# Go Functions Tutorial - Complete Learning Guide

Welcome to the comprehensive Go Functions Tutorial! This guide will teach you everything you need to know about writing functions in Go, from basic concepts to advanced patterns.

## Table of Contents



---

## Function Basics

### What is a Function?

A function is a reusable block of code that performs a specific task. Functions help organize code, reduce repetition, and make programs easier to understand and maintain.

### Basic Function Syntax

go

```go
func functionName(parameter1 type1, parameter2 type2) returnType {
    // function body
    return value
}
```

### Your First Function

go

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

func main() {
    greet() // Call the function
}
```

**Key Points:**

- Use `func` keyword to declare a function
- Function names should be descriptive
- Use parentheses `()` for parameters (even if empty)
- Use curly braces `{}` for the function body

---

## Parameters and Return Values

### Functions with Parameters

```go
func greetPerson(name string) {
    fmt.Printf("Hello, %s!\n", name)
}

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

// Shorthand for same-type parameters
func multiply(a, b int) int {
    return a * b
}
```

### Practice Example

```go
func calculateArea(length, width float64) float64 {
    return length * width
}

func main() {
    area := calculateArea(5.0, 3.0)
    fmt.Printf("Area: %.2f\n", area)
}
```

**Exercise 1:** Write a function that takes a person's name and age, then prints a personalized message.

---

## Multiple Return Values

Go functions can return multiple values, which is especially useful for error handling.

go

```go
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("cannot divide by zero")
    }
    return a / b, nil
}

func main() {
    result, err := divide(10, 2)
    if err != nil {
        fmt.Printf("Error: %v\n", err)
        return
    }
    fmt.Printf("Result: %.2f\n", result)
}
```

### Common Pattern: Value and Error

go

```go
func parseAge(ageStr string) (int, error) {
    age, err := strconv.Atoi(ageStr)
    if err != nil {
        return 0, fmt.Errorf("invalid age: %s", ageStr)
    }
    if age < 0 || age > 150 {
        return 0, errors.New("age out of valid range")
    }
    return age, nil
}
```

**Exercise 2:** Write a function that returns both the quotient and remainder of integer division.

---

## Named Return Values

You can name return values in the function signature:

go

```go
func rectangleInfo(length, width float64) (area, perimeter float64) {
    area = length * width
    perimeter = 2 * (length + width)
    return // naked return
}

// Equivalent to:
func rectangleInfoExplicit(length, width float64) (float64, float64) {
    area := length * width
    perimeter := 2 * (length + width)
    return area, perimeter
}
```

**When to Use Named Returns:**

- For short functions where it improves readability
- When you need to modify return values in defer statements
- Avoid in long functions as it can reduce clarity

---

## Variadic Functions

Variadic functions accept a variable number of arguments:

go

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

func main() {
    fmt.Println(sum(1, 2, 3))        // 6
    fmt.Println(sum(1, 2, 3, 4, 5))  // 15

    // Pass slice with spread operator
    nums := []int{10, 20, 30}
    fmt.Println(sum(nums...))        // 60
}
```

### Mixing Regular and Variadic Parameters

go

```go
func logMessage(level string, messages ...string) {
    fmt.Printf("[%s] ", level)
    for i, msg := range messages {
        if i > 0 {
            fmt.Print(" | ")
        }
        fmt.Print(msg)
    }
    fmt.Println()
}

func main() {
    logMessage("INFO", "User logged in")
    logMessage("ERROR", "Database error", "Connection timeout", "Retry failed")
}
```

**Exercise 3:** Create a function that finds the maximum value among any number of integers.

---

## Function Values and Types

In Go, functions are first-class values - they can be assigned to variables, passed as arguments, and returned from functions.

go

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

func subtract(a, b int) int {
    return a - b
}

func main() {
    // Assign function to variable
    var operation func(int, int) int
    operation = add
    fmt.Println(operation(5, 3)) // 8

    operation = subtract
    fmt.Println(operation(5, 3)) // 2
}
```

## Function Types

go

```go
type MathOperation func(int, int) int
type StringProcessor func(string) string

func processString(s string, processor StringProcessor) string {
    return processor(s)
}

func toUpper(s string) string {
    return strings.ToUpper(s)
}

func main() {
    result := processString("hello", toUpper)
    fmt.Println(result) // HELLO
}
```

---

## Anonymous Functions

Anonymous functions are functions without names, often used for short-lived operations:

```go
func main() {
    // Anonymous function assigned to variable
    square := func(x int) int {
        return x * x
    }
    fmt.Println(square(5)) // 25

    // Immediately invoked function expression (IIFE)
    result := func(a, b int) int {
        return a + b
    }(10, 20)
    fmt.Println(result) // 30

    // Anonymous function in slice
    operations := []func(int, int) int{
        func(a, b int) int { return a + b },
        func(a, b int) int { return a - b },
        func(a, b int) int { return a * b },
    }

    for i, op := range operations {
        fmt.Printf("Operation %d: %d\n", i, op(6, 3))
    }
}
```

**Exercise 4:** Create a slice of anonymous functions that perform different string transformations.

---

## Closures

A closure is a function that references variables from outside its body:

go

```go
func counter() func() int {
    count := 0
    return func() int {
        count++
        return count
    }
}

func main() {
    myCounter := counter()
    fmt.Println(myCounter()) // 1
    fmt.Println(myCounter()) // 2
    fmt.Println(myCounter()) // 3

    // Each counter is independent
    anotherCounter := counter()
    fmt.Println(anotherCounter()) // 1
}
```

### Practical Closure Example

go

```go
func fileProcessor(prefix string) func(string) string {
    return func(filename string) string {
        return fmt.Sprintf("%s_%s", prefix, filename)
    }
}

func main() {
    logProcessor := fileProcessor("log")
    dataProcessor := fileProcessor("data")

    fmt.Println(logProcessor("error.txt"))    // log_error.txt
    fmt.Println(dataProcessor("users.csv"))   // data_users.csv
}
```

**Exercise 5:** Create a closure that generates a sequence of numbers with a custom step size.

---

## Higher-Order Functions

Higher-order functions either take functions as parameters or return functions:

```go
// Function that takes a function as parameter
func applyToSlice(slice []int, fn func(int) int) []int {
    result := make([]int, len(slice))
    for i, v := range slice {
        result[i] = fn(v)
    }
    return result
}

// Function that returns a function
func createMultiplier(factor int) func(int) int {
    return func(x int) int {
        return x * factor
    }
}

func main() {
    numbers := []int{1, 2, 3, 4, 5}

    // Double all numbers
    doubled := applyToSlice(numbers, func(x int) int { return x * 2 })
    fmt.Println(doubled) // [2 4 6 8 10]

    // Square all numbers
    squared := applyToSlice(numbers, func(x int) int { return x * x })
    fmt.Println(squared) // [1 4 9 16 25]

    // Using function factory
    triple := createMultiplier(3)
    tripled := applyToSlice(numbers, triple)
    fmt.Println(tripled) // [3 6 9 12 15]
}
```

**Exercise 6:** Write a function that filters a slice based on a predicate function.

---

## Methods

Methods are functions with a receiver - they belong to a specific type:

go

```go
type Rectangle struct {
    Width  float64
    Height float64
}

// Method with value receiver
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

// Method with pointer receiver
func (r *Rectangle) Scale(factor float64) {
    r.Width *= factor
    r.Height *= factor
}

func main() {
    rect := Rectangle{Width: 5, Height: 3}
    fmt.Printf("Area: %.2f\n", rect.Area()) // 15.00

    rect.Scale(2)
    fmt.Printf("After scaling - Width: %.2f, Height: %.2f\n", rect.Width, rect.Height)
    fmt.Printf("New area: %.2f\n", rect.Area()) // 60.00
}
```

## Value vs Pointer Receivers

- **Value receiver**: Method receives a copy of the value
- **Pointer receiver**: Method receives a pointer to the original value

go

```go
type Counter struct {
    count int
}

// This WON'T modify the original counter
func (c Counter) IncrementWrong() {
    c.count++
}

// This WILL modify the original counter
func (c *Counter) Increment() {
    c.count++
}

func (c Counter) Value() int {
    return c.count
}
```

**Exercise 7:** Create a `BankAccount` struct with methods for deposit, withdraw, and balance inquiry.

---

## Recursive Functions

Recursive functions call themselves:

go

```go
func factorial(n int) int {
    if n <= 1 {
        return 1 // base case
    }
    return n * factorial(n-1) // recursive case
}

func fibonacci(n int) int {
    if n <= 1 {
        return n
    }
    return fibonacci(n-1) + fibonacci(n-2)
}

// More efficient fibonacci with memoization
func fibonacciMemo(n int, memo map[int]int) int {
    if n <= 1 {
        return n
    }
    if val, exists := memo[n]; exists {
        return val
    }
    memo[n] = fibonacciMemo(n-1, memo) + fibonacciMemo(n-2, memo)
    return memo[n]
}
```

## Tree Traversal Example

go

```go
type TreeNode struct {
    Value int
    Left  *TreeNode
    Right *TreeNode
}

func (t *TreeNode) PrintInOrder() {
    if t != nil {
        t.Left.PrintInOrder()
        fmt.Printf("%d ", t.Value)
        t.Right.PrintInOrder()
    }
}
```

**Exercise 8:** Write a recursive function to calculate the sum of digits in a number.

---

## Defer Statements

`defer` schedules a function call to run when the surrounding function returns:

go

```go
func readFile(filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close() // Ensures file is closed even if error occurs

    // Read file content...
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        fmt.Println(scanner.Text())
    }

    return scanner.Err()
}
```

### Multiple Defers (LIFO - Last In, First Out)

go

```go
func deferExample() {
    fmt.Println("Start")
    defer fmt.Println("First defer")
    defer fmt.Println("Second defer")
    defer fmt.Println("Third defer")
    fmt.Println("End")
}
// Output:
// Start
// End
// Third defer
// Second defer
// First defer
```

### Defer with Parameters

go

```go
func deferWithParams() {
    i := 0
    defer fmt.Println("Deferred:", i) // i is evaluated now (0)
    i++
    fmt.Println("Current:", i) // 1
}
// Output:
// Current: 1
// Deferred: 0
```

**Exercise 9:** Write a function that measures and prints the execution time using defer.

---

## Error Handling Patterns

Go uses explicit error handling with multiple return values:

### Basic Error Handling

go

```
import (
    "errors"
    "fmt"
    "strconv"
)

func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division by zero")
    }
    return a / b, nil
}
```

## Custom Error Types

go

```
type ValidationError struct {
    Field string
    Value string
    Issue string
}

func (e ValidationError) Error() string {
    return fmt.Sprintf("validation failed for %s: %s (value: %s)",
        e.Field, e.Issue, e.Value)
}

func validateEmail(email string) error {
    if !strings.Contains(email, "@") {
        return ValidationError{
            Field: "email",
            Value: email,
            Issue: "missing @ symbol",
        }
    }
    return nil
}
```

## Error Wrapping (Go 1.13+)

go

```go
import "fmt"

func processFile(filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return fmt.Errorf("failed to open file %s: %w", filename, err)
    }
    defer file.Close()

    // Process file...
    return nil
}
```

**Exercise 10:** Create a user registration function that validates multiple fields and returns appropriate errors.

---

## Best Practices

### 1. Function Naming

- Use clear, descriptive names
- Start with a verb when possible
- Use camelCase for private functions
- Use PascalCase for exported functions

go

```go
// Good
func calculateTotalPrice(items []Item) float64 { ... }
func (u *User) UpdateProfile(profile Profile) error { ... }

// Avoid
func calc(items []Item) float64 { ... }
func do(u *User, p Profile) error { ... }
```

### 2. Function Length

- Keep functions short and focused
- One function should do one thing well

- If a function is too long, break it into smaller functions

### 3. Parameter Guidelines

- Limit the number of parameters (prefer structs for many parameters)
- Put the most important parameters first
- Use meaningful parameter names

go

```go
// Good
type OrderRequest struct {
    CustomerID string
    Items      []Item
    ShippingAddress Address
    PaymentMethod   string
}

func CreateOrder(req OrderRequest) (*Order, error) { ... }

// Avoid
func CreateOrder(customerID string, item1 Item, item2 Item,
    addr1 string, addr2 string, city string, payment string) (*Order, error) { ... }
```

### 4. Return Value Guidelines

- Return errors as the last value
- Use named returns sparingly and only for short functions
- Don't ignore errors

### 5. Documentation

go

```go
// CalculateDistance calculates the distance between two points.
// It returns the distance as a float64 and any error encountered.
func CalculateDistance(p1, p2 Point) (float64, error) {
    // Implementation...
}
```

---

## Practice Exercises

### Exercise 1: Basic Functions

Write a function that takes a person's name and age, then prints a personalized message.

<details> <summary>Solution</summary>

```go
func greetPerson(name string, age int) {
    fmt.Printf("Hello, %s! You are %d years old.\n", name, age)
}
```

</details>

### Exercise 2: Multiple Return Values

Write a function that returns both the quotient and remainder of integer division.

<details> <summary>Solution</summary>

```go
func divideWithRemainder(a, b int) (int, int, error) {
    if b == 0 {
        return 0, 0, errors.New("division by zero")
    }
    return a / b, a % b, nil
}
```

</details>

### Exercise 3: Variadic Functions

Create a function that finds the maximum value among any number of integers.

<details> <summary>Solution</summary>

go

```go
func max(numbers ...int) (int, error) {
    if len(numbers) == 0 {
        return 0, errors.New("no numbers provided")
    }

    maximum := numbers[0]
    for _, num := range numbers[1:] {
        if num > maximum {
            maximum = num
        }
    }
    return maximum, nil
}
```

</details>

### Exercise 4: Anonymous Functions

Create a slice of anonymous functions that perform different string transformations.

<details> <summary>Solution</summary>

go

```go
func main() {
    transforms := []func(string) string{
        func(s string) string { return strings.ToUpper(s) },
        func(s string) string { return strings.ToLower(s) },
        func(s string) string { return strings.Title(s) },
        func(s string) string {
            runes := []rune(s)
            for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 {
                runes[i], runes[j] = runes[j], runes[i]
            }
            return string(runes)
        },
    }

    text := "hello world"
    for i, transform := range transforms {
        fmt.Printf("Transform %d: %s\n", i+1, transform(text))
    }
}
```

</details>

### Exercise 5: Closures

Create a closure that generates a sequence of numbers with a custom step size.

<details> <summary>Solution</summary>

go

```go
func sequenceGenerator(start, step int) func() int {
    current := start - step // Start one step before to get correct first value
    return func() int {
        current += step
        return current
    }
}

func main() {
    evenNumbers := sequenceGenerator(0, 2)
    fmt.Println(evenNumbers()) // 0
    fmt.Println(evenNumbers()) // 2
    fmt.Println(evenNumbers()) // 4

    oddNumbers := sequenceGenerator(1, 2)
    fmt.Println(oddNumbers()) // 1
    fmt.Println(oddNumbers()) // 3
    fmt.Println(oddNumbers()) // 5
}
```

</details>

### Exercise 6: Higher-Order Functions

Write a function that filters a slice based on a predicate function.

<details> <summary>Solution</summary>

go

```go
func filter(slice []int, predicate func(int) bool) []int {
    var result []int
    for _, value := range slice {
        if predicate(value) {
            result = append(result, value)
        }
    }
    return result
}

func main() {
    numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    // Filter even numbers
    evens := filter(numbers, func(n int) bool { return n%2 == 0 })
    fmt.Println("Evens:", evens) // [2 4 6 8 10]

    // Filter numbers greater than 5
    greaterThan5 := filter(numbers, func(n int) bool { return n > 5 })
    fmt.Println("Greater than 5:", greaterThan5) // [6 7 8 9 10]
}
```

</details>

### Exercise 7: Methods

Create a `BankAccount` struct with methods for deposit, withdraw, and balance inquiry.

<details> <summary>Solution</summary>

go

```go
type BankAccount struct {
    accountNumber string
    balance       float64
}

func NewBankAccount(accountNumber string) *BankAccount {
    return &BankAccount{
        accountNumber: accountNumber,
        balance:       0.0,
    }
}

func (ba *BankAccount) Deposit(amount float64) error {
    if amount <= 0 {
        return errors.New("deposit amount must be positive")
    }
    ba.balance += amount
    return nil
}

func (ba *BankAccount) Withdraw(amount float64) error {
    if amount <= 0 {
        return errors.New("withdrawal amount must be positive")
    }
    if amount > ba.balance {
        return errors.New("insufficient funds")
    }
    ba.balance -= amount
    return nil
}

func (ba BankAccount) Balance() float64 {
    return ba.balance
}

func (ba BankAccount) AccountNumber() string {
    return ba.accountNumber
}
```

</details>

### Exercise 8: Recursion

Write a recursive function to calculate the sum of digits in a number.

<details> <summary>Solution</summary>

```go
func sumOfDigits(n int) int {
    // Handle negative numbers
    if n < 0 {
        n = -n
    }

    // Base case
    if n < 10 {
        return n
    }

    // Recursive case
    return n%10 + sumOfDigits(n/10)
}

func main() {
    fmt.Println(sumOfDigits(123))   // 6 (1+2+3)
    fmt.Println(sumOfDigits(9876))  // 30 (9+8+7+6)
    fmt.Println(sumOfDigits(-456))  // 15 (4+5+6)
}
```

</details>

### Exercise 9: Defer

Write a function that measures and prints the execution time using defer.

<details> <summary>Solution</summary>

```go
import "time"

func measureTime(name string) func() {
    start := time.Now()
    return func() {
        fmt.Printf("%s took %v\n", name, time.Since(start))
    }
}

func expensiveOperation() {
    defer measureTime("expensiveOperation")()

    // Simulate some work
    time.Sleep(100 * time.Millisecond)

    // Do some calculations
    total := 0
    for i := 0; i < 1000000; i++ {
        total += i
    }
}
```

</details>

### Exercise 10: Error Handling

Create a user registration function that validates multiple fields and returns appropriate errors.

<details> <summary>Solution</summary>

go

```go
type User struct {
    Username string
    Email    string
    Age      int
}

type RegistrationError struct {
    Field   string
    Message string
}

func (e RegistrationError) Error() string {
    return fmt.Sprintf("registration error - %s: %s", e.Field, e.Message)
}

func validateUsername(username string) error {
    if len(username) < 3 {
        return RegistrationError{"username", "must be at least 3 characters"}
    }
    if len(username) > 20 {
        return RegistrationError{"username", "must be no more than 20 characters"}
    }
    return nil
}

func validateEmail(email string) error {
    if !strings.Contains(email, "@") {
        return RegistrationError{"email", "must contain @ symbol"}
    }
    if !strings.Contains(email, ".") {
        return RegistrationError{"email", "must contain a domain"}
    }
    return nil
}

func validateAge(age int) error {
    if age < 13 {
        return RegistrationError{"age", "must be at least 13"}
    }
    if age > 120 {
        return RegistrationError{"age", "must be realistic"}
    }
    return nil
```

```
}

func RegisterUser(username, email string, age int) (*User, error) {
    if err := validateUsername(username); err != nil {
        return nil, err
    }
    if err := validateEmail(email); err != nil {
        return nil, err
    }
    if err := validateAge(age); err != nil {
        return nil, err
    }

    user := &User{
        Username: username,
        Email:    email,
        Age:      age,
    }

    return user, nil
}
```

</details>

---

## Conclusion

Congratulations! You've completed the Go Functions Tutorial. You've learned:

- How to write basic functions with parameters and return values
- Advanced function patterns like variadic functions, closures, and higher-order functions
- How to work with methods and receivers
- Error handling patterns in Go
- Best practices for writing clean, maintainable functions

### Next Steps

1. Practice writing functions for real-world problems
2. Study the Go standard library to see function patterns in action
3. Learn about interfaces and how they work with methods
4. Explore Go's concurrency features (goroutines and channels)

5. Build projects that combine all these concepts

Keep practicing, and happy coding in Go! 🚀