Go Training OReilly - Notes

```
package main

import "math"

func main() {
    math.Floor(3.1415)        // valid...
    math.Floor()              // not enough arguments
    math.Floor(3.1415, 12.34) // too many arguments
    math.Floor("a string")    // wrong type
}


---
./prog.go:7:12: not enough arguments in call to math.Floor
    have ()
    want (float64)
./prog.go:8:12: too many arguments in call to math.Floor
    have (number, number)
    want (float64)
./prog.go:9:13: cannot use "a string" (type untyped string) as type float64 in argument to math.Floor
```

```
package main

func main() {
    fmt.Println(math.Floor(2.75))
    fmt.Println(strings.Title("head first go"))
}

---

./prog.go:4:2: undefined: fmt
./prog.go:4:14: undefined: math
./prog.go:5:2: undefined: fmt
./prog.go:5:14: undefined: strings
```

```
This works

package main

import (
    "fmt"
    "math"
    "strings"
)

func main() {
    fmt.Println(math.Floor(2.75))
    fmt.Println(strings.Title("head first go"))
}

---
2
Head First Go

BUT this does not:

package main

import ("fmt" "math" "strings")

func main() {
    fmt.Println(math.Floor(2.75))
    fmt.Println(strings.Title("head first go"))
}

---
prog.go:3:15: expected ';', found "math"

This does

package main

import ("fmt"; "math"; "strings")

func main() {
    fmt.Println(math.Floor(2.75))
    fmt.Println(strings.Title("head first go"))
}


```

```
Unused import is error

"goimports" - wrapper for go fmt + add/remove imports

 $ go get golang.org/x/tools/cmd/goimports


```

```
VARIABLES

var NAME type
=> strongly type

int
```

```
float64

reflect.TypeOf(varname)

// example

var myInteger int
myInteger = 1
var myFloat float64
myFloat = 3.1415
fmt.Println(myInteger) // => 1
fmt.Println(myFloat) // => 3.1415
fmt.Println(reflect.TypeOf(myInteger)) // => int
fmt.Println(reflect.TypeOf(myFloat)) // => float64

// short declare - derives the type
// := ... declare AND assign, guess type

myInteger := 1
myFloat := 3.1415
fmt.Println(myInteger) // => 1
fmt.Println(myFloat) // => 3.1415
fmt.Println(reflect.TypeOf(myInteger)) // => int
fmt.Println(reflect.TypeOf(myFloat)) // => float64

// must use every variable
// this is error

subtotal := 24.70
tax := 1.89
fmt.Println(subtotal)
---
Compile error:
prog.go:9:2: tax declared and not used


```

```
// NO IMPLICIT TYPE CONV

var length float64 = 1.2
var width int = 2
// Can't assign an `int` value
// to a `float64` variable:
length = width
fmt.Println(length)

Compile error:
cannot use width (type int) as type float64 in assignment

---

```

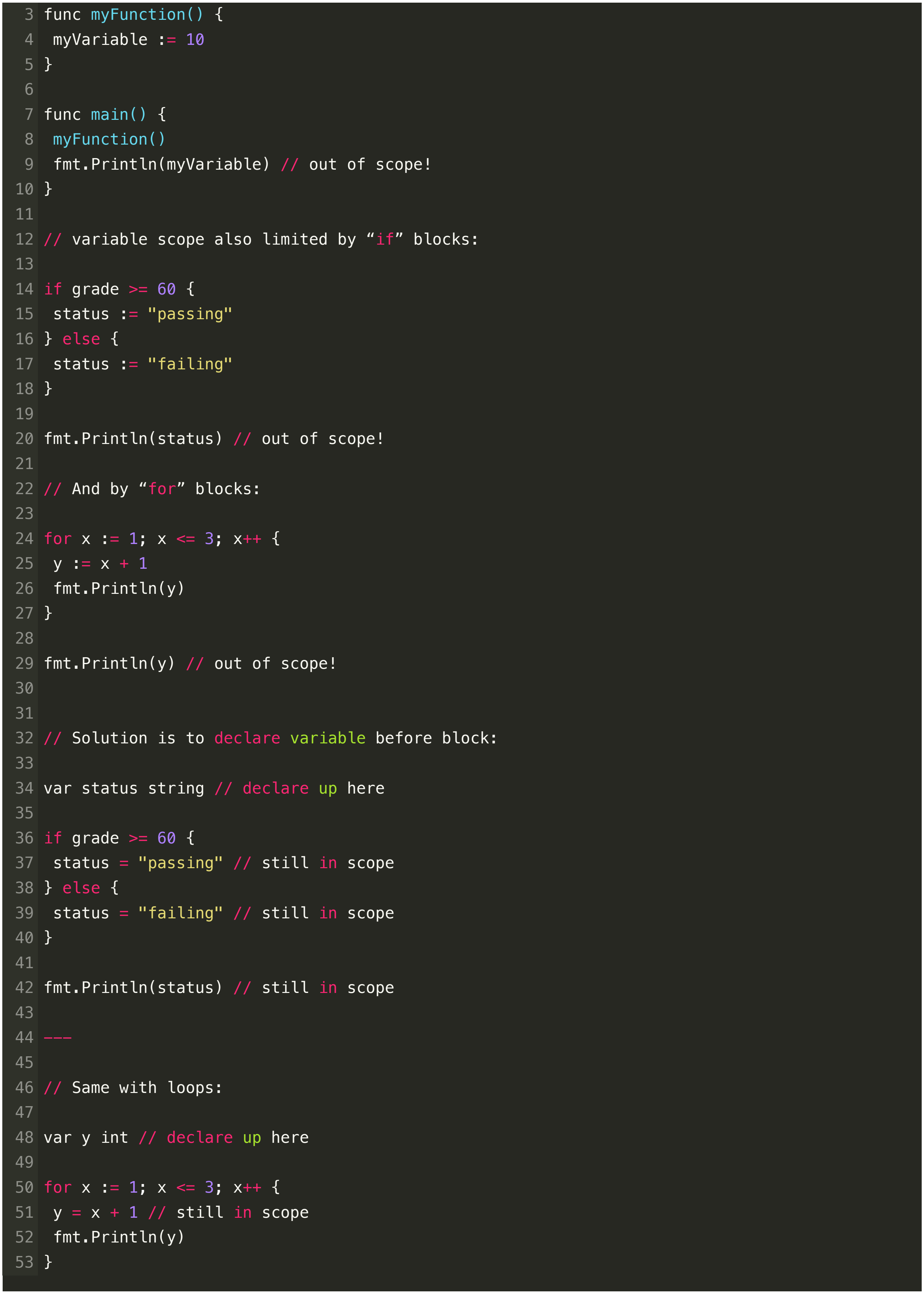
```
var length float64 = 1.2
var width int = 2
// But you can if you do a type
// conversion!
length = float64(width)
fmt.Println(length) // => 2


Same in MATH ops

package main

import (
    "fmt"
)

func main() {
    var length float64 = 1.2
    var width int = 2
    // Can't do a math operation with a float64 and an int:
    fmt.Println("Area is", length*width)
    // Or a comparison:
    fmt.Println("length > width?", length > width)
}

./prog.go:11:31: invalid operation: length * width (mismatched types float64 and int)
./prog.go:13:40: invalid operation: length > width (mismatched types float64 and int)

---
package main

import (
    "fmt"
)

func main() {
    var length float64 = 1.2
    var width int = 2
    // But you can if you do type conversions!
    fmt.Println("Area is", length*float64(width))
    fmt.Println("length > width?", length > float64(width))
}

---
Area is 2.4
length > width? false

```

```
IF

if 1 < 2 {
 fmt.Println("It's true!")
}

---
It's true!

parens around condition - dicouraged, fmt removes them

Opening curly brace must be on same line as if. This is a syntax error:

if (1 < 2)
{
 fmt.Println("It's true!")
}


```

```
QA

- can compile for any arch, distributes multiple binaries

```

```
Function names

• Use CamelCase: capitalize each word after the first.

• If the first letter of a function name is Capitalized, it’s considered exported: it can be used
from other packages.

• If the first letter of a function name is uncapitalized, it’s considered unexported: it can only
be used within its package.

• This is why all the names of standard library functions we’ve been calling are capitalized. (E.g.
fmt.Println, math.Floor, etc.)


Func Params

func say(phrase string, times int) {
    for i := 0; i < times; i++ {
        fmt.Print(phrase)
    }
    fmt.Print("\n")
}


```

```
Variable scope limited to function where it’s declared.

```

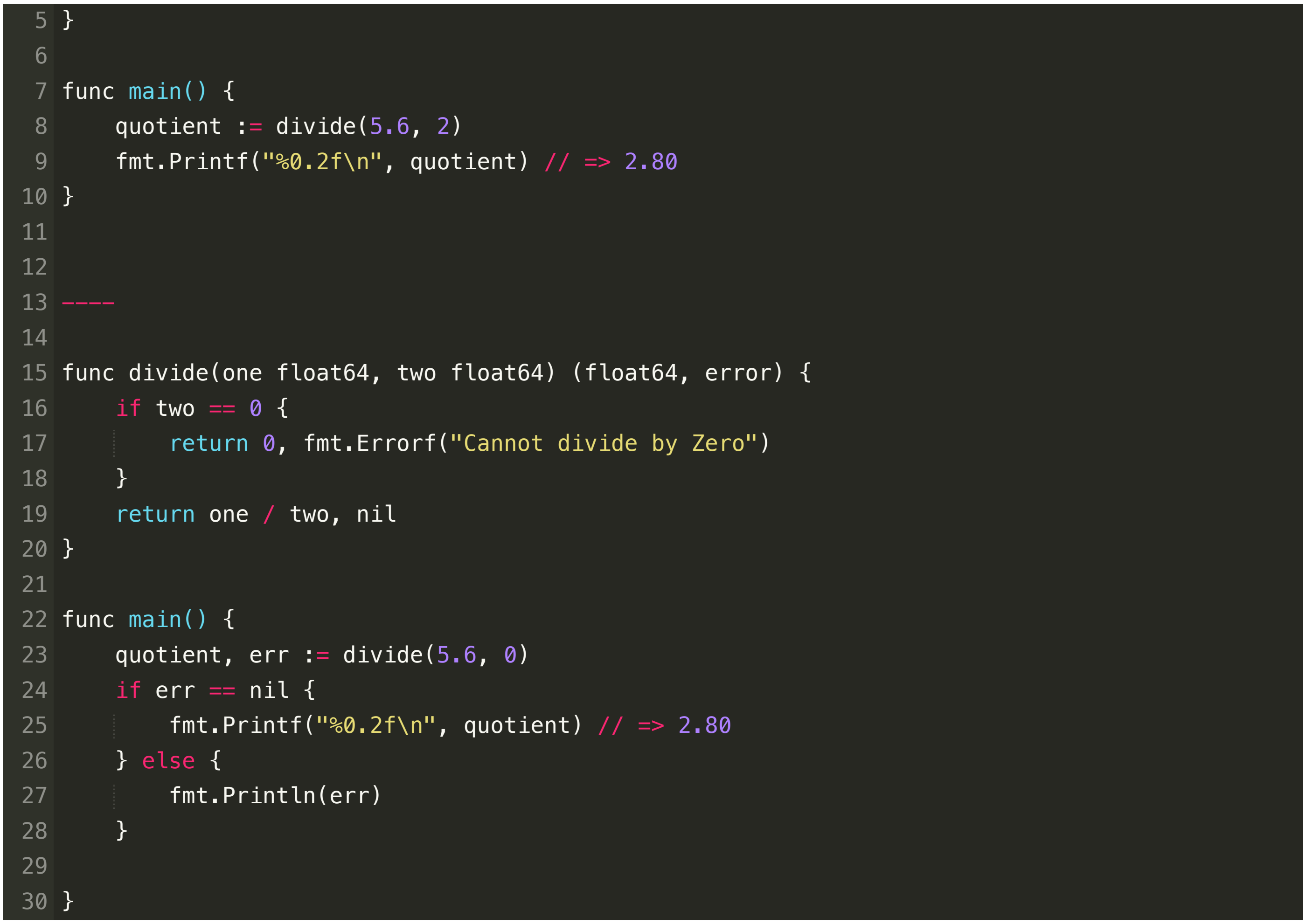


```
func myFunction() {
 myVariable := 10
}

func main() {
 myFunction()
 fmt.Println(myVariable) // out of scope!
}

// variable scope also limited by “if” blocks:

if grade >= 60 {
 status := "passing"
} else {
 status := "failing"
}

fmt.Println(status) // out of scope!

// And by “for” blocks:

for x := 1; x <= 3; x++ {
 y := x + 1
 fmt.Println(y)
}

fmt.Println(y) // out of scope!


// Solution is to declare variable before block:

var status string // declare up here

if grade >= 60 {
 status = "passing" // still in scope
} else {
 status = "failing" // still in scope
}

fmt.Println(status) // still in scope

---

// Same with loops:

var y int // declare up here

for x := 1; x <= 3; x++ {
 y = x + 1 // still in scope
 fmt.Println(y)
}
```

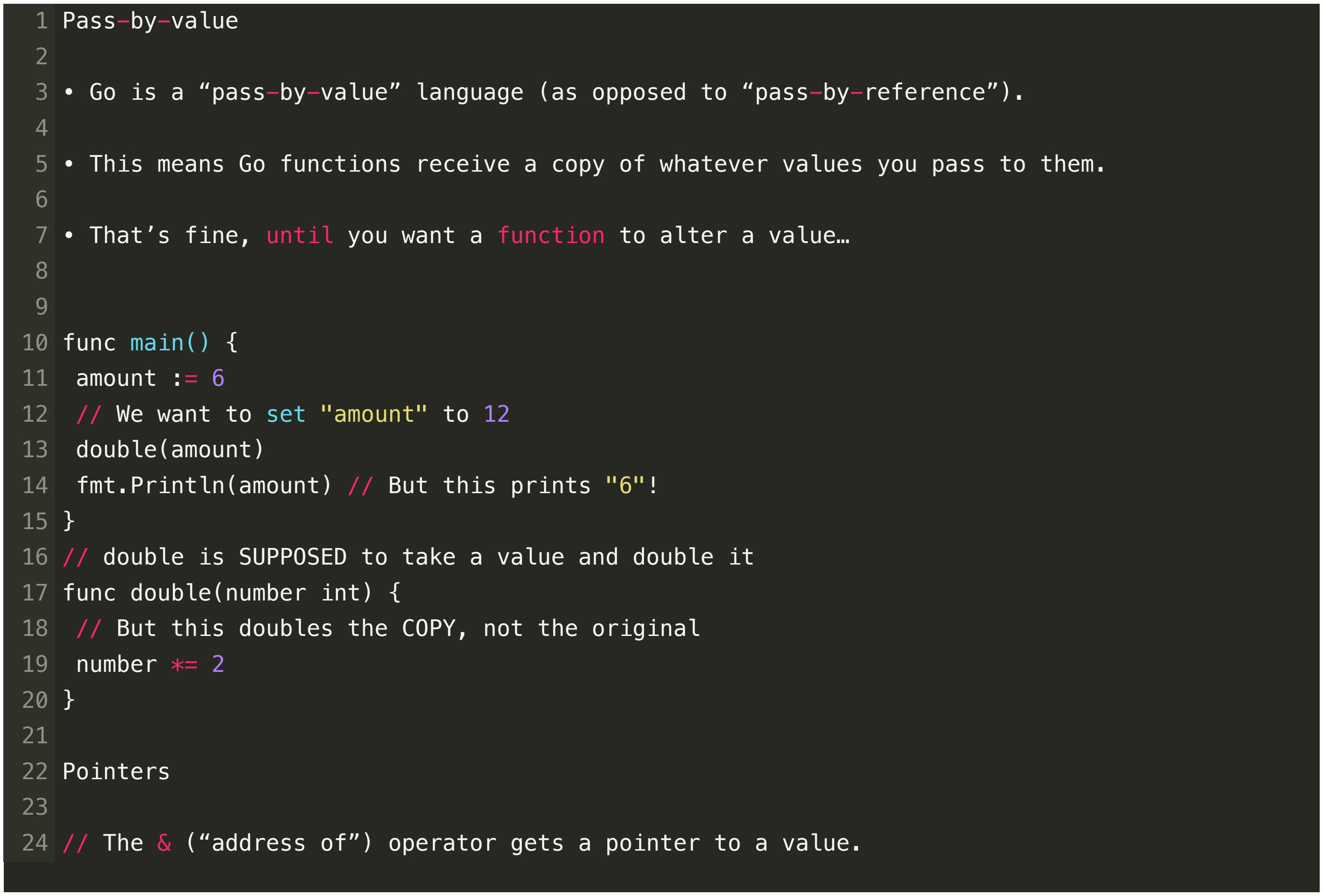
```
fmt.Println(y) // still in scope

---

RETURN

// Add return value type after parentheses
func myFunction() int {
 // Use "return" keyword
 return 10
}

func main() {
 // Assign returned value to variable
 myVariable := myFunction()
 fmt.Println(myVariable) // => 10
}
```

```
Multiple values

func main() {
 flag := strconv.ParseBool("true")
 flag = strconv.ParseBool("foobar")
 fmt.Println(flag)
}

Compile error:
prog.go:9:7: assignment mismatch: 1 variable but strconv.ParseBool returns 2 values
prog.go:10:7: assignment mismatch: 1 variable but strconv.ParseBool returns 2 values

need second vars


func main() {
 flag, err := strconv.ParseBool("true")
 if err != nil {
 log.Fatal(err)
 }
 fmt.Println(flag) // => true
 flag, err = strconv.ParseBool("foobar")
 if err != nil {
 log.Fatal(err)
 // => 2009/11/10 23:00:00 strconv.ParseBool:
 // => parsing "foobar": invalid syntax
 }
 fmt.Println(flag)
}
```

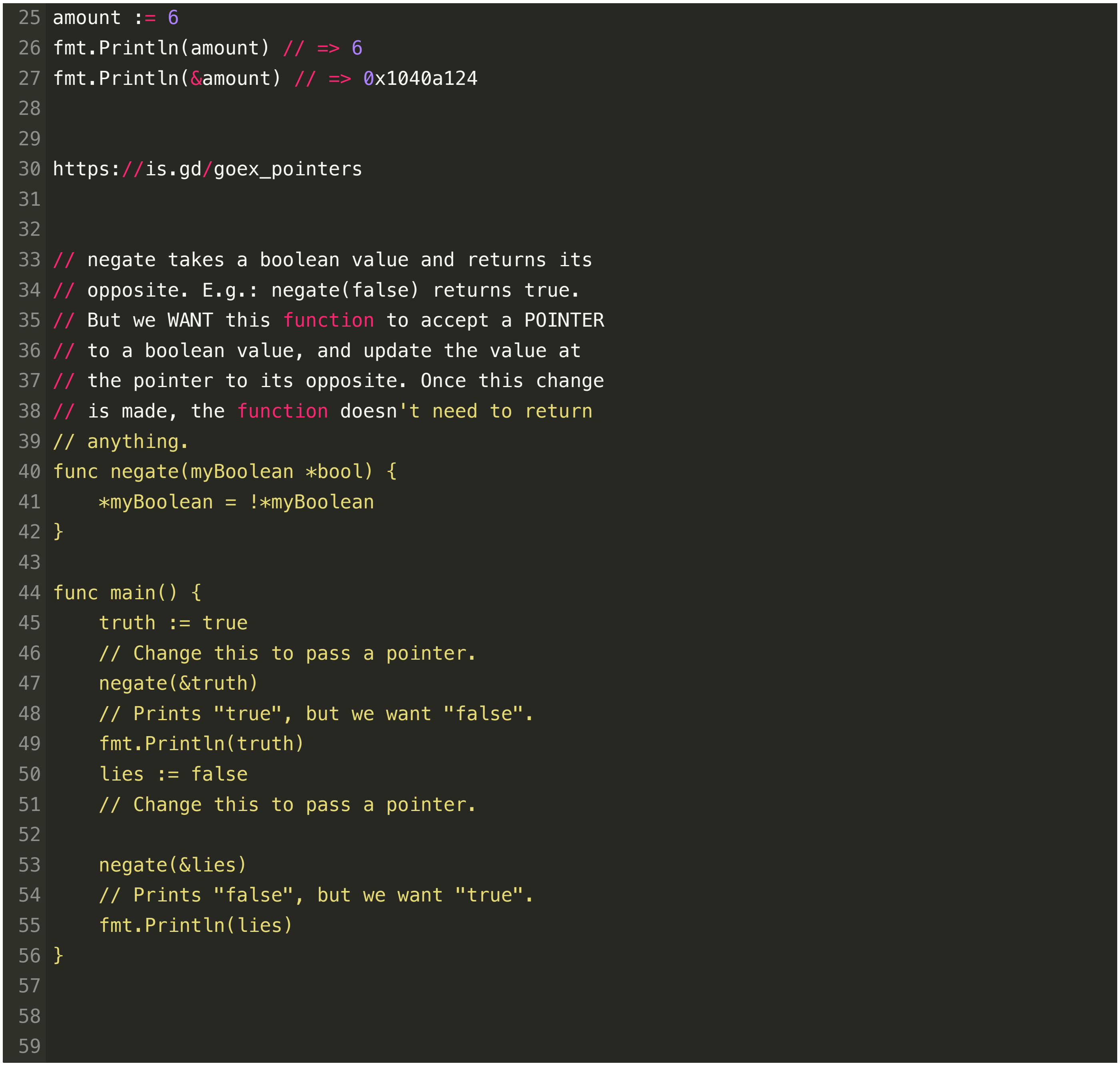
```
EXERC 2

func divide(one float64, two float64) float64 {
    return one / two
```



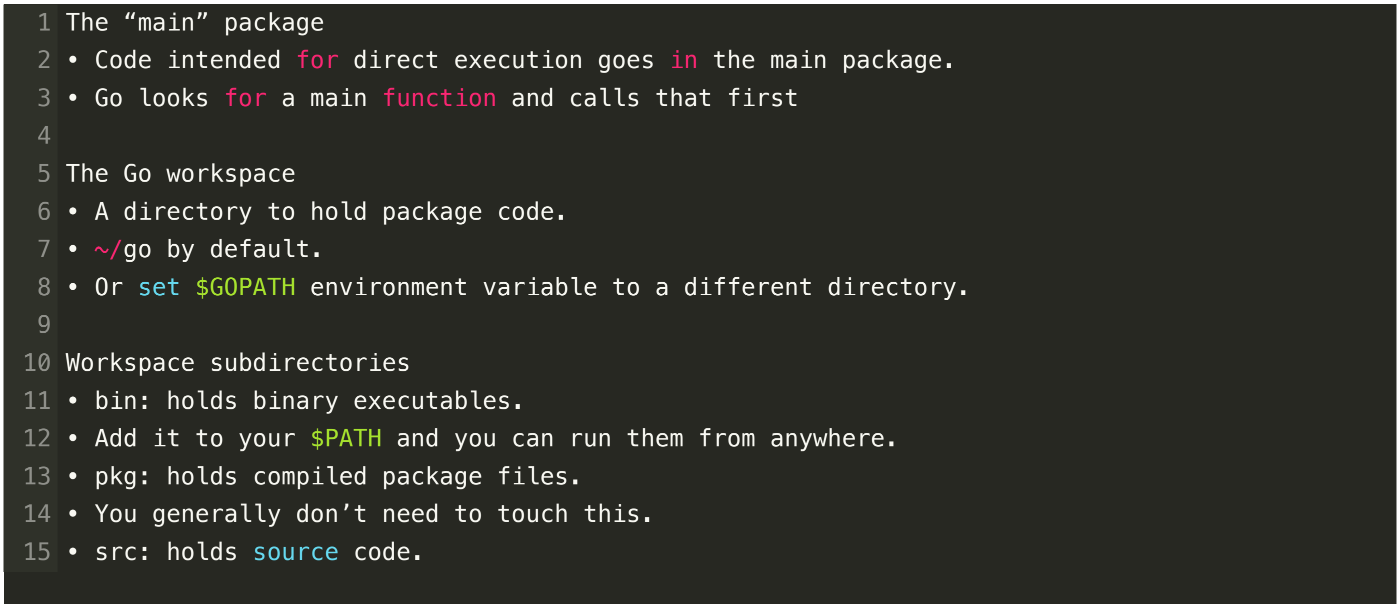
```
}

func main() {
    quotient := divide(5.6, 2)
    fmt.Printf("%0.2f\n", quotient) // => 2.80
}


----

func divide(one float64, two float64) (float64, error) {
    if two == 0 {
        return 0, fmt.Errorf("Cannot divide by Zero")
    }
    return one / two, nil
}

func main() {
    quotient, err := divide(5.6, 0)
    if err == nil {
        fmt.Printf("%0.2f\n", quotient) // => 2.80
    } else {
        fmt.Println(err)
    }

}
```



```
Pass-by-value

• Go is a “pass-by-value” language (as opposed to “pass-by-reference”).

• This means Go functions receive a copy of whatever values you pass to them.

• That’s fine, until you want a function to alter a value…


func main() {
 amount := 6
 // We want to set "amount" to 12
 double(amount)
 fmt.Println(amount) // But this prints "6"!
}
// double is SUPPOSED to take a value and double it
func double(number int) {
 // But this doubles the COPY, not the original
 number *= 2
}

Pointers

// The & (“address of”) operator gets a pointer to a value.
```



```
amount := 6
fmt.Println(amount) // => 6
fmt.Println(&amount) // => 0x1040a124


https://is.gd/goex_pointers


// negate takes a boolean value and returns its
// opposite. E.g.: negate(false) returns true.
// But we WANT this function to accept a POINTER
// to a boolean value, and update the value at
// the pointer to its opposite. Once this change
// is made, the function doesn't need to return
// anything.
func negate(myBoolean *bool) {
    *myBoolean = !*myBoolean
}

func main() {
    truth := true
    // Change this to pass a pointer.
    negate(&truth)
    // Prints "true", but we want "false".
    fmt.Println(truth)
    lies := false
    // Change this to pass a pointer.

    negate(&lies)
    // Prints "false", but we want "true".
    fmt.Println(lies)
}
```



```
The “main” package
• Code intended for direct execution goes in the main package.
• Go looks for a main function and calls that first

The Go workspace
• A directory to hold package code.
• ~/go by default.
• Or set $GOPATH environment variable to a different directory.

Workspace subdirectories
• bin: holds binary executables.
• Add it to your $PATH and you can run them from anywhere.
• pkg: holds compiled package files.
• You generally don’t need to touch this.
• src: holds source code.
• Including your code!
```

```

Unexported

So why would you ever make functions unexported?

• Unexported methods are Go’s equivalent to Java’s private methods.
• Use for helper functions that other packages shouldn’t call.
• Once you export a function, you shouldn’t change it any more.
• You can change how it works internally…
• But you shouldn’t change its parameters, return value, etc.
• If you do, you risk breaking others’ code!
• But you can change unexported functions all you want!

i18n

Alt-Shift-? == ¿
Alt-1 == ¡

Notice the import paths are not the same as the package names!

• Package name is whatever is used in package clause in files: package dansk
• By convention, last segment of import path is used as package name.

Import Path <=> Package Name
greeting  <=> greeting
greeting/dansk  <=> dansk
greeting/deutsch  <=> deutsch


```

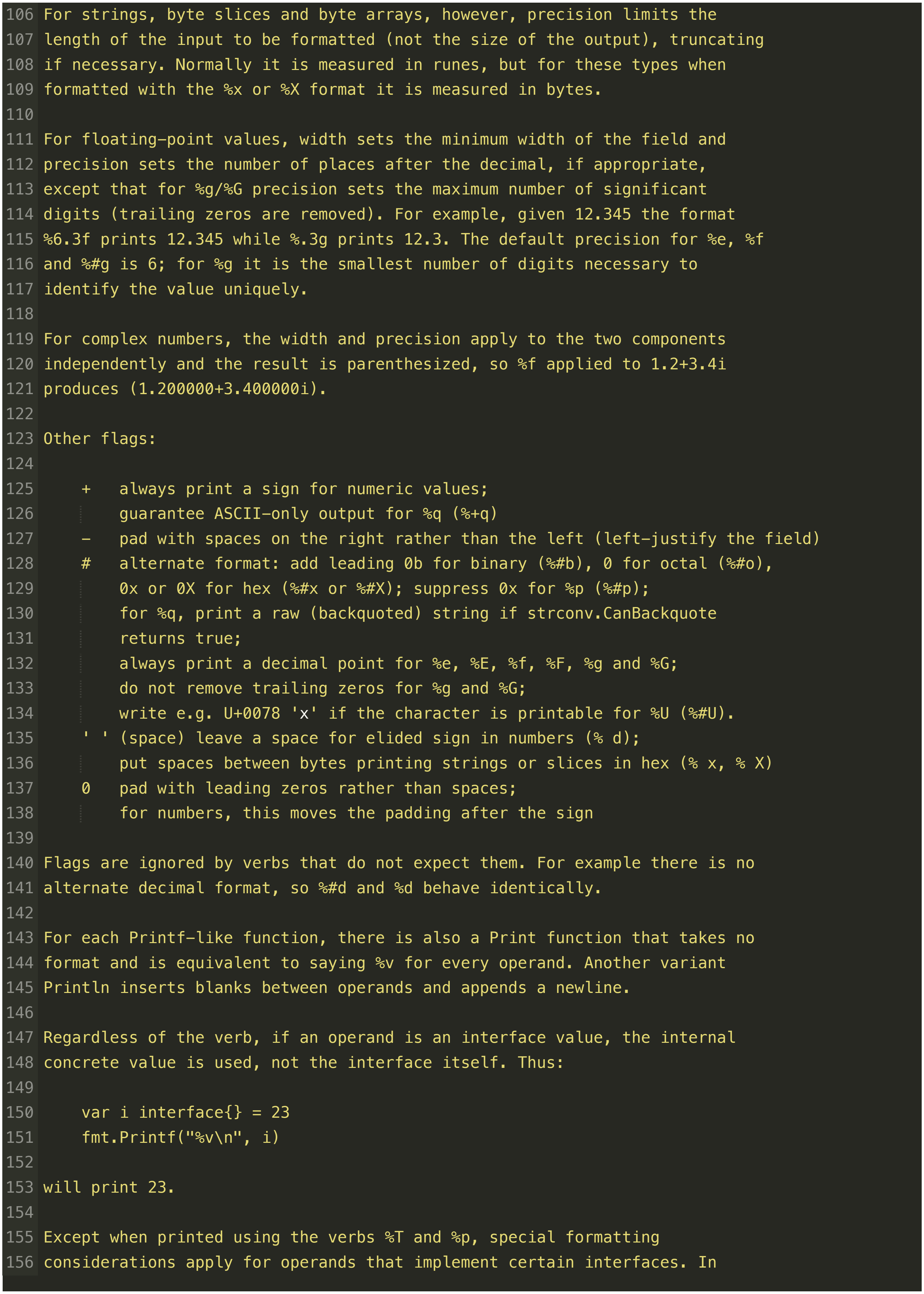
```
go get github.com/headfirstgo/greeting


```

```
Fist comment before package => doc

➜  s-oly-lt-go-introduction go doc github.com/headfirstgo/greeting
package greeting // import "github.com/headfirstgo/greeting"

Package greeting greets the user in English.

func Hello()
func Hi()



```

```
➜  s-oly-lt-go-introduction go doc fmt
package fmt // import "fmt"

```

```
Package fmt implements formatted I/O with functions analogous to C's printf
and scanf. The format 'verbs' are derived from C's but are simpler.


Printing

The verbs:

General:

	%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

Boolean:

	%t	the word true or false

Integer:

	%b	base 2
	%c	the character represented by the corresponding Unicode code point
	%d	base 10
	%o	base 8
	%O	base 8 with 0o prefix
	%q	a single-quoted character literal safely escaped with Go syntax.
	%x	base 16, with lower-case letters for a-f
	%X	base 16, with upper-case letters for A-F
	%U	Unicode format: U+1234; same as "U+%04X"

Floating-point and complex constituents:

	%b	decimalless scientific notation with exponent a power of two,
		in the manner of strconv.FormatFloat with the 'b' format,
		e.g. -123456p-78
	%e	scientific notation, e.g. -1.234456e+78
	%E	scientific notation, e.g. -1.234456E+78
	%f	decimal point but no exponent, e.g. 123.456
	%F	synonym for %f
	%g	%e for large exponents, %f otherwise. Precision is discussed below.
	%G	%E for large exponents, %F otherwise
	%x	hexadecimal notation (with decimal power of two exponent), e.g. -0x1.23abcp+20
	%X	upper-case hexadecimal notation, e.g. -0X1.23ABCP+20

String and slice of bytes (treated equivalently with these verbs):

	%s	the uninterpreted bytes of the string or slice
	%q	a double-quoted string safely escaped with Go syntax
	%x	base 16, lower-case, two characters per byte
```

```
    %X  base 16, upper-case, two characters per byte

Slice:

    %p  address of 0th element in base 16 notation, with leading 0x

Pointer:

    %p  base 16 notation, with leading 0x
    The %b, %d, %o, %x and %X verbs also work with pointers,
    formatting the value exactly as if it were an integer.

The default format for %v is:

    bool:                    %t
    int, int8 etc.:          %d
    uint, uint8 etc.:        %d, %#x if printed with %#v
    float32, complex64, etc: %g
    string:                  %s
    chan:                    %p
    pointer:                 %p

For compound objects, the elements are printed using these rules,
recursively, laid out like this:

    struct:             {field0 field1 ...}
    array, slice:       [elem0 elem1 ...]
    maps:               map[key1:value1 key2:value2 ...]
    pointer to above:   &{}, &[], &map[]

Width is specified by an optional decimal number immediately preceding the
verb. If absent, the width is whatever is necessary to represent the value.
Precision is specified after the (optional) width by a period followed by a
decimal number. If no period is present, a default precision is used. A
period with no following number specifies a precision of zero. Examples:

    %f     default width, default precision
    %9f    width 9, default precision
    %.2f   default width, precision 2
    %9.2f  width 9, precision 2
    %9.f   width 9, precision 0

Width and precision are measured in units of Unicode code points, that is,
runes. (This differs from C's printf where the units are always measured in
bytes.) Either or both of the flags may be replaced with the character '*',
causing their values to be obtained from the next operand (preceding the one
to format), which must be of type int.

For most values, width is the minimum number of runes to output, padding the
formatted form with spaces if necessary.

```



```
For strings, byte slices and byte arrays, however, precision limits the
length of the input to be formatted (not the size of the output), truncating
if necessary. Normally it is measured in runes, but for these types when
formatted with the %x or %X format it is measured in bytes.

For floating-point values, width sets the minimum width of the field and
precision sets the number of places after the decimal, if appropriate,
except that for %g/%G precision sets the maximum number of significant
digits (trailing zeros are removed). For example, given 12.345 the format
%6.3f prints 12.345 while %.3g prints 12.3. The default precision for %e, %f
and %#g is 6; for %g it is the smallest number of digits necessary to
identify the value uniquely.

For complex numbers, the width and precision apply to the two components
independently and the result is parenthesized, so %f applied to 1.2+3.4i
produces (1.200000+3.400000i).

Other flags:

    +   always print a sign for numeric values;
        guarantee ASCII-only output for %q (%+q)
    -   pad with spaces on the right rather than the left (left-justify the field)
    #   alternate format: add leading 0b for binary (%#b), 0 for octal (%#o),
        0x or 0X for hex (%#x or %#X); suppress 0x for %p (%#p);
        for %q, print a raw (backquoted) string if strconv.CanBackquote
        returns true;
        always print a decimal point for %e, %E, %f, %F, %g and %G;
        do not remove trailing zeros for %g and %G;
        write e.g. U+0078 'x' if the character is printable for %U (%#U).
    ' ' (space) leave a space for elided sign in numbers (% d);
        put spaces between bytes printing strings or slices in hex (% x, % X)
    0   pad with leading zeros rather than spaces;
        for numbers, this moves the padding after the sign

Flags are ignored by verbs that do not expect them. For example there is no
alternate decimal format, so %#d and %d behave identically.

For each Printf-like function, there is also a Print function that takes no
format and is equivalent to saying %v for every operand. Another variant
Println inserts blanks between operands and appends a newline.

Regardless of the verb, if an operand is an interface value, the internal
concrete value is used, not the interface itself. Thus:

    var i interface{} = 23
    fmt.Printf("%v\n", i)

will print 23.

Except when printed using the verbs %T and %p, special formatting
considerations apply for operands that implement certain interfaces. In
```

```
order of application:

1. If the operand is a reflect.Value, the operand is replaced by the
concrete value that it holds, and printing continues with the next rule.

2. If an operand implements the Formatter interface, it will be invoked.
Formatter provides fine control of formatting.

3. If the %v verb is used with the # flag (%#v) and the operand implements
the GoStringer interface, that will be invoked.

If the format (which is implicitly %v for Println etc.) is valid for a
string (%s %q %v %x %X), the following two rules apply:

4. If an operand implements the error interface, the Error method will be
invoked to convert the object to a string, which will then be formatted as
required by the verb (if any).

5. If an operand implements method String() string, that method will be
invoked to convert the object to a string, which will then be formatted as
required by the verb (if any).

For compound operands such as slices and structs, the format applies to the
elements of each operand, recursively, not to the operand as a whole. Thus
%q will quote each element of a slice of strings, and %6.2f will control
formatting for each element of a floating-point array.

However, when printing a byte slice with a string-like verb (%s %q %x %X),
it is treated identically to a string, as a single item.

To avoid recursion in cases such as

    type X string
    func (x X) String() string { return Sprintf("<%s>", x) }

convert the value before recurring:

    func (x X) String() string { return Sprintf("<%s>", string(x)) }

Infinite recursion can also be triggered by self-referential data
structures, such as a slice that contains itself as an element, if that type
has a String method. Such pathologies are rare, however, and the package
does not protect against them.

When printing a struct, fmt cannot and therefore does not invoke formatting
methods such as Error or String on unexported fields.

Explicit argument indexes:

In Printf, Sprintf, and Fprintf, the default behavior is for each formatting
verb to format successive arguments passed in the call. However, the
```

notation [n] immediately before the verb indicates that the nth one-indexed argument is to be formatted instead. The same notation before a '*' for a width or precision selects the argument index holding the value. After processing a bracketed expression [n], subsequent verbs will use arguments n+1, n+2, etc. unless otherwise directed.

For example,

```
    fmt.Sprintf("%[2]d %[1]d\n", 11, 22)
```

will yield "22 11", while

```
    fmt.Sprintf("%[3]*.[2]*[1]f", 12.0, 2, 6)
```

equivalent to

```
    fmt.Sprintf("%6.2f", 12.0)
```

will yield " 12.00". Because an explicit index affects subsequent verbs, this notation can be used to print the same values multiple times by resetting the index for the first argument to be repeated:

```
    fmt.Sprintf("%d %d %#[1]x %#x", 16, 17)
```

will yield "16 17 0x10 0x11".

Format errors:

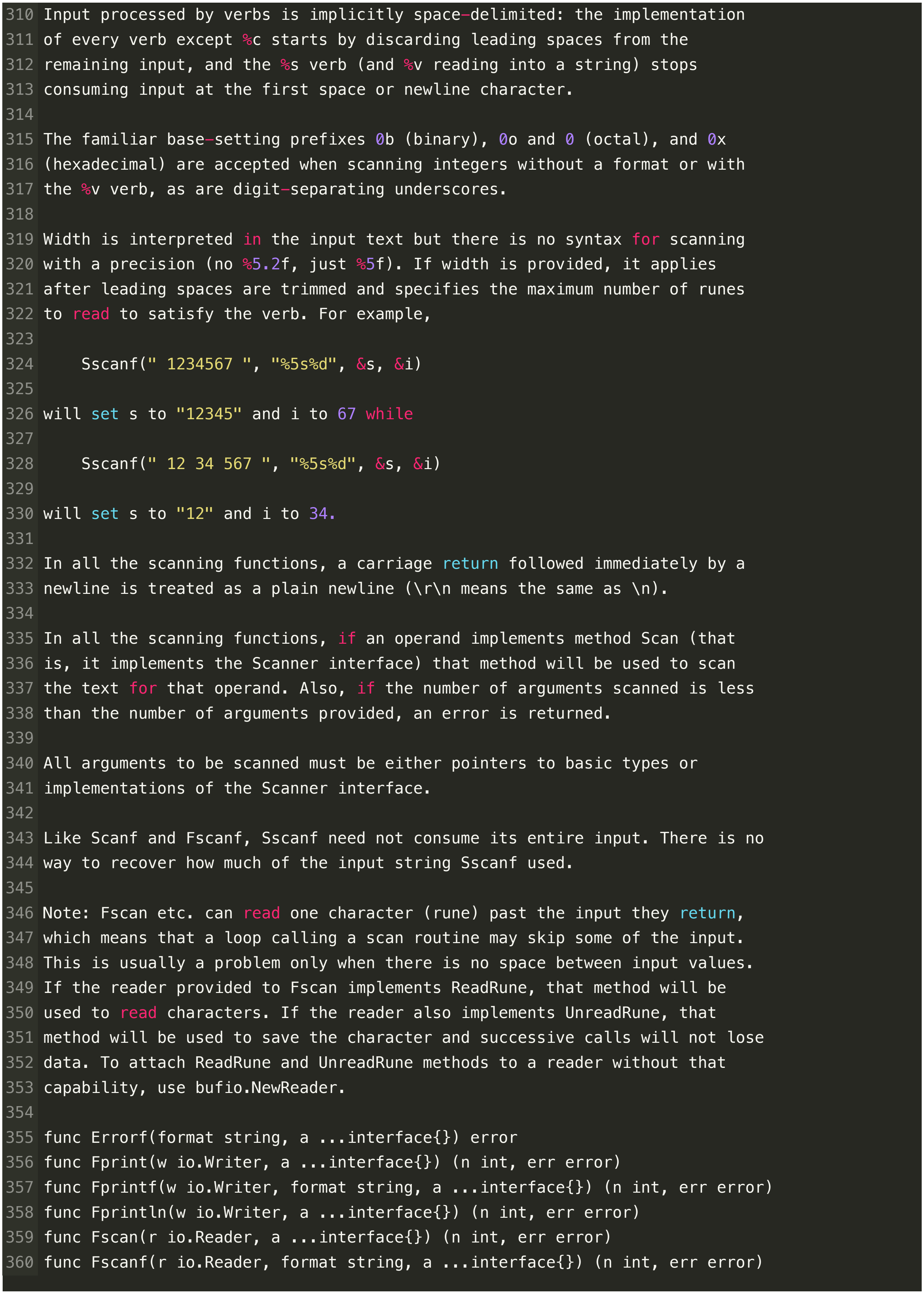
If an invalid argument is given for a verb, such as providing a string to %d, the generated string will contain a description of the problem, as in these examples:

```
    Wrong type or unknown verb: %!verb(type=value)
        Printf("%d", "hi"):        %!d(string=hi)
    Too many arguments: %!(EXTRA type=value)
        Printf("hi", "guys"):      hi%!(EXTRA string=guys)
    Too few arguments: %!verb(MISSING)
        Printf("hi%d"):            hi%!d(MISSING)
    Non-int for width or precision: %!(BADWIDTH) or %!(BADPREC)
        Printf("%*s", 4.5, "hi"):  %!(BADWIDTH)hi
        Printf("%.*s", 4.5, "hi"): %!(BADPREC)hi
    Invalid or invalid use of argument index: %!(BADINDEX)
        Printf("%*[2]d", 7):       %!d(BADINDEX)
        Printf("%.[2]d", 7):       %!d(BADINDEX)
```

All errors begin with the string "%!" followed sometimes by a single character (the verb) and end with a parenthesized description.

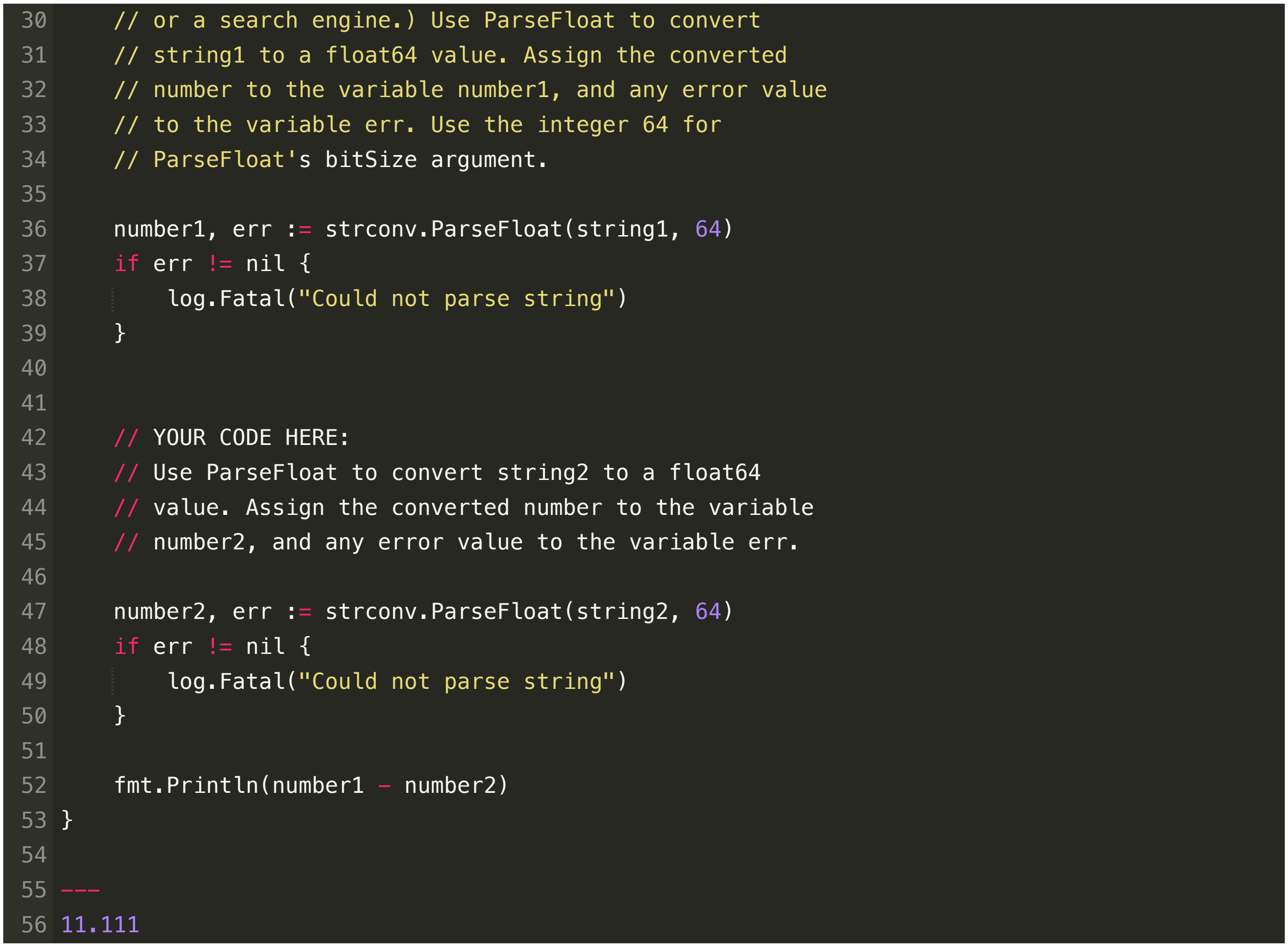
If an Error or String method triggers a panic when called by a print routine, the fmt package reformats the error message from the panic, decorating it with an indication that it came through the fmt package. For

```
example, if a String method calls panic("bad"), the resulting formatted
message will look like

    %!s(PANIC=bad)

The %!s just shows the print verb in use when the failure occurred. If the
panic is caused by a nil receiver to an Error or String method, however, the
output is the undecorated string, "<nil>".


Scanning

An analogous set of functions scans formatted text to yield values. Scan,
Scanf and Scanln read from os.Stdin; Fscan, Fscanf and Fscanln read from a
specified io.Reader; Sscan, Sscanf and Sscanln read from an argument string.

Scan, Fscan, Sscan treat newlines in the input as spaces.

Scanln, Fscanln and Sscanln stop scanning at a newline and require that the
items be followed by a newline or EOF.

Scanf, Fscanf, and Sscanf parse the arguments according to a format string,
analogous to that of Printf. In the text that follows, 'space' means any
Unicode whitespace character except newline.

In the format string, a verb introduced by the % character consumes and
parses input; these verbs are described in more detail below. A character
other than %, space, or newline in the format consumes exactly that input
character, which must be present. A newline with zero or more spaces before
it in the format string consumes zero or more spaces in the input followed
by a single newline or the end of the input. A space following a newline in
the format string consumes zero or more spaces in the input. Otherwise, any
run of one or more spaces in the format string consumes as many spaces as
possible in the input. Unless the run of spaces in the format string appears
adjacent to a newline, the run must consume at least one space from the
input or find the end of the input.

The handling of spaces and newlines differs from that of C's scanf family:
in C, newlines are treated as any other space, and it is never an error when
a run of spaces in the format string finds no spaces to consume in the
input.

The verbs behave analogously to those of Printf. For example, %x will scan
an integer as a hexadecimal number, and %v will scan the default
representation format for the value. The Printf verbs %p and %T and the
flags # and + are not implemented. For floating-point and complex values,
all valid formatting verbs (%b %e %E %f %F %g %G %x %X and %v) are
equivalent and accept both decimal and hexadecimal notation (for example:
"2.3e+7", "0x4.5p-8") and digit-separating underscores (for example:
"3.14159_26535_89793").

```

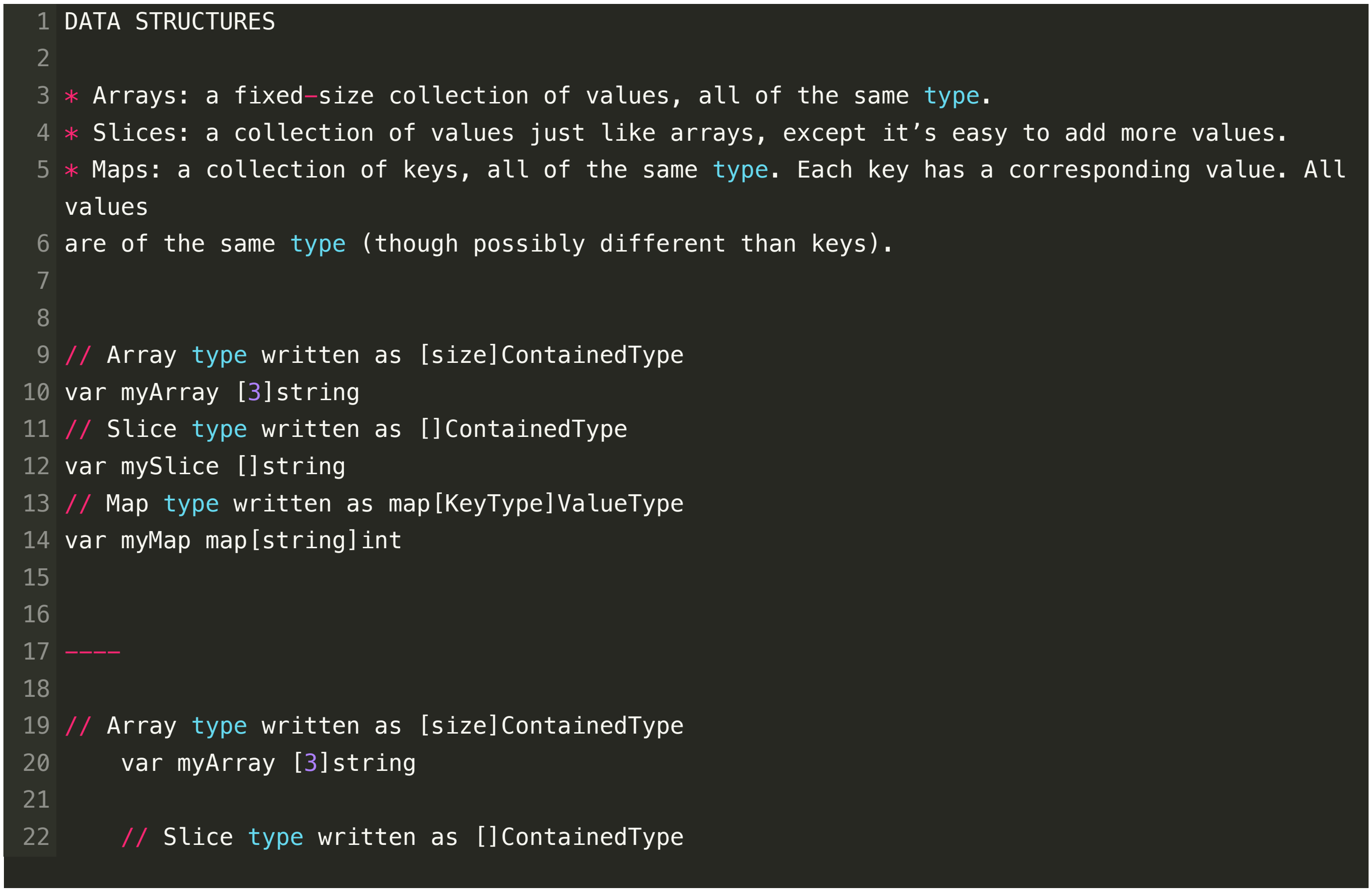


```
Input processed by verbs is implicitly space-delimited: the implementation
of every verb except %c starts by discarding leading spaces from the
remaining input, and the %s verb (and %v reading into a string) stops
consuming input at the first space or newline character.

The familiar base-setting prefixes 0b (binary), 0o and 0 (octal), and 0x
(hexadecimal) are accepted when scanning integers without a format or with
the %v verb, as are digit-separating underscores.

Width is interpreted in the input text but there is no syntax for scanning
with a precision (no %5.2f, just %5f). If width is provided, it applies
after leading spaces are trimmed and specifies the maximum number of runes
to read to satisfy the verb. For example,

    Sscanf(" 1234567 ", "%5s%d", &s, &i)

will set s to "12345" and i to 67 while

    Sscanf(" 12 34 567 ", "%5s%d", &s, &i)

will set s to "12" and i to 34.

In all the scanning functions, a carriage return followed immediately by a
newline is treated as a plain newline (\r\n means the same as \n).

In all the scanning functions, if an operand implements method Scan (that
is, it implements the Scanner interface) that method will be used to scan
the text for that operand. Also, if the number of arguments scanned is less
than the number of arguments provided, an error is returned.

All arguments to be scanned must be either pointers to basic types or
implementations of the Scanner interface.

Like Scanf and Fscanf, Sscanf need not consume its entire input. There is no
way to recover how much of the input string Sscanf used.

Note: Fscan etc. can read one character (rune) past the input they return,
which means that a loop calling a scan routine may skip some of the input.
This is usually a problem only when there is no space between input values.
If the reader provided to Fscan implements ReadRune, that method will be
used to read characters. If the reader also implements UnreadRune, that
method will be used to save the character and successive calls will not lose
data. To attach ReadRune and UnreadRune methods to a reader without that
capability, use bufio.NewReader.

func Errorf(format string, a ...interface{}) error
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)
func Print(a ...interface{}) (n int, err error)
func Printf(format string, a ...interface{}) (n int, err error)
func Println(a ...interface{}) (n int, err error)
func Scan(a ...interface{}) (n int, err error)
func Scanf(format string, a ...interface{}) (n int, err error)
func Scanln(a ...interface{}) (n int, err error)
func Sprint(a ...interface{}) string
func Sprintf(format string, a ...interface{}) string
func Sprintln(a ...interface{}) string
func Sscan(str string, a ...interface{}) (n int, err error)
func Sscanf(str string, format string, a ...interface{}) (n int, err error)
func Sscanln(str string, a ...interface{}) (n int, err error)
type Formatter interface{ ... }
type GoStringer interface{ ... }
type ScanState interface{ ... }
type Scanner interface{ ... }
type State interface{ ... }
type Stringer interface{ ... }
➜  s-oly-lt-go-introduction

```

```
Web based

go get -v  golang.org/x/tools/cmd/godoc


➜  s-oly-lt-go-introduction ./bin/godoc -http=:6060
using GOPATH mode


http://localhost:6060/pkg/

EX: https://play.golang.org/p/0IoS8oGzrnw

---
package main

import (
    "fmt"
    "log"
    "strconv"
)

func main() {
    string1 := "12.345"
    string2 := "1.234"

    // YOUR CODE HERE:
    // Look up documentation for the "strconv" package's
    // ParseFloat function. (You can use either "go doc"
```

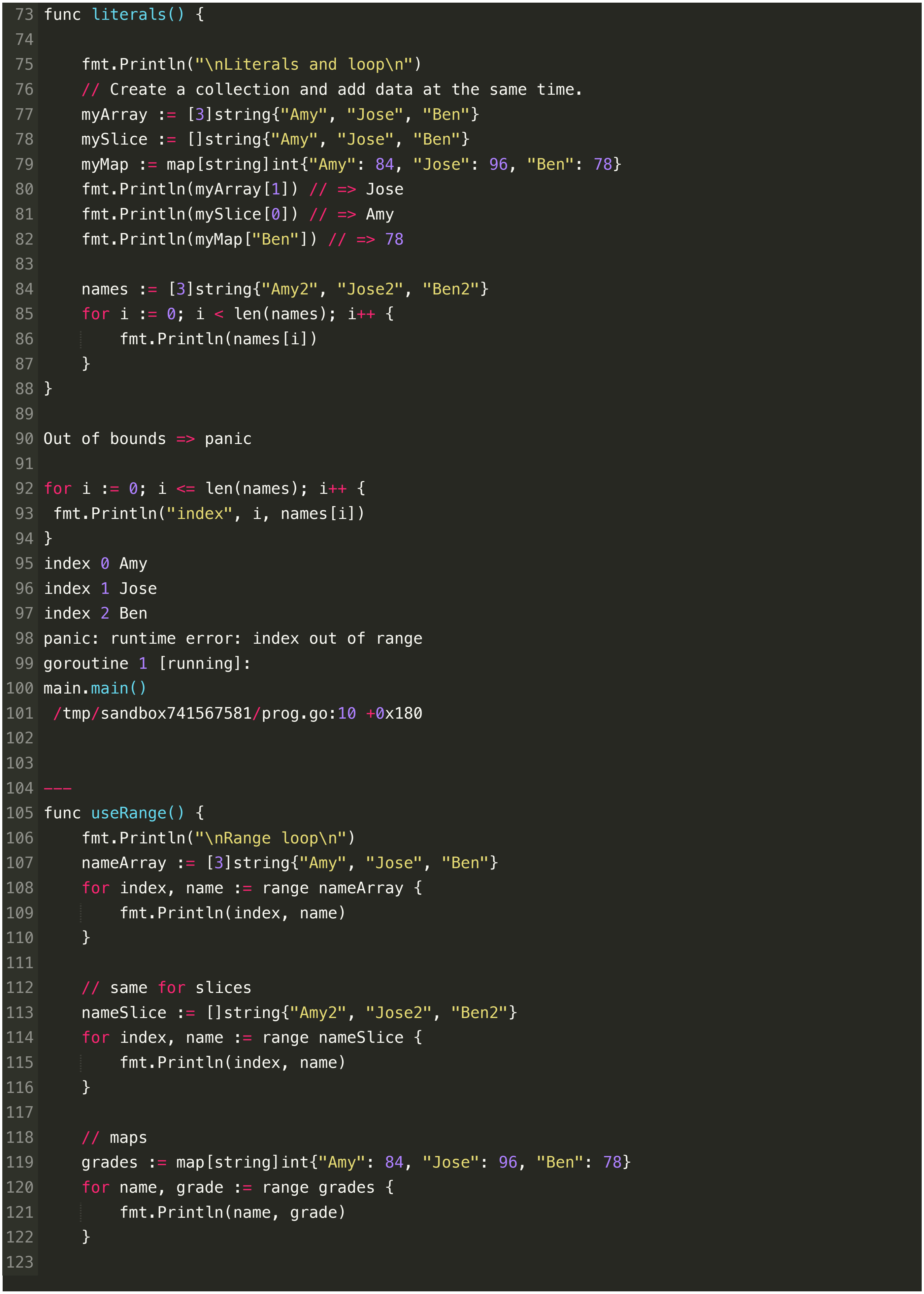


```
    // or a search engine.) Use ParseFloat to convert
    // string1 to a float64 value. Assign the converted
    // number to the variable number1, and any error value
    // to the variable err. Use the integer 64 for
    // ParseFloat's bitSize argument.

    number1, err := strconv.ParseFloat(string1, 64)
    if err != nil {
        log.Fatal("Could not parse string")
    }


    // YOUR CODE HERE:
    // Use ParseFloat to convert string2 to a float64
    // value. Assign the converted number to the variable
    // number2, and any error value to the variable err.

    number2, err := strconv.ParseFloat(string2, 64)
    if err != nil {
        log.Fatal("Could not parse string")
    }

    fmt.Println(number1 - number2)
}

---
11.111
```

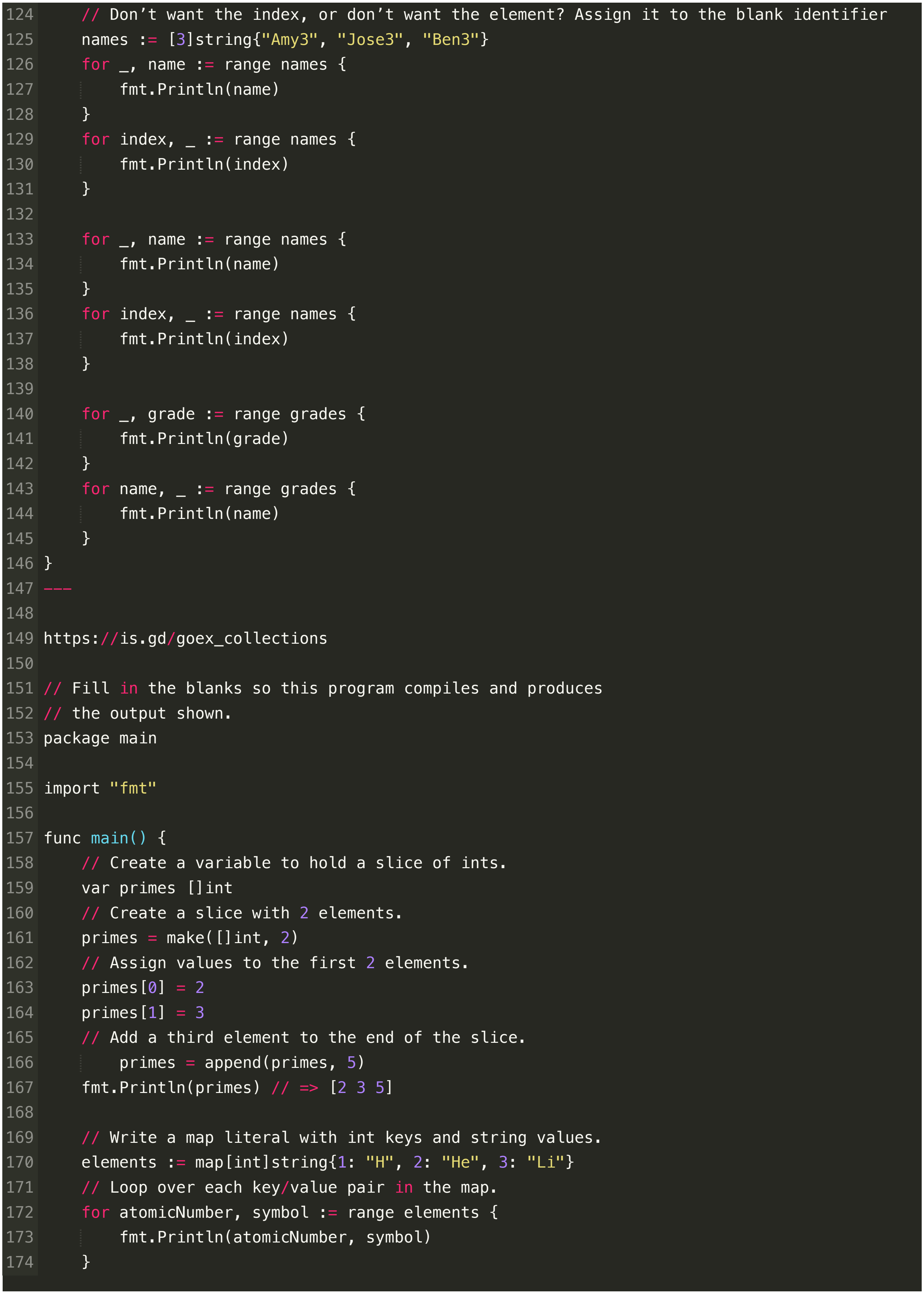


```
DATA STRUCTURES

* Arrays: a fixed-size collection of values, all of the same type.
* Slices: a collection of values just like arrays, except it’s easy to add more values.
* Maps: a collection of keys, all of the same type. Each key has a corresponding value. All values
are of the same type (though possibly different than keys).


// Array type written as [size]ContainedType
var myArray [3]string
// Slice type written as []ContainedType
var mySlice []string
// Map type written as map[KeyType]ValueType
var myMap map[string]int


----

// Array type written as [size]ContainedType
    var myArray [3]string

    // Slice type written as []ContainedType
```

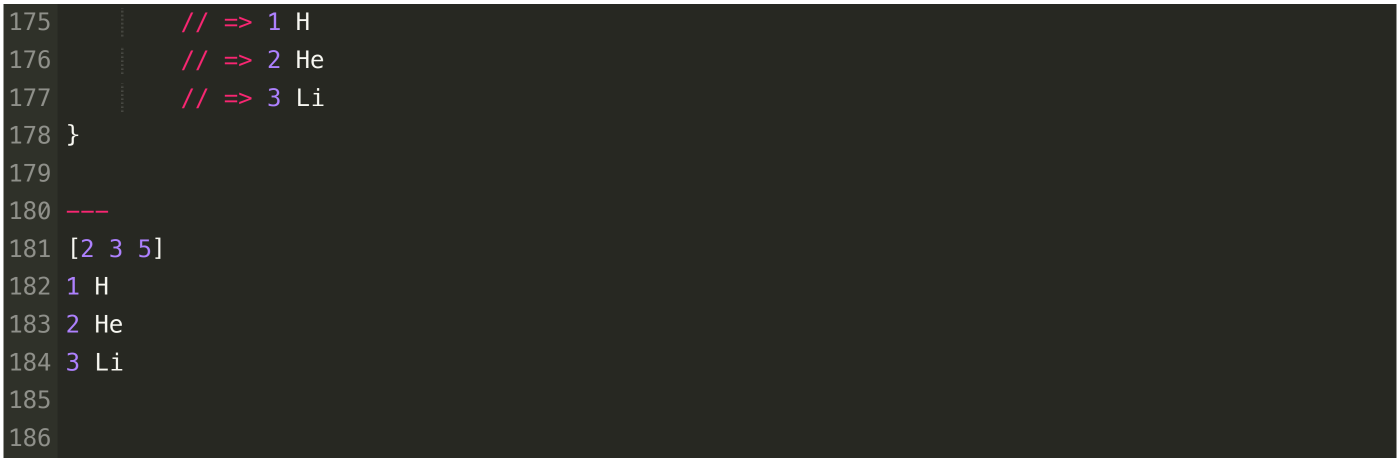
```go
    var mySlice []string
    mySlice = make([]string, 2)

    // Map type written as map[KeyType]ValueType
    var myMap map[string]int
    myMap = make(map[string]int)


    myArray[0] = "Amy"
    fmt.Println(myArray[0]) // => Amy

    mySlice[1] = "Jose"
    fmt.Println(mySlice[1]) // => Jose

    myMap["Ben"] = 78
    fmt.Println(myMap["Ben"]) // => 78


---
func shortDeclarations() {

    fmt.Println("\nShort declarations\n")

    var myArray [3]string
    mySlice := make([]string, 2)
    myMap := make(map[string]int)
    myArray[0] = "Amy2"
    fmt.Println(myArray[0]) // => Amy
    mySlice[1] = "Jose2"
    fmt.Println(mySlice[1]) // => Jose
    myMap["Ben"] = 79
    fmt.Println(myMap["Ben"]) // => 78
}


---
func expandingSlices() {
    fmt.Println("\nExpanding slices\n")
    primes := make([]int, 2)
    primes[0] = 2
    primes[1] = 3
    primes = append(primes, 5)
    primes = append(primes, 7)
    fmt.Println(primes) // => [2 3 5 7]
    // Want to do the same with an array? Have to throw it out and restart with a bigger one.
    //• In most cases you should use slices instead of arrays.
}

---

```



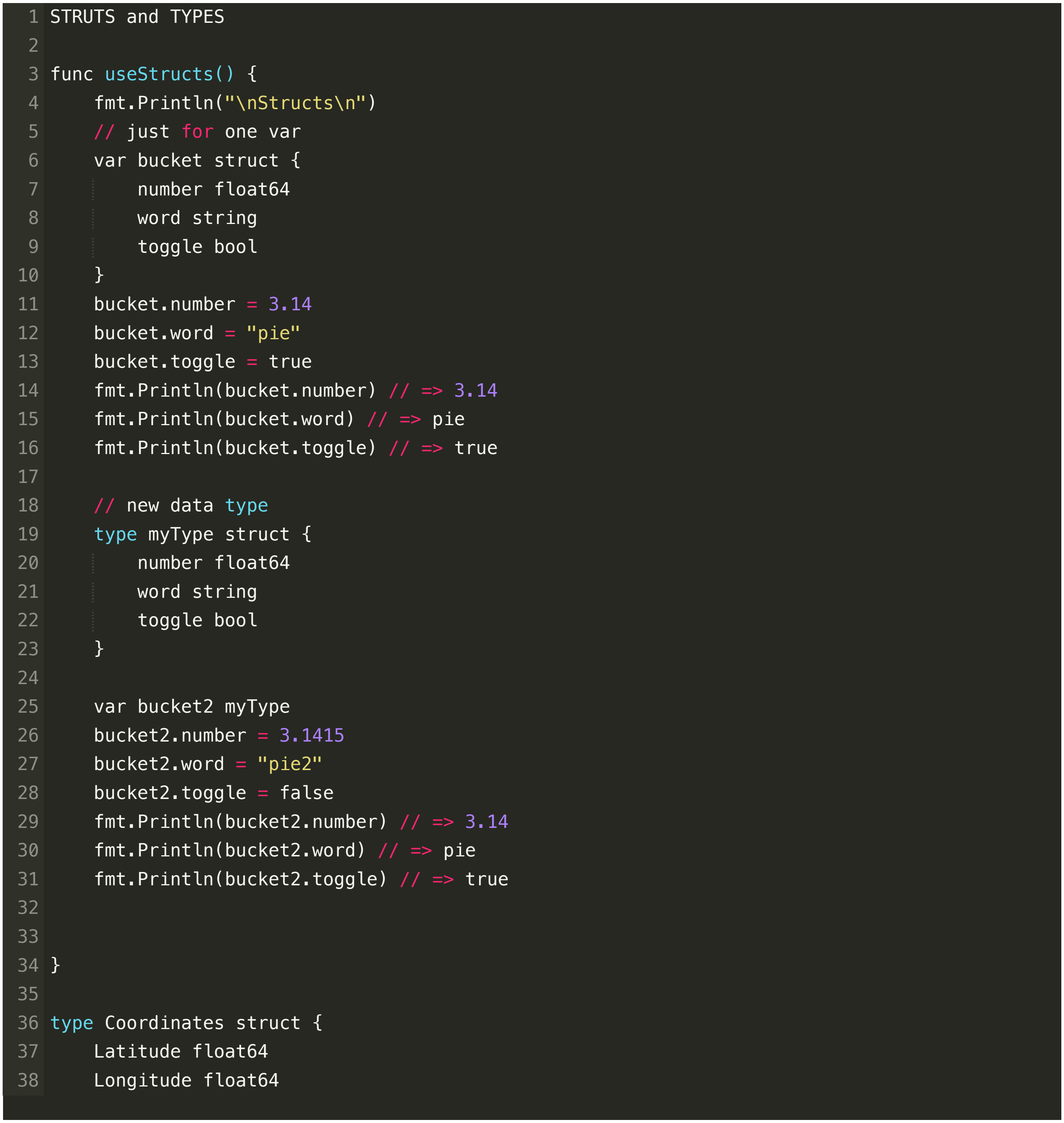
```
func literals() {

    fmt.Println("\nLiterals and loop\n")
    // Create a collection and add data at the same time.
    myArray := [3]string{"Amy", "Jose", "Ben"}
    mySlice := []string{"Amy", "Jose", "Ben"}
    myMap := map[string]int{"Amy": 84, "Jose": 96, "Ben": 78}
    fmt.Println(myArray[1]) // => Jose
    fmt.Println(mySlice[0]) // => Amy
    fmt.Println(myMap["Ben"]) // => 78

    names := [3]string{"Amy2", "Jose2", "Ben2"}
    for i := 0; i < len(names); i++ {
        fmt.Println(names[i])
    }
}

Out of bounds => panic

for i := 0; i <= len(names); i++ {
 fmt.Println("index", i, names[i])
}
index 0 Amy
index 1 Jose
index 2 Ben
panic: runtime error: index out of range
goroutine 1 [running]:
main.main()
 /tmp/sandbox741567581/prog.go:10 +0x180


---
func useRange() {
    fmt.Println("\nRange loop\n")
    nameArray := [3]string{"Amy", "Jose", "Ben"}
    for index, name := range nameArray {
        fmt.Println(index, name)
    }

    // same for slices
    nameSlice := []string{"Amy2", "Jose2", "Ben2"}
    for index, name := range nameSlice {
        fmt.Println(index, name)
    }

    // maps
    grades := map[string]int{"Amy": 84, "Jose": 96, "Ben": 78}
    for name, grade := range grades {
        fmt.Println(name, grade)
    }

```



```
    // Don’t want the index, or don’t want the element? Assign it to the blank identifier
    names := [3]string{"Amy3", "Jose3", "Ben3"}
    for _, name := range names {
        fmt.Println(name)
    }
    for index, _ := range names {
        fmt.Println(index)
    }

    for _, name := range names {
        fmt.Println(name)
    }
    for index, _ := range names {
        fmt.Println(index)
    }

    for _, grade := range grades {
        fmt.Println(grade)
    }
    for name, _ := range grades {
        fmt.Println(name)
    }
}
---

https://is.gd/goex_collections

// Fill in the blanks so this program compiles and produces
// the output shown.
package main

import "fmt"

func main() {
    // Create a variable to hold a slice of ints.
    var primes []int
    // Create a slice with 2 elements.
    primes = make([]int, 2)
    // Assign values to the first 2 elements.
    primes[0] = 2
    primes[1] = 3
    // Add a third element to the end of the slice.
        primes = append(primes, 5)
    fmt.Println(primes) // => [2 3 5]

    // Write a map literal with int keys and string values.
    elements := map[int]string{1: "H", 2: "He", 3: "Li"}
    // Loop over each key/value pair in the map.
    for atomicNumber, symbol := range elements {
        fmt.Println(atomicNumber, symbol)
    }
```



```
        // => 1 H
        // => 2 He
        // => 3 Li
}

---
[2 3 5]
1 H
2 He
3 Li


```

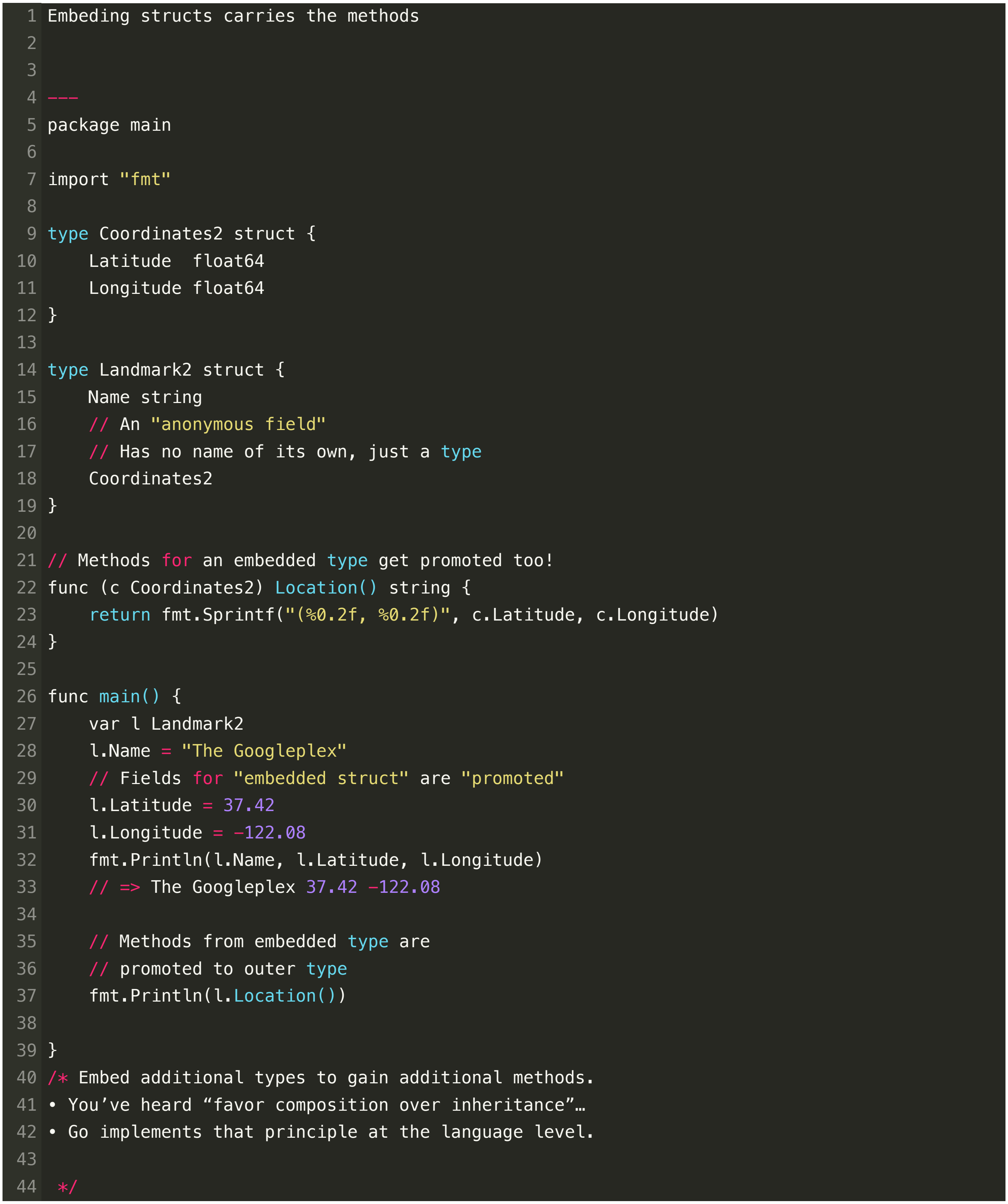


```
STRUTS and TYPES

func useStructs() {
    fmt.Println("\nStructs\n")
    // just for one var
    var bucket struct {
        number float64
        word string
        toggle bool
    }
    bucket.number = 3.14
    bucket.word = "pie"
    bucket.toggle = true
    fmt.Println(bucket.number) // => 3.14
    fmt.Println(bucket.word) // => pie
    fmt.Println(bucket.toggle) // => true

    // new data type
    type myType struct {
        number float64
        word string
        toggle bool
    }

    var bucket2 myType
    bucket2.number = 3.1415
    bucket2.word = "pie2"
    bucket2.toggle = false
    fmt.Println(bucket2.number) // => 3.14
    fmt.Println(bucket2.word) // => pie
    fmt.Println(bucket2.toggle) // => true


}

type Coordinates struct {
    Latitude float64
    Longitude float64
}
type Landmark struct {
    Name string
    // An "anonymous field"
    // Has no name of its own, just a type
    Coordinates
}

func useStructs2() {
    var l Landmark
    l.Name = "The Googleplex"
    // Fields for "embedded struct" are "promoted"
    l.Latitude = 37.42
    l.Longitude = -122.08
    fmt.Println(l.Name, l.Latitude, l.Longitude)
}

```

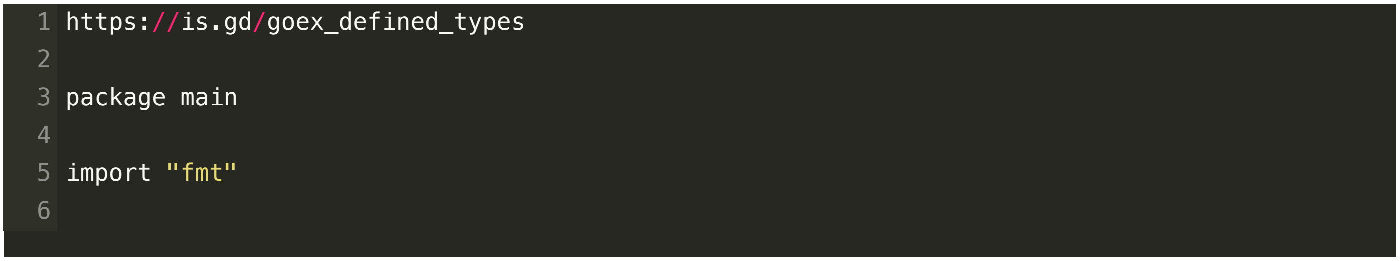
```
https://is.gd/goex_structs

package main

import (
    "fmt"
)

type Subscriber struct {
    Name   string
    Rate   float64
    Active bool
    Address
}

type Employee struct {
    Name   string
    Salary float64
    Address
}

type Address struct {
    Street  string
    City    string
    State   string
    PostalCode string
}

// YOUR CODE HERE:
// Define a struct type named Address that has Street, City, State,
// and PostalCode fields, each with a type of "string".
// Then embed the Address type within the Subscriber and Employee
// types using anonymous fields, so that the code in "main" will
// compile, run, and produce the output shown.

func main() {
    var subscriber Subscriber
    subscriber.Name = "Aman Singh"
    subscriber.Street = "123 Oak St"
    subscriber.City = "Omaha"
    subscriber.State = "NE"
    subscriber.PostalCode = "68111"
    fmt.Println("Name:", subscriber.Name)             // => Name: Aman Singh
    fmt.Println("Street:", subscriber.Street)         // => Street: 123 Oak St
    fmt.Println("City:", subscriber.City)             // => City: Omaha
    fmt.Println("State:", subscriber.State)           // => State: NE
    fmt.Println("Postal Code:", subscriber.PostalCode) // => Postal Code: 68111

    var employee Employee
    employee.Name = "Joy Carr"
    employee.Street = "456 Elm St"
    employee.City = "Portland"
    employee.State = "OR"
    employee.PostalCode = "97222"
    fmt.Println("Name:", employee.Name)             // => Name: Joy Carr
    fmt.Println("Street:", employee.Street)         // => Street: 456 Elm St
    fmt.Println("City:", employee.City)             // => City: Portland
    fmt.Println("State:", employee.State)           // => State: OR
    fmt.Println("Postal Code:", employee.PostalCode) // => Postal Code: 97222
}


Name: Aman Singh
Street: 123 Oak St
City: Omaha
State: NE
Postal Code: 68111
Name: Joy Carr
Street: 456 Elm St
City: Portland
State: OR
Postal Code: 97222
```

```
DEFINED TYPES

Custom type with underlying basic type

---
package main

import "fmt"

type Liters float64
type Gallons float64
```

```go

type MyType string
// Specify a "receiver parameter" within a function
// definition to make it a method. The receiver
// parameter's type will be the type the method
// gets defined on.
func (m MyType) sayHi() {
    fmt.Println("Hi")
}

func (m MyType) sayHi2() {
    fmt.Println("Hi from", m)
}

func main() {
    var carFuel Gallons
    var busFuel Liters
    // Defining a type defines a conversion
    // from the underlying type to the new type
    carFuel = Gallons(10.0)
    busFuel = Liters(240.0)
    fmt.Println(carFuel) // => 10
    fmt.Println(busFuel) // => 240

    // call sayHi

    value := MyType("a MyType value")
    value.sayHi() // => Hi
    anotherValue := MyType("another value")
    anotherValue.sayHi() // => Hi

    value.sayHi2()
    anotherValue.sayHi2()
}



---
10
240
Hi
Hi
Hi from a MyType value
Hi from another value

Process finished with exit code 0

----
Underlying type is not a superclass


```



```
Embeding structs carries the methods


---
package main

import "fmt"

type Coordinates2 struct {
    Latitude  float64
    Longitude float64
}

type Landmark2 struct {
    Name string
    // An "anonymous field"
    // Has no name of its own, just a type
    Coordinates2
}

// Methods for an embedded type get promoted too!
func (c Coordinates2) Location() string {
    return fmt.Sprintf("(%0.2f, %0.2f)", c.Latitude, c.Longitude)
}

func main() {
    var l Landmark2
    l.Name = "The Googleplex"
    // Fields for "embedded struct" are "promoted"
    l.Latitude = 37.42
    l.Longitude = -122.08
    fmt.Println(l.Name, l.Latitude, l.Longitude)
    // => The Googleplex 37.42 -122.08

    // Methods from embedded type are
    // promoted to outer type
    fmt.Println(l.Location())

}
/* Embed additional types to gain additional methods.
• You’ve heard “favor composition over inheritance”…
• Go implements that principle at the language level.

 */
```



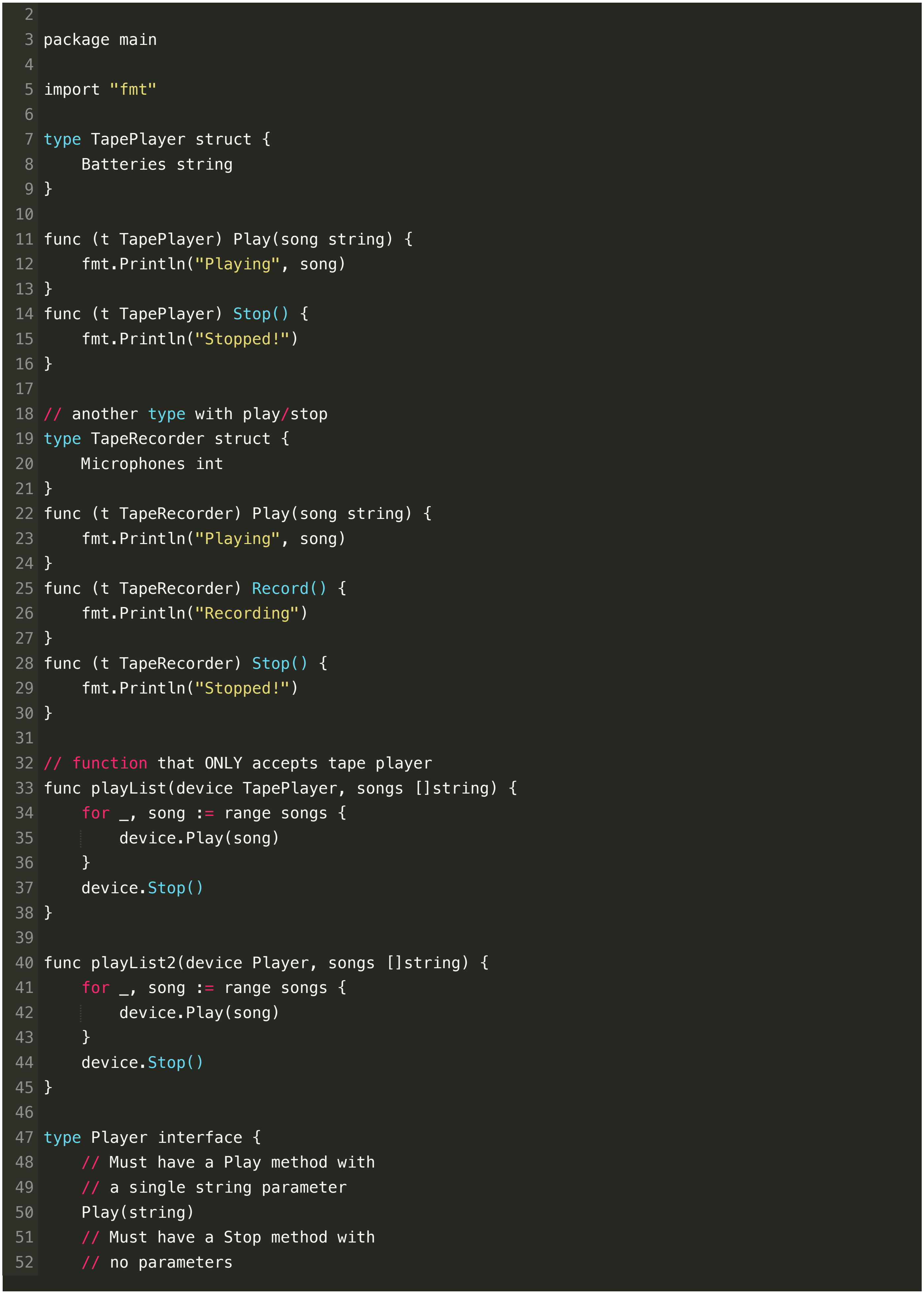
```
https://is.gd/goex_defined_types

package main

import "fmt"

```

```
// YOUR CODE HERE:
// Define a Rectangle struct type with Length and Width
// fields, each of which has a type of float64.

type Rectangle struct {
    Length float64
    Width float64
}


// YOUR CODE HERE:
// Define an Area method on the Rectangle type. It should
// accept no parameters (other than the receiver parameter).
// It should return a float64 value calculated by multiplying
// the receiver's Length by its Width.
func (a Rectangle) Area() float64 {
    return a.Length * a.Width
}

// YOUR CODE HERE:
// Define a Perimeter method on the Rectangle type. It should
// accept no parameters. It should return a float64 value
// representing the receiver's perimeter (2 times its Length
// plus 2 times its Width).

func (a Rectangle) Perimeter() float64 {
    return 2 * a.Length + 2 * a.Width
}

func main() {
    // Once you've defined the above code correctly,
    // this code should compile and run.
    var myRectangle Rectangle
    myRectangle.Length = 2
    myRectangle.Width = 3
    fmt.Println("Area:", myRectangle.Area())           // => Area: 6
    fmt.Println("Perimeter:", myRectangle.Perimeter()) // => Perimeter: 10
}

---
Area: 6
Perimeter: 10


```

```
INTERFACES - MOTIVATION

package main

import "fmt"

```

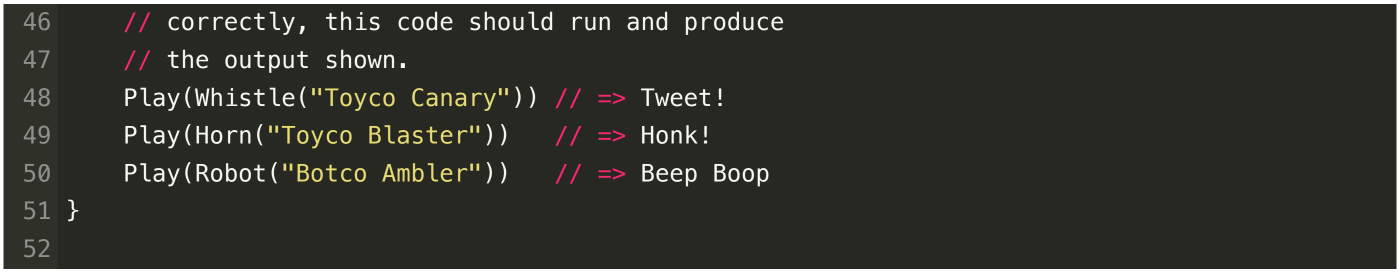
```
type TapePlayer struct {
    Batteries string
}

func (t TapePlayer) Play(song string) {
    fmt.Println("Playing", song)
}
func (t TapePlayer) Stop() {
    fmt.Println("Stopped!")
}

// another type with play/stop
type TapeRecorder struct {
    Microphones int
}
func (t TapeRecorder) Play(song string) {
    fmt.Println("Playing", song)
}
func (t TapeRecorder) Record() {
    fmt.Println("Recording")
}
func (t TapeRecorder) Stop() {
    fmt.Println("Stopped!")
}

// function that ONLY accepts tape player
func playList(device TapePlayer, songs []string) {
    for _, song := range songs {
        device.Play(song)
    }
    device.Stop()
}

func main() {
    mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
    var player TapePlayer
    playList(player, mixtape)
}

/* Cannot use Recorder - even if has same methods

    mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
    var recorder TapeRecorder
    playList(recorder, mixtape)

    prog.go:40:10: cannot use recorder (type TapeRecorder) as type TapePlayer in argument to playList

 */
```

```
INTERFACES
```

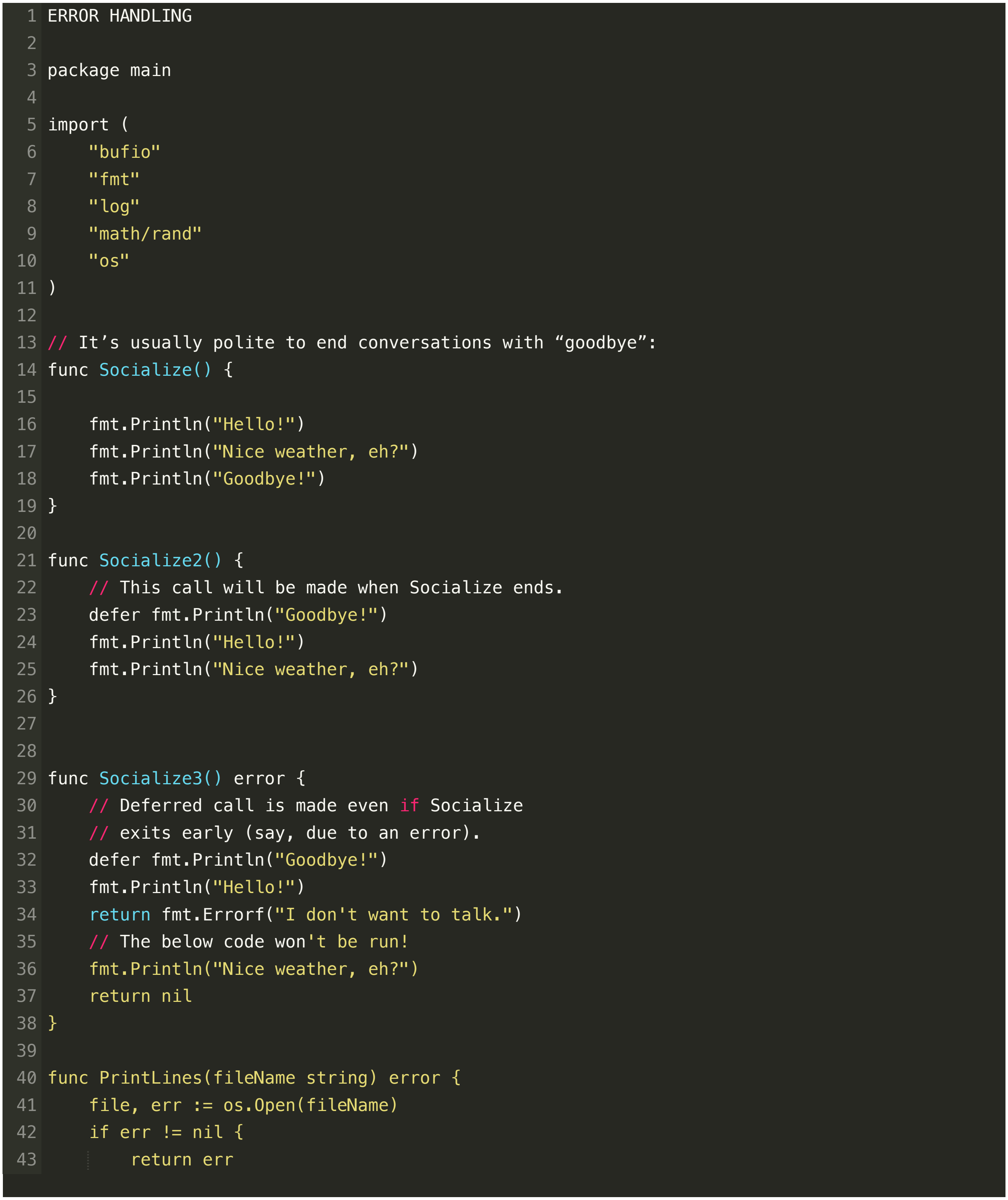


```go

package main

import "fmt"

type TapePlayer struct {
    Batteries string
}

func (t TapePlayer) Play(song string) {
    fmt.Println("Playing", song)
}
func (t TapePlayer) Stop() {
    fmt.Println("Stopped!")
}

// another type with play/stop
type TapeRecorder struct {
    Microphones int
}
func (t TapeRecorder) Play(song string) {
    fmt.Println("Playing", song)
}
func (t TapeRecorder) Record() {
    fmt.Println("Recording")
}
func (t TapeRecorder) Stop() {
    fmt.Println("Stopped!")
}

// function that ONLY accepts tape player
func playList(device TapePlayer, songs []string) {
    for _, song := range songs {
        device.Play(song)
    }
    device.Stop()
}

func playList2(device Player, songs []string) {
    for _, song := range songs {
        device.Play(song)
    }
    device.Stop()
}

type Player interface {
    // Must have a Play method with
    // a single string parameter
    Play(string)
    // Must have a Stop method with
    // no parameters
    Stop()
}
/*

This Works:

    mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
    var player TapePlayer
    playList(player, mixtape)

Cannot use Recorder - even if has same methods

mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
var recorder TapeRecorder
playList(recorder, mixtape)

prog.go:40:10: cannot use recorder (type TapeRecorder) as type TapePlayer in argument to playList

*/


func main() {
    mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
    var player TapePlayer
    playList(player, mixtape)

    // This works
    var recorder TapeRecorder
    playList2(player, mixtape)
    playList2(recorder, mixtape)

    TryOut(TapeRecorder{})
    TryOut2(TapeRecorder{})
}

// Type assertions

func TryOut(player Player) {
    player.Play("Test Track")
    player.Stop()
    // Player interface doesn't include this method!
    // THIS WILL NOT WORK
    // player.Record()
    // even if we call it TryOut(recorder)
}

// This will work
func TryOut2(player Player) {
    player.Play("Test Track")
    player.Stop()
    // Do a type assertion to get the concrete value back...
    recorder := player.(TapeRecorder)
    // Then you can call Record on that.
    recorder.Record()
}
```

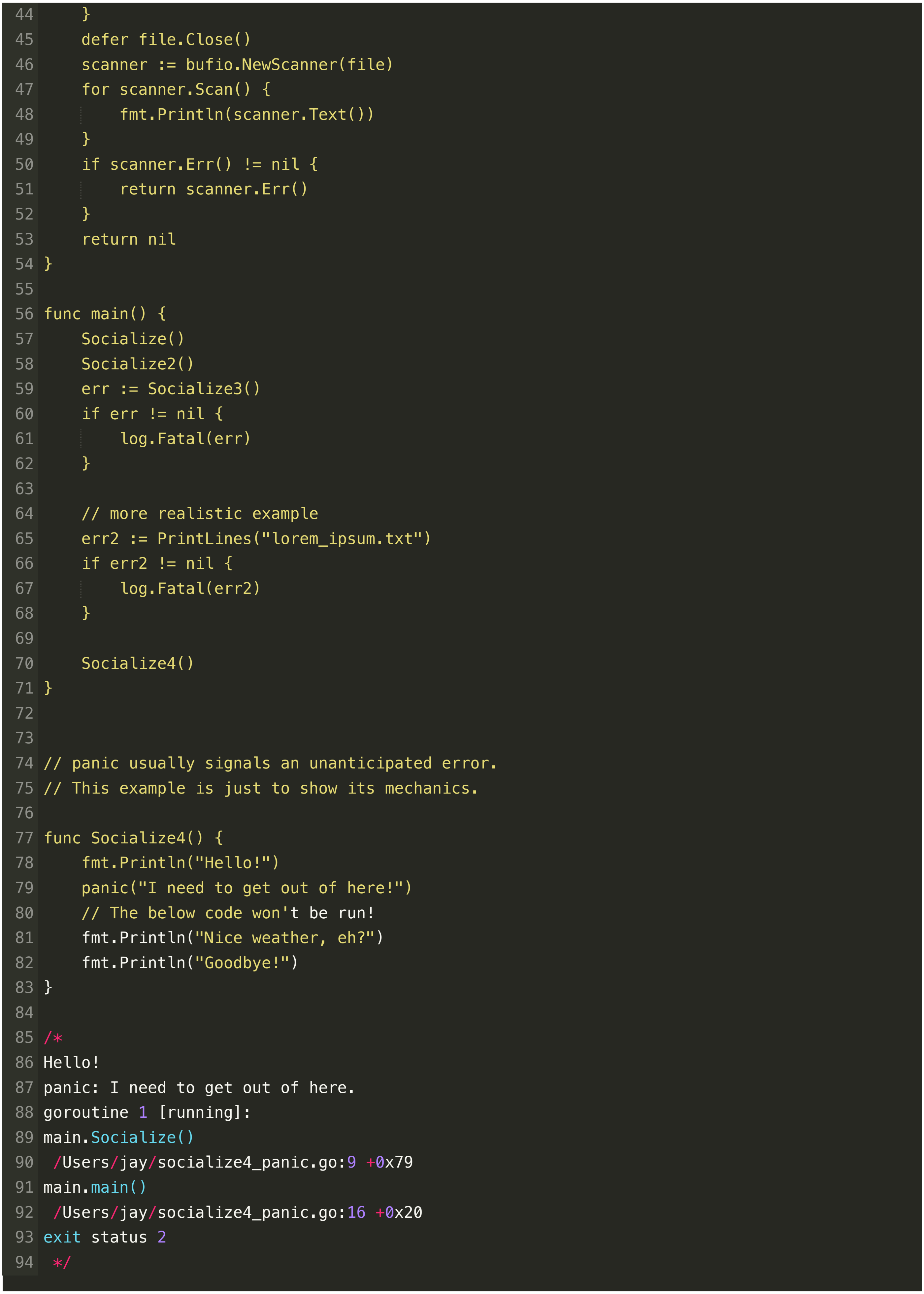
```
https://is.gd/goex_interfaces

package main

import "fmt"

type Whistle string
func (w Whistle) MakeSound() {
    fmt.Println("Tweet!")
}

type Horn string
func (h Horn) MakeSound() {
    fmt.Println("Honk!")
}

type Robot string
func (r Robot) MakeSound() {
    fmt.Println("Beep Boop")
}
func (r Robot) Walk() {
    fmt.Println("Powering legs")
}

// YOUR CODE HERE:
// Define a NoiseMaker interface type, which the above
// Whistle, Horn, and Robot types will all satisfy.
// It should require one method, MakeSound, which has
// no parameters and no return values.

type NoiseMaker interface {
    MakeSound()
}

// YOUR CODE HERE:
// Define a Play function that accepts a parameter with
// the NoiseMaker interface. Play should call MakeSound
// on the parameter it receives.

func Play(dev NoiseMaker) {
    dev.MakeSound()
}

func main() {
    // When the above code has been implemented
```



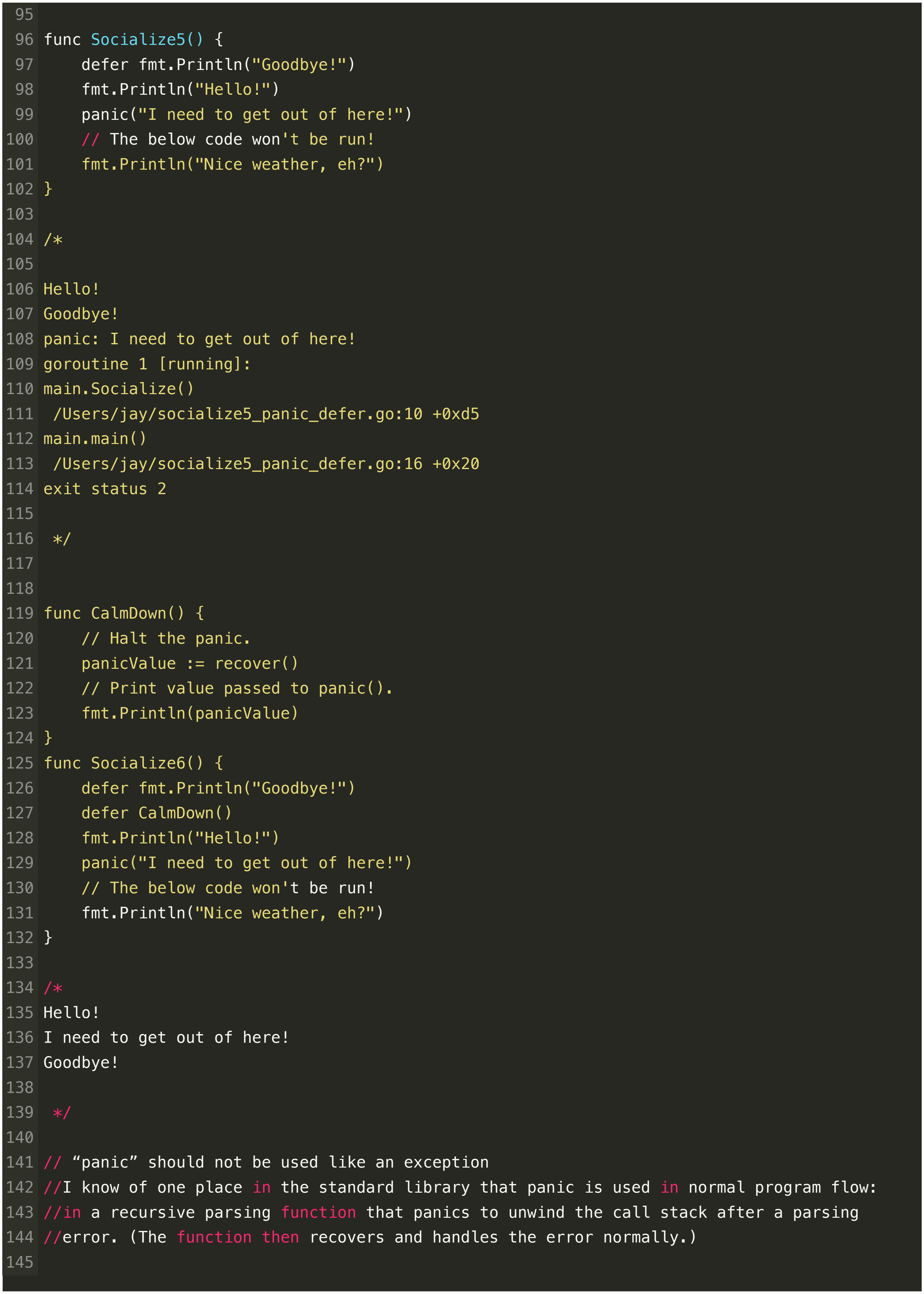
```
    // correctly, this code should run and produce
    // the output shown.
    Play(Whistle("Toyco Canary")) // => Tweet!
    Play(Horn("Toyco Blaster"))   // => Honk!
    Play(Robot("Botco Ambler"))   // => Beep Boop
}

```



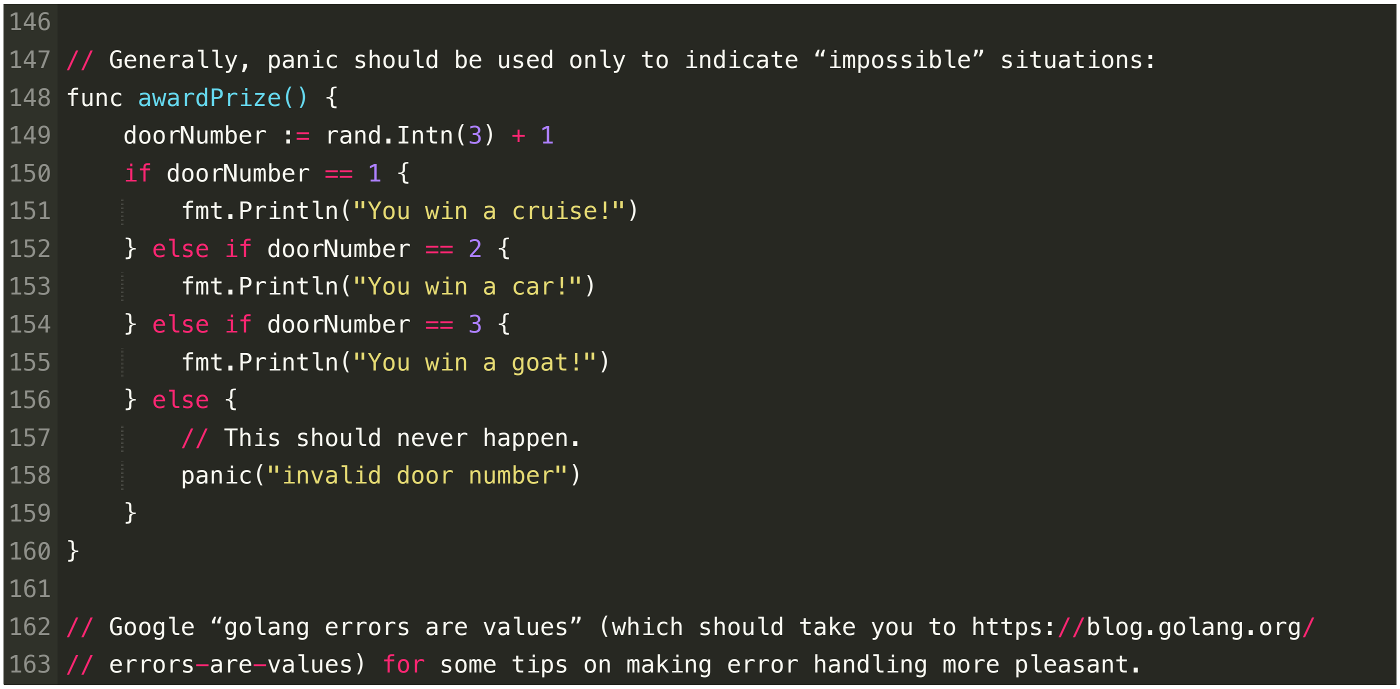
```
ERROR HANDLING

package main

import (
    "bufio"
    "fmt"
    "log"
    "math/rand"
    "os"
)

// It’s usually polite to end conversations with “goodbye”:
func Socialize() {

    fmt.Println("Hello!")
    fmt.Println("Nice weather, eh?")
    fmt.Println("Goodbye!")
}

func Socialize2() {
    // This call will be made when Socialize ends.
    defer fmt.Println("Goodbye!")
    fmt.Println("Hello!")
    fmt.Println("Nice weather, eh?")
}


func Socialize3() error {
    // Deferred call is made even if Socialize
    // exits early (say, due to an error).
    defer fmt.Println("Goodbye!")
    fmt.Println("Hello!")
    return fmt.Errorf("I don't want to talk.")
    // The below code won't be run!
    fmt.Println("Nice weather, eh?")
    return nil
}

func PrintLines(fileName string) error {
    file, err := os.Open(fileName)
    if err != nil {
        return err
```



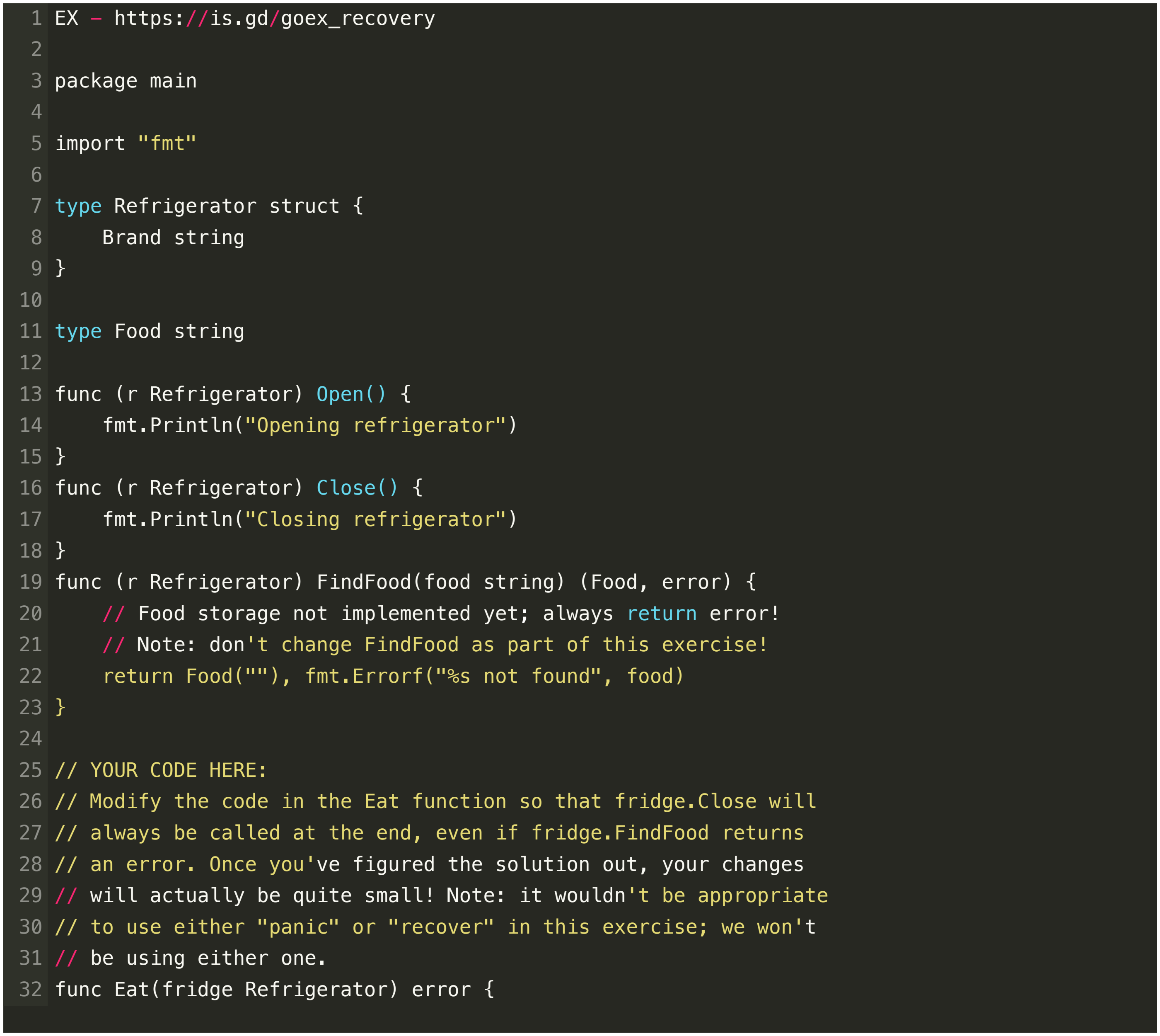
```
    }
    defer file.Close()
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        fmt.Println(scanner.Text())
    }
    if scanner.Err() != nil {
        return scanner.Err()
    }
    return nil
}

func main() {
    Socialize()
    Socialize2()
    err := Socialize3()
    if err != nil {
        log.Fatal(err)
    }

    // more realistic example
    err2 := PrintLines("lorem_ipsum.txt")
    if err2 != nil {
        log.Fatal(err2)
    }

    Socialize4()
}


// panic usually signals an unanticipated error.
// This example is just to show its mechanics.

func Socialize4() {
    fmt.Println("Hello!")
    panic("I need to get out of here!")
    // The below code won't be run!
    fmt.Println("Nice weather, eh?")
    fmt.Println("Goodbye!")
}

/*
Hello!
panic: I need to get out of here.
goroutine 1 [running]:
main.Socialize()
 /Users/jay/socialize4_panic.go:9 +0x79
main.main()
 /Users/jay/socialize4_panic.go:16 +0x20
exit status 2
 */
```



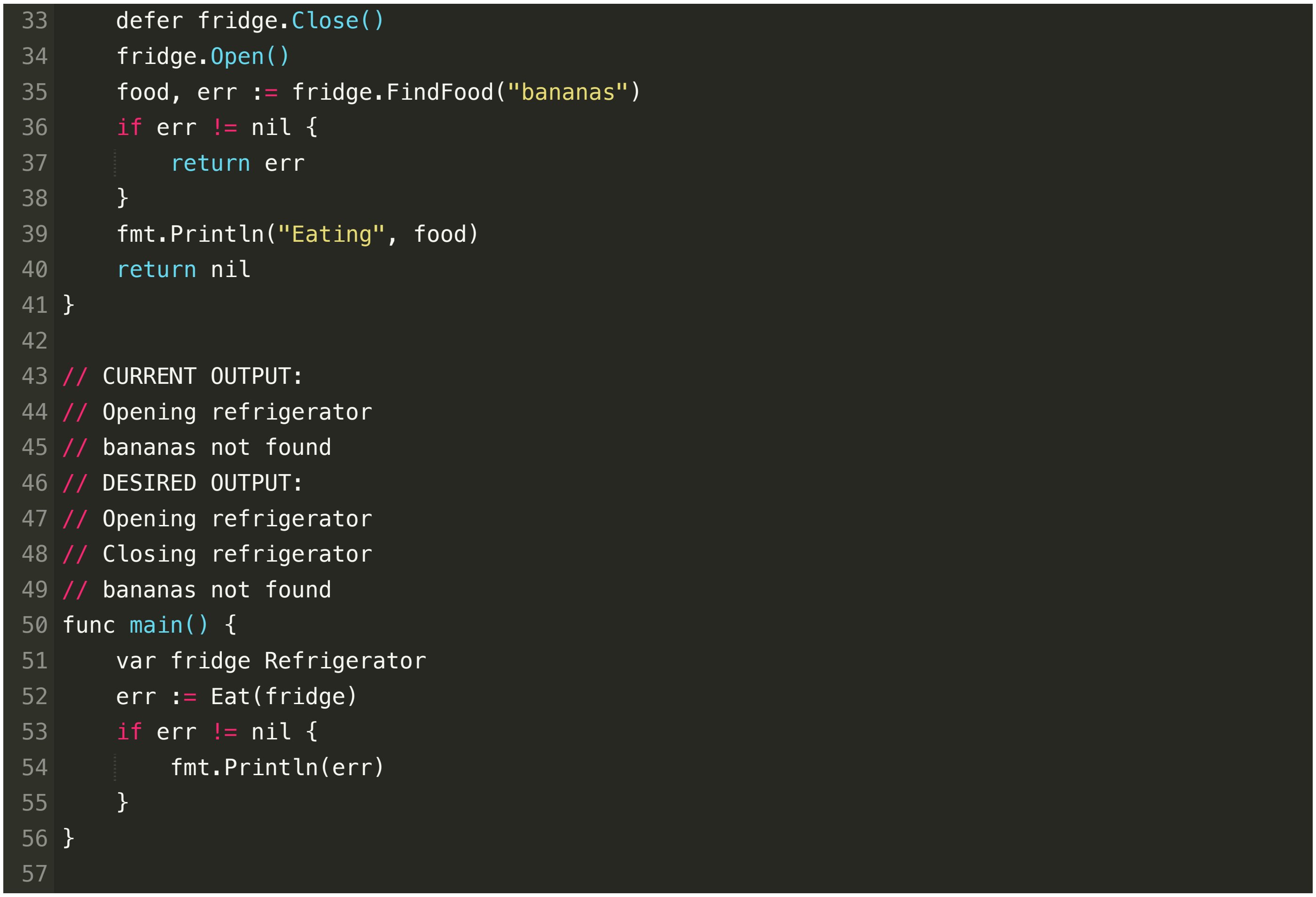
```go

func Socialize5() {
    defer fmt.Println("Goodbye!")
    fmt.Println("Hello!")
    panic("I need to get out of here!")
    // The below code won't be run!
    fmt.Println("Nice weather, eh?")
}

/*

Hello!
Goodbye!
panic: I need to get out of here!
goroutine 1 [running]:
main.Socialize()
 /Users/jay/socialize5_panic_defer.go:10 +0xd5
main.main()
 /Users/jay/socialize5_panic_defer.go:16 +0x20
exit status 2

 */


func CalmDown() {
    // Halt the panic.
    panicValue := recover()
    // Print value passed to panic().
    fmt.Println(panicValue)
}
func Socialize6() {
    defer fmt.Println("Goodbye!")
    defer CalmDown()
    fmt.Println("Hello!")
    panic("I need to get out of here!")
    // The below code won't be run!
    fmt.Println("Nice weather, eh?")
}

/*
Hello!
I need to get out of here!
Goodbye!

 */

// “panic” should not be used like an exception
//I know of one place in the standard library that panic is used in normal program flow:
//in a recursive parsing function that panics to unwind the call stack after a parsing
//error. (The function then recovers and handles the error normally.)

```



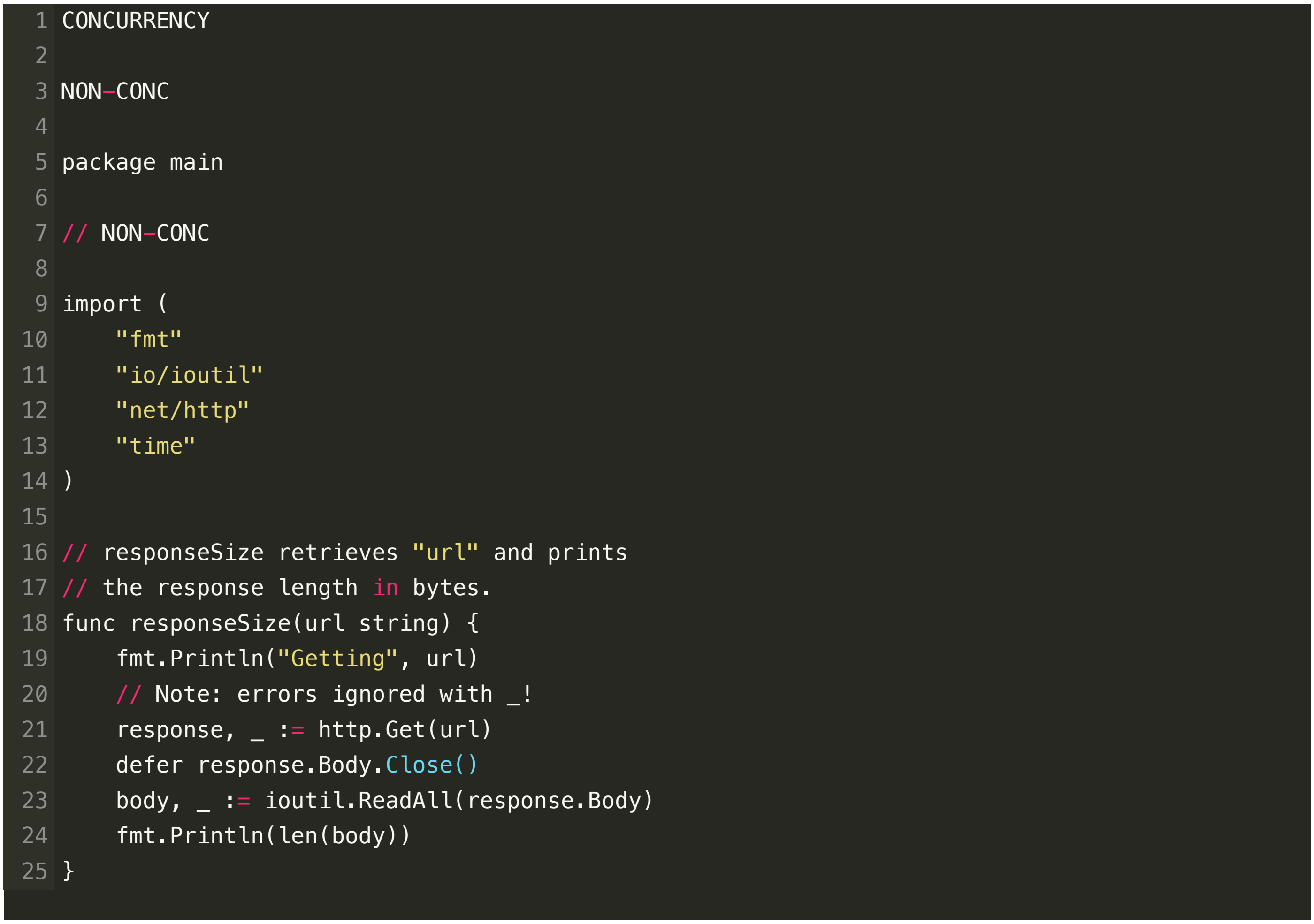
```go

// Generally, panic should be used only to indicate “impossible” situations:
func awardPrize() {
    doorNumber := rand.Intn(3) + 1
    if doorNumber == 1 {
        fmt.Println("You win a cruise!")
    } else if doorNumber == 2 {
        fmt.Println("You win a car!")
    } else if doorNumber == 3 {
        fmt.Println("You win a goat!")
    } else {
        // This should never happen.
        panic("invalid door number")
    }
}

// Google “golang errors are values” (which should take you to https://blog.golang.org/
// errors-are-values) for some tips on making error handling more pleasant.
```



```go
EX - https://is.gd/goex_recovery

package main

import "fmt"

type Refrigerator struct {
    Brand string
}

type Food string

func (r Refrigerator) Open() {
    fmt.Println("Opening refrigerator")
}
func (r Refrigerator) Close() {
    fmt.Println("Closing refrigerator")
}
func (r Refrigerator) FindFood(food string) (Food, error) {
    // Food storage not implemented yet; always return error!
    // Note: don't change FindFood as part of this exercise!
    return Food(""), fmt.Errorf("%s not found", food)
}

// YOUR CODE HERE:
// Modify the code in the Eat function so that fridge.Close will
// always be called at the end, even if fridge.FindFood returns
// an error. Once you've figured the solution out, your changes
// will actually be quite small! Note: it wouldn't be appropriate
// to use either "panic" or "recover" in this exercise; we won't
// be using either one.
func Eat(fridge Refrigerator) error {
```

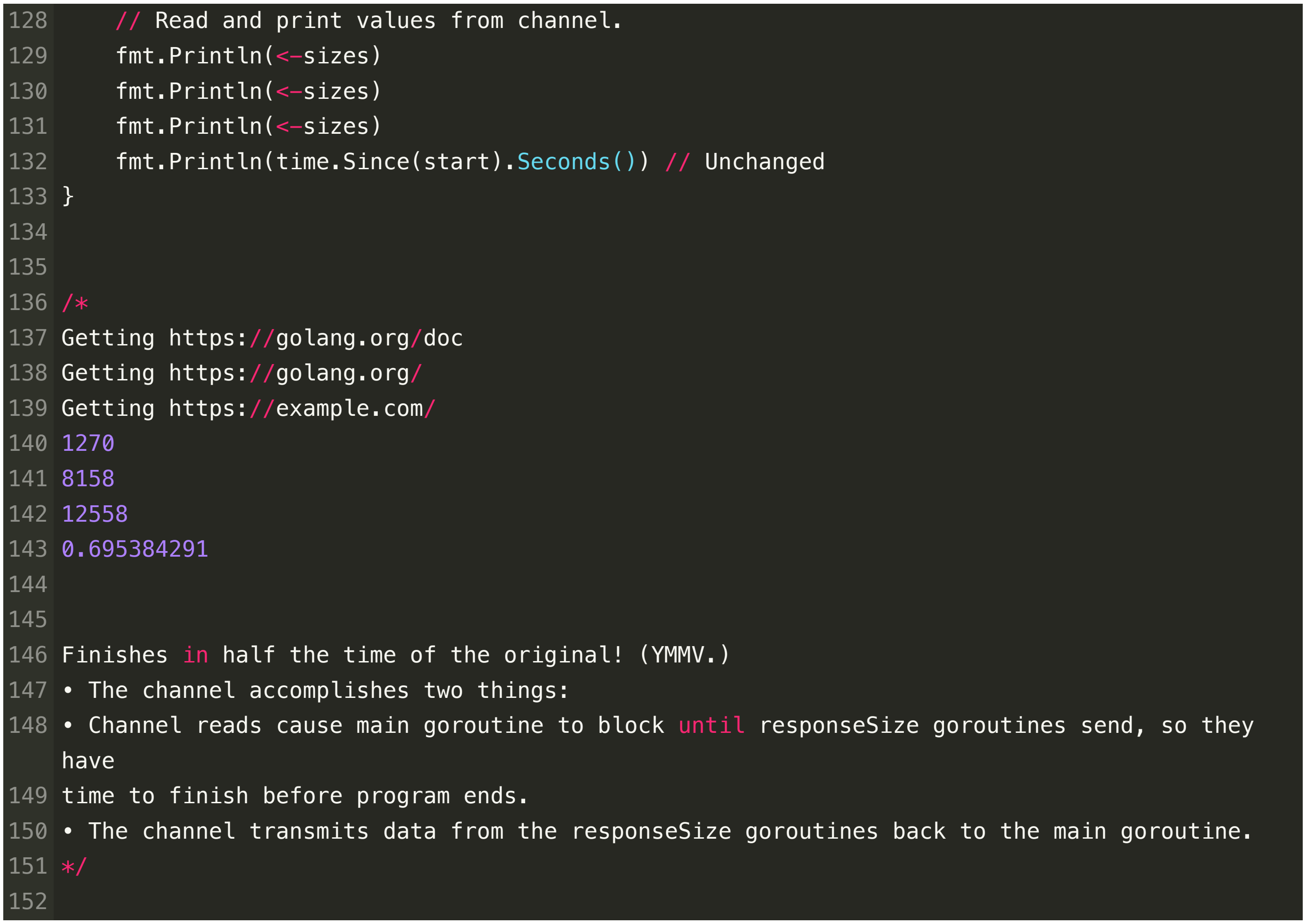


```go
    defer fridge.Close()
    fridge.Open()
    food, err := fridge.FindFood("bananas")
    if err != nil {
        return err
    }
    fmt.Println("Eating", food)
    return nil
}

// CURRENT OUTPUT:
// Opening refrigerator
// bananas not found
// DESIRED OUTPUT:
// Opening refrigerator
// Closing refrigerator
// bananas not found
func main() {
    var fridge Refrigerator
    err := Eat(fridge)
    if err != nil {
        fmt.Println(err)
    }
}

```

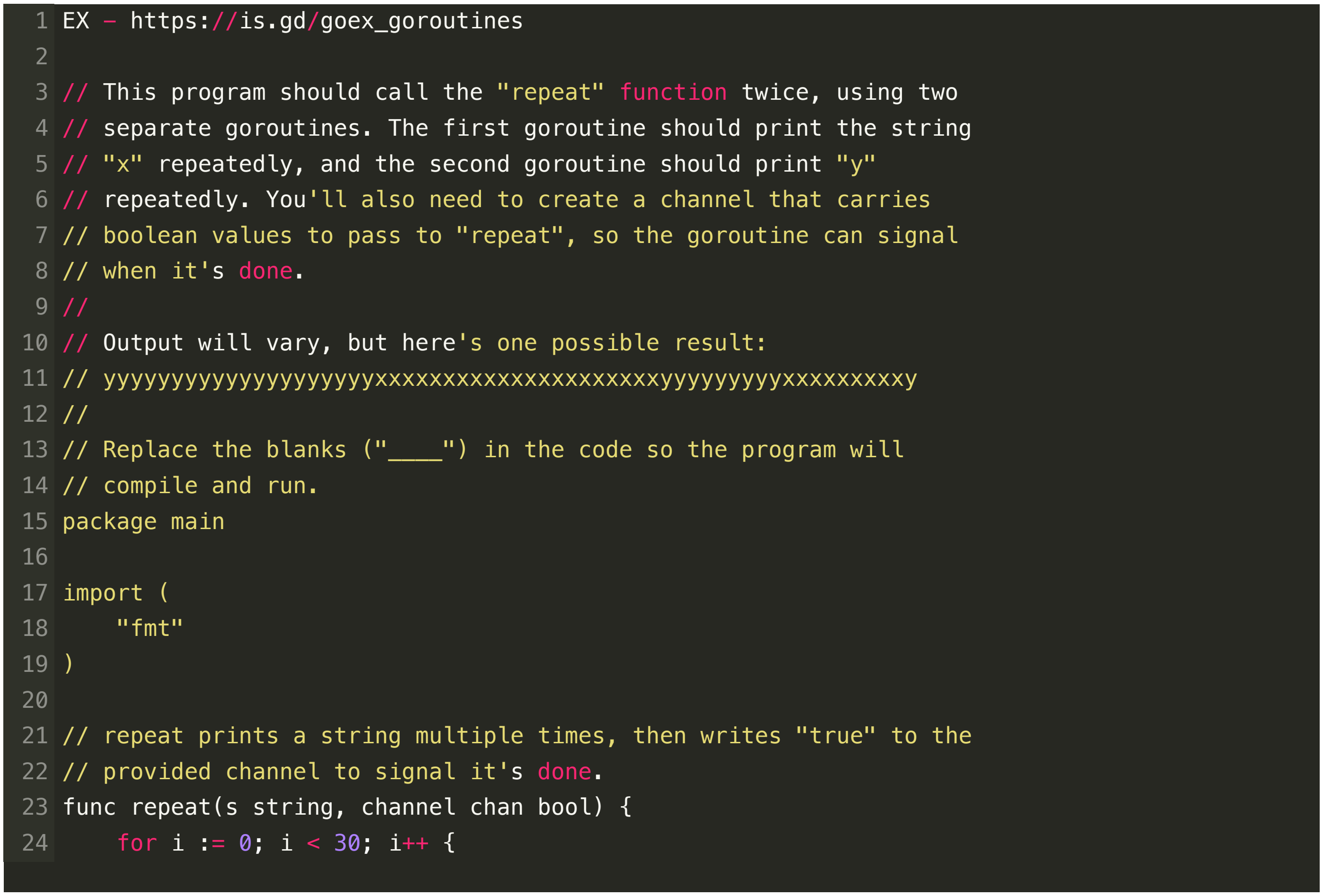


```go
CONCURRENCY

NON-CONC

package main

// NON-CONC

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "time"
)

// responseSize retrieves "url" and prints
// the response length in bytes.
func responseSize(url string) {
    fmt.Println("Getting", url)
    // Note: errors ignored with _!
    response, _ := http.Get(url)
    defer response.Body.Close()
    body, _ := ioutil.ReadAll(response.Body)
    fmt.Println(len(body))
}
```

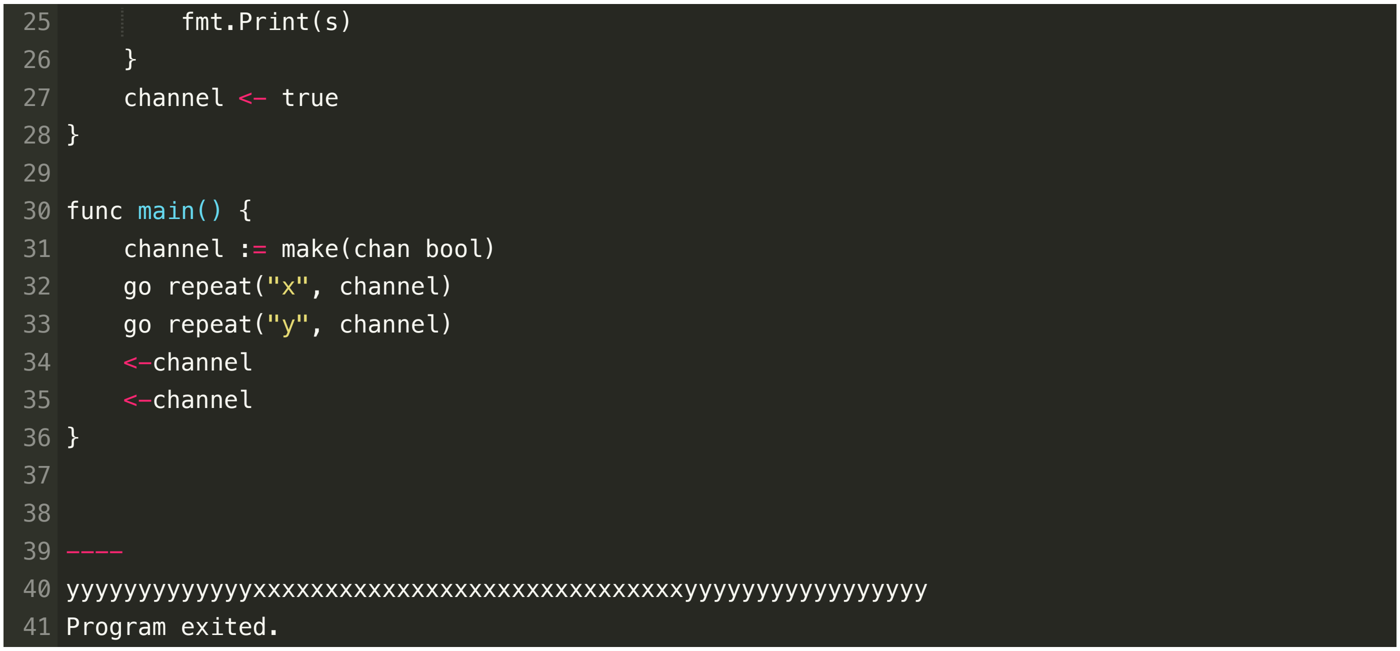
```

func main() {
    // Note the time we started.
    start := time.Now()
    responseSize("https://example.com/")
    responseSize("https://golang.org/")
    responseSize("https://golang.org/doc")
    // Print how long everything took.
    fmt.Println(time.Since(start).Seconds(), "seconds")
}

/*
Getting https://example.com/
1270
Getting https://golang.org/
8158
Getting https://golang.org/doc
12558
1.5341211000000001 seconds
*/



---

package main

// NON-CONC

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "time"

)

// responseSize retrieves "url" and prints
// the response length in bytes.
// UNCHANGED func responseSize(url string) {
func responseSize2(url string) {
    fmt.Println("Getting", url)
    // Note: errors ignored with _!
    response, _ := http.Get(url)
    defer response.Body.Close()
    body, _ := ioutil.ReadAll(response.Body)
    fmt.Println(len(body))
}

func main() {
    // Note the time we started.
    start := time.Now()
    go responseSize2("https://example.com/")
    go responseSize2("https://golang.org/")
    go responseSize2("https://golang.org/doc")
    // Print how long everything took.
    fmt.Println(time.Since(start).Seconds(), "seconds")
}

/*
Getting https://example.com/
Getting https://golang.org/
9.378e-06 seconds
Getting https://golang.org/doc

DOES NOT WAIT to finish
*/

---

package main

// NON-CONC

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "time"

)

// responseSize retrieves "url" and prints
// the response length in bytes.

func responseSize3(url string, channel chan int) {
    fmt.Println("Getting", url)
    // Note: errors ignored with _!
    response, _ := http.Get(url)
    defer response.Body.Close()
    body, _ := ioutil.ReadAll(response.Body)
    channel <- len(body)
}

func main() {
    start := time.Now() // Unchanged
    // Make a channel to carry ints.
    sizes := make(chan int)
    // Pass channel to each call to responseSize.
    go responseSize3("https://example.com/", sizes)
    go responseSize3("https://golang.org/", sizes)
    go responseSize3("https://golang.org/doc", sizes)
```



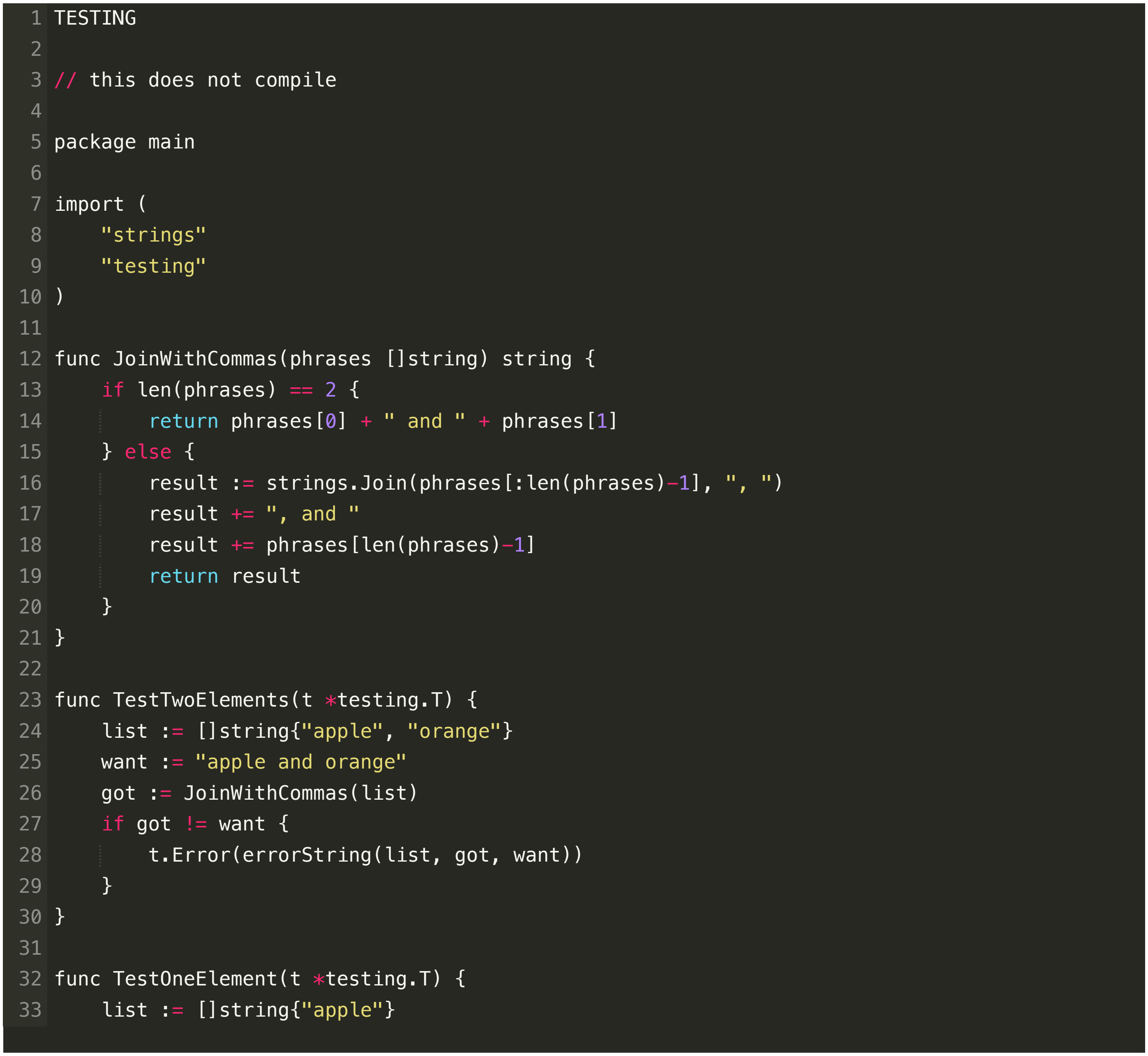
```
    // Read and print values from channel.
    fmt.Println(<-sizes)
    fmt.Println(<-sizes)
    fmt.Println(<-sizes)
    fmt.Println(time.Since(start).Seconds()) // Unchanged
}


/*
Getting https://golang.org/doc
Getting https://golang.org/
Getting https://example.com/
1270
8158
12558
0.695384291


Finishes in half the time of the original! (YMMV.)
• The channel accomplishes two things:
• Channel reads cause main goroutine to block until responseSize goroutines send, so they have
time to finish before program ends.
• The channel transmits data from the responseSize goroutines back to the main goroutine.
*/

```



```
EX - https://is.gd/goex_goroutines

// This program should call the "repeat" function twice, using two
// separate goroutines. The first goroutine should print the string
// "x" repeatedly, and the second goroutine should print "y"
// repeatedly. You'll also need to create a channel that carries
// boolean values to pass to "repeat", so the goroutine can signal
// when it's done.
//
// Output will vary, but here's one possible result:
// yyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyxxxxxxxxxy
//
// Replace the blanks ("____") in the code so the program will
// compile and run.
package main

import (
    "fmt"
)

// repeat prints a string multiple times, then writes "true" to the
// provided channel to signal it's done.
func repeat(s string, channel chan bool) {
    for i := 0; i < 30; i++ {
```



```
        fmt.Print(s)
    }
    channel <- true
}

func main() {
    channel := make(chan bool)
    go repeat("x", channel)
    go repeat("y", channel)
    <-channel
    <-channel
}


----
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyy
Program exited.
```



```
TESTING

// this does not compile

package main

import (
    "strings"
    "testing"
)

func JoinWithCommas(phrases []string) string {
    if len(phrases) == 2 {
        return phrases[0] + " and " + phrases[1]
    } else {
        result := strings.Join(phrases[:len(phrases)-1], ", ")
        result += ", and "
        result += phrases[len(phrases)-1]
        return result
    }
}

func TestTwoElements(t *testing.T) {
    list := []string{"apple", "orange"}
    want := "apple and orange"
    got := JoinWithCommas(list)
    if got != want {
        t.Error(errorString(list, got, want))
    }
}

func TestOneElement(t *testing.T) {
    list := []string{"apple"}
```

```go
    want := "apple"
    got := JoinWithCommas(list)
    if got != want {
        t.Error(errorString(list, got, want))
    }
}

func TestThreeElements(t *testing.T) {
    list := []string{"apple", "orange", "pear"}
    want := "apple, orange, and pear"
    got := JoinWithCommas(list)
    if got != want {
        t.Error(errorString(list, got, want))
    }
}



```

```
TESTING

moved to package prose - see https://github.com/headfirstgo/prose


➜  s-oly-lt-go-introduction go test prose
ok      prose   0.011s

```

```
WEB


```