Go for Beginners Cheatsheet

educative

1. Introduction to Go

Go is a statically typed, compiled language designed for simplicity and efficiency.

Go Hello, World!

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

Run the Program

```
go run main.go
```

2. Variables

Variables store data and are declared using the `var` keyword or the `:=` shorthand for short declarations.

Variable Declaration

```
var x int = 14
y := 55 // Short declaration
```

Multiple Declarations

```
var a, b, c int = 1, 2, 3
```

3. Constants

Constants are fixed values that cannot be changed once declared.

```
const Pi = 3.14
```

```
const (
    A = 1
    B = 2
)
```

4. Type Conversion

Type conversion allows you to change one data type into another.

```
var x int = 21
var y float64 = float64(x)
```

5. Data Types

Go has several built-in types that define data, such as integers, strings, and booleans.

Basic Types

Go for Beginners Cheatsheet

educative

```
var num int = 42
var str string = "Go"
var flag bool = true
```

Strings

```
s := "Hello"
fmt.Println(len(s))    // Length of the string
fmt.Println(s[1:4])    // Substring
```

Arrays: Arrays are fixed-size collections of elements of the same type.

```
var arr [3]int = [3]int{1, 2, 3}
fmt.Println(arr[0])    // Access element
```

Slices: Slices are flexible, dynamic-sized collections that are backed by arrays.

```
slice := []int{1, 2, 3}
slice = append(slice, 4) // Append elements
```

Pointers: Pointers store the memory address of another variable.

```
var x int = 5
var p *int = &x
fmt.Println(*p) // Dereferencing
```

6. Flow Control

Go has several built-in types that define data, such as integers, strings, and booleans.

Conditional Statements

```
if x := 10; x > 5 {
    fmt.Println("Greater than 5")
} else {
    fmt.Println("5 or less")
}
```

Switch Statement: Switch statement allows multi-way branching based on conditions.

```
switch day := "Monday"; day {
case "Monday":
    fmt.Println("Start of the week")
case "Friday":
    fmt.Println("End of the week")
```

```
default:
    fmt.Println("Midweek")
}
```

For loop: The `for` loop is used to repeat a block of code a specific number of times.

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

Go for Beginners Cheatsheet

For-range loop: The `for-range` loop iterates over arrays or slices and returns index and value.

```
nums := []int{1, 2, 3}
for i, v := range nums {
    fmt.Printf("Index: %d, Value: %d\n", i, v)
}
```

While loop (using for): Go doesn't have a `while` loop but uses `for` as a while loop with a condition.

```
n := 0
for n < 5 {
    fmt.Println(n)
    n++
}
```

7. Functions

Functions are blocks of reusable code designed to perform a specific task.

Basic Function

```
func add(a int, b int) int {
    return a + b
}
```

Multiple return values: Functions can return multiple values, which is useful for returning error codes or complex results.

```
func divide(a int, b int) (int, int) {
    return a / b, a % b
}
result, remainder := divide(10, 3)
```

Named return values: Functions can return named values, making the code more readable.

```
func calc(a int, b int) (sum int, product int) {
    sum = a + b
    product = a * b
    return
}
```

Anonymous function (Lambda): Anonymous functions allow you to define a function without naming it.

```
sum := func(a int, b int) int {
    return a + b
}
fmt.Println(sum(2, 3))
```

8. Error Handling

Error handling in Go is done explicitly by returning error types and checking them.

Basic Error Handling

```
file, err := os.Open("file.txt")
if err != nil {
    log.Fatal(err)
}
defer file.Close()
```

Defer statement: The `defer` statement ensures a function call is delayed until the surrounding function returns.

```
func main() {
    defer fmt.Println("Executed last")
    fmt.Println("Executed first")
}
```

Multiple deferred functions: Deferred functions are executed in LIFO (Last In, First Out) order.

```
func main() {
    defer fmt.Println("1")
    defer fmt.Println("2")
    defer fmt.Println("3")
    fmt.Println("Main executed")
}
```

9. Concurrency (Goroutines and Channels)

Goroutines: Go provides goroutines for concurrency, allowing multiple functions to run simultaneously.

```
go func() {
    fmt.Println("Running concurrently")
}()
```

Channels: Channels allow goroutines to communicate with each other.

```
ch := make(chan string)
go func() {
    ch <- "Hello from goroutine"
}()
msg := <-ch
fmt.Println(msg)
```

10. Packages and Libraries

Go programs are organized into packages. You can import standard libraries or third-party ones to extend functionality.

```
import "fmt"
import "math"

func main() {
    fmt.Println(math.Sqrt(16))
}
```