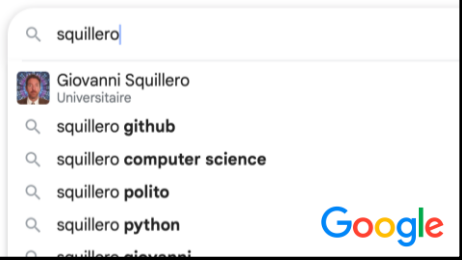
Go vs. Python

Day 1: Choose your weapon

Giovanni Squillero
giovanni.squillero@polito.it

… but first

- **Giovanni Squillero**
  - Enrolled in an M.S. in Electronics Engineering, then earned a Ph.D. in Computer Engineering, ended up working in Computer Science.
  - Now serving as a full professor at Politecnico di Torino (Italy).
  - Playing with bio-inspired meta heuristics since mid-1990s.
  - Well-known to “expanded west” search engines (Google, Bing, Yandex, …), less so to eastern ones (Baidu, Sogou, …)



Agenda

- Introduction.
- Basic syntax and primitive types.
- Functions.
- Composite types.
- Error handling.
- Concurrency.

Agenda: Composite types

- Go arrays and slices ↔ Pythonic lists.
- Go maps ↔ Pythonic dicts.
- Go structs ↔ Pythonic classes.

Agenda: Functions

- Void functions and multiple return values in Go.
- Methods on structs!!!

Agenda: Duck typing

- Polymorphism ↔ Duck typing:
  - Go structs + composition.
  - Pythonic objects + inheritance.
  - Pythonic instance method dispatch (bound methods).

Agenda

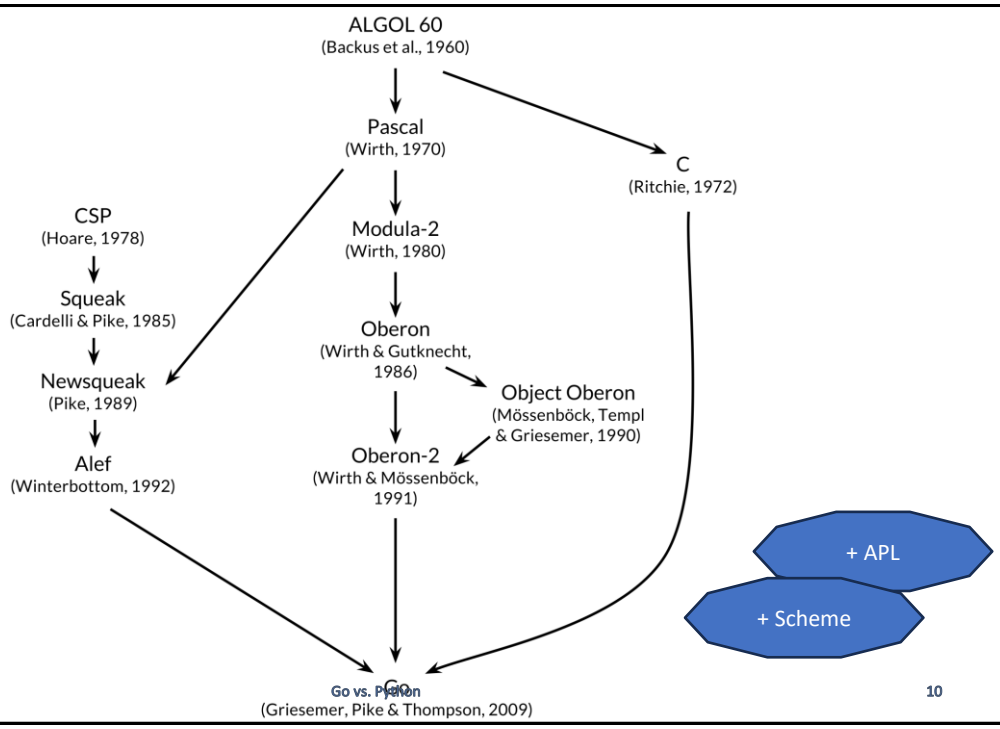
- Introduction.
- Basic syntax an
- Functions.
- Composite types.
- Error handling.
- Concurrency.

Go

- Modern, compact, concise, general-purpose.
- Imperative, statically type-checked, dynamically type-safe.
- Garbage-collected.
- No warnings, unused local vars and imports are an error.
- Strong support for concurrency.
- Fast compilation to native code, statically linked.
- Reasonably efficient execution.

Go

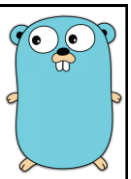
- Started in 2007 by Rob Pike, Robert Griesemer, Ken Thompson.
- Russ Cox and Ian Lance Taylor joined soon after.
- Folklore: “Designed while waiting for a C++ program to compile”.



Introduction

- Everything is Unicode!
- Programs are organized in packages.
- A package is a set of package files.
- A package file expresses its dependencies on other packages via import declarations.
- The remainder of a package file is a list of (constant, variable, type, and function) declarations.

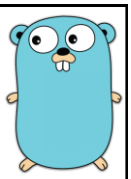
Python vs. Go



- Focus: Readability.
- Speed of development.
- De-facto standard for ML and AI research.
- Probably *the* most used programming language in 2025.

- Focus: Simplicity.
- Usable on large codebases (scalable).
- Cloud-Native and Infrastructure Tooling.
- High-traffic backends API and microservices.
- Significant user base, typically ranked among the top10/top20 most widely used languages.

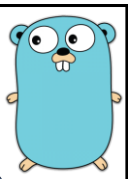
Python vs. Go



- Highly portable.
- Interpreted (Bytecode):
  - Code is translated into bytecode.
  - Run by a Virtual Machine (PVM).

- Highly portable.
- Compiled (Machine Code):
  - Compiled and statically linked.
  - Native binary runs on the hardware.

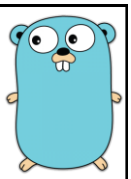
Python vs. Go



- Dynamic name binding.
- Strongly typed objects.
- Type checking possible only at run time.

- Type declared at compile-time.
- Very precise typing.
- No automatic casting.
- Quite effective static type checking.

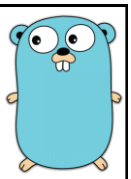
Python vs. Go



- Inherently sequential.
- GIL!
- Heavy process-based parallelism.
- `asyncio`.
- Note: Since Python v3.13 experimental *free-threaded mode.*

- Native parallelism.
- Goroutines (lightweight threads).
- Channels.

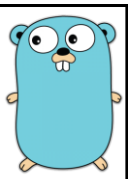
Python vs. Go



- Moderate performances.
- Slower for CPU-heavy tasks due to overheads and dynamic checking.
- Ideal as a “glue” language with optimized libraries.

- High performance.
- Slower than Rust.
- Ideal for high-throughput backend services.

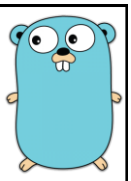
Python vs. Go



- Heavy overhead in memory management.
- Reference counting.
- Background garbage collector.

- Highly efficient memory management.
- Better cache locality.
- Low-latency background garbage collector.

Python vs. Go



- Error handling through Exceptions.
- Uses `try/except` blocks to catch errors at runtime.

- Explicit error handling.
- Functions return error values that the developer must check manually.

Kickstart

```
> winget install GoLang.Go
```


Kickstart

```
$ brew install golang
```

```
> winget install GoLang.Go
```

```
$ cd ~/Downloads
$ wget -c https://go.dev/dl/go1.26.1.linux-amd64.tar.gz
$ sudo tar -C /usr/local/ -xzf go1.26.1.linux-amd64.tar.gz
$ export PATH=$PATH:/usr/local/go/bin
```

Kickstart

```
$ brew install golang
```

```
> winget install GoLang.Go
```

```
$ cd ~/Downloads
$ wget -c https://go.dev/dl/go1.26.1.linux-amd64.tar.gz
$ sudo tar -C /usr/local/ -xzf go1.26.1.linux-amd64.tar.gz
$ export PATH=$PATH:/usr/local/go/bin
```

```
> go version
go version go1.26.1
> mkdir hello-world
> cd hello-world
> go mod init hello-world
go: creating new go.mod: module hello-world
> code main.go
> go run hello-world
Hello, AROL!
```

main.go > ...

```go
// Copyright 2026 Giovanni Squillero
// SPDX-License-Identifier: 0BSD

package main

import "fmt"

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


import

- Modify the program (blue arrow).
- Save.
- Imports are *automagically* updated.

main.go > ...

```go
// Copyright 2026 Giovanni Squillero
// SPDX-License-Identifier: 0BSD

package main

import (
    "fmt"
    "os"
)

func main() {
    name := os.Getenv("USER")
    fmt.Printf("Well done %s!\n", name)
}
```

go mod

- Explicitly define dependencies.
- Ensure reproducibility.
- Essential commands:
  - `go mod init <name>`
  - `go mod tidy`
  - `go mod verify`
  - `go get <package>@version`

import (external)

main.go > ...

```go
// Copyright 2026 Giovanni Squillero
// SPDX-License-Identifier: 0BSD

package main

import (
    "time"

    "github.com/briandowns/spinner"
)

func main() {
    s := spinner.New(spinner.CharSets[9], 100*time.Millisecond)
    s.Start()
    time.Sleep(4 * time.Second)
    s.Stop()
}
```

External Imports

```
> go run hello-world
main.go:9:2: no required module provides package github.com/briandowns/spinner
> go get github.com/briandowns/spinner
go: downloading github.com/briandowns/spinner v1.23.2
go: downloading github.com/fatih/color v1.7.0
go: downloading golang.org/x/term v0.1.0
go: downloading github.com/mattn/go-colorable v0.1.2
go: downloading github.com/mattn/go-isatty v0.0.8
go: downloading golang.org/x/sys v0.0.0-20220412211240-33da011f77ad
go: added github.com/briandowns/spinner v1.23.2
go: added github.com/fatih/color v1.7.0
go: added github.com/mattn/go-colorable v0.1.2
go: added github.com/mattn/go-isatty v0.0.8
go: added golang.org/x/sys v0.0.0-20220412211240-33da011f77ad
go: added golang.org/x/term v0.1.0
> go run hello-world
```


go mod tidy

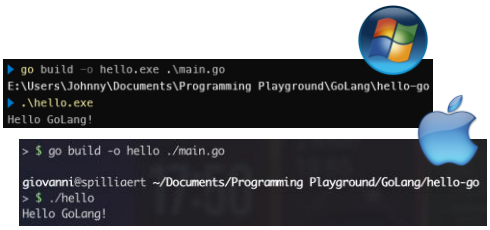
- Run: `go mod tidy`

go.mod

```
1  // Copyright 2026 Giovanni Squillero
2  // SPDX-License-Identifier: 0BSD
3
   Reset go.mod diagnostics | Run govulncheck | Run go mod tidy | Create vendor directory
4  module hello-world
5
6  go 1.26.1
7
   Check for upgrades | Upgrade transitive dependencies | Upgrade direct dependencies
8  require github.com/briandowns/spinner v1.23.2
9
10 require (
11     github.com/fatih/color v1.7.0 // indirect
12     github.com/mattn/go-colorable v0.1.2 // indirect
13     github.com/mattn/go-isatty v0.0.8 // indirect
14     golang.org/x/sys v0.0.0-20220412211240-33da011f77ad // indirect
15     golang.org/x/term v0.1.0 // indirect
16 )
17
```

go build

- Creates a single, standalone binary executable:
  - Dependency Resolution.
  - Compilation.
  - Linking.
- Note: Go compiler is so fast that you can safely skip compilation.



go vet

- Static linter.
- Examines code for “suspicious constructs”.
- Find potential bugs/errors.

Agenda

- Introduction.
- Basic syntax and primitive types.
- Functions.
- Composite types.
- Error handling.
- Concurrency.

25 reserved 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

Programming Language	Number of Keywords
C (ANSI C89)	32
C (C11)	44
C++ (C++20)	92
C# (8.0)	107
Dart (2.2)	33
Go (1.18)	25
Java (SE 17 LTS)	67
JavaScript (6th edition)	46
Python 3 (3.10)	38
Ruby (2.7)	41
Rust (1.46)	53
Swift (5.3)	97
Visual Basic (2019)	217

Predeclared names

- Constants:
  - true, false, iota, nil
- Types:
  - Integers: int, int8, int16, int32, int64
  - Unsigned Integers: uint, uint8, uint16, uint32, uint64, uintptr
  - Floating/Complex: float32, float64, complex128, complex64
  - Others: bool, byte, rune, string, error

Predeclared names

- Functions:
  - Memory/Allocation: make, new
  - Collections: len, cap, append, copy, delete
  - Channel Handling: close
  - Complex Numbers: complex, real, imag
  - Error Handling: panic, recover

Linking class

- Exported names are **Capitalized**
- Local names are **lowercase**

Literals

```
42, 4_294_967_296, 0xff, 0x10p2

3.14, 1.2e3, 3.14E0

'G', '\107', '\x47', '\u0047', '\U00000047',
'\t', '\n'

"Hello\nGoodbye",
`Hello
Goodbye`
```

Variables

- Explicit declaration:

```
var name type
```

- Explicit declaration and initialization:

```
var name type = value
```

- Explicit declaration with inferred type and initialization:

```
var name = value
```

- Initialization and implicit declaration:

```
name := value
```


Variables

- Explicit declaration:

```
var n1, n2, … type
```

- Explicit declaration and initialization:

```
var n1, n2, … type = v1, v2, …
```

- Explicit declaration with inferred types and initialization:

```
var n1, n2, … = v1, v2, …
```

Data types: Integer

- Integer, all standard operators are supported.
- No mixed data, explicit casting.

```
var a, b int8    // 8-bit integer (also 16, 32, 64)
var c uint64 = 3 // 64-bit unsigned integer  (also 8, 16, 32)
var d int        // 32/64-bit integer (platform dependent)
e := 4           // same as int

f := d + int(a) // explicit cast is always required
```

Type name	Value range
int8	–128 to 127
int16	–32768 to 32767
int32	–2147483648 to 2147483647
int64	–9223372036854775808 to 9223372036854775807
uint8	0 to 255
uint16	0 to 65535
uint32	0 to 4294967295
uint64	0 to 18446744073709551615

Data types: Floating point

- Everything as expected.

```
var a float32    // single precision
var b float64    // double precision
c := 3.14        // default is float64
var d complex64  // two float32
var e complex128 // two float64
f := 23 + 10i    // default is complex128
```

Type name	Largest absolute value	Smallest (nonzero) absolute value
float32	3.402823466385288598117041834845169254 40e+38	1.401298464324817070923729583289916131280e-45
float64	1.797693134862315708145274237317043567981e+308	4.940656458412465441765687928682213723651e-324

Data types: Runes and Strings

- Strings support indexing and concatenation, stored in UTF-8.
- Runes are Unicode points stored as int32.

```
    var a rune   // unicode point
    b := '无'     // a rune
    var c string // sequence of runes
    bts := `알앰
슈가
제이홉
진
정국
뷔
지민` // multiline string
```


Data types: `any`

- Introduced in Go1.18
- A variable declared as **`any`** has a static type of **`any`**
- At runtime, it possesses a dynamic type (the type of the value it currently holds).

Data types: pointers

- A pointer is a variable that stores the memory address of another value.
- Designed for memory efficiency and controlled mutability, not arbitrary memory manipulation.
- No pointer arithmetic (e.g., `p++`).
- Escape Analysis!
- Caveat: Slices, Maps, and Channels all contain pointers to underlying arrays.

```
a := 41
b := &a
*b++
```

```
func foo(a int) *int {
    var b int
    b = a*2 + 1
    return &b
}
```


Type definition and type alias

- **Type definition** is the mechanism used to create a new, distinct type from an existing one.
- Introduced in Go 1.9, a **type alias** creates an alternative name for an existing one.
- Remember all the static type checking!

```
type BYTE uint8   // BYTE + uint8 is illegal (type mismatch)
type BYTE2 BYTE   // BYTE + BYTE2 is illegal (type mismatch)
type BYTE3 = BYTE // BYTE + BYTE3 is legal!
```

Blank identifier

- The blank identifier _ (underscore) can be assigned or declared with any value of any type, with its value discarded harmlessly.
- A write-only syntactic place-holder where a variable is needed but the actual value is irrelevant.

Constants and `iota`

- Constants may or may not have a type.

```
const BIRTHDAY = 23

var a int8 = BIRTHDAY   // valid: BIRTHDAY is "23"
var b uint32 = BIRTHDAY // valid: BIRTHDAY is "23"

const BYTE uint8 = 23

var c uint8 = BYTE  // valid
var d int8 = BYTE   // invalid: BYTE is a uint8
var e uint32 = BYTE // invalid: BYTE is a uint8
```

```
const (
    JOHN = iota
    PAUL
    GEORGE
    RINGO
)
```

Flow control: `if`

- The usual if, with a possible operation before.

```
if name, surname := getUser(); name == "Giovanni" && surname == "Squillero" {
    log.Println("Whoa!")
}
```


Flow control: `switch`

- Several improvements over C/C++.
- Spoiler alert: type switch!

```
switch num := rand.Int() % 100; num {
case 0, 1, 3:
    log.Println("How small!")
case 97, 98, 99:
    log.Println("How large!")
default:
    log.Println("D'ho!?")
}
```

```
switch num := rand.Int() % 100; {
case num < 10:
    log.Println("How small!")
case num > 90:
    log.Println("How large!")
}
```

```
func crazy() any {
    switch rand.Int() % 6 {
    case 0:
        return int(42)
    case 1:
        return int8(42)
    case 2:
        return int16(42)
    case 3:
        return int16(42)
    case 4:
        return int32(42)
    case 5:
        return int64(42)
    }
    return nil
}
```

Flow control: `for`

- Classical, modern, and fancier for loops (spoiler alert)

```go
// classical for loop
for i := 0; i < 10; i++ {
    log.Println(i)
}

// classical for loop (modern)
for i := range 10 {
    log.Println(i)
}
```

```go
// a foreach loop
for i, v := range []int{1, 2, 3, 4} {
    log.Println(i, v)
}
```

```go
// a while loop
for num != 1 {
    if num%2 == 0 {
        num = num / 2
    } else {
        num = 3*num + 1
    }
}
```

```go
// an idiomatic endless loop
for {
    log.Println("Forever")
}
```


Agenda

- Introduction.
- Basic syntax and primitive types.
- Functions.
- Composite types.
- Error handling.
- Concurrency.

Functions

- First-class citizens.
- Call by sharing (name binding).
- Types are checked at runtime.
- Support named arguments.
- Return values packing.
- Anonymous functions through `lambda` (1 line).
- Full variadic params `*args` and `**kwargs`.

- First-class citizens.
- Call by values
- Statically typed.
- Support named return values.
- Native multiple return values.
- Full support for anonymous functions.
- Limited variadic params through `...type`.

Function definition (by example)

- `func simple_average(a, b int) float64`
- `func simple_average(a, b int) (x float64)`
  - Takes two integers, returns a float.
- `func average(values ...int) float64`
  - Takes any number of integers, returns a float.
- `func divmod(a, b int) (int, int)`
  - Takes any number of integers, returns two integers.

Function definition (by example)

- `func simple_average(a, b int) float64`
- `func simple_average(a, b int) (avg float64)`
  - Takes two intege
- `func average(`
  - Takes any numbe
- `func divmod(`
  - Takes any number of integers, returns two integers.
- `func average(values ...int) float64`
  - Any number of integer arguments, returning a float.

```
func sumUp(values ...int) (total int) {
    total = 0
    for _, n := range values {
        total += n
    }
    return
}
```


Recursion & Deferred executions

```
func zap(n int) {
    defer log.Println(n)

    switch {
    case n == 1:
        return
    case n%2 == 0:
        zap(n / 2)
    case n%2 == 1:
        zap(3*n + 1)
    }
}
```

```
16:10:50.856076 bar_file.go:50: 1
16:10:50.856091 bar_file.go:57: 2
16:10:50.856093 bar_file.go:57: 4
16:10:50.856095 bar_file.go:57: 8
16:10:50.856097 bar_file.go:57: 16
16:10:50.856099 bar_file.go:57: 5
16:10:50.856101 bar_file.go:57: 10
16:10:50.856103 bar_file.go:57: 20
16:10:50.856105 bar_file.go:57: 40
16:10:50.856106 bar_file.go:57: 13
16:10:50.856108 bar_file.go:57: 26
16:10:50.856110 bar_file.go:57: 52
16:10:50.856111 bar_file.go:57: 17
16:10:50.856113 bar_file.go:57: 34
16:10:50.856115 bar_file.go:57: 11
16:10:50.856117 bar_file.go:57: 22
16:10:50.856118 bar_file.go:57: 7
```

Scope

```go
foo := 1
bar := 2
slog.Info("Outer scope:", "foo", foo, "bar", bar)
{
    slog.Info("Inner scope:", "foo", foo, "bar", bar)
    foo = 42
    bar := 42
    slog.Info("Inner scope:", "foo", foo, "bar", bar)
}
slog.Info("Outer scope:", "foo", foo, "bar", bar)
```

```
main.go:16: INFO Outer scope: foo=1 bar=2
main.go:18: INFO Inner scope: foo=1 bar=2
main.go:21: INFO Inner scope: foo=42 bar=42
main.go:23: INFO Outer scope: foo=42 bar=2
```


Scope and Closures

- (Lexical) closure or function value: a record storing a function together with an environment.
- Lexical (static) scope.

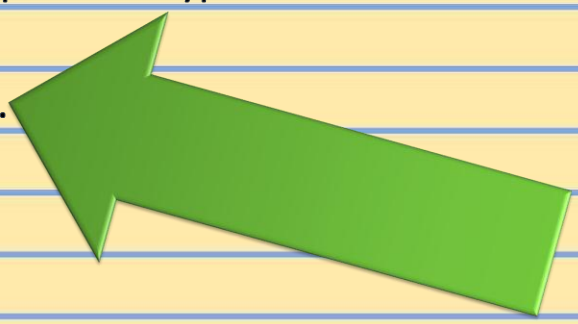
```go
func makeInc(offset int) func(n int) int {
    return func(n int) int {
        offset *= 2
        return offset + n
    }
}

func Bar() {
    i10 := makeInc(10)
    i100 := makeInc(100)
    log.Println(i10(1), i10(1), i100(1), i10(1))
}
```

```
bar_file.go:51: 21 41 202 81
```

Agenda

- Introduction.
- Basic syntax and primitive types.
- Functions.
- Composite types.
- Error handling.
- Concurrency.



Agenda: Composite types

- Go arrays and slices ↔ Pythonic lists.
- Go maps ↔ Pythonic dicts.
- Go structs ↔ Pythonic classes.

Arrays

- Numbered sequences of elements of a specific length and specific type:
  - Single, contiguous block of memory — size must be known at compile time (e.g., defined via a constant).
  - The length is part of the type.
  - "..." represent the number of items provided in the initialization.

```
var beatles1 [4]string = [4]string{"John", "Paul", "George", "Ringo"}
var beatles2 [4]string = [...]string{"John", "Paul", "George", "Ringo"}
var beatles4 = [4]string{"John", "Paul", "George", "Ringo"}
var beatles3 = [...]string{"John", "Paul", "George", "Ringo"}
beatles5 := [4]string{"John", "Paul", "George", "Ringo"}
beatles6 := [...]string{"John", "Paul", "George", "Ringo"}
```


Arrays

- Arrays are always initialized to default type value.
- Partial initialization is allowed.

```
var partial = [1024]int{23: 1, 10: 1, 18: 2, 5: 2}
```

Arrays

- Bounds are always checked.

```
var array [2]int
array[0] = 23
array[2] = -1 // error: out of bounds
```


Slices

- Descriptors for an underlying array.
- Lightweight data structure that provides a window into an array's data.

```
var beatles7 []string = []string{"John", "Paul", "George", "Ringo"}
var beatles8 = []string{"John", "Paul", "George", "Ringo"}
beatles9 := []string{"John", "Paul", "George", "Ringo"}
```

Slice functions

- `len(slice)`
  - Number of element in the underlying array.
- `cap(slice)`
  - Capacity of the underlying array.
- `make(type, length[, capacity])`
  - Creates a slice with an underlying array of size *length* that coul. expand to *capacity* without relocation
- `slice = append(slice, e1, …)`
  - Appends the elements to the slice, with possible relocation.

Arrays & Slices (conversion)

- Native, built-in feature (Go1.17+ and Go1.20+).

```
var array = [...]int{23, 10, 18, 5}
var slice = array[:]
```

```
var slice = []int{23, 10, 18, 5}
var array = [2]int(slice)
```

List vs. Slice

- Reasonably fast.
- Heterogeneous.
- Can grow (`append`).
- Can shrink (`pop`, slicing, …).
- Powerful slicing.
- `l1 = l2` create an alias.
- Hidden details.
- `l1 == l2` compare content.

- Very fast.
- Homogeneous.
- Can grow (`append`) .
- Sort of shrink (`s = s[:n]`).
- Some slicing (windows).
- `s1 := s2` copy the header.
- Visible details (`cap`).
- `s1 == s2` is invalid.

copy

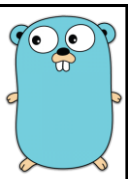
- Copies elements from a source slice into a destination slice
- The source and destination may overlap.
- Returns the number of elements copied, which will be the minimum of `len(src)` and `len(dst)` .

Maps

- Built-in, unordered collections of key-value pairs.
- A map is a reference to a hash-table data structure.
- The key type must be comparable (i.e., **`int`**, **`string`**, **`float`**, **`bool`**, **`pointers`**, **`channels`**, and **`structs`** where all fields are comparable.

```
map[keyType]dataType
```


Dictionary vs. Map



- Heterogeneous.
- Keys must be hashable.
- Add key: **`d[k] = v`**
- Growth strategy: Resizes the entire table.
- Delete key: **`del d[k]`**
- Clear all: **`d.clear()`**
- Missing key: Error.
- **`d2 == d2`** compare content.
- Predictable traversing order.

- Homogeneous.
- Keys must be comparable.
- Add key: **`m[k] = v`**
- Growth strategy: *Incremental evacuation* (no latency spikes).
- Delete key: **`delete(m, k)`**
- Clear all: **`clear(m)`**
- Missing key: Safe (default zero).
- **`m1 == m2`** is invalid.
- Unpredictable traversing order.

Structs

- Composite literal types that groups together zero or more named values (“fields”) of arbitrary types as a single entity.
- A struct is a contiguous block of memory where each field is laid out in the order it is defined.

```
struct {
      field type [`field tag`]
      …
}
```


Structs

- “Usually” used with type definitions, but anonymous structs are legit.

```
type Person struct {
    name    string
    surname string
}
var foo, bar Person

var point struct {
    x, y int
}
point.x = 12
```

Structs

- **Field tags** allow to attach metadata to fields.
- Tags can be read at runtime using the **`reflect`** package.
- The most common use cases are for serialization, DB mapping, and validation.

```
type User struct {
    FirstName string `json:"first_name"`      // Rename "FirstName" to "first_name" in JSON
    LastName  string `json:"last_name"`       // Rename "LastName" to "last_name" in JSON
    Password  string `json:"-"`               // "-" means: always ignore this field
    Email     string `json:"email,omitempty"` // Hide this field if it is an empty string
}
```

```
type Product struct {
    ID    uint   `gorm:"primaryKey"`          // Marks this as the DB Primary Key
    Code  string `gorm:"column:product_code"` // Maps to column named "product_code"
    Price int64  `gorm:"default:100"`         // Sets a default value in the DB schema
}
```

```
type SignupRequest struct {
    Username string `validate:"required,min=3"` // Must exist and be at least 3 chars
    Age      int    `validate:"gte=18,lte=130"` // "Greater Than or Equal to 18"
    Email    string `validate:"required,email"` // Must be a valid email format
}
```


Methods (syntactic sugar)

```
type Friend struct {
      name, surname string
}

func (f Friend) Greet() {
      fmt.Printf("Hello %v!\n", f.name)
}

func main() {
      giovanni := Friend{name: "Giovanni", surname: "Squillero"}
      giovanni.Greet()
}
```

```
> $ go run data.go
Hello Giovanni!
```

Class vs. Struct

- Dynamic (unless `__slots__`).
- Methods defined inside the class.
- Fragmented memory layout.
- `s1 = s2` creates an alias.
- Custom default values.
- Metadata: Decorators and type hints.

- Static (compile time).
- Receiver functions.
- Contiguous memory layout.
- `s1 = s2` copies all fields.
- Zero defaults value.
- Metadata: Field tags and inspection.

Interfaces

- A type defined strictly by a method set.
- Implicitly satisfied (no need to declare that it is implemented).
- Static vs. dynamic type.

```
interface {
        function(arg, arg, …) type
        …
}
```

Interfaces

- Idiomatic interfaces:
  - Used with type definitions.
  - Very small (few functions).
  - Functional nomenclature (-er).

```
type jumper interface {
    jump() int
}

var carl interface {
    run() int
}
```


Interfaces

- Used for decoupling and abstraction (sometime with *type assertion and type switch*).
- **`any`**, i.e., the empty interface **`interface{}`**, matches any type.

```
func jump(kid jumper) int {
```

Pointers

- Go automatically handles conversion between values and pointers for method calls.
- Use a pointer receiver type to avoid copying on method calls or to allow the method to mutate the receiving struct.

Methods (syntactic sugar)

```
type Friend struct {
    name, surname string
}

func (f Friend) Greet() {
    fmt.Printf("Hello %v!\n", f.name)
}

func main() {
    g1 := Friend{name: "Giovanni", surname: "Squillero"}
    g2 := &g1
    g1.Greet()
    g2.Greet()
}
```

```
> $ go run data.go
Hello Giovanni!
Hello Giovanni!
```

Methods (syntactic sugar)

```
type Friend struct {
    name, surname string
}

func (f *Friend) Greet() {
    fmt.Printf("Hello %v!\n", f.name)
}

func main() {
    g1 := Friend{name: "Giovanni", surname: "Squillero"}
    g2 := &g1
    g1.Greet()
    g2.Greet()
}
```

```
> $ go run data.go
Hello Giovanni!
Hello Giovanni!
```


Interfaces: Type assertion

- Runtime operation to extract an interface underlying concrete value (or to convert it to a different interface type).
- Unlike type conversion, which changes the representation, a type assertion is a dynamic check of the reified type metadata.

```
var unknown any = "Bob"

n1 := int(unknown)      // error: Invalid Conversion
n2 := unknown.(int)     // unsafe
n3, ok := unknown.(int) // safe
```

Interfaces: Type switch

- A specialized control structure that performs a series of type assertions in a single block.
- I.e., a multi-way conditional branch that matches the dynamic type of an interface against a list of specific types.

```
switch unknown.(type) {
case int:
    n4 = 23
case string:
    n4 = 10
default:
    n4 = -1
}
```


Channels

- First-class, thread-safe, FIFO queues managed by the Go runtime:
  - Unbuffered with no capacity to store values — used as rendezvous to guarantee synchronization between goroutines.
  - Buffered with fixed-size capacity — used to decouples sender and receiver, allowing them to operate at different speeds.

```
channel := make(chan type, size)
```

Channel operation

- `channel <- value`
  - Send element to channel.
- `<-channel`
  - Receive from channel.
- `close(channel)`
  - Close channel.

```go
buffered_channel := make(chan int, 1)
buffered_channel <- 42
log.Println(<-buffered_channel)

unbuffered_channel := make(chan int)
unbuffered_channel <- 42 // all goroutines are asleep - deadlock!
log.Println(<-unbuffered_channel)
```


range

- Provides a way to iterate over an array, slice, string, map, or channel.
- Built-in immutable sequence type used for generating arithmetic progressions of integers.

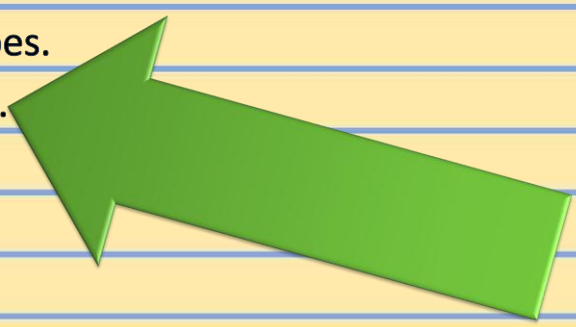
range vs. range

- `range(s, [e [, i]])`
- Built-in immutable sequence type generating arithmetic progressions of integers.
- Integers are generated on-demand (lazily).
- Highly memory-efficient.

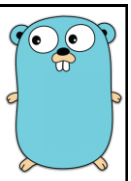
- `range(collection)`
- Keyword: language construct designed for traversing collections.
- Returns 2 values, types depend on collection type.
- In Go1.22+ `range n` mocks a Pythonic `range(0, n, 1)`.
- Loop variable capture in Go1.22+
- Highly memory-efficient.

Agenda

- Introduction.
- Basic syntax and primitive types.
- Functions.
- Composite types.
- Error handling.
- Concurrency.



Errors



- Implicit propagation (bubble up).
- `try/except/raise`
- Overhead for stack unwinding.
- Errors can be hidden.
- Separates error logic from business logic.
- Both expected failures and emergencies.

- Explicit propagation.
- `val, err := f()`
- No overhead.
- Errors in function signature.
- Interleaves error handling with business logic.
- Emergencies: `panic()` and `recover()` .

The “ok” idiom

- Resolves ambiguity in returning a single value from a data structure that might not contain the requested key.
- Available for `map`, `interface`, and `channel` .

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

```
val, ok := <-ch
```

```
val, ok := i.(int)
```

The “err” idiom

- Used for fallible operations (IO, network, logic).
- Provides an error explaining the failure.
- Functions return (value, error).

```go
val, err = doSomething()
if err == nil {
    log.Println(val)
}
```

```go
func doSomething() (value int, err error) {
    err = nil
    value = 42

    if 1 != 1 {
        err = fmt.Errorf("Panic")
    }

    return
}
```

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


Panic & Recover

```go
func trap() {
    if p := recover(); p != nil {
        log.Printf("Panicking (%v), but recovering...\n", p)
    }
}

func Bar() {
    defer trap()
    panic("serious error")
}
```

Go vs. Python

Day 2: Master the blade

Giovanni Squillero
giovanni.squillero@polito.it

Day1 quick recap

- Basic syntax: data types, variables, loops, functions, etc.
- Mastering modules: `go mod` & external references.
- Handling errors: the value/ok and value/err idioms.
- Duck typing:
  - interfaces.
  - Note: *"The bigger the interface, the weaker the abstraction"*!

Day1 quick recap

- Basic sy… c.
- Master…
- Handlir…
- Duck ty…
  - interf…
  - Note:… "!

```go
package main

import (
    "log"

    "github.com/pterm/pterm"
)

func main() {
    var err error

    pterm.Info.Println("Hello PTerm!")

    beatles := []string{"John", "Paul", "George", "Ringo"}
    selectedOptions, err := pterm.DefaultInteractiveMultiselect.WithOptions(beatles).Show()
    if err != nil {
        log.Panicf("Yeuch: %v\n", err.Error())
    }
    pterm.Info.Println("Selected options: %s", pterm.Green(selectedOptions))
}
```

Go generics

- Available in Go1.18+.
- Compile-time parametric polymorphism.
- Stronger type safety.
- Faster than interfaces.
- GCShape stenciling (not full monomorphization as in C++).
- Size–performance trade-off.

- Available i
- Compile-ti
- Stronger t
- Faster thar
- GCShape s
- Size–perfo

```
import (
    "log"

    "golang.org/x/exp/constraints"
)

func main() {
    log.SetFlags(log.Lmicroseconds + log.Lshortfile)
    log.Println("Yo!")

    z1 := zap(1, 2)
    log.Printf("(%T)%v\n", z1, z1)
    z2 := zap(uint(1), uint(2))
    log.Printf("(%T)%v\n", z2, z2)
    z3 := zap(int8(1), int8(2))
    log.Printf("(%T)%v\n", z3, z3)
}

func zap[T constraints.Integer](a, b T) T {
    return a + b
}
```

```
18:35:32.564745 main.go:11: Yo!
18:35:32.565075 main.go:14: (int)3
18:35:32.565102 main.go:16: (uint)3
18:35:32.565106 main.go:18: (int8)3
```

C++).

Notez Bien

```
// identity(1) + 2 would be an error (MismatchedTypes)
func identity(value any) any {
    return value
}
```

```
// identity(1) + 2 is ok!
func identity[T any](value T) T {
    return value
}
```

Constraints

- Use interfaces!
  - Approximation interfaces (Go1.18+)
- Built-in constraints for generic:
  - **`any`**
  - **`comparable`**
  - **`cmp.Ordered`**

```
type MyInt int
type Number interface {
    ~int | ~float64 // MyInt is ok!
}
```


golang.org/x/exp/constraints

- Signed
- Unsigned
- Integer (i.e., either Signed or Unsigned)
- Float
- Ordered (i.e., either Integer or Float)
- Complex

Comparison

- Go: generics
  - GCShape stenciling and runtime dictionary: reasonably fast.
- C++: templates
  - Full monomorphization: fastest possible execution, but code bloat.
- Java: generics
  - Type erasure: performance impaired by boxing.
- Rust: traits
  - ???

Module `math/rand/v2`

- Non-cryptographic pseudo-random number generators.
- Random numbers are generated by a Source, usually wrapped in a Rand.
- Sharing among multiple goroutines requires some kind of synchronization.
- Use: `rand.Type()` or `rand.Int*N(max)`

Module os

- Unix-like
- Low-level file I/O
- Low-level commandline arguments
- Process management
- Environment variables
- System metadata (e.g., `os.Hostname`)
- Error handling (platform-agnostic error types)

- Unix-lik
- Low-lev
- Low-lev
- Process
- Enviror
- System
- Error h

```go
func printSong_os(filename string) error {
    var err error
    var file *os.File

    file, err = os.Open(filename) // also checkout: os.OpenFile()
    if err != nil {
        return fmt.Errorf("[printSong_os] %v", err)
    }
    defer file.Close()

    const BUFFER_SIZE = 1 << 13 // ie. 8k
    buffer := make([]byte, BUFFER_SIZE)

    // *os.File implements the io.Reader interface
    file.Read(buffer) // nb: max size from len(), not cap()

    log.Println(buffer[:128])
    log.Println(string(buffer))

    return nil
}
```

`bufio.Reader` interface

- Internally buffered (4KiB or more).
- Extra features, e.g., **`ReadString`**, **`ReadLine`**, **`Peek`**.

`Reader` interface

```
reader := bufio.NewReader(file)
for {
    b, err := reader.ReadByte() // Returns a single uint8
    if err == io.EOF {
        break
    }
    fmt.Printf("%c", b) // print the byte as a character, yeuch
}
```

`Reader` interface

```go
file.Seek(0, io.SeekStart) // rewind the file
for {
    rune, _, err := reader.ReadRune() // _ is the size of the rune
    if err == io.EOF {
        break
    } else if err != nil {
        return fmt.Errorf("[printSong_reader] %s", err)
    }
    fmt.Printf("%c", rune) // ;-)
}
return nil
```


`bufio.Scanner` interface

- Probably the most *idiomatic* way to process text files in Go.
- Internally buffered (4KiB or more).
- Fills internal buffer looking for a delimiter via **`Read()`**.
- Expose the data via **`Text()`** or **`Bytes()`**.
- Delimiter can be customized via **`Split()`**.

- Probably
- Internall
- Fills inte
- Expose t
- Delimite

```go
var file *os.File

file, err = os.Open(filename) // checkout: os.OpenFile()
if err != nil {
    return fmt.Errorf("[printSong_scanner] %v", err)
}
defer file.Close()

scanner := bufio.NewScanner(file)
for scanner.Scan() {
    line := scanner.Text() // current line as a string
    fmt.Println(line)
}

if err := scanner.Err(); err != nil {
    return fmt.Errorf("[printSong_scanner] %v", err)
}
return nil
```

in Go.

).

Day 1

https://adventofcode.com/2025/day/1

```
--- Day 1: Secret Entrance ---

The Elves have good news and bad news.

The good news is that they've discovered project management! This has given
them the tools they need to prevent their usual Christmas emergency. For
example, they now know that the North Pole decorations need to be finished
soon so that other critical tasks can start on time.

The bad news is that they've realized they have a different emergency:
according to their resource planning, none of them have any time left to
decorate the North Pole!

To save Christmas, the Elves need you to finish decorating the North Pole by
December 12th.
```


i/o

- Use **os** to open file.
- Use **bufio** to get a Scanner.
- Read line by line with **Scan**.
- Use **Atoi** to convert strings into integers.

```go
file, err := os.Open(filename)
if err != nil {
    log.Fatalf("readRotations: %v\n", err)
}
defer file.Close()

scanner := bufio.NewScanner(file)
for scanner.Scan() {
    line := scanner.Text() // current line as a string
```

Day 2

https://adventofcode.com/2025/day/2

```
--- Day 2: Gift Shop ---

You get inside and take the elevator to its only other stop: the gift shop.
"Thank you for visiting the North Pole!" gleefully exclaims a nearby sign.
You aren't sure who is even allowed to visit the North Pole, but you know
you can access the lobby through here, and from there you can access the
rest of the North Pole base.

As you make your way through the surprisingly extensive selection, one of
the clerks recognizes you and asks for your help.

As it turns out, one of the younger Elves was playing on a gift shop
computer and managed to add a whole bunch of invalid product IDs to their
gift shop database! Surely, it would be no trouble for you to identify the
invalid product IDs for them, right?
```


i/o

- Use **os** to open file.
- Use **bufio** to get a **Scanner**, slurp the only line.
- Get the different blocks with **strings.Split**.
- Use **Atoi** to convert strings into integers.

```go
scanner := bufio.NewScanner(file)
scanner.Scan() // slurp first (only) line
line := scanner.Text()

var ranges []Range
for _, block := range strings.Split(line, ",") {
    tok := strings.Split(block, "-")
    from, _ := strconv.Atoi(tok[0]) // can't fail ;-)
    to, _ := strconv.Atoi(tok[1])   // can't fail ;-)
    ranges = append(ranges, Range{
        Min: uint64(from),
        Max: uint64(to),
    })
}
```

i/o (alt)

- Use **`os`** to open file.
- Use **`bufio`** to get a Scanner with comma as separator.
- Get the different blocks with **`Scan`** .

```
scanner := bufio.NewScanner(file)
scanner.Split(func(data []byte, atEOF bool) (int, []byte, error) {
    switch i := bytes.IndexByte(data, ','); {
    case i >= 0:
        return i + 1, data[0:i], nil // found a comma!
    case atEOF && len(data) > 0:
        return len(data), data, nil // no commas, return last chunk of the file
    default:
        return 0, nil, nil // plz read more data and try again with a longer slice...
    }
})
```


Algorithm

- Generate all id in ranges, convert them to string, check validity.
- The RE2 engine does not support back references as in **`"^(.+)\1$"`** .

```
checksum := 0

for t := rng.Min; t <= rng.Max; t += 1 {
    id := strconv.Itoa(t)
    if len(id)%2 == 0 && id[:len(id)/2] == id[len(id)/2:] {
        checksum += t
    }
}
return checksum
```

Agenda

- Introduction.
- Basic syntax and primitive types.
- Functions.
- Composite types.
- Error handling.
- Concurrency.

Concurrency

- Concurrency vs. Parallelism.
- Squeak language.
- Goroutines.
- Channels.

Goroutine

- Multiplexed onto a small number of threads using an M:N scheduler (the G-P-M model)
  - **G**oroutine; Logical **P**rocessor; **M**achine (OS Thread)
  - Go runtime's context switches are way faster than OS's ones
- The runtime handles context switching (avoids the overhead of kernel-level context switches).
- Non-Blocking I/O.
- Dynamic stack (grows and shrinks in the heap as needed).
- 0.1–1M concurrent goroutines on the same machine…

Concurrency

- Based on Hoare's "Communicating Sequential Processes"
- **Do not communicate by sharing memory; share memory by communicating!**
- Channels are thread-safe FIFO queue with built-in blocking semantics. Under the hood:
  - A circular buffer.
  - 1 mutex.
  - 2 wait lists.

Concurrency

```go
func part1(ranges []Range) int {
    checksum := 0
    for _, r := range ranges {
        checksum += checkValidity(r)
    }
    return int(checksum)
}
```

```go
func part1_concurrent(ranges []Range) int {
    checksum := 0
    channel := make(chan int, 256)
    for _, r := range ranges {
        go func() {
            channel <- checkValidity(r)
        }()
    }
    for range ranges {
        checksum += <-channel
    }
    return int(checksum)
}
```


Concurrency

```go
func part1_concurrent(ranges []Range) int {
    checksum := 0
    channel := make(chan int, 256)
    for _, rng := range ranges {
        go func(r Range) {
            channel <- checkValidity(r)
        }(rng) // loop variable capture (ie. no closure on loop variable!)
    }
    for range ranges {
        checksum += <-channel
    }
    return int(checksum)
}
```

Non-blocking channels operations

- The **`select`** statement is the control structure that enables multiplexing across multiple channel operations.
  - Go's implementation of CSP's "alternative composition".
- The "receive & Check" pattern.

```
select {
case value := <-ch:
    log.Println("Got", value)
default:
    log.Println("D'ho!")
}
```


Select and channels

- The select statement is the control structure that enables multiplexing across multiple channel operations.
- Go's implementation of CSP "Alternative" composition.

Select and channels

- The sel…les multipl…
- Go's im…

```
output := make(chan string)

go func() {
    time.Sleep(time.Duration(time.Duration(rand.IntN(5)) * time.Second))
    output <- "Oh yeah!"
}()

select {
case result := <-output:
    fmt.Println(result)
case <-time.After(2 * time.Second):
    fmt.Println("Error: Operation timed out!")
}
```


Switch vs. Select

`switch`

- Value matching.
- Cases: Expressions or values.
- Deterministic execution order: first matching case.
- Never blocks.

`select`

- Multiplexing channel.
- Cases: Channel sends or receives.
- Uniform execution order: random matching case.
- Blocks unless **`default`** exists.

Non-blocking channels operations

- The **`select`** statement is the control structure that enables multiplexing across multiple channel operations.
  - Go's implementation of CSP's "alternative composition".
- The "receive & Check" pattern.

Non-blocking channels operations

- The **`select`** statement is the control structure that enables multiplexing across multiple channel operations.
  - Go's implementation of CSP's "alternative composition".
- The "receive & Check" pattern.

```go
select {
case value := <-ch:
    log.Println("Got", value)
default:
    log.Println("D'ho!")
}
```

Non-blocking channels operations

- The **`select`** statement is the control structure that enables multiplexing acrc ons.
  - Go's implementa osition".
- The "receive & C

```
select {
case value, ok := <-ch:
    if ok {
        log.Println("Got", value)
    } else {
        log.Println("Bad channel")
    }
default:
    log.Println("D'ho!")
}
```

```
    log.Println("D'ho!")
}
```


Problem of Shared Memory

- Beware of **race condition**.
- Operations like **+=** and **−=** are not atomic.
- Results will not be 0.
- Do not share variables!

```
for range NUM_GOROUTINES {
    go func() {
        for range NUM_OPERATION {
            sharedVar += 1
        }
    }()
    go func() {
        for range NUM_OPERATION {
            sharedVar -= 1
        }
    }()
}
```

Problem of Shared Memory

- Beware of ra
- Operations lik
  atomic.
- Results will n
- Do not share

```
var sharedVar int64

synch := make(chan int)
for range NUM_GOROUTINES {
    go func() {
        for range NUM_OPERATION {
            atomic.AddInt64(&sharedVar, 1)
        }
    }()
    go func() {
        for range NUM_OPERATION {
            atomic.AddInt64(&sharedVar, -1)
        }
    }()
}
```

```
M_GOROUTINES {
) {
range NUM_OPERATION {
sharedVar += 1

) {
range NUM_OPERATION {
sharedVar -= 1
```


WaitGroup

- Thread-safe counter.
- Caveat:
  - Don't add negative values.
  - Always add before start .
  - Call **Done** with **defer** .

```
var sharedVar int64
var wg sync.WaitGroup

for range NUM_GOROUTINES {
    wg.Add(1)
    go func() {
        defer wg.Done()
        for range NUM_OPERATION {
            atomic.AddInt64(&sharedVar, 1)
        }
    }()
    wg.Add(1)
    go func() {
        defer wg.Done()
        for range NUM_OPERATION {
            atomic.AddInt64(&sharedVar, -1)
        }
    }()
}
wg.Wait() // Block here until all workers call Done()

return sharedVar
```

Contexts

- Provides a powerful toolset for managing concurrent operations.
- A mechanism to control the lifecycle, cancellation, and propagation of requests across multiple goroutines.
- Notez Bien: **`Done`** is provided for use in select statements.
- Context with timeout.

Contexts

- Provide operati
- A mech propag
- Notez B ents.
- Contex

```go
ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
defer cancel()

if deadline, ok := ctx.Deadline(); ok {
    log.Printf("Started context with timeout @ %v\n", deadline)
}

output := make(chan Result, 4096)
randomInt := 1 + rand.IntN(1_000_000_000)
go collaz(ctx, randomInt, output)

select {
case <-ctx.Done():
    log.Println("Timed out!")
case result := <-output:
    log.Printf("Received result: %v\n", result)
}
```