

# FRE6871 R in Finance

## Lecture#6, Spring 2023

Jerzy Pawlowski [jp3900@nyu.edu](mailto:jp3900@nyu.edu)

*NYU Tandon School of Engineering*

May 1, 2023



**NYU**

**TANDON SCHOOL  
OF ENGINEERING**

# Lists

Lists are a type of vector that contain elements of different *types*.

Lists are recursive object types, meaning each list element can contain other vectors or lists.

The function `list()` creates a list from a list of vectors.

`list()` creates a named list from a list of symbol-value pairs.

The function `is.list()` returns `TRUE` if its argument is a list, and `FALSE` otherwise.

The function `unlist()` collapses a list with atomic elements into a vector (which can cause type coercion).

```
> # Create a list with two elements
> listv <- list(c("a", "b"), 1:4)
> listv
[[1]]
[1] "a" "b"

[[2]]
[1] 1 2 3 4
> c(typeof(listv), mode(listv), class(listv))
[1] "list" "list" "list"
> # Lists are also vectors
> c(is.vector(listv), is.list(listv))
[1] TRUE TRUE
> NROW(listv)
[1] 2
> # Create named list
> listv <- list(first=c("a", "b"), second=1:4)
> listv
$first
[1] "a" "b"

$second
[1] 1 2 3 4
> names(listv)
[1] "first" "second"
> unlist(listv)
first1 first2 second1 second2 second3 second4
  "a"   "b"   "1"   "2"   "3"   "4"
```

# Subsetting Lists

Lists can be subset (indexed) using:

- the "[" operator (returns sublist),
- the "[[" operator (returns an element),
- the "\$" operator (for named listv only),

Partial name matching allows subsetting with partial name, as long as it can be resolved.

```
> listv[2] # Extract second element as sublist
$second
[1] 1 2 3 4
> listv[[2]] # Extract second element
[1] 1 2 3 4
> listv[[2]][3] # Extract third element of second element
[1] 3
> listv[[c(2, 3)]] # Third element of second element
[1] 3
> listv$second # Extract second element
[1] 1 2 3 4
> listv$s # Extract second element - partial name matching
[1] 1 2 3 4
> listv$second[3] # Third element of second element
[1] 3
> listv <- list() # Empty list
> listv$a <- 1
> listv[2] <- 2
> listv
$a
[1] 1

[[2]]
[1] 2
> names(listv)
[1] "a" ""
```

# Coercing Vectors Into Lists Using `as.list()`

The function `as.list()` coerces vectors and other objects into lists.

`as.list()` returns a list with the same elements as the vector.

`list()` called on a vector returns a single element equal to the vector.

```
> # Convert vector elements to list elements
> as.list(1:3)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
> # Convert whole vector to single list element
> list(1:3)
[[1]]
[1] 1 2 3
```

# Data Frames

Data frames are 2-D objects (like matrices), but their columns can be of different *types*.

Data frames can be thought of as *listv* of vectors of the same length.

The function `data.frame()` creates a *data frame* from vectors assigned to column names.

```
> dframe <- data.frame( # Create a data frame
+   type=c("rose", "daisy", "tulip"),
+   color=c("red", "white", "yellow"),
+   price=c(1.5, 0.5, 1.0)
+ ) # end data.frame

> dframe
  type color price
1 rose   red   1.5
2 daisy white  0.5
3 tulip yellow 1.0

> dim(dframe) # Get dimension attribute
[1] 3 3

> colnames(dframe) # Get the colnames attribute
[1] "type" "color" "price"

> rownames(dframe) # Get the rownames attribute
[1] "1" "2" "3"

> class(dframe) # Get object class
[1] "data.frame"

> typeof(dframe) # Data frames are listv
[1] "list"

> is.data.frame(dframe)
[1] TRUE

>
> class(dframe$type) # Get column class
[1] "character"

> class(dframe$price) # Get column class
[1] "numeric"
```

# Subsetting Data Frames

Data frames can be subset in a similar way to `listv` and matrices.

Depending on how a data frame is subset, the result can be either a data frame or a vector.

Extracting a single column from a data frame produces a vector.

The data frame class attribute can be preserved by using the parameter `"drop=FALSE"`.

Extracting a single row from a data frame produces a data frame.

The function `unlist()` applied to a single row extracted from a data frame coerces it to a vector.

```
> dframe[, 3] # Extract third column as vector
[1] 1.5 0.5 1.0
> dframe[[3]] # Extract third column as vector
[1] 1.5 0.5 1.0
> dframe[3] # Extract third column as data frame
  price
1  1.5
2  0.5
3  1.0
> dframe[, 3, drop=FALSE] # Extract third column as data frame
  price
1  1.5
2  0.5
3  1.0
> dframe[[3]][2] # Second element from third column
[1] 0.5
> dframe$price[2] # Second element from "price" column
[1] 0.5
> is.data.frame(dframe[[3]]); is.vector(dframe[[3]])
[1] FALSE
[1] TRUE
> dframe[2, ] # Extract second row
  type color price
2 daisy white  0.5
> dframe[2, ][3] # Third element from second column
  price
2  0.5
> dframe[2, 3] # Third element from second column
[1] 0.5
> unlist(dframe[2, ]) # Coerce to vector
  type  color  price
"daisy" "white" "0.5"
> is.data.frame(dframe[2, ]); is.vector(dframe[2, ])
[1] TRUE
[1] FALSE
```

# Data Frames and Factors

By default `data.frame()` does not coerce character vectors to factors, so no need for the option `stringsAsFactors=FALSE`.

The function `options()` sets global *options*, that determine how R computes and displays its results.

If the global option `stringsAsFactors=FALSE` is set, then character vectors will not be coerced to factors in all subsequent data frame operations.

The default is `stringsAsFactors=FALSE` since R version 4.0.

```
> dframe <- data.frame( # Create a data frame
+   type=c("rose", "daisy", "tulip"),
+   color=c("red", "white", "yellow"),
+   price=c(1.5, 0.5, 1.0),
+   row.names=c("flower1", "flower2", "flower3")
+ ) # end data.frame
> dframe
      type color price
flower1 rose   red   1.5
flower2 daisy  white  0.5
flower3 tulip  yellow 1.0
> class(dframe$type) # Get column class
[1] "character"
> class(dframe$price) # Get column class
[1] "numeric"
> # Set option to not coerce character vectors to factors
> options("stringsAsFactors")
$stringsAsFactors
NULL
> default.stringsAsFactors()

Error: 'default.stringsAsFactors' is defunct.
Use ''stringsAsFactors = FALSE'' instead.
See help("Defunct")

> options(stringsAsFactors=FALSE)
```

# Exploring Data Frames

The function `str()` displays the structure of an R object.

The functions `head()` and `tail()` display the first and last rows of an R object.

```
> str(dframe) # Display the object structure
'data.frame': 3 obs. of 3 variables:
 $ type : chr  "rose" "daisy" "tulip"
 $ color: chr  "red" "white" "yellow"
 $ price: num  1.5 0.5 1
> dim(cars) # The cars data frame has 50 rows
[1] 50 2
> head(cars, n=5) # Get first five rows
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
> tail(cars, n=5) # Get last five rows
  speed dist
46    24   70
47    24   92
48    24   93
49    24  120
50    25   85
```



# Sorting Vectors

The function `sort()` returns a vector sorted into ascending order.

A permutation is a re-ordering of the elements of a vector.

The permutation index specifies how the elements are re-ordered in a permutation.

The function `order()` calculates the permutation index to sort a given vector into ascending order.

Applying the function `order()` twice: `order(order())`, calculates the permutation index to sort the vector from ascending order into its unsorted (original) order.

So the permutation index produced by: `order(order())` is the reverse of the permutation index produced by: `order()`.

`order()` can take several vectors as input, to break any ties.

Data frames can be sorted on any column.

```
> # Create a named vector of student scores
> scorev <- sample(round(runif(5, min=1, max=10), digits=2))
> names(scorev) <- c("Angie", "Chris", "Suzie", "Matt", "Liz")
> # Sort the vector into ascending order
> sort(scorev)
  Liz  Matt Suzie Chris Angie
3.63  6.63  7.09  8.35  9.02
> # Calculate index to sort into ascending order
> order(scorev)
[1] 5 4 3 2 1
> # Sort the vector into ascending order
> scorev[order(scorev)]
  Liz  Matt Suzie Chris Angie
3.63  6.63  7.09  8.35  9.02
> # Calculate the sorted (ordered) vector
> sortv <- scorev[order(scorev)]
> # Calculate index to sort into unsorted (original) order
> order(order(scorev))
[1] 5 4 3 2 1
> sortv[order(order(scorev))]
Angie Chris Suzie  Matt  Liz
9.02  8.35  7.09  6.63  3.63
> scorev
Angie Chris Suzie  Matt  Liz
9.02  8.35  7.09  6.63  3.63
> # Examples for sort() with ties
> order(c(2, 1:4)) # There's a tie
[1] 2 1 3 4 5
> order(c(2, 1:4), 1:5) # There's a tie
[1] 2 1 3 4 5
```

# Sorting Data Frames

Data frames can be sorted on any one of its columns.

```
> # Create a vector of student ranks
> rankv <- c("fifth", "fourth", "third", "second", "first")
> # Reverse sort the student ranks according to students
> rankv[order(order(scorev))]
[1] "first" "second" "third" "fourth" "fifth"
> # Create a data frame of students and their ranks
> rosterdf <- data.frame(score=scorev,
+   rank=rankv[order(order(scorev))])
> rosterdf
      score  rank
Angie  9.02  first
Chris  8.35  second
Suzie  7.09  third
Matt   6.63  fourth
Liz    3.63  fifth

> # Permutation index on price column
> order(dframe$price)
[1] 2 3 1
> # Sort dframe on price column
> dframe[order(dframe$price), ]
      type color price
flower2 daisy  white   0.5
flower3 tulip  yellow  1.0
flower1 rose   red    1.5
> # Sort dframe on color column
> dframe[order(dframe$color), ]
      type color price
flower1 rose   red    1.5
flower2 daisy  white  0.5
flower3 tulip  yellow  1.0
```

# Coercing Data Frames Into Matrices Using `as.matrix()`

The function `as.matrix()` coerces vectors and data frames into matrices.

Coercing a data frame into a matrix causes coercion of numeric values into character.

`as.matrix()` coerces vectors into single column matrices, as opposed to `matrix()`, which produces a matrix.

```
> as.matrix(dframe)
      type color price
flower1 "rose"  "red"  "1.5"
flower2 "daisy" "white" "0.5"
flower3 "tulip" "yellow" "1.0"
> vectorv <- sample(9)
> matrix(vectorv, ncol=3)
      [,1] [,2] [,3]
[1,]    9    4    6
[2,]    1    2    7
[3,]    3    5    8
> as.matrix(vectorv, ncol=3)
      [,1]
[1,]    9
[2,]    1
[3,]    3
[4,]    4
[5,]    2
[6,]    5
[7,]    6
[8,]    7
[9,]    8
```

# Coercing Matrices Into Data Frames

The generic function `as.data.frame()` coerces matrices and other objects into data frames.

The method `as.data.frame.matrix()` coerces only matrices into data frames.

`as.data.frame.matrix()` is about 50% faster than `as.data.frame()`, because it skips extra R code in `as.data.frame()` needed for argument validation, error checking, and method dispatch.

As a general rule, calling generic functions is slower than directly calling individual methods, because generic functions must execute extra R code for method dispatch.

The function `data.frame()` can also be used to coerce matrices into data frames, but is much slower than even `as.data.frame()`.

`as.data.frame()` is about three times faster than `data.frame()`, because it doesn't require extra R code in `data.frame()` needed for handling different types of vectors, and for method dispatch.

```
> library(microbenchmark)
> # Call method instead of generic function
> as.data.frame.matrix(matrixv)
  col1 col2 col3
row1   5    7    9
row2   6    8   10
> # A few methods for generic function as.data.frame()
> sample(methods(as.data.frame), size=4)
[1] "as.data.frame.person" "as.data.frame.array"  "as.data.frame.s"
[4] "as.data.frame.default"
> # Function method is faster than generic function
> summary(microbenchmark(
+   as_dframe_matrix=
+     as.data.frame.matrix(matrixv),
+   as_dframe=as.data.frame(matrixv),
+   dframe=data.frame(matrixv),
+   times=10))[, c(1, 4, 5)] # end microbenchmark summary
      expr   mean median
1 as_dframe_matrix  7.24   7.24
2       as_dframe  7.99   7.87
3         dframe 39.99  34.32
```

# Coercing Matrices Into Lists

Matrices can be coerced into lists in at least two different ways.

Matrices can be first coerced into a data frame, and then into a list using function `as.list()`.

Matrices can be directly coerced into a list using function `lapply()`.

Using `lapply()` is the faster of the two methods, because `lapply()` is a *compiled* function.

```
> # lapply is faster than coercion function
> summary(microbenchmark(
+   aslist=as.list(as.data.frame.matrix(matrixv)),
+   lapply=lapply(seq_along(matrixv[1, ]),
+     function(indeks) matrixv[, indeks]),
+   times=10))[, c(1, 4, 5)] # end microbenchmark summary
      expr mean median
1 aslist   12    8.3
2 lapply  838    3.3
```

# The iris Data Frame

The iris data frame is included in the datasets base package.

iris contains sepal and petal dimensions of 50 flowers from 3 species of iris.

The function unique() extracts unique elements of an object.

sapply() applies a function to a list or a vector of objects and returns a vector.

sapply() performs a loop over the list of objects, and can replace "for" loops in R.

```
> # ?iris # Get information on iris
> dim(iris)
[1] 150 5
> head(iris, 2)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1         5.1         3.5          1.4          0.2  setosa
2         4.9         3.0          1.4          0.2  setosa
> colnames(iris)
[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"
> unique(iris$Species) # List of unique elements of iris
[1] setosa versicolor virginica
Levels: setosa versicolor virginica
> class(unique(iris$Species))
[1] "factor"
> # Find which columns of iris are numeric
> sapply(iris, is.numeric)
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          TRUE          TRUE          TRUE          TRUE        FALSE
> # Calculate means of iris columns
> sapply(iris, mean) # Returns NA for Species
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          5.84          3.06          3.76          1.20         NA
```

# The mtcars Data Frame

The mtcars data frame is included in the datasets base package, and contains design and performance data for 32 automobiles.

```
> # ?mtcars # mtcars data from 1974 Motor Trend magazine
> # mpg Miles/(US) gallon
> # qsec 1/4 mile time
> # hp Gross horsepower
> # wt Weight (lb/1000)
> # cyl Number of cylinders
> dim(mtcars)
[1] 32 11
> head(mtcars, 2)
      mpg   cyl  disp    hp  drat    wt  qsec vs am gear carb
Mazda RX4     21    6  160   110   3.9 2.62 16.5  0  1    4    4
Mazda RX4 Wag  21    6  160   110   3.9 2.88 17.0  0  1    4    4
> colnames(mtcars)
[1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am"
[11] "carb"
> head(rownames(mtcars), 3)
[1] "Mazda RX4"      "Mazda RX4 Wag"  "Datsun 710"
> unique(mtcars$cyl) # Extract list of car cylinders
[1] 6 4 8
> sapply(mtcars, mean) # Calculate means of mtcars columns
      mpg      cyl    disp      hp      drat      wt      qsec      vs
20.091  6.188 230.722 146.688   3.597   3.217 17.849   0.438 0
      carb
2.812
```

# The Cars93 Data Frame

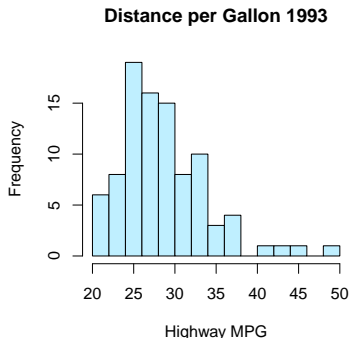
The Cars93 data frame is included in the MASS package, and contains design and performance data for 93 automobiles.

The function `hist()` calculates and plots a histogram, and returns its data *invisibly*.

The parameter `breaks` is the number of cells of the histogram.

"FD" stands for the Freedman-Diaconis rule for calculating histogram breaks,

```
> library(MASS)
> # ?Cars93 # Get information on Cars93
> dim(Cars93)
> head(colnames(Cars93))
> # head(Cars93, 2)
> unique(Cars93$Type) # Extract list of car types
> # sapply(Cars93, mean) # Calculate means of Cars93 columns
> # Plot histogram of Highway MPG using the Freedman-Diaconis rule
> hist(Cars93$MPG.highway, col="lightblue1",
+       main="Distance per Gallon 1993", xlab="Highway MPG", breaks="FD")
```





# Types of Bad Data

Possible sources of bad data are: imported data, class coercion, numeric overflow.

Types of bad data:

- NA (not available) is a logical constant indicating missing data,
- NaN means Not a Number data,
- Inf means numeric overflow - divide by zero,

When a function produces NA or NaN values, then it also produces a *warning* condition, but not an *error*.

NA or NaN values are not *errors*.

The functions `is.na()` and `is.nan()` test for NA and NaN values.

Many functions have a `na.rm` parameter to remove NAs from input data.

```
> as.numeric(c(1:3, "a")) # NA from coercion
[1] 1 2 3 NA
> 0/0 # NaN from ambiguous math
[1] NaN
> 1/0 # Inf from divide by zero
[1] Inf
> is.na(c(NA, NaN, 0/0, 1/0)) # Test for NA
[1] TRUE TRUE TRUE FALSE
> is.nan(c(NA, NaN, 0/0, 1/0)) # Test for NaN
[1] FALSE TRUE TRUE FALSE
> NA*1:4 # Create vector of NAs
[1] NA NA NA NA
> # Create vector with some NA values
> datav <- c(1, 2, NA, 4, NA, 5)
> datav
[1] 1 2 NA 4 NA 5
> mean(datav) # Returns NA, when NAs are input
[1] NA
> mean(datav, na.rm=TRUE) # remove NAs from input data
[1] 3
> datav[!is.na(datav)] # Delete the NA values
[1] 1 2 4 5
> sum(!is.na(datav)) # Count non-NA values
[1] 4
```

# Scrubbing Bad Data

The function `complete.cases()` returns TRUE if a row has no NA values.

```
> # airquality data has some NAs
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1   41    190  7.4   67     5   1
2   36    118  8.0   72     5   2
3   12    149 12.6   74     5   3
4   18    313 11.5   62     5   4
5   NA     NA 14.3   56     5   5
6   28     NA 14.9   66     5   6
> dim(airquality)
[1] 153  6
> # Number of NA elements
> sum(is.na(airquality))
[1] 44
> # Number of rows with NA elements
> sum(!complete.cases(airquality))
[1] 42
> # Display rows containing NAs
> head(airquality[!complete.cases(airquality), ])
  Ozone Solar.R Wind Temp Month Day
5     NA     NA 14.3   56     5   5
6     28     NA 14.9   66     5   6
10    NA    194  8.6   69     5  10
11     7     NA  6.9   74     5  11
25    NA     66 16.6   57     5  25
26    NA    266 14.9   58     5  26
```

# Scrubbing Data Using Carry Forward

Rows containing bad data may be either removed or replaced with an estimated value.

The function `stats::na.omit()` removes individual NA values from vectors, and it also removes whole rows of data containing NA values from matrices and data frames.

Bad data can also be replaced with the most recent prior values (carry forward good data).

The function `zoo::na.locf()` replaces NA values with the most recent non-NA values prior to it (*locf* stands for *last observation carry forward*).

Copying the last non-NA values forward causes less data loss than removing whole rows of data.

The function `na.locf()` with argument `fromLast=TRUE` replaces NA values with non-NA values in reverse order, starting from the end.

```
> # Create vector containing NA values
> vectorv <- sample(22)
> vectorv[sample(NROW(vectorv), 4)] <- NA
> # Replace NA values with the most recent non-NA values
> zoo::na.locf(vectorv)
[1] 13 13 6 8 8 5 5 18 17 17 20 21 11 9 3 12 19 10 2 7 15
> # Remove rows containing NAs
> good_air <- airquality[complete.cases(airquality), ]
> dim(good_air)
[1] 111 6
> # NAs removed
> head(good_air)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
8    19      99 13.8   59     5   8
> # Another way of removing NAs
> fresh_air <- na.omit(airquality)
> all.equal(fresh_air, good_air, check.attributes=FALSE)
[1] TRUE
> # Replace NAs
> good_air <- zoo::na.locf(airquality)
> dim(good_air)
[1] 153 6
> # NAs replaced
> head(good_air)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    18     313 14.3   56     5   5
6    28     313 14.9   66     5   6
```

# Scrubbing Time Series Data

Missing asset prices and returns can be replaced with the most recent prior values (carry forward good data).

But missing asset returns should not be replaced with values from the future. Instead, missing returns should be replaced with zero values.

The function `na.locf.xts()` from package `xts` is faster than `zoo::na.locf()`, but it only operates on time series of class "xts".

```
> # Replace NAs in xts time series
> library(rutils) # load package rutils
> pricev <- rutils::etfenv$prices[, 1]
> head(pricev, 3)
          DBC
2003-09-10 NA
2003-09-11 NA
2003-09-12 NA
> sum(is.na(pricev))
[1] 605
> pricez <- zoo::na.locf(pricev, fromLast=TRUE)
> pricex <- xts::na.locf.xts(pricev, fromLast=TRUE)
> all.equal(pricez, pricex, check.attributes=FALSE)
[1] TRUE
> head(pricex, 3)
          DBC
2003-09-10 24.5
2003-09-11 24.5
2003-09-12 24.5
> library(microbenchmark)
> summary(microbenchmark(
+   zoo=zoo::na.locf(pricev, fromLast=TRUE),
+   xts=xts::na.locf.xts(pricev, fromLast=TRUE),
+   times=10))[, c(1, 4, 5)] # end microbenchmark summary
      expr mean median
1  zoo 21.3   17.3
2  xts 17.1   16.4
```

# NULL Values

NULL represents a null object, and is a legitimate value, not bad data.

NULL is often returned by functions whose value is undefined.

NULL can also be used to initialize vectors.

NULL is not the same as NA values or zero-length (empty) vectors.

The functions `numeric()` and `character()` return empty (zero-length) vectors of the specified *type*.

The function `is.null()` tests for NULL values.

Very often variables are initialized to NULL before the start of iteration.

A more efficient way to perform iteration is by pre-allocating the vector.

```
> # NULL values have no mode or type
> c(mode(NULL), mode(NA))
[1] "NULL"      "logical"
> c(typeof(NULL), typeof(NA))
[1] "NULL"      "logical"
> c(length(NULL), length(NA))
[1] 0 1
> # Check for NULL values
> is.null(NULL)
[1] TRUE
> # NULL values are ignored when combined into a vector
> c(1, 2, NULL, 4, 5)
[1] 1 2 4 5
> # But NA value isn't ignored
> c(1, 2, NA, 4, 5)
[1] 1 2 NA 4 5
> # Vectors can be initialized to NULL
> vectorv <- NULL
> is.null(vectorv)
[1] TRUE
> # Grow the vector in a loop - very bad code!!!
> for (indeks in 1:5)
+   vectorv <- c(vectorv, indeks)
> # Initialize empty vector
> vectorv <- numeric()
> # Grow the vector in a loop - very bad code!!!
> for (indeks in 1:5)
+   vectorv <- c(vectorv, indeks)
> # Allocate vector
> vectorv <- numeric(5)
> # Assign to vector in a loop - good code
> for (indeks in 1:5)
+   vectorv[indeks] <- runif(1)
```

## Flattening a List of Vectors to a Matrix Using `do.call()`

A list of vectors can be flattened into a matrix using the functions `do.call()` and either `rbind()` or `cbind()`.

If the list contains vectors of different lengths, then R applies the recycling rule.

If the list contains a `NULL` element, that element is skipped.

```
> # Create list of vectors
> listv <- lapply(1:3, function(x) sample(6))
> # Bind list elements into matrix - doesn't work
> rbind(listv)
> # Bind list elements into matrix - tedious
> rbind(listv[[1]], listv[[2]], listv[[3]])
> # Bind list elements into matrix - works!
> do.call(rbind, listv)
> # Create numeric list
> listv <- list(1, 2, 3, 4)
> do.call(rbind, listv) # Returns single column matrix
> do.call(cbind, listv) # Returns single row matrix
> # Recycling rule applied
> do.call(cbind, list(1:2, 3:5))
> # NULL element is skipped
> do.call(cbind, list(1, NULL, 3, 4))
> # NA element isn't skipped
> do.call(cbind, list(1, NA, 3, 4))
```

# Efficient Binding of Lists Into Matrices

A list of vectors can be flattened into a matrix using the functions `do.call()` and either `rbind()` or `cbind()`.

But for large vectors this procedure can be very slow, and often causes an out of memory error.

The function `do_call_rbind()` efficiently combines a list of vectors into a matrix.

`do_call_rbind()` produces the same result as `do.call(rbind, list_var)`, but using recursion.

`do_call_rbind()` calls `lapply` in a loop, each time binding neighboring list elements and dividing the length of the list by half.

`do_call_rbind()` is the same function as `do.call.rbind()` from package *qmao*:

[https://r-forge.r-project.org/R/?group\\_id=1113](https://r-forge.r-project.org/R/?group_id=1113)

```
> list_vectors <- lapply(1:5, rnorm, n=10)
> matrixv <- do.call(rbind, list_vectors)
> dim(matrixv)
> do_call_rbind <- function(listv) {
+   while (NROW(listv) > 1) {
+     # Index of odd list elements
+     odd_index <- seq(from=1, to=NROW(listv), by=2)
+     # Bind odd and even elements, and divide listv by half
+     listv <- lapply(odd_index, function(indeks) {
+       if (indeks==NROW(listv)) return(listv[[indeks]])
+       rbind(listv[[indeks]], listv[[indeks+1]])
+     }) # end lapply
+   } # end while
+   # listv has only one element - return it
+   listv[[1]]
+ } # end do_call_rbind
> identical(matrixv, do_call_rbind(list_vectors))
```

# Filtering Data Frames Using subset()

*Filtering* means extracting rows from a *data frame* that satisfy a logical condition.

*Data frames* can be filtered using Boolean vectors and brackets "[]" operators.

The function `subset()` filters *data frames*, by applying logical conditions to its columns, using the column names.

`subset()` provides a succinct notation and discards NA values, but it's slightly slower than using Boolean vectors and brackets "[]" operators.

```
> airquality[(airquality$Solar.R > 320 &
+             !is.na(airquality$Solar.R)), ]
> subset(x=airquality, subset=(Solar.R > 320))
> summary(microbenchmark(
+   subset=subset(x=airquality, subset=(Solar.R > 320)),
+   brackets=airquality[(airquality$Solar.R > 320 &
+                         !is.na(airquality$Solar.R)), ],
+   times=10))[, c(1, 4, 5)] # end microbenchmark summary
```



# Splitting Data Frames Using factor Categorical Variables

The function `split()` divides an object into a list of objects, according to a factor (categorical variable).

The list's `names` attribute is equal to the factor levels.

```
> unique(iris$Species) # Species has three distinct values
> # Split into separate data frames by hand
> setosa <- iris[iris$Species=="setosa", ]
> versicolor <- iris[iris$Species=="versicolor", ]
> virginica <- iris[iris$Species=="virginica", ]
> dim(setosa)
> head(setosa, 2)
> # Split iris into list based on Species
> split_iris <- split(iris, iris$Species)
> str(split_iris, max.conf=1)
> names(split_iris)
> dim(split_iris$setosa)
> head(split_iris$setosa, 2)
> all.equal(setosa, split_iris$setosa)
```

# The *split-apply-combine* Procedure

The *split-apply-combine* procedure consists of:

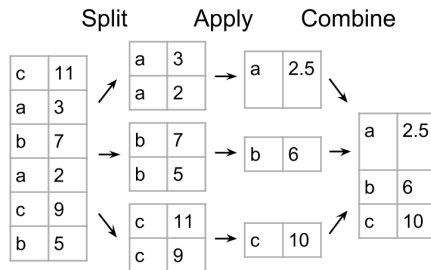
- dividing an object into a list, according to a factor (attribute).
- applying a function to each list element.
- combining the results.

The *split-apply-combine* procedure is also called the *map-reduce* procedure, or simply *data pivoting*, and it's similar to *pivot tables* in *Excel*.

*Data pivoting* can be performed *data frames*, by aggregating its columns based on categorical data stored in one of its columns.

You can read more about the *split-apply-combine* procedure in Hadley Wickham's paper:

<http://www.jstatsoft.org/v40/i01/paper>



## Data Pivoting Example

*Data pivoting* can be performed through successive applications of functions `split()`, `apply()`, and `unlist()`.

A *data frame* can be *pivoted* either by first splitting it into a list of *data frames* and then aggregating, or by splitting just a single column and aggregating it.

The function `split()` divides an object into a list of objects, according to a factor (categorical variable).

The list's `namesv` attribute is equal to the factor levels.

The functional `aggregate()` *pivots* the columns of a *data frame*.

`aggregate()` can accept a "formula" argument with the column names, or it can accept "x" and "by" arguments with the columns.

`aggregate()` returns a *data frame* containing the names of the groups (factor `confls`).

```
> unique(mtcars$cyl) # cyl has three unique values
> # Split mpg column based on number of cylinders
> split(mtcars$mpg, mtcars$cyl)
> # Split mtcars data frame based on number of cylinders
> split_cars <- split(mtcars, mtcars$cyl)
> str(split_cars, max.conf=1)
> names(split_cars)
> # Aggregate the mean mpg over split mtcars data frame
> sapply(split_cars, function(x) mean(x$mpg))
> # Or: split mpg column and aggregate the mean
> sapply(split(mtcars$mpg, mtcars$cyl), mean)
> # Same but using with()
> with(mtcars, sapply(split(mpg, cyl), mean))
> # Or: aggregate() using formula syntax
> aggregate(x=(mpg ~ cyl), data=mtcars, FUN=mean)
> # Or: aggregate() using data frame syntax
> aggregate(x=mtcars$mpg, by=list(cyl=mtcars$cyl), FUN=mean)
> # Or: using name for mpg
> aggregate(x=list(mpg=mtcars$mpg), by=list(cyl=mtcars$cyl), FUN=mean)
> # Aggregate() all columns
> aggregate(x=mtcars, by=list(cyl=mtcars$cyl), FUN=mean)
```

# The tapply() Functional

The functional `tapply()` is a specialized version of the `apply()` functional, that applies a function to elements of a *jagged array*.

A *jagged array* is a list consisting of vectors or matrices of different lengths.

`tapply()` accepts a vector of values "X", a factor "INDEX", and a function "FUN".

`tapply()` first groups the elements of "X" according to the factor "INDEX", transforming it into a *jagged array*, and then applies "FUN" to each element of the *jagged array*.

`tapply()` applies a function to sub-vectors aggregated using a factor, and performs *data pivoting* in a single function call.

The `by()` function is a wrapper for `tapply()`.

The `with()` function evaluates an expression in an environment constructed from the data.

```
> # Mean mpg for each cylinder group
> tapply(X=mtcars$mpg, INDEX=mtcars$cyl, FUN=mean)
> # using with() environment
> with(mtcars, tapply(X=mpg, INDEX=cyl, FUN=mean))
> # Function sapply() instead of tapply()
> with(mtcars, sapply(sort(unique(cyl)), function(x) {
+   structure(mean(mpg[x==cyl]), names=x)
+   }))) # end with
> # Function by() instead of tapply()
> with(mtcars, by(data=mpg, INDICES=cyl, FUN=mean))
```

## Data Pivoting Returning a Matrix

Sometimes *data pivoting* returns a list of vectors.

A list of vectors can be flattened into a matrix using the functions `do.call()` and either `rbind()` or `cbind()`.

The function `do.call()` executes a function call using a function name and a list of arguments.

`do.call()` passes the list elements individually, instead of passing the whole list as one argument:

```
do.call(fun, list)= fun(list[[1]], list[[2]],
...)
```

```
> # Get several mpg stats for each cylinder group
> data_cars <- sapply(split_cars,
+   function(x) {
+     c(mean=mean(x$mpg), max=max(x$mpg), min=min(x$mpg))
+   } # end anonymous function
+ ) # end sapply
> data_cars # sapply() produces a matrix
> data_cars <- lapply(split_cars, # Now same using lapply
+   function(x) {
+     c(mean=mean(x$mpg), max=max(x$mpg), min=min(x$mpg))
+   } # end anonymous function
+ ) # end lapply
> is.list(data_cars) # lapply produces a list
> # do.call flattens list into a matrix
> do.call(cbind, data_cars)
```

## Data Pivoting of Panel Data

The *data frame* `panel_data` contains fundamental financial data for *S&P500* stocks.

The *Industry* column has 22 unique elements, while the *Sector* column has 10 unique elements.

Each *Industry* belongs to a single *Sector*, but each *Sector* may have several *Industries* that belong to it.

The functional `aggregate()` allows aggregating over the *Industry* column, by performing *data pivoting*.

```
> # Download CRSPanel.txt from Brightspace
> # Read the file using read.table() with header and sep arguments
> panel_data <- read.table(file="/Users/jerzy/Develop/lecture_slides/
+   header=TRUE, sep="\t")
> attach(panel_data)
> # Split panel_data based on Industry column
> split_panel <- split(panel_data, panel_data$Industry)
> # Number of companies in each Industry
> sapply(split_panel, NROW)
> # Number of Sectors that each Industry belongs to
> sapply(split_panel, function(x) {
+   NROW(unique(x$Sector))
+ }) # end sapply
> # Or
> aggregate(x=(Sector ~ Industry),
+   data=panel_data, FUN=function(x) NROW(unique(x)))
> # Industries and the Sector to which they belong
> aggregate(x=(Sector ~ Industry), data=panel_data, FUN=unique)
> # Or
> aggregate(x=Sector, by=list(Industry), FUN=unique)
> # Or
> sapply(unique(Industry), function(x) {
+   Sector[match(x, Industry)]
+ }) # end sapply
```

## Data Pivoting Returning a Jagged Array

A *jagged array* is a list consisting of vectors or matrices of different lengths.

The functional `aggregate()` returns a *data frame*, so its output must be coerced if the *data pivoting* attempts to return a *jagged array*.

The functional `tapply()` returns an array, so its output must be coerced if the *data pivoting* attempts to return a *jagged array*.

`tapply()` accepts a vector of values "X", a factor "INDEX", and a function "FUN".

`tapply()` first groups the elements of "X" according to the factor "INDEX", transforming it into a *jagged array*, and then applies "FUN" to each element of the *jagged array*.

`tapply()` applies a function to sub-vectors aggregated using a factor, and performs *data pivoting* in a single function call.

```
> # Split panel_data based on Sector column
> split_panel <- split(panel_data, panel_data$Sector)
> # Number of companies in each Sector
> sapply(split_panel, NROW)
> # Industries belonging to each Sector (jagged array)
> sec_ind <- sapply(split_panel,
+   function(x) unique(x$Industry))
> # Or use aggregate() (returns a data frame)
> sec_ind2 <- aggregate(x=(Industry ~ Sector),
+   data=panel_data, FUN=function(x) unique(x))
> # Or use aggregate() with "by" argument
> sec_ind2 <- aggregate(x=Industry, by=list(Sector),
+   FUN=function(x) as.vector(unique(x)))
> # Coerce sec_ind2 into a jagged array
> namesv <- as.vector(sec_ind2[, 1])
> sec_ind2 <- sec_ind2[, 2]
> names(sec_ind2) <- namesv
> all.equal(sec_ind2, sec_ind)
> # Or use tapply() (returns an array)
> sec_ind2 <- tapply(X=Industry, INDEX=Sector, FUN=unique)
> # Coerce sec_ind2 into a jagged array
> sec_ind2 <- drop(as.matrix(sec_ind2))
> all.equal(sec_ind2, sec_ind)
```

# Data Pivoting Over Multiple Columns

*Data pivoting* over multiple columns can be performed by splitting the *data frame* and then performing an `sapply()` loop using an anonymous function.

Splitting the *data frame* allows aggregations over multiple columns.

An anonymous function allows applying different aggregations on the same column.

```
> # Average ROE in each Industry
> sapply(split(ROE, Industry), mean)
> # Average, min, and max ROE in each Industry
> t(sapply(split(ROE, Industry), FUN=function(x)
+   c(mean=mean(x), max=max(x), min=min(x))))
> # Split panel_data based on Industry column
> split_panel <- split(panel_data, panel_data$Industry)
> # Average ROE and EPS in each Industry
> t(sapply(split_panel, FUN=function(x)
+   c(mean_roe=mean(x$ROE),
+     mean_eps=mean(x$EPS.EXCLUDE.EI))))
> # Or: split panel_data based on Industry column
> split_panel <- split(panel_data[, c("ROE", "EPS.EXCLUDE.EI")],
+   panel_data$Industry)
> # Average ROE and EPS in each Industry
> t(sapply(split_panel, FUN=function(x) sapply(x, mean)))
> # Average ROE and EPS using aggregate()
> aggregate(x=panel_data[, c("ROE", "EPS.EXCLUDE.EI")],
+   by=list(panel_data$Industry), FUN=mean)
```



# Exception Conditions: Errors and Warnings

*Exception conditions* are R objects containing information about *errors* or *warnings* produced while evaluating expressions.

The function `warning()` produces a *warning* condition, but doesn't halt function execution, and returns its message to the warning handler.

The function `stop()` produces an *error* condition, halts function execution, and returns its message to the error handler.

The handling of *warning* conditions depends on the value of `options("warn")`:

- *negative* then warnings are ignored,
- *zero* then warnings are stored and printed after the top-level function has completed,
- *one* - warnings are printed as they occur,
- *two or larger* - warnings are turned into errors,

The function `suppressWarnings()` evaluates its expressions and ignores all warnings.

```
> # ?options # Get info on global options
> getOption("warn") # Global option for "warn"
> options("warn") # Global option for "warn"
> getOption("error") # Global option for "error"
> sqrt_safe <- function(input) {
+ # Returns its argument
+   if (input<0) {
+     warning("sqrt_safe: input is negative")
+     NULL # Return NULL for negative argument
+   } else {
+     sqrt(input)
+   } # end if
+ } # end sqrt_safe
> sqrt_safe(5)
> sqrt_safe(-1)
> options(warn=-1)
> sqrt_safe(-1)
> options(warn=0)
> sqrt_safe()
> options(warn=1)
> sqrt_safe()
> options(warn=3)
> sqrt_safe()
```

# Validating Function Arguments

Argument validation consists of first determining if any arguments are *missing*, and then determining if the arguments are of the correct *type*.

An argument is *missing* when the formal argument is not bound to an actual value in the function call.

The function `missing()` returns `TRUE` if an argument is missing, and `FALSE` otherwise.

Missing arguments can be detected by:

- assigning a `NULL` default value to formal arguments and then calling `is.null()` on them,
- calling the function `missing()` on the arguments.

The argument *type* can be validated using functions such as `is.numeric()`, `is.character()`, etc.

The function `return()` returns its argument and terminates further function execution.

```
> # Function valido validates its arguments
> valido <- function(input=NULL) {
+ # Check if argument is valid and return double
+   if (is.null(input)) {
+     return("valido: input is missing")
+   } else if (is.numeric(input)) {
+     2*input
+   } else cat("valido: input not numeric")
+ } # end valido
> valido(3)
> valido("a")
> valido()
> # valido validates arguments using missing()
> valido <- function(input) {
+ # Check if argument is valid and return double
+   if (missing(input)) {
+     return("valido: input is missing")
+   } else if (is.numeric(input)) {
+     2*input
+   } else cat("valido: input is not numeric")
+ } # end valido
> valido(3)
> valido("a")
> valido()
```

# Validating Assertions Inside Functions

If assertions about variables inside a function are FALSE, then `stop()` can be called to halt its execution.

Calling `stop()` is preferable to calling `return()`, or inserting `cat()` statements into the code.

Using `stop()` inside a function allows calling the function `traceback()`, if an error was produced.

The function `traceback()` prints the call stack, showing the function that produced the *error* condition.

`cat()` statements inside the function body provide information about the state of its variables.

```
> # valido() validates its arguments and assertions
> valido <- function(input) {
+ # Check if argument is valid and return double
+   if (missing(input)) {
+     stop("valido: input is missing")
+   } else if (!is.numeric(input)) {
+     cat("input =", input, "\n")
+     stop("valido: input is not numeric")
+   } else 2*input
+ } # end valido
> valido(3)
> valido("a")
> valido()
> # Print the call stack
> traceback()
```

# Validating Assertions Using stopifnot()

R provides robust validation and debugging tools through *type* validation functions, and functions `missing()`, `stop()`, and `stopifnot()`.

If the argument to function `stopifnot()` is `FALSE`, then it produces an *error* condition, and halts function execution.

`stopifnot()` is a convenience wrapper for `stop()`, and eliminates the need to use `if ()` statements.

`stopifnot()` is often used to check the validity of function arguments.

`stopifnot()` can be inserted anywhere in the function body in order to check assertions about its variables.

```
> valido <- function(input) {  
+ # Check argument using long form '&&' operator  
+   stopifnot(!missing(input) && is.numeric(input))  
+   2*input  
+ } # end valido  
> valido(3)  
> valido()  
> valido("a")  
  
> valido <- function(input) {  
+ # Check argument using logical '&' operator  
+   stopifnot(!missing(input) & is.numeric(input))  
+   2*input  
+ } # end valido  
> valido()  
> valido("a")
```

# Validating Function Arguments and Assertions

The functions `stop()` and `stopifnot()` halt function execution and produce *error* conditions if certain assertions are FALSE.

The *type* validation functions, such as `is.numeric()`, `is.na()`, etc., and `missing()`, allow for validation of arguments and variables inside functions.

`cat()` statements can provide information about the state of variables inside a function.

`cat()` statements don't return values, so they provide information even when a function produces an error.

```
> # sumtwo() returns the sum of its two arguments
> sumtwo <- function(input1, input2) { # Even more robust
+   # Check if at least one argument is not missing
+   stopifnot(!missing(input1) &&
+             !missing(input2))
+   # Check if arguments are valid and return sum
+   if (is.numeric(input1) && is.numeric(input2)) {
+     input1 + input2 # Both valid
+   } else if (is.numeric(input1)) {
+     cat("input2 is not numeric\n")
+     input1 # input1 is valid
+   } else if (is.numeric(input2)) {
+     cat("input1 is not numeric\n")
+     input2 # input2 is valid
+   } else {
+     stop("none of the arguments are numeric")
+   }
+ } # end sumtwo
> sumtwo(1, 2)
> sumtwo(5, 'a')
> sumtwo('a', 5)
> sumtwo('a', 'b')
> sumtwo()
```

# The R Debugger Facility

The function `debug()` flags a function for future debugging, but doesn't invoke the debugger.

After a function is flagged for debugging with the call `"debug(myfun)"`, then the function call `"myfun()"` automatically invokes the debugger (browser).

When the debugger is first invoked, it prints the function code to the console, and produces a *browser* prompt: `"Browse[2]>"`.

Once inside the debugger, the user can execute the function code one command at a time by pressing the *Enter* key.

The user can examine the function arguments and variables with standard R commands, and can also change the values of objects or create new ones.

The command `"c"` executes the remainder of the function code without pausing.

The command `"Q"` exits the debugger (browser).

The call `"undebug(myfun)"` at the R prompt unflags the function for debugging.

```
> # Flag "valido" for debugging
> debug(valido)
> # Calling "valido" starts debugger
> valido(3)
> # unflag "valido" for debugging
> undebug(valido)
```

## Debugging Using browser()

As an alternative to flagging a function for debugging, the user can insert the function `browser()` into the function body.

`browser()` pauses the execution of a function and invokes the debugger (browser) at the point where `browser()` was called.

Once inside the debugger, the user can execute all the same browser commands as when using `debug()`.

`browser()` is usually inserted just before the command that is suspected of producing an *error* condition.

Another alternative to flagging a function for debugging, or inserting `browser()` calls, is setting the "error" option equal to "recover".

Setting the "error" option equal to "recover" automatically invokes the debugger when an *error* condition is produced.

```
> valido <- function(input) {  
+   browser() # Pause and invoke debugger  
+ # Check argument using long form '&&' operator  
+   stopifnot(!missing(input) && is.numeric(input))  
+   2*input  
+ } # end valido  
> valido() # Invokes debugger  
> options("error") # Show default NULL "error" option  
> options(error=recover) # Set "error" option to "recover"  
> options(error=NULL) # Set back to default "error" option
```

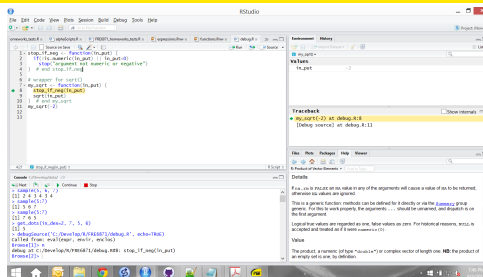
# Using the Debugger in RStudio

RStudio has several built-in debugging facilities that complement those already installed in R:

- toggling breakpoints, instead of inserting `browser()` commands,
- stepping into functions,
- environment pane with environment stack, instead of calling `ls()`,
- traceback pane, instead of calling `traceback()`,

RStudio provides an online debugging tutorial:

<https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio>





# Handling Exception Conditions

The function `tryCatch()` executes functions and expressions, and handles any *exception conditions* produced when they are evaluated.

`tryCatch()` first evaluates its "expression" argument.

If no error or warning condition is produced then `tryCatch()` just returns the value of the expression.

If an exception condition is produced then `tryCatch()` invokes error and warning *handlers* and executes other expressions to provide information about the exception condition.

If a *handler* is provided to `tryCatch()` then the error is captured by the *handler*, instead of being broadcast to the console.

At the end, `tryCatch()` evaluates the expression provided to the `finally` argument.

```
> str(tryCatch) # Get arguments of tryCatch()
> tryCatch( # Without error handler
+ { # Evaluate expressions
+   numv <- 101 # Assign
+   stop('my error') # Produce error
+ },
+ finally=print(paste("numv=", numv))
+ ) # end tryCatch
>
> tryCatch( # With error handler
+ { # Evaluate expressions
+   numv <- 101 # Assign
+   stop('my error') # Produce error
+ },
+ # Error handler captures error condition
+ error=function(error_cond) {
+   print(paste("error handler: ", error_cond))
+ }, # end error handler
+ # Warning handler captures warning condition
+ warning=function(warning_cond) {
+   print(paste("warning handler: ", warning_cond))
+ }, # end warning handler
+ finally=print(paste("numv=", numv))
+ ) # end tryCatch
```

# Error Conditions in Loops

If an *error* occurs in an `apply()` loop, then the loop exits without returning any result.

`apply()` collects the values returned by the function supplied to its `FUN` argument, and returns them only after the loop is finished.

If one of the function calls produces an error, then the loop is interrupted and `apply()` exits without returning any result.

The function `tryCatch()` captures errors, allowing loops to continue after the error condition.

```
> # Apply loop without tryCatch
> apply(matrix(1:5), 1, function(numv) { # Anonymous function
+   stopifnot(!(numv == 3)) # Check for error
+   # Broadcast message to console
+   cat("(cat) numv =", numv, "\n")
+   # Return a value
+   paste("(return) numv =", numv)
+ }) # end anonymous function
+ ) # end apply
```

# Exception Handling in Loops

If the body of the function supplied to the `FUN` argument is wrapped in `tryCatch()`, then the loop can finish without interruption and return its results.

The messages produced by *errors* and *warnings* can be caught by *handlers* (functions) that are supplied to `tryCatch()`.

The *error* and *warning* messages are bound (passed) to the formal arguments of the *handler* functions that are supplied to `tryCatch()`.

`tryCatch()` always evaluates the expression provided to the *finally* argument, even after an *error* occurs.

```
> # Apply loop with tryCatch
> apply(as.matrix(1:5), 1, function(numv) { # Anonymous function
+   tryCatch( # With error handler
+ { # Body
+   stopifnot(numv != 3) # Check for error
+   # Broadcast message to console
+   cat("(cat) numv =", numv, "\t")
+   # Return a value
+   paste("(return) numv =", numv)
+ },
+ # Error handler captures error condition
+ error=function(error_cond)
+   paste("handler: ", error_cond),
+ finally=print(paste("(finally) numv =", numv))
+ ) # end tryCatch
+ } # end anonymous function
+ ) # end apply
(cat) numv = 1 [1] "(finally) numv = 1"
(cat) numv = 2 [1] "(finally) numv = 2"
[1] "(finally) numv = 3"
(cat) numv = 4 [1] "(finally) numv = 4"
(cat) numv = 5 [1] "(finally) numv = 5"
[1] "(return) numv = 1"
[2] "(return) numv = 2"
[3] "handler: Error in FUN(newX[, i], ...): numv != 3 is not TRUE"
[4] "(return) numv = 4"
[5] "(return) numv = 5"
```

# Homework Assignment

## Required

- Study all the lecture slides in *FRE6871\_Lecture\_6.pdf*, and run all the code in *FRE6871\_Lecture\_6.R*

## Recommended

- Read about *PCA* in:  
*pca-handout.pdf*  
*pcaTutorial.pdf*
- Read about *optimization methods*:  
*Bolker Optimization Methods.pdf*  
*Yollin Optimization.pdf*  
*Boudt DEoptim Large Portfolio Optimization.pdf*