5
8
13
21
34
```