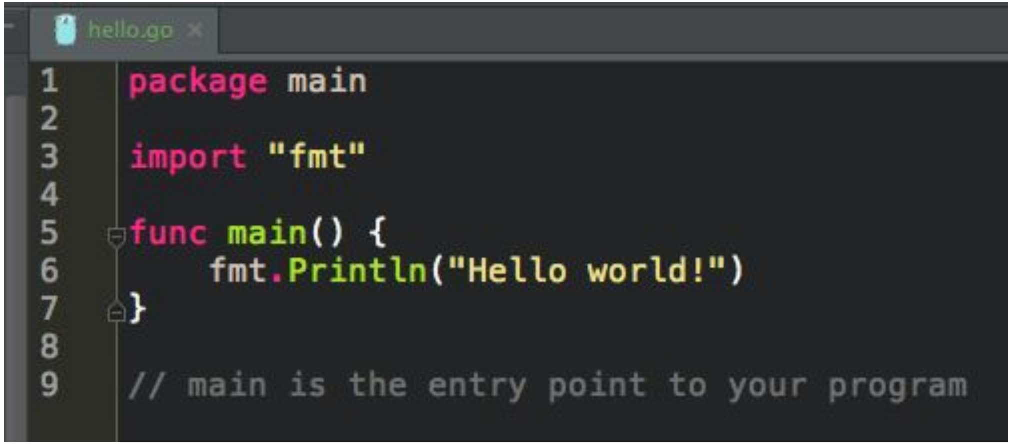
functions

funcs, func expressions, closure, returning funcs, recursion, the stack

functions

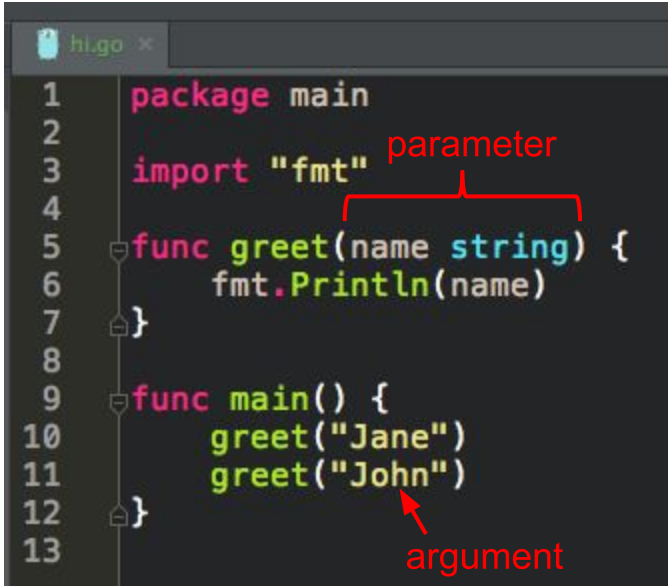
- functions in go are types
    - functions behave as types in go
    - use like any other type
        - declare them as variables
        - pass functions around just as you'd pass types around
        - pass functions just like any other argument / parameter
            - pass them into functions as arguments
            - return them from functions
        - declare functions inside other functions
    - similar to JavaScript



```go
package main

import "fmt"

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

// main is the entry point to your program
```

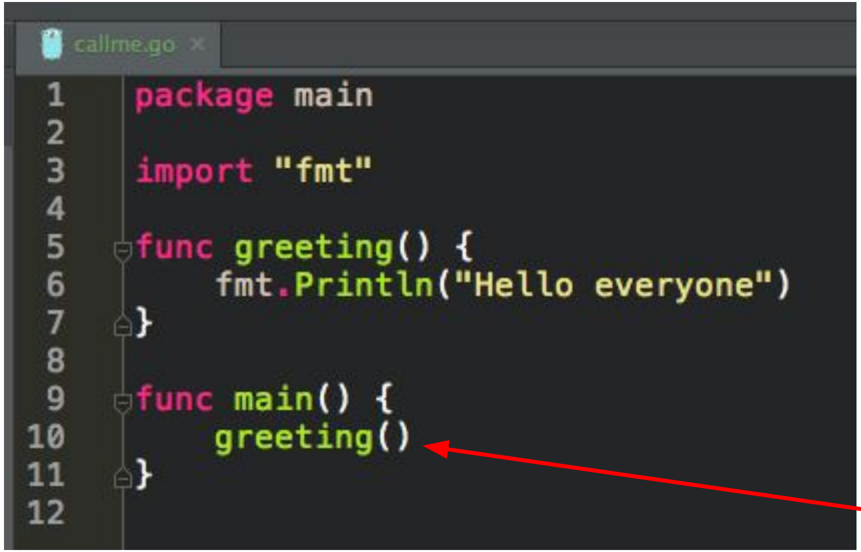
func main

the entry point for your program



```go
package main

import "fmt"

func greet(name string) {
	fmt.Println(name)
}

func main() {
	greet("Jane")
	greet("John")
}
```

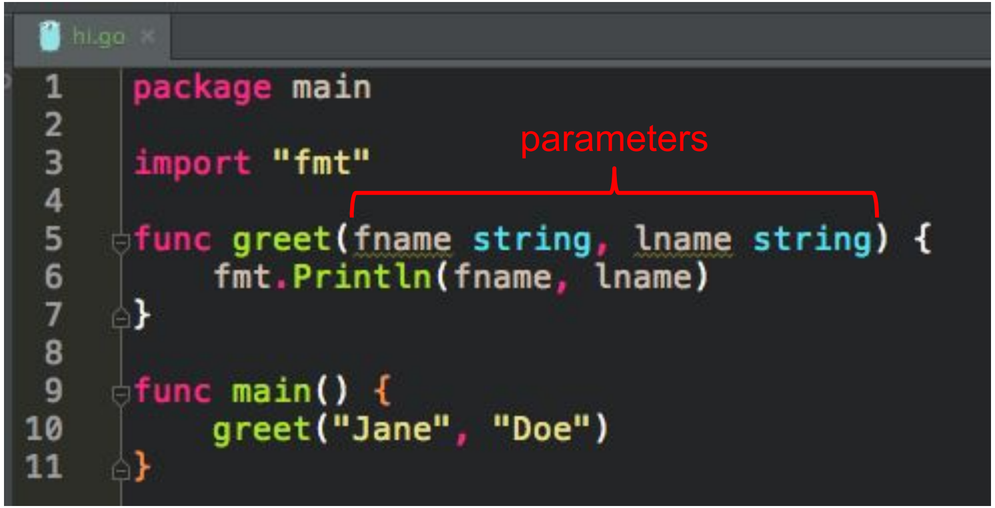
parameters & arguments



```go
package main

import "fmt"

func greeting() {
    fmt.Println("Hello everyone")
}

func main() {
    greeting()
}

```

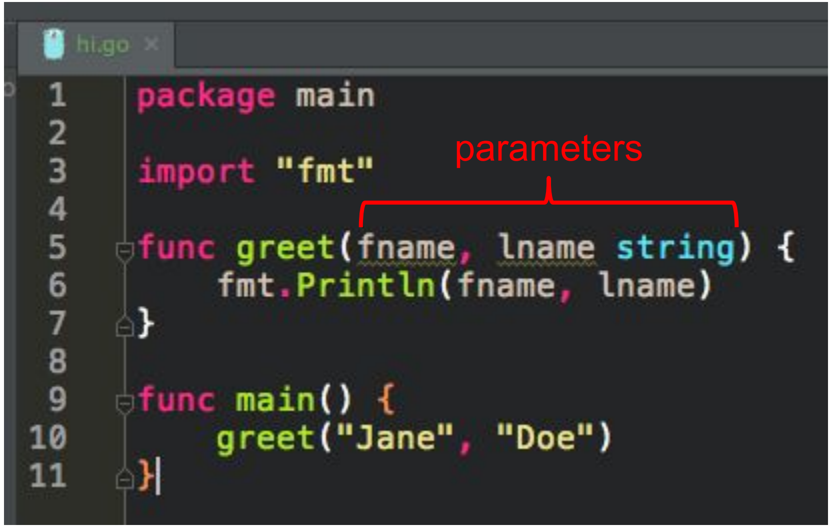
You need the ()

calling a function



```go
package main

import "fmt"

func greet(fname string, lname string) {
	fmt.Println(fname, lname)
}

func main() {
	greet("Jane", "Doe")
}
```

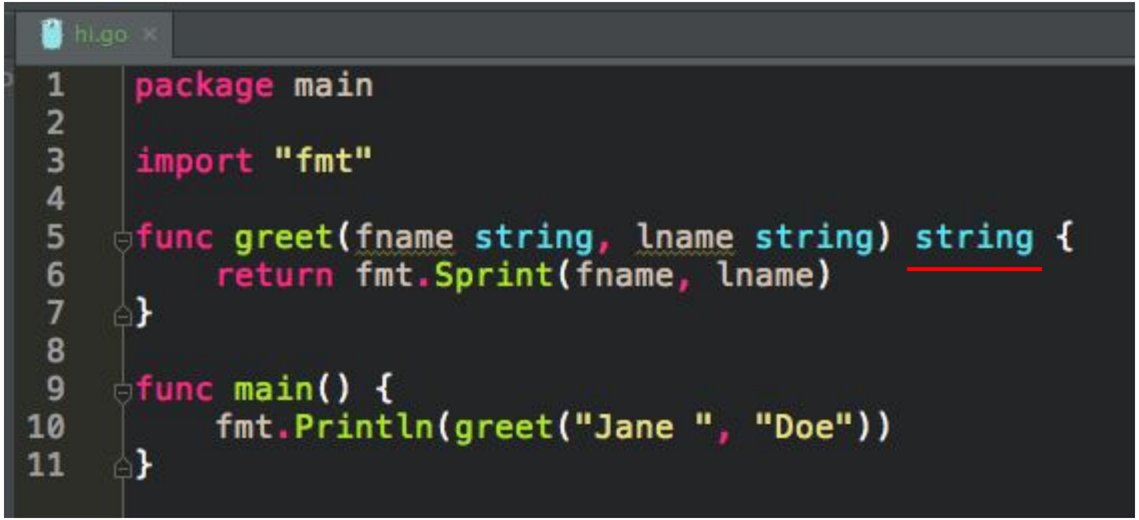
two params



parameters

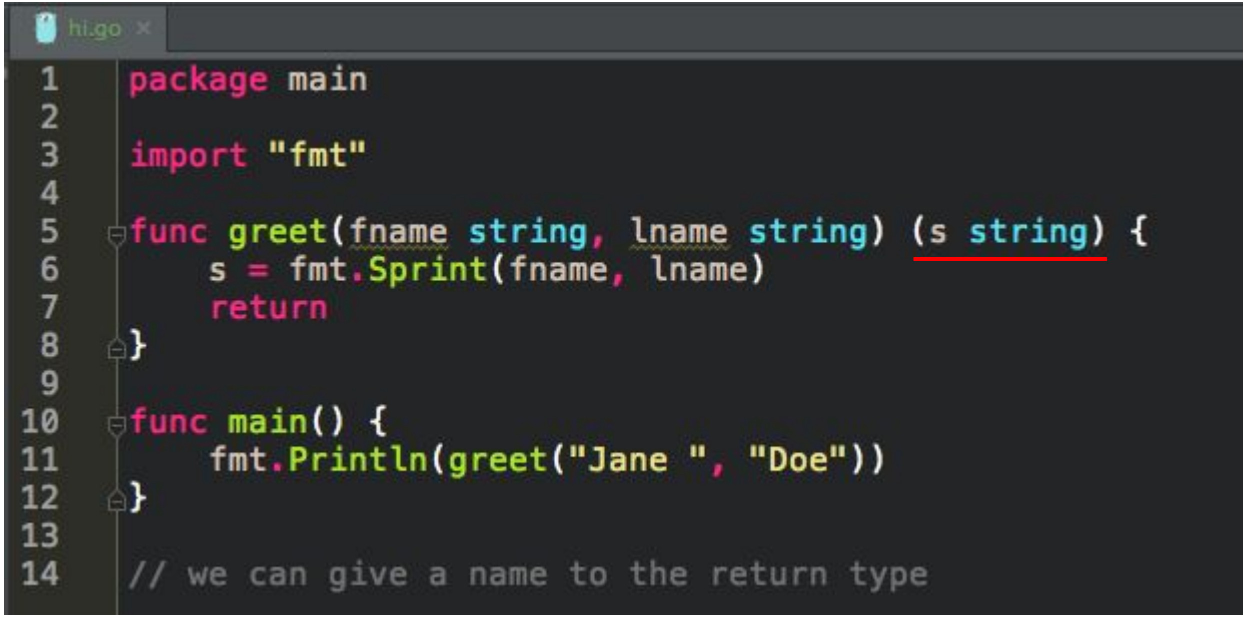
```go
package main

import "fmt"

func greet(fname, lname string) {
	fmt.Println(fname, lname)
}

func main() {
	greet("Jane", "Doe")
}
```

two params



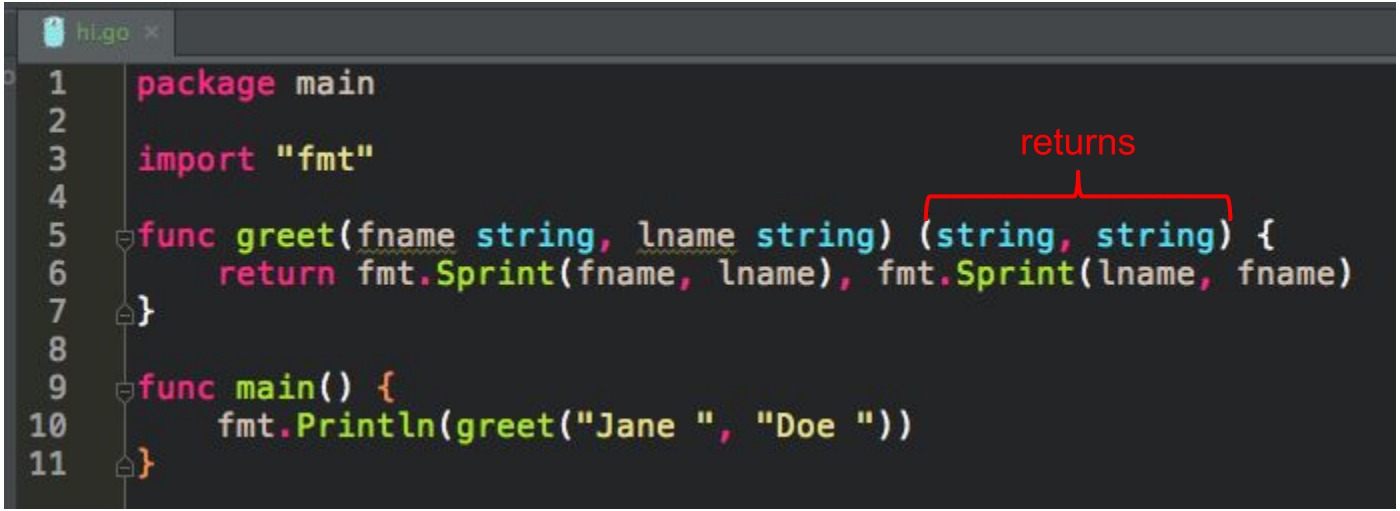
```go
package main

import "fmt"

func greet(fname string, lname string) string {
	return fmt.Sprint(fname, lname)
}

func main() {
	fmt.Println(greet("Jane ", "Doe"))
}
```

return



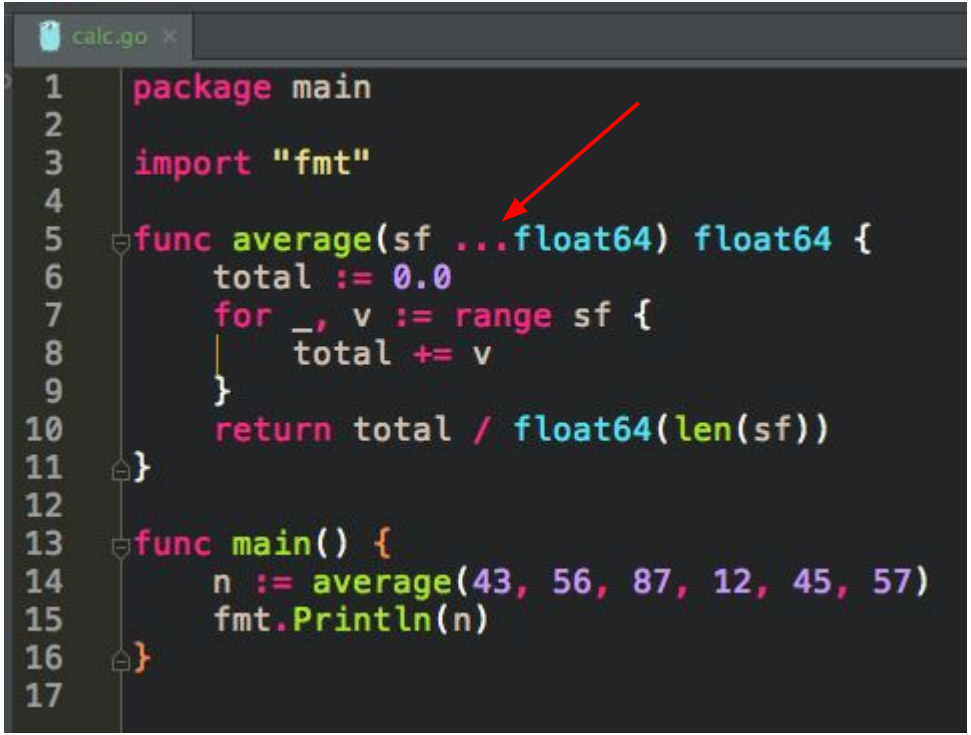
```go
package main

import "fmt"

func greet(fname string, lname string) (s string) {
    s = fmt.Sprint(fname, lname)
    return
}

func main() {
    fmt.Println(greet("Jane ", "Doe"))
}

// we can give a name to the return type
```

named return



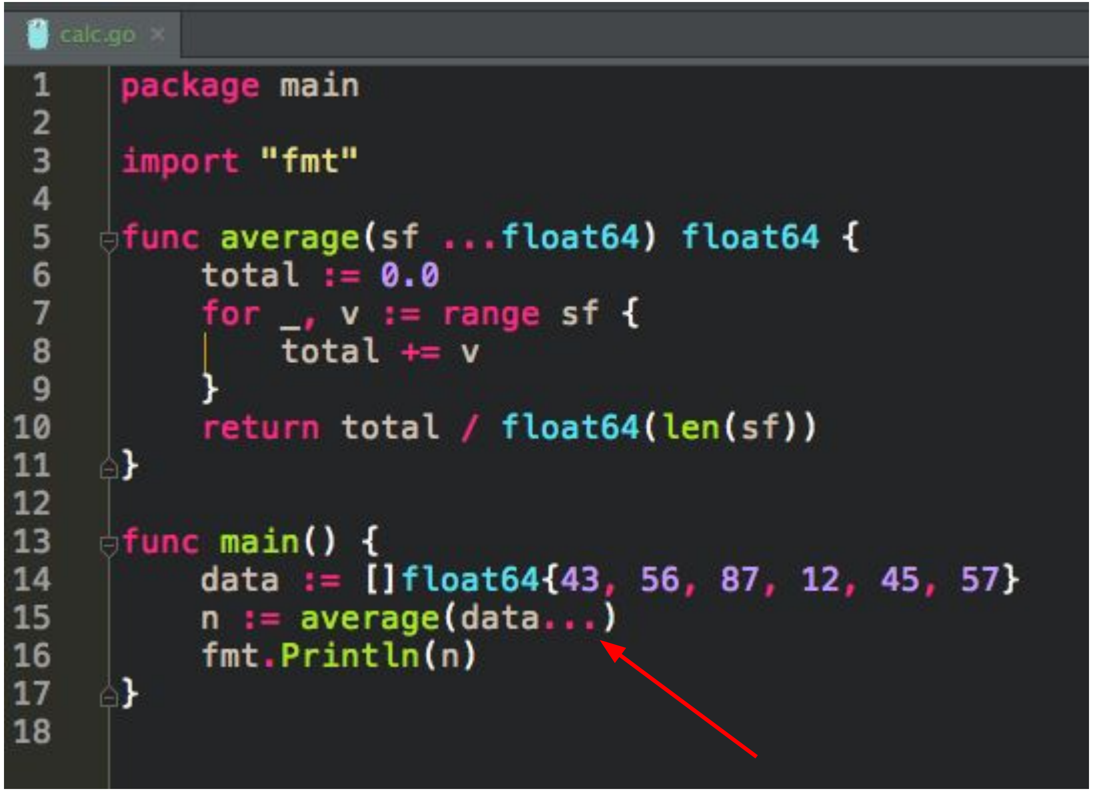
```go
package main

import "fmt"

func greet(fname string, lname string) (string, string) {
	return fmt.Sprint(fname, lname), fmt.Sprint(lname, fname)
}

func main() {
	fmt.Println(greet("Jane ", "Doe "))
}
```

return multiple



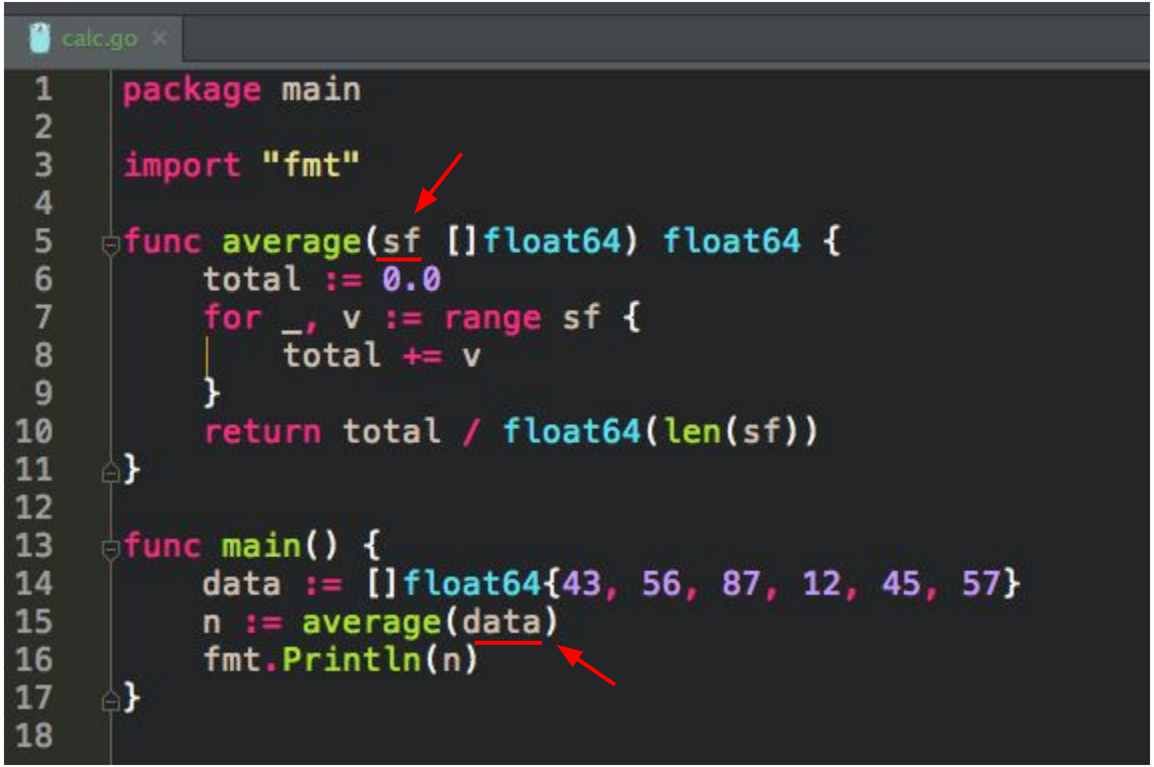
```go
package main

import "fmt"

func average(sf ...float64) float64 {
	total := 0.0
	for _, v := range sf {
		total += v
	}
	return total / float64(len(sf))
}

func main() {
	n := average(43, 56, 87, 12, 45, 57)
	fmt.Println(n)
}

```

variadic parameters



```go
package main

import "fmt"

func average(sf ...float64) float64 {
	total := 0.0
	for _, v := range sf {
		total += v
	}
	return total / float64(len(sf))
}

func main() {
	data := []float64{43, 56, 87, 12, 45, 57}
	n := average(data...)
	fmt.Println(n)
}
```

variadic arguments



```go
package main

import "fmt"

func average(sf []float64) float64 {
	total := 0.0
	for _, v := range sf {
		total += v
	}
	return total / float64(len(sf))
}

func main() {
	data := []float64{43, 56, 87, 12, 45, 57}
	n := average(data)
	fmt.Println(n)
}
```

parameter name does not have to match argument name

exercise

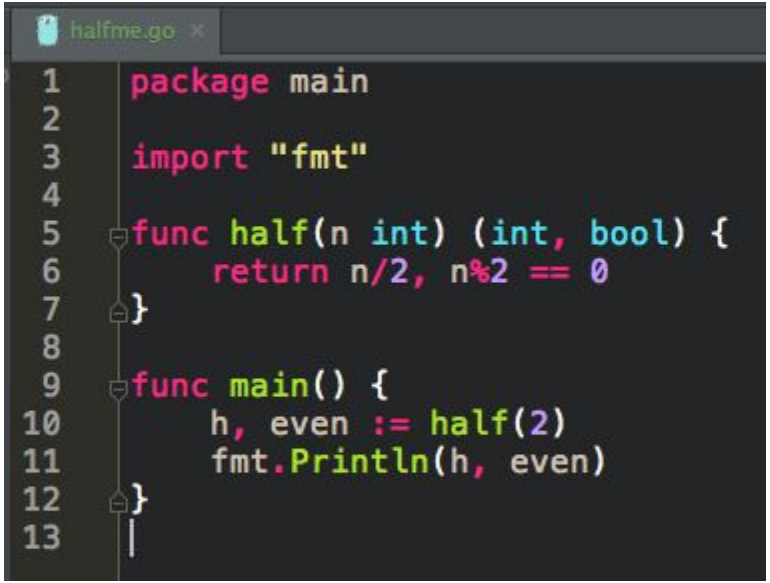
Write a function which takes an integer and returns two values:

- the integer divided by 2
- whether or not the integer is even (true, false)

For example

- half(1) should return (0, false)
- half(2) should return (1, true).

credit: http://www.golang-book.com/books/intro/7



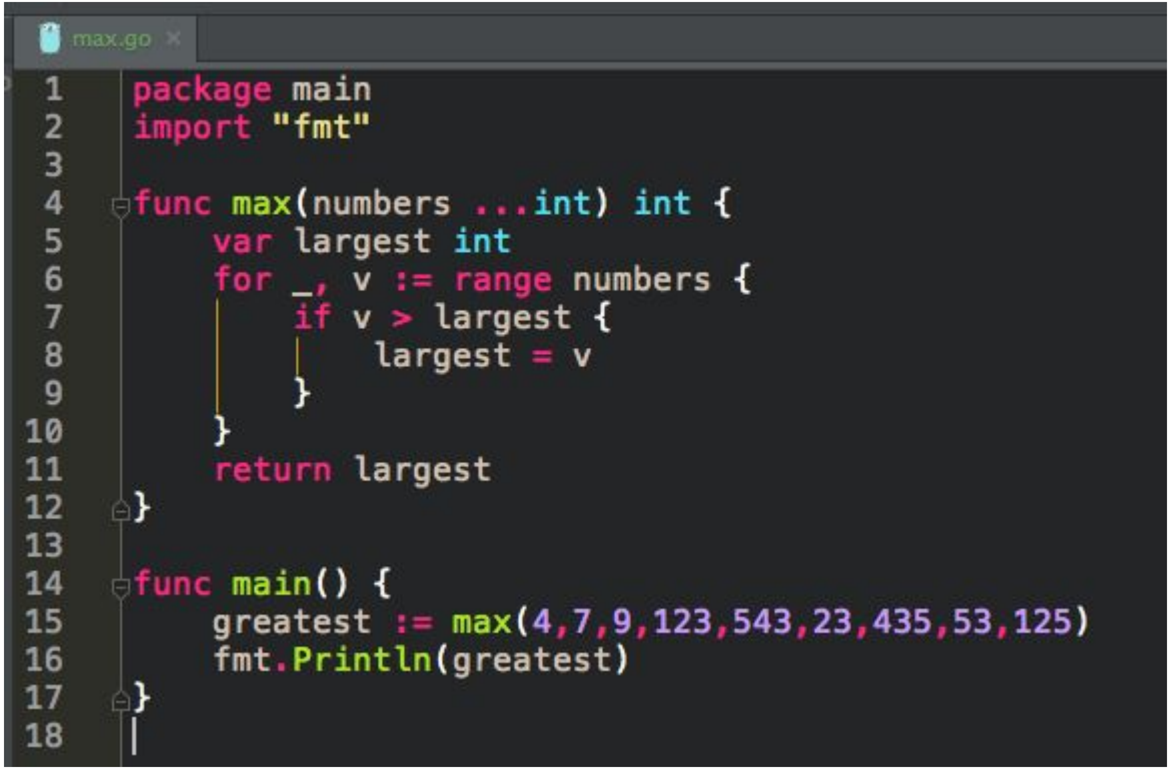
```go
package main

import "fmt"

func half(n int) (int, bool) {
	return n/2, n%2 == 0
}

func main() {
	h, even := half(2)
	fmt.Println(h, even)
}
```

solution to exercise

exercise

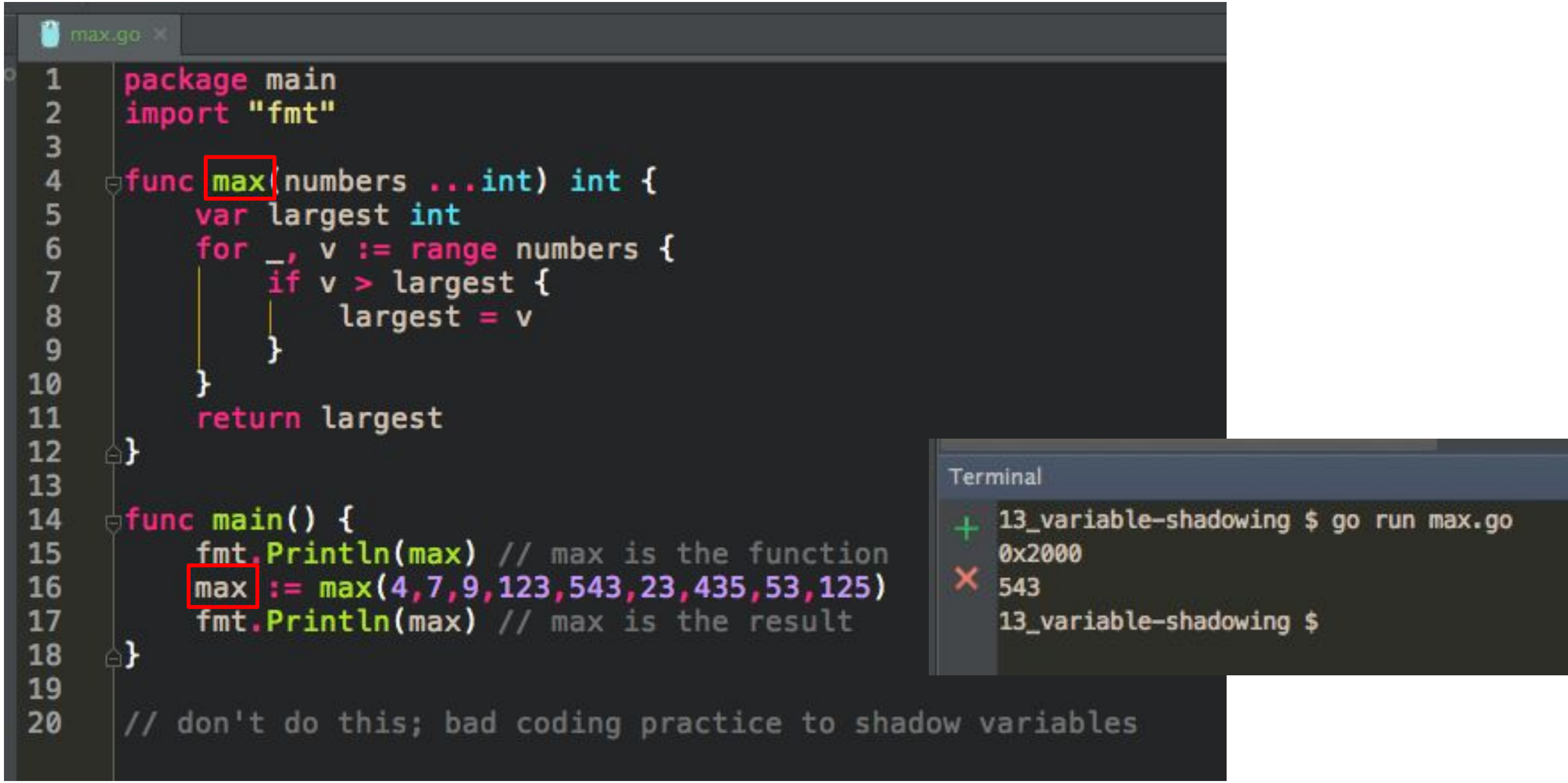
Write a function with one variadic parameter that finds the greatest number in a list of numbers.

credit: http://www.golang-book.com/books/intro/7



```go
package main
import "fmt"

func max(numbers ...int) int {
	var largest int
	for _, v := range numbers {
		if v > largest {
			largest = v
		}
	}
	return largest
}

func main() {
	greatest := max(4,7,9,123,543,23,435,53,125)
	fmt.Println(greatest)
}
```

solution to exercise

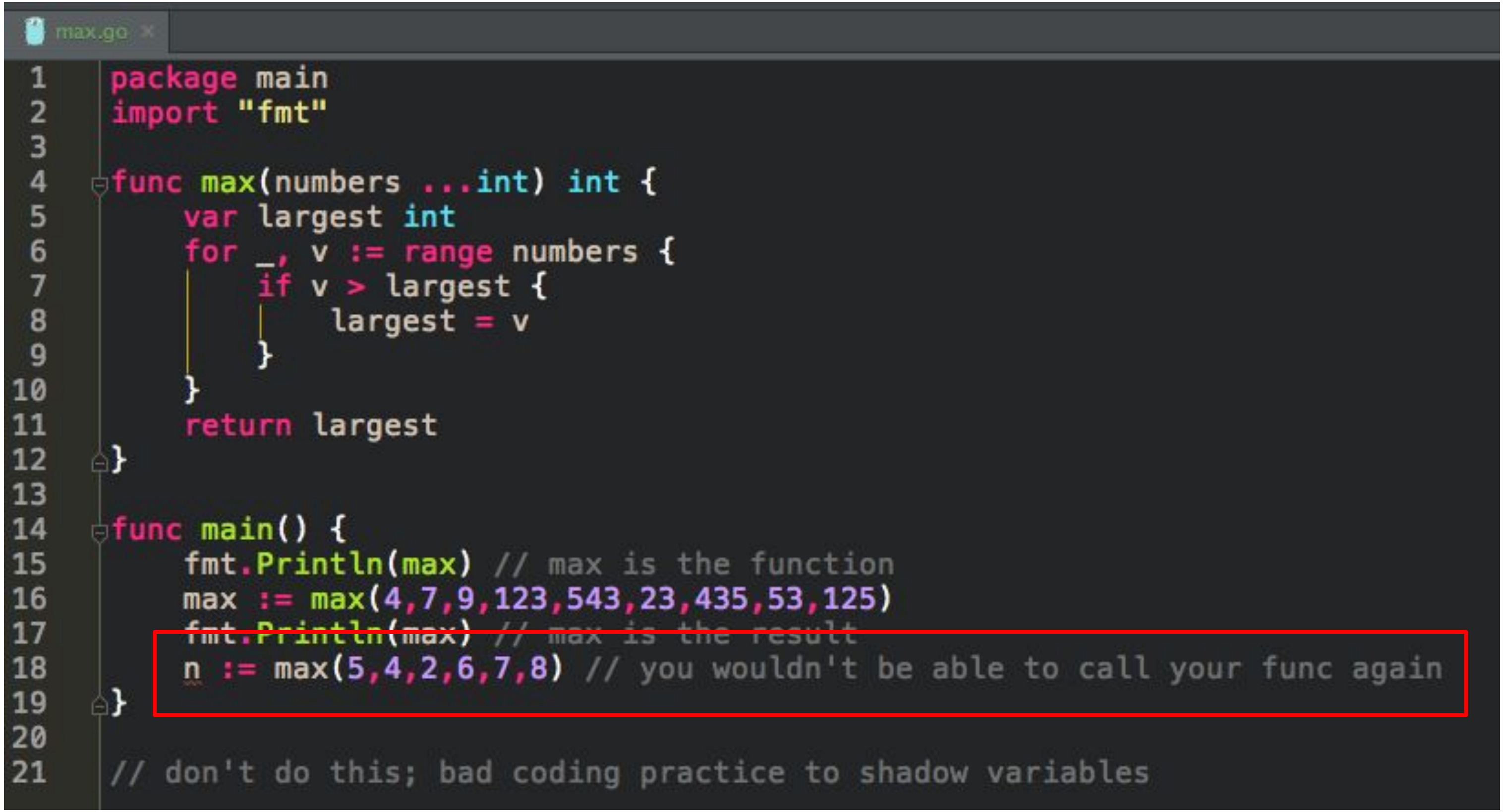


```go
package main
import "fmt"

func max(numbers ...int) int {
    var largest int
    for _, v := range numbers {
        if v > largest {
            largest = v
        }
    }
    return largest
}

func main() {
    fmt.Println(max) // max is the function
    max := max(4,7,9,123,543,23,435,53,125)
    fmt.Println(max) // max is the result
}

// don't do this; bad coding practice to shadow variables
```

```
13_variable-shadowing $ go run max.go
0x2000
543
13_variable-shadowing $
```

bad coding practice

variable shadowing



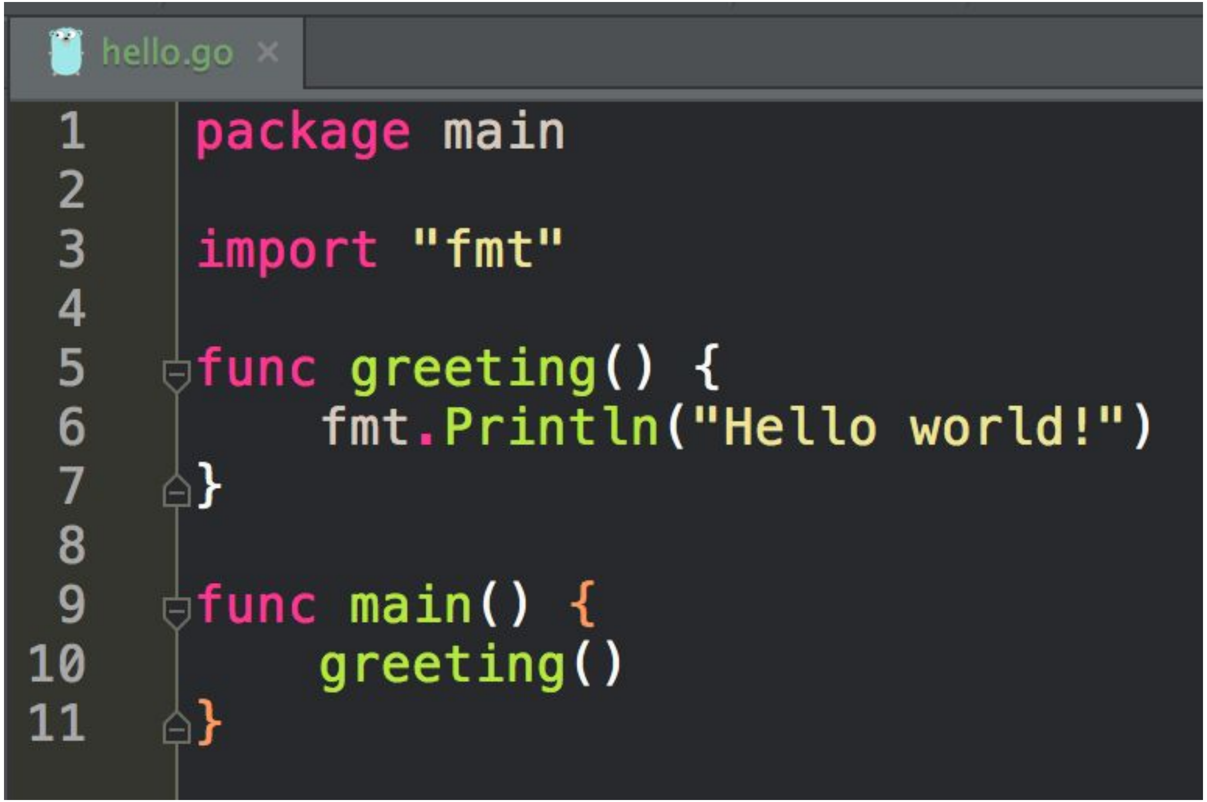
```go
package main
import "fmt"

func max(numbers ...int) int {
	var largest int
	for _, v := range numbers {
		if v > largest {
			largest = v
		}
	}
	return largest
}

func main() {
	fmt.Println(max) // max is the function
	max := max(4,7,9,123,543,23,435,53,125)
	fmt.Println(max) // max is the result
	n := max(5,4,2,6,7,8) // you wouldn't be able to call your func again
}

// don't do this; bad coding practice to shadow variables
```

bad coding practice

variable shadowing

func expression

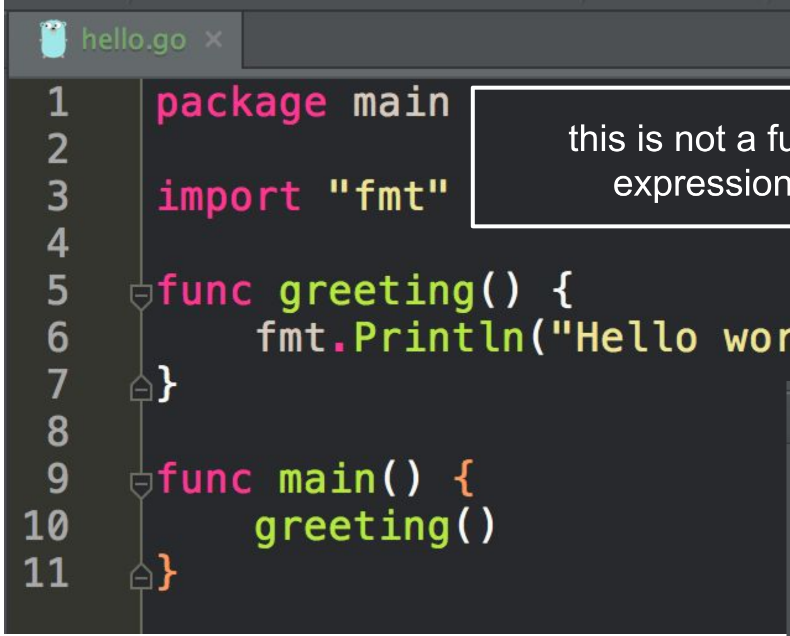
setting a variable equal to a function



```go
package main

import "fmt"

func greeting() {
	fmt.Println("Hello world!")
}

func main() {
	greeting()
}
```

this is not a func expression

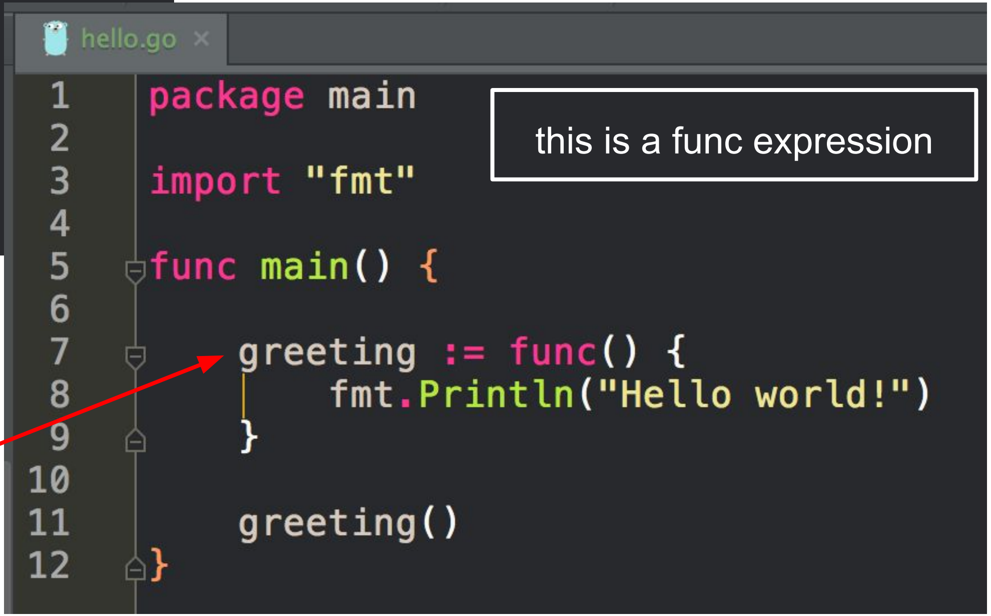
this is our code before using a func expression



hello.go

```go
package main

import "fmt"

func greeting() {
	fmt.Println("Hello world!")
}

func main() {
	greeting()
}
```

this is not a func expression



hello.go

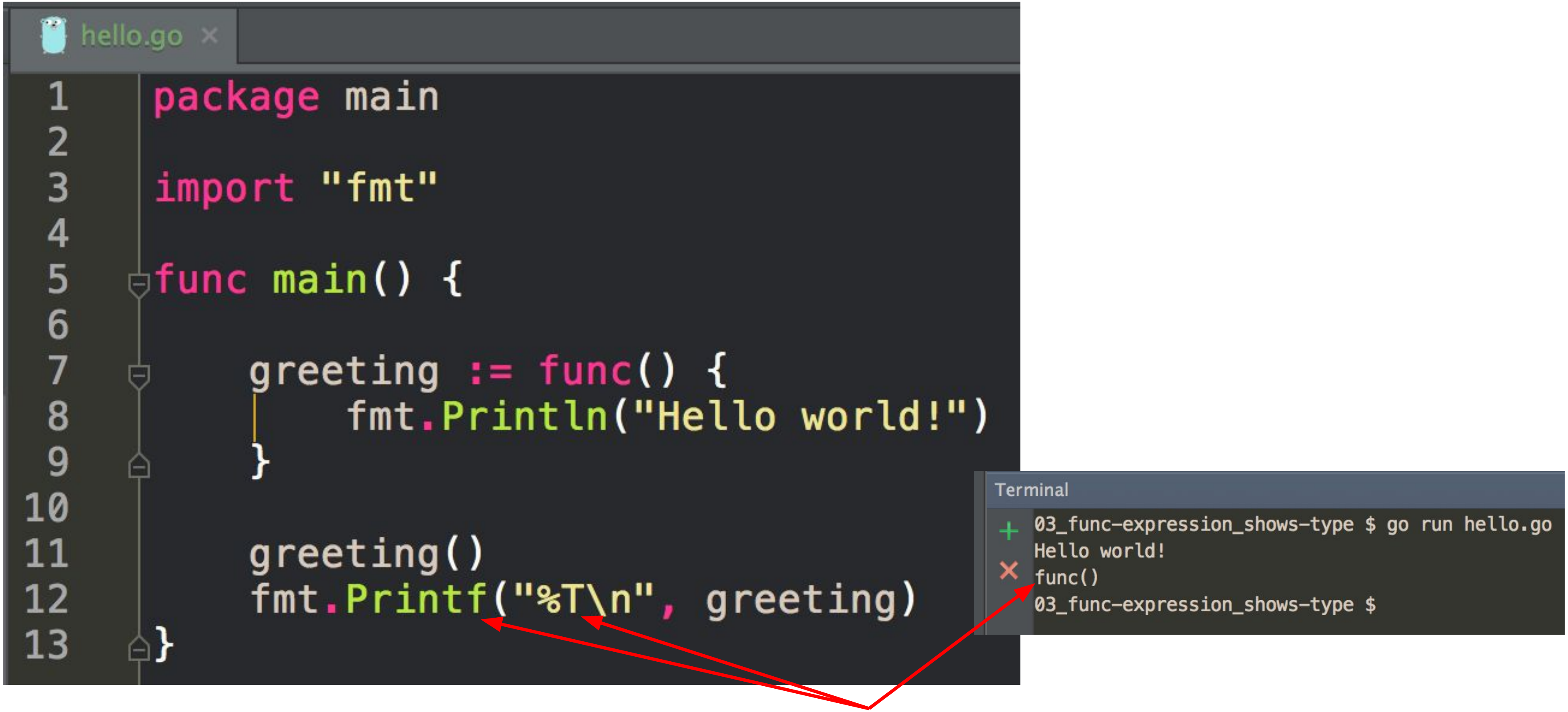
```go
package main

import "fmt"

func main() {

	greeting := func() {
		fmt.Println("Hello world!")
	}

	greeting()
}
```

this is a func expression

func expression

setting a variable equal to a func

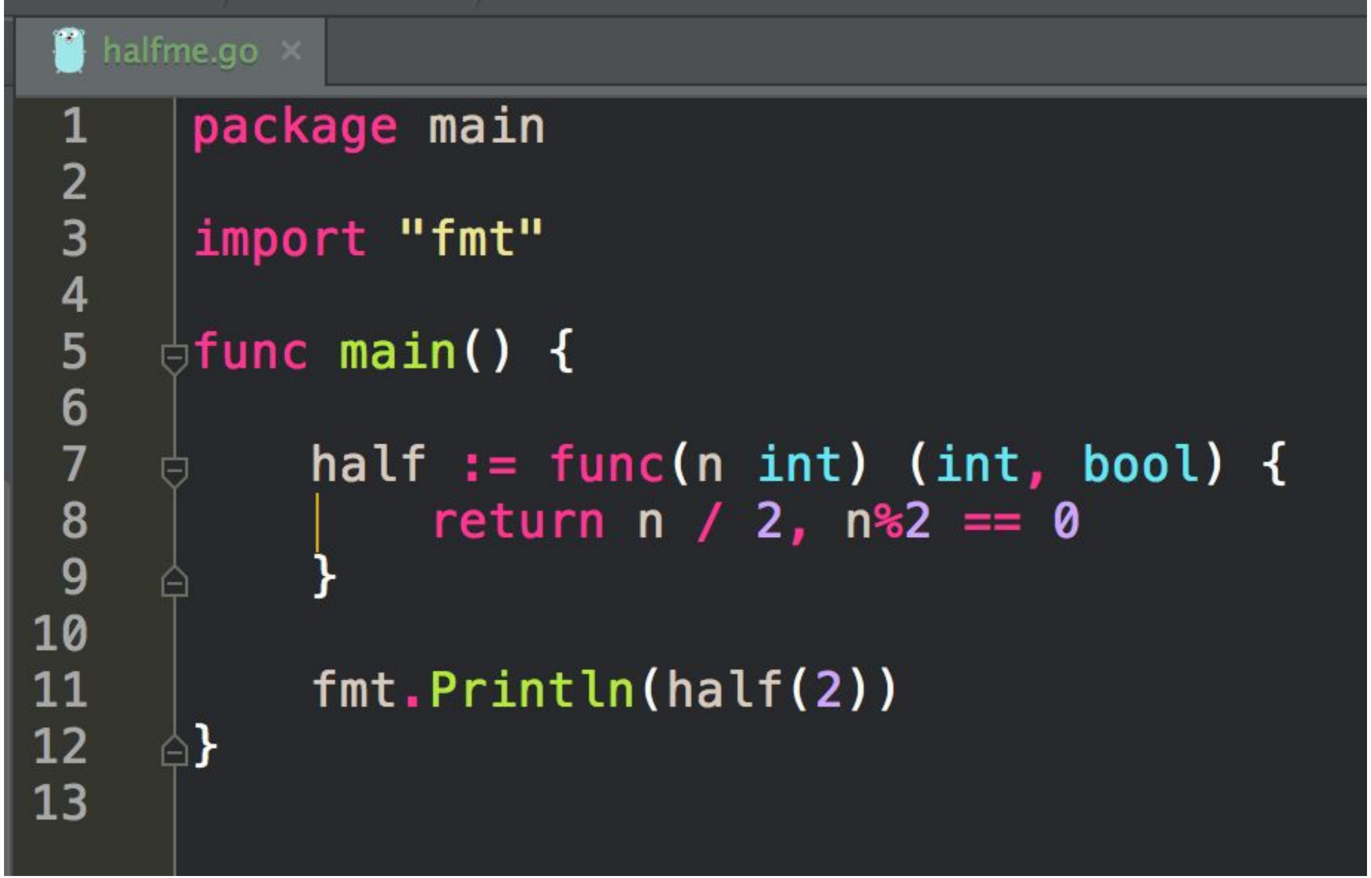
the **scope** of **greeting** is func main()



```go
package main

import "fmt"

func main() {

	greeting := func() {
		fmt.Println("Hello world!")
	}

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

```
03_func-expression_shows-type $ go run hello.go
Hello world!
func()
03_func-expression_shows-type $
```

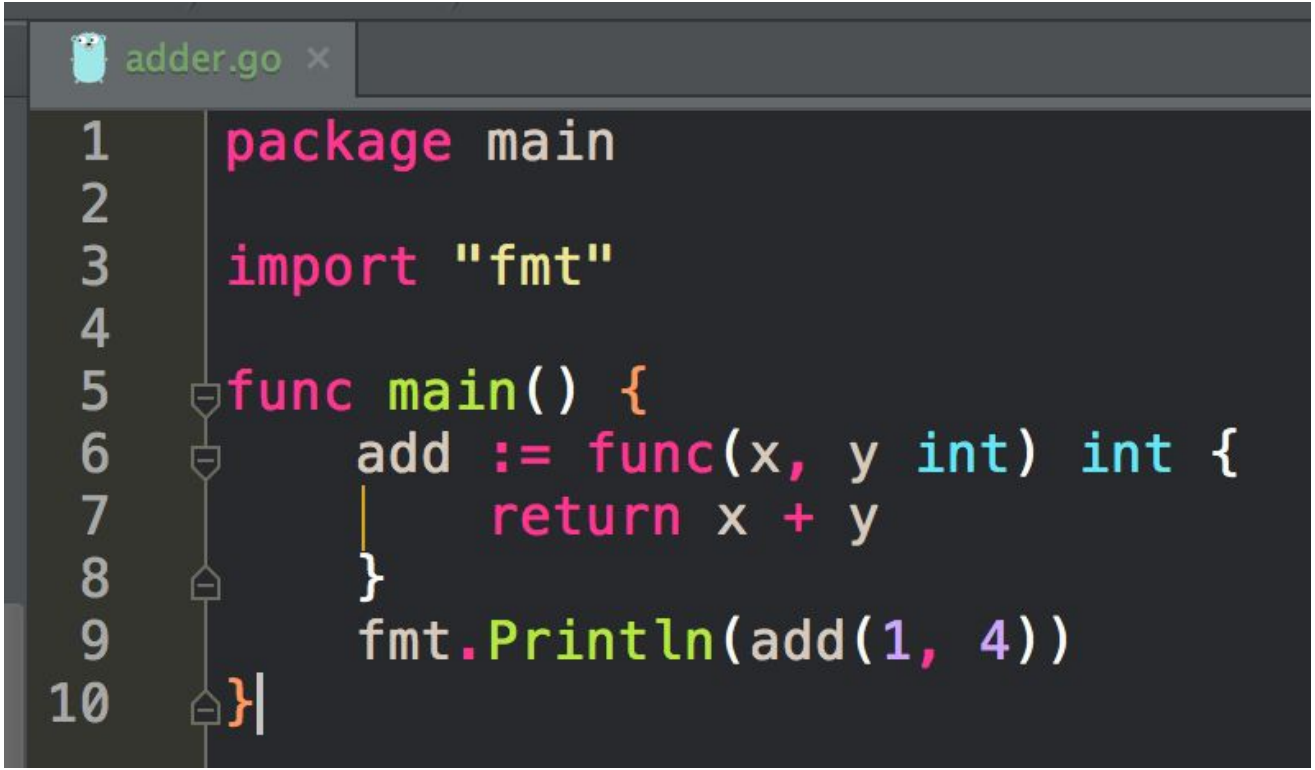
interesting to look at greeting’s type



```go
package main

import "fmt"

func main() {

	half := func(n int) (int, bool) {
		return n / 2, n%2 == 0
	}

	fmt.Println(half(2))
}

```

another func expression

setting a variable equal to a func



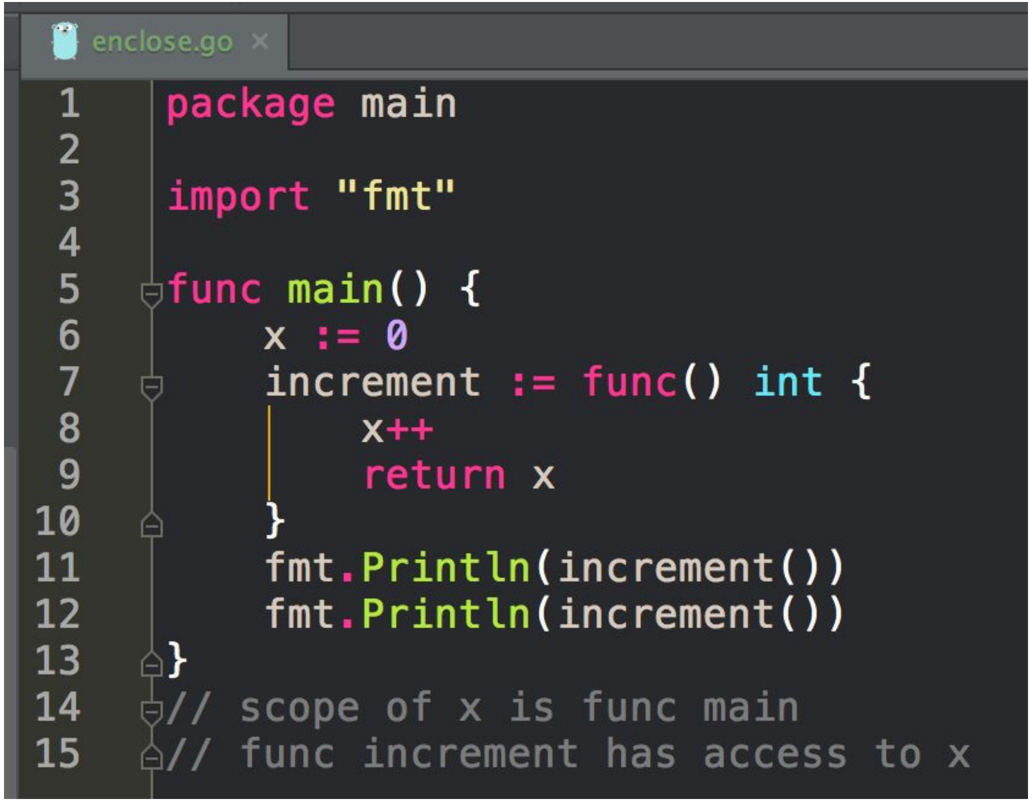
```go
package main

import "fmt"

func main() {
	add := func(x, y int) int {
		return x + y
	}
	fmt.Println(add(1, 4))
}
```

another func expression

setting a variable equal to a func

closure

my definition: *“one thing enclosing another thing”*



```go
package main

import "fmt"

func main() {
	x := 0
	increment := func() int {
		x++
		return x
	}
	fmt.Println(increment())
	fmt.Println(increment())
}
// scope of x is func main
// func increment has access to x
```

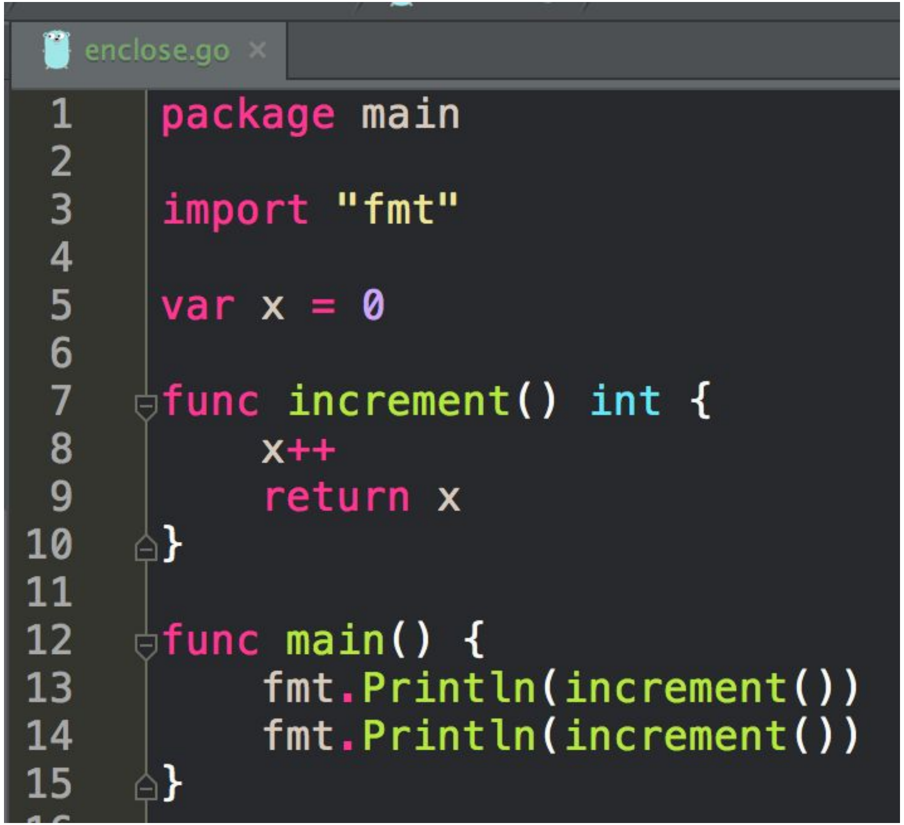
closure

func main encloses func increment

closure helps us limit the scope of variables that are used by multiple functions

without closure, for two or more funcs to have access to the same variable, that variable would need to be package scope

func main is enclosing increment; increment is enclosing x

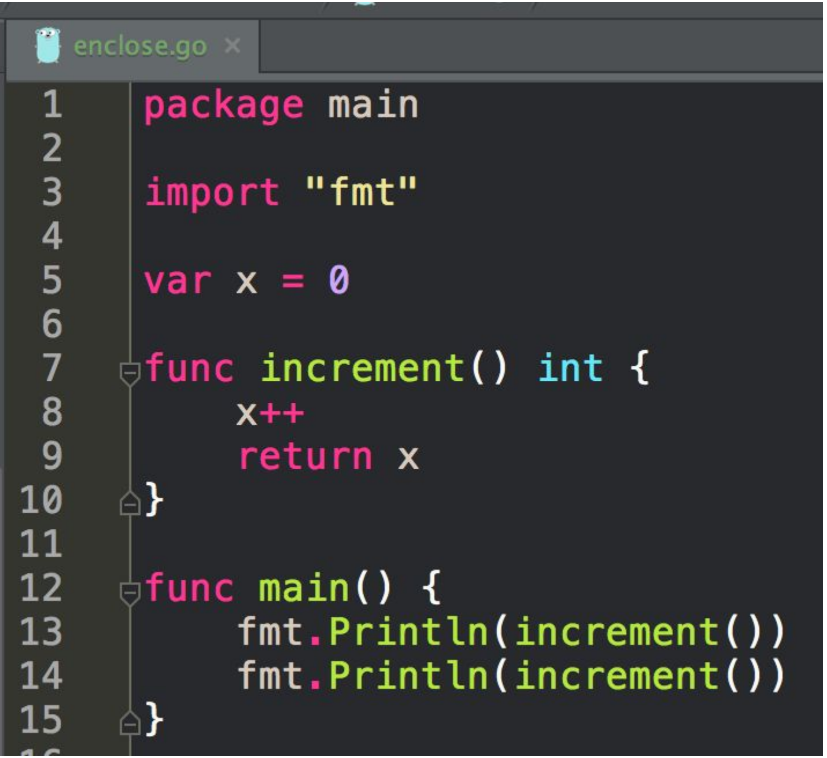


```go
package main

import "fmt"

var x = 0

func increment() int {
	x++
	return x
}

func main() {
	fmt.Println(increment())
	fmt.Println(increment())
}
```

not using closure

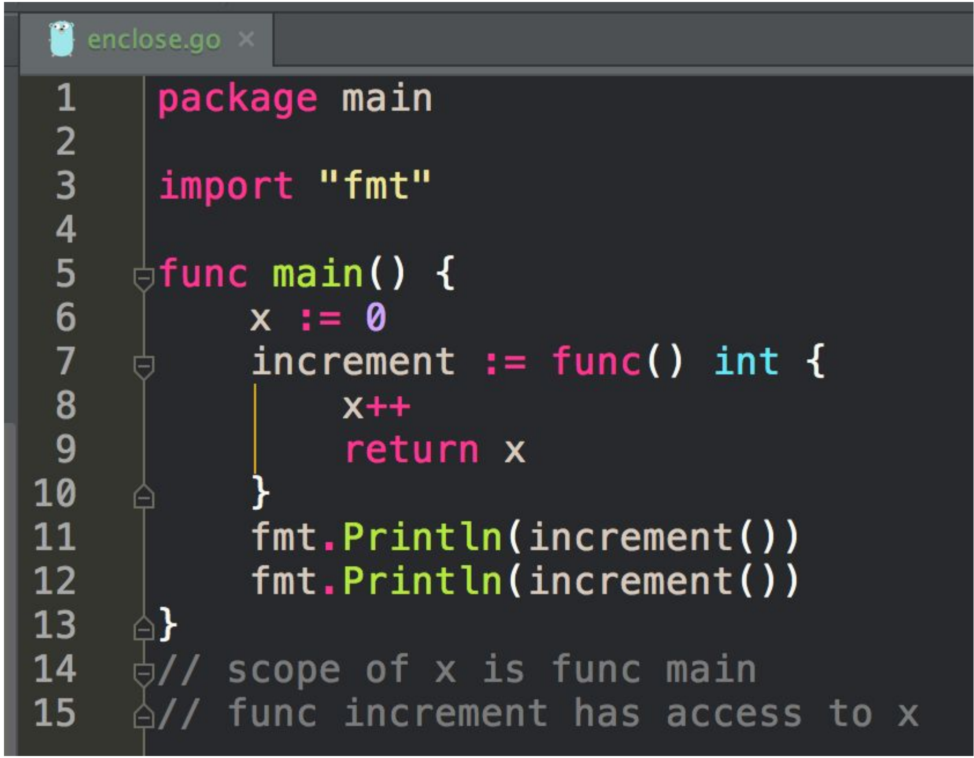
closure helps us limit the scope of variables that are used by multiple functions

without closure, for two or more funcs to have access to the same variable, that variable would need to be package scope



```go
package main

import "fmt"

var x = 0

func increment() int {
	x++
	return x
}

func main() {
	fmt.Println(increment())
	fmt.Println(increment())
}
```

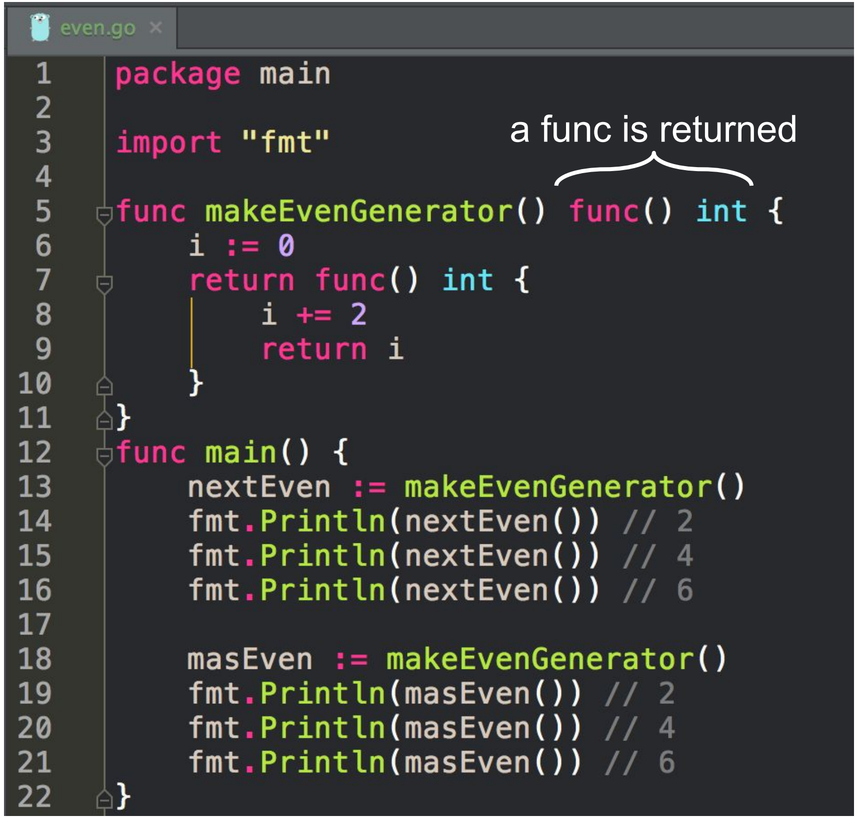
not using closure



```go
package main

import "fmt"

func main() {
	x := 0
	increment := func() int {
		x++
		return x
	}
	fmt.Println(increment())
	fmt.Println(increment())
}
// scope of x is func main
// func increment has access to x
```

closure

returning a func



a func is returned

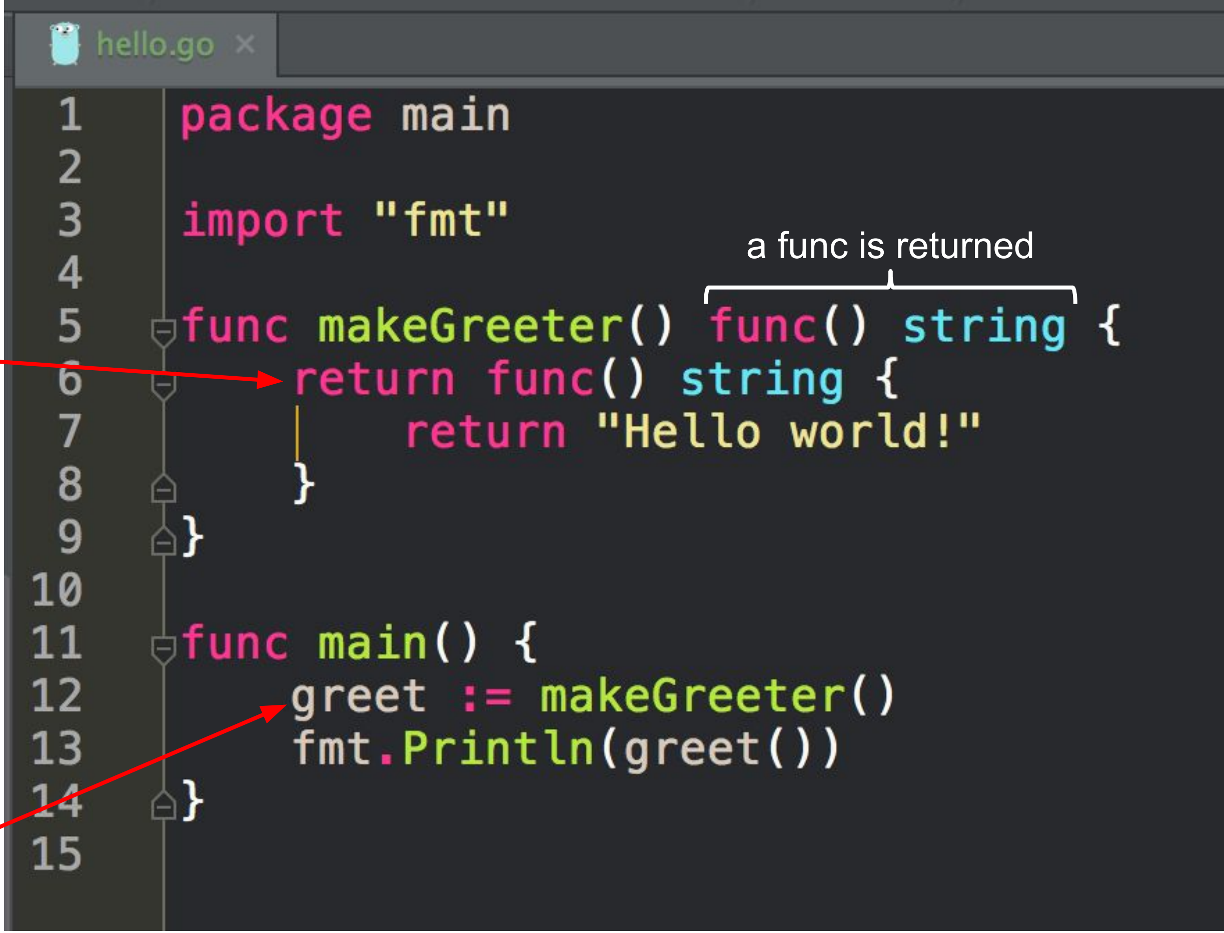
```go
package main

import "fmt"

func makeEvenGenerator() func() int {
    i := 0
    return func() int {
        i += 2
        return i
    }
}
func main() {
    nextEven := makeEvenGenerator()
    fmt.Println(nextEven()) // 2
    fmt.Println(nextEven()) // 4
    fmt.Println(nextEven()) // 6

    masEven := makeEvenGenerator()
    fmt.Println(masEven()) // 2
    fmt.Println(masEven()) // 4
    fmt.Println(masEven()) // 6
}
```

closure

closure helps us limit the scope of variables that are used by multiple functions

without closure, for two or more funcs to have access to the same variable, that variable would need to be package scope

nextEven & masEven are each holding/enclosing the variable i



```go
package main

import "fmt"

func makeGreeter() func() string {
	return func() string {
		return "Hello world!"
	}
}

func main() {
	greet := makeGreeter()
	fmt.Println(greet())
}

```

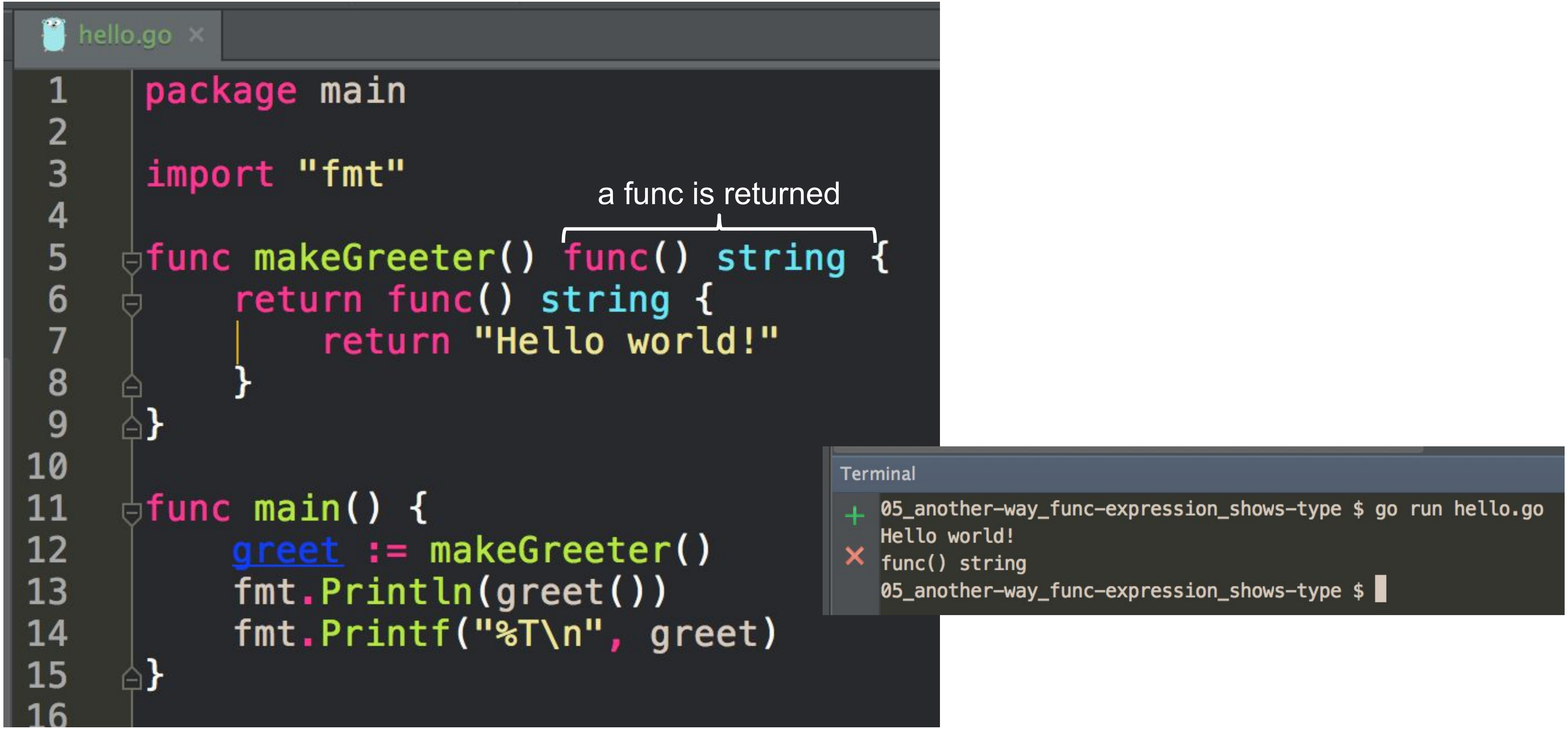
a func is returned

returning a func

(not part of func expression)

another func expression

setting a variable equal to a func



a func is returned

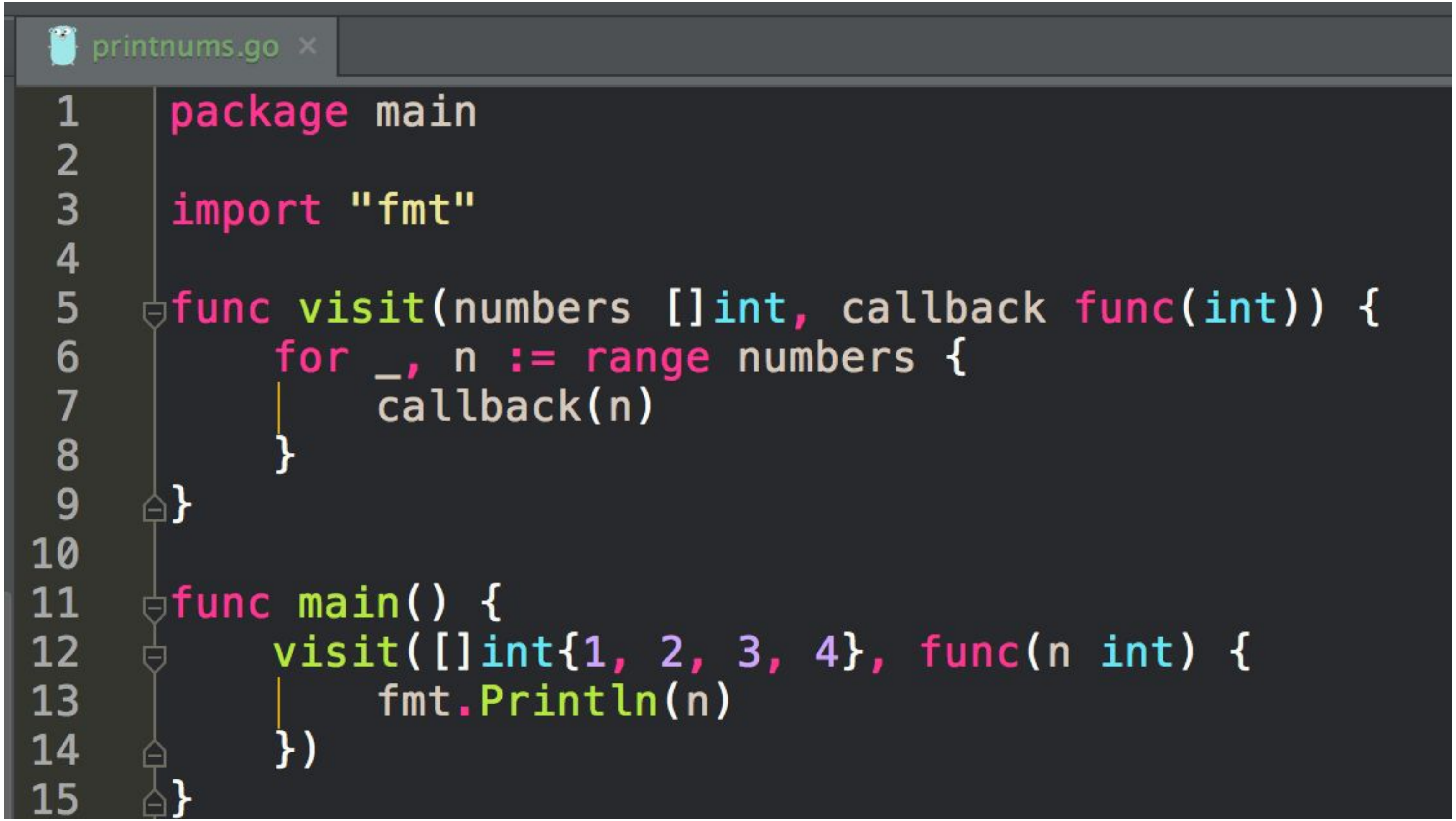
```go
package main

import "fmt"

func makeGreeter() func() string {
	return func() string {
		return "Hello world!"
	}
}

func main() {
	greet := makeGreeter()
	fmt.Println(greet())
	fmt.Printf("%T\n", greet)
}
```

```
05_another-way_func-expression_shows-type $ go run hello.go
Hello world!
func() string
05_another-way_func-expression_shows-type $
```

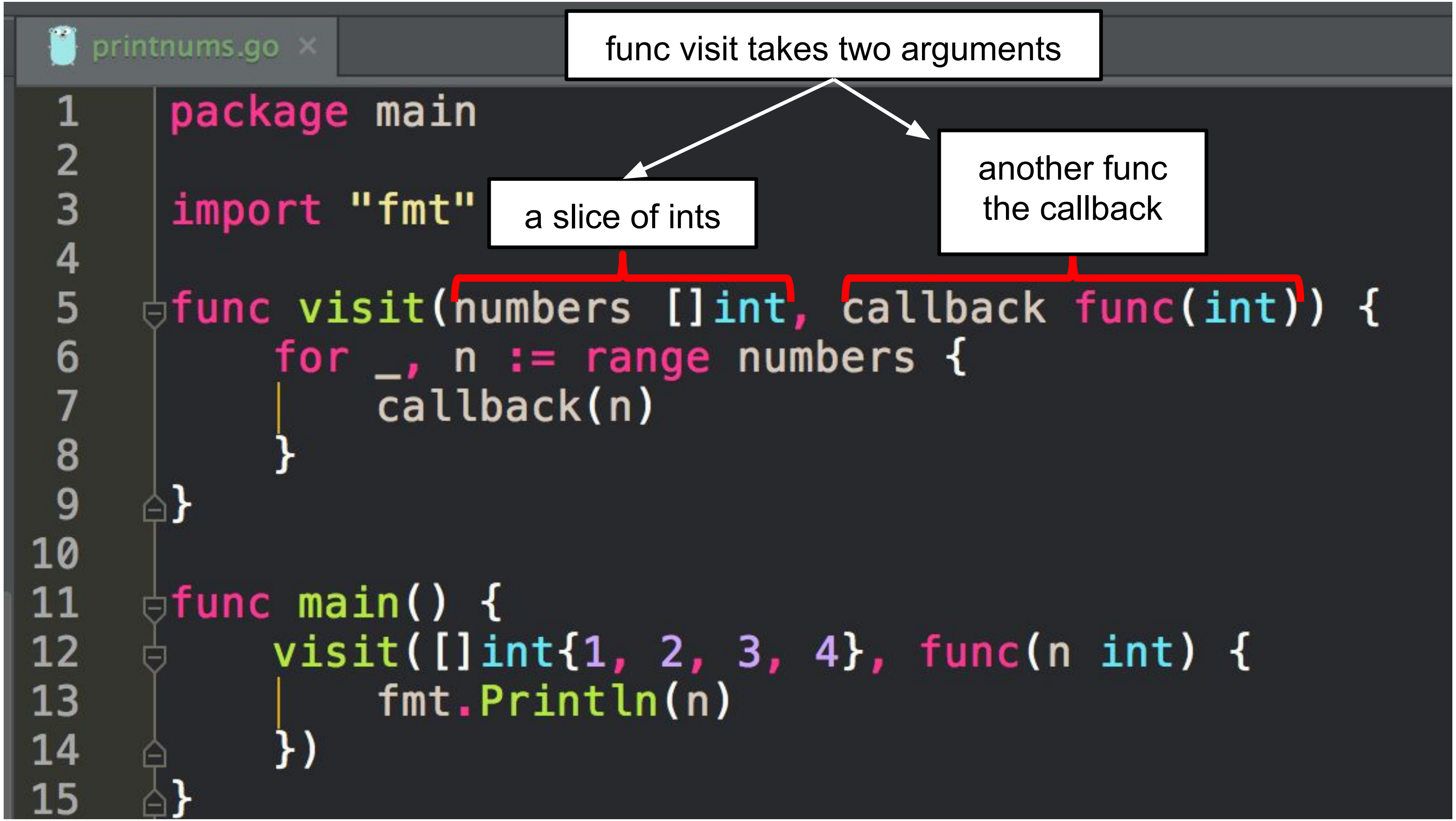
interesting to look at greet’s type

callback

passing a func as an argument



```go
package main

import "fmt"

func visit(numbers []int, callback func(int)) {
	for _, n := range numbers {
		callback(n)
	}
}

func main() {
	visit([]int{1, 2, 3, 4}, func(n int) {
		fmt.Println(n)
	})
}
```

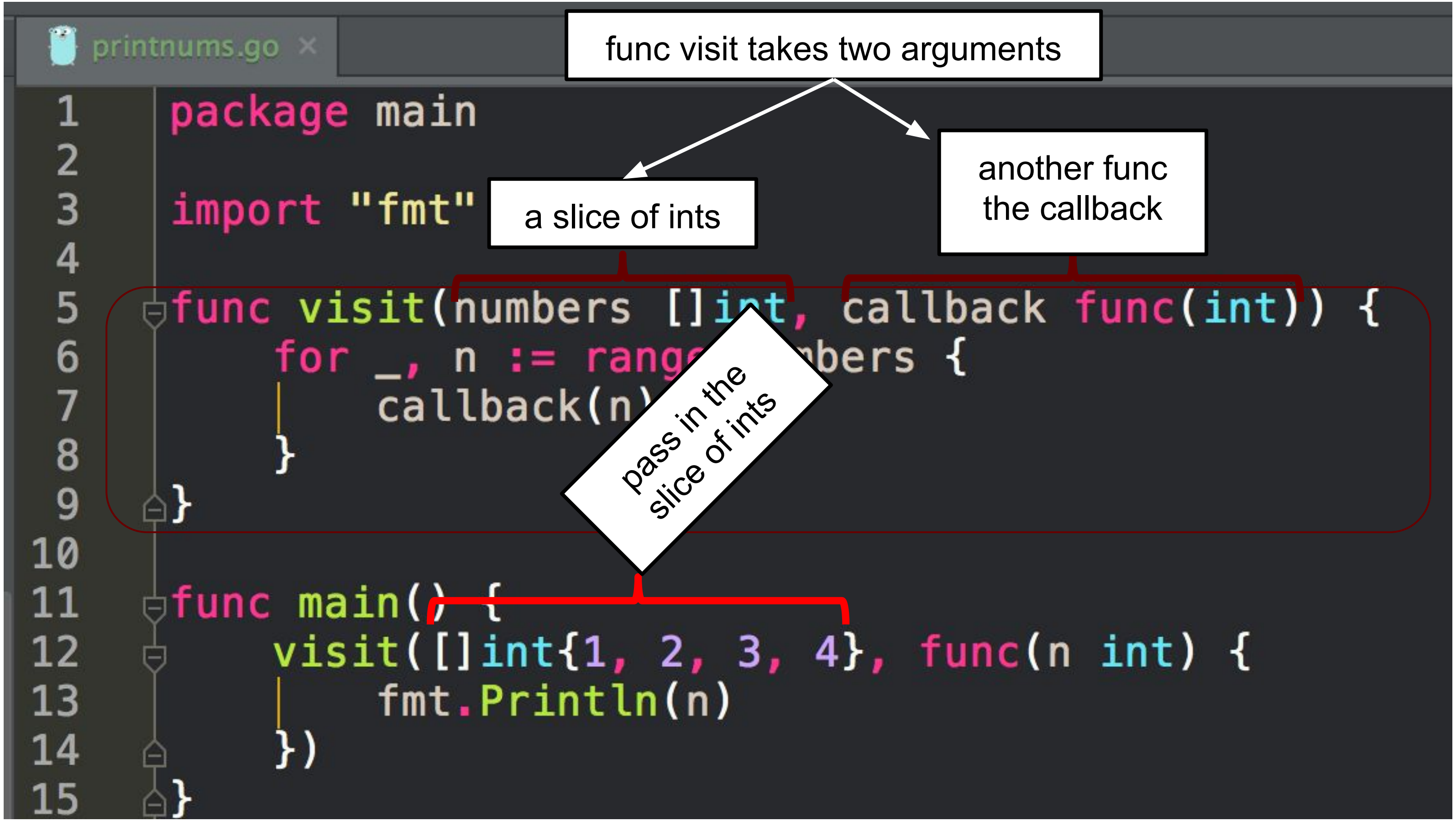


func visit takes two arguments

a slice of ints

another func
the callback

```go
package main

import "fmt"

func visit(numbers []int, callback func(int)) {
    for _, n := range numbers {
        callback(n)
    }
}

func main() {
    visit([]int{1, 2, 3, 4}, func(n int) {
        fmt.Println(n)
    })
}
```



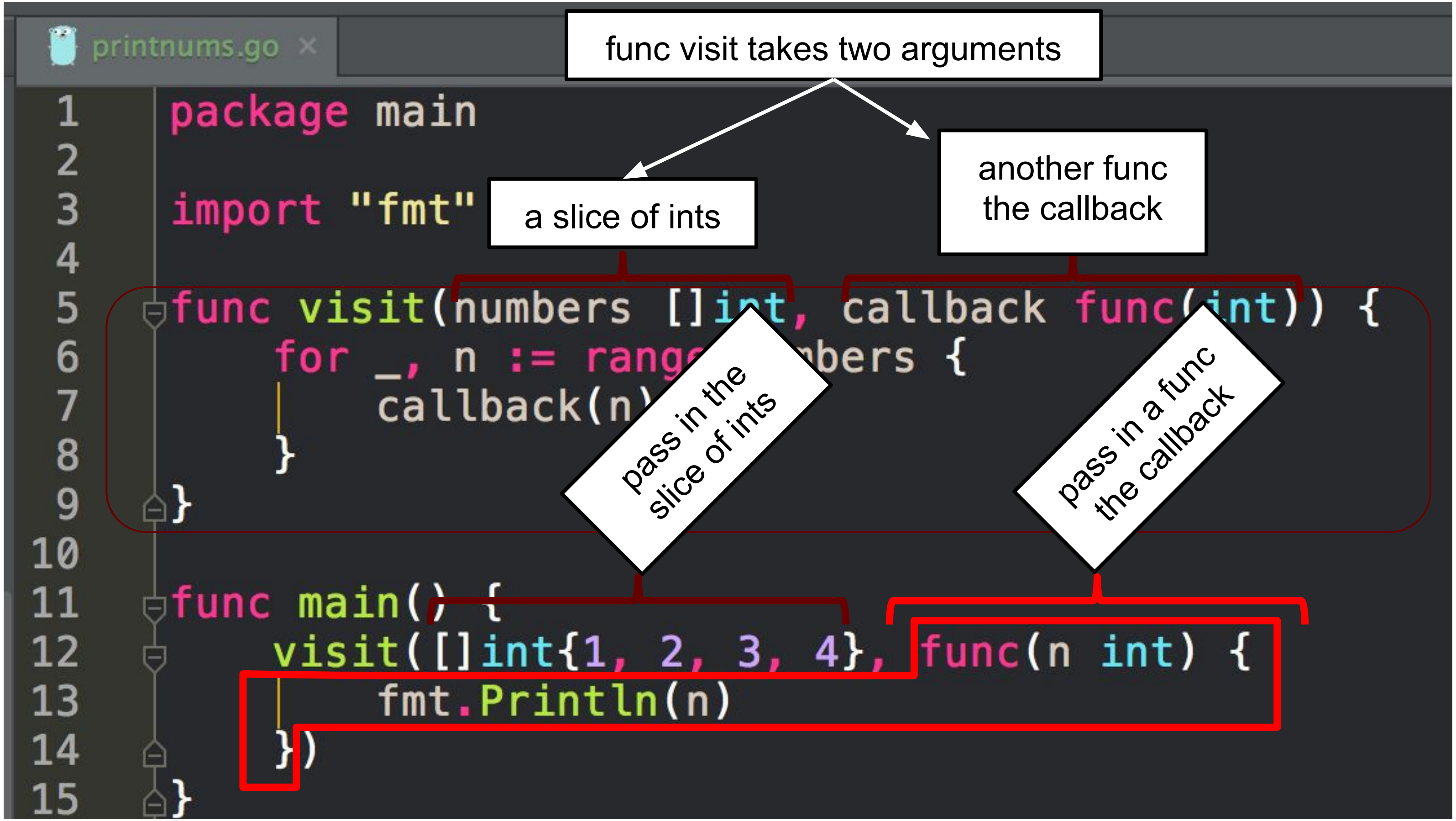
```go
package main

import "fmt"

func visit(numbers []int, callback func(int)) {
	for _, n := range numbers {
		callback(n)
	}
}

func main() {
	visit([]int{1, 2, 3, 4}, func(n int) {
		fmt.Println(n)
	})
}
```

func visit takes two arguments

a slice of ints

another func
the callback

pass in the
slice of ints



```go
package main

import "fmt"

func visit(numbers []int, callback func(int)) {
	for _, n := range numbers {
		callback(n)
	}
}

func main() {
	visit([]int{1, 2, 3, 4}, func(n int) {
		fmt.Println(n)
	})
}
```

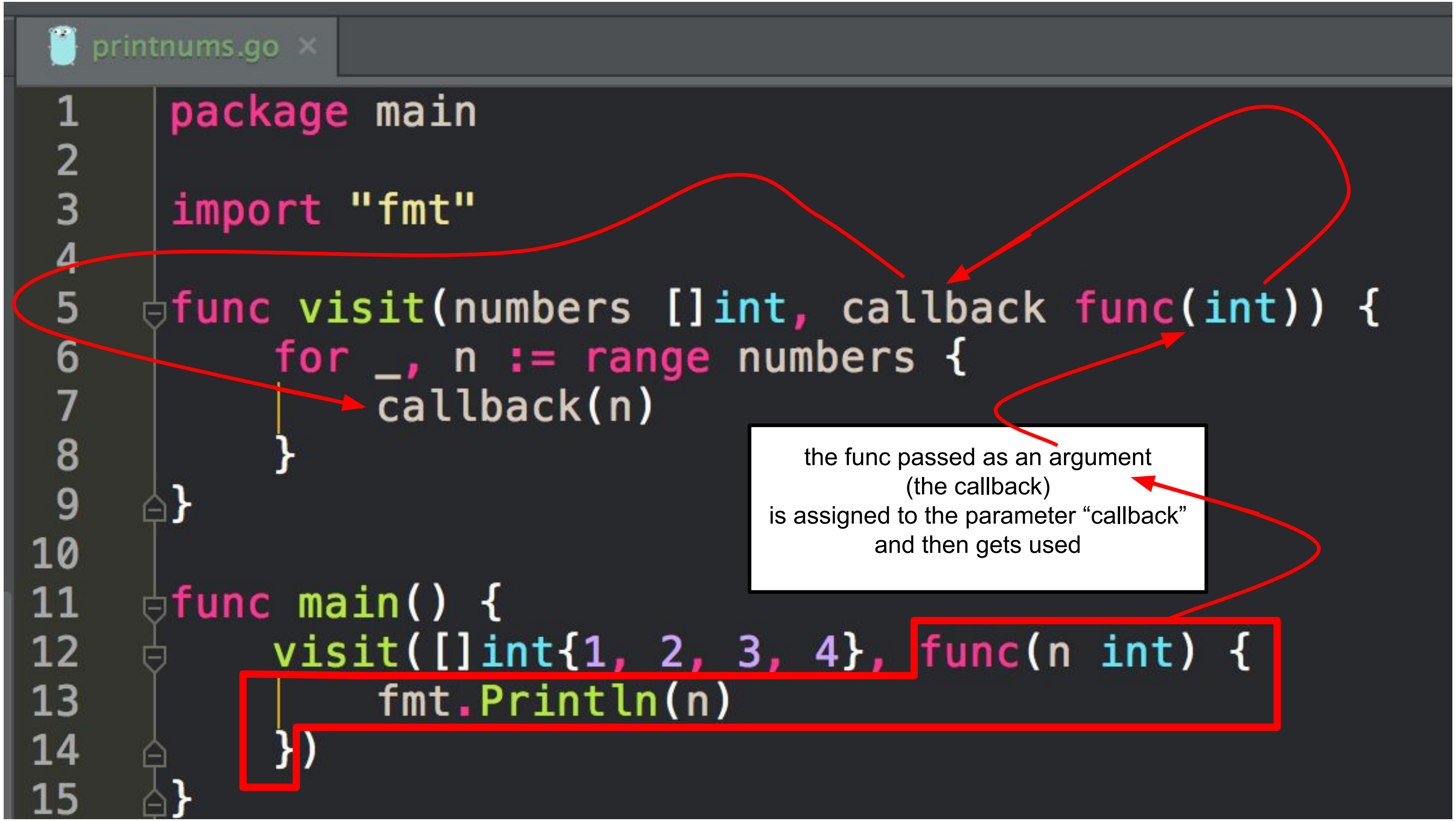
func visit takes two arguments

a slice of ints

another func
the callback

pass in the
slice of ints

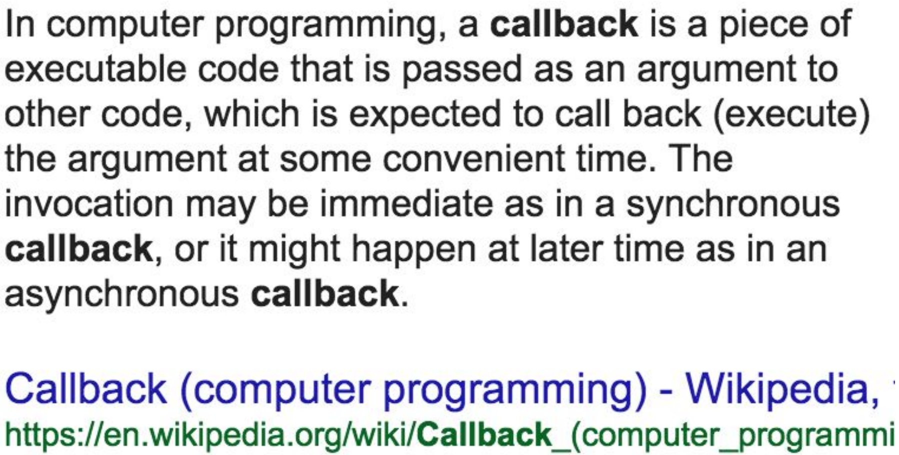
pass in a func
the callback



```go
package main

import "fmt"

func visit(numbers []int, callback func(int)) {
    for _, n := range numbers {
        callback(n)
    }
}

func main() {
    visit([]int{1, 2, 3, 4}, func(n int) {
        fmt.Println(n)
    })
}
```

the func passed as an argument
(the callback)
is assigned to the parameter “callback”
and then gets used

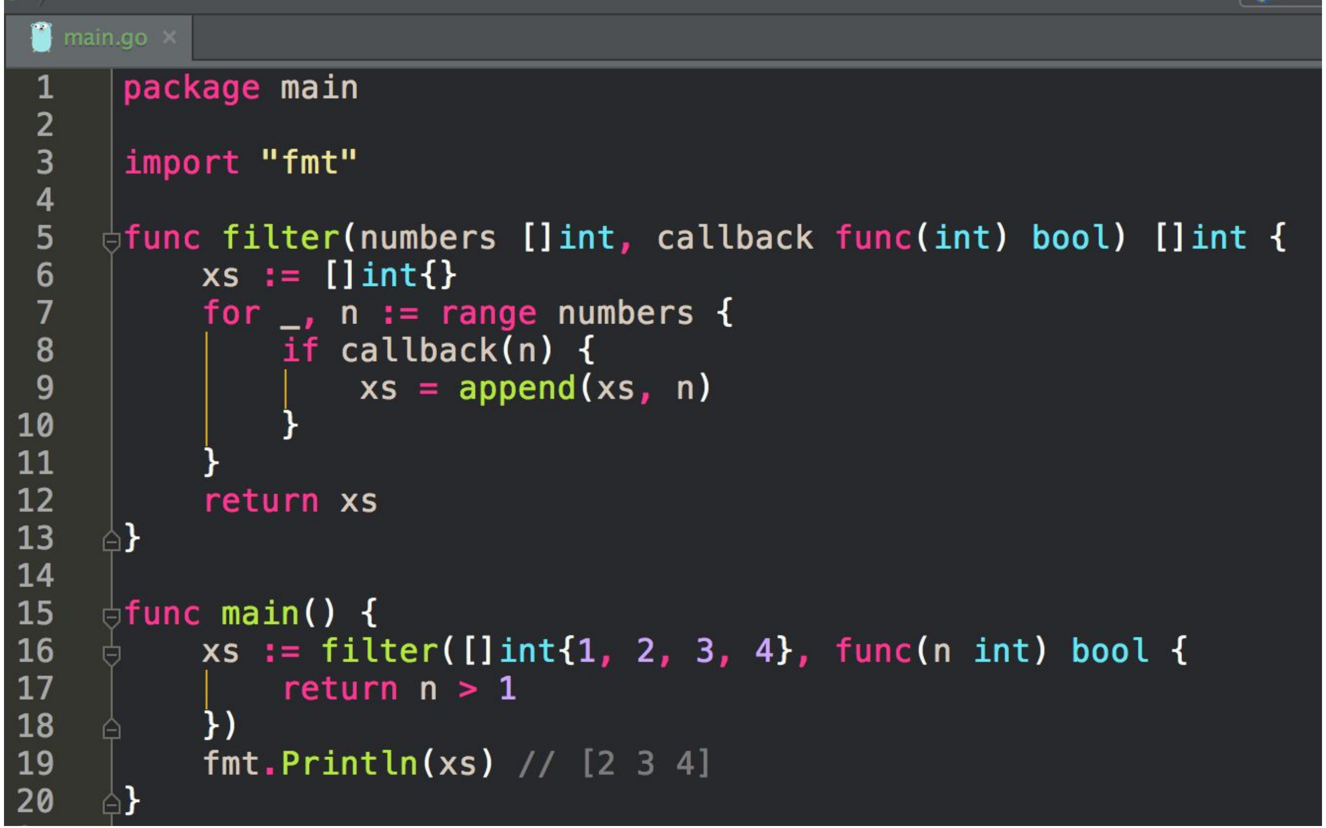
wikipedia’s description



In computer programming, a **callback** is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time. The invocation may be immediate as in a synchronous **callback**, or it might happen at later time as in an asynchronous **callback**.

Callback (computer programming) - Wikipedia,

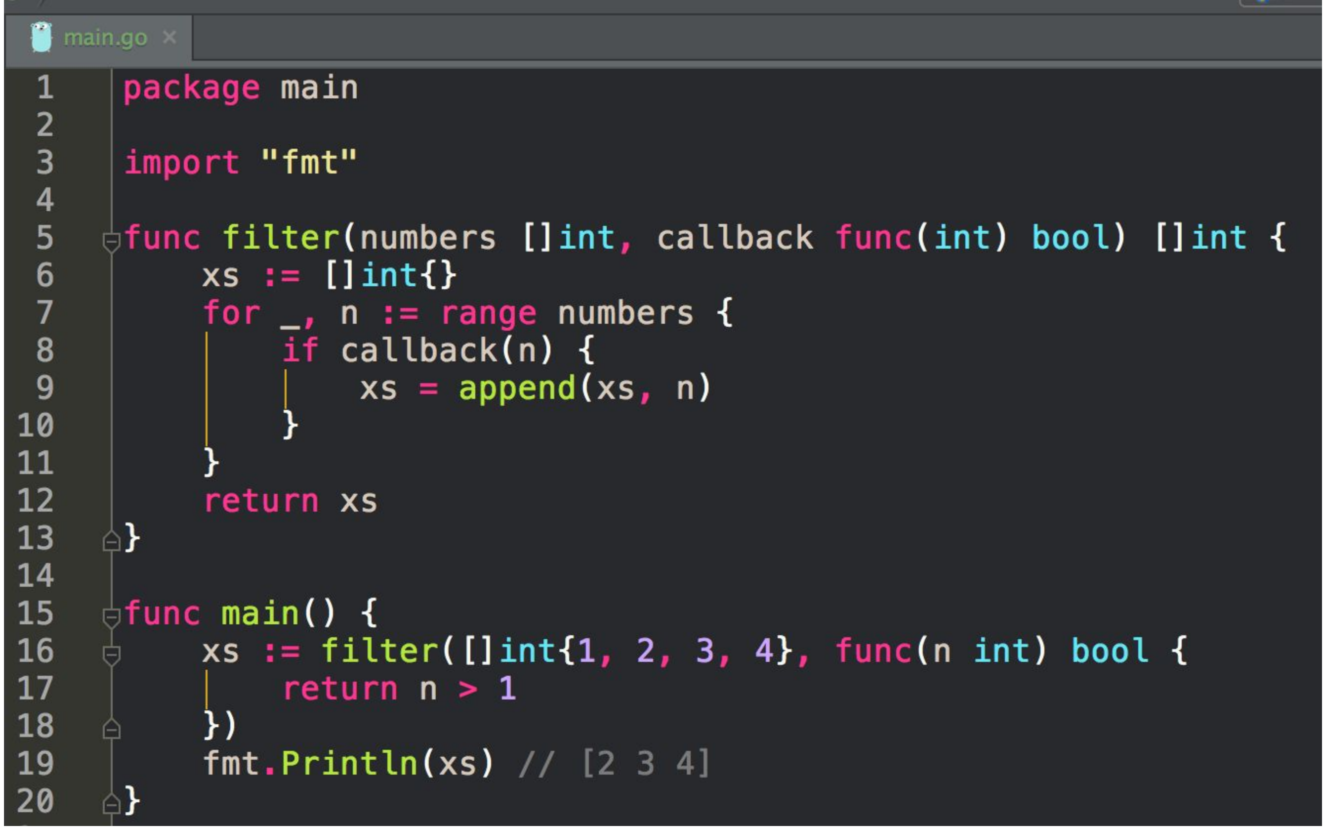
https://en.wikipedia.org/wiki/**Callback**_(computer_programmi



```go
package main

import "fmt"

func filter(numbers []int, callback func(int) bool) []int {
	xs := []int{}
	for _, n := range numbers {
		if callback(n) {
			xs = append(xs, n)
		}
	}
	return xs
}

func main() {
	xs := filter([]int{1, 2, 3, 4}, func(n int) bool {
		return n > 1
	})
	fmt.Println(xs) // [2 3 4]
}
```

another callback

can you explain this code?



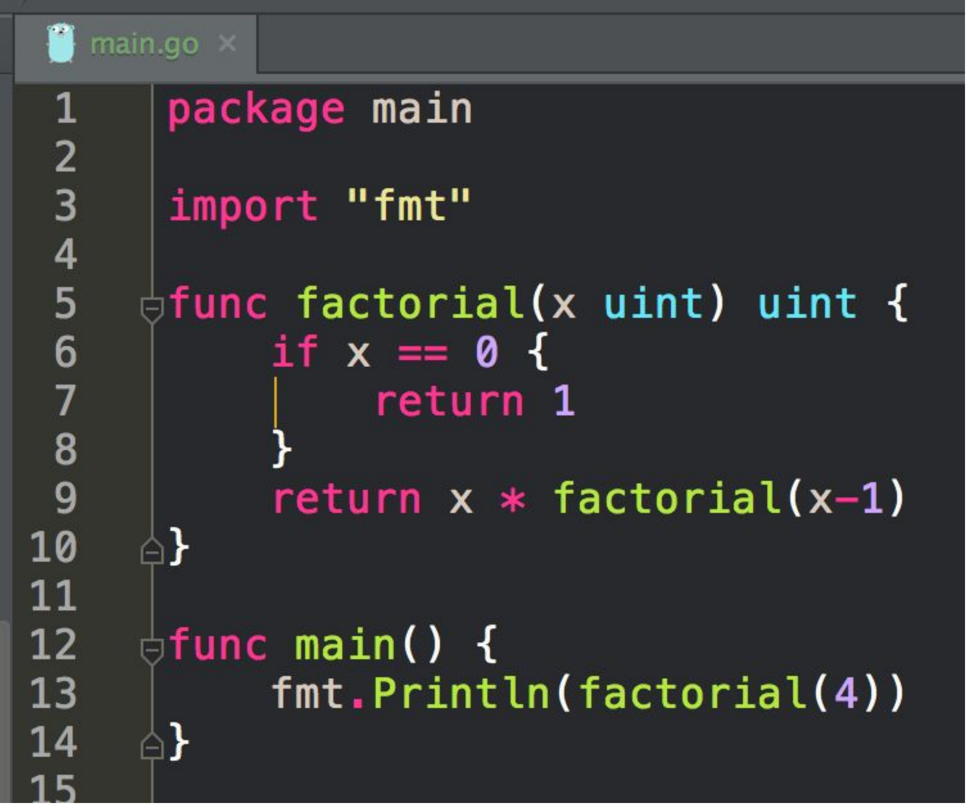
```go
package main

import "fmt"

func filter(numbers []int, callback func(int) bool) []int {
	xs := []int{}
	for _, n := range numbers {
		if callback(n) {
			xs = append(xs, n)
		}
	}
	return xs
}

func main() {
	xs := filter([]int{1, 2, 3, 4}, func(n int) bool {
		return n > 1
	})
	fmt.Println(xs) // [2 3 4]
}
```

“If you’ve done functional programming like Lisp or Haskell, this way of dealing with functions is super common; it’s an approach to development; you get used to passing functions around. Go allows you to do that [passing functions around] but it’s not the most common way of writing code. The more normal way you’d write code [for something like the code above] would just be a simple for loop. For loops are easy to understand.”

~ Caleb Doxsey

recursion

a func that can call itself



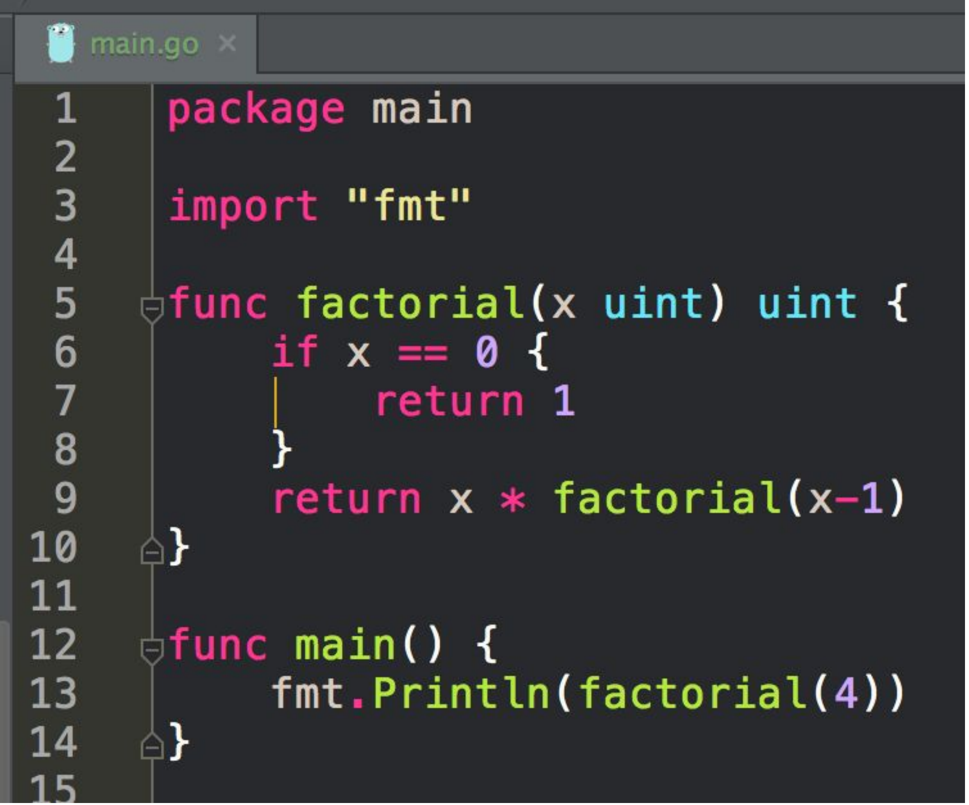
```go
package main

import "fmt"

func factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * factorial(x-1)
}

func main() {
    fmt.Println(factorial(4))
}
```

The End Result:

- 24

Can you pencil out how the answer, 24, was reached?

recursion



```go
package main

import "fmt"

func factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * factorial(x-1)
}

func main() {
    fmt.Println(factorial(4))
}
```

- factorial(4)
    - returns: 4 * factorial(3)
- factorial(3)
    - returns: 3 * factorial(2)
- factorial(2)
    - returns: 2 * factorial(1)
- factorial(1)
    - returns: 1 * factorial(0)
- factorial(0)
    - returns: 1

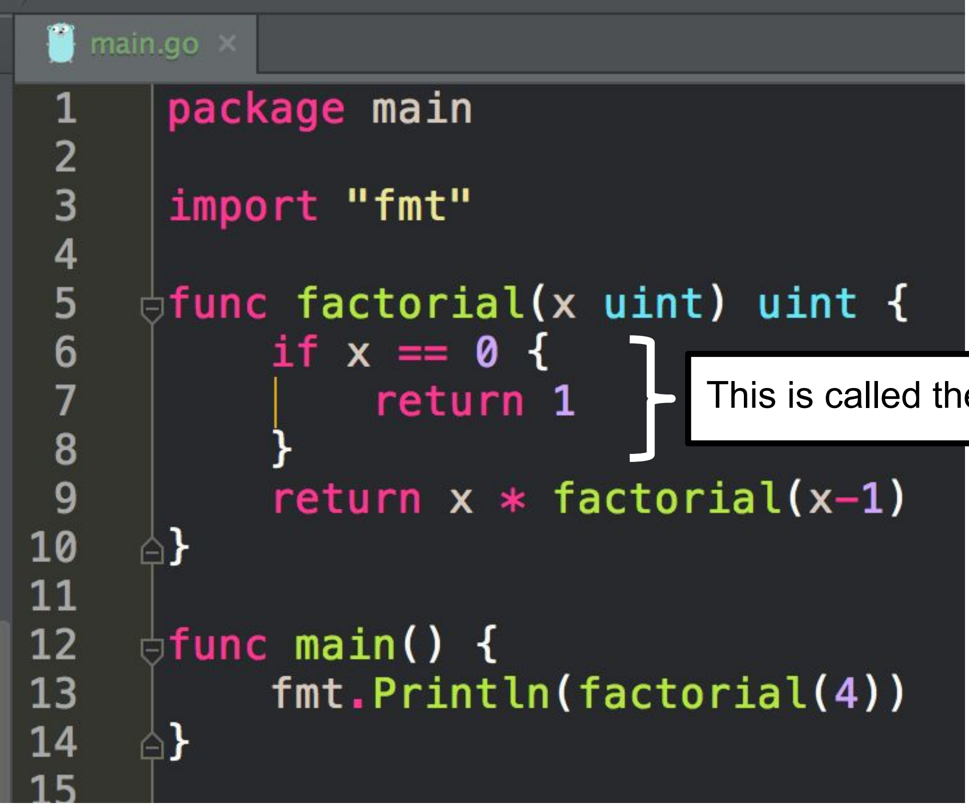
----

returns: 4 * 3 * 2 * 1 * 1

The End Result:

- 4 * 3 * 2 * 1

recursion



```go
package main

import "fmt"

func factorial(x uint) uint {
	if x == 0 {
		return 1
	}
	return x * factorial(x-1)
}

func main() {
	fmt.Println(factorial(4))
}

```

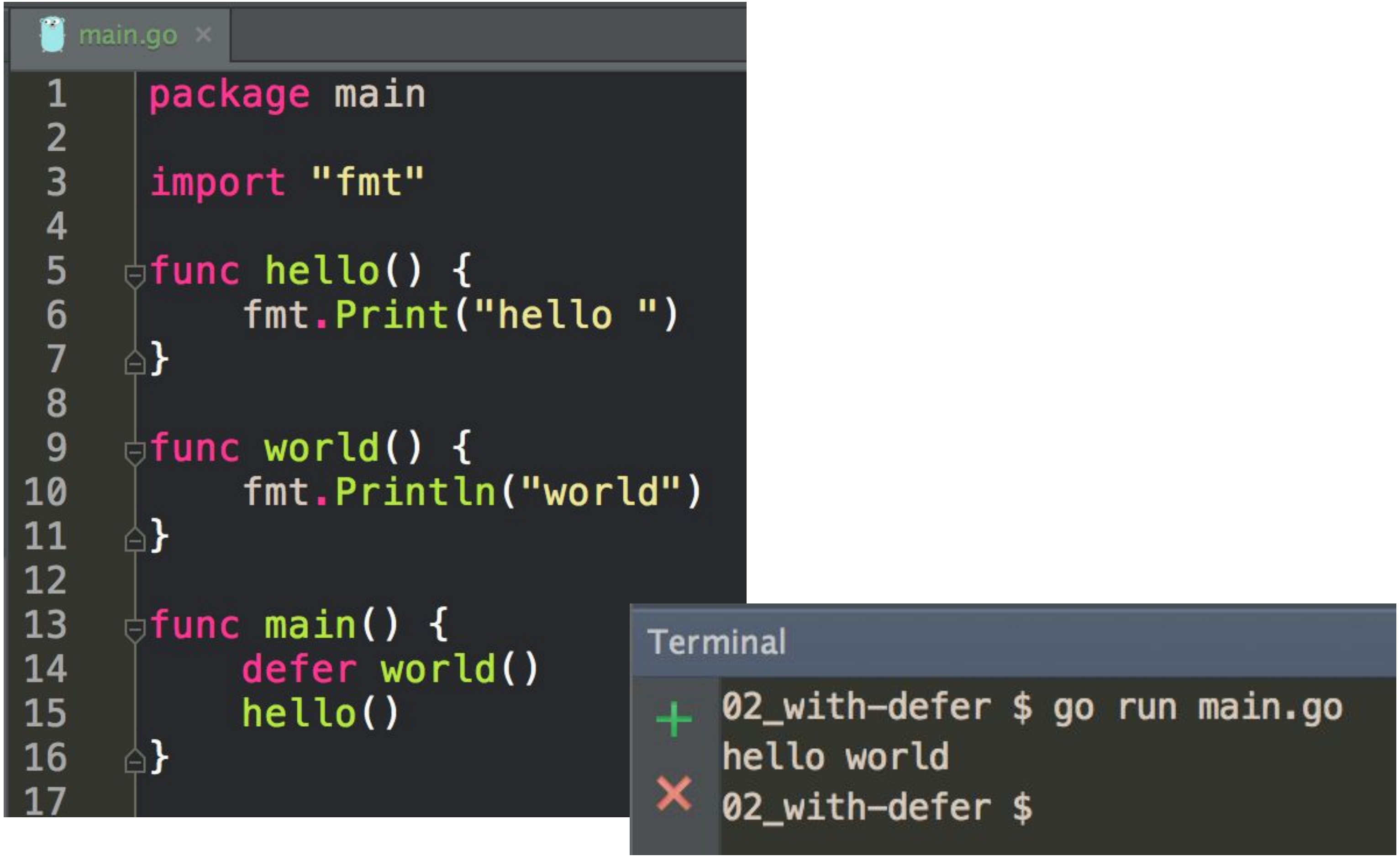
This is called the base case

- You can always use loops to solve any problem that can be solved with recursion.
- Loops are more performant than recursion.

recursion

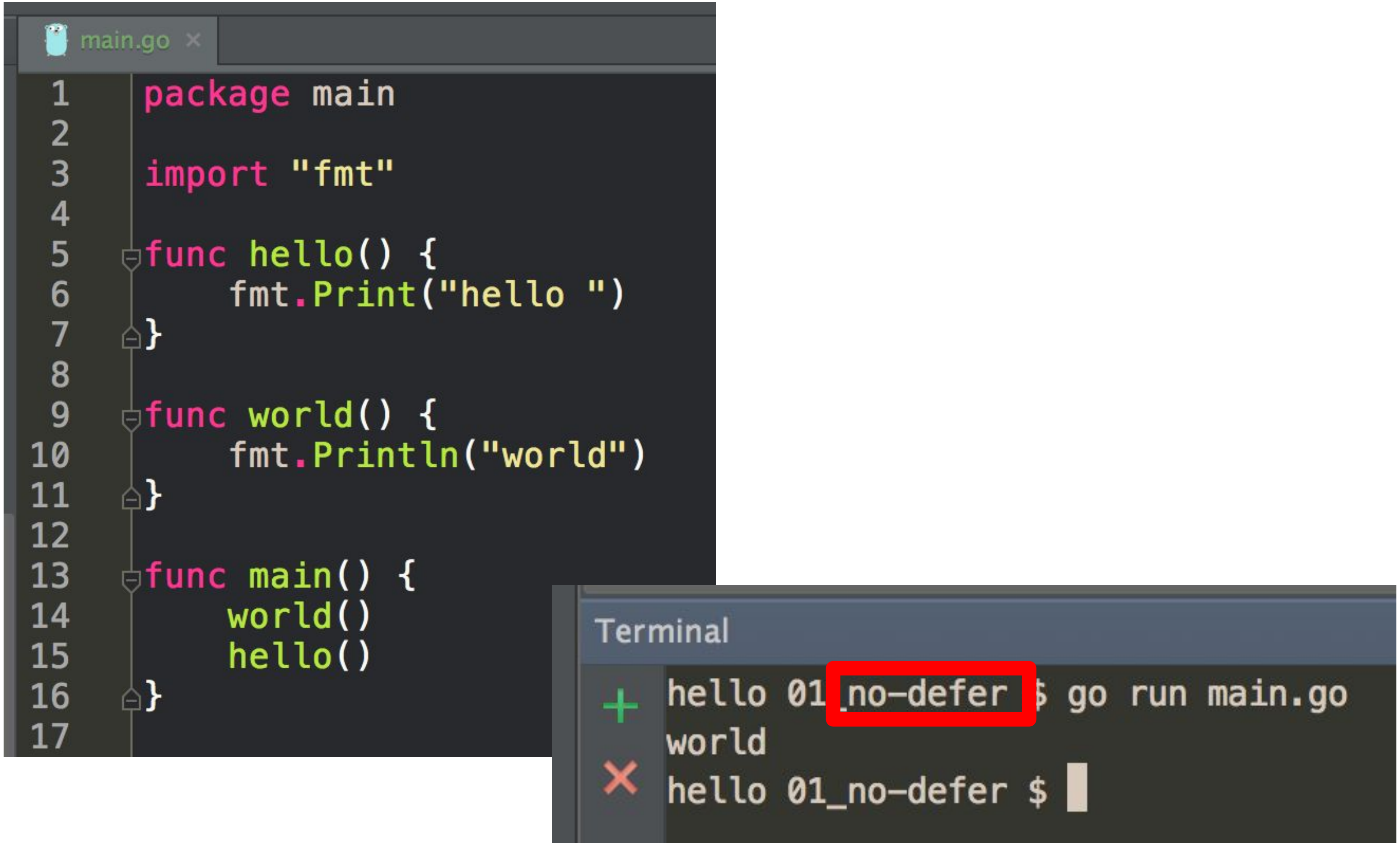
defer

run this at the last possible moment



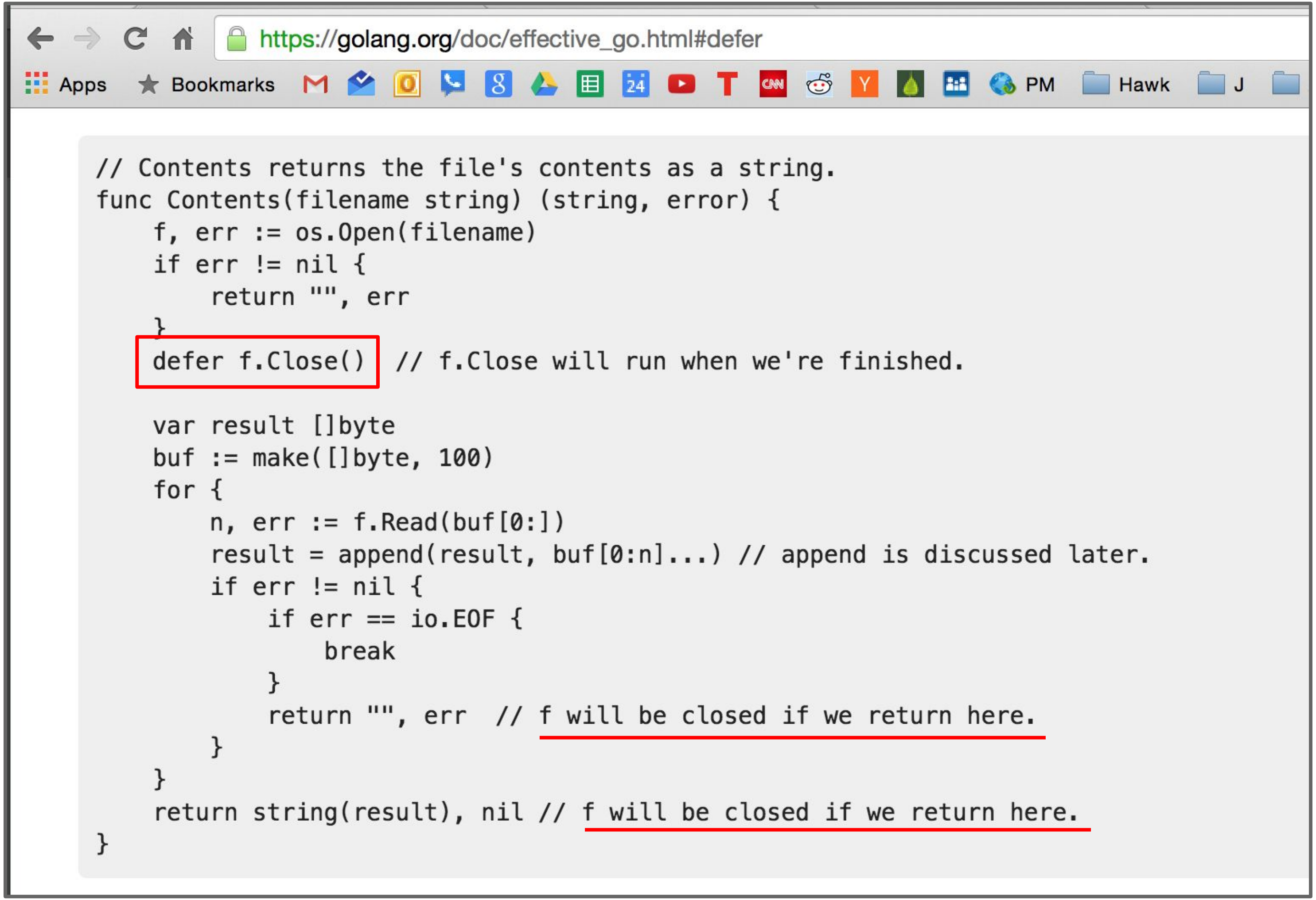
```go
package main

import "fmt"

func hello() {
	fmt.Print("hello ")
}

func world() {
	fmt.Println("world")
}

func main() {
	defer world()
	hello()
}

```

```
02_with-defer $ go run main.go
hello world
02_with-defer $
```

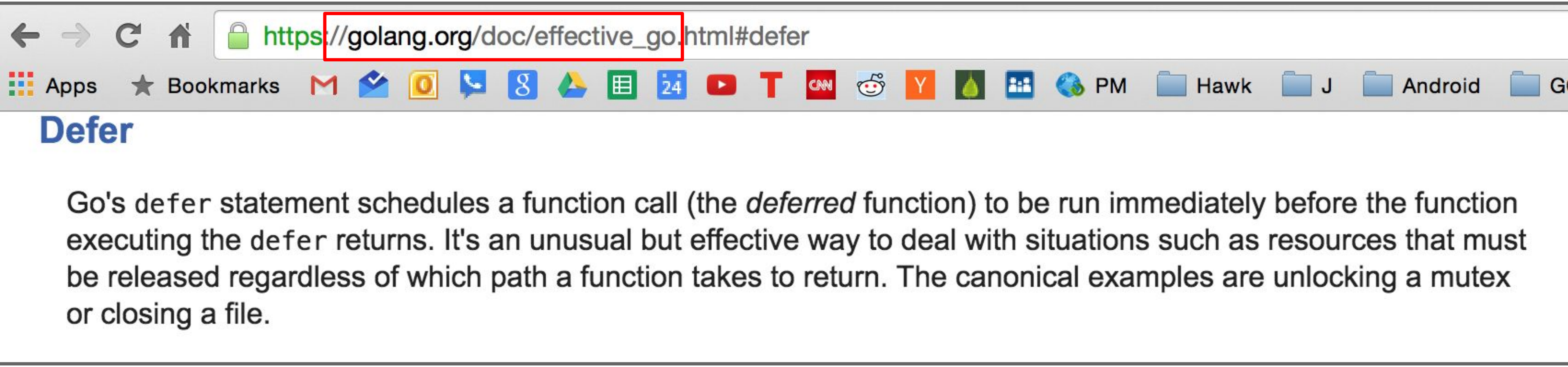


```go
package main

import "fmt"

func hello() {
	fmt.Print("hello ")
}

func world() {
	fmt.Println("world")
}

func main() {
	world()
	hello()
}

```

```
hello 01_no-defer $ go run main.go
world
hello 01_no-defer $ 
```



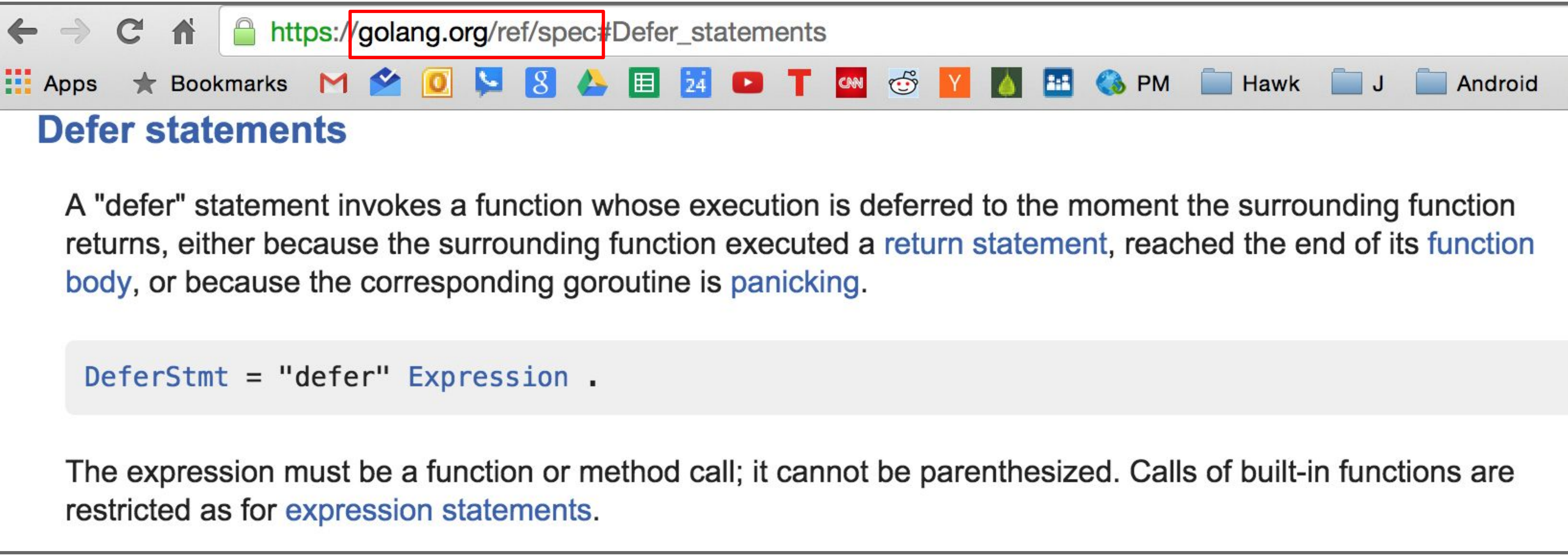
```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed later.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```



https://golang.org/doc/effective_go.html#defer

Defer

Go's `defer` statement schedules a function call (the *deferred* function) to be run immediately before the function executing the `defer` returns. It's an unusual but effective way to deal with situations such as resources that must be released regardless of which path a function takes to return. The canonical examples are unlocking a mutex or closing a file.



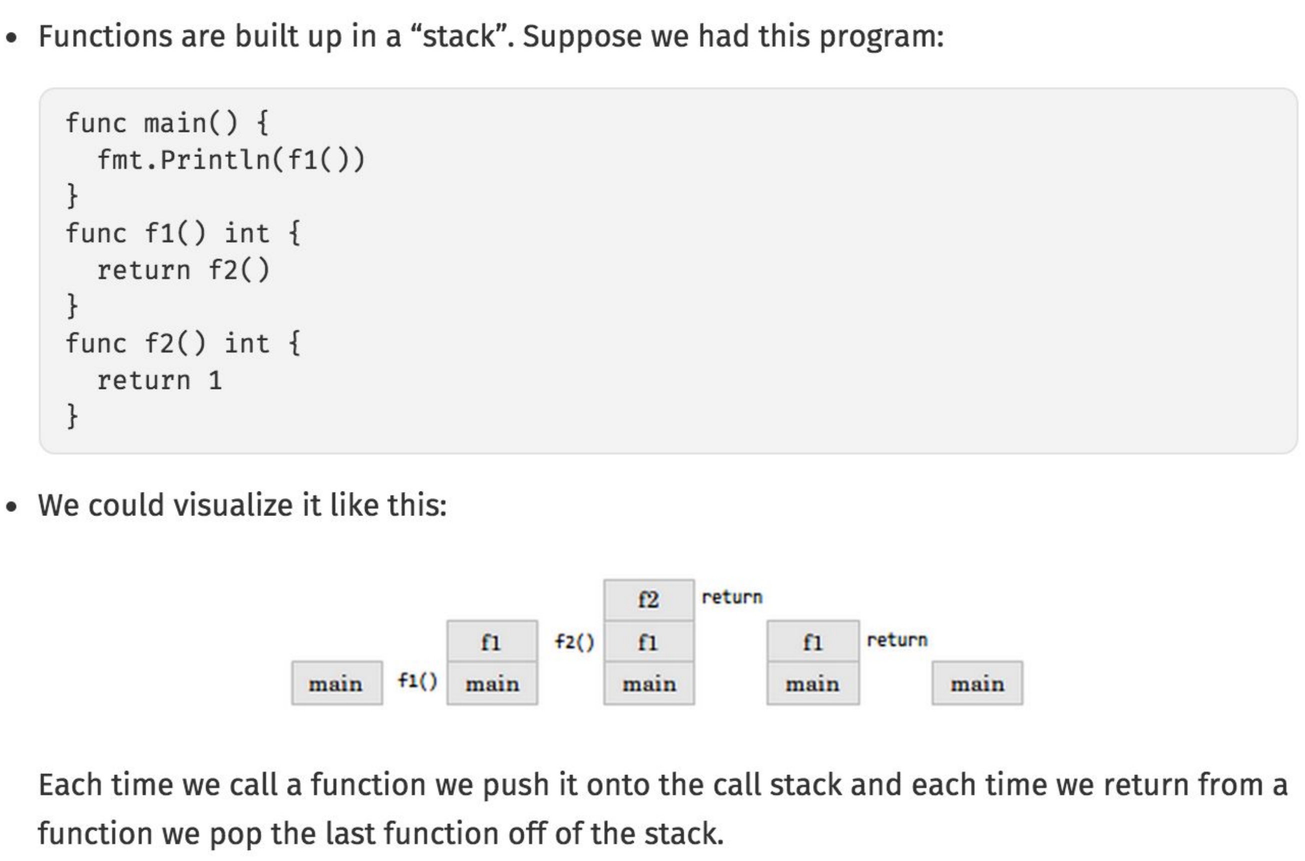
Defer statements

A "defer" statement invokes a function whose execution is deferred to the moment the surrounding function returns, either because the surrounding function executed a return statement, reached the end of its function body, or because the corresponding goroutine is panicking.

```
DeferStmt = "defer" Expression .
```

The expression must be a function or method call; it cannot be parenthesized. Calls of built-in functions are restricted as for expression statements.

the stack



- Functions are built up in a “stack”. Suppose we had this program:

```
func main() {
  fmt.Println(f1())
}
func f1() int {
  return f2()
}
func f2() int {
  return 1
}
```

- We could visualize it like this:

Each time we call a function we push it onto the call stack and each time we return from a function we pop the last function off of the stack.

http://www.golang-book.com/books/intro/7

Review

- func main() {}
- calling a function
- greeting()
- parameters vs arguments
  - two params
  - variadic
    - …params
    - args...
- returns
  - named returns
  - multiple returns
- variable shadowing
- **func expression**
  - setting a variable equal to a function
  - greeting := func(){<code here>}
    - greeting’s type is func
- **closure**
  - one thing enclosing another
  - helps us limit scope of variables
- **returning a func**
  - **functional programming**
- **callback**
  - passing a func as an argument
- recursion
- **defer**
- **the stack**
  - the order in which functions are called

exercises

bool

Write a program that prints the value of this expression:
(true && false) || (false && true) || !(false && false)

two params

Write a program that calls a function which takes first name and age then returns a string like this, “John is 27 years old.”

two returns

Write a program that calls a function which takes first name and age
then returns an int and a bool
the int: person’s age * 7 (dog years)
the bool: whether or not the person is old (age > 25)
use those two returns in a sentence like this,
(“John is 140 in dog years and is not old”)
or like this, (“Jane is 280 in dog years and is old”)

named return

Write a program that calls a function which takes age then returns **dogYears int** which is age * 7

variadic parameters

Write a program that has variadic parameters
use that function in a program

variadic arguments

Write a program that has variadic parameters
use that function in a program, passing in variadic arguments

func expression

Write a program that uses a func expression

variable type

You wrote a program that uses a func expression
now add a print statement that shows
the type of the variable to which the function is assigned

closure

create a program that uses closure

returning a func

create a func that returns a func
use that func in a program

callback

create a program that uses a callback

(a func is being passed in as an argument)

recursion

The Fibonacci sequence is defined as: fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2). Write a recursive function which can find fib(n).

defer

create a program that uses defer

review questions

Answer These Questions

- What is variable shadowing?
- What is a func expression?
- What is closure?
- What is a callback?
- How does defer work?
- What is the stack and how does it work?