Carl Kingsford, 02-201, Fall 2015

Lecture 5: Variables and Functions

Structure of a Go program

- The Go program is stored in a plain text file
- A Go program consists of a sequence of statements, usually 1 per line
- Go programs start with the statement `package main` as the first line

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.

Creating variables

The `var` statement creates (declares) new variables:

```
var NAME int
```

This creates a new “box” named “NAME” that can hold an integer (stay tuned for how to create boxes that can hold other things)

You must declare a variable before you can use it.

> Test yourself: Where is the value for this variable stored?

Examples:

```go
var c int
var bobsAge int
var salary int
```

You can set the value of a variable when you declare it:

```go
var c int = 2
var bobsAge int = 40
var salary int = 111
```

If you don't set the value, Go will assign it a "0" value.

Varibles that can hold different kinds of values

In the statement:

```go
var a int = 32
```

The `int` is called the type of the variable. This means that this variable can contain only integers.

If you want a variable to hold a real number, you use the type `float64`:

```go
var a float64 = 3.14159
```

We'll see soon the types that Go supports. For now, we need only `int` and `float64`.

> **Test yourself!** Where do you think the number "64" comes from in `float64`?

Creating multiple variables at once

You can create several variables at once to save typing or to group related variables together:

```go
var a,bobsAge, salary int = -2, 0, 2
```

This can also be written as:

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

Changing the value of a variable

The value of a variable can be changed anytime with an assignment statement:

```
VARIABLE = VALUE
```

Examples:

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

You should think of `=` as ←, which would be a better symbol, but is hard to type. Assignment is *not* the same as "equals": you are changing the value of a variable. The following statement is perfectly fine:

```
    a = 2*a
```

even though it doesn't make sense mathematically. It means you are changing `a` to have the value of 2 times its current value.

How to name variables

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:

- Must start with a letter (upper or lowercase) or underscore _
- The rest of the identifier is a sequence of letters, digits, and _ (underscore)

- Can't be a special keyword in Go (e.g. `var`)

Examples:

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

BAD identifiers:

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

Variable names are case sensitive: `n` is different than `N` and `numOfPeople` is not the same as `numofpeople` .

> Conventions in Go:
>
> - use `camelCaseLikeThis` for long identifiers.
> - 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.

Use descriptive names: `numOfPeople` is better than `n`

Use `i` , `j` , `k` for integers that don't last long in your program.

Functions

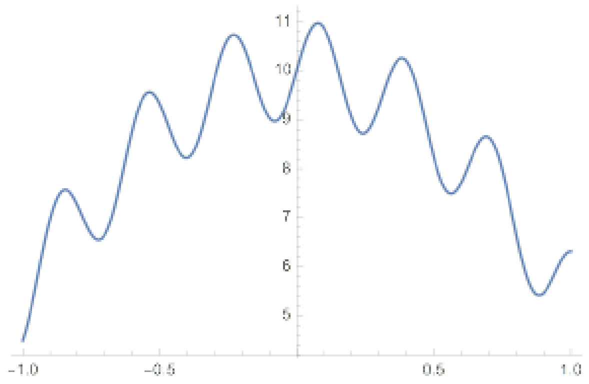
Functions in calculus give a rule for mapping input values to an output:

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

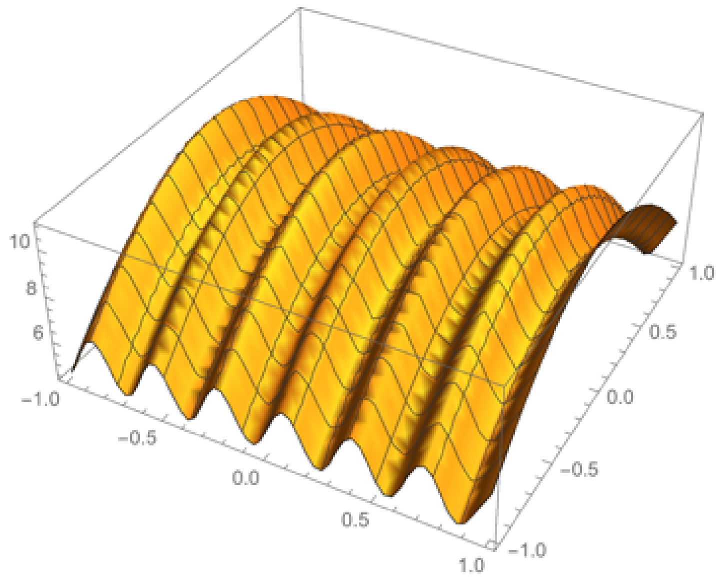
Calculus function may take multiple inputs:

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

Functions encapsulate some expression and allow it to be reused:



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



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

Functions play the same central role in programming too.

Functions in Go

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

The first line is the signature of the function. It contains:

- `func` starts a new function
- `square` is our name for this function. Can be any indentifer
- Inside the `()` we list the parameters (aka arguments) of the function.
- Following the `()' we list the return type of the function: why kind of value will this function compute.
- Then there is an opening `{' that starts the body of the function.

The signature must be on a single line in the file.

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

Functions can do a lot

Functions can call other functions that have been previously defined:

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

The `square(x)` is a function call.

Functions can take mulitple parameters:

```go
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.

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

(We'll see how to understand this soon.)

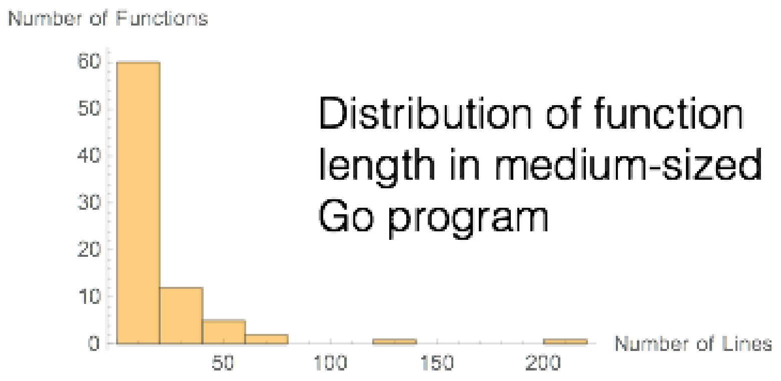
Functions are the Paragraphs of Programming

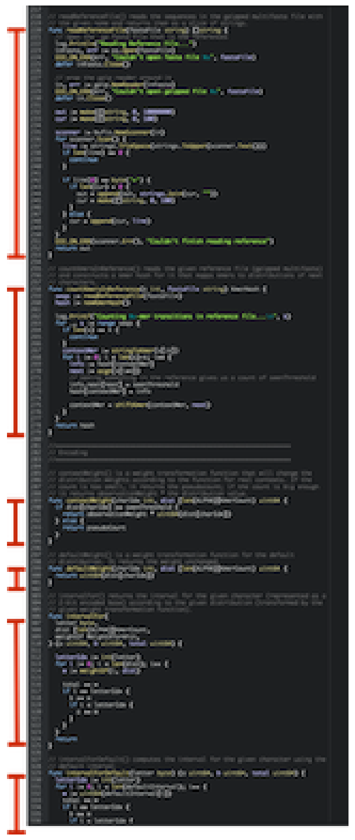
Your program will typically consist of a long sequence of functions.

These functions will call one another to make the program do whatever it is designed to do.

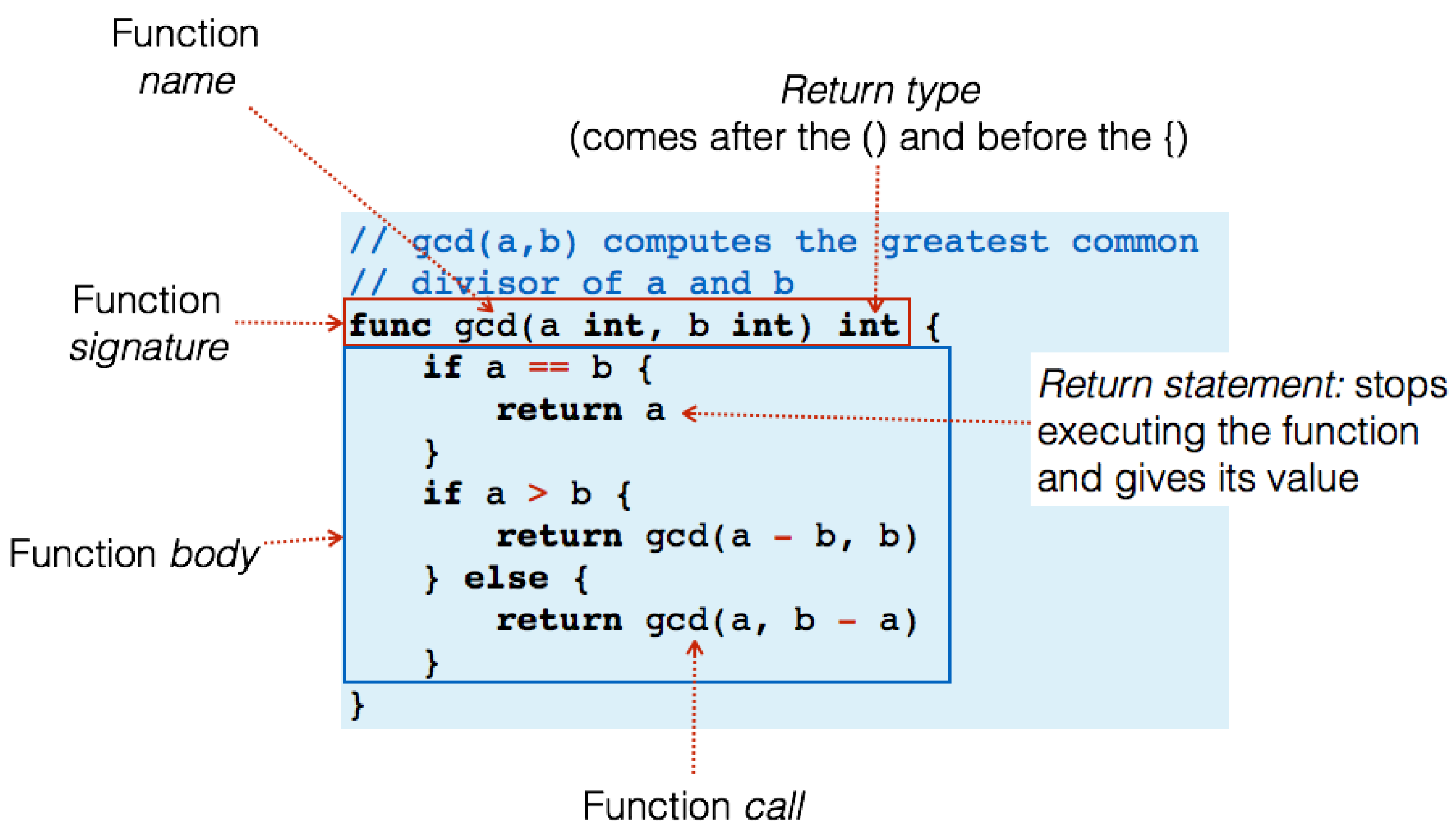
Just as with paragraphs, functions should be well written:

- They should do one thing only.
- Name + signature ≈ “topic sentence”
- They should have a good "interface" with the rest of your program.
- Functions should be short.





A longer example:


Function name
Return type (comes after the () and before the {})
Function signature
Return statement: stops executing the function and gives its value
Function body
Function call

```
// 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)
    }
}
```

More 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 after the `()` in the signature.

The `main()` function

Your program starts by running the `main` function:

```go
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)
    }
}
```

`main()` shouldn't take any parameters or return anything.

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

Example: Approximating e

The number "e" can be approximated with the following expression

$$1 + \sum_{i=1}^{k} \frac{1}{i!}$$

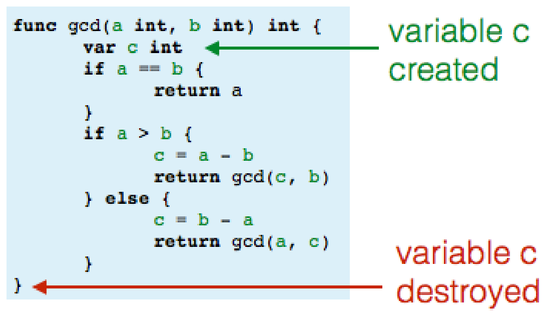
The bigger k is, the more accurate the approximation. Let's write a program to compute e.

```go
package main
import "fmt"

func factorial(n int) float64 {
    var out = 1
    for i := 1; i <= n; i++ {
        out = out * i
    }
    return float64(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))
}
```

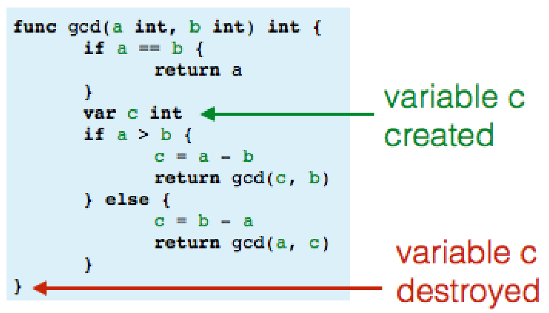
We haven't seen yet how to write loops (`for` in Go). We will see this soon.

Scope: How long do variables last?

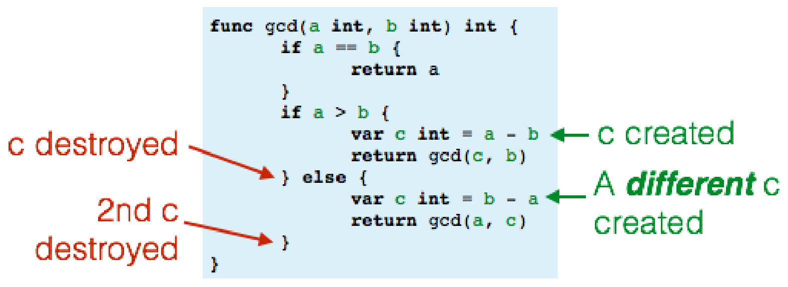
- Variables persist from when they are created until the end of the `{}` block that they were created in.
- Parameters to functions are variables. They last until the end of the function.



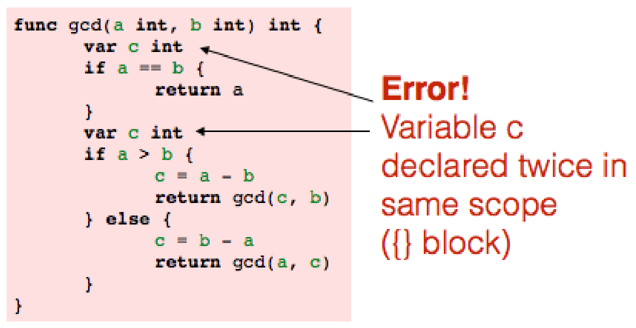
```
func gcd(a int, b int) int {
        var c int
        if a == b {
                return a
        }
        if a > b {
                c = a - b
                return gcd(c, b)
        } else {
                c = b - a
                return gcd(a, c)
        }
}
```



```
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)
        }
}
```



```
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)
        }
}
```



```
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)
        }
}
```

- The concept of scope is borrowed from mathematics.

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

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*

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

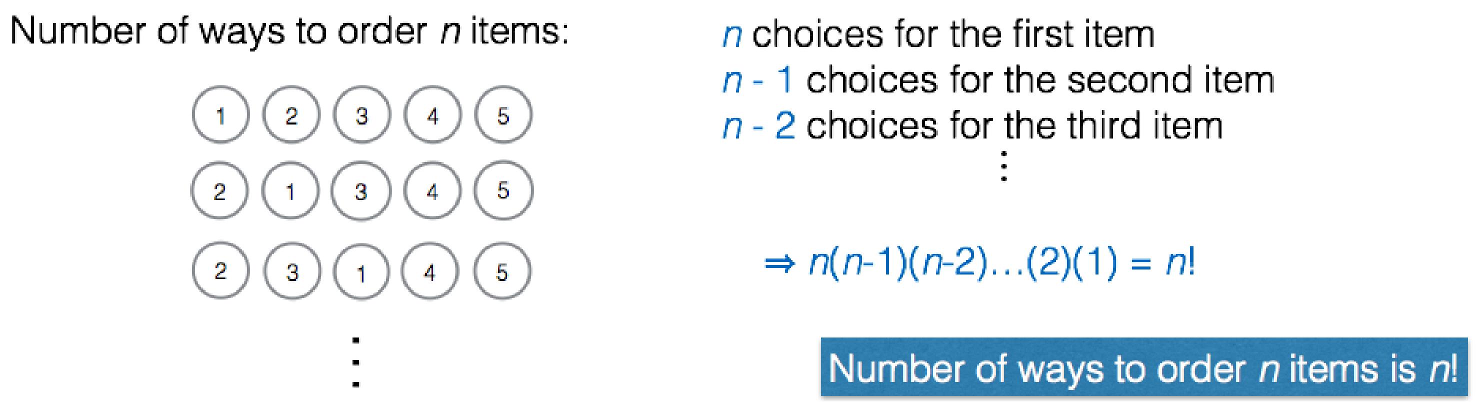
$$\sum_{i=0}^{n} \overbrace{\underbrace{\sum_{j=i}^{n} ij}_{\text{Scope of j}}}^{\text{Scope of i}}$$

> **Test yourself!** What's the error here:
>
> ```
> // BAD CODE BAD CODE BAD CODE BAD CODE BAD CODE
> 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)
>     }
> }
> // BAD CODE BAD CODE BAD CODE BAD CODE BAD CODE
> ```

Example: Permutations & Combinations

Definition of Permutations & Combinations

A permutation is an ordering of some objects:



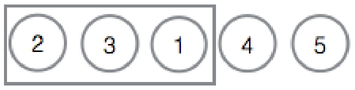
n choices for the first item
n - 1 choices for the second item
n - 2 choices for the third item
⋮

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

Number of ways to order n items is n!

A combination is the number of ways of choose k items from a set of n items:

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 $\binom{n}{k}$

Program to compute 10 choose 4

```go
package main
import "fmt"

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

func main() {
    print10Choose4()
}
```

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.

Glossary

- variable: holds a value used in your program. Can change over the course of running your program.
- assignment statement: changes the value of a variable.
- type: a variable's "type" is the kind of value it can hold.
- int: the type of a variable that can hold an integer.
- float64: the type of a variable that can hold a real number.
- identifiers: names you give to variables and functions in your program.
- function: a unit of computation.
- function call: the use of a function by another part of your program.
- scope: the time for which a particular variable exists.
- parameters (aka arguments): the things that are passed into your function; your function's input.
- return type: the type of value that a function returns.
- keywords: reserved words that have special meaning to Go.
- function signature: the name, paramters, and return type of a function.