University of Chicago Professional Education

MSCA 37010 01/02 Programming for Analytics
Week 2 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: Wednesdays (5:30pm – 7:00pm or until last students)
Saturdays (4:30pm – 6:00pm or until last students)
Sundays (1:00pm – 2:00pm or until last students)

Notes: 1) First ten-minute quiz; Two 10-minute breaks
2) Set up a weekly discussion group on canvas, breakout groups in zoom; allow 24 hours to respond
3) Email questions first with Section # - reply/set up a time/post in the weekly discussion group
4) If it is urgent, feel free to text me directly (847-800-9979)

Week 2 Class Agenda

- Data Structure - Vector
  (Vector Basics, Operations, Indexing, and Slicing)
- Data Structure – Matrix
  (Matrix Basics, Operations, Referencing, and apply() Function)
- Data Structure – Array
  (Array Basics, Operations, Referencing, and apply() Function)
- Data Structure – List
  (List Basics, Operations, and Referencing)
- Data Structure – Data Frame
  (Data Frame Basics, Importing and Exporting, and more next week)

❑ Data Structures

Data Structures

- **Vectors**: A vector is a one-dimensional array and an ordered collection of objects of the **same type**
- **Matrices**: A matrix is just a two-dimensional generalization of a vector
- **Arrays:** An array is a multi-dimensional generalization of a vector
- **Lists**: A list is a general form of a vector, where the elements don't need to be of the same type or dimension.
- **Dataframes**: R refers to datasets as dataframes

❑ Data Structure - Vector

- A vector is an ordered collection of objects of **the same type**. We can create a vector with all the basic data types. The simplest way to create a vector in R, is to use the **c()** command
- The function **c**(...) concatenates its arguments to form a vector
- Vector Names - use the **names()** function to assign names to each element in our vector
- To create a patterned vector
  - "**:**" sequence of integers
  - **seq()** general sequence
  - **rep()** vector of replicated elements

```
> v1 <- c(2.5, 4, 7.3, 0.1); v1
[1] 2.5 4.0 7.3 0.1
> v2 <- c("A", "B", "C", "D"); v2
[1] "A" "B" "C" "D"
> #: Sequence of integers
> v3 <- -3:3; v3
[1] -3 -2 -1  0  1  2  3
> # seq() General sequence
> seq(0, 2, by=0.5); seq(0, 2, len=6)
[1] 0.0 0.5 1.0 1.5 2.0
[1] 0.0 0.4 0.8 1.2 1.6 2.0
> # rep() Vector of replicated elements
> rep(1:5, each=2); rep(1:5, times=2)
 [1] 1 1 2 2 3 3 4 4 5 5
 [1] 1 2 3 4 5 1 2 3 4 5
```

❑ Data Structure – Reference Elements of a Vector (Indexing and Slicing)

- Use bracket notation **"[ ]"** with a vector/scalar of positions to index and or access individual elements from a vector/scalar of positions to reference elements
- A **minus sign** before the vector/scalar to **remove** elements
- Slicing - use a colon (**:**) to indicate a slice of a vector. The format is: vector[start_index:stop_index]

Example:

```
> x <- c(4, 7, 2, 10, 1, 0)
> x[4]
[1] 10
> x[1:3]
[1] 4 7 2
> x[c(2,5,6)]
[1] 7 1 0
> x[-3]
[1]  4  7 10  1  0
> x[-c(4,5)]
[1] 4 7 2 0
> x[x>4]
[1]  7 10
> x[3] <- 99
> x
[1]  4  7 99 10  1  0
```

❑ Data Structure – which() and match()

- Additional functions that will return the indices of a vector

  - ***which***() Indices of a logical vector where the condition is TRUE

  - ***which.max***() Location of the (rst) maximum element of a numeric vector

  - ***which.min***() Location of the (rst) minimum element of a numeric vector

  - ***match***() First position of an element in a vector

```
> x <- c(4, 7, 2, 10, 1, 0)
> x>=4
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
> which(x>=4)
[1] 1 2 4
> which.max(x)
[1] 4
> x[which.max(x)]
[1] 10
> max(x)
[1] 10
```

```
> y <- rep(1:5, times=5:1);y
 [1] 1 1 1 1 1 2 2 2 2 3 3 3 4 4 5
> match(1:5, y)
[1]  1  6 10 13 15
> match(unique(y), y)
[1]  1  6 10 13 15
> ?match
```

❑ Data Structure – Vector Operations

- When vectors are used in math expressions, the operations are performed **element by element**

  Example:

```
> x <- c(4,7,2,10,1,0)
> y <- x^2 + 1;
> y
[1]  17  50   5 101   2   1
> x*y
[1]   68  350   10 1010    2    0
```

❑ Data Structure – Useful Vector Functions

`sum(x)`	`prod(x)`	Sum/product of the elements of `x`
`cumsum(x)`	`cumprod(x)`	Cumulative sum/product of the elements of `x`
`min(x)`	`max(x)`	Minimum/Maximum element of `x`
`mean(x)`	`median(x)`	Mean/median of `x`
`var(x)`	`sd(x)`	Variance/standard deviation of `x`
`cov(x,y)`	`cor(x,y)`	Covariance/correlation of `x` and `y`
`range(x)`		Range of `x`
`quantile(x)`		Quantiles of `x` for the given probabilities
`fivenum(x)`		Five number summary of `x`
`length(x)`		Number of elements in `x`
`unique(x)`		Unique elements of `x`
`rev(x)`		Reverse the elements of `x`
`sort(x)`		Sort the elements of `x`
`which()`		Indices of `TRUE`s in a logical vector
`which.max(x)`	`which.min(x)`	Index of the max/min element of `x`
`match()`		First position of an element in a vector
`union(x, y)`		Union of `x` and `y`
`intersect(x, y)`		Intersection of `x` and `y`
`setdiff(x, y)`		Elements of `x` that are not in `y`
`setequal(x, y)`		Do `x` and `y` contain the same elements?

❑ Data Structures

Data Structures Continued

- Vectors: A vector is a one-dimensional and ordered collection of objects of the same type
- Matrices: A matrix is just a two-dimensional generalization of a vector
- Arrays: An array is a multi-dimensional generalization of a vector
- Lists: A list is a general form of a vector, where the elements don't need to be of the same type or dimension
- Data Frames: R refers to datasets as dataframes

❑ Data Structure – Matrices

- A matrix is just a two-dimensional generalization of a vector.
- To create a matrix, ***matrix***(***data***=NA, ***nrow***=1, ***ncol***=1, ***byrow*** = FALSE, ***dimnames*** = NULL)
  - ***data***: a vector that gives data to fill the matrix; if data does not have enough elements to fill the matrix, then the elements are recycled
  - ***nrow***: desired number of rows; ***ncol***: desired number of columns
  - ***byrow***: if FALSE (default) matrix is filled by columns, otherwise by rows
  - ***dimnames***: (optional) list of length 2 giving the row and column names respectively, list names will be used as names for the dimensions

```
> x <- matrix(c(5,0,6,1,3,5,9,5,7,1,5,3), nrow=3, ncol=4, byrow=TRUE,
+             dimnames=list(rows=c("r.1", "r.2", "r.3"),cols=c("c.1", "c.2", "c.3", "c.4")))
> x
     cols
rows  c.1 c.2 c.3 c.4
  r.1   5   0   6   1
  r.2   3   5   9   5
  r.3   7   1   5   3
```

❑ Data Structure – Referencing Elements of a Matrix

- Reference matrix elements using the **"[ ]"** just like with vectors, but now with 2-dimensions

Example:

```
> x <- matrix(c(5,0,6,1,3,5,9,5,7,1,5,3), nrow=3, ncol=4, byrow=TRUE,
+              dimnames=list(rows=c("r.1", "r.2", "r.3"),cols=c("c.1", "c.2", "c.3", "c.4")))
> x
     cols
rows  c.1 c.2 c.3 c.4
  r.1   5   0   6   1
  r.2   3   5   9   5
  r.3   7   1   5   3
> x[2,3] # Row 2, Column 3
[1] 9
> x[1,]  # Row 1
c.1 c.2 c.3 c.4
  5   0   6   1
> x[,2]  # Column 2
r.1 r.2 r.3
  0   5   1
> x[c(1,3),] # Rows 1 and 3, all Columns
     cols
rows  c.1 c.2 c.3 c.4
  r.1   5   0   6   1
  r.3   7   1   5   3
```

❑ Data Structure – Referencing Elements of a Matrix

- We can also reference parts of a matrix by using the row or column names; Sometimes it is better to reference a row/column by its name rather than by the numeric index.

  - Reference matrix elements using the "**[ ]**" but now use the column or row name, with quotations, in place of the index number

- You don't have to specify the names when you create a matrix. To get or set the column, row, or both dimension names of A: colnames(A) / rownames(A) / dimnames(A)
- Can also name the elements of a vector, c("name.1"=1, "name.2"=2)
- Use the function ***names***() to get or set the names of vector elements

❑ Data Structure – Referencing Elements of a Matrix

- Example:

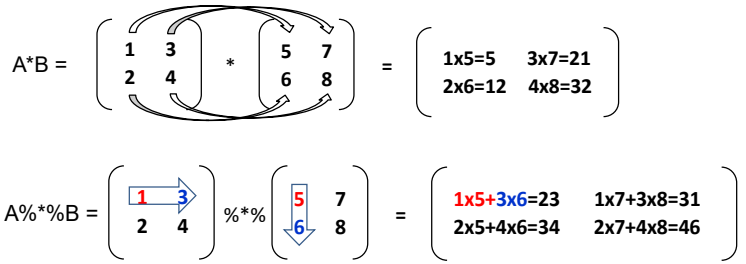
```
> N <- matrix(c(5,8,3,0,4,1), nrow=2, ncol=3, byrow=TRUE)
> colnames(N) <- c("c.1", "c.2", "c.3")
> N
     c.1 c.2 c.3
[1,]   5   8   3
[2,]   0   4   1
> N[,"c.2"] # Column named "c.2"
[1] 8 4
> colnames(N)
[1] "c.1" "c.2" "c.3"
>
> M <- diag(2)
> (MN <- cbind(M, N)) # Placing the expression in parentheses
     c.1 c.2 c.3
[1,] 1 0   5   8   3
[2,] 0 1   0   4   1
> MN[,2] # Column 2
[1] 0 1
> MN[,"c.2"] # Column named "c.2"
[1] 8 4
```

❑ Data Structure – Matrix Operations

- When matrices are used in **math expressions**, "*" the operations are performed **element by element**.
- For **matrix multiplication** use the "**%*%**" operator

Example:

```
> A <- matrix(1:4, nrow=2); A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> B <- matrix(5:8, nrow=2, ncol=2); B
     [,1] [,2]
[1,]    5    7
[2,]    6    8
> A*B
     [,1] [,2]
[1,]    5   21
[2,]   12   32
> A%*%B
     [,1] [,2]
[1,]   23   31
[2,]   34   46
```



❑ Data Structure – Matrix Operations

- If a vector is used in matrix multiplication, it will be coerced to a row or column matrix to make the arguments conformable. Using %*% on two vectors will return the inner product (%o% for outer product) as a matrix and not a scalar. Use either **c**() or **as.vector**() to convert to a scalar.

  Example:

```
> y <- 1:3
> y*y
[1] 1 4 9
> y%*%y
     [,1]
[1,]   14
> A/(y%*%y)
Error in A/(y %*% y) : non-conformable arrays
> A/c(y%*%y)
           [,1]      [,2]
[1,] 0.07142857 0.2142857
[2,] 0.14285714 0.2857143
```

❑ Data Structure – Matrix Operations

- Exercise:

```
> A <- matrix(11:14, nrow=2); A
> B <- matrix(15:18, nrow=2, ncol=2); B
> A*B=?
> A%*%B=?
```

❑ Data Structure – Useful Matrix Functions

`t(A)`	Transpose of `A`
`det(A)`	Determinate of `A`
`solve(A, b)`	Solves the equation `Ax=b` for `x`
`solve(A)`	Matrix inverse of `A`
`MASS::ginv(A)`	Generalized inverse of `A` (MASS package)
`eigen(A)`	Eigenvalues and eigenvectors of `A`
`chol(A)`	Choleski factorization of `A`
`diag(n)`	Create a `n`×`n` identity matrix
`diag(A)`	Returns the diagonal elements of a matrix `A`
`diag(x)`	Create a diagonal matrix from a vector `x`
`lower.tri(A),upper.tri(A)`	Matrix of logicals indicating lower/upper triangular matrix
`apply()`	Apply a function to the margins of a matrix
`rbind(...)`	Combines arguments by rows
`cbind(...)`	Combines arguments by columns and
`dim(A)`	Dimensions of `A`
`nrow(A), ncol(A)`	Number of rows/columns of `A`
`colnames(A), rownames(A)`	Get or set the column/row names of `A`
`dimnames(A)`	Get or set the dimension names of `A`

❑ Data Structure – apply() Function

- The ***apply***() function is used for applying functions to the margins of a matrix, array, or dataframes
  apply(X, MARGIN, FUN, ...)
  - X: A matrix, array or dataframe
  - MARGIN: Vector of subscripts indicating which margins to apply the function to
    1=rows, 2=columns, c(1,2)=rows and columns
  - FUN: Function to be applied
  - ... Optional arguments for FUN
- You can also use your own function (more on this later)

Example:

```
> x <- matrix(1:12, nrow=3, ncol=4); x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> apply(x, 1, sum) # Row totals
[1] 22 26 30
> apply(x, 2, mean) # Column means
[1]  2  5  8 11
```

❑ Data Structure – apply() Function

- Exercise:

```
> x <- matrix(1:18, nrow=3, ncol=6); > x
> apply(x, 1, max)       # Row max
> apply(x, 2, median)  # Column median
```

❑ Data Structure

Data Structures

- Vectors: A vector is a one-dimensional and ordered collection of objects of the same type
- Matrices: A matrix is just a two-dimensional generalization of a vector
- Arrays: An array is a multi-dimensional generalization of a vector
- Lists: A list is a general form of a vector, where the elements don't need to be of the same type or dimension.
- Dataframes: R refers to datasets as dataframes

❑ Data Structure – Arrays

- To create an array, ***array(data*** = *NA*, ***dim*** = *length(data),* ***dimnames*** = *NULL)*

  - **data:** A vector that gives data to fill the array; if data does not have enough elements to fill the matrix, then the elements are recycled

  - **dim:** Dimension of the array, a vector of length one or more giving the maximum indices in each dimension

  - **dimnames:** Name of the dimensions, list with one component for each dimension, either NULL or a character vector of the length given by dim for that dimension. The list can be named, and the list names will be used as names for the dimensions.

- Values are entered by columns
- Like with vectors and matrices, when arrays are used in math expressions so the operations are performed element by element.
- Also like vectors and matrices, the elements of an array must all be **of the same type** (numeric, character, logical, etc.)

❑ Data Structure – Arrays

- Example: 2x2x3 array

```
> z <- array(1:12,dim =c(2,2,3),dimnames=list(c("Row1","Row2"), c("Col1","Col2"), c("Mat1","Mat2","Mat3"))); z
, , Mat1

     Col1 Col2
Row1    1    3
Row2    2    4

, , Mat2

     Col1 Col2
Row1    5    7
Row2    6    8

, , Mat3

     Col1 Col2
Row1    9   11
Row2   10   12
```

❑ Data Structure – Arrays

- Example: 2x3x4 array

```
> w <- array(1:24, dim=c(2,3,4),dimnames=list(c("A","B"), c("X","Y","Z"), c("N","M","O","P")))
> w
, , N

  X Y Z
A 1 3 5
B 2 4 6

, , M

  X  Y  Z
A 7  9 11
B 8 10 12

, , O

   X  Y  Z
A 13 15 17
B 14 16 18

, , P

   X  Y  Z
A 19 21 23
B 20 22 24
```

❑ Data Structure – Arrays

- Exercise: Create your own 2x2x4 array

```
> z <- array(1:16,dim =c(2,2,4),dimnames=list(c("A","B"), c("C","D"), c("E","F","G","H")))
> z
, , E

  C D
A 1 3
B 2 4

, , F

  C D
A 5 7
B 6 8

, , G

   C  D
A  9 11
B 10 12

, , H

   C  D
A 13 15
B 14 16
```

❑ Data Structure – Referencing of Elements of Arrays

- Reference array elements using the "**[ ]**" just like with vectors and matrices, but now with more dimensions

Example:

```
> w <- array(1:24, dim=c(2,3,4),dimnames=list(c("A","B"), c("X","Y","Z"), c("N","M","O","P")))
> w[2,3,1] # Row 2, Column 3, Matrix 1
[1] 6
> w[,"Y",] # Column named "Y"
  N  M  O  P
A 3  9 15 21
B 4 10 16 22
> w[1,,] # Row 1
  N  M  O  P
X 1  7 13 19
Y 3  9 15 21
Z 5 11 17 23
> w[1:2,,"M"] # Rows 1 and 2, Matrix "M"
  X  Y  Z
A 7  9 11
B 8 10 12
```

❑ Data Structure – Referencing of Elements of Arrays

- Exercise:

```
> v<-array(1:12, dim=c(2,3,2), dimnames=list(c("a","b"),c("x","y","z"),c("n","m")))
> v[2,3,2]      # row 2, column 3, matrix 2
> v[,"y",]      # column named "y"
> v[2,,]        # row 2
> v[1,,"m"]
```

❑ Data Structure – Arrays: Functions

`apply()`	Apply a function to the margins of an array
`aperm()`	Transpose an array by permuting its dimensions
`dim(x)`	Dimensions of `x`
`dimnames(x)`	Get or set the dimension names of `x`

❑ Data Structure – Arrays: apply()

- We can use the apply() function for more than one dimension
- For a 3-dimensional array there are now three margins to apply the function to:

  1=rows, 2=columns, and 3=matrices.

Example:

```
> z <- array(1:12,dim =c(2,2,3),dimnames=list(c("Row1","Row2"), c("Col1","Col2"), c("Mat1","Mat2","Mat3")));z
, , Mat1

     Col1 Col2
Row1    1    3
Row2    2    4

, , Mat2

     Col1 Col2
Row1    5    7
Row2    6    8

, , Mat3

     Col1 Col2
Row1    9   11
Row2   10   12

> apply(z, 2, sum)       # Column sums
Col1 Col2
  33   45
> apply(z, 1, sum)       # row sums
Row1 Row2
  36   42
> apply(z, c(1,3), sum)  # Row and matrix sums
     Mat1 Mat2 Mat3
Row1    4   12   20
Row2    6   14   22
```

❑ Data Structure

Data Structures

- Vectors: A vector is an ordered collection of objects of the same type
- Matrices: A matrix is just a two-dimensional generalization of a vector
- Arrays: An array is a multi-dimensional generalization of a vector
- Lists: A list is a general form of a vector, where the elements don't need to be of the same type or dimension
- Data Frames: R refers to datasets as dataframes

❑ Data Structure – Lists

- A list is a general form of a vector, where ***the elements don't need to be of the same type, dimension, and storage mode.***
- The function ***list(...)*** creates a list of the arguments
- Arguments have the form name=value. Arguments can be specified with and without names.

Example:

```
> x <- list(num=c(1,2,3), "Nick", identity=diag(2)); x
```

R Output:

```
> x <- list(num=c(1,2,3), "Nick", identity=diag(2)); x
$num
[1] 1 2 3

[[2]]
[1] "Nick"

$identity
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

❑ Data Structure – List Functions

- To convert a vector to a list, you can use the ***as.list()*** function; The ***unlist()*** function converts a list to a vector

  Example:

  > student.vec<-c(name="John",year=2, classtaken=c("CS101","CS102","CS103"), GPA=3.9)

  > student.list<-as.list(student.vec)

  > str(student.list)

  > unlist(student.list)

```
> student.vec<-c(name="John",year=2, classtaken=c("CS101","CS102","CS103"), GPA=3.9)
> student.list<-as.list(student.vec)
> str(student.list)
List of 6
 $ name       : chr "John"
 $ year       : chr "2"
 $ classtaken1: chr "CS101"
 $ classtaken2: chr "CS102"
 $ classtaken3: chr "CS103"
 $ GPA        : chr "3.9"
> unlist(student.list)
      name       year classtaken1 classtaken2 classtaken3        GPA
    "John"        "2"     "CS101"     "CS102"     "CS103"      "3.9"
```

❑ Data Structure – Reference Elements of a List

- Elements of a list can be referenced using "[ ]" as well as "[[ ]]" or "$"

Exercise:

```
> x <- list(num=c(1,2,3), "Nick", identity=diag(2))
> x[[2]]          # Second element of x
> x[["num"]]      # Element named "num"
> x$identity      # Element named "identity"
> x[[3]][1,]      # First row of the third element
> x[1:2]          # Create a sublist of the first two elements
```

R Output:

```
> x <- list(num=c(1,2,3), "Nick", identity=diag(2))
> x[[2]] # Second element of x
[1] "Nick"
> x[["num"]] # Element named "num"
[1] 1 2 3
> x$identity # Element named "identity"
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> x[[3]][1,] # First row of the third element
[1] 1 0
> x[1:2] # Create a sublist of the first two elements
$num
[1] 1 2 3

[[2]]
[1] "Nick"
```

❑ Data Structure – Reference Elements of a List

- Using single square brackets, [ ], instead of double square brackets [[ ]], returns a list with the selected component

  Exercise:

  > student.vec<-c(name="John",year=2, classtaken=c("CS101","CS102","CS103"), GPA=3.9)

  > student[3]

  > student[[3]]

❑ Data Structure – Concatenating List *c()* Function

- Using the concatenate function, **c()**, to concatenate lists

Example:

```
> list1<-c(list(letters[1:3],2:4),list(c(1,3,5)))
> str(list1)
```

R Output:

```
> list1<-c(list(letters[1:3],2:4),list(c(1,3,5)))
> str(list1)
List of 3
 $ : chr [1:3] "a" "b" "c"
 $ : int [1:3] 2 3 4
 $ : num [1:3] 1 3 5
```

❑ Data Structure – List Example

- Lists are sometimes called ***recursive vector*** because a list contains other lists

  Example:

  > nestedList<-list(c1=1, letters, list(c1=2,c2=LETTERS))

  > str(nestedList)

```
> nestedList<-list(c1=1, letters, list(c1=2,c2=LETTERS))
> str(nestedList)
List of 3
 $ c1: num 1
 $   : chr [1:26] "a" "b" "c" "d" ...
 $   :List of 2
  ..$ c1: num 2
  ..$ c2: chr [1:26] "A" "B" "C" "D" ...
```

❑ Data Structure – Useful List Functions

`lapply()`	Apply a function to each element of a list, returns a list
`sapply()`	Same as `lapply()`, but returns a vector or matrix by default
`vapply()`	Similiar to `sapply()`, but has a pre-specified type of return value
`replicate()`	Repeated evaluation of an expression, useful for replicating lists
`unlist(x)`	Produce a vector of all the components that occur in `x`
`length(x)`	Number of objects in `x`
`names(x)`	Names of the objects in `x`

❑ Data Structure

Data Structures

- Vectors: A vector is an ordered collection of objects of the same type
- Matrices: A matrix is just a two-dimensional generalization of a vector
- Arrays: An array is a multi-dimensional generalization of a vector
- Lists: A list is a general form of a vector, where the elements don't need to be of the same type or dimension
- Data Frames: R refers to datasets as dataframes

❑ Data Structure – Data Frame Basics

- **R refers to datasets as dataframes**
- A dataframe is a matrix-like structure, where the columns can be of different types. You can also think of a dataframe as a list. Each column is an element of the list and each element has the same length.
- A dataframe is the fundamental data structure used by R 's statistical modeling functions

Example:

State	Population	Income	Life.Exp	State.Region
Alabama	3615	3624	69.05	South
Alaska	365	6315	69.31	West
Arizona	2212	4530	70.55	West
Arkansas	2110	3378	70.66	South
California	21198	5114	71.71	West
Colorado	2541	4884	72.06	West
Connecticut	3100	5348	72.48	Northeast

❑ Data Structure – Data Frame Basics

- To get/replace elements of a dataframe use either "[ ]" or '$". The "[ ]" are used to access rows and columns and the "$" is used to get entire columns

state.x77, is a built-in R dataset of state facts stored as a matrix; Type data(), to see a list of built-in datasets

```
data()                              # Type data(), to see a list of built-in datasets
data <- data.frame(state.x77)       # First, convert to a dataframe
head(data) / tail(data)             # Print the first / last few rows of a dataset/matrix
names(data) or colnames(data)       # Column names
rownames(data)                      # Row names
dim(data)                           # Dimension of the dataframe
data[,c("Population", "Income")]    # "Population" and "Income" columns
data$Area                           # Get the column "Area"
data[1:5,]                          # Get the first five rows
```

❑ Data Structure – Data Frame Importing By Using *scan()*

- The ***scan***() can also be used to import datasets including from keyboard. It is a very exible function but is also **harder to use**. The function **read.table()** provides an easier interface.
- R scan Function: ***scan()*** function read data from screen or file.

scan(file = "", what = double(), nmax = -1, n = -1, sep = "", quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",

skip = 0, nlines = 0, na.strings = "NA", flush = FALSE, fill = FALSE, strip.white = FALSE,

quiet = FALSE, blank.lines.skip = TRUE, multi.line = TRUE, comment.char = "", allowEscapes = FALSE,

fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)

file: the name of a file to read from, if "", then read in from the keyboard or from ***stdin()***

what: the type of data, including logical, integer, numeric, complex, character, raw, and list

For more detailed description about the arguments of ***scan()*** function, please type in ***"?scan***" to search

❑ Data Structure – Data Frame Importing By Using *scan()*

- Examples:

```
> setwd("C:/TeachingUChicago/Spring2020/Week3")

> getwd() [1] "C:/TeachingUChicago/Spring2020/Week2"

> x <- scan("scandata.csv",what="character",skip=1,quiet=TRUE); x
```

[1] "r1,1,0,1,0,0,1,0,2" "r2,1,2,5,1,2,1,2,1" "r3,0,0,9,2,1,1,0,1" "r4,0,0,2,1,2,0,0,0" [5] "r5,0,2,15,1,1,0,0,0" "r6,2,2,3,1,1,1,0,0" "r7,2,2,3,1,1,1,0,1"

```
> y <- scan("scandata.csv",what="character",quiet=TRUE); y
```

[1] ",t1,t2,t3,t4,t5,t6,t7,t8" "r1,1,0,1,0,0,1,0,2" "r2,1,2,5,1,2,1,2,1" [4] "r3,0,0,9,2,1,1,0,1" "r4,0,0,2,1,2,0,0,0" "r5,0,2,15,1,1,0,0,0" [7] "r6,2,2,3,1,1,1,0,0" "r7,2,2,3,1,1,1,0,1"

❑ Data Structure – Data Frame Importing By Using *read_table()*

- Make sure ***readr*** is installed which is part of the core ***tidyverse*** package (Included via Anaconda Installation)
- The function ***read_table()*** is the easiest way to import data into R. The preferred raw data format is either .txt or a tab delimited text file or a comma-separate file (CSV).

  - The simplest and recommended way to import Excel files is to do a **Save As** in Excel and save the file as a tab delimited or CSV file and then import this file into R.

  - Similarly, for SAS files export the file as a tab delimited or CSV file using proc export.

- Two commonly used functions for importing data and they are almost identical:

  read_table() Reads a file in table format and creates a dataframe

  read_csv() Reads comma-delimited files (same as read.table() where sep=",“)

  read_csv2() Reads semicolon-separated files

  read_tsv() Reads tab-delimiter files

❑ Data Structure – Data Frame Importing By Using *read_table()*

- ***read_table***(**file**, header = FALSE, **sep** = "", skip, as.is, stringsAsFactors=TRUE)

  - **file**: the name of the file to import

  - header: logical, does the first row contain column labels

  - **sep**: field separator character: sep=" " space (default); sep="nt" tab-delimited; sep="," comma-separated

  - skip: number of lines to skip before reading data

  - as.is: vector of numeric or character indices which specify which columns should not be converted to factors

  - stringsAsFactors: logical should character vectors be converted to factors

- There are many more arguments for read.table that allow you to adjust for the format of your data

❑ Data Structure – Data Frame Importing By Using *read_table()*

- The default field separator is a space, if you use this separator keep in mind that,
  - Column names cannot have spaces
  - Character data cannot have spaces
  - There can be trouble interpreting extra blank spaces as missing data. Need to include missing data explicitly in the dataset by using NA
- Recommended approach is to use commas to separate the fields. By importing a CSV file we don't have any of the problems that can occur when you use a space.

❑ Data Structure – Data Frame Importing By Using *read_table()*

- Example:

```
> data <- read_table("KaggleHealthcare.csv")
> str(data) # Gives the structure of data
> View(data)
```

- Exercise: Import a .csv file by using read_table()

❑ Data Structure – Data Frame Importing By Using *read_csv()*

- ***read_csv:*** use similar syntax as other functions

  - **The first argument to read_csv() is most important as it is the path to the file to read**

    Example: > datacsv <- read_csv("KaggleHealthcare.csv")

    > View(datacsv)

  - If there are a few lines of metadata at the top of the file, use skip=n to skip the first n lines

    Example: > datacsvskip3 <- read_csv("KaggleHealthcare.csv", skip=3)

    > View(datacsvskip3)

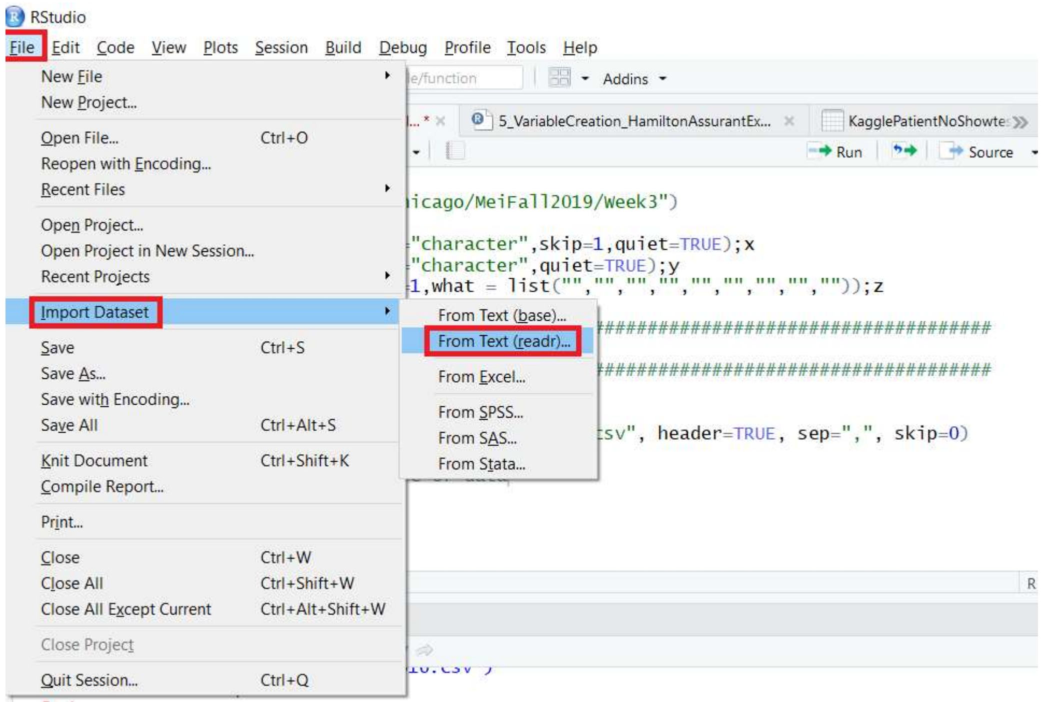
❑ Data Structure – Data Frame Importing By Using *read_excel()*

- Make sure to ***readxl*** is installed which reads Excel files (Including in Anaconda Installation)
- The function ***read_excel()*** is to import Excel data into R.

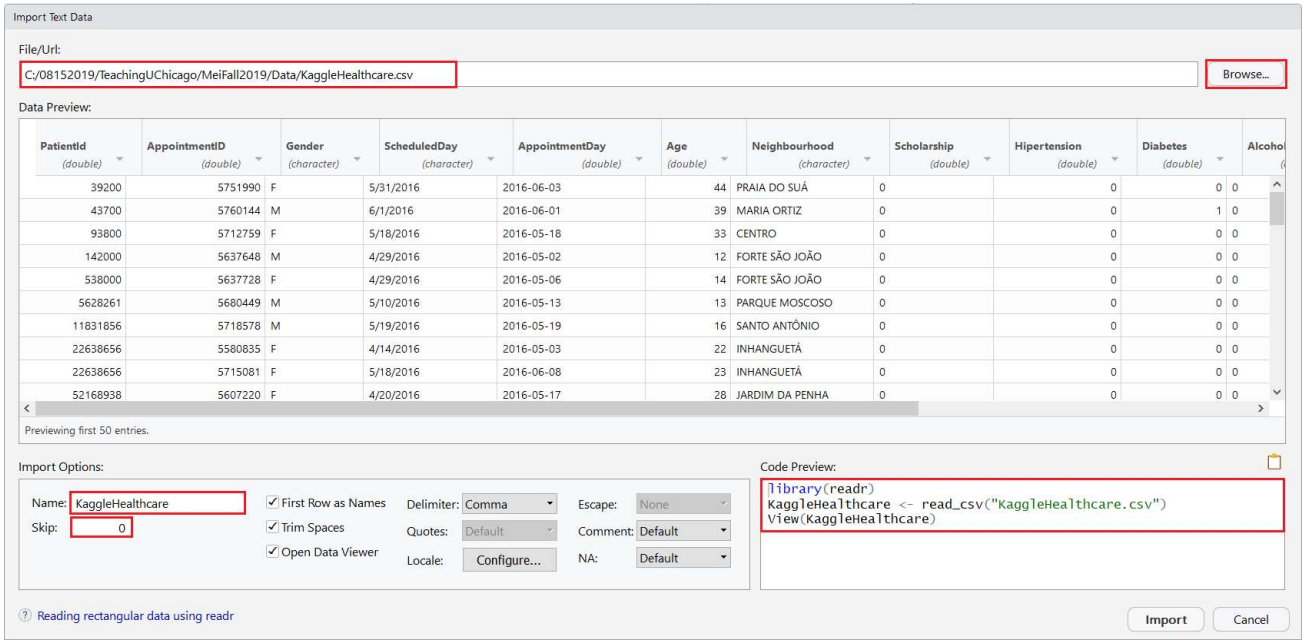
  Example:

```
> library(readxl)
> Testxlsx <- read_excel("AutoCollisionxlsx.xlsx")
> View(Testxlsx)
> Testxls <- read_excel("AutoCollisionxls.xls")
> View(Testxls)
```

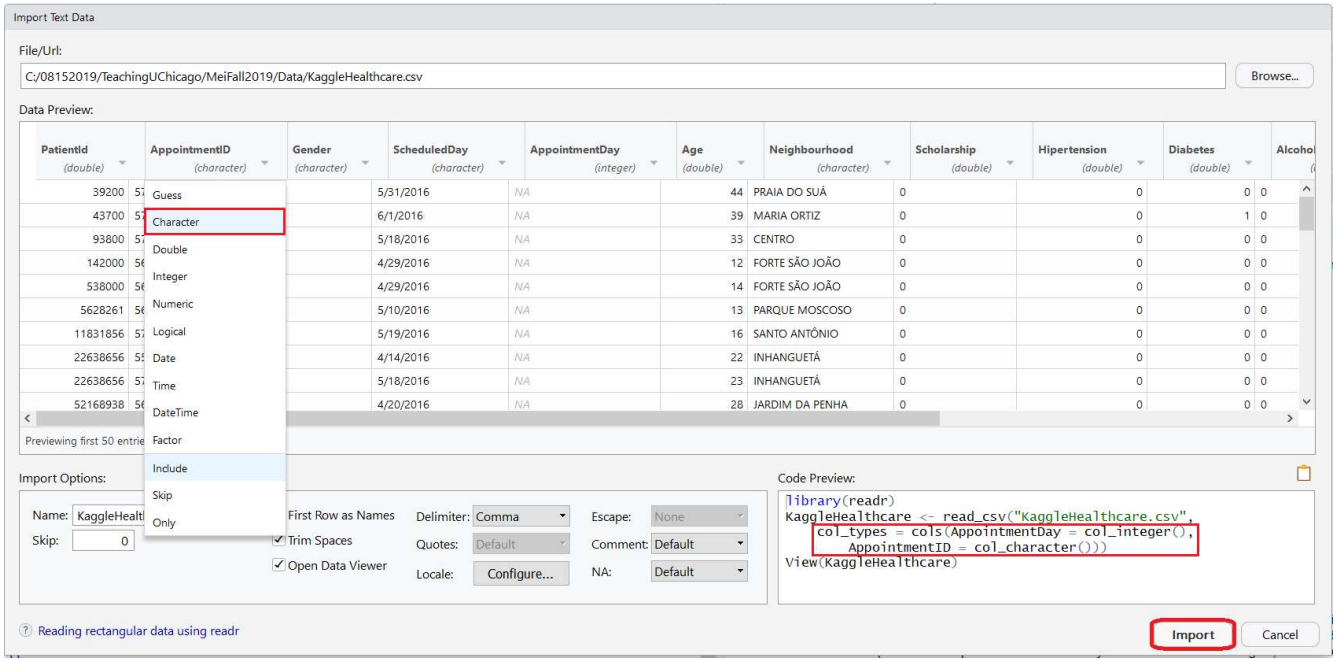
❑ Data Structure – Data Frame Importing Data By Clicking on buttons 1



❑ Data Structure – Data Frame Importing Data By Clicking on buttons 2



❑ Data Structure – Data Frame Importing Data By Clicking on buttons 3



❑ Data Structure – Data Frame Importing Data Comparison with Base R

If you've used R before, you might wonder the difference between read.csv() and read_csv(). There are a few good reasons to favor readr functions over the base R equivalents:

- read_csv() is from readr and much faster for large csv files. They are typically much faster (~10x) than their base equivalents. Long-running jobs have a progress bar, so you can see what's happening. If you are looking for raw speed, try ***data.table::fread()***
- They produce tibbles, and they don't convert character vectors to factors, use row names, or munge the column names. These are common sources of frustration with the base R functions
- They are more reproducible. Base R functions inherit some behavior from your operating system and environment variables, so import code that works on your computer might not work on someone else's

❑ Data Structure – Data Frame Writing to a File *write_csv()*

- ***readr*** also comes with two useful functions for writing data back to disk (exporting):

  write_csv() and write_tsv().

- Both functions increase the chances of the output file read back in correctly by:

  - Always encoding strings in **UTF-8**

  - Saving dates and date-times in **ISO8601** format so they are easily parsed elsewhere

  Example:

  write_csv(data,"test_export.csv")

❑ Data Structure – Data Frame Writing to a File *write.table()*

- ***write.table***(x, file="", sep=" ", row.names=TRUE, col.names=TRUE)

  - x: the object to be saved, either a matrix or dataframe

  - file: file name

  - sep: field separator

  - row.names: logical, include row.names

  - col.names: logical, include col.names

- There is also a wrapper function, write.csv() for creating a CSV file by calling write.table() with sep=",".

  # Export R dataframe as a CSV file

  > write.table(data, "export.example.csv", sep=",", row.names=FALSE)

❑ Data Structure – Data Frame Attaching a File Using *attach()*

- R objects that reside in other R objects can require a lot of typing to access.

  Example:
  To refer to a variable x in a dataframe df, one could type ***df$x***. This is no problem when the dataframe and variable names are short, but can become burdensome when longer names or repeated references are required, or objects in complicated structures must be accessed.

- The function ***search()*** displays the search path for R objects. When R looks for an object it first looks in the global environment then proceeds through the search path looking for the object. The search path lists attached dataframes and loaded libraries.
- The function ***attach()*** (***detach()***) attaches (detaches) a dataframe to the search path. This means that the column names of the dataframe are searched by R when evaluating a variable, so variables in the dataframe can be accessed by simply giving their names.

❑ Data Structure – Data Frame Attaching a File Using *attach()*

Example:

```
ds <- read_csv ("KaggleHealthcare.csv",skip=0)
head(ds)
mean(ds$Age)  # Average Age
search()          # Search path

attach(ds)        # Attach dataset
search()          # If R can find ds in the global environment

mean(Age)         # Average Age
detach(ds)        # Detach dataset
search()          # Check if ds is not in the global environment
```

❑ Data Structure – Caution with Attaching a File

- ***attach()*** is okay if you are just working on one dataset and your purpose is mostly on analysis, but if you are going to have several datasets and lots of variables avoid using *attach().*
- If you attach a dataframe and use simple names like x and y, it is very possible to have very different objects with the same name which can cause problems

  Note: R prints a warning message if attaching a dataframe causes a duplication of one or more names.
- Several modeling functions like lm() and glm() have a data argument so there is no need to attach a dataframe
- For functions without a data argument use ***with().*** This function evaluates an R expression in an environment constructed from the dataframe.
- If you do use attach() call detach() when you are finished working with the dataframe to avoid errors.

❑ Data Structure – Caution with Attaching a File

- Example - Caution with attach()

- Type defined in the global environment

```
search()
attach(ds)    # Attach ds
detach(ds)
```

- Use ***with*** instead of attach, can also be simpler than using the $

```
with(ds, table(No_show, Age))
```

- Some modeling functions have a data argument

```
lm(No_show~Age, data=ds)
```

❑ Summary Statistics - Vectors

Functions for calculating summary statistics of vector elements

`mean(x)`	Mean of `x`
`median(x)`	Median of `x`
`var(x)`	Variance of `x`
`sd(x)`	Standard deviation of `x`
`cov(x,y)`	Covariance of `x` and `y`
`cor(x,y)`	Correlation of `x` and `y`
`min(x)`	Minimum of `x`
`max(x)`	Maximum of `x`
`range(x)`	Range of `x`
`quantile(x)`	Quantiles of `x` for the given probabilities

❑ Summary Statistics - Dataframes

- Functions for calculating summary statistics of the columns of a dataframe

`summary()`	Summary statistics of each column; type of statistics depends on data type
`apply()`	Apply a function to each column, works best if all columns are the same data type
`tapply()`	Divide the data into subsets and apply a function to each subset, returns an array
`by()`	Similar to `tapply()`, return an object of class by
`ave()`	Similar to `tapply()`, returns a vector the same length as the argument vector
`aggregate()`	Similar to `tapply()`, returns a dataframe
`sweep()`	"Sweep out" a summary statistic from a dataframe, matrix or array

Note: The major difference between tapply(), ave(), by(), and aggregate() is the format of the returned object

Q & A

Thank You

Contact Information:

Your comments and questions are valued and encouraged.

Mei Najim
E-mail: mnajim@uchicago.edu

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