Online Academic Data Analysis Bootcamp Using Open-Access Program R

Session 3

Data Management

Joseph Ndiba (BSc, MSc)
Climate Change & Adaptation

Patrick Njage (BSc, MSc, Mstat, PhD)
Technical University of Denmark
Academic Data Analysts (WWW.ACADEMICDATAANALYSTS.ORG/)
Institute of Statistics and Data Science (ISADS) (WWW.ISADS.ORG)

Outline

Part 1: Common Data Management Options in R

Part 2: Transforming Your Data with dplyr

Part 1: Common Data Management Options in R Outline

- **Sorting Data**
- **Merging Data**
- **Aggregating Data**
- **Reshaping Data**
- **Subsetting Data**
- **Data Type Conversion**

Sorting Data

order( ) function used to sort dataframes- default is sorting ASCENDING.

Prepend the sorting variable by a minus sign to indicate DESCENDING order. Here are some examples.

```
# sorting examples using the mtcars dataset
attach(mtcars)

# sort by mpg
newdata <- mtcars[order(mpg),]

# sort by mpg and cyl
newdata <- mtcars[order(mpg, cyl),]

#sort by mpg (ascending) and cyl (descending)
newdata <- mtcars[order(mpg, -cyl),]

detach(mtcars)
```

Merging Data: Adding Columns

To merge two data frames (datasets) horizontally, use the **merge** function. In most cases, you join two data frames by one or more common key variables (i.e., an inner join).

```
# merge two data frames by ID
total <- merge(data frameA,data frameB,by="ID")
```

```
# merge two data frames by ID and Country
total <- merge(data frameA,data frameB,by=c("ID","Country"))
```

Horizontal joins like this are typically used to add variables to a data frame.

Merging Data: Adding Columns

Examples: Two simple dataframes

Dataframe “x” and ”y”

```
#Dataframe "x"
name <- c("John", "Paul", "George", "Ringo", "Stuart", "Pete")
instrument <- c("guitar", "bass", "guitar", "drums", "bass", "drums")
x <- data.frame(name, instrument)
#Dataframe “y”
 name <- c("John", "Paul", "George", "Ringo", "Brian")
band <- c("TRUE", "TRUE", "TRUE", "TRUE", "FALSE")
y <- data.frame(name, band)
```

Merging Data: Adding Columns

```
total <- merge(x, y,by=“name")
```

```
> x
    name instrument
1   John      guitar
2   Paul        bass
3 George      guitar
4  Ringo       drums
5 Stuart        bass
6   Pete       drums
> y
    name  band
1   John  TRUE
2   Paul  TRUE
3 George  TRUE
4  Ringo  TRUE
5  Brian FALSE
> total
    name instrument band
1 George      guitar TRUE
2   John      guitar TRUE
3   Paul        bass TRUE
4  Ringo       drums TRUE
>
```

Merging Data: Adding Rows

To join two data frames (datasets) vertically, use the **rbind** function. The two data frames **must** have the same variables, but they do not have to be in the same order.

```
total <- rbind(data frameA, data frameB)
        dataframeA <- mtcars[1:4,]
        dataframeB <- mtcars[5:8,]
        dataframe_both <- rbind(dataframeA, dataframeB) # combine objects as rows
```

If data frameA has variables that data frameB does not, then either:

- Delete the extra variables in data frameA or

- Create the additional variables in data frameB and set them to NA (missing) before joining them with **rbind( )**.

Merging Data: Adding Rows

```
> dataframeA
                  mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4        21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag    21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710       22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive   21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
> dataframeB
                   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
Hornet Sportabout 18.7   8 360.0 175 3.15 3.44 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.46 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.57 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.19 20.00  1  0    4    2
> dataframe_both
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
> |
```

Aggregating Data

It is relatively easy to collapse data in R using one or more BY variables and a defined function.

Example: aggregate data frame mtcars by cyl and vs, returning means for numeric variables

```
attach(mtcars) # to make the variables in the dataframe directy available in the workspace
aggdata <-aggregate(mtcars, by=list(cyl,vs),  FUN=mean, na.rm=TRUE)
print(aggdata)
detach(mtcars)
```

```
  Group.1 Group.2      mpg cyl   disp       hp     drat       wt     qsec vs        am
1       4       0 26.00000   4 120.30  91.0000 4.430000 2.140000 16.70000  0 1.0000000
2       6       0 20.56667   6 155.00 131.6667 3.806667 2.755000 16.32667  0 1.0000000
3       8       0 15.10000   8 353.10 209.2143 3.229286 3.999214 16.77214  0 0.1428571
4       4       1 26.73000   4 103.62  81.8000 4.035000 2.300300 19.38100  1 0.7000000
5       6       1 19.12500   6 204.55 115.2500 3.420000 3.388750 19.21500  1 0.0000000
      gear     carb
1 5.000000 2.000000
2 4.333333 4.666667
3 3.285714 3.500000
4 4.000000 1.500000
5 3.500000 2.500000
```

When using the **aggregate()** function, the by variables must be in a list (even if there is only one).

Reshaping Data: Transpose

Use the t() function to transpose a matrix or a data frame. In the later case, rownames become variable (column) names.

```
# example using built-in dataset
mtcars
t(mtcars)
```

Reshaping Data: The Reshape Package

Hadley Wickham has created a comprehensive package called reshape to massage data. Both an introduction and article are available. There is even a video!

Basically, you "melt" data so that each row is a unique id-variable combination. Then you "cast" the melted data into any shape you would like.

Reshaping Data: The Reshape Package

Creating the data frame

```
id <- c(1, 1, 2, 2)
time <- c(1, 2, 1, 2)
measure_1 <- c(5, 3, 6, 2)
measure_2 <- c(6, 5, 1, 4)
mydata <- data.frame(id, time, measure_1, measure_2)
```

```
> 
> mydata
  id time measure_1 measure_2
1  1    1         5         6
2  1    2         3         5
3  2    1         6         1
4  2    2         2         4
> 
```

Reshaping Data: The Reshape Package

Melt function

```
library(reshape)
mdata <- melt(mydata, id=c("id","time"))
```

You must specify the variables needed to uniquely identify each measurement (ID and Time)

The variable indicating the measurement variable names (measure_1 or measure_2) is created for you automatically.

```
> mdata
   id time  variable value
1   1    1 measure_1     5
2   1    2 measure_1     3
3   2    1 measure_1     6
4   2    2 measure_1     2
5   1    1 measure_2     6
6   1    2 measure_2     5
7   2    1 measure_2     1
8   2    2 measure_2     4
>
```

Reshaping Data: The Reshape Package

Cast function

Now that the data is in a melted form, it can be recast into any shape, using the cast() function.

newdata <- cast(md, formula, FUN)

Cast the melted data
cast(data, formula, function)

Example 1. Average variable for each subject

subjmeans <- cast(mdata, id~variable, mean)

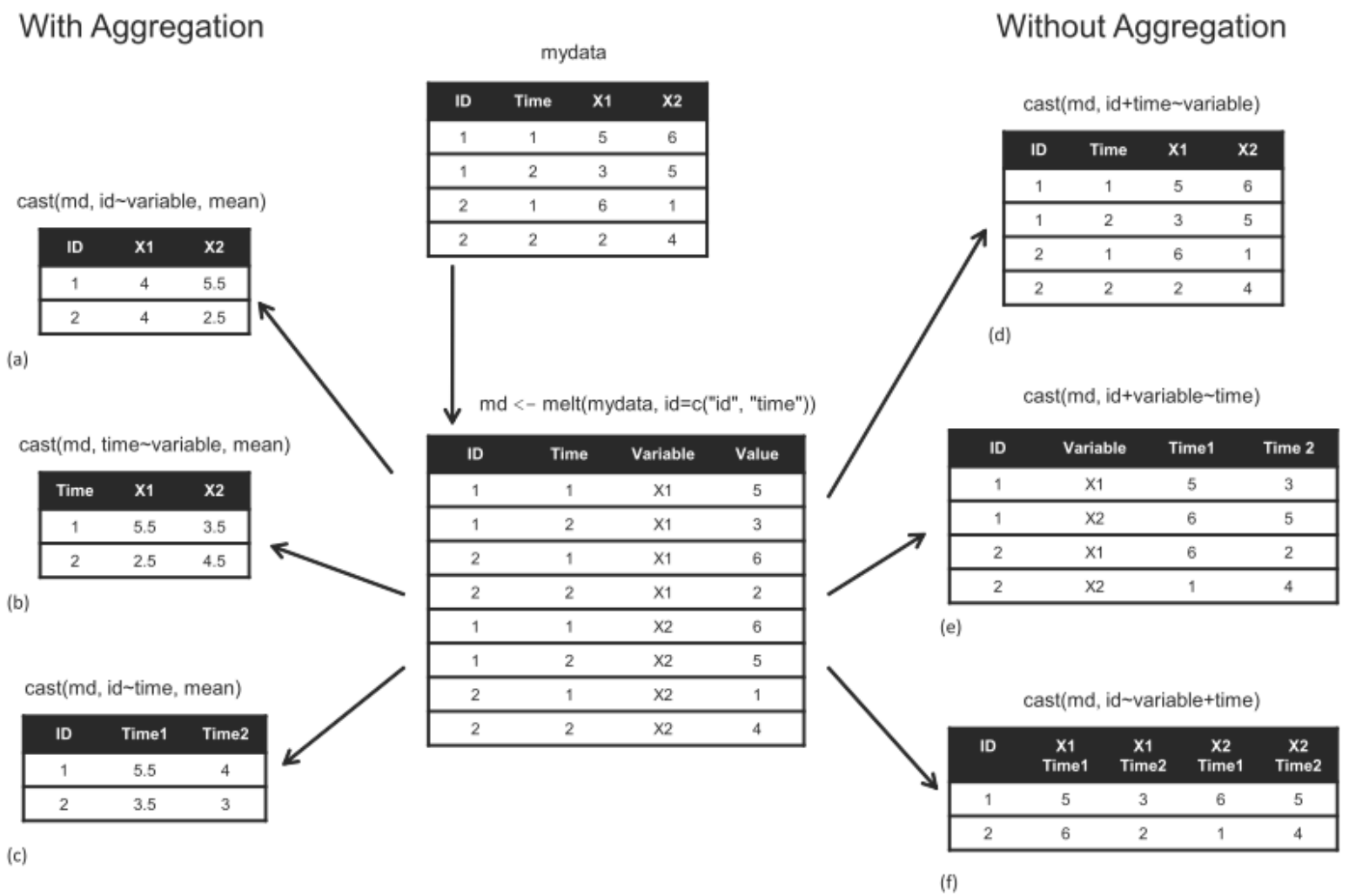
Example 2. Average variable for each time

timemeans <- cast(mdata, time~variable, mean)

```
> subjmeans
  id measure_1 measure_2
1  1         4       5.5
2  2         4       2.5
```

```
> timemeans
  time measure_1 measure_2
1    1       5.5       3.5
2    2       2.5       4.5
>
```

Reshaping a Dataset



Reshaping data with the `melt()` and `cast()` functions

Subsetting Data

- R has powerful indexing features for accessing object elements.
- These features can be used to select and exclude variables and observations.
- This section demonstrates:
  - how to keep or delete variables,
  - how to keep or delete observations, and
  - how to take random samples from a dataset.

Subsetting Data: Selecting (Keeping) Variables

```
# select variables v1, v2, v3
myvars <- c("v1", "v2", "v3")
newdata <- mydata[myvars]

myvars_mtcars <- c("mpg" , "cyl" , "disp")

newdata_mtcars <- mtcars[myvars_mtcars]

# select 1st and 5th thru 10th variables
newdata <- mydata[c(1,5:10)]

newdata2_mtcars <- mtcars[c(1,5:10)]
```

Subsetting Data: Excluding (DROPPING) Variables

```
# exclude variables v1, v2, v3
myvars <- names(mydata) %in% c("v1",
"v2", "v3")
newdata_mtcars <- mydata[!myvars]


# exclude 3rd and 5th variable
newdata <- mydata[c(-3,-5)]

# delete variables v3 and v5
mydata$v3 <- mydata$v5 <- NULL
```

Example: exclude variables am, gear, carb

```
myvars <- names(mtcars) %in% c("am",   "gear",
"carb")
newdata_mtcars <- mtcars[!myvars]

# exclude 3rd and 5th variable
newdata_mtcars_2 <- mtcars[c(-3,-5)]

# delete variables am and gear from DataframeA
dataframeA$am <- dataframeA$gear <- NULL
```

Subsetting Data: Selecting Observations

```
# first 5 observations
newdata_mtcars_5 <- mtcars[1:5,]

# based on variable values
newdata_mtcars_carb_mpg <- mtcars[ which(mtcars$carb==2 & mtcars$mpg >
20.09), ]

# or
attach(mtcars)
newdata_mtcars_carb_mpg <- mtcars[ which(mtcars$carb==2 & mtcars$mpg >
20.09), ]
detach(mtcars)
```

Subsetting Data: Selection using the Subset Function

The **subset( )** function is the easiest way to select variables and observations. In the following example, we select all rows that have a value of weight greater than or equal to 3.610 or wt less then 2.58. We keep the mpg, cyl and disp columns.

```
# using subset function
newdata_mtcars_wt <- subset(mtcars, wt >= 3.610 | wt < 2.58, select=c(mpg, cyl, disp))
```

In the next example, we select cars with miles per gallon greater than 20.09 and 2 carburetors but we keep variables horsepower *through* 1/4 mile time (hp, drat, wt and qsec).

```
# using subset function (part 2)
newdata_mtcars_range <- subset(mtcars, mtcars$carb==2 & mtcars$mpg > 20.09,
select=hp:qsec)
```

Subsetting Data: Random Samples

Use the **sample( )** function to take a **random sample of size n** from a dataset.

Example: take a random sample of size 15 from the dataset *mtcars* sample without replacement

```
mysample_mtcars <- mtcars[sample(1:nrow(mtcars), 15, replace=FALSE),]
```

Part 2: Transforming Your Data with dplyr

Outline

Dplyr is aimed at simplifying manipulating, sorting, summarizing, and joining data frames.

Basic dplyr package functions are introduced here including:

select() selects variables

filter() provides basic filtering capabilities

group_by() groups data by categorical levels

summarise() summarizes data by functions of choice

arrange() orders data

join() joins separate dataframes

Dplyr: package and %>% Operator

- Package Utilized

install.packages("dplyr")

library(dplyr)

%>% Operator

- Although not required, the tidyr and dplyr packages make use of the pipe operator %>%
- Key advantage: ability to string multiple functions together by incorporating %>%.
- For instance a function to filter data can be written as:

filter(data, variable == numeric_value)

Same as

data %>% filter(variable == numeric_value)

Dplyr: select( ) function

Objective: Reduce dataframe size to only desired variables for current task

Description: When working with a sizable dataframe, often we desire to only assess specific variables. The select() function allows you to select and/or rename variables.

Function: select(data, ...)

Same as: data %>% select(...)

Arguments:

data: data frame

...: call variables by name or by function

Dplyr: select( ) function

Read the provided “expenditure” data into R

Example: our goal is to only assess the 5 most recent years worth of expenditure data. Applying the select() function we can select only the variables of concern.

```
sub.exp <- expenditures %>% select(Division, State, X2007:X2011)

head(sub.exp)  # for brevity only display first 6 rows
```

	Division	State	X2007	X2008	X2009	X2010	X2011
1	6	Alabama	6245031	6832439	6683843	6670517	6592925
2	9	Alaska	1634316	1918375	2007319	2084019	2201270
3	8	Arizona	7815720	8403221	8726755	8482552	8340211
4	7	Arkansas	3997701	4156368	4240839	4459910	4578136
5	9	California	57352599	61570555	60080929	58248662	57526835
6	8	Colorado	6579053	7338766	7187267	7429302	7409462

Dplyr: select( ) function

We can also apply some of the special functions within select(). For instance we can select all variables that start with ‘X’:

```
head(expenditures %>% select(starts_with("X")))
```

```
   X1980   X1990   X2000   X2001   X2002   X2003   X2004
1 1146713 2275233 4176082 4354794 4444390 4657643 4812479
2  377947  828051 1183499 1229036 1284854 1326226 1354846
3  949753 2258660 4288739 4846105 5395814 5892227 6071785
4  666949 1404545 2380331 2505179 2822877 2923401 3109644
5 9172158 21485782 38129479 42908787 46265544 47983402 49215866
6 1243049 2451833 4401010 4758173 5151003 5551506 5666191
   X2005   X2006   X2007   X2008   X2009   X2010   X2011
1 5164406 5699076 6245031 6832439 6683843 6670517 6592925
2 1442269 1529645 1634316 1918375 2007319 2084019 2201270
3 6579957 7130341 7815720 8403221 8726755 8482552 8340211
4 3546999 3808011 3997701 4156368 4240839 4459910 4578136
5 50918654 53436103 57352599 61570555 60080929 58248662 57526835
6 5994440 6368289 6579053 7338766 7187267 7429302 7409462
>
```

Dplyr: filter( ) function

Objective: Reduce rows/observations with matching conditions

Description: Filtering data is a common task to identify/select observations in which a particular variable matches a specific value/condition. The filter() function provides this capability.

Function: filter(data, ...)

Same as: data %>% filter(...)

Arguments:

data: data frame

...: conditions to be met

Dplyr: filter( ) function

Example: Continuing with the sub.exp dataframe which includes only the recent 5 years worth of expenditures, we can filter by Division:

```
sub.exp %>% filter(Division == 6)
```

```
  Division   State   X2007   X2008   X2009   X2010   X2011
1        6 Alabama 6245031 6832439 6683843 6670517 6592925
```

We can apply multiple logic rules in the filter() function such as:

<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	is NA
<=	Less than or equal to	!is.na	is not NA
>=	Greater than or equal to	&,|,!	Boolean operators

Dplyr: filter( ) function

Filtering by multiple criteria within a single logical expression (starwars is an R dataset)

filter(starwars, hair_color == "none" & eye_color == "black")

```
# A tibble: 9 x 14
  name      height  mass hair_color skin_color     eye_color birth_year sex    gender
  <chr>      <int> <dbl> <chr>      <chr>          <chr>           <dbl> <chr>  <chr>
1 Nien Nu~     160    68 none       grey           black              NA male   mascul~
2 Gasgano      122    NA none       white, blue    black              NA male   mascul~
3 Kit Fis~     196    87 none       green          black              NA male   mascul~
4 Plo Koon     188    80 none       orange         black              22 male   mascul~
5 Lama Su      229    88 none       grey           black              NA male   mascul~
6 Taun We      213    NA none       grey           black              NA female femini~
7 Shaak Ti     178    57 none       red, blue, wh~ black              NA female femini~
8 Tion Me~     206    80 none       grey           black              NA male   mascul~
9 BB8           NA    NA none       none           black              NA none   mascul~
# ... with 5 more variables: homeworldÂ <chr>, speciesÂ <chr>, filmsÂ <list>,
#   vehiclesÂ <list>, starshipsÂ <list>
```

Dplyr: group_by( ) function

Objective: Group data by categorical variables

Description: Often, observations are nested within groups or categories and our goals is to perform statistical analysis both at the observation level and also at the group level.

The group_by() function allows us to create these categorical groupings.

Function: group_by(data, ...)

Same as: data %>% group_by(...)

Dplyr: group_by( ) function

Example: The group_by() function is a silent function. No observable manipulation of the data is performed after applying the function.

Only change: on top of the actual dataframe, an indicator of what variable the data is grouped by will be provided.

The real importance of the group_by() function comes when we perform summary statistics which we will cover shortly.

```
group.exp <- sub.exp %>% group_by(Division)

head(group.exp)
```

```
# A tibble: 6 x 7
# Groups:   Division [4]
  Division State       X2007   X2008   X2009   X2010   X2011
     <int> <fct>       <int>   <int>   <int>   <int>   <int>
1        6 Alabama   6245031 6832439 6683843 6670517 6592925
2        9 Alaska    1634316 1918375 2007319 2084019 2201270
3        8 Arizona   7815720 8403221 8726755 8482552 8340211
4        7 Arkansas  3997701 4156368 4240839 4459910 4578136
5        9 California 57352599 61570555 60080929 58248662 57526835
6        8 Colorado  6579053 7338766 7187267 7429302 7409462
> |
```

Dplyr: summarise( ) function

Objective: Perform summary statistics on variables

Description: key goal of data management is to be able to support statistical analysis on the data.

The summarise() function allows us to perform the majority of the initial summary statistics when performing exploratory data analysis.

Function: summarise(data, ...)

Same as: data %>% summarise(...)

Dplyr: summarise( ) function

Examples

Lets get the mean expenditure value across all states in 2011

```
sub.exp %>% summarise(Mean_2011 = mean(X2011))
```

```
  Mean_2011
1  14441473
```

Dplyr: summarise( ) function

Examples: some more summary stats

```
sub.exp %>% summarise(Min = min(X2011, na.rm=TRUE),
                      Median = median(X2011, na.rm=TRUE),
                      Mean = mean(X2011, na.rm=TRUE),
                      Var = var(X2011, na.rm=TRUE),
                     SD = sd(X2011, na.rm=TRUE),
                     Max = max(X2011, na.rm=TRUE))
```

```
      Min   Median      Mean          Var       SD      Max
1 2201270 7001194 14441473 4.503461e+14 21221360 57526835
>
```

Dplyr: summarise( ) function

Previous slide: useful summaries. Comparison of summary statistics at multiple levels reveals important insights.

This is where the group_by() function comes in.

Group by Division and see how the different regions compared in by 2010 and 2011.

```
sub.exp %>%
    group_by(Division)%>%
    summarise(Mean_2010 = mean(X2010, na.rm=TRUE),
              Mean_2011 = mean(X2011, na.rm=TRUE))
```

```
  Mean_2010 Mean_2011
1  14562494  14441473
```

Dplyr: arrange( ) function

Objective: Order variable values

Description: Often, we desire to view observations in rank order for a particular variable(s). The arrange() function allows us to order data by variables in accending or descending order.

Function: arrange(data, ...)

Same as: data %>% arrange(...)

Arguments:

data: data frame

...: Variable(s) to order

*use desc(x) to sort variable in descending order

Dplyr: arrange( ) function

Examples

- Sort mtcars data by cylinder and displacement

```
mtcars[with(mtcars, order(cyl, disp)), ]
```

- Same result using arrange: no need to use with(), as the context is implicit

NOTE: plyr functions do NOT preserve row.names

```
arrange(mtcars, cyl, disp)
```

- Let's keep the row.names in this example

```
myCars = cbind(vehicle=row.names(mtcars), mtcars)
arrange(myCars, cyl, disp)
```

- Sort with displacement in descending order

```
arrange(myCars, cyl, desc(disp))
```

Dplyr: join( ) functions

Objective: Join two datasets together

Description: Often we have separate dataframes that can have common and differing variables for similar observations and we wish to join these dataframes together.

The multiple xxx_join() functions provide multiple ways to join dataframes.

Dplyr: join( ) functions

Description: Join two datasets

Function:

inner_join(x, y, by = NULL)

left_join(x, y, by = NULL)

right_join(x, y, by = NULL)

full_join(x, y, by = NULL)

semi_join(x, y, by = NULL)

anti_join(x, y, by = NULL)

Arguments:

x,y: data frames to join

by: a character vector of variables to join by. If NULL, the default, join will do a natural join, using all variables with common names across the two tables.

Dplyr: join( ) functions

Examples: Two simple dataframes

Dataframe “x” and ”y”

#Dataframe "x"

```
name <- c("John", "Paul", "George", "Ringo", "Stuart", "Pete")

instrument <- c("guitar", "bass", "guitar", "drums", "bass", "drums")

x <- data.frame(name, instrument)
```

#Dataframe “y”

```
name <- c("John", "Paul", "George", "Ringo", "Brian")

band <- c("TRUE", "TRUE", "TRUE", "TRUE", "FALSE")

y <- data.frame(name, band)
```

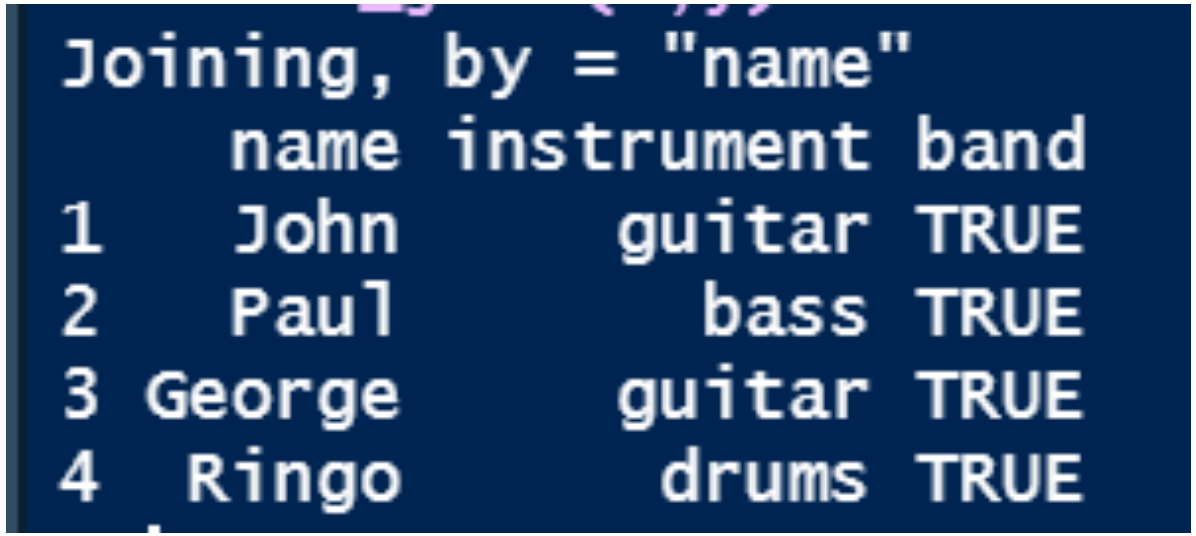
```
> y
    name  band
1   John  TRUE
2   Paul  TRUE
3 George  TRUE
4  Ringo  TRUE
5  Brian FALSE
```

```
> x
    name instrument
1   John     guitar
2   Paul       bass
3 George     guitar
4  Ringo      drums
5 Stuart       bass
6   Pete      drums
```

Dplyr: join( ) functions

inner_join(): Include only rows in both x and y that have a matching value

inner_join(x,y)



```
Joining, by = "name"
    name instrument band
1   John       guitar TRUE
2   Paul         bass TRUE
3 George       guitar TRUE
4  Ringo        drums TRUE
```

Dplyr: join( ) functions

left_join(): Include all of x, and matching rows of y

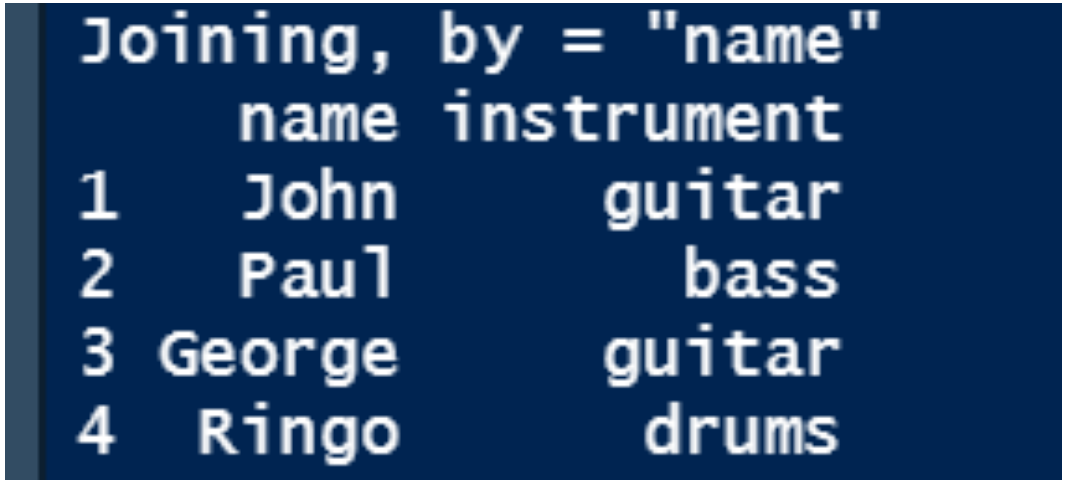
left_join(x,y)

```
Joining, by = "name"
    name instrument band
1   John      guitar TRUE
2   Paul        bass TRUE
3 George      guitar TRUE
4  Ringo       drums TRUE
5 Stuart        bass <NA>
6   Pete       drums <NA>
```

Dplyr: join( ) functions

semi_join(): Include rows of x that match y but only keep the columns from x

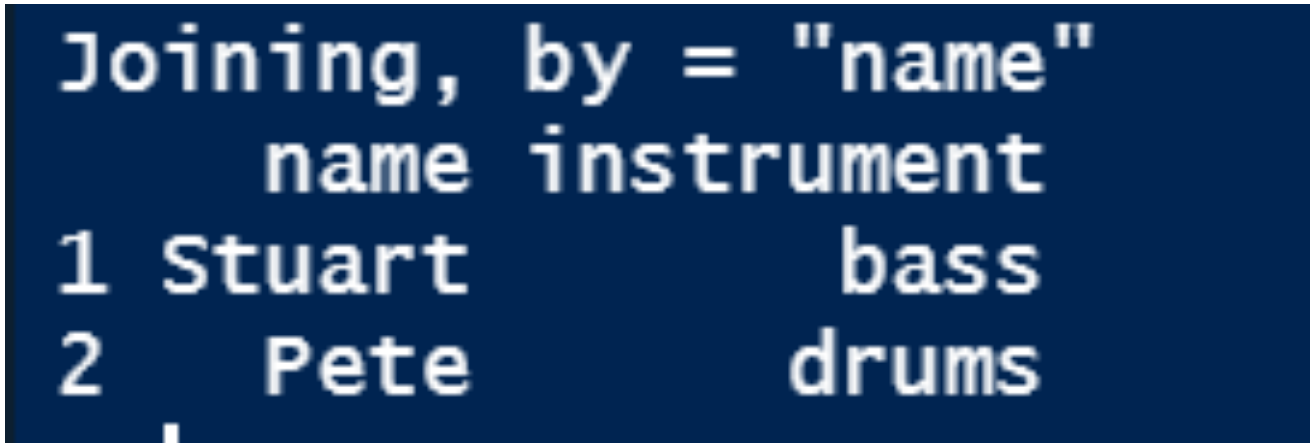
semi_join(x,y)



Dplyr: join( ) functions

anti_join(): Opposite of semi_join

anti_join(x,y)



Thank You