

Functions & Variables

02-201 / 02-601

Functions

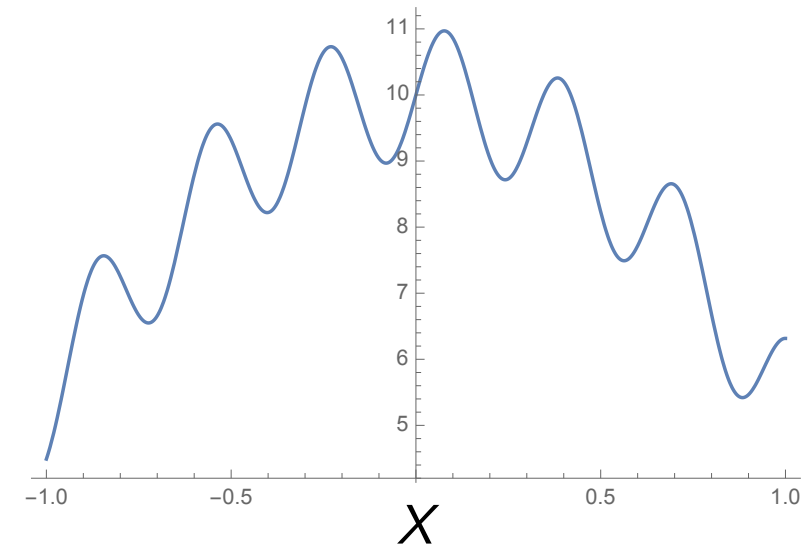
- Functions in calculus give a rule for mapping input values to an output:

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

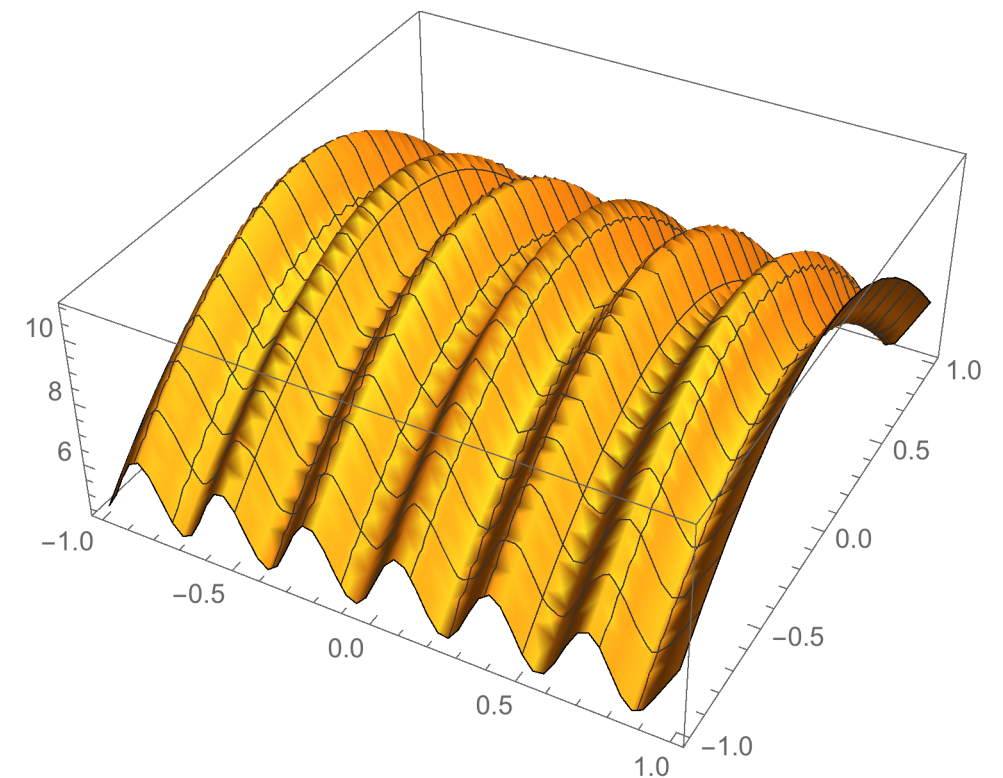
- May take multiple inputs:

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

- Functions encapsulate some expression and allow it to be reused.
- Functions play the same central role in programming too.



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



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

Functions in Go

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

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  
}
```

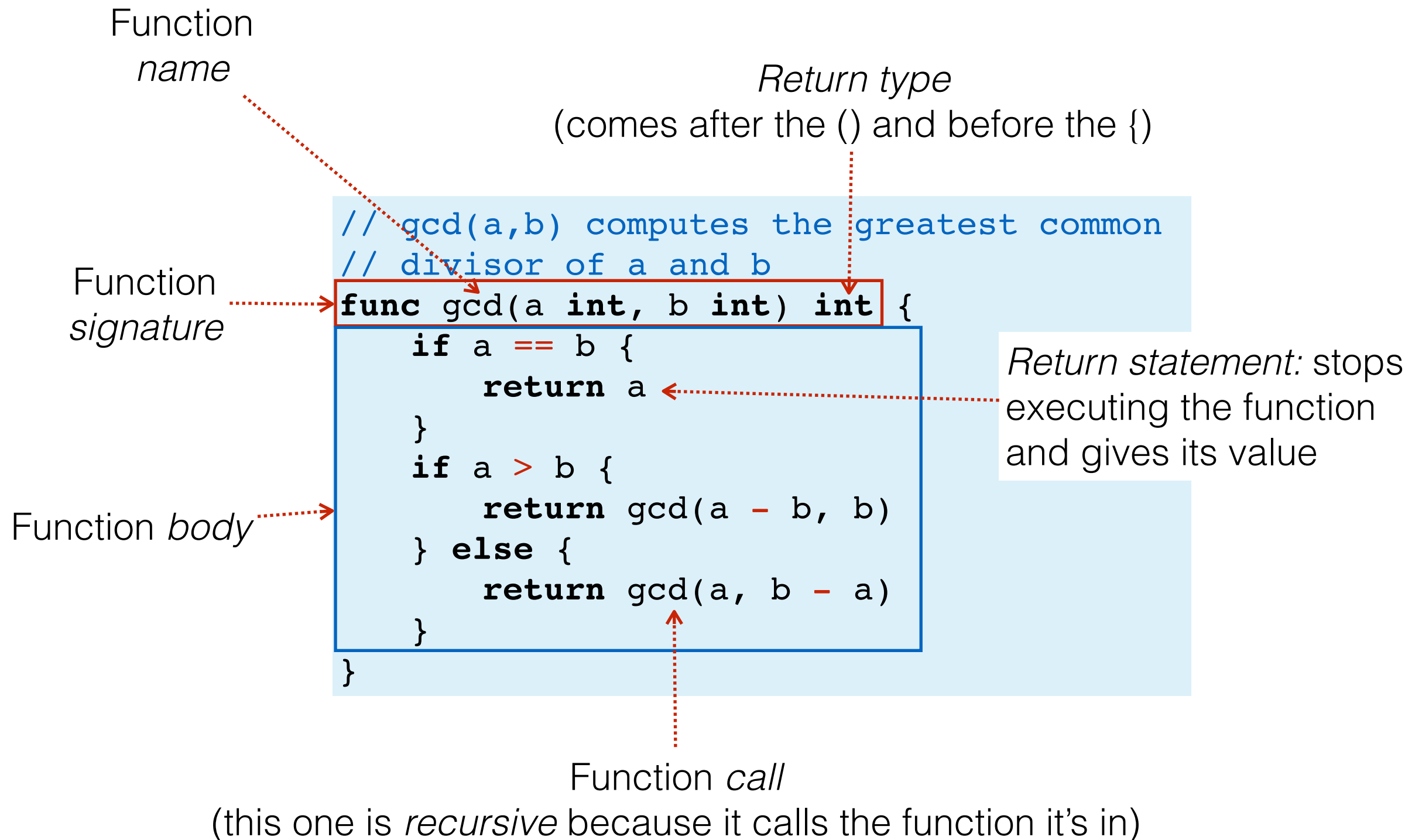
- Functions can have side effects: they can affect the screen, network, disk, etc:

← *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*

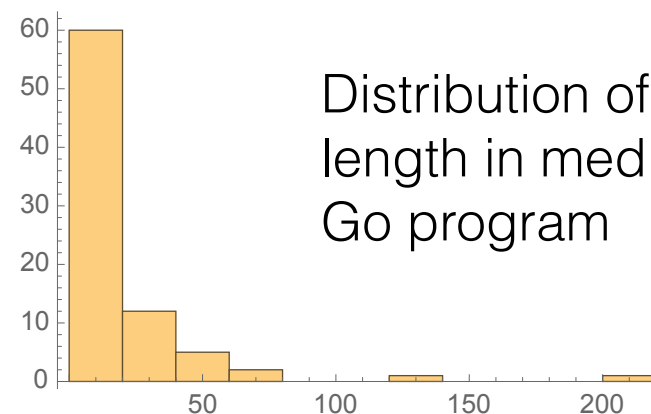
A Longer Example: Greatest Common Divisor



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 \approx “topic sentence”
 3. Functions should be short.
 4. They should have a good “interface” with the rest of your program.

Number of Functions



Distribution of function length in medium-sized Go program

```
217 // readReferenceFile() reads the sequences in the gripped multifasta file with
218 // the given name and returns them as a slice of strings.
219 func readReferenceFile(fastaFile string) []string {
220 // open the .gz fasta file that is the reference
221 log.Println("Reading Reference File...")
222 infasta, err := os.Open(fastaFile)
223 if err != nil {
224     log.Fatalf("Couldn't open fasta file %s", fastaFile)
225 }
226 defer infasta.Close()
227
228 // wrap the gzip reader around it
229 in, err := gzip.NewReader(infasta)
230 if err != nil {
231     log.Fatalf("Couldn't open gripped file %s", fastaFile)
232 }
233 defer in.Close()
234
235 out := make([]string, 0, 10000000)
236 cur := make([]string, 0, 100)
237
238 scanner := bufio.NewScanner(in)
239 for scanner.Scan() {
240     line := strings.TrimSpace(strings.ToUpper(scanner.Text()))
241     if len(line) == 0 {
242         continue
243     }
244     if line[0] == byte('>') {
245         if len(cur) > 0 {
246             out = append(out, strings.Join(cur, ""))
247             cur = make([]string, 0, 100)
248         } else {
249             cur = append(cur, line)
250         }
251     } else {
252         cur = append(cur, line)
253     }
254 }
255 if scanner.Err() != nil {
256     log.Fatalf("Couldn't finish reading reference")
257 }
258 return out
259 }
260
261 // countKmersInReference() reads the given reference file (gripped multifasta)
262 // and constructs a kmer hash for it that maps kmers to distributions of next
263 // characters.
264 func countKmersInReference(k int, fastaFile string) KmerHash {
265     seqs := readReferenceFile(fastaFile)
266     hash := newKmerHash()
267
268     log.Printf("Counting %d-mer transitions in reference file...\n", k)
269     for _, s := range seqs {
270         if len(s) <= k {
271             continue
272         }
273         contextKmer := stringToKmer(s[:k])
274         for i := 0; i < len(s)-k; i++ {
275             info := hash[contextKmer]
276             next := acgt[s[i+k]]
277             // seeing something in the reference gives us a count of seenThreshold
278             info.next[next] = seenThreshold
279             hash[contextKmer] = info
280         }
281         contextKmer = shiftKmer(contextKmer, next)
282     }
283     return hash
284 }
285
286 // Encoding
287 // =====
288 // contextWeight() is a weight transformation function that will change the
289 // distribution weights according to the function for real contexts. If the
290 // count is too small, it returns the pseudocount; if the count is big enough
291 // it returns observationWeight * the distribution value.
292 func contextWeight(charIdx int, dist [len(ALPHA)]KmerCount) uint64 {
293     if dist[charIdx] >= seenThreshold {
294         return observationWeight * uint64(dist[charIdx])
295     } else {
296         return pseudoCount
297     }
298 }
299
300 // defaultWeight() is a weight transformation function for the default
301 // distribution. It returns the weight unchanged.
302 func defaultWeight(charIdx int, dist [len(ALPHA)]KmerCount) uint64 {
303     return uint64(dist[charIdx])
304 }
305
306 // intervalFor() returns the interval for the given character (represented as a
307 // 2-bit encoded base) according to the given distribution (transformed by the
308 // given weight transformation function).
309 func intervalFor(
310     letter byte,
311     dist [len(ALPHA)]KmerCount,
312     weightOf WeightTransform,
313 ) (a uint64, b uint64, total uint64) {
314
315     letterIdx := int(letter)
316     for i := 0; i < len(dist); i++ {
317         w := weightOf(i, dist)
318         total += w
319         if i <= letterIdx {
320             b += w
321         }
322     }
323     return
324 }
325
326 // intervalForDefault() computes the interval for the given character using the
327 // default interval
328 func intervalForDefault(letter byte) (a uint64, b uint64, total uint64) {
329     letterIdx := int(letter)
330     for i := 0; i < len(defaultInterval); i++ {
331         w := uint64(defaultInterval[i])
332         total += w
333         if i <= letterIdx {
334             b += w
335         }
336     }
337     return
338 }
```

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`

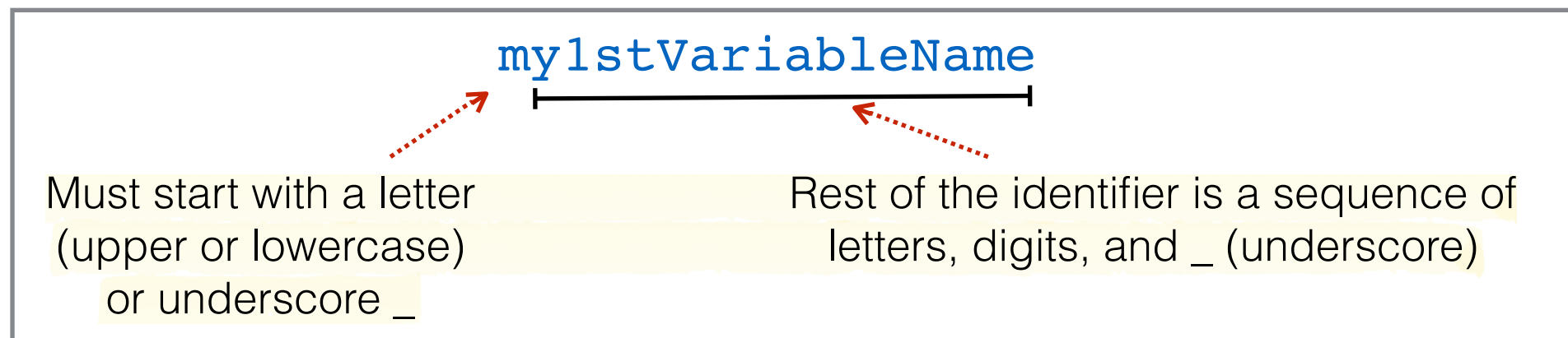
This uses the *current* value of a.

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`
`Αμ`
`℞μ`

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.

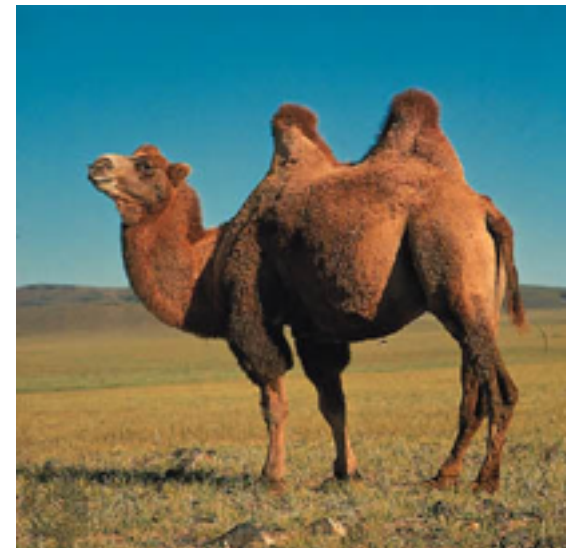
We'll see a lot more about variables of different types soon.

```
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 destroyed

2nd c destroyed

c created

A *different* c created

```
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

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

i only defined inside the sum

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

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

$$x \leq 3 \wedge \forall x \overbrace{\exists y. x = y}^{\text{Scope of } y}$$

Scope of x

$$\sum_{i=0}^n \sum_{j=i}^n \overbrace{ij}^{\text{Scope of } i}$$

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.)

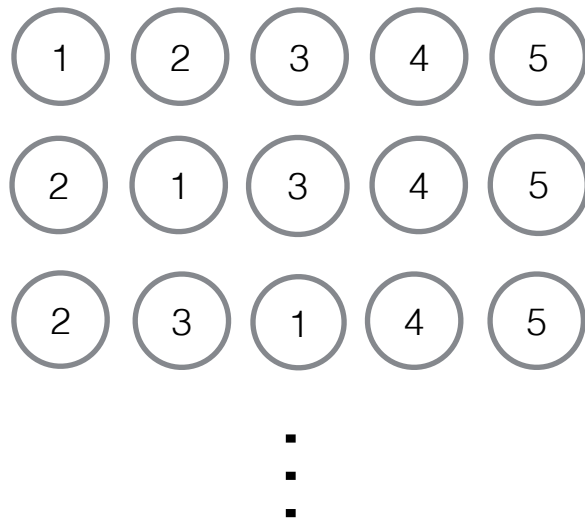
These two code snippets are equivalent:

```
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

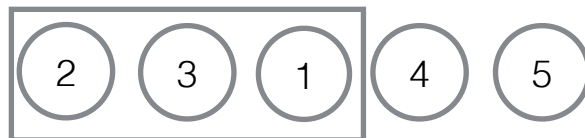
$n - 2$ choices for the third item

⋮

$$\Rightarrow n(n-1)(n-2)\dots(2)(1) = n!$$

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

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



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

$n!$ 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  
}
```

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

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

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  
}
```

```
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)  
}
```

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

Go Functions Can Return Multiple Values

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

```
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

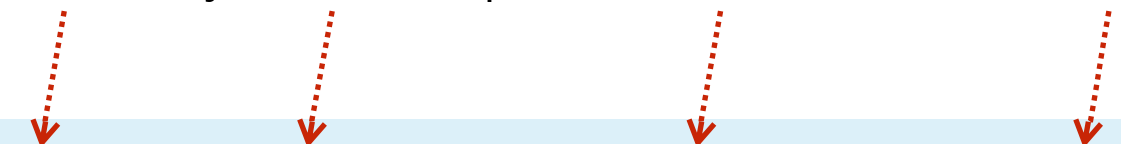
Return types listed in
parentheses

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 **ints** 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.: $e \rightarrow \text{sum}$ and factorial)