Data management with R

What we have covered

- Data management with Python
  - List/Dictionary/Pandas series/dataframe
  - Filtering
  - Grouping
  - Joining
  - Datetime management
- Introduction of machine learning
  - Supervised learning/unsupervised learning
  - Data mining process
  - Decision tree algorithm
  - Clustering algorithm

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
  - Dealing with characters
  - Dealing with factors
  - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Agenda

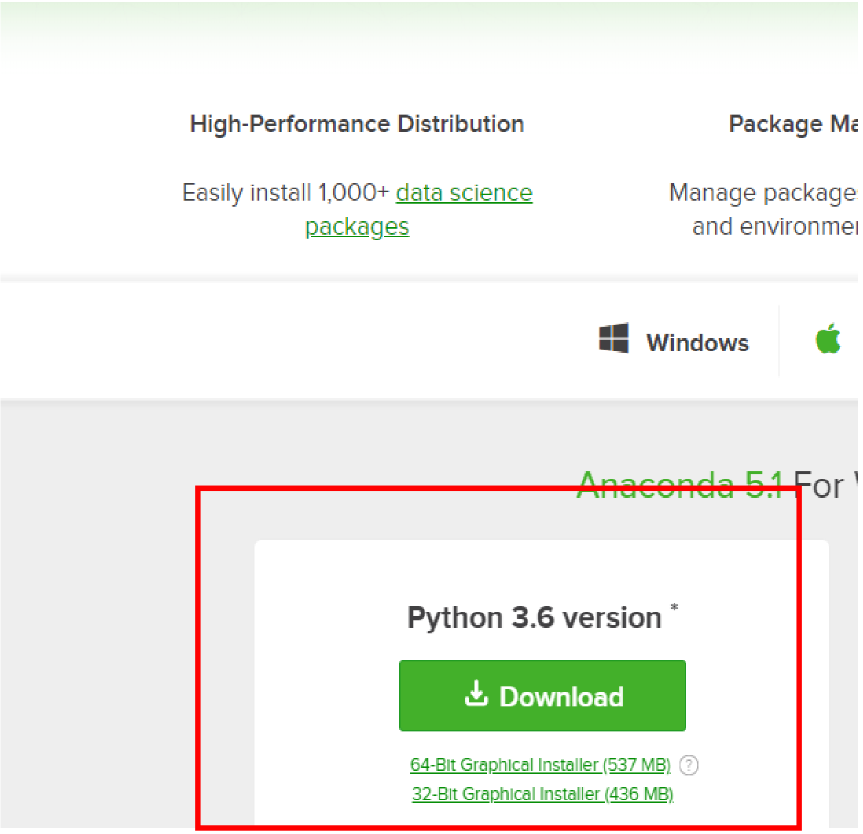
- **What is R and Why R**
- Vectorization
  - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
  - Dealing with characters
  - Dealing with factors
  - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

What is R

- **R is a comprehensive statistical platform, offering all manner of data-analytic techniques. Just about any type of data analysis can be done in R**
- **R contains advanced statistical routines not yet available in other packages**
- **R has the state-of-art graphics capability. If you want to visualize complex data, R has the most comprehensive and power feature set available.**
- **http://cran.r-project.org**
- **R studio (https://www.rstudio.com/) is a good GUI platform with R**

Install Jupyter

- https://www.anaconda.com/download/
- conda install -c r r-essentials



Why R

- Open source
- Flexibile
- It comes with lots of useful modules which support different topics on data analytic
- Lots of good supporting material available
- It comes with very good features on the following items:
    - Data visualization support (e.g. ggplot2)
    - Reporting feature (e.g. Rmarkup)
    - Data management
    - Data analytic

Agenda

- What is R and Why R
- **Vectorization**
- Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
    - Dealing with characters
    - Dealing with factors
    - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Vectorization

X<-c(1,2,3)

Y<-c(4,5,6)

Z<-X+Y

Z

+ is a vector operation in R, and it will operate on the entire vectors at once

Recycling

- When performing an operation on two or more vectors of unequal length, R will recycle elements of the shorter vector(s) to match the longest vector.

Example

long <- 1: 10

short <- 1: 5

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

short ## [1] 1 2 3 4 5

long + short ## [1] 2 4 6 8 10 7 9 11 13 15

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- **Getting help**
- Workspace
- Working with packages
- Working with different types of data in R
  - Dealing with characters
  - Dealing with factors
  - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Getting help

- To get the supporting material on the function funtionname which is already loaded

Help(functionname) or

?functionname

- To get the supporting material on the function functionname which has been installed but not yet loaded

help(functionname, package = "packagename").

Agenda

- What is R and Why R
- Vectorization
    - Recycling
- Getting help
- **Workspace**
- Working with packages
- Working with different types of data in R
    - Dealing with characters
    - Dealing with factors
    - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Workspace

- To define the workspace use setwd() function
- To fetch the path of the workspace use getwd() function
- Note the forward slashes in the pathname of the setwd() command. R treats the backslash (\) as an escape character. Even when you are using R on a Windows platform, use forward slashes in pathnames.
- Example

Setwd(“d:/test”)

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- Getting help
- Workspace
- **Working with packages**
- Working with different types of data in R
  - Dealing with characters
  - Dealing with factors
  - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Working with packages

- Install package
  - Install.packages(“packagename”)
- Load package
  - library(packagename)
- Get help on the use of the package
  - E.g. help(package=“caret”)
- Check which library packages have been installed
  - library()
- Check packages already loaded
  - search()
- List vignettes available for a specified package
  - Vignette(package=“packagename”)
- View specific vignette
  - E.g. Vignette(“caret”)

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- Getting help
- Workspace
- Working with packages
- **Working with different types of data in R**
  - Dealing with characters
  - Dealing with factors
  - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Working with different types of data in R

- Integer
- Numeric
- Character
- logical

Check the type

- is.numeric
- is.character
- is.logical
- as.numeric
- as.character
- as.logical

To check if the data type is of type numeric,character or logical respectively

To convert the data type to the type numeric,character or logical respectively

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
  - **Dealing with characters**
  - Dealing with factors
  - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Dealing with character

- Create string
- Counting number of elements or characters
- Subset string
- Set operator for character strings
- Regular expression
- Split string

Create string-1/2

- Use Paste() function

Example 1

```
a <- "learning to create"  # create string a
b <- "character strings" # create string b
paste(a,b)
```

Example 2

```
paste(" I", "love", "R", sep = "-")
## [1] "I-love-R"
```

Create string-2/2

- # use paste0() to paste without spaces btwn characters

paste0(" I", "love", "R") ## [1] "IloveR"

paste objects with different lengths

paste(" R", 1: 5, sep = " v1.")

[1] "R v1.1" "R v1.2" "R v1.3" "R v1.4" "R v1.5"

Counting number of elements or characters

- To count the number of elements in a string use length():

```
length(" How many elements are in this string?")
## [1] 1
length( c(" How", "many", "elements", "are", "in", "this", "string?"))
## [1] 7
```

- To count the number of characters in a string use nchar():

```
nchar(" How many characters are in this string?")
## [1] 39
nchar( c("How", "many", "characters", "are", "in", "this", "string?"))
## [1] 3 4 10 3 2 4 7
```

Subset string

```
alphabet <- paste( LETTERS, collapse = "")
# extract 18th through last character
substring( alphabet, first = 18)
## [1] "RSTUVWXYZ“
 # recursive extraction; specify start position only
substring( alphabet, first = 18: 24)
## [1] "RSTUVWXYZ" "STUVWXYZ" "TUVWXYZ" "UVWXYZ" "VWXYZ" "WXYZ"
## [7] "XYZ"
```

Quiz: What is the output from the following statement ?

```
substring( alphabet, first = 1: 5, last = 3: 7)
```

Set operator for character strings-1/6

- Set Union
- Set Intersection
- Identifying different elements
- Test for element equality
- Test for exact equality
- Testing if elements are contained in a string
- Sorting a string

Set operator for character strings-2/6

```
set_1 <- c(" lagunitas", "bells", "dogfish", "summit", "odell")
set_2 <- c(" sierra", "bells", "harpoon", "lagunitas", "founders")
#Set Union
union( set_1, set_2)
#Set Intersect
intersect( set_1, set_2)
# returns elements in set_1 not in set_2
setdiff( set_1, set_2)
## [1] "dogfish" "summit" "odell"
# returns elements in set_2 not in set_1
setdiff( set_2, set_1)
## [1] "sierra" "harpoon" "founders"
```

Set operator for character strings-3/6

To test if two vectors contain the same elements regardless of order use setequal():

```
set_3 <- c(" woody", "buzz", "rex")
set_4 <- c(" woody", "andy", "buzz")
set_5 <- c(" andy", "buzz", "woody")
setequal( set_3, set_4)
## [1] FALSE
setequal( set_4, set_5) ## [1] TRUE
```

Set operator for character strings-4/6

To test if two character vectors are equal in content and order use identical():

```
set_6 <- c(" woody", "andy", "buzz")
set_7 <- c(" andy", "buzz", "woody")
set_8 <- c(" woody", "andy", "buzz")
identical( set_6, set_7)
## [1] FALSE
identical( set_6, set_8)
## [1] TRUE
```

Set operator for character strings-5/6

To test if an element is contained within a character vector use is.element() or %in%:

good <- "andy"

bad <- "sid"

is.element(good, set_8)

[1] TRUE

good %in% set_8

[1] TRUE

Quiz: what is the output of the following statement

Bad %in% set_8

Set operator for character strings-6/6

To sort a character vector use sort():

```
sort( set_8)
## [1] "andy" "buzz" "woody"
sort( set_8, decreasing = TRUE)
## [1] "woody" "buzz" "andy"
```

Regular expression-1/2

# use the built in data set ` state.division `
head(as.character(state.division))

[1] "East South Central" "Pacific" "Mountain" ## [4] "West South Central" "Pacific" "Mountain"

find the elements which match the pattern

grep(" North", state.division)

[1] 13 14 15 16 22 23 25 27 34 35 41 49

use 'value = TRUE' to show the element value

grep(" North", state.division, value = TRUE)

Regular expression-2/2

# can use the 'invert' argument to show the non-matching elements
grep(" North | South", state.division, invert = TRUE)

[1] 2 3 5 6 7 8 9 10 11 12 19 20 21 26 28 29 30 31 32 33 37 38 39 ## [24] 40 44 45 46 47 48 50

String splitting

To split the elements of a character string use str_split().

z <- "The day after I will take a break and drink a beer."

str_split(z, pattern = " ") ## [[1]] ## [1] "The" "day" "after" "I" "will" "take" "a" "break" ## [9] "and" "drink" "a" "beer."

a <- "Alabama-Alaska-Arizona-Arkansas-California"

str_split(a, pattern = "-")

[[1]] ## [1] "Alabama" "Alaska" "Arizona" "Arkansas" "California"

Note that the output of strs_plit() is a list. To convert the output to a simple atomic vector simply wrap in unlist():

unlist(str_split(a, pattern = "-"))

[1] "Alabama" "Alaska" "Arizona" "Arkansas" "California"

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
  - Dealing with characters
  - **Dealing with factors**
  - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Dealing with Factors

- Factors are variables in R which take on a limited number of different values; such variables are often referred to as categorical variables.
- One of the most important uses of factors is in statistical modeling; since categorical variables enter into statistical models such as lm and glm differently than continuous variables, storing data as factors insures that the modeling functions will treat such data correctly.
- One can think of a factor as an integer vector where each integer has a label. In fact, factors are built on top of integer vectors using two attributes: the class() “factor”, which makes them behave differently from regular integer vectors, and the levels(), which defines the set of allowed values.

Dealing with Factors

- Factor objects can be created with the factor() function:

create a factor string
gender <- factor(c(" male", "female", "female", "male", "female"))
gender
[1] male female female male female
Levels: female male # inspect to see if it is a factor class
class(gender)
[1] "factor"
show that factors are just built on top of integers
typeof(gender)
[1] "integer"

Dealing with Factors

- # See the underlying representation of factor

unclass(gender)
[1] 2 1 1 2 1
attr(," levels")
[1] "female" "male"
what are the factor levels?
levels(gender)
[1] "female" "male"
show summary of counts
summary(gender)
female male
3 2

Ordering levels

- When creating a factor we can control the ordering of the levels by using the levels argument:
- # when not specified the default puts order as alphabetical

gender <- factor(c(" male", "female", "female", "male", "female"))

gender

[1] male female female male female

Levels: female male

specifying order

gender <- factor(c(" male", "female", "female", "male", "female"), levels = c(" male", "female"))

gender

[1] male female female male female

Levels: male female

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
  - Dealing with characters
  - Dealing with factors
  - **Dealing with dates**
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Dealing with Dates

To convert a string that is already in a date format (YYYY-MM-DD) into a date object use as.Date():

```
x <- c(" 2015-07-01", "2015-08-01", "2015-09-01")
as.Date( x)
## [1] "2015-07-01" "2015-08-01" "2015-09-01"
```

Dealing with Dates

Note that the default date format is YYYY-MM-DD; therefore, if your string is of different format you must incorporate the format argument. There are multiple formats that dates can be in; for a complete list of formatting code options in R type ?strftime in your console.

```
y <- c(" 07/ 01/ 2015", "07/ 01/ 2015", "07/ 01/ 2015")
as.Date( y, format = "% m/% d/% Y")
## [1] "2015-07-01" "2015-07-01" "2015-07-01"
```

Dealing with Dates: Date format

Symbol	Meaning	Example
%d	Day as a number (0-31)	01-31
%a	Abbreviated weekday	Mon
%A	Unabbreviated weekday	Monday
%m	Month(00-12)	00-12
%b	Abbreviated month	Jan
%B	Unabbreviated month	January
%y	Two-digit year	07
%Y	Four-digit year	2007

In R the default format for date value is yyyy-mm-dd

Dealing with Dates: Example on date management in R

```
strDates <- c("01/05/1965", "08/16/1975")
dates <- as.Date(strDates, "%m/%d/%Y")

today <- Sys.Date()
format(today, format="%B %d %Y")
format(today, format="%A")
```

Dealing with Dates

To create a sequence of dates we can leverage the seq() function.

As with numeric vectors, you have to specify at least three of the four arguments (from, to, by, and length.out).

```
seq( as.Date(" 2010-1-1"), as.Date(" 2015-1-1"), by = "years")
```

```
seq( as.Date(' 2015-09-15'), as.Date(' 2015-09-30'), by = "2 days")
```

```
seq( as.POSIXct(" 2015-1-1 0: 00"), as.POSIXct(" 2015-1-1 12: 00"), by = "hour")
```

Using lubridate package to work with dates-1/3

```
x <- c(" 2015-07-01", "2015-08-01", "2015-09-01")
y <- c(" 07/ 01/ 2015", "07/ 01/ 2015", "07/ 01/ 2015")

library( lubridate)
ymd( x)
## [1] "2015-07-01 UTC" "2015-08-01 UTC" "2015-09-01 UTC"
mdy( y) ## [1] "2015-07-01 UTC" "2015-07-01 UTC" "2015-07-01 UTC"
```

Using lubridate package to work with dates-2/3

```
x <- c(" 2015-07-01", "2015-08-01", "2015-09-01")
year( x)
## [1] 2015 2015 2015 # default is numerical value
month( x)
 ## [1] 7 8 9
# show abbreviated name
month( x, label = TRUE)
 ## [1] Jul Aug Sep ## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
# show unabbreviated name
month( x, label = TRUE, abbr = FALSE)
## [1] July August September ## 12 Levels: January < February < March < April < May < June < ... < December
wday( x, label = TRUE, abbr = FALSE)
## [1] Wednesday Saturday Tuesday ## 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

Using lubridate package to work with dates-3/3

most recent daylight savings time

ds <- ymd_hms(" 2015-03-08 01: 59: 59", tz = "US/ Eastern")

if we add a duration of 1 sec we gain an extra hour

ds + dseconds(1)

[1] "2015-03-08 03: 00: 00 EDT"

Agenda

- What is R and Why R
- Vectorization
    - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
    - Dealing with characters
    - Dealing with factors
    - Dealing with dates
- **Working with different data structures in R**
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Data structure in R

- Vector
- List
- Matrix
- Data frames

Basic data management technique

- Checking the structure of the data
  - Use str() function
- Adding elements into the data structure
- Adding attributes into the data structure
  - To check the existing attributes use attributes() function
- Subset the data structure
- Preserving or simplifying the data structure

Adding attributes in vector

assigning names to a pre-existing vector

names(v1) <- letters[1: length(v1)]

V1

a b c d e f g h i j

8 9 10 11 12 13 14 15 16 17

attributes(v1)

$ names ## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j“

adding names when creating vectors

v2 <- c(name1 = 1, name2 = 2, name3 = 3)

v2

name1 name2 name3

1 2 3

attributes(v2) ## $ names ## [1] "name1" "name2" "name3"

Subsetting with vector

- The four main ways to subset a vector include combining square brackets [ ] with:
  - Positive integers
  - Negative integers
  - Logical values
  - Names

Subsetting with vector

```
v1
## a b c d e f g h i j
## 8 9 10 11 12 13 14 15 16 17
v1[ 2]
## b
## 9
v1[ 2: 4]
## b c d
## 9 10 11
v1[ c( 2, 4, 6, 8)]
## b d f h
## 9 11 13 15
# note that you can duplicate index positions
v1[ c( 2, 2, 4)]
## b b d
## 9 9 11
```

Subsetting with vector

v1[-1]

b c d e f g h i j

9 10 11 12 13 14 15 16 17

v1[-c(2, 4, 6, 8)]

a c e g i j

8 10 12 14 16 17

Creating List-Example

```
l <- list( 1: 3, "a", c( TRUE, FALSE, TRUE), c( 2.5, 4.2))
str( l)
## List of 4
## $ : int [1: 3] 1 2 3
## $ : chr "a“
 ## $ : logi [1: 3] TRUE FALSE TRUE
## $ : num [1: 2] 2.5 4.2
# a list containing a list
l <- list( 1: 3, list( letters[ 1: 5], c( TRUE, FALSE, TRUE)))
str( l)
## List of 2
 ## $ : int [1: 3] 1 2 3
## $ :List of 2
## .. $ : chr [1: 5] "a" "b" "c" "d" ...
 ## .. $ : logi [1: 3] TRUE FALSE TRUE
```

Adding elements into List-1/2

How to add the element vector into the existing list ?

```
l2 <- list( l1, c( 2.5, 4.2))
str( l2)
## List of 2
## $ :List of 3
## .. $ : int [1: 3] 1 2 3
## .. $ : chr "a"
## .. $ : logi [1: 3] TRUE FALSE TRUE
```

Correct ?

Adding elements into List-2/2

```
l3 <- append( l1, list( c( 2.5, 4.2)))
str( l3)
## List of 4
## $ : int [1: 3] 1 2 3
## $ : chr "a"
## $ : logi [1: 3] TRUE FALSE TRUE
## $ : num [1: 2] 2.5 4.2
```

Alternatively, we can also add a new list component by utilizing the ‘$’ sign and naming the new item:

```
l3 $ item4 <- "new list item"
str( l3)
## List of 5
## $ : int [1: 3] 1 2 3
 ## $ : chr "a" ## $ : logi [1: 3] TRUE FALSE TRUE ## $ : num [1: 2] 2.5 4.2 ## $ item4: chr "new list item"
```

Adding attributes to list-1/5

adding names to a pre-existing list

names(l1) <- c(" item1", "item2", "item3")

str(l1)

List of 3

$ item1: int [1: 6] 1 2 3 4 5 6

$ item2: chr [1: 4] "a" "dding" "to a" "list"

$ item3: logi [1: 3] TRUE FALSE TRUE attributes(l1)

$ names ## [1] "item1" "item2" "item3"

adding names when creating lists

l2 <- list(item1 = 1: 3, item2 = letters[1: 5], item3 = c(T, F, T, T))

str(l2)

List of 3

$ item1: int [1: 3] 1 2 3

$ item2: chr [1: 5] "a" "b" "c" "d" ...

$ item3: logi [1: 4] TRUE FALSE TRUE TRUE attributes(l2)

$ names ## [1] "item1" "item2" "item3"

To subset the list-2/5

- To subset lists we can utilize the single bracket [ ], double brackets [[ ]], and dollar sign $ operators.
- Each approach provides a specific purpose and can be combined in different ways to achieve the following subsetting objectives:
  - Subset list and preserve output as a list
  - Subset list and simplify output
  - Subset list to get elements out of a list
  - Subset list with a nested list

To subset the list-3/5

```
# extract first list item
l2[ 1]
## $ item1
## [1] 1 2 3
# same as above but using the item's name
l2[" item1"]
## $ item1
## [1] 1 2 3
# extract multiple list items l2[ c( 1,3)]
## $ item1
## [1] 1 2 3
## ## $ item3
## [1] TRUE FALSE TRUE TRUE
```

To subset the list-4/5

- To extract one or more list items while simplifying the output use the [[ ]] or $ operator:

```
# extract first list item and simplify to a vector
l2[[ 1]]
## [1] 1 2 3
# same as above but using the item's name
l2[[" item1"]]
## [1] 1 2 3
# same as above but using the ` $ ` operator
l2 $ item1
## [1] 1 2 3
```

To subset the list-5/5

- To extract individual elements out of a specific list item combine the [[(or $) operator with the [operator:

```
# extract third element from the second list item
l2[[ 2]][ 3]
## [1] "c"
# same as above but using the item's name
l2[[" item2"]][ 3]
## [1] "c"
# same as above but using the ` $ ` operator
l2 $ item2[ 3]
## [1] "c"
```

Dealing with Matrix->Create matrix

- # numeric matrix

```
m1 <- matrix( 1: 6, nrow = 2, ncol = 3)
m1
##        [, 1] [, 2] [, 3]
 ##[1,] 1       3     5
## [2,]  2      4     6
```

Adding row or column in matrix

v1 <- 1: 4

v2 <- 5: 8

cbind(v1, v2)

```
##       v1 v2
## [1,] 1 5
## [2,] 2 6
## [3,] 3 7
## [4,] 4 8
```

rbind(v1, v2)

```
##     [, 1] [, 2] [, 3] [, 4]
## v1 1 2 3 4
## v2 5 6 7 8
```

Adding attributes in matrix

```
# add column names
colnames( m2) <- c(" col1", "col2", "col3")
# add row names
rownames( m2) <- c(" row1", "row2", "row3", "row4")
```

To subset matrix-1/3

- To subset matrices we use the [operator;
- however, since matrices have 2 dimensions we need to incorporate subsetting arguments for both row and column dimensions.
- A generic form of matrix subsetting looks like: matrix[rows, columns].

To subset matrix-2/3

```
m2
##          col_1  col_2  col_3
## row_1    1      5      9
## row_2    2      6      10
## row_3    3      7      11
## row_4    4      8      12

# subset for rows 1 and 2 but keep all columns
m2[ 1: 2, ]
# subset for columns 1 and 3 but keep all rows
m2[ , c( 1, 3)]
# subset for both rows and columns
m2[ 1: 2, c( 1, 3)]
```

To subset matrix-3/3

- Note that subsetting matrices with the [operator will simplify the results to the lowest possible dimension.
- To avoid this you can introduce the drop = FALSE argument:

```
# simplifying results in a named vector
m2[, 2]
## row_1 row_2 row_3 row_4
## 5 6 7 8
# preserving results in a 4x1 matrix
m2[, 2, drop = FALSE]
## col_2
## row_1 5
## row_2 6
## row_3 7
## row_4 8
```

Create data frame-1/2

df <- data.frame(col1 = 1: 3, col2 = c(" this", "is", "text"), col3 = c(TRUE, FALSE, TRUE), col4 = c(2.5, 4.2, pi))

assess the structure of a data frame

str(df)

'data.frame': 3 obs. of 4 variables:

$ col1: int 1 2 3

$ col2: Factor w/ 3 levels "is"," text"," this": 3 1 2

$ col3: logi TRUE FALSE TRUE

$ col4: num 2.5 4.2 3.14

number of rows

nrow(df)

[1] 3

number of columns

ncol(df)

[1] 4

Create data frame-2/2

- Note how col2 in df was converted to a column of factors.
- This is because there is a default setting in data.frame() that converts character columns to factors.
- We can turn this off by setting the stringsAsFactors = FALSE argument:

```
df <- data.frame( col1 = 1: 3, col2 = c(" this", "is", "text"), col3 = c( TRUE, FALSE, TRUE), col4 = c( 2.5, 4.2, pi), stringsAsFactors = FALSE)
```

To subset data frame

- Data frames possess the characteristics of both lists and matrices:
- if you subset with a single vector, they behave like lists and will return the selected columns with all rows;
- if you subset with two vectors, they behave like matrices and can be subset by row and column:

Missing value-1/5

- A common task in data analysis is dealing with missing values.
- In R, missing values are often represented by NA or some other value that represents missing values (i.e. 99).

Missing value-2/5

```
# vector with missing data
x <- c( 1: 4, NA, 6: 7, NA)
x
## [1] 1 2 3 4 NA 6 7 NA
is.na( x)
## [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE
# data frame with missing data
df <- data.frame( col1 = c( 1: 3, NA), col2 = c(" this", NA," is", "text"), col3 = c( TRUE, FALSE, TRUE, TRUE), col4 = c( 2.5, 4.2, 3.2, NA), stringsAsFactors = FALSE)
 # identify NAs in full data frame
is.na( df)
## col1 col2 col3 col4
## [1,] FALSE FALSE FALSE FALSE
 ## [2,] FALSE TRUE FALSE FALSE
## [3,] FALSE FALSE FALSE FALSE
## [4,] TRUE FALSE FALSE TRUE
# identify NAs in specific data frame column
is.na( df $ col4)
## [1] FALSE FALSE FALSE TRUE
```

Missing value-3/5

- To identify the location or the number of NAs we can leverage the which() and sum() functions:

identify location of NAs in vector

which(is.na(x))

[1] 5 8

identify count of NAs in data frame

sum(is.na(df))

[1] 3

Missing value-4/5

- # recode missing values with the mean

x[is.na(x)] <- mean(x, na.rm = TRUE)

round(x, 2)

[1] 1.00 2.00 3.00 4.00 3.83 6.00 7.00 3.83

data frame that codes missing values as 99

df <- data.frame(col1 = c(1: 3, 99), col2 = c(2.5, 4.2, 99, 3.2)) # change 99s to NAs

df[df = = 99] <- NA

Missing value-5/5

- First, to find complete cases we can leverage the complete.cases() function which returns a logical vector identifying rows which are complete cases.

df[complete.cases(df),]

df[! complete.cases(df),]

na.omit() would be the shorthand alternative

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
  - Dealing with characters
  - Dealing with factors
  - Dealing with dates
- Working with different data structures in R
- **Different basic looping functions in R (apply(),lapply(),sapply(),tapply())**
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- Case study

Apply function

- The apply family consists of vectorized functions which minimize your need to explicitly create loops.
- These functions will apply a specified function to a data object and there primary difference is in the object class in which the function is applied to (list vs. matrix, etc) and the object class that will be returned from the function.
- apply() (Input:matrix or dataframe)
- lapply() (input:list Output:list)
- sapply()
- tapply()

Apply()

- The syntax for apply() is as follows

apply(x, MARGIN, FUN, ...)

  - where x is the matrix, dataframe
  - array MARGIN is a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns.
  - FUN is the function to be applied
  - ... is for any other arguments to be passed to the function

Apply()-Example

#To calculate the mean for every column of the dataset mtcars

```
apply( mtcars, 2, mean)
```

```
# get column quantiles (notice the quantile percents as row names)
apply( mtcars, 2, quantile, probs = c( 0.10, 0.25, 0.50, 0.75, 0.90))
```

lapply()

- The lapply() function does the following simple series of operations:
- it loops over a list, iterating over each element in that list it applies a function to each element of the list (a function that you specify) and returns a list (the l is for “list”).
- The syntax for lapply() is as follows
  - where x is the list FUN is the function to be applied
  - ... is for any other arguments to be passed to the function
- syntax of lapply function

lapply(x, FUN, ...)

lapply-Example

```
data <- list( item1 = 1: 4, item2 = rnorm( 10), item3 = rnorm( 20, 1),
item4 = rnorm( 100, 5))

# get the mean of each list item

lapply( data, mean)
```

lapply-Example

\# list of R's built in beaver data

beaver_data <- list(beaver1 = beaver1, beaver2 = beaver2)

\# get the mean of each list item

lapply(beaver_data, function(x) round(apply(x, 2, mean), 2))

\## $ beaver1

\## day time temp activ

\## 346.20 1312.02 36.86 0.05

\## ## $ beaver2

\## day time temp activ

\## 307.13 1446.20 37.60 0.62

sapply()

- The sapply() function behaves similarly to lapply();
- the only real difference is in the return value.
- sapply() will try to simplify the result of lapply() if possible.
- Essentially, sapply() calls lapply() on its input and then applies the following algorithm:
- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If neither of the above simplifications can be performed then a list is returned

tapply()

- tapply() is used to apply a function over subsets of a vector.
- It is primarily used when we have the following circumstances:
- A dataset that can be broken up into groups (via categorical variables - aka factors)
- We desire to break the dataset up into groups Within each group, we want to apply a function

tapply()

- The arguments to tapply() are as follows:
- x is a vector
- INDEX is a factor or a list of factors (or else they are coerced to factors) FUN is a function to be applied
- ... contains other arguments to be passed FUN
- simplify, should we simplify the result?
- syntax of tapply function

tapply(x, INDEX, FUN, ..., simplify = TRUE)

tapply()-Example

- tapply(mtcars $ mpg, mtcars $ cyl, mean)

- apply(mtcars, 2, function(x) tapply(x, mtcars $ cyl, mean))

Agenda

- What is R and Why R
- Vectorization
    - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
    - Dealing with characters
    - Dealing with factors
    - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- **Importing and exporting data from/into other formats (e.g. csv and excel)**
- Other mathematical functions
- Case study

Importing data from a delimited text file-1/2

First, save the following 4 lines in a file named

"studentgrades.csv" in the current working directory

```
StudentID,First,Last,Math,Science,Social Studies
011,Bob,Smith,90,80,67
012,Jane,Weary,75,,80
010,Dan,"Thornton, III",65,75,70
040,Mary,"O'Leary",90,95,92
```

Next, read the data into R using the read.table() function

```
grades <- read.table("studentgrades.csv", header=TRUE,
              row.names="StudentID", sep=",")
grades # print data frame
str(grades) # view data frame structure
```

Importing data from a delimited text file-2/2

```
# Alternatively, import the data while specifying column classes
    grades <- read.table("studentgrades.csv", header=TRUE,
                  row.names="StudentID", sep=",",
                  colClasses=c("character", "character", "character",
                         "numeric", "numeric", "numeric"))
    grades # print data frame
    str(grades) # view data frame structure
```

Importing/exporting data from/to Excel

```
library(xlsx)
workbook <- “C:/myworkbook.xlsx”
mydataframe <- read.xlsx(workbook,1)
write.xlsx(workbook,”test.xlsx”)
```

Agenda

- What is R and Why R
- Vectorization
  - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
  - Dealing with characters
  - Dealing with factors
  - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- **Other mathematical functions**
- Case study

Mathematical functions in R
Common mathematical functions used

Function	Description
abs(x)	Absolute values abs(-4) is 4
signif(x,digits=n)	Rounds x to the specified number of decimal digits signif(3.475,digits=2) returns 3.5
mean(x)	The mean value of a vector Mean(c(1,2,3,4)) returns 2.5
median(x)	The median value of a vector median(c(1,2,3,4)) returns 2.5
Sd(x)	The standard deviation of a vector Sd(c(1,2,3,4)) returns 1.29
Quantile(x,probs)	Quantiles where x is the numeric vector, where quantiles are desired and probs is a numeric vector with probabilities in [0,1] #75th and 25th percentiles of x Y <- quantile(x, c(0.3,0.84))
Range(x)	The range of the numeric values within the numeric vector X <- c(1,2,3,4) Range(x) returns c(1,4) Diff(range(x)) returns 3
Scale(x,center=TRUE,scale=TRUE)	Column center (center=TRUE) or standardize (center=TRUE,scale=TRUE) data object x

Agenda

- What is R and Why R
- Vectorization
    - Recycling
- Getting help
- Workspace
- Working with packages
- Working with different types of data in R
    - Dealing with characters
    - Dealing with factors
    - Dealing with dates
- Working with different data structures in R
- Different basic looping functions in R (apply(),lapply(),sapply(),tapply())
- Importing and exporting data from/into other formats (e.g. csv and excel)
- Other mathematical functions
- **Case study**

Case study

- In this example you are studying how men and women differ in the ways they lead their organizations.
- Typical questions might be:
  - Do men and women in management positions differ in the degree to which they defer to superiors ?
  - Does this vary from country to country, or are these gender differences universal
- One way to address these questions is to have bosses in multiple countries rate their managers on deferential behavior

Sample dataset used in this example

Manager	Date	Country	Gender	Age	q1	q2	q3	q4	q5
1	10/24/14	US	M	32	5	4	5	5	5
2	10/28/14	US	F	45	3	5	2	5	5
3	10/01/14	UK	F	25	3	5	5	5	2
4	10/12/14	UK	M	39	3	3	4		
5	05/01/14	UK	F	99	2	2	1	2	1

Requirements on the data processing

- The five ratings (q1 to q5) need to be combined, yielding a single mean deferential score from each manager
- In surveys, respondents often skip questions. For example, the boss rating manager 4 skipped questions 4 and 5. You need a method of handling incomplete data. You also need to recode values like 99 for age to missing
- There may be hundreds of variables in a dataset, but you may only be interested in a few. To simplify matters you will want to create a new dataset with only the variables of interest
- Past research suggests that leadership behavior may change as a function of the managers' age. To examine this you may want to recode the current values of ages into a new categorical age grouping (for example,young, middle-aged,elder)
- Leadership behavior may change over time. You might want to focus on deferential behavior during the recent global financial crisis. To do so, you may want to focus on deferential behavior during the recent global financial crisis. You may want to limit the study to data gathered during a specific period of time (say, January 1,2009 to December 31,2009)

Create the dataframe

```
manager <- c(1,2,3,4,5)
date <- c("10/24/08","10/28/08","10/1/08","10/12/08","5/1/09")
gender <- c("M","F","F","M","F")
age <- c(32,45,25,39,99)
q1 <- c(5,3,3,3,2)
q2 <- c(4,5,5,3,2)
q3 <- c(5,2,5,4,1)
q4 <- c(5,5,5,NA,2)
q5 <- c(5,5,2,NA,1)
leadership <- data.frame(manager,date,gender,age,q1,q2,q3,q4,q5,
                         stringsAsFactors=FALSE)
```

Creating new attributes
Three methods are available

Example

```
mydata<-data.frame(x1 = c(2, 2, 6, 4),
                   x2 = c(3, 4, 2, 8))
```

Method 1

```
mydata$sumx <- mydata$x1 + mydata$x2
mydata$meanx <- (mydata$x1 + mydata$x2)/2
```

Method 2

```
attach(mydata)
mydata$sumx <- x1 + x2
mydata$meanx <- (x1 + x2)/2
detach(mydata)
```

Missing values

- In R, missing values are represented by the symbol NA

1. **Checking if it is missing**

- Use is.na() function to check if it is NA value

2. **Recoding values into missing**

Example:

```
leadership$age[age==99] <-NA
```

3. **Excluding missing values from analysis**

- Using “na.rm=TRUE”

Example:

```
x <- c(1, 2, NA, 3)
y <- sum(x, na.rm=TRUE)
```

Example: Create new categorical attribute

```
leadership <- within(leadership,{
  agecat <- NA
  agecat[age > 75] <- "Elder"
  agecat[age >= 55 & age <= 75] <- "Middle Aged"
  agecat[age < 55] <- "Young" })
```