# ****Data types****

// Boolean type

Bool // true or false

// String type

String // Immutable sequence of bytes (typically UTF-8)

// Signed integer types

int8 // 8-bit signed integer (-128 to 127)

int16 // 16-bit signed integer (-32768 to 32767)

int32 // 32-bit signed integer (-2147483648 to 2147483647)

int64 // 64-bit signed integer (-2^63 to 2^63-1)

int // Platform-dependent signed integer (32 or 64 bits)

// Unsigned integer types

uint8 // 8-bit unsigned integer (0 to 255)

uint16 // 16-bit unsigned integer (0 to 65535)

uint32 // 32-bit unsigned integer (0 to 4294967295)

uint64 // 64-bit unsigned integer (0 to 2^64-1)

uint // Platform-dependent unsigned integer

uintptr // Unsigned integer large enough to store pointer value

// Floating-point types

float32 // 32-bit IEEE-754 floating-point

float64 // 64-bit IEEE-754 floating-point (default for floating-point literals)

// Complex number types

complex64 // Complex number with float32 real and imaginary parts

complex128 // Complex number with float64 real and imaginary parts

// Special aliases

Byte // Alias for uint8 (raw data)

Rune // Alias for int32 (Unicode code point)

// Aggregate types

arr [n]T // Fixed-size array

slice []T // Dynamic-size slice

struct { } // Structure

// Reference types

map[T]T // Map of keys to values

\*T // Pointer

var fn func(T) T // Function that takes int and returns int

chan T // Channel for communicating T data type

interface{} // Empty interface (can hold any value)

any // Alias for interface{} (since Go 1.18)

# ****Operators****

## **Arithmetic Operators**

+ // Addition (numbers, strings, slices)

- // Subtraction (numbers)

\* // Multiplication (numbers) / dereference (pointers)

/ // Division (numbers)

% // Remainder (integers)

& // Bitwise AND / address-of (pointers)

| // Bitwise OR

^ // Bitwise XOR / unary negation (integers)

&^ // Bit clear (AND NOT)

<< // Left shift

>> // Right shift

## **Comparison Operators**

== // Equal

!= // Not equal

< // Less than

<= // Less or equal

> // Greater than

>= // Greater or equal

## **Logical Operators**

&& // Logical AND

|| // Logical OR

! // Logical NOT

## **Assignment Operators**

= // Simple assignment

:= // Short variable declaration

+= // Add and assign

-= // Subtract and assign

\*= // Multiply and assign

/= // Divide and assign

%= // Modulo and assign

&= // Bitwise AND and assign

|= // Bitwise OR and assign

^= // Bitwise XOR and assign

<<= // Left shift and assign

>>= // Right shift and assign

<- // Send/receive (channels)

## **Pointer Operators**

& // Address-of

\* // Dereference (also multiplication)

## **Bitwise Operators**

& // AND

| // OR

^ // XOR

&^ // AND NOT

<< // Shift left

>> // Shift right

## **Type Operators**

.(T) // Type assertion (interfaces)

... // Variadic unpack / array length

## **Memory Operators**

& // Address-of

\* // Pointer dereference

New // Memory allocation

Make // Slice/map/channel initialization

## **Basic Syntax Symbols**

, // Expression separator

; // Statement terminator

: // Label/type declaration

() // Grouping / function call

[] // Array/slice index/make

{} // Block/composite literal

// // Single-line comment

/\*…\*/ // Multi-line comment

[T any] // Type parameter

~int // Underlying type constraint

# Functions:

## **Basic Function**

func FuncName(param1 Type1, param2 Type2) ReturnType {} // Declaration

FuncName(arg1, arg2) // Call

## **Multiple Return Values**

func FuncName() (Type1, Type2) {} // Declaration

val1, val2 := FuncName() // Call

## **Named Return Values**

func FuncName() (name1 Type1, name2 Type2) {} // Declaration

val1, val2 := FuncName() // Call (same as multiple returns)

## **Variadic Function**

func FuncName(params ...Type) ReturnType {} // Declaration

FuncName(arg1, arg2, arg3) // Call // Any number of args

## **Method (Receiver Function)**

func (receiver ReceiverType) MethodName() ReturnType {} // Declaration

receiver.MethodName() // Call

## **Anonymous Function**

funcVar := func(params) ReturnType {} // Declaration

funcVar(args) // Call

## **Higher-Order Function**

func FuncName(fn func(Type1) Type2, param Type1) Type2 {} // Declaration

FuncName(otherFunc, arg) // Call

## **Closure**

func OuterFunc() func() ReturnType {} // Declaration

closure := OuterFunc() // Call

closure()

## **Generic Function**

func FuncName[T Constraint](param T) ReturnType {} // Declaration

FuncName[Type](arg) // Call // Type inference often possible

## **Deferred Call**

defer FuncName(args) // Called when surrounding function exits

## **Interface Implementation**

type InterfaceName interface { MethodName() ReturnType} // Declaration

var i InterfaceName = MyType{} // Call (via interface variable)

i.MethodName()

# Slices

## **Creation & Initialization**

// Empty slice

var s []T // nil slice (len=0, cap=0)

s := make([]T, len) // With length (zero-valued elements)

s := make([]T, len, cap) // With length and capacity

// Literal initialization

s := []T{v1, v2, v3} // Slice with values

s := []T{} // Empty initialized slice (len=0, cap=0)

// From array

arr := [3]T{v1, v2, v3}

s := arr[:] // Slice referencing entire array

## **Basic Operations**

// Length and capacity

len(s) // Current length

cap(s) // Current capacity

// Access elements

s[0] // First element

s[len(s)-1] // Last element

// Sub-slicing (creates new slice header)

s[1:4] // Elements from index 1 to 3

s[:3] // First 3 elements

s[3:] // From index 3 to end

s[:] // Copy of entire slice (shallow)

## **Modification**

// Append (may trigger reallocation)

s = append(s, v) // Append single value

s = append(s, v1, v2) // Append multiple values

s = append(s, s2...) // Append another slice

// Copy (returns number of elements copied)

n := copy(dest, src) // Copy elements between slices

// Insert at index

s = append(s[:i], append([]T{v}, s[i:]...)...)

// Delete element at index

s = append(s[:i], s[i+1:]...)

// Delete range [i,j)

s = append(s[:i], s[j:]...)

## **Memory Efficiency**

// Preallocate when size is known

s := make([]T, 0, expectedSize)

// Trim capacity to match length

s = s[:len(s):len(s)] // cap(s) == len(s)

// Clear slice (keep memory)

s = s[:0] // len=0, cap unchanged

// Release memory (GC can collect)

s = nil // len=0, cap=0

// Force new allocation (if reducing size significantly)

s = append([]T(nil), s...)

## **Conversion Tricks**

// String ↔ []byte

str := string(bytes) // Copies bytes

bytes := []byte(str) // Copies bytes

// String ↔ []rune

runes := []rune(str) // Copies runes

str := string(runes) // Copies runes

// []T1 ↔ []T2 (unsafe conversion)

import "unsafe"

t2Slice := \*(\*[]T2)(unsafe.Pointer(&t1Slice)) // Use with caution!

# ****Conditionals in Go****

## Basic if

if condition {}

## if with Initialization

if val := compute(); val > 0 {}

## if-else

if condition {} else {}

## if-else if Chain

if cond1 {} else if cond2 {} else {}

## Switch (Value)

switch variable {

case val1: // ...

case val2: // ...

default: // ... }

## Switch (Expression)

switch {

case score > 90: // ...

case score > 50: // ... }

## Switch with Initialization

switch res := compute(); res {

case 1: // ... }

## Type Switch

switch v := x.(type) {

case int: // v is int

case string: // v is string

default: } // v is original type

# ****Special Conditional Cases****

## Empty switch

switch {} // Equivalent to switch true {}

## Fallthrough

switch v {

case 1:

fmt.Println("1")

fallthrough // Continues to next case

case 2:

fmt.Println("2")

}

# ****Loops in Go****

## Basic for Loop

for i := 0; i < 10; i++ {} // Declaration

## While-style Loop

for condition {} // Works like while(condition)

## Infinite Loop

for {} // Runs forever (break to exit)

## Range Loop

for index, value := range collection {} // Arrays/slices/maps/strings

for key, value := range map {} // Maps

for ch := range channel {} // Channels (until closed)

## Range Loop (Value Only)

for \_, value := range collection {} // Ignore index/key

## Range Loop (Index Only)

for index := range collection {} // Keys for maps, indices for slices

## Loop Control

for {

if condition {

break }} // Exit loop

for i := 0; i < 10; i++ {

if i%2 == 0 { continue }} // Skip iteration

// Labeled Break

OuterLoop:

for i := 0; i < 5; i++ {

for j := 0; j < 5; j++ {

if condition {

break OuterLoop }} // Breaks outer loop

## **Common Patterns**

// Slice Iteration

for i := 0; i < len(s); i++ { // Standard for loop

// s[i] }

for i := range s { // Range loop (index only)

// i, s[i] }

for i, v := range s { // Range loop (index + value)

// i, v }

for \_, v := range s { // Range loop (value only)

// v }

// Multi-dimensional Slices

matrix := make([][]T, rows) // 2D slice

for i := range matrix {

matrix[i] = make([]T, cols) }

jagged := make([][]T, size) // Jagged slice

for i := range jagged {

jagged[i] = make([]T, i+1) } // Varying lengths

var flat []T // Flatten 2D slice

for \_, row := range matrix {

flat = append(flat, row...) }

n := 0 // Filter elements (in-place)

for \_, x := range s {

if keep(x) {

s[n] = x

n++ } }

s = s[:n]

for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 { // Reverse slice

s[i], s[j] = s[j], s[i] }

if len(s) > 0 { // Remove duplicates (sorted slice)

j := 0

for i := 1; i < len(s); i++ {

if s[j] != s[i] {

j++

s[j] = s[i] } }

s = s[:j+1] }

batchSize := 10 // Batch processing

for i := 0; i < len(s); i += batchSize {

end := i + batchSize

if end > len(s) {

end = len(s) }

batch := s[i:end]

// Process batch }

# ****Type****

type MyInt int // New type from existing type

type MyAlias = int // Alias (compile-time equivalent)

type MathFunc func(int, int) int // Function type

type Reader interface { // Interface type

Read(p []byte) (n int, err error) }

type Signal chan struct{} // Channel type

# struct

## **Struct Declarations**

type Person struct { // Basic struct

Name string

Age int }

type Employee struct { // Embedded struct (composition)

Person // Embedded field

ID int

Salary float64 }

type User struct { // Struct with tags

Name string `json:"name" xml:"name"`

Email string `json:"email,omitempty"`

private bool } // Unexported field

point := struct { // Anonymous struct

X, Y int

}{10, 20}

## **3. Method Declarations**

func (p Person) Greet() string { // Value receiver (operates on copy)

return fmt.Sprintf("Hello, %s", p.Name) }

func (p \*Person) Birthday() { // Pointer receiver (modifies original)

p.Age++ }

func (m MyInt) Double() MyInt { // Method on custom type

return m \* 2 }

func (p Person) String() string { // Interface implementation

return fmt.Sprintf("%s (%d)", p.Name, p.Age) }

## **4. Constructor Pattern**

// Conventional constructor

func NewPerson(name string, age int) \*Person {

return &Person{

Name: name,

Age: age, } }

## **5. Type Constraints (Generics)**

// Generic type

type Stack[T any] struct {

items []T }

// Generic method

func (s \*Stack[T]) Push(item T) {

s.items = append(s.items, item) }

# ****Channels and Select****

## **Channels Syntax**

// Creation

ch := make(chan T) // Unbuffered channel

ch := make(chan T, buf) // Buffered channel (size=buf)

// Operations

ch <- v // Send (blocks if unbuffered/full)

v := <-ch // Receive (blocks if empty)

close(ch) // Close channel

v, ok := <-ch // ok=false if closed

// Directional channels

func f(ch chan<- T) {} // Send-only

func f(ch <-chan T) {} // Receive-only

// Range over channel

for v := range ch {

// Loops until ch is closed }

## **Select Statement Syntax**

select {

case x := <-ch1: // Basic receive

// Handle ch1 value

case ch2 <- y: // Send operation

// Send succeeded

case v, ok := <-ch3: // Non-blocking check

if !ok { /\* channel closed \*/ }

case <-time.After(1\*time.Second): // Timeout

// Handle timeout

default: // Default fallthrough

// Non-blocking if no cases ready }

## **Key Patterns**

// Basic synchronization (unbuffered channel)

ch := make(chan struct{})

go func() { ch <- struct{}{} }()

<-ch // Wait for signal

// Worker pool (buffered channels)

jobs := make(chan Job, 10)

results := make(chan Result, 10)

// Start workers

for w := 1; w <= 3; w++ {

go worker(jobs, results) }

// Multiplexing (select with multiple channels)

select {

case msg1 := <-ch1:

fmt.Println(msg1)

case msg2 := <-ch2:

fmt.Println(msg2) }

// Graceful shutdown

done := make(chan struct{})

go func() {

defer close(done)

// Work here }()

select {

case <-done:

fmt.Println("Done")

case <-time.After(2\*time.Second):

fmt.Println("Timeout")

}

// Non-blocking check

select {

case v := <-ch:

fmt.Println(v)

default:

// Skip if no data }

# Builtin Functions

## **Rune/Byte Conversions**

rune(x) // Converts to Unicode code point (int32 alias)

byte(x) // Converts to 8-bit unsigned (uint8 alias)

## **String Conversions**

string(rune) // Converts rune to UTF-8 string (1-4 bytes)

string([]rune) // Converts rune slice to UTF-8 string

string(byte) // Converts byte to 1-character string

string([]byte) // Converts byte slice to string

## **Numeric Type Conversions**

uint(x) // Unsigned int (platform-dependent size)

uintptr(x) // Integer large enough to store pointer bits

## **Special Numeric Cases**

float32(x) // 32-bit floating point

float64(x) // 64-bit floating point (default for literals)

complex64(x) // 64-bit complex (float32 components)

complex128(x) // 128-bit complex (float64 components)

## **Bool Conversion**

bool(x) // Converts to true/false (only from numeric/string in some cases)

## **append() - All Variants**

slice = append(slice, element) // Single element

slice = append(slice, el1, el2, el3) // Multiple elements

slice = append(slice, anotherSlice...) // Append another slice (must unpack)

slice = append(append(slice, el1), el2) // Chain appends

## **make() - All Types**

make([]T, length) // Slice (with optional capacity)

make([]T, length, capacity)

make(map[K]V) // Map (with optional size hint)

make(map[K]V, initialSize)

make(chan T) // Channel (with optional buffer)

make(chan T, bufferSize)

## **copy() - Variants**

copied := copy(dstSlice, srcSlice) // Full copy (if dst has enough capacity)

copied := copy(dstSlice[:n], srcSlice) // Partial copy (limited by dst length)

## **Type Conversion - Common Cases**

int(float64) // Numeric conversions

float64(int)

string([]byte) // String/byte conversions

[]byte(string)

value.(ConcreteType) // Interface assertions

value.(interface{Method()})

## **delete() - Map Operations**

delete(myMap, key) // Single key deletion

if \_, exists := myMap[key]; exists { // Safe deletion (check exists first)

delete(myMap, key)}

## **Channel Operations**

close(myChan) // Basic close

if myChan != nil { close(myChan)} // Safe close (avoid panic)

## **len()/cap() - All Supported Types**

len("string") // String length

len([3]int{}) // Array length

len([]int{1,2}) // Slice length

len(map[string]int) // Map item count

len(chan int) // Channel buffer items

cap([3]int{}) // Array capacity

cap([]int{1,2}) // Slice capacity

cap(chan int) // Channel buffer size

## **Complex Number Operations**

complex(1.0, 2.0) // float64 → complex128

// Creation variants

complex(float32(1), float32(2)) // float32 → complex64

// Extraction variants

real(complex128) // → float64

imag(complex64) // → float32

## **Special Cases Cheat Box**

var s []int // Append to nil slice

s = append(s, 1) // Works (creates slice)

var dst []int // Copy to nil slice (does nothing)

copied := copy(dst, src) // copied == 0

var i interface{} = "hello" // Type conversion panics

num := i.(int) // PANICS (use type assertion)

## **Memory Allocation**

new(T) // Allocates zeroed value, returns \*T

## **Panic/Recovery**

panic(x) // Triggers runtime panic (any type)

recover() // Captures panic (only in defer)

## **Debug Prints (Avoid in production)**

print(args...) // Low-level print (no formatting)

println(args...) // Print with spaces + newline

## **Unsafe Operations (Special cases)**

unsafe.Sizeof(x) // Memory size in bytes

unsafe.Alignof(x) // Memory alignment

unsafe.Offsetof(s.f) // Field offset in struct

## **Pointer Arithmetic (Unsafe)**

unsafe.Add(ptr, len) // Pointer addition

unsafe.Slice(ptr, len) // Creates slice from pointer

## **Type Identity**

unsafe.Pointer(x) // Converts to raw pointer (bypasses type system)

# ****fmt Package Functions****

## **Special Format Verbs**

%v // Default format (any type)

%+v // Add field names (structs)

%#v // Go-syntax representation

%T // Type name

%% // Percent sign

// Type-Specific Verbs

%d // Integer (base 10)

%x // Hexadecimal

%f // Floating-point

%t // Boolean

%s // String

%p // Pointer address

%c // Unicode character

%U // Unicode code point

// Formatting Options

%5d // Pad with spaces (width 5)

%-5d // Left-justified

%05d // Pad with zeros

%5.2f // Width 5, precision 2

%.2f // Precision 2 (auto width)

## Basic Printing

fmt.Print(args...) // Prints raw values (no auto-spacing)

fmt.Printf(format, args...) // Formatted print (uses % verbs)

fmt.Println(args...) // Prints values + newline (auto-spaced)

## String Generators (Return Strings)

fmt.Sprint(args...) string // Concatenates to string (no formatting)

fmt.Sprintf(format, args...) string // Returns formatted string (uses % verbs)

fmt.Sprintln(args...) string // Returns string with spaces + newline

## Error Constructor

fmt.Errorf(format, args...) error // Creates error (supports %w wrapping)

## Input Scanners

fmt.Scan(&vars...) // Reads whitespace-separated input

fmt.Scanf(format, &vars...) // Formatted input scan (uses % verbs)

fmt.Scanln(&vars...) // Reads line (whitespace-separated)

## String Scanners

fmt.Sscan(str, &vars...) // Scans whitespace-separated from string

fmt.Sscanln(str, &vars...) // Scans line from string

fmt.Sscanf(str, format, &vars...) // Formatted string scan (uses % verbs)

## **Required Interfaces**

// String Representation

type Stringer interface {

String() string // Implement to customize print behavior

}

// Custom Formatting

type Formatter interface {

Format(f State, verb rune) // Implement for advanced % verb support

}

// Format State

type State interface {

Write(b []byte) (n int, err error) // Required for Formatter

Width() (wid int, ok bool) // Reads width specifier (e.g., %10s)

Precision() (prec int, ok bool) // Reads precision (e.g., %.2f)

Flag(c int) bool } // Checks formatting flags

# ****Strings****

## **Basic String Operations**

strings.Compare(a, b string) int // Returns -1/0/1 (lex order)

strings.Contains(s, substr string) bool // Checks substring existence

strings.Count(s, substr string) int // Counts non-overlapping instances

## **Case Conversion**

strings.ToLower(s string) string // Converts to lowercase

strings.ToUpper(s string) string // Converts to uppercase

strings.ToTitle(s string) string // Converts to title case

## **Trimming & Padding**

strings.Trim(s, cutset string) string // Removes edge chars

strings.TrimSpace(s string) string // Removes whitespace

strings.TrimPrefix(s, prefix string) string // Removes exact prefix

strings.TrimSuffix(s, suffix string) string // Removes exact suffix

strings.Repeat(s string, count int) string // Repeats string

## **Splitting & Joining**

strings.Split(s, sep string) []string // Splits at separator

strings.SplitN(s, sep string, n int) []string // Splits n times

strings.Join(elems []string, sep string) string // Joins with separator

## **Replacement**

strings.Replace(s, old, new string, n int) string // Replaces n instances

strings.ReplaceAll(s, old, new string) string // Replaces all instances

## **Index Searching**

strings.Index(s, substr string) int // Finds first substring index

strings.LastIndex(s, substr string) int // Finds last substring index

strings.IndexAny(s, chars string) int // Finds any char's first index

## **Predicate Checks**

strings.HasPrefix(s, prefix string) bool // Checks string start

strings.HasSuffix(s, suffix string) bool // Checks string end

strings.EqualFold(s, t string) bool // Case-insensitive compare

## **Specialized Functions**

strings.Map(mapping func(rune) rune, s string) string // Char-by-char mapping

strings.Fields(s string) []string // Splits at whitespace

strings.FieldsFunc(s string, f func(rune) bool) []string // Custom split

## **Utility Functions**

strings.Clone(s string) string // Returns copy of string

strings.Cut(s, sep string) (before, after string, found bool) // Splits at first sep

## **strings.Reader Type**

type Reader struct {} // Contains unexported fields

func NewReader(s string) \*Reader // Creates reader from string

// Methods

func (r \*Reader) Len() int // Returns unread bytes

func (r \*Reader) Read(b []byte) (n int, err error) // Implements io.Reader

func (r \*Reader) ReadAt(b []byte,off int64)(n int,err error) // Implements io.ReaderAt

func (r \*Reader) ReadByte() (byte, error) // Reads single byte

func (r \*Reader) ReadRune() (ch rune, size int, err error) // Reads UTF-8 rune

func (r \*Reader) Reset(s string) // Resets reader with new string

func (r \*Reader) Seek(offset int64, whence int) (int64, error) // Implements io.Seeker

func (r \*Reader) Size() int64 // Returns original string length

func (r \*Reader) UnreadByte() error // Puts back last byte

func (r \*Reader) UnreadRune() error // Puts back last rune

func (r \*Reader) WriteTo(w io.Writer) (n int64, err error) // Implements io.WriterTo

## **strings.Builder Type**

type Builder struct {} // Contains unexported fields

// Methods

func (b \*Builder) Cap() int // Returns capacity

func (b \*Builder) Grow(n int) // Pre-allocates space

func (b \*Builder) Len() int // Returns length

func (b \*Builder) Reset() // Clears contents

func (b \*Builder) String() string // Returns built string

func (b \*Builder) Write(p []byte) (int, error) // Implements io.Writer

func (b \*Builder) WriteByte(c byte) error // Appends single byte

func (b \*Builder) WriteRune(r rune) (int, error) // Appends UTF-8 rune

func (b \*Builder) WriteString(s string) (int, error) // Appends string

# strconv

## **String ↔ Number Conversions**

// Integer conversions

strconv.Atoi(s) (int, error) // Base-10 only

strconv.ParseInt(s, base, bits) // Custom base (2-36), bits (8/16/32/64)

strconv.ParseUint(s, base, bits) // Unsigned version

strconv.Itoa(i) string // Base-10

strconv.FormatInt(i, base) string // Custom base

strconv.FormatUint(i, base) string // Unsigned version

// Float conversions

strconv.ParseFloat(s, bits) // bits=32/64

strconv.FormatFloat(f, fmt, prec, bits) // fmt='b','e','E','f','g','G'

// Bool conversions

strconv.ParseBool(s) // Accepts "1/t/true" or "0/f/false"

strconv.FormatBool(b) // Returns "true"/"false"

## **Quoting Utilities**

// String quoting

strconv.Quote(s) // Adds "" and escapes

strconv.QuoteToASCII(s) // Escapes non-ASCII

strconv.QuoteToGraphic(s) // Keeps printable Unicode

// Rune quoting

strconv.QuoteRune(r) // Single rune version

strconv.QuoteRuneToASCII(r) // ASCII-only rune

strconv.QuoteRuneToGraphic(r) // Graphic rune

// Unquoting

strconv.Unquote(s) // Removes "" or ``

strconv.UnquoteChar(s) // Low-level unescape

## **Append Variants**

// Append to byte slices (avoid allocations)

strconv.AppendInt(dst, i, base) // Like FormatInt but appends

strconv.AppendUint(dst, i, base) // Unsigned append

strconv.AppendFloat(dst, f, fmt, prec, bits)

strconv.AppendBool(dst, b)

strconv.AppendQuote(dst, s)

strconv.AppendQuoteRune(dst, r)

## **Validation Helpers**

strconv.IsPrint(r) // Printable (incl. space)

strconv.IsGraphic(r) // Visible characters only

strconv.CanBackquote(s) // Safe for raw strings (`s`)

# os

## **File Operations**

// Basic file operations

os.Create(name) (\*File, error) // Create or truncate file

os.Open(name) (\*File, error) // Open read-only

os.OpenFile(name,flag,perm)(\*File,error) // Full control (flag=O\_RDONLY/etc, perm=0644)

// File info

os.Stat(name) (FileInfo, error) // Get file info

os.IsExist(err) bool // Check if error is "already exists"

os.IsNotExist(err) bool // Check if error is "doesn't exist"

// File manipulation

os.Rename(old, new) error // Move/rename

os.Remove(path) error // Delete file

os.RemoveAll(path) error // Recursive delete

os.Truncate(name, size) error // Resize file

## **Directory Operations**

// Directory handling

os.Mkdir(name, perm) error // Create single dir

os.MkdirAll(path, perm) error // Create all dirs in path

os.ReadDir(name) ([]DirEntry, error) // Read directory entries (modern)

os.ReadFile(name) ([]byte, error) // Read entire file

os.WriteFile(name, data, perm) error // Write entire file

## **Process & Environment**

// Process control

os.Exit(code int) // Exit immediately

os.Getpid() int // Get current process ID

os.Getppid() int // Get parent process ID

// Environment variables

os.Getenv(key) string // Get env var

os.Setenv(key, value) error // Set env var

os.Unsetenv(key) error // Unset env var

os.Environ() []string // Get all env vars

os.ExpandEnv(s) string // Replace $VAR in string

os.Clearenv() // Clear all env vars

## **File Info & Permissions**

// FileInfo methods (from Stat())

fi.Name() string // Base filename

fi.Size() int64 // File size

fi.Mode() FileMode // Permission bits

fi.ModTime() time.Time // Modification time

fi.IsDir() bool // Is directory?

fi.Sys() interface{} // Underlying data source

// FileMode constants

os.ModeDir // Directory

os.ModePerm // Unix permissions (0777 etc)

os.ModeNamedPipe // Named pipe

os.ModeSocket // Unix domain socket

## **Standard Streams**

os.Stdin \*File // Standard input

os.Stdout \*File // Standard output

os.Stderr \*File // Standard error

## **User & Host Info**

os.Hostname() (string, error) // System hostname

os.UserCacheDir() (string, error) // User cache dir

os.UserConfigDir() (string, error) // User config dir

os.UserHomeDir() (string, error) // User home dir

os.Getwd() (string, error) // Current working dir

os.Chdir(dir) error // Change working dir

## **File Descriptors**

os.NewFile(fd, name) \*File // Create File from fd

os.Pipe() (r \*File, w \*File, err error) // Create pipe

## **Symlinks & Special Files**

os.Readlink(name) (string, error) // Read symlink target

os.Symlink(old, new) error // Create symlink

os.Lstat(name) (FileInfo, error) // Like Stat but for symlinks

os.Chmod(name, mode) error // Change permissions

os.Chown(name, uid, gid) error // Change owner (Unix)

os.Lchown(name, uid, gid) error // Change symlink owner

## **Temporary Files**

os.CreateTemp(dir, pattern) (\*File, error) // Create temp file

os.MkdirTemp(dir, pattern) (string, error) // Create temp dir

## **Process Signals**

os.FindProcess(pid) (\*Process, error) // Get process handle

p.Signal(sig os.Signal) error // Send signal

p.Wait() (\*ProcessState, error) // Wait for process

p.Kill() error // Kill process

# errors

## **Basic Error Creation**

errors.New(text) error // Create simple error with text

fmt.Errorf(format, args...) error // Formatted error (often used instead)

## **Error Inspection**

errors.Is(err, target) bool // Check if error matches target

errors.As(err, target) bool // Extract error to target type

errors.Unwrap(err) error // Get next error in chain

## **Error Wrapping (Go 1.13+)**

// Wrap errors with context (fmt.Errorf with %w)

wrapped := fmt.Errorf("context: %w", originalErr)

// Example checking wrapped errors:

var ErrNotFound = errors.New("not found")

if errors.Is(err, ErrNotFound) {

// Handle specific error

}

## **Custom Error Types**

type MyError struct {

Code int

Message string

}

func (e \*MyError) Error() string {

return fmt.Sprintf("%d: %s", e.Code, e.Message)

}

// Usage with errors.As:

var me \*MyError

if errors.As(err, &me) {

// Access me.Code and me.Message

}

## **Sentinel Errors**

// Define at package level

var (

ErrInvalidInput = errors.New("invalid input")

ErrTimeout = errors.New("operation timed out")

)

// Compare with errors.Is

if errors.Is(err, ErrInvalidInput) {

// Handle specific error case

}

## **Error Chaining**

err1 := errors.New("error 1")

err2 := fmt.Errorf("error 2: %w", err1)

err3 := fmt.Errorf("error 3: %w", err2)

// Unwrap chain:

// err3 → err2 → err1 → nil

## **Best Practices**

* Use errors.New for simple static errors
* Use fmt.Errorf with %w to wrap errors
* Export sentinel errors as variables (ErrXxx)
* Use errors.Is instead of == for comparisons
* Use errors.As for type assertions on errors

## **Common Patterns**

// Checking error types:

var e \*MyError

if errors.As(err, &e) {

// Handle MyError case }

// Adding context:

if err := doSomething(); err != nil {

return fmt.Errorf("doSomething failed: %w", err) }

// Creating rich errors:

type APIError struct {

StatusCode int

Err error }

func (a \*APIError) Error() string {

return fmt.Sprintf("status %d: %v", a.StatusCode, a.Err) }

### ****time****

## **Time Construction**

time.Now() time.Time // Current time

// Create specific time

time.Date(year, month, day, hour, min, sec, nsec, loc) time.Time

time.Unix(sec, nsec) time.Time // From Unix timestamp

time.UnixMilli(ms) time.Time // From milliseconds

time.UnixMicro(μs) time.Time // From microseconds

// Zero time

time.Time{} // Zero value: 0001-01-01 00:00:00

## **Time Parsing & Formatting**

// Parse from string (layout must match input)

time.Parse(layout, value) (time.Time, error)

time.ParseInLocation(layout, value, loc) (time.Time, error)

// Format to string

t.Format(layout) string

// Standard layouts (predefined constants):

time.RFC3339 // "2006-01-02T15:04:05Z07:00"

time.RFC822 // "02 Jan 06 15:04 MST"

time.Kitchen // "3:04PM"

time.Stamp // "Jan \_2 15:04:05"

time.DateOnly // "2006-01-02" (Go 1.20+)

time.TimeOnly // "15:04:05" (Go 1.20+)

## **Time Components**

t.Year() int

t.Month() time.Month // January-December

t.Day() int

t.Hour() int

t.Minute() int

t.Second() int

t.Nanosecond() int

t.YearDay() int // Day of year (1-366)

t.Weekday() time.Weekday // Sunday-Saturday

t.ISOWeek() (year, week int) // ISO 8601 week

t.Location() \*time.Location // Timezone info

t.Zone() (name string, offset int) // Timezone name and offset

## **Time Manipulation**

// Duration arithmetic

t.Add(d time.Duration) time.Time

t.AddDate(years, months, days int) time.Time

t.Sub(u time.Time) time.Duration // Time between two points

t.UTC() time.Time // Convert to UTC

t.Local() time.Time // Convert to local time

t.Round(d time.Duration) time.Time // Round to nearest duration

t.Truncate(d time.Duration) time.Time // Truncate to duration

## **Durations**

// Construction

time.Duration(n) \* time.Second // 5\*time.Second

time.ParseDuration(s) (time.Duration, error) // "300ms", "2h45m"

// Constants

time.Nanosecond

time.Microsecond

time.Millisecond

time.Second

time.Minute

time.Hour

// Methods

d.Seconds() float64

d.Milliseconds() int64

d.Microseconds() int64

d.Nanoseconds() int64

d.Round(m time.Duration) time.Duration

d.Truncate(m time.Duration) time.Duration

## **Time Comparison**

t.Before(u time.Time) bool

t.After(u time.Time) bool

t.Equal(u time.Time) bool // Exact equality

t.Compare(u time.Time) int // -1/0/+1 (Go 1.20+)

t.IsZero() bool // Checks for zero time

## **Timezone Handling**

time.LoadLocation(name) (\*time.Location, error) // "UTC", "Local", "America/New\_York"

time.FixedZone(name, offset) \*time.Location // Custom offset

time.UTC \*time.Location // UTC timezone

time.Local \*time.Location // System local timezone

## **Tickers & Timers**

// Execute after duration

time.AfterFunc(d, func()) \*time.Timer

time.After(d) <-chan time.Time // Channel version

// Repeating timer

time.NewTicker(d) \*time.Ticker

ticker.Stop() // Stop the ticker

// One-shot timer

time.NewTimer(d) \*time.Timer

timer.Reset(d) // Restart timer

## **Sleep & Delay**

time.Sleep(d time.Duration) // Block for duration

## **Useful Patterns**

// Measure execution time

start := time.Now()

// ... operation ...

elapsed := time.Since(start) // Or: time.Now().Sub(start)

// Timeout pattern

select {

case res := <-ch:

// use res

case <-time.After(1 \* time.Second):

// timeout

}

// Periodic execution

ticker := time.NewTicker(1 \* time.Minute)

defer ticker.Stop()

for t := range ticker.C {

// runs every minute

}

## **Constants**

time.January ... time.December // Months

time.Sunday ... time.Saturday // Weekdays

time.Layout = "01/02 03:04:05PM '06 -0700" // Reference time format

# ****math****

## **Basic Constants**

math.Pi // 3.141592653589793

math.E // 2.718281828459045

math.Phi // 1.618033988749895 (golden ratio)

math.Sqrt2 // 1.4142135623730951

math.SqrtE // 1.6487212707001282

math.SqrtPi // 1.772453850905516

math.SqrtPhi // 1.272019649514069

math.Ln2 // 0.6931471805599453

math.Ln10 // 2.302585092994046

## **Numeric Limits**

// Float limits

math.MaxFloat32 // 3.4028234663852886e+38

math.SmallestNonzeroFloat32 // 1.401298464324817e-45

math.MaxFloat64 // 1.7976931348623157e+308

math.SmallestNonzeroFloat64 // 4.940656458412465e-324

// Integer limits (via math/bits)

bits.UintSize // 32 or 64

math.MaxInt // Max int value (32 or 64-bit)

math.MinInt // Min int value

math.MaxInt8 // 127

math.MinInt8 // -128

math.MaxUint8 // 255

math.MaxInt16 // 32767

math.MinInt16 // -32768

math.MaxUint16 // 65535

math.MaxInt32 // 2147483647

math.MinInt32 // -2147483648

math.MaxUint32 // 4294967295

math.MaxInt64 // 9223372036854775807

math.MinInt64 // -9223372036854775808

math.MaxUint64 // 18446744073709551615

## **Basic Arithmetic**

math.Abs(x float64) float64 // Absolute value

math.Ceil(x float64) float64 // Round up

math.Floor(x float64) float64 // Round down

math.Round(x float64) float64 // Round to nearest

math.RoundToEven(x float64) float64 // Round to nearest even

math.Trunc(x float64) float64 // Truncate decimal

math.Mod(x, y float64) float64 // Floating-point remainder

math.Remainder(x, y float64) float64 // IEEE remainder

math.Max(x, y float64) float64 // Maximum of two numbers

math.Min(x, y float64) float64 // Minimum of two numbers

math.Dim(x, y float64) float64 // x-y if x>y, else 0

## **Exponents and Roots**

math.Sqrt(x float64) float64 // Square root

math.Cbrt(x float64) float64 // Cube root

math.Pow(x, y float64) float64 // x raised to y

math.Pow10(n int) float64 // 10 raised to n

math.Exp(x float64) float64 // e^x

math.Exp2(x float64) float64 // 2^x

math.Expm1(x float64) float64 // e^x - 1 (more accurate for small x)

math.Log(x float64) float64 // Natural log (ln)

math.Log1p(x float64) float64 // ln(1+x) (accurate near 0)

math.Log2(x float64) float64 // Base-2 log

math.Log10(x float64) float64 // Base-10 log

math.Hypot(x, y float64) float64 // sqrt(x² + y²)

## **Trigonometry**

math.Sin(x float64) float64 // Sine (radians)

math.Cos(x float64) float64 // Cosine (radians)

math.Tan(x float64) float64 // Tangent (radians)

math.Asin(x float64) float64 // Arc sine

math.Acos(x float64) float64 // Arc cosine

math.Atan(x float64) float64 // Arc tangent

math.Atan2(y, x float64) float64 // Arc tangent of y/x

math.Sincos(x float64) (sin, cos float64) // Both sin and cos

## **Hyperbolic Functions**

math.Sinh(x float64) float64 // Hyperbolic sine

math.Cosh(x float64) float64 // Hyperbolic cosine

math.Tanh(x float64) float64 // Hyperbolic tangent

math.Asinh(x float64) float64 // Inverse hyperbolic sine

math.Acosh(x float64) float64 // Inverse hyperbolic cosine

math.Atanh(x float64) float64 // Inverse hyperbolic tangent

## **Special Functions**

math.Erf(x float64) float64 // Error function

math.Erfc(x float64) float64 // Complementary error function

math.Gamma(x float64) float64 // Gamma function

math.Lgamma(x float64) (float64, int) // Natural log of Gamma

math.J0(x float64) float64 // Bessel function of 1st kind, order 0

math.J1(x float64) float64 // Bessel function of 1st kind, order 1

math.Jn(n int, x float64) float64 // Bessel function of 1st kind, order n

math.Y0(x float64) float64 // Bessel function of 2nd kind, order 0

math.Y1(x float64) float64 // Bessel function of 2nd kind, order 1

math.Yn(n int, x float64) float64 // Bessel function of 2nd kind, order n

## **Floating-point Manipulation**

math.Frexp(f float64) (frac float64, exp int) // Split into mantissa and exponent

math.Ldexp(frac float64, exp int) float64 // Combine mantissa and exponent

math.Modf(f float64) (intpart, fracpart float64) // Split integer and fraction

math.Nextafter(x, y float64) float64 // Next representable float after x toward y

math.Nextafter32(x, y float32) float32 // 32-bit version

math.Ilogb(x float64) int // Binary exponent

math.Logb(x float64) float64 // Binary exponent as float

math.Copysign(x, y float64) float64 // Copy sign of y to x

math.Signbit(x float64) bool // Check if negative

math.IsNaN(f float64) bool // Check if NaN

math.IsInf(f float64, sign int) bool // Check if infinite

math.NaN() float64 // Returns NaN

math.Inf(sign int) float64 // Returns ±Inf

## **Bit Manipulation (via math/bits)**

bits.OnesCount(n uint) int // Count set bits

bits.LeadingZeros(n uint) int // Count leading zeros

bits.TrailingZeros(n uint) int // Count trailing zeros

bits.Len(n uint) int // Bit length

bits.Reverse(n uint) uint // Reverse bits

bits.RotateLeft(n uint, k int) uint // Rotate bits left

## **Random Numbers (via math/rand)**

rand.Int() int // Random int

rand.Intn(n int) int // Random int in [0,n)

rand.Float64() float64 // Random float in [0.0,1.0)

rand.NormFloat64() float64 // Normally distributed float

rand.ExpFloat64() float64 // Exponentially distributed float

rand.Seed(seed int64) // Set random seed

# ****unicode****

## **Character Categories**

unicode.IsControl(r) bool // Control characters

unicode.IsDigit(r) bool // 0-9

unicode.IsLetter(r) bool // Letters

unicode.IsLower(r) bool // Lowercase letters

unicode.IsUpper(r) bool // Uppercase letters

unicode.IsSpace(r) bool // Whitespace (incl. \t, \n, etc.)

unicode.IsPunct(r) bool // Punctuation

unicode.IsSymbol(r) bool // Symbols

unicode.IsMark(r) bool // Diacritical marks

unicode.IsNumber(r) bool // Numbers (incl. Roman numerals)

unicode.IsGraphic(r) bool // Printable (excluding space)

unicode.IsPrint(r) bool // Printable (including space)

## **Case Conversion**

unicode.ToLower(r) rune // Convert to lowercase

unicode.ToUpper(r) rune // Convert to uppercase

unicode.ToTitle(r) rune // Convert to title case (usually same as Upper)

unicode.SimpleFold(r) rune // Case-fold equivalent rune

## **Character Ranges**

// Range tables (useful for custom checks)

unicode.Latin // All Latin characters(Example)

unicode.Digit // All digits

unicode.Letter // All letters

unicode.Space // All whitespace chars

// Example usage:

unicode.Is(unicode.Latin, r) // Check if rune is Latin

## **Custom Range Checks**

// Create custom range tables

var MyRange = &unicode.RangeTable{

R16: []unicode.Range16{

{Lo: 0x0041, Hi: 0x005A, Stride: 1},},} // A-Z

// Check against custom range

unicode.Is(MyRange, r) bool

## **Unicode Properties**

unicode.In(r, ranges...) bool // Check if in any of the given ranges

unicode.IsOneOf(ranges []\*RangeTable, r) bool // Same as In()

## **Normalization (via unicode/norm package)**

Unicode.norm.NFC.String(s) string // Normalize to NFC form

Unicode.norm.NFD.String(s) string // Normalize to NFD form

Unicode.norm.NFKC.String(s) string // Normalize to NFKC form

Unicode.norm.NFKD.String(s) string // Normalize to NFKD form

## **UTF-8 Helpers**

utf8.RuneCount(s) int // Count runes in string

utf8.ValidString(s) bool // Check if valid UTF-8

utf8.EncodeRune(buf []byte, r) int // Encode rune to UTF-8

utf8.DecodeRune(b []byte) (r, size int) // Decode first rune

## **Common Constants**

unicode.MaxRune // '\U0010FFFF' - maximum valid Unicode code point

unicode.ReplacementChar // '\uFFFD' - replacement character

unicode.Version // Current Unicode version (e.g., "13.0.0")

## **Script Checks**

unicode.Is(unicode.Scripts["Han"], r) // Check if character is Han

unicode.Is(unicode.Scripts["Arabic"], r) // Check if Arabic, etc.

# ****bytes****

## **Basic Operations**

bytes.Contains(b, subslice []byte) bool // Check if subslice exists

bytes.Count(s, sep []byte) int // Count non-overlapping instances

bytes.Equal(a, b []byte) bool // Compare byte slices

bytes.EqualFold(s, t []byte) bool // Case-insensitive comparison

bytes.HasPrefix(s, prefix []byte) bool // Check prefix

bytes.HasSuffix(s, suffix []byte) bool // Check suffix

bytes.Index(s, sep []byte) int // First index of sep

bytes.LastIndex(s, sep []byte) int // Last index of sep

bytes.IndexAny(s []byte, chars string) int // First index of any char in chars

bytes.LastIndexAny(s []byte, chars string) int // Last index of any char in chars

bytes.IndexByte(s []byte, c byte) int // First index of byte

bytes.LastIndexByte(s []byte, c byte) int // Last index of byte

bytes.IndexFunc(s []byte, f func(r rune) bool) int // First index where f returns true

// Last index where f returns true

bytes.LastIndexFunc(s []byte, f func(r rune) bool) int

## **Manipulation**

bytes.Join(s [][]byte, sep []byte) []byte // Join slices with separator

bytes.Repeat(b []byte, count int) []byte // Repeat slice count times

bytes.Replace(s,old, new []byte,n int) []byte // Replace old with new (n = -1 for all)

bytes.ReplaceAll(s, old, new []byte) []byte // Replace all occurrences

bytes.Map(mapping func(r rune) rune, s []byte) []byte // Map runes using function

bytes.Runes(s []byte) []rune // Convert to rune slice

bytes.Split(s, sep []byte) [][]byte // Split by separator

bytes.SplitN(s, sep []byte, n int) [][]byte // Split into at most n slices

bytes.SplitAfter(s, sep []byte) [][]byte // Split after separator

bytes.SplitAfterN(s, sep []byte, n int) [][]byte // Split after into n slices

bytes.Fields(s []byte) [][]byte // Split by whitespace

bytes.FieldsFunc(s []byte, f func(rune) bool) [][]byte // Split using function

## **Transformation**

bytes.Title(s []byte) []byte // Deprecated: Use cases.Title instead

bytes.ToLower(s []byte) []byte // Convert to lowercase

bytes.ToUpper(s []byte) []byte // Convert to uppercase

bytes.ToTitle(s []byte) []byte // Convert to title case

bytes.ToLowerSpecial(c unicode.SpecialCase,s[]byte)[]byte // Case conversion with rules

bytes.ToUpperSpecial(c unicode.SpecialCase, s []byte) []byte

bytes.ToTitleSpecial(c unicode.SpecialCase, s []byte) []byte

bytes.Trim(s []byte, cutset string) []byte // Trim cutset from both ends

bytes.TrimLeft(s []byte, cutset string) []byte // Trim from left

bytes.TrimRight(s []byte, cutset string) []byte // Trim from right

bytes.TrimSpace(s []byte) []byte // Trim whitespace

bytes.TrimPrefix(s, prefix []byte) []byte // Trim prefix

bytes.TrimSuffix(s, suffix []byte) []byte // Trim suffix

bytes.TrimFunc(s []byte, f func(r rune) bool) []byte // Trim using function

bytes.TrimLeftFunc(s []byte, f func(r rune) bool) []byte

bytes.TrimRightFunc(s []byte, f func(r rune) bool) []byte

## **Comparison & Sorting**

bytes.Compare(a, b []byte) int // Returns -1/0/+1

bytes.Clone(b []byte) []byte // Create a copy

bytes.Compact(s []byte) []byte // Remove adjacent duplicates

bytes.CompactFunc(s []byte, f func(r rune) bool) []byte // Compact using function

bytes.Cut(s, sep []byte) (before, after []byte, found bool) // Split around separator

## **Buffer Operations**

// Buffer type (implements io.Reader, io.Writer, etc.)

buf := bytes.NewBuffer(b []byte) // Create buffer from slice

buf := bytes.NewBufferString(s string) // Create buffer from string

// Buffer methods

buf.Bytes() []byte // Get underlying bytes

buf.String() string // Get as string

buf.Len() int // Get length

buf.Cap() int // Get capacity

buf.Grow(n int) // Preallocate space

buf.Reset() // Reset buffer

buf.Truncate(n int) // Truncate to first n bytes

buf.Read(p []byte) (n int, err error) // Read into p

buf.Write(p []byte) (n int, err error) // Write p to buffer

buf.WriteString(s string) (n int, err error) // Write string to buffer

buf.WriteByte(c byte) error // Write single byte

buf.WriteRune(r rune) (n int, err error) // Write rune

buf.ReadByte() (byte, error) // Read single byte

buf.UnreadByte() error // Unread last byte

buf.ReadRune() (r rune, size int, err error) // Read rune

buf.UnreadRune() error // Unread last rune

buf.Next(n int) []byte // Read next n bytes

## **Reader Operations**

r := bytes.NewReader(b []byte) // Create reader from slice

// Reader methods

r.Len() int // Unread bytes

r.Size() int64 // Original length

r.Read(p []byte) (n int, err error) // Read into p

r.ReadAt(p []byte, off int64) (n int, err error) // Read at offset

r.ReadByte() (byte, error) // Read single byte

r.UnreadByte() error // Unread last byte

r.ReadRune() (r rune, size int, err error) // Read rune

r.UnreadRune() error // Unread last rune

r.Seek(offset int64, whence int) (int64, error) // Change read position

r.Reset(b []byte) // Reset with new slice

# ****slices package**** (Go 1.21+)

## **Basic Operations**

slices.Equal(s1, s2 []T) bool // Deep equality check

slices.EqualFunc(s1, s2 []T, eq func(T, T) bool) bool // Custom equality

slices.Compare(s1, s2 []T) int // Returns -1/0/+1 (like bytes.Compare)

slices.CompareFunc(s1, s2 []T, cmp func(T, T) int) int // Custom comparison

slices.Clone(s []T) []T // Create a new copy

slices.Contains(s []T, v T) bool // Check if value exists

slices.Index(s []T, v T) int // First index of value (-1 if not found)

slices.IndexFunc(s []T, f func(T) bool) int // First index where f returns true

slices.LastIndex(s []T, v T) int // Last index of value

slices.LastIndexFunc(s []T, f func(T) bool) int // Last index where f returns true

## **Sorting & Searching**

slices.Sort(s []T) // Ascending sort (uses < operator)

slices.SortFunc(s []T, cmp func(a, T b) int) // Custom sort (returns -1/0/+1)

slices.SortStableFunc(s []T, cmp func(a, T b) int) // Stable sort

slices.IsSorted(s []T) bool // Check if sorted

slices.IsSortedFunc(s []T, cmp func(a, T b) int) bool // Custom sorted check

slices.BinarySearch(s []T, v T) (int, bool) // Search sorted slice (index, found)

slices.BinarySearchFunc(s []T, v T, cmp func(T, T) int) (int, bool) // Custom compare

## **Manipulation**

slices.Insert(s []T, i int, v ...T) []T // Insert values at index

slices.Delete(s []T, i, j int) []T // Remove elements [i,j)

slices.DeleteFunc(s []T, del func(T) bool) []T // Remove elements where del returns true

slices.Replace(s []T, i, j int, v ...T) []T // Replace elements [i,j) with v

slices.Compact(s []T) []T // Remove consecutive duplicates

slices.CompactFunc(s []T, eq func(T, T) bool) []T // Custom duplicate check

slices.Grow(s []T, n int) []T // Increase capacity by n

slices.Clip(s []T) []T // Remove unused capacity

## **Functional Operations**

slices.Reverse(s []T) // Reverse elements in-place

slices.Concat(s ...[]T) []T // Concatenate multiple slices

slices.Repeat(s []T, count int) []T // Create repeated copy (like bytes.Repeat)

slices.Map[S~[]E1,T []E2](s S,f func(E1) E2) T // Map elements to new type (Go 1.22+)

slices.Filter(s []T, keep func(T) bool) []T // Filter elements (Go 1.22+)

slices.Reduce(s []T, f func(T, T) T) T // Reduce elements (Go 1.22+)

## **Specialized Helpers**

slices.Min(s []T) T // Minimum value (panics if empty)

slices.Max(s []T) T // Maximum value (panics if empty)

slices.MinFunc(s []T, cmp func(a, T b) int) T // Custom min comparison

slices.MaxFunc(s []T, cmp func(a, T b) int) T // Custom max comparison

# ****maps package**** (Go 1.21+)

## **Basic Operations**

// Built-in

m := make(map[K]V) // Create map

m := map[K]V{k1: v1, k2: v2} // Literal initialization

v, ok := m[key] // Check existence (ok=true if exists)

delete(m, key) // Remove key-value pair

len(m) // Number of elements

// maps package

maps.Clone(m map[K]V) map[K]V // Create shallow copy

maps.Copy(dst, src map[K]V) // Copy all key-values

maps.DeleteFunc(m map[K]V, del func(K, V) bool) // Conditional delete

maps.Equal(m1, m2 map[K]V) bool // Deep equality check

maps.EqualFunc(m1, m2 map[K]V, cmp func(V, V) bool) bool // Custom value comparison

## **Iteration**

// Built-in (random order)

for k, v := range m {

// k = key, v = value }

// Keys only

for k := range m {

// k = key }

// Values only

for \_, v := range m {

// v = value }

## **Common Patterns**

// Initialize if nil

if m == nil {

m = make(map[K]V) }

// Set default value

if v, ok := m[key]; !ok {

m[key] = defaultValue }

// Invert map (values become keys)

func invert[K, V comparable](m map[K]V) map[V]K {

inv := make(map[V]K, len(m))

for k, v := range m {

inv[v] = k }

return inv }

// Merge maps (later keys overwrite)

func merge[K comparable, V any](maps ...map[K]V) map[K]V {

result := make(map[K]V)

for \_, m := range maps {

maps.Copy(result, m) }

return result }

# ****sort package****

## **Sorting Slices**

// Basic types (int, float64, string)

sort.Ints(s []int)

sort.Float64s(s []float64)

sort.Strings(s []string)

// Custom sorting

sort.Slice(s interface{}, less func(i, j int) bool)

sort.SliceStable(s interface{}, less func(i, j int) bool) // Maintains equal elements' order

// Example custom sort:

people := []struct{

Name string

Age int

}{

{"Alice", 25},

{"Bob", 30},

}

sort.Slice(people, func(i, j int) bool {

return people[i].Age < people[j].Age

})

## **Checking Sort Status**

sort.IntsAreSorted(s []int) bool

sort.Float64sAreSorted(s []float64) bool

sort.StringsAreSorted(s []string) bool

sort.IsSorted(data Interface) bool // For custom types

sort.SliceIsSorted(s interface{}, less func(i, j int) bool) bool

## **Searching Sorted Slices**

// Sorting

sort.Slice(s, func(i, j int) bool {

return s[i] < s[j] // Custom comparison

})

// Binary search (slice must be sorted)

idx := sort.Search(len(s), func(i int) bool {

return s[i] >= target

})

if idx < len(s) && s[idx] == target {

// Found

}

// Check if sorted

isSorted := sort.SliceIsSorted(s, func(i, j int) bool {

return s[i] < s[j]

})

// Predefined searches

sort.SearchInts(s []int, x int) int

sort.SearchFloat64s(s []float64, x float64) int

sort.SearchStrings(s []string, x string) int

// General binary search

sort.Search(n int, f func(int) bool) int

// Example:

x := 42

i := sort.Search(len(data), func(i int) bool { return data[i] >= x })

if i < len(data) && data[i] == x {

// Found

}

## **Sorting Custom Types**

type Interface interface {

Len() int

Less(i, j int) bool

Swap(i, j int)

}

// Example:

type Person struct {

Name string

Age int

}

type ByAge []Person

func (a ByAge) Len() int { return len(a) }

func (a ByAge) Swap(i, j int) { a[i], a[j] = a[j], a[i] }

func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }

// Usage:

people := []Person{{"Bob", 31}, {"Alice", 25}}

sort.Sort(ByAge(people))

## **Reverse Sorting**

// Using sort.Reverse:

sort.Sort(sort.Reverse(ByAge(people)))

// For primitive types:

sort.Slice(s, func(i, j int) bool {

return s[i] > s[j] // Note: > instead of <

})

## **Sorting Multi-dimensional Slices**

matrix := [][]int{{3, 4}, {1, 2}, {5, 6}}

sort.Slice(matrix, func(i, j int) bool {

return matrix[i][0] < matrix[j][0] // Sort by first column

})

## **Stable vs Unstable Sort**

// Stable sort (maintains original order of equal elements)

sort.SliceStable(s, lessFunc)

// Unstable sort (may be faster)

sort.Slice(s, lessFunc)

## **Practical Examples**

// Case-insensitive string sort

sort.Slice(names, func(i, j int) bool {

return strings.ToLower(names[i]) < strings.ToLower(names[j])

})

// Sort map keys

keys := make([]string, 0, len(myMap))

for k := range myMap {

keys = append(keys, k)

}

sort.Strings(keys)

// Sort by multiple fields

sort.Slice(people, func(i, j int) bool {

if people[i].LastName != people[j].LastName {

return people[i].LastName < people[j].LastName

}

return people[i].FirstName < people[j].FirstName

})

# ****container package****

## **container/list (Doubly Linked List)**

import "container/list"

// Initialization

l := list.New() // Creates new empty list

var l list.List // Zero-value list (ready to use)

// Insertion

e := l.PushFront(value)` // O(1)

e := l.PushBack(value) // O(1)

e := l.InsertBefore(v, mark) // O(1)

e := l.InsertAfter(v, mark) // O(1)

// Removal

val := l.Remove(e) // O(1) (panics if e not in list)

// Movement

l.MoveToFront(e) // O(1)

l.MoveToBack(e) // O(1)

l.MoveBefore(e, mark) // O(1)

l.MoveAfter(e, mark) // O(1)

// Access

front := l.Front() // \*Element (nil if empty)

back := l.Back() // \*Element (nil if empty)

next := e.Next() // \*Element

prev := e.Prev() // \*Element

length := l.Len() // O(1)

// Iteration

for e := l.Front(); e != nil; e = e.Next() { /\* ... \*/ } // Forward

for e := l.Back(); e != nil; e = e.Prev() { /\* ... \*/ } // Backward

// Practical Examples: LRU Cache

type LRUCache struct {

capacity int

list \*list.List

cache map[int]\*list.Element }

func (c \*LRUCache) Get(key int) int {

if elem, ok := c.cache[key]; ok {

c.list.MoveToFront(elem)

return elem.Value.(int) }

return -1 }

## **container/heap (Priority Queue)**

import "container/heap"

// Interface (must implement)

type Interface interface {

sort.Interface

Push(x any)

Pop() any }

// Basic operations

heap.Init(h Interface) // O(n) heapify

heap.Push(h Interface, x any) // O(log n)

heap.Pop(h Interface) any // O(log n)

heap.Remove(h Interface, i int) any // O(log n)

heap.Fix(h Interface, i int) // O(log n)

// Example IntHeap implementation:

type IntHeap []int

func (h IntHeap) Len() int { return len(h) }

func (h IntHeap) Less(i, j int) bool { return h[i] < h[j] }

func (h IntHeap) Swap(i, j int) { h[i], h[j] = h[j], h[i] }

func (h \*IntHeap) Push(x any) { \*h = append(\*h, x.(int)) }

func (h \*IntHeap) Pop() any {

old := \*h

n := len(old)

x := old[n-1]

\*h = old[0 : n-1]

return x }

// Usage:

h := &IntHeap{3, 1, 4}

heap.Init(h)

heap.Push(h, 2)

min := heap.Pop(h) // 1

// Practical Examples

// Item with priority

type Item struct {

value string

priority int

index int }

// Update modifies priority and value

func (pq \*PriorityQueue) Update(item \*Item, value string, priority int) {

item.value = value

item.priority = priority

heap.Fix(pq, item.index) }

## **container/ring (Circular List)**

import "container/ring"

// Initialization

r := ring.New(n) // Creates ring with n elements

// Navigation

next := r.Next() // Next element

prev := r.Prev() // Previous element

// Operations

r.Do(func(p any) { /\* ... \*/ }) // Execute function on each element

r.Link(s \*Ring) // Link rings O(n)

r.Unlink(n int) \*Ring // Remove n elements O(n)

r.Move(n int) \*Ring // Move pointer O(n)

// Example usage:

r := ring.New(3)

for i := 1; i <= 3; i++ {

r.Value = i

r = r.Next() }

// Sum all elements

sum := 0

r.Do(func(p any) {

sum += p.(int) })

// Practical Examples: Circular Buffer:

func NewFixedSizeBuffer(size int) \*FixedSizeBuffer {

return &FixedSizeBuffer{

ring: ring.New(size),

size: size, } }

func (b \*FixedSizeBuffer) Write(data []byte) {

for \_, d := range data {

b.ring.Value = d

b.ring = b.ring.Next() } }

# ****io package****

## **Basic Interfaces**

type Reader interface {

Read(p []byte) (n int, err error) }

type Writer interface {

Write(p []byte) (n int, err error) }

type Closer interface {

Close() error }

type Seeker interface {

Seek(offset int64, whence int) (int64, error) }

## **Core Functions**

// Copy Operations

io.Copy(dst Writer, src Reader) (written int64, err error)

io.CopyBuffer(dst Writer, src Reader, buf []byte) (written int64, err error)

io.CopyN(dst Writer, src Reader, n int64) (written int64, err error)

// Reading

io.ReadFull(r Reader, buf []byte) (n int, err error) // Exactly len(buf) bytes

io.ReadAtLeast(r Reader, buf []byte, min int) (n int, err error)

io.LimitReader(r Reader, n int64) Reader // Reads max N bytes

// Writing

io.WriteString(w Writer, s string) (n int, err error)

// Multi-Readers/Writers

io.MultiReader(readers ...Reader) Reader

io.MultiWriter(writers ...Writer) Writer

// Tee Reader

io.TeeReader(r Reader, w Writer) Reader // Writes while reading

// Pipe

io.Pipe() (\*PipeReader, \*PipeWriter) // Synchronous in-memory pipe

## **Utility Functions**

// Offset Handling

io.SeekStart // Whence for Seek (start of file)

io.SeekCurrent // Whence for Seek (current position)

io.SeekEnd // Whence for Seek (end of file)

// EOF Handling

io.EOF // Error returned when no more input available

// Interface Checks

var \_ io.Reader = (\*MyReader)(nil) // Compile-time interface check

## **Combined Interfaces**

type ReadWriter interface {

Reader

Writer }

type ReadCloser interface {

Reader

Closer }

type WriteCloser interface {

Writer

Closer }

type ReadWriteCloser interface {

Reader

Writer

Closer }

type ReadSeeker interface {

Reader

Seeker }

type WriteSeeker interface {

Writer

Seeker }

type ReadWriteSeeker interface {

Reader

Writer

Seeker }

## **Practical Examples**

// Custom Reader

type AlphaReader struct {

src string

pos int }

func (a \*AlphaReader) Read(p []byte) (n int, err error) {

// Implementation that reads only alphabetic chars }

// Usage:

r := &AlphaReader{src: "Hello 123"}

io.Copy(os.Stdout, r) // Outputs "Hello"

// Multi-Writer

file, \_ := os.Create("log.txt")

mw := io.MultiWriter(os.Stdout, file)

fmt.Fprintln(mw, "Hello") // Writes to both console and file

// Limited Reader

r := strings.NewReader("Some long text")

lr := io.LimitReader(r, 4)

io.Copy(os.Stdout, lr) // Outputs "Some"

// Tee Reader

var buf bytes.Buffer

r := io.TeeReader(strings.NewReader("Hello"), &buf)

io.Copy(os.Stdout, r) // Outputs "Hello" and copies to buf

## **Error Handling Patterns**

// Checking for EOF

n, err := r.Read(buf)

if err == io.EOF {

// Handle end of file

} else if err != nil {

// Handle other errors }

// Ensuring Full Read

buf := make([]byte, 1024)

if \_, err := io.ReadFull(r, buf); err != nil {

// Handle partial read }

# ****io/fs package**** (Go 1.16+)

## **Core Interfaces**

type FS interface {

Open(name string) (File, error) } // Primary interface

type File interface {

Stat() (FileInfo, error)

Read([]byte) (int, error)

Close() error }

type FileInfo interface { // Same as os.FileInfo

Name() string

Size() int64

Mode() FileMode

ModTime() time.Time

IsDir() bool

Sys() interface{} }

type DirEntry interface { // Optimized for directory reading

Name() string

IsDir() bool

Type() FileMode

Info() (FileInfo, error) }

type ReadDirFile interface { // For directories

File

ReadDir(n int) ([]DirEntry, error) }

## **Key Functions**

// Read file contents

fs.ReadFile(fsys FS, name string) ([]byte, error)

// Write file (not all FS implementations support this)

fs.WriteFile(fsys FS, name string, data []byte, perm FileMode) error

// Directory operations

fs.ReadDir(fsys FS, name string) ([]DirEntry, error)

fs.Glob(fsys FS, pattern string) ([]string, error)

fs.Sub(fsys FS, dir string) (FS, error) // Get subdirectory FS

// File information

fs.Stat(fsys FS, name string) (FileInfo, error)

fs.FileModeToPerm(mode FileMode) fs.FileMode // Conversion

## **File Modes (fs.FileMode)**

// Permission bits

fs.ModeDir // Directory

fs.ModeAppend // Append-only

fs.ModeExclusive // Exclusive use

fs.ModeTemporary // Temporary file

fs.ModeSymlink // Symbolic link

fs.ModeDevice // Device file

fs.ModeNamedPipe // Named pipe

fs.ModeSocket // Unix domain socket

fs.ModeIrregular // Non-regular file

// Example checking:

if info.Mode().IsDir() { ... }

if info.Mode()&fs.ModeSymlink != 0 { ... }

## **Built-in Implementations**

// OS filesystem

os.DirFS(root string) fs.FS // Creates FS from directory

// In-memory filesystem

fstest.MapFS(map[string]\*fstest.MapFile) fs.FS

// Archive files

zip.NewReader(r io.ReaderAt, size int64) (\*zip.Reader, error) // Implements fs.FS

## **Walk Directory Tree**

fs.WalkDir(fsys FS, root string, fn fs.WalkDirFunc) error

// WalkDirFunc signature:

func(path string, d DirEntry, err error) error

// Example:

fs.WalkDir(os.DirFS("."), ".", func(path string, d fs.DirEntry, err error) error {

if err != nil {

return err }

if !d.IsDir() {

fmt.Println(path) }

return nil })

## **Creating Custom Filesystems**

type MemFS struct {

files map[string]\*MemFile }

func (m \*MemFS) Open(name string) (fs.File, error) {

if file, ok := m.files[name]; ok {

return file, nil }

return nil, &fs.PathError{Op: "open", Path: name, Err: fs.ErrNotExist} }

// Implement File, FileInfo, etc. for MemFile

## **Error Handling**

// Common errors

fs.ErrInvalid // Invalid arguments

fs.ErrPermission // Permission denied

fs.ErrExist // Already exists

fs.ErrNotExist // Does not exist

fs.ErrClosed // Already closed

// Path error wrapping

&fs.PathError{

Op: "open",

Path: "file.txt",

Err: fs.ErrNotExist, }

## **Practical Examples**

// Read Config Files

func ReadConfig(fsys fs.FS) (Config, error) {

data, err := fs.ReadFile(fsys, "config.json")

if err != nil {

return Config{}, err }

// Parse data... }

// Embedded Files (go:embed)

//:embed static/\*

var staticFiles embed.FS // Implements fs.FS

// Usage:

fs.WalkDir(staticFiles, "static", walkFn)

// Mock Filesystem for Testing

mockFS := fstest.MapFS{

"file.txt": {

Data: []byte("hello"),

Mode: 0644,

ModTime: time.Now(), }, }

data, \_ := fs.ReadFile(mockFS, "file.txt")

# ****bufio package****

## **Buffered Readers**

// Creation

reader := bufio.NewReader(io.Reader) // Default buffer size (4096)

reader := bufio.NewReaderSize(io.Reader, size) // Custom buffer size

// Reading methods

reader.Read(p []byte) (n int, err error) // Implements io.Reader

reader.ReadByte() (byte, error) // Single byte

reader.UnreadByte() error // Put back last byte

reader.ReadRune() (r rune, size int, err error) // UTF-8 rune

reader.UnreadRune() error // Put back last rune

reader.ReadLine() (line []byte, isPrefix bool, err error) // Deprecated

reader.ReadString(delim byte) (string, error) // Reads until delimiter

reader.ReadBytes(delim byte) ([]byte, error) // Same but returns bytes

reader.Peek(n int) ([]byte, error) // Preview without consuming

// Example:

r := bufio.NewReader(os.Stdin)

input, \_ := r.ReadString('\n')

## **Buffered Writers**

// Creation

writer := bufio.NewWriter(io.Writer) // Default buffer size (4096)

writer := bufio.NewWriterSize(io.Writer, size) // Custom buffer size

// Writing methods

writer.Write(p []byte) (n int, err error) // Implements io.Writer

writer.WriteByte(c byte) error // Single byte

writer.WriteRune(r rune) (size int, err error) // UTF-8 rune

writer.WriteString(s string) (n int, err error)

writer.Flush() error // Write buffer to underlying writer

writer.Reset(w io.Writer) // Reuse buffer with new writer

writer.Available() int // Bytes unused in buffer

writer.Buffered() int // Bytes in buffer

// Example:

w := bufio.NewWriter(os.Stdout)

w.WriteString("Hello, ")

w.WriteString("World!")

w.Flush() // Don't forget!

## **Scanner (Line-by-Line Processing)**

scanner := bufio.NewScanner(io.Reader)

// Key methods

scanner.Scan() bool // Advances to next token

scanner.Text() string // Current token as string

scanner.Bytes() []byte // Current token as bytes

scanner.Err() error // Any scanning error

// Configuration

scanner.Split(splitFunc bufio.SplitFunc) // Custom split function

bufio.ScanLines // Default (line-based)

bufio.ScanWords // Word-based

bufio.ScanRunes // Rune-based

bufio.ScanBytes // Byte-based

// Example:

scanner := bufio.NewScanner(os.Stdin)

for scanner.Scan() {

fmt.Println("You typed:", scanner.Text()) }

if err := scanner.Err(); err != nil {

// Handle error }

## **ReadWriter (Combined Reader/Writer)**

rw := bufio.NewReadWriter(

bufio.NewReader(io.Reader),

bufio.NewWriter(io.Writer), )

// Inherits all methods from Reader and Writer

## **Custom Split Functions**

// Example: Split at commas

func commaSplit(data []byte, atEOF bool) (advance int, token []byte, err error) {

for i := 0; i < len(data); i++ {

if data[i] == ',' {

return i + 1, data[:i], nil }}

// Handle remaining data when EOF

return 0, data, bufio.ErrFinalToken }

// Usage:

scanner := bufio.NewScanner(strings.NewReader("a,b,c"))

scanner.Split(commaSplit)

## **Performance Tips**

// Buffer Sizes:

// For large files, use bigger buffers (e.g., 64KB)

bufio.NewReaderSize(file, 64\*1024)

// Batch Operations:

// Write multiple strings before flushing

w.WriteString("a"); w.WriteString("b"); w.Flush()

// Reuse Buffers:

// Reset readers/writers to reuse buffers

reader.Reset(newSource)

writer.Reset(newDest)

// Scanner Memory:

// For very long lines, increase buffer:

scanner := bufio.NewScanner(reader)

buf := make([]byte, 0, 64\*1024)

scanner.Buffer(buf, 1024\*1024) // Max token size 1MB

## **Common Patterns**

// Copy with Buffer:

func Copy(dst io.Writer, src io.Reader) (written int64, err error) {

buf := make([]byte, 32\*1024) // 32KB buffer

for {

nr, er := src.Read(buf)

if nr > 0 {

nw, ew := dst.Write(buf[0:nr])

if ew != nil {

return written, ew }

written += int64(nw) }

if er == io.EOF {

break }

if er != nil {

return written, er } }

return written, nil }

// High-Performance Logging:

var logBuffer = bufio.NewWriterSize(os.Stderr, 256\*1024)

func logMessage(msg string) {

logBuffer.WriteString(time.Now().Format(time.RFC3339))

logBuffer.WriteByte(' ')

logBuffer.WriteString(msg)

logBuffer.WriteByte('\n')

// Auto-flush every 1000 messages

if msgCount%1000 == 0 {

logBuffer.Flush() } }

## **Error Handling**

// Reader Errors:

data, err := reader.Peek(5)

if err == bufio.ErrBufferFull {

// Handle buffer full

} else if err != nil {

// Handle other errors }

// Writer Errors:

if writer.Buffered() > 0 {

if err := writer.Flush(); err != nil {

// Handle flush error } }

# ****encoding package****

## **encoding/json**

import "encoding/json"

// Marshal/Unmarshal

json.Marshal(v interface{}) ([]byte, error) // Struct → JSON

json.Unmarshal(data []byte, v interface{}) error // JSON → Struct

// Streaming

json.NewEncoder(w io.Writer) \*json.Encoder

json.NewDecoder(r io.Reader) \*json.Decoder

enc.Encode(v interface{}) error // Write JSON

dec.Decode(v interface{}) error // Read into struct

// Customization

json.MarshalIndent(v, prefix, indent string) // Pretty-print

json.Valid(data []byte) bool // Validate JSON

type Marshaler interface { MarshalJSON() ([]byte, error) }

type Unmarshaler interface { UnmarshalJSON([]byte) error }

// Example:

type Person struct {

Name string `json:"name"`

Age int `json:"age"`

}

data, \_ := json.Marshal(Person{"Alice", 30})

// Practical Examples: JSON Custom Marshaler:

type CustomTime struct {

time.Time }

func (ct CustomTime) MarshalJSON() ([]byte, error) {

return []byte(`"` + ct.Format("2006-01-02") + `"`), nil }

// Error Handling Patterns

if err := json.Unmarshal(data, &obj); err != nil {

if ute, ok := err.(\*json.UnmarshalTypeError); ok {

// Handle type error } }

## **encoding/xml**

import "encoding/xml"

// Marshal/Unmarshal

xml.Marshal(v interface{}) ([]byte, error)

xml.Unmarshal(data []byte, v interface{}) error

// Streaming

xml.NewEncoder(w io.Writer) \*xml.Encoder

xml.NewDecoder(r io.Reader) \*xml.Decoder

// Tags

type Book struct {

Title string `xml:"title"`

ISBN string `xml:"isbn,attr"` }

// Example:

data := `<Book isbn="123"><title>Go</title></Book>`

var b Book

xml.Unmarshal([]byte(data), &b)

## **encoding/csv**

import "encoding/csv"

// Reader

csv.NewReader(r io.Reader) \*csv.Reader

reader.Read() ([]string, error) // Read one record

reader.ReadAll() ([][]string, error) // Read all records

// Writer

csv.NewWriter(w io.Writer) \*csv.Writer

writer.Write(record []string) error

writer.WriteAll(records [][]string) error

writer.Flush()

// Example:

data := "first,last\nJohn,Doe"

r := csv.NewReader(strings.NewReader(data))

records, \_ := r.ReadAll()

// Practical Examples: CSV with Different Delimiter**:**

reader := csv.NewReader(file)

reader.Comma = ';' // Use semicolon delimiter

// Error Handling Patterns

for {

record, err := reader.Read()

if err == io.EOF {

break

}

if perr, ok := err.(\*csv.ParseError); ok {

// Handle parse error } }

## **encoding/gob**

import "encoding/gob"

// Registration

gob.Register(concreteType interface{})

// Encoding/Decoding

gob.NewEncoder(w io.Writer) \*gob.Encoder

gob.NewDecoder(r io.Reader) \*gob.Decoder

enc.Encode(v interface{}) error

dec.Decode(v interface{}) error

// Example:

var network bytes.Buffer

enc := gob.NewEncoder(&network)

enc.Encode(42) // Encode value

## **encoding/base64**

import "encoding/base64"

// Std Encoding

base64.StdEncoding.EncodeToString(src []byte) string

base64.StdEncoding.DecodeString(s string) ([]byte, error)

// URL Encoding

base64.URLEncoding.EncodeToString(src []byte) string

// Custom Encoding

encoding := base64.NewEncoding("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/")

// Example:

data := []byte("hello")

encoded := base64.StdEncoding.EncodeToString(data)

## **encoding/hex**

import "encoding/hex"

hex.EncodeToString(src []byte) string

hex.DecodeString(s string) ([]byte, error)

hex.NewEncoder(w io.Writer) io.Writer

hex.NewDecoder(r io.Reader) io.Reader

// Example:

src := []byte("Hello")

encoded := hex.EncodeToString(src) // "48656c6c6f"

## **encoding/binary**

import "encoding/binary"

// Byte Order

binary.LittleEndian

binary.BigEndian

// Read/Write

binary.Read(r io.Reader, order ByteOrder, data interface{}) error

binary.Write(w io.Writer, order ByteOrder, data interface{}) error

// Varints

binary.PutVarint(buf []byte, x int64) int

binary.Varint(buf []byte) (int64, int)

// Example:

var pi float64

binary.Read(r, binary.LittleEndian, &pi)

// Practical Examples: Binary Data Packing**:**

buf := make([]byte, 4)

binary.BigEndian.PutUint32(buf, 123456)

## **encoding/asn1**

import "encoding/asn1"

asn1.Marshal(v interface{}) ([]byte, error)

asn1.Unmarshal(b []byte, val interface{}) (rest []byte, err error)

// Example:

type Certificate struct {

Raw asn1.RawContent

// ... fields }

## **encoding/pem**

import "encoding/pem"

pem.Encode(w io.Writer, b \*pem.Block) error

pem.Decode(data []byte) (p \*pem.Block, rest []byte)

// Example:

block := &pem.Block{

Type: "CERTIFICATE",

Bytes: certBytes, }

pem.Encode(os.Stdout, block)

# ****sync package****

## **Mutual Exclusion**

var mu sync.Mutex

mu.Lock() // Acquire lock

// Critical section

mu.Unlock() // Release lock

// Example:

var counter int

mu.Lock()

counter++

mu.Unlock()

// Practical Patterns: Debouncing:

var mu sync.Mutex

var timer \*time.Timer

func debounce(f func(), d time.Duration) {

mu.Lock()

defer mu.Unlock()

if timer != nil {

timer.Stop() }

timer = time.AfterFunc(d, f) }

## **RWMutex (Read-Write Lock)**

var rw sync.RWMutex

rw.RLock() // Multiple readers can acquire

// Read operation

rw.RUnlock()

rw.Lock() // Single writer exclusive

// Write operation

rw.Unlock()

## **WaitGroup**

var wg sync.WaitGroup

wg.Add(n) // Add n workers

go func() {

defer wg.Done() // Worker completed

// Task }()

wg.Wait() // Block until all done

// Example:

for i := 0; i < 5; i++ {

wg.Add(1)

go worker(i, &wg) }

wg.Wait()

// Practical Patterns: Worker Pool:

var wg sync.WaitGroup

workCh := make(chan Work)

for i := 0; i < runtime.NumCPU(); i++ {

wg.Add(1)

go func() {

defer wg.Done()

for work := range workCh {

process(work) } }() }

// Add work and close channel

close(workCh)

wg.Wait()

## **Once**

var once sync.Once

once.Do(func() {

// Runs exactly once })

// Example:

var config map[string]string

var loadConfig = func() {

config = readConfig()

}

// Thread-safe singleton:

once.Do(loadConfig)

## **Pool**

var pool = sync.Pool{

New: func() interface{} {

return &Buffer{} }, }

// Get/recycle objects

buf := pool.Get().(\*Buffer)

defer pool.Put(buf)

// Example (byte buffer pool):

var bufPool = sync.Pool{

New: func() interface{} {

return bytes.NewBuffer(make([]byte, 0, 4096)) }, }

## **Cond (Condition Variable)**

var cond = sync.NewCond(&sync.Mutex{})

var ready bool

// Waiter:

cond.L.Lock()

for !ready {

cond.Wait() } // Releases lock while waiting

// Proceed

cond.L.Unlock()

// Signaler:

cond.L.Lock()

ready = true

cond.Broadcast() // or cond.Signal()

cond.L.Unlock()

## **Map (Concurrent Map)**

var cm sync.Map

cm.Store(key, value) // Thread-safe write

val, ok := cm.Load(key) // Thread-safe read

cm.Delete(key)

cm.LoadOrStore(key, value) // Atomic get-or-set

cm.Range(func(key, value interface{}) bool {

// Iteration

return true }) // continue

// Example:

var userMap sync.Map

userMap.Store("alice", User{})

if v, ok := userMap.Load("alice"); ok {

// ... }

## **sync/atomic package**

// Core Operations

// Integer Operations (int32, int64, uint32, uint64, uintptr)

atomic.AddInt32(&var, delta) // Atomic addition

atomic.AddUint64(&var, delta) // Unsigned addition

atomic.LoadInt32(&var) // Atomic read

atomic.LoadUintptr(&var)

atomic.StoreInt64(&var, value) // Atomic write

atomic.StoreUint32(&var, value)

atomic.SwapInt32(&var, new) // Swap and return old value

atomic.SwapUint64(&var, new)

atomic.CompareAndSwapInt32(&var, old, new) // CAS operation (returns bool)

// Pointer Operations (unsafe.Pointer)

atomic.LoadPointer(&ptr) // \*unsafe.Pointer

atomic.StorePointer(&ptr, new)

atomic.SwapPointer(&ptr, new)

atomic.CompareAndSwapPointer(&ptr, old, new)

// Practical Examples

var counter int64 // Atomic Counter:

atomic.AddInt64(&counter, 1) // Increment

current := atomic.LoadInt64(&counter) // Read

old := atomic.SwapInt64(&counter, 0) // Reset

var flag int32 // 0=off, 1=on // Thread-Safe Flag:

atomic.StoreInt32(&flag, 1) // Set

if atomic.LoadInt32(&flag) == 1 {// flag is set } // Check

type Data struct { value int } // Pointer Swap:

var dataPtr unsafe.Pointer = unsafe.Pointer(&Data{42})

newData := &Data{100} // Atomic update

old := atomic.SwapPointer(&dataPtr, unsafe.Pointer(newData))

// Advanced Patterns

// Lock-Free Data Structure:

type Node struct {

value int

next unsafe.Pointer }

func (n \*Node) append(value int) {

newNode := &Node{value: value}

for {

last := atomic.LoadPointer(&n.next)

if atomic.CompareAndSwapPointer(&n.next, last, unsafe.Pointer(newNode)) {

break } } }

// Spinlock:

var lock int32

func acquire() {

for !atomic.CompareAndSwapInt32(&lock, 0, 1) {

runtime.Gosched() // Yield } }

func release() {

atomic.StoreInt32(&lock, 0) }

// Value Type (Go 1.19+)

var shared atomic.Value

shared.Store(42) // Store

val := shared.Load().(int) // Load // Type assertion

// Example with struct:

config := atomic.Value{}

config.Store(Config{Timeout: 10})

current := config.Load().(Config)

// Performance Considerations

// Cache Line Padding:

type Counter struct {

\_ [64]byte // Padding

value int64

\_ [64]byte } // Padding

// Error Handling

// Safe type assertion

val := shared.Load()

if num, ok := val.(int); ok {

// Use num }

// CAS failure handling

for {

old := atomic.LoadInt32(&counter)

new := old + 1

if atomic.CompareAndSwapInt32(&counter, old, new) {

break } }

// Special Cases

// Bool as int32:

var enabled int32

atomic.StoreInt32(&enabled, 1) // true

atomic.StoreInt32(&enabled, 0) // false

// Bit Flags:

const (

flagA = 1 << 0

flagB = 1 << 1 )

var flags uint32

// Set flag

atomic.OrUint32(&flags, flagA)

// Clear flag

atomic.AndUint32(&flags, ^flagB)

# ****context package****

## **Context Creation**

ctx := context.Background() // Background context (root)

ctx := context.TODO() // TODO context (placeholder)

ctx, cancel := context.WithCancel(parent) // WithCancel (manual cancellation)

defer cancel() // Releases resources

// WithTimeout (auto-cancel after duration)

ctx, cancel := context.WithTimeout(parent, 2\*time.Second)

defer cancel()

// WithDeadline (auto-cancel at specific time)

ctx, cancel := context.WithDeadline(parent, time.Now().Add(2\*time.Second))

defer cancel()

// WithValue (carry request-scoped values)

ctx := context.WithValue(parent, key, value)

## **Context Inspection**

// Deadline check

deadline, ok := ctx.Deadline() // (time.Time, bool)

// Cancellation check

select {

case <-ctx.Done():

// Context cancelled

err := ctx.Err() // context.Canceled or context.DeadlineExceeded

default:

// Still active }

// Value lookup

val := ctx.Value(key) // Returns nil if key not found

## **Propagation Patterns**

// HTTP Server:

func handler(w http.ResponseWriter, r \*http.Request) {

ctx := r.Context()

// Pass to downstream calls

result, err := database.Query(ctx, "SELECT...") }

// gRPC Client:

ctx, cancel := context.WithTimeout(context.Background(), 1\*time.Second)

defer cancel()

response, err := client.GetUser(ctx, &pb.UserRequest{Id: 123})

## **Common Patterns**

// Timeout Cascade:

func worker(ctx context.Context) error {

ctx, cancel := context.WithTimeout(ctx, 100\*time.Millisecond)

defer cancel()

// Fast path attempt

if result, err := fastOperation(ctx); err == nil {

return result

}

// Fallback to slow path

return slowOperation(ctx) }

// Value Propagation:

type contextKey string

const (

requestIDKey contextKey = "request-id"

authTokenKey contextKey = "auth-token" )

// Set values

ctx = context.WithValue(ctx, requestIDKey, "12345")

ctx = context.WithValue(ctx, authTokenKey, "bearer xyz")

// Get values

reqID := ctx.Value(requestIDKey).(string) // Type assertion

## **Error Handling**

func DoWork(ctx context.Context) error {

select {

case <-ctx.Done():

return ctx.Err() // Properly propagate cancellation

case result := <-workCh:

return process(result) } }

## **Integration Examples**

// Database Timeout:

ctx, cancel := context.WithTimeout(context.Background(), 3\*time.Second)

defer cancel()

rows, err := db.QueryContext(ctx, "SELECT...")

if errors.Is(err, context.DeadlineExceeded) {

// Handle timeout }

// Parallel Requests:

func gatherData(ctx context.Context) (Result, error) {

ctx, cancel := context.WithCancel(ctx)

defer cancel()

var wg sync.WaitGroup

result := make(chan partialResult, 3)

// Launch parallel fetchers

fetch := func(url string) {

defer wg.Done()

data, err := fetchURL(ctx, url)

if err == nil {

result <- data } }

wg.Add(3)

go fetch("https://api1")

go fetch("https://api2")

go fetch("https://api3")

go func() {

wg.Wait()

close(result) }()

return aggregateResults(result) }

# testing package

import "testing"

## Core test types

type T struct{} // Test context

type B struct{} // Benchmark context

type F struct{} // Fuzz test context

type M struct{} // Test main context

type PB struct{} // Parallel benchmark control

## Test methods (T)

t.Error(args...) // Log error and fail (continue)

t.Errorf(fmt, args...) // Formatted error

t.Fail() // Mark failed (continue)

t.FailNow() // Mark failed (stop)

t.Fatal(args...) // Log error and fail (stop)

t.Fatalf(fmt, args...) // Formatted fatal

t.Log(args...) // Log message

t.Logf(fmt, args...) // Formatted log

t.Skip(args...) // Skip test (stop)

t.Skipf(fmt, args...) // Formatted skip

t.SkipNow() // Skip immediately

t.Parallel() // Run parallel

t.Run(name, testFunc) // Run subtest

t.Cleanup(func()) // Register cleanup

t.Helper() // Mark helper func

t.Name() string // Get test name

t.TempDir() string // Get temp dir

t.Setenv(key, value) // Set env var

t.Deadline() (time.Time, bool) // Get deadline

## Benchmark methods (B)

b.ReportAllocs() // Report allocations

b.ReportMetric(n, unit) // Report custom metric

b.ResetTimer() // Reset measurements

b.StartTimer() // Start timing

b.StopTimer() // Stop timing

b.SetBytes(n) // Set bytes processed

b.Run(name, benchFunc) // Sub-benchmark

b.RunParallel(func(\*PB)) // Parallel benchmark

## Fuzz test methods (F)

f.Add(args...) // Add seed values

f.Fuzz(func(t \*T, args...)) // Fuzz target func

## Test main (M)

m.Run() int // Run all tests

## Parallel benchmark (PB)

pb.Next() bool // Report iteration

## Package functions

testing.Short() bool // Check -short flag

testing.Verbose() bool // Check -v flag

testing.AllocsPerRun(int, func()) float64

testing.CoverMode() string

testing.Coverage() float64

# flag

## **Flag Definition**

flag.String(name, value, usage) // String flag

flag.Int(name, value, usage) // Integer flag

flag.Bool(name, value, usage) // Boolean flag

flag.Float64(name, value, usage) // Float flag

flag.Duration(name, value, usage) // Time duration flag

// Custom type flag

flag.Var(&myVar, name, usage) // Implements flag.Value interface

## **Special Definitions**

flag.StringVar(&var, name, value, usage) // Bind to existing variable

flag.IntVar(&var, name, value, usage) // (Same pattern for all types)

flag.Func(name, usage, func(string) error) // Custom parsing

## **Parsing & Usage**

flag.Parse() // Parse command-line flags

flag.Args() []string // Non-flag arguments

flag.NArg() int // Count of non-flag arguments

flag.NFlag() int // Count of set flags

flag.Usage() // Print usage message

flag.PrintDefaults() // Print all defined flags

## **Lookup & Settings**

flag.Lookup(name) \*flag.Flag // Get flag by name

flag.Set(name, value string) error // Set flag value

flag.Unset(name) // Reset flag to default

flag.Visit(fn func(\*Flag)) // Visit set flags

flag.VisitAll(fn func(\*Flag)) // Visit all flags

## **Flag Object (Returned by Lookup)**

f.Name // Flag name

f.Value // Current value (as String())

f.DefValue // Default value

f.Usage // Usage string

## **Example Setup**

var (

port = flag.Int("port", 8080, "server port")

verbose = flag.Bool("v", false, "verbose mode")

config = flag.String("config", "", "config file") )

func main() {

flag.Parse()

if \*verbose {

log.Println("Starting server...") } }

# flag

## **Basic Logging**

log.Print(v...) // Print log with timestamp

log.Printf(format, v...) // Formatted print

log.Println(v...) // Print with newline

## **Fatal Logging (Exits)**

log.Fatal(v...) // Print + os.Exit(1)

log.Fatalf(format, v...) // Formatted fatal

log.Fatalln(v...) // Fatal with newline

## **Panic Logging**

log.Panic(v...) // Print + panic()

log.Panicf(format, v...) // Formatted panic

log.Panicln(v...) // Panic with newline

## **Configuration**

log.SetPrefix("PREFIX") // Set log line prefix

log.SetFlags(flag) // Set formatting flags

log.SetOutput(w io.Writer) // Set output destination

## **Standard Flags**

log.Ldate // 2023/01/01

log.Ltime // 01:23:45

log.Lmicroseconds // 01:23:45.123456

log.Llongfile // /a/b/c.go:23

log.Lshortfile // c.go:23

log.LUTC // UTC time

log.Lmsgprefix // Put prefix after file/line

## **Custom Logger**

logger := log.New(os.Stderr, "PREFIX", flag)

logger.Println("Message") // Use custom logger

## **Example Setup**

log.SetFlags(log.Ldate | log.Ltime | log.Lshortfile)

// Output: 2023/01/01 01:23:45 main.go:10: Server started

log.Println("Server started")

# ****runtime Package****

## **Goroutine Control**

runtime.Goexit() // Terminates calling goroutine

runtime.Gosched() // Yields processor

runtime.NumGoroutine() int // Returns current goroutine count

## **Memory Management**

runtime.GC() // Runs garbage collection

runtime.ReadMemStats(\*MemStats) // Populates memory statistics

runtime.SetFinalizer(obj, func) // Sets object finalizer

runtime.KeepAlive(interface{}) // Prevents GC before this point

## **System Configuration**

runtime.GOMAXPROCS(n int) int // Sets max CPU cores (returns previous)

runtime.NumCPU() int // Returns system CPU count

runtime.GOROOT() string // Returns Go install path

## **Stack Traces**

runtime.Caller(skip int) (pc uintptr, file string, line int, ok bool)

runtime.Callers(skip int, pc []uintptr) int

runtime.Stack(buf []byte, all bool) int

## **Version Info**

runtime.Version() string // Returns Go version (e.g. "go1.21.0")

runtime.Compiler string // Returns compiler ("gc")

## **Thread Control**

runtime.LockOSThread() // Binds goroutine to OS thread

runtime.UnlockOSThread() // Releases thread binding

# ****runtime/debug Package****

## **Memory Dumps**

debug.FreeOSMemory() // Forces memory return to OS

debug.ReadGCStats(\*GCStats) // Gets GC statistics

## **Stack Traces**

debug.Stack() []byte // Returns current stack trace

debug.PrintStack() // Prints stack to stderr

## **Binary Info**

debug.ReadBuildInfo() (\*BuildInfo, bool) // Module build info

## **Crash Handling**

debug.SetPanicOnFault(bool) bool // Controls crash behavior

debug.SetTraceback(string) // Controls crash output verbosity

## **Performance**

debug.SetGCPercent(int) int // Sets GC target percentage

debug.SetMaxStack(bytes int) int // Sets goroutine stack limit

debug.SetMaxThreads(int) int // Sets thread count limit

## **Key Data Structures**

// runtime.MemStats

type MemStats struct {

Alloc uint64 // Bytes allocated and not freed

TotalAlloc uint64 // Bytes allocated (including freed)

Sys uint64 // Bytes obtained from system

HeapAlloc uint64 // Bytes in heap objects

// ...40+ other fields }

// debug.BuildInfo

type BuildInfo struct {

GoVersion string

Path string

Main Module

Deps []\*Module }

// debug.GCStats

type GCStats struct {

LastGC time.Time

NumGC int64

PauseTotal time.Duration

Pause []time.Duration }

## **Practical Examples**

// Memory Analysis

var m runtime.MemStats

runtime.ReadMemStats(&m)

fmt.Printf("HeapAlloc = %.2f MB", float64(m.HeapAlloc)/1024/1024)

// Force GC

runtime.GC() // Run immediate GC

debug.SetGCPercent(-1) // Disable automatic GC

// Get Build Info

if info, ok := debug.ReadBuildInfo(); ok {

fmt.Println("Built with:", info.GoVersion)

for \_, dep := range info.Deps {

fmt.Println(dep.Path, dep.Version) } }

// Handle Crashes

debug.SetPanicOnFault(true)

debug.SetTraceback("crash")

# ****net Package Core Functions****

## **TCP Connections**

net.Dial("tcp", "host:port") // TCP connection

net.Listen("tcp", ":port") // TCP listener

net.ResolveTCPAddr("tcp", "host:port") // Parse TCP address

## **UDP Connections**

net.DialUDP("udp", nil, addr) // UDP connection

net.ListenUDP("udp", addr) // UDP listener

## **DNS Lookup**

net.LookupHost("domain.com") // [IP] and error

net.LookupIP("domain.com") // [net.IP] and error

net.LookupCNAME("domain.com") // Canonical name

## **IP Address Handling**

net.ParseIP("192.168.1.1") // Returns net.IP

ip.To4() // IPv4 representation

ip.To16() // IPv6 representation

# ****net/http Package****

## **Client**

http.Get("http://example.com") // GET request

http.Post(url, contentType, body) // POST request

http.Head(url) // HEAD request

client := &http.Client{ // Custom client

Timeout: 10 \* time.Second, }

client.Do(req) // Execute request

## **Server**

http.HandleFunc("/path", handler) // Route handler

http.ListenAndServe(":8080", nil) // Start server

http.Handle("/", http.FileServer(http.Dir("/static"))) // File server

srv := &http.Server{ // Custom server

Addr: ":8080",

Handler: mux, }

srv.ListenAndServe()

## **Request/Response**

req, \_ := http.NewRequest("GET", url, nil) // Create request

req.Header.Add("Key", "Value") // Add headers

resp, \_ := client.Do(req) // Get response

body, \_ := io.ReadAll(resp.Body) // Read body

resp.StatusCode // HTTP status

# ****net/url Package****

## **URL Parsing**

u, \_ := url.Parse("http://example.com/path?q=val")

u.Scheme // "http"

u.Host // "example.com"

u.Path // "/path"

u.Query() // Values map

## **URL Construction**

values := url.Values{} // Query values

values.Add("key", "value") // Add param

u.RawQuery = values.Encode() // Set query string

u.String() // Full URL string

## **Encoding**

url.QueryEscape("string with spaces") // %20encoding

url.QueryUnescape("%20decoded") // Original string

url.PathEscape("/path/with spaces") // Path encoding

## **Key Utilities**

// HTTP Middleware

func middleware(next http.Handler) http.Handler {

return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {

// Pre-processing

next.ServeHTTP(w, r)

// Post-processing }) }

// WebSocket Upgrade

var upgrader = websocket.Upgrader{}

conn, \_ := upgrader.Upgrade(w, r, nil) // WS connection

conn.WriteMessage(websocket.TextMessage, []byte(msg))

// Reverse Proxy

proxy := httputil.NewSingleHostReverseProxy(targetUrl)

http.Handle("/", proxy)

# **regexp**

## **Compilation & Matching**

re := regexp.Compile(`pattern`) // Compile regex (panics on error)

re, err := regexp.Compile(`pattern`) // Compile with error

re := regexp.MustCompile(`pattern`) // Panic if invalid

re.Match([]byte) bool // Match against []byte

re.MatchString(string) bool // Match against string

re.MatchReader(io.RuneReader) bool // Match against reader

## **Finding Matches**

re.FindString(string) string // First match

re.FindAllString(string, n int) []string // All matches (n = -1 for all)

re.FindStringSubmatch(string) []string // First match + submatches

re.FindAllStringSubmatch(string, n) [][]string // All matches + submatches

// Byte slice versions:

re.Find([]byte) []byte

re.FindAll([]byte, n) [][]byte

## **Replacement**

re.ReplaceAllString(src, repl) string // Replace all matches

re.ReplaceAllLiteralString(src, repl) // Literal replacement

re.ReplaceAllFunc(src, func([]byte) []byte) []byte // Custom replacement

## **Splitting**

re.Split(string, n int) []string // Split by regex (n = -1 for all)

## **Subexpressions**

re.NumSubexp() int // Number of capture groups

re.SubexpNames() []string // Named capture groups

re.SubexpIndex(name string) int // Index of named group

## **Utility Functions**

regexp.QuoteMeta(string) string // Escape regex metacharacters

## **Regexp Object Methods**

re.String() string // Returns source pattern

re.Longest() // Prefer longest match

re.Copy() \*Regexp // Thread-safe copy

## **Common Patterns**

// Email validation

emailRegex := regexp.MustCompile(`^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`)

// URL extraction

urlRegex := regexp.MustCompile(`https?://[^\s]+`)

// Capture groups

dateRegex := regexp.MustCompile(`(\d{4})-(\d{2})-(\d{2})`)

matches := dateRegex.FindStringSubmatch("2023-01-15")

// matches[0] = "2023-01-15"

// matches[1] = "2023", etc.

# Compress package

## **compress/flate (DEFLATE)**

// Compression

w, \_ := flate.NewWriter(file, flate.BestSpeed) // or BestCompression

w.Write(data) // Compress data

w.Close() // Flush buffer

// Decompression

r := flate.NewReader(compressedFile)

io.Copy(output, r)

r.Close()

## **compress/gzip (.gz files)**

// Compression

w := gzip.NewWriter(file)

w.Write(data)

w.Close() // Required to write footer

// Decompression

r, \_ := gzip.NewReader(compressedFile)

io.Copy(output, r)

r.Close()

// Example (GZIP File)

import (

"compress/gzip"

"os" )

func compressFile(src, dst string) error {

in, \_ := os.Open(src)

defer in.Close()

out, \_ := os.Create(dst)

defer out.Close()

gz := gzip.NewWriter(out)

defer gz.Close()

\_, err := io.Copy(gz, in)

return err }

## **compress/zlib (zlib format)**

// Compression

w, \_ := zlib.NewWriterLevel(file, zlib.BestCompression)

w.Write(data)

w.Close()

// Decompression

r, \_ := zlib.NewReader(compressedFile)

io.Copy(output, r)

r.Close()

## **compress/bzip2 (.bz2 files)**

// Decompression only (no compression in stdlib)

r := bzip2.NewReader(compressedFile)

io.Copy(output, r)

## **compress/lzw (Lempel-Ziv-Welch)**

// Compression

w := lzw.NewWriter(file, lzw.LSB, 8)

w.Write(data)

w.Close()

// Decompression

r := lzw.NewReader(compressedFile, lzw.LSB, 8)

io.Copy(output, r)

r.Close()

## **compress/zstd (Go 1.19+)**

// Compression

w, \_ := zstd.NewWriter(file)

w.Write(data)

w.Close()

// Decompression

r, \_ := zstd.NewReader(compressedFile)

io.Copy(output, r)

r.Close()

## **Key Methods (All Packages)**

w.Write([]byte) // Compress data

w.Close() // Flush buffers (critical!)

r.Read([]byte) // Decompress data

io.Copy(dst, r) // Stream decompression

# **crypto**

## **Hash Functions**

// MD5 (insecure)

import "crypto/md5"

h := md5.New()

h.Write(data)

sum := h.Sum(nil)

// SHA

import "crypto/sha1" // SHA-1 (insecure)

import "crypto/sha256"

import "crypto/sha512"

h := sha256.New()

h.Write(data)

sum := h.Sum(nil)

## **HMAC**

import "crypto/hmac"

h := hmac.New(sha256.New, key)

h.Write(data)

sum := h.Sum(nil)

## **Symmetric Encryption**

import "crypto/aes"

import "crypto/cipher"

// AES

block, \_ := aes.NewCipher(key)

ciphertext := make([]byte, len(data))

block.Encrypt(ciphertext, data)

// GCM Mode

gcm, \_ := cipher.NewGCM(block)

nonce := make([]byte, gcm.NonceSize())

ciphertext := gcm.Seal(nil, nonce, data, nil)

## **Public Key Crypto**

import "crypto/rsa"

import "crypto/rand"

// RSA Key Generation

key, \_ := rsa.GenerateKey(rand.Reader, 2048)

// Encryption

ciphertext, \_ := rsa.EncryptOAEP(sha256.New(), rand.Reader, &key.PublicKey, data, nil)

// Signing

sig, \_ := rsa.SignPKCS1v15(rand.Reader, key, crypto.SHA256, hashed)

## **Elliptic Curves**

import "crypto/ecdsa"

import "crypto/elliptic"

// Key Generation

key, \_ := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)

// Signing

sig, \_ := ecdsa.Sign(rand.Reader, key, hashed)

// Verification

valid := ecdsa.Verify(&key.PublicKey, hashed, r, s)

## **TLS/SSL**

import "crypto/tls"

import "crypto/x509"

// TLS Config

config := &tls.Config{

Certificates: []tls.Certificate{cert},

RootCAs: pool, }

## **X.509 Certificates**

import "crypto/x509"

// Parse Certificate

cert, \_ := x509.ParseCertificate(certBytes)

// Verify

opts := x509.VerifyOptions{

DNSName: "example.com",

Roots: rootPool, }

chains, \_ := cert.Verify(opts)

## **Key Derivation**

import "golang.org/x/crypto/pbkdf2"

key := pbkdf2.Key(password, salt, iter, keyLen, sha256.New)

## **Useful Constants**

crypto.SHA256 // Hash identifier

crypto.MD5SHA1 // Combined hash

# ****embed****

## **Basic Embedding**

import "embed"

// Embed single file

var configFile string //go:embed config.json

// Embed multiple files

var templateFS embed.FS //go:embed templates/\*

// Embed binary file

var logoData []byte //go:embed logo.png

## **FS Methods**

fs.Open(name string) (fs.File, error) // Open file

fs.ReadDir(name string) ([]fs.DirEntry, error) // List directory

fs.ReadFile(name string) ([]byte, error) // Read file content

## **Usage Patterns**

// Read embedded file

data, \_ := templateFS.ReadFile("templates/index.html")

// Serve embedded files

http.Handle("/", http.FileServer(http.FS(templateFS)))

// Access at compile time

func init() {

fmt.Println(configFile) } // Directly use embedded string

## **Special Cases**

// Directory Inclusion: Trailing slash includes all files recursively

var staticFiles embed.FS //go:embed static/\*

// File Patterns: Supports \* and ? wildcards

var textFiles embed.FS //go:embed \*.txt

// Multiple Directives: Combine multiple files

var files embed.FS //go:embed file1.txt file2.txt

# **reflect**

## **Basic Type Inspection**

import "reflect"

t := reflect.TypeOf(x) // Get type of x

v := reflect.ValueOf(x) // Get value of x

kind := v.Kind() // Primitive kind (Int, Struct, etc.)

name := t.Name() // Type name

## **Value Manipulation**

v.Interface() // Convert back to interface{}

v.Int() // Get as int64 (panics if not int)

v.String() // Get as string

v.Bool() // Get as bool

v.Float() // Get as float64

## **Struct Operations**

numField := t.NumField() // Number of fields

field := t.Field(i) // Get StructField

fieldValue := v.Field(i) // Get field value

tag := field.Tag.Get("json") // Get struct tag

## **Slice/Array Operations**

length := v.Len() // Get length

elem := v.Index(i) // Get element

newSlice := reflect.MakeSlice(t, 0, 10) // Create new slice

## **Map Operations**

keys := v.MapKeys() // Get all keys

val := v.MapIndex(key) // Get value by key

newMap := reflect.MakeMap(t) // Create new map

## **Function Operations**

args := []reflect.Value{v1, v2}

results := v.Call(args) // Call function

numIn := t.NumIn() // Number of input params

numOut := t.NumOut() // Number of return values

## **Pointer Operations**

ptr := reflect.New(t) // Create new pointer

elem := v.Elem() // Dereference pointer

## **Special Cases**

v.IsNil() // Check if nil (pointer, map, etc.)

v.CanSet() // Check if value is settable

v.CanAddr() // Check if addressable

v.Addr() // Get pointer to value

v.Convert(t) // Type conversion

# **path**

## **path package (for URL-style paths)**

import "path"

// Path manipulation

path.Base("/a/b/c") // "c" (last element)

path.Dir("/a/b/c") // "/a/b" (parent directory)

path.Ext("file.txt") // ".txt" (extension)

path.Join("a", "b/c") // "a/b/c" (join paths)

path.Clean("/a/../b") // "/b" (normalize path)

// Path inspection

path.IsAbs("/a/b") // true (is absolute path)

path.Split("a/b.txt") // ("a/", "b.txt") (split dir/file)

// Pattern matching

path.Match("\*.txt", "file.txt") // true (pattern match)

## **path/filepath package (for OS filesystem paths)**

import "path/filepath"

// Path manipulation

filepath.Base(`C:\a\b`) // "b" (OS-aware)

filepath.Dir(`C:\a\b`) // `C:\a`

filepath.Ext("file.txt") // ".txt"

filepath.Join("a", "b/c") // "a/b/c" (OS-specific separators)

filepath.Clean("a/../b") // "b" (normalize)

// Path inspection

filepath.IsAbs(`C:\a`) // true (OS-aware)

filepath.Split("a/b.txt") // ("a/", "b.txt")

// Filesystem operations

filepath.Glob("\*.go") // []string of matches

filepath.Walk(".", walkFn) // Recursive directory walk

filepath.Rel("/a/b", "/a/b/c") // "c" (relative path)

// OS-specific

filepath.FromSlash("a/b/c") // Converts to OS separators

filepath.ToSlash(`C:\a\b`) // "C:/a/b" (converts to slashes)

filepath.VolumeName(`C:\a`) // "C:"

# **go/ast**

## **Import**

import "go/ast"

## **Core Types**

ast.File // Represents a Go source file

ast.Node // All AST nodes implement this

ast.Expr // Expression nodes

ast.Stmt // Statement nodes

ast.Decl // Declaration nodes

ast.Comment // Comment nodes

## **AST Inspection**

ast.Inspect(node, func(n ast.Node) bool) // Traverse AST

ast.Print(fset, node) // Print AST structure

ast.Walk(v ast.Visitor, n ast.Node) // Recursive walk

## **Common Node Types**

// Expressions

&ast.Ident{Name: "x"} // Identifier

&ast.BasicLit{Value: "42"} // Literal value

&ast.BinaryExpr{X: x, Op: +, Y: y} // Binary operation

// Statements

&ast.AssignStmt{Lhs: []Expr{x}, Tok: =, Rhs: []Expr{y}} // Assignment

&ast.IfStmt{Cond: cond, Body: block} // If statement

&ast.ReturnStmt{Results: []Expr{x}} // Return statement

// Declarations

&ast.FuncDecl{Name: name, Type: typ, Body: block} // Function

&ast.GenDecl{Tok: token.VAR, Specs: []Spec{spec}} // Variable declaration

## **Utility Functions**

ast.NewIdent(name string) \*ast.Ident // Create identifier

ast.IsExported(name string) bool // Check if name is exported

ast.MergePackageFiles(pkg \*Package) // Merge files in package

## **Visitor Pattern**

type visitor struct{}

func (v visitor) Visit(n ast.Node) ast.Visitor {

if n != nil {

// Process node }

return v }

ast.Walk(visitor{}, node)

## **Special Cases**

// Position Handling:

fset := token.NewFileSet()

file, \_ := parser.ParseFile(fset, "file.go", src, 0)

pos := fset.Position(node.Pos())

// Comment Handling:

ast.NewCommentMap(fset, file, file.Comments)

// Package Inspection:

ast.Package{Files: map[string]\*ast.File}

// Key Notes: Use with go/parser to create ASTs:

f, \_ := parser.ParseFile(fset, filename, src, parser.ParseComments)