Module 2: Function Types
Topic 1.1: First-class Values

Functions are First-class

- Functions can be treated like other types
  - Variables can be declared with a function type
  - Can be **created dynamically**
  - Can be **passed as arguments** and **returned as values**
  - Can be **stored in data structures**

Variables as Functions

- Declare a variable as a func
- Function is on right-hand side, without ()

```
var funcVar func(int) int
func incFn(x int) int {
    return x + 1
}
func main() {
    funcVar = incFn
    fmt.Print(funcVar(1))
}
```

Functions as Arguments

- Function can be passed to another function as an argument

```
func applyIt(afunct func (int) int,
   val int) int {
    return afunct(val)
}
```

Functions as Arguments

```
func applyIt(afunct func (int) int,
   val int) int {
    return afunct(val)
}

func incFn(x int) int {return x + 1}
func decFn(x int) int {return x - 1}

func main() {
   fmt.Println(applyIt(incFn, 2))
   fmt.Println(applyIt(decFn, 2))
}
```

Anonymous Functions

- Don’t need to name a function

```
func applyIt(afunct func (int) int,
   val int) int {
    return afunct(val)
}

func main() {
    v := applyIt(func (x int) int
          {return x + 1}, 2)
    fmt.Println(v)
}
```

Module 2: Function Types
Topic 1.2: Returning Functions

Functions as Return Values

- Functions can return functions
- Might create a function with controllable parameters
- Example: **Distance to Origin function**
  - Takes a point (x, y, coordinates)
  - Returns distance to origin
- What if I want to change the origin?
  - Option 1: Pass origin as argument
  - Option 2: Define function with new origin

Function Defines a Function

```
1. func MakeDistOrigin(o_x, o_y float64)
2.         func (float64, float64) float64 {
3.     fn := func (x, y float64) float64 {
4.             return math.Sqrt(math.Pow(x - o_x, 2) +
5.                 math.Pow(y - o_y, 2))
6.         }
7.     return fn
8. }
```

- Origin location is passed as an argument
- Origin is built into the returned function

Special-Purpose Functions

```
func main() {
    Dist1 := MakeDistOrigin(0,0)
    Dist2 := MakeDistOrigin(2,2)
    fmt.Println(Dist1(2,2))
    fmt.Println(Dist2(2,2))
}
```

- `Dist1()` and `Dist2()` have different origins

Environment of a Function

- Set of all names that are valid inside a function
- Names defined locally, in the function
- **Lexical Scoping**
- Environment includes names defined in block where the function is defined

```
var x int
funct foo(y int) {
    z := 1
    …
}
```

Closure

- Function + its environment
- When functions are passed/returned, their environment comes with them!

```
func MakeDistOrigin(o_x, o_y float64)
        func (float64, float64) float64 {
    fn := func (x, y float64) float64 {
            return math.Sqrt(math.Pow(x - o_x, 2) +
                math.Pow(y - o_y, 2))
        }
```

- `o_x` and `o_y` are in the closure of `fn()`

Module 2: Functions and Organization
Topic 2.1: Variadic and Deferred

Variable Argument Number

- Functions can take a variable number of arguments
- Use ellipsis ... to specify
- Treated as a slice inside function

```
func getMax(vals ...int) int {
   maxV := -1
   for _, v := range vals {
      if v > maxV {
         maxV = v
      }
   }
   return maxV
}
```

Variadic Slice Argument

- Can pass a slice to a variadic function
- Need the ... suffix

```
func main() {
   fmt.Println(getMax(1, 3, 6, 4))

   vslice := []int{1, 3, 6, 4}

   fmt.Println(getMax(vslice...))
}
```

Deferred Function Calls

- Call can be **deferred** until the surrounding function completes
- Typically used for cleanup activities

```
func main() {
   defer fmt.Println(“Bye!”)

   fmt.Println(“Hello!”)
}
```

Deferred Call Arguments

- Arguments of a deferred call are evaluated immediately

```
func main() {
    i := 1
    defer fmt.Println(i+1)
    i++
    fmt.Println("Hello!")
}
```