

A photograph of a server room with rows of server racks illuminated by blue light. The racks are filled with server units, and the perspective shows a long aisle leading into the distance. The lighting is predominantly blue, with some yellow light from the server units themselves.

Introduction to Programming in Go

4. Basic Functions in Go

Module Topics

1. Function syntax
2. Multiple return values
3. Deferred execution
4. Recursion
5. Varadic functions
6. Error handling
7. Panics and recoveries

Function Syntax

Basic Function Syntax

1. Function form with unnamed return value:

```
func fname(var1 type1, ..varn typen) type { ... }
```

2. Function form with named return value:

```
func fname(var1 type1, ..varn typen) (var type) { ... }
```

3. Named return values are initialized to their zero values.
4. Named return values are used like normal variables in the function.
5. Unnamed values require an explicit **return** argument list.
6. Named return values only need a naked return statement.

Basic Function Form

```
// Example 04-01 Basic function sytnax
```

```
...
```

```
func square(x int) int {  
    y := x * x  
    return y  
}
```

```
func main() {  
    x := 4  
    retval := square(x)  
    fmt.Println("x = ", x, "square = ", retval)  
}
```

```
[Module04]$ go run ex04-01.go  
x = 4 square = 16
```

Named Return Value

```
// Example 04-02 Named Return Value
```

```
...
```

```
func square(x int) (result int) {  
    y := x * x  
    return  
}
```

```
func main() {  
    x := 4  
    fmt.Println("x = ", x, "square = ", square(x))  
}
```

```
[Module04]$ go run ex04-02.go  
x = 4 square = 16
```

Multiple Parameters

```
// Example 04-03 Multiple Parameters
```

```
...
```

```
func divides(x, y int) (div bool) {  
    div = (y % x) == 0  
    return div  
}
```

```
func main() {  
    x, y := 2, 21  
    fmt.Println(x, "|", y, " is ", divides(x, y))  
}
```

```
[Module04]$ go run ex04-03.go  
2 | 21 is false
```

Multiple Return Values

Multiple Return Values

1. Function form with multiple return values:

```
func fname(v1 t1...) (type1, type2, ..) { ... }
```

2. Function form with multiple named return values:

```
func fname(v1 t1...) (var1 type1, var2 type2) { ... }
```

3. Same rules apply as for single return values.

Multiple Return Values

```
// Example 04-04 Multiple return values
```

```
...
```

```
func divides(x, y int) (int, int) {  
    return (y/x), (y%x)  
}
```

```
func main() {  
    x, y := 3, 23  
    quote, rem := divides(x,y)  
    fmt.Println(x, "/", y, " is ", quot, "R", rem)  
}
```

```
[Module04]$ go run ex04-04.go  
3 / 23 is 7 R 2
```

Multiple Named Return Values

```
// Example 04-05 Multiple named return values
```

```
...
```

```
func divides(x, y int) (int q, int r) {  
    q = y / x  
    r = y % x  
    return  
}
```

```
func main() {  
    x, y := 3, 23  
    quote, rem := divides(x,y)  
    fmt.Println(x, "/", y, " is ", quot, "R", rem)  
}
```

```
[Module04]$ go run ex04-05.go  
3 / 23 is 7 R 2
```

Deferred Execution

Deferred Execution

1. Function execution is deferred using the `defer` operator:

```
defer fname(x,y)
```

2. Defers function execution until the calling function is about to exit.
3. Parameters are evaluated when called, not when executed.
4. Multiple functions can be deferred: executed in a LIFO order.
5. Often used for cleanup and recovery from panics.

Simple Defer

```
// Example 04-06 Deferred execution
...

func f(message string) {
    fmt.Println(message)
}

func main() {
    m := "before defer"
    defer f(m)
    m = "after defer"
    fmt.Println(m)
}
```

```
[Module04]$ go run ex04-06.go
after defer
before defer
```

Stacked Defer

```
// Example 04-07 Stacked defer
...
func f(k int) {
    fmt.Println("executing f(%d)\n", k)
}

func main() {
    for i := 1; i < 4; i++ {
        fmt.Printf("called f(%d)\n", i)
        defer f(i)
    }
    fmt.Println("end main")
}
```

```
[Module04]$ go run ex04-07.go
called f(1)
called f(2)
called f(3)
end main
executed f(3)
executed f(2)
executed f(1)
```

Recursion

Recursion

```
// Example 04-08 Recursion
...
func sum(k int) int {
    fmt.Println("executing sum(%d)\n", k)
    if k > 0 {
        return k + sum(k-1)
    } else {
        return 0
    }
}

func main() {
    fmt.Println(sum(3))
}
```

```
[Module04]$ go run ex04-08.go
executed sum(3)
executed sum(2)
executed sum(1)
executed sum(0)
6
```

Varadic Functions

Varadic Functions

1. Varadic functions take a variable number of parameters

```
func fname(msg string, x ... int)
```

2. Ellipsis indicates variable number of parameters.
3. Only the last parameter can use ellipsis.
4. Varadic parameters are loaded into an array of the parameter type.

Varadic Functions

```
// Example 04-09 Varadic functions
...
func addup(nums ... int) (sum int) {
    for _, val := range nums {
        sum += val
    }
    return
}

func main() {
    fmt.Println(sum(3))
}
```

```
[Module04]$ go run ex04-09.go
15
```

Error Handling

Error Handling

1. Go does not use exceptions to handle errors.
2. When a function fails, an error object is created and returned.
3. The error object must be the last return value in the return value list.
4. When no error occurs, the error object is nil.
5. When an error occurs, any return values are considered invalid.
6. Calling function then tests for the error occurrence.
7. Errors are not created automatically, the program has to create them.
8. A simpler form is the "comma ok" idiom with bools instead of errors.

Error Handling

```
// Example 04-10 Error Handling
...
import "errors"

func division(num, denom int) (int, error) {
    if denom == 0 {
        return 0, errors.New("Divide by zero")
    }
    return (num/denom), nil
}

func main() {
    res, e := division(56, 0)
    if e != nil {
        fmt.Println(e)
    } else {
        fmt.Println(res)
    }
}
```

```
[Module04]$ go run ex04-10.go
Divide by zero
```


Error Handling

```
// Example 04-11 Error Handling
...
import "errors"

func division(num, denom int) (int, error) {
    if denom == 0 {
        return 0, errors.New("Divide by zero")
    }
    return (num/denom), nil
}

func main() {
    res, e := division(56, 2)
    if e != nil {
        fmt.Println(e)
    } else {
        fmt.Println(res)
    }
}
```

```
[Module04]$ go run ex04-11.go
28
```

Comma OK Idiom

```
// Example 04-12 Comma OK Idiom
...
func division(num, denom int) (int, bool) {
    if denom == 0 {
        return 0, false
    }
    return (num/denom), true
}

func main() {
    res, ok := division(56, 2)
    if ok {
        fmt.Println(res)
    }
}
```

```
[Module04]$ go run ex04-12.go
28
```

Panics and Recoveries

Panics and Recoveries

1. A panic is generated by Go when a runtime error occurs.
2. During a panic, all deferred functions are called, then execution halts.
3. Panics can be generated by calling the `panic()` function.
4. Panics can be recovered from by executing the `recover()` function.
5. The `recover()` function only works in deferred functions.
6. Panics should be used only for critical problems, use errors otherwise.
7. The `recover()` function returns the panic item, nil otherwise.

Runtime Panic

```
// Example 04-13 Runtime Panic
```

```
...
```

```
func main() {  
    x, y := 1, 0  
    fmt.Println(x / y)  
}
```

```
[Module04]$ go run ex04-13.go  
panic: runtime error: integer divide by zero  
[signal 0x8 code=0x1 addr=0x40102c pc=0x40102c]  
(stacktrace omitted)  
exit status 2
```


Generated Panic

```
// Example 04-14 Generated Panic
...
func division(num, denom int) int {
    if denom == 0 {
        panic("Dividing by zero?!")
    }
    return (num/denom)
}

func main() {
    res := division(56, 0)
    fmt.Println(res)
}
```

```
[Module04]$ go run ex04-14.go
panic: Dividing by zero?!
(stacktrace omitted)
exit status 2
```

Panic and Recovery

```
// Example 04-15 Panic and Recovery
...
func rec() {
    r := recover()
    if r != nil {
        fmt.Println("recovered value = ", r)
    }
}

func main() {
    x, y := 1, 0
    defer rec()
    fmt.Println(x / y)
}
```

```
[Module04]$ go run ex04-15.go
recovered value = runtime error: integer divide by zero
```

Lab 4: Functions