

Go by Example

Go is an open source programming language designed for building scalable, secure and reliable software. Please read the [official documentation](#) to learn more.

Go by Example is a hands-on introduction to Go using annotated example programs. Check out the [first example](#) or browse the full list below.

Unless stated otherwise, examples here assume the latest major release Go and may use new language features. Try to upgrade to the latest version if something isn't working.

- [Hello World](#)
- [Values](#)
- [Variables](#)
- [Constants](#)
- [For](#)
- [If/Else](#)
- [Switch](#)
- [Arrays](#)
- [Slices](#)
- [Maps](#)
- [Functions](#)
- [Multiple Return Values](#)
- [Variadic Functions](#)
- [Closures](#)
- [Recursion](#)
- [Range over Built-in Types](#)
- [Pointers](#)
- [Strings and Runes](#)
- [Structs](#)
- [Methods](#)
- [Interfaces](#)
- [Enums](#)
- [Struct Embedding](#)
- [Generics](#)
- [Range over Iterators](#)
- [Errors](#)
- [Custom Errors](#)
- [Goroutines](#)
- [Channels](#)
- [Channel Buffering](#)
- [Channel Synchronization](#)
- [Channel Directions](#)
- [Select](#)
- [Timeouts](#)
- [Non-Blocking Channel Operations](#)
- [Closing Channels](#)
- [Range over Channels](#)
- [Timers](#)
- [Tickers](#)
- [Worker Pools](#)
- [WaitGroups](#)
- [Rate Limiting](#)
- [Atomic Counters](#)
- [Mutexes](#)
- [Stateful Goroutines](#)
- [Sorting](#)
- [Sorting by Functions](#)
- [Panic](#)
- [Defer](#)
- [Recover](#)
- [String Functions](#)
- [String Formatting](#)
- [Text Templates](#)

[Regular Expressions](#)
[JSON](#)
[XML](#)
[Time](#)
[Epoch](#)
[Time Formatting / Parsing](#)
[Random Numbers](#)
[Number Parsing](#)
[URL Parsing](#)
[SHA256 Hashes](#)
[Base64 Encoding](#)
[Reading Files](#)
[Writing Files](#)
[Line Filters](#)
[File Paths](#)
[Directories](#)
[Temporary Files and Directories](#)
[Embed Directive](#)
[Testing and Benchmarking](#)
[Command-Line Arguments](#)
[Command-Line Flags](#)
[Command-Line Subcommands](#)
[Environment Variables](#)
[Logging](#)
[HTTP Client](#)
[HTTP Server](#)
[Context](#)
[Spawning Processes](#)
[Exec'ing Processes](#)
[Signals](#)
[Exit](#)

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Switch

Switch statements express conditionals across many branches.

Here's a basic switch.

You can use commas to separate multiple expressions in the same case statement. We use the optional default case in this example as well.

switch without an expression is an alternate way to express if/else logic. Here we also show how the case expressions can be non-constants.

A type switch compares types instead of values. You can use this to discover the type of an interface value. In this example, the variable `t` will have the type corresponding to its clause.

```
package main

import (
    "fmt"
    "time"
)

func main() {

    i := 2
    fmt.Print("Write ", i, " as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("It's the weekend")
    default:
        fmt.Println("It's a weekday")
    }

    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("It's before noon")
    default:
        fmt.Println("It's after noon")
    }

    whatAmI := func(i interface{}) {
        switch t := i.(type) {
        case bool:
            fmt.Println("I'm a bool")
        case int:
            fmt.Println("I'm an int")
        default:
            fmt.Printf("Don't know type %T\n", t)
        }
    }
    whatAmI(true)
    whatAmI(1)
    whatAmI("hey")
}
```

```
$ go run switch.go
Write 2 as two
It's a weekday
It's after noon
I'm a bool
I'm an int
Don't know type string
```

Next example: [Arrays](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Worker Pools

In this example we'll look at how to implement a *worker pool* using goroutines and channels.

Here's the worker, of which we'll run several concurrent instances. These workers will receive work on the jobs channel and send the corresponding results on results. We'll sleep a second per job to simulate an expensive task.

In order to use our pool of workers we need to send them work and collect their results. We make 2 channels for this.

This starts up 3 workers, initially blocked because there are no jobs yet.

Here we send 5 jobs and then close that channel to indicate that's all the work we have.

Finally we collect all the results of the work. This also ensures that the worker goroutines have finished. An alternative way to wait for multiple goroutines is to use a [WaitGroup](#).

Our running program shows the 5 jobs being executed by various workers. The program only takes about 2 seconds despite doing about 5 seconds of total work because there are 3 workers operating concurrently.

```
package main

import (
    "fmt"
    "time"
)

func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "started job", j)
        time.Sleep(time.Second)
        fmt.Println("worker", id, "finished job", j)
        results <- j * 2
    }
}

func main() {

    const numJobs = 5
    jobs := make(chan int, numJobs)
    results := make(chan int, numJobs)

    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }

    for j := 1; j <= numJobs; j++ {
        jobs <- j
    }
    close(jobs)

    for a := 1; a <= numJobs; a++ {
        <-results
    }
}
```

```
$ time go run worker-pools.go
worker 1 started job 1
worker 2 started job 2
worker 3 started job 3
worker 1 finished job 1
worker 1 started job 4
worker 2 finished job 2
worker 2 started job 5
worker 3 finished job 3
worker 1 finished job 4
worker 2 finished job 5

real    0m2.358s
```

Next example: [WaitGroups](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

This is the site of **Mark McGranaghan**, engineer and entrepreneur.

I'm currently working with the engineering team at [Stainless](#).

Previously I co-founded [Muse](#), was a principal investigator with the industrial research lab [Ink & Switch](#), led engineering teams at [Stripe](#) and [Heroku](#), and built [Go by Example](#).

2019-12 [Thoughts on Recruiting](#)

2019-10 [CloudFront Analytics](#)

2019-08 [Alt VC Math](#)

2019-03 [Lessons from Stripe](#)

∞ [Interests](#)

∞ [Contact](#)

∞ [Colophon](#)

Go by Example: Closures

Go supports *anonymous functions*, which can form *closures*. Anonymous functions are useful when you want to define a function inline without having to name it.

This function `intSeq` returns another function, which we define anonymously in the body of `intSeq`. The returned function *closes over* the variable `i` to form a closure.

We call `intSeq`, assigning the result (a function) to `nextInt`. This function value captures its own `i` value, which will be updated each time we call `nextInt`.

See the effect of the closure by calling `nextInt` a few times.

To confirm that the state is unique to that particular function, create and test a new one.

```
package main

import "fmt"

func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}

func main() {

    nextInt := intSeq()

    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    newInts := intSeq()
    fmt.Println(newInts())
}
```

```
$ go run closures.go
1
2
3
1
```

The last feature of functions we'll look at for now is recursion.

Next example: [Recursion](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Regular Expressions

Go offers built-in support for [regular expressions](#). Here are some examples of common regexp-related tasks in Go.

This tests whether a pattern matches a string.

Above we used a string pattern directly, but for other regexp tasks you'll need to Compile an optimized Regexp struct.

Many methods are available on these structs. Here's a match test like we saw earlier.

This finds the match for the regexp.

This also finds the first match but returns the start and end indexes for the match instead of the matching text.

The Submatch variants include information about both the whole-pattern matches and the submatches within those matches. For example this will return information for both `p([a-z]+)ch` and `([a-z]+)`.

Similarly this will return information about the indexes of matches and submatches.

The All variants of these functions apply to all matches in the input, not just the first. For example to find all matches for a regexp.

These All variants are available for the other functions we saw above as well.

Providing a non-negative integer as the second argument to these functions will limit the number of matches.

Our examples above had string arguments and used names like `MatchString`. We can also provide `[]byte` arguments and drop `String` from the function name.

When creating global variables with regular expressions you can use the `MustCompile` variation of `Compile`. `MustCompile` panics instead of returning an error, which makes it safer to use for global variables.

The `regexp` package can also be used to replace subsets of strings with other values.

The `Func` variant allows you to transform matched text with a given function.

```
package main

import (
    "bytes"
    "fmt"
    "regexp"
)

func main() {

    match, _ := regexp.MatchString("p([a-z]+)ch", "peach")
    fmt.Println(match)

    r, _ := regexp.Compile("p([a-z]+)ch")

    fmt.Println(r.MatchString("peach"))

    fmt.Println(r.FindString("peach punch"))

    fmt.Println("idx:", r.FindStringIndex("peach punch"))

    fmt.Println(r.FindStringSubmatch("peach punch"))

    fmt.Println(r.FindStringSubmatchIndex("peach punch"))

    fmt.Println(r.FindAllString("peach punch pinch", -1))

    fmt.Println("all:", r.FindAllStringSubmatchIndex(
        "peach punch pinch", -1))

    fmt.Println(r.FindAllString("peach punch pinch", 2))

    fmt.Println(r.Match([]byte("peach")))

    r = regexp.MustCompile("p([a-z]+)ch")
    fmt.Println("regexp:", r)

    fmt.Println(r.ReplaceAllString("a peach", "<fruit>"))

    in := []byte("a peach")
    out := r.ReplaceAllFunc(in, bytes.ToUpper)
    fmt.Println(string(out))
}
```

```
$ go run regular-expressions.go
```

```
true
true
peach
idx: [0 5]
[peach ea]
[0 5 1 3]
[peach punch pinch]
all: [[0 5 1 3] [6 11 7 9] [12 17 13 15]]
[peach punch]
true
regexp: p([a-z]+)ch
a <fruit>
a PEACH
```

For a complete reference on Go regular expressions check the [regexp](#) package docs.

Next example: [JSON](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Closing Channels

Closing a channel indicates that no more values will be sent on it. This can be useful to communicate completion to the channel's receivers.

In this example we'll use a jobs channel to communicate work to be done from the main() goroutine to a worker goroutine. When we have no more jobs for the worker we'll close the jobs channel.

Here's the worker goroutine. It repeatedly receives from jobs with `j, more := <-jobs`. In this special 2-value form of receive, the `more` value will be `false` if jobs has been closed and all values in the channel have already been received. We use this to notify on done when we've worked all our jobs.

This sends 3 jobs to the worker over the jobs channel, then closes it.

We await the worker using the [synchronization](#) approach we saw earlier.

Reading from a closed channel succeeds immediately, returning the zero value of the underlying type. The optional second return value is `true` if the value received was delivered by a successful send operation to the channel, or `false` if it was a zero value generated because the channel is closed and empty.

The idea of closed channels leads naturally to our next example: range over channels.

Next example: [Range over Channels](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import "fmt"

func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)

    go func() {
        for {
            j, more := <-jobs
            if more {
                fmt.Println("received job", j)
            } else {
                fmt.Println("received all jobs")
                done <- true
                return
            }
        }
    }()

    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job", j)
    }
    close(jobs)
    fmt.Println("sent all jobs")

    <-done

    _, ok := <-jobs
    fmt.Println("received more jobs:", ok)
}
```

```
$ go run closing-channels.go
sent job 1
received job 1
sent job 2
received job 2
sent job 3
received job 3
sent all jobs
received all jobs
received more jobs: false
```

Go by Example: Range over Channels

In a [previous](#) example we saw how `for` and `range` provide iteration over basic data structures. We can also use this syntax to iterate over values received from a channel.

We'll iterate over 2 values in the queue channel.

This `range` iterates over each element as it's received from `queue`. Because we closed the channel above, the iteration terminates after receiving the 2 elements.

This example also showed that it's possible to close a non-empty channel but still have the remaining values be received.

Next example: [Timers](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import "fmt"

func main() {

    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)

    for elem := range queue {
        fmt.Println(elem)
    }
}
```

```
$ go run range-over-channels.go
one
two
```

Go by Example: SHA256 Hashes

[*SHA256 hashes*](#) are frequently used to compute short identities for binary or text blobs. For example, TLS/SSL certificates use SHA256 to compute a certificate's signature. Here's how to compute SHA256 hashes in Go.

Go implements several hash functions in various `crypto/*` packages.

Here we start with a new hash.

`Write` expects bytes. If you have a string `s`, use `[]byte(s)` to coerce it to bytes.

This gets the finalized hash result as a byte slice. The argument to `Sum` can be used to append to an existing byte slice: it usually isn't needed.

Running the program computes the hash and prints it in a human-readable hex format.

You can compute other hashes using a similar pattern to the one shown above. For example, to compute SHA512 hashes import `crypto/sha512` and use `sha512.New()`.

Note that if you need cryptographically secure hashes, you should carefully research [hash strength](#)!

Next example: [Base64 Encoding](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "crypto/sha256"
    "fmt"
)

func main() {
    s := "sha256 this string"

    h := sha256.New()

    h.Write([]byte(s))

    bs := h.Sum(nil)

    fmt.Println(s)
    fmt.Printf("%x\n", bs)
}
```

```
$ go run sha256-hashes.go
sha256 this string
1af1dfa857bf1d8814fe1af8983c18080019922e557f15a8a...
```

Go by Example: Embed Directive

//go:embed is a [compiler directive](#) that allows programs to include arbitrary files and folders in the Go binary at build time. Read more about the embed directive [here](#).

Import the embed package; if you don't use any exported identifiers from this package, you can do a blank import with `_ "embed"`.

embed directives accept paths relative to the directory containing the Go source file. This directive embeds the contents of the file into the string variable immediately following it.

Or embed the contents of the file into a `[]byte`.

We can also embed multiple files or even folders with wildcards. This uses a variable of the [embed.FS type](#), which implements a simple virtual file system.

Print out the contents of `single_file.txt`.

Retrieve some files from the embedded folder.

Use these commands to run the example. (Note: due to limitation on go playground, this example can only be run on your local machine.)

```
package main
```

```
import (  
    "embed"  
)
```

```
//go:embed folder/single_file.txt  
var fileString string
```

```
//go:embed folder/single_file.txt  
var fileByte []byte
```

```
//go:embed folder/single_file.txt  
//go:embed folder/*.hash  
var folder embed.FS
```

```
func main() {  
  
    print(fileString)  
    print(string(fileByte))  
  
    content1, _ := folder.ReadFile("folder/file1.hash")  
    print(string(content1))  
  
    content2, _ := folder.ReadFile("folder/file2.hash")  
    print(string(content2))  
}
```

```
$ mkdir -p folder  
$ echo "hello go" > folder/single_file.txt  
$ echo "123" > folder/file1.hash  
$ echo "456" > folder/file2.hash
```

```
$ go run embed-directive.go  
hello go  
hello go  
123  
456
```

Next example: [Testing and Benchmarking](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Exec'ing Processes

In the previous example we looked at [spawning external processes](#). We do this when we need an external process accessible to a running Go process. Sometimes we just want to completely replace the current Go process with another (perhaps non-Go) one. To do this we'll use Go's implementation of the classic [exec](#) function.

For our example we'll exec `ls`. Go requires an absolute path to the binary we want to execute, so we'll use `exec.LookPath` to find it (probably `/bin/ls`).

Exec requires arguments in slice form (as opposed to one big string). We'll give `ls` a few common arguments. Note that the first argument should be the program name.

Exec also needs a set of [environment variables](#) to use. Here we just provide our current environment.

Here's the actual `syscall.Exec` call. If this call is successful, the execution of our process will end here and be replaced by the `/bin/ls -a -l -h` process. If there is an error we'll get a return value.

When we run our program it is replaced by `ls`.

Note that Go does not offer a classic Unix `fork` function. Usually this isn't an issue though, since starting goroutines, spawning processes, and exec'ing processes covers most use cases for `fork`.

Next example: [Signals](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "os"
    "os/exec"
    "syscall"
)

func main() {

    binary, lookErr := exec.LookPath("ls")
    if lookErr != nil {
        panic(lookErr)
    }

    args := []string{"ls", "-a", "-l", "-h"}

    env := os.Environ()

    execErr := syscall.Exec(binary, args, env)
    if execErr != nil {
        panic(execErr)
    }
}
```

```
$ go run execing-processes.go
total 16
drwxr-xr-x  4 mark 136B Oct 3 16:29 .
drwxr-xr-x 91 mark 3.0K Oct 3 12:50 ..
-rw-r--r--  1 mark 1.3K Oct 3 16:28 execing-processes.go
```

Go by Example: WaitGroups

To wait for multiple goroutines to finish, we can use a *wait group*.

This is the function we'll run in every goroutine.

Sleep to simulate an expensive task.

This WaitGroup is used to wait for all the goroutines launched here to finish. Note: if a WaitGroup is explicitly passed into functions, it should be done *by pointer*.

Launch several goroutines and increment the WaitGroup counter for each.

Wrap the worker call in a closure that makes sure to tell the WaitGroup that this worker is done. This way the worker itself does not have to be aware of the concurrency primitives involved in its execution.

Block until the WaitGroup counter goes back to 0; all the workers notified they're done.

Note that this approach has no straightforward way to propagate errors from workers. For more advanced use cases, consider using the [errgroup package](#).

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int) {
    fmt.Printf("Worker %d starting\n", id)

    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}

func main() {

    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        wg.Add(1)

        go func() {
            defer wg.Done()
            worker(i)
        }()

        wg.Wait()
    }
```

```
$ go run waitgroups.go
Worker 5 starting
Worker 3 starting
Worker 4 starting
Worker 1 starting
Worker 2 starting
Worker 4 done
Worker 1 done
Worker 2 done
Worker 5 done
Worker 3 done
```

The order of workers starting up and finishing is likely to be different for each invocation.

Next example: [Rate Limiting](#).

Go by Example: Non-Blocking Channel Operations

Basic sends and receives on channels are blocking. However, we can use `select` with a default clause to implement *non-blocking* sends, receives, and even non-blocking multi-way selects.

Here's a non-blocking receive. If a value is available on `messages` then `select` will take the `<-messages` case with that value. If not it will immediately take the default case.

A non-blocking send works similarly. Here `msg` cannot be sent to the `messages` channel, because the channel has no buffer and there is no receiver. Therefore the default case is selected.

We can use multiple cases above the default clause to implement a multi-way non-blocking select. Here we attempt non-blocking receives on both `messages` and `signals`.

```
package main

import "fmt"

func main() {
    messages := make(chan string)
    signals := make(chan bool)

    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    default:
        fmt.Println("no message received")
    }

    msg := "hi"
    select {
    case messages <- msg:
        fmt.Println("sent message", msg)
    default:
        fmt.Println("no message sent")
    }

    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    case sig := <-signals:
        fmt.Println("received signal", sig)
    default:
        fmt.Println("no activity")
    }
}
```

```
$ go run non-blocking-channel-operations.go
no message received
no message sent
no activity
```

Next example: [Closing Channels](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Random Numbers

Go's `math/rand/v2` package provides [pseudorandom number](#) generation.

For example, `rand.IntN` returns a random int `n`, $0 \leq n < 100$.

`rand.Float64` returns a float64 `f`, $0.0 \leq f < 1.0$.

This can be used to generate random floats in other ranges, for example $5.0 \leq f' < 10.0$.

If you want a known seed, create a new `rand.Source` and pass it into the `New` constructor. `NewPCG` creates a new [PCG](#) source that requires a seed of two uint64 numbers.

Some of the generated numbers may be different when you run the sample.

See the [math/rand/v2](#) package docs for references on other random quantities that Go can provide.

Next example: [Number Parsing](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "math/rand/v2"
)

func main() {
    fmt.Print(rand.IntN(100), ",")
    fmt.Print(rand.IntN(100))
    fmt.Println()

    fmt.Println(rand.Float64())

    fmt.Print((rand.Float64()*5)+5, ",")
    fmt.Print((rand.Float64() * 5) + 5)
    fmt.Println()

    s2 := rand.NewPCG(42, 1024)
    r2 := rand.New(s2)
    fmt.Print(r2.IntN(100), ",")
    fmt.Print(r2.IntN(100))
    fmt.Println()

    s3 := rand.NewPCG(42, 1024)
    r3 := rand.New(s3)
    fmt.Print(r3.IntN(100), ",")
    fmt.Print(r3.IntN(100))
    fmt.Println()
}
```

```
$ go run random-numbers.go
68,56
0.8090228139659177
5.840125017402497,6.937056298890035
94,49
94,49
```


Go by Example: Enums

Enumerated types (enums) are a special case of [sum types](#). An enum is a type that has a fixed number of possible values, each with a distinct name. Go doesn't have an enum type as a distinct language feature, but enums are simple to implement using existing language idioms.

Our enum type `ServerState` has an underlying `int` type.

The possible values for `ServerState` are defined as constants. The special keyword `iota` generates successive constant values automatically; in this case 0, 1, 2 and so on.

By implementing the [fmt.Stringer](#) interface, values of `ServerState` can be printed out or converted to strings.

This can get cumbersome if there are many possible values. In such cases the [stringer tool](#) can be used in conjunction with `go:generate` to automate the process. See [this post](#) for a longer explanation.

If we have a value of type `int`, we cannot pass it to `transition` - the compiler will complain about type mismatch. This provides some degree of compile-time type safety for enums.

`transition` emulates a state transition for a server; it takes the existing state and returns a new state.

Suppose we check some predicates here to determine the next state...

```
package main

import "fmt"

type ServerState int

const (
    StateIdle ServerState = iota
    StateConnected
    StateError
    StateRetrying
)

var stateName = map[ServerState]string{
    StateIdle:    "idle",
    StateConnected: "connected",
    StateError:    "error",
    StateRetrying: "retrying",
}

func (ss ServerState) String() string {
    return stateName[ss]
}

func main() {
    ns := transition(StateIdle)
    fmt.Println(ns)

    ns2 := transition(ns)
    fmt.Println(ns2)
}

func transition(s ServerState) ServerState {
    switch s {
    case StateIdle:
        return StateConnected
    case StateConnected, StateRetrying:
        return StateIdle
    case StateError:
        return StateError
    default:
        panic(fmt.Errorf("unknown state: %s", s))
    }
}
```

```
$ go run enums.go
connected
idle
```

Next example: [Struct Embedding](#).

Go by Example: URL Parsing

URLs provide a [uniform way to locate resources](#). Here's how to parse URLs in Go.

We'll parse this example URL, which includes a scheme, authentication info, host, port, path, query params, and query fragment.

Parse the URL and ensure there are no errors.

Accessing the scheme is straightforward.

User contains all authentication info; call Username and Password on this for individual values.

The Host contains both the hostname and the port, if present. Use SplitHostPort to extract them.

Here we extract the path and the fragment after the #.

To get query params in a string of k=v format, use RawQuery. You can also parse query params into a map. The parsed query param maps are from strings to slices of strings, so index into [0] if you only want the first value.

Running our URL parsing program shows all the different pieces that we extracted.

```
package main

import (
    "fmt"
    "net"
    "net/url"
)

func main() {

    s := "postgres://user:pass@host.com:5432/path?k=v#f"

    u, err := url.Parse(s)
    if err != nil {
        panic(err)
    }

    fmt.Println(u.Scheme)

    fmt.Println(u.User)
    fmt.Println(u.User.Username())
    p, _ := u.User.Password()
    fmt.Println(p)

    fmt.Println(u.Host)
    host, port, _ := net.SplitHostPort(u.Host)
    fmt.Println(host)
    fmt.Println(port)

    fmt.Println(u.Path)
    fmt.Println(u.Fragment)

    fmt.Println(u.RawQuery)
    m, _ := url.ParseQuery(u.RawQuery)
    fmt.Println(m)
    fmt.Println(m["k"][0])
}
```

```
$ go run url-parsing.go
postgres
user:pass
user
pass
host.com:5432
host.com
5432
/path
f
k=v
map[k:[v]]
v
```

Next example: [SHA256 Hashes](#).

Go by Example: String Formatting

Go offers excellent support for string formatting in the `printf` tradition. Here are some examples of common string formatting tasks.

Go offers several printing “verbs” designed to format general Go values. For example, this prints an instance of our `point` struct.

If the value is a struct, the `%+v` variant will include the struct’s field names.

The `%#v` variant prints a Go syntax representation of the value, i.e. the source code snippet that would produce that value.

To print the type of a value, use `%T`.

Formatting booleans is straight-forward.

There are many options for formatting integers. Use `%d` for standard, base-10 formatting.

This prints a binary representation.

This prints the character corresponding to the given integer.

`%x` provides hex encoding.

There are also several formatting options for floats. For basic decimal formatting use `%f`.

`%e` and `%E` format the float in (slightly different versions of) scientific notation.

For basic string printing use `%s`.

To double-quote strings as in Go source, use `%q`.

As with integers seen earlier, `%x` renders the string in base-16, with two output characters per byte of input.

To print a representation of a pointer, use `%p`.

When formatting numbers you will often want to control the width and precision of the resulting figure. To specify the width of an integer, use a number after the `%` in the verb. By default the result will be right-justified and padded with spaces.

You can also specify the width of printed floats, though usually you’ll also want to restrict the decimal precision at the same time with the `width.precision` syntax.

```
package main

import (
    "fmt"
    "os"
)

type point struct {
    x, y int
}

func main() {

    p := point{1, 2}
    fmt.Printf("struct1: %v\n", p)

    fmt.Printf("struct2: %+v\n", p)

    fmt.Printf("struct3: %#v\n", p)

    fmt.Printf("type: %T\n", p)

    fmt.Printf("bool: %t\n", true)

    fmt.Printf("int: %d\n", 123)

    fmt.Printf("bin: %b\n", 14)

    fmt.Printf("char: %c\n", 33)

    fmt.Printf("hex: %x\n", 456)

    fmt.Printf("float1: %f\n", 78.9)

    fmt.Printf("float2: %e\n", 123400000.0)
    fmt.Printf("float3: %E\n", 123400000.0)

    fmt.Printf("str1: %s\n", "\"string\"")

    fmt.Printf("str2: %q\n", "\"string\"")

    fmt.Printf("str3: %x\n", "hex this")

    fmt.Printf("pointer: %p\n", &p)

    fmt.Printf("width1: |%6d|%6d|\n", 12, 345)

    fmt.Printf("width2: |%6.2f|%6.2f|\n", 1.2, 3.45)
```

To left-justify, use the `-` flag.

You may also want to control width when formatting strings, especially to ensure that they align in table-like output. For basic right-justified width.

To left-justify use the `-` flag as with numbers.

So far we've seen `Printf`, which prints the formatted string to `os.Stdout`. `Sprintf` formats and returns a string without printing it anywhere.

You can format+print to `io.Writers` other than `os.Stdout` using `Fprintf`.

```
fmt.Printf("width3:  |%-6.2f|%-6.2f|\n", 1.2, 3.45)

fmt.Printf("width4:  |%6s|%6s|\n", "foo", "b")

fmt.Printf("width5:  |%-6s|%-6s|\n", "foo", "b")

s := fmt.Sprintf("sprintf: a %s", "string")
fmt.Println(s)

fmt.Fprintf(os.Stderr, "io: an %s\n", "error")
}
```

```
$ go run string-formatting.go
struct1: {1 2}
struct2: {x:1 y:2}
struct3: main.point{x:1, y:2}
type: main.point
bool: true
int: 123
bin: 1110
char: !
hex: 1c8
float1: 78.900000
float2: 1.234000e+08
float3: 1.234000E+08
str1: "string"
str2: "\"string\""
str3: 6865782074686973
pointer: 0xc0000ba000
width1: |   12|   345|
width2: |  1.20|  3.45|
width3: |1.20 |3.45 |
width4: |   foo|    b|
width5: |foo  |b    |
sprintf: a string
io: an error
```

Next example: [Text Templates](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Range over Built-in Types

range iterates over elements in a variety of built-in data structures. Let's see how to use *range* with some of the data structures we've already learned.

Here we use *range* to sum the numbers in a slice. Arrays work like this too.

range on arrays and slices provides both the index and value for each entry. Above we didn't need the index, so we ignored it with the blank identifier `_`. Sometimes we actually want the indexes though.

range on map iterates over key/value pairs.

range can also iterate over just the keys of a map.

range on strings iterates over Unicode code points. The first value is the starting byte index of the rune and the second the rune itself. See [Strings and Runes](#) for more details.

```
package main

import "fmt"

func main() {

    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }

    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }

    for k := range kvs {
        fmt.Println("key:", k)
    }

    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

```
$ go run range-over-built-in-types.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
0 103
1 111
```

Next example: [Pointers](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Hello World

Our first program will print the classic “hello world” message. Here’s the full source code.

```
package main

import "fmt"

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

To run the program, put the code in `hello-world.go` and use `go run`.

Sometimes we’ll want to build our programs into binaries. We can do this using `go build`.

We can then execute the built binary directly.

Now that we can run and build basic Go programs, let’s learn more about the language.

Next example: [Values](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
$ go run hello-world.go
hello world

$ go build hello-world.go
$ ls
hello-world  hello-world.go

$ ./hello-world
hello world
```

Go by Example: Maps

Maps are Go's built-in [associative data type](#) (sometimes called *hashes* or *dicts* in other languages).

To create an empty map, use the builtin `make`:
`make(map[key-type]val-type)`.

Set key/value pairs using typical `name[key] = val` syntax.

Printing a map with e.g. `fmt.Println` will show all of its key/value pairs.

Get a value for a key with `name[key]`.

If the key doesn't exist, the [zero value](#) of the value type is returned.

The builtin `len` returns the number of key/value pairs when called on a map.

The builtin `delete` removes key/value pairs from a map.

To remove *all* key/value pairs from a map, use the `clear` builtin.

The optional second return value when getting a value from a map indicates if the key was present in the map. This can be used to disambiguate between missing keys and keys with zero values like `0` or `""`. Here we didn't need the value itself, so we ignored it with the *blank identifier* `_`.

You can also declare and initialize a new map in the same line with this syntax.

The `maps` package contains a number of useful utility functions for maps.

Note that maps appear in the form `map[k:v k:v]` when printed with `fmt.Println`.

```
package main

import (
    "fmt"
    "maps"
)

func main() {

    m := make(map[string]int)

    m["k1"] = 7
    m["k2"] = 13

    fmt.Println("map:", m)

    v1 := m["k1"]
    fmt.Println("v1:", v1)

    v3 := m["k3"]
    fmt.Println("v3:", v3)

    fmt.Println("len:", len(m))

    delete(m, "k2")
    fmt.Println("map:", m)

    clear(m)
    fmt.Println("map:", m)

    _, prs := m["k2"]
    fmt.Println("prs:", prs)

    n := map[string]int{"foo": 1, "bar": 2}
    fmt.Println("map:", n)

    n2 := map[string]int{"foo": 1, "bar": 2}
    if maps.Equal(n, n2) {
        fmt.Println("n == n2")
    }
}
```

```
$ go run maps.go
map: map[k1:7 k2:13]
v1: 7
v3: 0
len: 2
map: map[k1:7]
map: map[]
prs: false
map: map[bar:2 foo:1]
n == n2
```

Next example: [Functions](#).

Go by Example: Writing Files

Writing files in Go follows similar patterns to the ones we saw earlier for reading.

To start, here's how to dump a string (or just bytes) into a file.

For more granular writes, open a file for writing.

It's idiomatic to defer a `Close` immediately after opening a file.

You can write byte slices as you'd expect.

A `WriteString` is also available.

Issue a `Sync` to flush writes to stable storage.

`bufio` provides buffered writers in addition to the buffered readers we saw earlier.

Use `Flush` to ensure all buffered operations have been applied to the underlying writer.

Try running the file-writing code.

Then check the contents of the written files.

Next we'll look at applying some of the file I/O ideas we've just seen to the `stdin` and `stdout` streams.

Next example: [Line Filters](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {

    d1 := []byte("hello\ngo\n")
    err := os.WriteFile("/tmp/dat1", d1, 0644)
    check(err)

    f, err := os.Create("/tmp/dat2")
    check(err)

    defer f.Close()

    d2 := []byte{115, 111, 109, 101, 10}
    n2, err := f.Write(d2)
    check(err)
    fmt.Printf("wrote %d bytes\n", n2)

    n3, err := f.WriteString("writes\n")
    check(err)
    fmt.Printf("wrote %d bytes\n", n3)

    f.Sync()

    w := bufio.NewWriter(f)
    n4, err := w.WriteString("buffered\n")
    check(err)
    fmt.Printf("wrote %d bytes\n", n4)

    w.Flush()

}
```

```
$ go run writing-files.go
wrote 5 bytes
wrote 7 bytes
wrote 9 bytes

$ cat /tmp/dat1
hello
go
$ cat /tmp/dat2
some
writes
buffered
```


Go by Example: Structs

Go's *structs* are typed collections of fields. They're useful for grouping data together to form records.

This person struct type has name and age fields.

`newPerson` constructs a new person struct with the given name.

Go is a garbage collected language; you can safely return a pointer to a local variable - it will only be cleaned up by the garbage collector when there are no active references to it.

This syntax creates a new struct.

You can name the fields when initializing a struct.

Omitted fields will be zero-valued.

An `&` prefix yields a pointer to the struct.

It's idiomatic to encapsulate new struct creation in constructor functions

Access struct fields with a dot.

You can also use dots with struct pointers - the pointers are automatically dereferenced.

Structs are mutable.

If a struct type is only used for a single value, we don't have to give it a name. The value can have an anonymous struct type. This technique is commonly used for [table-driven tests](#).

```
package main

import "fmt"

type person struct {
    name string
    age  int
}

func newPerson(name string) *person {

    p := person{name: name}
    p.age = 42
    return &p
}

func main() {

    fmt.Println(person{"Bob", 20})

    fmt.Println(person{name: "Alice", age: 30})

    fmt.Println(person{name: "Fred"})

    fmt.Println(&person{name: "Ann", age: 40})

    fmt.Println(newPerson("Jon"))

    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    sp := &s
    fmt.Println(sp.age)

    sp.age = 51
    fmt.Println(sp.age)

    dog := struct {
        name  string
        isGood bool
    }{
        "Rex",
        true,
    }
    fmt.Println(dog)
}
```

```
$ go run structs.go
{Bob 20}
{Alice 30}
{Fred 0}
&{Ann 40}
&{Jon 42}
Sean
50
51
{Rex true}
```

Next example: [Methods](#).

Go by Example: Command-Line Subcommands

Some command-line tools, like the `go` tool or `git` have many *subcommands*, each with its own set of flags. For example, `go build` and `go get` are two different subcommands of the `go` tool. The `flag` package lets us easily define simple subcommands that have their own flags.

We declare a subcommand using the `NewFlagSet` function, and proceed to define new flags specific for this subcommand.

For a different subcommand we can define different supported flags.

The subcommand is expected as the first argument to the program.

Check which subcommand is invoked.

For every subcommand, we parse its own flags and have access to trailing positional arguments.

```
package main

import (
    "flag"
    "fmt"
    "os"
)

func main() {

    fooCmd := flag.NewFlagSet("foo", flag.ExitOnError)
    fooEnable := fooCmd.Bool("enable", false, "enable")
    fooName := fooCmd.String("name", "", "name")

    barCmd := flag.NewFlagSet("bar", flag.ExitOnError)
    barLevel := barCmd.Int("level", 0, "level")

    if len(os.Args) < 2 {
        fmt.Println("expected 'foo' or 'bar' subcommands")
        os.Exit(1)
    }

    switch os.Args[1] {

    case "foo":
        fooCmd.Parse(os.Args[2:])
        fmt.Println("subcommand 'foo'")
        fmt.Println("  enable:", *fooEnable)
        fmt.Println("  name:", *fooName)
        fmt.Println("  tail:", fooCmd.Args())
    case "bar":
        barCmd.Parse(os.Args[2:])
        fmt.Println("subcommand 'bar'")
        fmt.Println("  level:", *barLevel)
        fmt.Println("  tail:", barCmd.Args())
    default:
        fmt.Println("expected 'foo' or 'bar' subcommands")
        os.Exit(1)
    }
}
```

First invoke the `foo` subcommand.

Now try `bar`.

But `bar` won't accept `foo`'s flags.

```
$ go build command-line-subcommands.go

$ ./command-line-subcommands foo -enable -name=joe a1 a2
subcommand 'foo'
  enable: true
  name: joe
  tail: [a1 a2]

$ ./command-line-subcommands bar -level 8 a1
subcommand 'bar'
  level: 8
  tail: [a1]

$ ./command-line-subcommands bar -enable a1
flag provided but not defined: -enable
Usage of bar:
  -level int
     level
```

Next we'll look at environment variables, another common way to parameterize programs.

Next example: [Environment Variables](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: HTTP Client

The Go standard library comes with excellent support for HTTP clients and servers in the `net/http` package. In this example we'll use it to issue simple HTTP requests.

Issue an HTTP GET request to a server. `http.Get` is a convenient shortcut around creating an `http.Client` object and calling its `Get` method; it uses the `http.DefaultClient` object which has useful default settings.

Print the HTTP response status.

Print the first 5 lines of the response body.

```
package main

import (
    "bufio"
    "fmt"
    "net/http"
)

func main() {
    resp, err := http.Get("https://gobyexample.com")
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()

    fmt.Println("Response status:", resp.Status)

    scanner := bufio.NewScanner(resp.Body)
    for i := 0; scanner.Scan() && i < 5; i++ {
        fmt.Println(scanner.Text())
    }

    if err := scanner.Err(); err != nil {
        panic(err)
    }
}
```

```
$ go run http-clients.go
Response status: 200 OK
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Go by Example</title>
```

Next example: [HTTP Server](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Text Templates

Go offers built-in support for creating dynamic content or showing customized output to the user with the `text/template` package. A sibling package named `html/template` provides the same API but has additional security features and should be used for generating HTML.

We can create a new template and parse its body from a string. Templates are a mix of static text and “actions” enclosed in `{{...}}` that are used to dynamically insert content.

Alternatively, we can use the `template.Must` function to panic in case `Parse` returns an error. This is especially useful for templates initialized in the global scope.

By “executing” the template we generate its text with specific values for its actions. The `{{.}}` action is replaced by the value passed as a parameter to `Execute`.

Helper function we’ll use below.

If the data is a struct we can use the `{{.FieldName}}` action to access its fields. The fields should be exported to be accessible when a template is executing.

The same applies to maps; with maps there is no restriction on the case of key names.

if/else provide conditional execution for templates. A value is considered false if it’s the default value of a type, such as 0, an empty string, nil pointer, etc. This sample demonstrates another feature of templates: using `-` in actions to trim whitespace.

range blocks let us loop through slices, arrays, maps or channels. Inside the range block `{{.}}` is set to the current item of the iteration.

```
package main

import (
    "os"
    "text/template"
)

func main() {

    t1 := template.New("t1")
    t1, err := t1.Parse("Value is {{.}}\n")
    if err != nil {
        panic(err)
    }

    t1 = template.Must(t1.Parse("Value: {{.}}\n"))

    t1.Execute(os.Stdout, "some text")
    t1.Execute(os.Stdout, 5)
    t1.Execute(os.Stdout, []string{
        "Go",
        "Rust",
        "C++",
        "C#",
    })

    Create := func(name, t string) *template.Template {
        return template.Must(template.New(name).Parse(t))
    }

    t2 := Create("t2", "Name: {{.Name}}\n")

    t2.Execute(os.Stdout, struct {
        Name string
    }{"Jane Doe"})

    t2.Execute(os.Stdout, map[string]string{
        "Name": "Mickey Mouse",
    })

    t3 := Create("t3",
        "{{if . -}} yes {{else -}} no {{end}}\n")
    t3.Execute(os.Stdout, "not empty")
    t3.Execute(os.Stdout, "")

    t4 := Create("t4",
        "Range: {{range .}}{{.}} {{end}}\n")
    t4.Execute(os.Stdout,
        []string{
            "Go",
            "Rust",
            "C++",
            "C#",
        })
}
```

```
$ go run templates.go
```

```
Value: some text
Value: 5
Value: [Go Rust C++ C#]
Name: Jane Doe
Name: Mickey Mouse
yes
no
Range: Go Rust C++ C#
```

Next example: [Regular Expressions](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Variadic Functions

Variadic functions can be called with any number of trailing arguments. For example, `fmt.Println` is a common variadic function.

Here's a function that will take an arbitrary number of ints as arguments.

Within the function, the type of `nums` is equivalent to `[]int`. We can call `len(nums)`, iterate over it with `range`, etc.

Variadic functions can be called in the usual way with individual arguments.

If you already have multiple args in a slice, apply them to a variadic function using `func(slice...)` like this.

```
package main

import "fmt"

func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0

    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {

    sum(1, 2)
    sum(1, 2, 3)

    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

```
$ go run variadic-functions.go
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

Another key aspect of functions in Go is their ability to form closures, which we'll look at next.

Next example: Closures.

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: If/Else

Branching with `if` and `else` in Go is straight-forward.

Here's a basic example.

You can have an `if` statement without an `else`.

Logical operators like `&&` and `||` are often useful in conditions.

A statement can precede conditionals; any variables declared in this statement are available in the current and all subsequent branches.

Note that you don't need parentheses around conditions in Go, but that the braces are required.

```
package main

import "fmt"

func main() {

    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    if 8%2 == 0 || 7%2 == 0 {
        fmt.Println("either 8 or 7 are even")
    }

    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

```
$ go run if-else.go
7 is odd
8 is divisible by 4
either 8 or 7 are even
9 has 1 digit
```

There is no [ternary if](#) in Go, so you'll need to use a full `if` statement even for basic conditions.

Next example: [Switch](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Errors

In Go it's idiomatic to communicate errors via an explicit, separate return value. This contrasts with the exceptions used in languages like Java and Ruby and the overloaded single result / error value sometimes used in C. Go's approach makes it easy to see which functions return errors and to handle them using the same language constructs employed for other, non-error tasks.

See the documentation of the [errors package](#) and [this blog post](#) for additional details.

By convention, errors are the last return value and have type error, a built-in interface.

`errors.New` constructs a basic error value with the given error message.

A `nil` value in the error position indicates that there was no error.

A sentinel error is a predeclared variable that is used to signify a specific error condition.

We can wrap errors with higher-level errors to add context. The simplest way to do this is with the `%w` verb in `fmt.Errorf`. Wrapped errors create a logical chain (A wraps B, which wraps C, etc.) that can be queried with functions like `errors.Is` and `errors.As`.

It's common to use an inline error check in the `if` line.

`errors.Is` checks that a given error (or any error in its chain) matches a specific error value. This is especially useful with wrapped or nested errors, allowing you to identify specific error types or sentinel errors in a chain of errors.

```
package main

import (
    "errors"
    "fmt"
)

func f(arg int) (int, error) {
    if arg == 42 {

        return -1, errors.New("can't work with 42")
    }

    return arg + 3, nil
}

var ErrOutOfTea = fmt.Errorf("no more tea available")
var ErrPower = fmt.Errorf("can't boil water")

func makeTea(arg int) error {
    if arg == 2 {
        return ErrOutOfTea
    } else if arg == 4 {

        return fmt.Errorf("making tea: %w", ErrPower)
    }
    return nil
}

func main() {
    for _, i := range []int{7, 42} {

        if r, e := f(i); e != nil {
            fmt.Println("f failed:", e)
        } else {
            fmt.Println("f worked:", r)
        }
    }

    for i := range 5 {
        if err := makeTea(i); err != nil {

            if errors.Is(err, ErrOutOfTea) {
                fmt.Println("We should buy new tea!")
            } else if errors.Is(err, ErrPower) {
                fmt.Println("Now it is dark.")
            } else {
                fmt.Printf("unknown error: %s\n", err)
            }
            continue
        }

        fmt.Println("Tea is ready!")
    }
}
```

```
$ go run errors.go
```

```
f worked: 10
f failed: can't work with 42
Tea is ready!
Tea is ready!
We should buy new tea!
Tea is ready!
Now it is dark.
```

Next example: [Custom Errors](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Teaching coding with JavaScript and p5.js (<https://eli.thegreenplace.net/2025/teaching-coding-with-javascript-and-p5js/>)

📅 May 10, 2025 at 08:57

When asked which programming language to learn first - especially for kids - my usual answer is JavaScript [1]. Nothing beats the direct feedback you get from code that's able to paint things on the screen, without having to install anything.

One library that makes it a particularly pleasant process is [p5.js](https://p5js.org/) (<https://p5js.org/>), which was created specifically for this educational purpose. I've had good experience teaching kids basic programming using p5.js. Here's a simple example of what I mean:

(You should see some circles moving on the canvas; click on the canvas to add more)

Even though this demo is so trivial, it has many of the elements of creating simple games - colorful stuff is drawn on the screen, things are moving around according to simple physical laws, and there's interactivity!

Here's the entire code required to implement this with p5.js:

```

let circles = [];

function setup() {
  createCanvas(400, 400);

  for (let i = 0; i < 5; i++) {
    circles.push(randomCircleAtPos(
      width / 2 + random(-100, 100),
      height / 2 + random(-100, 100)));
  }
}

function draw() {
  background(240);

  for (let c of circles) {
    c.x += c.xSpeed;
    c.y += c.ySpeed;

    // Bounce off the walls
    if (c.x + c.size / 2 > width || c.x - c.size / 2 < 0) {
      c.xSpeed *= -1;
    }
    if (c.y + c.size / 2 > height || c.y - c.size / 2 < 0) {
      c.ySpeed *= -1;
    }
    fill(c.color);
    circle(c.x, c.y, c.size);
  }
}

function mousePressed() {
  circles.push(randomCircleAtPos(mouseX, mouseY));
}

function randomCircleAtPos(x, y) {
  return {
    x: x,
    y: y,
    size: random(20, 80),
    color: color(random(255), random(255), random(255)),
    xSpeed: random(-2, 2),
    ySpeed: random(-2, 2)
  };
}

```

There are many niceties provided by p5.js here:

- No need to write any HTML! The `createCanvas` call will create a canvas element and all subsequent drawing and interaction happens on it [2].
- `setup` is a "magic" function that gets invoked once at the beginning of the program.
- `draw` is another magic function that gets automatically called for every animation frame (p5.js arranges the correct `requestAnimationFrame` calls behind the scenes).
- There are many useful helper functions for drawing, without having to deal with the HTML canvas API, e.g. `background`, `fill`, `circle`. In particular, there's no need to deal with canvas contexts or paths (<https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/beginPath>).
- `mousePressed` is yet another magic function that is called on mouse clicks within the canvas; `mouseX` and `mouseY` are magic globals that specify the mouse location within the canvas (without having to worry about client position offsets, etc.)
- Utility functions like `color` and `random` have convenient, simple APIs. p5.js has many others, specifically tuned for developing simulations and games. There are vectors (<https://p5js.org/reference/p5/p5.Vector/>), utilities for smooth random noise (<https://p5js.org/reference/p5/noise/>), functions for mapping between linear ranges (<https://p5js.org/reference/p5/map/>), etc.

I also wrote a version of the same animation without p5.js, using plain JS and canvas APIs instead. It's available on GitHub (<https://github.com/eliben/code-for-blog/tree/main/2025/p5-moving-circles>) - feel free to compare!

Some history

It all started with [Processing \(https://processing.org/\)](https://processing.org/), a 25-year-old Java library designed to teach people how to code by creating animations and games (and conversely, giving artists and animators simple tools to enhance their work with code). In 2007, John Resig (of jQuery fame) developed [Processing.js \(https://github.com/processing-js/processing-js\)](https://github.com/processing-js/processing-js) - a JS clone of Processing; Khan Academy started using Processing.js for a programming unit on their website.

p5.js was created in 2013; here's a brief history from their GitHub repository:

p5.js was created by [Lauren Lee McCarthy \(https://github.com/lmccart\)](https://github.com/lmccart) in 2013 as a new interpretation of Processing for the context of the web. Since then we have allowed ourselves space to deviate and grow, while drawing inspiration from Processing and our shared community. p5.js is sustained by a community of contributors, with support from the Processing Foundation.

As a result of p5's growth in popularity, Processing.js has been archived a few years ago and new users are directed to p5.js.

While p5.js has a very similar *feel* to the original Java Processing library, I strongly recommend the former. With the ubiquity of the web these days, there's really no reason to use the Java variant with all the complexity of installation and running separate tools it requires. The browser is all you need!

Educational resources

At the time of writing, Khan Academy's [Computer Programming course \(https://www.khanacademy.org/computing/computer-programming\)](https://www.khanacademy.org/computing/computer-programming) (starting at unit 4) still uses Processing.js, but it's similar enough to p5.js that I recommend ignoring the difference and just doing it - it's a great resource.

The Coding Train (<https://thecodingtrain.com/>) is another fantastic resource that uses p5.js directly to teach programming for beginners in a friendly and engaging style. If you prefer a book format with more advanced material, check out [Nature of Code \(https://natureofcode.com/\)](https://natureofcode.com/) from the same author.

Finally, p5.js comes with its own [online editor \(https://editor.p5js.org/\)](https://editor.p5js.org/), where you can create an arbitrary number of projects (each with multiple files, if you want), and have a live preview of everything in the browser.



Using p5.js for professional programming?

OK, so p5.js is a great resource for beginners learning to write code. Is it recommended for professional programming, though? Should you incorporate p5.js in your frontend work?

This depends, but in general I'd recommend against it. At the end of the day, it all comes down to the [benefits of dependencies as a function of effort \(https://eli.thegreenplace.net/2017/benefits-of-dependencies-in-software-projects-as-a-function-of-effort/\)](https://eli.thegreenplace.net/2017/benefits-of-dependencies-in-software-projects-as-a-function-of-effort/). p5.js has a very wide and shallow API - if you're already a seasoned programmer familiar with JS, the additional functionality p5.js provides is fairly trivial [3]. Sooner or later you'll find yourself at odds with p5.js's abstraction or implementation of some concept and will start looking for a way out. For experienced programmers, the raw canvas API isn't that bad to deal with, so the biggest benefits of p5.js dissipate rather quickly.

That said, if you want to hack together a quick game or simulation and p5.js makes your life easier - why not! Just remember the benefit vs. effort curve.

- [1] This sometimes raises eyebrows for experienced programmers, because JavaScript has a certain reputation. My view is that none of this matters for beginners - they typically don't care about our pedantic nuances, and for them JS is as good as any other language. But the *environment* in which JS executes is the real boon. Imagine you're a kid with a Chromebook; to create a simple game with JS, you don't have to install *anything*. Just open any web-based JS IDE (e.g. CodePen, JSFiddle, or - better yet - p5.js's own online editor (<https://editor.p5js.org/>)) and start coding. With some honest copy pasting, you can go from blank screen to colorful objects moving around and interacting with your mouse in less than a minute.
- [2] If you follow the source of my animation on this page, you'll notice I'm cheating a bit - I *do* need to create an explicit canvas element in HTML because I have to properly embed it within this blog post. But for demos where all you have on the screen is that canvas - this isn't needed. Beginning programmers can completely ignore the existence of HTML and CSS when starting with p5.js!
- [3] As an example of what I mean, here's p5min (<https://github.com/eliben/code-for-blog/tree/main/2025/p5-moving-circles/p5min>) - a minimal clone of p5.js sufficient to run the circles demo shown earlier in this post. It's not hard to keep extending it gradually to implement additional functionality, as needed.
-

Recent posts

- 2025.05.01: Bloom filters (<https://eli.thegreenplace.net/2025/bloom-filters/>)
- 2025.04.18: Sparsely-gated Mixture Of Experts (MoE) (<https://eli.thegreenplace.net/2025/sparsely-gated-mixture-of-experts-moe/>)
- 2025.04.12: Cross-entropy and KL divergence (<https://eli.thegreenplace.net/2025/cross-entropy-and-kl-divergence/>)
- 2025.04.05: Reproducing word2vec with JAX (<https://eli.thegreenplace.net/2025/reproducing-word2vec-with-jax/>)
- 2025.03.31: Summary of reading: January - March 2025 (<https://eli.thegreenplace.net/2025/summary-of-reading-january-march-2025/>)
- 2025.03.26: Notes on implementing Attention (<https://eli.thegreenplace.net/2025/notes-on-implementing-attention/>)
- 2025.03.22: Understanding Numpy's einsum (<https://eli.thegreenplace.net/2025/understanding-numpys-einsum/>)
- 2025.02.22: Making any integer with four 2s (<https://eli.thegreenplace.net/2025/making-any-integer-with-four-2s/>)
- 2025.02.18: Benchmarking utility for Python (<https://eli.thegreenplace.net/2025/benchmarking-utility-for-python/>)
- 2025.02.03: Decorator JITs - Python as a DSL (<https://eli.thegreenplace.net/2025/decorator-jits-python-as-a-dsl/>)

See Archives (<https://eli.thegreenplace.net/archives/all>) for a full list.

Go by Example: String Functions

The standard library's `strings` package provides many useful string-related functions. Here are some examples to give you a sense of the package.

We alias `fmt.Println` to a shorter name as we'll use it a lot below.

Here's a sample of the functions available in `strings`. Since these are functions from the package, not methods on the string object itself, we need pass the string in question as the first argument to the function. You can find more functions in the [strings](#) package docs.

```
package main

import (
    "fmt"
    s "strings"
)

var p = fmt.Println

func main() {

    p("Contains: ", s.Contains("test", "es"))
    p("Count:    ", s.Count("test", "t"))
    p("HasPrefix: ", s.HasPrefix("test", "te"))
    p("HasSuffix: ", s.HasSuffix("test", "st"))
    p("Index:     ", s.Index("test", "e"))
    p("Join:      ", s.Join([]string{"a", "b"}, "-"))
    p("Repeat:    ", s.Repeat("a", 5))
    p("Replace:   ", s.Replace("foo", "o", "0", -1))
    p("Replace:   ", s.Replace("foo", "o", "0", 1))
    p("Split:     ", s.Split("a-b-c-d-e", "-"))
    p("ToLower:   ", s.ToLower("TEST"))
    p("ToUpper:   ", s.ToUpper("test"))

}
```

```
$ go run string-functions.go
Contains:  true
Count:    2
HasPrefix: true
HasSuffix: true
Index:    1
Join:     a-b
Repeat:   aaaaa
Replace:  f00
Replace:  f0o
Split:    [a b c d e]
ToLower:  test
ToUpper:  TEST
```

Next example: [String Formatting](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Command-Line Arguments

Command-line arguments are a common way to parameterize execution of programs. For example, `go run hello.go` uses `run` and `hello.go` arguments to the `go` program.

`os.Args` provides access to raw command-line arguments. Note that the first value in this slice is the path to the program, and `os.Args[1:]` holds the arguments to the program.

You can get individual args with normal indexing.

To experiment with command-line arguments it's best to build a binary with `go build` first.

Next we'll look at more advanced command-line processing with flags.

Next example: [Command-Line Flags](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "os"
)

func main() {

    argsWithProg := os.Args
    argsWithoutProg := os.Args[1:]

    arg := os.Args[3]

    fmt.Println(argsWithProg)
    fmt.Println(argsWithoutProg)
    fmt.Println(arg)
}
```

```
$ go build command-line-arguments.go
$ ./command-line-arguments a b c d
[./command-line-arguments a b c d]
[a b c d]
c
```

Go by Example: Environment Variables

[Environment variables](#) are a universal mechanism for [conveying configuration information to Unix programs](#). Let's look at how to set, get, and list environment variables.

To set a key/value pair, use `os.Setenv`. To get a value for a key, use `os.Getenv`. This will return an empty string if the key isn't present in the environment.

Use `os.Environ` to list all key/value pairs in the environment. This returns a slice of strings in the form `KEY=value`. You can `strings.SplitN` them to get the key and value. Here we print all the keys.

Running the program shows that we pick up the value for `F00` that we set in the program, but that `BAR` is empty.

The list of keys in the environment will depend on your particular machine.

If we set `BAR` in the environment first, the running program picks that value up.

Next example: [Logging](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "os"
    "strings"
)

func main() {

    os.Setenv("F00", "1")
    fmt.Println("F00:", os.Getenv("F00"))
    fmt.Println("BAR:", os.Getenv("BAR"))

    fmt.Println()
    for _, e := range os.Environ() {
        pair := strings.SplitN(e, "=", 2)
        fmt.Println(pair[0])
    }
}
```

```
$ go run environment-variables.go
F00: 1
BAR:

TERM_PROGRAM
PATH
SHELL
...
F00

$ BAR=2 go run environment-variables.go
F00: 1
BAR: 2
...
```

Go by Example: Time Formatting / Parsing

Go supports time formatting and parsing via pattern-based layouts.

Here's a basic example of formatting a time according to RFC3339, using the corresponding layout constant.

Time parsing uses the same layout values as Format.

Format and Parse use example-based layouts. Usually you'll use a constant from time for these layouts, but you can also supply custom layouts. Layouts must use the reference time Mon Jan 2 15:04:05 MST 2006 to show the pattern with which to format/parse a given time/string. The example time must be exactly as shown: the year 2006, 15 for the hour, Monday for the day of the week, etc.

For purely numeric representations you can also use standard string formatting with the extracted components of the time value.

Parse will return an error on malformed input explaining the parsing problem.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    p := fmt.Println

    t := time.Now()
    p(t.Format(time.RFC3339))

    t1, e := time.Parse(
        time.RFC3339,
        "2012-11-01T22:08:41+00:00")
    p(t1)

    p(t.Format("3:04PM"))
    p(t.Format("Mon Jan _2 15:04:05 2006"))
    p(t.Format("2006-01-02T15:04:05.999999-07:00"))
    form := "3 04 PM"
    t2, e := time.Parse(form, "8 41 PM")
    p(t2)

    fmt.Printf("%d-%02d-%02dT%02d:%02d:%02d-00:00\n",
        t.Year(), t.Month(), t.Day(),
        t.Hour(), t.Minute(), t.Second())

    ansic := "Mon Jan _2 15:04:05 2006"
    _, e = time.Parse(ansic, "8:41PM")
    p(e)
}
```

```
$ go run time-formatting-parsing.go
2014-04-15T18:00:15-07:00
2012-11-01 22:08:41 +0000 +0000
6:00PM
Tue Apr 15 18:00:15 2014
2014-04-15T18:00:15.161182-07:00
0000-01-01 20:41:00 +0000 UTC
2014-04-15T18:00:15-00:00
parsing time "8:41PM" as "Mon Jan _2 15:04:05 2006": ...
```

Next example: [Random Numbers](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Exit

Use `os.Exit` to immediately exit with a given status.

defers will *not* be run when using `os.Exit`, so this `fmt.Println` will never be called.

Exit with status 3.

Note that unlike e.g. C, Go does not use an integer return value from `main` to indicate exit status. If you'd like to exit with a non-zero status you should use `os.Exit`.

If you run `exit.go` using `go run`, the exit will be picked up by go and printed.

By building and executing a binary you can see the status in the terminal.

Note that the `!` from our program never got printed.

```
package main

import (
    "fmt"
    "os"
)

func main() {

    defer fmt.Println("!")

    os.Exit(3)
}
```

```
$ go run exit.go
exit status 3

$ go build exit.go
$ ./exit
$ echo $?
3
```

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Reading Files

Reading and writing files are basic tasks needed for many Go programs. First we'll look at some examples of reading files.

Reading files requires checking most calls for errors. This helper will streamline our error checks below.

Perhaps the most basic file reading task is slurping a file's entire contents into memory.

You'll often want more control over how and what parts of a file are read. For these tasks, start by opening a file to obtain an `os.File` value.

Read some bytes from the beginning of the file. Allow up to 5 to be read but also note how many actually were read.

You can also `Seek` to a known location in the file and Read from there.

Other methods of seeking are relative to the current cursor position,

and relative to the end of the file.

The `io` package provides some functions that may be helpful for file reading. For example, reads like the ones above can be more robustly implemented with `ReadAtLeast`.

There is no built-in rewind, but `Seek(0, io.SeekStart)` accomplishes this.

The `bufio` package implements a buffered reader that may be useful both for its efficiency with many small reads and because of the additional reading methods it provides.

Close the file when you're done (usually this would be scheduled immediately after opening with `defer`).

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {

    dat, err := os.ReadFile("/tmp/dat")
    check(err)
    fmt.Print(string(dat))

    f, err := os.Open("/tmp/dat")
    check(err)

    b1 := make([]byte, 5)
    n1, err := f.Read(b1)
    check(err)
    fmt.Printf("%d bytes: %s\n", n1, string(b1[:n1]))

    o2, err := f.Seek(6, io.SeekStart)
    check(err)
    b2 := make([]byte, 2)
    n2, err := f.Read(b2)
    check(err)
    fmt.Printf("%d bytes @ %d: ", n2, o2)
    fmt.Printf("%v\n", string(b2[:n2]))

    _, err = f.Seek(2, io.SeekCurrent)
    check(err)

    _, err = f.Seek(-4, io.SeekEnd)
    check(err)

    o3, err := f.Seek(6, io.SeekStart)
    check(err)
    b3 := make([]byte, 2)
    n3, err := io.ReadAtLeast(f, b3, 2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n3, o3, string(b3))

    _, err = f.Seek(0, io.SeekStart)
    check(err)

    r4 := bufio.NewReader(f)
    b4, err := r4.Peek(5)
    check(err)
    fmt.Printf("5 bytes: %s\n", string(b4))

    f.Close()
}
```

```
$ echo "hello" > /tmp/dat
$ echo "go" >> /tmp/dat
$ go run reading-files.go
hello
```



```
go
5 bytes: hello
2 bytes @ 6: go
2 bytes @ 6: go
5 bytes: hello
```

Next we'll look at writing files.

Next example: [Writing Files](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Defer

Defer is used to ensure that a function call is performed later in a program's execution, usually for purposes of cleanup. `defer` is often used where e.g. `ensure` and `finally` would be used in other languages.

Suppose we wanted to create a file, write to it, and then close when we're done. Here's how we could do that with `defer`.

Immediately after getting a file object with `createFile`, we defer the closing of that file with `closeFile`. This will be executed at the end of the enclosing function (`main`), after `writeFile` has finished.

It's important to check for errors when closing a file, even in a deferred function.

Running the program confirms that the file is closed after being written.

Next example: [Recover](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "os"
)

func main() {

    f := createFile("/tmp/defer.txt")
    defer closeFile(f)
    writeFile(f)
}

func createFile(p string) *os.File {
    fmt.Println("creating")
    f, err := os.Create(p)
    if err != nil {
        panic(err)
    }
    return f
}

func writeFile(f *os.File) {
    fmt.Println("writing")
    fmt.Fprintln(f, "data")
}

func closeFile(f *os.File) {
    fmt.Println("closing")
    err := f.Close()

    if err != nil {
        panic(err)
    }
}
```

```
$ go run defer.go
creating
writing
closing
```

Go by Example: Variables

In Go, *variables* are explicitly declared and used by the compiler to e.g. check type-correctness of function calls.

`var` declares 1 or more variables.

You can declare multiple variables at once.

Go will infer the type of initialized variables.

Variables declared without a corresponding initialization are *zero-valued*. For example, the zero value for an `int` is `0`.

The `:=` syntax is shorthand for declaring and initializing a variable, e.g. for `var f string = "apple"` in this case. This syntax is only available inside functions.

```
package main

import "fmt"

func main() {

    var a = "initial"
    fmt.Println(a)

    var b, c int = 1, 2
    fmt.Println(b, c)

    var d = true
    fmt.Println(d)

    var e int
    fmt.Println(e)

    f := "apple"
    fmt.Println(f)
}
```

```
$ go run variables.go
initial
1 2
true
0
apple
```

Next example: [Constants](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Panic

A panic typically means something went unexpectedly wrong. Mostly we use it to fail fast on errors that shouldn't occur during normal operation, or that we aren't prepared to handle gracefully.

We'll use panic throughout this site to check for unexpected errors. This is the only program on the site designed to panic.

A common use of panic is to abort if a function returns an error value that we don't know how to (or want to) handle. Here's an example of panicking if we get an unexpected error when creating a new file.

Running this program will cause it to panic, print an error message and goroutine traces, and exit with a non-zero status.

When first panic in main fires, the program exits without reaching the rest of the code. If you'd like to see the program try to create a temp file, comment the first panic out.

Note that unlike some languages which use exceptions for handling of many errors, in Go it is idiomatic to use error-indicating return values wherever possible.

Next example: [Defer](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import "os"

func main() {

    panic("a problem")

    _, err := os.Create("/tmp/file")
    if err != nil {
        panic(err)
    }
}
```

```
$ go run panic.go
panic: a problem

goroutine 1 [running]:
main.main()
    ../panic.go:12 +0x47
...
exit status 2
```

Go by Example: Recursion

Go supports *recursive functions*. Here's a classic example.

This fact function calls itself until it reaches the base case of fact(0).

Anonymous functions can also be recursive, but this requires explicitly declaring a variable with var to store the function before it's defined.

Since fib was previously declared in main, Go knows which function to call with fib here.

```
package main

import "fmt"

func fact(n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}

func main() {
    fmt.Println(fact(7))

    var fib func(n int) int

    fib = func(n int) int {
        if n < 2 {
            return n
        }

        return fib(n-1) + fib(n-2)
    }

    fmt.Println(fib(7))
}
```

```
$ go run recursion.go
5040
13
```

Next example: [Range over Built-in Types](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Channel Buffering

By default channels are *unbuffered*, meaning that they will only accept sends (`chan <-`) if there is a corresponding receive (`<- chan`) ready to receive the sent value. *Buffered channels* accept a limited number of values without a corresponding receiver for those values.

Here we make a channel of strings buffering up to 2 values.

Because this channel is buffered, we can send these values into the channel without a corresponding concurrent receive.

Later we can receive these two values as usual.

```
package main

import "fmt"

func main() {

    messages := make(chan string, 2)

    messages <- "buffered"
    messages <- "channel"

    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

```
$ go run channel-buffering.go
buffered
channel
```

Next example: [Channel Synchronization](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Timeouts

Timeouts are important for programs that connect to external resources or that otherwise need to bound execution time. Implementing timeouts in Go is easy and elegant thanks to channels and select.

For our example, suppose we're executing an external call that returns its result on a channel `c1` after 2s. Note that the channel is buffered, so the send in the goroutine is nonblocking. This is a common pattern to prevent goroutine leaks in case the channel is never read.

Here's the select implementing a timeout. `res := <-c1` awaits the result and `<-time.After` awaits a value to be sent after the timeout of 1s. Since select proceeds with the first receive that's ready, we'll take the timeout case if the operation takes more than the allowed 1s.

If we allow a longer timeout of 3s, then the receive from `c2` will succeed and we'll print the result.

Running this program shows the first operation timing out and the second succeeding.

Next example: [Non-Blocking Channel Operations](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "time"
)

func main() {

    c1 := make(chan string, 1)
    go func() {
        time.Sleep(2 * time.Second)
        c1 <- "result 1"
    }()

    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(1 * time.Second):
        fmt.Println("timeout 1")
    }

    c2 := make(chan string, 1)
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "result 2"
    }()
    select {
    case res := <-c2:
        fmt.Println(res)
    case <-time.After(3 * time.Second):
        fmt.Println("timeout 2")
    }
}
```

```
$ go run timeouts.go
timeout 1
result 2
```

Go by Example: Logging

The Go standard library provides straightforward tools for outputting logs from Go programs, with the [log](#) package for free-form output and the [log/slog](#) package for structured output.

Simply invoking functions like `Println` from the `log` package uses the *standard* logger, which is already pre-configured for reasonable logging output to `os.Stderr`. Additional methods like `Fatal*` or `Panic*` will exit the program after logging.

Loggers can be configured with *flags* to set their output format. By default, the standard logger has the `log.Ldate` and `log.Ltime` flags set, and these are collected in `log.LstdFlags`. We can change its flags to emit time with microsecond accuracy, for example.

It also supports emitting the file name and line from which the `log` function is called.

It may be useful to create a custom logger and pass it around. When creating a new logger, we can set a *prefix* to distinguish its output from other loggers.

We can set the prefix on existing loggers (including the standard one) with the `SetPrefix` method.

Loggers can have custom output targets; any `io.Writer` works.

This call writes the log output into `buf`.

This will actually show it on standard output.

The `slog` package provides *structured* log output. For example, logging in JSON format is straightforward.

In addition to the message, `slog` output can contain an arbitrary number of key=value pairs.

Sample output; the date and time emitted will depend on when the example ran.

These are wrapped for clarity of presentation on the website; in reality they are emitted on a single line.

Next example: [HTTP Client](#).

```
package main
```

```
import (  
    "bytes"  
    "fmt"  
    "log"  
    "os"  
  
    "log/slog"  
)
```

```
func main() {
```

```
    log.Println("standard logger")
```

```
    log.SetFlags(log.LstdFlags | log.Lmicroseconds)  
    log.Println("with micro")
```

```
    log.SetFlags(log.LstdFlags | log.Lshortfile)  
    log.Println("with file/line")
```

```
    mylog := log.New(os.Stdout, "my:", log.LstdFlags)  
    mylog.Println("from mylog")
```

```
    mylog.SetPrefix("ohmy:")  
    mylog.Println("from mylog")
```

```
    var buf bytes.Buffer  
    buflog := log.New(&buf, "buf:", log.LstdFlags)
```

```
    buflog.Println("hello")
```

```
    fmt.Print("from buflog:", buf.String())
```

```
    jsonHandler := slog.NewJSONHandler(os.Stderr, nil)  
    myslog := slog.New(jsonHandler)  
    myslog.Info("hi there")
```

```
    myslog.Info("hello again", "key", "val", "age", 25)
```

```
}
```

```
$ go run logging.go
```

```
2023/08/22 10:45:16 standard logger  
2023/08/22 10:45:16.904141 with micro  
2023/08/22 10:45:16 logging.go:40: with file/line  
my:2023/08/22 10:45:16 from mylog  
ohmy:2023/08/22 10:45:16 from mylog  
from buflog:buf:2023/08/22 10:45:16 hello
```

```
{"time":"2023-08-22T10:45:16.904166391-07:00",  
 "level":"INFO","msg":"hi there"}  
{"time":"2023-08-22T10:45:16.904178985-07:00",  
 "level":"INFO","msg":"hello again",  
 "key":"val","age":25}
```


Go by Example: Base64 Encoding

Go provides built-in support for [base64 encoding/decoding](#).

This syntax imports the `encoding/base64` package with the `b64` name instead of the default `base64`. It'll save us some space below.

Here's the string we'll encode/decode.

Go supports both standard and URL-compatible base64. Here's how to encode using the standard encoder. The encoder requires a `[]byte` so we convert our string to that type.

Decoding may return an error, which you can check if you don't already know the input to be well-formed.

This encodes/decodes using a URL-compatible base64 format.

The string encodes to slightly different values with the standard and URL base64 encoders (trailing `+` vs `-`) but they both decode to the original string as desired.

```
package main

import (
    b64 "encoding/base64"
    "fmt"
)

func main() {

    data := "abc123!?$*&'-'=@~"

    sEnc := b64.StdEncoding.EncodeToString([]byte(data))
    fmt.Println(sEnc)

    sDec, _ := b64.StdEncoding.DecodeString(sEnc)
    fmt.Println(string(sDec))
    fmt.Println()

    uEnc := b64.URLEncoding.EncodeToString([]byte(data))
    fmt.Println(uEnc)
    uDec, _ := b64.URLEncoding.DecodeString(uEnc)
    fmt.Println(string(uDec))
}
```

```
$ go run base64-encoding.go
YWJjMTIzIT8kKiYoKSctPUB+
abc123!?$*&'-'=@~
```

```
YWJjMTIzIT8kKiYoKSctPUB-
abc123!?$*&'-'=@~
```

Next example: [Reading Files](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Number Parsing

Parsing numbers from strings is a basic but common task in many programs; here's how to do it in Go.

The built-in package `strconv` provides the number parsing.

With `ParseFloat`, this 64 tells how many bits of precision to parse.

For `ParseInt`, the 0 means infer the base from the string. 64 requires that the result fit in 64 bits.

`ParseInt` will recognize hex-formatted numbers.

A `ParseUint` is also available.

`Atoi` is a convenience function for basic base-10 int parsing.

Parse functions return an error on bad input.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    f, _ := strconv.ParseFloat("1.234", 64)
    fmt.Println(f)

    i, _ := strconv.ParseInt("123", 0, 64)
    fmt.Println(i)

    d, _ := strconv.ParseInt("0x1c8", 0, 64)
    fmt.Println(d)

    u, _ := strconv.ParseUint("789", 0, 64)
    fmt.Println(u)

    k, _ := strconv.Atoi("135")
    fmt.Println(k)

    _, e := strconv.Atoi("wat")
    fmt.Println(e)
}
```

```
$ go run number-parsing.go
1.234
123
456
789
135
strconv.ParseInt: parsing "wat": invalid syntax
```

Next we'll look at another common parsing task: URLs.

Next example: [URL Parsing](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Temporary Files and Directories

Throughout program execution, we often want to create data that isn't needed after the program exits. *Temporary files and directories* are useful for this purpose since they don't pollute the file system over time.

The easiest way to create a temporary file is by calling `os.CreateTemp`. It creates a file *and* opens it for reading and writing. We provide `"` as the first argument, so `os.CreateTemp` will create the file in the default location for our OS.

Display the name of the temporary file. On Unix-based OSes the directory will likely be `/tmp`. The file name starts with the prefix given as the second argument to `os.CreateTemp` and the rest is chosen automatically to ensure that concurrent calls will always create different file names.

Clean up the file after we're done. The OS is likely to clean up temporary files by itself after some time, but it's good practice to do this explicitly.

We can write some data to the file.

If we intend to write many temporary files, we may prefer to create a temporary *directory*. `os.MkdirTemp`'s arguments are the same as `CreateTemp`'s, but it returns a directory *name* rather than an open file.

Now we can synthesize temporary file names by prefixing them with our temporary directory.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {

    f, err := os.CreateTemp("", "sample")
    check(err)

    fmt.Println("Temp file name:", f.Name())

    defer os.Remove(f.Name())

    _, err = f.Write([]byte{1, 2, 3, 4})
    check(err)

    dname, err := os.MkdirTemp("", "sampledir")
    check(err)
    fmt.Println("Temp dir name:", dname)

    defer os.RemoveAll(dname)

    fname := filepath.Join(dname, "file1")
    err = os.WriteFile(fname, []byte{1, 2}, 0666)
    check(err)
}
```

```
$ go run temporary-files-and-directories.go
Temp file name: /tmp/sample610887201
Temp dir name: /tmp/sampledir898854668
```

Next example: [Embed Directive](#).

Go by Example: Struct Embedding

Go supports *embedding* of structs and interfaces to express a more seamless *composition* of types. This is not to be confused with `//go:embed` which is a go directive introduced in Go version 1.16+ to embed files and folders into the application binary.

A container *embeds* a base. An embedding looks like a field without a name.

When creating structs with literals, we have to initialize the embedding explicitly; here the embedded type serves as the field name.

We can access the base's fields directly on `co`, e.g. `co.num`.

Alternatively, we can spell out the full path using the embedded type name.

Since `container` embeds `base`, the methods of `base` also become methods of a `container`. Here we invoke a method that was embedded from `base` directly on `co`.

Embedding structs with methods may be used to bestow interface implementations onto other structs. Here we see that a `container` now implements the `describer` interface because it embeds `base`.

```
package main

import "fmt"

type base struct {
    num int
}

func (b base) describe() string {
    return fmt.Sprintf("base with num=%v", b.num)
}

type container struct {
    base
    str string
}

func main() {

    co := container{
        base: base{
            num: 1,
        },
        str: "some name",
    }

    fmt.Printf("co={num: %v, str: %v}\n", co.num, co.str)

    fmt.Println("also num:", co.base.num)

    fmt.Println("describe:", co.describe())

    type describer interface {
        describe() string
    }

    var d describer = co
    fmt.Println("describer:", d.describe())
}
```

```
$ go run struct-embedding.go
co={num: 1, str: some name}
also num: 1
describe: base with num=1
describer: base with num=1
```

Next example: [Generics](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Command-Line Flags

Command-line flags are a common way to specify options for command-line programs. For example, in `wc -l` the `-l` is a command-line flag.

Go provides a `flag` package supporting basic command-line flag parsing. We'll use this package to implement our example command-line program.

Basic flag declarations are available for string, integer, and boolean options. Here we declare a string flag `word` with a default value `"foo"` and a short description. This `flag.String` function returns a string pointer (not a string value); we'll see how to use this pointer below.

This declares `numb` and `fork` flags, using a similar approach to the `word` flag.

It's also possible to declare an option that uses an existing var declared elsewhere in the program. Note that we need to pass in a pointer to the flag declaration function.

Once all flags are declared, call `flag.Parse()` to execute the command-line parsing.

Here we'll just dump out the parsed options and any trailing positional arguments. Note that we need to dereference the pointers with e.g. `*wordPtr` to get the actual option values.

To experiment with the command-line flags program it's best to first compile it and then run the resulting binary directly.

Try out the built program by first giving it values for all flags.

Note that if you omit flags they automatically take their default values.

Trailing positional arguments can be provided after any flags.

Note that the `flag` package requires all flags to appear before positional arguments (otherwise the flags will be interpreted as positional arguments).

Use `-h` or `--help` flags to get automatically generated help text for the command-line program.

```
package main

import (
    "flag"
    "fmt"
)

func main() {

    wordPtr := flag.String("word", "foo", "a string")

    numbPtr := flag.Int("numb", 42, "an int")
    forkPtr := flag.Bool("fork", false, "a bool")

    var svar string
    flag.StringVar(&svar, "svar", "bar", "a string var")

    flag.Parse()

    fmt.Println("word:", *wordPtr)
    fmt.Println("numb:", *numbPtr)
    fmt.Println("fork:", *forkPtr)
    fmt.Println("svar:", svar)
    fmt.Println("tail:", flag.Args())
}
```

```
$ go build command-line-flags.go
```

```
$ ./command-line-flags -word=opt -numb=7 -fork -svar=flag
word: opt
numb: 7
fork: true
svar: flag
tail: []
```

```
$ ./command-line-flags -word=opt
word: opt
numb: 42
fork: false
svar: bar
tail: []
```

```
$ ./command-line-flags -word=opt a1 a2 a3
word: opt
...
tail: [a1 a2 a3]
```

```
$ ./command-line-flags -word=opt a1 a2 a3 -numb=7
word: opt
numb: 42
fork: false
svar: bar
tail: [a1 a2 a3 -numb=7]
```

```
$ ./command-line-flags -h
Usage of ./command-line-flags:
```

If you provide a flag that wasn't specified to the flag package, the program will print an error message and show the help text again.

```
-fork=false: a bool
-numb=42: an int
-svar="bar": a string var
-word="foo": a string

$ ./command-line-flags -wat
flag provided but not defined: -wat
Usage of ./command-line-flags:
...
```

Next example: [Command-Line Subcommands](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Goroutines

A *goroutine* is a lightweight thread of execution.

Suppose we have a function call `f(s)`. Here's how we'd call that in the usual way, running it synchronously.

To invoke this function in a goroutine, use `go f(s)`. This new goroutine will execute concurrently with the calling one.

You can also start a goroutine for an anonymous function call.

Our two function calls are running asynchronously in separate goroutines now. Wait for them to finish (for a more robust approach, use a [WaitGroup](#)).

When we run this program, we see the output of the blocking call first, then the output of the two goroutines. The goroutines' output may be interleaved, because goroutines are being run concurrently by the Go runtime.

Next we'll look at a complement to goroutines in concurrent Go programs: channels.

Next example: [Channels](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "time"
)

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {
    f("direct")

    go f("goroutine")

    go func(msg string) {
        fmt.Println(msg)
    }("going")

    time.Sleep(time.Second)
    fmt.Println("done")
}
```

```
$ go run goroutines.go
direct : 0
direct : 1
direct : 2
goroutine : 0
going
goroutine : 1
goroutine : 2
done
```


Go by Example: Multiple Return Values

Go has built-in support for *multiple return values*. This feature is used often in idiomatic Go, for example to return both result and error values from a function.

The `(int, int)` in this function signature shows that the function returns 2 ints.

Here we use the 2 different return values from the call with *multiple assignment*.

If you only want a subset of the returned values, use the blank identifier `_`.

```
package main

import "fmt"

func vals() (int, int) {
    return 3, 7
}

func main() {

    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    _, c := vals()
    fmt.Println(c)
}
```

```
$ go run multiple-return-values.go
3
7
7
```

Accepting a variable number of arguments is another nice feature of Go functions; we'll look at this next.

Next example: [Variadic Functions](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Sorting

Go's `slices` package implements sorting for builtins and user-defined types. We'll look at sorting for builtins first.

Sorting functions are generic, and work for any *ordered* built-in type. For a list of ordered types, see [cmp.Ordered](#).

An example of sorting ints.

We can also use the `slices` package to check if a slice is already in sorted order.

```
package main

import (
    "fmt"
    "slices"
)

func main() {

    strs := []string{"c", "a", "b"}
    slices.Sort(strs)
    fmt.Println("Strings:", strs)

    ints := []int{7, 2, 4}
    slices.Sort(ints)
    fmt.Println("Ints:   ", ints)

    s := slices.IsSorted(ints)
    fmt.Println("Sorted: ", s)
}
```

```
$ go run sorting.go
Strings: [a b c]
Ints:    [2 4 7]
Sorted:  true
```

Next example: [Sorting by Functions](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: JSON

Go offers built-in support for JSON encoding and decoding, including to and from built-in and custom data types.

We'll use these two structs to demonstrate encoding and decoding of custom types below.

Only exported fields will be encoded/decoded in JSON. Fields must start with capital letters to be exported.

First we'll look at encoding basic data types to JSON strings. Here are some examples for atomic values.

And here are some for slices and maps, which encode to JSON arrays and objects as you'd expect.

The JSON package can automatically encode your custom data types. It will only include exported fields in the encoded output and will by default use those names as the JSON keys.

You can use tags on struct field declarations to customize the encoded JSON key names. Check the definition of `response2` above to see an example of such tags.

Now let's look at decoding JSON data into Go values. Here's an example for a generic data structure.

We need to provide a variable where the JSON package can put the decoded data. This `map[string]interface{}` will hold a map of strings to arbitrary data types.

Here's the actual decoding, and a check for associated errors.

In order to use the values in the decoded map, we'll need to convert them to their appropriate type. For example here we convert the value in `num` to the

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
    "strings"
)

type response1 struct {
    Page    int
    Fruits []string
}

type response2 struct {
    Page    int    `json:"page"`
    Fruits []string `json:"fruits"`
}

func main() {

    bolB, _ := json.Marshal(true)
    fmt.Println(string(bolB))

    intB, _ := json.Marshal(1)
    fmt.Println(string(intB))

    fltB, _ := json.Marshal(2.34)
    fmt.Println(string(fltB))

    strB, _ := json.Marshal("gopher")
    fmt.Println(string(strB))

    slcD := []string{"apple", "peach", "pear"}
    slcB, _ := json.Marshal(slcD)
    fmt.Println(string(slcB))

    mapD := map[string]int{"apple": 5, "lettuce": 7}
    mapB, _ := json.Marshal(mapD)
    fmt.Println(string(mapB))

    res1D := &response1{
        Page:    1,
        Fruits: []string{"apple", "peach", "pear"}}
    res1B, _ := json.Marshal(res1D)
    fmt.Println(string(res1B))

    res2D := &response2{
        Page:    1,
        Fruits: []string{"apple", "peach", "pear"}}
    res2B, _ := json.Marshal(res2D)
    fmt.Println(string(res2B))

    byt := []byte(`{"num":6.13,"strs":["a","b"]}`)

    var dat map[string]interface{}

    if err := json.Unmarshal(byt, &dat); err != nil {
        panic(err)
    }
    fmt.Println(dat)

    num := dat["num"].(float64)
    fmt.Println(num)
```

expected `float64` type.

Accessing nested data requires a series of conversions.

We can also decode JSON into custom data types. This has the advantages of adding additional type-safety to our programs and eliminating the need for type assertions when accessing the decoded data.

In the examples above we always used bytes and strings as intermediates between the data and JSON representation on standard out. We can also stream JSON encodings directly to `os.Writers` like `os.Stdout` or even HTTP response bodies.

Streaming reads from `os.Readers` like `os.Stdin` or HTTP request bodies is done with `json.Decoder`.

```
strs := dat["strs"].([]interface{})
str1 := strs[0].(string)
fmt.Println(str1)

str := `{"page": 1, "fruits": ["apple", "peach"]}`
res := response2{}
json.Unmarshal([]byte(str), &res)
fmt.Println(res)
fmt.Println(res.Fruits[0])

enc := json.NewEncoder(os.Stdout)
d := map[string]int{"apple": 5, "lettuce": 7}
enc.Encode(d)

dec := json.NewDecoder(strings.NewReader(str))
res1 := response2{}
dec.Decode(&res1)
fmt.Println(res1)
}
```

```
$ go run json.go
true
1
2.34
"gopher"
["apple","peach","pear"]
{"apple":5,"lettuce":7}
{"Page":1,"Fruits":["apple","peach","pear"]}
{"page":1,"fruits":["apple","peach","pear"]}
map[num:6.13 strs:[a b]]
6.13
a
{1 [apple peach]}
apple
{"apple":5,"lettuce":7}
{1 [apple peach]}
```

We've covered the basic of JSON in Go here, but check out the [JSON and Go](#) blog post and [JSON package docs](#) for more.

Next example: [XML](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example

gobyexample.com

7.6k stars 1.3k forks Branches Tags Activity

☆ Star

🔔 Notifications

<> Code Issues 10 Pull requests 5 Actions Projects 0 Security Insights

master Branches Tags

🔍 Go to file

Go to file

Code

...

📁 .github/workflows

📁 examples

📁 public

📁 templates

📁 tools

📄 .gitattributes

📄 .gitignore

📄 CONTRIBUTING.md

📄 README.md

📄 examples.txt

📄 go.mod

📄 go.sum

README

Go by Example

Content and build toolchain for [Go by Example](#), a site that teaches Go via annotated example programs.

Overview

The Go by Example site is built by extracting code and comments from source files in `examples` and rendering them using `templates` into a static `public` directory. The programs implementing this build process are in `tools`, along with dependencies specified in the `go.mod` file.

The built `public` directory can be served by any static content system. The production site uses S3 and CloudFront, for example.

Building

test passing

To build the site you'll need Go installed. Run:

```
$ tools/build
```

To build continuously in a loop:

```
$ tools/build-loop
```

To see the site locally:

```
$ tools/serve
```

and open `http://127.0.0.1:8000/` in your browser.

[🔗](#) Publishing

To upload the site:

```
$ export AWS_ACCESS_KEY_ID=...  
$ export AWS_SECRET_ACCESS_KEY=...  
$ tools/upload
```

[🔗](#) License

This work is copyright Mark McGranaghan and licensed under a [Creative Commons Attribution 3.0 Unported License](#).

The Go Gopher is copyright [Renée French](#) and licensed under a [Creative Commons Attribution 3.0 Unported License](#).

[🔗](#) Translations

Contributor translations of the Go by Example site are available in:

- [Chinese](#) by [gobyexample-cn](#)
- [French](#) by [keirua](#)
- [Japanese](#) by [spinute](#)
- [Korean](#) by [mingrammer](#)
- [Russian](#) by [badkaktus](#)
- [Ukrainian](#) by [butuzov](#)
- [Brazilian Portuguese](#) by [lcslitx](#)
- [Burmese](#) by [Set Kyar Wa Lar](#)

[🔗](#) Thanks

Thanks to [Jeremy Ashkenas](#) for [Docco](#), which inspired this project.

[🔗](#) FAQ

[🔗](#) I found a problem with the examples; what do I do?

We're very happy to fix problem reports and accept contributions! Please submit [an issue](#) or send a Pull Request. See `CONTRIBUTING.md` for more details.

[🔗](#) What version of Go is required to run these examples?

Given Go's strong [backwards compatibility guarantees](#), we expect the vast majority of examples to work on the latest released version of Go as well as many older releases going back years.

That said, some examples show off new features added in recent releases; therefore, it's recommended to try running examples with the latest officially released Go version (see Go's [release history](#) for details).

[↗](#) **I'm getting output in a different order from the example. Is the example wrong?**

Some of the examples demonstrate concurrent code which has a non-deterministic execution order. It depends on how the Go runtime schedules its goroutines and may vary by operating system, CPU architecture, or even Go version.

Similarly, examples that iterate over maps may produce items in a different order from what you're getting on your machine. This is because the order of iteration over maps in Go is [not specified and is not guaranteed to be the same from one iteration to the next](#).

It doesn't mean anything is wrong with the example. Typically the code in these examples will be insensitive to the actual order of the output; if the code is sensitive to the order - that's probably a bug - so feel free to report it.

Releases

No releases published

Packages 0

No packages published

Contributors 130

[+ 116 contributors](#)

Languages



Go by Example: Channel Synchronization

We can use channels to synchronize execution across goroutines. Here's an example of using a blocking receive to wait for a goroutine to finish. When waiting for multiple goroutines to finish, you may prefer to use a [WaitGroup](#).

This is the function we'll run in a goroutine. The done channel will be used to notify another goroutine that this function's work is done.

Send a value to notify that we're done.

Start a worker goroutine, giving it the channel to notify on.

Block until we receive a notification from the worker on the channel.

If you removed the `<- done` line from this program, the program would exit before the worker even started.

Next example: [Channel Directions](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "time"
)

func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")

    done <- true
}

func main() {
    done := make(chan bool, 1)
    go worker(done)

    <-done
}
```

\$ go run channel-synchronization.go
working...done

Go by Example: Strings and Runes

A Go string is a read-only slice of bytes. The language and the standard library treat strings specially - as containers of text encoded in [UTF-8](#). In other languages, strings are made of “characters”. In Go, the concept of a character is called a *rune* - it’s an integer that represents a Unicode code point. [This Go blog post](#) is a good introduction to the topic.

`s` is a string assigned a literal value representing the word “hello” in the Thai language. Go string literals are UTF-8 encoded text.

Since strings are equivalent to `[]byte`, this will produce the length of the raw bytes stored within.

Indexing into a string produces the raw byte values at each index. This loop generates the hex values of all the bytes that constitute the code points in `s`.

To count how many *runes* are in a string, we can use the `utf8` package. Note that the run-time of `RuneCountInString` depends on the size of the string, because it has to decode each UTF-8 rune sequentially. Some Thai characters are represented by UTF-8 code points that can span multiple bytes, so the result of this count may be surprising.

A range loop handles strings specially and decodes each rune along with its offset in the string.

We can achieve the same iteration by using the `utf8.DecodeRuneInString` function explicitly.

This demonstrates passing a rune value to a function.

Values enclosed in single quotes are *rune literals*. We can compare a rune value to a rune literal directly.

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {

    const s = "สวัสดี"

    fmt.Println("Len:", len(s))

    for i := 0; i < len(s); i++ {
        fmt.Printf("%x ", s[i])
    }
    fmt.Println()

    fmt.Println("Rune count:", utf8.RuneCountInString(s))

    for idx, runeValue := range s {
        fmt.Printf("%#U starts at %d\n", runeValue, idx)
    }

    fmt.Println("\nUsing DecodeRuneInString")
    for i, w := 0, 0; i < len(s); i += w {
        runeValue, width := utf8.DecodeRuneInString(s[i:])
        fmt.Printf("%#U starts at %d\n", runeValue, i)
        w = width

        examineRune(runeValue)
    }
}

func examineRune(r rune) {

    if r == 't' {
        fmt.Println("found tee")
    } else if r == 'า' {
        fmt.Println("found so sua")
    }
}
```

```
$ go run strings-and-runes.go
Len: 18
e0 b8 aa e0 b8 a7 e0 b8 b1 e0 b8 aa e0 b8 94 e0 b8 b5
Rune count: 6
U+0E2A 'า' starts at 0
U+0E27 'จ' starts at 3
U+0E31 'ิ' starts at 6
U+0E2A 'า' starts at 9
U+0E14 'อ' starts at 12
U+0E35 'ว' starts at 15
```

```
Using DecodeRuneInString
U+0E2A 'ᩃ' starts at 0
found so sua
U+0E27 'ᨾ' starts at 3
U+0E31 '᩠' starts at 6
U+0E2A 'ᩃ' starts at 9
found so sua
U+0E14 'ᨣ' starts at 12
U+0E35 '᩵' starts at 15
```

Next example: [Structs](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Signals

Sometimes we'd like our Go programs to intelligently handle [Unix signals](#). For example, we might want a server to gracefully shutdown when it receives a SIGTERM, or a command-line tool to stop processing input if it receives a SIGINT. Here's how to handle signals in Go with channels.

Go signal notification works by sending `os.Signal` values on a channel. We'll create a channel to receive these notifications. Note that this channel should be buffered.

`signal.Notify` registers the given channel to receive notifications of the specified signals.

We could receive from `sigs` here in the main function, but let's see how this could also be done in a separate goroutine, to demonstrate a more realistic scenario of graceful shutdown.

This goroutine executes a blocking receive for signals. When it gets one it'll print it out and then notify the program that it can finish.

The program will wait here until it gets the expected signal (as indicated by the goroutine above sending a value on `done`) and then exit.

When we run this program it will block waiting for a signal. By typing `ctrl-C` (which the terminal shows as `^C`) we can send a SIGINT signal, causing the program to print `interrupt` and then exit.

Next example: [Exit](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
)

func main() {

    sigs := make(chan os.Signal, 1)

    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)

    done := make(chan bool, 1)

    go func() {

        sig := <-sigs
        fmt.Println()
        fmt.Println(sig)
        done <- true
    }()

    fmt.Println("awaiting signal")
    <-done
    fmt.Println("exiting")
}
```

```
$ go run signals.go
awaiting signal
^C
interrupt
exiting
```

Go by Example: XML

Go offers built-in support for XML and XML-like formats with the `encoding/xml` package.

Plant will be mapped to XML. Similarly to the JSON examples, field tags contain directives for the encoder and decoder. Here we use some special features of the XML package: the `XMLName` field name dictates the name of the XML element representing this struct; `id,attr` means that the `Id` field is an XML *attribute* rather than a nested element.

Emit XML representing our plant; using `MarshalIndent` to produce a more human-readable output.

To add a generic XML header to the output, append it explicitly.

Use `Unmarshal` to parse a stream of bytes with XML into a data structure. If the XML is malformed or cannot be mapped onto `Plant`, a descriptive error will be returned.

The `parent>child>plant` field tag tells the encoder to nest all plants under `<parent><child>...`

```
package main

import (
    "encoding/xml"
    "fmt"
)

type Plant struct {
    XMLName xml.Name `xml:"plant"`
    Id      int    `xml:"id,attr"`
    Name    string `xml:"name"`
    Origin  []string `xml:"origin"`
}

func (p Plant) String() string {
    return fmt.Sprintf("Plant id=%v, name=%v, origin=%v",
        p.Id, p.Name, p.Origin)
}

func main() {
    coffee := &Plant{Id: 27, Name: "Coffee"}
    coffee.Origin = []string{"Ethiopia", "Brazil"}

    out, _ := xml.MarshalIndent(coffee, " ", " ")
    fmt.Println(string(out))

    fmt.Println(xml.Header + string(out))

    var p Plant
    if err := xml.Unmarshal(out, &p); err != nil {
        panic(err)
    }
    fmt.Println(p)

    tomato := &Plant{Id: 81, Name: "Tomato"}
    tomato.Origin = []string{"Mexico", "California"}

    type Nesting struct {
        XMLName xml.Name `xml:"nesting"`
        Plants  []*Plant `xml:"parent>child>plant"`
    }

    nesting := &Nesting{}
    nesting.Plants = []*Plant{coffee, tomato}

    out, _ = xml.MarshalIndent(nesting, " ", " ")
    fmt.Println(string(out))
}
```

```
$ go run xml.go
<plant id="27">
  <name>Coffee</name>
  <origin>Ethiopia</origin>
  <origin>Brazil</origin>
</plant>
<?xml version="1.0" encoding="UTF-8"?>
<plant id="27">
  <name>Coffee</name>
  <origin>Ethiopia</origin>
  <origin>Brazil</origin>
</plant>
Plant id=27, name=Coffee, origin=[Ethiopia Brazil]
<nesting>
  <parent>
    <child>
```

```
<plant id="27">
  <name>Coffee</name>
  <origin>Ethiopia</origin>
  <origin>Brazil</origin>
</plant>
<plant id="81">
  <name>Tomato</name>
  <origin>Mexico</origin>
  <origin>California</origin>
</plant>
</child>
</parent>
</nesting>
```

Next example: [Time](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Arrays

In Go, an *array* is a numbered sequence of elements of a specific length. In typical Go code, [slices](#) are much more common; arrays are useful in some special scenarios.

Here we create an array `a` that will hold exactly 5 ints. The type of elements and length are both part of the array's type. By default an array is zero-valued, which for ints means 0s.

We can set a value at an index using the `array[index] = value` syntax, and get a value with `array[index]`.

The builtin `len` returns the length of an array.

Use this syntax to declare and initialize an array in one line.

You can also have the compiler count the number of elements for you with `...`

If you specify the index with `:`, the elements in between will be zeroed.

Array types are one-dimensional, but you can compose types to build multi-dimensional data structures.

You can create and initialize multi-dimensional arrays at once too.

Note that arrays appear in the form `[v1 v2 v3 ...]` when printed with `fmt.Println`.

```
package main

import "fmt"

func main() {

    var a [5]int
    fmt.Println("emp:", a)

    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    fmt.Println("len:", len(a))

    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    b = [...]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    b = [...]int{100, 3: 400, 500}
    fmt.Println("idx:", b)

    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)

    twoD = [2][3]int{
        {1, 2, 3},
        {1, 2, 3},
    }
    fmt.Println("2d: ", twoD)
}
```

```
$ go run arrays.go
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
dcl: [1 2 3 4 5]
idx: [100 0 0 400 500]
2d: [[0 1 2] [1 2 3]]
2d: [[1 2 3] [1 2 3]]
```

Next example: [Slices](#).

Go by Example: Channel Directions

When using channels as function parameters, you can specify if a channel is meant to only send or receive values. This specificity increases the type-safety of the program.

This ping function only accepts a channel for sending values. It would be a compile-time error to try to receive on this channel.

The pong function accepts one channel for receives (pings) and a second for sends (pongs).

```
package main

import "fmt"

func ping(pings chan<- string, msg string) {
    pings <- msg
}

func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

```
$ go run channel-directions.go
passed message
```

Next example: [Select](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Range over Iterators

Starting with version 1.23, Go has added support for [iterators](#), which lets us range over pretty much anything!

Let's look at the `List` type from the [previous example](#) again. In that example we had an `AllElements` method that returned a slice of all elements in the list. With Go iterators, we can do it better - as shown below.

`All` returns an *iterator*, which in Go is a function with a [special signature](#).

The iterator function takes another function as a parameter, called `yield` by convention (but the name can be arbitrary). It will call `yield` for every element we want to iterate over, and note `yield`'s return value for a potential early termination.

Iteration doesn't require an underlying data structure, and doesn't even have to be finite! Here's a function returning an iterator over Fibonacci numbers: it keeps running as long as `yield` keeps returning `true`.

Since `List.All` returns an iterator, we can use it in a regular range loop.

Packages like [slices](#) have a number of useful functions to work with iterators. For example, `Collect` takes any iterator and collects all its values into a slice.

```
package main

import (
    "fmt"
    "iter"
    "slices"
)

type List[T any] struct {
    head, tail *element[T]
}

type element[T any] struct {
    next *element[T]
    val  T
}

func (lst *List[T]) Push(v T) {
    if lst.tail == nil {
        lst.head = &element[T]{val: v}
        lst.tail = lst.head
    } else {
        lst.tail.next = &element[T]{val: v}
        lst.tail = lst.tail.next
    }
}

func (lst *List[T]) All() iter.Seq[T] {
    return func(yield func(T) bool) {

        for e := lst.head; e != nil; e = e.next {
            if !yield(e.val) {
                return
            }
        }
    }
}

func genFib() iter.Seq[int] {
    return func(yield func(int) bool) {
        a, b := 1, 1

        for {
            if !yield(a) {
                return
            }
            a, b = b, a+b
        }
    }
}

func main() {
    lst := List[int]{}
    lst.Push(10)
    lst.Push(13)
    lst.Push(23)

    for e := range lst.All() {
        fmt.Println(e)
    }

    all := slices.Collect(lst.All())
    fmt.Println("all:", all)
}
```


Once the loop hits break or an early return, the yield function passed to the iterator will return false.

```
for n := range genFib() {  
    if n >= 10 {  
        break  
    }  
    fmt.Println(n)  
}
```

```
$ go run range-over-iterators.go  
10  
13  
23  
all: [10 13 23]  
1  
1  
2  
3  
5  
8
```

Next example: [Errors](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Generics

Starting with version 1.18, Go has added support for *generics*, also known as *type parameters*.

As an example of a generic function, `SlicesIndex` takes a slice of any comparable type and an element of that type and returns the index of the first occurrence of `v` in `s`, or `-1` if not present. The comparable constraint means that we can compare values of this type with the `==` and `!=` operators. For a more thorough explanation of this type signature, see [this blog post](#). Note that this function exists in the standard library as [slices.Index](#).

As an example of a generic type, `List` is a singly-linked list with values of any type.

We can define methods on generic types just like we do on regular types, but we have to keep the type parameters in place. The type is `List[T]`, not `List`.

`AllElements` returns all the `List` elements as a slice. In the next example we'll see a more idiomatic way of iterating over all elements of custom types.

When invoking generic functions, we can often rely on *type inference*. Note that we don't have to specify the types for `S` and `E` when calling `SlicesIndex` - the compiler infers them automatically.

... though we could also specify them explicitly.

```
package main

import "fmt"

func SlicesIndex[S ~[]E, E comparable](s S, v E) int {
    for i := range s {
        if v == s[i] {
            return i
        }
    }
    return -1
}

type List[T any] struct {
    head, tail *element[T]
}

type element[T any] struct {
    next *element[T]
    val  T
}

func (lst *List[T]) Push(v T) {
    if lst.tail == nil {
        lst.head = &element[T]{val: v}
        lst.tail = lst.head
    } else {
        lst.tail.next = &element[T]{val: v}
        lst.tail = lst.tail.next
    }
}

func (lst *List[T]) AllElements() []T {
    var elems []T
    for e := lst.head; e != nil; e = e.next {
        elems = append(elems, e.val)
    }
    return elems
}

func main() {
    var s = []string{"foo", "bar", "zoo"}

    fmt.Println("index of zoo:", SlicesIndex(s, "zoo"))

    _ = SlicesIndex[[]string, string](s, "zoo")

    lst := List[int]{}
    lst.Push(10)
    lst.Push(13)
    lst.Push(23)
    fmt.Println("list:", lst.AllElements())
}
```

```
$ go run generics.go
index of zoo: 2
list: [10 13 23]
```

Next example: [Range over Iterators](#).

Go by Example: Recover

Go makes it possible to *recover* from a panic, by using the `recover` built-in function. A `recover` can stop a panic from aborting the program and let it continue with execution instead.

An example of where this can be useful: a server wouldn't want to crash if one of the client connections exhibits a critical error. Instead, the server would want to close that connection and continue serving other clients. In fact, this is what Go's `net/http` does by default for HTTP servers.

This function panics.

`recover` must be called within a deferred function. When the enclosing function panics, the `defer` will activate and a `recover` call within it will catch the panic.

The return value of `recover` is the error raised in the call to `panic`.

This code will not run, because `mayPanic` panics. The execution of `main` stops at the point of the panic and resumes in the deferred closure.

```
package main

import "fmt"

func mayPanic() {
    panic("a problem")
}

func main() {

    defer func() {
        if r := recover(); r != nil {

            fmt.Println("Recovered. Error:\n", r)
        }
    }()

    mayPanic()

    fmt.Println("After mayPanic()")
}
```

```
$ go run recover.go
Recovered. Error:
a problem
```

Next example: [String Functions](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Methods

Go supports *methods* defined on struct types.

This area method has a *receiver type* of `*rect`.

Methods can be defined for either pointer or value receiver types. Here's an example of a value receiver.

Here we call the 2 methods defined for our struct.

Go automatically handles conversion between values and pointers for method calls. You may want to use a pointer receiver type to avoid copying on method calls or to allow the method to mutate the receiving struct.

Next we'll look at Go's mechanism for grouping and naming related sets of methods: interfaces.

Next example: [Interfaces](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import "fmt"

type rect struct {
    width, height int
}

func (r *rect) area() int {
    return r.width * r.height
}

func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

func main() {
    r := rect{width: 10, height: 5}

    fmt.Println("area: ", r.area())
    fmt.Println("perim:", r.perim())

    rp := &r
    fmt.Println("area: ", rp.area())
    fmt.Println("perim:", rp.perim())
}
```

```
$ go run methods.go
area: 50
perim: 30
area: 50
perim: 30
```

Go by Example: Select

Go's *select* lets you wait on multiple channel operations. Combining goroutines and channels with *select* is a powerful feature of Go.

For our example we'll select across two channels.

Each channel will receive a value after some amount of time, to simulate e.g. blocking RPC operations executing in concurrent goroutines.

We'll use *select* to await both of these values simultaneously, printing each one as it arrives.

We receive the values "one" and then "two" as expected.

Note that the total execution time is only ~2 seconds since both the 1 and 2 second *Sleeps* execute concurrently.

Next example: [Timeouts](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "time"
)

func main() {

    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(1 * time.Second)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "two"
    }()

    for i := 0; i < 2; i++ {
        select {
            case msg1 := <-c1:
                fmt.Println("received", msg1)
            case msg2 := <-c2:
                fmt.Println("received", msg2)
        }
    }
}
```

```
$ time go run select.go
received one
received two

real    0m2.245s
```

Go by Example: Mutexes

In the previous example we saw how to manage simple counter state using [atomic operations](#). For more complex state we can use a [mutex](#) to safely access data across multiple goroutines.

Container holds a map of counters; since we want to update it concurrently from multiple goroutines, we add a Mutex to synchronize access. Note that mutexes must not be copied, so if this struct is passed around, it should be done by pointer.

Lock the mutex before accessing counters; unlock it at the end of the function using a [defer](#) statement.

Note that the zero value of a mutex is usable as-is, so no initialization is required here.

This function increments a named counter in a loop.

Run several goroutines concurrently; note that they all access the same Container, and two of them access the same counter.

Wait for the goroutines to finish

Running the program shows that the counters updated as expected.

Next we'll look at implementing this same state management task using only goroutines and channels.

Next example: [Stateful Goroutines](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "sync"
)

type Container struct {
    mu      sync.Mutex
    counters map[string]int
}

func (c *Container) inc(name string) {

    c.mu.Lock()
    defer c.mu.Unlock()
    c.counters[name]++
}

func main() {
    c := Container{

        counters: map[string]int{"a": 0, "b": 0},
    }

    var wg sync.WaitGroup

    doIncrement := func(name string, n int) {
        for i := 0; i < n; i++ {
            c.inc(name)
        }
        wg.Done()
    }

    wg.Add(3)
    go doIncrement("a", 10000)
    go doIncrement("a", 10000)
    go doIncrement("b", 10000)

    wg.Wait()
    fmt.Println(c.counters)
}
```

```
$ go run mutexes.go
map[a:20000 b:10000]
```

Go by Example: Directories

Go has several useful functions for working with *directories* in the file system.

Create a new sub-directory in the current working directory.

When creating temporary directories, it's good practice to defer their removal. `os.RemoveAll` will delete a whole directory tree (similarly to `rm -rf`).

Helper function to create a new empty file.

We can create a hierarchy of directories, including parents with `MkdirAll`. This is similar to the command-line `mkdir -p`.

`ReadDir` lists directory contents, returning a slice of `os.DirEntry` objects.

`Chdir` lets us change the current working directory, similarly to `cd`.

Now we'll see the contents of `subdir/parent/child` when listing the *current* directory.

`cd` back to where we started.

We can also visit a directory *recursively*, including all its sub-directories. `WalkDir` accepts a callback function to handle every file or directory visited.

`visit` is called for every file or directory found recursively by `filepath.WalkDir`.

```
package main

import (
    "fmt"
    "io/fs"
    "os"
    "path/filepath"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {

    err := os.Mkdir("subdir", 0755)
    check(err)

    defer os.RemoveAll("subdir")

    createEmptyFile := func(name string) {
        d := []byte("")
        check(os.WriteFile(name, d, 0644))
    }

    createEmptyFile("subdir/file1")

    err = os.MkdirAll("subdir/parent/child", 0755)
    check(err)

    createEmptyFile("subdir/parent/file2")
    createEmptyFile("subdir/parent/file3")
    createEmptyFile("subdir/parent/child/file4")

    c, err := os.ReadDir("subdir/parent")
    check(err)

    fmt.Println("Listing subdir/parent")
    for _, entry := range c {
        fmt.Println(" ", entry.Name(), entry.IsDir())
    }

    err = os.Chdir("subdir/parent/child")
    check(err)

    c, err = os.ReadDir(".")
    check(err)

    fmt.Println("Listing subdir/parent/child")
    for _, entry := range c {
        fmt.Println(" ", entry.Name(), entry.IsDir())
    }

    err = os.Chdir("../..")
    check(err)

    fmt.Println("Visiting subdir")
    err = filepath.WalkDir("subdir", visit)
}

func visit(path string, d fs.DirEntry, err error) error {
    if err != nil {
        return err
    }
}
```



```
}  
    fmt.Println(" ", path, d.IsDir())  
    return nil  
}
```

```
$ go run directories.go  
Listing subdir/parent  
  child true  
  file2 false  
  file3 false  
Listing subdir/parent/child  
  file4 false  
Visiting subdir  
  subdir true  
  subdir/file1 false  
  subdir/parent true  
  subdir/parent/child true  
  subdir/parent/child/file4 false  
  subdir/parent/file2 false  
  subdir/parent/file3 false
```

Next example: [Temporary Files and Directories](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Custom Errors

It's possible to use custom types as errors by implementing the `Error()` method on them. Here's a variant on the example above that uses a custom type to explicitly represent an argument error.

A custom error type usually has the suffix "Error".

Adding this `Error` method makes `argError` implement the error interface.

Return our custom error.

`errors.As` is a more advanced version of `errors.Is`. It checks that a given error (or any error in its chain) matches a specific error type and converts to a value of that type, returning `true`. If there's no match, it returns `false`.

```
package main

import (
    "errors"
    "fmt"
)

type argError struct {
    arg    int
    message string
}

func (e *argError) Error() string {
    return fmt.Sprintf("%d - %s", e.arg, e.message)
}

func f(arg int) (int, error) {
    if arg == 42 {
        return -1, &argError{arg, "can't work with it"}
    }
    return arg + 3, nil
}

func main() {
    _, err := f(42)
    var ae *argError
    if errors.As(err, &ae) {
        fmt.Println(ae.arg)
        fmt.Println(ae.message)
    } else {
        fmt.Println("err doesn't match argError")
    }
}
```

```
$ go run custom-errors.go
42
can't work with it
```

Next example: [Goroutines](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Pointers

Go supports *pointers*, allowing you to pass references to values and records within your program.

We'll show how pointers work in contrast to values with 2 functions: `zeroval` and `zeroptr`. `zeroval` has an `int` parameter, so arguments will be passed to it by value. `zeroval` will get a copy of `ival` distinct from the one in the calling function.

`zeroptr` in contrast has an `*int` parameter, meaning that it takes an `int` pointer. The `*iptr` code in the function body then *dereferences* the pointer from its memory address to the current value at that address. Assigning a value to a dereferenced pointer changes the value at the referenced address.

The `&i` syntax gives the memory address of `i`, i.e. a pointer to `i`.

Pointers can be printed too.

`zeroval` doesn't change the `i` in `main`, but `zeroptr` does because it has a reference to the memory address for that variable.

Next example: [Strings and Runes](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import "fmt"

func zeroval(ival int) {
    ival = 0
}

func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    zeroptr(&i)
    fmt.Println("zeroptr:", i)

    fmt.Println("pointer:", &i)
}
```

```
$ go run pointers.go
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0x42131100
```

Go by Example: Epoch

A common requirement in programs is getting the number of seconds, milliseconds, or nanoseconds since the [Unix epoch](#). Here's how to do it in Go.

Use `time.Now` with `Unix`, `UnixMilli` or `UnixNano` to get elapsed time since the Unix epoch in seconds, milliseconds or nanoseconds, respectively.

You can also convert integer seconds or nanoseconds since the epoch into the corresponding time.

Next we'll look at another time-related task: time parsing and formatting.

Next example: [Time Formatting / Parsing](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "time"
)

func main() {

    now := time.Now()
    fmt.Println(now)

    fmt.Println(now.Unix())
    fmt.Println(now.UnixMilli())
    fmt.Println(now.UnixNano())

    fmt.Println(time.Unix(now.Unix(), 0))
    fmt.Println(time.Unix(0, now.UnixNano()))
}
```

```
$ go run epoch.go
2012-10-31 16:13:58.292387 +0000 UTC
1351700038
1351700038292
1351700038292387000
2012-10-31 16:13:58 +0000 UTC
2012-10-31 16:13:58.292387 +0000 UTC
```

Go by Example: Spawning Processes

Sometimes our Go programs need to spawn other, non-Go processes.

We'll start with a simple command that takes no arguments or input and just prints something to stdout. The `exec.Command` helper creates an object to represent this external process.

The `Output` method runs the command, waits for it to finish and collects its standard output. If there were no errors, `dateOut` will hold bytes with the date info.

`Output` and other methods of `Command` will return `*exec.Error` if there was a problem executing the command (e.g. wrong path), and `*exec.ExitError` if the command ran but exited with a non-zero return code.

Next we'll look at a slightly more involved case where we pipe data to the external process on its `stdin` and collect the results from its `stdout`.

Here we explicitly grab input/output pipes, start the process, write some input to it, read the resulting output, and finally wait for the process to exit.

We omitted error checks in the above example, but you could use the usual `if err != nil` pattern for all of them. We also only collect the `StdoutPipe` results, but you could collect the `StderrPipe` in exactly the same way.

Note that when spawning commands we need to provide an explicitly delineated command and argument array, vs. being able to just pass in one command-line string. If you want to spawn a full command with a string, you can use `bash`'s `-c` option:

The spawned programs return output that is the same as if we had run them directly from the command-line.

`date` doesn't have a `-x` flag so it will exit with an error message and non-zero return code.

```
package main

import (
    "fmt"
    "io"
    "os/exec"
)

func main() {

    dateCmd := exec.Command("date")

    dateOut, err := dateCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> date")
    fmt.Println(string(dateOut))

    _, err = exec.Command("date", "-x").Output()
    if err != nil {
        switch e := err.(type) {
        case *exec.Error:
            fmt.Println("failed executing:", err)
        case *exec.ExitError:
            fmt.Println("command exit rc =", e.ExitCode())
        default:
            panic(err)
        }
    }

    grepCmd := exec.Command("grep", "hello")

    grepIn, _ := grepCmd.StdinPipe()
    grepOut, _ := grepCmd.StdoutPipe()
    grepCmd.Start()
    grepIn.Write([]byte("hello grep\ngoodbye grep"))
    grepIn.Close()
    grepBytes, _ := io.ReadAll(grepOut)
    grepCmd.Wait()

    fmt.Println("> grep hello")
    fmt.Println(string(grepBytes))

    lsCmd := exec.Command("bash", "-c", "ls -a -l -h")
    lsOut, err := lsCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> ls -a -l -h")
    fmt.Println(string(lsOut))
}
```

```
$ go run spawning-processes.go
> date
Thu 05 May 2022 10:10:12 PM PDT

command exited with rc = 1
> grep hello
```

```
hello grep
```

```
> ls -a -l -h
```

```
drwxr-xr-x  4 mark 136B Oct 3 16:29 .
```

```
drwxr-xr-x 91 mark 3.0K Oct 3 12:50 ..
```

```
-rw-r--r--  1 mark 1.3K Oct 3 16:28 spawning-processes.go
```

Next example: [Exec'ing Processes](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Atomic Counters

The primary mechanism for managing state in Go is communication over channels. We saw this for example with [worker pools](#). There are a few other options for managing state though. Here we'll look at using the `sync/atomic` package for *atomic counters* accessed by multiple goroutines.

We'll use an atomic integer type to represent our (always-positive) counter.

A `WaitGroup` will help us wait for all goroutines to finish their work.

We'll start 50 goroutines that each increment the counter exactly 1000 times.

To atomically increment the counter we use `Add`.

Wait until all the goroutines are done.

Here no goroutines are writing to 'ops', but using `Load` it's safe to atomically read a value even while other goroutines are (atomically) updating it.

We expect to get exactly 50,000 operations. Had we used a non-atomic integer and incremented it with `ops++`, we'd likely get a different number, changing between runs, because the goroutines would interfere with each other. Moreover, we'd get data race failures when running with the `-race` flag.

Next we'll look at mutexes, another tool for managing state.

Next example: [Mutexes](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

func main() {

    var ops atomic.Uint64

    var wg sync.WaitGroup

    for i := 0; i < 50; i++ {
        wg.Add(1)

        go func() {
            for c := 0; c < 1000; c++ {

                ops.Add(1)
            }

            wg.Done()
        }()
    }

    wg.Wait()

    fmt.Println("ops:", ops.Load())
}
```

```
$ go run atomic-counters.go
ops: 50000
```

Go by Example: Timers

We often want to execute Go code at some point in the future, or repeatedly at some interval. Go's built-in *timer* and *ticker* features make both of these tasks easy. We'll look first at timers and then at [tickers](#).

Timers represent a single event in the future. You tell the timer how long you want to wait, and it provides a channel that will be notified at that time. This timer will wait 2 seconds.

The `<-timer1.C` blocks on the timer's channel `C` until it sends a value indicating that the timer fired.

If you just wanted to wait, you could have used `time.Sleep`. One reason a timer may be useful is that you can cancel the timer before it fires. Here's an example of that.

Give the `timer2` enough time to fire, if it ever was going to, to show it is in fact stopped.

The first timer will fire ~2s after we start the program, but the second should be stopped before it has a chance to fire.

Next example: [Tickers](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "time"
)

func main() {

    timer1 := time.NewTimer(2 * time.Second)

    <-timer1.C
    fmt.Println("Timer 1 fired")

    timer2 := time.NewTimer(time.Second)
    go func() {
        <-timer2.C
        fmt.Println("Timer 2 fired")
    }()
    stop2 := timer2.Stop()
    if stop2 {
        fmt.Println("Timer 2 stopped")
    }

    time.Sleep(2 * time.Second)
}
```

```
$ go run timers.go
Timer 1 fired
Timer 2 stopped
```


Go by Example

Go is an open source programming language designed for building scalable, secure and reliable software. Please read the [official documentation](#) to learn more.

Go by Example is a hands-on introduction to Go using annotated example programs. Check out the [first example](#) or browse the full list below.

Unless stated otherwise, examples here assume the latest major release Go and may use new language features. Try to upgrade to the latest version if something isn't working.

- [Hello World](#)
- [Values](#)
- [Variables](#)
- [Constants](#)
- [For](#)
- [If/Else](#)
- [Switch](#)
- [Arrays](#)
- [Slices](#)
- [Maps](#)
- [Functions](#)
- [Multiple Return Values](#)
- [Variadic Functions](#)
- [Closures](#)
- [Recursion](#)
- [Range over Built-in Types](#)
- [Pointers](#)
- [Strings and Runes](#)
- [Structs](#)
- [Methods](#)
- [Interfaces](#)
- [Enums](#)
- [Struct Embedding](#)
- [Generics](#)
- [Range over Iterators](#)
- [Errors](#)
- [Custom Errors](#)
- [Goroutines](#)
- [Channels](#)
- [Channel Buffering](#)
- [Channel Synchronization](#)
- [Channel Directions](#)
- [Select](#)
- [Timeouts](#)
- [Non-Blocking Channel Operations](#)
- [Closing Channels](#)
- [Range over Channels](#)
- [Timers](#)
- [Tickers](#)
- [Worker Pools](#)
- [WaitGroups](#)
- [Rate Limiting](#)
- [Atomic Counters](#)
- [Mutexes](#)
- [Stateful Goroutines](#)
- [Sorting](#)
- [Sorting by Functions](#)
- [Panic](#)
- [Defer](#)
- [Recover](#)
- [String Functions](#)
- [String Formatting](#)
- [Text Templates](#)

[Regular Expressions](#)
[JSON](#)
[XML](#)
[Time](#)
[Epoch](#)
[Time Formatting / Parsing](#)
[Random Numbers](#)
[Number Parsing](#)
[URL Parsing](#)
[SHA256 Hashes](#)
[Base64 Encoding](#)
[Reading Files](#)
[Writing Files](#)
[Line Filters](#)
[File Paths](#)
[Directories](#)
[Temporary Files and Directories](#)
[Embed Directive](#)
[Testing and Benchmarking](#)
[Command-Line Arguments](#)
[Command-Line Flags](#)
[Command-Line Subcommands](#)
[Environment Variables](#)
[Logging](#)
[HTTP Client](#)
[HTTP Server](#)
[Context](#)
[Spawning Processes](#)
[Exec'ing Processes](#)
[Signals](#)
[Exit](#)

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example

gobyexample.com

7.6k stars 1.3k forks Branches Tags Activity

☆ Star

🔔 Notifications

<> Code Issues 10 Pull requests 5 Actions Projects 0 Security Insights

master Branches Tags

🔍 Go to file

Go to file

Code

...

📁 .github/workflows

📁 examples

📁 public

📁 templates

📁 tools

📄 .gitattributes

📄 .gitignore

📄 CONTRIBUTING.md

📄 README.md

📄 examples.txt

📄 go.mod

📄 go.sum

README

Go by Example

Content and build toolchain for [Go by Example](#), a site that teaches Go via annotated example programs.

Overview

The Go by Example site is built by extracting code and comments from source files in `examples` and rendering them using `templates` into a static `public` directory. The programs implementing this build process are in `tools`, along with dependencies specified in the `go.mod` file.

The built `public` directory can be served by any static content system. The production site uses S3 and CloudFront, for example.

Building

test passing

To build the site you'll need Go installed. Run:

```
$ tools/build
```

To build continuously in a loop:

```
$ tools/build-loop
```

To see the site locally:

```
$ tools/serve
```

and open `http://127.0.0.1:8000/` in your browser.

[🔗](#) Publishing

To upload the site:

```
$ export AWS_ACCESS_KEY_ID=...  
$ export AWS_SECRET_ACCESS_KEY=...  
$ tools/upload
```

[🔗](#) License

This work is copyright Mark McGranaghan and licensed under a [Creative Commons Attribution 3.0 Unported License](#).

The Go Gopher is copyright [Renée French](#) and licensed under a [Creative Commons Attribution 3.0 Unported License](#).

[🔗](#) Translations

Contributor translations of the Go by Example site are available in:

- [Chinese](#) by [gobyexample-cn](#)
- [French](#) by [keirua](#)
- [Japanese](#) by [spinute](#)
- [Korean](#) by [mingrammer](#)
- [Russian](#) by [badkaktus](#)
- [Ukrainian](#) by [butuzov](#)
- [Brazilian Portuguese](#) by [lcslitx](#)
- [Burmese](#) by [Set Kyar Wa Lar](#)

[🔗](#) Thanks

Thanks to [Jeremy Ashkenas](#) for [Docco](#), which inspired this project.

[🔗](#) FAQ

[🔗](#) I found a problem with the examples; what do I do?

We're very happy to fix problem reports and accept contributions! Please submit [an issue](#) or send a Pull Request. See `CONTRIBUTING.md` for more details.

[🔗](#) What version of Go is required to run these examples?

Given Go's strong [backwards compatibility guarantees](#), we expect the vast majority of examples to work on the latest released version of Go as well as many older releases going back years.

That said, some examples show off new features added in recent releases; therefore, it's recommended to try running examples with the latest officially released Go version (see Go's [release history](#) for details).

[↗](#) **I'm getting output in a different order from the example. Is the example wrong?**

Some of the examples demonstrate concurrent code which has a non-deterministic execution order. It depends on how the Go runtime schedules its goroutines and may vary by operating system, CPU architecture, or even Go version.

Similarly, examples that iterate over maps may produce items in a different order from what you're getting on your machine. This is because the order of iteration over maps in Go is [not specified and is not guaranteed to be the same from one iteration to the next](#).

It doesn't mean anything is wrong with the example. Typically the code in these examples will be insensitive to the actual order of the output; if the code is sensitive to the order - that's probably a bug - so feel free to report it.

Releases

No releases published

Packages 0

No packages published

Contributors 130

[+ 116 contributors](#)

Languages



Go by Example: Time

Go offers extensive support for times and durations; here are some examples.

We'll start by getting the current time.

You can build a time struct by providing the year, month, day, etc. Times are always associated with a Location, i.e. time zone.

You can extract the various components of the time value as expected.

The Monday-Sunday Weekday is also available.

These methods compare two times, testing if the first occurs before, after, or at the same time as the second, respectively.

The Sub methods returns a Duration representing the interval between two times.

We can compute the length of the duration in various units.

You can use Add to advance a time by a given duration, or with a - to move backwards by a duration.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    p := fmt.Println

    now := time.Now()
    p(now)

    then := time.Date(
        2009, 11, 17, 20, 34, 58, 651387237, time.UTC)
    p(then)

    p(then.Year())
    p(then.Month())
    p(then.Day())
    p(then.Hour())
    p(then.Minute())
    p(then.Second())
    p(then.Nanosecond())
    p(then.Location())

    p(then.Weekday())

    p(then.Before(now))
    p(then.After(now))
    p(then.Equal(now))

    diff := now.Sub(then)
    p(diff)

    p(diff.Hours())
    p(diff.Minutes())
    p(diff.Seconds())
    p(diff.Nanoseconds())

    p(then.Add(diff))
    p(then.Add(-diff))
}
```

```
$ go run time.go
2012-10-31 15:50:13.793654 +0000 UTC
2009-11-17 20:34:58.651387237 +0000 UTC
2009
November
17
20
34
58
651387237
UTC
Tuesday
true
false
false
25891h15m15.142266763s
25891.25420618521
1.5534752523711128e+06
9.320851514226677e+07
93208515142266763
2012-10-31 15:50:13.793654 +0000 UTC
2006-12-05 01:19:43.509120474 +0000 UTC
```

Next we'll look at the related idea of time relative to the Unix epoch.

Next example: [Epoch](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Tutorial: Get started with Go

In this tutorial, you'll get a brief introduction to Go programming. Along the way, you will:

- Install Go (if you haven't already).
- Write some simple "Hello, world" code.
- Use the `go` command to run your code.
- Use the Go package discovery tool to find packages you can use in your own code.
- Call functions of an external module.

Note: For other tutorials, see [Tutorials](#).

Prerequisites

- **Some programming experience.** The code here is pretty simple, but it helps to know something about functions.
- **A tool to edit your code.** Any text editor you have will work fine. Most text editors have good support for Go. The most popular are VSCode (free), GoLand (paid), and Vim (free).
- **A command terminal.** Go works well using any terminal on Linux and Mac, and on PowerShell or `cmd` in Windows.

Install Go

Just use the [Download and install](#) steps.

Write some code

Get started with Hello, World.

1. Open a command prompt and `cd` to your home directory.

On Linux or Mac:

```
cd
```

On Windows:

```
cd %HOMEPATH%
```

2. Create a `hello` directory for your first Go source code.

For example, use the following commands:

```
mkdir hello
cd hello
```

3. Enable dependency tracking for your code.

When your code imports packages contained in other modules, you manage those dependencies through your code's own module. That module is defined by a `go.mod` file that tracks the modules that provide those packages. That `go.mod` file stays with your code, including in your source code repository.

To enable dependency tracking for your code by creating a `go.mod` file, run the `go mod init` command, giving it the name of the module your code will be in. The name is the module's module path.

In actual development, the module path will typically be the repository location where your source code will be kept. For example, the module path might be `github.com/mymodule`. If you plan to publish your module for others to use, the module path *must* be a location from which Go tools can download your module. For more about naming a module with a module path, see [Managing dependencies](#).

For the purposes of this tutorial, just use `example/hello`.

```
$ go mod init example/hello
go: creating new go.mod: module example/hello
```


4. In your text editor, create a file `hello.go` in which to write your code.
5. Paste the following code into your `hello.go` file and save the file.

```
package main

import "fmt"

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

This is your Go code. In this code, you:

- Declare a main package (a package is a way to group functions, and it's made up of all the files in the same directory).
 - Import the popular [fmt package](#), which contains functions for formatting text, including printing to the console. This package is one of the [standard library](#) packages you got when you installed Go.
 - Implement a main function to print a message to the console. A main function executes by default when you run the main package.
6. Run your code to see the greeting.

```
$ go run .
Hello, World!
```

The `go run` command is one of many go commands you'll use to get things done with Go. Use the following command to get a list of the others:

```
$ go help
```

Call code in an external package

When you need your code to do something that might have been implemented by someone else, you can look for a package that has functions you can use in your code.

1. Make your printed message a little more interesting with a function from an external module.
 1. Visit [pkg.go.dev](#) and [search for a "quote" package](#).
 2. Locate and click the `rsc.io/quote` package in search results (if you see `rsc.io/quote/v3`, ignore it for now).
 3. In the **Documentation** section, under **Index**, note the list of functions you can call from your code. You'll use the `Go` function.
 4. At the top of this page, note that package `quote` is included in the `rsc.io/quote` module.

You can use the [pkg.go.dev](#) site to find published modules whose packages have functions you can use in your own code. Packages are published in modules -- like `rsc.io/quote` -- where others can use them. Modules are improved with new versions over time, and you can upgrade your code to use the improved versions.

2. In your Go code, import the `rsc.io/quote` package and add a call to its Go function.

After adding the highlighted lines, your code should include the following:

```
package main

import "fmt"

import "rsc.io/quote"

func main() {
    fmt.Println(quote.Go())
}
```

3. Add new module requirements and sums.

Go will add the `quote` module as a requirement, as well as a `go.sum` file for use in authenticating the module. For more, see [Authenticating modules](#) in the Go Modules Reference.

```
$ go mod tidy
go: finding module for package rsc.io/quote
```

```
go: found rsc.io/quote in rsc.io/quote v1.5.2
```

4. Run your code to see the message generated by the function you're calling.

```
$ go run .  
Don't communicate by sharing memory, share memory by communicating.
```

Notice that your code calls the Go function, printing a clever message about communication.

When you ran `go mod tidy`, it located and downloaded the `rsc.io/quote` module that contains the package you imported. By default, it downloaded the latest version -- `v1.5.2`.

Write more code

With this quick introduction, you got Go installed and learned some of the basics. To write some more code with another tutorial, take a look at [Create a Go module](#).

Go by Example: File Paths

The `filepath` package provides functions to parse and construct *file paths* in a way that is portable between operating systems; `dir/file` on Linux vs. `dir\file` on Windows, for example.

`Join` should be used to construct paths in a portable way. It takes any number of arguments and constructs a hierarchical path from them.

You should always use `Join` instead of concatenating `/s` or `\s` manually. In addition to providing portability, `Join` will also normalize paths by removing superfluous separators and directory changes.

`Dir` and `Base` can be used to split a path to the directory and the file. Alternatively, `Split` will return both in the same call.

We can check whether a path is absolute.

Some file names have extensions following a dot. We can split the extension out of such names with `Ext`.

To find the file's name with the extension removed, use `strings.TrimSuffix`.

`Rel` finds a relative path between a *base* and a *target*. It returns an error if the target cannot be made relative to base.

```
package main

import (
    "fmt"
    "path/filepath"
    "strings"
)

func main() {

    p := filepath.Join("dir1", "dir2", "filename")
    fmt.Println("p:", p)

    fmt.Println(filepath.Join("dir1/", "filename"))
    fmt.Println(filepath.Join("dir1/../dir1", "filename"))

    fmt.Println("Dir(p):", filepath.Dir(p))
    fmt.Println("Base(p):", filepath.Base(p))

    fmt.Println(filepath.IsAbs("dir/file"))
    fmt.Println(filepath.IsAbs("/dir/file"))

    filename := "config.json"

    ext := filepath.Ext(filename)
    fmt.Println(ext)

    fmt.Println(strings.TrimSuffix(filename, ext))

    rel, err := filepath.Rel("a/b", "a/b/t/file")
    if err != nil {
        panic(err)
    }
    fmt.Println(rel)

    rel, err = filepath.Rel("a/b", "a/c/t/file")
    if err != nil {
        panic(err)
    }
    fmt.Println(rel)
}
```

```
$ go run file-paths.go
p: dir1/dir2/filename
dir1/filename
dir1/filename
Dir(p): dir1/dir2
Base(p): filename
false
true
.json
config
t/file
../c/t/file
```

Next example: [Directories](#).

Go by Example: Values

Go has various value types including strings, integers, floats, booleans, etc. Here are a few basic examples.

Strings, which can be added together with +.

Integers and floats.

Booleans, with boolean operators as you'd expect.

```
package main

import "fmt"

func main() {

    fmt.Println("go" + "lang")

    fmt.Println("1+1 =", 1+1)
    fmt.Println("7.0/3.0 =", 7.0/3.0)

    fmt.Println(true && false)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

```
$ go run values.go
golang
1+1 = 2
7.0/3.0 = 2.3333333333333335
false
true
false
```

Next example: [Variables](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Release History

This page summarizes the changes between official stable releases of Go. The [change log](#) has the full details.

To update to a specific release, use:

```
git fetch --tags
git checkout goX.Y.Z
```

Release Policy

Each major Go release is supported until there are two newer major releases. For example, Go 1.5 was supported until the Go 1.7 release, and Go 1.6 was supported until the Go 1.8 release. We fix critical problems, including [critical security problems](#), in supported releases as needed by issuing minor revisions (for example, Go 1.6.1, Go 1.6.2, and so on).

go1.24.0 (released 2025-02-11)

Go 1.24.0 is a major release of Go. Read the [Go 1.24 Release Notes](#) for more information.

Minor revisions

go1.24.1 (released 2025-03-04) includes security fixes to the `net/http` package, as well as bug fixes to `cgo`, the compiler, the `go` command, and the `reflect`, `runtime`, and `syscall` packages. See the [Go 1.24.1 milestone](#) on our issue tracker for details.

go1.24.2 (released 2025-04-01) includes security fixes to the `net/http` package, as well as bug fixes to the compiler, the `runtime`, the `go` command, and the `crypto/tls`, `go/types`, `net/http`, and `testing` packages. See the [Go 1.24.2 milestone](#) on our issue tracker for details.

go1.24.3 (released 2025-05-06) includes security fixes to the `os` package, as well as bug fixes to the `runtime`, the compiler, the linker, the `go` command, and the `crypto/tls` and `os` packages. See the [Go 1.24.3 milestone](#) on our issue tracker for details.

go1.23.0 (released 2024-08-13)

Go 1.23.0 is a major release of Go. Read the [Go 1.23 Release Notes](#) for more information.

Minor revisions

go1.23.1 (released 2024-09-05) includes security fixes to the `encoding/gob`, `go/build/constraint`, and `go/parser` packages, as well as bug fixes to the compiler, the `go` command, the `runtime`, and the `database/sql`, `go/types`, `os`, `runtime/trace`, and `unique` packages. See the [Go 1.23.1 milestone](#) on our issue tracker for details.

go1.23.2 (released 2024-10-01) includes fixes to the compiler, `cgo`, the `runtime`, and the `maps`, `os`, `os/exec`, `time`, and `unique` packages. See the [Go 1.23.2 milestone](#) on our issue tracker for details.

go1.23.3 (released 2024-11-06) includes fixes to the linker, the `runtime`, and the `net/http`, `os`, and `syscall` packages. See the [Go 1.23.3 milestone](#) on our issue tracker for details.

go1.23.4 (released 2024-12-03) includes fixes to the compiler, the `runtime`, the `trace` command, and the `syscall` package. See the [Go 1.23.4 milestone](#) on our issue tracker for details.

go1.23.5 (released 2025-01-16) includes security fixes to the `crypto/x509` and `net/http` packages, as well as bug fixes to the compiler, the `runtime`, and the `net` package. See the [Go 1.23.5 milestone](#) on our issue tracker for details.

go1.23.6 (released 2025-02-04) includes security fixes to the `crypto/elliptic` package, as well as bug fixes to the compiler and the `go` command. See the [Go 1.23.6 milestone](#) on our issue tracker for details.

go1.23.7 (released 2025-03-04) includes security fixes to the `net/http` package, as well as bug fixes to `cgo`, the compiler, and the `reflect`, `runtime`, and `syscall` packages. See the [Go 1.23.7 milestone](#) on our issue tracker for details.

go1.23.8 (released 2025-04-01) includes security fixes to the `net/http` package, as well as bug fixes to the `runtime` and the `go` command. See the [Go 1.23.8 milestone](#) on our issue tracker for details.

go1.23.9 (released 2025-05-06) includes fixes to the `runtime` and the linker. See the [Go 1.23.9 milestone](#) on our issue tracker for details.

go1.22.0 (released 2024-02-06)

Go 1.22.0 is a major release of Go. Read the [Go 1.22 Release Notes](#) for more information.

Minor revisions

go1.22.1 (released 2024-03-05) includes security fixes to the `crypto/x509`, `html/template`, `net/http`, `net/http/cookiejar`, and `net/mail` packages, as well as bug fixes to the compiler, the `go` command, the runtime, the `trace` command, and the `go/types` and `net/http` packages. See the [Go 1.22.1 milestone](#) on our issue tracker for details.

go1.22.2 (released 2024-04-03) includes a security fix to the `net/http` package, as well as bug fixes to the compiler, the `go` command, the linker, and the `encoding/gob`, `go/types`, `net/http`, and `runtime/trace` packages. See the [Go 1.22.2 milestone](#) on our issue tracker for details.

go1.22.3 (released 2024-05-07) includes security fixes to the `go` command and the `net` package, as well as bug fixes to the compiler, the runtime, and the `net/http` package. See the [Go 1.22.3 milestone](#) on our issue tracker for details.

go1.22.4 (released 2024-06-04) includes security fixes to the `archive/zip` and `net/netip` packages, as well as bug fixes to the compiler, the `go` command, the linker, the runtime, and the `os` package. See the [Go 1.22.4 milestone](#) on our issue tracker for details.

go1.22.5 (released 2024-07-02) includes security fixes to the `net/http` package, as well as bug fixes to the compiler, `cgo`, the `go` command, the linker, the runtime, and the `crypto/tls`, `go/types`, `net`, `net/http`, and `os/exec` packages. See the [Go 1.22.5 milestone](#) on our issue tracker for details.

go1.22.6 (released 2024-08-06) includes fixes to the `go` command, the compiler, the linker, the `trace` command, the `covdata` command, and the `bytes`, `go/types`, and `os/exec` packages. See the [Go 1.22.6 milestone](#) on our issue tracker for details.

go1.22.7 (released 2024-09-05) includes security fixes to the `encoding/gob`, `go/build/constraint`, and `go/parser` packages, as well as bug fixes to the `fix` command and the runtime. See the [Go 1.22.7 milestone](#) on our issue tracker for details.

go1.22.8 (released 2024-10-01) includes fixes to `cgo`, and the `maps` and `syscall` packages. See the [Go 1.22.8 milestone](#) on our issue tracker for details.

go1.22.9 (released 2024-11-06) includes fixes to the linker. See the [Go 1.22.9 milestone](#) on our issue tracker for details.

go1.22.10 (released 2024-12-03) includes fixes to the runtime and the `syscall` package. See the [Go 1.22.10 milestone](#) on our issue tracker for details.

go1.22.11 (released 2025-01-16) includes security fixes to the `crypto/x509` and `net/http` packages, as well as bug fixes to the runtime. See the [Go 1.22.11 milestone](#) on our issue tracker for details.

go1.22.12 (released 2025-02-04) includes security fixes to the `crypto/elliptic` package, as well as bug fixes to the compiler and the `go` command. See the [Go 1.22.12 milestone](#) on our issue tracker for details.

go1.21.0 (released 2023-08-08)

Go 1.21.0 is a major release of Go. Read the [Go 1.21 Release Notes](#) for more information.

Minor revisions

go1.21.1 (released 2023-09-06) includes four security fixes to the `cmd/go`, `crypto/tls`, and `html/template` packages, as well as bug fixes to the compiler, the `go` command, the linker, the runtime, and the `context`, `crypto/tls`, `encoding/gob`, `encoding/xml`, `go/types`, `net/http`, `os`, and `path/filepath` packages. See the [Go 1.21.1 milestone](#) on our issue tracker for details.

go1.21.2 (released 2023-10-05) includes one security fix to the `cmd/go` package, as well as bug fixes to the compiler, the `go` command, the linker, the runtime, and the `runtime/metrics` package. See the [Go 1.21.2 milestone](#) on our issue tracker for details.

go1.21.3 (released 2023-10-10) includes a security fix to the `net/http` package. See the [Go 1.21.3 milestone](#) on our issue tracker for details.

go1.21.4 (released 2023-11-07) includes security fixes to the `path/filepath` package, as well as bug fixes to the linker, the runtime, the compiler, and the `go/types`, `net/http`, and `runtime/cgo` packages. See the [Go 1.21.4 milestone](#) on our issue

tracker for details.

go1.21.5 (released 2023-12-05) includes security fixes to the go command, and the net/http and path/filepath packages, as well as bug fixes to the compiler, the go command, the runtime, and the crypto/rand, net, os, and syscall packages. See the [Go 1.21.5 milestone](#) on our issue tracker for details.

go1.21.6 (released 2024-01-09) includes fixes to the compiler, the runtime, and the crypto/tls, maps, and runtime/pprof packages. See the [Go 1.21.6 milestone](#) on our issue tracker for details.

go1.21.7 (released 2024-02-06) includes fixes to the compiler, the go command, the runtime, and the crypto/x509 package. See the [Go 1.21.7 milestone](#) on our issue tracker for details.

go1.21.8 (released 2024-03-05) includes security fixes to the crypto/x509, html/template, net/http, net/http/cookiejar, and net/mail packages, as well as bug fixes to the go command and the runtime. See the [Go 1.21.8 milestone](#) on our issue tracker for details.

go1.21.9 (released 2024-04-03) includes a security fix to the net/http package, as well as bug fixes to the linker, and the go/types and net/http packages. See the [Go 1.21.9 milestone](#) on our issue tracker for details.

go1.21.10 (released 2024-05-07) includes security fixes to the go command, as well as bug fixes to the net/http package. See the [Go 1.21.10 milestone](#) on our issue tracker for details.

go1.21.11 (released 2024-06-04) includes security fixes to the archive/zip and net/netip packages, as well as bug fixes to the compiler, the go command, the runtime, and the os package. See the [Go 1.21.11 milestone](#) on our issue tracker for details.

go1.21.12 (released 2024-07-02) includes security fixes to the net/http package, as well as bug fixes to the compiler, the go command, the runtime, and the crypto/x509, net/http, net/netip, and os packages. See the [Go 1.21.12 milestone](#) on our issue tracker for details.

go1.21.13 (released 2024-08-06) includes fixes to the go command, the covdata command, and the bytes package. See the [Go 1.21.13 milestone](#) on our issue tracker for details.

go1.20 (released 2023-02-01)

Go 1.20 is a major release of Go. Read the [Go 1.20 Release Notes](#) for more information.

Minor revisions

go1.20.1 (released 2023-02-14) includes security fixes to the crypto/tls, mime/multipart, net/http, and path/filepath packages, as well as bug fixes to the compiler, the go command, the linker, the runtime, and the time package. See the [Go 1.20.1 milestone](#) on our issue tracker for details.

go1.20.2 (released 2023-03-07) includes a security fix to the crypto/elliptic package, as well as bug fixes to the compiler, the covdata command, the linker, the runtime, and the crypto/ecdh, crypto/rsa, crypto/x509, os, and syscall packages. See the [Go 1.20.2 milestone](#) on our issue tracker for details.

go1.20.3 (released 2023-04-04) includes security fixes to the go/parser, html/template, mime/multipart, net/http, and net/textproto packages, as well as bug fixes to the compiler, the linker, the runtime, and the time package. See the [Go 1.20.3 milestone](#) on our issue tracker for details.

go1.20.4 (released 2023-05-02) includes three security fixes to the html/template package, as well as bug fixes to the compiler, the runtime, and the crypto/subtle, crypto/tls, net/http, and syscall packages. See the [Go 1.20.4 milestone](#) on our issue tracker for details.

go1.20.5 (released 2023-06-06) includes four security fixes to the cmd/go and runtime packages, as well as bug fixes to the compiler, the go command, the runtime, and the crypto/rsa, net, and os packages. See the [Go 1.20.5 milestone](#) on our issue tracker for details.

go1.20.6 (released 2023-07-11) includes a security fix to the net/http package, as well as bug fixes to the compiler, cgo, the cover tool, the go command, the runtime, and the crypto/ecdsa, go/build, go/printer, net/mail, and text/template packages. See the [Go 1.20.6 milestone](#) on our issue tracker for details.

go1.20.7 (released 2023-08-01) includes a security fix to the crypto/tls package, as well as bug fixes to the assembler and the compiler. See the [Go 1.20.7 milestone](#) on our issue tracker for details.

go1.20.8 (released 2023-09-06) includes two security fixes to the html/template package, as well as bug fixes to the compiler, the go command, the runtime, and the crypto/tls, go/types, net/http, and path/filepath packages. See the

[Go 1.20.8 milestone](#) on our issue tracker for details.

go1.20.9 (released 2023-10-05) includes one security fix to the cmd/go package, as well as bug fixes to the go command and the linker. See the [Go 1.20.9 milestone](#) on our issue tracker for details.

go1.20.10 (released 2023-10-10) includes a security fix to the net/http package. See the [Go 1.20.10 milestone](#) on our issue tracker for details.

go1.20.11 (released 2023-11-07) includes security fixes to the path/filepath package, as well as bug fixes to the linker and the net/http package. See the [Go 1.20.11 milestone](#) on our issue tracker for details.

go1.20.12 (released 2023-12-05) includes security fixes to the go command, and the net/http and path/filepath packages, as well as bug fixes to the compiler and the go command. See the [Go 1.20.12 milestone](#) on our issue tracker for details.

go1.20.13 (released 2024-01-09) includes fixes to the runtime and the crypto/tls package. See the [Go 1.20.13 milestone](#) on our issue tracker for details.

go1.20.14 (released 2024-02-06) includes fixes to the crypto/x509 package. See the [Go 1.20.14 milestone](#) on our issue tracker for details.

go1.19 (released 2022-08-02)

Go 1.19 is a major release of Go. Read the [Go 1.19 Release Notes](#) for more information.

Minor revisions

go1.19.1 (released 2022-09-06) includes security fixes to the net/http and net/url packages, as well as bug fixes to the compiler, the go command, the pprof command, the linker, the runtime, and the crypto/tls and crypto/x509 packages. See the [Go 1.19.1 milestone](#) on our issue tracker for details.

go1.19.2 (released 2022-10-04) includes security fixes to the archive/tar, net/http/httputil, and regexp packages, as well as bug fixes to the compiler, the linker, the runtime, and the go/types package. See the [Go 1.19.2 milestone](#) on our issue tracker for details.

go1.19.3 (released 2022-11-01) includes security fixes to the os/exec and syscall packages, as well as bug fixes to the compiler and the runtime. See the [Go 1.19.3 milestone](#) on our issue tracker for details.

go1.19.4 (released 2022-12-06) includes security fixes to the net/http and os packages, as well as bug fixes to the compiler, the runtime, and the crypto/x509, os/exec, and sync/atomic packages. See the [Go 1.19.4 milestone](#) on our issue tracker for details.

go1.19.5 (released 2023-01-10) includes fixes to the compiler, the linker, and the crypto/x509, net/http, sync/atomic, and syscall packages. See the [Go 1.19.5 milestone](#) on our issue tracker for details.

go1.19.6 (released 2023-02-14) includes security fixes to the crypto/tls, mime/multipart, net/http, and path/filepath packages, as well as bug fixes to the go command, the linker, the runtime, and the crypto/x509, net/http, and time packages. See the [Go 1.19.6 milestone](#) on our issue tracker for details.

go1.19.7 (released 2023-03-07) includes a security fix to the crypto/elliptic package, as well as bug fixes to the linker, the runtime, and the crypto/x509 and syscall packages. See the [Go 1.19.7 milestone](#) on our issue tracker for details.

go1.19.8 (released 2023-04-04) includes security fixes to the go/parser, html/template, mime/multipart, net/http, and net/textproto packages, as well as bug fixes to the linker, the runtime, and the time package. See the [Go 1.19.8 milestone](#) on our issue tracker for details.

go1.19.9 (released 2023-05-02) includes three security fixes to the html/template package, as well as bug fixes to the compiler, the runtime, and the crypto/tls and syscall packages. See the [Go 1.19.9 milestone](#) on our issue tracker for details.

go1.19.10 (released 2023-06-06) includes four security fixes to the cmd/go and runtime packages, as well as bug fixes to the compiler, the go command, and the runtime. See the [Go 1.19.10 milestone](#) on our issue tracker for details.

go1.19.11 (released 2023-07-11) includes a security fix to the net/http package, as well as bug fixes to cgo, the cover tool, the go command, the runtime, and the go/printer package. See the [Go 1.19.11 milestone](#) on our issue tracker for details.

go1.19.12 (released 2023-08-01) includes a security fix to the crypto/tls package, as well as bug fixes to the assembler and the compiler. See the [Go 1.19.12 milestone](#) on our issue tracker for details.

go1.19.13 (released 2023-09-06) includes fixes to the go command, and the crypto/tls and net/http packages. See the [Go 1.19.13 milestone](#) on our issue tracker for details.

go1.18 (released 2022-03-15)

Go 1.18 is a major release of Go. Read the [Go 1.18 Release Notes](#) for more information.

Minor revisions

go1.18.1 (released 2022-04-12) includes security fixes to the crypto/elliptic, crypto/x509, and encoding/pem packages, as well as bug fixes to the compiler, linker, runtime, the go command, vet, and the bytes, crypto/x509, and go/types packages. See the [Go 1.18.1 milestone](#) on our issue tracker for details.

go1.18.2 (released 2022-05-10) includes security fixes to the syscall package, as well as bug fixes to the compiler, runtime, the go command, and the crypto/x509, go/types, net/http/httptest, reflect, and sync/atomic packages. See the [Go 1.18.2 milestone](#) on our issue tracker for details.

go1.18.3 (released 2022-06-01) includes security fixes to the crypto/rand, crypto/tls, os/exec, and path/filepath packages, as well as bug fixes to the compiler, and the crypto/tls and text/template/parse packages. See the [Go 1.18.3 milestone](#) on our issue tracker for details.

go1.18.4 (released 2022-07-12) includes security fixes to the compress/gzip, encoding/gob, encoding/xml, go/parser, io/fs, net/http, and path/filepath packages, as well as bug fixes to the compiler, the go command, the linker, the runtime, and the runtime/metrics package. See the [Go 1.18.4 milestone](#) on our issue tracker for details.

go1.18.5 (released 2022-08-01) includes security fixes to the encoding/gob and math/big packages, as well as bug fixes to the compiler, the go command, the runtime, and the testing package. See the [Go 1.18.5 milestone](#) on our issue tracker for details.

go1.18.6 (released 2022-09-06) includes security fixes to the net/http package, as well as bug fixes to the compiler, the go command, the pprof command, the runtime, and the crypto/tls, encoding/xml, and net packages. See the [Go 1.18.6 milestone](#) on our issue tracker for details.

go1.18.7 (released 2022-10-04) includes security fixes to the archive/tar, net/http/httputil, and regexp packages, as well as bug fixes to the compiler, the linker, and the go/types package. See the [Go 1.18.7 milestone](#) on our issue tracker for details.

go1.18.8 (released 2022-11-01) includes security fixes to the os/exec and syscall packages, as well as bug fixes to the runtime. See the [Go 1.18.8 milestone](#) on our issue tracker for details.

go1.18.9 (released 2022-12-06) includes security fixes to the net/http and os packages, as well as bug fixes to cgo, the compiler, the runtime, and the crypto/x509 and os/exec packages. See the [Go 1.18.9 milestone](#) on our issue tracker for details.

go1.18.10 (released 2023-01-10) includes fixes to cgo, the compiler, the linker, and the crypto/x509, net/http, and syscall packages. See the [Go 1.18.10 milestone](#) on our issue tracker for details.

go1.17 (released 2021-08-16)

Go 1.17 is a major release of Go. Read the [Go 1.17 Release Notes](#) for more information.

Minor revisions

go1.17.1 (released 2021-09-09) includes a security fix to the archive/zip package, as well as bug fixes to the compiler, linker, the go command, and the crypto/rand, embed, go/types, html/template, and net/http packages. See the [Go 1.17.1 milestone](#) on our issue tracker for details.

go1.17.2 (released 2021-10-07) includes security fixes to linker and the misc/wasm directory, as well as bug fixes to the compiler, runtime, the go command, and the text/template and time packages. See the [Go 1.17.2 milestone](#) on our issue tracker for details.

go1.17.3 (released 2021-11-04) includes security fixes to the archive/zip and debug/macho packages, as well as bug fixes to the compiler, linker, runtime, the go command, the misc/wasm directory, and the net/http and syscall packages. See the [Go 1.17.3 milestone](#) on our issue tracker for details.

go1.17.4 (released 2021-12-02) includes fixes to the compiler, linker, runtime, and the go/types, net/http, and time

packages. See the [Go 1.17.4 milestone](#) on our issue tracker for details.

go1.17.5 (released 2021-12-09) includes security fixes to the `net/http` and `syscall` packages. See the [Go 1.17.5 milestone](#) on our issue tracker for details.

go1.17.6 (released 2022-01-06) includes fixes to the compiler, linker, runtime, and the `crypto/x509`, `net/http`, and `reflect` packages. See the [Go 1.17.6 milestone](#) on our issue tracker for details.

go1.17.7 (released 2022-02-10) includes security fixes to the `go` command, and the `crypto/elliptic` and `math/big` packages, as well as bug fixes to the compiler, linker, runtime, the `go` command, and the `debug/macho`, `debug/pe`, and `net/http/httptest` packages. See the [Go 1.17.7 milestone](#) on our issue tracker for details.

go1.17.8 (released 2022-03-03) includes a security fix to the `regexp/syntax` package, as well as bug fixes to the compiler, runtime, the `go` command, and the `crypto/x509` and `net` packages. See the [Go 1.17.8 milestone](#) on our issue tracker for details.

go1.17.9 (released 2022-04-12) includes security fixes to the `crypto/elliptic` and `encoding/pem` packages, as well as bug fixes to the linker and runtime. See the [Go 1.17.9 milestone](#) on our issue tracker for details.

go1.17.10 (released 2022-05-10) includes security fixes to the `syscall` package, as well as bug fixes to the compiler, runtime, and the `crypto/x509` and `net/http/httptest` packages. See the [Go 1.17.10 milestone](#) on our issue tracker for details.

go1.17.11 (released 2022-06-01) includes security fixes to the `crypto/rand`, `crypto/tls`, `os/exec`, and `path/filepath` packages, as well as bug fixes to the `crypto/tls` package. See the [Go 1.17.11 milestone](#) on our issue tracker for details.

go1.17.12 (released 2022-07-12) includes security fixes to the `compress/gzip`, `encoding/gob`, `encoding/xml`, `go/parser`, `io/fs`, `net/http`, and `path/filepath` packages, as well as bug fixes to the compiler, the `go` command, the runtime, and the `runtime/metrics` package. See the [Go 1.17.12 milestone](#) on our issue tracker for details.

go1.17.13 (released 2022-08-01) includes security fixes to the `encoding/gob` and `math/big` packages, as well as bug fixes to the compiler and the runtime. See the [Go 1.17.13 milestone](#) on our issue tracker for details.

go1.16 (released 2021-02-16)

Go 1.16 is a major release of Go. Read the [Go 1.16 Release Notes](#) for more information.

Minor revisions

go1.16.1 (released 2021-03-10) includes security fixes to the `archive/zip` and `encoding/xml` packages. See the [Go 1.16.1 milestone](#) on our issue tracker for details.

go1.16.2 (released 2021-03-11) includes fixes to `cgo`, the compiler, linker, the `go` command, and the `syscall` and `time` packages. See the [Go 1.16.2 milestone](#) on our issue tracker for details.

go1.16.3 (released 2021-04-01) includes fixes to the compiler, linker, runtime, the `go` command, and the `testing` and `time` packages. See the [Go 1.16.3 milestone](#) on our issue tracker for details.

go1.16.4 (released 2021-05-06) includes a security fix to the `net/http` package, as well as bug fixes to the compiler, runtime, and the `archive/zip`, `syscall`, and `time` packages. See the [Go 1.16.4 milestone](#) on our issue tracker for details.

go1.16.5 (released 2021-06-03) includes security fixes to the `archive/zip`, `math/big`, `net`, and `net/http/httputil` packages, as well as bug fixes to the linker, the `go` command, and the `net/http` package. See the [Go 1.16.5 milestone](#) on our issue tracker for details.

go1.16.6 (released 2021-07-12) includes a security fix to the `crypto/tls` package, as well as bug fixes to the compiler, and the `net` and `net/http` packages. See the [Go 1.16.6 milestone](#) on our issue tracker for details.

go1.16.7 (released 2021-08-05) includes a security fix to the `net/http/httputil` package, as well as bug fixes to the compiler, linker, runtime, the `go` command, and the `net/http` package. See the [Go 1.16.7 milestone](#) on our issue tracker for details.

go1.16.8 (released 2021-09-09) includes a security fix to the `archive/zip` package, as well as bug fixes to the `archive/zip`, `go/internal/gccgoimporter`, `html/template`, `net/http`, and `runtime/pprof` packages. See the [Go 1.16.8 milestone](#) on our issue tracker for details.

go1.16.9 (released 2021-10-07) includes security fixes to linker and the `misc/wasm` directory, as well as bug fixes to runtime and the `text/template` package. See the [Go 1.16.9 milestone](#) on our issue tracker for details.

go1.16.10 (released 2021-11-04) includes security fixes to the archive/zip and debug/macho packages, as well as bug fixes to the compiler, linker, runtime, the misc/wasm directory, and the net/http package. See the [Go 1.16.10 milestone](#) on our issue tracker for details.

go1.16.11 (released 2021-12-02) includes fixes to the compiler, runtime, and the net/http, net/http/httptest, and time packages. See the [Go 1.16.11 milestone](#) on our issue tracker for details.

go1.16.12 (released 2021-12-09) includes security fixes to the net/http and syscall packages. See the [Go 1.16.12 milestone](#) on our issue tracker for details.

go1.16.13 (released 2022-01-06) includes fixes to the compiler, linker, runtime, and the net/http package. See the [Go 1.16.13 milestone](#) on our issue tracker for details.

go1.16.14 (released 2022-02-10) includes security fixes to the go command, and the crypto/elliptic and math/big packages, as well as bug fixes to the compiler, linker, runtime, the go command, and the debug/macho, debug/pe, net/http/httptest, and testing packages. See the [Go 1.16.14 milestone](#) on our issue tracker for details.

go1.16.15 (released 2022-03-03) includes a security fix to the regexp/syntax package, as well as bug fixes to the compiler, runtime, the go command, and the net package. See the [Go 1.16.15 milestone](#) on our issue tracker for details.

go1.15 (released 2020-08-11)

Go 1.15 is a major release of Go. Read the [Go 1.15 Release Notes](#) for more information.

Minor revisions

go1.15.1 (released 2020-09-01) includes security fixes to the net/http/cgi and net/http/fcgi packages. See the [Go 1.15.1 milestone](#) on our issue tracker for details.

go1.15.2 (released 2020-09-09) includes fixes to the compiler, runtime, documentation, the go command, and the net/mail, os, sync, and testing packages. See the [Go 1.15.2 milestone](#) on our issue tracker for details.

go1.15.3 (released 2020-10-14) includes fixes to cgo, the compiler, runtime, the go command, and the bytes, plugin, and testing packages. See the [Go 1.15.3 milestone](#) on our issue tracker for details.

go1.15.4 (released 2020-11-05) includes fixes to cgo, the compiler, linker, runtime, and the compress/flate, net/http, reflect, and time packages. See the [Go 1.15.4 milestone](#) on our issue tracker for details.

go1.15.5 (released 2020-11-12) includes security fixes to the go command and the math/big package. See the [Go 1.15.5 milestone](#) on our issue tracker for details.

go1.15.6 (released 2020-12-03) includes fixes to the compiler, linker, runtime, the go command, and the io package. See the [Go 1.15.6 milestone](#) on our issue tracker for details.

go1.15.7 (released 2021-01-19) includes security fixes to the go command and the crypto/elliptic package. See the [Go 1.15.7 milestone](#) on our issue tracker for details.

go1.15.8 (released 2021-02-04) includes fixes to the compiler, linker, runtime, the go command, and the net/http package. See the [Go 1.15.8 milestone](#) on our issue tracker for details.

go1.15.9 (released 2021-03-10) includes security fixes to the encoding/xml package. See the [Go 1.15.9 milestone](#) on our issue tracker for details.

go1.15.10 (released 2021-03-11) includes fixes to the compiler, the go command, and the net/http, os, syscall, and time packages. See the [Go 1.15.10 milestone](#) on our issue tracker for details.

go1.15.11 (released 2021-04-01) includes fixes to cgo, the compiler, linker, runtime, the go command, and the database/sql and net/http packages. See the [Go 1.15.11 milestone](#) on our issue tracker for details.

go1.15.12 (released 2021-05-06) includes a security fix to the net/http package, as well as bug fixes to the compiler, runtime, and the archive/zip, syscall, and time packages. See the [Go 1.15.12 milestone](#) on our issue tracker for details.

go1.15.13 (released 2021-06-03) includes security fixes to the archive/zip, math/big, net, and net/http/httputil packages, as well as bug fixes to the linker, the go command, and the math/big and net/http packages. See the [Go 1.15.13 milestone](#) on our issue tracker for details.

go1.15.14 (released 2021-07-12) includes a security fix to the crypto/tls package, as well as bug fixes to the linker and the

net package. See the [Go 1.15.14 milestone](#) on our issue tracker for details.

go1.15.15 (released 2021-08-05) includes a security fix to the net/http/httputil package, as well as bug fixes to the compiler, runtime, the go command, and the net/http package. See the [Go 1.15.15 milestone](#) on our issue tracker for details.

go1.14 (released 2020-02-25)

Go 1.14 is a major release of Go. Read the [Go 1.14 Release Notes](#) for more information.

Minor revisions

go1.14.1 (released 2020-03-19) includes fixes to the go command, tools, and the runtime. See the [Go 1.14.1 milestone](#) on our issue tracker for details.

go1.14.2 (released 2020-04-08) includes fixes to cgo, the go command, the runtime, and the os/exec and testing packages. See the [Go 1.14.2 milestone](#) on our issue tracker for details.

go1.14.3 (released 2020-05-14) includes fixes to cgo, the compiler, the runtime, and the go/doc and math/big packages. See the [Go 1.14.3 milestone](#) on our issue tracker for details.

go1.14.4 (released 2020-06-01) includes fixes to the go doc command, the runtime, and the encoding/json and os packages. See the [Go 1.14.4 milestone](#) on our issue tracker for details.

go1.14.5 (released 2020-07-14) includes security fixes to the crypto/x509 and net/http packages. See the [Go 1.14.5 milestone](#) on our issue tracker for details.

go1.14.6 (released 2020-07-16) includes fixes to the go command, the compiler, the linker, vet, and the database/sql, encoding/json, net/http, reflect, and testing packages. See the [Go 1.14.6 milestone](#) on our issue tracker for details.

go1.14.7 (released 2020-08-06) includes security fixes to the encoding/binary package. See the [Go 1.14.7 milestone](#) on our issue tracker for details.

go1.14.8 (released 2020-09-01) includes security fixes to the net/http/cgi and net/http/fcgi packages. See the [Go 1.14.8 milestone](#) on our issue tracker for details.

go1.14.9 (released 2020-09-09) includes fixes to the compiler, linker, runtime, documentation, and the net/http and testing packages. See the [Go 1.14.9 milestone](#) on our issue tracker for details.

go1.14.10 (released 2020-10-14) includes fixes to the compiler, runtime, and the plugin and testing packages. See the [Go 1.14.10 milestone](#) on our issue tracker for details.

go1.14.11 (released 2020-11-05) includes fixes to the runtime, and the net/http and time packages. See the [Go 1.14.11 milestone](#) on our issue tracker for details.

go1.14.12 (released 2020-11-12) includes security fixes to the go command and the math/big package. See the [Go 1.14.12 milestone](#) on our issue tracker for details.

go1.14.13 (released 2020-12-03) includes fixes to the compiler, runtime, and the go command. See the [Go 1.14.13 milestone](#) on our issue tracker for details.

go1.14.14 (released 2021-01-19) includes security fixes to the go command and the crypto/elliptic package. See the [Go 1.14.14 milestone](#) on our issue tracker for details.

go1.14.15 (released 2021-02-04) includes fixes to the compiler, runtime, the go command, and the net/http package. See the [Go 1.14.15 milestone](#) on our issue tracker for details.

go1.13 (released 2019-09-03)

Go 1.13 is a major release of Go. Read the [Go 1.13 Release Notes](#) for more information.

Minor revisions

go1.13.1 (released 2019-09-25) includes security fixes to the net/http and net/textproto packages. See the [Go 1.13.1 milestone](#) on our issue tracker for details.

go1.13.2 (released 2019-10-17) includes security fixes to the compiler and the crypto/dsa package. See the [Go 1.13.2 milestone](#) on our issue tracker for details.

go1.13.3 (released 2019-10-17) includes fixes to the go command, the toolchain, the runtime, and the crypto/ecdsa, net, net/http, and syscall packages. See the [Go 1.13.3 milestone](#) on our issue tracker for details.

go1.13.4 (released 2019-10-31) includes fixes to the net/http and syscall packages. It also fixes an issue on macOS 10.15 Catalina where the non-notarized installer and binaries were being [rejected by Gatekeeper](#). See the [Go 1.13.4 milestone](#) on our issue tracker for details.

go1.13.5 (released 2019-12-04) includes fixes to the go command, the runtime, the linker, and the net/http package. See the [Go 1.13.5 milestone](#) on our issue tracker for details.

go1.13.6 (released 2020-01-09) includes fixes to the runtime and the net/http package. See the [Go 1.13.6 milestone](#) on our issue tracker for details.

go1.13.7 (released 2020-01-28) includes two security fixes to the crypto/x509 package. See the [Go 1.13.7 milestone](#) on our issue tracker for details.

go1.13.8 (released 2020-02-12) includes fixes to the runtime, and the crypto/x509 and net/http packages. See the [Go 1.13.8 milestone](#) on our issue tracker for details.

go1.13.9 (released 2020-03-19) includes fixes to the go command, tools, the runtime, the toolchain, and the crypto/cypher package. See the [Go 1.13.9 milestone](#) on our issue tracker for details.

go1.13.10 (released 2020-04-08) includes fixes to the go command, the runtime, and the os/exec and time packages. See the [Go 1.13.10 milestone](#) on our issue tracker for details.

go1.13.11 (released 2020-05-14) includes fixes to the compiler. See the [Go 1.13.11 milestone](#) on our issue tracker for details.

go1.13.12 (released 2020-06-01) includes fixes to the runtime, and the go/types and math/big packages. See the [Go 1.13.12 milestone](#) on our issue tracker for details.

go1.13.13 (released 2020-07-14) includes security fixes to the crypto/x509 and net/http packages. See the [Go 1.13.13 milestone](#) on our issue tracker for details.

go1.13.14 (released 2020-07-16) includes fixes to the compiler, vet, and the database/sql, net/http, and reflect packages. See the [Go 1.13.14 milestone](#) on our issue tracker for details.

go1.13.15 (released 2020-08-06) includes security fixes to the encoding/binary package. See the [Go 1.13.15 milestone](#) on our issue tracker for details.

go1.12 (released 2019-02-25)

Go 1.12 is a major release of Go. Read the [Go 1.12 Release Notes](#) for more information.

Minor revisions

go1.12.1 (released 2019-03-14) includes fixes to cgo, the compiler, the go command, and the fmt, net/smtp, os, path/filepath, sync, and text/template packages. See the [Go 1.12.1 milestone](#) on our issue tracker for details.

go1.12.2 (released 2019-04-05) includes security fixes to the runtime, as well as bug fixes to the compiler, the go command, and the doc, net, net/http/httputil, and os packages. See the [Go 1.12.2 milestone](#) on our issue tracker for details.

go1.12.3 (released 2019-04-08) was accidentally released without its intended fix. It is identical to go1.12.2, except for its version number. The intended fix is in go1.12.4.

go1.12.4 (released 2019-04-11) fixes an issue where using the prebuilt binary releases on older versions of GNU/Linux [led to failures](#) when linking programs that used cgo. Only Linux users who hit this issue need to update.

go1.12.5 (released 2019-05-06) includes fixes to the compiler, the linker, the go command, the runtime, and the os package. See the [Go 1.12.5 milestone](#) on our issue tracker for details.

go1.12.6 (released 2019-06-11) includes fixes to the compiler, the linker, the go command, and the crypto/x509, net/http, and os packages. See the [Go 1.12.6 milestone](#) on our issue tracker for details.

go1.12.7 (released 2019-07-08) includes fixes to cgo, the compiler, and the linker. See the [Go 1.12.7 milestone](#) on our issue tracker for details.

go1.12.8 (released 2019-08-13) includes security fixes to the net/http and net/url packages. See the [Go 1.12.8 milestone](#) on our issue tracker for details.

go1.12.9 (released 2019-08-15) includes fixes to the linker, and the math/big and os packages. See the [Go 1.12.9 milestone](#) on our issue tracker for details.

go1.12.10 (released 2019-09-25) includes security fixes to the net/http and net/textproto packages. See the [Go 1.12.10 milestone](#) on our issue tracker for details.

go1.12.11 (released 2019-10-17) includes security fixes to the crypto/dsa package. See the [Go 1.12.11 milestone](#) on our issue tracker for details.

go1.12.12 (released 2019-10-17) includes fixes to the go command, runtime, and the net and syscall packages. See the [Go 1.12.12 milestone](#) on our issue tracker for details.

go1.12.13 (released 2019-10-31) fixes an issue on macOS 10.15 Catalina where the non-notarized installer and binaries were being [rejected by Gatekeeper](#). Only macOS users who hit this issue need to update.

go1.12.14 (released 2019-12-04) includes a fix to the runtime. See the [Go 1.12.14 milestone](#) on our issue tracker for details.

go1.12.15 (released 2020-01-09) includes fixes to the runtime and the net/http package. See the [Go 1.12.15 milestone](#) on our issue tracker for details.

go1.12.16 (released 2020-01-28) includes two security fixes to the crypto/x509 package. See the [Go 1.12.16 milestone](#) on our issue tracker for details.

go1.12.17 (released 2020-02-12) includes a fix to the runtime. See the [Go 1.12.17 milestone](#) on our issue tracker for details.

go1.11 (released 2018-08-24)

Go 1.11 is a major release of Go. Read the [Go 1.11 Release Notes](#) for more information.

Minor revisions

go1.11.1 (released 2018-10-01) includes fixes to the compiler, documentation, go command, runtime, and the crypto/x509, encoding/json, go/types, net, net/http, and reflect packages. See the [Go 1.11.1 milestone](#) on our issue tracker for details.

go1.11.2 (released 2018-11-02) includes fixes to the compiler, linker, documentation, go command, and the database/sql and go/types packages. See the [Go 1.11.2 milestone](#) on our issue tracker for details.

go1.11.3 (released 2018-12-12) includes three security fixes to "go get" and the crypto/x509 package. See the [Go 1.11.3 milestone](#) on our issue tracker for details.

go1.11.4 (released 2018-12-14) includes fixes to cgo, the compiler, linker, runtime, documentation, go command, and the go/types and net/http packages. It includes a fix to a bug introduced in Go 1.11.3 that broke go get for import path patterns containing "...". See the [Go 1.11.4 milestone](#) on our issue tracker for details.

go1.11.5 (released 2019-01-23) includes a security fix to the crypto/elliptic package. See the [Go 1.11.5 milestone](#) on our issue tracker for details.

go1.11.6 (released 2019-03-14) includes fixes to cgo, the compiler, linker, runtime, go command, and the crypto/x509, encoding/json, net, and net/url packages. See the [Go 1.11.6 milestone](#) on our issue tracker for details.

go1.11.7 (released 2019-04-05) includes fixes to the runtime and the net package. See the [Go 1.11.7 milestone](#) on our issue tracker for details.

go1.11.8 (released 2019-04-08) was accidentally released without its intended fix. It is identical to go1.11.7, except for its version number. The intended fix is in go1.11.9.

go1.11.9 (released 2019-04-11) fixes an issue where using the prebuilt binary releases on older versions of GNU/Linux [led to failures](#) when linking programs that used cgo. Only Linux users who hit this issue need to update.

go1.11.10 (released 2019-05-06) includes security fixes to the runtime, as well as bug fixes to the linker. See the [Go 1.11.10 milestone](#) on our issue tracker for details.

go1.11.11 (released 2019-06-11) includes a fix to the crypto/x509 package. See the [Go 1.11.11 milestone](#) on our issue tracker for details.

go1.11.12 (released 2019-07-08) includes fixes to the compiler and the linker. See the [Go 1.11.12 milestone](#) on our issue tracker for details.

go1.11.13 (released 2019-08-13) includes security fixes to the `net/http` and `net/url` packages. See the [Go 1.11.13 milestone](#) on our issue tracker for details.

go1.10 (released 2018-02-16)

Go 1.10 is a major release of Go. Read the [Go 1.10 Release Notes](#) for more information.

Minor revisions

go1.10.1 (released 2018-03-28) includes security fixes to the `go` command, as well as bug fixes to the compiler, runtime, and the `archive/zip`, `crypto/tls`, `crypto/x509`, `encoding/json`, `net`, `net/http`, and `net/http/pprof` packages. See the [Go 1.10.1 milestone](#) on our issue tracker for details.

go1.10.2 (released 2018-05-01) includes fixes to the compiler, linker, and `go` command. See the [Go 1.10.2 milestone](#) on our issue tracker for details.

go1.10.3 (released 2018-06-05) includes fixes to the `go` command, and the `crypto/tls`, `crypto/x509`, and `strings` packages. In particular, it adds [minimal support to the go command for the vgo transition](#). See the [Go 1.10.3 milestone](#) on our issue tracker for details.

go1.10.4 (released 2018-08-24) includes fixes to the `go` command, linker, and the `bytes`, `mime/multipart`, `net/http`, and `strings` packages. See the [Go 1.10.4 milestone](#) on our issue tracker for details.

go1.10.5 (released 2018-11-02) includes fixes to the `go` command, linker, runtime, and the `database/sql` package. See the [Go 1.10.5 milestone](#) on our issue tracker for details.

go1.10.6 (released 2018-12-12) includes three security fixes to "go get" and the `crypto/x509` package. It contains the same fixes as Go 1.11.3 and was released at the same time. See the [Go 1.10.6 milestone](#) on our issue tracker for details.

go1.10.7 (released 2018-12-14) includes a fix to a bug introduced in Go 1.10.6 that broke go get for import path patterns containing "...". See the [Go 1.10.7 milestone](#) on our issue tracker for details.

go1.10.8 (released 2019-01-23) includes a security fix to the `crypto/elliptic` package. See the [Go 1.10.8 milestone](#) on our issue tracker for details.

go1.9 (released 2017-08-24)

Go 1.9 is a major release of Go. Read the [Go 1.9 Release Notes](#) for more information.

Minor revisions

go1.9.1 (released 2017-10-04) includes two security fixes. See the [Go 1.9.1 milestone](#) on our issue tracker for details.

go1.9.2 (released 2017-10-25) includes fixes to the compiler, linker, runtime, documentation, `go` command, and the `crypto/x509`, `database/sql`, `log`, and `net/smtp` packages. It includes a fix to a bug introduced in Go 1.9.1 that broke go get of non-Git repositories under certain conditions. See the [Go 1.9.2 milestone](#) on our issue tracker for details.

go1.9.3 (released 2018-01-22) includes security fixes to the `net/url` package, as well as bug fixes to the compiler, runtime, and the `database/sql`, `math/big`, and `net/http` packages. See the [Go 1.9.3 milestone](#) on our issue tracker for details.

go1.9.4 (released 2018-02-07) includes a security fix to "go get". See the [Go 1.9.4 milestone](#) on our issue tracker for details.

go1.9.5 (released 2018-03-28) includes security fixes to the `go` command, as well as bug fixes to the compiler, `go` command, and the `net/http/pprof` package. See the [Go 1.9.5 milestone](#) on our issue tracker for details.

go1.9.6 (released 2018-05-01) includes fixes to the compiler and `go` command. See the [Go 1.9.6 milestone](#) on our issue tracker for details.

go1.9.7 (released 2018-06-05) includes fixes to the `go` command, and the `crypto/x509` and `strings` packages. In particular, it adds [minimal support to the go command for the vgo transition](#). See the [Go 1.9.7 milestone](#) on our issue tracker for details.

go1.8 (released 2017-02-16)

Go 1.8 is a major release of Go. Read the [Go 1.8 Release Notes](#) for more information.

Minor revisions

go1.8.1 (released 2017-04-07) includes fixes to the compiler, linker, runtime, documentation, `go` command and the

crypto/tls, encoding/xml, image/png, net, net/http, reflect, text/template, and time packages. See the [Go 1.8.1 milestone](#) on our issue tracker for details.

go1.8.2 (released 2017-05-23) includes a security fix to the crypto/elliptic package. See the [Go 1.8.2 milestone](#) on our issue tracker for details.

go1.8.3 (released 2017-05-24) includes fixes to the compiler, runtime, documentation, and the database/sql package. See the [Go 1.8.3 milestone](#) on our issue tracker for details.

go1.8.4 (released 2017-10-04) includes two security fixes. It contains the same fixes as Go 1.9.1 and was released at the same time. See the [Go 1.8.4 milestone](#) on our issue tracker for details.

go1.8.5 (released 2017-10-25) includes fixes to the compiler, linker, runtime, documentation, go command, and the crypto/x509 and net/smtp packages. It includes a fix to a bug introduced in Go 1.8.4 that broke go get of non-Git repositories under certain conditions. See the [Go 1.8.5 milestone](#) on our issue tracker for details.

go1.8.6 (released 2018-01-22) includes the same fix in math/big as Go 1.9.3 and was released at the same time. See the [Go 1.8.6 milestone](#) on our issue tracker for details.

go1.8.7 (released 2018-02-07) includes a security fix to "go get". It contains the same fix as Go 1.9.4 and was released at the same time. See the [Go 1.8.7 milestone](#) on our issue tracker for details.

go1.7 (released 2016-08-15)

Go 1.7 is a major release of Go. Read the [Go 1.7 Release Notes](#) for more information.

Minor revisions

go1.7.1 (released 2016-09-07) includes fixes to the compiler, runtime, documentation, and the compress/flate, hash/crc32, io, net, net/http, path/filepath, reflect, and syscall packages. See the [Go 1.7.1 milestone](#) on our issue tracker for details.

go1.7.2 should not be used. It was tagged but not fully released. The release was deferred due to a last minute bug report. Use go1.7.3 instead, and refer to the summary of changes below.

go1.7.3 (released 2016-10-19) includes fixes to the compiler, runtime, and the crypto/cipher, crypto/tls, net/http, and strings packages. See the [Go 1.7.3 milestone](#) on our issue tracker for details.

go1.7.4 (released 2016-12-01) includes two security fixes. See the [Go 1.7.4 milestone](#) on our issue tracker for details.

go1.7.5 (released 2017-01-26) includes fixes to the compiler, runtime, and the crypto/x509 and time packages. See the [Go 1.7.5 milestone](#) on our issue tracker for details.

go1.7.6 (released 2017-05-23) includes the same security fix as Go 1.8.2 and was released at the same time. See the [Go 1.8.2 milestone](#) on our issue tracker for details.

go1.6 (released 2016-02-17)

Go 1.6 is a major release of Go. Read the [Go 1.6 Release Notes](#) for more information.

Minor revisions

go1.6.1 (released 2016-04-12) includes two security fixes. See the [Go 1.6.1 milestone](#) on our issue tracker for details.

go1.6.2 (released 2016-04-20) includes fixes to the compiler, runtime, tools, documentation, and the mime/multipart, net/http, and sort packages. See the [Go 1.6.2 milestone](#) on our issue tracker for details.

go1.6.3 (released 2016-07-17) includes security fixes to the net/http/cgi package and net/http package when used in a CGI environment. See the [Go 1.6.3 milestone](#) on our issue tracker for details.

go1.6.4 (released 2016-12-01) includes two security fixes. It contains the same fixes as Go 1.7.4 and was released at the same time. See the [Go 1.7.4 milestone](#) on our issue tracker for details.

go1.5 (released 2015-08-19)

Go 1.5 is a major release of Go. Read the [Go 1.5 Release Notes](#) for more information.

Minor revisions

go1.5.1 (released 2015-09-08) includes bug fixes to the compiler, assembler, and the `fmt`, `net/textproto`, `net/http`, and `runtime` packages. See the [Go 1.5.1 milestone](#) on our issue tracker for details.

go1.5.2 (released 2015-12-02) includes bug fixes to the compiler, linker, and the `mime/multipart`, `net`, and `runtime` packages. See the [Go 1.5.2 milestone](#) on our issue tracker for details.

go1.5.3 (released 2016-01-13) includes a security fix to the `math/big` package affecting the `crypto/tls` package. See the [release announcement](#) for details.

go1.5.4 (released 2016-04-12) includes two security fixes. It contains the same fixes as Go 1.6.1 and was released at the same time. See the [Go 1.6.1 milestone](#) on our issue tracker for details.

go1.4 (released 2014-12-10)

Go 1.4 is a major release of Go. Read the [Go 1.4 Release Notes](#) for more information.

Minor revisions

go1.4.1 (released 2015-01-15) includes bug fixes to the linker and the `log`, `syscall`, and `runtime` packages. See the [Go 1.4.1 milestone on our issue tracker](#) for details.

go1.4.2 (released 2015-02-17) includes security fixes to the compiler, and bug fixes to the `go` command, the compiler and linker, and the `runtime`, `syscall`, `reflect`, and `math/big` packages. See the [Go 1.4.2 milestone on our issue tracker](#) for details.

go1.4.3 (released 2015-09-22) includes security fixes to the `net/http` package and bug fixes to the `runtime` package. See the [Go 1.4.3 milestone on our issue tracker](#) for details.

go1.3 (released 2014-06-18)

Go 1.3 is a major release of Go. Read the [Go 1.3 Release Notes](#) for more information.

Minor revisions

go1.3.1 (released 2014-08-13) includes bug fixes to the compiler and the `runtime`, `net`, and `crypto/rsa` packages. See the [change history](#) for details.

go1.3.2 (released 2014-09-25) includes security fixes to the `crypto/tls` package and bug fixes to `cgo`. See the [change history](#) for details.

go1.3.3 (released 2014-09-30) includes further bug fixes to `cgo`, the `runtime` package, and the `nacl` port. See the [change history](#) for details.

go1.2 (released 2013-12-01)

Go 1.2 is a major release of Go. Read the [Go 1.2 Release Notes](#) for more information.

Minor revisions

go1.2.1 (released 2014-03-02) includes bug fixes to the `runtime`, `net`, and `database/sql` packages. See the [change history](#) for details.

go1.2.2 (released 2014-05-05) includes a [security fix](#) that affects the `tour` binary included in the binary distributions (thanks to Guillaume T).

go1.1 (released 2013-05-13)

Go 1.1 is a major release of Go. Read the [Go 1.1 Release Notes](#) for more information.

Minor revisions

go1.1.1 (released 2013-06-13) includes a security fix to the compiler and several bug fixes to the compiler and `runtime`. See the [change history](#) for details.

go1.1.2 (released 2013-08-13) includes fixes to the `gc` compiler and `cgo`, and the `bufio`, `runtime`, `syscall`, and `time` packages. See the [change history](#) for details. If you use package `syscall`'s `Getrlimit` and `Setrlimit` functions under Linux on the ARM or 386 architectures, please note change [11803043](#) that fixes [issue 5949](#).

go1 (released 2012-03-28)

Go 1 is a major release of Go that will be stable in the long term. Read the [Go 1 Release Notes](#) for more information.

It is intended that programs written for Go 1 will continue to compile and run correctly, unchanged, under future versions of Go 1. Read the [Go 1 compatibility document](#) for more about the future of Go 1.

The go1 release corresponds to [weekly.2012-03-27](#).

Minor revisions

go1.0.1 (released 2012-04-25) was issued to [fix](#) an [escape analysis bug](#) that can lead to memory corruption. It also includes several minor code and documentation fixes.

go1.0.2 (released 2012-06-13) was issued to fix two bugs in the implementation of maps using struct or array keys: [issue 3695](#) and [issue 3573](#). It also includes many minor code and documentation fixes.

go1.0.3 (released 2012-09-21) includes minor code and documentation fixes.

See the [go1 release branch history](#) for the complete list of changes.

Older releases

See the [Pre-Go 1 Release History](#) page for notes on earlier releases.

Go by Example: Tickers

Timers are for when you want to do something once in the future - *tickers* are for when you want to do something repeatedly at regular intervals. Here's an example of a ticker that ticks periodically until we stop it.

Tickers use a similar mechanism to timers: a channel that is sent values. Here we'll use the `select` builtin on the channel to await the values as they arrive every 500ms.

Tickers can be stopped like timers. Once a ticker is stopped it won't receive any more values on its channel. We'll stop ours after 1600ms.

When we run this program the ticker should tick 3 times before we stop it.

Next example: [Worker Pools](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "time"
)

func main() {

    ticker := time.NewTicker(500 * time.Millisecond)
    done := make(chan bool)

    go func() {
        for {
            select {
            case <-done:
                return
            case t := <-ticker.C:
                fmt.Println("Tick at", t)
            }
        }
    }()

    time.Sleep(1600 * time.Millisecond)
    ticker.Stop()
    done <- true
    fmt.Println("Ticker stopped")
}
```

```
$ go run tickers.go
Tick at 2012-09-23 11:29:56.487625 -0700 PDT
Tick at 2012-09-23 11:29:56.988063 -0700 PDT
Tick at 2012-09-23 11:29:57.488076 -0700 PDT
Ticker stopped
```

Go by Example: Testing and Benchmarking

Unit testing is an important part of writing principled Go programs. The `testing` package provides the tools we need to write unit tests and the `go test` command runs tests.

For the sake of demonstration, this code is in package `main`, but it could be any package. Testing code typically lives in the same package as the code it tests.

We'll be testing this simple implementation of an integer minimum. Typically, the code we're testing would be in a source file named something like `intutils.go`, and the test file for it would then be named `intutils_test.go`.

A test is created by writing a function with a name beginning with `Test`.

`t.Error*` will report test failures but continue executing the test. `t.Fatal*` will report test failures and stop the test immediately.

Writing tests can be repetitive, so it's idiomatic to use a *table-driven style*, where test inputs and expected outputs are listed in a table and a single loop walks over them and performs the test logic.

`t.Run` enables running “subtests”, one for each table entry. These are shown separately when executing `go test -v`.

Benchmark tests typically go in `_test.go` files and are named beginning with `Benchmark`. Any code that's required for the benchmark to run but should not be measured goes before this loop.

The benchmark runner will automatically execute this loop body many times to determine a reasonable estimate of the run-time of a single iteration.

Run all tests in the current project in verbose mode.

```
package main

import (
    "fmt"
    "testing"
)

func IntMin(a, b int) int {
    if a < b {
        return a
    }
    return b
}

func TestIntMinBasic(t *testing.T) {
    ans := IntMin(2, -2)
    if ans != -2 {
        t.Errorf("IntMin(2, -2) = %d; want -2", ans)
    }
}

func TestIntMinTableDriven(t *testing.T) {
    var tests = []struct {
        a, b int
        want int
    }{
        {0, 1, 0},
        {1, 0, 0},
        {2, -2, -2},
        {0, -1, -1},
        {-1, 0, -1},
    }

    for _, tt := range tests {

        testname := fmt.Sprintf("%d,%d", tt.a, tt.b)
        t.Run(testname, func(t *testing.T) {
            ans := IntMin(tt.a, tt.b)
            if ans != tt.want {
                t.Errorf("got %d, want %d", ans, tt.want)
            }
        })
    }
}

func BenchmarkIntMin(b *testing.B) {
    for b.Loop() {

        IntMin(1, 2)
    }
}
```

```
$ go test -v
== RUN    TestIntMinBasic
--- PASS: TestIntMinBasic (0.00s)
=== RUN   TestIntMinTableDriven
=== RUN   TestIntMinTableDriven/0,1
```

Run all benchmarks in the current project. All tests are run prior to benchmarks. The bench flag filters benchmark function names with a regexp.

Next example: [Command-Line Arguments](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
=== RUN    TestIntMinTableDriven/1,0
=== RUN    TestIntMinTableDriven/2,-2
=== RUN    TestIntMinTableDriven/0,-1
=== RUN    TestIntMinTableDriven/-1,0
--- PASS: TestIntMinTableDriven (0.00s)
    --- PASS: TestIntMinTableDriven/0,1 (0.00s)
    --- PASS: TestIntMinTableDriven/1,0 (0.00s)
    --- PASS: TestIntMinTableDriven/2,-2 (0.00s)
    --- PASS: TestIntMinTableDriven/0,-1 (0.00s)
    --- PASS: TestIntMinTableDriven/-1,0 (0.00s)
PASS
ok      examples/testing-and-benchmarking    0.023s

$ go test -bench=.
goos: darwin
goarch: arm64
pkg: examples/testing
BenchmarkIntMin-8 1000000000 0.3136 ns/op
PASS
ok      examples/testing-and-benchmarking    0.351s
```

Build simple, secure, scalable systems with Go

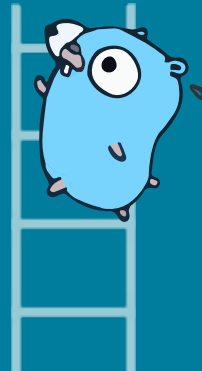
- ✓ An open-source programming language supported by Google
- ✓ Easy to learn and great for teams
- ✓ Built-in concurrency and a robust standard library
- ✓ Large ecosystem of partners, communities, and tools

Get Started

Download

Download packages for [Windows 64-bit](#), [macOS](#), [Linux](#), and [more](#)

The go command by default downloads and authenticates modules using the Go module mirror and Go checksum database run by Google. [Learn more.](#)



Companies using Go

Organizations in every industry use Go to power their software and services. [View all stories](#)




What's possible with Go

Use Go for a variety of software development purposes



Cloud & Network Services

With a strong ecosystem of tools and APIs on major cloud providers, it is easier than ever to build services with Go.

 Popular Packages:

cloud.google.com/go

[aws/client](#)


[Azure/azure-sdk-for-go](#)

[Learn More](#)



Command-line Interfaces

With popular open source packages and a robust standard library, use Go to create fast and elegant CLIs.

 Popular Packages:

[spf13/cobra](#)

[spf13/viper](#)

[urfave/cli](#)

[delve](#)


[chzyer/readline](#)

[Learn More](#)



Web Development

With enhanced memory performance and support for several IDEs, Go powers fast and scalable web applications.

 Popular Packages:

[net/http](#)

[html/template](#)

[flosch/pongo2](#)

[database/sql](#)


[elastic/go-elasticsearch](#)

[Learn More](#)



DevOps & Site Reliability

With fast build times, lean syntax, an automatic formatter and doc generator, Go is built to support both DevOps and SRE.

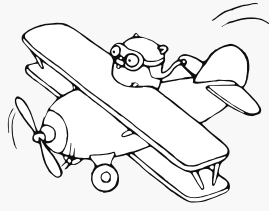
 Popular Packages:

[open-telemetry/opentelemetry-go](#)

[istio/istio](#)

[urfave/cli](#)

[Learn More](#)



[More use cases](#)

Get started with Go

Explore a wealth of learning resources, including guided journeys, courses, books, and more.

[Get Started](#)

[Download Go](#)

RESOURCES TO START ON YOUR OWN

Guided learning journeys

Step-by-step tutorials to get your feet wet

Online learning

Browse resources and learn at your own pace

Featured books

Read through structured chapters and theories

Cloud Self-paced labs

Jump in to deploying Go apps on GCP

IN-PERSON TRAININGS

Ardan Labs

Offering customized on-site live training classes.

Gopher Guides

Customized In-person, remote, and online training classes. Training for Developers by Developers.

Boss Sauce Creative

Personalized or track-based Go training for teams.

Shiju Varghese

On-site classroom training on Go and consulting on distributed systems architectures, in India.

Go by Example: For

for is Go's only looping construct. Here are some basic types of for loops.

The most basic type, with a single condition.

A classic initial/condition/after for loop.

Another way of accomplishing the basic “do this N times” iteration is range over an integer.

for without a condition will loop repeatedly until you break out of the loop or return from the enclosing function.

You can also continue to the next iteration of the loop.

```
package main

import "fmt"

func main() {

    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    for j := 0; j < 3; j++ {
        fmt.Println(j)
    }

    for i := range 3 {
        fmt.Println("range", i)
    }

    for {
        fmt.Println("loop")
        break
    }

    for n := range 6 {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

```
$ go run for.go
1
2
3
0
1
2
range 0
range 1
range 2
loop
1
3
5
```

We'll see some other for forms later when we look at range statements, channels, and other data structures.

Next example: [If/Else](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Slices

Slices are an important data type in Go, giving a more powerful interface to sequences than arrays.

Unlike arrays, slices are typed only by the elements they contain (not the number of elements). An uninitialized slice equals to nil and has length 0.

To create a slice with non-zero length, use the builtin `make`. Here we make a slice of strings of length 3 (initially zero-valued). By default a new slice's capacity is equal to its length; if we know the slice is going to grow ahead of time, it's possible to pass a capacity explicitly as an additional parameter to `make`.

We can set and get just like with arrays.

`len` returns the length of the slice as expected.

In addition to these basic operations, slices support several more that make them richer than arrays. One is the builtin `append`, which returns a slice containing one or more new values. Note that we need to accept a return value from `append` as we may get a new slice value.

Slices can also be copy'd. Here we create an empty slice `c` of the same length as `s` and copy into `c` from `s`.

Slices support a “slice” operator with the syntax `slice[low:high]`. For example, this gets a slice of the elements `s[2]`, `s[3]`, and `s[4]`.

This slices up to (but excluding) `s[5]`.

And this slices up from (and including) `s[2]`.

We can declare and initialize a variable for slice in a single line as well.

The `slices` package contains a number of useful utility functions for slices.

Slices can be composed into multi-dimensional data structures. The length of the inner slices can vary, unlike with multi-dimensional arrays.

```
package main

import (
    "fmt"
    "slices"
)

func main() {

    var s []string
    fmt.Println("uninit:", s, s == nil, len(s) == 0)

    s = make([]string, 3)
    fmt.Println("emp:", s, "len:", len(s), "cap:", cap(s))

    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    fmt.Println("len:", len(s))

    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)

    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    l := s[2:5]
    fmt.Println("sl1:", l)

    l = s[:5]
    fmt.Println("sl2:", l)

    l = s[2:]
    fmt.Println("sl3:", l)

    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)

    t2 := []string{"g", "h", "i"}
    if slices.Equal(t, t2) {
        fmt.Println("t == t2")
    }

    twoD := make([][]int, 3)
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

Note that while slices are different types than arrays, they are rendered similarly by `fmt.Println`.

```
$ go run slices.go
uninit: [] true true
emp: [ ] len: 3 cap: 3
set: [a b c]
get: c
len: 3
apd: [a b c d e f]
cpy: [a b c d e f]
sl1: [c d e]
sl2: [a b c d e]
sl3: [c d e f]
dcl: [g h i]
t == t2
2d: [[0] [1 2] [2 3 4]]
```

Check out this [great blog post](#) by the Go team for more details on the design and implementation of slices in Go.

Now that we've seen arrays and slices we'll look at Go's other key builtin data structure: maps.

Next example: [Maps](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Constants

Go supports *constants* of character, string, boolean, and numeric values.

`const` declares a constant value.

A `const` statement can appear anywhere a `var` statement can.

Constant expressions perform arithmetic with arbitrary precision.

A numeric constant has no type until it's given one, such as by an explicit conversion.

A number can be given a type by using it in a context that requires one, such as a variable assignment or function call. For example, here `math.Sin` expects a `float64`.

```
package main

import (
    "fmt"
    "math"
)

const s string = "constant"

func main() {
    fmt.Println(s)

    const n = 5000000000

    const d = 3e20 / n
    fmt.Println(d)

    fmt.Println(int64(d))

    fmt.Println(math.Sin(n))
}
```

```
$ go run constant.go
constant
6e+11
6000000000000
-0.28470407323754404
```

Next example: [For](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Channels

Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine.

Create a new channel with `make(chan val-type)`. Channels are typed by the values they convey.

Send a value into a channel using the channel `<-` syntax. Here we send "ping" to the messages channel we made above, from a new goroutine.

The `<-` channel syntax *receives* a value from the channel. Here we'll receive the "ping" message we sent above and print it out.

When we run the program the "ping" message is successfully passed from one goroutine to another via our channel.

By default sends and receives block until both the sender and receiver are ready. This property allowed us to wait at the end of our program for the "ping" message without having to use any other synchronization.

Next example: [Channel Buffering](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import "fmt"

func main() {

    messages := make(chan string)

    go func() { messages <- "ping" }()

    msg := <-messages
    fmt.Println(msg)
}
```

```
$ go run channels.go
ping
```

Go by Example: Interfaces

Interfaces are named collections of method signatures.

Here's a basic interface for geometric shapes.

For our example we'll implement this interface on `rect` and `circle` types.

To implement an interface in Go, we just need to implement all the methods in the interface. Here we implement geometry on `rects`.

The implementation for `circles`.

If a variable has an interface type, then we can call methods that are in the named interface. Here's a generic `measure` function taking advantage of this to work on any geometry.

Sometimes it's useful to know the runtime type of an interface value. One option is using a *type assertion* as shown here; another is a [type switch](#).

The `circle` and `rect` struct types both implement the `geometry` interface so we can use instances of these structs as arguments to `measure`.

```
package main

import (
    "fmt"
    "math"
)

type geometry interface {
    area() float64
    perim() float64
}

type rect struct {
    width, height float64
}

type circle struct {
    radius float64
}

func (r rect) area() float64 {
    return r.width * r.height
}

func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func detectCircle(g geometry) {
    if c, ok := g.(circle); ok {
        fmt.Println("circle with radius", c.radius)
    }
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    measure(r)
    measure(c)

    detectCircle(r)
    detectCircle(c)
}
```

```
$ go run interfaces.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
circle with radius 5
```

To understand how Go's interfaces work under the hood, check out this [blog post](#).

Next example: [Enums](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Stateful Goroutines

In the previous example we used explicit locking with [mutexes](#) to synchronize access to shared state across multiple goroutines. Another option is to use the built-in synchronization features of goroutines and channels to achieve the same result. This channel-based approach aligns with Go's ideas of sharing memory by communicating and having each piece of data owned by exactly 1 goroutine.

In this example our state will be owned by a single goroutine. This will guarantee that the data is never corrupted with concurrent access. In order to read or write that state, other goroutines will send messages to the owning goroutine and receive corresponding replies. These `readOp` and `writeOp` structs encapsulate those requests and a way for the owning goroutine to respond.

As before we'll count how many operations we perform.

The reads and writes channels will be used by other goroutines to issue read and write requests, respectively.

Here is the goroutine that owns the state, which is a map as in the previous example but now private to the stateful goroutine. This goroutine repeatedly selects on the reads and writes channels, responding to requests as they arrive. A response is executed by first performing the requested operation and then sending a value on the response channel `resp` to indicate success (and the desired value in the case of reads).

This starts 100 goroutines to issue reads to the state-owning goroutine via the reads channel. Each read requires constructing a `readOp`, sending it over the reads channel, and then receiving the result over the provided `resp` channel.

We start 10 writes as well, using a similar approach.

```
package main

import (
    "fmt"
    "math/rand"
    "sync/atomic"
    "time"
)

type readOp struct {
    key int
    resp chan int
}

type writeOp struct {
    key int
    val int
    resp chan bool
}

func main() {

    var readOps uint64
    var writeOps uint64

    reads := make(chan readOp)
    writes := make(chan writeOp)

    go func() {
        var state = make(map[int]int)
        for {
            select {
            case read := <-reads:
                read.resp <- state[read.key]
            case write := <-writes:
                state[write.key] = write.val
                write.resp <- true
            }
        }
    }()

    for r := 0; r < 100; r++ {
        go func() {
            for {
                read := readOp{
                    key: rand.Intn(5),
                    resp: make(chan int)}
                reads <- read
                <-read.resp
                atomic.AddUint64(&readOps, 1)
                time.Sleep(time.Millisecond)
            }
        }()
    }

    for w := 0; w < 10; w++ {
        go func() {
            for {
                write := writeOp{
                    key: rand.Intn(5),
                    val: rand.Intn(100),
                    resp: make(chan bool)}
                writes <- write
            }
        }()
    }
}
```

Let the goroutines work for a second.

Finally, capture and report the op counts.

Running our program shows that the goroutine-based state management example completes about 80,000 total operations.

For this particular case the goroutine-based approach was a bit more involved than the mutex-based one. It might be useful in certain cases though, for example where you have other channels involved or when managing multiple such mutexes would be error-prone. You should use whichever approach feels most natural, especially with respect to understanding the correctness of your program.

Next example: [Sorting](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
        <-write.resp
        atomic.AddUint64(&writeOps, 1)
        time.Sleep(time.Millisecond)
    }
}

time.Sleep(time.Second)

readOpsFinal := atomic.LoadUint64(&readOps)
fmt.Println("readOps:", readOpsFinal)
writeOpsFinal := atomic.LoadUint64(&writeOps)
fmt.Println("writeOps:", writeOpsFinal)
}
```

```
$ go run stateful-goroutines.go
readOps: 71708
writeOps: 7177
```

Go by Example: Sorting by Functions

Sometimes we'll want to sort a collection by something other than its natural order. For example, suppose we wanted to sort strings by their length instead of alphabetically. Here's an example of custom sorts in Go.

We implement a comparison function for string lengths. `cmp.Compare` is helpful for this.

Now we can call `slices.SortFunc` with this custom comparison function to sort `fruits` by name length.

We can use the same technique to sort a slice of values that aren't built-in types.

Sort people by age using `slices.SortFunc`.

Note: if the `Person` struct is large, you may want the slice to contain `*Person` instead and adjust the sorting function accordingly. If in doubt, [benchmark](#)!

```
package main

import (
    "cmp"
    "fmt"
    "slices"
)

func main() {
    fruits := []string{"peach", "banana", "kiwi"}

    lenCmp := func(a, b string) int {
        return cmp.Compare(len(a), len(b))
    }

    slices.SortFunc(fruits, lenCmp)
    fmt.Println(fruits)

    type Person struct {
        name string
        age  int
    }

    people := []Person{
        Person{name: "Jax", age: 37},
        Person{name: "TJ", age: 25},
        Person{name: "Alex", age: 72},
    }

    slices.SortFunc(people,
        func(a, b Person) int {
            return cmp.Compare(a.age, b.age)
        })
    fmt.Println(people)
}
```

```
$ go run sorting-by-functions.go
[kiwi peach banana]
[{TJ 25} {Jax 37} {Alex 72}]
```

Next example: [Panic](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

Go by Example: Functions

Functions are central in Go. We'll learn about functions with a few different examples.

Here's a function that takes two ints and returns their sum as an int.

Go requires explicit returns, i.e. it won't automatically return the value of the last expression.

When you have multiple consecutive parameters of the same type, you may omit the type name for the like-typed parameters up to the final parameter that declares the type.

Call a function just as you'd expect, with `name(args)`.

There are several other features to Go functions. One is multiple return values, which we'll look at next.

Next example: [Multiple Return Values](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import "fmt"

func plus(a int, b int) int {

    return a + b
}

func plusPlus(a, b, c int) int {
    return a + b + c
}

func main() {

    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

```
$ go run functions.go
1+2 = 3
1+2+3 = 6
```

Go by Example: Rate Limiting

Rate limiting is an important mechanism for controlling resource utilization and maintaining quality of service. Go elegantly supports rate limiting with goroutines, channels, and tickers.

First we'll look at basic rate limiting. Suppose we want to limit our handling of incoming requests. We'll serve these requests off a channel of the same name.

This limiter channel will receive a value every 200 milliseconds. This is the regulator in our rate limiting scheme.

By blocking on a receive from the limiter channel before serving each request, we limit ourselves to 1 request every 200 milliseconds.

We may want to allow short bursts of requests in our rate limiting scheme while preserving the overall rate limit. We can accomplish this by buffering our limiter channel. This `burstyLimiter` channel will allow bursts of up to 3 events.

Fill up the channel to represent allowed bursting.

Every 200 milliseconds we'll try to add a new value to `burstyLimiter`, up to its limit of 3.

Now simulate 5 more incoming requests. The first 3 of these will benefit from the burst capability of `burstyLimiter`.

Running our program we see the first batch of requests handled once every ~200 milliseconds as desired.

For the second batch of requests we serve the first 3 immediately because of the burstable rate limiting, then serve the remaining 2 with ~200ms delays each.

Next example: Atomic Counters.

```
package main

import (
    "fmt"
    "time"
)

func main() {

    requests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        requests <- i
    }
    close(requests)

    limiter := time.Tick(200 * time.Millisecond)

    for req := range requests {
        <-limiter
        fmt.Println("request", req, time.Now())
    }

    burstyLimiter := make(chan time.Time, 3)

    for i := 0; i < 3; i++ {
        burstyLimiter <- time.Now()
    }

    go func() {
        for t := range time.Tick(200 * time.Millisecond) {
            burstyLimiter <- t
        }
    }()

    burstyRequests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        burstyRequests <- i
    }
    close(burstyRequests)
    for req := range burstyRequests {
        <-burstyLimiter
        fmt.Println("request", req, time.Now())
    }
}
```

```
$ go run rate-limiting.go
request 1 2012-10-19 00:38:18.687438 +0000 UTC
request 2 2012-10-19 00:38:18.887471 +0000 UTC
request 3 2012-10-19 00:38:19.087238 +0000 UTC
request 4 2012-10-19 00:38:19.287338 +0000 UTC
request 5 2012-10-19 00:38:19.487331 +0000 UTC
```

```
request 1 2012-10-19 00:38:20.487578 +0000 UTC
request 2 2012-10-19 00:38:20.487645 +0000 UTC
request 3 2012-10-19 00:38:20.487676 +0000 UTC
request 4 2012-10-19 00:38:20.687483 +0000 UTC
request 5 2012-10-19 00:38:20.887542 +0000 UTC
```


Go by Example: Context

In the previous example we looked at setting up a simple [HTTP server](#). HTTP servers are useful for demonstrating the usage of `context.Context` for controlling cancellation. A `Context` carries deadlines, cancellation signals, and other request-scoped values across API boundaries and goroutines.

A `context.Context` is created for each request by the `net/http` machinery, and is available with the `Context()` method.

Wait for a few seconds before sending a reply to the client. This could simulate some work the server is doing. While working, keep an eye on the context's `Done()` channel for a signal that we should cancel the work and return as soon as possible.

The context's `Err()` method returns an error that explains why the `Done()` channel was closed.

As before, we register our handler on the `"/hello"` route, and start serving.

Run the server in the background.

Simulate a client request to `/hello`, hitting `Ctrl+C` shortly after starting to signal cancellation.

Next example: [Spawning Processes](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "net/http"
    "time"
)

func hello(w http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    fmt.Println("server: hello handler started")
    defer fmt.Println("server: hello handler ended")

    select {
    case <-time.After(10 * time.Second):
        fmt.Fprintf(w, "hello\n")
    case <-ctx.Done():

        err := ctx.Err()
        fmt.Println("server:", err)
        internalError := http.StatusInternalServerError
        http.Error(w, err.Error(), internalError)
    }
}

func main() {
    http.HandleFunc("/hello", hello)
    http.ListenAndServe(":8090", nil)
}
```

```
$ go run context.go &
```

```
$ curl localhost:8090/hello
server: hello handler started
^C
server: context canceled
server: hello handler ended
```

Go by Example: Line Filters

A *line filter* is a common type of program that reads input on stdin, processes it, and then prints some derived result to stdout. `grep` and `sed` are common line filters.

Here's an example line filter in Go that writes a capitalized version of all input text. You can use this pattern to write your own Go line filters.

Wrapping the unbuffered `os.Stdin` with a buffered scanner gives us a convenient `Scan` method that advances the scanner to the next token; which is the next line in the default scanner.

`Text` returns the current token, here the next line, from the input.

Write out the uppercased line.

Check for errors during `Scan`. End of file is expected and not reported by `Scan` as an error.

To try out our line filter, first make a file with a few lowercase lines.

Then use the line filter to get uppercase lines.

Next example: [File Paths](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

func main() {

    scanner := bufio.NewScanner(os.Stdin)

    for scanner.Scan() {

        ucl := strings.ToUpper(scanner.Text())

        fmt.Println(ucl)
    }

    if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "error:", err)
        os.Exit(1)
    }
}
```

```
$ echo 'hello' > /tmp/lines
$ echo 'filter' >> /tmp/lines

$ cat /tmp/lines | go run line-filters.go
HELLO
FILTER
```


Go by Example: HTTP Server

Writing a basic HTTP server is easy using the `net/http` package.

A fundamental concept in `net/http` servers is *handlers*. A handler is an object implementing the `http.Handler` interface. A common way to write a handler is by using the `http.HandlerFunc` adapter on functions with the appropriate signature.

Functions serving as handlers take a `http.ResponseWriter` and a `http.Request` as arguments. The response writer is used to fill in the HTTP response. Here our simple response is just “hello\n”.

This handler does something a little more sophisticated by reading all the HTTP request headers and echoing them into the response body.

We register our handlers on server routes using the `http.HandleFunc` convenience function. It sets up the *default router* in the `net/http` package and takes a function as an argument.

Finally, we call the `ListenAndServe` with the port and a handler. `nil` tells it to use the default router we’ve just set up.

Run the server in the background.

Access the `/hello` route.

Next example: [Context](#).

by [Mark McGranaghan](#) and [Eli Bendersky](#) | [source](#) | [license](#)

```
package main

import (
    "fmt"
    "net/http"
)

func hello(w http.ResponseWriter, req *http.Request) {

    fmt.Fprintf(w, "hello\n")
}

func headers(w http.ResponseWriter, req *http.Request) {
    for name, headers := range req.Header {
        for _, h := range headers {
            fmt.Fprintf(w, "%v: %v\n", name, h)
        }
    }
}

func main() {

    http.HandleFunc("/hello", hello)
    http.HandleFunc("/headers", headers)

    http.ListenAndServe(":8090", nil)
}
```

```
$ go run http-server.go &
```

```
$ curl localhost:8090/hello
hello
```