Go Programming Language Cheatsheet

A concise reference to the Go programming language, covering syntax, data types, control structures, and common libraries, to facilitate quick lookups and efficient coding.

Basics & Syntax

Basic Structure

```go
package main

import "fmt"

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

- `package main`: Declares the package as `main`, the entry point of the executable program.
- `import "fmt"`: Imports the "fmt" package, which provides formatted I/O.
- `func main()`: The main function where program execution begins.

Variables & Data Types

Declaration:	`var x int` or `x := 10` (short assignment)
Basic Types:	`int`, `float64`, `bool`, `string`
Constants:	`const PI = 3.14`
Arrays:	`var arr [5]int`
Slices:	`slice := []int{1, 2, 3}`
Maps:	`m := map[string]int{"a": 1}`

Control Structures

If-Else:

```go
if x > 0 {
    //...
} else {
    //...
}
```

For Loop:

```go
for i := 0; i < 10; i++ {
    //...
}
```

Range Loop:

```go
for index, value := range slice {
    //...
}
```

Switch:

```go
switch x {
case 1:
    //...
default:
    //...
}
```

Functions & Methods

Function Definition

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

- `func`: Keyword for defining a function.
- `add`: Function name.
- `(x int, y int)`: Parameters with their types.
- `int`: Return type.

Multiple Return Values

```go
func divide(x int, y int) (int, error) {
    if y == 0 {
        return 0, fmt.Errorf("division by
zero")
    }
    return x / y, nil
}
```

Functions can return multiple values. It's common to return a value and an error.

Methods

```go
type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}
```

- Methods are functions associated with a type. The `(c Circle)` part is the receiver.

Variadic Functions

```go
func sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}
```

- Variadic functions accept a variable number of arguments of the same type.

Concurrency

Goroutines

```go
func myFunc() {
    //...
}

go myFunc() // Launches myFunc in a goroutine
```

- Goroutines are lightweight, concurrent functions. Use the `go` keyword to start a goroutine.

Channels

Declaration:	`ch := make(chan int)`
Send:	`ch <- 10`
Receive:	`value := <-ch`
Buffered Channel:	`ch := make(chan int, 100)`
Close Channel:	`close(ch)`

Select Statement

```go
select {
case msg1 := <-ch1:
    fmt.Println("Received", msg1)
case msg2 := <-ch2:
    fmt.Println("Received", msg2)
default:
    fmt.Println("No message received")
}
```

- The `select` statement allows you to wait on multiple channel operations.

Mutex Locks

Declaration:	`var mu sync.Mutex`
Lock:	`mu.Lock()`
Unlock:	`mu.Unlock()`

Standard Library

fmt Package

Printing:	`fmt.Println("Hello")`
Formatted Printing:	`fmt.Printf("Value: %d", 10)`
Error Printing:	`fmt.Errorf("Error message")`
String Formatting:	`fmt.Sprintf("Value: %d", 10)`

net/http Package

```go
http.HandleFunc("/", func(w
http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, HTTP!")
})

http.ListenAndServe(":8080", nil)
```

- Used for creating HTTP servers. `HandleFunc` registers a function to handle requests to a specific path. `ListenAndServe` starts the server.

os Package

Environment Variables:	`os.Getenv("HOME")`
Command Line Args:	`os.Args`
Exit:	`os.Exit(1)`

io Package

Read from Reader:	`io.ReadAll(reader)`
Write to Writer:	`io.WriteString(writer, "data")`
Copy:	`io.Copy(destination, source)`