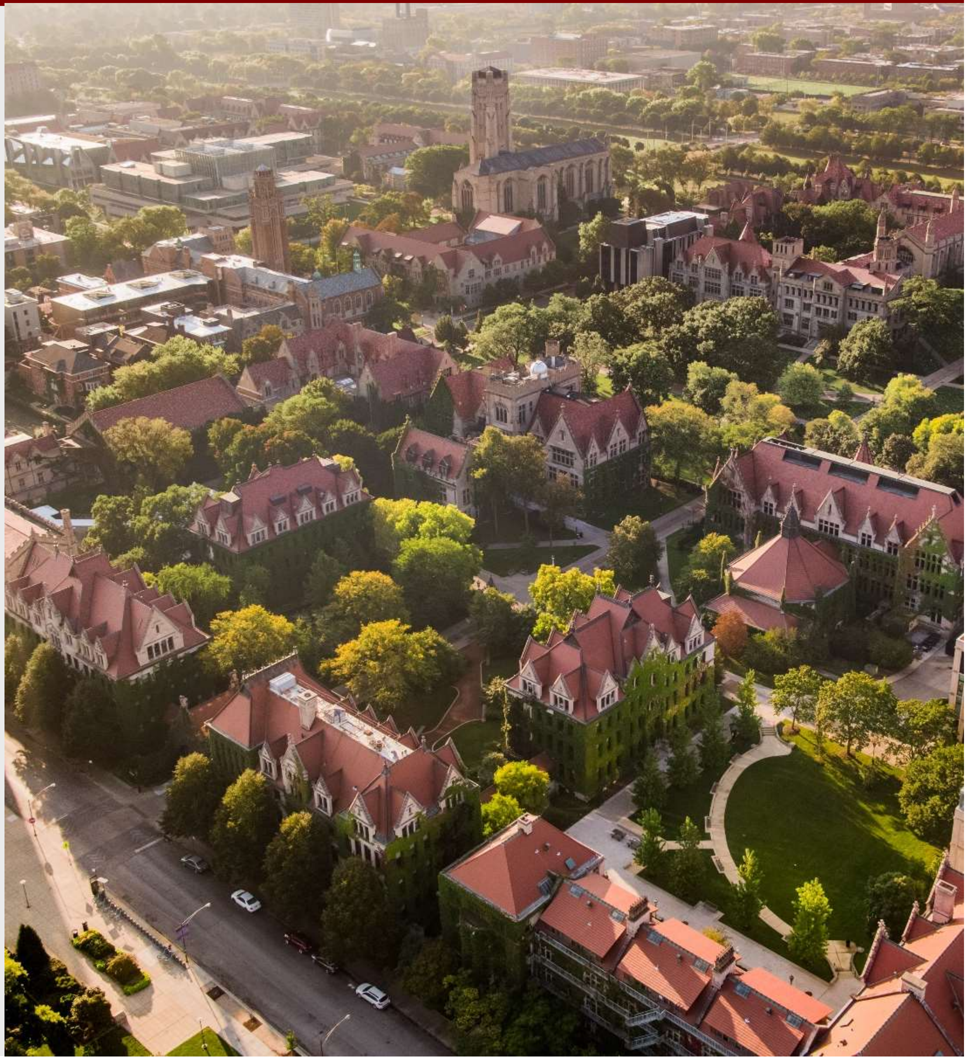
University of Chicago Professional Education

MSCA 37010 Programming for Analytics
Week 3 Lecture Notes

Autumn 2020



❑ Introduction

Instructor: Mei Najim
Email: mnajim@uchicago.edu
Tel: 847-800-9979 (C)
Class Meeting Time: 6:00 - 9:00pm, Mondays (01 Section)
1:30 - 4:30pm Saturdays (02 Section)
Tentative Office Hour: Mondays (9:00pm – 10:00pm or until last students)
Wednesdays (5:30pm – 6:30pm or until last students)
Saturdays (4:30pm – 5:30pm or until last students)

Notes:
1) First ten-minute quiz; Breakout groups in zoom; Two 10-minute breaks
2) Set up a weekly discussion group on canvas, allow 24 hours to respond
3) Email questions with the **Section Number (01/02)** in the subject line
4) If it is urgent, feel free to text me directly (847-800-9979)

Week 3 Class Agenda

- Data Preparation
  - Subsetting
  - Merging
  - Reviewing Data Types
  - Sorting
  - Duplicates
  - Missing values
  - Dplyr library

Week 3 Class Agenda

- Data Preparation
  - Subsetting
  - Merging
  - Reviewing Data Types
  - Sorting
  - Duplicates
  - Missing values
  - Dplyr library

Data Preparation – Subsetting

- Subsetting vectors, matrices, arrays, or data frame can also be referred to as indexing

  - 1) Subsetting Vectors

  - 2) Subsetting Data Frames

  - 3) Subsetting Operators in R

❑ Data Preparation – Subsetting Vectors

- Subsetting Vectors:
  - ✓ Subsetting the elements of a vector can be achieved by inserting an index vector in **square brackets** to the name of the vector
  - ✓ A logical vector:

  - The index vector should be the same length as the vector from which the elements are to be selected

  - Values corresponding to TRUE in index vector are being selected / to FALSE are being excluded

  - Values corresponding to NA returns NA

Data Preparation – Subsetting Vectors

- Subsetting Vectors:

Examples:

```
> a <- c(1, 3, 5, NA, 7)
> is.na(a)       [1] FALSE FALSE FALSE TRUE FALSE
> !is.na(a)      [1] TRUE TRUE TRUE FALSE TRUE
> a[!is.na(a)]   [1] 1 3 5 7
> a > 3          [1] FALSE FALSE TRUE NA TRUE
> a [a > 3]      [1] 5 NA 7
```

Data Preparation – Subsetting Data Frames

- Subsetting Data Frames

  - Data frame shares the common properties of matrices and lists. If you subset with two vectors at both positions, it behaves like subsetting a matrix

```
> d <- data.frame(L = c("A", "B", "C"), M = 1:3, N = c(T, F, NA))
> d[c(2,3), ]
  L M     N
2 B 2 FALSE
3 C 3    NA
> d[,c(1,3)]
  L     N
1 A  TRUE
2 B FALSE
3 C    NA
> d[d$L == "A", ]
  L M    N
1 A 1 TRUE
> d[, c("M", "N")]
  M     N
1 1  TRUE
2 2 FALSE
3 3    NA
> d[d$L == "A", c("M", "N")]
  M    N
1 1 TRUE
```

Data Preparation – Subsetting Operators

- Subsetting Operators: Three operators, “[ ]”, “[[ ]]”, and “$”, can be used to subset objects. Deciding which operator to use depends upon the object type. The main behavior difference across these three objects are:

- The type of object returned by using the “[“ operator is the same data type as the object “[“ applies to.

  Example: using “[“ to subset a list returns a list

- You can use “[“ to extract any numbers of elements of an object, while you can only use “[[“ and “$” to extract one element

- “$” does not evaluate its argument, while “[[“ and “[“ do. Thus, you can include an expression inside “[[“ or “[“

- “$” uses partial matching to extract elements, while “[[“ and “[“ do not

❑ Data Preparation – Subsetting Operators

- Examples

```
> alist <- list(name1 = c("john", "ken"), station = "AM640", time = "M-F: 3:00pm")
> alist[c(1,2)]
  $`name1` [1] "john" "ken"
  $station [1] "AM640
```

- The code above extracts the first two elements from a list. The resulting object is also a list. You can not write alist[[c(1,2)]] since you can only extract one element by using the [[operator

```
> alist["name1"]      $`name1` [1] "john" "ken"
> alist[["name1"]]    [1] "john" "ken"
> alist$name1         [1] "john" "ken"
```

Data Preparation – Subsetting Data Frames

- Simplifying : using the index vector with "[" operator at the second index position or using the "[[" operator when selecting a column
- Preserving : using the index vector the "[" operator at the second index position and setting the drop option to FALSE or using the "[" operator when selecting a column

```
> dat <- data.frame(V1=1:3, V2=c("a", "b", "c"), V3=c(T, T, F)); dat
  V1 V2    V3
1  1  a  TRUE
2  2  b  TRUE
3  3  c FALSE
> dat[, 2]
[1] a b c
Levels: a b c
> dat[[2]]
[1] a b c
Levels: a b c
> dat[, 2, drop = F]
  V2
1  a
2  b
3  c
> dat[2]
  V2
1  a
2  b
3  c
```

Data Preparation – Subsetting Data Frames

- Subsetting Data Frames by Using Index Vectors

We will use the data set, painters, from the MASS library to illustrate some examples

```
> library(MASS)
> head(painters)
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	A
Guilio Romano	15	16	4	14	A

❑ Data Preparation – Subsetting Data Frames

- Subsetting Data Frames by Using Index Vectors

Selecting the observations from a data frame is similar to selecting rows from a matrix by placing an index vector on the left side of the comma (,) inside the []

Example: To select the observations of painters with Colour greater than or equals 17:

```
> painters[painters$Colour>=17,]
```

	Composition	Drawing	Colour	Expression	School
Bassano	6	8	17	0	D
Giorgione	8	9	18	4	D
Pordenone	8	14	17	5	D
Titian	12	15	18	6	D
Rembrandt	15	6	17	12	G

Data Preparation – Subsetting Data Frames

- Subsetting Data Frames by Using Index Vectors

  Example: To select those from school A and D, you may want to use School == c('A', 'D’). This tests

```
> painters[painters$School %in% c('A', 'D'),]
```

	Composition	Drawing	Colour	Expression	School
Da Udine	10	8	16	3	A
Da Vinci	15	16	4	14	A
Del Piombo	8	13	16	7	A
Del Sarto	12	16	9	8	A
Fr. Penni	0	15	8	0	D

Data Preparation – Subsetting Data Frames

- Subsetting Data Frames by Using Index Vectors

  Selecting variables from a data frame is also similar to selecting columns from a matrix by placing an index vector on the right side of the comma (,) inside the []

  Example: To create a data set that contain the Colour (the third column) and School variables (the fifth column), you can write either one of the following statements

```
> d1 <- painters[, c('School', 'Colour’)]
> d2 <- painters[, c(5,3)]
> head(d2)
```

	School	Colour
Da Udine	A	16
Da Vinci	A	4
Del Piombo	A	16

❑ Data Preparation – Subsetting Data Frames

- Subsetting Data Frames by Using Index Vectors

  You can also select observations and variables at the same time by including two index vectors inside

  > d5<-painters[painters$School == "A", c('School', 'Colour')]; d5

	School	Colour
Da Udine	A	16
Da Vinci	A	4
Del Piombo	A	16
Del Sarto	A	9
Fr. Penni	A	8
Guilio Romano	A	4
Michelangelo	A	4

❑ Data Preparation – Subsetting Data Frames Example

- Subsetting Data Frames by Using ***subset()*** function

Examples: Use AutoCollision data

- Subsetting ClaimServerity>=300 **and** ClaimCount>=10

```
> subset(AutoColl, ClaimSeverity >=200 & ClaimCount>=10);
```

- Subsetting ClaimServerity>=300 **or** ClaimCount>=10

```
> subset(AutoColl, ClaimSeverity >=200 | ClaimCount>=10);
```

- Subsetting ClaimSeverity >=300 without AgeGroup column

```
> AutoColl[AutoColl$ClaimSeverity>=300, c('ClaimSeverity','VehicleUse’)]
> subset(AutoColl, ClaimSeverity >=300, select= - AgeGroup)
> subset.data.frame(AutoColl, ClaimSeverity >=300, select= - AgeGroup)
```

Week 3 Class Agenda

- Data Preparation
  - Subsetting
  - Merging
  - Reviewing Data Types
  - Sorting
  - Duplicates
  - Missing values
  - Dplyr library

Data Preparation – Merging

- Merging: Merge two datasets by an ID variable, where ID is the same for both datasets

```
> # Merge two datasets by an ID variable, where ID is the same for both datasets
> data1 <- data.frame(ID=1:5, x=letters[1:5]);data1
  ID x
1  1 a
2  2 b
3  3 c
4  4 d
5  5 e
> data2 <- data.frame(ID=1:5, y=letters[6:10]);data2
  ID y
1  1 f
2  2 g
3  3 h
4  4 i
5  5 j
> data3<-merge(data1, data2);data3
  ID x y
1  1 a f
2  2 b g
3  3 c h
4  4 d i
5  5 e j
```

Data Preparation – Merging

- Merging: Merge two datasets by an ID variable, where ID is not the same for both datasets

```
> data1 <- data.frame(ID=1:5, x=letters[1:5])
> data2 <- data.frame(ID=4:8, y=letters[6:10])
> merge(data1, data2)
  ID x y
1  4 d f
2  5 e g
> merge(data1, data2, all=TRUE)
  ID    x    y
1  1    a <NA>
2  2    b <NA>
3  3    c <NA>
4  4    d    f
5  5    e    g
6  6 <NA>    h
7  7 <NA>    i
8  8 <NA>    j
> merge(data1, data2, all.x=TRUE) # Only keep the rows from the 1st argument data1
  ID x    y
1  1 a <NA>
2  2 b <NA>
3  3 c <NA>
4  4 d    f
5  5 e    g
> merge(data1, data2, all.y=TRUE) # Only keep the rows from the 2nd argument data2
  ID    x y
1  4    d f
2  5    e g
3  6 <NA> h
4  7 <NA> i
5  8 <NA> j
```

Data Preparation – Merging

- Merging: Merge two datasets by an ID variable, where both dataset have the same names

```
> data1 <- data.frame(ID=1:5, x=letters[1:5])
> data2 <- data.frame(ID=1:5, x=letters[6:10])
> merge(data1, data2, all=TRUE)   # Add rows
   ID x
1   1 a
2   1 f
3   2 b
4   2 g
5   3 c
6   3 h
7   4 d
8   4 i
9   5 e
10  5 j
> merge(data1, data2, by="ID")  # Add columns
  ID x.x x.y
1 1   a   f
2 2   b   g
3 3   c   h
4 4   d   i
5 5   e   j
> merge(data1, data2, by="ID", suffixes=c(1, 2))
  ID x1 x2
1 1  a  f
2 2  b  g
3 3  c  h
4 4  d  i
5 5  e  j
```

Data Preparation – Merging

- Merging: Merge two datasets by an ID variable, where the ID variable has a different name

```
> data1 <- data.frame(ID1=1:5, x=letters[1:5]);data1
  ID1 x
1   1 a
2   2 b
3   3 c
4   4 d
5   5 e
> data2 <- data.frame(ID2=1:5, x=letters[6:10]);data2
  ID2 x
1   1 f
2   2 g
3   3 h
4   4 i
5   5 j
> merge(data1, data2, by.x="ID1", by.y="ID2")
  ID1 x.x x.y
1   1   a   f
2   2   b   g
3   3   c   h
4   4   d   i
5   5   e   j
```

Week 3 Class Agenda

- Data Preparation
  - Subsetting
  - Merging
  - Reviewing Data Types
  - Sorting
  - Duplicates
  - Missing values
  - Dplyr library

❑ Data Preparation – Reviewing Data Types: Numeric

- R Technically, numeric data in R can be either double or integer, but in practice numeric data is almost always double (type double refers to real numbers). See ?integer and ?double
- ***format***() formats an object for pretty printing. format() is a generic function that is used with other types of objects. See ?format() for additional arguments.

```
# trim - If FALSE right justified with common width

> format(c(1,10,100,1000), trim = FALSE)    [1] " 1" " 10" " 100" "1000"

> format(c(1,10,100,1000), trim = TRUE)     [1] "1" "10" "100" "1000"

# nsmall - Minimum number of digits to the right of the decimal point

  format(13.7, nsmall = 3)                  [1] "13.700"

# scientific - Use scientific notation

> format(2^16, scientific = TRUE)           [1] "6.5536e+04"
```

❑ Data Types – Numeric: Integer vs. Double

- The two most common numeric classes used in R are integer and double (for double precision floating point numbers). R automatically converts between these two classes when needed for mathematical purposes. As a result, it's feasible to use R and perform analyses for years without specifying these differences.
- Creating Integer and Double Vectors: By default, when you create a numeric vector using the c() function it will produce a vector of double precision numeric values. To create a vector of integers using c() you must specify explicitly by placing an L directly after each number.

```
# create a string of double-precision values
> dbl_var <- c(1, 2.5, 4.5) ; dbl_var        [1] 1.0 2.5 4.5
# placing an L after the values creates a string of integers
> int_var <- c(1L, 6L, 10L); int_var        [1]  1  6 10
```

❑ Data Types – Numeric: Integer vs. Double

- The two most common numeric classes used in R are integer and double (for double precision floating point Checking for Numeric Type

  To check whether a vector is made up of integer or double values:

  # identifies the vector type (double, integer, logical, or character)

  > typeof(dbl_var)   [1] "double"

  > typeof(int_var)   [1] "integer"

- Converting Between Integer and Double Values

  By default, if you read in data that has no decimal points or you create numeric values using the x <- 1:10 method the numeric values will be coded as integer. If you want to change a double to an integer or vice versa you can specify one of the following:

  # converts integers to double-precision values

  > as.double(int_var)   [1] 1 6 10   # identical to as.double()

  > as.numeric(int_var)   [1] 1 6 10   # converts doubles to integers

  > as.integer(dbl_var)   [1] 1 2 4

Data Types – Logical

- Logical values are represented by the reserved words **TRUE** and **FALSE** in all caps or simply **T** and **F**

`!x`	NOT x
`x & y`	x AND y elementwise, returns a vector
`x && y`	x AND y, returns a single value
`x | y`	x OR y elementwise, returns a vector
`x || y`	x OR y, returns a single value
`xor(x,y)`	Exclusive OR of x and y, elementwise
`x %in% y`	x IN y
`x < y`	$x < y$
`x > y`	$x > y$
`x <= y`	$x \leq y$
`x >= y`	$x \geq y$
`x == y`	$x = y$
`x != y`	$x \neq y$
`isTRUE(x)`	TRUE if x is TRUE
`all(...)`	TRUE if all arguments are TRUE
`any(...)`	TRUE if at least one argument is TRUE
`identical(x,y)`	Safe and reliable way to test two objects for being *exactly* equal
`all.equal(x,y)`	Test if two objects are *nearly* equal

❑ Data Types – Example Logical Operations

Example:

```
> x <- 1:10;x
  [1] 1 2 3 4 5 6 7 8 9 10
> (x%%2==0) | (x > 5) # What elements of x are even or greater than 5?
  [1] FALSE TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE
> x[(x%%2==0) | (x > 5)]
  [1] 2 4 6 7 8 9 10
> y <- 5:15 # What elements of x are in y? > x %in% y
  [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> x[x%in% y]
  [1] 5 6 7 8 9 10
> any(x>5) # Are any elements of x greater then 5?
  [1] TRUE
> all(x>5) # Are all the elements of x greater then 5?
  [1] FALSE
```

❑ Data Types – Isn't That Equal?

- In general, logical operators may not produce a single value and may return an **NA** if an element is **NA** or **NaN**.
- If you must get a single **TRUE** or **FALSE**, such as with **if** expressions, you should NOT use **==** or **!=**. Unless you are absolutely sure that nothing unusual can happen, you should use the ***identical***() function instead.
- ***identical()*** only returns a single logical value, TRUE or FALSE, never NA

```
> name <- "Nick“;   > if(name=="Nick") TRUE else FALSE          [1] TRUE

# But what if name is never set to "Nick"?

> name <- NA;        > if(name=="Nick") TRUE else FALSE

  Error in if (name == "Nick") TRUE else FALSE : missing value where TRUE/FALSE needed

> if(identical(name, "Nick")) TRUE else FALSE                   [1] FALSE
```

Data Types – Isn't That Equal?

- With ***all.equal***() objects are treated as equal if the only difference is probably the result of inexact floating-point calculations. Returns ***TRUE*** if the mean relative difference is less than the specified tolerance.
- all.equal() either returns TRUE or a character string that describes the difference. Therefore, do not use all.equal() directly in if expressions, instead use with ***isTRUE***() or ***identical***().

```
> (x <- sqrt(2))
 [1] 1.414214
> x^2
 [1] 2
> x^2==2
 [1] FALSE
```

```
> all.equal(x^2, 2)
 [1] TRUE
> all.equal(x^2, 1)
 [1] "Mean relative difference: 0.5"
> isTRUE(all.equal(x^2, 1))
 [1] FALSE
```

Data Types – Character

- Character strings are defined by quotation marks, single ' ' or double " "

`cat()`	Concatenate objects and print to console (\n for newline)
`paste()`	Concatenate objects and return a string
`print()`	Print an object
`substr()`	Extract or replace substrings in a character vector
`strtrim()`	Trim character vectors to specified display widths
`strsplit()`	Split elements of a character vector according to a substring
`grep()`	Search for matches to a pattern within a character vector, returns a vector of the indices that matched
`grepl()`	Like `grep()`, but returns a logical vector
`agrep()`	Similar to `grep()`, but searches for approximate matches
`regexpr()`	Similar to `grep()`, but returns the position of the first instance of a pattern *within* a string
`gsub()`	Replace all occurrences of a pattern with a character vector
`sub()`	Like `gsub()`, but only replaces the first occurrence
`tolower(), toupper()`	Convert to all lower/upper case
`noquote()`	Print a character vector without quotations
`nchar()`	Number of characters
`letters, LETTERS`	Built-in vector of lower and upper case letters

Data Types – Example Character Functions

- Character strings are defined by quotation marks, single ' ' or double " "

```
> animals <- c("bird", "horse", "fish"); home <- c("tree", "barn", "lake")
> length(animals)              # Number of strings                    R Output: [1] 3
> nchar(animals)               # Number of characters in each string  R Output: [1] 4 5 4
> cat("Animals:", animals)     # Need \n to move cursor to a newline  R Output: Animals: bird horse fish
> cat(animals, home, "\n")     # Joins one vector after the other     R Output: bird horse fish tree barn lake
> paste(animals, collapse=" ") # Create one long string of animals    R Output: [1] "bird horse fish"
> substr(animals, 2, 4)        # Get characters 2-4 of each animal    R Output: [1] "ird" "ors" "ish"
> strtrim(animals, 3)          # Print the first three characters     R Output: [1] "bir" "hor" "fis"
> toupper(animals)             # Print animals in all uppercase       R Output: [1] "BIRD" "HORSE" "FISH"
```

Data Types – Factors

- A factor is a categorical variable with a defined number of ordered/unordered levels. Use the function factor to create a factor variable.

```
> factor(rep(1:2, 4), labels=c("trt.1", "trt.2"))
[1] trt.1 trt.2 trt.1 trt.2 trt.1 trt.2 trt.1 trt.2
Levels: trt.1 trt.2

> factor(rep(1:3, 3), labels=c("low", "med", "high"), ordered=TRUE)
[1] low med high low med high low med high
Levels: low < med < high
```

`levels(x)`	Retrieve or set the levels of `x`
`nlevels(x)`	Return the number of levels of `x`
`relevel(x, ref)`	Levels of `x` are reordered so that the level specified by `ref` is first
`reorder()`	Reorders levels based on the values of a second variable
`gl()`	Generate factors by specifying the pattern of their levels
`cut(x, breaks)`	Divides the range of `x` into intervals (factors) determined by `breaks`

Data Types – Dates and Times

- R has objects that are dates only and objects that are dates and times. We will just focus on dates. Look at ?DateTimeClasses for information about how to handles dates and times.
- An R date object has the format: Year-Month-Day
- Operations with dates:

  - Days can be added or subtracted to a date

  - Dates can be subtracted

  - Dates can be compared using logical operators

`Sys.Date()`	Current date
`as.Date()`	Convert a character string to a date object
`format.Date()`	Change the format of a date object
`seq.Date()`	Generate sequence of dates
`cut.Date()`	Cut dates into intervals
`weekdays, months, quarters`	Extract parts of a date object
`julian`	Number of days since a given origin

.Date suffix is optional for calling `format.Date()`, `seq.Date()` and `cut.Date()`, but is necessary for viewing the appropriate documentation

Data Types – Dates and Times

- Converting a string to a date object requires specifying a format string that defines the date format
- Any character in the format string other then the % symbol is interpreted literally.
- Common conversion specifications (see ?strptime for a complete list),

- **%a** Abbreviated weekday name
- **%d** Day of the month
- **%b** Abbreviated month name
- **%y** Year without century (2 digits)
- **%A** Full weekday name
- **%B** Full month name
- **%m** Numeric month (01-12)
- **%Y** Year with century (4 digits)

Example:

```
> dates.1 <- c("5jan2008", "19aug2008", "2feb2009", "29sep2009")
> as.Date(dates.1, format="%d%b%Y")          [1] "2008-01-05" "2008-08-19" "2009-02-02" "2009-09-29"
> dates.2 <- c("5-1-2008", "19-8-2008", "2-2-2009", "29-9-2009")
> as.Date(dates.2, format="%d-%m-%Y")        [1] "2008-01-05" "2008-08-19" "2009-02-02" "2009-09-29"
```

Data Types – Sequence of Dates

- To create a sequence of dates, **seq.Date(from**, **to**, **by**, **length.out** = *NULL***)**

  - **from, to** Start and ending date objects
  - **by** A character string, containing one of "day", "week", "month" or "year". Can optionally be preceded by a (positive or negative) integer and a space, or followed by a "s".
  - **length.out** Integer, desired length of the sequence

Example:

```
> seq.Date(as.Date("2011/1/1"), as.Date("2011/1/31"), by="week")
[1] "2011-01-01" "2011-01-08" "2011-01-15" "2011-01-22" "2011-01-29"
> seq.Date(as.Date("2011/1/1"), as.Date("2011/1/31"), by="3 days")
[1] "2011-01-01" "2011-01-04" "2011-01-07" "2011-01-10" "2011-01-13" "2011-01-16" "2011-01-19"
[8] "2011-01-22" "2011-01-25" "2011-01-28" "2011-01-31"
> seq.Date(as.Date("2011/1/1"), by="week", length.out=10)
[1] "2011-01-01" "2011-01-08" "2011-01-15" "2011-01-22" "2011-01-29" "2011-02-05" "2011-02-12"
[8] "2011-02-19" "2011-02-26" "2011-03-05"
```

❑ Data Types – Cutting Dates

- To divide a sequence of dates into levels, cut.Date(x, breaks, start.on.monday = TRUE)

```
> jan <- seq.Date(as.Date("2011/1/1"), as.Date("2011/1/31"), by="days")
> cut(jan, breaks="weeks", start.on.monday=TRUE)
 [1] 2010-12-27 2010-12-27 2011-01-03 2011-01-03 2011-01-03 2011-01-03 2011-01-03 2011-01-03 2011-01-03 2011-01-10
[11] 2011-01-10 2011-01-10 2011-01-10 2011-01-10 2011-01-10 2011-01-10 2011-01-17 2011-01-17 2011-01-17 2011-01-17
[21] 2011-01-17 2011-01-17 2011-01-17 2011-01-24 2011-01-24 2011-01-24 2011-01-24 2011-01-24 2011-01-24 2011-01-24
[31] 2011-01-31
Levels: 2010-12-27 2011-01-03 2011-01-10 2011-01-17 2011-01-24 2011-01-31
```

January 2011

Sun	Mon	Tue	Wed	Thr	Fri	Sat
26	27	28	29	30	31	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	27	28	29
30	31	1	2	3	4	5

Data Types – Operations with Dates

- Operations with dates:

- Days can be added or subtracted to a date

- Dates can be subtracted

- Dates can be compared using logical operators

Example:

```
> jan1 <- as.Date("2011/1/1")
> (jan8 <- jan1 + 7)          # Add 7 days to 2011/1/1          [1] "2011-01-08"
> jan1 - 14                   # Subtract 2 weeks from 2011/1/8   [1] "2010-12-18"
> jan8 - jan1                 # Number of days between 2011/1/1 and 2011/1/8 Time difference of 7 days
> jan8 > jan1                 # Compare dates                    [1] TRUE
# Use format to extract parts of a date object or change the appearance
> format.Date(jan8, "%Y")       [1] "2011"
> format.Date(jan8, "%b-%d")    [1] "Jan-08"
```

❑ Data Types – Testing and Coercing Objects

- All objects in R have a type. We can test the type of an object using a ***is.type***() function.
- We can also attempt to **coerce** objects of one type to another using a ***as.type***() function.
- Automatic conversions:

  - Logical values are converted to numbers by setting **FALSE** as **0** and **TRUE** as **1**

  - Logical, numeric, factor and date types are converted to characters by converting each element/level individually
- Some general rules for coercion:

  - Numeric values are coerced to logical by treating all **non-zero values** as **TRUE**

  - Numeric characters can be coerced to numbers, but non-numeric characters cannot

  - Factor levels can be coerced to numeric and numbers can be coerced to factors with a level for each unique number

  - Vectors, matrices and arrays are coerced to lists by making each element a vector of length 1

  - Vectors, matrices, arrays can also be coerced from one form to another

❑ Data Types – Testing and Coercing Functions

Type	Testing	Coercing
Array	is.array()	as.array()
Character	is.character()	as.character()
Dataframe	is.data.frame()	as.data.frame()
Factor	is.factor()	as.factor()
List	is.list()	as.list()
Logical	is.logical()	as.logical()
Matrix	is.matrix()	as.matrix()
Numeric	is.numeric()	as.numeric()
Vector	is.vector()	as.vector()

❑ Data Types – Testing and Coercing Objects

Example:

```
> x <- 1:4; > x>3
  [1] FALSE FALSE FALSE TRUE
> sum(x>3)        # Automatic conversion to numeric vector; note TRUE=1, FALSE=0
  [1] 1
> is.vector(x)
  [1] TRUE
> is.numeric(x)
  [1] TRUE
> as.list(x)
  [[1]]
  [1] 1
  [[2]]
  [1] 2
  …
  [[4]]
  [1] 4
> as.numeric("123")
  [1] 123
```

Data Types – Testing and Coercing Functions

Type	Testing	Coercing
Array	`is.array()`	`as.array()`
Character	`is.character()`	`as.character()`
Dataframe	`is.data.frame()`	`as.data.frame()`
Factor	`is.factor()`	`as.factor()`
List	`is.list()`	`as.list()`
Logical	`is.logical()`	`as.logical()`
Matrix	`is.matrix()`	`as.matrix()`
Numeric	`is.numeric()`	`as.numeric()`
Vector	`is.vector()`	`as.vector()`

Week 3 Class Agenda

- Data Preparation
  - Subsetting
  - Merging
  - Reviewing Data Types
  - Sorting
  - Duplicates
  - Missing values
  - Dplyr library

❑ Data Preparation – Sorting: sort() vs. order() vs. rank() Function

- Sorting: sort() vs. order() vs. rank() Function

  - sort() sorts the vector in an ascending order.

  - rank() ranks the numbers in the vector the smallest number receiving the rank 1

  - order() returns the indices of the vector in a sorted order

```
> # Build a tibble (a simple data frame)
> set.seed(12345)
> t <- tibble(
+   unit = LETTERS[1:3],
+   a = rnorm(3),
+   b = rnorm(3)
+ )
> class(t)
[1] "tbl_df"     "tbl"        "data.frame"
>
> # Create two new columns: rank_a and rank_b, which, as the names imply,
> # contain the rank (or order) of ea?h value in their corresponding columns.
> t$sort_a <- sort(t$a)
> t$sort_b <- sort(t$b)
> t$order_a <- order(t$a)
> t$order_b <- order(t$b)
> t$rank_a <- rank(t$a)
> t$rank_b <- rank(t$b)
> t
# A tibble: 3 x 9
  unit       a      b sort_a sort_b order_a order_b rank_a rank_b
  <chr>  <dbl>  <dbl>  <dbl>  <dbl>   <int>   <int>  <dbl>  <dbl>
1 A      0.586 -0.453 -0.109 -1.82        3       3      2      2
2 B      0.709  0.606  0.586 -0.453       1       1      3      3
3 C     -0.109 -1.82   0.709  0.606       2       2      1      1
```

Data Preparation – Sorting a Vector

- The ***sort()*** function takes one vector argument, either numeric or character, and returns a vector of sorted values; To sort in decreasing order: rev(sort(x))

Example: `> x <- c(1,2.3,2,3,4,8,12,43,-4,-1,NA); sort(x)`

```
[1] -4.0 -1.0 1.0 2.0 2.3 3.0 4.0 8.0 12.0 43.0
```

- The ***order()*** function sorts a vector, matrix or data frame: ***order***(x, decreasing = FALSE, na.last = NA, ...)

- x: vector

- decreasing: decrease or not

- na.last: if TRUE, NAs are put at last position, FALSE at first, if NA, remove them (default)

Example: `> order(x)`

```
[1] -4.0 -1.0  1.0  2.0  2.3  3.0  4.0  8.0 12.0 43.0
```

❑ Data Preparation – Sorting a Vector

- Example: Sort Vector Continue:

```
> x <- c(1,2.3,2,3,4,8,12,43,-4,-1,NA)
> order(x)
[1] -4.0 -1.0  1.0  2.0  2.3  3.0  4.0  8.0 12.0 43.0
> order(x, decreasing=TRUE)
[1] 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0
> order(x, decreasing=TRUE, na.last=TRUE)
[1] 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0   NA
> order(x, decreasing=TRUE, na.last=FALSE)
[1]   NA 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0
```

Data Preparation – Sorting a Dataframe

- Example: Sort a Dataframe:

```
> library(readr)
> setwd("C:/TeachingUChicago/Spring2020/Data")
> Autodata <- read_csv("AutoCollision.csv")
> str(Autodata)       # Gives the structure of data
> View(Autodata)    # View imported data
# Sort by ClaimSeverity, ascending
> Autodata<-Autodata[order(Autodata$ClaimSeverity),]
# Sort by AgeGroup, descending
> Autodata<-Autodata[order(Autodata$AgeGroup, decreasing=TRUE),]
```

Week 3 Class Agenda

- Data Preparation
  - Subsetting
  - Merging
  - Reviewing Data Types
  - Sorting
  - Duplicates
  - Missing values
  - Dplyr library

❑ Data Preparation – Duplicates

- **Duplicates:**

  The function ***unique()*** will return a dataframe with the duplicate rows or columns removed.

  NOTE: ***unique()*** only work for imported dataframes and doesn't work for dataframes created during an R session

  Example:

```
> Dupdata <- read_csv("Dupautocoll.csv")      # 35obs. of 4 variables
> NoDupdata<-unique(Dupdata)                  # Dataset with 3 duplicated rows removed: 32obs. of 4 variables
```

Week 3 Class Agenda

- Data Preparation
  - Subsetting
  - Merging
  - Reviewing Data Types
  - Sorting
  - Duplicates
  - Missing values
  - Dplyr library

Data Preparation – Missing Data

- R denotes data that is not available by ***NA***. Quantities that are not a number, such as 0/0, are denoted by **NaN**. In R NaN implies NA (NaN refers to unavailable numeric data and NA refers to any type of unavailable data)
- How a function handles missing data depends on the function. For example, ***mean*** only ignores *NAs* if the argument *na.rm*=TRUE, whereas which always ignores missing data.

```
> x <- c(4, 7, 2, 0, 1, NA); > mean(x)      [1] NA
> mean(x, na.rm=TRUE)                       [1] 2.8
> which(x>4)                                [1] 2
```

- Undefined or null objects are denoted in R by NULL. This is useful when we do not want to add row labels to a matrix.

Example: `> x <- matrix(1:6, ncol=2, dimnames=list(NULL, c("c.1", "c.2"))); x`

```
     c.1  c.2
[1,]  1    4
[2,]  2    5
[3,]  3    6
```

❑ Data Preparation – Detecting Missing Data

- To test for missing data avoid using identical() and never use ==.; Using identical() relies on unreliable internal computations and "==" will always evaluate to NA or NaN.
- Functions used for detecting missing data,

  - is.na(x) Tests for NA or NaN data in x

  - is.nan(x) Tests for NaN data in x

  - is.null(x) Tests if x is NULL

  Example:

> x <- c(4, 7, 2, 0, 1, NA) ; x==NA	[1] NA NA NA NA NA NA
> is.na(x)	[1] FALSE FALSE FALSE FALSE FALSE TRUE
> any(is.na(x))	[1] TRUE
> (y <- x/0)	[1] Inf Inf Inf NaN Inf NA
> is.nan(y)	[1] FALSE FALSE FALSE TRUE FALSE FALSE

Data Preparation – Missing Data

- Use ***na.omit***() to remove missing data from a dataset
- Use ***na.fail***() to signal an error if a dataset contains NA
- Use ***complete.cases***() returns a logical vector indicating which rows have no missing data

Example:

```
> data <- data.frame(x=c(1,2,3), y=c(5, NA, 8))
> na.omit(data) # Remove all rows with missing data
# Use na.fail to test if a dataset is complete
NeedCompleteData <- function(data) {
  na.fail(data) # Return an error message if missing data
  lm(y, x, data=data)
}
NeedCompleteData(data)
sum(complete.cases(data))     # Get the number of complete cases
sum(!complete.cases(data))    # Get the number of incomplete cases
```

Week 3 Class Agenda

- Data Preparation
  - Subsetting
  - Merging
  - Reviewing Data Types
  - Sorting
  - Duplicates
  - Missing values
  - Dplyr library

❑ Data Preparation – Dplyr Library

- For analytics, the following functions from dplyr are useful for analytics – Let us see details in the related R scripts

  - 1. filter() (and slice())

  - 2. arrange()

  - 3. select() (and rename())

  - 4. distinct()

  - 5. mutate() (and transmute())

  - 6. summarise()

  - 7. sample_n() and sample_frac()

Q & A

THE UNIVERSITY OF CHICAGO

Thank You

THE UNIVERSITY OF
CHICAGO

Contact Information:

Your comments and questions are valued and encouraged.

Mei Najim
E-mail: mnajim@uchicago.edu

LinkedIn: https://www.linkedin.com/in/meinajim/