The Go Programming Language

Basic

Anuchit Prasertsang
Developer

Basic

Topic

- Introduction
- Variables
- Constants
- Control Flow
- Loop
- Data Structures
- Array
- Slices
- Map
- Functions
- Defer
- Pointers

Installing

Download https://golang.org/dl (https://golang.org/dl/)

mac/linux

Download file.tar.gz

```
~/go1.x.x
```

windows

Download file.zip

```
C:\go1.x.x
```

GOROOT

mac/linux

```
export GOROOT=~/go1.x.x
export GOBIN=$GOPATH/bin
export PATH=$GOBIN:$GOROOT/bin:$PATH
```

windows

```
set GOROOT=C:\go1.x.x
set GOBIN=%GOPATH%\bin
set PATH=%GOBIN%;%GOROOT%\bin;%PATH%
```

GOROOT vs GOPATH

GOROOT is a root of your Go installation.
If you are a Java user, $GOROOT is similar in effect to $JAVA_HOME

GOPATH is a environment variable specifies the location of your workspace.
It defaults to a directory named go inside your home directory

- Unix

```
$HOME/go
```

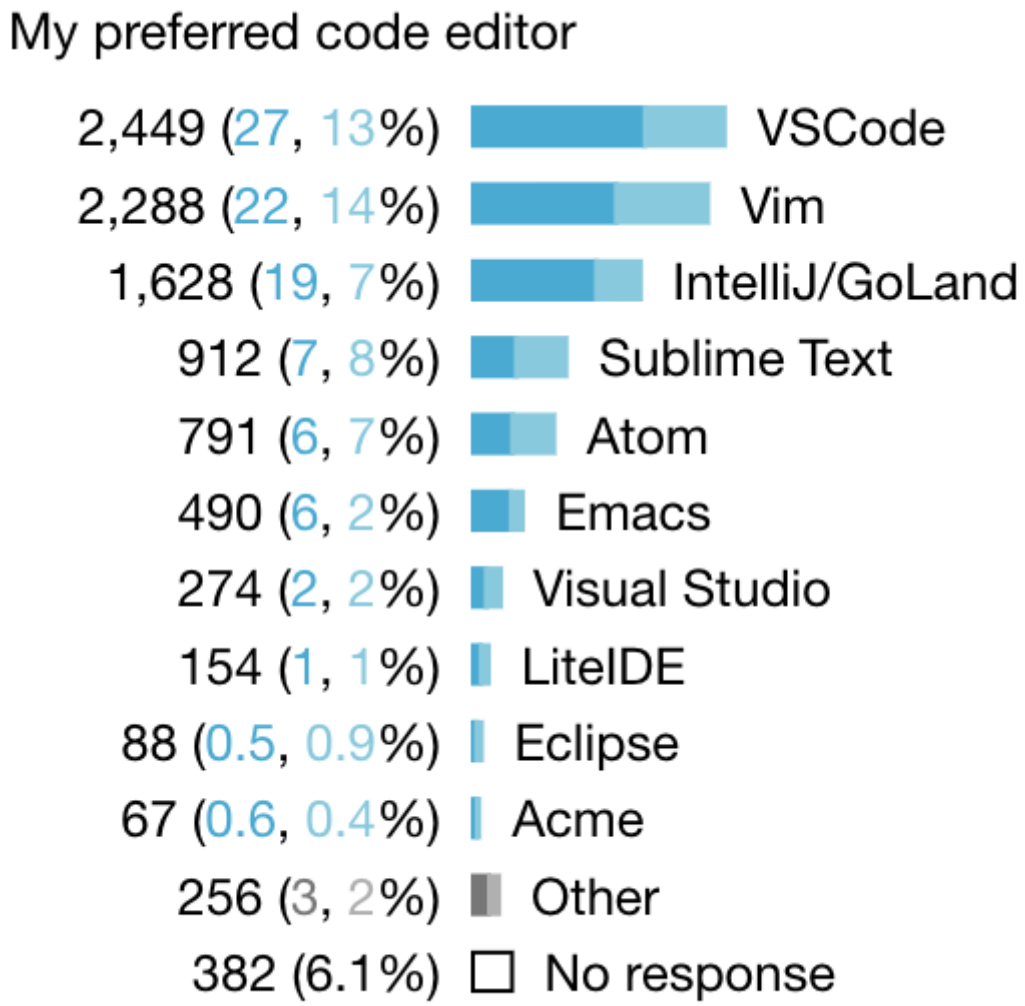
- Windows

```
%USERPROFILE%\go (usually C:\Users\YourName\go)
```

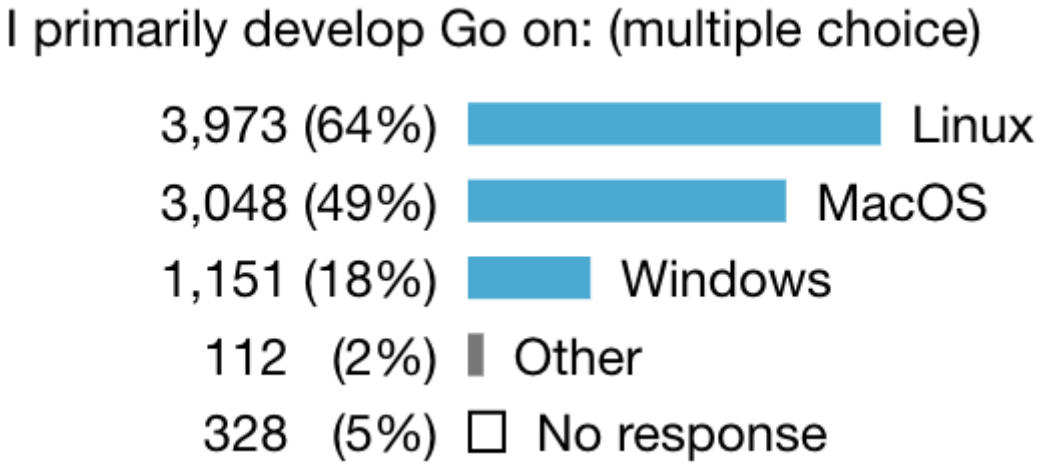
Project structure

```
bin/
    hello                           # command executable
    outyet                          # command executable
pkg/
    linux_amd64/
        github.com/golang/example/
            stringutil.a            # package object
src/
    github.com/golang/example/
        .git/                       # Git repository metadata
    hello/
        hello.go                # command source
    outyet/
        main.go                 # command source
        main_test.go            # test source
    stringutil/
        reverse.go              # package source
        reverse_test.go         # test source
    golang.org/x/image/
        .git/                       # Git repository metadata
    bmp/
        reader.go               # package source
        writer.go               # package source
    ... (many more repositories and packages omitted) ...
```

editor


My preferred code editor
2,449 (27, 13%) VSCode
2,288 (22, 14%) Vim
1,628 (19, 7%) IntelliJ/GoLand
912 (7, 8%) Sublime Text
791 (6, 7%) Atom
490 (6, 2%) Emacs
274 (2, 2%) Visual Studio
154 (1, 1%) LiteIDE
88 (0.5, 0.9%) Eclipse
67 (0.6, 0.4%) Acme
256 (3, 2%) Other
382 (6.1%) No response

OS


I primarily develop Go on: (multiple choice)
3,973 (64%) Linux
3,048 (49%) MacOS
1,151 (18%) Windows
112 (2%) Other
328 (5%) No response

Keywords

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Packages

- Every Go program is made up of packages.
- Programs start running in package main.

```
package main

import (
    "fmt"
    "math/rand"
)

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

Run

- By convention, the package name is the same as the last element of the import path. For instance, the **"math/rand"** package comprises files that begin with the statement package **rand**.

Imports

- This code groups the imports into a parenthesized, "factored" import statement.

```
package main

import (
    "fmt"
    "math"
)

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

- You can also write multiple import statements, like:

```
import "fmt"
import "math"
```

But it is good style to use the factored import statement.

Exported names

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

```
Pizza
Pi
Println
```

- When importing a package **math.Pi**
- Any "unexported" names are not accessible from outside the package.

Variables

- The **var** statement declares a list of variables; as in function argument lists, the type is last.
- A **var** statement can be at package or function level. We see both in this example.

```
package main

import "fmt"

var c, python, java bool

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

Run

Variables with initializers

- If an initializer is present, the type can be omitted; the variable will take the type of the initializer.

```
package main

import "fmt"

var i, j int = 1, 2

func main() {
    var c, python, java = true, false, "no!"

    fmt.Println(i, j, c, python, java)
}
```

Run

Short variable declarations

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

```
package main

import "fmt"

func main() {
    var i, j int = 1, 2
    k := 3
    c, python, java := true, false, "no!"

    fmt.Println(i, j, k, c, python, java)
}
```

Run

- Outside a function, every statement begins with a keyword (*var*, *func*, and so on) and so the *:=* construct is not available.

Basic types

Go's basic types are

```
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
```

The *int*, *uint*, and *uintptr* types 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.

Zero values

Variables declared without an explicit initial value are given their zero value.

The zero value is:
0 for numeric types,
false for the boolean type
"" (the empty string) for strings.

```
package main

import "fmt"

func main() {
    var i int
    var f float64
    var b bool
    var s string
    var r rune
    fmt.Printf("%v %v %v %v %q\n", i, f, b, r, s)
}
```

Run

%q a double-quoted string safely escaped with Go syntax

Operators and punctuation

The following character sequences represent operators (including assignment operators) and punctuation:

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=
```

Type conversions

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)
```

Unlike in C, in Go assignment between items of different type requires an explicit conversion.

Type conversions - example

Try removing the *float64* or *uint* conversions in the example and see what happens.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    var x, y int = 3, 4
    var f float64 = math.Sqrt(float64(x*x + y*y))
    var z uint = uint(f)
    fmt.Println(x, y, z)
}
```

Type inference

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.

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
```

Type inference - example

Try changing the initial value of *v* in the example code and observe how its type is affected.

```
package main

import "fmt"

func main() {
    v := 42 // change me!
    fmt.Printf("v is of type %T\n", v)
}
```

Constants

Constants are declared like variables, but with the const keyword.
Constants can be *character*, *string*, *boolean*, or *numeric* values.
Constants **cannot** be declared using the := syntax.

```
package main

import "fmt"

const Pi = 3.14

func main() {
    const world = "世界"
    fmt.Println("Hello", world)
    fmt.Println("Happy", Pi, "Day")

    const truth = true
    fmt.Println("Go rules?", truth)
}
```

Run

Constants - iota

iota expression is repeated by the other constants
until another assignment or type declaration shows up.

Functions

- A function can take zero or more arguments.
- In this example, **add** takes two parameters of type **int**.
- Notice that the *type* comes *after* the *variable name*.

```
package main

import "fmt"

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

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

```
func add(x, y int) int
```

Multiple results

- a function can return any number of results
- the **swap** function returns two string.

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

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

Run

Named return values

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

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

- A return statement without arguments returns the named return values. This is known as a **"naked"** return.
- Naked return statements should be used only in short functions.

Function values

Functions are values too. They can be **passed around** just like other values.
Function values may be used as function arguments and return 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))
}
```

Function closures

```
package main

import "fmt"

func adder() (func() int, func() int) {
    sum := 0
    return func() int {
            sum += 1
            return sum
        },
        func() int {
            return sum
        }
}

func main() {
    inc, cur := adder()
    fmt.Println(cur())

    fmt.Println(inc())
    fmt.Println(inc())

    fmt.Println(cur())
}
```

Exercise: Fibonacci closure

Implement a *fibonacci* function that returns a function (a closure) that returns successive fibonacci numbers
F0, the "0" is omitted (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 {
}

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

Flow Control

For

Go has only one looping construct, the **for** loop.

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

Run

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
```

The init statement will often be a short variable declaration,
and the variables declared there are visible only in the scope of the **for** statement.
The loop will stop iterating once the boolean condition evaluates to **false**.

Note: there are no parentheses surrounding

For continued

The init and post statements are optional.

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

Run

For is Go's "while"

At that point you can drop the semicolons: C's while is spelled for in Go.

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

Run

Forever

If you omit the loop condition it loops forever, so an infinite loop is compactly expressed.

```
func main() {
    names := []string{"game", "bank", "samui", "dog", "jiew"}
    for _, name := range names {
        fmt.Println(name)
    }

    for {
        fmt.Println("print forever.")
    }
}
```

Run

If

Go's **if** statements are like its *for* loops;the expression need not be surrounded by parentheses ( ) but the braces **{ }** are **required**.

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

Run

If with a short statement

Like *for*, the **if** statement can start with a short statement to execute before the condition. Variables declared by the statement are only in scope until the end of the *if*.

(Try using *v* in the last *return* statement.)

```
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),
    )
}
```

If and else

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

```
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
}
```

Exercise: Loops and Functions

Switch

A **switch** statement is a shorter way to write a sequence of *if - else* statements.
It runs the first case whose value is equal to the condition expression.

```
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.", os)
    }
}
```

Run

Go **only runs the selected case**, not all the cases that follow.
the **break** statement is provided automatically in Go.

Switch evaluation order

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

(For example,

```
switch i {
case 0:
case f():
}
```

does not call *f* **if i==0** .)

```
func main() {
    fmt.Println("When's Saturday?")
    today := time.Now().Weekday()
    switch time.Saturday {
    case today + 0:
        fmt.Println("Today.")
    case today + 1:
        fmt.Println("Tomorrow.")
    default:
        fmt.Println("Too far away.")
    }
}
```

Run

Switch with no condition

Switch without a condition is the same as **switch true**.

```
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.")
    }
}
```

Run

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.

```
package main

import "fmt"

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

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

Run

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

Stacking defers

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

```
package main

import "fmt"

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

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

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

Run

More types

Pointers

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.

Pointers - example

```
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
}
```

Structs

A *struct* is a collection of fields.

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

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

Run

Struct Fields

Struct fields are accessed using a dot.

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

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

Run

Pointers to structs

Struct fields can be accessed through a struct pointer.

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

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

Run

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.

Struct Literals

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.)

```
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
)
```

Run

The special prefix **&** returns a pointer to the struct value.

Arrays

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

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

declares a variable a as an array of ten integers.

```
func main() {
    var a [3]string
    a[0] = "Hello"
    a[1] = "World"
    fmt.Println(a[0], a[1], a[2])
    fmt.Println(a)

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

Run

An array's length is part of its type, so arrays *cannot* be *resized*.
This seems limiting, but don't worry; Go provides a convenient way of working with arrays.[53]

Slices

An array has a fixed size. A slice, on the other hand, is a dynamically-sized, flexible view into the elements of an array. In practice, slices are much more common than arrays.

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:

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

This selects a **half-open range** which *includes* the *first* element, but *excludes* the *last* one.

The following expression creates a slice which includes elements 1 through 3 of **a**:

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

Slices - example, create slices from array

```
func main() {
    primes := [6]int{2, 3, 5, 7, 11, 13}
    fmt.Printf("%T => %v\n", primes, primes)

    var s []int = primes[1:4]
    fmt.Printf("%T => %v\n", s, s)
}
```

Run

Slices are like references to arrays

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

```
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)
}
```

Run

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

Other slices that share the same underlying array will see those changes.

Slice literals

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

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

the array will be create, then builds a slice that references it:

```
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},
    }
    fmt.Println(s)
}
```

Run

Slice bound defaults

When slicing, you may omit the high or low bounds to use their defaults instead.
For the array

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

these slice expressions are equivalent:

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

The default is *zero* for the low bound and the *length* of the slice for the high bound.

Slice bound defaults - example

```
package main

import "fmt"

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

    s = s[1:4]
    fmt.Println(s)

    s = s[:2]
    fmt.Println(s)

    s = s[1:]
    fmt.Println(s)
}
```

Run

Slice length and capacity

A slice has both a *length* and a *capacity*.

The length of a slice is the number of elements it contains.

The capacity of a slice is the number of elements in the underlying array, counting from the first element in the slice.

The length and capacity of a slice **s** can be obtained using the expressions **len(s)** and **cap(s)**.

Slice length and capacity - example

You can extend a slice's length by re-slicing it, provided it has sufficient capacity. Try changing one of the slice operations in the example program to extend it beyond its capacity and see what happens.

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

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

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

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

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

Nil slices

The zero value of a slice is **nil**.
A nil slice has a length and capacity of 0 and has no underlying array.

```
package main

import "fmt"

func main() {
    var s []int
    fmt.Println(s, len(s), cap(s))
    if s == nil {
        fmt.Println("nil!")
    }
}
```

Run

Creating a slice with make

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
```

Creating a slice with make - example

```
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)
}
```

Run

Slices of slices

Slices can contain any type, including other slices.

```
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], " "))
    }
}
```

Run

Appending to a slice

It is common to **append** new elements to a slice, and so Go provides a built-in append function.

append documents (https://golang.org/pkg/builtin/#append)

```
func append(s []T, vs ...T) []T
```

The first parameter **s** of **append** is a slice of type **T**, and the rest are **T** values to append to the slice.

The resulting value of **append** is a slice containing all the elements of the original slice plus the provided values.

If the backing array of **s** is too small to fit all the given values a bigger array will be allocated. The returned slice will point to the newly allocated array.

slices usages and internals (https://blog.golang.org/go-slices-usage-and-internals)

Appending to a slice - example

```
package main

import "fmt"

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

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

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

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

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

Run

Range

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. The **first** is the **index**, and the **second** is a **copy of** the **element** at that index.

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

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

Run

Range continued

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

If you only want the index, drop the **, value** entirely.

```
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)
    }
}
```

Run

Exercise: Slices

Implement **Pic**. It should return a slice of length **dy**, each element of which is a slice of **dx** 8-bit unsigned integers. When you run the program, it will display your picture, interpreting the integers as grayscale (well, bluescale) values.

The choice of image is up to you. Interesting functions include *(x+y)/2*, *x*y*, and *x^y*.

(You need to use a loop to allocate each **[]uint8** inside the **[][]uint8**.)

(Use **uint8(intValue)** to convert between types.)

```
package main

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

func Pic(dx, dy int) [][]uint8 {
}

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

Exercise: Slices - example answer

```
package main

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

func Pic(dx, dy int) [][]uint8 {
    pic := [][]uint8{}

    for y := 0; y < dy; y++ {
        row := []uint8{}
        for x := 0; x < dx; x++ {
            if x%2 == 0 {
                row = append(row, uint8((x+y)/2))
            } else {
                row = append(row, uint8(x^y))
            }
        }
        pic = append(pic, row)
    }
    return pic
}

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

Run

Maps

A map maps keys to values.

The zero value of a map is **nil**. A **nil** map has no keys, nor can keys be added.

The make function returns a **map** of the given type, initialized and ready for use.

```
func main() {
    var m map[string]string
    m = make(map[string]string)
    m["nong"] = "AnuchitO"
    fmt.Println(m["nong"])
}
```

Run

Exercise: Maps I

- create map

```
key - student id
value - student information, name, class, age, no
```

Map literals

Map literals are like struct literals, but the keys are required.

```
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)
}
```

Run

Map literals continued

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

```
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)
}
```

Run

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.

Mutating Maps - example

```
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)
}
```

Run

Exercise: Maps II

Implement **WordCount**. It should return a map of the counts of each “word” in the string **s**.

```
package main

import "fmt"

func WordCount(s string) map[string]int {
    return map[string]int{"x": 1}
}

func main() {
    s := "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck
    w := WordCount(s)
    fmt.Println(w)
}
```

You might find **strings.Fields** helpful.

Thank you

Anuchit Prasertsang
Developer
anuchit.prasertsang@gmail.com (mailto:anuchit.prasertsang@gmail.com)
https://github.com/AnuchitO (https://github.com/AnuchitO)
@twitter_AnuchitO (http://twitter.com/twitter_AnuchitO)