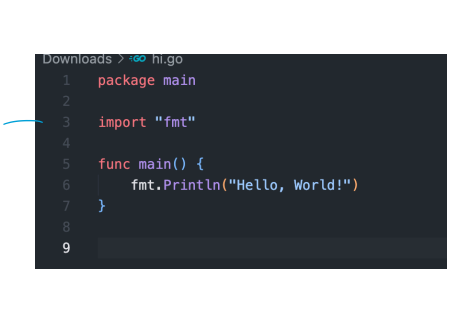
Go_basics

Thursday, January 9, 2025 10:37

1. Intro

- Faster than JS. Python. Ruby. PHP in execution speed.
- Faster than Rust. C. C++. C#. Java in compilation speed.
- Go:
  - strong & statically strong → type of var can't change often. and have to define before run
  - excellent community.
  - key feature: 1° simplicity.
    - 2° fast compile time.
    - 3° garbe collected → don't need to manage memory
    - 4° built-in concurrency
    - 5° compile to standalone binaries.
    - ↳ no need to worry about external libraries.

"fmt" for format



```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

the classic hello.world.

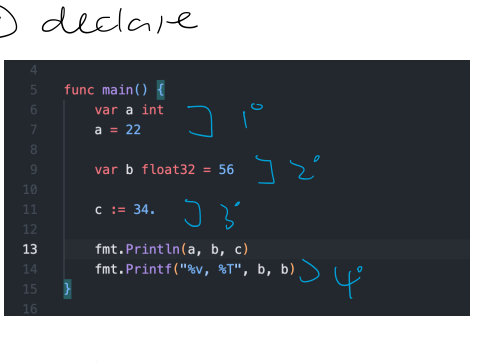
- GOROOT. GOPATH

正课开始.

2. Variables

- declare
- redeclaration & shadowing
- visibility
- naming conventions
- type conversion.

① declare



```
func main() {
    var a int
    a = 22

    var b float32 = 56

    c := 34.

    fmt.Println(a, b, c)
    fmt.Printf("%v, %T", b, b)
}
```

1° 先declare, 但不赋val

2° 直给

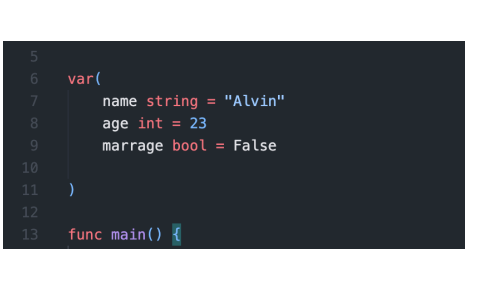
3° 直接给. 34. = default type: float64

34 = int

4° %v variable

%T type

if 是package level declaration, 3° 不能用



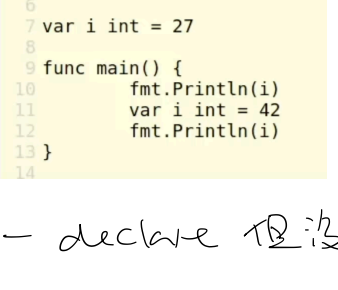
```
var(
    name string = "Alvin"
    age int = 23
    marrage bool = False

)

func main() {
```

什要一起declare, 就能更efficient.

② redeclare

- package level 和 main 两次 declare ✓
- main 里面两次 declare ✗

↳ 有priority, AKA shadowing



```
var i int = 27

func main() {
        fmt.Println(i)
        var i int = 42
        fmt.Println(i)
}
```

→ output:
27
42

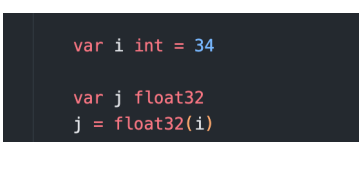
- declare 但没用, 会报错.

③ visibility

- 只有3种:
  - 1° package level w/ lower case
    - 任何file in package 可见
  - 2° package ·· upper case
    - globable visible, package 之外也可见
  - 3° block scope.
    - 在block之间的variable只能在block中可见
- visibility = life span of variable
- 命名规则: 1° 缩写全部大写 2° Pascal / camelCase 3° longer name for longer lifespan.

PascalCase
camelCase
snake_case

④ Conversion.

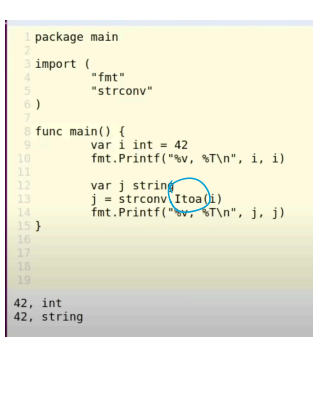


```
var i int = 34

var j float32
j = float32(i)
```

convert int to float32.

- 当float 变int 时, 可能会lose info. decimal part会丢失. 而这不会报错, 要自己注意.
- int/float 变string 时要用 string conversion package. 不然,就会变码 *

import "strconv"

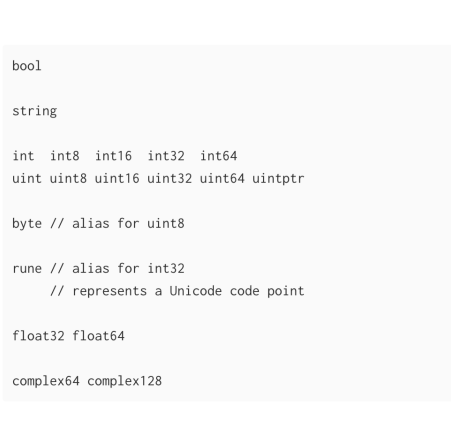


```
package main

import (
        "fmt"
        "strconv"
)

func main() {
        var i int = 42
        fmt.Printf("%v, %T\n", i, i)

        var j string
        j = strconv.Itoa(i)
        fmt.Printf("%v, %T\n", j, j)
}

42, int
42, string
```

Int to ASCII string

3. Primitives

- boolean type
- numeric types (integer, floating point, complex number)
- text type



```
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
```

default
false
" "
0

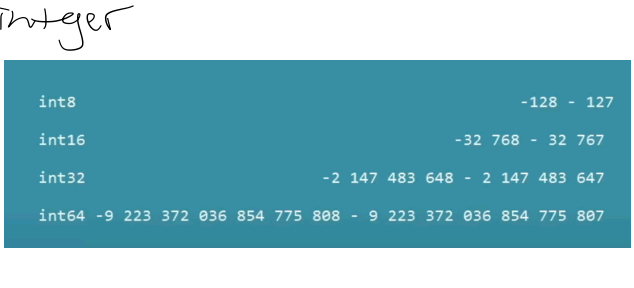
1. boolean

true. false

用处: ① logical test

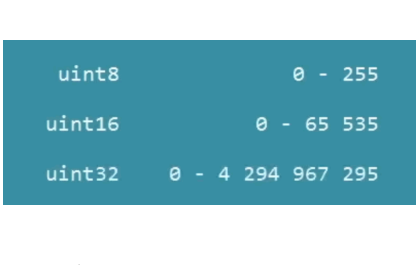
② mistake flag

2. Integer



```
int8                                  -128 - 127
int16                              -32 768 - 32 767
int32                   -2 147 483 648 - 2 147 483 647
int64 -9 223 372 036 854 775 808 - 9 223 372 036 854 775 807
```

$-2^8 \sim 2^{8}-1)$



```
uint8             0 - 255
uint16            0 - 65 535
uint32    0 - 4 294 967 295
```

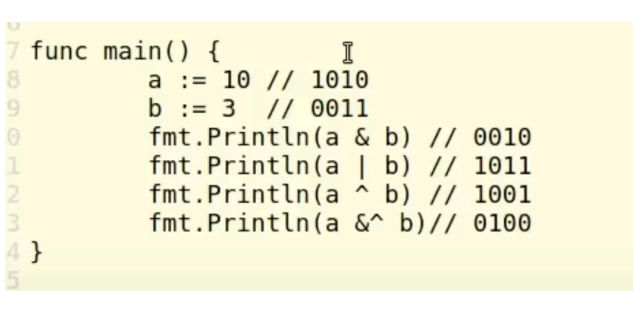
unsigned int

only same-type var can do calculation.

- bit operation

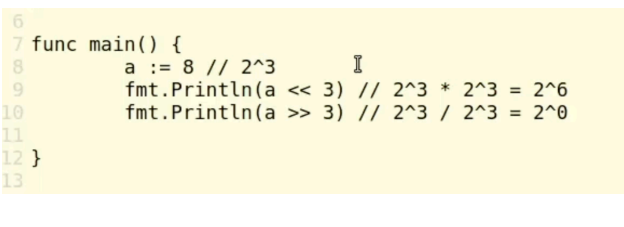
& | ^ &^

AND OR NOR ANDNOT



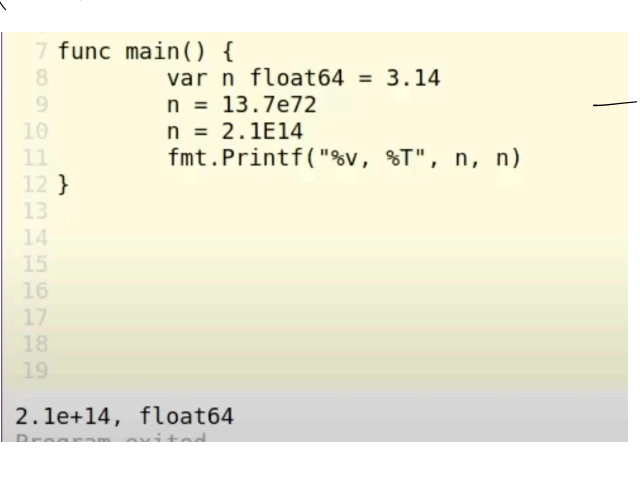
```
func main() {
        a := 10 // 1010
        b := 3  // 0011
        fmt.Println(a & b) // 0010
        fmt.Println(a | b) // 1011
        fmt.Println(a ^ b) // 1001
        fmt.Println(a &^ b)// 0100
}
```

- bit shifting



```
func main() {
      a := 8 // 2^3
      fmt.Println(a << 3) // 2^3 * 2^3 = 2^6
      fmt.Println(a >> 3) // 2^3 / 2^3 = 2^0
}
```

3. float



```
func main() {
        var n float64 = 3.14
        n = 13.7e72
        n = 2.1E14
        fmt.Printf("%v, %T", n, n)
}

2.1e+14, float64
```

— floating point

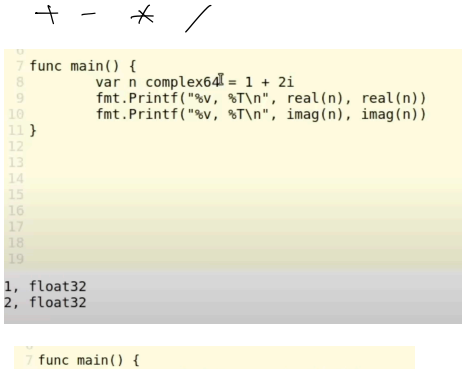
default type: float64

4. Complex number

- complex64 complex128

# of digits for the real & imaginary part.

- + - * /

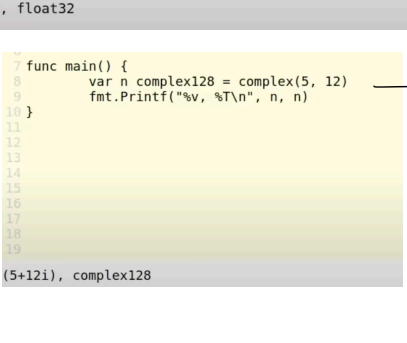


```
func main() {
        var n complex64 = 1 + 2i
        fmt.Printf("%v, %T\n", real(n), real(n))
        fmt.Printf("%v, %T\n", imag(n), imag(n))
}

1, float32
2, float32
```

var a: complex64

real(a) : float32

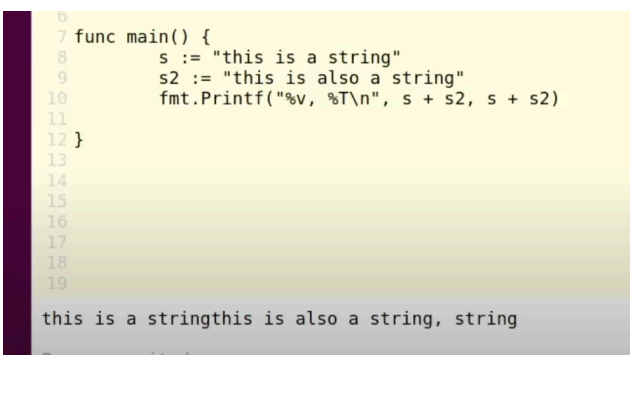
imag(a) : float32



```
func main() {
        var n complex128 = complex(5, 12)
        fmt.Printf("%v, %T\n", n, n)
}

(5+12i), complex128
```

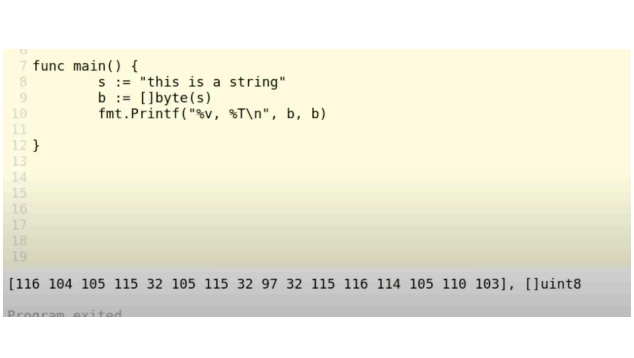
— 另一种 def

5. String



```
func main() {
        s := "this is a string"
        s2 := "this is also a string"
        fmt.Printf("%v, %T\n", s + s2, s + s2)
}

this is a stringthis is also a string, string
```

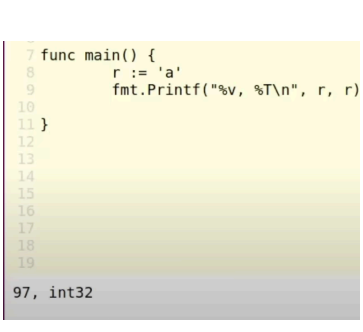
append



```
func main() {
        s := "this is a string"
        b := []byte(s)
        fmt.Printf("%v, %T\n", b, b)
}

[116 104 105 115 32 105 115 32 97 32 115 116 114 105 110 103], []uint8
```

type conversion

ASCII.



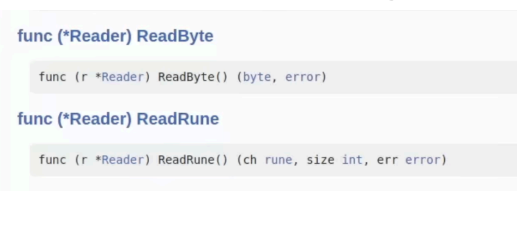
```
func main() {
        r := 'a'
        fmt.Printf("%v, %T\n", r, r)
}

97, int32
```

" " 和 ' ' 不同

' ' 叫 rune, 就是 int32

" " 叫 string, 可以来回换

想要 rune, string 转换. 要用 API. (golang.org 可查)



func (*Reader) ReadByte

```
func (r *Reader) ReadByte() (byte, error)
```

func (*Reader) ReadRune

```
func (r *Reader) ReadRune() (ch rune, size int, err error)
```

4. Constants

- Naming convention
- Typed constants
- Untyed constants
- Enumerated constants
- Enumeration expression

1. naming convention

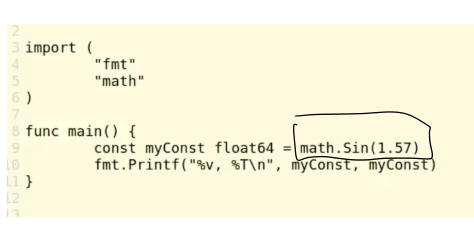
不用 const MY_CONST, 这种全部大写, 用_分隔 in the Go 中会因为首字母大写表示 want to export this symbol.

所以还是用 camelCase

2. typed constant

constants with defined type

Eg. const monthNumber int = 12



```
import (
        "fmt"
        "math"
)

func main() {
        const myConst float64 = math.Sin(1.57)
        fmt.Printf("%v, %T\n", myConst, myConst)
}
```

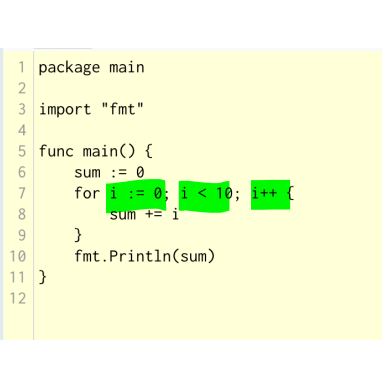
错误

值不能是 run time 才知道

5. Flow Control

- for
- if else
- switch
- defer

1. For



```
package main

import "fmt"

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

→ 0+1+2·· +9

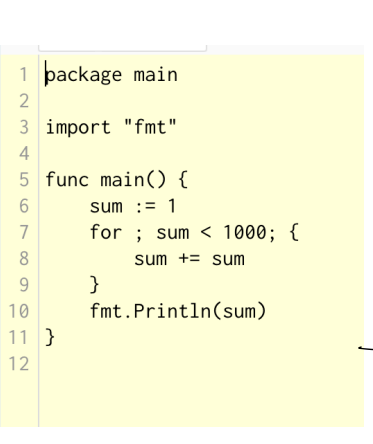
= (0+9)×10/2

= 45

① ② ③ ① init statement

② condition expression

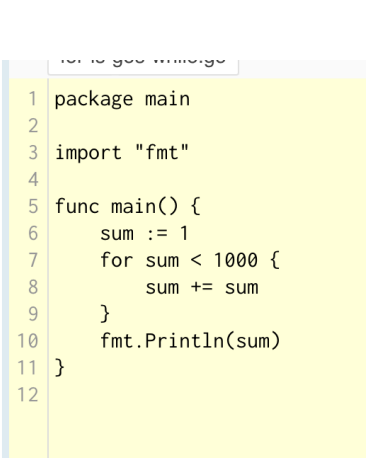
③ post statement



```
package main

import "fmt"

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

①. ③ are optional

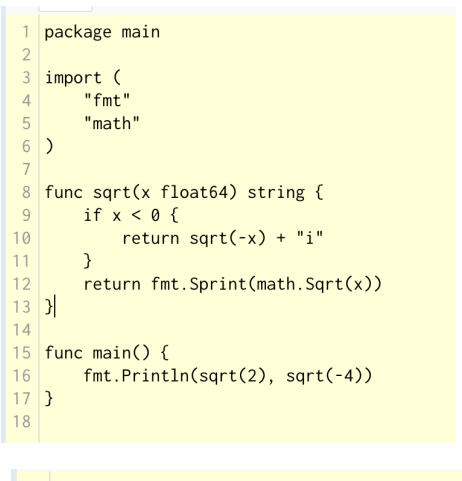
→ 1024 $2^0 \to 2^1 \to \cdots 2^{10}$



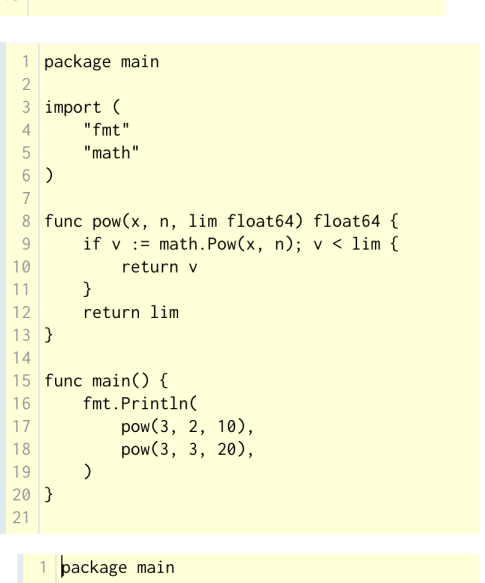
```
package main

import "fmt"

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

用 for 代替 while

2. if



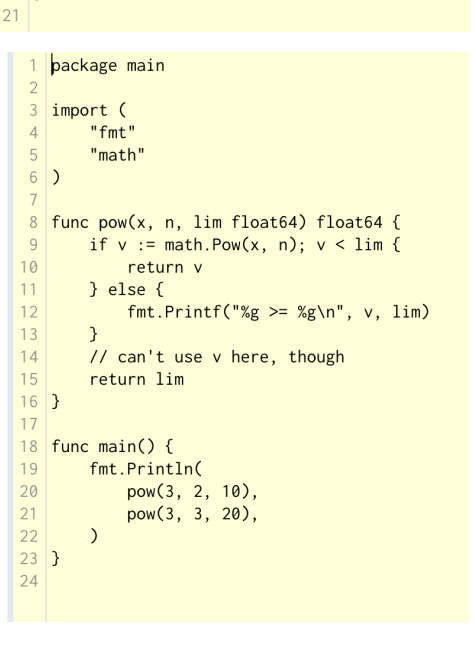
```
package main

import (
    "fmt"
    "math"
)

func sqrt(x float64) string {
    if x < 0 {
        return sqrt(-x) + "i"
    }
    return fmt.Sprint(math.Sqrt(x))
}

func main() {
    fmt.Println(sqrt(2), sqrt(-4))
}
```



```
package main

import (
    "fmt"
    "math"
)

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

func main() {
    fmt.Println(
        pow(3, 2, 10),
        pow(3, 3, 20),
    )
}
```

$3^2 = 9 < 10 \to 9$

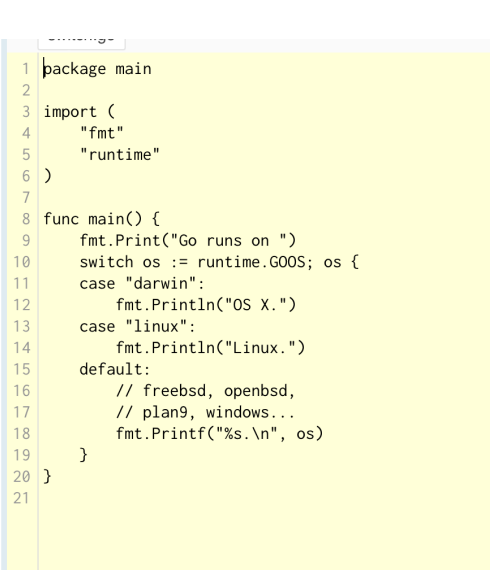
$3^3 = 27 > 20 \to 20$



```
package main

import (
    "fmt"
    "math"
)

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)
    }
    // can't use v here, though
    return lim
}

func main() {
    fmt.Println(
        pow(3, 2, 10),
        pow(3, 3, 20),
    )
}
```

if else

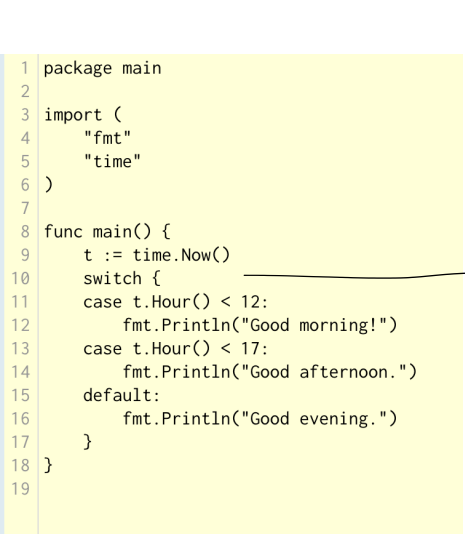
3. switch



```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("Go runs on ")
    switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd,
        // plan9, windows...
        fmt.Printf("%s.\n", os)
    }
}
```

① no need for break.
It is auto ~~added~~ when compile.

② cases can't be constants & value involved can't be int

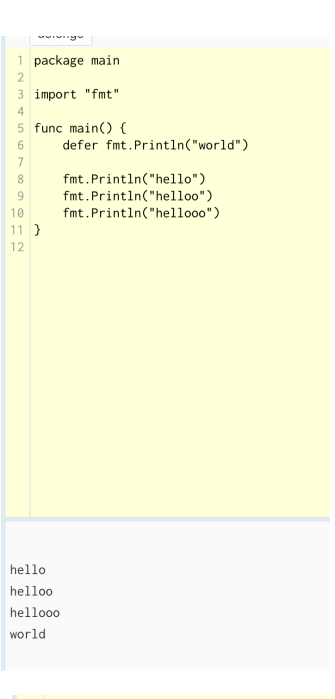


```
package main

import (
    "fmt"
    "time"
)

func main() {
    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.")
    }
}
```

→ switch without condition is same as switch true.

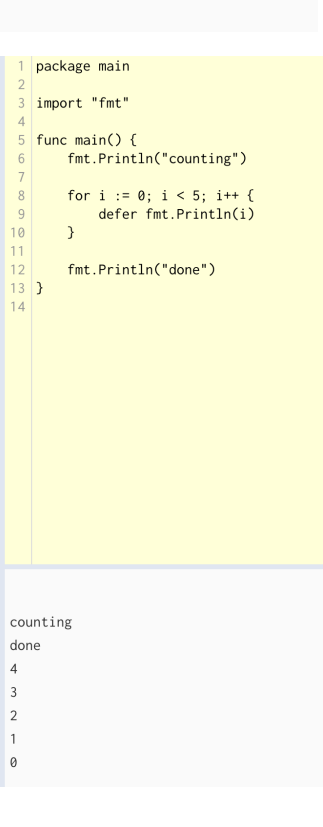
- run till killed

4. defer



```
package main

import "fmt"

func main() {
    defer fmt.Println("world")

    fmt.Println("hello")
    fmt.Println("helloo")
    fmt.Println("hellooo")
}

hello
helloo
hellooo
world
```

defer statement defers the execution of a function untill the surrounding function returns



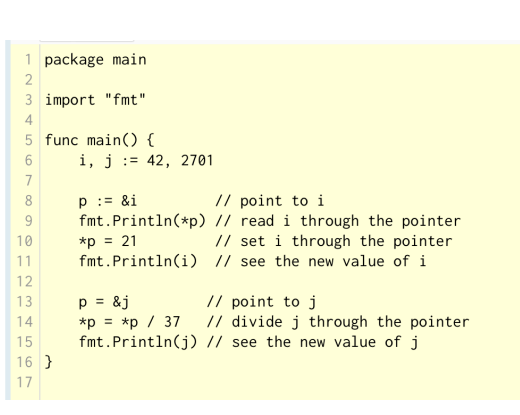
```
package main

import "fmt"

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

    for i := 0; i < 5; i++ {
        defer fmt.Println(i)
    }

    fmt.Println("done")
}

counting
done
4
3
2
1
0
```

stacking defer

LIFO

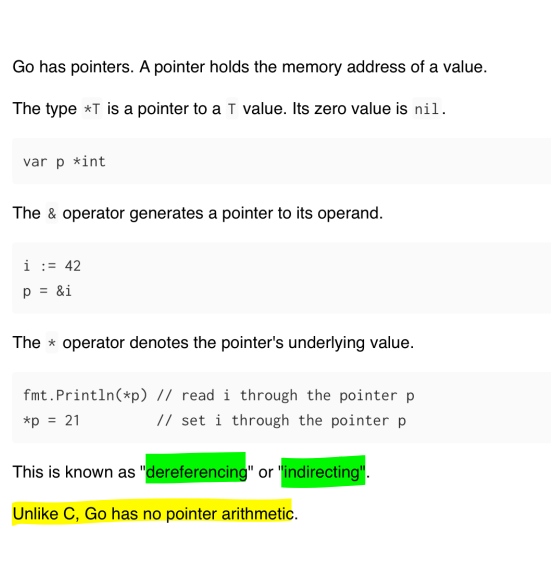
6. More types: structs, slices, and maps.

1. Pointers.



```
package main

import "fmt"

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

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

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

Output: 42
21
73



Go has pointers. A pointer holds the memory address of a value.

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

```
var p *int
```

The `&` operator generates a pointer to its operand.

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

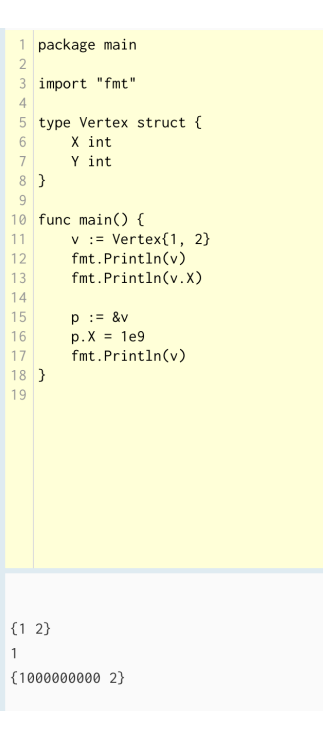
The `*` operator denotes the pointer's underlying value.

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

This is known as "dereferencing" or "indirecting".

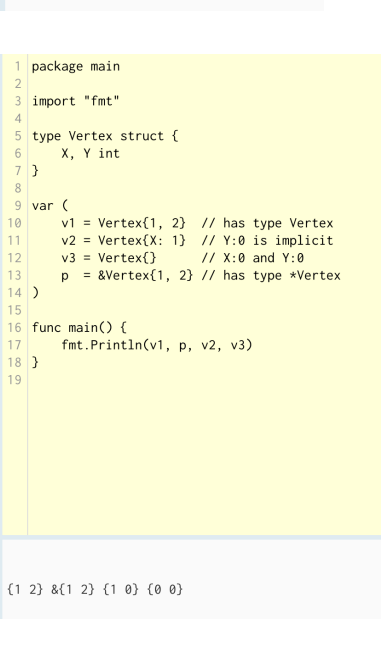
Unlike C, Go has no pointer arithmetic.

2. Structs



```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

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

    p := &v
    p.X = 1e9
    fmt.Println(v)
}

{1 2}
1
{1000000000 2}
```

- struct = collection of fields.
- use dot to access fields

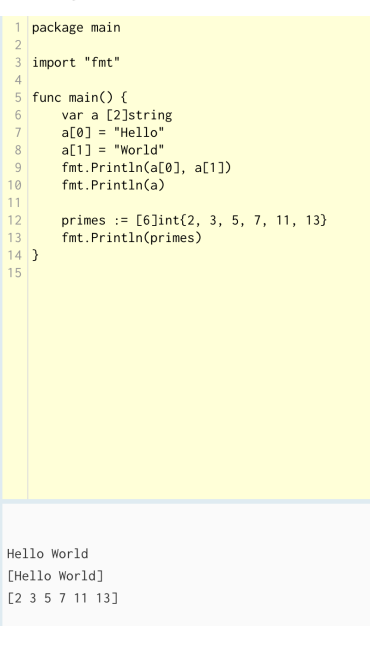


```
package main

import "fmt"

type Vertex 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
)

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

{1 2} &{1 2} {1 0} {0 0}
```

struct literal

[illegible]

3. Array

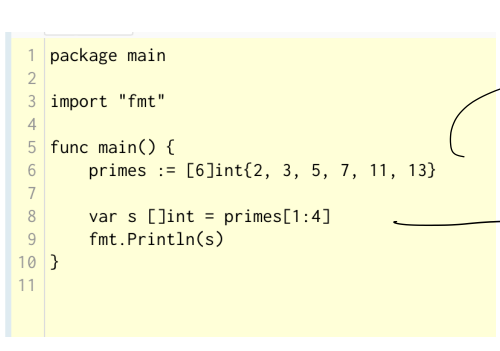


```
package main

import "fmt"

func main() {
    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)
}

Hello World
[Hello World]
[2 3 5 7 11 13]
```

var x [n]T

array of n values of type T.

4. Slices



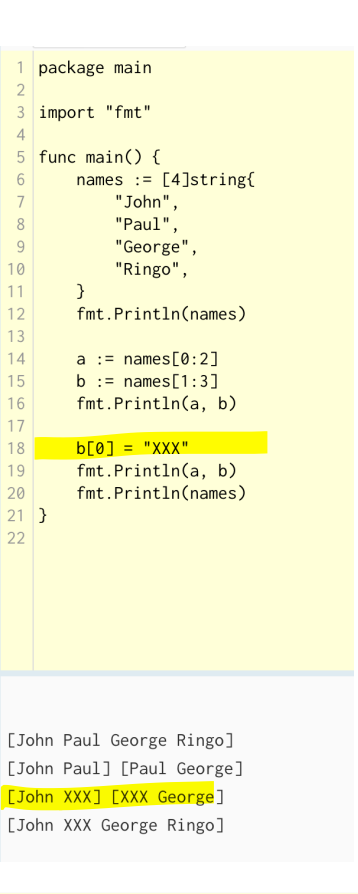
```
package main

import "fmt"

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

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

→ if the # of element is assure, no need to define length two.

→ slice时不用自己写 length

length = high − low

return [low, high)

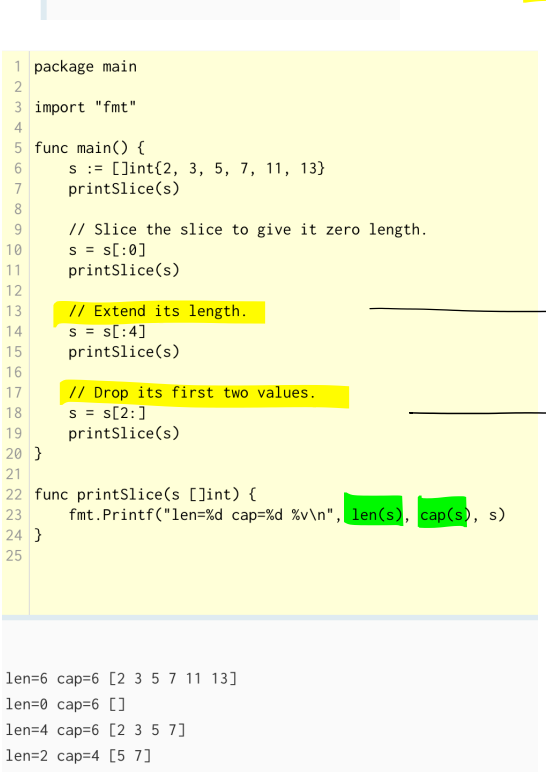


```
package main

import "fmt"

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

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

    b[0] = "XXX"
    fmt.Println(a, b)
    fmt.Println(names)
}

[John Paul George Ringo]
[John Paul] [Paul George]
[John XXX] [XXX George]
[John XXX George Ringo]
```

when b[0] change to "XXX"

a[1] is also changed ?

Changing element of a slice modifies the corresponding element of its underlying array.



```
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]
```

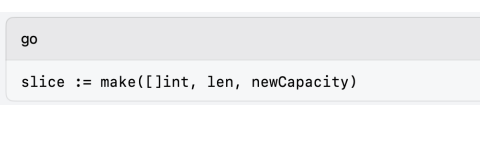
→ 不能超过 capacity.

→ this will cut down capacity.

To increase capacity:

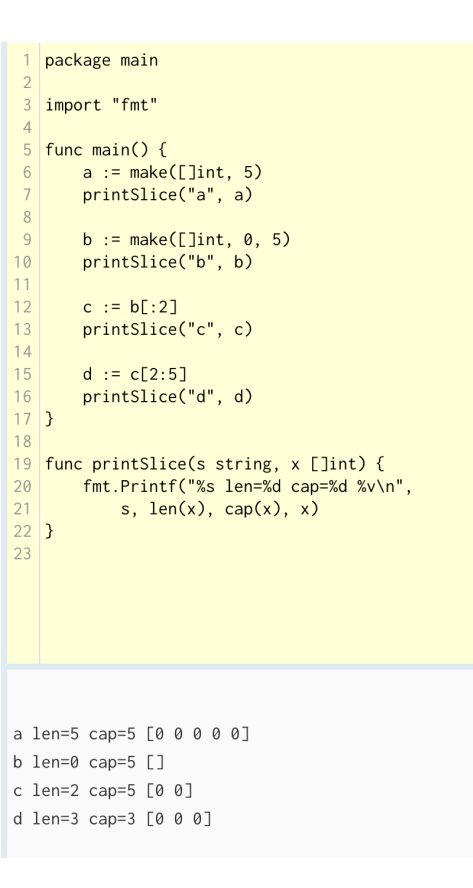
法1: 直接 append element.

法2: preallocate a larger capacity to minimize reallocations.



```
go

slice := make([]int, len, newCapacity)
```

用 nil 代替 NULL

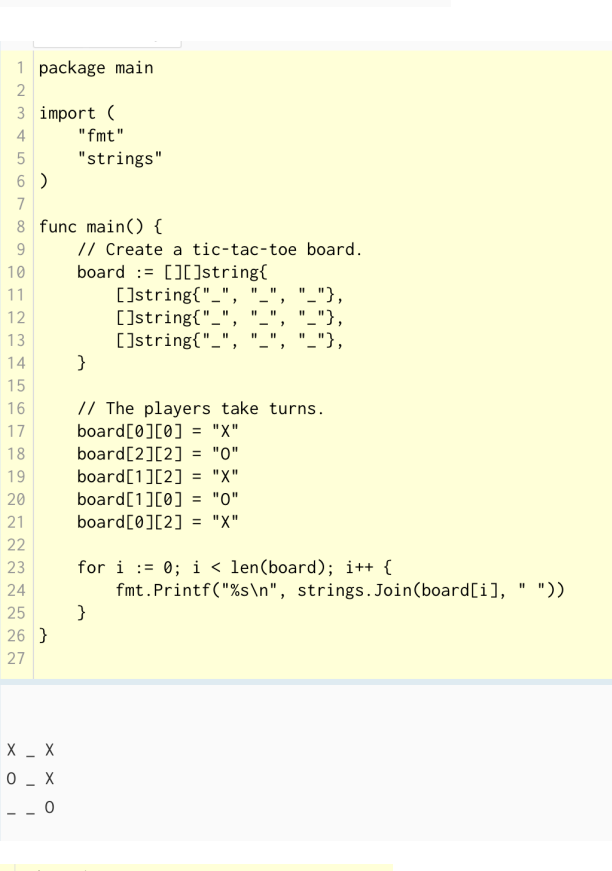


```
package main

import "fmt"

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

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

    c := b[:2]
    printSlice("c", c)

    d := c[2:5]
    printSlice("d", d)
}

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

a len=5 cap=5 [0 0 0 0 0]
b len=0 cap=5 []
c len=2 cap=5 [0 0]
d len=3 cap=3 [0 0 0]
```

create slice using make

make([]T, a, b)

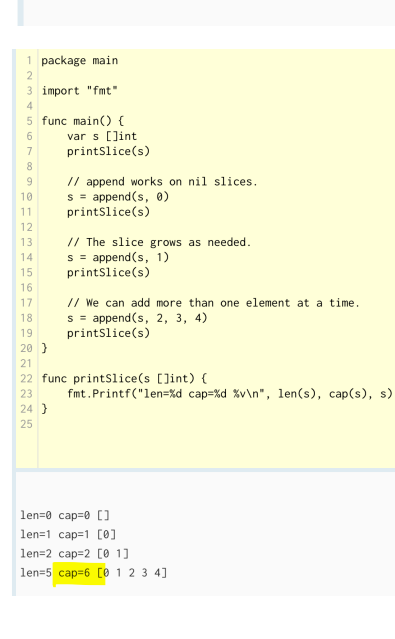
- array of type T
- length = a
- capacity = b
- if b is not specified, then b=a.



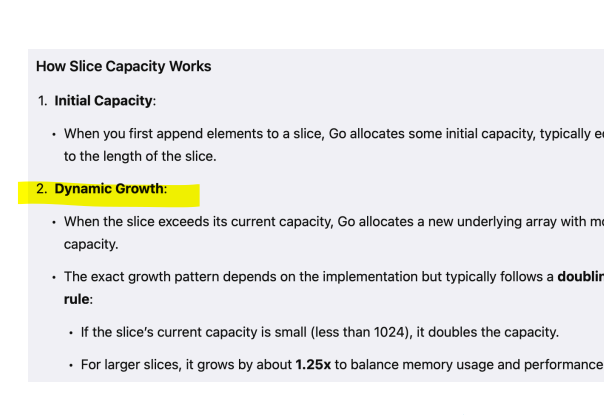
```
package main

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
```

slices of slices.

a 2D array.



```
package main

import "fmt"

func main() {
    var s []int
    printSlice(s)

    // append works on nil slices.
    s = append(s, 0)
    printSlice(s)

    // The slice grows as needed.
    s = append(s, 1)
    printSlice(s)

    // We can add more than one element at a time.
    s = append(s, 2, 3, 4)
    printSlice(s)
}

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

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

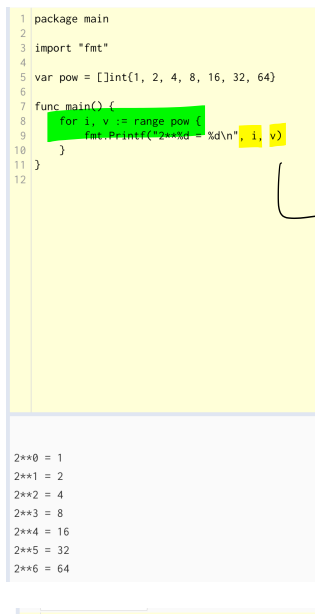


How Slice Capacity Works

1. **Initial Capacity:**
   - When you first append elements to a slice, Go allocates some initial capacity, typically equal to the length of the slice.
2. **Dynamic Growth:**
   - When the slice exceeds its current capacity, Go allocates a new underlying array with more capacity.
   - The exact growth pattern depends on the implementation but typically follows a **doubling rule**:
     - If the slice's current capacity is small (less than 1024), it doubles the capacity.
     - For larger slices, it grows by about **1.25x** to balance memory usage and performance.

感觉 cap 也没什么特别的意义.

5. Range

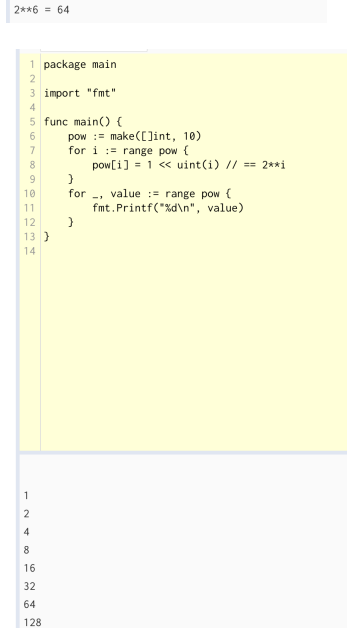


```
package main

import "fmt"

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

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
```

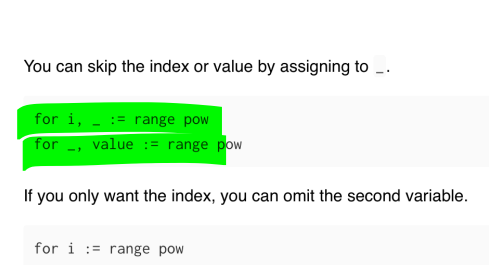
类似于 for n in pow

↳ i 是 index

v 是 pow[i] → v 其实可以不写, 直接用 pow[i].



```
package main

import "fmt"

func main() {
    pow := make([]int, 10)
    for i := range pow {
        pow[i] = 1 << uint(i) // == 2**i
    }
    for _, value := range pow {
        fmt.Printf("%d\n", value)
    }
}

1
2
4
8
16
32
64
128
```



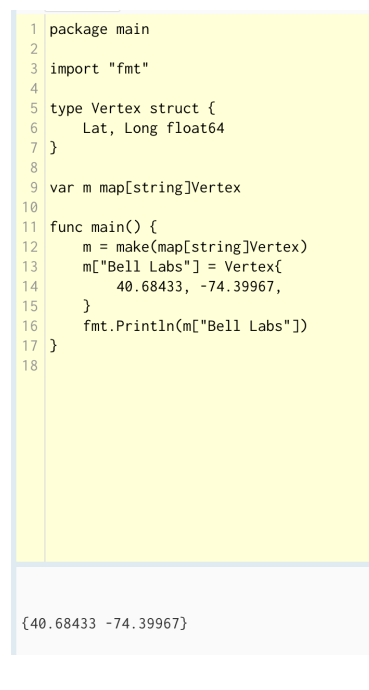
You can skip the index or value by assigning to `_`.

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

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

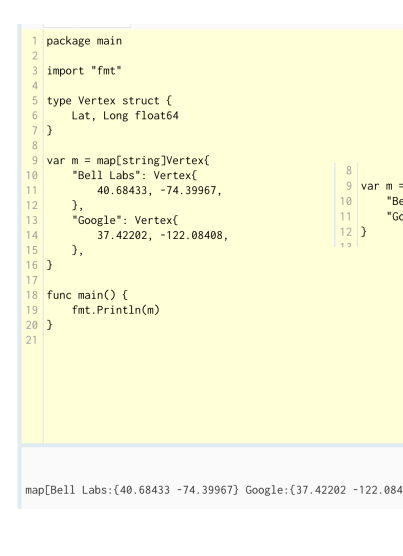
```
for i := range pow
```

6. Maps

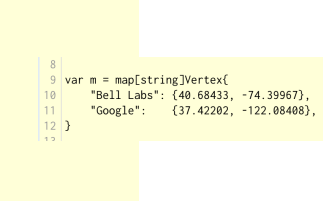


```
package main

import "fmt"

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}
```

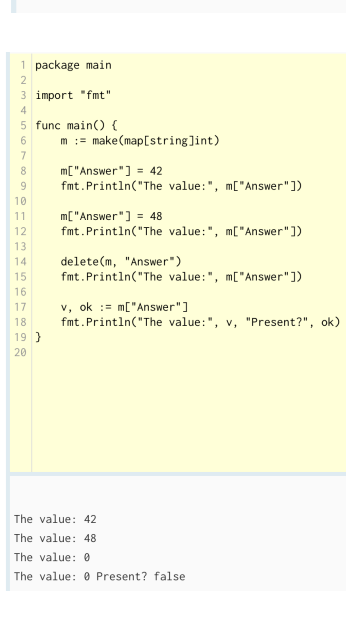
和 dictionary 差不多.



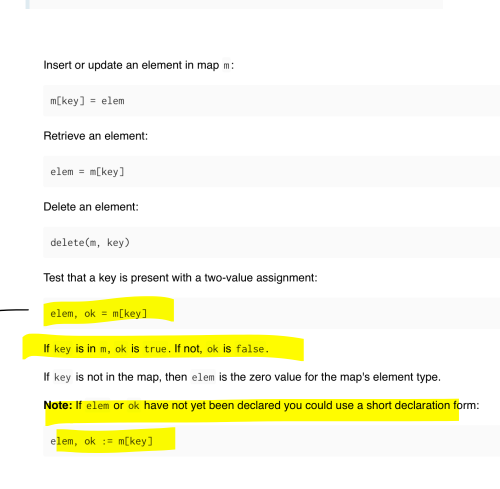
```
package main

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)
}

map[Bell Labs:{40.68433 -74.39967} Google:{37.42202 -122.08408}]
```



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



```
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)
}

The value: 42
The value: 48
The value: 0
The value: 0 Present? false
```



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`.

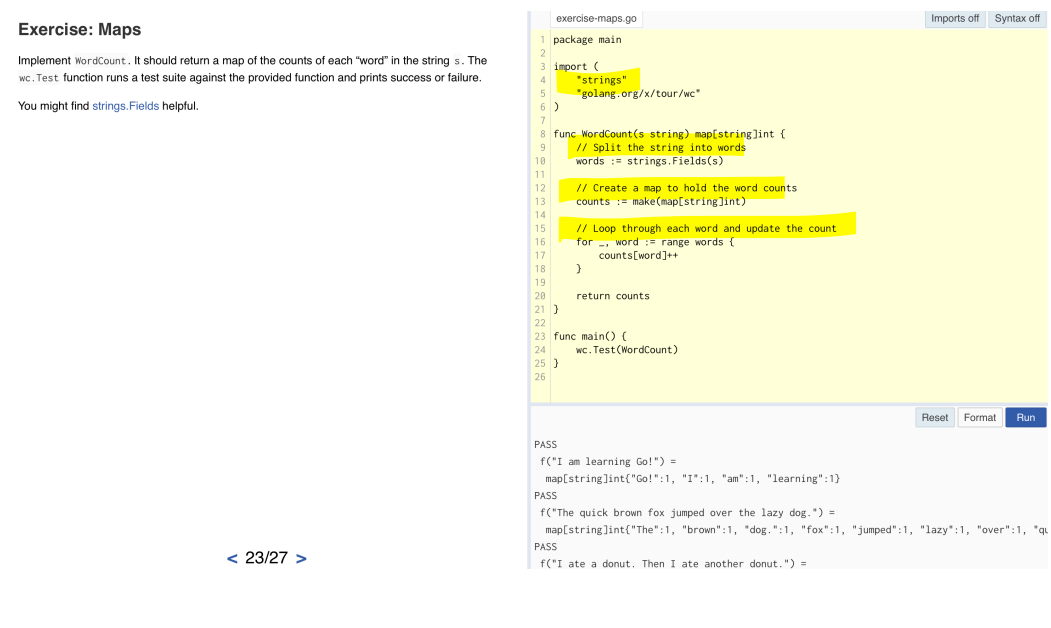
If `key` is not in the map, then `elem` is the zero value for the map's element type.

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

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

↓ 就是多一个 attribute 是 exist or not.

名字 var 啥都可以. 不是必须是 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.

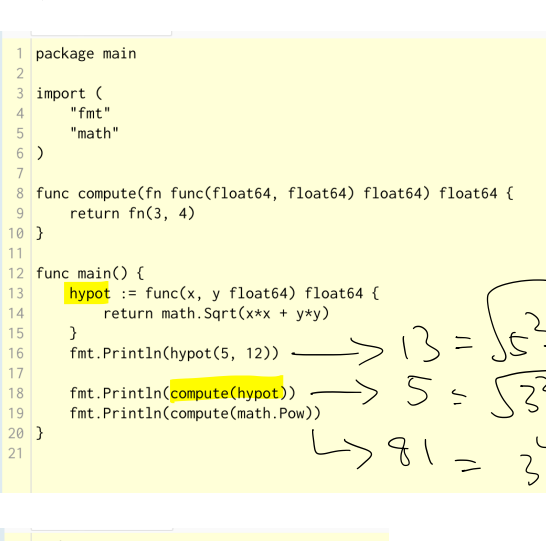
You might find strings.Fields helpful.

< 23/27 >

```
package main

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

func WordCount(s string) map[string]int {
    // Split the string into words
    words := strings.Fields(s)

    // Create a map to hold the word counts
    counts := make(map[string]int)

    // Loop through each word and update the count
    for _, word := range words {
        counts[word]++
    }

    return counts
}

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

```
PASS
 f("I am learning Go!") =
  map[string]int{"Go!":1, "I":1, "am":1, "learning":1}
PASS
 f("The quick brown fox jumped over the lazy dog.") =
  map[string]int{"The":1, "brown":1, "dog.":1, "fox":1, "jumped":1, "lazy":1, "over":1, "q
PASS
 f("I ate a donut. Then I ate another donut.") =
```

7. function values



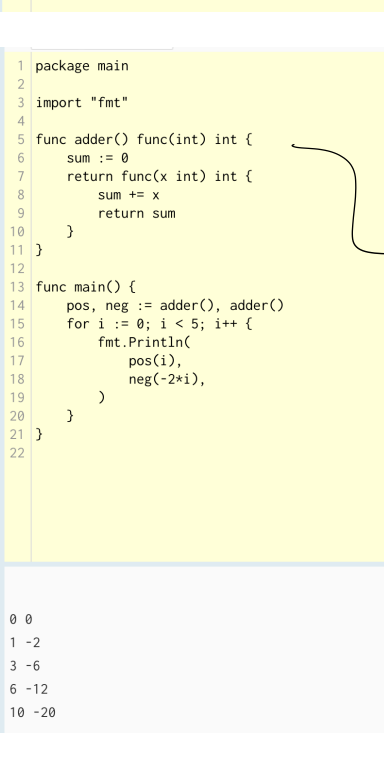
```
package main

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))
}
```

pass function as a value.

→ $13 = \sqrt{5^2+12^2}$

→ $5 = \sqrt{3^2+4^2}$

↳ $81 = 3^4$



```
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 < 5; i++ {
        fmt.Println(
            pos(i),
            neg(-2*i),
        )
    }
}

0 0
1 -2
3 -6
6 -12
10 -20
```

Function closure

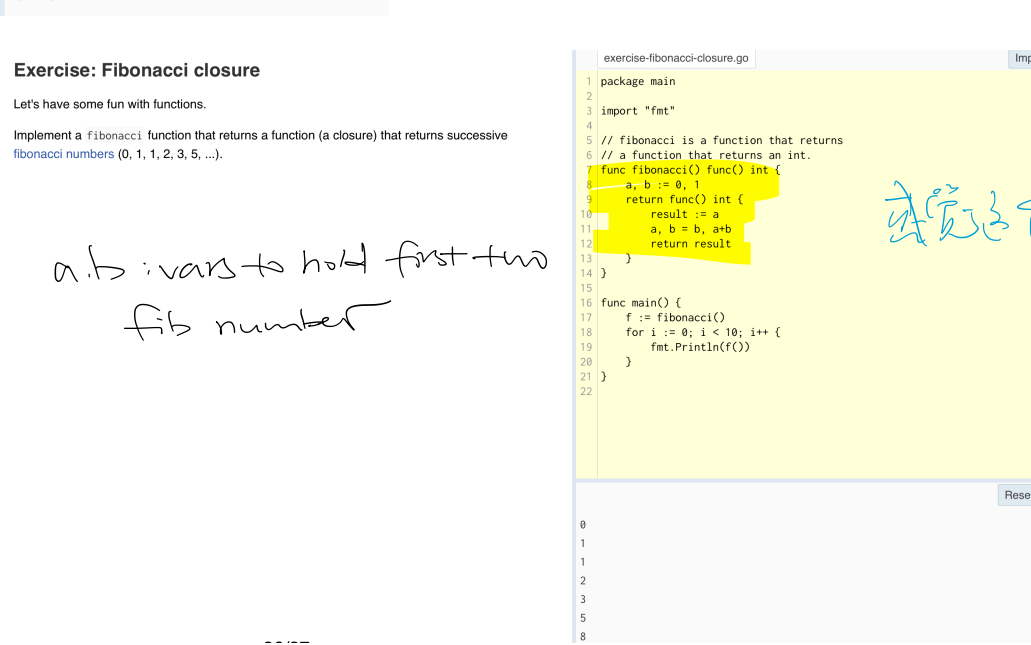
↳ adder 本身就是 return x itself.

i=0 pos(0) neg(0) → 0 0

i=1 pos(1) neg(-2) → 1 -2

i=2 pos(3) neg(-6) → 3 -6

⋮



Exercise: Fibonacci closure

Let's have some fun with functions.

Implement a `fibonacci` function that returns a function (a closure) that returns successive fibonacci numbers (0, 1, 1, 2, 3, 5, ...).

```
package main

import "fmt"

// fibonacci is a function that returns
// a function that returns an int.
func fibonacci() func() int {
    a, b := 0, 1
    return func() int {
        result := a
        a, b = b, a+b
        return result
    }
}

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

0
1
1
2
3
5
8
```

?

a.b: vars to hold first two fib number

感觉这个func 不会 step looping.