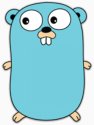
Programming in Go
Lesson 1: The Basics

Matt Holiday

16 April 2019

Cardinal Peak



Lesson # 1

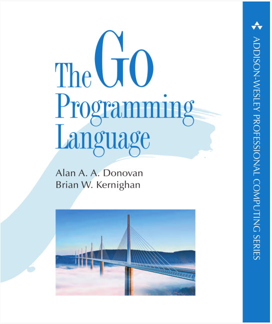
What we'll cover today:

- Installation
- Running a simple program
- Simple types
- Declarations
- Initialization
- Assignment & type conversion
- Reference types
- Basic control structures
- Standard I/O & simple formatting

The Book

Hereinafter referred to as *GOPL*

I will be taking exercise material from this book



Amazon paper: $28

`informit.com` PDF: $19
(with coupon `IUGD45`)

"Anything with Brian Kernighan's name on it is worth reading."
— Matt Holiday

Installation

Start from the Go language page: https://golang.org

Mac: run `brew install go` (or use the installer package)

Homebrew installation: https://brew.sh

Windows: open the installer (MSI) file and follow the prompts to install the Go tools

(otherwise you can download a ZIP file, but you have to set some environment stuff)

Linux: download the archive and extract it into `/usr/local`, creating a Go tree in `/usr/local/go`

```
sudo tar -C /usr/local -xzf go1.12.4.linux-amd64.tar.gz
```

and don't forget to add `/usr/local/go/bin` to `$PATH` (Linux)

GOPATH environment variable

If you work out of `$HOME/go/src` you don't need to set it

Otherwise for what we're doing in class you'll need it

The Go command page tells you more than you want to know

Hello, world!

What the simplest program looks like:

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

Running a program

From the command line:

```
$ go run hello.go
Hello, world!

$ go run sieve.go 49
15: [2 3 5 7 11 13 17 19 23 29 31 37 41 43 47]
```

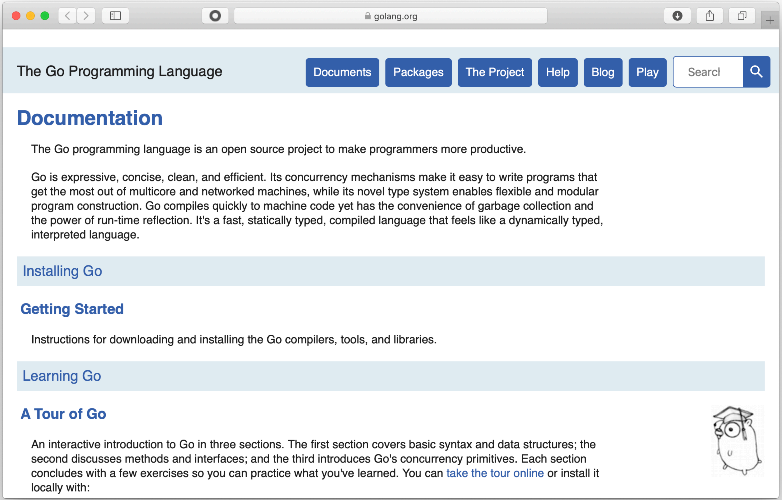
Later we'll talk about how to build binaries that stick around

Hello, playground!

Simple programs run at the Go playground

More information

Get all the info at https://golang.org/doc/



Something more complicated

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

Something more complicated

```go
package main

import "fmt"

// Find primes in the range 2..n and return them in a slice

func sieveOfEratosthenes(n int) []int {
    // Create a boolean array [0..n] as all true, and then
    // set entries false as they are found not to be prime

    integers := make([]bool, n+1) // why "n+1" and not "n"?

    for i := range integers {
        integers[i] = true
    }

    integers[0], integers[1] = false, false
```

Something more complicated

```go
// We "cross out" multiples of each prime because they
// are divisible by that prime number; e.g., when p =
// 2 we remove 4, 6, 8, ... and when p = 3  we remove
// 9, 12, 15, ... and skip 6 because 6 = 2*3 was seen

for p := 2; p*p <= n; p++ {
    // If integers[p] is not changed, then it is prime

    if integers[p] {
        // Update values x, x+p, x+2p, ... as not prime
        // starting with x = p*p (see above)

        for i := p * p; i <= n; i += p {
            integers[i] = false
        }
    }
}
```

Something more complicated

```go
    // Now pick out the primes and return only them

    var primes []int  // we don't know how many yet

    for p := range integers {
        if integers[p] {
            primes = append(primes, p)
        }
    }

    return primes
}

func main() {
    p := sieveOfEratosthenes(121)

    fmt.Printf("%d: %v\n", len(p), p)
}
```

Read a number from the command line

```go
import (
    "fmt"
    "os"
    "strconv"
)

. . .

func main() {
    s := os.Args[1]  // we skipped the error checking!

    if n, err := strconv.ParseInt(s, 10, 64); err != nil {
        fmt.Fprintln(os.Stderr, "invalid int:", s)
    } else {
        p := sieveOfEratosthenes(int(n))
        fmt.Printf("%d: %v\n", len(p), p)
    }
}
```

Running a program with a bug

From the command line:

```
$ go run sieve.go   ## no number given
panic:  runtime error:  index out of range
goroutine 1 [running]:
main.main()
/Users/mholiday/go/src/sieve1/sieve.go:55 +0x202
exit status 2
```

What went wrong? We read past the end of `os.Args`!

Basic Stuff

Keywords & symbols

Only 25 keywords; you may not use these as names:

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

Plus a bunch of operators & symbols:

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

Predeclared identifiers

You can use these as names, shadowing the built-in meaning, but you really don't want to do that!

Constants:

```
true false iota nil
```

Types:

```
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr
float32 float64 complex64 complex128
bool byte rune string error
```

Functions:

```
make len cap new append copy close delete
complex real imag
panic recover
```

Simple types

Integers:

- "unsized" (defaults to the machine's natural wordsize):
  `int`, `uint`
  - on my Core i7 laptop, these are 64 bits in size
  - on my Raspberry Pi, these are 32 bits in size

  `int` is the default type for integers in Go, even lengths

- sized, signed:
  `int8 int16 int32 int64`

- sized, unsigned:
  `uint8 uint16 uint32 uint64 uintptr`

Simple types

“Real” number types:

- floating point numbers:
  `float32 float64`

- complex (imaginary) floating point numbers:
  `complex64 complex128`

Don’t ever use floating point for monetary calculations!

https://www.exploringbinary.com/why-0-point-1-does-not-exist-in-floating-point/

Number conversions

Conversions may change the value

```
var size float32 = 1.25

y := int(size)        // truncated to 1
z := float32(y)       // still 1.0 from 1
```

Once the number's been rounded down, it stays that way

Integer conversions to a smaller size take the bits that fit

```
var a uint32 = 66000
var b uint32 = 2000000

m := int16(a)         // 464
n := int16(b)         // -31616
```

Number conversions

The 32-bit values get truncated; high bit set $\implies$ negative

```go
package main
import "fmt"

func main() {
    var a, b uint32 = 66000, 2000000

    m, n := int16(a), int16(b) // 464, -31616

    fmt.Printf("%032b %016b\n", a, uint16(m))
    fmt.Printf("%032b %016b\n", b, uint16(n))
}
```

```
00000000000000010000000111010000 0000000111010000
00000000000111101000010010000000 1000010010000000
```

Simple types

Types related to strings:

- `byte`: a synonym for `uint8`
- `rune`: a synonym for `int32` for characters
- `string`: an immutable sequence of “characters”
  - physically a sequence of `byte`
  - logically a sequence of `rune`

Runes (characters) are enclosed in single quotes: `'a'`

“Raw” strings use backtick quotes: `` `string with "quotes"` ``

They also don’t evaluate escape characters such as `\n`

String-related types

Let's see `rune` vs `byte` in a string:

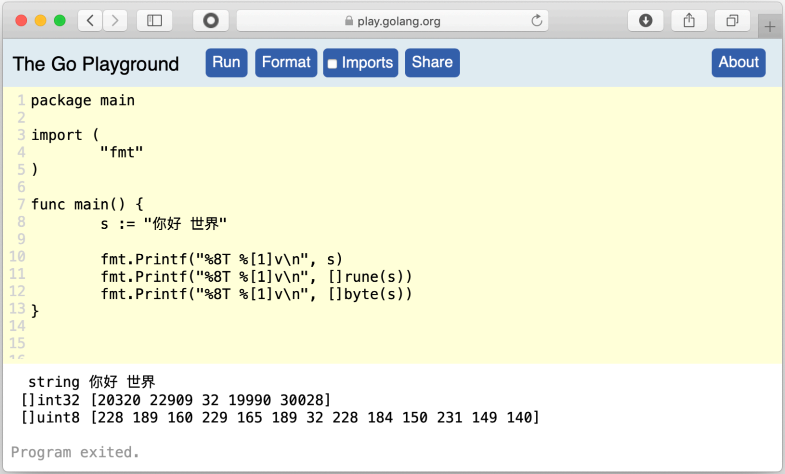
```go
package main
import "fmt"

func main() {
    s := "élite"
    fmt.Printf("%8T %[1]v\n", s)
    fmt.Printf("%8T %[1]v\n", []rune(s))
    fmt.Printf("%8T %[1]v\n", []byte(s))
}
```

é is one rune (character) but two bytes in UTF-8 encoding:

```
  string élite
 []int32 [233 108 105 116 101]
 []uint8 [195 169 108 105 116 101]
```

String-related types, in Chinese

I can’t do this in the slides:



Simple types

Special types:

- `bool` (boolean) has two values `false`, `true`

  these values are **not** convertible to/from integers!

- `error`: an interface type with one function

  `func (e *error) Error() string`

  an `error` may be nil or non-nil

- Pointers are physically addresses, logically opaque

  a pointer may be nil or non-nil
  no pointer manipulation except through package `unsafe`

Numeric literals

Go keeps “arbitrary” precision for literal values (at least 256 bits)

- Integer literals are untyped
    - assign a literal to any size integer without conversion
    - assign an integer literal to float, complex also

- Ditto float and complex; picked by syntax of the literal

- Mathematical constants can be very precise

```
Pi  = 3.14159265358979323846264338327950288419716939937510582097494459
```

- Constant arithmetic done at compile time doesn’t lose precision

Constants

Only numbers and strings can be constants

Constant can be a literal or a compile-time function of a constant

```
const (
    a = 1                   // int
    b = 2 * 1024            // 2048
    c = b << 3              // 16384

    g uint8 = 0x07          // 7
    h uint8 = g & 0x03      // 3

    s = "a string"
    t = len(s)              // 8
    u = s[2:]               // SYNTAX ERROR
)
```

Declaration

There are six ways to introduce a name:

- Constant declaration with `const`
- Type declaration with `type`
- Variable declaration with `var`
  (must have type or initial value, sometimes both)
- Short, initialized variable declaration of any type `:=`
  only inside a function
- Declaration of a function at package level with `func`
  (methods may *only* be declared at package level)
- Formal parameters and named returns of a function

Initialization

Go initializes all variables to “zero” by default if you don’t:

- All numerical types get `0` (float `0.0`, complex `0i`)
- `bool` gets `false`
- `string` gets `""` (the empty string, length 0)
- Everything else gets `nil`:
    - pointers
    - slices
    - maps
    - channels
    - functions (function variables)
    - interfaces
- For aggregate types, all members get their “zero” values

Examples

```go
// x and y get the values passed in by the caller
// the (unnamed) return value gets the "return" expression

func do(x, y int) int {
    const t = 21              // type int by default
    const z = false           // type bool from the value

    var i uint8 = 255         // explicit type uint8
    var j = 256               // type int by default
    var k int                 // 0 by default

    var m                     // SYNTAX ERROR, no type/value

    j := 0                    // short declaration, int
    v := func() { ... }       // short declaration, function

    return k
}
```

Examples

```go
// explicit conversion is required for integer types
// and inc-/decrement operators can't be expressions

func do(x, y int) int {
    k := x + y                  // k int
    m := uint32(k)              // int conversion

    k = m                       // TYPE MISMATCH

    var i uint8 = 255

    j := i++                    // SYNTAX ERROR
    --i                         // SYNTAX ERROR

    b := k = 0                  // SYNTAX ERROR

    return m                    // TYPE MISMATCH
}
```

Basic operators

Arithmetic: numbers only except + on string

```
+    -    *    /    %    ++    --
```

Comparison: only numbers/strings support order

```
==   !=   <    >    <=   >=
```

Boolean: only booleans, with shortcut evaluation

```
!    &&   ||
```

Bitwise: operate on integers

```
&    |    ^    <<   >>   &^
```

Assignment: as above for binary operations

```
=    +=   -=   *=   /=   %=
&=   |=   ^=   <<=  >>=  &^=
```

Operator gotchas

The division operator / has two meanings

```
i := 3/2            // integer division
fmt.Println(i)      // 1

j := 51/25
fmt.Println(j)      // 2

t := 3.0/2.0        // real division
fmt.Println(t)      // 1.5
```

Aside: why are i, j, and k often used as loop index variables?

They were integer variables by default in Fortran (names i–n)

Operator precedence

There are only five levels of precedence, otherwise left-to-right:

Operators like multiplication:

`*    /    %    <<    >>    &    &^`

Operators like addition:

`+    -    |    ^`

Comparison operators:

`==    !=    <    <=    >    >=`

Logical and:

`&&`

Logical or:

`||`

Examples

```go
// function has been called with x=1, y=2

func do(x, y int) int {
	k := 3 * x + y          // 5  by precedence
	m := 3 * (x + y)        // 9
	n := 3 * 5 * x + y      // 17
	p := 3 * (5 * x) + y    // 17
	v := 3 + 5 * x + y      // 10 by precedence
	w := (3 + 5) * x + y    // 10
	z := 3 + 5 * (x + y)    // 18

	b := n < 3 || p > 5     // true (2nd clause)
	c := n > 5 || z < 9     // true (1st clause)

	return k
}
```

Examples

```go
// function has been called with x=1, y=2

func do(x, y byte) byte {
    t := x & y              // 0
    u := x | y              // 3
    v := x &^ y             // 1
    w := ^y                 // 253
    z := x << y             // 4

    a := uint8(255)         // else signed

    a++                     // 0
    a--                     // 255

    b := x || y             // SYNTAX ERROR

    return 42               // ok, literal
}
```

Composite Types

Strings

Strings are a sequence of characters and are **immutable**

The built-in `len` function calculates the length

Strings overload the addition operator (+ and +=)

```
s := "the quick brown fox"

a := len(s)                       // 19
b := s[:3]                        // "the"
c := s[4:9]                       // "quick"
d := s[:4] + "slow" + s[9:]       // replaces "quick"

s[5] = 'a'                        // SYNTAX ERROR
s += "es"                         // now plural (copied)
```

Strings are passed *by reference*, thus they aren't copied

String functions

Package `strings` has many functions on strings

```
s := "a string"

x := len(s)                      // built-in, = 8

strings.Contains(s, "g")     // returns true
strings.Contains(s, "x")     // returns false

strings.HasPrefix(s, "a")    // returns true
strings.ToUpper(s)           // returns "A STRING"
```

Indexes in strings are numbered from 0 up to `len(s) - 1`

```
strings.Index(s, "string")   // returns 2
```

Arrays

Arrays are typed by size, which is *fixed* at compile time

```
// all these are equivalent

var a [3]int
var b [3]int{0, 0, 0}
var c [...]{0, 0, 0}   // sized by initializer

var d [3]int
d = b                  // elements copied

var m [...]int{1, 2, 3, 4}

c = m                  // TYPE MISMATCH
```

Arrays are passed *by value*, thus elements are copied

Slices

Slices have variable length, backed by some array; they are copied when they outgrow that array

```
var a []int              // nil, no storage
var b = []int{}          // empty but non-nil
var c = []int{1, 2}      // non-empty

a = append(a, 1)         // append to nil OK
len(a), cap(a)           // 2, 2

d := make([]int, 5)
e := make([]int, 0, 5)  // capacity but empty

len(d), cap(d)           // 5, 5
len(e), cap(e)           // 0, 5
```

Slices are passed *by reference;* no copying, updating OK

Slices

```go
package main
import "fmt"

func main() {
    t := []byte("string")   // 0:s 1:t 2:r 3:i 4:n 5:g

    fmt.Println(len(t), t)  // 6 bytes in t
    fmt.Println(t[2])       //   1 item
    fmt.Println(t[:2])      //   2 items
    fmt.Println(t[2:])      // 6-2 items
    fmt.Println(t[3:5])     // 5-3 items
}
```

```
6 [115 116 114 105 110 103]
114
[115 116]
[114 105 110 103]
[105 110]
```

Slices vs arrays

Most Go APIs take slices as inputs, not arrays

Slice	**Array**
Variable length	Length fixed at compile time
Passed by reference	Passed by value (copied)
Not comparable	Comparable (==)
Cannot be used as map key	Can be used as map key
Has `copy` & `append` helpers	—
Useful as function parameters	Useful as “pseudo” constants

Arrays as pseudo-constants

It can be useful to have fixed-size tables of values in some algorithms, treated as constant data

```go
// from the file crypto/des/const.go in the DES package

// Used to perform an initial permutation of a 64-bit
// input block.
var initialPermutation = [64]byte{
	6, 14, 22, 30, 38, 46, 54, 62,
	4, 12, 20, 28, 36, 44, 52, 60,
	2, 10, 18, 26, 34, 42, 50, 58,
	0,  8, 16, 24, 32, 40, 48, 56,
	7, 15, 23, 31, 39, 47, 55, 63,
	5, 13, 21, 29, 37, 45, 53, 61,
	3, 11, 19, 27, 35, 43, 51, 59,
	1,  9, 17, 25, 33, 41, 49, 57,
}
```

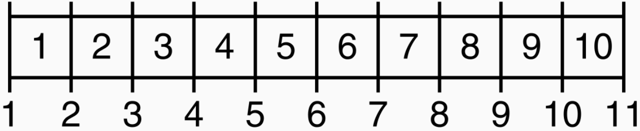
The off-by-one bug

Slices are indexed like `[1:3]`

(read as the starting element and *one past* the ending element, so this way we have $3 - 1 = 2$ elements in our slice)

For loops work the same way in most cases:

```
for i := 1; i < 5; i++ {  // in math written [1, 5)
    . . .
}
```



Read it on Wikipedia OB1

Examples

```go
var w = [...]int{1, 2, 3}   // array of len(3)
var x = []int{0, 0, 0}      // slice of len(3)

func do(a [3]int, b []int) []int {
    a = b                   // SYNTAX ERROR
    a[0] = 4                // w unchanged
    b[0] = 3                // x changed

    c := make([]int, 5)     // len/cap 5
    c[4] = 42
    copy(c, b)              // copies only 3 elts
    b = append(b, c...)     // reallocates!
    return c
}

y := do(w, x)
fmt.Println(w, x, y)        // [1 2 3] [3 0 0] [3 0 0 0 42]
```

Maps

Maps are dictionaries: indexed by key, returning a value

You can read from a nil map, but inserting will panic

```
var m map[string]int          // nil, no storage
p := make(map[string]int)    // non-nil but empty

a := m["the"]                 // returns 0
m["and"] = 1                  // PANIC
m = p
m["and"]++                    // OK, same map as p now
c := p["and"]                 // returns 1
```

Maps are passed *by reference;* no copying, updating OK

The type used for the key must have == and `!=` defined
not slices, maps, or functions

Maps

Maps can’t be compared to one another; maps can be compared only to `nil` as a special case

```
var m  = map[string]int{
    "and": 1,
    "the": 1,
    "or":  2,
}

var n map[string]int

b := m == n                    // SYNTAX ERROR
c := n == nil                  // true
d := len(m)                    // 3
e := cap(m)                    // TYPE MISMATCH
```

Maps

Maps have a special two-result lookup function

The second variable tells you if they key was there

```
p := map[string]int{}        // non-nil but empty

a := p["the"]                // returns 0
b, ok := p["and"]            // 0, false

p["the"]++

c, ok := p["the"]            // 1, true

if w, ok := p["the"]; ok {
    // we know w is not the default value
    . . .
}
```

Make nil useful

Nil is a type of zero: it indicates the absence of something

Many built-ins are safe: `len`, `cap`, `range`

```go
// nil -- no options
jar, err := cookiejar.New(nil)

// nil function -- use the default
http.ListenAndServe("localhost:8080", nil)

// nil []int slice -- skip the loop
for _, v := range values {
    total += v
}
```

“Make the zero value useful.” — Rob Pike

See Francesc Campoy’s video at https://www.youtube.com/watch?v=ynoY2xz-F8s

Built-ins

Each type has certain built-in functions

`len(s)`	`string`	`string length`
`len(a), cap(a)`	`array`	`array length, capacity (constant)`
`make(T, x)`	`slice`	`slice of type T with length x and capacity x`
`make(T, x, y)`	`slice`	`slice of type T with length x and capacity y`
`copy(c, d)`	`slice`	`copy from d to c; # = min of the two lengths`
`c=append(c, d)`	`slice`	`append d to c and return a new slice result`
`len(s), cap(s)`	`slice`	`slice length and capacity`
`make(T)`	`map`	`map of type T`
`make(T, x)`	`map`	`map of type T with space hint for x elements`
`delete(m, k)`	`map`	`delete key k (if present, else no change)`
`len(m)`	`map`	`map length`

Control Structures

Sequence

The simplest type of program has no “structures”

It just flows from top to bottom, executing statements in sequence

```go
package main
import (
	"fmt"
	"math"
)

func main() {
	a, b, c := -0.5, 0.5, 5.0
	x := math.Sqrt(b*b - 4*a*c) / (2 * a)
	y1, y2 := -b + x, -b - x

	fmt.Printf("%5.4f, %5.4f\n", y1, y2)
	// -3.7016, 2.7016
}
```

If-then-else

The next type of structure is a choice between alternatives

All if-then statements require braces

```
if a == b {
    fmt.Println("a equals b")
} else {
    fmt.Println("a is not equal to b")
}
```

They can start with a short declaration or statement

```
if err := doSomething(); err != nil {
    return err
}
```

Switch

A switch is another choice between alternatives

It is a shortcut replacing a series of if-then statements

```
switch a := f.Get(); a {
case 0, 1, 2:
    fmt.Println("underflow possible")

case 3, 4, 5, 6, 7, 8:

default:
    fmt.Println("warning: overload")
}
```

Alternatives may be empty and **do not fall through** (as in C) so a `break` is not required

Switch on true

Arbitrary comparisons may be made for an switch with no argument

```go
a := f.Get()

switch {
case a <= 2:
    fmt.Println("underflow possible")

case a <= 8:
    // evaluated in order

default:
    fmt.Println("warning: overload")
}
```

For loops

The loop control structure provides automatic repetition

There is only `for` (no `do` or `while`) but with options

1. Explicit control with an index variable

```
for i := 0; i < 10; i++ {
    fmt.Printf("(%d, %d)\n", i, i*i)
}

// prints (0, 0) up to (9, 81)
```

Three parts, all optional (initialize, check, increment)

The loop ends when the explict check fails (e.g., `i == 10`)

For loops

2. Implicit control through the `range` operator for arrays, slices, and maps (among others)

```
// one var: i is an index 0, 1, 2, ...

for i := range anArray {
    fmt.Println(i, anArray[i])
}

// two vars: i is the index, v is a value

for i, v := range anArray {
    fmt.Println(i, v)
}
```

The loop ends when the range is exhausted

For loops

3. An infinite loop with an explicit `break`

```go
i, j := 0, 3

// this loop must be made to stop

for {
    i, j = i + 50, j * j

    fmt.Println(i, j)

    if j > i {
        break       // when i = 150, j = 6561
    }
}
```

There is also `continue` to make an iteration start over

For loops

Here's a **common mistake**

If you only want `range` values, you need the blank identifier:

```
// two vars: _ is the index (ignored),
// v is a value

for _, v := range anArray {
    fmt.Println(v)
}
```

Sometimes you may get a compile error for a type mismatch

The _ is an untyped, reusable "variable" placeholder

Input/Output

Standard I/O

Unix has the notion of three standard I/O streams

They're open by default in every program

Most modern programming languages have followed this convention:

- Standard input
- Standard output
- Standard error (output)

These are normally mapped to the console/terminal but can be *redirected*

```
find . -name '*.go' | xargs grep -n "rintf" > print.txt
```

Formatted I/O

We've been using the `fmt` package to do I/O

By default, we've been printing to standard output

```go
package main

import (
	"fmt"
	"os"
)

func main() {
	fmt.Println("printing a line to standard output")

	fmt.Fprintln(os.Stderr, "printing to error output")
}
```

Printing command-line arguments

```go
package main

import "fmt"
import "os"

func main() {
    var s, sep string

    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }

    fmt.Println(s)
}
```

We'll skip `os.Arg[0]` because that's the program name:

```
/var/folders/6y/q8z5w4xn1dzb0_qz680mcs6h0000gn/T/go-build451715571/b001/exe/hello
```

A whole family of functions

The `fmt` package uses reflection and can print anything;
some of the functions take a *format string*

```
// always os.Stdout

fmt.Println(...interface{}) (int, error)
fmt.Printf(string, ...interface{}) (int, error)

// print to anything that has the correct Write() method

fmt.Fprintln(io.Writer, ...interface{}) (int, error)
fmt.Fprintf(io.Writer, string, ...interface{}) (int, error)

// return a string

fmt.Sprintln(...interface{}) string
fmt.Sprintf(string, ...interface{}) string
```

Format codes

The `fmt` package uses format codes reminiscent of C

```
%s   the uninterpreted bytes of the string or slice
%q   a double-quoted string safely escaped with Go syntax
%c   the character represented by the corresponding Unicode code point

%d   base 10
%x   base 16, with lower-case letters for a-f
%f   decimal point but no exponent, e.g. 123.456
%t   the word true or false

%v   the value in a default format
     when printing structs, the plus flag (%+v) adds field names

%#v  a Go-syntax representation of the value
%T   a Go-syntax representation of the type of the value

%%   a literal percent sign; consumes no value [escape]
```

Read the godoc, Luke: https://golang.org/pkg/fmt/

Format code examples

%#v and %T are very useful for describing what something is:

```
a := []int{1, 2, 3}
b := [3]rune{'a', 'b', 'c'}
p := map[string]int{"and":1, "or":2}

fmt.Printf("%T\n", a)   // []int
fmt.Printf("%v\n", a)   // [1 2 3]
fmt.Printf("%#v\n", a)  // []int{1, 2, 3}

fmt.Printf("%T\n", b)   // [3]int32
fmt.Printf("%q\n", b)   // ['a' 'b' 'c']
fmt.Printf("%v\n", b)   // [97 98 99]
fmt.Printf("%#v\n", b)  // [3]int32{97, 98, 99}

fmt.Printf("%T\n", p)   // map[string]int
fmt.Printf("%v\n", p)   // map[and:1 or:2]
fmt.Printf("%#v\n", p)  // map[string]int{"and":1, "or":2}
```

Reading a file

Here's another program that checks a file's size

```go
package main

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

func main() {
    fname := os.Args[1]
    if f, err := os.Open(fname); err != nil {
        fmt.Fprintln(os.Stderr, "bad file:", err)
    } else if d, err := ioutil.ReadAll(f); err != nil {
        fmt.Fprintln(os.Stderr, "can't read:", err)
    } else {
        fmt.Printf("The file has %d bytes\n", len(d))
    }
}
```

If run on itself (the source file), it prints "The file has 333 bytes"

Reading a file

Wait, what's going on here?

```
if f, err := os.Open(fname); err != nil {
    fmt.Fprintln(os.Stderr, "bad file:", err)
} . . .
```

An if-statement can have a short declaration as its first part

We often call functions whose 2nd return value is a possible error

```
func Open(name string) (*File, error)
```

where the `error` can be compared to `nil`, meaning no error

Always check the error — the file might not really be open!

Homework

Homework # 1

Write a program to take a list of files from the command line and count the words in them. A word is anything separated by whitespace (so detached punctuation will count).

You can test your first program with `wc`:

```
$ wc testing.txt
       9      35     180 testing.txt
```

Then create an option to count only *unique* words.
You can test this with:

```
$ awk '{for (i=1;i<=NF;i++) {print $i}}' testing.txt|sort|uniq|wc
      26      26     144
```

You may want to use `ioutil.ReadFile` and `strings.Fields`