

# **More Loops, Overflow, Debugging**

02-201

# For KthDigit() on HW1

Check out

```
math.Abs( )  
math.Pow10( )
```

in the math package (import “math”)

# Reading Package Documentation

The screenshot shows a browser window displaying the Go Programming Language documentation for the `math` package. The URL in the address bar is `https://golang.org/pkg/math/`. The page title is "math - The Go Programming Language". The top navigation bar includes links for "Documents", "Packages", "The Project", "Help", "Blog", "Play", and "Search". On the left, there's a sidebar with links for "Overview", "Index", and "Subdirectories". The main content area has a heading "Package math" and a code snippet showing the import statement `import "math"`. Below this, the "Overview" section is active, containing the text "Package math provides basic constants and mathematical functions.". The "Index" section is also visible, listing various mathematical functions like `Abs`, `Acos`, etc.

## Package math

```
import "math"
```

[Overview](#)  
[Index](#)  
[Subdirectories](#)

### Overview ▾

Package math provides basic constants and mathematical functions.

### Index ▾

Constants

```
func Abs(x float64) float64
func Acos(x float64) float64
func Acosh(x float64) float64
func Asin(x float64) float64
func Asinh(x float64) float64
func Atan(x float64) float64
func Atan2(y, x float64) float64
func Atanh(x float64) float64
func Cbrt(x float64) float64
func Ceil(x float64) float64
func Copysign(x, y float64) float64
func Cos(x float64) float64
func Cosh(x float64) float64
func Dim(x, y float64) float64
func Erf(x float64) float64
func Erfc(x float64) float64
func Exp(x float64) float64
func Exp2(x float64) float64
func Exp2m1(x float64) float64
func Floor(x float64) float64
func Gamma(x float64) float64
func Hypot(x, y float64) float64
func Ilogb(x float64) int
func Ilogbf(x float64) int
func Ilogbl(x float64) int
func Ldexp(x, n float64) float64
func Log(x float64) float64
func Log10(x float64) float64
func Log2(x float64) float64
func Modf(x float64) (float64, float64)
func Pow(x, y float64) float64
func Signbit(x float64) bool
func Sin(x float64) float64
func Sinh(x float64) float64
func Sqrt(x float64) float64
func Tan(x float64) float64
func Tanh(x float64) float64
```

# Warm-Up (Review of Loops)

**Exercise:** Write a Go function that takes integers  $k$  and  $n$  and prints  $k^n$  for every number between 0 and  $n$ .

# Warm-Up (Review of Loops)

**Exercise:** Write a Go function that takes integers  $k$  and  $n$  and prints  $k^n$  for every number between 0 and  $n$ .

```
import "math"

func PrintPowers(k, n int) int {
    for i:=0; i<=n; i++ {
        fmt.Println(math.Pow(k, i))
    }
}
```

**Error!** `math.Pow()` can only take `float64` variable types.

# Converting Between Types

Use `type(expression)` to convert `expression` to `type`:

```
var a float64 = 3.2          // ok!
var c int = int(a)           // ok: 3.2 converted to 3.0
var g int = int(3.2)         // ERROR! can't convert 3.2 to int
var f uint = uint(c)         // ok: c is converted to uint
var ok bool = bool(0)         // ERROR! can't convert int to bool
```

# Conversion Challenges

```
var u int = -70  
var q uint = uint(u)
```

**Exercise:** What value does q have?

# Conversion Challenges

```
var u int = -70  
var q uint = uint(u)
```

**Exercise:** What value does q have?

**Answer:** It depends on your computer, but probably  
 $q = 18446744073709551546$

# Conversion Challenges

```
var u int = -70  
var q uint = uint(u)
```

**Exercise:** What value does q have?

**Answer:** It depends on your computer, but probably  
 $q = 18446744073709551546$

```
var i uint = 10  
for ; i>=0; i = i-1 {  
    fmt.Println(i)  
}
```

**Exercise:** How many times does this loop run?

# Conversion Challenges

```
var u int = -70  
var q uint = uint(u)
```

**Exercise:** What value does q have?

**Answer:** It depends on your computer, but probably  
 $q = 18446744073709551546$

```
var i uint = 10  
for ; i>=0; i = i-1 {  
    fmt.Println(i)  
}
```

**Exercise:** How many times does this loop run?

**Answer:** It never stops!

# Variables Have Ranges

Type	Min	Max
int	- 9223372036854775808	9223372036854775807
uint	0	18446744073709551615
float64	-1.797693134862315708145274 237317043567981e+308	1.7976931348623157081452742373 17043567981e+308

If a uint gets too large, it “wraps around” to 0; if it is negative, it “wraps around” to the largest possible uint.

# Variables Have Ranges

Type	Min	Max
int	- 9223372036854775808	9223372036854775807
uint	0	18446744073709551615
float64	-1.797693134862315708145274 237317043567981e+308	1.7976931348623157081452742373 17043567981e+308

If a uint gets too large, it “wraps around” to 0; if it is negative, it “wraps around” to the largest possible uint.

```
var i int = 9223372036854775807
fmt.Println(i+1)
```

**Exercise:** What is printed?

# Variables Have Ranges

Type	Min	Max
int	- 9223372036854775808	9223372036854775807
uint	0	18446744073709551615
float64	-1.797693134862315708145274 237317043567981e+308	1.7976931348623157081452742373 17043567981e+308

If a uint gets too large, it “wraps around” to 0; if it is negative, it “wraps around” to the largest possible uint.

```
var i int = 9223372036854775807
fmt.Println(i+1)
```

**Exercise:** What is printed?

**Answer:** -9223372036854775808

# Variables Have Ranges

Type	Min	Max
int	- 9223372036854775808	9223372036854775807
uint	0	18446744073709551615
float64	-1.797693134862315708145274 237317043567981e+308	1.7976931348623157081452742373 17043567981e+308

If a uint gets too large, it “wraps around” to 0; if it is negative, it “wraps around” to the largest possible uint.

```
var i int = 9223372036854775807
fmt.Println(i+1)
```

**Exercise:** What is printed?

**Answer:** -9223372036854775808

**Lesson:** if you have very big or very small numbers, you have to do something special to prevent this “overflow”.

# Printing a “Rectangle”

# **Print Rectangle Problem:** *Draw a rectangle of symbols.*

- **Input:** Integers  $r$  and  $c$ .
  - **Output:** A rectangle of “#” symbols with  $r$  rows and  $c$  columns.

# Think: How do we do this?

# Nested Loops: Printing a “Rectangle”

```
// PrintRect(r, c) prints r x c
// rectangle of # symbols
func PrintRect(r, c int) {
    for i:=1; i<=r; i++ {
        for j:=1; j<=c; j++ {
            fmt.Print("#")
        }
        fmt.Println(" ")
    }
}
```

Note that nothing is returned by this function.

# Print a Diamond

**Print Diamond Problem:** *Draw a diamond of “#” symbols.*

- **Input:** An odd integer  $n$ .
- **Output:** A diamond of the form below having height  $n$ .

( $n = 9$ )

```
#  
# # #  
# # # # #  
# # # # # # #  
# # # # # # # #  
# # # # # # #  
# # # # #  
#
```

English



Pseudocode



Go

# Print a Diamond

**Print Diamond Problem:** *Draw a diamond of “#” symbols.*

- **Input:** An odd integer  $n$ .
- **Output:** A diamond of the form below having height  $n$ .

( $n = 9$ )

```
#  
# # #  
# # # # #  
# # # # # # #  
# # # # # # # #  
# # # # # # # #  
# # # # # #  
# # #
```

**Think:** How can we solve this problem? (Think top-down...)

English



Pseudocode



Go

# Print a Diamond

**Print Diamond Problem:** *Draw a diamond of “#” symbols.*

- **Input:** An odd integer  $n$ .
- **Output:** A diamond of the form below having height  $n$ .

( $n = 9$ )

```
#  
# # #  
# # # # #  
# # # # # # #  
# # # # # # # #  
# # # # # # #  
# # # # # #  
# # #  
#
```

**PrintDiamond( $n$ )**

**PrintTriangle( $n/2 + 1$ )**

**PrintInvertedTriangle( $n/2$ )**

# Print a Diamond

**Print Diamond Problem:** *Draw a diamond of “#” symbols.*

- **Input:** An odd integer  $n$ .
- **Output:** A diamond of the form below having height  $n$ .

( $n = 9$ )

```
#  
# # #  
# # # # #  
# # # # # # #  
# # # # # # # #  
# # # # # # #  
# # # # # #  
# # #  
#
```

**Note:** We have greatly simplified the problem already.

```
PrintDiamond(n)  
PrintTriangle(n/2 + 1)  
PrintInvertedTriangle(n/2)
```

# Print a Diamond

**Print Diamond Problem:** Draw a diamond of “#” symbols.

- **Input:** An odd integer  $n$ .
- **Output:** A diamond of the form below having height  $n$ .

( $n = 9$ )

```
#  
# # #  
# # # # #  
# # # # # # #  
# # # # # # # #  
# # # # # # #  
# # # # #  
# # #  
#
```

**Note:** We have greatly simplified the problem already.

```
PrintDiamond(n)  
  PrintTriangle(n/2 + 1)  
  PrintInvertedTriangle(n/2)
```

**Exercise:** Write pseudocode for PrintTriangle().

# Implementing PrintTriangle()

```
func PrintTriangle(k int) {
    var size int = 1 // num of symbols to print
    for row := 0; row < k; row++ {
        // print space to indent row
        for i := 0; i <= k - row; i++ {
            fmt.Print(" ")
        }
        // print correct number of symbols in a row
        for i := 1; i <= size; i++ {
            fmt.Print("#")
        }
        size = size + 2 // next row prints two more
        fmt.Println("") // start new line
    }
}
```

# Implementing PrintTriangle()

```
func PrintTriangle(k int) {
    var size int = 1 // num of symbols to print
    for row := 0; row < k; row++ {
        // print space to indent row
        for i := 0; i <= k - row; i++ {
            fmt.Print(" ")
        }
        // print correct number of symbols in a row
        for i := 1; i <= size; i++ {
            fmt.Print("#")
        }
        size = size + 2 // next row prints two more
        fmt.Println("") // start new line
    }
}
```

**Think:** Are there any ways in which this code could be improved?

# Think Style: A Better PrintTriangle()

```
func BetterPrintTriangle(k int) {
    for row := 0; row < k; row++ {
        // print space to indent row
        for i := 0; i <= k - row; i++ {
            fmt.Print(" ")
        }
        // print correct number of symbols in a row
        for i := 1; i <= 2*row-1; i++ {
            fmt.Print("#")
        }
        fmt.Println("") // start new line
    }
}
```

Shorter is not necessarily better, but this avoids an unnecessary additional variable.

# Best: Reusable PrintRow() Function

```
func BestPrintTriangle(k int) {
    for row := 0; row < k; row++ {
        PrintRow(k-row, 2*row+1)
    }
}

func PrintRow(numOfSpaces, numOfSymbols int) {
    for i:=numOfSpaces; i>=1; i-- {
        fmt.Print(" ")
    }
    for i:=1; i<=numOfSymbols; i++ {
        fmt.Print("#")
    }
    fmt.Println()
}
```

A little longer, but more modular and easier to read.