A Tour of Go

Jin Hyun Kim
Fall 2019

This Slides from

- https://tour.golang.org/

- http://golang.site/

In this class

- We will cover most of features of Go language so that you can create a small piece of program.

Hello World

- Install Go lang

- Install GoLanD from https:// www.jetbrains.com/idea/

- Create "helloworld" project

- Run

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Hello, World")
7  }
```

Run Go

- Compile and run

  - $ go build xxx.go

  - $ xxx

- Just run

  - $ go run xxx.go

1. Beginning

Package

```
1  package main
2
3  import (
4      "fmt"
5      "math/rand"
6  )
7
8  func main() {
9      fmt.Println("My favorite number is",
   rand.Intn(10))
10 }
11
```

- The package *main* is is a directive for compilers

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  func main() {
9      fmt.Printf("Now you have %g problems.\n",
   math.Sqrt(7))
10 }
11
```

- *import* read in a package

Function

```
 1  package main
 2
 3  import "fmt"
 4
 5  func add(x int, y int) int {
 6      return x + y
 7  }
 8
 9  func main() {
10      fmt.Println(add(42, 13))
11  }
12
```

In this example, we shortened

```
x int, y int
```

to

```
x, y int
```

- Notice how to declare a variable and a function

  - Function type - Return value type

  - Function parameters

Function

```
1  package main
2
3  import "fmt"
4
5  func swap(x, y string) (string, string) {
6      return y, x
7  }
8
9  func main() {
10     a, b := swap("hello", "world")
11     fmt.Println(a, b)
12 }
13
```

- Multiple results can be returned

Function

```go
1  package main
2
3  import "fmt"
4
5  func split(sum int) (x, y int) {
6      x = sum * 4 / 9
7      y = sum - x
8      return
9  }
10
11 func main() {
12     fmt.Println(split(17))
13 }
14
15
```

- Named results

  - Go's return values may be named. If so, they are treated as variables defined at the top of the function.

- A return statement without arguments returns the named return values. This is known as a "naked" return.

Variables

```
1  package main
2
3  import "fmt"
4
5  var c, python, java bool
6
7  func main() {
8      var i int
9      fmt.Println(i, c, python, java)
10 }
```

```
1  package main
2
3  import "fmt"
4
5  var i, j int = 1, 2
6
7  func main() {
8      var c, python, java = true, false, "no!"
9      fmt.Println(i, j, c, python, java)
10 }
11
```

- Use keyword *var* to declare variable types

- Initialize variables

Variables

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var i, j int = 1, 2
7      k := 3
8      c, python, java := true, false, "no!"
9
10     fmt.Println(i, j, k, c, python, java)
11 }
```

- Inside a function, the := short assignment statement can be used in place of a var declaration with implicit type.

- Outside a function,

  - := construct is not available.

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
```

Basic Types

```go
package main

import (
    "fmt"
    "math/cmplx"
)

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

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)
}
```

Basic Types

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var i int
7      var f float64
8      var b bool
9      var s string
10     fmt.Printf("%v %v %v %q\n", i, f, b, s)
11 }
12
```

- The zero value is:

  - **0** for numeric types,

  - **false** for the boolean type, and

  - **""** (the empty string) for strings.

Type Conversions

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  func main() {
9      var x, y int = 3, 4
10     var f float64 = math.Sqrt(float64(x*x + y*y))
11     var z uint = uint(f)
12     fmt.Println(x, y, z)
13 }
```

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

Some numeric conversions:

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

Or, put more simply:

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

Type Inference

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      v := 42 // change me!
7      fmt.Printf("v is of type %T\n", v)
8  }
```

- When declaring a variable without specifying an explicit type (either by using the := syntax or var = expression syntax),

  - the variable's type is inferred from the value on the right hand side.

Type Inference

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

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

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

- 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:

Constants

```go
1  package main
2
3  import "fmt"
4
5  const Pi = 3.14
6
7  func main() {
8      const World = "世界"
9      fmt.Println("Hello", World)
10     fmt.Println("Happy", Pi, "Day")
11
12     const Truth = true
13     fmt.Println("Go rules?", Truth)
14 }
```

- A constant variable can be one of character, string, and boolean

- Constants cannot be declared using the := syntax.

Constants

```go
package main

import "fmt"

func main() {
    var x, y, z int = 1, 2, 3
    c, python, java := true, false, "no!"

    fmt.Println(x, y, z, c, python, java)
}
```

- Initialize constant variables

Numeric Constants

```go
1  package main
2
3  import "fmt"
4
5  const (
6      // Create a huge number by shifting a 1 bit left 100 places.
7      // In other words, the binary number that is 1 followed by 100 zeroes.
8      Big = 1 << 100
9      // Shift it right again 99 places, so we end up with 1<<1, or 2.
10     Small = Big >> 99
11 )
12
13 func needInt(x int) int { return x*10 + 1 }
14 func needFloat(x float64) float64 {
15     return x * 0.1
16 }
17
18 func main() {
19     fmt.Println(needInt(Small))
20     fmt.Println(needFloat(Small))
21     fmt.Println(needFloat(Big))
22 }
23
```

- Numeric constants are high-precision values.

- An untyped constant takes the type needed by its context.

Operators

- Numeric : +, -, *, /, %

```
1  c = (a + b) / 5;
2  i++;
```

- Relational: ==, !=, >=

```
1  a == b
2  a != c
3  a >= b
```

- Logical: &&, ||, !

```
1  A && B
2  A || !(C && B)
```

- Bitwise: <<, >>

```
1  c = (a & b) << 5
```

- Assignment: += , &=, <<=

```
1  a = 100
2  a *= 10
3  a >>= 2
4  a |= 1
```

- Pointer

```
1  var k int = 10
2  var p = &k   //k의 주소를 할당
3  println(*p) //p가 가리키는 주소에 있는 실제 내용을 출력
```

2. Now, Walk on Go (Control)

For

```go
package main

import "fmt"

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

```go
package main

import "fmt"

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

- Iteration

- Go Lang have no more iteration than *for*

  - The basic for loop has three components separated by semicolons:

    - the *init* statement: executed before the first iteration

    - the *condition* expression: evaluated before every iteration

    - the *post* statement: executed at the end of every iteration

Like while

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       sum := 1
7       for sum < 1000 {
8           sum += sum
9       }
10      fmt.Println(sum)
11  }
```

- For is Go's "while"

Eternal Loop

- The same as

```python
while(True):
    print("Hello")
```

- in Python

```go
package main

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

if

```go
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))
}
```

- Conditional statements

- The expression need not be surrounded by parentheses ( ) but the braces { } are required.

if

```
1   package main
2
3   import (
4       "fmt"
5       "math"
6   )
7
8   func pow(x, n, lim float64) float64 {
9       if v := math.Pow(x, n); v < lim {
10          return v
11      }
12      return lim
13  }
14
15  func main() {
16      fmt.Println(
17          pow(3, 2, 10),
18          pow(3, 3, 20),
19      )
20  }
```

- Like for, the if statement can start with a short statement to execute before the condition.

- Variables declared by the statement, eg., v, are only in scope until the end of the if.

- Question: Explain how the result comes out

if-else

```go
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),
    )
}
```

- Variables declared inside an if short statement are also available inside any of the else blocks.

switch

```
1  package main
2
3  import (
4      "fmt"
5      "runtime"
6  )
7
8  func main() {
9      fmt.Print("Go runs on ")
10     switch os := runtime.GOOS; os {
11     case "darwin":
12         fmt.Println("OS X.")
13     case "linux":
14         fmt.Println("Linux.")
15     default:
16         // freebsd, openbsd,
17         // plan9, windows...
18         fmt.Printf("%s.\n", os)
19     }
20 }
```

- *Switch* statement in Go only runs the selected case, not all the cases that follow

- Switch cases evaluate cases from top to bottom, stopping when a case succeeds.

switch

```go
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 a condition is the same as switch true.

- This construct can be a clean way to write long if-then-else chains.

switch with no condition

```go
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.")
    }
}
```

- This construct can be a clean way to write long if-then-else chains.

defer

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

```go
package main

import "fmt"

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

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

```go
package main

import "os"

func main() {
    f, err := os.Open("1.txt")
    if err != nil {
        panic(err)
    }

    // main 마지막에 파일 close 실행
    defer f.Close()

    // 파일 읽기
    bytes := make([]byte, 1024)
    f.Read(bytes)
    println(len(bytes))
}
```

defer

```go
package main

import "fmt"

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

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

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

- Deferred function calls are pushed onto a stack. When a function returns, its deferred calls are executed in last-in-first-out order.

panic

```go
package main

import "os"

func main() {
    openFile("Invalid.txt")
    println("Done") //이 문장은 실행 안됨
}

func openFile(fn string) {
    f, err := os.Open(fn)
    if err != nil {
        panic(err)
    }
    // 파일 close 실행됨
    defer f.Close()
}
```

- Go 내장함수인 panic()함수는 현재 함수를 즉시 멈추고 현재 함수에 defer 함수들을 모두 실행한 후 즉시 리턴

Recover

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    openFile("1.txt")
    println("Done") // 이 문장 실행됨
}

func openFile(fn string) {
    // defere 함수. panic 호출시 실행됨
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("OPEN ERROR", r)
        }
    }()

    f, err := os.Open(fn)
    if err != nil {
        panic(err)
    }

    // 파일 close 실행됨
    defer f.Close()
}
```

- Go 내장함수인 recover()함수는 panic 함수에 의한 패닉상태를 다시 정상상태로 되돌리는 함수

Anonymous Functions

```go
package main

func main() {
    sum := func(n ...int) int { //익명함수 정의
        s := 0
        for _, i := range n {
            s += i
        }
        return s
    }

    result := sum(1, 2, 3, 4, 5) //익명함수 호출
    println(result)
}
```

- Call the anonymous function by the variable name

First Function

```go
package main

func main() {
    //변수 add 에 익명함수 할당
    add := func(i int, j int) int {
        return i + j
    }

    // add 함수 전달
    r1 := calc(add, 10, 20)
    println(r1)

    // 직접 첫번째 파라미터에 익명함수를 정의함
    r2 := calc(func(x int, y int) int { return x - y }, 10, 20)
    println(r2)

}

func calc(f func(int, int) int, a int, b int) int {
    result := f(a, b)
    return result
}
```

Variable Type

```go
package main

import (
    "fmt"
    "math/cmplx"
)

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

func main() {
    const f = "%T(%v)\n"
    fmt.Printf(f, ToBe, ToBe)
    fmt.Printf(f, MaxInt, MaxInt)
    fmt.Printf(f, z, z)
}
```

const f = "%T(%v)\n"

```
bool

string

int   int8   int16   int32   int64
uint uint8 uint16 uint32 uint64 uintptr

byte // uint8의 다른 이름(alias)

rune // int32의 다른 이름(alias)
     // 유니코드 코드 포인트 값을 표현합니다.

float32 float64

complex64 complex128
```

Variable Types

```go
package main

import "fmt"

func main() {
    var num1 int = 10
    var num2 float32 = 3.2
    var num3 complex64 = 2.5 + 8.1i
    var s string = "Hello, world!"
    var b bool = true
    var a []int = []int{1, 2, 3}
    var m map[string]int = map[string]int{"Hello": 1}
    var p *int = new(int)
    type Data struct{ a, b int }
    var data Data = Data{1, 2}
    var i interface{} = 1

    fmt.Println(num1) // 10: 정수 출력
    fmt.Println(num2) // 3.2: 실수 출력
    fmt.Println(num3) // (2.5+8.1i): 복소수 출력
    fmt.Println(s)    // Hello, world!: 문자열 출력
    fmt.Println(b)    // true: 불 출력
    fmt.Println(a)    // [1 2 3]: 슬라이스 출력
    fmt.Println(m)    // map[Hello:1]: 맵 출력
    fmt.Println(p)    // 0xc0820062d0: 포인터(메모리 주소) 출력
    fmt.Println(data) // {1 2}: 구조체 출력
    fmt.Println(i)    // 1: 인터페이스 출력

    fmt.Println(num1, num2, num3, s, b) // 10 3.2 (2.5+8.1i) Hello, world! true
    fmt.Println(p, a, m)                // 0xc0820062d0 [1 2 3] map[Hello:1]
    fmt.Println(data, i)                // {1 2} 1
}
```

http://pyrasis.com/book/GoForTheReallyImpatient/Unit41

3. Pointer, Struct

Pointer

```go
1  package main
2
3  import "fmt"
4
5  func main() {
6      i, j := 42, 2701
7
8      p := &i           // point to i
9      fmt.Println(*p) // read i through the pointer
10     *p = 21          // set i through the pointer
11     fmt.Println(i)  // see the new value of i
12
13     p = &j           // point to j
14     *p = *p / 37    // divide j through the pointer
15     fmt.Println(j) // see the new value of j
16 }
```

- 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.

Pointer

- 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.

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

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

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

Struct

```
 1  package main
 2
 3  import "fmt"
 4
 5  type Vertex struct {
 6      X int
 7      Y int
 8  }
 9
10  func main() {
11      fmt.Println(Vertex{1, 2})
12  }
13
```

- A struct is a collection of fields.

Struct

```
 1  package main
 2
 3  import "fmt"
 4
 5  type Vertex struct {
 6      X int
 7      Y int
 8  }
 9
10  func main() {
11      v := Vertex{1, 2}
12      v.X = 4
13      fmt.Println(v.X)
14  }
```

- Struct fields are accessed using a dot.

Struct

```go
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

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

- 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**

- **p.X** is allowed without the explicit dereference**.**

Struct Literals

```go
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)
}
```

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

- You can list just a subset of fields by using the Name: syntax. (And the order of named fields is irrelevant.)

4. Collections: Array, Slice, Map

Array

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

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

- An array's length is part of its type, so arrays cannot be resized.

Slice

```go
package main

import "fmt"

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

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

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

- A slice is formed by specifying two indices, a low and high bound, separated by a colon:

```go
package main
import "fmt"

func main() {
    var a []int          //슬라이스 변수 선언
    a = []int{1, 2, 3} //슬라이스에 리터럴값 지정
    a[1] = 10
    fmt.Println(a)       // [1, 10, 3]출력
}
```

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

```
a[1:4]
```

Slices

```go
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)
}
```

- A slice does not store any data, it just describes a section of an underlying array.

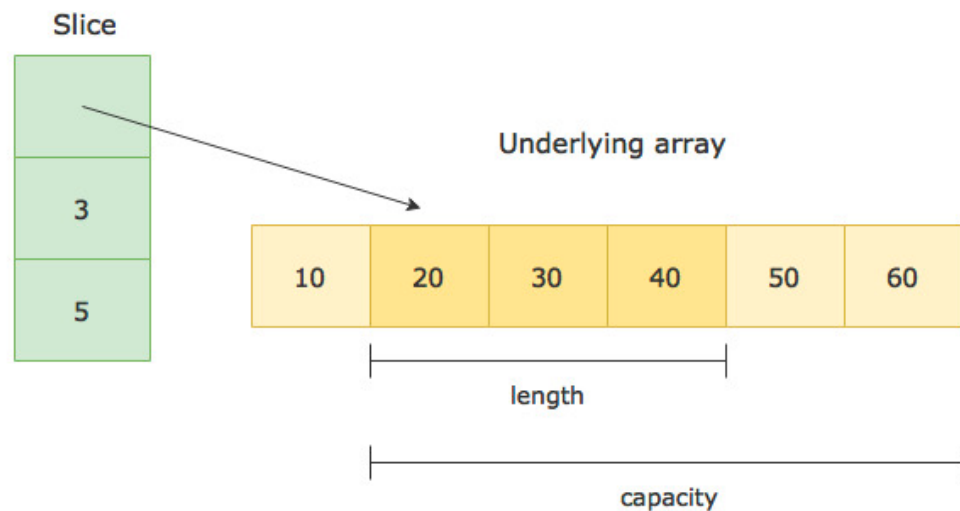- Changing the elements of a slice modifies the corresponding elements of its underlying array.

```go
s := []int{0, 1, 2, 3, 4, 5}
s = s[2:5]      // 2, 3, 4
s = s[1:]       // 3, 4
fmt.Println(s) // 3, 4 출력
```

Slice literals

```go
package main

import "fmt"

func main() {
    q := []int{2, 3, 5, 7, 11, 13}
    fmt.Println(q)

    r := []bool{true, false, true, true, false, true}
    fmt.Println(r)

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

- A slice literal is like an array literal without the length.

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

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

Length and Capacity


```go
func main() {
    s := make([]int, 5, 10)
    println(len(s), cap(s)) // len 5, cap 10
}
```

- Length (슬라이스의 길이)

- Capacity (내부 배열의 최대 길이)

  - 슬라이스 용량(capacity)이 아직 남아 있는 경우는 그 용량 내에서 슬라이스의 길이(length)를 변경하여 데이타를 추가

  - 용량(capacity)을 초과하는 경우 현재 용량의 2배에 해당하는 새로운 Underlying array 을 생성하고 기존 배열 값들을 모두 새 배열에 복제한 후 다시 슬라이스를 할당
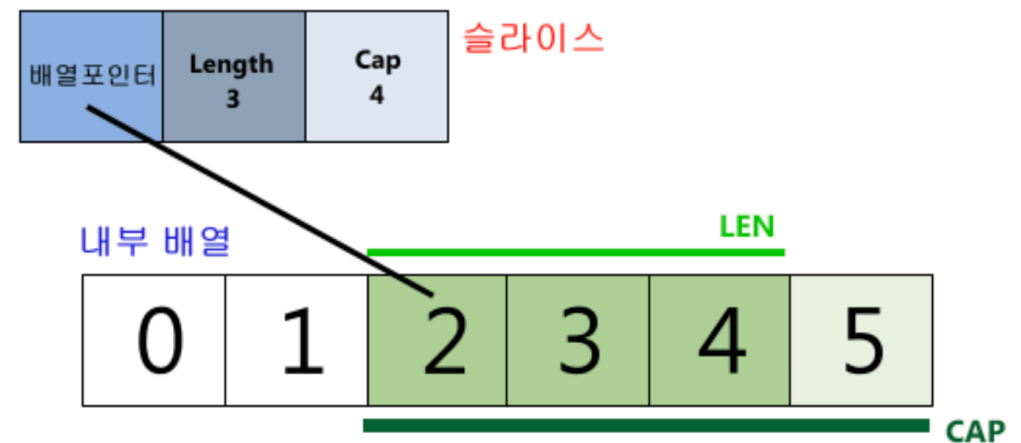
Slice Defaults

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       s := []int{2, 3, 5, 7, 11, 13}
7
8       s = s[1:4]
9       fmt.Println(s)
10
11      s = s[:2]
12      fmt.Println(s)
13
14      s = s[1:]
15      fmt.Println(s)
16  }
```

S := []int { 0, 1, 2, 3, 4, 5 } 실행시 슬라이스

S := S[2:5] 실행시 변경된 슬라이스

Creating a Slice

- Slices can be created with the built-in make function; this is how you create dynamically-sized arrays.

- 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
```

```
1  func main() {
2      s := make([]int, 5, 10)
3      println(len(s), cap(s)) // len 5, cap 10
4  }
```

Append to Slides

```go
func main() {
    s := []int{0, 1}

    // 하나 확장
    s = append(s, 2)         // 0, 1, 2
    // 복수 요소들 확장
    s = append(s, 3, 4, 5) // 0,1,2,3,4,5

    fmt.Println(s)
}
```

```go
package main

import "fmt"

func main() {
    // len=0, cap=3 인 슬라이스
    sliceA := make([]int, 0, 3)

    // 계속 한 요소씩 추가
    for i := 1; i <= 15; i++ {
        sliceA = append(sliceA, i)
        // 슬라이스 길이와 용량 확인
        fmt.Println(len(sliceA), cap(sliceA))
    }

    fmt.Println(sliceA) // 1 부터 15 까지 숫자 출력
}
```

Range

```
1  package main
2
3  import "fmt"
4
5  var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}
6
7  func main() {
8      for i, v := range pow {
9          fmt.Printf("2**%d = %d\n", i, v)
10     }
11 }
12
13
```

- The range form of
  the for loop iterates over a
  slice or map.

- When ranging over a slice,
  two values are returned for
  each iteration.

  - Index

  - A copy of the element at
    that index.

Range

```
 1  package main
 2
 3  import "fmt"
 4
 5  func main() {
 6      pow := make([]int, 10)
 7      for i := range pow {
 8          pow[i] = 1 << uint(i) // == 2**i
 9      }
10      for _, value := range pow {
11          fmt.Printf("%d\n", value)
12      }
13  }
14
```

- 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
```

Map

Key	Value
"Kim"	23
"Lee"	45
"Park"	37

- Map은 키(Key)에 대응하는 값 (Value)을 신속히 찾는 해시테이 블(Hash table)을 구현한 자료 구조

- 선언

- 초기화

- 리터럴 초기화

```
1  var idMap map[int]string
```

```
1  idMap = make(map[int]string)
```

```
1  //리터럴을 사용한 초기화
2  tickers := map[string]string{
3      "GOOG": "Google Inc",
4      "MSFT": "Microsoft",
5      "FB":   "FaceBook",
6  }
```

Map

```go
 1  package main
 2
 3  import "fmt"
 4
 5  type Vertex struct {
 6      Lat, Long float64
 7  }
 8
 9  var m = map[string]Vertex{
10      "Bell Labs": Vertex{
11          40.68433, -74.39967,
12      },
13      "Google": Vertex{
14          37.42202, -122.08408,
15      },
16  }
17
18  func main() {
19      fmt.Println(m)
20  }
21
```

- Initialize a map using literals

Map

```go
package main

func main() {
    var m map[int]string

    m = make(map[int]string)
    //추가 혹은 갱신
    m[901] = "Apple"
    m[134] = "Grape"
    m[777] = "Tomato"

    // 키에 대한 값 읽기
    str := m[134]
    println(str)

    noData := m[999] // 값이 없으면 nil 혹은 zero 리턴
    println(noData)

    // 삭제
    delete(m, 777)
}
```

Map

```
 1  package main
 2
 3  import "fmt"
 4
 5  func main() {
 6      m := make(map[string]int)
 7
 8      m["Answer"] = 42
 9      fmt.Println("The value:", m["Answer"])
10
11      m["Answer"] = 48
12      fmt.Println("The value:", m["Answer"])
13
14      delete(m, "Answer")
15      fmt.Println("The value:", m["Answer"])
16
17      v, ok := m["Answer"]
18      fmt.Println("The value:", v, "Present?", ok)
19  }
```

- 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.

Methods