Go

Clear syntax and everyday collections.

01 Run a Go program

```
package main
import "fmt"
func main() {
  fmt.Println("Hello, Go!")
}
```

Run go run main.go. For modules: go mod init example.com/app. Later snippets show imports: functions/types go outside main, other statements inside.

02 Variables & types

```
import "fmt"
var count int // zero value: 0
price := 19.95
ok := true
name := "Maya"
const limit = 10
fmt.Println(count, price, ok, name, limit)
```

Variables start with their type's zero value. := declares locals; var also works at package scope. byte is uint8; rune is int32. Unused local variables and imports are errors.

03 Operators & conversions

```
import "fmt"
whole := 7 / 2          // 3
ratio := float64(7) / 2 // 3.5
remainder := 7 % 2      // 1
ok := whole > 0 && remainder == 1
fmt.Println(whole, ratio, remainder, ok)
```

Use + - * / %, == != < <= > >=, and && || !. Numeric conversions are explicit. Integer division truncates toward zero. ++ and -- are statements, not expressions.

04 Conditions & switch

```
import "fmt"
score := 82
result := "Try"
if score >= 60 { result = "Pass" }
switch result {
case "Pass": fmt.Println("Good job")
}
```

Conditions must be bool. switch stops after each case, without break. Keep opening braces on the if/switch line.

05 Loops & range

```
import "fmt"
for i := 0; i < 3; i++ { fmt.Println(i) }
n := 2
for n > 0 { n-- }
for i, name := range []string{"a", "b"} {
  fmt.Println(i, name)
}
```

for is the loop keyword. range yields index/value here; _ discards a result. break exits; continue skips ahead.

06 Strings

```
import "fmt"
import "strings"
name := strings.TrimSpace(" Maya ")
fmt.Println(len(name), name == "Maya")
fmt.Println(strings.Contains(name, "ay"))
fmt.Println(strings.Split("a,b", ","))
```

Strings are immutable byte sequences. len counts bytes; range over a string decodes runes (Unicode code points). Convert to []rune when code-point indexing is needed.

07 Slices & shared storage

```
import "fmt"
nums := []int{10, 20, 30}
nums = append(nums, 40)
part := nums[1:3]
part[0] = 99
fmt.Println(nums, len(nums), cap(nums))
fmt.Println(part)
```

Slices share an underlying array: changing part changes nums here. Slicing excludes the end index. append may allocate a new array, so use its return value. [3]int is a fixed array.

08 Maps

```
import "fmt"
ages := map[string]int{"Maya": 28}
ages["Leo"] = 31
age, ok := ages["Maya"]
delete(ages, "Leo")
fmt.Println(age, ok, len(ages))
```

A missing key returns the value type's zero value; ok reports presence. Iteration order is unspecified. Allocate a map before writing: a map literal or make(map[string]int) works.

09 Functions

```
import "fmt"
func add(a, b int) int {
  return a + b
}
fmt.Println(add(2, 3))
```

Define add at package scope; call it inside main. Types follow names. Functions can return multiple values, often (value, error). Arguments are passed by value.

Go

Data, errors, and basic concurrency.

10 Structs & methods

```
import "fmt"
type Counter struct { n int }
func (c *Counter) Add() { c.n++ }
counter := Counter{}
counter.Add()
fmt.Println(counter.n)
```

Declare types and methods at package scope. A pointer receiver can change the original struct. Capitalized names are exported from a package; lower-case names are not.

11 Interfaces

```
import "fmt"
type Named interface { Name() string }
type User struct{}
func (User) Name() string { return "Maya" }
var named Named = User{}
fmt.Println(named.Name())
```

A type implements an interface by providing its methods; no implements declaration is needed. Keep interfaces small and focused on behavior. Put these definitions at package scope.

12 Pointers

```
import "fmt"
value := 7
ptr := &value
*ptr = 9
var empty *int
fmt.Println(value, *ptr, empty == nil)
```

& takes an address; * accesses its value. Go has no pointer arithmetic. Dereferencing nil panics. Go manages memory lifetime, but resources such as files still need closing.

13 Errors & wrapping

```
import "errors"
import "fmt"
err := errors.New("not found")
wrapped := fmt.Errorf("load: %w", err)
if errors.Is(wrapped, err) {
  fmt.Println("Missing item")
}
```

Check err != nil before using a result. %w wraps an error; errors.Is follows the chain. Return errors to callers.

14 defer & cleanup

```
import "fmt"
func show() {
  defer fmt.Println("last")
  defer fmt.Println("second")
  fmt.Println("first")
}
show()
```

Define show at package scope; call it in main. Deferred calls run when the function returns, last-in first-out. Their arguments are evaluated when defer executes.

15 Read files

```
import "fmt"
import "os"
data, err := os.ReadFile("notes.txt")
if err != nil { fmt.Println(err); return }
fmt.Print(string(data))
```

ReadFile loads the whole file and closes it. For streams, use os.Open and defer Close after a successful open. Check write/close errors for output.

16 JSON & exported fields

```
import "encoding/json"
import "fmt"
var user struct { Name string }
err := json.Unmarshal(
  []byte(`{"Name":"Maya"}`), &user)
if err != nil { fmt.Println(err); return }
fmt.Println(user.Name)
```

Only exported fields are used. A tag like `json:"name"` renames a field. Unmarshal needs a destination pointer; Marshal encodes. Check errors.

17 Goroutines & waiting

```
import "fmt"
done := make(chan bool)
go func() {
  fmt.Println("Hello")
  done <- true
}()
<-done
```

go starts a concurrent call. Wait before main returns; use channels or locks to synchronize shared data.

18 Channels & closing

```
import "fmt"
ch := make(chan int, 1)
ch <- 42
close(ch)
for value := range ch { fmt.Println(value) }
```

A buffer holds a limited number of values; an unbuffered send waits for a receiver. The sender closes a channel. range drains it and then ends; sending after close panics.