Making functions II

Danny Arends

Overview

- Algorithms as a concept
  - Design patterns
- Functions revisited
- Basic functions
  - Recursion
  - The dot dot dot parameter
- Higher order functions
  - Lapply and apply
  - Some other examples in R

Algorithms as a concept

- An algorithm is
  - An effective method expressed as a **finite list** of **well-defined instructions** for calculating a function. Starting from an **initial state** and **initial input** (perhaps empty), the instructions describe a computation that, when executed, proceeds through a **finite number of well-defined successive states**, eventually producing **"output"** and **terminating at a final ending state**. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

http://en.wikipedia.org/wiki/Algorithm

Classifications

- By implementation
  - Recursion or iteration
    - Example: towers of hanoi
  - Logical
  - Serial, parallel or distributed
  - Deterministic or non-deterministic
  - Exact or approximate
  - Quantum algorithm
- By design paradigm
  - Brute-force or exhaustive search
  - Divide and conquer
    - Example: Sorting
  - Search and enumeration
    - Example: Chess
  - Randomized algorithm
    - Monte Carlo algorithms
  - Reduction of complexity

Design patterns

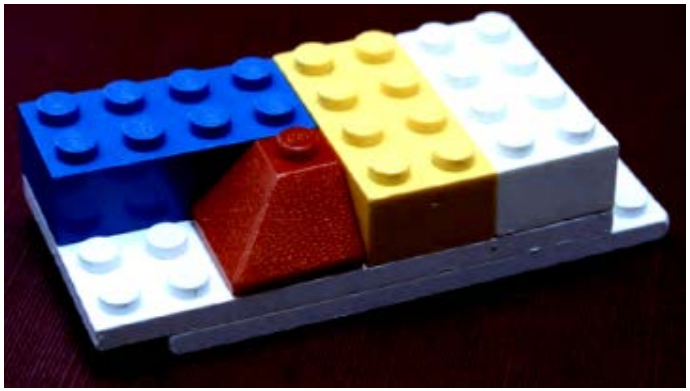
- A design pattern is a **general reusable solution** to a **commonly occurring problem** within a given context in software design.
- A design pattern is not a finished design that can be transformed directly into source or machine code. It is a **description or template** for how to solve a problem that can be used in **many different situations**.

Design patterns

- To avoid reinventing the wheel
- Best practices, proven to work
  - Some examples:
    - Singleton
    - Facade
    - Model View Controller
    - Locking
    - Scheduler

Design Patterns: Some examples

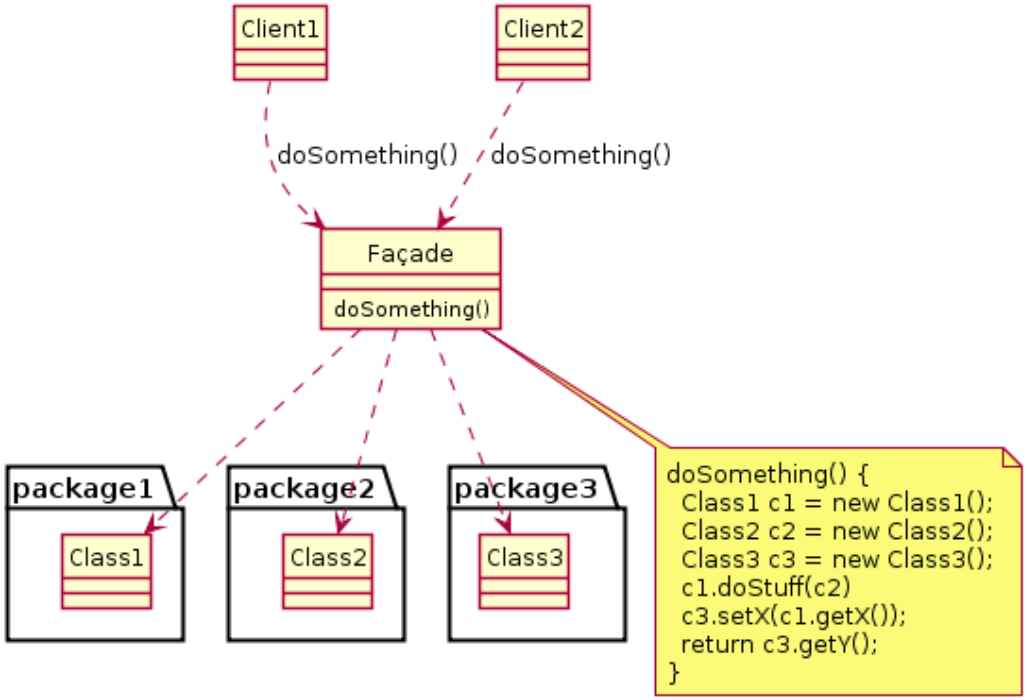
- Monte Carlo sampling device
  - LEGO and a washing machine
  - Random sampling, evaluate the outcome



http://www.improbable.com/airchives/paperair/volume20/v20i2/evolution-LEGO-washing-machine.pdf

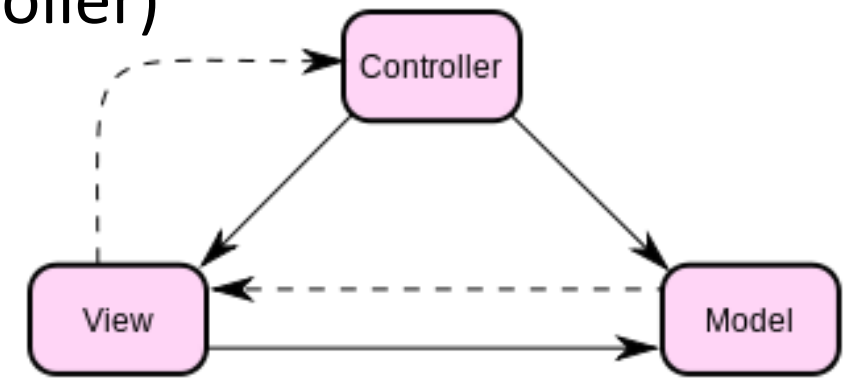
Design Patterns: Some examples

- Facade pattern
  - Provide an stable call-interface



Design Patterns: Some examples

- Model-View-Controller (MVC)
  - Data model (model)
    - Knows nothing
  - Data presentation (view)
    - Knows the Model
  - Application logic (controller)
    - Knows View and Model



Functions revisited

- Functions are factories
  - They can contain many boxes (variables)
  - But they are not visible from the outside
  - This is called the 'scope' of a variable

- Parent, variable scope
- Escaping our own scope

Functions revisited


```
someFunction <- function(inParam){          # Define a function
    intern <- (inParam)^2                   # Calculation
    return(intern)
}
somefunction(5)                             # Call the function
[1] 25
intern
Error: object 'intern' not found.
```

- **`intern`** is not visible from outside
  - This is called the ***scope*** of the variable
  - We can however access variables in our ***parent*** scope
    - This is considered **bad** practice, but sometimes we need to…

Functions revisited

- Possible reasons:
  - We are just being lazy
  - Save RAM, remember:

    Function parameters are copied in
  - Plot functions use it to read environment settings

```
# A bad example of just being lazy
exponent <- 5                          # Exponent in parent scope
someFunction <- function(inParam){     # Define a function
    intern <- (inParam)^exponent       # Calculation
    return(intern)
}
someFunction(5)
[1] 3125
```

Functions revisited

- We can also do the opposite and break out of our own ***scope***, creating/updating a variable visible outside of the function.
  - Possible reasons:
    - Save RAM, do not copy, but manipulate one object
    - Need to return multiple objects (this is being lazy)
      - Solution: use a List to return multiple objects
    - Set environmental settings, for (upcoming) plots
- Use the **<<-** assignment operator

Functions revisited

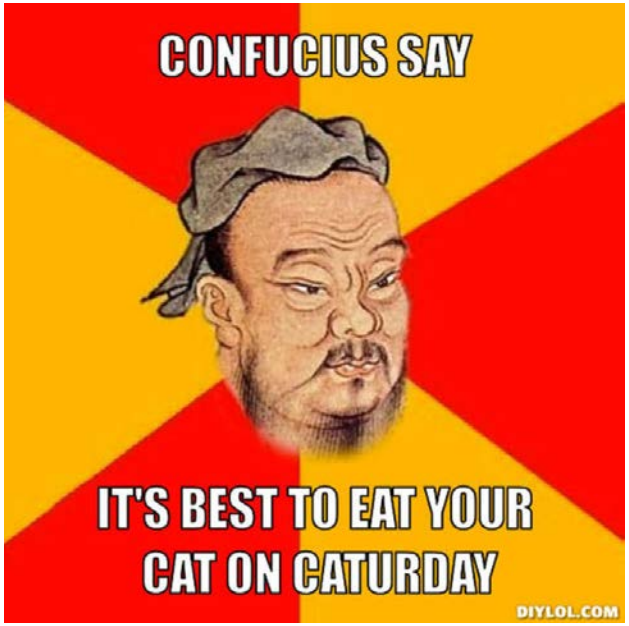

```
bigMatrix <- matrix(0, 10000, 10000)   # bigMatrix in parent scope
bigMatrix[1:3, 1:3]                    # Show a small piece
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0

someFunction <- function(){
    bigMatrix[1, 1] <<- 100            # Update bigMatrix
    invisible("Done")
}

someFunction()                         # Call our function
bigMatrix[1:3, 1:3]                    # Show a small piece
     [,1] [,2] [,3]
[1,]  100    0    0
[2,]    0    0    0
[3,]    0    0    0
```

Overview

- Taught you some ***bad*** programming practices
  - Useful for professionals
    - Even professionals often get these concepts wrong !!!
    - Powerful tools to save RAM
    - Modify the state of the R session
  - When used wrong,
    they will Eat Your Cat
  - Avoid these things if possible



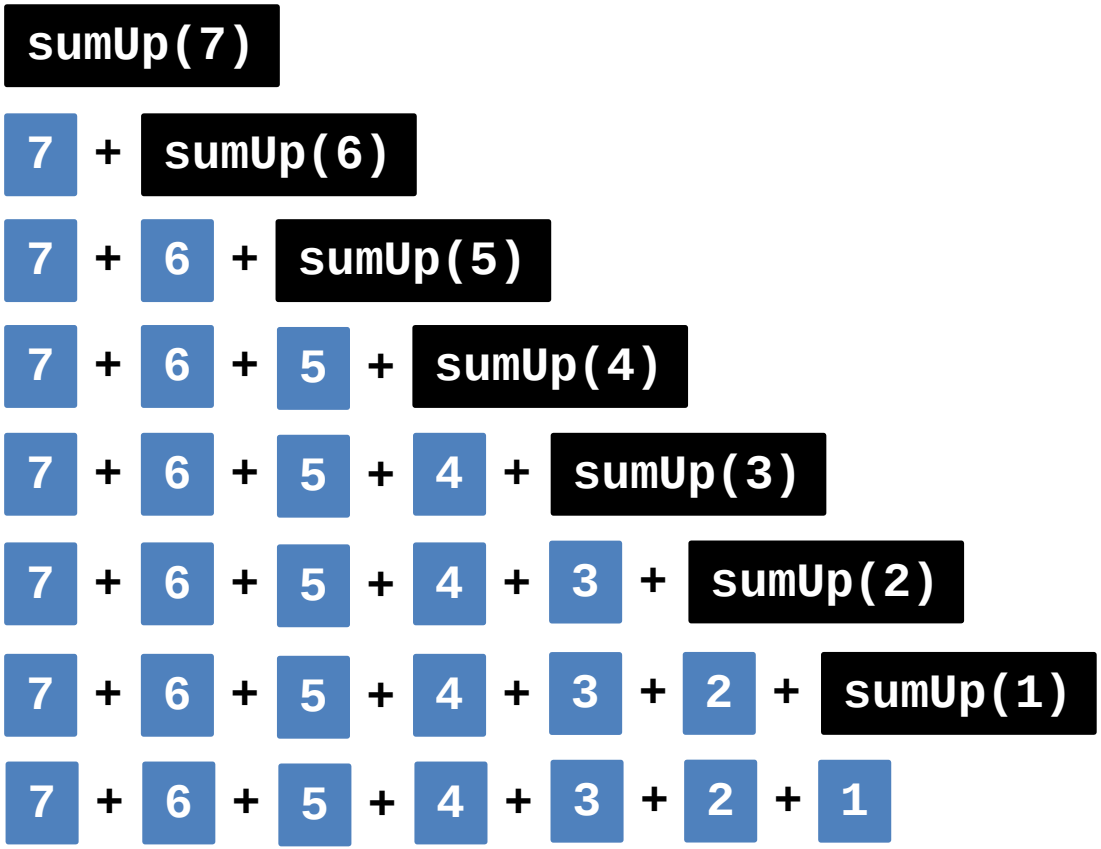
Advanced functions: Recursion

- In essence: A function calling itself
  - A simple base case (or cases)
  - Rules that reduce all other cases toward the base case

```
# The sum example from Computational Fundamentals
sumUp <- function(x){
    if(x <= 0) stop("Invalid input")  # Error on invalid input
    if(x == 1) return(1)              # Base case
    return(x + sumUp(x - 1))          # Reduction
}

sumUp(7)
[1] 28
```

Advanced functions: Recursion



Advanced functions: Recursion

- Some notes:
  - Handle weird/unexpected input
  - Infinite recursion (next slide)
    - Make sure to reduce to the base case
    - Check your invariant is increasing/decreasing properly
  - There is a recursion limit in R
    - Set the limit by: `options(expressions = 500000)`
    - Every time we call our function
      - A new function call is created
      - Internal variables need to be stored
    - 500k limit in R >= 320 Mb of memory

Advanced functions: Infinite recursion

- Recursion invariant

```
# The sum example from Computational Fundamentals
sumUp <- function(x){
                                        # No check incorrect input
    if(x == 1) return(1)                # Base case
    return(x + sumUp(x - 1))            # Reduction
}

sumUp(7)
7, 6, 5, 4, 3, 2, 1
[1] 28

sumUp(-7)
-7, -8, -9, ..., -Infinite
[ERROR] Evaluation is nested too deeply: infinite recursion?
```

Advanced functions: Indirect recusion

- Recursion using two (or more) functions

```
multUp <- function(x){
     if(x <= 0) stop("Invalid input")   # Error on 'wrong' input
     if(x == 1) return(1)               # Base case
     return(x * sumUp(x-1))             # Reduction to sumUp
}

sumUp <- function(x){
     if(x <= 0) stop("Invalid input")   # Error on 'wrong' input
     if(x == 1) return(1)               # Base case
     return(x + multUp(x-1))            # Reduction to multUp
}

sumUp(7)
[1] 157

multUp(7)
[1] 497
```

Basic functions: dot dot dot

- Pass arbitrary function parameters
  - Loose coupling between two functions
- Function A, defines 10 parameters, Function B calls function A, and passes parameters.
  - If function A adds another parameter, function B does not have to know about it
  - If function A has the names of its input parameter changed, function B does not have to know about it

Basic functions: dot dot dot


```
complexFunction <- function(x, a = 0, b = 0, c = 0){
    return(x+a+b+c)
}

usingDots <- function(x, ...){
    x = x + 6
    complexFunction(x = x, ...)
}
```

- Function usingDots now allows for 4 parameters: x, a, b and c
- **Note**: a, b, c have defaults

Overview

- Algorithms as concepts
- Revisited functions
  - Function scope & Bad programming practices
- (Indirect) Recursion
- Dot Dot Dot parameter

Higher order functions: Concepts

- Functions are factories
  - Factories can be put into a box (variable)
  - Another factory can then use this box
- Simple example, doTo

```
# Example using FUN on x
doTo <- function(x, FUN){
    return(FUN(x))
}

doTo(c(1, 2, 3, 4, 5), mean)

[1] 3
```

Higher order functions: lapply

- Higher order functions in R
  - lapply: To each element of the list do X

```
mlist <- list(x = c(1,2,3,4,5), y = c(3,4,5), z = c(5, 6, 7))
lapply(mlist, mean)            # Apply the mean function

$x
[1] 3
$y
[1] 4
$z
[1] 6
```

Higher order functions: apply

- apply works like lapply,
  - apply to a matrix, across: columns or rows

```
mmatrix <- matrix(runif(1000), 100, 10)
apply(mmatrix, 1, mean)                    # Row means
apply(mmatrix, 2, mean)                    # Column means
```

- These functions lapply and apply are much much more efficient then for/while

Higher order functions

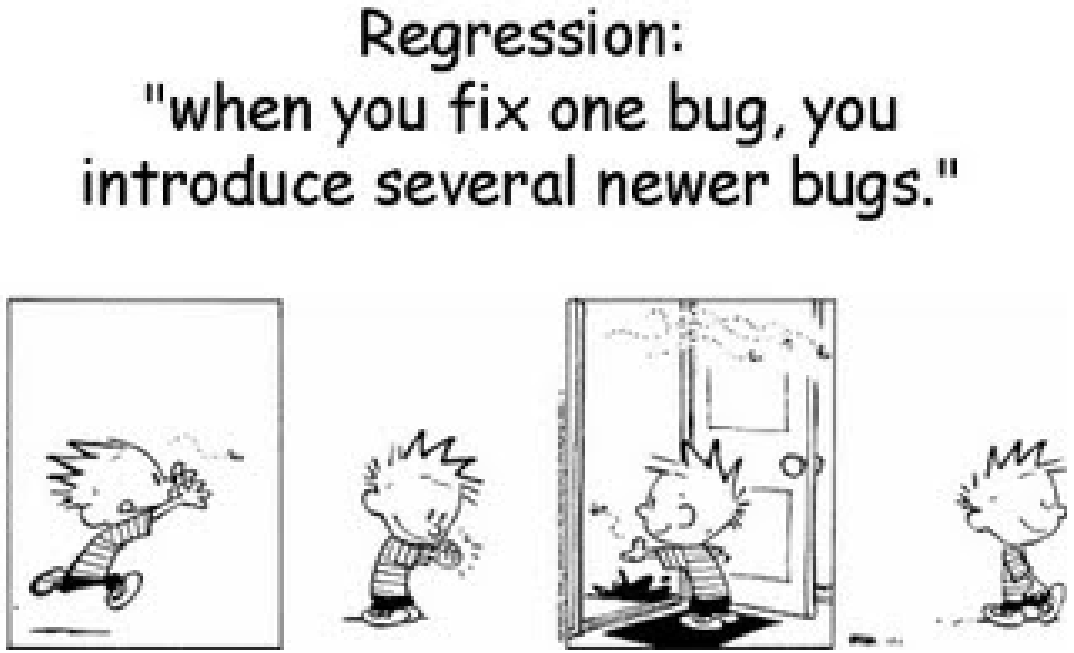
- Another example:

```
# We can check the type and then decide to use it
higherOrder <- function(x, FUN){
    if(class(FUN) != "function") stop("Please pass a function")
    if(x < 10) x = FUN(x)
    return(x)
}

class(higherOrder)                    # Check the class
[1] "function"
```

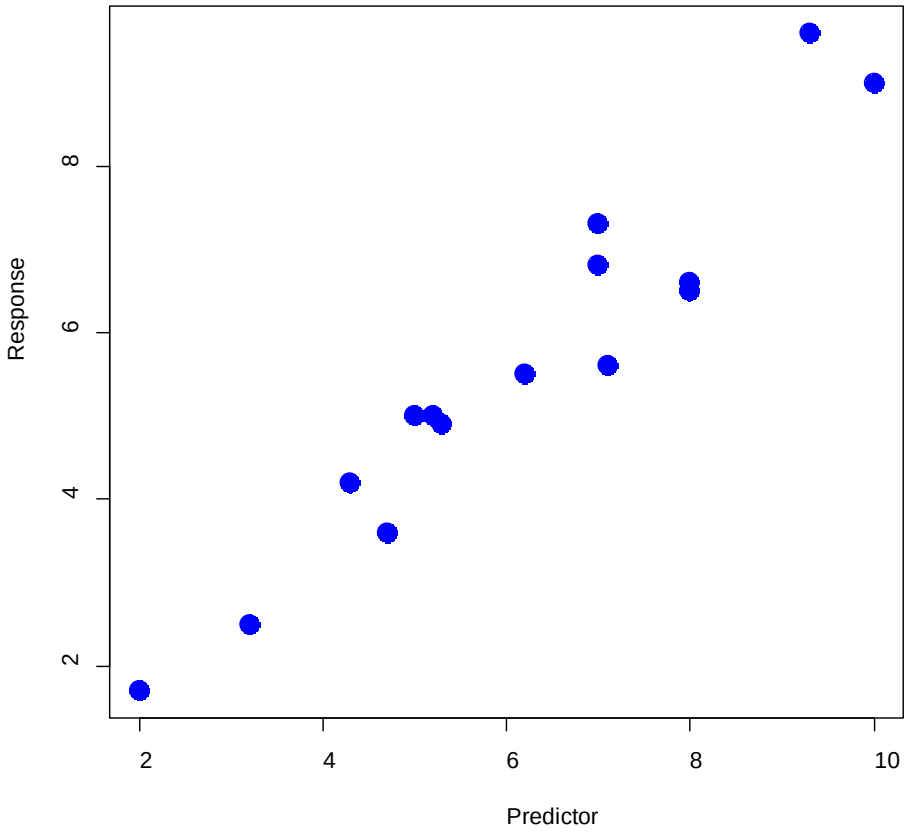
Linear modeling

- Regression
  - Best fit of a straight line through your data
  - **~** and **lm** and **glm** function
- **anova** function



Linear modeling

Reponse	Predictor
6.6	8.0
5.6	7.1
6.5	8.0
5.5	6.2
5.0	5.0
4.9	5.3
2.5	3.2
6.8	7.0
9.0	9.8
5.0	5.2
7.3	7.0
4.2	4.3
1.7	2.0
9.6	9.3
3.6	4.7



Linear modeling

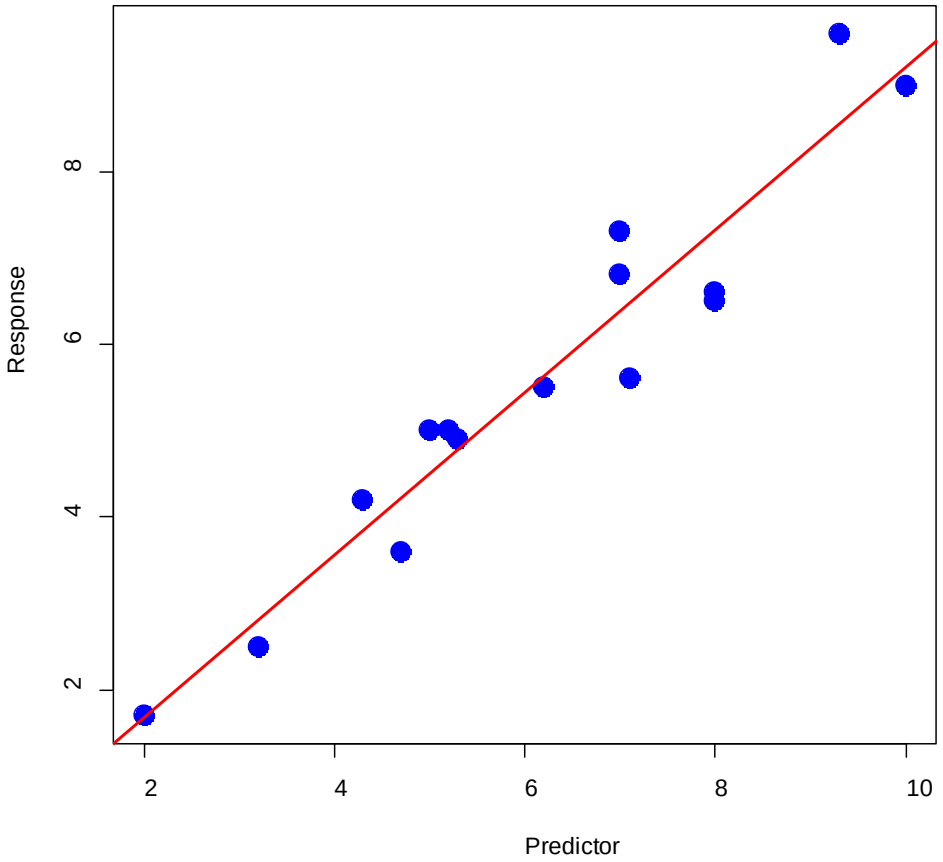
- Using the function:

```
lm(response ~ predictor)
```

```
Call:
lm(formula = response ~ predictor)

Coefficients:
(Intercept)    predictor
    -0.2068       0.9415
```

```
abline(a = -0.2068, b = 0.9415)
```



Linear modeling

- Significance
  - Use the **anova** function to test for significance of the predictor variable(s)

```
anova(lm(response ~ predictor))

Analysis of Variance Table

Response: response
          Df Sum Sq Mean Sq F value    Pr(>F)
predictor  1 60.578  60.578  153.82 1.405e-08 ***
Residuals 13  5.120   0.394
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Summary

- Higher order functions
  - Factories as input to other factories
  - How to write your own
  - lapply, apply
- Linear models
  - R has many functions to deal with linear models
  - We can use slope & intercept to beautify plots
  - Determine if something is significant

Questions

Many thanks to my London Support Team:
Joeri van der Velde, Anna Mulder, Diana Karweina & Konrad Zych