MODULE-2

Control Structures

1) Arrays
2) Slices and Maps
3) Function control structures:
    (a) The for Statement
    (b) If Statement
    (c) Switch Statement
    (d) Arrays
4) Slices & Maps: Arrays, Slices-append, Copy, Maps,
5) Functions: Your Second Function,
6) Variadic Functions
7) Closure,
8) Defer
9) Panic and recovery
10) Pointers
11) The * and & Operators, New.

Arrays

Control Structures

- Now that we know how to use variables it's time to start writing some useful programs. First let's write a program that counts to 10, starting from 1, with each number on its own line. Using what we've learned so far, we could write this:

```
package main

import "fmt"

func main() {
    fmt.Println(1)
    fmt.Println(2)
    fmt.Println(3)
    fmt.Println(4)
    fmt.Println(5)
    fmt.Println(6)
    fmt.Println(7)
    fmt.Println(8)
    fmt.Println(9)
    fmt.Println(10)
}
```

Or this:

```
package main
import "fmt"

func main() {
    fmt.Println(`1
2
3
4
5
6
7
8
9
10`)
}
```

But both of these programs are pretty tedious to write. What we need is a way of doing something multiple times.

FOR

- The for statement allows us to repeat a list of statements (a block) multiple times. Rewriting our previous program using a for statement looks like this:

```
package main

import "fmt"

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

- First, we create a variable called i that we use to store the number we want to print.
- Then we create a for loop by using the keyword for, providing a conditional expression which is either true or false and finally supplying a block to execute. The for loop works like this:

1. We evaluate (run) the expression i <= 10 ("i less than or equal to 10"). If this evaluates to true then we run the statements inside of the block. Otherwise, we jump to the next line of our program after the block. (in this case there is nothing after the for loop so we exit the program.

2. After we run the statements inside of the block we loop back to the beginning of the for statement and repeat step 1. Control Structures 50 The i = i + 1 line is extremely important, because without it i <= 10 would always evaluate to true and our program would never stop. (When this happens, this is referred to as an infinite loop)

As an exercise lets walk through the program like a computer would:

• Create a variable named i with the value 1
• Is i <= 10? Yes.
• Print i
• Set i to i + 1 (i now equals 2)
• Is i <= 10? Yes.
• Print i
• Set i to i + 1 (i now equals 3)

• …
• Set i to i + 1 (i now equals 11)
• Is i <= 10? No.
• Nothing left to do, so exit

- Other programming languages have a lot of different types of loops (while, do, until, foreach, …) but Go only has one that can be used in a variety of different ways. The previous program could also have been written like this:

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

- Now the conditional expression also contains two other statements with semicolons between them. First we have the variable initialization, then we have the condition to check each time and finally we "increment" the variable. (adding 1 to a variable is so

common that we have a special operator: ++. Similarly subtracting 1 can be done with --)

We will see additional ways of using the for loop in later chapters.

If

- Let's modify the program we just wrote so that instead of just printing the numbers 1-10 on each line it also specifies whether or not the number is even or odd. Like this:

```
1 odd
2 even
3 odd
4 even
5 odd
6 even
7 odd
8 even
9 odd
10 even
```

- First, we need a way of determining whether or not a number is even or odd.
- An easy way to tell is to divide the number by 2.
- If you have nothing left over then the number is even, otherwise it's odd.
- So how do we find the remainder after division in Go?
- We use the % operator. 1 % 2 equals 1, 2 % 2 equals 0, 3 % 2 equals 1 and so on.

- Next, we need a way of choosing to do different things based on a condition. For that we use the if statement:

```
if i % 2 == 0 {
  // even
} else {
  // odd
}
```

- An if statement is similar to a for statement in that it has a condition followed by a block. If statements also have an optional else part. If the condition evaluates to true then the block after the condition is run, otherwise either the block is skipped or if the else block is present that block is run.

If statements can also have else if parts:

```
if i % 2 == 0 {
    // divisible by 2
} else if i % 3 == 0 {
    // divisible by 3
} else if i % 4 == 0 {
    // divisible by 4
}
```

The conditions are checked top down and the first one to result in true will have its associated block executed. None of the other blocks will execute, even if their conditions also pass. (So, for example the number 8 is divisible by both 4 and 2, but the // divisible by 4 blocks will never execute because the // divisible by 2 blocks is done first)

Putting it all together we have

```
func main() {
    for i := 1; i <= 10; i++ {
        if i % 2 == 0 {
            fmt.Println(i, "even")
        } else {
            fmt.Println(i, "odd")
        }
    }
}
```

Let's walk through this program:

Create a variable i of type int and give it the value 1

• Is i less than or equal to 10? Yes: jump to the block

• Is the remainder of i ÷ 2 equal to 0? No: jump to the else block

• Print i followed by odd

• Increment i (the statement after the condition)

• Is i less than or equal to 10? Yes: jump to the block

• Is the remainder of i ÷ 2 equal to 0? Yes: jump to the if block

• Print i followed by even

• …

- The remainder operator, while rarely seen outside of elementary school, turns out to be really useful when programming. You'll see it turn up everywhere from zebra striping tables to partitioning data sets.

Switch

- Suppose we wanted to write a program that printed the English names for numbers. Using what we've learned so far, we might start by doing this:

```
if i == 0 {
    fmt.Println("Zero")
} else if i == 1 {
    fmt.Println("One")
} else if i == 2 {
    fmt.Println("Two")
} else if i == 3 {
    fmt.Println("Three")
} else if i == 4 {
    fmt.Println("Four")
} else if i == 5 {
    fmt.Println("Five")
}
```

- Since writing a program in this way would be pretty tedious Go provides another statement to make this easier: the switch statement. We can rewrite our program to look like this
-

```
switch i {
case 0: fmt.Println("Zero")
case 1: fmt.Println("One")
case 2: fmt.Println("Two")
case 3: fmt.Println("Three")
case 4: fmt.Println("Four")
case 5: fmt.Println("Five")
default: fmt.Println("Unknown Number")
}
```

  Like an if statement each case is checked top down and the first one to succeed is chosen. A switch also supports a default case which will happen if none of the cases matches the value. (Kind of like the else in an if statement)

- These are the main control flow statements. Additional statements will be explored in later chapters.

Problems

1. What does the following program print: i: = 10 if i > 10 {fmt. Println("Big")} else {fmt. Println("Small")}
2. Write a program that prints out all the numbers evenly divisible by 3 between 1 and 100. (3, 6, 9, etc.)
3. Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "Fizz Buzz"

Arrays, Slices and Maps

- In module-1 we learned about Go's basic types. In this module we will look at three more built-in types: arrays, slices and maps.

Arrays

- An array is a numbered sequence of elements of a single type with a fixed length. In Go they look like this:

```
var x [5]int
```

- x is an example of an array which is composed of 5 ints. Try running the following program:

```
package main

import "fmt"

func main() {
     var x [5]int
     x[4] = 100
     fmt.Println(x)
}
```

You should see this:

```
[0 0 0 0 100]
```

- x [4] = 100 should be read "set the 5th element of the array x to 100". It might seem strange that x [4] represents the 5th element instead of the 4th but like strings, arrays are indexed starting from 0. Arrays are accessed in a similar way. We could change fmt. Println(x) to fmt. Println (x [4]) and we would get 100.

Here's an example program that uses arrays

```
func main() {
     var x [5]float64
     x[0] = 98
     x[1] = 93
     x[2] = 77
     x[3] = 82
     x[4] = 83

     var total float64 = 0
     for i := 0; i < 5; i++ {
          total += x[i]
     }
     fmt.Println(total / 5)
}
```

This program computes the average of a series of test scores. If you run it you should see 86.6. Let's walk through the program:

• First, we create an array of length 5 to hold our test scores, then we fill up each element with a grade

• Next, we setup a for loop to compute the total score

• Finally, we divide the total score by the number of elements to find the average

This program works, but Go provides some features we can use to improve it. First these 2 parts: i < 5 and total / 5 should throw up a red flag for us. Say we changed the number of grades from 5 to 6. We would also need to change both of these parts. It would be better to use the length of the array instead:

```
var total float64 = 0
for i := 0; i < len(x); i++ {
      total += x[i]
}
fmt.Println(total / len(x))
```

Go ahead and make these changes and run the program. You should get an error:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:19: invalid operation: total / 5
(mismatched types float64 and int)
```

- The issue here is that len(x) and total have different types. total is a float64 while len(x) is an int. So, we need to convert len(x) into a float64:

```
fmt.Println(total / float64(len(x)))
```

- This is an example of a type conversion. In general, to convert between types you use the type name like a function.

- Another change to the program we can make is to use a special form of the for loop:

```
var total float64 = 0
for i, value := range x {
    total += value
}
fmt.Println(total / float64(len(x)))
```

- In this for loop i represents the current position in the array and value is the same as x[i]. We use the keyword range followed by the name of the variable we want to loop over.

Running this program will result in another error:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:16: i declared and not used
```

The Go compiler won't allow you to create variables that you never use. Since we don't use `i` inside of our loop we need to change it to this:

```
var total float64 = 0
for _, value := range x {
    total += value
}
fmt.Println(total / float64(len(x)))
```

- A single _ (underscore) is used to tell the compiler that we don't need this. (In this case we don't need the iterator variable).

- Go also provides a shorter syntax for creating arrays:

```
x := [5]float64{ 98, 93, 77, 82, 83 }
```

- We no longer need to specify the type because Go can figure it out. Sometimes arrays like this can get too long to fit on one line, so Go allows you to break it up like this:

```
x := [5]float64{
     98,
     93,
     77,
     82,
     83,
}
```

- Notice the extra trailing, after 83. This is required by
- Go and it allows us to easily remove an element from the array by commenting out the line:

```
x := [4]float64{
     98,
     93,
     77,
     82,
     // 83,
}
```

  - This example illustrates a major issue with arrays: their length is fixed and part of the array's type name. In order to remove the last item, we actually had to change the type as well. Go's solution to this problem is to use a different type: slices.

Slices

- A slice is a segment of an array. Like arrays slices are indexable and have a length. Unlike arrays this length is allowed to change. Here's an example of a slice:

```
var x []float64
```

- The only difference between this and an array is the missing length between the brackets. In this case x has been created with a length of 0.
- If you want to create a slice you should use the built-in make function:

```
x := make([]float64, 5)
```

- This creates a slice that is associated with an underlying float64 array of length 5. Slices are always associated with some array, and although they can never be longer than the array, they can be smaller. The make function also allows a 3rd parameter:

```
x := make([]float64, 5, 10)
```

- 10 represents the capacity of the underlying array which the slice points to:

- Another way to create slices is to use the [low: high] expression:

```
arr := [5]float64{1,2,3,4,5}
x := arr[0:5]
```

- low is the index of where to start the slice and high is the index where to end it (but not including the index itself). For example, while arr [0:5] returns [1,2,3,4,5], arr [1:4] returns [2,3,4].

- For convenience we are also allowed to omit low, high or even both low and high. arr [0:] is the same as arr [0: len(arr)], arr [:5] is the same as arr [0:5] and arr [:] is the same as arr [0: len(arr)].

Slice Functions

- Go includes two built-in functions to assist with slices: append and copy. Here is an example of append:

```
func main() {
     slice1 := []int{1,2,3}
     slice2 := append(slice1, 4, 5)
     fmt.Println(slice1, slice2)
}
```

- After running this program slice1 has [1,2,3] and slice2 has [1,2,3,4,5]. append creates a new slice by taking an existing slice (the first argument) and appending all the following arguments to it.
- Here is an example of copy:

```
func main() {
     slice1 := []int{1,2,3}
     slice2 := make([]int, 2)
     copy(slice2, slice1)
     fmt.Println(slice1, slice2)
}
```

- After running this program slice1 has [1,2,3] and slice2 has [1,2]. The contents of slice1 are copied into slice2, but since slice2 has room for only two elements only the first two elements of slice1 are copied

Maps

- A map is an unordered collection of key-value pairs. Also known as an associative array, a hash table or a dictionary, maps are used to look up a value by its associated key.

Here's an example of a map in Go:

```
var x map[string]int
```

- The map type is represented by the keyword map, followed by the key type in brackets and finally the value type. If you were to read this out loud you would say “x is a map of strings to ints.”
- Like arrays and slices maps can be accessed using brackets. Try running the following program:

```
var x map[string]int
x["key"] = 10
fmt.Println(x)
```

You should see an error similar to this:

```
panic: runtime error: assignment to entry in nil
map

goroutine 1 [running]:
main.main()
  main.go:7 +0x4d

goroutine 2 [syscall]:
created by runtime.main

C:/Users/ADMINI~1/AppData/Local/Temp/2/bindi
t269497170/go/src/pkg/runtime/proc.c:221
exit status 2
```

- Up till now we have only seen compile-time errors. This is an example of a runtime error. As the name would imply, runtime errors happen when you run the program, while compile-time errors happen when you try to compile the program.

- The problem with our program is that maps have to be initialized before they can be used. We should have written this:

```
x := make(map[string]int)
x["key"] = 10
fmt.Println(x["key"])
```

- If you run this program you should see 10 displayed. The statement x["key"] = 10 is similar to what we saw with arrays but the key, instead of being an integer, is a string because the map's key type is string. We can also create maps with a key type of int:

```
x := make(map[int]int)
x[1] = 10
fmt.Println(x[1])
```

- This looks very much like an array but there are a few differences. First the length of a map (found by doing len(x)) can change as we add new items to it. When first created it has a length of 0, after x[1] = 10 it has a length of 1. Second maps are not sequential. We have x[1], and with an array that would imply there must be an x[0], but maps don't have this requirement.

- We can also delete items from a map using the built-in delete function

```
delete(x, 1)
```

Let's look at an example program that uses a map:

```
package main

import "fmt"

func main() {
    elements := make(map[string]string)
    elements["H"] = "Hydrogen"
    elements["He"] = "Helium"
    elements["Li"] = "Lithium"
    elements["Be"] = "Beryllium"
    elements["B"] = "Boron"
    elements["C"] = "Carbon"
    elements["N"] = "Nitrogen"
    elements["O"] = "Oxygen"
    elements["F"] = "Fluorine"
    elements["Ne"] = "Neon"

    fmt.Println(elements["Li"])
}
```

- elements is a map that represents the first 10 chemical elements indexed by their symbol. This is a very common way of using maps: as a lookup table or a dictionary. Suppose we tried to look up an element that doesn't exist:

```
fmt.Println(elements["Un"])
```

- If you run this you should see nothing returned. Technically a map returns the zero value for the value type (which for strings is the empty string). Although we could check for the zero value in a condition (elements["Un"] == "") Go provides a better way:

```
name, ok := elements["Un"]
fmt.Println(name, ok)
```

- Accessing an element of a map can return two values instead of just one. The first value is the result of the lookup, the second tells us whether or not the lookup was successful. In Go we often see code like this:

- First, we try to get the value from the map, then if it's successful we run the code inside of the block.
- Like we saw with arrays there is also a shorter way to create maps:

```
elements := map[string]string{
    "H": "Hydrogen",
    "He": "Helium",
    "Li": "Lithium",
    "Be": "Beryllium",
    "B": "Boron",
    "C": "Carbon",
    "N": "Nitrogen",
    "O": "Oxygen",
    "F": "Fluorine",
    "Ne": "Neon",
}
```

- Maps are also often used to store general information. Let's modify our program so that instead of just storing the name of the element we store its standard state (state at room temperature) as well:

```
func main() {
     elements := map[string]map[string]string{
           "H": map[string]string{
                 "name":"Hydrogen",
                 "state":"gas",
           },
           "He": map[string]string{
                 "name":"Helium",
                 "state":"gas",
           },
           "Li": map[string]string{
                 "name":"Lithium",
                 "state":"solid",
           },
```

```
            "Be": map[string]string{
                  "name":"Beryllium",
                  "state":"solid",
            },
            "B":  map[string]string{
                  "name":"Boron",
                  "state":"solid",
            },
            "C":  map[string]string{
                  "name":"Carbon",
                  "state":"solid",
            },
            "N":  map[string]string{
                  "name":"Nitrogen",
                  "state":"gas",
            },
            "O":  map[string]string{
                  "name":"Oxygen",
                  "state":"gas",
            },
            "F":  map[string]string{
                  "name":"Fluorine",
                  "state":"gas",
            },
            "Ne":  map[string]string{
                  "name":"Neon",
                  "state":"gas",
            },
      }

      if el, ok := elements["Li"]; ok {
            fmt.Println(el["name"], el["state"])
      }
}
```

Notice that the type of our map has changed from map[string]string to map[string]map[string]string. We now have a map of strings to maps of strings to strings. The outer map is used as a lookup table based on the element's symbol, while the inner maps are used to store general information about the elements. Although maps are often used like this, in chapter 9 we will see a better way to store structured information.

Problems
1. How do you access the 4th element of an array or slice?
2. What is the length of a slice created using: make ([]int, 3, 9)?
3. Given the array: x:= [6]string{"a","b","c","d","e","f"} what would x[2:5] give you?
4. Write a program that finds the smallest number in this list: x:= []int{ 48,96,86,68, 57,82,63,70, 37,34,83,27, 19,97, 9,17,

Functions

- A function is an independent section of code that maps zero or more input parameters to zero or more output parameters. Functions (also known as procedures or subroutines) are often represented as a black box: (the black box represents the function)

Until now the programs we have written in Go have used only one function:

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

- We will now begin writing programs that use more than one function

Your Second Function

```
func main() {
     xs := []float64{98,93,77,82,83}

     total := 0.0
     for _, v := range xs {
          total += v
     }
     fmt.Println(total / float64(len(xs)))
}
```

- This program computes the average of a series of numbers. Finding the average like this is a very general problem, so it's an ideal candidate for definition as a function.

- The average function will need to take in a slice of float64s and return one float64. Insert this before the main function:

```
func average(xs []float64) float64 {
     panic("Not Implemented")
}
```

- Functions start with the keyword func, followed by the function's name. The parameters (inputs) of the function are defined like this: name type, name type, …. Our function has one parameter (the list of scores) that we named xs. After the parameters we put the return type. Collectively the parameters and the return type are known as the function's signature.

- Finally, we have the function body which is a series of statements between curly braces. In this body we invoke a built-in function called panic which causes a run time error. (We'll see more about panic later in this chapter) Writing functions can be difficult so it's a good idea to break the process into manageable chunks, rather than trying to implement the entire thing in one large step.

- Now let's take the code from our main function and move it into our average function:

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

- Notice that we changed the fmt. Println to be a return instead. The return statement causes the function to immediately stop and return the value after it to the function that called this one. Modify main to look like this:

```
func main() {
     xs := []float64{98,93,77,82,83}
     fmt.Println(average(xs))
}
```

- Running this program should give you exactly the same result as the original. A few things to keep in mind:

• The names of the parameters don't have to match in the calling function. For example, we could have done this:

```
func main() {
     someOtherName := []float64{98,93,77,82,83}
     fmt.Println(average(someOtherName))
}
```

- And our program would still work.

• Functions don't have access to anything in the calling function. This won't work:

```
func f() {
     fmt.Println(x)
}
func main() {
     x := 5
     f()
}
```

We need to either do this:

```
func f(x int) {
     fmt.Println(x)
}
func main() {
     x := 5
     f(x)
}
```

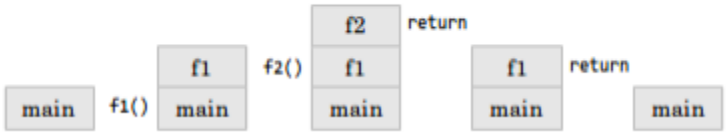
Or this:

```
var x int = 5
func f() {
     fmt.Println(x)
}
func main() {
     f()
}
```

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.

  - We can also name the return type:

```
func f2() (r int) {
    r = 1
    return
}
```

Returning Multiple Values

- Go is also capable of returning multiple values from a function:

```
func f() (int, int) {
     return 5, 6
}

func main() {
     x, y := f()
}
```

- Three changes are necessary: change the return type to contain multiple types separated by, change the expression after the return so that it contains multiple expressions separated by, and finally change the assignment statement so that multiple values are on the left side of the: = or =.

- Multiple values are often used to return an error value along with the result (x, err: = f ()), or a Boolean to indicate success (x, ok: = f ()).

Variadic Functions

There is a special form available for the last parameter in a Go function:

```
func add(args ...int) int {
     total := 0
     for _, v := range args {
          total += v
     }
     return total
}
func main() {
     fmt.Println(add(1,2,3))
}
```

- By using ... before the type name of the last parameter you can indicate that it takes zero or more of those parameters. In this case we take zero or more ints. We invoke the function like any other function except we can pass as many ints as we want.
- This is precisely how the fmt. Println function is implemented

```
func Println(a ...interface{}) (n int, err
error)
```

- The Println function takes any number of values of any type.
- We can also pass a slice of ints by following the slice with ...:

```
func main() {
     xs := []int{1,2,3}
     fmt.Println(add(xs...))
}
```

Closure

- It is possible to create functions inside of functions:

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

add is a local variable that has the type func (int, int) int (a function that takes two ints and returns an int).

```
func main() {
     x := 0
     increment := func() int {
          x++
          return x
     }
     fmt.Println(increment())
     fmt.Println(increment())
}
```

- increment adds 1 to the variable x which is defined in the main function's scope. This x variable can be accessed and modified by the increment function. This is why the first time we call increment we see 1 displayed, but the second time we call it we see 2 displayed.
- A function like this together with the non-local variables it references is known as a closure. In this case increment and the variable x form the closure.
- One way to use closure is by writing a function which returns another function which – when called – can generate a sequence of numbers. For example, here's how we might generate all the even numbers:

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

- makeEvenGenerator returns a function which generates even numbers. Each time it's called it adds 2 to the local i variable which – unlike normal local variables – persists between calls.

Recursion

- Finally, a function is able to call itself. Here is one way to compute the factorial of a number:

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

     return x * factorial(x-1)
}
```

- factorial calls itself, which is what makes this function recursive. In order to better understand how this function works, lets walk through factorial (2):

• Is x == 0? No. (x is 2)

• Find the factorial of x – 1

• Is x == 0? No. (x is 1)

• Find the factorial of x – 1

• Is x == 0? Yes, return 1.

• return 1 * 1

• return 2 * 1

- Closure and recursion are powerful programming techniques which form the basis of a paradigm known as functional programming. Most people will find functional programming more difficult to understand than an approach based on for loops, if statements, variables and simple functions.

Defer, Panic & Recover

- Go has a special statement called defer which schedules a function call to be run after the function completes. Consider the following example:

```
package main

import "fmt"

func first() {
     fmt.Println("1st")
}
func second() {
     fmt.Println("2nd")
}
func main() {
     defer second()
     first()
}
```

```
package main

import "fmt"

func first() {
     fmt.Println("1st")
}
func second() {
     fmt.Println("2nd")
}
func main() {
     defer second()
     first()
}
```

This program prints 1st followed by 2nd. Basically, defer moves the call to second to the end of the function:

```
func main() {
    first()
    second()
}
```

defer is often used when resources need to be freed in some way. For example, when we open a file we need to make sure to close it later. With defer:

```
f, _ := os.Open(filename)
defer f.Close()
```

This has 3 advantages:

(1) it keeps our Close call near our Open call so it's easier to understand,

(2) if our function had multiple return statements (perhaps one in an if and one in an else) Close will happen before both of them and

(3) deferred functions are run even if a run-time panic occurs.

Panic & Recover

- Earlier we created a function that called the panic function to cause a run time error. We can handle a run-time panic with the built-in recover function. recover stops the panic and returns the value that was passed to the call to panic. We might be tempted to use it like this:

```
package main

import "fmt"

func main() {
     panic("PANIC")
     str := recover()
     fmt.Println(str)
}
```

But the call to `recover` will never happen in this case because the call to `panic` immediately stops execution of the function. Instead we have to pair it with `defer`:

```
package main

import "fmt"

func main() {
     defer func() {
           str := recover()
           fmt.Println(str)
     }()
     panic("PANIC")
}
```

- A panic generally indicates a programmer error (for example attempting to access an index of an array that's out of bounds, forgetting to initialize a map, etc.) or an exceptional condition that there's no easy way to recover from. (Hence the name “panic”).

Problems
1. sum is a function which takes a slice of numbers and adds them together. What would its function signature look like in Go?
2. Write a function which takes an integer and halves it and returns true if it was even or false if it was odd. For example, half (1) should return (0, false) and half (2) should return (1, true).
3. Write a function with one variadic parameter that finds the greatest number in a list of numbers
. 4. Using makeEvenGenerator as an example, write a makeOddGenerator function that generates odd numbers.
5. 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). 6. What are defer, panic and recover? How do you recover from a run-time panic?