R Programming

Coursera Course by John Hopkins University

INSTRUCTORS: Dr. Jeff Leek, Dr. Roger D. Peng, Dr. Brian Caffo

Contents

Overview of R, R data types and objects, reading and writing data

Installing R & RStudio

- This was covered in the previous course.

RMarkdown reference site

- I found a site that expands on some features of R-Markdown and have been referencing it pretty regularly

Swirl

- swirl teaches you R programming and data science interactively, at your own pace, and right in the R console.

- Start swirl
  - install the package "swirl" if you haven't yet

  - Everytime you want to run swirl execute:
    * library("swirl")

    * swirl()

  - You'll then be prompted to install a course

  - Help page for swirl

History of S and R programming

- What is S?
  - R is a dialect of S

  - S was developed by John Chambers and others at Bell Labs

  - Initiated in 1976 as an internal statistical analysis environment, implemented as Fortran libraries
    * Early versions did not contain functions for statistical modeling

  - Version 3 was released in 1988, which was rewritten in C and began to resemble the system that we have today.

  - Version 4 was released in 1998 and is the version we use today.
    * This version is documented in *Programming with Data* by John Chambers (the green book)
  - Insightful sells its implementation of the S language under the name *S-PLUS*, which includes a number of fancy features, mostly GUIs.
  - S won the Association for Computing Machinery's Software System Award in '98

  - (More about S)[https://web.archive.org/web/20181014111802/ect.bell-labs.com/sl/S/]
- What is R?
  - R was developed by Ross Ihaka and Robert Gentleman, they documented thier experience in a (1996 JCGS paper)[https://amstat.tandfonline.com/doi/abs/10.1080/10618600.1996.10474713].

  - In 1995, R become free software after Martin Machler convinced Ross & Robert to use the GNU (General Public License)

- Versions
    - R version 1.0.0 was released in 2000
    - R version 3.0.2 is released in Dec. 2013
- Syntax is similar to S, making it easy for S-PLUS users to switch over
- Runs on almost any standard computing platform/OS (even on the PS3)
- Frequent releases; active development and communities
- Funtionality is divided into modular packages as to keep it "lean"
- It's free!
- What is free about Free Software?
    - Freedom 0: freedom to run the program, for any purpose
    - Freedom 1: freedom to study how the program works, and adapt it to one's needs. Which implies access to the source code
    - Freedom 2: freedom to redistribute copies
    - Freedom 3: freedom to improve the program, and release your improvements to the public, or to sell them.
    - These are outlined by the (Free Software Foundation)[https://www.fsf.org/]

- Drawbacks of R
    - Essentially based on 40 year old technology,the original S language
    - Little build support for dynamic or 3D graphics. Although there are packages for such
    - Functionality is based on consumer demand and use contributions, if a feature is not present you'll have to build it.
    - Objects that are manipulated in R have to be stored in the physical memory of the computer, as such if an object is bigger than the memory you'll be unable to load it into memory
    - Not ideal for all possible situations, such as calling to order pizza (but this is a drawback of all software packages)

*Design of the R System
+ "base" R system that can be downloaded from (CRAN)[http://cran.r-project.org] (krey-an) which...
- contains the packages: **utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4**.
- and "Recommends" the packages: **boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix**.
+ Packages are available all around the web, but packages on CRAN have to meet a certain level of quality.

- Some Useful Books on S/R
    - Chambers (2008). *Software for Data Analysis*, Springer.

- Chambers (1998). *Programming with Data*, Springer.
- Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.
- Venables & Ripley (2000). *S Programming*, Springer.
- Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-Plus*, Springer.
- Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press.
- (Additional Books)[http://www.r-project.org/doc/bib/R-books.html]

Review of getting help

- Covered in previous course

Input and Evaluation: Vocabulary/Syntax

- **Expressions -** The code that is typed into the R prompt.
- **Assignment Operator -** assigns a value to a symbol, Ex:
  `x <- 1`
- *Output a variable*:

```
x <- 36
print(x) ##explicit printing
```

```
## [1] 36
```

```
## or one can just type the variable
x ##auto-printing
```

```
## [1] 36
```

- *Comment:* Use a Hash(#) symbol to make a comment to the right of #
- *[1]* is indicating the following variable is the first element of the vector

```
x <- 1:30 ##Loads x with the numbers 1 to 30
print(x)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30
```

```
## here, [26] is telling you the next number is the 26th element of the vector
```

- `Inf` - represents infinity and can be used in ordinary calculations (Ex: 1 / `Inf` is 0)

- `Nan` - represents an undefined value ("not a number") (Ex: 0/0 is `NaN`).
  - Can also be thought of as a missing value

- **Attributes -** Some objects in R come with attributes. These attributes can be set or modified with the expression `attributes()`. They are:
  - names, dimnames (dimension names)
  - dimensions (e.g. matrices, arrays) - number of rows & cols, or more depending on dimensions of array
  - class - the data type of the object
  - length - number of elements
  - other user-defined attributes/metadata can be added

- **Coercion -** occurs so that every element of a vector is of the same class (Covered further in Vector section)

Different atomic data types

- R has five basic, or "atomic", classes of objects:
  - character
    - In `R` there is no `string` data type. It is also considered part of the `character` data type
  - numeric (real numbers)
    - R thinks as numbers as these by default

  - integer
    - Must be explicity declared with the L suffix; `x <- 1` assigns a numeric object, but `x <- 1L` explicitly assigns an integer

  - complex

  - logical (True/False)
- A vector can only contain objects of the same class
  - an empty vector can be created with `vector()`

- However, a **list** is represented as a vector but can contain objects of different classes (as such we usually use these)

Vectors, Lists, and Matrices

- The `c()` function (can be thought to stand for "concatenate")
  - Can be used to create vectors of objects

```
x <- c(0.5, 0.6) ## numeric
x <- c(TRUE, FALSE) ## logical
x <- c(T, F) ## logical
x <- c("a", "b", "c") ## character
x <- c(1+0i, 2+4i) ## complex
```

- The `vector()` function
  - Can also be used to create, you guessed it, vectors

```
x <- vector() ## Creates an empty vector
x ## Prints as code that evaluates as FALSE
```

```
## logical(0)
```

```
x <- vector(mode = "numeric", length = 10) ## Creates a vector with length "10" of
## numeric data type,default value is 0
x
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
x <- vector("numeric", 5) ##The parameter names are not required, but can easily clarify code
x
```

```
## [1] 0 0 0 0 0
```

- When different objects are mixed in a vector, **coercion** occurs so all objects are of the same class.
  - R will implicitly create the "Least Common Denominator" of the mixed classes

```
y <- c(1.7, "a") ## character
y
```

```
## [1] "1.7" "a"
```

```
y <- c(TRUE, 2) ## numeric
y
```

```
## [1] 1 2
```

```
y <- c("a", TRUE) ## character
y
```

```
## [1] "a"    "TRUE"
```

```
y[2] ## "TRUE" is a string stored as a "character" data type
```

```
## [1] "TRUE"
```

```
y[3] ## The third element does not exist
```

```
## [1] NA
```

- Objects can be **explicitly coerced** from onne class to another using the `as.*` functions, if available.
  - Nonsensical coercion reasults in `NA`s

```
x <- 0:6
class(x)
```

```
## [1] "integer"
```

```r
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```r
as.logical(x)
```

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

```r
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```r
as.complex(x)
```

```
## [1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

```r
x
```

```
## [1] 0 1 2 3 4 5 6
```

```r
y <- as.character(x)
y
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```r
x <- c("a", "b", "c")
as.numeric(x) ##Nonsensical coercion will also show a warning
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA
```

```r
as.logical(x)
```

```
## [1] NA NA NA
```

```r
as.complex(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA
```

- Lists (Important data type in R that you should get to know well)
  - Lists are a type of vector that can contain elements of different classes.
  - Doesn't print like a vector because every element is different
    - prints index of element with double brackets bordering it: `[[1]]`

```r
x <- list(1, "a", TRUE, 1 + 4i, 16 +18i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
```

```
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
##
## [[5]]
## [1] 16+18i
```

- **Matrices -** a type of vector with a *dimension* attribute.
    - The *dimension* attribute is itself an integer vector of length 2 (numRows, numCols)
    - Constructed *column-wise*, so entries can be thought of starting in the "upper left" corner, then running down the columns

    - Matrices can also be created by adding a *dimension* attribute to an existing vector

```
m <- matrix(nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

```
dim(m)##reports num of rows then cols
```

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

```
attributes(m) ## dim is an attribute of the vector
```

```
## $dim
## [1] 2 3
```

```
m <- matrix(1:6, 2, 3) ## Demonstrating column-wise filling of matrix
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
m <- 1:10 ## m is now just a vector
m
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
dim(m) <- c(2,5) ## adding the dimension attribute
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

- Creating a matrix with **cbind** and **rbind**
    - cbind fills the columns with the elements of the vectors that are passed as the respective parameters

    - likewise, rbind fills the rows with the elements of the respective parameters

```r
x <- 1:3
y <- 10:12
cbind(x,y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```r
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Other data types

- Factors
    - Used to represent categorical data

    - can be unordered or ordered
    - Kinda like enumerated data, where it's an integer at heart, and each integer has a *label*
    - Using factors with labels is *better* than using integers because factors are self-describing
        * consider "Male" and "Female" as opposed to just the values 1 and 2
    - Prints differently than a character value, does not include quotations and displays *Levels*

```r
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

```r
table(x) ## displays a frequency table of the factors
```

```
## x
##  no yes
##   2   3
```

```r
unclass(x) ## strips out the class and displays the underlying integer vector
```

```
## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

- The order of the levels can be set with the `levels` argument to `factor()`
    - This can be important in linear modelling because the first level is sued as the baseline level.
    - default levels are based alphabetically

```
x <- factor(
        c("yes", "yes", "no", "yes", "no"),
        levels = c("yes", "no")
        )
x
```

```
## [1] yes yes no  yes no
## Levels: yes no
```

- Missing Values (`NA` or `NaN`)
  - `NaN` is for undefined mathematical operations
  - `is.na()` and `is.nan()` are logical tests for the respective missing values
  - `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
  - a `NaN` is also a `NA`, however the converse is not true

```
x <- c(1, 2, NA, 10, 3)
is.na(x)
```

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

```
is.nan(x)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

```
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

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

```
is.nan(x)
```

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

- Data Frames
  - Used to store tabular data
  - Special type of list where every element has to have the same length
  - Each element is like a column and the length of each element is the number of rows
  - like lists, Data Frames can store different classes in each column
  - Attribute: `row.names`
    - Useful for annotating data
    - However, often the row names are not interesting and we use "1, 2, 3…"
  - Usually created by calling `read.table()` or `read.csv()`
  - Can be converted to a matrix with `data.matrix()`
    - Forces each object to be coerced

```
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))## cols are named here
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```r
nrow(x)
```

```
## [1] 4
```

```r
ncol(x)
```

```
## [1] 2
```

```r
row.names(x)
```

```
## [1] "1" "2" "3" "4"
```

- Names Attribute, useful for writing readable code and self-describing objects
    - Any R object can have names

```r
x <- 1:3
names(x)## by default there are no names
```

```
## NULL
```

```r
names(x) <- c("foo", "bar", "norf")
x
```

```
##  foo  bar norf
##    1    2    3
```

```r
names(x)
```

```
## [1] "foo"  "bar"  "norf"
```

```r
##Lists can also have names
x <- list(a=1, b=2, c=3) ## here, names are assigned as list is established
x
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

```r
## Matrices can also have names, called dimnames
m <- matrix(1:4, nrow = 2, ncol = 2)
m
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```r
dimnames(m) <- list(c("a", "b"), c("c", "d")) ##First vector is rownames, second is colnames
m
```

```
##   c d
```

```
## a 1 3
## b 2 4
```

Reading Data

Tabular Data

- Functions for **reading** data into R
  - `read.table`,`read.csv` - for reading tabular data
    - most common

    - reads in data that's organized into rows and cols
    - returns a data frame
  - `readLines`, for reading lines of a text file

  - `source`, for reading in R code files (inverse of `dump`)

  - `dget`, for reading in R code files (inverse of `dput`)

  - `load`, for reading in saved workspaces

  - `unserialize`, for reading single R objects in binary form

- Functions for **writing** data from R to files
  - `write.table`

  - `writeLines`

  - `dump`

  - `dput`

  - `save`

  - `serialize`

- Arguments of `read.table` function
  - `file` - the name of a file or connection

  - `header` - logical that indicates if the file has a header line

  - `sep` - a string that indicates how the columns are separated (tokens)

  - `colClasses` - a character vector that indicates the class (Data type) of each column

  - `nrows`

  - `comment.char` - character string that indicates the comment character (default is '#')

  - `skip` - number of lines to skip from the beginning

  - `stringsAsFactors` - (default = TRUE) should character variables be coded as factors?

- Implicit actions `R` takes

```
data <- read.table("foo.txt")
## Header must not have a label for the row labels for R to implicitly determine them
data
```

```
##             Price Num_Sold In_Stock Complex_Num
## Chips&Salsa  2.55     1729     TRUE       1+ 2i
## Drink        1.99     3435     TRUE       5+18i
## Taco         3.49       36    FALSE       3+ 0i
```

- Skips lines that begin with a #
- figures out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table.
  - Telling R all these things directly will make it run faster and more efficiently
- `read.csv` is identical to `read.table` except that the default separator is a comma
  - `.csv` files are common output from excel or other spreadsheet programs.

Large Datasets

- Doing the following things will make your life easier and prevent R from "choking"
  - Read the help page for read.table, which contains many hints

  - Make a rough calculation of the memory required to store your dataset.
    * Say for example, you have a data frame with 1,500,000 rows and 120 columns (not *that* big), all of which are numeric data. To roughly claculate how much memory is requrired..

    * 1,500,000 * 120 * 8 bytes/numeric = 1440000000 bytes

    * 1440000000 bytes / 2^{20} bytes/MB = 1,373.29 MB

    * 1,373.29 MB = 1.37 GB

    * Rule of thumb is that you'll need twice the amount of RAM to be able to read in the dataset

  - If the dataset is larger than the amount of RAM on your computer you can probably stop right here.
    * Type `free -k` in terminal to return amount of RAM in kilobytes (`-b` for bytes, `-m` for megabytes and `-g` for gigabytes)
  - Set `comment.char = ""` if there are no commented lines in your file.

  - Use the `colClasses` argument.
    - Specifying this option instead of using the default can make `read.table` run *MUCH* faster.
    - To use this option you have to know the class of each column in your data frame.
    - If all of the columns are of the same data type, for example "numeric", then you can just set `colClasses = "numeric"`
    - A quick and dirty way to figure out the clases of each column is to take a small sample and determine it from that.

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
classes ##Coded file wrong, I'm not gonna fix it now, Boolean should have said "TRUE" or "FALS
```

```
##     Index   Boolean
## "integer" "integer"
```

```
tabAll <- read.table("datatable.txt",
                     colClasses = classes)
```

- Set `nrows`
  - This doesn't make R run faster but it helps with memory usage.
  - A mild overestimate is okay.
  - You can type `wc <filename>` in terminal to return the number of: lines, strings, characters; "lines" are the `nrows`.
- When using R with larger datasets it's useful to know a few things about your system
  - How much memory is available
    - Type `free -k` into terminal
  - What other applications are in use
    - Type `ps aux` in terminal
  - Are there other users logged into the same system
    - Type `w` in terminal (Note: `last` will report a history)
  - What OS are you using
    - Type `lsb_release -a` into terminal
  - Is the OS 32 or 64 bit
    - Type `lscpu`, listed under first two returns
    - On a 64 bit system you'll generallly be able to access more memory

Textual Formats

- Contains the metadata, such as classes of columns, making transfering data more efficent as the metadata doesn't need to be determined again.

- Known as `dumping` and `dputing`.
- Edit-able, which in the case of corruption allows for a potential recovery.
- Textual formats can work much better with version control programs.

- Adhere to the "Unix philosophy", which is to store data as text
- *Downside:* The format is not very space-efficient and as such ussually requires compression

- `dput` will deparse an R object, and `dget` can read the data back in from a file

```
y <- data.frame(a=1, b="a")
dput(y) ## If file is not specified the output is displayed in the console
```

```
## structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), class = "data.fra
## -1L))
```

```
dput(y, file = "y.R")
new.y <- dget("y.R")##dget retrieves the object from a file
new.y
```

```
##   a b
## 1 1 a
```

- Multiple objects can be deparsed using the `dump` function, then read back in with `source`
  - The parameter for `dump` is a character vector that contains characters for the names of the variables one wishes to dump

```
x <- "foo"
y <- data.frame(a=1, b="a")
dump(c("x", "y"))
dump(c("x", "y"), file = "data.R")
rm(x, y) ## removes the variables
source("data.R") ## reconstructs y and x objects
y
```

```
##   a b
## 1 1 a
```

```
x
```

```
## [1] "foo"
```

Connections (Interfaces to the outside world)

- Connections can be made to files or to other, more "exotic" things.
  - `file` - opens a connection to a file

  - `gzfile` - opens a connection to a file compressed with *gzip*.

  - `bzfile` - opens a connection to a file compressed with *bzip2*.

  - `url` - opens a connection to a webpage (in HTML format).

- Arguments
  - `description` is the name of the file

- `open` indicates how the file is opened
    * **"r" -** read only
    * **"w" -** writing (and initializing a new file)
    * **"a" -** appending
    * **"rb", "wb", "ab" -** reading, writing, or appending in binary mode (Windows)
    * There are other options but they aren't uber important
- Connections are powerful tools that allow you to navigate files or other external objects in a more "sophisticated" way.
  - However, one does not need to deal with the connection interface in many case

```
con <- file("foo.txt", "r")
data <- read.csv(con)
close(con)
```

- ^This is the same as..

```
data <- read.csv("foo.txt")
```

- As such, the connection was not nessacary for this case
- Reading lines of a text file with `con` from a *gzip* file

```
con <- gzfile("words.gz")
x <- readLines(con, 10) ##reads in first 10 lines
x
```

```
##  [1] "1080"     "10-point" "10th"     "11-point" "12-point" "16-point"
##  [7] "18-point" "1st"      "2"        "20-point"
```

- `writeLines` takes a character vector and writes each element one line at a time to a text file
- `readLines` can be used for reading in lines of webpages.

```
## This might take time
con <- url("http://www.jhsph.edu", "r") ##John Hopkin's School of Public Health
x <- readLines(con)
head(x) ##Displays the header
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html lang=\"en\">"
## [3] ""
## [4] "<head>"
## [5] "<meta charset=\"utf-8\" />"
## [6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

Subsetting R objects using the "[", "[[", and "$" operators and logical vectors

Basics

- Operators to extract subsets of R objects
  - `[` always returns an object of the same class as the original
    - subsetting a vector will return a vector, a list will return a list, etc.
    - Can be used to select more than one element (there is one exception)
  - `[[` is used to extract elements of a *list* or *data frame*
    - Can only be used to extract a single element
    - The class of the returned object will not necessarily be a list or data frame
  - `$` is used to extract elements of a *list* or *data frame* by name
    - Similar to `[[` as it may not be of the same class
- Numerical Index for subsetting:

```
x <- c("a", "b", "c", "c", "d", "a")
x[1] ## Returns first element
```

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

```
x[2] ## Returns second element
```

```
## [1] "b"
```

```
x[1:4] ## Returns first to fourth elements
```

```
## [1] "a" "b" "c" "c"
```

- Logical Index for subsetting:

```
x <- c("a", "b", "c", "c", "d", "a")
x[x > "a"]## returns all elements that are greater than "a"
```

```
## [1] "b" "c" "c" "d"
```

```
u <- x > "a" ## u is a logical vector that indicates which elements of x are greater than "a"
u
```

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

```
x[u] ## subsets all elements of x such that u reports that index as TRUE; elements that are >
```

```
## [1] "b" "c" "c" "d"
```

Lists

- Lists can be subsetted with the `[[` or `$` operators

```
x <- list(foo = 1:4, bar = 0.6)
x[1]##Extracts the first element as a list, since the orginal set was a list class
```

```
## $foo
## [1] 1 2 3 4
```

```r
x[[1]]##Extracts the first element as a sequence, not a list
```

```
## [1] 1 2 3 4
```

```r
x$bar ##returns the element that is associated with the name "bar"
```

```
## [1] 0.6
```

```r
x[["bar"]]##same as x$bar
```

```
## [1] 0.6
```

```r
x["bar"]##returns a list with the element "bar" in it
```

```
## $bar
## [1] 0.6
```

- subsetting with the name is helpful when the index isn't known
- To extract multiple elements of a list, one must use the single bracket operator `[`

```r
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]##extracts the first and third element of the list
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

- The `[[` operator can be used with *computed* indices, whereas `$` can only be used with literal names.

```r
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"
x[[name]] ## computed index for 'foo'
```

```
## [1] 1 2 3 4
```

```r
x$name ## element 'name' doesn't exist!
```

```
## NULL
```

```r
x$foo ## element 'foo' does exist
```

```
## [1] 1 2 3 4
```

- The `[[` can also take an integer sequence instead of a single number

```r
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1,3)]] ##extracts first element, then the third element of said first element
```

```
## [1] 14
```

```r
x[[1]][[3]] ##equivelent
```

```
## [1] 14
```

```r
x[[c(2,1)]]##extracts first element of the second element of x
```

```
## [1] 3.14
```

Matrices

Partial Matching

Removing missing (NA) values from a vector

Basic Arithmetic operations

Control structures, functions, scoping rules, dates and times

Loop functions, debugging tools

Simulation, code profiling