

# Programming for Data Analytics

## 2. Lists and Functions

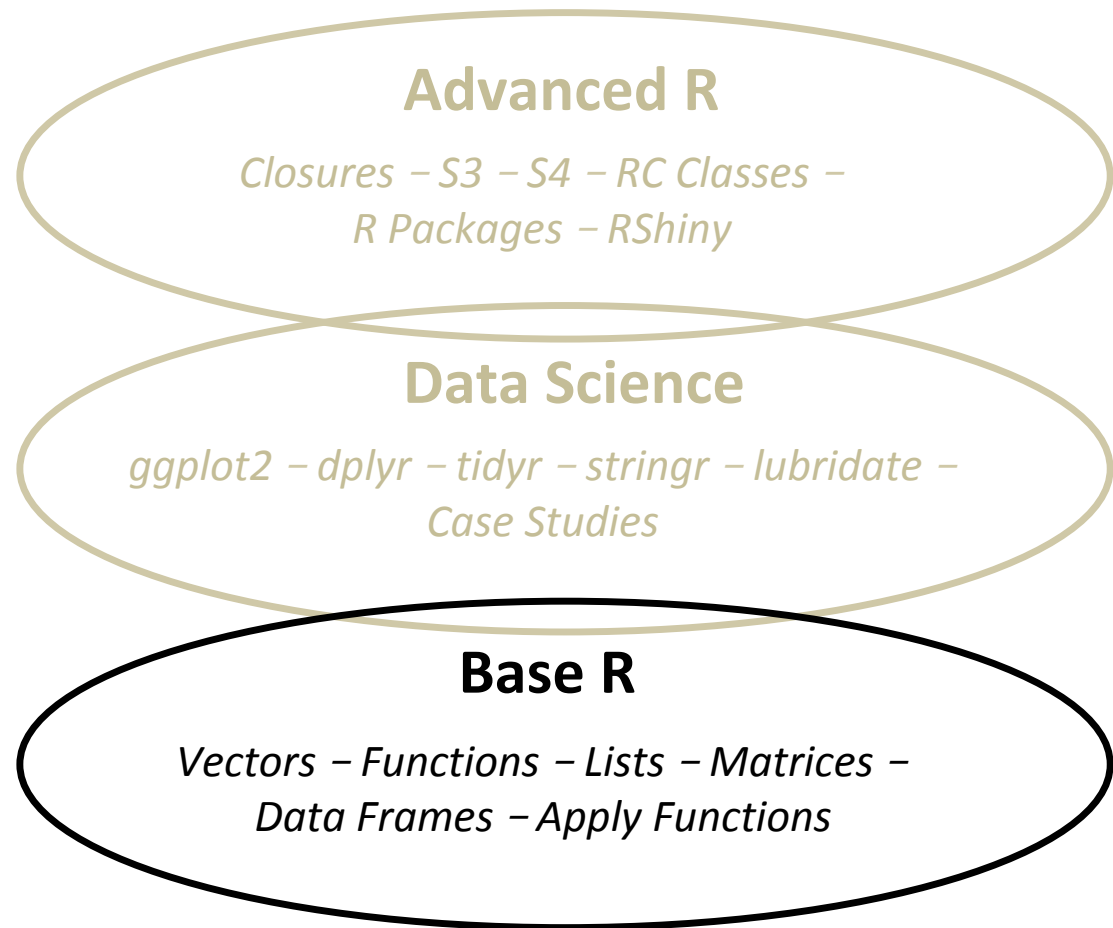
Dr. Jim Duggan,  
School of Engineering & Informatics  
National University of Ireland Galway.

[https://twitter.com/\\_jimduggan](https://twitter.com/_jimduggan)



# Lecture 02 - Overview

1. Lists
2. Functions
3. Environments



# (1) Lists

	Homogenous	Heterogenous
1d	Atomic Vector	List
2d	Matrix	Data Frame
nd	Array	

- Lists are different from atomic vectors because their elements can be of **any type**, including lists.
- `list()` creates a list, instead of `c()`

# Creating a list...

```
>  
> x<- list(1:3, "a", c(T,F,T), c(2.3, 5.9))  
>  
> str(x)  
List of 4  
 $ : int [1:3] 1 2 3  
 $ : chr "a"  
 $ : logi [1:3] TRUE FALSE TRUE  
 $ : num [1:2] 2.3 5.9
```

# Subsetting lists

- Works in the same way as subsetting an atomic vector
- Using `[` will always return a list
- `[[` and `$` pull out the contents of a list

If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5, `x[4:6]` is a train of cars 4-6" @RLangTip



# Example

```
> x<- list(1:3, let="a", c(T,F,T))
```

```
>
```

```
> x
```

```
[[1]]
```

```
[1] 1 2 3
```

```
$let
```

```
[1] "a"
```

```
[[3]]
```

```
[1] TRUE FALSE TRUE
```

```
> x[1]
```

```
[[1]]
```

```
[1] 1 2 3
```

```
> str(x[1])
```

```
List of 1
```

```
$ : int [1:3] 1 2 3
```

```
>
```

```
> x[[1]]
```

```
[1] 1 2 3
```

```
>
```

```
> str(x[[1]])
```

```
int [1:3] 1 2 3
```

# Simplifying vs Preserving Subsetting

- Simplifying subsets returns the simplest possible data structure that can represent the output
- Preserving subsetting keeps the structure of the input the same as the output.
- Omitting drop = FALSE when subsetting matrices and data frames is a common source of programming error.

	Simplifying	Preserving
Vector	x[[1]]	x[1]
List	x[[1]]	x[1]

# \$ operator

- \$ is a shorthand operator, where x\$y is equivalent to **x[["y",exact=FALSE]]**
- Often used to access variables in a data frame
- \$ does partial matching

```
> x  
[[1]]  
[1] 1 2 3
```

```
$let  
[1] "a"
```

```
[[3]]  
[1] TRUE FALSE TRUE
```

```
> x$let  
[1] "a"  
>  
> x$l  
[1] "a"
```

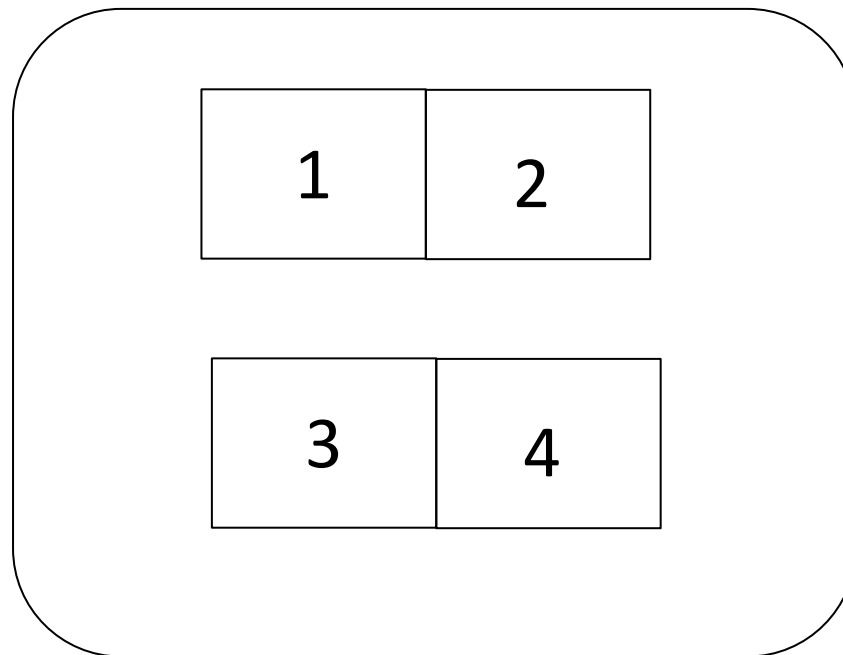


# Visualising Lists and Subsetting

- Lists have rounded corners. Atomic vectors have square corners
- Children are drawn inside their parent, and have a slightly darker background

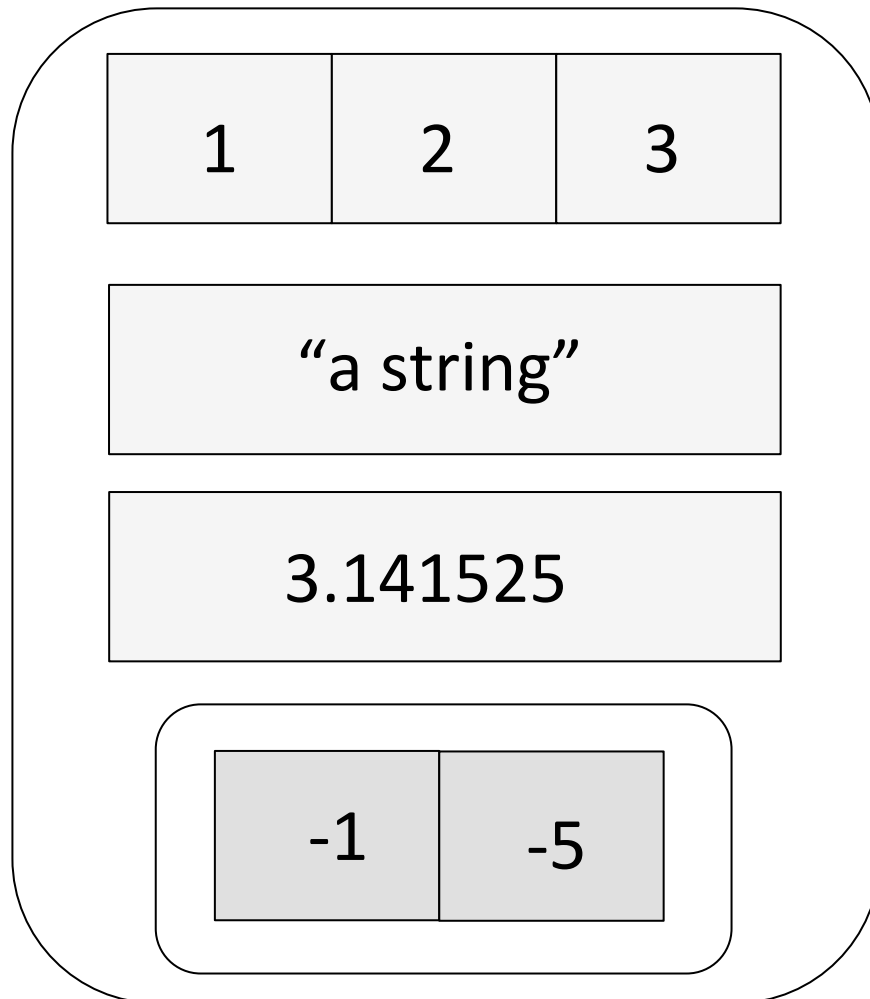
```
x1 <- list(c(1,2),c(3,4))
```

x1

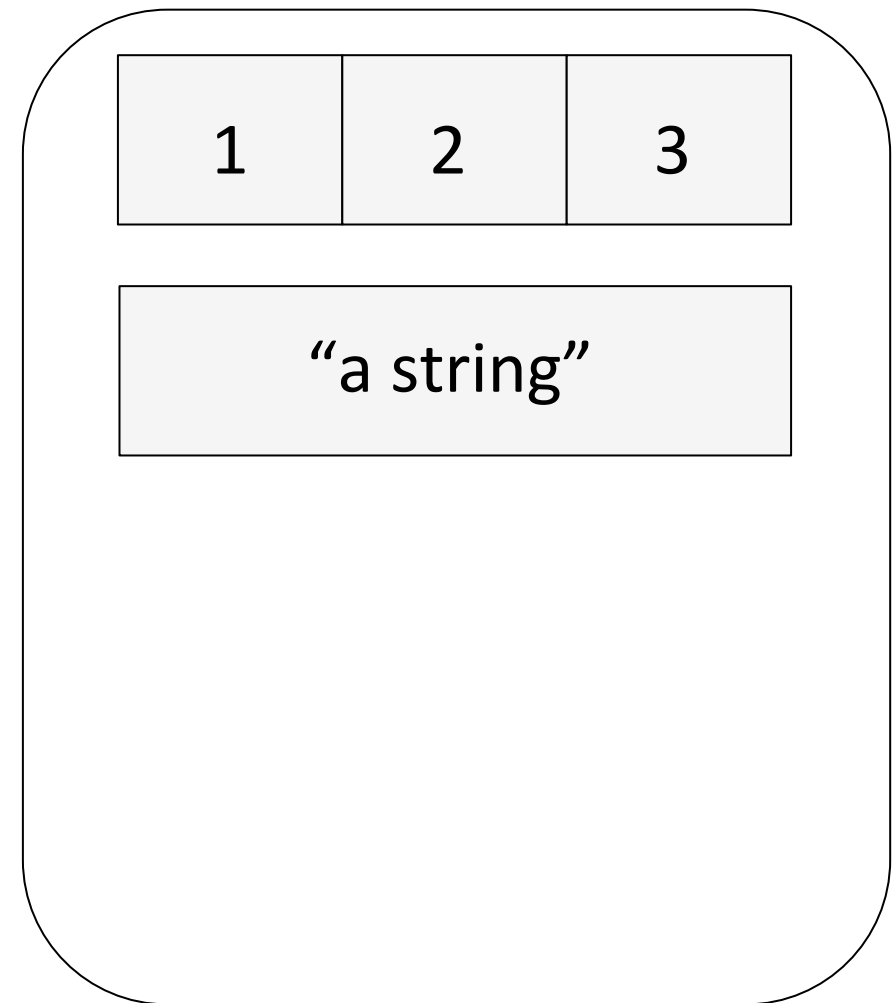


```
a <- list(a=1:3, b = "a string", c=pi, d = list(-1,-5))
```

a

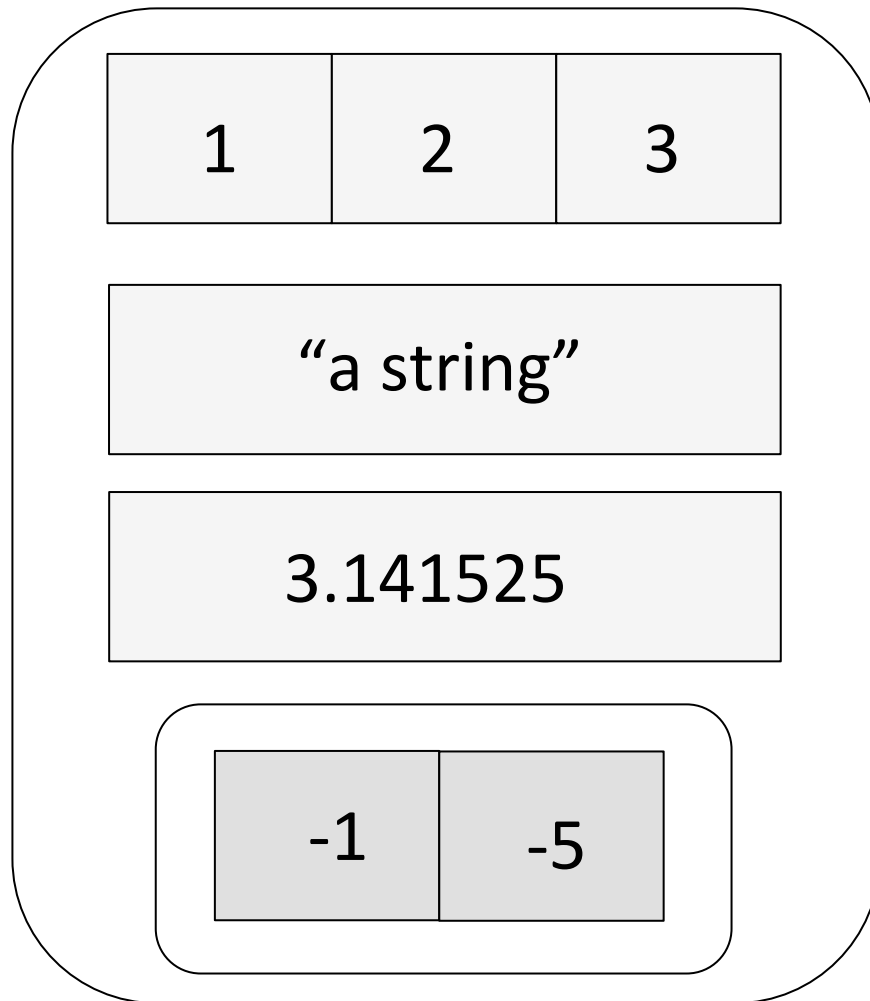


a[1:2]

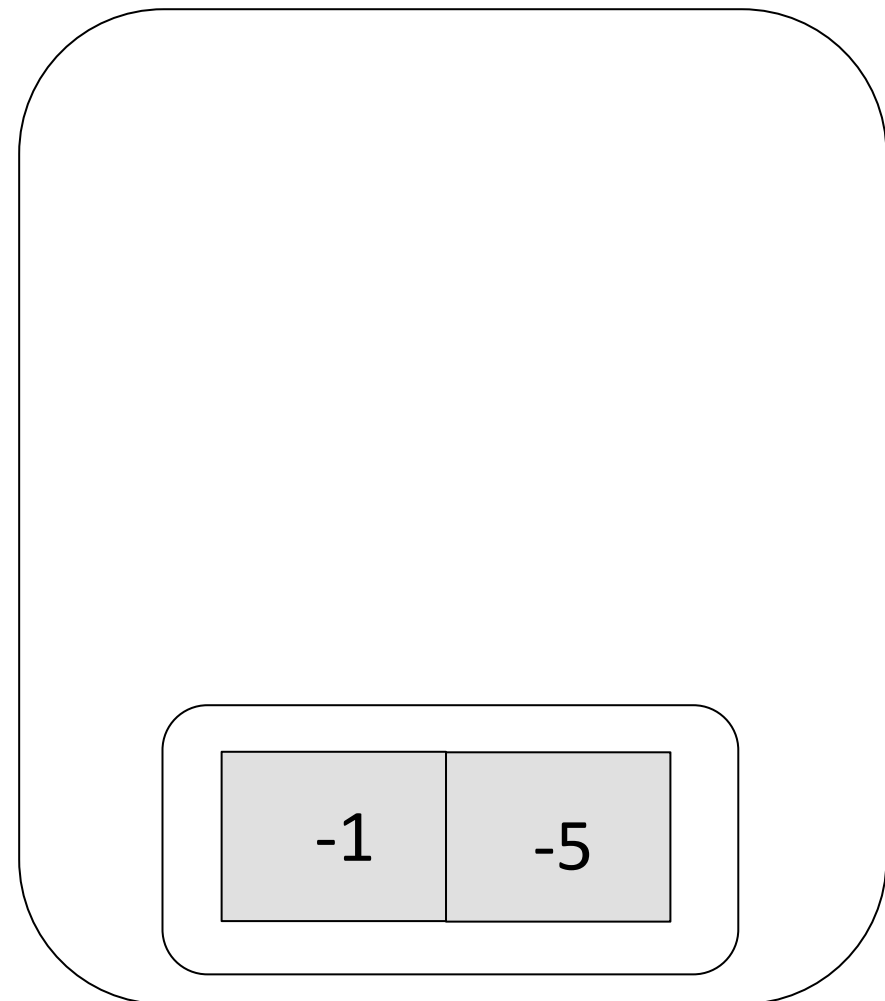


```
a <- list(a=1:3, b = "a string", c=pi, d = list(-1,-5))
```

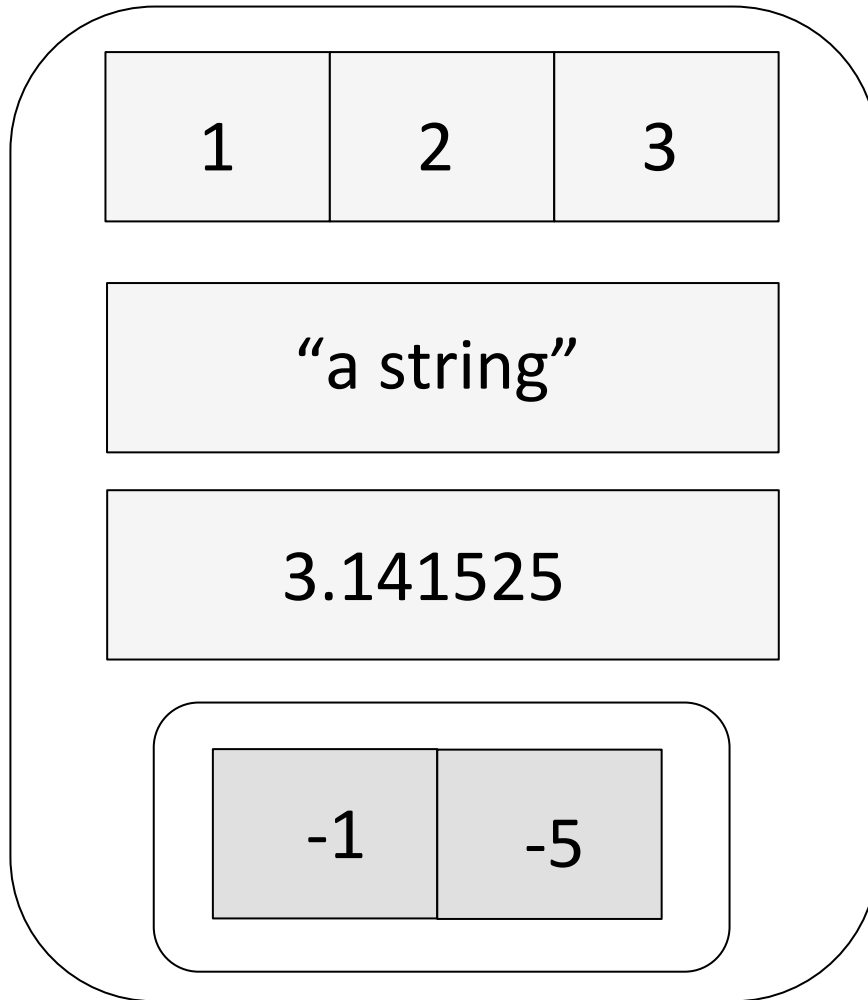
a



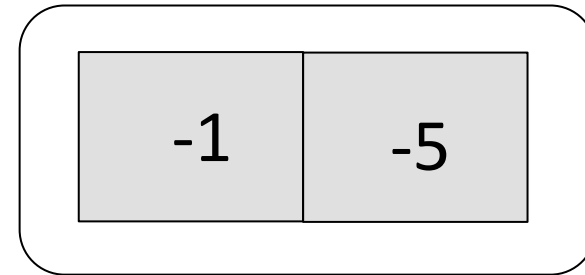
a[4]



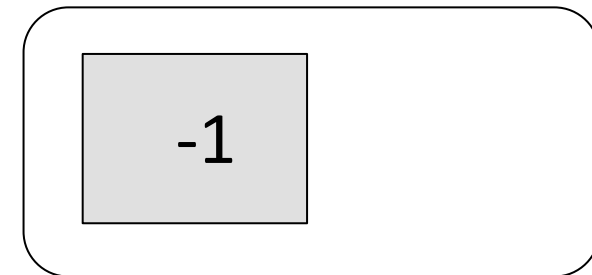
a



a[[4]]



a[[4]][1]



a[[4]][[1]]



a[[4]][[2]]



## Challenge 2.1

```
l <- list(1:4, list(2:4, c(T, F)), c("A", "B"))
```

- Based on the list:
  - Extract the 2<sup>nd</sup> list element, and calculate the sum of all the resulting elements. What do you expect the sum to be?
  - Calculate the sum of the first list element
  - Convert the entire list to a vector and predict what the vector type will be

## (2) Functions

- A function is a group of instructions that:
  - takes input,
  - uses the input to compute other value, and
  - returns a result (Matloff 2009).
- Functions are a fundamental building block of R (Wickham 2015)
- Users of R should adopt the habit of creating simple functions which will make their work more effective and also more trustworthy (Chambers 2008).
- Functions are declared:
  - using the **function** reserved word
  - are objects



# General Form

`function(arguments)`  
*expression*



- *arguments* gives the arguments, separated by commas.
- *Expression* (body of the function) is any legal R expression, usually enclosed in { }
- **Last evaluation is returned**
- `return()` can also be used, but usually for exceptions.



# Example

```
> f <- function(x){x^2}
```

```
>
```

```
> f(4)
```

```
[1] 16
```

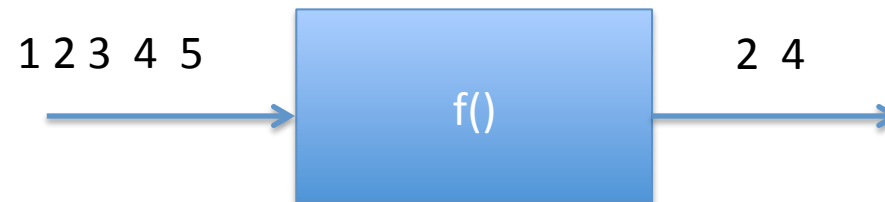


```
> f(1:10)
```

```
[1] 1 4 9 16 25 36 49 64 81 100
```

## Challenge 2.2

- Write an R function that filters a vector to return all the even numbers



# Function Arguments

- It is useful to distinguish between *formal arguments* and the *actual arguments*
  - **Formal arguments** are the property of the function
  - **Actual arguments** can vary each time the function is called.
- When calling functions, arguments can be specified by
  - Complete name
  - Partial name
  - Position



```
f <- function(abcdef, bcde1, bcde2){
  c(a=abcdef, b1=bcde1, b2=bcde2)
}
```

```
f(1,2,3)
```

```
f(2,3,abcdef = 1)
```

```
f(2,3,a = 1)
```

```
f(2,3,b = 1)
```

```
> f(1,2,3)
```

```
  a b1 b2
  1  2  3
```

```
>
```

```
> f(2,3,abcdef = 1)
```

```
  a b1 b2
  1  2  3
```

```
>
```

```
> f(2,3,a = 1)
```

```
  a b1 b2
  1  2  3
```

```
>
```

```
> f(2,3,b = 1)
```

```
Error in f(2, 3, b = 1) : c
```

# Guidelines (Wickham 2015)

- Use positional mapping for the first one or two arguments (most commonly used)
- Avoid using positional mapping for less commonly used attributes
- Named arguments should always come after unnamed arguments



# Default and missing arguments

- Function arguments in R can have default values
- R function arguments are “lazy” – only evaluated if actually used

```
g <- function(a=1, b=1){  
  c(a,b)  
}
```

```
>  
> g()  
[1] 1 1  
> g(1)  
[1] 1 1  
> g(2,4)  
[1] 2 4
```

# Status of an argument

- You can determine if an argument was supplied by using the `missing()` function

```
h <- function(a=1, b=1){  
  c(missing(a),missing(b))  
}
```

```
>  
> h()  
[1] TRUE TRUE  
>  
> h(1)  
[1] FALSE TRUE  
>  
> h(1,1)  
[1] FALSE FALSE  
,
```

# Creating robust functions

- Defensive programming “the art of making code fail in a well-defined manner even when something unexpected occurs” (Wickham 2015)
- Key principle:
  - Fail fast
  - As soon as something is wrong, signal an error
- Generally good style to reserve the use of an explicit **return()** when returning early from a function



# Examples

```
my_min <- function(v){  
  if(!is.numeric(v))  
    stop("Error, type should be numeric")  
  min(v)  
}
```

```
> my_min("ABC")
```

```
Error in my_min("ABC") : Error, type should be numeric
```

```
>
```

```
> my_min(c(T,F))
```

```
Error in my_min(c(T, F)) : Error, type should be numeric
```

```
>
```

```
> my_min(c(T,F,1))
```

```
[1] 0
```

```
>
```

```
> my_min(300:400)
```

```
[1] 300
```

# Passing functions as objects

- Functions are first class objects, so they can be passed to other functions
- Provides flexibility, and widely used in R

```
> p1 <- function(f, v){  
+   f(v)  
+ }  
>  
> p1 (min,1:10)  
[1] 1  
>  
> p1(max,1:10)  
[1] 10  
>  
> p1(mean,1:10)  
[1] 5.5
```

# Challenge 2.3

Write a function that takes in a vector and returns a vector with no duplicates. Make use of the R function `uplicated()`

`uplicated {base}`

R Documentation

## Determine Duplicate Elements

### Description

`uplicated()` determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.

`anyDuplicated(.)` is a “generalized” more efficient shortcut for `any(uplicated(.))`.

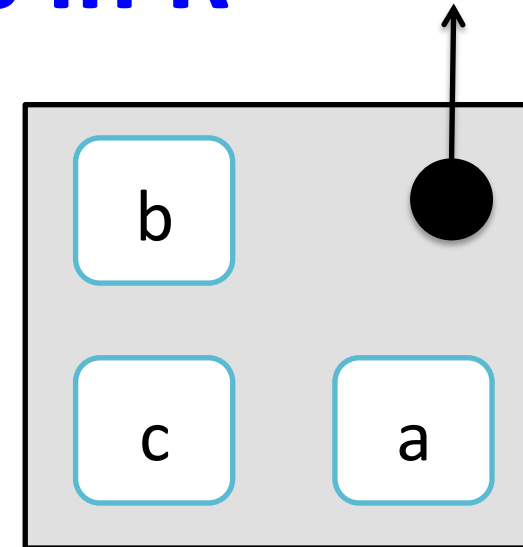
### Usage

```
uplicated(x, incomparables = FALSE, ...)
```



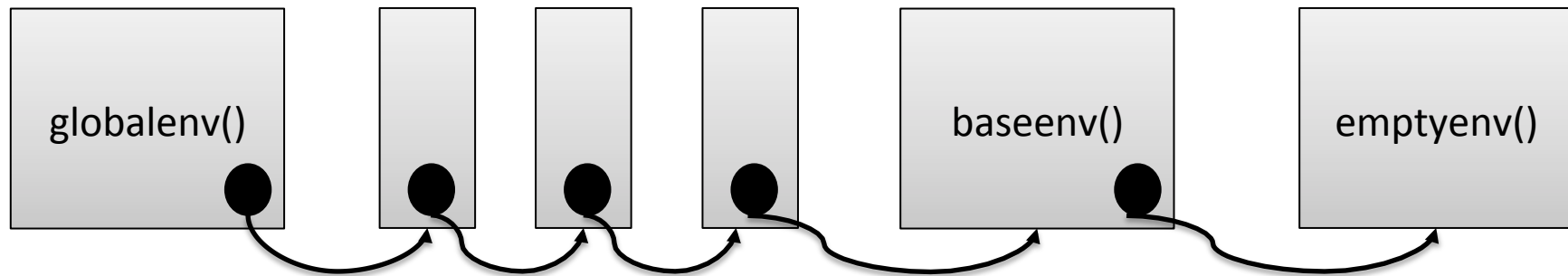
## (3) Environments in R

- Environments can be thought of as consisting of two things: a frame, which is a set of symbol-value pairs, and an enclosure, a pointer to an enclosing environment
- Every object (variable or function) in an environment has a unique name
- The working environment is known as the Global Environment



```
> a <- 10
> b <- 20
> c <- 30
>
> environment()
<environment: R_GlobalEnv>
```

# Hierarchy of Environments



- Environments form a tree structure in which the enclosures play the role of parents. The tree of environments is rooted in an empty environment, available through `emptyenv()`, which has no parent

```
> ls()  
[1] "a" "b" "c"
```

# Examining the hierarchy with search()

```
> search()
```

```
[1] ".GlobalEnv"      "tools:rstudio"    "package:stats"
[4] "package:graphics" "package:grDevices" "package:utils"
[7] "package:datasets" "package:methods"   "Autoloads"
[10] "package:base"
```

```
>
```

```
> library(ggplot2)
```

```
>
```

```
> search()
```

```
[1] ".GlobalEnv"      "package:ggplot2"  "tools:rstudio"
[4] "package:stats"    "package:graphics" "package:grDevices"
[7] "package:utils"    "package:datasets" "package:methods"
[10] "Autoloads"        "package:base"
```



# Functions and their Environment

- Functions are first class objects that exist in an environment
- **Functions can access all variables contained in their enclosing environment**
- When functions are executed, they have their own environment (to be covered later in the advanced topic)

```
> ls()
character(0)
>
> f1 <- function(x)x^2
>
> ls()
[1] "f1"
>
> x <- f1(2)
>
> ls()
[1] "f1" "x"
```

# Looking for variables...

- If a name isn't defined inside a function, R will look one level up to the enclosing environment.

```
x <- 2
g <- function(){
  y <- 1
  c(x,y)
}

g()
```

```
>
> g()
[1] 2 1
```



# Similar search rules apply for nested functions

```
x <- 1
h <- function(){
  y <- 2
  i <- function(){
    z <- 3
    c(x,y,z)
  }
  i()
}

h()
```

```
> h()
[1] 1 2 3
```

## Challenge 2.4

- Predict the output of the following call h()

```
x <- 1; y <- 2; z <- 3
h <- function(){
  x <- 10
  z <- 30
  i <- function(){
    x <- 100
    z <- 300
    j <- function(){
      x <- 1
      c(x,y,z)
    }# end of j
    j()
  } # end of i
  i()
}# end of h
```

# References

- Wickham, H. 2015.  
Advanced R. Taylor &  
Francis

