TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Introduction to Go

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2024/25

Valeria Cardellini

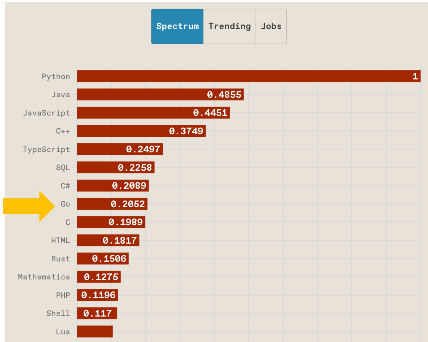
Laurea Magistrale in Ingegneria Informatica

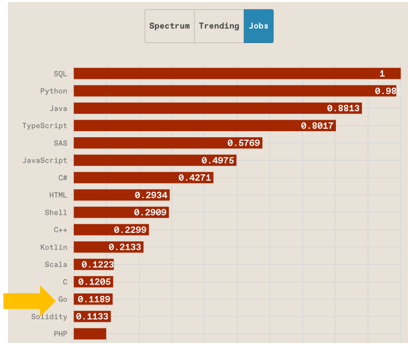
What is Go?

- "An open source programming language that makes it easy to build simple, secure, and scalable systems" https://go.dev/
- Conceived in 2007 at Google by R. Griesemer, R. Pike and K. Thompson, and announced in 2009
- Goals of language and its tools:
  - To be expressive, efficient in both compilation and execution, and effective in writing reliable and robust programs
  - Strong and statically, fast compiled language that feels like a dynamically typed, interpreted language
- Go's ancestors: mainly C and CSP (communicating sequential processes) formal language by T. Hoare https://en.wikipedia.org/wiki/Communicating_sequential_processes

Top programming languages and Go

- IEEE Spectrum's 2024 rankings of most popular programming languages https://spectrum.ieee.org/top-programming-languages-2024





Go and C

- Go: "C-like language" or "C for the 21st century"
- From C, Go inherited
  - Expression syntax
  - Control-flow statements
  - Basic data types
  - Call-by-value parameter passing
  - Pointers
  - Run-time efficiency
  - Static typing

Go and other languages

- New and efficient facilities for concurrency
- Flexible approach to data abstraction and object-oriented programming
- Automatic memory management (*garbage collection*)
- Readability and usability

Go and distributed systems

- Go allows programmers to focus on distributed system problems
  - good support for concurrency
  - good support for RPC
  - garbage-collected (no use-after-freeing problems)
  - type safe
- Simple language to learn

Go and cloud

- Also language for cloud native applications
- E.g., Go Cloud: library and tools for open cloud development in Go https://github.com/google/go-cloud
  - Goal: allow application developers to seamlessly deploy cloud applications on any combination of cloud providers
  - E.g., read from blob storage

```go
ctx := context.Background()
bucket, err := blob.OpenBucket(ctx, "s3://my-bucket")
if err != nil {
    return err
}
defer bucket.Close()
blobReader, err := bucket.NewReader(ctx, "my-blob", nil)
if err != nil {
    return err
}
```


Editor plugins and IDEs

- GoLand https://www.jetbrains.com/go/
- vim-go: plugin for vim https://github.com/fatih/vim-go
- Go extension for Visual Studio Code https://code.visualstudio.com/docs/languages/go
- Can be integrated with gopls https://github.com/golang/tools/tree/master/gopls
  - Go language server (What is it? https://langserver.org)

Hello world example

```go
package main

import "fmt"

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


Some notes on the first example

- No semicolon at the end of statements or declarations
- Go natively handles Unicode
- Every Go program is made up of packages (similar to C libraries or Python packages)
  – Package: one or more .go source files in a single directory
- Source file begins with package declaration (which package the file belongs to), followed by list of other imported packages
  – Programs start running in `main`
  – `fmt` package contains functions for printing formatted output and scanning input

Go tool

- Go is a compiled language
- Go tool: fetch, build, and install Go packages and commands
  - A zero configuration tool
- To run the program: **`go run`**

```
[√ go/src % go run helloworld.go
Hello, 世界
√ go/src %
```

- To build the program into binary: **`go build`**

```
[√ go/src % go build helloworld.go
[√ go/src % ls helloworld*
helloworld      helloworld.go
[√ go/src % ./helloworld
Hello, 世界
√ go/src %
```


Packages

- Go program is made up of packages
- Programs start running in package `main`
- Packages contain type, function, variable, and constant declarations
- Packages can even be very small or very large
- Case determines visibility: a name is exported if it begins with a capital letter
  - `Foo` is exported, `foo` is not
  - E.g., `fmt.Println(math.pi)`

    `./prog.go:9:19: undefined: math.pi`

Imports

- **`Import`** statement: groups imports into a parenthesized, "factored" statement

```
package main
import (
    "fmt"
    "math")

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


Functions

- Function can take zero or more arguments

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

  - add takes as input two arguments of type `int`
- Type comes *after* variable name
- Shorter version for input arguments:

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

- Function can return any number of results

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

  - Also useful to return both result and error values

Functions

```
package main

import "fmt"

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

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


Functions

- Return values may be named

```
package main

import "fmt"

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

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

Variables

- **var** statement: declares a list of variables
  - Type is last
- Can be at package or function level

```
package main
import "fmt"

var c, python, java bool

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

- Can include initializers, one per variable
  - If initializer is present, type can be omitted
- Variables declared without an explicit initial value are given their *zero value*
- Short variable declaration using `:=` (use only inside functions)

Types

- Usual basic types
  - `bool, string, int, uint, float32, float64, …`
- Type conversion

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

  - Unlike in C, in Go assignment between items of different type requires an explicit conversion
- Type inference
  - Variable's type inferred from value on right hand side

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

Flow control statements

- `for`, `if` (and `else`), `switch`
- `defer`

Looping construct

- Go has only one looping construct: **`for`** loop
- 3 components
  - *Init* statement
  - *Condition* expression
  - *Post* statement

```
sum := 0
for i := 0; i < 10; i++ {
        sum += i
}
```

- No parentheses surrounding the 3 components of `for` statement
- Braces `{ }` are always required

Looping construct

- Init and post statements are optional: in this way, `for` is Go's "while"

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

- If you omit condition, infinite loop

```
for {
}
```


Example: echo

```
// Echo prints its command-line arguments.
package main
import (
      "fmt"
      "os"
)
func main() {
      var s, sep string
      for i := 1; i < len(os.Args); i++ {
            s += sep + os.Args[i]
            sep = " "
      }
      fmt.Println(s)
}
```

s and sep implicitly initialized to empty strings

`os.Args` is a slice of strings (see next slides)

Conditional statements: if

- Go's `if` (and `else`) statement is like `for` loop:
  - Expression is not surrounded by parentheses ( )
  - Braces { } are always required
  - `if...else if...else` statement to combine multiple `if...else` statements

```
if optionalStatement1; booleanExpression1 {
    block1
} else if optionalStatement2; booleanExpression2 {
    block2
} else {
    block3
}
```


Example: if

- An example

```
if v := math.Pow(x, n); v < limit {
        return v
  } else {
        fmt.Printf("%g >= %g\n", v, limit)
  }
```

  - Remember that `} else` must be on the same line
  - Variable v is in scope only within `if` statement

Conditional statements: switch

- `switch` statement selects one of many cases to be executed
  – Cases evaluated from top to bottom, stopping when a case succeeds
- Differences from C
  – Go only runs the selected case, not all the cases that follow (i.e., C's break is provided automatically in Go)
  – Switch cases need not be constants, and values involved need not be integers

Defer statement

- New mechanism to defer the execution of a function *until* the surrounding function returns
  – Deferred call's arguments are evaluated immediately, but function call is not executed until surrounding function that contains `defer` has terminated

```
package main
import "fmt"

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

```
hello
world
```

- Deferred function calls pushed onto a stack
  – Deferred calls executed in LIFO order
- Great for cleanup things, like closing files or connections!

Pointers

- Pointer: value that contains the address of a variable
  - Usual operators * and &: & operator yields the address of a variable, and * operator retrieves the variable that the pointer refers to

```
var p *int
i := 1
p = &i    // p, of type *int, points to i
fmt.Println(*p) // "1"
*p = 2 // equivalent to i = 2
fmt.Println(i) // "2"
```

- Unlike C, Go has no pointer arithmetic
- Zero value for a pointer is `nil`
- Safe for a function to return the address of a local variable, because local variable will survive function scope

Composite data types: structs and array

- Aggregate data types: structs and arrays
- **Struct**: typed collection of fields
  - Syntax similar to C, fixed size

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

  - Struct fields are accessed using dot notation, e.g., `fmt.Println(v.X)`
  - Can also be accessed through a struct pointer

- **Array**: `[n]T` is an array of n values of type T
  - Fixed size (cannot be resized)

```
var a [2]string
a[0] = "Hello"
```

Composite data types: slices

- Slice: key data type in Go, more powerful than array

- `[ ]T` is a **slice** with elements of type `T`: dynamically-sized, flexible view into the elements of an array
  - Create a slice by slicing an existing array or slice
  - Specify two indices, a low and high bound, separated by a colon: `s[i : j]`
  - Slice includes the first element, but excludes the last

```
primes := [6]int{2, 3, 5, 7, 11, 13}
var s []int = primes[1:4]    [3 5 7]
```

- Slice: section of *underlying array*
  - Change slice element: modify corresponding element of underlying array

Slices: operations

- Length of slice `s`: number of elements it contains, use **`len(s)`**
- Capacity of slice `s`: number of elements in the underlying array, counting from the first element in the slice, use **`cap(s)`**
- Compile or run-time error if array length is exceeded: Go performs bounds check (memory-safe language)

- Slices can also be created using **`make`**
  - Length and capacity can be specified

Slices: operations

- Let's create an empty slice

```
package main
import "fmt"
func main() {
  a := make([]int, 0, 5)     // len(s)=0, cap(s)=5
  printSlice("a", a)
}

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

```
a len=0 cap=5 []
```


Slices: operations

- New items can be appended to a slice using **append**

  `func append(slice []T, elems ...T) []T`

  – When append a slice, slice may be enlarged if necessary

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

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

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

    s = append(s, 2, 3, 4) // more than one element
    printSlice(s)
}
```

Composite data types: maps

- **map** maps keys to values
  - Map type `map[K]V` is a reference to a hash table where `K` and `V` are the types of its keys and values
  - Use `make` to create a map

```
m = make(map[string]Vertex)
m["Bell Labs"] = Vertex{
   40.68433, -74.39967,
}
```

- Operations on map: insert or update element, retrieve element, delete element, test if key is present

```
m[key] = element       // insert or update
elem = m[key]          // retrieve
delete(m, key)         // delete
elem, ok = m[key]      // test
```


Range

- **range** iterates over entries in a variety of data structures
  - range on arrays and slices provides both index and value for each entry
  - range on map iterates over key/value pairs

```
package main
import "fmt"

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)
    }
}
```

Range: example

```go
func main() {
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    for k := range kvs {
        fmt.Println("key:", k)
    }
}
```

Skip index or value by assigning to _

```
$ go run range2.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
```

Key is first, value is second but doesn't have to be present
https://go.dev/ref/spec#For_statements

Go functions: anonymous and closures

- Go functions can be anonymous
  - Useful when you want to define a function inline without having to name it
- Go functions can be closures
  - Go closure: anonymous nested function which retains bindings to variables defined outside closure's body
  - Closure can hold a unique state of its own; state becomes isolated as you create new function instances
  - See example https://gobyexample.com/closures
- See https://www.calhoun.io/5-useful-ways-to-use-closures-in-go/
  - E.g., middleware pattern to independently act on a request before or after the normal request handler (e.g., to wrap HTTP request's handler and measure its processing time)

Closure: example

```
package main

import "fmt"

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

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


Methods

- Go does not have classes, but supports **methods** defined on struct types
- A method is a function with a special *receiver* argument (extra parameter before function name)
  - The receiver appears in its own argument list between `func` and method name

```
type Vertex struct {
    X, Y float64
}

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

Interfaces

- *Interface type*: named collection of method signatures
- Any type (struct) that implements the required methods, implements that interface
  - Instead of designing the abstraction in terms of what kind of data our type can hold, you design the abstraction in terms of *what actions* your type can execute
- A type is not explicitly declared to be of a certain interface, it is *implicit*
  - Just implement the required methods
- Let's code a basic interface for geometric shapes

Interface: example

```
package main

import "fmt"
import "math"

// A basic interface for geometric shapes
type geometry interface {
    area() float64
    perim() float64
}

// For example, implement this interface on rect and circle types
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}
```

Interface: example

```
// To implement an interface in Go, you just need to
// implement all the methods in the interface.

// Here you implement geometry on rect
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

// Here you implement geometry on circle
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```


Interface: example

```
// If a variable has an interface type, then you can call
// methods that are in the named interface. Here's a
// generic measure function taking advantage of this
// to work on any geometry
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // The circle and rect struct types both implement the
    // geometry interface so you can use instances of these
    // structs as arguments to measure
    measure(r)
    measure(c)
}
```

```
$ go run interfaces.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```

Concurrency in Go

- Go provides concurrency features as part of the core language
- **Goroutines** and **channels**
  - Support CSP concurrency model
- Can be used to implement different concurrency patterns

Goroutines

- Goroutine: lightweight thread managed by Go runtime
- Very easy to use: just prefix go to function call

```
go f(x, y, z)   // start a new goroutine running
                // f(x, y, z)
```

- Goroutines run in same address space, so access to shared memory must be synchronized
- Be careful: when `main` function returns, program exits without waiting for other (non-main) goroutines to complete
  - See example goroutine_termination.go

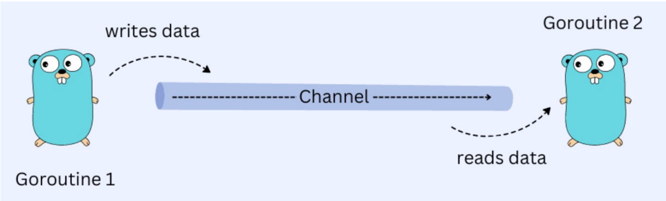
Goroutines

- Are goroutines threads?
  - No, they are lightweight abstractions over threads
    - Scheduled over OS threads by Go scheduler
    - A single OS thread can run many goroutines
  - Goroutine creation and destruction are cheaper as compared to OS threads (at least 5x) and less memory consuming (~500x)
- Are goroutines called in the declared order?
  - No, since goroutines are abstractions over threads, they all have the same priority and you therefore cannot control the order in which they run
- How to control goroutine performance?
  - You can set an environment variable (`GOMAXPROCS`) which determines how many threads your program will use simultaneously
    - Normally set to number of virtual CPU cores

https://dev.to/gophers/what-are-goroutines-and-how-are-they-scheduled-2nj3

Channels

- Communication mechanism that lets one goroutine sends values of a given type to another goroutine
  - Channel: thread-safe queue managed by Go and its runtime
- Hides a lot of pain of inter-thread communication
  - Internally, a channel uses mutexes and semaphores just as one might expect



- Multiple senders can write to same channel
  - Useful for notifications, multiplexing, etc.
  - And totally thread-safe!
- Be careful: only one can `close` channel, and can't send after close (panic!)

Channels

- Channel: a typed conduit through which a goroutine can send and receive values using the channel operator `<-`

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

Data flows in the arrow direction

- A conduit for values of a particular type (e.g., `int`, `struct`)
  - By default bidirectional
- Create channel with `make` before using it

```
ch := make(chan int)
```

- Send and receive block until the other side is ready
  - Goroutines can synchronize without explicit locks or condition variables
  - See https://gobyexample.com/channel-synchronization

Channels: example

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

func main() {
        s := []int{7, 2, 8, -9, 4, 0}
        c := make(chan int)
        go sum(s[:len(s)/2], c)
        go sum(s[len(s)/2:], c)
        x, y := <-c, <-c // receive from c
        fmt.Println(x, y, x+y)
}
```

- Distributed sum: sum is distributed between two goroutines
- Example of applying the common SPMD pattern for parallelism

Channels: example

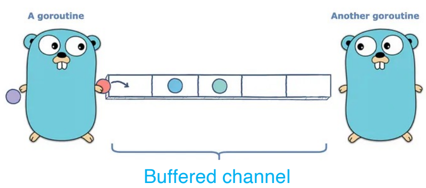
```
package main
import "fmt"
func fib(c chan int) {
        x, y := 0, 1
        for {
                c <- x
                x, y = y, x+y
        }
}
func main() {
        c := make(chan int)
        go fib(c)
        for i := 0; i < 10; i++ {
                fmt.Println(<-c)
        }
}
```

- Fibonacci sequence: iterative version using channel

`x, y = y, x+y` → Elegant and efficient!

Buffered channels

- By default (i.e., *unbuffered channel*), channel ops block
  – Go spec.: *If the capacity is zero or absent, the channel is unbuffered and communication succeeds only when both a sender and receiver are ready. If the channel is unbuffered, the sender blocks until the receiver has received the value*
  https://go.dev/ref/spec#Channel_types
- Buffered channels do not block if they are not full or not empty
  – Specify buffer capacity as `make`'s second argument
    `ch := make(chan int, 100)`
    - If capacity is zero or absent, channel is unbuffered
  – Send to buffered channel blocks only when buffer is full
  – Receive from buffered channel blocks only when buffer is empty (no data to receive)



More on channels: close and range

- How to close channel
  - Use `close` function
  - Receiver can test whether a channel has been closed by assigning a second value to receive

    `v, ok := <- ch`

    - ok is false if there are no more values to receive and channel has been closed
  - Only sender should `close` a channel, never receiver
    - Sending on closed channel causes run-time panic

      `panic: send on closed channel`
  - See example https://gobyexample.com/closing-channels
- Use `range` to receive values from channel repeatedly until it is closed

```
for elem := range ch {
    fmt.Println(elem)
}
```


More on channels: select

- **`select`** allows a goroutine to wait on multiple channels at once
  - Blocks until one of its cases can run, then executes that case
  - One at random if multiple cases are ready

    Go spec.: *If one or more of the communications can proceed, a single one that can proceed is chosen via a uniform pseudo-random selection. Otherwise, if there is a default case, that case is chosen. If there is no default case, the "select" statement blocks until at least one of the communications can proceed.* https://go.dev/ref/spec#Select_statements

```
select {
    case mgs1 := <-ch1:  // receive on ch1
          // ...
    case msg2 := <-ch2:  // receive on ch2
          // ...use x...
    }
```

Using select: example

- Fibonacci sequence: iterative version using two channels, the latter being used to quit

```
package main
import "fmt"

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


Using select: example

```
func main() {
        c := make(chan int)   // unbuffered channel
        quit := make(chan int)
        go func() {           // anonymous function
                for i := 0; i < 10; i++ {
                        fmt.Println(<-c) // receive Fibonacci val.
                }
                quit <- 0
        }()
        fibonacci(c, quit)
}
```

More on channels: select

- You can use `select` with a `default` clause to implement *non-blocking* sends, receives, and even non-blocking multi-way selects

```
select {
    case mgs1 := <-ch1:        // receive
        // ...
    case msg2 := <-ch2:        // receive
        // ...use x...
    case ch3 <-msg3:           // send
        // ...
    default:
        // ...
    }
```

See example with non-blocking channel operations
https://gobyexample.com/non-blocking-channel-operations

Timers

- You can implement timeouts by using a timer channel
  - You tell the timer how long you want to wait, and it provides a channel that will be notified at that time

```
// to wait 2 seconds
timer := time.NewTimer(time.Second * 2)
    <- timer.C
```

  - `<-timer.C` blocks on timer's channel `C` until it sends a value indicating that the timer fired
  - Timer can be canceled before it fires using `Stop()`
  - See example https://gobyexample.com/timers

Exercise: Implement mutex using channel

- Go also provides mutexes to safely access shared data across multiple goroutines
  - See example https://gobyexample.com/mutexes
- Let’s implement mutex using channel

```
type Lock struct {
    // ?
}
func NewLock() Lock {
    // ?
}
func (l *Lock) Lock() {
    // ?
}
func (l *Lock) Unlock() {
    // ?
}
```


Exercise: Implement mutex using channel

```
type Lock struct {
        ch chan bool
}
func NewLock() Lock {
        lock := Lock{make(chan bool, 1)}
        lock.ch <- true // send
        return lock
}
func (l *Lock) Lock() {
        <-l.ch // receive
}
func (l *Lock) Unlock() {
        l.ch <- true // send
}
```

Wait group

- Another synchronization primitive is `sync.WaitGroup`
- Allows co-operating goroutines to collectively wait for an event before proceeding independently again
- Like a concurrent-safe counter: functions `Add`, `Done`, and `Wait`
- When to use
  1. When cleaning up, to ensure that all goroutines (main included) wait before all terminating
     - See example https://gobyexample.com/waitgroups
  2. Cyclic algorithm with a set of goroutines that work independently for a while, then wait on a barrier, before proceeding independently again; data might be exchanged at barrier
     - Aka **Bulk Synchronous Parallel (BSP)** pattern https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

A few more things

- Modules
- Variadic functions
- Error handling
- Go tools
- Testing and benchmarking
- RPC in Go
  http://www.ce.uniroma2.it/courses/sdcc2425/slides/DS_Communication1.pdf

- Many other things, but this is just an introduction!
  - E.g., HTTP support in `net/http` package

Go modules

- **Module**: collection of related Go packages stored in a file tree with a `go.mod` file at its root
- go.mod file defines:
  - module path, which is also the import path used for root directory
  - minimum version of Go required by module
  - its dependency requirements, which are the other modules needed for a successful build with their minimum version
- To generate `go.mod` file:

```
$ go mod init <module_name>
```

```
module mymodule

go 1.16

require (
	github.com/inconshreveable/mousetrap v1.0.0 // indirect
	github.com/spf13/cobra v1.2.1 // indirect
	github.com/spf13/pflag v1.0.5 // indirect
)
```

- To add missing (and remove unused) module requirements:

```
$ go mod tidy
```

https://go.dev/doc/tutorial/create-module
https://www.digitalocean.com/community/tutorials/how-to-use-go-modules

Variadic functions

- Go functions can be called with a varying number of arguments: variadic functions
  - E.g., `fmt.Println` is a variadic function

```
package main

import "fmt"

func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}
```

```
func main() {

    sum(1, 2)
    sum(1, 2, 3)

    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

```
$ go run variadic-functions.go
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

Error handling

- Go code uses error values to indicate abnormal state
- Errors are communicated via explicit, separate return value

  "Error handling [in Go] does not obscure the flow of control." (R. Pike)
  - By convention, last value returned by functions
  - `nil` value in error position means no error

```
result, err := SomeFunction()
if err != nil {
    // handle error
}
```

- Built-in `error` interface type in package `errors`

```
type error interface {
    Error() string
}
```

  - `errors.New` constructs a basic error value with its message

Common errors and recommended tools

- Go can be somewhat picky
  - Unused variables raise errors, not warnings
    - Use blank identifier "_" for variables you don't care about (e.g., the loop index when you need only the value)
  - In `if-else` statement `{` must be placed at the end of the same line, e.g.

```
} else {
} else if … {
```

  - Unused imports raise errors

- Recommended command-line tools:
  - gofmt to format code https://pkg.go.dev/cmd/gofmt

```
$ gofmt -w yourcode.go
```

  - goimports to automatically add/remove imports https://pkg.go.dev/golang.org/x/tools/cmd/goimports
  - godoc to browse package documentation https://pkg.go.dev/golang.org/x/tools/cmd/godoc

Testing and benchmarking in Go

- Go `testing` package provides tools to write unit tests
  https://pkg.go.dev/testing
- To run tests:
  `$ go test`
- Code to be tested is in a given source file (e.g., math.go)
- Test file for it ends `_test.go` (e.g., math_test.go)
  - Call `func TestXxx(*testing.T)` where `Xxx` is the name of the tested function

```
func TestAbs(t *testing.T) {
  got := Abs(-1)
  if got != 1 {
    t.Errorf("Abs(-1) = %d; want 1", got)
  }
}
```


Testing and benchmarking in Go

- Use benchmarking to measure code performance
- Benchmark tests are in `_test.go` files and are named beginning with `Benchmark`
- The testing runner executes each benchmark function several times, increasing `b.N` on each run until it collects a precise measurement
  - A benchmark runs a function in a loop `b.N` times

```
func BenchmarkXxx(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Xxx(…)
    }
}
```

- To run benchmarks
  `$ go test -bench=.`
- Example: let's benchmark `make` vs. `append` on slice

References

- Go website https://go.dev/
- Go standard library https://pkg.go.dev/std
- Online Go tutorial https://go.dev/tour
- Go playground https://go.dev/play
- Go by examples https://gobyexample.com

- Donovan and Kernighan, The Go programming language, Addison-Wesley, 2016 https://www.gopl.io/
- Learn Go programming (7 hours video) https://www.youtube.com/watch?v=YS4e4q9oBaU
- How to code in Go https://www.digitalocean.com/community/tutorial-series/how-to-code-in-go
- More resources https://go.dev/learn