Multiple Return Values Go

```go
package main
import "fmt"

func divide(a, b int) (int, int, error) {
    if b == 0 {
        return 0, 0, fmt.Errorf("division by zero")
    }
    return a / b, a % b, nil
}

func main() {
    q, r, err := divide(10, 3)
    fmt.Println(q, r, err)
}
```

Ignoring Return Values Go

```go
package main
import "fmt"

func calculate() (int, int, int) {
    return 1, 2, 3
}

func main() {
    a, _, c := calculate()
    fmt.Println(a, c)
}
```

Named Return Values Go

```go
package main
import "fmt"

func divide(a, b int) (quotient, remainder int) {
    quotient = a / b
    remainder = a % b
    return
}

func main() {
    q, r := divide(17, 5)
    fmt.Println(q, r)
}
```

Named Returns Shadowing Trap Go

```go
package main
import "fmt"

func calculate() (result int) {
    result = 10
    if true {
        result := 20
        fmt.Println(result)
    }
    return
}

func main() {
    fmt.Println(calculate())
}
```

Variadic Function Basics Go

```go
package main
import "fmt"

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

func main() {
    fmt.Println(sum(1, 2, 3, 4, 5))
    fmt.Println(sum())
}
```

Variadic Unpacking Go

```go
package main
import "fmt"

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

func main() {
    nums := []int{1, 2, 3, 4, 5}
    fmt.Println(sum(nums...))
}
```

Variadic Must Be ... Parameter Go

```go
package main
import "fmt"

func process(prefix string, nums ...int) {
    fmt.Print(prefix, ": ")
    for _, n := range nums {
        fmt.Print(n, " ")
    }
    fmt.Println()
}

func main() {
    process("Numbers", 1, 2, 3, 4)
}
```

Functions as Values Go

```go
package main
import "fmt"

func main() {
    f := func(x int) int {
        return x * 2
    }
    fmt.Println(f(5))
}
```

Function Type Declaration Go

```go
package main
import "fmt"

type MathOp func(int, int) int

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

func main() {
    var op MathOp = add
    fmt.Println(op(3, 4))
}
```

Closures Capture Variables Go

```go
package main
import "fmt"

func main() {
    x := 10
    f := func() {
        fmt.Println(x)
    }
    x = 20
    f()
}
```

Closure Loop Variable Trap Go

```go
package main
import "fmt"

func main() {
    funcs := []func(){}
    for i := 0; i < 3; i++ {
        funcs = append(funcs, func() {
            fmt.Println(i)
        })
    }
    for _, f := range funcs {
        f()
    }
}
```

Fixing Closure Loop Variable Go

```go
package main
import "fmt"

func main() {
    funcs := []func(){}
    for i := 0; i < 3; i++ {
        i := i // Shadow with copy
        funcs = append(funcs, func() {
            fmt.Println(i)
        })
    }
    for _, f := range funcs {
        f()
    }
}
```

Defer Basics Go

```go
package main
import "fmt"

func main() {
    defer fmt.Println("world")
    fmt.Println("hello")
}
```

Multiple Defers (LIFO) Go

```go
package main
import "fmt"

func main() {
    defer fmt.Println("1")
    defer fmt.Println("2")
    defer fmt.Println("3")
    fmt.Println("4")
}
```

Defer Captures Arguments Go

```go
package main
import "fmt"

func main() {
    x := 10
    defer fmt.Println(x)
    x = 20
    fmt.Println(x)
}
```

Defer with Named Returns Go

```go
package main
import "fmt"

func calculate() (result int) {
    defer func() {
        result++
    }()
    result = 10
    return
}

func main() {
    fmt.Println(calculate())
}
```

Defer in Loop Go

```go
package main
import "fmt"

func main() {
    for i := 0; i < 3; i++ {
        defer fmt.Println(i)
    }
}
```

Defer with Closure Go

```go
package main
import "fmt"

func main() {
    x := 10
    defer func() {
        fmt.Println(x)
    }()
    x = 20
}
```

Function as Parameter Go

```go
package main
import "fmt"

func apply(f func(int) int, x int)
int {
    return f(x)
}

func double(x int) int {
    return x * 2
}

func main() {
    result := apply(double, 5)
    fmt.Println(result)
}
```

Returning Functions Go

```go
package main
import "fmt"

func makeMultiplier(factor int)
func(int) int {
    return func(x int) int {
        return x * factor
    }
}

func main() {
    double := makeMultiplier(2)
    triple := makeMultiplier(3)
    fmt.Println(double(5))
    fmt.Println(triple(5))
}
```

Go

Blank Identifier in Function Signature

```go
package main
import "fmt"

func process(_ int, name string) {
    fmt.Println(name)
}

func main() {
    process(42, "Go")
}
```

Function Parameter Grouping Go

```go
package main
import "fmt"

func add(a, b, c int, x, y float64)
{
    fmt.Println(a + b + c)
    fmt.Println(x + y)
}

func main() {
    add(1, 2, 3, 4.5, 5.5)
}
```

Function Parameter Grouping Go

```go
package main
import "fmt"

func factorial(n int) int {
    if n <= 1 {
        return 1
    }
    return n * factorial(n-1)
}

func main() {
    fmt.Println(factorial(5))
}
```

Closure Modifying Outer Variable Go

```go
package main
import "fmt"

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

func main() {
    c := counter()
    fmt.Println(c())
    fmt.Println(c())
    fmt.Println(c())
}
```

Named Return Value Initialized to ... Go

```go
package main
import "fmt"

func getValues() (x int, y string, z bool) {
    // No assignments
    return
}

func main() {
    a, b, c := getValues()
    fmt.Printf("%v %q %v\n", a, b, c)
}
```

🎯 BONUS CHALLENGE Go

```go
package main
import "fmt"

func mystery() (result int) {
    x := 10

    defer func() {
        result += x
    }()

    defer func(val int) {
        result += val
    }(x)

    x = 20
    result = 5
    return
}

func main() {
    fmt.Println(mystery())
}
```

Recap Markdown

```
**Basic Functions:**
- Multiple return values (idiomatic for errors)
- Named return values with bare return
- Variadic parameters (`...type`)
- Functions as first-class values

**Function Types:**
- `type` can define function signatures
- Functions can be parameters
- Functions can be return values
- Enables callbacks and higher-order functions

**Closures:**
- Anonymous functions that capture variables
- Capture by reference, not value
- Can maintain state across calls
- Loop variable trap is famous gotcha

**Defer:**
- Postpones execution until function exit
- LIFO order for multiple defers
- Parameters evaluated immediately
- Closures capture current values
- Can modify named return values
- Don't overuse in loops!

**Common Patterns:**
- Error handling with multiple returns
- Callbacks with function parameters
- Function factories returning closures
- Stateful functions with closures
- Cleanup with defer

**Pitfalls to Avoid:**
- Loop variable closure trap
- Shadowing named returns
- Confusing defer parameter vs closure
- Defer in loops accumulating
```