Revision

Primiative Data types

1. boolean type `bool`
2. numeric type
    1. integer `int`, `int8`, `int32`, `int64`
    2. un-signed integers `uint`,`uint8`,`uint16`,`uint32`,`uint64`
3. `byte` the same as `uint8`
4. floating type
    1. `float32`
    2. `float64`
5. string types `string`
6. Derived types
    1. Pointers
    2. Arrays
    3. Structures
    4. union-find types
    5. functions (*can be treated as a data type*)
    6. slice (*like arrays i.e. a slice of arrays*)
    7. Interface types
    8. Map type
    9. Channel type

Variables

The `var` statement declares a list of variables; as in function argument lists, the type is last.

A `var` statement can be at package or function level. We see both in this example.

```
package main

import "fmt"

var c, python, java bool
```

```
func main() {
    var i int
    fmt.Println(i, c, python, java)
}
```

```
0 false false false
```

Variables with initializers

A `var` declaration can include initializers, one per variable.

If an initializer is present, the type can be omitted; the variable will take the type of the initializer.

```
package main

import "fmt"
// notice here that the variables are global
var i, j int = 1, 2

func main() {
    var c, python, java = true, false, "no!"
    fmt.Println(i, j, c, python, java)
}
```

```
1 2 true false no!
```

Short variable declarations

Inside a function, the `:=` short assignment statement can be used in place of a `var` declaration with implicit type.

Outside a function, every statement begins with a keyword (`var`, `func`, and so on) and so the `:=` construct is not available.

```
package main

import "fmt"
// := construct can't be used outside functions
func main() {
    var i, j int = 1, 2
    k := 3
    c, python, java := true, false, "no!"

    fmt.Println(i, j, k, c, python, java)
}
```

Zero values

Zero values Variables declared without an explicit initial value are given their zero value.

The zero value is:

`0` for numeric types,

`false` for the boolean type, and

`""` (the empty string) for strings.

```
package main

import "fmt"

func main() {
    var i int
    var f float64
    var b bool
    var s string
    fmt.Printf("%v %v %v %q\n", i, f, b, s)
}
```

```
0 0 false ""
```

`var` delcars 1 or more variables. You can declare mutiple varables at once. Go will infer the type of initilized values.

Variables declared without a corresponding initialization are *zero-valued*. For example, the zero value for an int is 0.

```
package main

import "fmt"

func main() {

    var a = "initial"
    fmt.Println(a)

    var b, c int = 1, 2
    fmt.Println(b, c)

    var d = true
    fmt.Println(d)

    var e int
    fmt.Println(e)
```

```
    f := "apple"
    fmt.Println(f)

    var (
        alpha = 112
        beta  = 113
    )
    fmt.Println(alpha, beta)
}
```

```
initial
1 2
true
0
apple
112 113
```

More Examples

```
package main

import "fmt"

func main() {
    var integer int = 3
    str := "ahmed"
    float := 7.5
    boolean := true

    fmt.Printf("%T\n", integer)
    fmt.Printf("%T\n", str)
    fmt.Printf("%T\n", float)
    fmt.Printf("%T\n", boolean)
}
```

```
int
string
float64
bool
```

```
package main

import "fmt"

func main() {
    var operation string
```

```
    var x, y, z float64
    var correct_operation = true
    fmt.Printf("Enter the mathematical operation: ")
    fmt.Scanln(&operation)
    fmt.Printf("Enter the first operand: ")
    fmt.Scanln(&x)
    fmt.Printf("Enter the second operand: ")
    fmt.Scanln(&y)

    if operation == "+" {
        z = x + y
    } else if operation == "-" {
        z = x - y
    } else if operation == "/" && y != 0 {
        z = x / y
    } else if operation == "*" {
        z = x * y
    } else {
        correct_operation = false
    }
    if correct_operation {
        fmt.Println(x, operation, y, "=", z)
    } else {
        fmt.Println("The operation you have entered is incorrect")
    }
}
```

```
Enter the mathematical operation: +
Enter the first operand: 1
Enter the second operand: 2
1 + 2 = 3
```

Functions

A function can take zero or more arguments.

In this example, add takes two parameters of type int.

Notice that the type comes after the variable name.

```
package main

import "fmt"

func add(x int, y int) int {
    return x + y
```

```
}

func main() {
    fmt.Println(add(42, 13))
}
```

```
55
```

When two or more consecutive named function parameters share a type, you can omit the type from all but the last.

In this example, we shortened

```
x int, y int
```

to

```
x,y int
```

```
package main

import "fmt"

func add(x, y int) int {
    return x + y
}

func main() {
    fmt.Println(add(42, 13))
}
```

Multiple results

A function can return any number of results.

The `swap` function returns two strings.

```
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}
```

```
world hello
```

Call by reference and call by value

- Call by Value

  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

e.g.

```go
package main
import "fmt"
func swap(a,b int){
    temp := a
    a = b
    b = temp
    fmt.Println(a,b)
}
func main(){
    x := 1
    y := 2
    swap(x,y)
    fmt.Println(x,y)
}
```

```
2 1
1 2
```

- Call by reference

  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

```go
package main
import "fmt"
func swap(a,b *int){
    temp := *a
    *a = *b
    *b = temp
    fmt.Println(*a,*b)
}
func main(){
    x := 1
    y := 2
```

```
    swap(x,y)
    fmt.Println(x,y)
}
```

```
2 1
2 1
```

Veradic functions

Variadic functions can be called with any number of trailing arguments. For example, fmt.Println is a common variadic function. Here's a function that will take an arbitrary number of ints as arguments.

```
package main

import "fmt"

func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0

    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {

    sum(1, 2)
    sum(1, 2, 3)

    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

```
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

Here is another example where the funciton takes strings and joins them.

```
package main

import (
    "fmt"
    "strings"
```

```
)

func joinstr(elements ...string) string {
    return strings.Join(elements, "-")
}
func main() {
    fmt.Println(joinstr())
    fmt.Println(joinstr("A", "B"))
    fmt.Println(joinstr("A", "and", "B"))
    fmt.Println(joinstr("A", "and", "B", "and", "c"))
}
```

```
A-B
A-and-B
A-and-B-and-c
```

Arrays

an *array* is a numbered sequence of elements of a specific length. In typical Go code, slices are much more common; arrays are useful in some special scenarios.

Struct

A `struct` is a collection of fields.

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    fmt.Println(Vertex{1, 2})
}
```

```
{1 2}
```

Struct Fields

Struct fields are accessed using a dot.

```go
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    v.X = 4
    fmt.Println(v.X)
}
```

```
4
```

An example of using structs and slices

```go
package main

import "fmt"

type class struct {
    className string
    students  []student
}

type student struct {
    name   string
    rollNo int
    city   string
}

func main() {
    goerge := student{"Goerge", 35, "Newyork"}
    john := student{"Goerge", 25, "London"}

    students := []student{goerge, john}
    class := class{"firstA", students}
    fmt.Println(class)
}
```

```
{firstA [{Goerge 35 Newyork} {Goerge 25 London}]}
```

Exercises

Primitive Data and definite loops

1. In physics, a common useful equation for finding the position s of a body in linear motion at a given time t, based on its initial position s_0, initial velocity v_0, and rate of acceleration a, is the following:

$$s = s_0 + v_0 t + \frac{1}{2} a t^2$$

   Write code to declare variables for s_0,v_0,a, and t, and then write the code to compute s on the basis of these values.

```go
package main

import (
    "fmt"
)

func position(s_0, v_0, t, a float64) float64 {
    return (s_0 + v_0*t + (0.5 * a * t * t))
}
func main() {
    var s_0, v_0, t, a float64
    fmt.Print("Enter s_0: ")
    fmt.Scanln(&s_0)
    fmt.Print("Enter v_0: ")
    fmt.Scanln(&v_0)
    fmt.Print("Enter t: ")
    fmt.Scanln(&t)
    fmt.Print("Enter a: ")
    fmt.Scanln(&a)
    fmt.Print("The s position is ", position(s_0, v_0, t, a))
}
```

2. Write a `for` loop that produces the following output:

```
1 4 9 16 25 36 49 64 81 100
```

```go
package main

import "fmt"

const end = 10

func sol_1() {

    for i := 0; i < end; i++ {
```

```
        fmt.Printf("%d ", (i+1)*(i+1))
    }
}

func sol_2() {
    num := 1
    diff := 3
    for i := 0; i < end; i, diff = i+1, diff+2 {
        fmt.Printf("%d ", num)
        num = num + diff
    }
}
func main() {
    sol_1()
    fmt.Println()
    sol_2()
}
```

```
1 4 9 16 25 36 49 64 81 100
1 4 9 16 25 36 49 64 81 100
```

3. The Fibonacci numbers are a sequence of integers in which the first two elements are 1, and each following element is the sum of the two preceding elements. The mathematical definition of each kth Fibonacci number is the following:

$$F(k) = \begin{cases} F(k-1) + F(k-2) & k > 2 \\ 1 & k \leq 2 \end{cases}$$

The first 12 Fibonacci numbers are `1 1 2 3 5 8 13 21 34 55 89 144`

```
package main

import "fmt"

func fib(n int) int {
    if n <= 2 {
        return 1
    }
    return fib(n-1) + fib(n-2)
}
func main() {
    for i := 1; i < 13; i++ {
        fmt.Printf("%d ", fib(i))
    }
}
```

4. Write nested `for` loops to produce the following output:

```
*****
*****
*****
*****
```

```go
package main

import "fmt"

func main() {
    const rows = 4
    const cols = 4
    for i := 0; i < rows; i++ {
        for j := 0; j < cols; j++ {
            fmt.Printf("*")
        }
        fmt.Println()
    }
}
```

5. Write nested `for` loops to produce the following output:

```
*
**
***
****
*****
```

```go
package main

import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        for j := 0; j <= i; j++ {
            fmt.Print("*")
        }
        fmt.Println()
    }
}
```

6. Write nested `for` loops to produce the following output:

```
1
22
333
4444
55555
```

```
666666
7777777
```

```go
package main

import "fmt"

func main() {
    for i := 1; i < 8; i++ {
        for j := 0; j < i; j++ {
            fmt.Print(i)
        }
        fmt.Println()
    }
}
```

7. Write nested `for` loops to produce the following output:

```
    1
   2
  3
 4
5
```

8. Write nested `for` loops to produce the following output:

```
    1
   22
  333
 4444
55555
```

Functions

1. Write a method called `print_numbers` that accepts a maximum number as an argument and prints each number from 1 up to that maximum, inclusive, boxed by square brackets. For example, consider the following calls:

```
print_numbers(15)
print_numbers(5)
```

These calls should produce the following output:

```
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15]

[1] [2] [3] [4] [5]
```

2. Write a method called `print_powers_of2` that accepts a maximum number as an argument and prints each power of 2 from 20 (1) up to that maximum

power, inclusive. For example, consider the following calls:

```
print_powers_of2(3)
print_powers_of2(10)
```

These calls should produce the following output:

```
1 2 4 8
1 2 4 8 16 32 64 128 256 512 1024
```

Structs

Arrays and Slices

1. Write a program that scans the element values of an array from user and pass it to a method that separates even and odd elements

```
array[0] = 50
array[1] = 11
array[2] = 744
array[3] = 101
array[4] = 5
50 is an even element whose index is 0
11 is an odd element whose index is 1
744 is an even element whose index is 2
101 is an odd element whose index is 3
5 is an odd element whose index is 4
```

```
package main

import "fmt"

func main() {
    // define a fixed-size array
    const len = 5
    var a [len]int

    for i := 0; i < len; i++ {
        fmt.Printf("array[%d] = ", i)
        fmt.Scanln(&a[i])
    }
    for i := 0; i < len; i++ {
        if a[i]%2 == 0 {
            fmt.Printf("%d is an even element whose index is %d \n", a[i], i)
        } else {
            fmt.Printf("%d is an odd element whose index is %d\n", a[i], i)
        }
    }
```

```
}
```

2. Solve the (1.) using slices

```
array[0] = 5
array[1] = 4
array[2] = 11
array[3] = 222
array[4] = 10
Even part : [4 222 10]
Odd part : [5 11]
```

```
package main

import "fmt"

func main() {
    var a [5]int
    var even_part []int
    var odd_part []int
    for i := 0; i < len(a); i++ {
        fmt.Printf("array[%d] = ", i)
        fmt.Scanln(&a[i])
    }
    for i := 0; i < len(a); i++ {
        if a[i]%2 == 0 {
            fmt.Printf("%d is an Even element whose index is %d\n",
            a[i],i)
            even_part = append(even_part, a[i])
        } else {
            fmt.Printf("%d is an Odd element whose index is %d\n",
             a[i], i)
            odd_part = append(odd_part, a[i])
        }
    }
    fmt.Println("Even part:", even_part)
    fmt.Println("Odd part:", odd_part)
}
```

3. Write a program that asks the user to fill a data of new person
   - Person is a struct
   - Process of entering a new Person is a loop
   - if user enters 1, user shall create a new Person, otherwise the program ends

```
Enter 1 to add a new Person, press any key to end program:
1
```

```
Enter Person's id: 01
Enter Person's name: FirstName LastName
Enter Person's age: 24
New Person add
Enter 1 to add a new Person, press any key to end program:
2
Program ended
```

```go
package main

import (
    "bufio"
    "fmt"
    "os"
)

type Person struct {
    id, name string
    age      int
}

func main() {
    var persons []Person
    fmt.Println(len(persons))
    fmt.Println(cap(persons))
    scanner := bufio.NewScanner(os.Stdin)

    for input := ""; true; input = "" {
        fmt.Print("For adding a new Person enter 1 or to the end to
        program press any key\n")
        fmt.Scanln(&input)
        fmt.Println("you have entered:", input)
        if input != "1" {
            break
        }
        persons = append(persons, func() Person {
            var id, name string
            var age int
            fmt.Print("Enter The person's id: ")
            fmt.Scanln(&id)
            fmt.Print("Enter the person's name: ")
            scanner.Scan()
            name = scanner.Text()
            fmt.Print("Enter the person's age: ")
            fmt.Scanln(&age)
            return Person{id, name, age}
```

```
        }())
    }

    fmt.Println("Length of the persons slice:", len(persons))
    fmt.Println("Capacity of the persons slice:", cap(persons))
    func(persons_slice []Person) {
        for i := 0; i < len(persons_slice); i++ {
            fmt.Println("--------")
            fmt.Printf("Person %d\n", i)
            fmt.Printf("ID: %s \t Name: %s \t Age: %d\n",
             persons_slice[i].id, persons_slice[i].name,
             persons_slice[i].age)
            fmt.Println("--------")
        }
    }(persons)
}
```

4. Write a method called `last_index_of` that accepts an array of integers and an integer value as its parameters and returns the last index at which the value occurs in the array. The method should return –1 if the value is not found. For example, in the array `[74, 85, 102, 99, 101, 85, 56]`, the last index of the value `85` is `5`.

5. Write a method called `range` that returns the range of values in an array of integers. The range is defined as 1 more than the difference between the maximum and minimum values in the array. For example, if an array called `list` contains the values `[36, 12, 25, 19, 46, 31, 22]`, the call of range(list) should return 35 (46 - 12 + 1). You may assume that the array has at least one element.

6. Write a method called `count_in_range` that accepts an array of integers, a minimum value, and a maximum value as parameters and returns the count of how many elements from the array fall between the minimum and maximum (inclusive). For example, in the array `[14, 1, 22, 17, 36, 7, -43, 5]`, for minimum value 4 and maximum value 17, there are four elements whose values fall between 4 and 17.

7. Write a method called `is_sorted` that accepts an array of real numbers as a parameter and returns true if the list is in sorted (nondecreasing) order and false otherwise. For example, if arrays named list1 and list2 store `[16.1, 12.3, 22.2, 14.4]` and `[1.5, 4.3, 7.0, 19.5, 25.1, 46.2]` respectively, the calls `is_sorted(list1)` and `is_sorted(list2)` should return `false` and `true` respectively. Assume the array has at least one element. A one-element array is considered to be sorted.

8. Write a method called `stdev` that returns the standard deviation of an array of integers. Standard deviation is computed by taking the square

root of the sum of the squares of the differences between each element and the mean, divided by one less than the number of elements. (It's just that simple!) For example, if the array passed contains the values `[1, -2, 4, -4, 9, -6, 16, -8, 25, -10]`, your method should return approximately `11.237`. More concisely and mathematically, the standard deviation of an array a is written as follows:

$$stdev(a) = \sqrt{\frac{\sum_{i=0}^{a.length-1} (a[i] - average(a))^2}{a.length - 1}}$$

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

9. Write a method called `kth_largest` that accepts an integer k and an array a as its parameters and returns the element such that k elements have greater or equal value. If k = 0, return the largest element; if k = 1, return the second-largest element, and so on. For example, if the array passed contains the values `[74, 85, 102, 99,101, 56, 84]` and the integer `k` passed is `2`, your method should return `99` because there are two values at least as large as `99` (`101` and `102`). Assume that $0 \leq k < a.\text{length}$. (Hint: Consider sorting the array or a copy of the array first.)

10. Write a method called `median` that accepts an array of integers as its parameter and returns the median of the numbers in the array. The median is the number that appears in the middle of the list if you arrange the elements in order. Assume that the array is of odd size (so that one sole element constitutes the median) and that the numbers in the array are between 0 and 99 inclusive. For example, the median of `[5, 2, 4, 17, 55, 4, 3, 26, 18, 2, 17]` is `5` and the median of `[42, 37, 1, 97, 1, 2, 7, 42, 3, 25, 89, 15, 10, 29, 27]` is `25`