Go

Go was created out of the need to get work done. It’s not the latest trend in computer science, but it is the newest fastest way to solve real-world problems.

It has familiar concepts of imperative languages with static typing. It’s fast to compile and fast to execute, it adds easy-to-understand concurrency to leverage today’s multi-core CPUs, and has features to help with large-scale programming.

Go comes with a great standard library and an enthusiastic community.

```go
// Single line comment
/* Multi-
 line comment */

// A package clause starts every source file.
// Main is a special name declaring an executable rather than a library.
package main

// Import declaration declares library packages referenced in this file.
import (
    "fmt"       // A package in the Go standard library.
    "io/ioutil" // Implements some I/O utility functions.
    m "math"    // Math library with local alias m.
    "net/http"  // Yes, a web server!
    "strconv"   // String conversions.
)

// A function definition. Main is special. It is the entry point for the
// executable program. Love it or hate it, Go uses brace brackets.
func main() {
    // Println outputs a line to stdout.
    // Qualify it with the package name, fmt.
    fmt.Println("Hello world!")

    // Call another function within this package.
    beyondHello()
}

// Functions have parameters in parentheses.
// If there are no parameters, empty parentheses are still required.
func beyondHello() {
    var x int // Variable declaration. Variables must be declared before use.
    x = 3     // Variable assignment.
    // "Short" declarations use := to infer the type, declare, and assign.
    y := 4
    sum, prod := learnMultiple(x, y)        // Function returns two values.
    fmt.Println("sum:", sum, "prod:", prod) // Simple output.
    learnTypes()                            // < y minutes, learn more!
}

/* <- multiline comment
Functions can have parameters and (multiple!) return values.
Here `x`, `y` are the arguments and `sum`, `prod` is the signature (what's returned).
Note that `x` and `sum` receive the type `int`.
```

```go
*/
func learnMultiple(x, y int) (sum, prod int) {
    return x + y, x * y // Return two values.
}

// Some built-in types and literals.
func learnTypes() {
    // Short declaration usually gives you what you want.
    str := "Learn Go!" // string type.

    s2 := `A "raw" string literal
can include line breaks.` // Same string type.

    // Non-ASCII literal. Go source is UTF-8.
    g := 'Σ' // rune type, an alias for int32, holds a unicode code point.

    f := 3.14195 // float64, an IEEE-754 64-bit floating point number.
    c := 3 + 4i  // complex128, represented internally with two float64's.

    // var syntax with initializers.
    var u uint = 7 // Unsigned, but implementation dependent size as with int.
    var pi float32 = 22. / 7

    // Conversion syntax with a short declaration.
    n := byte('\n') // byte is an alias for uint8.

    // Arrays have size fixed at compile time.
    var a4 [4]int           // An array of 4 ints, initialized to all 0.
    a3 := [...]int{3, 1, 5} // An array initialized with a fixed size of three
    // elements, with values 3, 1, and 5.

    // Slices have dynamic size. Arrays and slices each have advantages
    // but use cases for slices are much more common.
    s3 := []int{4, 5, 9}    // Compare to a3. No ellipsis here.
    s4 := make([]int, 4)    // Allocates slice of 4 ints, initialized to all 0.
    var d2 [][]float64      // Declaration only, nothing allocated here.
    bs := []byte("a slice") // Type conversion syntax.

    // Because they are dynamic, slices can be appended to on-demand.
    // To append elements to a slice, the built-in append() function is used.
    // First argument is a slice to which we are appending. Commonly,
    // the array variable is updated in place, as in example below.
    s := []int{1, 2, 3}     // Result is a slice of length 3.
    s = append(s, 4, 5, 6)  // Added 3 elements. Slice now has length of 6.
    fmt.Println(s) // Updated slice is now [1 2 3 4 5 6]

    // To append another slice, instead of list of atomic elements we can
    // pass a reference to a slice or a slice literal like this, with a
    // trailing ellipsis, meaning take a slice and unpack its elements,
    // appending them to slice s.
    s = append(s, []int{7, 8, 9}...) // Second argument is a slice literal.
    fmt.Println(s)  // Updated slice is now [1 2 3 4 5 6 7 8 9]

    p, q := learnMemory() // Declares p, q to be type pointer to int.
    fmt.Println(*p, *q)   // * follows a pointer. This prints two ints.

    // Maps are a dynamically growable associative array type, like the
    // hash or dictionary types of some other languages.
    m := map[string]int{"three": 3, "four": 4}
    m["one"] = 1

    // Unused variables are an error in Go.
    // The underscore lets you "use" a variable but discard its value.
    _, _, _, _, _, _, _, _, _, _ = str, s2, g, f, u, pi, n, a3, s4, bs
    // Output of course counts as using a variable.
    fmt.Println(s, c, a4, s3, d2, m)

    learnFlowControl() // Back in the flow.
}

// It is possible, unlike in many other languages for functions in go
// to have named return values.
// Assigning a name to the type being returned in the function declaration line
// allows us to easily return from multiple points in a function as well as to
// only use the return keyword, without anything further.
func learnNamedReturns(x, y int) (z int) {
    z = x * y
    return // z is implicit here, because we named it earlier.
}

// Go is fully garbage collected. It has pointers but no pointer arithmetic.
// You can make a mistake with a nil pointer, but not by incrementing a pointer.
func learnMemory() (p, q *int) {
    // Named return values p and q have type pointer to int.
    p = new(int) // Built-in function new allocates memory.
    // The allocated int is initialized to 0, p is no longer nil.
    s := make([]int, 20) // Allocate 20 ints as a single block of memory.
    s[3] = 7             // Assign one of them.
    r := -2              // Declare another local variable.
    return &s[3], &r     // & takes the address of an object.
}

func expensiveComputation() float64 {
    return m.Exp(10)
}

func learnFlowControl() {
    // If statements require brace brackets, and do not require parentheses.
    if true {
        fmt.Println("told ya")
    }
    // Formatting is standardized by the command line command "go fmt."
    if false {
        // Pout.
    } else {
        // Gloat.
    }
    // Use switch in preference to chained if statements.
```

```go
x := 42.0
switch x {
case 0:
case 1:
case 42:
    // Cases don't "fall through".
    /*
    There is a `fallthrough` keyword however, see:
      https://github.com/golang/go/wiki/Switch#fall-through
    */
case 43:
    // Unreached.
default:
    // Default case is optional.
}
// Like if, for doesn't use parens either.
// Variables declared in for and if are local to their scope.
for x := 0; x < 3; x++ { // ++ is a statement.
    fmt.Println("iteration", x)
}
// x == 42 here.

// For is the only loop statement in Go, but it has alternate forms.
for { // Infinite loop.
    break    // Just kidding.
    continue // Unreached.
}

// You can use range to iterate over an array, a slice, a string, a map, or a channel.
// range returns one (channel) or two values (array, slice, string and map).
for key, value := range map[string]int{"one": 1, "two": 2, "three": 3} {
    // for each pair in the map, print key and value
    fmt.Printf("key=%s, value=%d\n", key, value)
}

// As with for, := in an if statement means to declare and assign
// y first, then test y > x.
if y := expensiveComputation(); y > x {
    x = y
}
// Function literals are closures.
xBig := func() bool {
    return x > 10000 // References x declared above switch statement.
}
fmt.Println("xBig:", xBig()) // true (we last assigned e^10 to x).
x = 1.3e3                    // This makes x == 1300
fmt.Println("xBig:", xBig()) // false now.

// What's more is function literals may be defined and called inline,
// acting as an argument to function, as long as:
// a) function literal is called immediately (),
// b) result type matches expected type of argument.
fmt.Println("Add + double two numbers: ",
    func(a, b int) int {
            return (a + b) * 2
        }(10, 2)) // Called with args 10 and 2
    // => Add + double two numbers: 24

    // When you need it, you'll love it.
    goto love
love:

    learnFunctionFactory() // func returning func is fun(3)(3)
    learnDefer()      // A quick detour to an important keyword.
    learnInterfaces() // Good stuff coming up!
}

func learnFunctionFactory() {
    // Next two are equivalent, with second being more practical
    fmt.Println(sentenceFactory("summer")("A beautiful", "day!"))

    d := sentenceFactory("summer")
    fmt.Println(d("A beautiful", "day!"))
    fmt.Println(d("A lazy", "afternoon!"))
}

// Decorators are common in other languages. Same can be done in Go
// with function literals that accept arguments.
func sentenceFactory(mystring string) func(before, after string) string {
    return func(before, after string) string {
        return fmt.Sprintf("%s %s %s", before, mystring, after) // new string
    }
}

func learnDefer() (ok bool) {
    // Deferred statements are executed just before the function returns.
    defer fmt.Println("deferred statements execute in reverse (LIFO) order.")
    defer fmt.Println("\nThis line is being printed first because")
    // Defer is commonly used to close a file, so the function closing the
    // file stays close to the function opening the file.
    return true
}

// Define Stringer as an interface type with one method, String.
type Stringer interface {
    String() string
}

// Define pair as a struct with two fields, ints named x and y.
type pair struct {
    x, y int
}

// Define a method on type pair. Pair now implements Stringer.
func (p pair) String() string { // p is called the "receiver"
    // Sprintf is another public function in package fmt.
    // Dot syntax references fields of p.
    return fmt.Sprintf("(%d, %d)", p.x, p.y)
```

```
}

func learnInterfaces() {
    // Brace syntax is a "struct literal". It evaluates to an initialized
    // struct. The := syntax declares and initializes p to this struct.
    p := pair{3, 4}
    fmt.Println(p.String()) // Call String method of p, of type pair.
    var i Stringer          // Declare i of interface type Stringer.
    i = p                   // Valid because pair implements Stringer
    // Call String method of i, of type Stringer. Output same as above.
    fmt.Println(i.String())

    // Functions in the fmt package call the String method to ask an object
    // for a printable representation of itself.
    fmt.Println(p) // Output same as above. Println calls String method.
    fmt.Println(i) // Output same as above.

    learnVariadicParams("great", "learning", "here!")
}

// Functions can have variadic parameters.
func learnVariadicParams(myStrings ...interface{}) {
    // Iterate each value of the variadic.
    // The underbar here is ignoring the index argument of the array.
    for _, param := range myStrings {
        fmt.Println("param:", param)
    }

    // Pass variadic value as a variadic parameter.
    fmt.Println("params:", fmt.Sprintln(myStrings...))

    learnErrorHandling()
}

func learnErrorHandling() {
    // ", ok" idiom used to tell if something worked or not.
    m := map[int]string{3: "three", 4: "four"}
    if x, ok := m[1]; !ok { // ok will be false because 1 is not in the map.
        fmt.Println("no one there")
    } else {
        fmt.Print(x) // x would be the value, if it were in the map.
    }
    // An error value communicates not just "ok" but more about the problem.
    if _, err := strconv.Atoi("non-int"); err != nil { // _ discards value
        // prints 'strconv.ParseInt: parsing "non-int": invalid syntax'
        fmt.Println(err)
    }
    // We'll revisit interfaces a little later. Meanwhile,
    learnConcurrency()
}

// c is a channel, a concurrency-safe communication object.
func inc(i int, c chan int) {
    c <- i + 1 // <- is the "send" operator when a channel appears on the left.
```

```go
}

// We'll use inc to increment some numbers concurrently.
func learnConcurrency() {
    // Same make function used earlier to make a slice. Make allocates and
    // initializes slices, maps, and channels.
    c := make(chan int)
    // Start three concurrent goroutines. Numbers will be incremented
    // concurrently, perhaps in parallel if the machine is capable and
    // properly configured. All three send to the same channel.
    go inc(0, c) // go is a statement that starts a new goroutine.
    go inc(10, c)
    go inc(-805, c)
    // Read three results from the channel and print them out.
    // There is no telling in what order the results will arrive!
    fmt.Println(<-c, <-c, <-c) // channel on right, <- is "receive" operator.

    cs := make(chan string)       // Another channel, this one handles strings.
    ccs := make(chan chan string) // A channel of string channels.
    go func() { c <- 84 }()       // Start a new goroutine just to send a value.
    go func() { cs <- "wordy" }() // Again, for cs this time.
    // Select has syntax like a switch statement but each case involves
    // a channel operation. It selects a case at random out of the cases
    // that are ready to communicate.
    select {
    case i := <-c: // The value received can be assigned to a variable,
        fmt.Printf("it's a %T", i)
    case <-cs: // or the value received can be discarded.
        fmt.Println("it's a string")
    case <-ccs: // Empty channel, not ready for communication.
        fmt.Println("didn't happen.")
    }
    // At this point a value was taken from either c or cs. One of the two
    // goroutines started above has completed, the other will remain blocked.

    learnWebProgramming() // Go does it. You want to do it too.
}

// A single function from package http starts a web server.
func learnWebProgramming() {

    // First parameter of ListenAndServe is TCP address to listen to.
    // Second parameter is an interface, specifically http.Handler.
    go func() {
        err := http.ListenAndServe(":8080", pair{})
        fmt.Println(err) // don't ignore errors
    }()

    requestServer()
}

// Make pair an http.Handler by implementing its only method, ServeHTTP.
func (p pair) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // Serve data with a method of http.ResponseWriter.
    w.Write([]byte("You learned Go in Y minutes!"))
}

func requestServer() {
    resp, err := http.Get("http://localhost:8080")
    fmt.Println(err)
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    fmt.Printf("\nWebserver said: `%s`", string(body))
}
```

Further Reading

The root of all things Go is the official Go web site. There you can follow the tutorial, play interactively, and read lots. Aside from a tour, the docs contain information on how to write clean and effective Go code, package and command docs, and release history.

The language definition itself is highly recommended. It's easy to read and amazingly short (as language definitions go these days.)

You can play around with the code on Go playground. Try to change it and run it from your browser! Note that you can use https://play.golang.org as a REPL to test things and code in your browser, without even installing Go.

On the reading list for students of Go is the source code to the standard library. Comprehensively documented, it demonstrates the best of readable and understandable Go, Go style, and Go idioms. Or you can click on a function name in the documentation and the source code comes up!

Another great resource to learn Go is Go by example.

Go Mobile adds support for mobile platforms (Android and iOS). You can write all-Go native mobile apps or write a library that contains bindings from a Go package, which can be invoked via Java (Android) and Objective-C (iOS). Check out the Go Mobile page for more information.