Go by Example

Go is an open source programming language designed for building simple, fast, and reliable software.

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.

by @mmcgrana | feedback | source | license

Go by Example: Hello World

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

```go
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`.

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

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

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

We can then execute the built binary directly.

```
$ ./hello-world
hello world
```

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

Next example: Values.

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.

Go by Example: Variables

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

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

`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 = "short"` in this case.

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

Next example: Constants.

Go by Example: Constants

Go supports *constants* of character, string, boolean, and numeric values.

```
package main

import "fmt"
import "math"

const s string = "constant"

func main() {
    fmt.Println(s)

    const n = 500000000


    const d = 3e20 / n
    fmt.Println(d)

    fmt.Println(int64(d))


    fmt.Println(math.Sin(n))
}
```

`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 cast.

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`.

```
$ go run constant.go
constant
6e+11
600000000000
-0.28470407323754404
```

Next example: For.

Go by Example: For

`for` is Go's only looping construct. Here are three basic types of `for` loops.

The most basic type, with a single condition.

A classic initial/condition/after `for` loop.

`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 := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    for {
        fmt.Println("loop")
        break
    }

    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

```
$ go run for.go
1
2
3
7
8
9
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.

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.

A statement can precede conditionals; any variables declared in this statement are available in all 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 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
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.

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"
import "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 @mmcgrana | feedback | source | license

Go by Example: Arrays

In Go, an *array* is a numbered sequence of elements of a specific length.

```
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)

    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)
}
```

Here we create an array `a` that will hold exactly 5 `int`s. The type of elements and length are both part of the array's type. By default an array is zero-valued, which for `int`s means `0`s.

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.

Array types are one-dimensional, but you can compose types to build multi-dimensional data structures.

```
$ 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]
2d:  [[0 1 2] [1 2 3]]
```

Note that arrays appear in the form `[v1 v2 v3 ...]` when printed with `fmt.Println`.

You'll see *slices* much more often than arrays in typical Go. We'll look at slices next.

Next example: Slices.

Go by Example: Slices

Slices are a key data type in Go, giving a more powerful interface to sequences than arrays.

```
package main

import "fmt"

func main() {
```

Unlike arrays, slices are typed only by the elements they contain (not the number of elements). To create an empty slice with non-zero length, use the builtin `make`. Here we make a slice of `string`s of length `3` (initially zero-valued).

```
    s := make([]string, 3)
    fmt.Println("emp:", s)
```

We can set and get just like with arrays.

```
    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])
```

`len` returns the length of the slice as expected.

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

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.

```
    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)
```

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`.

```
    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)
```

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]`.

```
    l := s[2:5]
    fmt.Println("sl1:", l)
```

This slices up to (but excluding) `s[5]`.

```
    l = s[:5]
    fmt.Println("sl2:", l)
```

And this slices up from (and including) `s[2]`.

```
    l = s[2:]
    fmt.Println("sl3:", l)
```

We can declare and initialize a variable for slice in a single line as well.

```
    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)
```

Slices can be composed into multi-dimensional data structures. The length of the inner slices can vary, unlike with multi-dimensional arrays.

```
    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
emp: [  ]
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]
```

```
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.

Go by Example: Maps

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

```
package main

import "fmt"

func main() {
```

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

```
    m := make(map[string]int)
```

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

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

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

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

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

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

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

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

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

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

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* `_`.

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

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

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

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

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

Next example: Range.

Go by Example: Range

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

```
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)
    }
}
```

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.

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

Next example: Functions.

Go by Example: Functions

Functions are central in Go. We'll learn about functions with a few different examples.

```
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)
}
```

Here's a function that takes two `int`s 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)`.

```
$ go run functions.go
1+2 = 3
1+2+3 = 6
```

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 @mmcgrana | feedback | source | license

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.

```
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)
}
```

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

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 `_`.

```
$ 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.

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.

```
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...)
}
```

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.

```
$ 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.

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.

```
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())
}
```

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.

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

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

Next example: Recursion.

Go by Example: Recursion

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

```
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))
}
```

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

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

Next example: Pointers.

Go by Example: Pointers

Go supports *pointers*, allowing you to pass references to values and records within your program.

```
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)
}
```

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.

```
$ go run pointers.go
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0x42131100
```

`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: Structs.

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.

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.

Access struct fields with a dot.

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

Structs are mutable.

```
package main

import "fmt"

type person struct {
    name string
    age  int
}

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})

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

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

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

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

Next example: Methods.

Go by Example: Methods

Go supports *methods* defined on struct types.

```go
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())
}
```

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.

```
$ go run methods.go
area:  50
perim: 30
area:  50
perim: 30
```

Next we'll look at Go's mechanism for grouping and naming related sets of methods: interfaces.

Next example: Interfaces.

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`.

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"
import "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 main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    measure(r)
    measure(c)
}
```

```
$ go run interfaces.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```

To learn more about Go's interfaces, check out this great blog post.

Next example: Errors.

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 any other, non-error tasks.

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.

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.

In this case we use `&argError` syntax to build a new struct, supplying values for the two fields `arg` and `prob`.

The two loops below test out each of our error-returning functions. Note that the use of an inline error check on the `if` line is a common idiom in Go code.

If you want to programmatically use the data in a custom error, you'll need to get the error as an instance of the custom error type via type assertion.

```
package main

import "errors"
import "fmt"

func f1(arg int) (int, error) {
    if arg == 42 {

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


    }

    return arg + 3, nil
}

type argError struct {
    arg  int
    prob string
}


func (e *argError) Error() string {
    return fmt.Sprintf("%d - %s", e.arg, e.prob)
}

func f2(arg int) (int, error) {
    if arg == 42 {

        return -1, &argError{arg, "can't work with it"}
    }
    return arg + 3, nil
}

func main() {

    for _, i := range []int{7, 42} {
        if r, e := f1(i); e != nil {
            fmt.Println("f1 failed:", e)
        } else {
            fmt.Println("f1 worked:", r)
        }
    }
    for _, i := range []int{7, 42} {
        if r, e := f2(i); e != nil {
            fmt.Println("f2 failed:", e)
        } else {
            fmt.Println("f2 worked:", r)
        }
    }

    _, e := f2(42)
    if ae, ok := e.(*argError); ok {
        fmt.Println(ae.arg)
        fmt.Println(ae.prob)
    }
}
```

```
$ go run errors.go

f1 worked: 10
```

```
f1 failed: can't work with 42
f2 worked: 10
f2 failed: 42 - can't work with it
42
can't work with it
```

See this great post on the Go blog for more on error handling.

Next example: Goroutines.

Go by Example: Goroutines

A *goroutine* is a lightweight thread of execution.

```
package main

import "fmt"

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")

    fmt.Scanln()
    fmt.Println("done")
}
```

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, so execution falls through to here. This `Scanln` requires we press a key before the program exits.

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

When we run this program, we see the output of the blocking call first, then the interleaved output of the two goroutines. This interleaving reflects the goroutines being run concurrently by the Go runtime.

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

Next example: Channels.

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.

```
package main

import "fmt"

func main() {

    messages := make(chan string)


    go func() { messages <- "ping" }()



    msg := <-messages
    fmt.Println(msg)
}
```

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.

```
$ go run channels.go
ping
```

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.

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.

```
package main

import "fmt"

func main() {

    messages := make(chan string, 2)

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



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

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.

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

Next example: Channel Synchronization.

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.

```
package main

import "fmt"
import "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
}
```

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.

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

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

Next example: Channel Directions.

by @mmcgrana | feedback | source | license

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.

```
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)
}
```

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`).

```
$ go run channel-directions.go
passed message
```

Next example: Select.

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.

```
package main

import "time"
import "fmt"

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)
        }
    }
}
```

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.

```
$ time go run select.go
received one
received two

real 0m2.245s
```

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.

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`.

```
package main

import "time"
import "fmt"

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")
    }
}
```

For our example, suppose we're executing an external call that returns its result on a channel `c1` after 2s.

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.

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

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

Using this `select` timeout pattern requires communicating results over channels. This is a good idea in general because other important Go features are based on channels and `select`. We'll look at two examples of this next: timers and tickers.

Next example: Non-Blocking Channel Operations.

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`.

```
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")
    }
}
```

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`.

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

Next example: Closing Channels.

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.

```
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
}
```

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.

```
$ 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
```

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

Next example: Range over Channels.

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.

```
package main

import "fmt"

func main() {

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

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

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

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

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

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 @mmcgrana | feedback | source | license

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.

```
package main

import "time"
import "fmt"

func main() {

    timer1 := time.NewTimer(2 * time.Second)




    <-timer1.C
    fmt.Println("Timer 1 expired")


    timer2 := time.NewTimer(time.Second)
    go func() {
        <-timer2.C
        fmt.Println("Timer 2 expired")
    }()
    stop2 := timer2.Stop()
    if stop2 {
        fmt.Println("Timer 2 stopped")
    }
}
```

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 expired.

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 expires. Here's an example of that.

```
$ go run timers.go
Timer 1 expired
Timer 2 stopped
```

The first timer will expire ~2s after we start the program, but the second should be stopped before it has a chance to expire.

Next example: Tickers.

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.

```
package main

import "time"
import "fmt"

func main() {

    ticker := time.NewTicker(500 * time.Millisecond)
    go func() {
        for t := range ticker.C {
            fmt.Println("Tick at", t)
        }
    }()

    time.Sleep(1600 * time.Millisecond)
    ticker.Stop()
    fmt.Println("Ticker stopped")
}
```

Tickers use a similar mechanism to timers: a channel that is sent values. Here we'll use the `range` builtin on the channel to iterate over 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.

```
$ 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
```

Next example: Worker Pools.

Go by Example: Worker Pools

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

```
package main

import "fmt"
import "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() {

    jobs := make(chan int, 100)
    results := make(chan int, 100)


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

    for j := 1; j <= 5; j++ {
        jobs <- j
    }
    close(jobs)

    for a := 1; a <= 5; a++ {
        <-results
    }
}
```

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.

```
$ 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
```

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.

Next example: Rate Limiting.

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.

```
package main

import "time"
import "fmt"

func main() {
```

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.

```
    requests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        requests <- i
    }
    close(requests)
```

This `limiter` channel will receive a value every 200 milliseconds. This is the regulator in our rate limiting scheme.

```
    limiter := time.Tick(200 * time.Millisecond)
```

By blocking on a receive from the `limiter` channel before serving each request, we limit ourselves to 1 request every 200 milliseconds.

```
    for req := range requests {
        <-limiter
        fmt.Println("request", req, time.Now())
    }
```

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.

```
    burstyLimiter := make(chan time.Time, 3)
```

Fill up the channel to represent allowed bursting.

```
    for i := 0; i < 3; i++ {
        burstyLimiter <- time.Now()
    }
```

Every 200 milliseconds we'll try to add a new value to `burstyLimiter`, up to its limit of 3.

```
    go func() {
        for t := range time.Tick(200 * time.Millisecond) {
            burstyLimiter <- t
        }
    }()
```

Now simulate 5 more incoming requests. The first 3 of these will benefit from the burst capability of `burstyLimiter`.

```
    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())
    }
}
```

Running our program we see the first batch of requests handled once every ~200 milliseconds as desired.

```
$ 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
```

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.

```
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
```

Next example: Atomic Counters.

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.

```
package main

import "fmt"
import "time"
import "sync/atomic"

func main() {

    var ops uint64


    for i := 0; i < 50; i++ {
        go func() {
            for {

                atomic.AddUint64(&ops, 1)




                time.Sleep(time.Millisecond)
            }
        }()
    }

    time.Sleep(time.Second)

    opsFinal := atomic.LoadUint64(&ops)
    fmt.Println("ops:", opsFinal)
}
```

We'll use an unsigned integer to represent our (always-positive) counter.

To simulate concurrent updates, we'll start 50 goroutines that each increment the counter about once a millisecond.

To atomically increment the counter we use `AddUint64`, giving it the memory address of our `ops` counter with the `&` syntax.

Wait a bit between increments.

Wait a second to allow some ops to accumulate.

In order to safely use the counter while it's still being updated by other goroutines, we extract a copy of the current value into `opsFinal` via `LoadUint64`. As above we need to give this function the memory address `&ops` from which to fetch the value.

```
$ go run atomic-counters.go
ops: 41419
```

Running the program shows that we executed about 40,000 operations.

Next we'll look at mutexes, another tool for managing state.

Next example: Mutexes.

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.

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "sync/atomic"
    "time"
)

func main() {
```

For our example the `state` will be a map.

```
    var state = make(map[int]int)
```

This `mutex` will synchronize access to `state`.

```
    var mutex = &sync.Mutex{}
```

We'll keep track of how many read and write operations we do.

```
    var readOps uint64
    var writeOps uint64
```

Here we start 100 goroutines to execute repeated reads against the state, once per millisecond in each goroutine.

```
    for r := 0; r < 100; r++ {
        go func() {
            total := 0
            for {
```

For each read we pick a key to access, `Lock()` the `mutex` to ensure exclusive access to the `state`, read the value at the chosen key, `Unlock()` the mutex, and increment the `readOps` count.

```
                key := rand.Intn(5)
                mutex.Lock()
                total += state[key]
                mutex.Unlock()
                atomic.AddUint64(&readOps, 1)
```

Wait a bit between reads.

```
                time.Sleep(time.Millisecond)
            }
        }()
    }
```

We'll also start 10 goroutines to simulate writes, using the same pattern we did for reads.

```
    for w := 0; w < 10; w++ {
        go func() {
            for {
                key := rand.Intn(5)
                val := rand.Intn(100)
                mutex.Lock()
                state[key] = val
                mutex.Unlock()
                atomic.AddUint64(&writeOps, 1)
                time.Sleep(time.Millisecond)
            }
        }()
    }
```

Let the 10 goroutines work on the `state` and `mutex` for a second.

```
    time.Sleep(time.Second)
```

Take and report final operation counts.

```
    readOpsFinal := atomic.LoadUint64(&readOps)
    fmt.Println("readOps:", readOpsFinal)
    writeOpsFinal := atomic.LoadUint64(&writeOps)
    fmt.Println("writeOps:", writeOpsFinal)
```

With a final lock of `state`, show how it ended up.

```
    mutex.Lock()
    fmt.Println("state:", state)
    mutex.Unlock()
}
```

Running the program shows that we executed about

```
$ go run mutexes.go
```

Running the program shows that we executed about 90,000 total operations against our `mutex`-synchronized `state`.

```
readOps: 83285
writeOps: 8320
state: map[1:97 4:53 0:33 2:15 3:2]
```

Next we'll look at implementing this same state management task using only goroutines and channels.

Next example: Stateful Goroutines.

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 the 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
                <-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)
}
```

Let the goroutines work for a second.

Finally, capture and report the op counts.

```
$ go run stateful-goroutines.go
readOps: 71708
writeOps: 7177
```

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.

Go by Example: Sorting

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

```
package main

import "fmt"
import "sort"

func main() {
```

Sort methods are specific to the builtin type; here's an example for strings. Note that sorting is in-place, so it changes the given slice and doesn't return a new one.

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

An example of sorting `ints`.

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

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

```
    s := sort.IntsAreSorted(ints)
    fmt.Println("Sorted: ", s)
}
```

Running our program prints the sorted string and int slices and `true` as the result of our `AreSorted` test.

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

Next example: Sorting by Functions.

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.

```
package main

import "sort"
import "fmt"

type byLength []string



func (s byLength) Len() int {
    return len(s)
}
func (s byLength) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}
func (s byLength) Less(i, j int) bool {
    return len(s[i]) < len(s[j])
}

func main() {
    fruits := []string{"peach", "banana", "kiwi"}
    sort.Sort(byLength(fruits))
    fmt.Println(fruits)
}
```

In order to sort by a custom function in Go, we need a corresponding type. Here we've created a `byLength` type that is just an alias for the builtin `[]string` type.

We implement `sort.Interface` - `Len`, `Less`, and `Swap` - on our type so we can use the `sort` package's generic `Sort` function. `Len` and `Swap` will usually be similar across types and `Less` will hold the actual custom sorting logic. In our case we want to sort in order of increasing string length, so we use `len(s[i])` and `len(s[j])` here.

With all of this in place, we can now implement our custom sort by casting the original `fruits` slice to `byLength`, and then use `sort.Sort` on that typed slice.

```
$ go run sorting-by-functions.go
[kiwi peach banana]
```

Running our program shows a list sorted by string length, as desired.

By following this same pattern of creating a custom type, implementing the three `Interface` methods on that type, and then calling sort.Sort on a collection of that custom type, we can sort Go slices by arbitrary functions.

Next example: Panic.

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.

```
package main

import "os"

func main() {

    panic("a problem")



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

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 `panic`king 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.

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



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

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.

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.

```
package main

import "fmt"
import "os"

func main() {
```

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.

```
    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")
    f.Close()
}
```

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

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

Next example: Collection Functions.

Go by Example: Collection Functions

We often need our programs to perform operations on collections of data, like selecting all items that satisfy a given predicate or mapping all items to a new collection with a custom function.

In some languages it's idiomatic to use generic data structures and algorithms. Go does not support generics; in Go it's common to provide collection functions if and when they are specifically needed for your program and data types.

Here are some example collection functions for slices of `strings`. You can use these examples to build your own functions. Note that in some cases it may be clearest to just inline the collection-manipulating code directly, instead of creating and calling a helper function.

```
package main

import "strings"
import "fmt"
```

Index returns the first index of the target string `t`, or `-1` if no match is found.

```
func Index(vs []string, t string) int {
    for i, v := range vs {
        if v == t {
            return i
        }
    }
    return -1
}
```

Include returns `true` if the target string t is in the slice.

```
func Include(vs []string, t string) bool {
    return Index(vs, t) >= 0
}
```

Any returns `true` if one of the strings in the slice satisfies the predicate `f`.

```
func Any(vs []string, f func(string) bool) bool {
    for _, v := range vs {
        if f(v) {
            return true
        }
    }
    return false
}
```

All returns `true` if all of the strings in the slice satisfy the predicate `f`.

```
func All(vs []string, f func(string) bool) bool {
    for _, v := range vs {
        if !f(v) {
            return false
        }
    }
    return true
}
```

Filter returns a new slice containing all strings in the slice that satisfy the predicate `f`.

```
func Filter(vs []string, f func(string) bool) []string {
    vsf := make([]string, 0)
    for _, v := range vs {
        if f(v) {
            vsf = append(vsf, v)
        }
    }
    return vsf
}
```

Map returns a new slice containing the results of applying the function `f` to each string in the original slice.

```
func Map(vs []string, f func(string) string) []string {
    vsm := make([]string, len(vs))
    for i, v := range vs {
        vsm[i] = f(v)
    }
    return vsm
```

Here we try out our various collection functions.

```
}

func main() {

    var strs = []string{"peach", "apple", "pear", "plum"}

    fmt.Println(Index(strs, "pear"))

    fmt.Println(Include(strs, "grape"))

    fmt.Println(Any(strs, func(v string) bool {
        return strings.HasPrefix(v, "p")
    }))

    fmt.Println(All(strs, func(v string) bool {
        return strings.HasPrefix(v, "p")
    }))

    fmt.Println(Filter(strs, func(v string) bool {
        return strings.Contains(v, "e")
    }))

    fmt.Println(Map(strs, strings.ToUpper))


}
```

The above examples all used anonymous functions, but you can also use named functions of the correct type.

```
$ go run collection-functions.go
2
false
true
false
[peach apple pear]
[PEACH APPLE PEAR PLUM]
```

Next example: String Functions.

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.

```
package main

import s "strings"
import "fmt"

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"))
    p()

    p("Len: ", len("hello"))
    p("Char:", "hello"[1])
}
```

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.

Not part of `strings`, but worth mentioning here, are the mechanisms for getting the length of a string in bytes and getting a byte by index.

Note that `len` and indexing above work at the byte level. Go uses UTF-8 encoded strings, so this is often useful as-is. If you're working with potentially multi-byte characters you'll want to use encoding-aware operations. See strings, bytes, runes and characters in Go for more information.

```
$ 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

Len:  5
Char: 101
```

Next example: String Formatting.

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.

```
package main

import "fmt"
import "os"

type point struct {
    x, y int
}

func main() {
```

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

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

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

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

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

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

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

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

Formatting booleans is straight-forward.

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

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

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

This prints a binary representation.

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

This prints the character corresponding to the given integer.

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

`%x` provides hex encoding.

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

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

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

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

```
    fmt.Printf("%e\n", 123400000.0)
    fmt.Printf("%E\n", 123400000.0)
```

For basic string printing use `%s`.

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

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

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

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

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

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

```
    fmt.Printf("%p\n", &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.

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

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.

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

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



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

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



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

```
$ go run string-formatting.go
{1 2}
{x:1 y:2}
main.point{x:1, y:2}
main.point
true
123
1110
!
1c8
78.900000
1.234000e+08
1.234000E+08
"string"
"\"string\""
6865782074686973
0x42135100
|    12|   345|
|  1.20|  3.45|
|1.20  |3.45  |
|   foo|     b|
|foo   |b     |
a string
an error
```

Next example: Regular Expressions.

Go by Example: Regular Expressions

Go offers built-in support for regular expressions. Here are some examples of common regexp-related tasks in Go.

```
package main

import "bytes"
import "fmt"
import "regexp"

func main() {
```

This tests whether a pattern matches a string.

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

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

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

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

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

This finds the match for the regexp.

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

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

```
    fmt.Println(r.FindStringIndex("peach punch"))
```

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]+)`.

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

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

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

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.

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

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

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

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

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

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

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

When creating constants with regular expressions you can use the `MustCompile` variation of `Compile`. A plain `Compile` won't work for constants because it has 2 return values.

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

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

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

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

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

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

peach
```

```
[0 5]
[peach ea]
[0 5 1 3]
[peach punch pinch]
[[0 5 1 3] [6 11 7 9] [12 17 13 15]]
[peach punch]
true
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 @mmcgrana | feedback | 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.

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 cast them to their appropriate type. For example here we cast the value in `num` to the expected `float64` type.

Accessing nested data requires a series of casts.

```
package main

import "encoding/json"
import "fmt"
import "os"

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)



    strs := dat["strs"].([]interface{})
```

Accessing nested data requires a series of casts.

```
    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)
}
```

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.

```
$ 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}
```

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: Time.

Go by Example: Time

Go offers extensive support for times and durations; here are some examples.

```
package main

import "fmt"
import "time"

func main() {
    p := fmt.Println
```

We'll start by getting the current time.

```
    now := time.Now()
    p(now)
```

You can build a `time` struct by providing the year, month, day, etc. Times are always associated with a `Location`, i.e. time zone.

```
    then := time.Date(
        2009, 11, 17, 20, 34, 58, 651387237, time.UTC)
    p(then)
```

You can extract the various components of the time value as expected.

```
    p(then.Year())
    p(then.Month())
    p(then.Day())
    p(then.Hour())
    p(then.Minute())
    p(then.Second())
    p(then.Nanosecond())
    p(then.Location())
```

The Monday-Sunday `Weekday` is also available.

```
    p(then.Weekday())
```

These methods compare two times, testing if the first occurs before, after, or at the same time as the second, respectively.

```
    p(then.Before(now))
    p(then.After(now))
    p(then.Equal(now))
```

The `Sub` methods returns a `Duration` representing the interval between two times.

```
    diff := now.Sub(then)
    p(diff)
```

We can compute the length of the duration in various units.

```
    p(diff.Hours())
    p(diff.Minutes())
    p(diff.Seconds())
    p(diff.Nanoseconds())
```

You can use `Add` to advance a time by a given duration, or with a `-` to move backwards by a duration.

```
    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

Next we'll look at the related idea of time relative to the Unix epoch.

Next example: Epoch.

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.

```
package main

import "fmt"
import "time"

func main() {

    now := time.Now()
    secs := now.Unix()
    nanos := now.UnixNano()
    fmt.Println(now)

    millis := nanos / 1000000
    fmt.Println(secs)
    fmt.Println(millis)
    fmt.Println(nanos)

    fmt.Println(time.Unix(secs, 0))
    fmt.Println(time.Unix(0, nanos))
}
```

Use `time.Now` with `Unix` or `UnixNano` to get elapsed time since the Unix epoch in seconds or nanoseconds, respectively.

Note that there is no `UnixMillis`, so to get the milliseconds since epoch you'll need to manually divide from nanoseconds.

You can also convert integer seconds or nanoseconds since the epoch into the corresponding `time`.

```
$ 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
```

Next we'll look at another time-related task: time parsing and formatting.

Next example: Time Formatting / Parsing.

Go by Example: Time Formatting / Parsing

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

```
package main

import "fmt"
import "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)
}
```

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.

```
$ 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.

Go by Example: Random Numbers

Go's `math/rand` package provides pseudorandom number generation.

```
package main

import "time"
import "fmt"
import "math/rand"

func main() {
```

For example, `rand.Intn` returns a random `int` n, `0 <= n < 100`.

```
    fmt.Print(rand.Intn(100), ",")
    fmt.Print(rand.Intn(100))
    fmt.Println()
```

`rand.Float64` returns a `float64` `f`, `0.0 <= f < 1.0`.

```
    fmt.Println(rand.Float64())
```

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

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

The default number generator is deterministic, so it'll produce the same sequence of numbers each time by default. To produce varying sequences, give it a seed that changes. Note that this is not safe to use for random numbers you intend to be secret, use `crypto/rand` for those.

```
    s1 := rand.NewSource(time.Now().UnixNano())
    r1 := rand.New(s1)
```

Call the resulting `rand.Rand` just like the functions on the `rand` package.

```
    fmt.Print(r1.Intn(100), ",")
    fmt.Print(r1.Intn(100))
    fmt.Println()
```

If you seed a source with the same number, it produces the same sequence of random numbers.

```
    s2 := rand.NewSource(42)
    r2 := rand.New(s2)
    fmt.Print(r2.Intn(100), ",")
    fmt.Print(r2.Intn(100))
    fmt.Println()
    s3 := rand.NewSource(42)
    r3 := rand.New(s3)
    fmt.Print(r3.Intn(100), ",")
    fmt.Print(r3.Intn(100))
}
```

```
$ go run random-numbers.go
81,87
0.6645600532184904
7.123187485356329,8.434115364335547
0,28
5,87
5,87
```

See the `math/rand` package docs for references on other random quantities that Go can provide.

Next example: Number Parsing.

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 "strconv"
import "fmt"

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.

Go by Example: URL Parsing

URLs provide a uniform way to locate resources. Here's how to parse URLs in Go.

```
package main

import "fmt"
import "net"
import "net/url"

func main() {
```

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

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

Parse the URL and ensure there are no errors.

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

Accessing the scheme is straightforward.

```
    fmt.Println(u.Scheme)
```

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

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

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

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

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

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

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.

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

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

```
$ 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: SHA1 Hashes.

Go by Example: SHA1 Hashes

SHA1 hashes are frequently used to compute short identities for binary or text blobs. For example, the git revision control system uses SHA1s extensively to identify versioned files and directories. Here's how to compute SHA1 hashes in Go.

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

```
package main

import "crypto/sha1"
import "fmt"

func main() {
    s := "sha1 this string"

    h := sha1.New()

    h.Write([]byte(s))

    bs := h.Sum(nil)

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

The pattern for generating a hash is `sha1.New()`, `sha1.Write(bytes)`, then `sha1.Sum([]byte{})`. 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.

SHA1 values are often printed in hex, for example in git commits. Use the `%x` format verb to convert a hash results to a hex string.

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

```
$ go run sha1-hashes.go
sha1 this string
cf23df2207d99a74fbe169e3eba035e633b65d94
```

You can compute other hashes using a similar pattern to the one shown above. For example, to compute MD5 hashes import `crypto/md5` and use `md5.New()`.

Note that if you need cryptographically secure hashes, you should carefully research hash strength!

Next example: Base64 Encoding.

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 cast 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.

```
package main

import b64 "encoding/base64"
import "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))
}
```

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.

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

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

Next example: Reading Files.

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.

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)
```

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

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

func main() {
```

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

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

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.

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

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

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

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

```
    o2, err := f.Seek(6, 0)
    check(err)
    b2 := make([]byte, 2)
    n2, err := f.Read(b2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n2, o2, string(b2))
```

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`.

```
    o3, err := f.Seek(6, 0)
    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))
```

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

```
    _, err = f.Seek(0, 0)
    check(err)
```

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.

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

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

```
    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.

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.

```
package main

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

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

func main() {

    d1 := []byte("hello\ngo\n")
    err := ioutil.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")
    fmt.Printf("wrote %d bytes\n", n3)

    f.Sync()

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

    w.Flush()


}
```

Try running the file-writing code.

Then check the contents of the written files.

```
$ 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
```

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.

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.

```
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)
    }
}
```

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.

```
$ echo 'hello'   > /tmp/lines
$ echo 'filter' >> /tmp/lines

$ cat /tmp/lines | go run line-filters.go
HELLO
FILTER
```

Next example: Command-Line Arguments.

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.

```
package main

import "os"
import "fmt"

func main() {

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



    arg := os.Args[3]

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

`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.

```
$ 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
```

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

Next example: Command-Line Flags.

by @mmcgrana | feedback | 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.

```
package main

import "flag"
import "fmt"


func main() {

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




    numbPtr := flag.Int("numb", 42, "an int")
    boolPtr := 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:", *boolPtr)
    fmt.Println("svar:", svar)
    fmt.Println("tail:", flag.Args())
}
```

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

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

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

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

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

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

Trailing positional arguments can be provided after any flags.

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

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

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

Use `-h` or `--help` flags to get automatically generated help

```
$ ./command-line-flags -h
```

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

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

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.

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

Next example: Environment Variables.

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.

```
package main

import "os"
import "strings"
import "fmt"

func main() {

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


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

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.Split` them to get the key and value. Here we print all the keys.

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

TERM_PROGRAM
PATH
SHELL
...

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

Running the program shows that we pick up the value for `FOO` 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: Spawning Processes.

Go by Example: Spawning Processes

Sometimes our Go programs need to spawn other, non-Go processes. For example, the syntax highlighting on this site is implemented by spawning a `pygmentize` process from a Go program. Let's look at a few examples of spawning processes from Go.

```
package main

import "fmt"
import "io/ioutil"
import "os/exec"

func main() {
```

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.

```
    dateCmd := exec.Command("date")
```

`.Output` is another helper that handles the common case of running a command, waiting for it to finish, and collecting its output. If there were no errors, `dateOut` will hold bytes with the date info.

```
    dateOut, err := dateCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> date")
    fmt.Println(string(dateOut))
```

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`.

```
    grepCmd := exec.Command("grep", "hello")
```

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.

```
    grepIn, _ := grepCmd.StdinPipe()
    grepOut, _ := grepCmd.StdoutPipe()
    grepCmd.Start()
    grepIn.Write([]byte("hello grep\ngoodbye grep"))
    grepIn.Close()
    grepBytes, _ := ioutil.ReadAll(grepOut)
    grepCmd.Wait()

    fmt.Println("> grep hello")
    fmt.Println(string(grepBytes))
```

We ommited 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:

```
    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))
}
```

The spawned programs return output that is the same as if we had run them directly from the command-line.

```
$ go run spawning-processes.go
> date
Wed Oct 10 09:53:11 PDT 2012

> 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.

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.

```
package main

import "syscall"
import "os"
import "os/exec"

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)
    }
}
```

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 apposed 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.

```
$ 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
```

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.

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.

```
package main

import "fmt"
import "os"
import "os/signal"
import "syscall"

func main() {

    sigs := make(chan os.Signal, 1)
    done := make(chan bool, 1)



    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)



    go func() {
        sig := <-sigs
        fmt.Println()
        fmt.Println(sig)
        done <- true
    }()

    fmt.Println("awaiting signal")
    <-done
    fmt.Println("exiting")
}
```

Go signal notification works by sending `os.Signal` values on a channel. We'll create a channel to receive these notifications (we'll also make one to notify us when the program can exit).

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

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.

```
$ go run signals.go
awaiting signal
^C
interrupt
exiting
```

Next example: Exit.

Go by Example: Exit

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

```
package main

import "fmt"
import "os"

func main() {

    defer fmt.Println("!")


    os.Exit(3)
}
```

`defer`s 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.

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


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

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

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