Go Language Tutorial

6/5/2014

Contents

1 The Official Go Language Tutorial

(Actually, this is an unofficial in SageMathCloud of the official tutorial.)

Lets take the official Go tour: `http://tour.golang.org` using SageMathCloud.
Instructions:

- To use this, open the file go.sagews in any SageMathCloud project at `https://cloud.sagemath.com`

- Edit code in cells and press shift+enter to evaluate it.
- Insert new cells by clicking the blue bar between cells.
- Put
- Double click on text if you want to change it.

```
# Here's the documentation for the "%go" magic command.  Put %go at the \
   beginning of each cell
# (or put '%default_mode go' in a cell)
go?
```

```
   File: /projects/ebf87cdc-c0e2-46c9-b242-87d96eaee5f6
   Docstring:
   x.__init__(...) initializes x; see help(type(x)) for signature
```

2 Hello,

```
%go

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

```
Hello,
```

3 This Go Playground

The SageMathCloud sandbox is actually not restrictive like the official go one. You can do anything.

```
%go
import (
    "net"
    "os"
    "time"
)

func main() {
    fmt.Println("Welcome to the playground!")

    fmt.Println("The time is", time.Now())

    fmt.Println("And if you try to open a file:")
    fmt.Println(os.Open("filename"))

    fmt.Println("Or access the network:")
    fmt.Println(net.Dial("tcp", "google.com"))
}
```

```
Welcome to the playground!
The time is 2014-06-04 15:42:37.395352624 +0000 UTC
And if you try to open a file:
<nil> open filename: no such file or directory
Or access the network:
<nil> dial tcp: missing port in address google.com
```

4 Packages

Every Go program is made up of packages.

Programs start running in package main.

This program is using the packages with import paths "fmt" and "math/rand".

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.

```
%go

import (
    "time"
    "math/rand"
)

func main() {
    rand.Seed(2);
    fmt.Println("My favorite number is", rand.Intn(10))
    fmt.Println("The time is", time.Now())
}
```

```
My favorite number is 6
The time is 2014-06-04 15:43:14.665529256 +0000 UTC
```

5 Imports

The above code groups the imports into a parenthesized, factored import statement. You can also write multiple import statements, like this:

```
%go

import "time"
import "math"

func main() {
    fmt.Printf("Now you have %g problems.",
        math.Nextafter(2, 3))
    fmt.Println("The time is", time.Now())
}
```

```
Now you have 2.0000000000000004 problems.The time is 2014-06-04 15:43:27.118208726 +0000
UTC
```

6 Exported names

After importing a package, you can refer to the names it exports.

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

Foo is an exported name, as is FOO. The name foo is not exported.

Run the code. Then rename math.pi to math.Pi and try it again.

You should see an error below before you change math.pi to math.Pi, then press shift+enter.

```
%go
import (
    "math"
)

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

```
# command-line-arguments
./616efe99-b8c6-4979-a40b-d7147925506e.go:8: cannot refer to unexported name math.pi
./616efe99-b8c6-4979-a40b-d7147925506e.go:8: undefined: math.pi
```

7 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.

```
%go

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


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

```
55
```

8 Functions continued

When two or more consecutive named function parameters share a type, you can omit the type from all but the last.

In this example, we shortened

x int, y int

to

x, y int

```
%go
```

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

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

```
55
```

9 Multiple results

A function can return any number of results.

This function returns two strings.

```
%go

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

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

```
world hello
```

10 Named results

Functions take parameters. In Go, functions can return multiple result parameters, not just a single value. They can be named and act just like variables.

If the result parameters are named, a return statement without arguments returns the current values of the results.

```
%go

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

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

```
7 10
```

11 Variables

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

```
%go

var i int
var c, python, java bool

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

```
0 false false false
```

12 Variables with initializers

A var declaration can include initializers, one per variable.

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

```
%go

var i, j int = 1, 2
var c, python, java = true, false, "no!"

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

```
1 2 true false no!
```

13 Short variable declarations

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

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

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

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

```
1 2 3 true false no!
```

14 Basic types

Gos 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

```
%go

import "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)
}
```

```
bool(false)
uint64(18446744073709551615)
complex128((2+3i))
```

15 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. Try removing the float64 or int conversions in the example and see what happens.

```
%go
import "math"

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

```
3 4 5
```

16 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.

```
%go

const Pi = 3.14

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

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

```
Hello
Happy 3.14 Day
Go rules? true
```

17 Numeric Constants

- Numeric constants are high-precision values.
- An untyped constant takes the type needed by its context.
- Try printing needInt(Big) too.

```
%go
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))
    fmt.Println(needFloat(Small))
    fmt.Println(needFloat(Big))
}
```

```
21
0.2
1.2676506002282295e+29
```

18 For

- Go has only one looping construct, the for loop.
- The basic for loop looks as it does in C or Java, except that the ( ) are gone (they are not even optional) and the   are required.

```
%go

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

```
45
```

(Aside: you can combine Sages time and go magics to find the total time to compile and run the program)

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

```
45
CPU time: 0.03 s, Wall time: 0.63 s
```

19 For continued

As in C or Java, you can leave the pre and post statements empty.

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

```
1024
```

20 For is Gos while

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

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

```
1024
```

21 Forever

If you omit the loop condition it loops forever, so an infinite loop is compactly expressed.
HINT: In SageMathCloud, interrupt this by clicking the Stop button above.

```
%go

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

22 If

The if statement looks as it does in C or Java, except that the ( ) are gone and the are required.
(Sound familiar?)

```
%go

import "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))
}
```

```
1.4142135623730951 2i
```

23 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.)

```
%go

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),
    )
}
```

```
9 20
```

24 If and else

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

```
%go
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),
    )
}
```

```
27 >= 20
9 20
```

25 Exercise: Loops and Functions

As a simple way to play with functions and loops, implement the square root function using Newtons method.
In this case, Newtons method is to approximate Sqrt(x) by picking a starting point z and then repeating:

$$z = z - \frac{z^2 - x}{2z}$$

To begin with, just repeat that calculation 10 times and see how close you get to the answer for various values (1, 2, 3,).

Next, change the loop condition to stop once the value has stopped changing (or only changes by a very small delta). See if thats more or fewer iterations. How close are you to the math.Sqrt?

Hint: to declare and initialize a floating point value, give it floating point syntax or use a conversion:

z := float64(1) z := 1.0

```
%go

func Sqrt(x float64) float64 {
}

func main() {
    fmt.Println(Sqrt(2))
}
```

```
# command-line-arguments
./9f3a5c04-7754-4606-8005-b90ee486d2c4.go:5: missing return at end of function
```

26 Structs

A struct is a collection of fields.
(And a type declaration does what youd expect.)

```
%go

type Vertex struct {
    X int
    Y int
}

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

```
{1 2}
```

27 Pointers

Go has pointers, but no pointer arithmetic.

Struct fields can be accessed through a struct pointer. The indirection through the pointer is transparent.

```
%go

type Vertex struct {
    X int
    Y int
}

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

```
{1000000000 2}
```

28 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.)

The special prefix constructs a pointer to a newly allocated struct.

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

var (
    p = Vertex{1, 2}  // has type Vertex
    q = &Vertex{1, 2} // has type *Vertex
    r = Vertex{X: 1}  // Y:0 is implicit
    s = Vertex{}      // X:0 and Y:0
)

func main() {
    fmt.Println(p, q, r, s)
}
```

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

29 The new function

The expression new(T) allocates a zeroed T value and returns a pointer to it.
var t *T = new(T)
or
t := new(T)

```
%go

type Vertex struct {
    X, Y int
}

func main() {
    v := new(Vertex)
    fmt.Println(v)
    v.X, v.Y = 11, 9
    fmt.Println(v)
}
```

```
&{0 0}
&{11 9}
```

30 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.
An arrays length is part of its type, so arrays cannot be resized. This seems limiting, but dont worry; Go provides a convenient way of working with arrays.

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

```
Hello World
[Hello World]
```

31 Slices

A slice points to an array of values and also includes a length.
[]T is a slice with elements of type T.

```
%go
func main() {
    p := []int{2, 3, 5, 7, 11, 13}
    fmt.Println("p ==", p)

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

```
p == [2 3 5 7 11 13]
p[0] == 2
p[1] == 3
p[2] == 5
p[3] == 7
p[4] == 11
p[5] == 13
```

32 Slicing slices

Slices can be re-sliced, creating a new slice value that points to the same array.
The expression
s[lo:hi]
evaluates to a slice of the elements from lo through hi-1, inclusive. Thus
s[lo:lo]
is empty and
s[lo:lo+1]
has one element.

```
%go

func main() {
    p := []int{2, 3, 5, 7, 11, 13}
    fmt.Println("p ==", p)
    fmt.Println("p[1:4] ==", p[1:4])

    // missing low index implies 0
    fmt.Println("p[:3] ==", p[:3])

    // missing high index implies len(s)
    fmt.Println("p[4:] ==", p[4:])
}
```

```
p == [2 3 5 7 11 13]
p[1:4] == [3 5 7]
p[:3] == [2 3 5]
p[4:] == [11 13]
```

33 Making slices

Slices are created with the make function. It works by allocating a zeroed array and returning 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

```
%go
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]
```

34 Nil slices

The zero value of a slice is nil.

A nil slice has a length and capacity of 0.

(To learn more about slices, read the `Slices:usageandinternalsarticle`.)

```
%go

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

```
[] 0 0
nil!
```

35 Range

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

```
%go
```

```
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
```

36 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, $(x + y)/2, andx * y$.

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

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

NOTE: I wasnt even able to figure out how to import that pic library; and I doubt it would work Email wstein@uw.edu if you figure out how to make this work.

```
%go

import "code.google.com/p/go-tour/pic"

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

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

```
9212d767-eb6d-4980-9425-8e882e33df76.go:3:8: cannot find package "code.google.com/p/go-
tour/pic" in any of:
        /usr/lib/go/src/pkg/code.google.com/p/go-tour/pic (from $GOROOT)
        ($GOPATH not set)
```

37 Maps

- A map maps keys to values.
- Maps must be created with make (not new) before use; the nil map is empty and cannot be assigned to.

```
%go

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}
```

38 Map literals

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

```
%go

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}]
```

39 Map literals continued

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

```
%go

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)
}
```

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

40 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 and elem is the zero value for the maps element type.

Similarly, when reading from a map if the key is not present the result is the zero value for the maps element type.

```
%go
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
```

41 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 `findstrings.Fields` helpful.

```
%go

// WARNING: I don't know how to import this...
import (
    "code.google.com/p/go-tour/wc"
)

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

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

```
fdac019e-be43-4660-9194-dfaf97f543e3.go:5:5: cannot find package "code.google.com/p/go-
tour/wc" in any of:
        /usr/lib/go/src/pkg/code.google.com/p/go-tour/wc (from $GOROOT)
        ($GOPATH not set)
```

42 Function values

Functions are values too.

```
%go

import (
    "fmt"
    "math"
)

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

    fmt.Println(hypot(3, 4))
}
```

```
5
```

43 Function closures

Go functions may be closures. A closure is a function value that references variables from outside its body.

The function may access and assign to the referenced variables; in this sense the function is bound to the variables.

For example, the adder function returns a closure. Each closure is bound to its own sum variable.

```
%go
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),
        )
    }
}
```

```
0 0
1 -2
3 -6
6 -12
10 -20
15 -30
21 -42
28 -56
36 -72
45 -90
```

44 Exercise: Fibonacci closure

Lets have some fun with functions.

Implement a fibonacci function that returns a function (a closure) that returns successive fibonacci numbers.

```
%go

// 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())
    }
}
```

```
# command-line-arguments
./ca433c87-2dc0-4f0b-8cfa-994ab0b18de6.go:7: missing return at end of function
```

45 Switch

You probably knew what switch was going to look like.

A case body breaks automatically, unless it ends with a fallthrough statement.

```
%go

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("SageMathCloud is running ")
    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)
    }
}
```

```
SageMathCloud is running Linux.
```

46 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.

```
%go
import (
    "fmt"
    "time"
)

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.")
    case today + 2:
        fmt.Println("In two days.")
    case today + 3:
        fmt.Println("In three days.")
    default:
        fmt.Println("Too far away.")
    }
}
```

```
When's Saturday?
In three days.
```

47 Switch with no condition

Switch without a condition is the same as switch true.

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

NOTE: In SageMathCloud the clock is set to the UTC time zone (i.e., England).

```
%go

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.")
    }
}
```

```
Good afternoon.
```

48 Advanced Exercise: Complex cube roots

Lets explore Gos built-in support for complex numbers via the complex64 and complex128 types. For cube roots, Newtons method amounts to repeating:

$$z = z - \frac{z^3 - x}{3z^2}$$

Find the cube root of 2, just to make sure the algorithm works. There is a `Pow` function in the math/cmplx package.

```
%go

func Cbrt(x complex128) complex128 {
}

func main() {
    fmt.Println(Cbrt(2))
}
```

```
# command-line-arguments
./12caf984-f34b-48cb-97be-3c32f3615f38.go:5: missing return at end of function
```

49 Methods and Interfaces

The next group of slides covers methods and interfaces, the constructs that define objects and their behavior.

50 Methods

Go does not have classes. However, you can define methods on struct types.

The method receiver appears in its own argument list between the func keyword and the method name.

```
%go
import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := &Vertex{3, 4}
    fmt.Println(v.Abs())
}
```

```
5
```

51 Methods continued

In fact, you can define a method on any type you define in your package, not just structs.

You cannot define a method on a type from another package, or on a basic type.

```
%go
import (
    "fmt"
    "math"
)

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
}
```

```
1.4142135623730951
```

52 Methods with pointer receivers

Methods can be associated with a named type or a pointer to a named type.

We just saw two Abs methods. One on the *Vertex pointer type and the other on the MyFloat value type.

There are two reasons to use a pointer receiver. First, to avoid copying the value on each method call (more efficient if the value type is a large struct). Second, so that the method can modify the value that its receiver points to.

Try changing the declarations of the Abs and Scale methods to use Vertex as the receiver, instead of *Vertex.

The Scale method has no effect when v is a Vertex. Scale mutates v. When v is a value (non-pointer) type, the method sees a copy of the Vertex and cannot mutate the original value.

Abs works either way. It only reads v. It doesnt matter whether it is reading the original value (through a pointer) or a copy of that value.

```
%go
import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := &Vertex{3, 4}
    v.Scale(5)
    fmt.Println(v, v.Abs())
}
```

```
&{15 20} 25
```

53 Interfaces

An interface type is defined by a set of methods.

A value of interface type can hold any value that implements those methods.

Note: The code below fails to compile.

Vertex doesnt satisfy Abser because the Abs method is defined only on *Vertex, not Vertex. Fix this by changing (v *Vertex) to (v Vertex).

```
%go

import (
    "fmt"
    "math"
)

type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f  // a MyFloat implements Abser
    a = &v // a *Vertex implements Abser

    // In the following line, v is a Vertex (not *Vertex)
    // and does NOT implement Abser.
    a = v

    fmt.Println(a.Abs())
}

type MyFloat float64
```

```
func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

```
# command-line-arguments
./ddbdfba8-0e23-4052-991d-ae6075709bf8.go:22: cannot use v (type Vertex) as type Abser in
assignment:
        Vertex does not implement Abser (Abs method has pointer receiver)
```

54 Interfaces are satisfied implicitly

A type implements an interface by implementing the methods.

There is no explicit declaration of intent.

Implicit interfaces decouple implementation packages from the packages that define the interfaces: neither depends on the other.

It also encourages the definition of precise interfaces, because you dont have to find every implementation and tag it with the new interface name.

`Packageio` defines Reader and Writer; you dont have to.

```
%go
import (
    "fmt"
    "os"
)

type Reader interface {
    Read(b []byte) (n int, err error)
}

type Writer interface {
    Write(b []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}

func main() {
    var w Writer
    // os.Stdout implements Writer
    w = os.Stdout

    fmt.Fprintf(w, "hello, writer\n")
}
```

```
hello, writer
```

55 Errors

An error is anything that can describe itself as an error string. The idea is captured by the predefined, built-in interface type, error, with its single method, Error, returning a string:

type error interface Error() string

The fmt packages various print routines automatically know to call the method when asked to print an error.

```
%go

import (
    "fmt"
    "time"
)

type MyError struct {
    When time.Time
    What string
}

func (e *MyError) Error() string {
    return fmt.Sprintf("at %v, %s",
        e.When, e.What)
}

func run() error {
    return &MyError{
        time.Now(),
        "it didn't work",
    }
}

func main() {
    if err := run(); err != nil {
        fmt.Println(err)
    }
}
```

```
at 2014-06-04 17:04:19.051890965 +0000 UTC, it didn't work
```

56 Exercise: Errors

Copy your ‘Sqrt function from the earlier exercises and modify it to return an error value.

Sqrt should return a non-nil error value when given a negative number, as it doesnt support complex numbers.

Create a new type

type ErrNegativeSqrt float64

and make it an error by giving it a

func (e ErrNegativeSqrt) Error() string

method such that ErrNegativeSqrt(-2).Error() returns "cannot Sqrt negative number: -2".

Note: a call to fmt.Print(e) inside the Error method will send the program into an infinite loop. You can avoid this by converting e first: fmt.Print(float64(e)). Why?

Change your Sqrt function to return an ErrNegativeSqrt value when given a negative number.

57 Web servers

`Packagehttp` serves HTTP requests using any value that implements http.Handler:

package http

type Handler interface ServeHTTP(w ResponseWriter, r *Request)

In this example, the type Hello implements http.Handler.

IMPORTANT

- Visit https://cloud.sagemath.com/project_id/port/4000/ to see the greeting, where project_id is the uuid of the project youre using right now (its in the url or you can get it by evaluating the line directly below).
- Click the Stop button above to terminate the server and continue being able to edit code in the worksheet.

```
salvus.project_info()['project_id']
```

```
u'ebf87cdc-c0e2-46c9-b242-87d96eaee5f6'
```

```
%go
import (
    "fmt"
    "net/http"
)

type Hello struct{}

func (h Hello) ServeHTTP(
    w http.ResponseWriter,
    r *http.Request) {
    fmt.Fprint(w, "Hello!")
}

func main() {
    var h Hello
    http.ListenAndServe(":4000", h)
}
```

58 Exercise: HTTP Handlers

Implement the following types and define ServeHTTP methods on them. Register them to handle specific paths in your web server.

type String string

type Struct struct Greeting string Punct string Who string

For example, you should be able to register handlers using:

http.Handle("/string", String("I'm a frayed knot.")) http.Handle("/struct", Struct"Hello", ":", "Gophers!")

WARNING: In SageMathCloud you have to instead handle urls like /project_id/port/4000/string, unfortunately.

```
%go

import (
    "net/http"
)

func main() {
    // your http.Handle calls here
    http.ListenAndServe("localhost:4000", nil)
}
```

59 Images

Package image defines the Image interface:

package image

type Image interface ColorModel() color.Model Bounds() Rectangle At(x, y int) color.Color

(See `thedocumentation` for all the details.)

Also, color.Color and color.Model are interfaces, but well ignore that by using the predefined implementations color.RGBA and color.RGBAModel. These interfaces and types are specified by the `image/colorpackage`.

```
%go
import (
    "fmt"
    "image"
)

func main() {
    m := image.NewRGBA(image.Rect(0, 0, 100, 100))
    fmt.Println(m.Bounds())
    fmt.Println(m.At(0, 0).RGBA())
}
```

```
(0,0)-(100,100)
0 0 0 0
```

60 Exercise: Images

Remember the picture generator you wrote earlier? Lets write another one, but this time it will return an implementation of image.Image instead of a slice of data.

Define your own Image type, implement `thenecessarymethods`, and call pic.ShowImage.

Bounds should return a image.Rectangle, like image.Rect(0, 0, w, h).

ColorModel should return color.RGBAModel.

At should return a color; the value v in the last picture generator corresponds to color.RGBAv, v, 255, 255 in this one.

WARNING: I have no idea how to make this work in SageMathCloud.

```
%go

import (
    "code.google.com/p/go-tour/pic"
    "image"
)

type Image struct{}

func main() {
    m := Image{}
    pic.ShowImage(m)
}
```

61 Exercise: Rot13 Reader

A common pattern is an io.Reader that wraps another io.Reader, modifying the stream in some way.

For example, the gzip.NewReader function takes an io.Reader (a stream of gzipped data) and returns a *gzip.Reader that also implements io.Reader (a stream of the decompressed data).

Implement a rot13Reader that implements io.Reader and reads from an io.Reader, modifying the stream by applying the ROT13 substitution cipher to all alphabetical characters.

The rot13Reader type is provided for you. Make it an io.Reader by implementing its Read method.

```
%go


import (
    "io"
    "os"
    "strings"
)

type rot13Reader struct {
    r io.Reader
}

func main() {
    s := strings.NewReader(
        "Lbh penpxrq gur pbqr!")
    r := rot13Reader{s}
    io.Copy(os.Stdout, &r)
}
```

```
# command-line-arguments
./cfc58d42-5c15-4fbe-a1fc-cd76012e544f.go:18: cannot use &r (type *rot13Reader) as type
io.Reader in function argument:
	*rot13Reader does not implement io.Reader (missing Read method)
```

62 Concurrency

The next section covers Gos concurrency primitives.

63 Goroutines

A goroutine is a lightweight thread managed by the Go runtime.

go f(x, y, z)

starts a new goroutine running

f(x, y, z)

The evaluation of f, x, y, and z happens in the current goroutine and the execution of f happens in the new goroutine.

Goroutines run in the same address space, so access to shared memory must be synchronized. The sync package provides useful primitives, although you wont need them much in Go as there are other primitives. (See the next slide.)

```
%go

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("sage")
    go say("world")
    say("hello")
}
```

```
hello
sage
world
hello
sage
```

```
world
hello
sage
world
hello
sage
world
hello
```

64 Channels

Channels are a typed conduit through which you can send and receive values with the channel operator, -.

ch - v // Send v to channel ch. v := -ch // Receive from ch, and // assign value to v.

(The data flows in the direction of the arrow.)

Like maps and slices, channels must be created before use:

ch := make(chan int)

By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

```
%go

func sum(a []int, c chan int) {
    sum := 0
    for _, v := range a {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    a := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(a[:len(a)/2], c)
    go sum(a[len(a)/2:], c)
    x, y := <-c, <-c // receive from c

    fmt.Println(x, y, x+y)
}
```

```
17 -5 12
```

65 Buffered Channels

Channels can be buffered. Provide the buffer length as the second argument to make to initialize a buffered channel:

ch := make(chan int, 100)

Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

Modify the example to overfill the buffer and see what happens.

```
%go

func main() {
    c := make(chan int, 2)
    c <- 1
    c <- 2
    fmt.Println(<-c)
    fmt.Println(<-c)
}
```

```
1
2
```

66 Range and Close

A sender can close a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: after

v, ok := -ch

ok is false if there are no more values to receive and the channel is closed.

The loop for i := range c receives values from the channel repeatedly until it is closed.

Note: Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic.

Another note: Channels arent like files; you dont usually need to close them. Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a range loop.

```
%go

func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

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

```
21
34
```

67 Select

The select statement lets a goroutine wait on multiple communication operations.

A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

```
%go

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

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

68 Default Selection

The default case in a select is run if no other case is ready.

Use a default case to try a send or receive without blocking:

select case i := -c: // use i default: // receiving from c would block

```
%go

import (
    "fmt"
    "time"
)

func main() {
    tick := time.Tick(100 * time.Millisecond)
    boom := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")
            return
        default:
            fmt.Println("    .")
            time.Sleep(50 * time.Millisecond)
        }
    }
}
```

```
    .
    .
tick.
    .
    .
tick.
    .
    .
tick.
    .
    .
tick.
    .
    .
BOOM!
```

69 Exercise: Equivalent Binary Trees

There can be many different binary trees with the same sequence of values stored at the leaves. For example, here are two binary trees storing the sequence 1, 1, 2, 3, 5, 8, 13.

A function to check whether two binary trees store the same sequence is quite complex in most languages.

Well use Gos concurrency and channels to write a simple solution.
This example uses the tree package, which defines the type:
type Tree struct Left *Tree Value int Right *Tree

70 Exercise: Equivalent Binary Trees

1. Implement the Walk function.

2. Test the Walk function.

The function tree.New(k) constructs a randomly-structured binary tree holding the values k, 2k, 3k, , 10k.
Create a new channel ch and kick off the walker:
go Walk(tree.New(1), ch) Then read and print 10 values from the channel. It should be the numbers 1, 2, 3, , 10.

1. Implement the Same function using Walk to determine whether t1 and t2 store the same values.

2. Test the Same function.

Same(tree.New(1), tree.New(1)) should return true, and Same(tree.New(1), tree.New(2)) should return false.

```
%go

import "code.google.com/p/go-tour/tree"

// Walk walks the tree t sending all values
// from the tree to the channel ch.
func Walk(t *tree.Tree, ch chan int)

// Same determines whether the trees
// t1 and t2 contain the same values.
func Same(t1, t2 *tree.Tree) bool

func main() {
}
```

```
dee55870-8c5d-4cdd-848e-52794891c767.go:3:8: cannot find package "code.google.com/p/go-
tour/tree" in any of:
        /usr/lib/go/src/pkg/code.google.com/p/go-tour/tree (from $GOROOT)
        ($GOPATH not set)
```

71 Exercise: Web Crawler

In this exercise youll use Gos concurrency features to parallelize a web crawler.
Modify the Crawl function to fetch URLs in parallel without fetching the same URL twice.
NOTE: Heh, Im not sure I want you running this in SageMathCloud :-)

```
%go
```

```
import (
    "fmt"
)

type Fetcher interface {
    // Fetch returns the body of URL and
    // a slice of URLs found on that page.
    Fetch(url string) (body string, urls []string, err error)
}

// Crawl uses fetcher to recursively crawl
// pages starting with url, to a maximum of depth.
func Crawl(url string, depth int, fetcher Fetcher) {
    // TODO: Fetch URLs in parallel.
    // TODO: Don't fetch the same URL twice.
    // This implementation doesn't do either:
    if depth <= 0 {
        return
    }
    body, urls, err := fetcher.Fetch(url)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("found: %s %q\n", url, body)
    for _, u := range urls {
        Crawl(u, depth-1, fetcher)
    }
    return
}

func main() {
    Crawl("http://golang.org/", 4, fetcher)
}

// fakeFetcher is Fetcher that returns canned results.
type fakeFetcher map[string]*fakeResult

type fakeResult struct {
    body string
    urls []string
}

func (f fakeFetcher) Fetch(url string) (string, []string, error) {
    if res, ok := f[url]; ok {
        return res.body, res.urls, nil
    }
    return "", nil, fmt.Errorf("not found: %s", url)
}

// fetcher is a populated fakeFetcher.
```

```
var fetcher = fakeFetcher{
    "http://golang.org/": &fakeResult{
        "The Go Programming Language",
        []string{
            "http://golang.org/pkg/",
            "http://golang.org/cmd/",
        },
    },
    "http://golang.org/pkg/": &fakeResult{
        "Packages",
        []string{
            "http://golang.org/",
            "http://golang.org/cmd/",
            "http://golang.org/pkg/fmt/",
            "http://golang.org/pkg/os/",
        },
    },
    "http://golang.org/pkg/fmt/": &fakeResult{
        "Package fmt",
        []string{
            "http://golang.org/",
            "http://golang.org/pkg/",
        },
    },
    "http://golang.org/pkg/os/": &fakeResult{
        "Package os",
        []string{
            "http://golang.org/",
            "http://golang.org/pkg/",
        },
    },
}
```

72 Where to Go from here

- See `http://tour.golang.org/#74` for documentation and other resources.
- In SageMathCloud you can write go code in files such as foo.go, then compile and run them using the command line terminal by typing go build foo.go. To create a file, click +New, type the file name, then press enter.