Functions & Variables

02-201 / 02-601

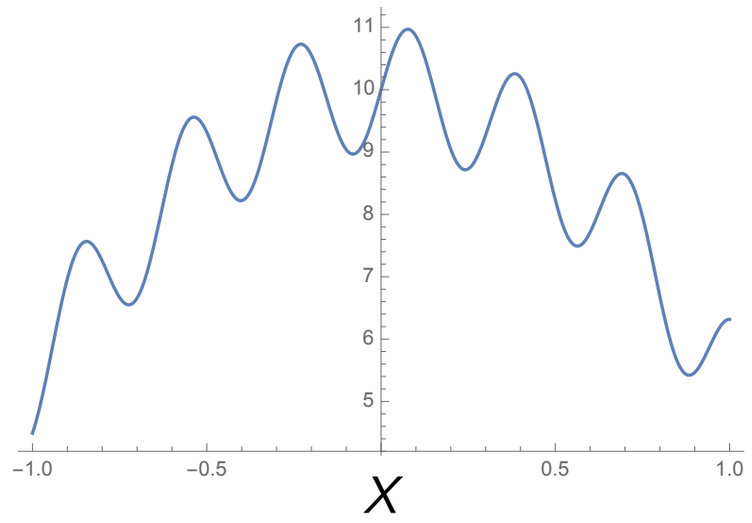
Functions

- Functions in calculus give a rule for mapping input values to an output:

$$f : \mathbb{R} \to \mathbb{R}$$

- May take multiple inputs:

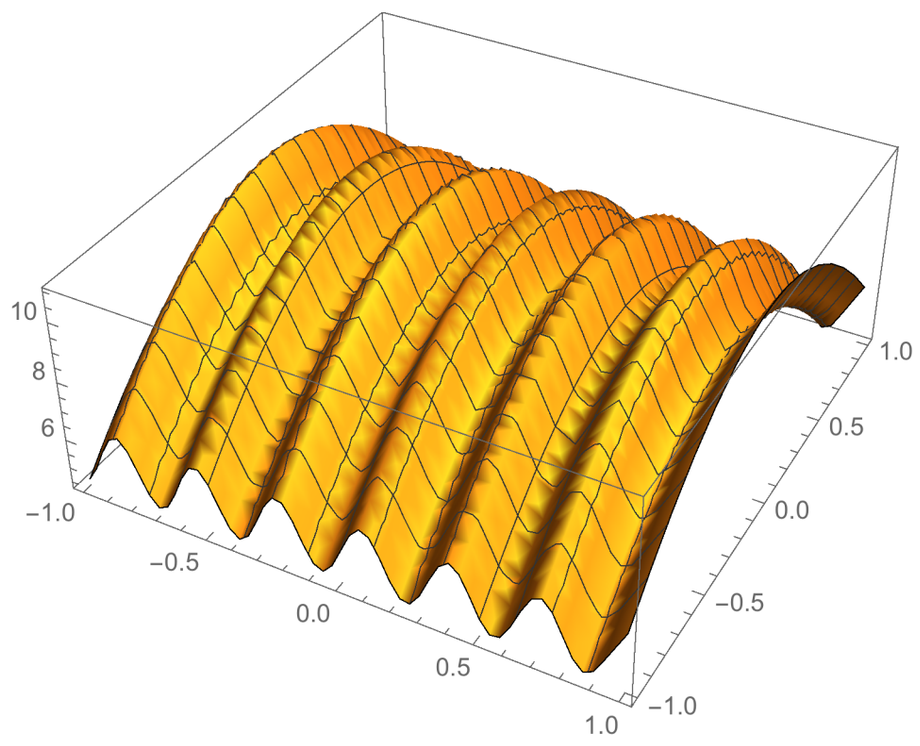
$$g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$



f(x) = sin(20x) + 10cos(x)

- Functions encapsulate some expression and allow it to be reused.

- Functions play the same central role in programming too.



g(x, y) = sin(20x) + 10cos(y)

Functions in Go

$$\text{square} : \mathbb{Z} \to \mathbb{Z}$$

Function name

Parameter

This **int** gives the type of the input parameter

```
func square(x int) int {
    return x*x
}
```

{ } are used for grouping related statements in Go.

func, **return**, and **int** are *keywords:* special words defined by the Go language.

This **int** after the () gives the type of the value returned by the function

Functions Can Do A Lot

- Functions can *call* other functions that have been previously defined:

```
func forthPower(x int) int {
    return square(x) * square(x)
}
```

a function call

- Functions can take multiple parameters:

```
func P(a int, b int, c int, x int) int {
    return a*square(x) + b*x + c
}
```

- Functions can have side effects: they can affect the screen, network, disk, etc:

This is what makes them so useful!

```
func print4thPower(x int) {
    fmt.Println(x, " to the fourth is ", forthPower(x))
}
```

A call to the builtin fmt.Println function to print text to the screen

A Longer Example: Greatest Common Divisor

Function *name*

Return type
(comes after the () and before the {)

Function *signature*

```
// gcd(a,b) computes the greatest common
// divisor of a and b
func gcd(a int, b int) int {
    if a == b {
        return a
    }
    if a > b {
        return gcd(a - b, b)
    } else {
        return gcd(a, b - a)
    }
}
```

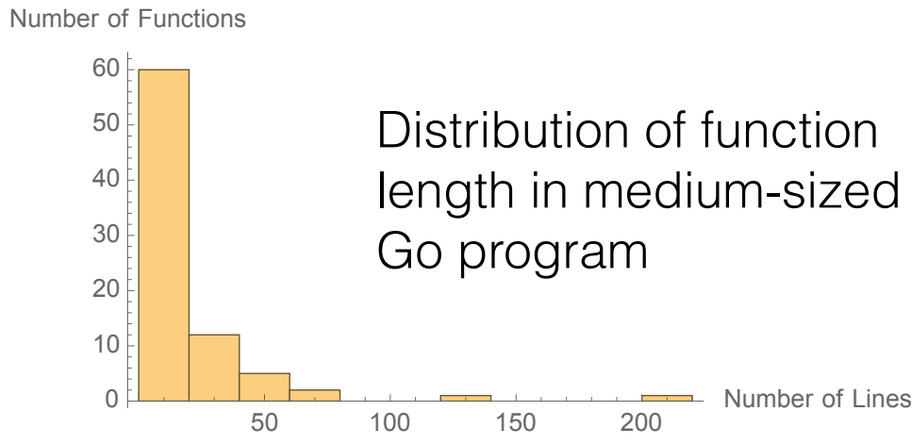
Return statement: stops executing the function and gives its value

Function *body*

Function *call*
(this one is *recursive* because it calls the function it's in)

Functions are the Paragraphs of Programming

- Your program will typically consist of a long sequence of functions.
- These functions will call one another to make the program do whatever it is designed to do.
- Just as with paragraphs, functions should be well written:
  1. They should do one thing only.
  2. Comment + signature ≈ “topic sentence”
  3. Functions should be short.
  4. They should have a good “interface” with the rest of your program.



Distribution of function length in medium-sized Go program

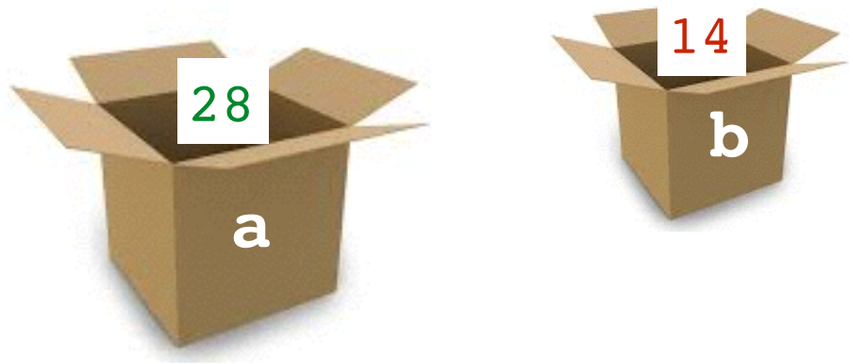
Compute e example

```
package main
import "fmt"

func factorial(n int) int {
    var out = 1
    for i := 1; i <= n; i++ {
        out = out * i
    }
    return out
}


func approxE(k int) float64 {
    var out = 1.0
    for i := 1; i <= k; i++ {
        out = out + 1.0 / float64(factorial(i))
    }
    return out
}

func main() {
    fmt.Println(approxE(10))
}
```

Variables

Variables hold values (just as in calculus).

Can think of these values as stored in boxes in the computer with the name of the variable equal to the label of the box.



Variables

```
func gcd(a int, b int) int {
    if a == b {
        return a
    }
    if a > b {
        a = a - b
        return gcd(a, b)
    } else {
        b = b - a
        return gcd(a, b)
    }
}
```

Assignment statement: changes the value of a variable

```
varName = EXPRESSION
```

Example assignment statements:

```
cat = 2 + f(20)
Pi = 3.1459
y = m*x + b
a = 2*b + 3*b + 7*a
```

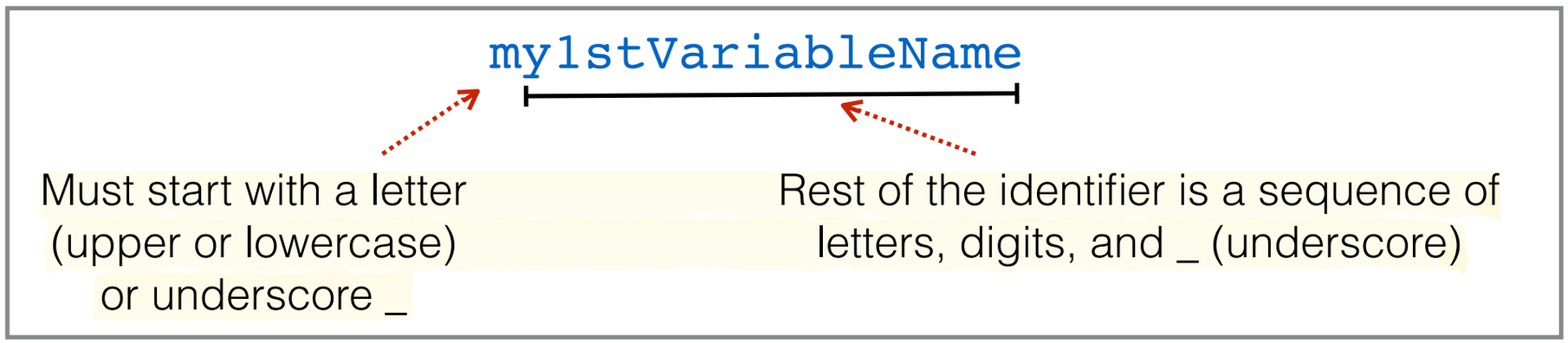
This uses the *current* value of a.

If a = 28, and b = 42, then this statement sets a to

2*42 + 3*42 + 7*28 = 406

Identifiers: Rules for Naming Variables and Functions

- Functions, variables, and other things in your program will have names.
- These names are called *identifiers.*
- There are some rules for what an identifier can look like:



- Identifier can't be a reserved Go **keyword** (e.g. **func**, **if**, **return**, …)

Example identifiers:

```
aLongVariableName
a_long_variable_name
A312789
_dont
i
Aμ
Rαμ
```

Convention in Go: use camelCaseLikeThis for long identifiers.

Convention in Go: don't start identifiers with _: these are "reserved" for special purposes.

Go supports unicode letters in identifiers; this is neat, but **tip**: only use letters you and others can easily type.

BAD identifiers:

```
3littlePigs
func
if
Once.Again
```

Creating New Variables

```
func gcd(a int, b int) int {
    var c int
    if a == b {
        return a
    }
    if a > b {
        c = a - b
        return gcd(c, b)
    } else {
        c = b - a
        return gcd(a, c)
    }
}
```

var statement creates (*declares)* new variables.

You can create several variables at once:

```
var a,b,c int
```

You can also write this as:

```
var (
    a int
    b int
    c int
)
```

You must declare a variable before you can use it.
Parameter names count as declarations.

You can assign values when you create a variable:

If you don't, the variable will have its default "0" value.

```
var c int = 10
var a,b, c int = -2, 0, 2
var (
    a int = -2
    b int = 0
    c int = 2
)
```

Real Valued Variables

Use float64 instead of **int** to create a real valued variable.

We'll see a lot more about variables of different types soon.

```
var c float64 = 3.14159
var e float64 = 2.718
```

Style: Choosing Good Names

- Use descriptive names: `numOfPeople` is better than `n`
- **Variable names are case sensitive:**
  `n` is different than `N` and `numOfPeople` is not the same as `numofpeople`.
- Use i, j, k for integers that don't last long in your program.
- Use camelCase to connect words together.
- Good function names usually involve verbs:

```
printFullName
encodeSingleRead
writeCounts
listBuckets
```

- Avoid the verb "compute" though
- Start names with lowercase letter (we'll see a required exception to this later)
- Don't use abbreviations (Bad: `nerr`, `ptf_name`, …)

Scope: How Long do Variables Last

Variables persist from when they are created until the end of the innermost {} block that they are in.

```
func gcd(a int, b int) int {
      var c int
      if a == b {
            return a
      }
      if a > b {
            c = a - b
            return gcd(c, b)
      } else {
            c = b - a
            return gcd(a, c)
      }
}
```

variable c created

variable c destroyed

```
func gcd(a int, b int) int {
      if a == b {
            return a
      }
      var c int
      if a > b {
            c = a - b
            return gcd(c, b)
      } else {
            c = b - a
            return gcd(a, c)
      }
}
```

variable c created

variable c destroyed

```
func gcd(a int, b int) int {
      if a == b {
            return a
      }
      if a > b {
            var c int = a - b
            return gcd(c, b)
      } else {
            var c int = b - a
            return gcd(a, c)
      }
}
```

c created

A ***different*** c created

c destroyed

2nd c destroyed

```
func gcd(a int, b int) int {
      var c int
      if a == b {
            return a
      }
      var c int
      if a > b {
            c = a - b
            return gcd(c, b)
      } else {
            c = b - a
            return gcd(a, c)
      }
}
```

Error!
Variable c declared twice in same scope ({} block)

Concept of “Scope” is Borrowed From Math

$$a = \sum_{i=1}^{n} \frac{1}{i}$$

i only defined inside the sum

$$a = i + \sum_{i=1}^{n} \frac{1}{i}$$

either this is a mistake, or this is a **different** *i*

$$x \leq 3 \wedge \forall x \exists y . x = y$$

Scope of x

Scope of y

$$\sum_{i=0}^{n} \sum_{j=i}^{n} ij$$

Scope of i

Scope of j

What's the error here?

```
func gcd(a int, b int) int {
    if a == b {
        return a
    }
    if a > b {
        var c int
        c = a - b
        return gcd(c, b)
    } else {
        c = b - a
        return gcd(a, c)
    }
}
```

Error!
Variable c doesn't exist in this block

Scope of Parameter Variables

```
func gcd(a int, b int) int {
    var c int
    if a == b {
        return a
    }
    if a > b {
        c = a - b
        return gcd(c, b)
    } else {
        c = b - a
        return gcd(a, c)
    }
}
```

variables a and b created

variables a and b destroyed

When you call a function:

```
gcd(12,6)
```

the parameter variables are created and set to the values you are passing into the function.

Defining a Variable During Assignment

You can use the := assignment operator to simultaneously define a variable and give it a value

Lets you omit the **int** and **var.**
(This will be more useful when we see variables that aren't integers.)

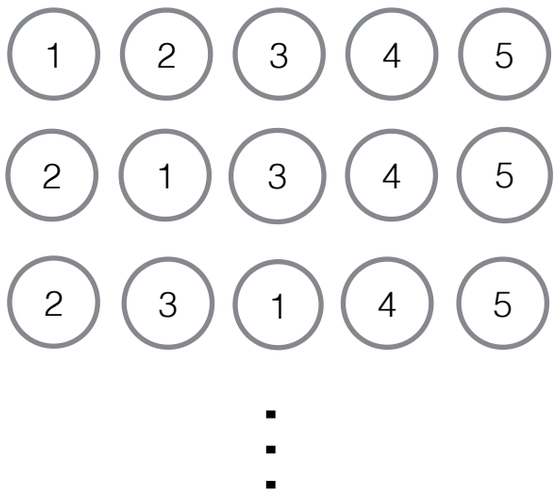
These two code snippets are equivalent:

```
func gcd(a int, b int) int {
    if a == b {
        return a
    }
    if a > b {
        c := a - b
        return gcd(c, b)
    } else {
        c := b - a
        return gcd(a, c)
    }
}
```

```
func gcd(a int, b int) int {
    if a == b {
        return a
    }
    if a > b {
        var c int = a - b
        return gcd(c, b)
    } else {
        var c int = b - a
        return gcd(a, c)
    }
}
```

Example: Permutations & Combinations

Number of ways to order *n* items:

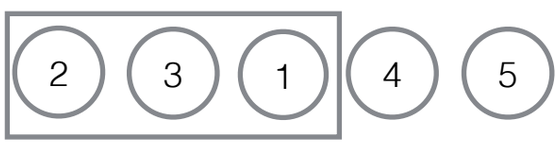


n choices for the first item
n - 1 choices for the second item
n - 2 choices for the third item

⇒ $n(n\text{-}1)(n\text{-}2)\ldots(2)(1) = n!$

Number of ways to order *n* items is *n*!

Number of ways to choose *k* items from a set of *n* items:



$$\Longrightarrow \frac{n!}{k!(n-k)!} = \binom{n}{k}$$

n! orderings of the whole sequence
k! equivalent orderings of the items that fall in the box
(n-k)! equivalent orderings of items that fall outside the box

Number of ways to chose k items from *n* items is $\binom{n}{k}$

How a function is called

```
func factorial(n int) int {
    var f,i int = 1, 0
    for i = 1; i <= n; i++ {
        f = f * i
    }
    return f
}
```

n=10 for factorial(n) call
n=4 for factorial(k) call
n=6 for factorial(n-k) call

These are different variables that happen to have the same name

```
func nChooseK(n int, k int) int {
    var numerator, denominator int
    numerator = factorial(n)
    denominator = factorial(k) * factorial(n-k)
    return numerator / denominator
}

func print10Choose4() {
    var answer int
    answer = nChooseK(10,4)
    fmt.Println("10 choose 4 =", answer)
}
```

n=10; k=4
numerator=0; denominator=0
numerator=3628800; denominator=0
denominator=24 * 720 = 744

Function Question 1

```
func factorial(n int) int {
    var f,i int = 1, 0
    for i = 1; i <= n; i++ {
        f = f * i
    }
    n = 24
    return f
}
```

How does this red statement change what is printed by the program?

```
func nChooseK(n int, k int) int {
    var numerator, denominator int
    numerator = factorial(n)
    denominator = factorial(k) * factorial(n-k)
    return numerator / denominator
}

func print10Choose4() {
    var answer int
    answer = nChooseK(10,4)
    fmt.Println("10 choose 4 =", answer)
}
```

Go Functions Can Return Multiple Values

$$\text{translatePoint} : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$$

```
func translatePoint(x int, y int, deltaX int, deltaY int) (int,int) {
    return x+deltaX, y+deltaY
}
```

Return statement now
has comma-separated list of values
to return

Return types listed in
parentheses

Another example:

```
func scalePoint(x int, y int, alpha int) (int,int) {
    return alpha*x, alpha*y
}
```

Can group together types if they are the same

These **int**s say that the parameters will be integers.

```
func translatePoint(x int, y int, deltaX int, deltaY int) (int,int) {
    return x+deltaX, y+deltaY
}
```

is equivalent to

```
func translatePoint(x, y, deltaX, deltaY int) (int,int) {
    return x+deltaX, y+deltaY
}
```

Rule: type of a parameter is the next type listed

Returning Multiple Values

```
func translatePoint(x, y, deltaX, deltaY int) (int,int) {
    return x+deltaX, y+deltaY
}
```

```
func scalePoint(x, y, alpha int) (int,int) {
    return alpha*x, alpha*y
}
```

Assignment statements can assign to multiple variables at once

```
func xlateAndScale(x, y, deltaX, deltaY, alpha int) (int,int) {
    x, y = translatePoint(x,y,deltaX,deltaY)
    return scalePoint(x,y,alpha)
}
```

Return values can “pass along” multiple values

These two functions do the same thing

```
func xlateAndScale(x, y, deltaX, deltaY, alpha int) (int,int) {
    x, y = translatePoint(x,y,deltaX,deltaY)
    x, y = scalePoint(x,y,alpha)
    return x,y
}
```

Additional Details About Functions

- Functions can take 0 parameters:

```
func pi() int {
    return 3
}
```

In this case, they are called using `pi()`. You still need the `()` following the function name.

- Functions can return 0 return values:

```
func printInt(a int) {
    fmt.Println(a)
}
```

You indicate this by not providing any return types.

The main() function

- Your program starts by running the main() function.

```
package main
import "fmt"

func main() {
    fmt.Println("GCD =", gcd(42,28))
}

func gcd(a int, b int) int {
     if a == b {
          return a
     }
     if a > b {
          return gcd(a-b, b)
     } else {
          return gcd(a, b-a)
     }
}
```

Your program should start with this statement; don't worry what it means for now.

Your program starts running here with a function call main()

main() can then call any other functions you've defined (or that are defined for you by the system).

Note: you can call a function that is defined later in the file: Go will find it for you.

- main() shouldn't take any parameters or return any thing.

Summary

- Functions are the “paragraphs” of programming.
- They let you extend the kinds of things that the computer can do
  - defining a function is like creating a new operation the computer can perform.
- Functions should be short, have well-defined behavior, and have a small “interface” with the rest of your program.
- Top-down design: break your big problem into smaller problems and write functions to solve those smaller problems (e.g.: e → sum and factorial)