Go Programming Language Cheatsheet

A comprehensive cheat sheet for the Go programming language, covering syntax, data types, control flow, concurrency, and more. Perfect for quick reference and learning.

Basics and Syntax

Hello, World!

The classic first program.

```
package main

import "fmt"

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

Explanation:

- `package main` declares this file as part of the main package, the entry point of the application.
- `import "fmt"` imports the `fmt` package, providing formatting and printing functions.
- `func main()` defines the main function where the program execution begins.
- `fmt.Println` prints the given string to standard output.

Variable Declaration

`var` Keyword	Explicitly declares a variable with optional type.
	`var name string = "John"`
	`var age int`
	`var height float64`

`:=` Short Assignment	Declares and initializes a variable, inferring the type. Only usable inside functions.
	`message := "Hello"`
	`count := 10`

Multiple Declarations	Declare multiple variables at once.
	`var ( name string = "Jane" age int = 30 )`
	`firstName, lastName := "John", "Doe"`

Constants

Declaration	Constants are declared like variables, but with the `const` keyword.
	`const PI = 3.14159`
	`const MAX_SIZE int = 100`

Iota	Used for creating enumerated constants. `const ( Sunday = iota Monday Tuesday )`

Data Types

Basic Data Types

Integers: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `uintptr`

Floating-Point Numbers: `float32`, `float64`

Complex Numbers: `complex64`, `complex128`

Boolean: `bool` (`true` or `false`)

String: `string` (UTF-8 encoded)

Example:

```
var age int = 30
var price float64 = 99.99
var isValid bool = true
var message string = "Hello, Go!"
```

Composite Types

Arrays	Fixed-size sequence of elements of the same type.
	`var numbers [5]int`
	`numbers := [5]int{1, 2, 3, 4, 5}`

Slices	Dynamically-sized sequence of elements of the same type. Built on top of arrays.
	`slice := []int{1, 2, 3}`
	`slice = append(slice, 4)`

Maps	Key-value pairs where keys are unique. `ages := map[string]int{ "John": 30, "Jane": 25, }`

Pointers

Declaration	A pointer holds the memory address of a value.
	`var p *int`
	`i := 42`
	`p = &i`

Dereferencing	Accessing the value pointed to by the pointer. `fmt.Println(*p) // Output: 42`

Control Flow

Conditional Statements

`if` Statement:

```
if age >= 18 {
    fmt.Println("Eligible to vote")
} else {
    fmt.Println("Not eligible to vote")
}
```

`if` with Short Statement:

```
if err := doSomething(); err != nil {
    fmt.Println("Error occurred:", err)
}
```

Switch Statement

Basic Switch	Evaluates a variable against a list of cases. `switch day { case "Sunday": fmt.Println("It's Sunday!") case "Monday": fmt.Println("It's Monday!") default: fmt.Println("It's another day.") }`
Fallthrough	Forces execution to continue to the next case. `switch i { case 1: fmt.Println("One") fallthrough case 2: fmt.Println("Two") }`

Looping Constructs

`for` Loop:

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

`for...range` Loop: Iterates over elements in an array, slice, string, map, or channel.

```
numbers := []int{1, 2, 3}
for index, value := range numbers {
    fmt.Println(index, value)
}
```

`while` Loop (simulated with `for`):

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

Functions and Packages

Function Definition

Basic Syntax	`func functionName(parameter1 type1, parameter2 type2) returnType { // Function body return value }`
Multiple Return Values	`func getValues() (int, string) { return 10, "Hello" }` `x, message := getValues()`
Named Return Values	`func split(sum int) (x, y int) { x = sum * 4 / 9 y = sum - x return }`

Packages and Imports

Importing Packages	`import "fmt" // Single import` `import ( "fmt" "math" )`
Package Aliases	`import f "fmt"` `f.Println("Hello")`

Exported Names	Identifiers that start with a capital letter are exported from the package.
	`package mypackage`
	`func MyFunction() {} // Exported`
	`func myFunction() {} // Not exported`

Concurrency

Goroutines

Lightweight, concurrent functions.

```
package main

import (
    "fmt"
    "time"
)

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

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

Channels

Declaration	Used for communication between goroutines. `ch := make(chan int)`
Sending and Receiving	`ch <- 42 // Send value` `value := <-ch // Receive value`
Buffered Channels	Channels with a capacity. `ch := make(chan int, 10)`

WaitGroup

Waits for a collection of goroutines to finish.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int, wg *sync.WaitGroup)
{
    defer wg.Done()

    fmt.Printf("Worker %d starting\n",
id)

    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}

func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 3; i++ {
        wg.Add(1)
        go worker(i, &wg)
    }

    wg.Wait()

    fmt.Println("All workers done!")
}
```