MCA372D – Go Lang

Unit-1

Programming Fundamentals

Why Go? Variables, values & type – introduction to packages, short declaration operator, var keyword, exploring type, zero values, fmt package, creating your own type, conversion, not casting. Control flow – Understanding control flow, loop, conditional.

Go, also called Golang or Go language, is an Open Source programming language that Google developed. It is often referred to as Golang because of its former domain name, golang.org , but its proper name is Go. Software developers use Go in an array of operating systems and frameworks to develop web applications, cloud and networking services, and other types of software.Some Google cloud engineers began to feel that this large and complex codebase was slowing them down. So they decided that they needed a new programming language focused on simplicity and quick performance. Robert Griesemer, Rob Pike, and Ken Thompson designed Go.The goals of the Go project were to eliminate the slowness and clumsiness of software development at Google, and thereby to make the process more productive and scalable. The language was designed by and for people who write—and read and debug and maintain—large software systems.

Go is a compiled language, not interpreted language. Go speeds up the compilation process of Go code from the binary file directly to machine code without working through the virtual machine.Go has built-in support for concurrency.Golang excels in networking, web development and microservices which making it a great choice for projects that heavily rely on these features. The main reason for Golang's popularity is its speed. Many programmers have compared Golang to its competitors and found Golang to have the fastest compile time.

It is syntactically similar to C, but also has memory safety, garbage collection, structural typing, and CSP-style concurrency. Golang is generally faster than Java for certain types of applications, particularly those that require a lot of concurrencies. However, Java has better memory management, which can make it more suitable for applications such as financial systems, healthcare applications, e-commerce platforms, etc.

The bootstrapping process of the Go programming language, where the Go compiler is written in Go itself, is a testament to the language's simplicity, self-containment, and pragmatic design. It showcases the ingenuity of the Go team in creating a language that is not only powerful but also self-sustaining.Go has at least two compilers, gc and gccgo . The former was written in C, but is now written in Go itself. While the latter is a gcc frontend written mainly in C++. Go's libraries are written in Go.

Go's architecture emphasizes simplicity, performance, and concurrency, making it well-suited for building scalable and efficient software systems.Golang, has a simple and straightforward architecture. The key components can be summarised as below;

- Compiler: Go comes with a compiler suite that transforms Go source code into executable machine code. The compiler includes several tools, such as go build for building executables, go run for running Go programs directly, and go install for installing packages and executables.

- Runtime: Go has a lightweight runtime that manages things like memory allocation, garbage collection, and goroutines (concurrent execution units). The runtime is responsible for coordinating the execution of Go programs and providing essential services to them.

- Standard Library: Go ships with a comprehensive standard library that covers various aspects of programming, including I/O operations, networking, text processing, cryptography, and much more. This standard library is well-designed and optimized for performance.

- Goroutines: Goroutines are a core feature of Go's concurrency model. They are lightweight threads of execution managed by the Go runtime. Goroutines enable concurrent programming in Go, allowing developers to write highly concurrent and scalable applications easily.

- Channels: Channels are used for communication and synchronization between goroutines. They provide a safe and efficient way for goroutines to communicate and share data. Channels facilitate the creation of clean and expressive concurrent code in Go.

- Packages and Modules: Go promotes a modular approach to software development through packages and modules. Packages are units of encapsulation, while modules provide versioning and dependency management capabilities. The go toolchain supports fetching, building, and managing dependencies automatically.

- Interfaces and Type System: Go has a statically typed system with support for interfaces. Interfaces in Go allow for polymorphism and decoupling, enabling code reuse and flexible design. Go's type

system is simple yet powerful, providing a balance between expressiveness and safety.

- Tooling: Go comes with a rich set of tools for development, testing, and profiling. Tools like go fmt for formatting code, go test for running tests, go vet for static analysis, and go prof for profiling performance are all part of the Go ecosystem.

Note:

In Go, variables are stored in memory. The exact location and management of memory depend on several factors, including the type of variable, its scope, and whether it's allocated on the stack or the heap.

Stack: Variables with a fixed size and lifetime tied to the scope of a function are typically allocated on the stack. This includes basic types like integers, floats, and pointers.

Heap: Variables with dynamic lifetimes or sizes, such as slices and maps, are allocated on the heap. The heap is managed by Go's garbage collector, which automatically reclaims memory that's no longer needed.

Globals: Global variables are stored in a special region of memory and persist throughout the entire program's execution.

Closure Variables: Variables captured by closures are stored on the heap if the closure itself escapes the scope where it's defined.

Comments

Comments serve as program documentation. There are two forms:

1. *Line comments* start with the character sequence // and stop at the end of the line.
2. *General comments* start with the character sequence /* and stop with the first subsequent character sequence */.

A comment cannot start inside a rune or string literal, or inside a comment. A general comment containing no newlines acts like a space. Any other comment acts like a newline.

Tokens

Tokens form the vocabulary of the Go language. There are four classes: *identifiers*, *keywords*, *operators and punctuation*, and *literals*. *White space*, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token. Also, a newline or end of file may trigger the insertion of a semicolon. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

Semicolons

The formal syntax uses semicolons ";" as terminators in a number of productions. Go programs may omit most of these semicolons using the following two rules:

1. When the input is broken into tokens, a semicolon is automatically inserted into the token stream immediately after a line's final token if that token is
    - an identifier
    - an integer, floating-point, imaginary, rune, or string literal
    - one of the keywords break, continue, fallthrough, or return
    - one of the operators and punctuation ++, --,),], or }
2. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing ")" or "}".

To reflect idiomatic use, code examples in this document elide semicolons using these rules.

Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

identifier = letter { letter | unicode_digit } .

Keywords

The following keywords are reserved and may not be used as identifiers.

break default func interface select
case defer go map struct
chan else goto package switch
const fallthrough if range type
continue for import return var

Operators and punctuation

The following character sequences represent operators (including assignment operators) and punctuation

+ & += &= && == != ( )
- | -= |= || < <= [ ]
* ^ *= ^= <- > >= { }
/ << /= <<= ++ = := , ;
% >> %= >>= -- ! ... . :
&^ &^= ~

Basic format 'verbs' in GoLang : Refer https://pkg.go.dev/fmt#hdr-Printing

```
package main

import (
    "fmt"
)
```

```
func main() {
    // Integer formatting
    num := 42
    fmt.Printf("Decimal: %d\n", num)
    fmt.Printf("Binary: %b\n", num)
    fmt.Printf("Hexadecimal: %x\n", num)
    fmt.Printf("Octal: %o\n", num)

    // Floating point formatting
    f := 3.14159
    fmt.Printf("Normal float: %f\n", f)
    fmt.Printf("Exponential notation: %e\n", f)
    fmt.Printf("Shorter Exponential notation: %.2e\n", f)

    // String formatting
    name := "John"
    fmt.Printf("String: %s\n", name)

    // Boolean formatting
    isTrue := true
    fmt.Printf("Boolean: %t\n", isTrue)

    // Width and Precision
    width, precision := 8, 2
    fmt.Printf("Width and Precision: %*.*f\n", width, precision, f)

    // Padding
    fmt.Printf("Padding with zeros: %08d\n", num)
```

```
    // Printing data type
    fmt.Printf("Data type: %T\n", f)
}
```

Output

Binary: 101010

Hexadecimal: 2a

Octal: 52

Normal float: 3.141590

Exponential notation: 3.141590e+00

Shorter Exponential notation: 3.14e+00

String: John

Boolean: true

Width and Precision: 3.14

Padding with zeros: 00000042

Data type: float64

Math package : refer https://pkg.go.dev/math

```
package main

import (
    "fmt"
    "math"
)

func main() {
    x := 2.5
    y := 3.0

    fmt.Println("Absolute value of", x, "is", math.Abs(x))
    fmt.Println("Sine of", x, "is", math.Sin(x))
```

```
    fmt.Println("Cosine of", x, "is", math.Cos(x))
    fmt.Println("Tangent of", x, "is", math.Tan(x))
    fmt.Println("Arcsine of", x, "is", math.Asin(x))
    fmt.Println("Arccosine of", x, "is", math.Acos(x))
    fmt.Println("Arctangent of", x, "is", math.Atan(x))
    fmt.Println("Arctangent of", y, "/", x, "is", math.Atan2(y, x))
    fmt.Println(x, "raised to the power of", y, "is", math.Pow(x, y))
    fmt.Println("Natural logarithm of", x, "is", math.Log(x))
    fmt.Println("e raised to the power of", x, "is", math.Exp(x))
    fmt.Println("Floor of", x, "is", math.Floor(x))
    fmt.Println("Ceil of", x, "is", math.Ceil(x))
    fmt.Println("Minimum of", x, "and", y, "is", math.Min(x, y))
    fmt.Println("Maximum of", x, "and", y, "is", math.Max(x, y))
    fmt.Println("Square root of", x, "is", math.Sqrt(x))
    fmt.Println("Rounded value of", x, "is", math.Round(x))
    fmt.Println("Remainder of", x, "divided by", y, "is", math.Mod(x, y))
    fmt.Println("Copysign of", x, "with the sign of", y, "is", math.Copysign(x, y))
    fmt.Println("Is", x, "NaN?", math.IsNaN(x))
}
```

Output

```
Absolute value of 2.5 is 2.5
Sine of 2.5 is 0.5984721441039564
Cosine of 2.5 is -0.8011436155469337
Tangent of 2.5 is -0.7470222972386603
Arcsine of 2.5 is NaN
Arccosine of 2.5 is NaN
Arctangent of 2.5 is 1.1902899496825317
Arctangent of 3 / 2.5 is 0.8760580505981934
```

2.5 raised to the power of 3 is 15.625

Natural logarithm of 2.5 is 0.9162907318741551

e raised to the power of 2.5 is 12.182493960703473

Floor of 2.5 is 2

Ceil of 2.5 is 3

Minimum of 2.5 and 3 is 2.5

Maximum of 2.5 and 3 is 3

Square root of 2.5 is 1.5811388300841898

Rounded value of 2.5 is 3

Remainder of 2.5 divided by 3 is 2.5

Copysign of 2.5 with the sign of 3 is 2.5

Is 2.5 NaN? False

string Package: Refer https://pkg.go.dev/strings

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    str := "Hello, world!"

    // 1. Length of the string
    fmt.Println("Length of the string:", len(str))

    // 2. To uppercase
    fmt.Println("Uppercase:", strings.ToUpper(str))
```

```
    // 3. To lowercase
    fmt.Println("Lowercase:", strings.ToLower(str))

    // 4. Contains
    fmt.Println("Contains 'world'?", strings.Contains(str, "world"))

    // 5. Index of
    fmt.Println("Index of 'world':", strings.Index(str, "world"))

    // 6. Replace
    fmt.Println("Replace 'world' with 'Go':", strings.Replace(str, "world", "Go", -
1))

    // 7. Split
    words := strings.Split(str, ",")
    fmt.Println("Split by comma:", words)

    // 8. Join
    joined := strings.Join(words, "-")
    fmt.Println("Joined with hyphen:", joined)

    // 9. Trim space
    spaces := "    Hello, world!    "
    fmt.Println("Trimmed spaces:", strings.TrimSpace(spaces))

    // 10. Trim prefix
    fmt.Println("Trimmed prefix 'Hello':", strings.TrimPrefix(str, "Hello"))
```

```
    // 11. Trim suffix
    fmt.Println("Trimmed suffix 'world!' :", strings.TrimSuffix(str, "world!"))

    // 12. Repeat
    fmt.Println("Repeated 3 times:", strings.Repeat(str, 3))

    // 13. Compare
    str2 := "Hello, world!"
    fmt.Println("Compare:", strings.Compare(str, str2))

    // 14. EqualFold
    fmt.Println("EqualFold (ignoring case):", strings.EqualFold(str, "HELLO, WORLD!"))

    // 15. Title
    fmt.Println("Title case:", strings.Title(str))
}
```

```
Length of the string: 13
Uppercase: HELLO, WORLD!
Lowercase: hello, world!
Contains 'world'? true
Index of 'world': 7
Replace 'world' with 'Go': Hello, Go!
Split by comma: [Hello  world!]
Joined with hyphen: Hello- world!
Trimmed spaces: Hello, world!
Trimmed prefix 'Hello': , world!
```

Trimmed suffix 'world!' : Hello,

Repeated 3 times: Hello, world!Hello, world!Hello, world!

Compare: 0

EqualFold (ignoring case): true

Title case: Hello, World!

// Understand = , == and :=

```
package main

import "fmt"

func main() {
    // Using ==
    a := 5
    b := 5
    if a == b {
        fmt.Println("Using ==: a is equal to b")
    }

    // Using =
    x := 10
    y := 0
    y = x // Assigning the value of x to y
    fmt.Println("Using =: y =", y)

    // Using :=
    c := 42
    d := "hello"
```

```
    fmt.Println("Using :=: c =", c)
    fmt.Println("Using :=: d =", d)
}
```

Output

Using ==: a is equal to b

Using =: y = 10

Using :=: c = 42

Using :=: d = hello

/*

create a go program to demonstate the scope of a variable.

In this program:

globalVar is a global variable accessible throughout the package.

localMainVar is a local variable declared within the main() function and is only accessible within that function.

localVar is a local variable declared within the functionWithLocalVar() function and is only accessible within that function.

Attempting to access a variable outside of its scope will result in a compile-time error.*/

```
package main

import "fmt"

// Global variable
var globalVar = "I am a global variable"
```

```
func main() {
    // Local variable within the main function
    var localMainVar = "I am a local variable in the main function"
    fmt.Println(localMainVar) // Output: I am a local variable in the main function

    // Accessing global variable
    fmt.Println(globalVar) // Output: I am a global variable

    // Calling a function where a variable is declared
    functionWithLocalVar()
}

func functionWithLocalVar() {
    // Local variable within another function
    var localVar = "I am a local variable in another function"
    fmt.Println(localVar) // Output: I am a local variable in another function

    // Trying to access the localMainVar from main function would result in a compile-time error
    // fmt.Println(localMainVar) // Uncommenting this line would result in an error
}
```

Constants:There are boolean constants, rune constants, integer constants, floating-point constants, complex constants, and string constants. Rune, integer, floating-point, and complex constants are collectively called numeric constants.

Variables:A variable is a storage location for holding a value. The set of permissible values is determined by the variable's type.A variable's value is retrieved by referring to the variable in an expression; it is the most recent value assigned to the variable. If a variable has not yet been assigned a value, its value is the zero value for its type.

Types :A type determines a set of values together with operations and methods specific to those values. A type may be denoted by a type name, if it has one, which must be followed by type arguments if the type is generic. A type may also be specified using a type literal, which composes a type from existing types.

- Boolean types :A boolean type represents the set of Boolean truth values denoted by the predeclared constants true and false. The predeclared boolean type is bool; it is a defined type.
- Numeric types:An integer, floating-point, or complex type represents the set of integer, floating-point, or complex values, respectively. They are collectively called numeric types.
- String types: string type represents the set of string values. A string value is a (possibly empty) sequence of bytes. The number of bytes is called the length of the string and is never negative. Strings are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is string; it is a defined type.

The length of a string s can be discovered using the built-in function len. The length is a compile-time constant if the string is a constant. A string's bytes can be accessed by integer indices 0 through len(s)-1. It is illegal to take the address of such an element; if s[i] is the i'th byte of a string, &s[i] is invalid.

- Array types :An array is a numbered sequence of elements of a single type, called the element type. The number of elements is called the length of the array and is never negative.

The length is part of the array's type; it must evaluate to a non-negative constant representable by a value of type int. The length of array a can be discovered using the built-in function len. The elements can be addressed by integer indices 0 through len(a)-1. Array types are always one-dimensional but may be composed to form multi-dimensional types.

- Slice types: A slice is a descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The number of elements is called the length of the slice and is never negative. The value of an uninitialized slice is nil.The length of a slice s can be discovered by the built-in function len; unlike with arrays it may change during execution. The elements can be addressed by integer indices 0 through len(s)-1. The slice index of a given element may be less than the index of the same element in the underlying array.A slice, once initialized, is always associated with an underlying array that holds its elements. A slice therefore shares storage with its array and with other slices of the same array; by contrast, distinct arrays always represent distinct storage.

The array underlying a slice may extend past the end of the slice. The capacity is a measure of that extent: it is the sum of the length of the slice and the length of the array beyond the slice; a slice of length up to that capacity can be created by slicing a new one from the original slice. The capacity of a slice a can be discovered using the built-in function cap(a).

A new, initialized slice value for a given element type T may be made using the built-in function make, which takes a slice type and parameters specifying the

length and optionally the capacity. A slice created with make always allocates a new, hidden array to which the returned slice value refers. That is, executing
make([]T, length, capacity)
produces the same slice as allocating an array and slicing it, so these two expressions are equivalent:
make([]int, 50, 100)
new([100]int)[0:50]
Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. With arrays of arrays, the inner arrays are, by construction, always the same length; however with slices of slices (or arrays of slices), the inner lengths may vary dynamically. Moreover, the inner slices must be initialized individually.

- Struct types: A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (EmbeddedField). Within a struct, non-blank field names must be unique.

```
// An empty struct.
struct {}

// A struct with 6 fields.
struct {
        x, y int
        u float32
        _ float32  // padding
        A *[]int
        F func()
}
```

- Pointer types :A pointer type denotes the set of all pointers to variables of a given type, called the base type of the pointer. The value of an uninitialized pointer is nil.
- Function types:A function type denotes the set of all functions with the same parameter and result types. The value of an uninitialized variable of function type is nil.
- Interface types: An interface type defines a type set. A variable of interface type can store a value of any type that is in the type set of the interface. Such a type is said to implement the interface. The value of an uninitialized variable of interface type is nil.
- Map types : A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type. The value of an uninitialized map is nil.
- Channel types: A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. The value of an uninitialized channel is nil.

Terminating statements

A terminating statement interrupts the regular flow of control in a block. The following statements are terminating:

- A "return" or "goto" statement.
- A call to the built-in function panic.
- A block in which the statement list ends in a terminating statement.
- An "if" statement in which:
    - the "else" branch is present, and
    - both branches are terminating statements.
- A "for" statement in which:

    - there are no "break" statements referring to the "for" statement, and
    - the loop condition is absent, and
    - the "for" statement does not use a range clause.
- A "switch" statement in which:
    - there are no "break" statements referring to the "switch" statement,
    - there is a default case, and
    - the statement lists in each case, including the default, end in a terminating statement, or a possibly labeled "fallthrough" statement.
- A "select" statement in which:
    - there are no "break" statements referring to the "select" statement, and
    - the statement lists in each case, including the default if present, end in a terminating statement.
- A labeled statement labeling a terminating statement.

All other statements are not terminating.

If statements

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

IfStmt = "if" [SimpleStmt ";"] Expression Block ["else" (IfStmt | Block)] .

```
if x > max {
        x = max
}
```

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
if x := f(); x < y {
        return x
} else if x > z {
        return z
} else {
        return y
}
```

Switch statements

"Switch" statements provide multi-way execution. An expression or type is compared to the "cases" inside the "switch" to determine which branch to execute.

SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .

There are two forms: expression switches and type switches. In an expression switch, the cases contain expressions that are compared against the value of the switch expression. In a type switch, the cases contain types that are compared against the type of a specially annotated switch expression. The switch expression is evaluated exactly once in a switch statement.

For statements

A "for" statement specifies repeated execution of a block. There are three forms: The iteration may be controlled by a single condition, a "for" clause, or a "range" clause.

ForStmt = "for" [Condition | ForClause | RangeClause] Block .

Condition = Expression .

- **For statements with single condition**

In its simplest form, a "for" statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is

evaluated before each iteration. If the condition is absent, it is equivalent to the boolean value true.

```
for a < b {
        a *= 2
}
```

- **For statements with for clause**

A "for" statement with a ForClause is also controlled by its condition, but additionally it may specify an init and a post statement, such as an assignment, an increment or decrement statement. The init statement may be a short variable declaration, but the post statement must not.

```
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .
for i := 0; i < 10; i++ {
        f(i)
}
```

If non-empty, the init statement is executed once before evaluating the condition for the first iteration; the post statement is executed after each execution of the block (and only if the block was executed). Any element of the ForClause may be empty but the semicolons are required unless there is only a condition. If the condition is absent, it is equivalent to the boolean value true.

```
for cond { S() }    is the same as    for ; cond ; { S() }
for      { S() }    is the same as    for true     { S() }
```

Each iteration has its own separate declared variable (or variables) [Go 1.22]. The variable used by the first iteration is declared by the init statement. The variable used by each subsequent iteration is declared implicitly before

executing the post statement and initialized to the value of the previous iteration's variable at that moment.

```
var prints []func()
for i := 0; i < 5; i++ {
	prints = append(prints, func() { println(i) })
	i++
}
for _, p := range prints {
	p()
}
prints
```

```
1
3
5
```

Prior to [Go 1.22], iterations share one set of variables instead of having their own separate variables. In that case, the example above prints

```
6
6
6
```

- **For statements with range clause**

A "for" statement with a "range" clause iterates through all entries of an array, slice, string or map, values received on a channel, or integer values from zero to an upper limit [Go 1.22]. For each entry it assigns iteration values to corresponding iteration variables if present and then executes the block.

```
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .
```

The expression on the right in the "range" clause is called the range expression, its core type must be an array, pointer to an array, slice, string, map, channel permitting receive operations, or an integer. As with an assignment, if present the operands on the left must be addressable or map index expressions; they denote the iteration variables. If the range expression is a channel or integer, at most one iteration variable is permitted, otherwise there may be up to two. If the last iteration variable is the blank identifier, the range clause is equivalent to the same clause without that identifier.

The range expression x is evaluated once before beginning the loop, with one exception: if at most one iteration variable is present and len(x) is constant, the range expression is not evaluated.

Comments

Comments serve as program documentation. There are two forms:

3. *Line comments* start with the character sequence // and stop at the end of the line.
4. *General comments* start with the character sequence /* and stop with the first subsequent character sequence */.

A comment cannot start inside a rune or string literal, or inside a comment. A general comment containing no newlines acts like a space. Any other comment acts like a newline.

Tokens

Tokens form the vocabulary of the Go language. There are four classes: *identifiers*, *keywords*, *operators and punctuation*, and *literals*. *White space*, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token. Also, a newline or end of file may trigger the insertion of a semicolon. While breaking the input into

tokens, the next token is the longest sequence of characters that form a valid token.

Semicolons

The formal syntax uses semicolons ";" as terminators in a number of productions. Go programs may omit most of these semicolons using the following two rules:

3. When the input is broken into tokens, a semicolon is automatically inserted into the token stream immediately after a line's final token if that token is
    - an identifier
    - an integer, floating-point, imaginary, rune, or string literal
    - one of the keywords break, continue, fallthrough, or return
    - one of the operators and punctuation ++, --,),], or }
4. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing ")" or "}".

To reflect idiomatic use, code examples in this document elide semicolons using these rules.

Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

identifier = letter { letter | unicode_digit } .

Keywords

The following keywords are reserved and may not be used as identifiers.

break default func interface select
case defer go map struct
chan else goto package switch
const fallthrough if range type
continue for import return var

Operators and punctuation

The following character sequences represent operators (including assignment operators) and punctuation

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=          ~
```

Basic format 'verbs' in GoLang : Refer https://pkg.go.dev/fmt#hdr-Printing

```
package main

import (
    "fmt"
)

func main() {
    // Integer formatting
    num := 42
    fmt.Printf("Decimal: %d\n", num)
    fmt.Printf("Binary: %b\n", num)
    fmt.Printf("Hexadecimal: %x\n", num)
    fmt.Printf("Octal: %o\n", num)

    // Floating point formatting
    f := 3.14159
    fmt.Printf("Normal float: %f\n", f)
    fmt.Printf("Exponential notation: %e\n", f)
```

```
    fmt.Printf("Shorter Exponential notation: %.2e\n", f)

    // String formatting
    name := "John"
    fmt.Printf("String: %s\n", name)

    // Boolean formatting
    isTrue := true
    fmt.Printf("Boolean: %t\n", isTrue)

    // Width and Precision
    width, precision := 8, 2
    fmt.Printf("Width and Precision: %*.*f\n", width, precision, f)

    // Padding
    fmt.Printf("Padding with zeros: %08d\n", num)

    // Printing data type
    fmt.Printf("Data type: %T\n", f)
}
```

Output

```
Binary: 101010
Hexadecimal: 2a
Octal: 52
Normal float: 3.141590
Exponential notation: 3.141590e+00
Shorter Exponential notation: 3.14e+00
String: John
Boolean: true
```

Width and Precision: 3.14

Padding with zeros: 00000042

Data type: float64

Math package : refer https://pkg.go.dev/math

```
package main

import (
    "fmt"
    "math"
)

func main() {
    x := 2.5
    y := 3.0

    fmt.Println("Absolute value of", x, "is", math.Abs(x))
    fmt.Println("Sine of", x, "is", math.Sin(x))
    fmt.Println("Cosine of", x, "is", math.Cos(x))
    fmt.Println("Tangent of", x, "is", math.Tan(x))
    fmt.Println("Arcsine of", x, "is", math.Asin(x))
    fmt.Println("Arccosine of", x, "is", math.Acos(x))
    fmt.Println("Arctangent of", x, "is", math.Atan(x))
    fmt.Println("Arctangent of", y, "/", x, "is", math.Atan2(y, x))
    fmt.Println(x, "raised to the power of", y, "is", math.Pow(x, y))
    fmt.Println("Natural logarithm of", x, "is", math.Log(x))
    fmt.Println("e raised to the power of", x, "is", math.Exp(x))
    fmt.Println("Floor of", x, "is", math.Floor(x))
    fmt.Println("Ceil of", x, "is", math.Ceil(x))
    fmt.Println("Minimum of", x, "and", y, "is", math.Min(x, y))
```

```
    fmt.Println("Maximum of", x, "and", y, "is", math.Max(x, y))
    fmt.Println("Square root of", x, "is", math.Sqrt(x))
    fmt.Println("Rounded value of", x, "is", math.Round(x))
    fmt.Println("Remainder of", x, "divided by", y, "is", math.Mod(x, y))
    fmt.Println("Copysign of", x, "with the sign of", y, "is", math.Copysign(x, y))
    fmt.Println("Is", x, "NaN?", math.IsNaN(x))
}
```

Output

Absolute value of 2.5 is 2.5

Sine of 2.5 is 0.5984721441039564

Cosine of 2.5 is -0.8011436155469337

Tangent of 2.5 is -0.7470222972386603

Arcsine of 2.5 is NaN

Arccosine of 2.5 is NaN

Arctangent of 2.5 is 1.1902899496825317

Arctangent of 3 / 2.5 is 0.8760580505981934

2.5 raised to the power of 3 is 15.625

Natural logarithm of 2.5 is 0.9162907318741551

e raised to the power of 2.5 is 12.182493960703473

Floor of 2.5 is 2

Ceil of 2.5 is 3

Minimum of 2.5 and 3 is 2.5

Maximum of 2.5 and 3 is 3

Square root of 2.5 is 1.5811388300841898

Rounded value of 2.5 is 3

Remainder of 2.5 divided by 3 is 2.5

Copysign of 2.5 with the sign of 3 is 2.5

Is 2.5 NaN? False

string Package: Refer https://pkg.go.dev/strings

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    str := "Hello, world!"

    // 1. Length of the string
    fmt.Println("Length of the string:", len(str))

    // 2. To uppercase
    fmt.Println("Uppercase:", strings.ToUpper(str))

    // 3. To lowercase
    fmt.Println("Lowercase:", strings.ToLower(str))

    // 4. Contains
    fmt.Println("Contains 'world'?", strings.Contains(str, "world"))

    // 5. Index of
    fmt.Println("Index of 'world':", strings.Index(str, "world"))

    // 6. Replace
    fmt.Println("Replace 'world' with 'Go':", strings.Replace(str, "world", "Go", -
1))
```

```
    // 7. Split
    words := strings.Split(str, ",")
    fmt.Println("Split by comma:", words)

    // 8. Join
    joined := strings.Join(words, "-")
    fmt.Println("Joined with hyphen:", joined)

    // 9. Trim space
    spaces := "   Hello, world!   "
    fmt.Println("Trimmed spaces:", strings.TrimSpace(spaces))

    // 10. Trim prefix
    fmt.Println("Trimmed prefix 'Hello':", strings.TrimPrefix(str, "Hello"))

    // 11. Trim suffix
    fmt.Println("Trimmed suffix 'world!' :", strings.TrimSuffix(str, "world!"))

    // 12. Repeat
    fmt.Println("Repeated 3 times:", strings.Repeat(str, 3))

    // 13. Compare
    str2 := "Hello, world!"
    fmt.Println("Compare:", strings.Compare(str, str2))

    // 14. EqualFold
    fmt.Println("EqualFold (ignoring case):", strings.EqualFold(str, "HELLO,
WORLD!"))
```

```
    // 15. Title
    fmt.Println("Title case:", strings.Title(str))
}
```

Length of the string: 13
Uppercase: HELLO, WORLD!
Lowercase: hello, world!
Contains 'world'? true
Index of 'world': 7
Replace 'world' with 'Go': Hello, Go!
Split by comma: [Hello world!]
Joined with hyphen: Hello- world!
Trimmed spaces: Hello, world!
Trimmed prefix 'Hello': , world!
Trimmed suffix 'world!' : Hello,
Repeated 3 times: Hello, world!Hello, world!Hello, world!
Compare: 0
EqualFold (ignoring case): true
Title case: Hello, World!

// Understand = , == and :=

```
package main

import "fmt"

func main() {
```

```
    // Using ==
    a := 5
    b := 5
    if a == b {
        fmt.Println("Using ==: a is equal to b")
    }

    // Using =
    x := 10
    y := 0
    y = x // Assigning the value of x to y
    fmt.Println("Using =: y =", y)

    // Using :=
    c := 42
    d := "hello"
    fmt.Println("Using :=: c =", c)
    fmt.Println("Using :=: d =", d)
}
```

Output

Using ==: a is equal to b

Using =: y = 10

Using :=: c = 42

Using :=: d = hello

/*

create a go program to demonstate the scope of a variable.

In this program:
globalVar is a global variable accessible throughout the package.
localMainVar is a local variable declared within the main() function and is only accessible within that function.
localVar is a local variable declared within the functionWithLocalVar() function and is only accessible within that function.
Attempting to access a variable outside of its scope will result in a compile-time error.*/

```
package main

import "fmt"

// Global variable
var globalVar = "I am a global variable"

func main() {
    // Local variable within the main function
    var localMainVar = "I am a local variable in the main function"
    fmt.Println(localMainVar) // Output: I am a local variable in the main function

    // Accessing global variable
    fmt.Println(globalVar) // Output: I am a global variable

    // Calling a function where a variable is declared
    functionWithLocalVar()
}
```

```
func functionWithLocalVar() {
    // Local variable within another function
    var localVar = "I am a local variable in another function"
    fmt.Println(localVar) // Output: I am a local variable in another function

    // Trying to access the localMainVar from main function would result in a compile-time error
    // fmt.Println(localMainVar) // Uncommenting this line would result in an error
}
```

Constants:There are boolean constants, rune constants, integer constants, floating-point constants, complex constants, and string constants. Rune, integer, floating-point, and complex constants are collectively called numeric constants.
Variables:A variable is a storage location for holding a value. The set of permissible values is determined by the variable's type.A variable's value is retrieved by referring to the variable in an expression; it is the most recent value assigned to the variable. If a variable has not yet been assigned a value, its value is the zero value for its type.
Types :A type determines a set of values together with operations and methods specific to those values. A type may be denoted by a type name, if it has one, which must be followed by type arguments if the type is generic. A type may also be specified using a type literal, which composes a type from existing types.

- Boolean types :A boolean type represents the set of Boolean truth values denoted by the predeclared constants true and false. The predeclared boolean type is bool; it is a defined type.

- Numeric types:An integer, floating-point, or complex type represents the set of integer, floating-point, or complex values, respectively. They are collectively called numeric types.
- String types: string type represents the set of string values. A string value is a (possibly empty) sequence of bytes. The number of bytes is called the length of the string and is never negative. Strings are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is string; it is a defined type.

The length of a string s can be discovered using the built-in function len. The length is a compile-time constant if the string is a constant. A string's bytes can be accessed by integer indices 0 through len(s)-1. It is illegal to take the address of such an element; if s[i] is the i'th byte of a string, &s[i] is invalid.

- Array types :An array is a numbered sequence of elements of a single type, called the element type. The number of elements is called the length of the array and is never negative.

The length is part of the array's type; it must evaluate to a non-negative constant representable by a value of type int. The length of array a can be discovered using the built-in function len. The elements can be addressed by integer indices 0 through len(a)-1. Array types are always one-dimensional but may be composed to form multi-dimensional types.

- Slice types: A slice is a descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The number of elements is called the length of the slice and is never negative. The value of an uninitialized slice is nil.The length of a slice s can be discovered by the built-in function len; unlike with arrays it may change during execution. The elements can be addressed by integer

indices 0 through len(s)-1. The slice index of a given element may be less than the index of the same element in the underlying array.A slice, once initialized, is always associated with an underlying array that holds its elements. A slice therefore shares storage with its array and with other slices of the same array; by contrast, distinct arrays always represent distinct storage.

The array underlying a slice may extend past the end of the slice. The capacity is a measure of that extent: it is the sum of the length of the slice and the length of the array beyond the slice; a slice of length up to that capacity can be created by slicing a new one from the original slice. The capacity of a slice a can be discovered using the built-in function cap(a).

A new, initialized slice value for a given element type T may be made using the built-in function make, which takes a slice type and parameters specifying the length and optionally the capacity. A slice created with make always allocates a new, hidden array to which the returned slice value refers. That is, executing

make([]T, length, capacity)

produces the same slice as allocating an array and slicing it, so these two expressions are equivalent:

make([]int, 50, 100)

new([100]int)[0:50]

Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. With arrays of arrays, the inner arrays are, by construction, always the same length; however with slices of slices (or arrays of slices), the inner lengths may vary dynamically. Moreover, the inner slices must be initialized individually.

- Struct types: A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly

(IdentifierList) or implicitly (EmbeddedField). Within a struct, non-blank field names must be unique.

```
// An empty struct.
struct {}

// A struct with 6 fields.
struct {
        x, y int
        u float32
        _ float32  // padding
        A *[]int
        F func()
}
```

- Pointer types :A pointer type denotes the set of all pointers to variables of a given type, called the base type of the pointer. The value of an uninitialized pointer is nil.
- Function types:A function type denotes the set of all functions with the same parameter and result types. The value of an uninitialized variable of function type is nil.
- Interface types: An interface type defines a type set. A variable of interface type can store a value of any type that is in the type set of the interface. Such a type is said to implement the interface. The value of an uninitialized variable of interface type is nil.
- Map types : A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type. The value of an uninitialized map is nil.

- Channel types: A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. The value of an uninitialized channel is nil.

Terminating statements

A terminating statement interrupts the regular flow of control in a block. The following statements are terminating:

- A "return" or "goto" statement.
- A call to the built-in function panic.
- A block in which the statement list ends in a terminating statement.
- An "if" statement in which:
    - the "else" branch is present, and
    - both branches are terminating statements.
- A "for" statement in which:
    - there are no "break" statements referring to the "for" statement, and
    - the loop condition is absent, and
    - the "for" statement does not use a range clause.
- A "switch" statement in which:
    - there are no "break" statements referring to the "switch" statement,
    - there is a default case, and
    - the statement lists in each case, including the default, end in a terminating statement, or a possibly labeled "fallthrough" statement.
- A "select" statement in which:
    - there are no "break" statements referring to the "select" statement, and

    - the statement lists in each case, including the default if present, end in a terminating statement.
- A labeled statement labeling a terminating statement.

All other statements are not terminating.

If statements

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

IfStmt = "if" [SimpleStmt ";"] Expression Block ["else" (IfStmt | Block)] .

```
if x > max {
        x = max
}
```

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
if x := f(); x < y {
        return x
} else if x > z {
        return z
} else {
        return y
}
```

Switch statements

"Switch" statements provide multi-way execution. An expression or type is compared to the "cases" inside the "switch" to determine which branch to execute.

SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .

There are two forms: expression switches and type switches. In an expression switch, the cases contain expressions that are compared against the value of the switch expression. In a type switch, the cases contain types that are compared against the type of a specially annotated switch expression. The switch expression is evaluated exactly once in a switch statement.

For statements

A "for" statement specifies repeated execution of a block. There are three forms: The iteration may be controlled by a single condition, a "for" clause, or a "range" clause.

ForStmt = "for" [Condition | ForClause | RangeClause] Block .

Condition = Expression .

- **For statements with single condition**

In its simplest form, a "for" statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is evaluated before each iteration. If the condition is absent, it is equivalent to the boolean value true.

```
for a < b {
        a *= 2
}
```

- **For statements with for clause**

A "for" statement with a ForClause is also controlled by its condition, but additionally it may specify an init and a post statement, such as an assignment, an increment or decrement statement. The init statement may be a short variable declaration, but the post statement must not.

ForClause = [InitStmt] ";" [Condition] ";" [PostStmt] .

InitStmt = SimpleStmt .
PostStmt = SimpleStmt .

```
for i := 0; i < 10; i++ {
        f(i)
}
```

If non-empty, the init statement is executed once before evaluating the condition for the first iteration; the post statement is executed after each execution of the block (and only if the block was executed). Any element of the ForClause may be empty but the semicolons are required unless there is only a condition. If the condition is absent, it is equivalent to the boolean value true.

for cond { S() } is the same as for ; cond ; { S() }
for { S() } is the same as for true { S() }

Each iteration has its own separate declared variable (or variables) [Go 1.22]. The variable used by the first iteration is declared by the init statement. The variable used by each subsequent iteration is declared implicitly before executing the post statement and initialized to the value of the previous iteration's variable at that moment.

```
var prints []func()
for i := 0; i < 5; i++ {
        prints = append(prints, func() { println(i) })
        i++
}
for _, p := range prints {
        p()
}
prints
```

1

3

5

Prior to [Go 1.22], iterations share one set of variables instead of having their own separate variables. In that case, the example above prints

6

6

6

- **For statements with range clause**

A "for" statement with a "range" clause iterates through all entries of an array, slice, string or map, values received on a channel, or integer values from zero to an upper limit [Go 1.22]. For each entry it assigns iteration values to corresponding iteration variables if present and then executes the block.
RangeClause = [ExpressionList "=" | IdentifierList ":="] "range" Expression .
The expression on the right in the "range" clause is called the range expression, its core type must be an array, pointer to an array, slice, string, map, channel permitting receive operations, or an integer. As with an assignment, if present the operands on the left must be addressable or map index expressions; they denote the iteration variables. If the range expression is a channel or integer, at most one iteration variable is permitted, otherwise there may be up to two. If the last iteration variable is the blank identifier, the range clause is equivalent to the same clause without that identifier.
The range expression x is evaluated once before beginning the loop, with one exception: if at most one iteration variable is present and len(x) is constant, the range expression is not evaluated.

Packages

In Go, packages are a way to organize and reuse code. They provide a mechanism for grouping related Go source files together into a single unit,

making it easier to manage and maintain large codebases. Packages also facilitate code reuse by allowing developers to import and use functionality from other packages in their programs.

There are a significant number of built-in packages in Go, covering a wide range of functionalities. Some of the main built-in packages in Go along with their functionalities are mentioned below:

- **fmt** - Formatting package: Provides functions for formatted I/O operations like printing to the console, formatting strings, and scanning input.
- **os** - Operating system interface: Offers functions for interacting with the operating system, including file operations, environment variables, and process management.
- **io** - Input/output utilities: Defines interfaces for I/O operations and provides basic implementations. It includes functions for working with readers and writers.
- **net** - Network communications: Contains utilities for network programming, including support for TCP/IP, UDP, DNS resolution, and HTTP clients and servers.
- **http** - HTTP client and server: Provides functionality for building HTTP clients and servers. It includes support for handling HTTP requests and responses, routing, middleware, and more.
- **encoding/json** - JSON encoding and decoding: Offers functions for encoding Go data structures into JSON format and decoding JSON data into Go data structures.
- **time** - Time-related functions: Provides functionality for working with dates, times, durations, and time zones.

- **strconv** - String conversion utilities: Includes functions for converting strings to other data types like integers and floats, and vice versa.
- **math** - Mathematical functions: Contains basic mathematical functions like trigonometric functions, exponential and logarithmic functions, and constants like π and ε.
- **sync** - Synchronization primitives: Provides synchronization mechanisms such as mutexes, wait groups, and atomic operations for safe concurrent programming.

Naming conventions for packages and their contents:

- Package names should be short, descriptive, and lowercase. Avoid using underscores or mixed-case names.
- Package names should be singular, not plural.
- Function names, variable names, and other identifiers within a package should follow the same conventions as package names, but with **initial uppercase letters if they are intended to be exported** (i.e., accessible from outside the package).

For example, a package named "utils" might contain exported functions such as "FormatString" or "CalculateSum", along with non-exported functions or variables like "parseString" or "tempVar".

Creating a user-defined package

Creating a user-defined package involves creating a directory with Go source files and organizing them properly.

Suppose that we want to create a package called mathutils which provides some basic mathematical operations.

Create a directory for your package:
Create a directory named mathutils somewhere in your Go workspace.

Create Go source file(s) inside the directory:
In this case, we'll create a file named math.go inside the mathutils directory. This file will contain our package's functions.

Define your package inside the source file:
Let's define some basic mathematical functions inside math.go.

```
// mathutils/math.go

package mathutils

// Add returns the sum of two integers.
func Add(a, b int) int {
    return a + b
}

// Subtract returns the difference between two integers.
func Subtract(a, b int) int {
    return a - b
}
```

Use your package in another Go file:
Now, let's create another Go file outside of the mathutils directory where we'll import and use the mathutils package.

```
// main.go
package main
```

```
import (
    "fmt"
    "your_module_path/mathutils" // Import your package
)
func main() {
    sum := mathutils.Add(5, 3)
    difference := mathutils.Subtract(10, 4)

    fmt.Println("Sum:", sum)
    fmt.Println("Difference:", difference)
}
```

Build and run your program:
Now, navigate to the directory containing main.go and run your program:

```
go run main.go
```

This will output:
Sum: 8
Difference: 6

Short Declaration Operator

In Go, the short declaration operator := is a concise way to declare and initialize variables. It's particularly useful when you want to declare variables without specifying their types explicitly. This operator infers the type of the variable from the assigned value.

```
//Example
package main
import "fmt"
```

```
func main() {
    // Short declaration with type inference
    a := 10
    b := "hello"

    // Displaying values
    fmt.Println(a, b)
}
```

In this example, a is inferred as an int and b as a string.

Var Keyword

The var keyword in Go is used to declare variables explicitly, allowing you to specify their type if needed. Unlike the short declaration operator, var requires you to explicitly state the type.

```
//Example
package main
import "fmt"

func main() {
    // Explicit variable declaration with var keyword
    var a int
    var b string

    // Initializing values
    a = 10
    b = "hello"

    // Displaying values
    fmt.Println(a, b)
```

```
}
```

In this example, a is declared as an int and b as a string.

Exploring Type

In Go, you can explore the type of a variable or an expression using the reflect package. This package allows you to examine the type of variables at runtime.

```
//Example
package main
import (
    "fmt"
    "reflect"
)
func main() {
    var a int
    a = 10

    // Using reflect.TypeOf() to get the type
    fmt.Println(reflect.TypeOf(a)) // Output: int
}
```

In this example, reflect.TypeOf(a) returns the type of a, which is int.

Zero Values

In Go, when a variable is declared but not explicitly initialized, it's assigned a default value known as its "zero value." The zero value depends on the type of the variable. Here are some examples:

For numeric types (like int, float32, float64, etc.), the zero value is 0.
For strings, the zero value is an empty string "".
For boolean types, the zero value is false.

For pointers, the zero value is nil.

Here's an example illustrating zero values:

```
package main

import "fmt"

func main() {
    var a int
    var b float64
    var c string
    var d bool
    var e *int

    // Displaying zero values
    fmt.Println(a) // Output: 0
    fmt.Println(b) // Output: 0
    fmt.Println(c) // Output:
    fmt.Println(d) // Output: false
    fmt.Println(e) // Output: <nil>
}
```

In this example, variables a, b, and d are assigned their respective zero values, and c is assigned an empty string. Variable e, being a pointer, is assigned nil.

Creating Your Own Type

Syntax:

```
type MyType underlyingType
```

MyType: The name of your custom type.

underlyingType: The existing type that your custom type will be based on.

Example:

type Celsius float64

In this example, Celsius is a custom type based on the float64 type, representing temperatures in Celsius.

Conversion (Not Casting)

Conversion vs. Casting:

Conversion in Go involves changing the type of a value to another type. It's done explicitly and follows Go's type system rules.

Casting, commonly seen in languages like C or C++, involves forcefully interpreting the bits of a value as if it belonged to another type. This is not idiomatic in Go.

Syntax:

variableOfTypeT := T(expressionOfTypeT)

T: Target type.

expressionOfTypeT: Expression of the source type T.

Example:

```
func main() {
    var f Fahrenheit = 32
    var c Celsius

    // Convert Fahrenheit to Celsius
    c = Celsius((f - 32) * 5 / 9)

    fmt.Println("Temperature in Celsius:", c)
}
```

In this example, f is converted to Celsius using a conversion. It ensures type safety and clarity.

Note:

Safety: Go's conversions ensure type safety, reducing runtime errors.

Clarity: Explicit conversions enhance code readability and maintainability.

Avoid Casting: Casting is discouraged in Go due to its potential for runtime errors and loss of type safety.

Unit-2

Grouping Data

Array. Slice - composite literal, for range, slicing a slice, append to a slice, delete from a slice, make, multi-dimensional slice. Map - introduction, add element & range, delete. Struct – introduction, embedded structs, anonymous structs.

Go Array

Composite Literal

Arrays in Go are fixed-size sequences of elements of the same type.

Composite literals allow the creation of arrays with predefined values.

```
arr := [3]int{1, 2, 3} // Array of size 3 with predefined values
```

For Range

The for range loop allows iteration over each element of an array.

```
for index, value := range arr {
    // Access index and value
}
```

Slicing a Slice

Slicing creates a new slice by specifying a range of indices from an existing slice.

slice := arr[start:end] // Slice from start index (inclusive) to end index (exclusive)

Append to a Slice

append() function adds elements to the end of a slice and returns a new slice.

slice = append(slice, value)

Delete from a Slice

There's no direct built-in method to delete elements from a slice. You can use append and slicing to achieve this.

slice = append(slice[:index], slice[index+1:]...)

Make

make() function is used to create slices dynamically with a specified length and capacity.

slice := make([]int, length, capacity)

Multi-dimensional Slice

Slices can be multi-dimensional by creating slices of slices.

matrix := [][]int{{1, 2}, {3, 4}} // 2D slice

Go Map

Introduction

Maps in Go are unordered collections of key-value pairs. Keys must be of a comparable type.

Add Element & Range

Elements can be added to a map using the key.

m := make(map[keyType]valueType)

m[key] = value

range can be used to iterate over keys and values.

for key, value := range m {

```
    // Access key and value
}
```

Delete

delete() function is used to remove an element from the map.

delete(m, key)

Go Struct

Introduction

Structs allow the creation of custom data types composed of fields.

Fields can be of any type, including other structs.

Embedded Structs

Structs can be embedded within another struct.

```
type InnerStruct struct {
    Field1 int
    Field2 string
}

type OuterStruct struct {
    InnerStruct
    Field3 float64
}
```

Anonymous Structs

Anonymous structs are defined without a name and are useful for temporary data structures.

```
anon := struct {
    Field1 int
    Field2 string
}{
```

```
    Field1: 10,
    Field2: "hello",
}
```

Unit-3

Functions:

Introduction, variadic parameter, Defer, Panic, Methods, Interfaces & polymorphism, Anonymous function, returning a function, recursion. Error handling – introduction Recover, Errors with info.

Functions in Go (Golang)

Declaration

Functions in Go are declared using the func keyword followed by the function name, parameter list, and return type (if any).

Example:

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

Parameters

Parameters are defined inside the parentheses following the function name.

Parameters can have both a name and a type.

Multiple parameters are separated by commas.

Example:

```
func greet(name string) {
    fmt.Println("Hello,", name)
}
```

Return Values

Functions can return one or more values.

Return types are specified after the parameter list.

Example:

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

Multiple Return Values

Go supports returning multiple values from a function.

Example:

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

Anonymous Functions

Functions can be declared anonymously.

Useful in scenarios like defining functions as arguments to other functions or for concurrent execution.

Example:

```
func() {
    fmt.Println("Anonymous function")
}()
```

Function as Types

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

Example:

```
type MathFunc func(int, int) int
```

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

func main() {
    var mf MathFunc
    mf = add
    fmt.Println(mf(2, 3)) // Output: 5
}
```

Variadic Functions

Functions can accept a variable number of arguments.

Denoted by ... before the type in the parameter list.

Example:

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

Defer, Panic, and Recover

Go has built-in functions defer, panic, and recover for handling exceptional conditions.

defer postpones the execution of a function until the surrounding function returns.

panic is used to terminate a function if it encounters an unrecoverable error.

recover is used to regain control of a panicking goroutine.

Example:

```
func main() {
```

```
    defer fmt.Println("Deferred")
    fmt.Println("Hello")
    panic("Panic")
}
```

Closures

Functions can be defined within other functions, forming closures.

Closures have access to variables declared in the parent function.

Example:

```
func outer() func() {
    count := 0
    return func() {
        count++
        fmt.Println("Count:", count)
    }
}

func main() {
    counter := outer()
    counter() // Output: Count: 1
    counter() // Output: Count: 2
}
```

Methods

Methods are functions associated with a particular type.

They allow adding behaviors to user-defined types.

Example:

```
type Rectangle struct {
    width, height float64
}
```

```
func (r Rectangle) Area() float64 {
    return r.width * r.height
}

func main() {
    rect := Rectangle{width: 10, height: 5}
    fmt.Println("Area:", rect.Area()) // Output: 50
}
```

Function Literals

Function literals are anonymous functions that can form closures.

They are useful for defining functions inline, especially for concurrent operations.

Example:

```
func main() {
    add := func(a, b int) int {
        return a + b
    }
    fmt.Println(add(3, 4)) // Output: 7
}
```

Recursion

Go supports recursive functions.

A recursive function calls itself with modified arguments to solve smaller instances of the same problem.

Example:

```
func factorial(n int) int {
    if n <= 1 {
        return 1
```

```
    }
    return n * factorial(n-1)
}
```

A function is a group of statements that exist within a program for the purpose of performing a specific task. At a high level, a function takes an input and returns an output. Function allows you to extract commonly used block of code into a single component.
The single most popular Go function is main(), which is used in every independent Go program.

Creating a Function in Golang

A declaration begins with the func keyword, followed by the name you want the function to have, a pair of parentheses (), and then a block containing the function's code. The following example has a function with the name SimpleFunction. It takes no parameter and returns no values.

Example

```
package main
import "fmt"
// SimpleFunction prints a message
func SimpleFunction() {
        fmt.Println("Hello World")
}

func main() {
        SimpleFunction()
}
```

Output

Hello World

Simple function with parameters in Golang

Information can be passed to functions through arguments. An argument is just like a variable. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with two arguments of int type. When the add() function is called, we pass two integer values (e.g. 20,30).

If the functions with names that start with an uppercase letter will be exported to other packages. If the function name starts with a lowercase letter, it won't be exported to other packages, but you can call this function within the same package.

Example

```
package main

import "fmt"

// Function accepting arguments
func add(x int, y int) {
	total := 0
	total = x + y
	fmt.Println(total)
}

func main() {
	// Passing arguments
```

```
	add(20, 30)
}
```

Output

50

Simple function with return value in Golang

In this example, the add() function takes input of two integer numbers and returns an integer value with a name of total. Note the return statement is required when a return value is declared as part of the function's signature. The types of input and return value must match with function signature. If we will modify the above program and pass some string value in argument then program will throw an exception "cannot use "test" (type string) as type int in argument to add".

Example

```
package main

import "fmt"

// Function with int as return type
func add(x int, y int) int {
	total := 0
	total = x + y
	return total
}

func main() {
	// Accepting return value in varaible
	sum := add(20, 30)
```

```
	fmt.Println(sum)
}
```

User Defined Function Types in Golang

Golang also support to define our own function types.

The modified version of above program with function types as below:

Example

```
package main
import "fmt"

type First func(int) int
type Second func(int) First

func squareSum(x int) Second {
	return func(y int) First {
		return func(z int) int {
			return x*x + y*y + z*z
		}
	}
}

func main() {
	// 5*5 + 6*6 + 7*7
	fmt.Println(squareSum(5)(6)(7))
}
```

Golang Passing Address to a Function

Passing the address of variable to the function and the value of a variables modified using dereferencing inside body of function.

Example

```
package main
import "fmt"

func update(a *int, t *string) {
        *a = *a + 5      // defrencing pointer address
        *t = *t + " Doe" // defrencing pointer address
        return
}

func main() {
        var age = 20
        var text = "John"
        fmt.Println("Before:", text, age)

        update(&age, &text)

        fmt.Println("After :", text, age)
}
```

Output

```
Before: John 20
After : John Doe 25
```

Golang Functions Returning Multiple Values

Functions in Golang can return multiple values, which is a helpful feature in many practical scenarios. This example declares a function with two return values and calls it from a main function.

Example

```
package main
```

```
import "fmt"
func rectangle(l int, b int) (area int, parameter int) {
        parameter = 2 * (l + b)
        area = l * b
        return // Return statement without specify variable name
}

func main() {
        var a, p int
        a, p = rectangle(20, 30)
        fmt.Println("Area:", a)
        fmt.Println("Parameter:", p)
}
```

Anonymous Functions in Golang

An anonymous function is a function that was declared without any named identifier to refer to it. Anonymous functions can accept inputs and return outputs, just as standard functions do.

Assigning function to the variable.

Anonymous functions can be used for containing functionality that need not be named and possibly for short-term use.

Example

```
package main
import "fmt"
var (
        area = func(l int, b int) int {
                return l * b
```

```
        }
)

func main() {
        fmt.Println(area(20, 30))
}
```

Example

Passing arguments to anonymous functions.

```
package main
import "fmt"
func main() {
        func(l int, b int) {
                fmt.Println(l * b)
        }(20, 30)
}
```

Example

Function defined to accept a parameter and return value.

```
package main
import "fmt"
func main() {
        fmt.Printf(
                "100 (°F) = %.2f (°C)\n",
                func(f float64) float64 {
                        return (f - 32.0) * (5.0 / 9.0)
                }(100),
        )
}
```

Explain with few cases the application of anonymous functions in Go.

In Go (Golang), anonymous functions are functions defined without a name. They are also known as function literals or lambda functions. The below mentioned are just a few examples of how anonymous functions can be used in Go to enhance the flexibility and expressiveness of the code. They are particularly useful in situations where you need to create functions on the fly or where functions need to operate within a specific context.

The primary use of anonymous functions in Go includes:

- Passing functions as arguments: Anonymous functions are commonly used when you want to pass a function as an argument to another function. This is often seen in scenarios like sorting, filtering, or mapping data.

```
package main
import "fmt"
func main() {
    numbers := []int{1, 2, 3, 4, 5}
        // Using anonymous function to filter even numbers
    even := filter(numbers, func(n int) bool {
        return n%2 == 0
    })
        fmt.Println(even) // Output: [2 4]
}
// filter applies the given function f to each element of numbers
// and returns a new slice containing only the elements for which f returns true.
func filter(numbers []int, f func(int) bool) []int {
    var result []int
    for _, num := range numbers {
        if f(num) {
```

```
            result = append(result, num)
        }
    }
    return result
}
```

- Creating closures: Anonymous functions are often used to create closures, which capture and retain the surrounding state. This is useful for creating functions that depend on some external context.

```
package main
import "fmt"
func main() {
    message := "Hello"

    // Anonymous function capturing message variable
    greet := func() {
        fmt.Println(message)
    }

    // Invoking the anonymous function
    greet() // Output: Hello
}
```

- As a function return value: Anonymous functions can also be returned by other functions, allowing for dynamic creation of functions.

```
package main
import "fmt"
func main() {
    add := adder(10)
    fmt.Println(add(5)) // Output: 15
}
```

```
// adder returns a function which adds x to the argument passed to it.
func adder(x int) func(int) int {
    return func(y int) int {
        return x + y
    }
}
```

Refer: https://go.dev/doc/tutorial/handle-errors

Importance of Error Handling in Go

Error handling helps programmers create robust and maintainable applications by ensuring that errors are identified and addressed systematically. Go takes a unique approach to error handling by eschewing exception-based mechanisms and instead using a simple yet powerful error interface. This approach encourages developers to handle errors explicitly and minimizes the risk of unhandled errors propagating through the application.

The error Interface in Go

In Go, errors are represented by the error interface, which is a built-in interface with a single method, Error() string. Any type that implements this method can be used as an error. Here's the definition of the error interface:

```
type error interface {
    Error() string
}
```

You can create custom error types by implementing the error interface, or you can use the standard errors package to create simple error values. Here's an example of creating an error using the errors.New() function:

```
package main
import (
    "errors"
    "fmt"
```

```
)
func main() {
    err := errors.New("an error occurred")
    fmt.Println(err)
}
```

Common Error Handling Patterns

- **Returning Errors**

In Go, it's a common pattern to return errors as the last value in the function signature. If a function can return an error, its signature should include an error as the last return value.

Here's an example of a function that returns an error:

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

- **Handling Errors**

When calling a function that returns an error, it's essential to check the error and handle it appropriately. The idiomatic way to do this in Go is to use the if err != nil pattern:

```
result, err := divide(10, 0)
if err != nil {
    fmt.Println("Error:", err)
} else {
    fmt.Println("Result:", result)
}
```

- **Wrapping Errors**

Sometimes, you may want to add additional context to an error before returning it to the caller. You can do this using the fmt.Errorf() function, which allows you to create a formatted error message and wrap the original error:

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

Use Custom Error Types

Using custom error types allows you to provide additional information about the error and makes it easier for the caller to handle specific error cases. Here's an example of a custom error type:

```
type ValidationError struct {
    Field string
    Msg   string
}

func (e ValidationError) Error() string {
    return fmt.Sprintf("%s: %s", e.Field, e.Msg)
}
```

Use Error Sentinels

Error sentinels are predefined error values that can be used to represent specific error cases. They allow the caller to compare errors directly using the ==

operator, making error handling more explicit and easier to understand. Here's an example of an error sentinel:

```
var ErrDivisionByZero = errors.New("division by zero is not allowed")
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, ErrDivisionByZero
    }
    return a / b, nil
}
```

Then, you can check for this specific error when handling the error:

```
result, err := divide(10, 0)
if err == ErrDivisionByZero {
    fmt.Println("Error: Division by zero")
} else if err != nil {
    fmt.Println("Error:", err)
} else {
    fmt.Println("Result:", result)
}
```

Use the errors.Is() and errors.As() Functions

Go 1.13 introduced the errors.Is() and errors.As() functions to make it easier to work with wrapped errors. The errors.Is() function checks if a specific error is present in the error chain, while the errors.As() function allows you to extract a specific error type from the error chain.

Here’s an example using both errors.Is() and errors.As():

```
package main
import (
    "errors"
    "fmt"
```

```
    "os"
)

func main() {
    err := processFile("nonexistent.txt")
    if errors.Is(err, os.ErrNotExist) {
        fmt.Println("File does not exist")
    } else if err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println("File processed successfully")
    }
    var validationErr *ValidationError
    if errors.As(err, &validationErr) {
        fmt.Printf("Validation error: field=%s, message=%s\n",
validationErr.Field, validationErr.Msg)
    }
```