Let's get Go-ing

An introduction to Go
programming for COS 316

Today's Agenda

Just enough Go to get started on Assignment 1.

- What is Go?
- Variables, loops, and functions in Go
- Navigating the standard library documentation

Why learn Go?

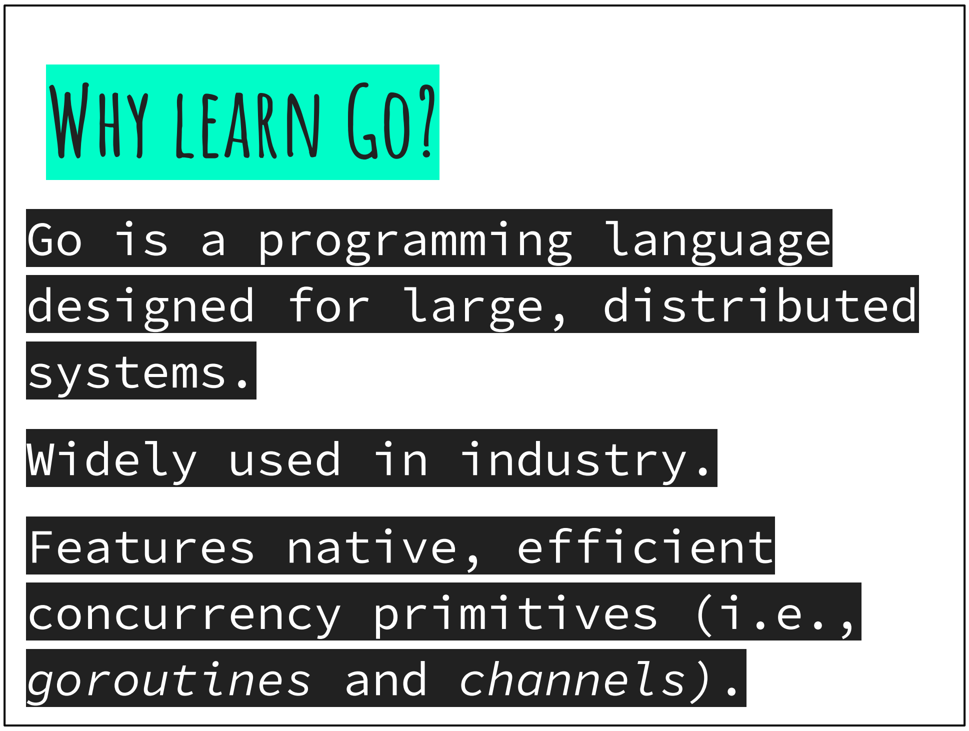
Why learn Go?

Go is a programming language designed for large, distributed systems.

Why learn Go?

Go is a programming language designed for large, distributed systems.

Widely used in industry.



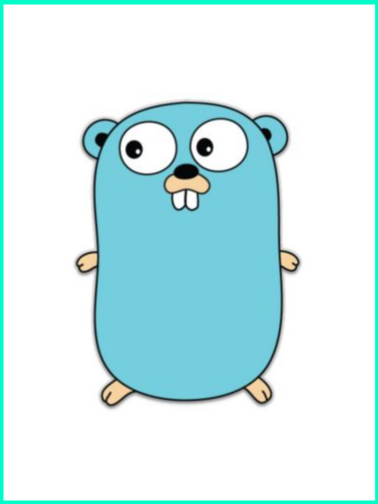
Why learn Go?

Go is a programming language designed for large, distributed systems.

Widely used in industry.

Features native, efficient concurrency primitives (i.e., *goroutines* and *channels*).

Worth mentioning Go is also used in COS 418.



Okay, let's write our first program

Variables

`https://go.dev/play`

Variables

```
package main

func main() {

}
```

Variables

```
package main

func main() {
  var a int = 3
}
```

Variables

```
package main

func main() {
  var a int = 3
}
```

Variable types come *after* variable names

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
}
```

Variable types come *after* variable names

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
  c := 1
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
  c := 1
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
  c := 1
  var d int
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
  c := 1
  var d int
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Can choose to accept default value (i.e., 0)

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
  c := 1
  var d int
  var e, f int = -1, -2
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Can choose to accept default value (i.e., 0)

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
  c := 1
  var d int
  var e, f int = -1, -2
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Can choose to accept default value (i.e., 0)

Can declare and init. multiple vars in 1 line

Variables

```
package main

func main() {
  var
  var
  c :=
  var
  var
}
```

Okay, looks good!
Let's run our code.

Variable types come *after* variable names

Variable types can be omitted and inferred

ccept
e., 0)

Can declare and init. multiple vars in 1 line

Variables

```
package main

func main() {
  var
  var
  c :=
  var
  var
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

ccept

e., 0)

Can declare and init. multiple vars in 1 line

```
Okay, looks good!
Let's run our code.

> go run main.go
```

Variables

```
package main
```

Variable types come *after* variable names

Variable types can be

default value (i.e., 0)

Can declare and init. multiple vars in 1 line

Compiler says nope!

X

```
./main.go:4:7: a declared and not used
./main.go:5:7: b declared and not used
./main.go:6:3: c declared and not used
./main.go:7:7: d declared and not used
./main.go:8:7: e declared and not used
./main.go:8:10: f declared and not used
```

Variables

```
package main

func main() {
  var
  var
  c :=
  var
  var
}
```

Go prevents you from compiling code with unused variables, so let's print them out

Variable types come *after* variable names

Variable types can be omitted and inferred

ccept

e., 0)

Can declare and init. multiple vars in 1 line

Variables

```
package main

func main() {
  var a int = 3
  var b = 2
  c := 1
  var d int
  var e, f int = -1, -2
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Can choose to accept default value (i.e., 0)

Can declare and init. multiple vars in 1 line

Variables

```
package main

import "fmt"

func main() {
  var a int = 3
  var b = 2
  c := 1
  var d int
  var e, f int = -1, -2
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Can choose to accept default value (i.e., 0)

Can declare and init. multiple vars in 1 line

Variables

```
package main

import "fmt"

func main() {
  var a int = 3
  var b = 2
  c := 1
  var d int
  var e, f int = -1, -2

  fmt.Println(a, b, c)
}
```

Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Can choose to accept default value (i.e., 0)

Can declare and init. multiple vars in 1 line

Variables

```
package main

import "fmt"

func main() {
  var a int = 3
  var b = 2
  c := 1
  var d int
  var e, f int = -1, -2

  fmt.Println(a, b, c)
  fmt.Println(d, e, f)
}
```

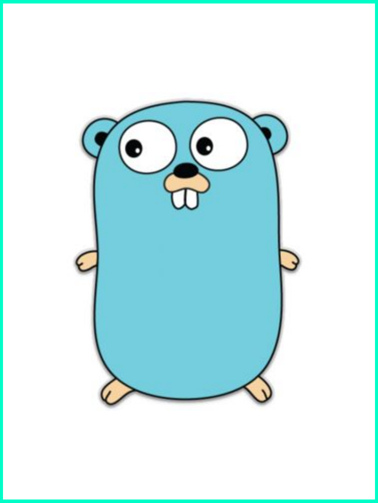
Variable types come *after* variable names

Variable types can be omitted and inferred

A shorthand for 'var c =' is 'c :='

Can choose to accept default value (i.e., 0)

Can declare and init. multiple vars in 1 line



Let's see this in action!

Play time!

"Go" to go.dev/play and try out some variable declarations.

Play time!

"Go" to go.dev/play and try out some variable declarations.

Here are some ideas.

1. Can you declare multiple variables with different types on the same line?

Play time!

"Go" to go.dev/play and try out some variable declarations.

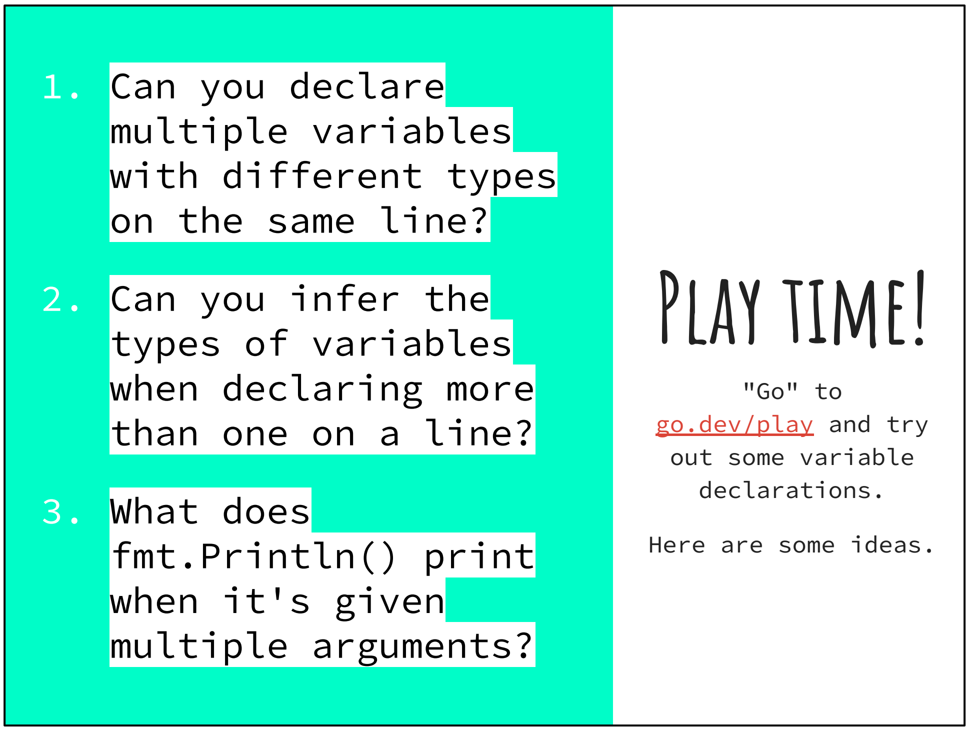
Here are some ideas.

1. Can you declare multiple variables with different types on the same line?
2. Can you infer the types of variables when declaring more than one on a line?

PLAY TIME!

"Go" to go.dev/play and try out some variable declarations.

Here are some ideas.



1. Can you declare multiple variables with different types on the same line?
2. Can you infer the types of variables when declaring more than one on a line?
3. What does fmt.Println() print when it's given multiple arguments?

Play time!

"Go" to go.dev/play and try out some variable declarations.

Here are some ideas.

1. Yes, if you instantiate the variables without the type
2. Yes
3. It gives space-separated values

Loops

Loops

```go
package main

func main() {

}
```

LOOPS

```go
package main

import "fmt"

func main() {
  for i := 1; i <= 3; i++ {
     fmt.Println(i)
  }
}
```

```
package main

import "fmt"

func main() {
  for i := 1; i <= 3; i++ {
     fmt.Println(i)
  }
}
```

Loops

'for' loops work like in Java/C, but don't require ()

Must use { }, even for 1-line loops

Loops

```go
package main

import "fmt"

func main() {
  for i := 1; i <= 3; i++ {
     fmt.Println(i)
  }
  i := 4
  for i <= 10 {
    fmt.Println(i)
    i++
  }
}
```

'for' loops work like in Java/C, but don't require ()

Must use { }, even for 1-line loops

```
package main

import "fmt"

func main() {
  for i := 1; i <= 3; i++ {
     fmt.Println(i)
  }
  i := 4
  for i <= 10 {
    fmt.Println(i)
    i++
  }
}
```

Loops

'for' loops work like in Java/C, but don't require ()

Must use { }, even for 1-line loops

No such thing as 'while' loops in Go

Loops

```
package main

import "fmt"

func main() {
  for i := 1; i <= 3; i++ {
     fmt.Println(i)
  }
  i := 4
  for i <= 10 {
    fmt.Println(i)
    i++
  }
  for {
    fmt.Println("done!")
    break
  }
}
```

'for' loops work like in Java/C, but don't require ()

Must use { }, even for 1-line loops

No such thing as 'while' loops in Go

Loops

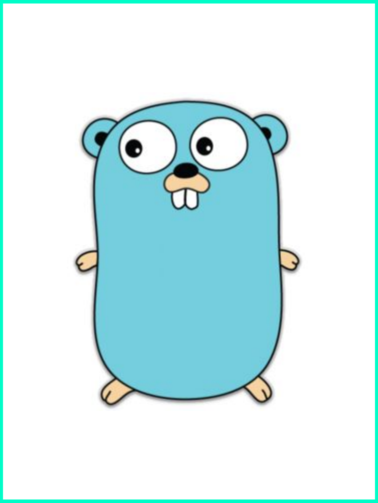
```
package main

import "fmt"

func main() {
  for i := 1; i <= 3; i++ {
     fmt.Println(i)
  }
  i := 4
  for i <= 10 {
    fmt.Println(i)
    i++
  }
  for {
    fmt.Println("done!")
    break
  }
}
```

'for' loops work like in Java/C, but don't require ()

Must use { }, even for 1-line loops

No such thing as 'while' loops in Go

Can use 'break' and 'continue'



Let's try it ourselves

Let's Get Loopy

Navigate to go.dev/play and write a few Go loops.

1. Does the scoping of the index variable in a Go 'for' loop extend beyond the loop?

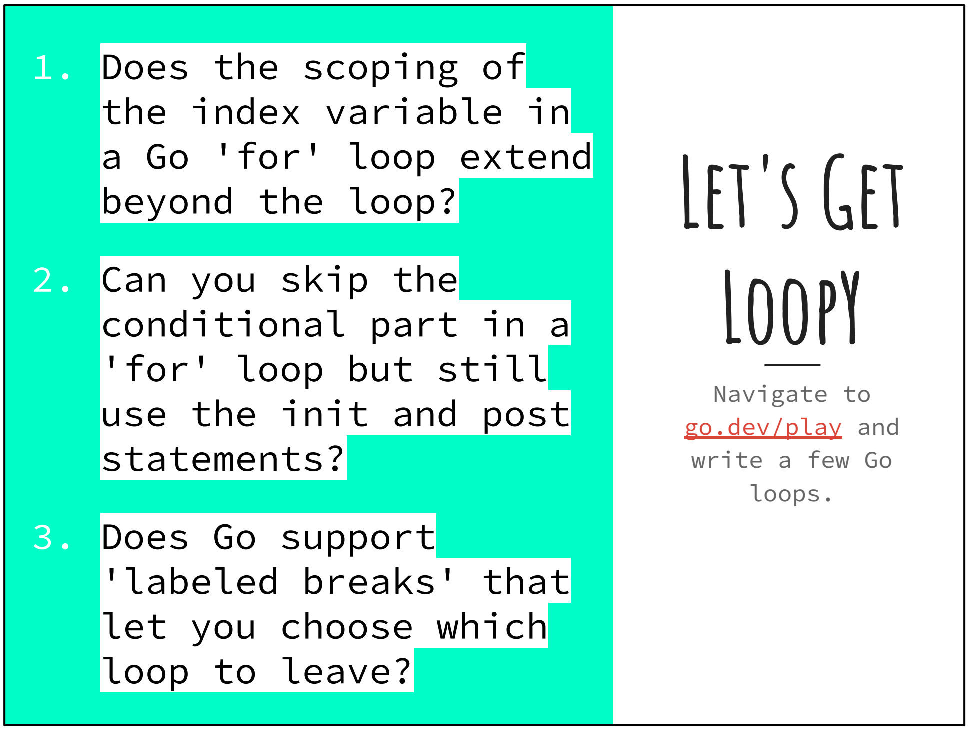
Let's Get Loopy

Navigate to go.dev/play and write a few Go loops.

1. Does the scoping of the index variable in a Go 'for' loop extend beyond the loop?
2. Can you skip the conditional part in a 'for' loop but still use the init and post statements?

LET'S GET LOOPY

Navigate to go.dev/play and write a few Go loops.



1. Does the scoping of the index variable in a Go 'for' loop extend beyond the loop?
2. Can you skip the conditional part in a 'for' loop but still use the init and post statements?
3. Does Go support 'labeled breaks' that let you choose which loop to leave?

LET'S GET LOOPY

Navigate to go.dev/play and write a few Go loops.

1. If the variable is declared as part of the loop invocation, then its scope doesn't extend beyond the loop.
2. Yes
3. Yes

Functions

Functions

```
func f(a int, b int) int {
  return a + b
}
```

FUNCTIONS

```
func f(a int, b int) int {
  return a + b
}
```

A function's return type is listed after its args

Functions

```
func f(a int, b int) int {
  return a + b
}

func g(a, b int) int {
  return a * b
}
```

A function's return type is listed after its args

Functions

```
func f(a int, b int) int {
  return a + b
}

func g(a, b int) int {
  return a * b
}
```

A function's return type is listed after its args

If args are same type, can specify type once at end

Functions

```
func f(a int, b int) int {
  return a + b
}

func g(a, b int) int {
  return a * b
}

func h(a, b int) (int,int) {
  return f(a, b), g(a, b)
}
```

A function's return type is listed after its args

If args are same type, can specify type once at end

FUNCTIONS

```
func f(a int, b int) int {
  return a + b
}

func g(a, b int) int {
  return a * b
}

func h(a, b int) (int,int) {
  return f(a, b), g(a, b)
}
```

A function's return type is listed after its args

If args are same type, can specify type once at end

Functions can return more than one result

FUNCTIONS

```
func f(a int, b int) int {
  return a + b
}

func g(a, b int) int {
  return a * b
}

func h(a, b int) (int,int) {
  return f(a, b), g(a, b)
}

func main() {
  a, b := h(1, 2)
  _, c := h(3, 4)
}
```

A function's return type is listed after its args

If args are same type, can specify type once at end

Functions can return more than one result

Functions

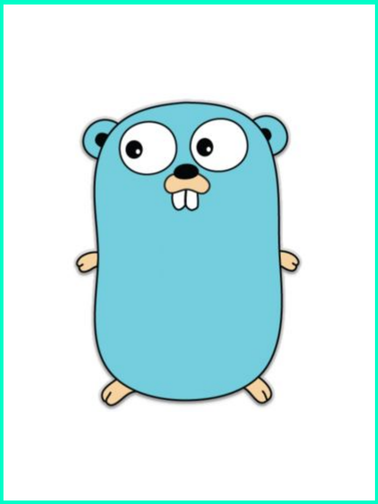
```
func f(a int, b int) int {
  return a + b
}

func g(a, b int) int {
  return a * b
}

func h(a, b int) (int,int) {
  return f(a, b), g(a, b)
}

func main() {
  a, b := h(1, 2)
  _, c := h(3, 4)
}
```

A function's return type is listed after its args

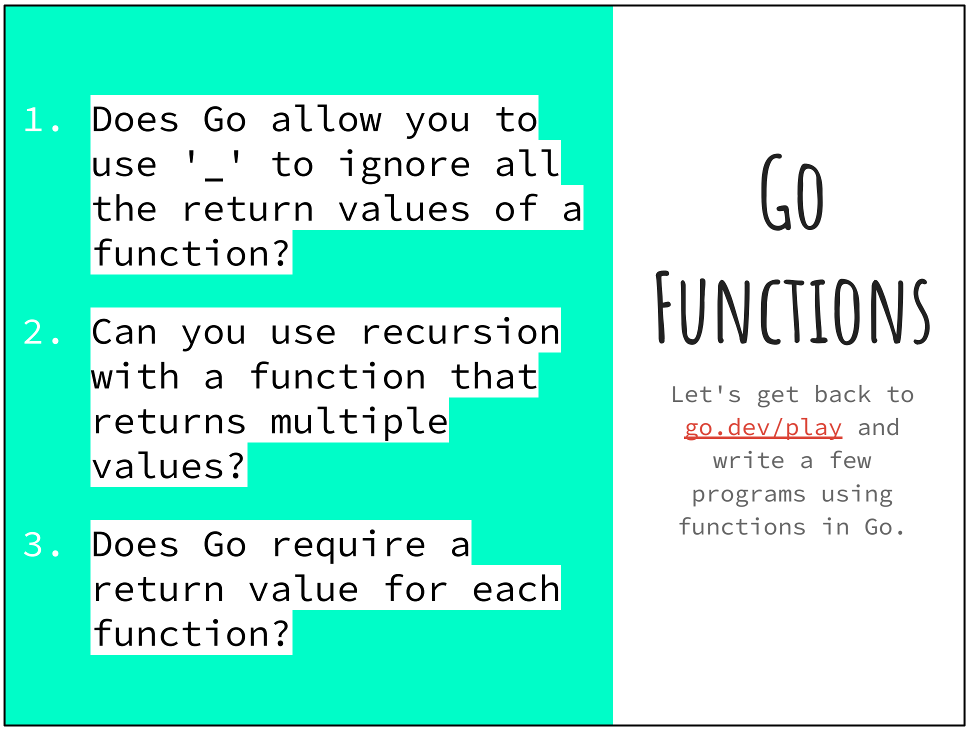
If args are same type, can specify type once at end

Functions can return more than one result

'_' throws away a return value



Last programming exercise!



1. Does Go allow you to use '_' to ignore all the return values of a function?
2. Can you use recursion with a function that returns multiple values?
3. Does Go require a return value for each function?

Go Functions

Let's get back to go.dev/play and write a few programs using functions in Go.

1. No
2. Yes
3. No

Go Standard Library

Go Standard Library

All Go programs have access to to a massive standard library of packages. (See pkg.go.dev/std)

Go Standard Library

All Go programs have access to to a massive standard library of packages. (See pkg.go.dev/std)

This collection of officially supported packages is one of the reasons Go is a useful language for systems programmers.

Reading The Documentation

Reading The Documentation

Navigating the documentation is hard.

Reading The Documentation

Navigating the documentation is hard.

There's a lot of it and you'll be learning about the language as you read it.

Reading The Documentation

Navigating the documentation is hard.

There's a lot of it and you'll be learning about the language as you read it.

Expect to spend some time pouring over it.

External Sources

External Sources

Googling is allowed, even encouraged, in this course. You may use any online resource.

External Sources

Googling is allowed, even encouraged, in this course. You may use any online resource.

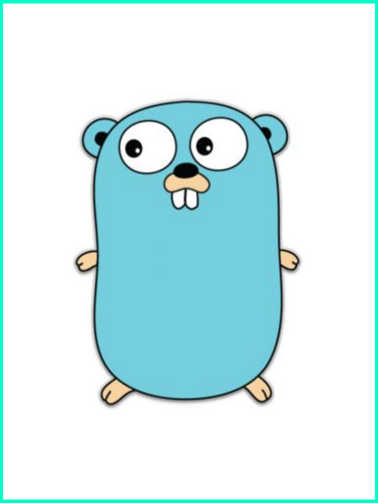
If you base a significant portion of your code on it, cite it in a comment in your code.

External Sources

Googling is allowed, even encouraged, in this course. You may use any online resource.

If you base a significant portion of your code on it, cite it in a comment in your code.

Search for “golang” instead.



Let's see the docs

Doc Hunt

1. Find some "interesting" packages
2. Can you experiment using the provided examples?

Navigate to
pkg.go.dev

Use
go.dev/play

Questions?

Please don't hesitate to ask!

Additional Resources

- Go.dev
- Go Tutorial
- go.dev/play
- gobyexample.com
- "Learn Go Programming" (7 hour YouTube tutorial)