Go String Formatting

```go
type point struct {
    x, y int
}

func main() {

    p := point{ x: 1,  y: 2}
    //For example, this prints an instance of our point struct.
    fmt.Printf( format: "%v\n", p)  // {1 2}

    //If the value is a struct, the %+v variant will include the struct's field names.
    fmt.Printf( format: "%+v\n", p) // {x:1 y:2}

    //The %#v variant prints a Go syntax representation of the value, i.e.  the source code snippet that would produce that value.
    fmt.Printf( format: "%#v\n", p)  // main.point{x:1, y:2}

    //To print the type of a value, use %T.
    fmt.Printf( format: "%T\n", p) // main.point

    // Formatting booleans is straight-forward.
    fmt.Printf( format: "%t\n",  a: true) // true

    //There are many options for formatting integers. Use %d for standard, base-10 formatting.
    fmt.Printf( format: "%d\n",  a: 123) // 123

    //This prints a binary representation.
    fmt.Printf( format: "%b\n",  a: 14) // 1110

    //This prints the character corresponding to the given integer.
    fmt.Printf( format: "%c\n",  a: 33) // !

    //%x provides hex encoding.
    fmt.Printf( format: "%x\n",  a: 456) // 1c8

    //There are also several formatting options for floats. For basic decimal formatting use %f.
    fmt.Printf( format: "%f\n",  a: 78.9) // 78.900000

    //%e and %E format the float in (slightly different versions of) scientific notation.
    fmt.Printf( format: "%e\n",  a: 123400000.0) // 1.234000e+08
    fmt.Printf( format: "%E\n",  a: 123400000.0) // 1.234000E+08

    //For basic string printing use %s.
    fmt.Printf( format: "%s\n",  a: "\"string\"") // "string"

    //To double-quote strings as in Go source, use %q.
    fmt.Printf( format: "%q\n",  a: "\"string\"") // "\"string\""

    //As with integers seen earlier, %x renders the string in base-16, with two output characters per byte of input.
    fmt.Printf( format: "%x\n",  a: "hex this") // 6865782074686973

    //To print a representation of a pointer, use %p.
    fmt.Printf( format: "%p\n", &p) // 0xc042060080

    //When formatting numbers you will often want to control the width and precision of theresulting figure. To specify the width
    //of an integer, use a number after the % in the verb. By default the result will be right-justified and padded with spaces.
    fmt.Printf( format: "|%6d|%6d|\n",  a: 12, 345) // |    12|   345|

    //You can also specify the width of printed floats, though usually you'll also want to restrict the
    //decimal precision at the same time with the width.precision syntax.
    fmt.Printf( format: "|%6.2f|%6.2f|\n",  a: 1.2, 3.45) // |  1.20|  3.45|

    //To left-justify, use the - flag.
    fmt.Printf( format: "|%-6.2f|%-6.2f|\n",  a: 1.2, 3.45) // |1.20  |3.45  |

    //You may also want to control width when formatting strings, especially to ensure that
    // they align in table-like output. For basic right-justified width.
    fmt.Printf( format: "|%6s|%6s|\n",  a: "foo", "b") // |   foo|     b|

    //To left-justify use the - flag as with numbers.
    fmt.Printf( format: "|%-6s|%-6s|\n",  a: "foo", "b") // |foo   |b     |

    //So far we've seen Printf, which prints the formatted string to os.Stdout.
    //Sprintf formats and returns a string without printing it anywhere.
    s := fmt.Sprintf( format: "a %s",  a: "string")
    fmt.Println(s) // a string

    //You can format+print to io.Writers other than os.Stdout using Fprintf.
    fmt.Fprintf(os.Stderr,  format: "an %s\n",  a: "error")
```

https://gobyexample.com/string-formatting

Go Strings

cheatsheet

String contains

```
fmt.Println(strings.Contains( s: "New York",  substr: "ew")) //true
fmt.Println(strings.ContainsAny( s: "team",  chars: "i")) //false
fmt.Println(strings.ContainsAny( s: "Hello",  chars: "aeiou&y")) //true
fmt.Println(strings.HasPrefix( s: "Hello World",  prefix: "Hello")) //true
fmt.Println(strings.HasSuffix( s: "Hello World",  suffix: "World")) //true
```

Regex

```
matched, err := regexp.MatchString( pattern: "Pop.*",  s: "Marry Poppins")
fmt.Println( a: "Matched:", matched, "Error:", err) //Matched: true Error: <nil>
matched, err = regexp.MatchString( pattern: "Rex.*",  s: "Hello World")
fmt.Println( a: "Matched:", matched, "Error:", err) //Matched: false Error: <nil>
matched, err = regexp.MatchString( pattern: "(invalid regexp",  s: "any string")
fmt.Println( a: "Matched:", matched, "Error:", err) //Matched: false Error: error
                                    // parsing regexp: missing closing ): `(bad regexp`
```

Random String

```
func GenerateRandomString(length int) string {
    b := make([]byte, 64) //or increase
    _, err := rand.Read(b)
    if err != nil {
        fmt.Printf( format: "error %s", err)
    }
    return base32.StdEncoding.EncodeToString(b)[:length]
}
```

Split Strings

```
s := strings.Split( s: "192.168.0.1:8000",  sep: ":")
ip, port := s[0], s[1]
fmt.Println(ip, port)
```

Read (small) file into lines

```
content, err := ioutil.ReadFile( filename: "README.md")
if err != nil {
    fmt.Println(err)
}
lines := strings.Split(string(content),  sep: "/n")
fmt.Println(lines)
```

Other String Functions

```
strings.Count( s: "tests",  substr: "s") //2
strings.Index( s: "test",  substr: "s") //true
strings.Join([]string{"a", "b"},  sep: "-") //a-b
strings.Repeat( s: "x",  count: 5)
strings.Replace( s: "good morning",  old: "morning",  new: "evening",  n: -1)
strings.Replace( s: "1aaa",  old: "a",  new: "x",  n: 1) //1xaa
strings.ToLower( s: "MARK TWAIN")
strings.ToUpper( s: "mark twain")
```

Split words into array

```
theString := "one two three four."
theArray := strings.Fields(theString) //[one two three four.]

x := func(c rune) bool {
    return !unicode.IsLetter(c) //<- replace this with whatever
}
anotherExample := strings.FieldsFunc( s: `the Bitcoin Test Network. lnd has several pluggable
back-end chain services including btcd (a full-node)`,x)
//[the Bitcoin Test Network lnd has several pluggable back end chain services including btcd a full node]
```

Go Beginners part 2

Range

```go
nums := []int{1, 2, 3}
sum := 0
for _, num := range nums {
    sum += num
}
fmt.Printf( format: "Sum: %d\n", sum)

for i, num := range nums {
    if num == 2 {
        fmt.Println( a: "index:", i)
    }
}
```

```go
kvs := map[string]string{"city": "Seoul", "country": "Korea"}
for k, v := range kvs {
    fmt.Printf( format: "%s -> %s\n", k, v)
}
```

Functions

```go
func addition(a int, b float64) float64 {
    return float64(a) + b
}

func plusPlus(x, y, z string) {
    fmt.Println(x + y + z)
}

func multipleReturnFunc() (int, int) {
    return 3, 7
}

func main() {

    addition( a: 10, b: 2.23)
    plusPlus( x: "a", y: "b", z: "c")
    a, b := multipleReturnFunc()

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

Variadic Functions

```go
// Variadic functions can be called with any number of trailing arguments.
// fmt.Println is a common variadic function.
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}
func main() {
    sum( nums: 1, 2)
    sum( nums: 1, 2, 3)

    //If you already have multiple args in a slice,
    // apply them to a variadic function using func(slice...)
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

Closures

```go
func main() {
    gen := makeFibGen()
    for i := 0; i < 10; i++ {
        fmt.Println(gen())
    }
}

func makeFibGen() func() int {
    f1 := 0
    f2 := 1
    return func() int {
        f2, f1 = (f1 + f2), f2
        return f1
    }
}
```

Structs

```go
type person struct {
    name string
    age  int
}

func main() {

    fmt.Println(person{ name: "Bob",  age: 20})

    fmt.Println(person{name: "Alice", age: 30})

    //Omitted fields will be zero-valued.
    fmt.Println(person{name: "Fred"})

    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    s.age = 51
}
```

https://gobyexample.com

Go Beginners

Variables

```go
var a = "initial"
var b, c int = 1, 2
var d = true
var e int
f := "short"

fmt.Println(a,b,c,d,e,f)
//initial 1 2 true 0 short
```

Constants

```go
const s string = "constant"
const n = 500000000

const d = 3e20 / n
// A numeric constant has no type
// until it's given one, such as
// by an explicit cast
fmt.Println(int64(d))
```

Loops

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

for j := 7; j <= 9; j++ {
    fmt.Println(j)
}

for {
    fmt.Println( a: "loop")
    break
}
```

Switch

```go
i := 2
switch i {
case 1:
    fmt.Println( a: "one")
case 2:
    fmt.Println( a: "two")
case 3:
    fmt.Println( a: "three")
}

switch time.Now().Weekday() {
case time.Saturday, time.Sunday:
    fmt.Println( a: "It's the weekend")
default:
    fmt.Println( a: "It's a weekday")
}

t := time.Now()
switch {
case t.Hour() < 12:
    fmt.Println( a: "It's before noon")
default:
    fmt.Println( a: "It's after noon")
}
```

Arrays

```go
// an array is a numbered sequence
// of elements OF A SPECIFIC LENGTH!
var a [5]int
fmt.Println( a: "emp:", a)

a[4] = 100
fmt.Println( a: "set:", a)
fmt.Println( a: "get:", a[4])
fmt.Println( a: "len:", len(a))

b := [5]int{1, 2, 3, 4, 5}
fmt.Println( a: "dcl:", b)

var twoD [2][3]int
for i := 0; i < 2; i++ {
    for j := 0; j < 3; j++ {
        twoD[i][j] = i + j
    }
}
```

Slices

```go
s := make([]string, 3)
s[0] = "a"
s[1] = "b"
s[2] = "c"

s = append(s, elems: "d")
s = append(s, elems: "e", "f")
fmt.Println( a: "apd:", s)

c := make([]string, len(s))
copy(c, s)
fmt.Println( a: "cpy:", c)
// This gets a slice of the
// elements s[2], s[3], and s[4].
x := s[2:5]

//This slices up to (but excluding) s[5].
l := s[:5]

t := []string{"g", "h", "i"}
fmt.Println( a: "dcl:", t)
```

Maps

```go
//To create an empty map, use the builtin make
m := make(map[string]int)
m["k1"] = 7
m["k2"] = 13

fmt.Println( a: "map:", m)

v1 := m["k1"]

fmt.Println( a: "len:", len(m))

delete(m, key: "k2")

// The optional second return value when getting a value
// from a map indicates if the key was present in the map.
_, prs := m["k2"]
fmt.Println( a: "prs:", prs)

// Another option of initializing maps
n := map[string]int{"foo": 1, "bar": 2}
fmt.Println( a: "map:", n)
```

https://gobyexample.com