Manipulating And Processing Data in R

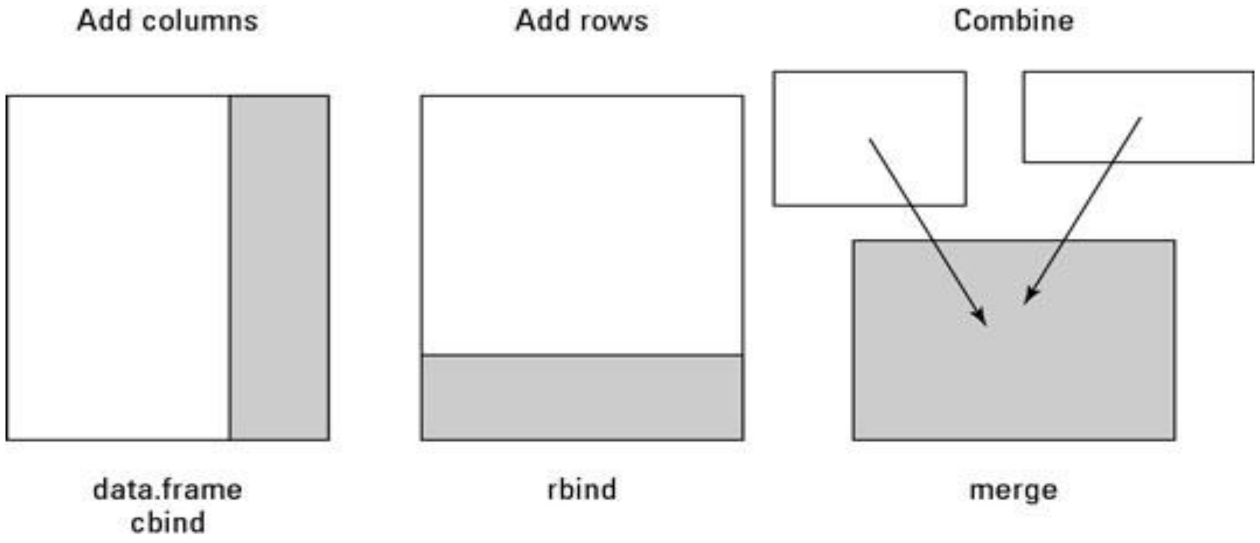
Pavan Kumar A
Senior Project Engineer
Big Data Analytics Team
CDAC-KP

Reshaping Data - Need

- Reshaping data is a general practice in the data analysis and it is very tedious task.
- Data often has multiple levels of grouping and typically requires investigation at multiple levels.
- For example,
  - From a long term clinical study we may be interested in investigating relationships over time, or between times or patients or treatments.
  - Performing these investigations fluently requires the data to be reshaped in different ways, but most software packages make it difficult to generalize these tasks and code needs to be written for each specific case.

Merging Datasets in R

- Similar datasets obtained from the same data sources, need to be merged together for further processing.
- R provides following functions for merging different data sets
  - The merge() function : Used to merge the data contained in different data frames on the basis of common columns
  - The cbind() function: Used to add the columns of datasets having an **equal set and identical order of rows**.
  - The rbind() function: Used to add rows in datasets having equal number of columns



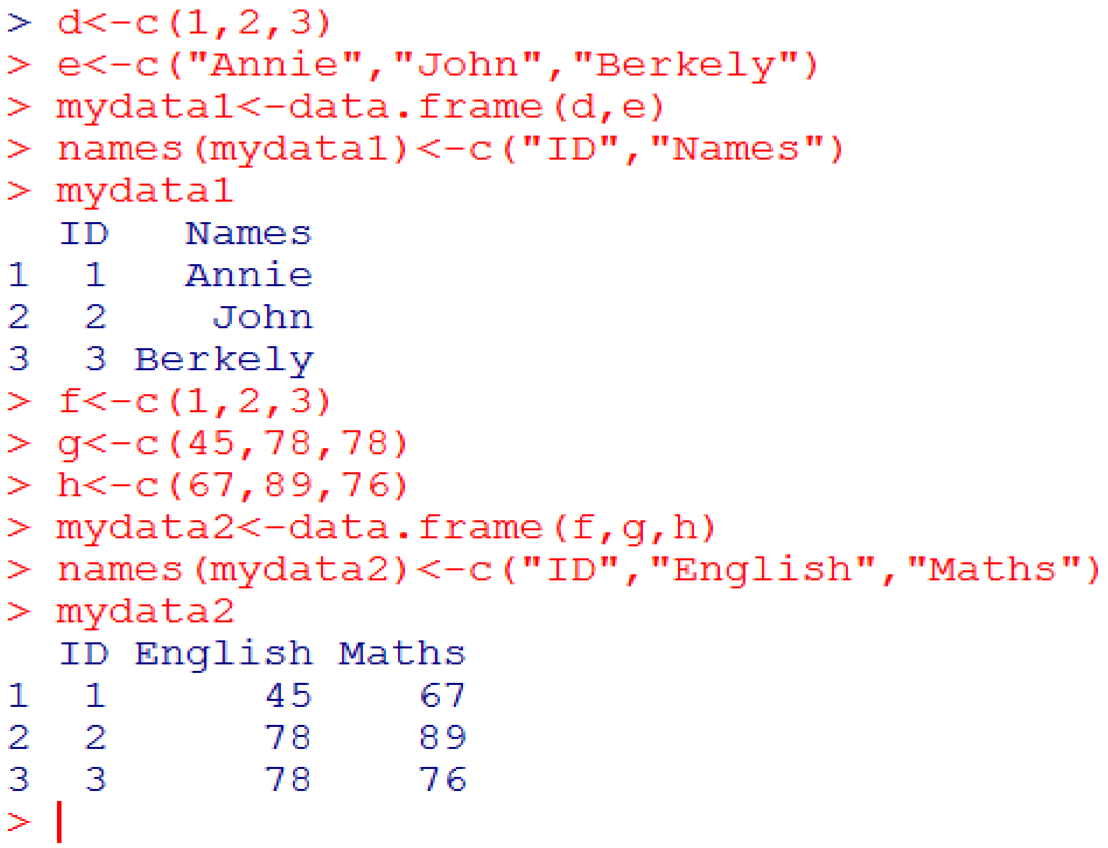
Merging Datasets in R- merge()

- The merge() function combines the data of two data frames on the basis of the existence of a common column between the two.

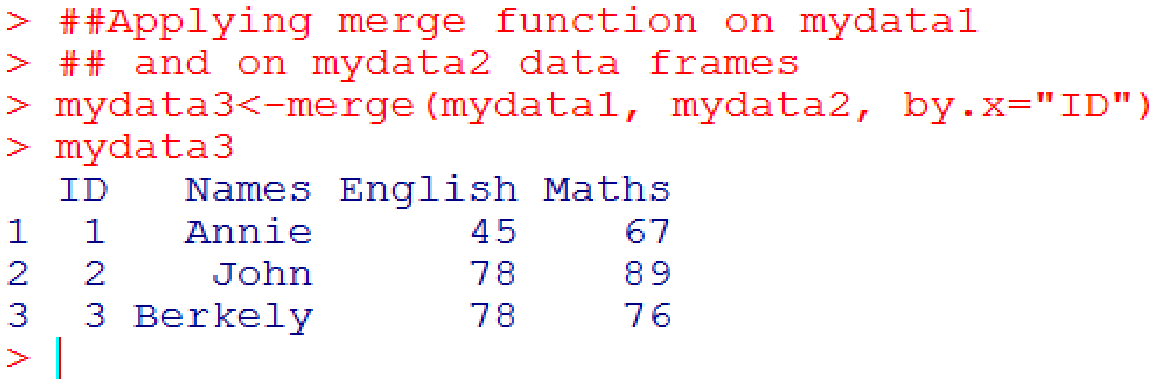
- Following are the arguments taken by merge() funciton
  - x: specifies a data frame
  - y: specifies a data frame
  - by, by.x, by.y: specifies the names of the common columns in both x and y

MERGING DATASETS IN R- MERGE()

- Example merge is shown.
- Data Frames mydata1 and mydata2 are merged based on common column “ID”



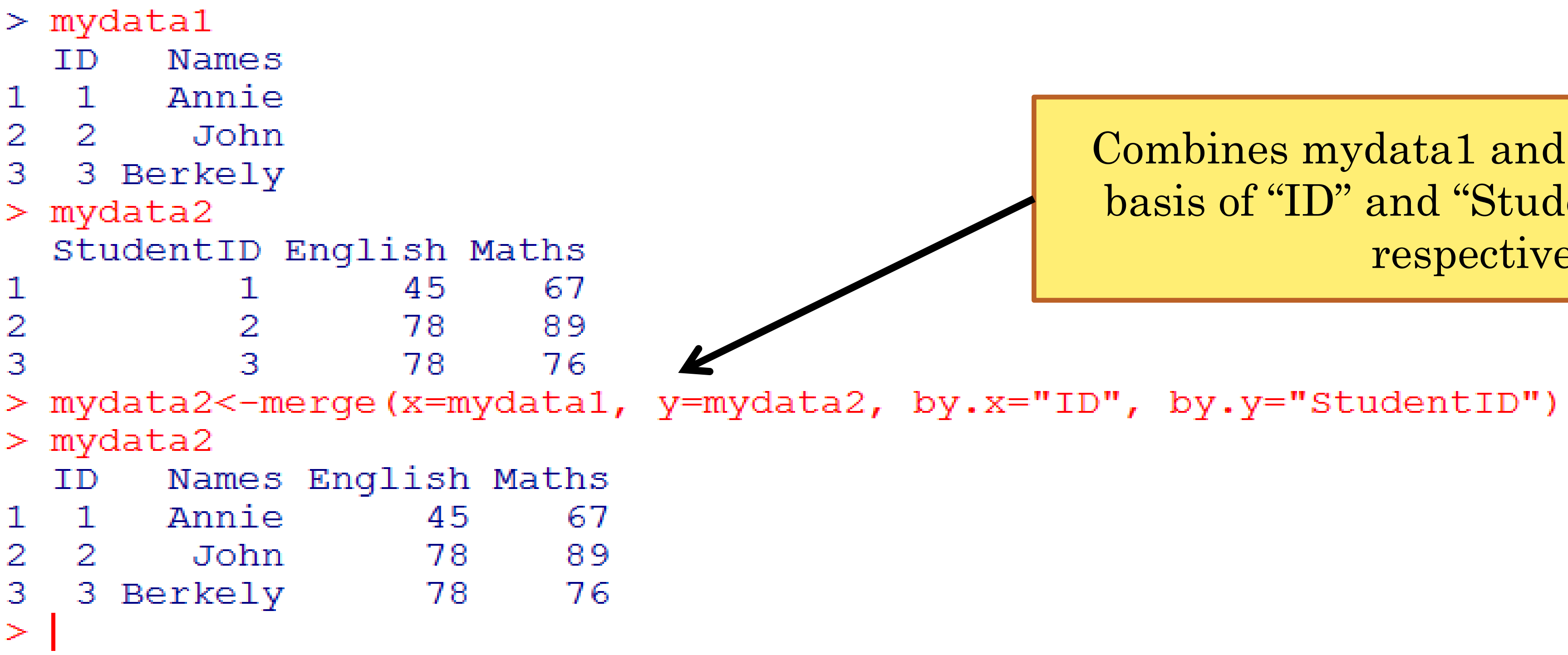
```
> d<-c(1,2,3)
> e<-c("Annie","John","Berkely")
> mydata1<-data.frame(d,e)
> names(mydata1)<-c("ID","Names")
> mydata1
  ID   Names
1  1   Annie
2  2    John
3  3 Berkely
> f<-c(1,2,3)
> g<-c(45,78,78)
> h<-c(67,89,76)
> mydata2<-data.frame(f,g,h)
> names(mydata2)<-c("ID","English","Maths")
> mydata2
  ID English Maths
1  1      45    67
2  2      78    89
3  3      78    76
> 
```



```
> ##Applying merge function on mydata1
> ## and on mydata2 data frames
> mydata3<-merge(mydata1, mydata2, by.x="ID")
> mydata3
  ID   Names English Maths
1  1   Annie      45    67
2  2    John      78    89
3  3 Berkely      78    76
> 
```

Merging Datasets in R- merge()

- Example merge is shown.
- Data Frames mydata1 and mydata2 are merged based on different columns.

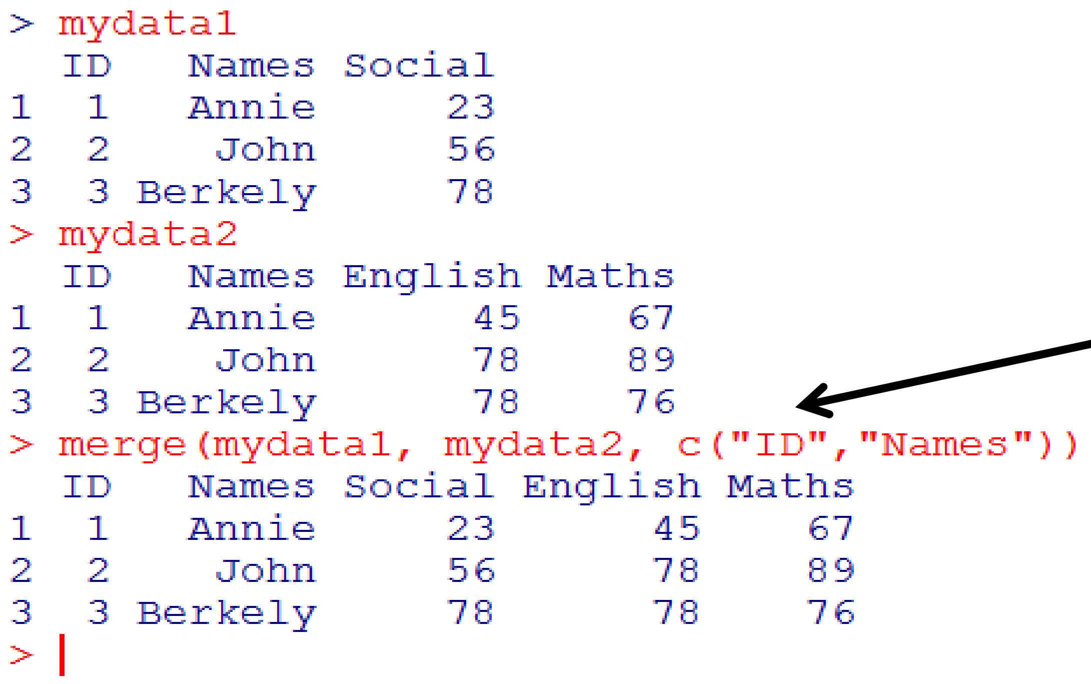


```
> mydata1
  ID   Names
1  1   Annie
2  2    John
3  3 Berkely
> mydata2
  StudentID English Maths
1         1      45    67
2         2      78    89
3         3      78    76
> mydata2<-merge(x=mydata1, y=mydata2, by.x="ID", by.y="StudentID")
> mydata2
  ID   Names English Maths
1  1   Annie      45    67
2  2    John      78    89
3  3 Berkely      78    76
> 
```

Combines mydata1 and mydata2 on the basis of “ID” and “StudentID” columns respectively

Merging Datasets in R- merge()

- Example merge is shown.
- Data Frames mydata1 and mydata2 are merged based on two common columns.

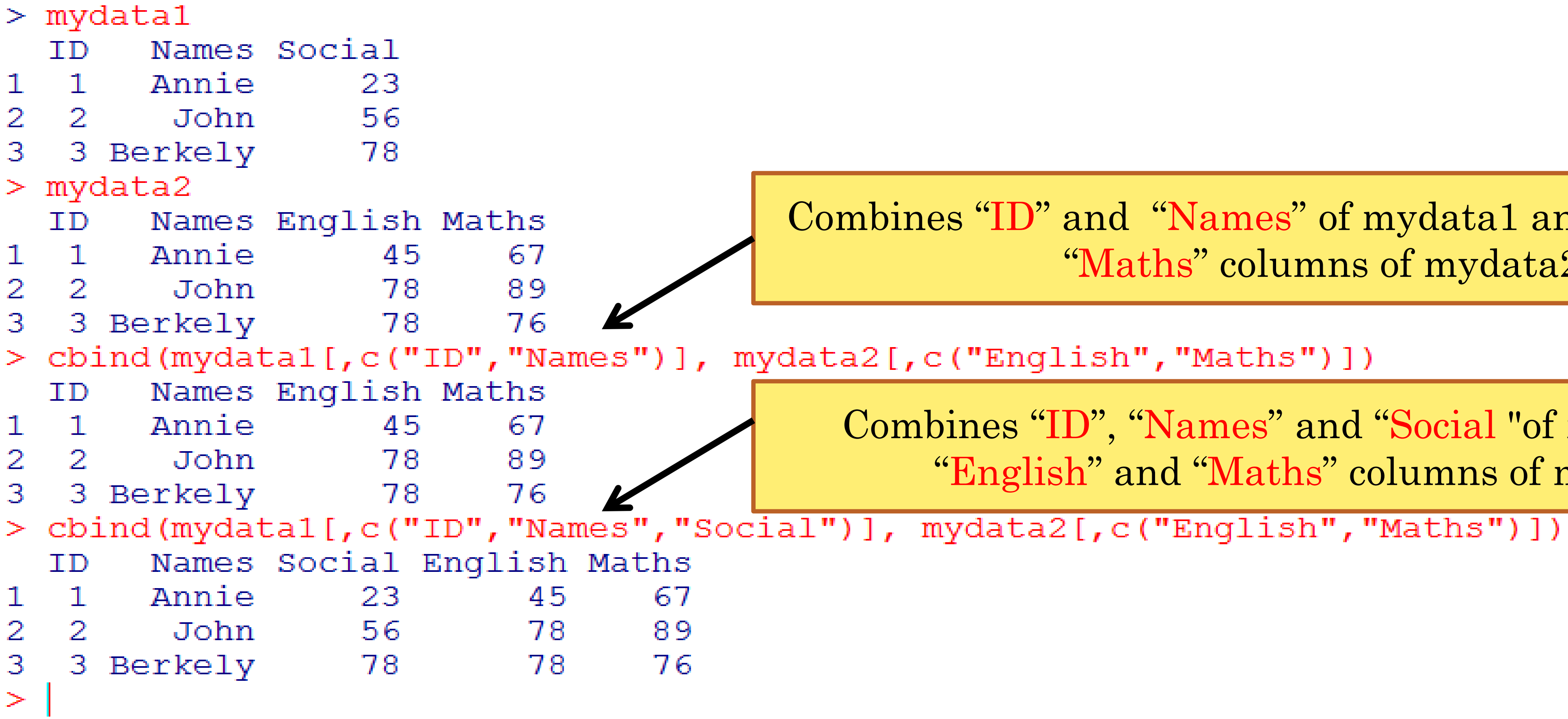


```
> mydata1
  ID   Names Social
1  1   Annie     23
2  2    John     56
3  3 Berkely     78
> mydata2
  ID   Names English Maths
1  1   Annie      45    67
2  2    John      78    89
3  3 Berkely      78    76
> merge(mydata1, mydata2, c("ID","Names"))
  ID   Names Social English Maths
1  1   Annie     23      45    67
2  2    John     56      78    89
3  3 Berkely     78      78    76
> |
```

Combines the data of mydata1 and mydata2 on the basis of "ID" and "Names" columns respectively

Merging Datasets in R- cbind()

- The cbind() function is used to bind the columns of two datasets.
- It helps in restricting the number of columns to be included in the new dataset.



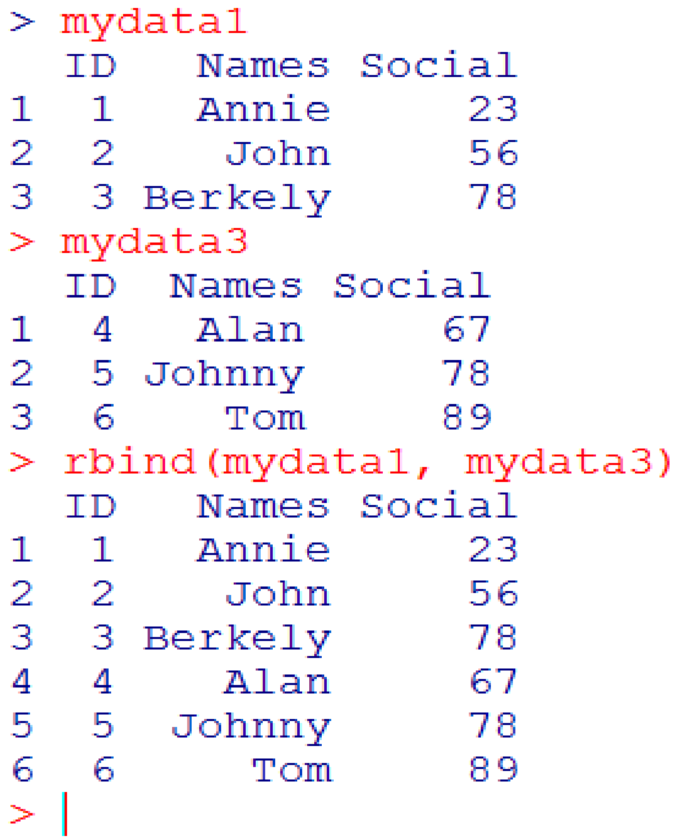
```
> mydata1
  ID    Names Social
1  1    Annie     23
2  2     John     56
3  3 Berkely     78
> mydata2
  ID    Names English Maths
1  1    Annie      45    67
2  2     John      78    89
3  3 Berkely      78    76
> cbind(mydata1[,c("ID","Names")], mydata2[,c("English","Maths")])
  ID    Names English Maths
1  1    Annie      45    67
2  2     John      78    89
3  3 Berkely      78    76
> cbind(mydata1[,c("ID","Names","Social")], mydata2[,c("English","Maths")])
  ID    Names Social English Maths
1  1    Annie     23      45    67
2  2     John     56      78    89
3  3 Berkely     78      78    76
> 
```

Combines “ID” and “Names” of mydata1 and “English” and “Maths” columns of mydata2

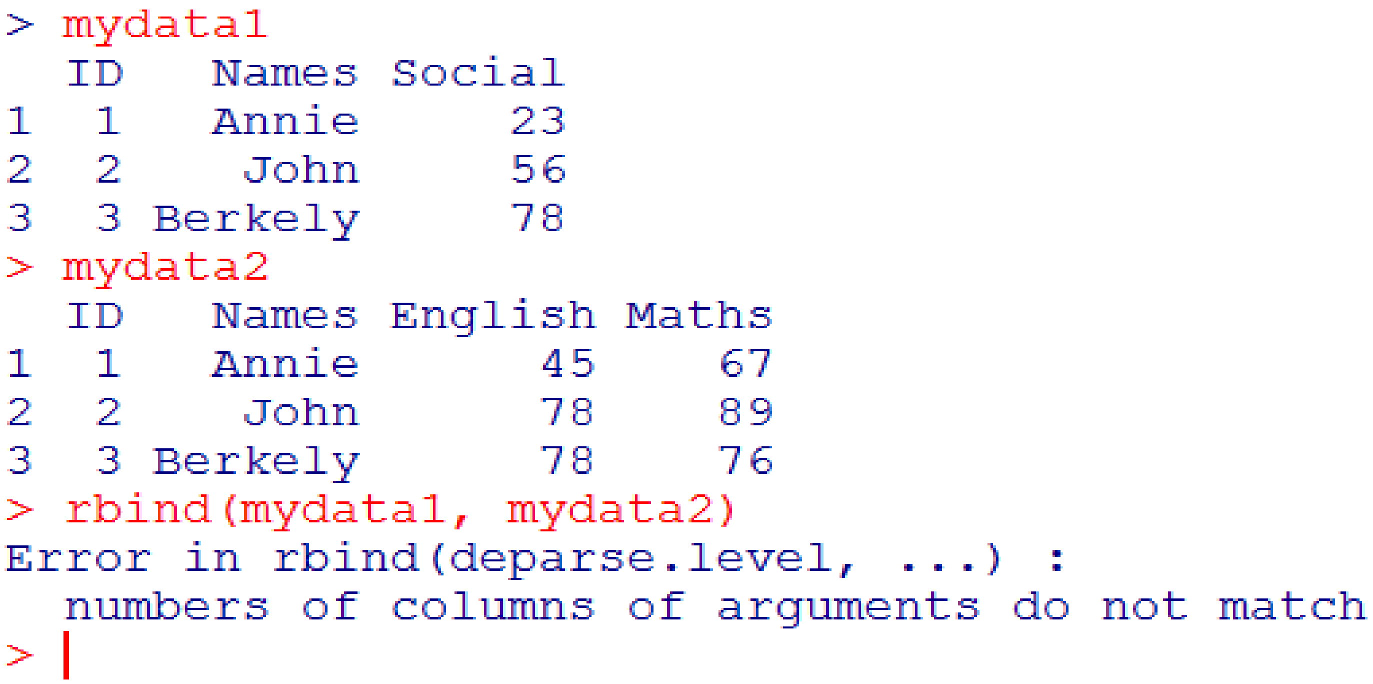
Combines “ID”, “Names” and “Social "of mydata1 and “English” and “Maths” columns of mydata2

MERGING DATASETS IN R- RBIND()

- The rbind() function is used to bind the rows of two datasets.
- The rbind() function combines vector, matrix or data frame by rows.



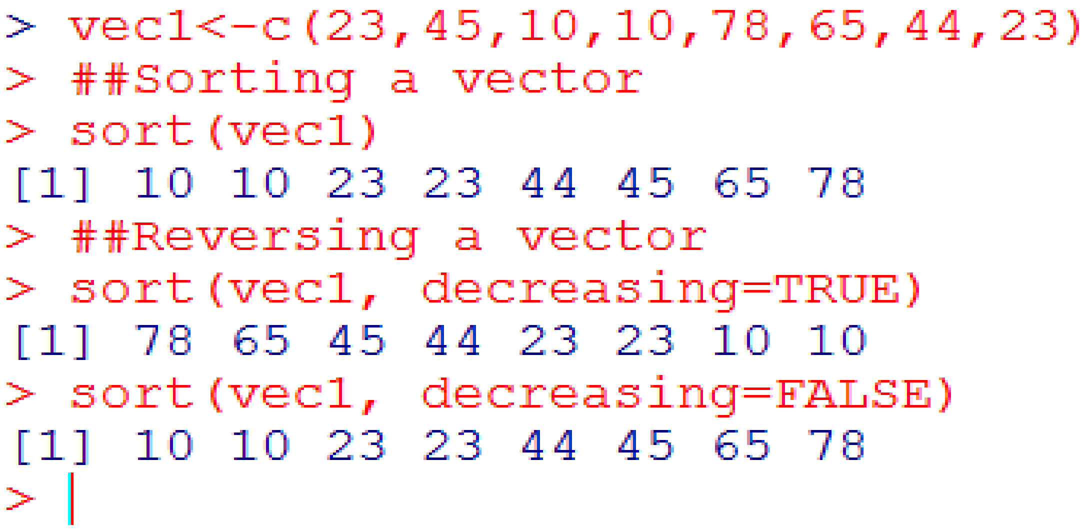
```
> mydata1
  ID   Names Social
1  1   Annie     23
2  2    John     56
3  3 Berkely     78
> mydata3
  ID  Names Social
1  4   Alan     67
2  5 Johnny     78
3  6    Tom     89
> rbind(mydata1, mydata3)
  ID   Names Social
1  1   Annie     23
2  2    John     56
3  3 Berkely     78
4  4    Alan     67
5  5  Johnny     78
6  6     Tom     89
> 
```



```
> mydata1
  ID   Names Social
1  1   Annie     23
2  2    John     56
3  3 Berkely     78
> mydata2
  ID   Names English Maths
1  1   Annie      45    67
2  2    John      78    89
3  3 Berkely      78    76
> rbind(mydata1, mydata2)
Error in rbind(deparse.level, ...) : 
  numbers of columns of arguments do not match
> 
```

Sorting Data

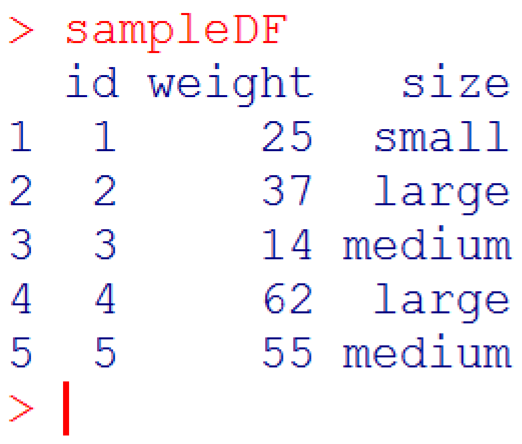
- R provides various functions that allow you to define the order of your data in a data structure.
- The following functions are used for sorting the data.
  - sort() : Used to sort the values contained in a vector
  - order(): Used to organize/arrange values or columns in a dataset
- **Example** : **Sorting and Reverse Sorting of Vector**



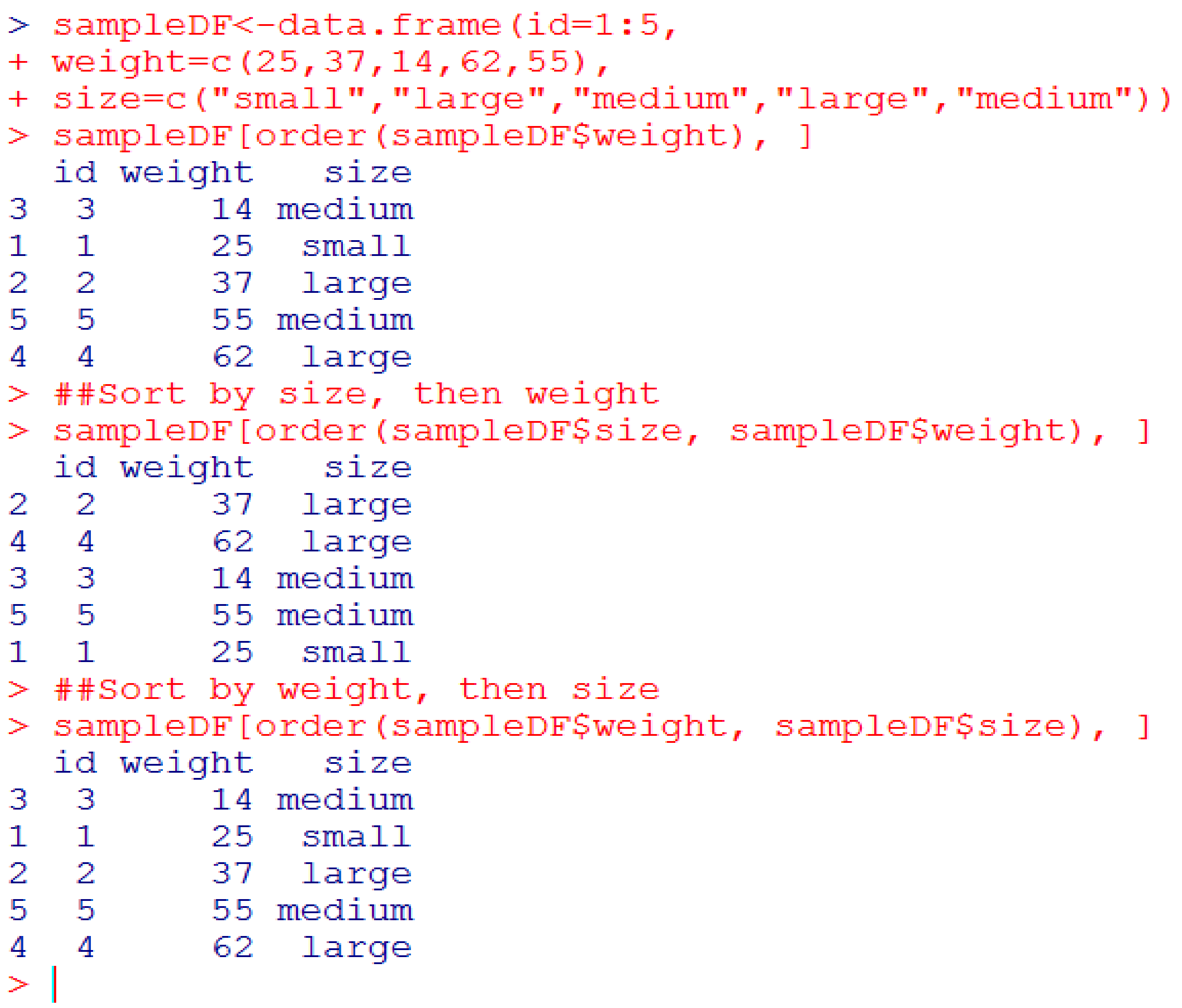
```
> vec1<-c(23,45,10,10,78,65,44,23)
> ##Sorting a vector
> sort(vec1)
[1] 10 10 23 23 44 45 65 78
> ##Reversing a vector
> sort(vec1, decreasing=TRUE)
[1] 78 65 45 44 23 23 10 10
> sort(vec1, decreasing=FALSE)
[1] 10 10 23 23 44 45 65 78
> 
```

ORDERING DATA

- The order() function is used to organize or arrange values or columns in a dataset.



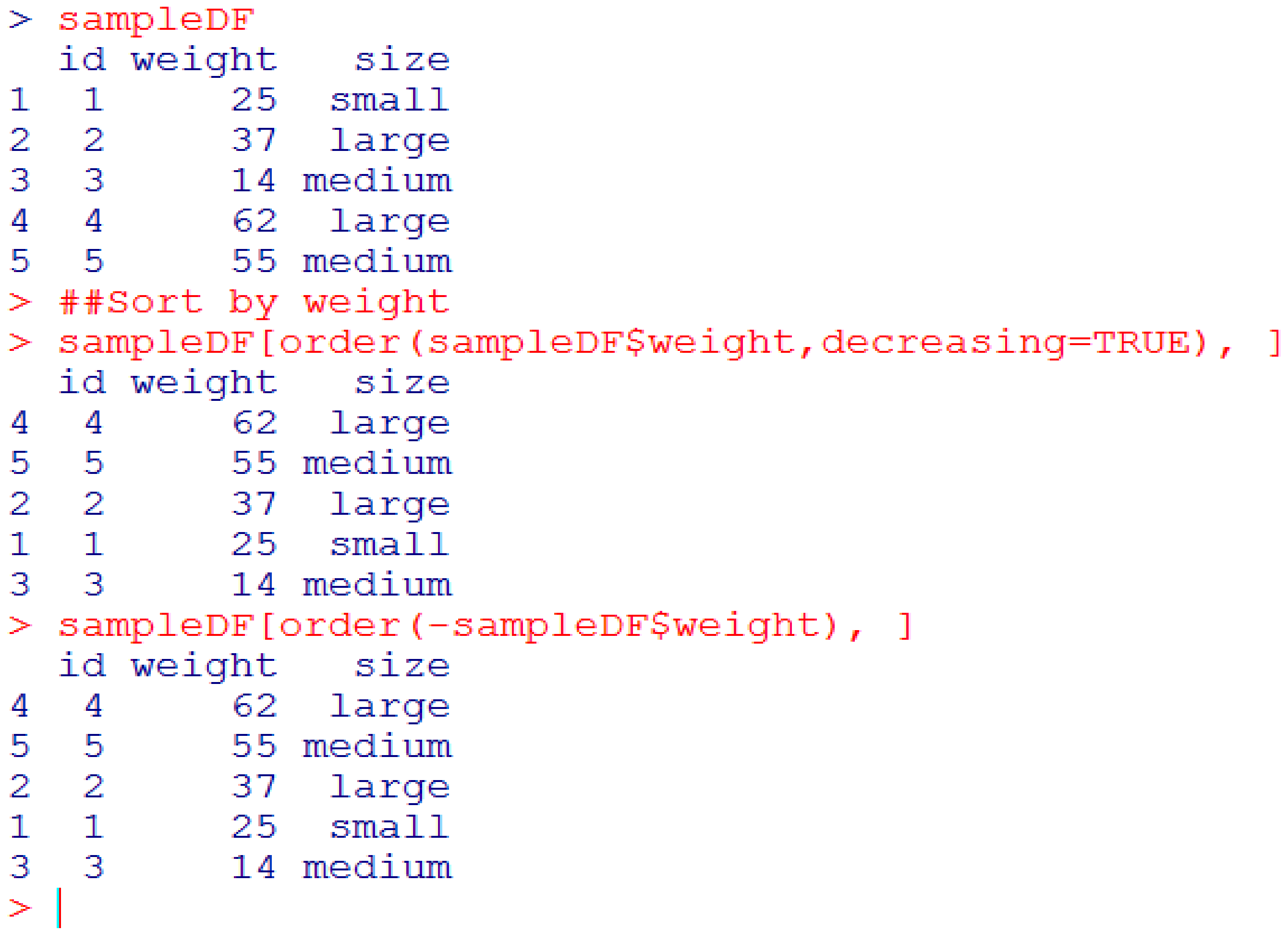
```
> sampleDF
  id weight   size
1  1     25  small
2  2     37  large
3  3     14 medium
4  4     62  large
5  5     55 medium
> 
```



```
> sampleDF<-data.frame(id=1:5,
+ weight=c(25,37,14,62,55),
+ size=c("small","large","medium","large","medium"))
> sampleDF[order(sampleDF$weight), ]
  id weight   size
3  3     14 medium
1  1     25  small
2  2     37  large
5  5     55 medium
4  4     62  large
> ##Sort by size, then weight
> sampleDF[order(sampleDF$size, sampleDF$weight), ]
  id weight   size
2  2     37  large
4  4     62  large
3  3     14 medium
5  5     55 medium
1  1     25  small
> ##Sort by weight, then size
> sampleDF[order(sampleDF$weight, sampleDF$size), ]
  id weight   size
3  3     14 medium
1  1     25  small
2  2     37  large
5  5     55 medium
4  4     62  large
> 
```

Reverse Order

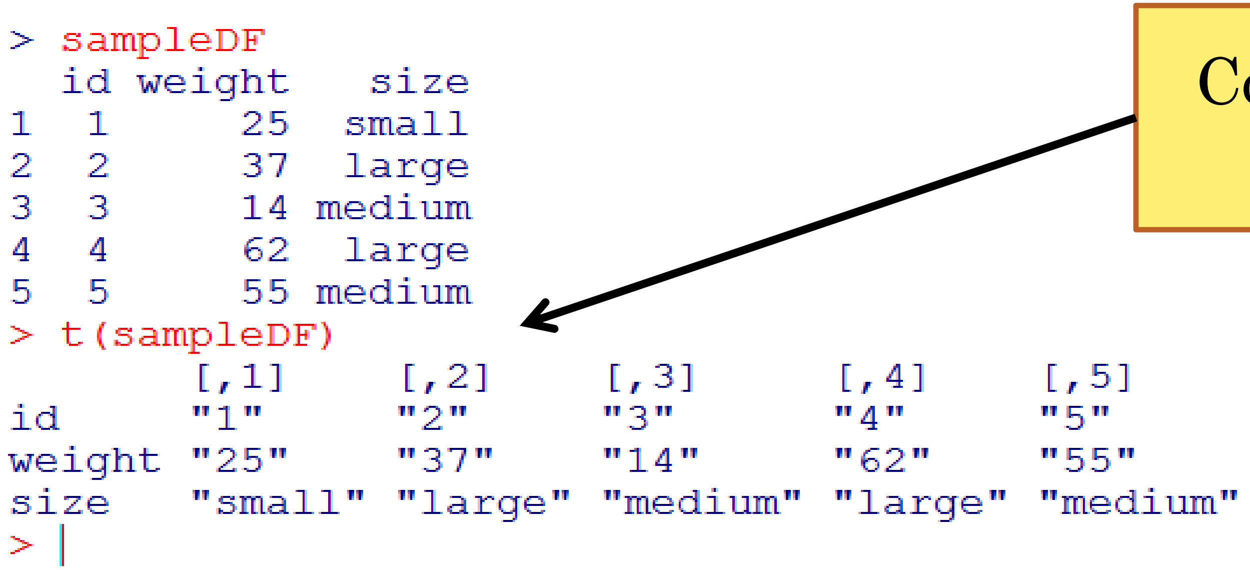
- You can reverse the order of the data contained in a column of a data frame in 2 ways.
  - By using `decreasing=TRUE,` in the order() function
  - By using - (minus) before the column name



```
> sampleDF
  id weight   size
1  1     25  small
2  2     37  large
3  3     14 medium
4  4     62  large
5  5     55 medium
> ##Sort by weight
> sampleDF[order(sampleDF$weight,decreasing=TRUE), ]
  id weight   size
4  4     62  large
5  5     55 medium
2  2     37  large
1  1     25  small
3  3     14 medium
> sampleDF[order(-sampleDF$weight), ]
  id weight   size
4  4     62  large
5  5     55 medium
2  2     37  large
1  1     25  small
3  3     14 medium
> 
```

Transposing The Data

- You can use t() function to transpose a matrix or a data frame
- This function converts rows to columns and columns to rows.



```
> sampleDF
  id weight   size
1  1     25  small
2  2     37  large
3  3     14 medium
4  4     62  large
5  5     55 medium
> t(sampleDF)
       [,1]    [,2]    [,3]     [,4]    [,5]    
id     "1"     "2"     "3"      "4"     "5"     
weight "25"    "37"    "14"     "62"    "55"    
size   "small" "large" "medium" "large" "medium"
> 
```

Converts rows to columns and columns to rows

Data Wrangling Tools

Data Wrangling Tools

- Some of the freely available data wrangling tools are
  - **Tabula :** Extracting **tabular data** from PDF's mainly tables.
  - **OpenRefine :** Tool for working with messy data, cleaning it up, transforming it from one format into another.
  - **"R" packages :** R is a open source programming/scripting language that's useful both for statistics and data science.
  - **DataWrangler:** Data Wrangler is an interactive tool for data cleaning and transformation. It is a web application
  - **CSVkit:** Suite of utilities for converting to and working with **CSV files**
  - **Python:** Pandas package for data cleaning**.**
  - **Mr. Data Converter:** It will convert your Excel data into one of several web-friendly formats, including **HTML**, **JSON** and **XML**.

Data Wrangling Tools

- **Tabula**
  - **Type:** Desktop application
  - **Technology:** Ruby, JavaScript
  - **License:** Open source
  - **Author:** Manuel Aristarán, Mike Tigas and Jeremy B. Merrill
  - **Links:**
    - **Website:** http://tabula.technology/

- A web application that lets you easily extract tabular data/images/text from PDF files.

Data Wrangling Tools

- **Open Refine**
  - **Type:** Desktop application
  - **Technology:** Java
  - **License:** Free
  - **Author:** Google Inc. (United States)
  - **Links:**
    - **Website:** http://code.google.com/p/google-refine/
    - **Documentation for users:** http://code.google.com/p/googlerefine/wiki/DocumentationForUsers
    - **Documentation for developers:** http://code.google.com/p/googlerefine/wiki/DocumentationForDevelopers
    - **Tutorials** https://github.com/OpenRefine/OpenRefine/wiki/External-Resources

Data Wrangling Tools

- **Open Refine**
  - **Input Formats supported:** TSV, CSV, Excel (. xls and xlsx), JSON, XML and Google Data documents.
  - **Output Formats:** TSV, CSV, Excel and in table
  - **Types of Data source:**
    - Upload a file from local system
    - Can provide URL (importing data from tables in web pages, in XML documents)
    - Copy and Paste data
    - Provide link of Google Docs.
  - **Features**
    - Data cleaning, Data transformation, Creation of new fields

Data Wrangling Tools

DataWrangler alpha

- **Data Wrangler**
  - **Type:** Web application
  - **Technology:** HTML
  - **License:** Free to use
  - **Author:** The Stanford Visualization Group (United States)
  - **Links:**
    - **Website:** http://vis.stanford.edu/wrangler/
    - **Research:** http://vis.stanford.edu/papers/wrangler
- **Interactive** web application for transformation and cleaning
- It combines direct manipulation of visualized data with automatic inference of relevant data transformation.

Data Wrangling Tools

csvkit

- **CSVkit**
  - **Type:** Library
  - **Technology:** Python
  - **License:** MIT
  - **Author:** Christopher Groskopf
  - **Links:**
    - **Repository:** https://github.com/onyxfish/csvkit
    - **Issues:** https://github.com/onyxfish/csvkit/issues
    - **Documentation:** http://csvkit.rtfd.org/
    - **Schemas:** https://github.com/onyxfish/ffs
- CSVkit is a suite of utilities for converting to and working with CSV

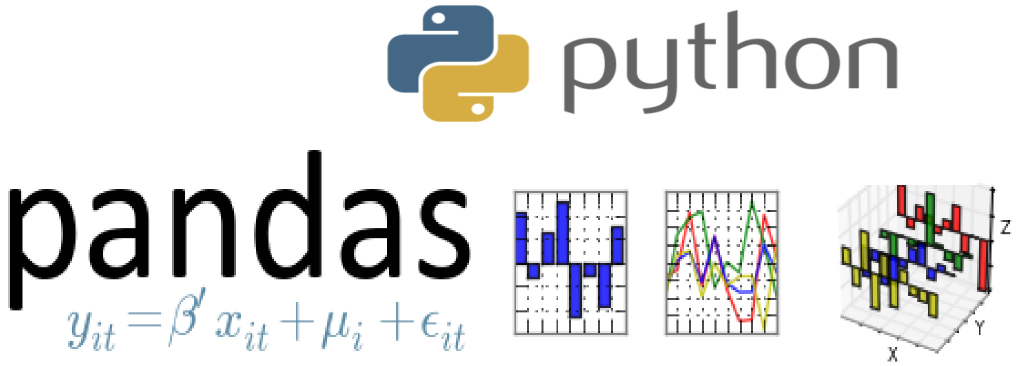
Data Wrangling Tools

Features of CSVkit

- Convert Excel to CSV
- Convert JSON to CSV
- csvcut: data scalpel
- csvstat: statistics on the data
- csvgrep: find the data you need
- csvsort: ordering
- csvjoin: merging related data
- csvstack: combining subsets

Data Wrangling Tools

- **Pandas: Python Data Analysis Library**
  - **Type:** Library
  - **Technology:** Python
  - **License:** Open source
  - **Links:**
    - **Website:** http://pandas.pydata.org/

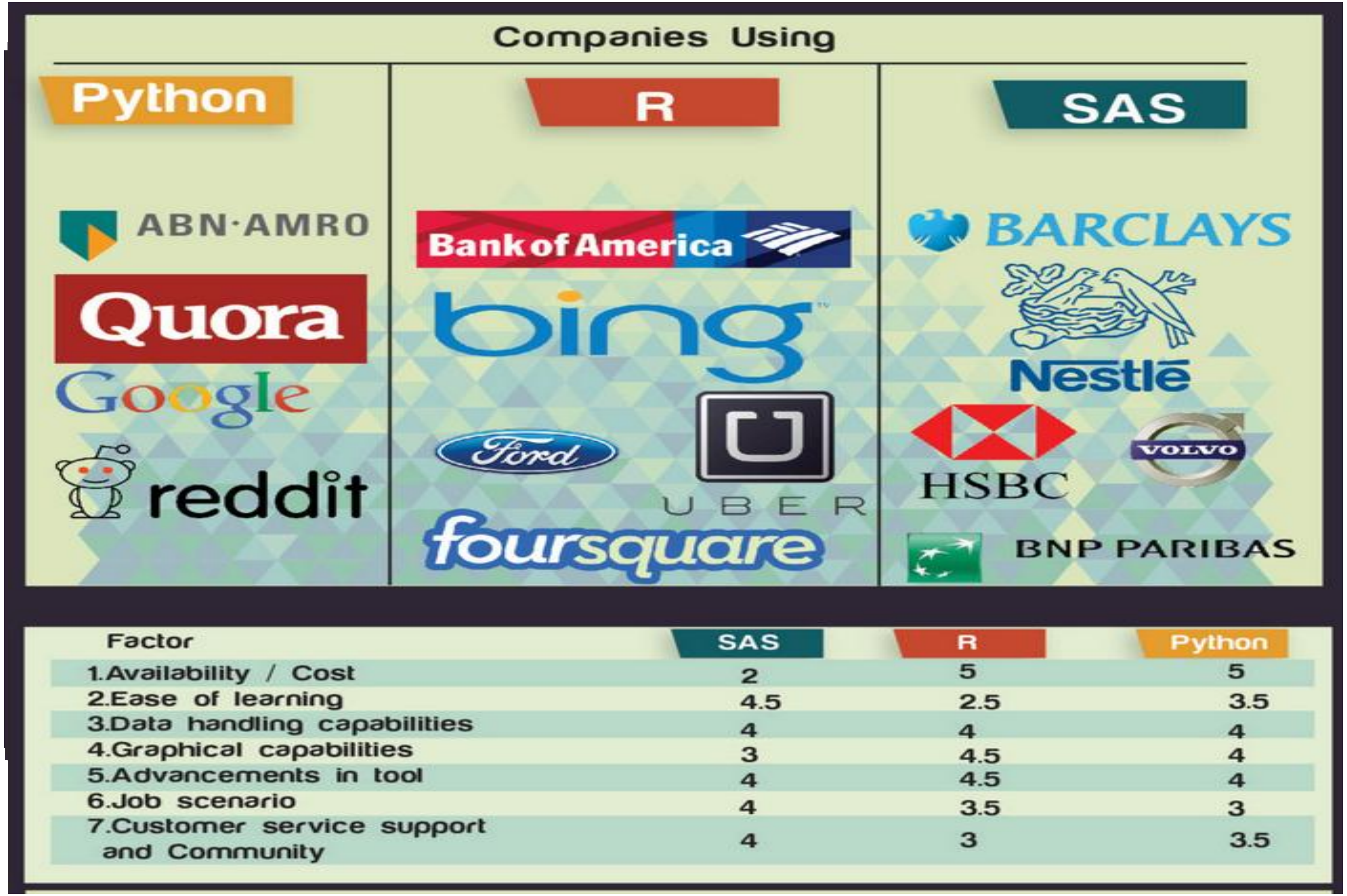


- Python with pandas is in use in a wide variety of academic and commercial domains, including Finance, Neuroscience, Economics, Statistics, Web Analytics, and more.

Data Wrangling Tools

- **Features of Pandas:**
  - Tools for reading and writing data (CSV and text files, Microsoft Excel, SQL databases)
  - merging and joining of data sets;
  - Flexible reshaping and pivoting of data sets;
  - A fast and efficient DataFrame object for data manipulation.
  - Aggregating or transforming data with a powerful group by engine allowing split-apply-combine operations on data sets;

Comparison



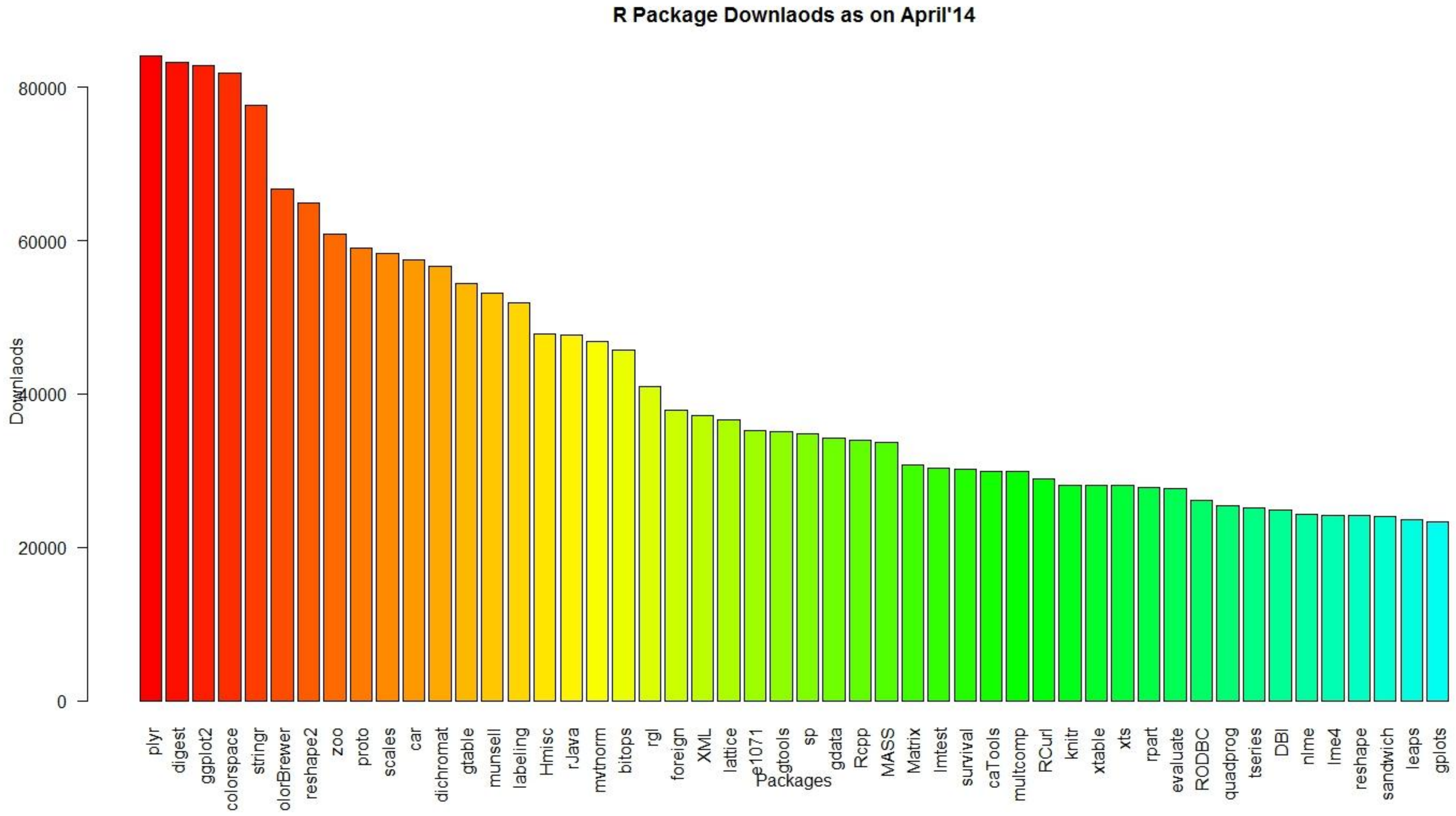
Companies Using

Python	R	SAS
ABN·AMRO	Bank of America	BARCLAYS
Quora	bing	Nestlé
Google	Ford	HSBC
reddit	UBER	VOLVO
	foursquare	BNP PARIBAS

Factor	SAS	R	Python
1.Availability / Cost	2	5	5
2.Ease of learning	4.5	2.5	3.5
3.Data handling capabilities	4	4	4
4.Graphical capabilities	3	4.5	4
5.Advancements in tool	4	4.5	4
6.Job scenario	4	3.5	3
7.Customer service support and Community	4	3	3.5

Source: http://www.analyticsvidhya.com/blog/2015/05/infographic-quick-guide-sas-python/

Why R for Data Wrangling



R Packages for Data Wrangling

- The **sqldf**: R package for running SQL Statements on R data frames
- The **tidyr**: Easily makes Tidy Data with spread() and gather() Functions
- The **plyr** & **dplyr**: The split-apply-combine strategy for R.
- The **reshape2**: For restructure and aggregate data.
- The **Data.table**: Speed with large data sets
- The **Stringr**: Package for text manipulation

- To use the above packages; install and load
- **Installing:**

```
install.packages(“Package_Name”)
```

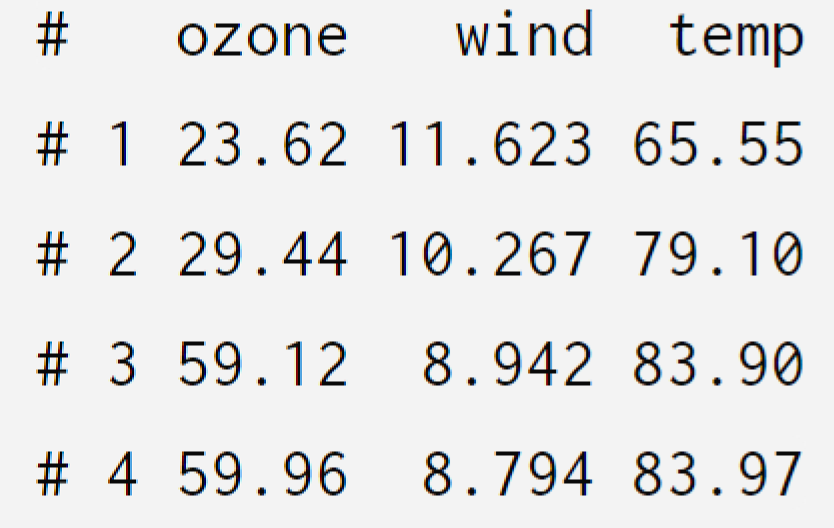
- **Loading**

```
library(Package_Name)
```

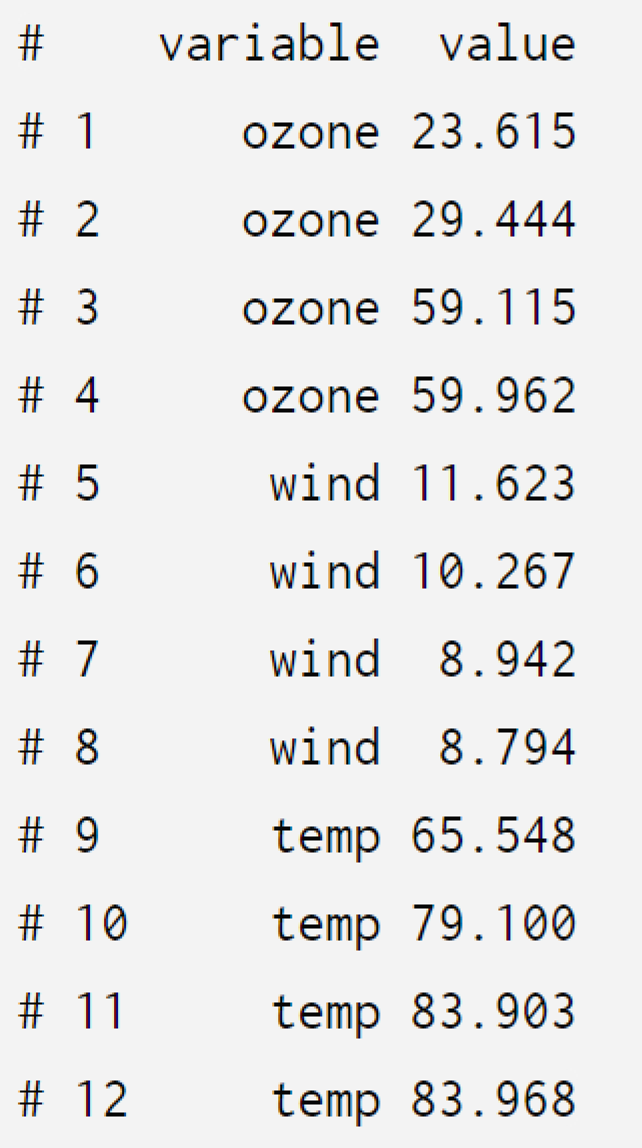
Reshaping The Data In R

Converting Data to Wide or Long Formats

- **Wide and Long data**
  - Wide data has more number of columns than rows
  - Long data has more number of rows than columns
- We can convert from One form to another form in R



```
#   ozone   wind  temp
# 1 23.62 11.623 65.55
# 2 29.44 10.267 79.10
# 3 59.12  8.942 83.90
# 4 59.96  8.794 83.97
```



```
#     variable  value
# 1      ozone 23.615
# 2      ozone 29.444
# 3      ozone 59.115
# 4      ozone 59.962
# 5       wind 11.623
# 6       wind 10.267
# 7       wind  8.942
# 8       wind  8.794
# 9       temp 65.548
# 10      temp 79.100
# 11      temp 83.903
# 12      temp 83.968
```

Converting Data to Wide or Long Formats

- Wide data has a column for each variable.
- Long-format data has a column for all possible variable types and a column for the values of those variables.
  - It is not necessarily 2 columns; it can be more than that
- In some data analysis, you need long data format and vice-versa.
- In reality, you need long-format data much more commonly than wide-format data.
- For example
  - **The ggplot2 requires wide-format data.**
  - **The plyr requires long-format data, and most modelling functions (such as lm(), glm() require long-format data.**
- But people often find it easier to record their data in wide format.

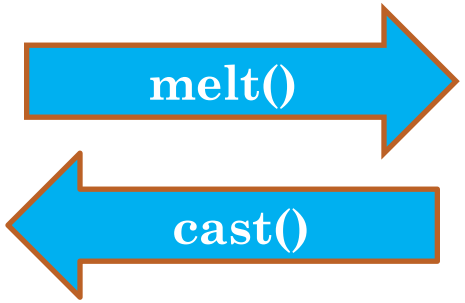
Converting Data to Wide or Long Formats

- R provides the reshape2() package to convert data into wide to long format and vice-versa.
- Two functions we use
  - Use melt() function to convert wide data to long format
  - Use dcast() function to convert long data to wide format
- When converting data from long to wider format, it is important to understand the identifier variables and measured variables.
  - Identifier variables identifies the observations
  - Measured variables represents the observed measurements

Melting Data To Long Format

- The melt() function is used for converting the data from wide format to long format.
- The melt() function contained in reshape2 package.
- So, reshape2 package should be installed and loaded.
- Sample example is shown here.

id	time	x1	x2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4



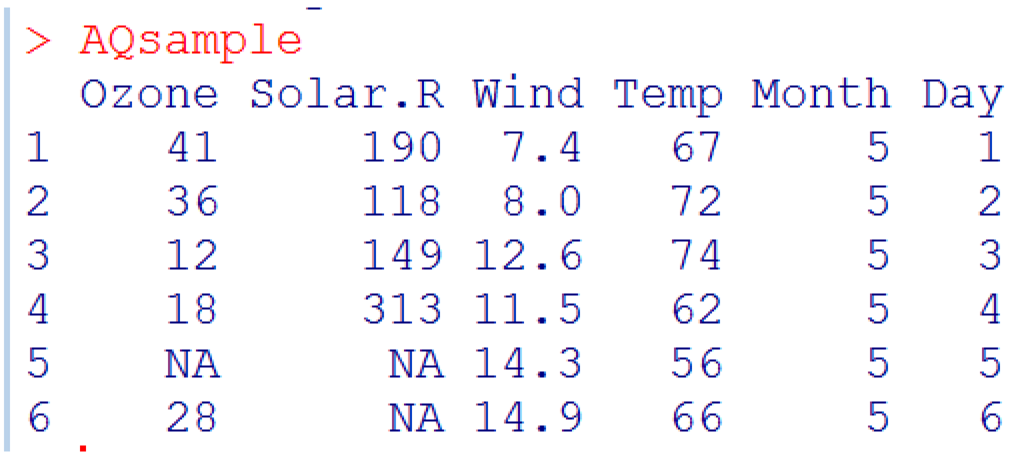
melt()

cast()

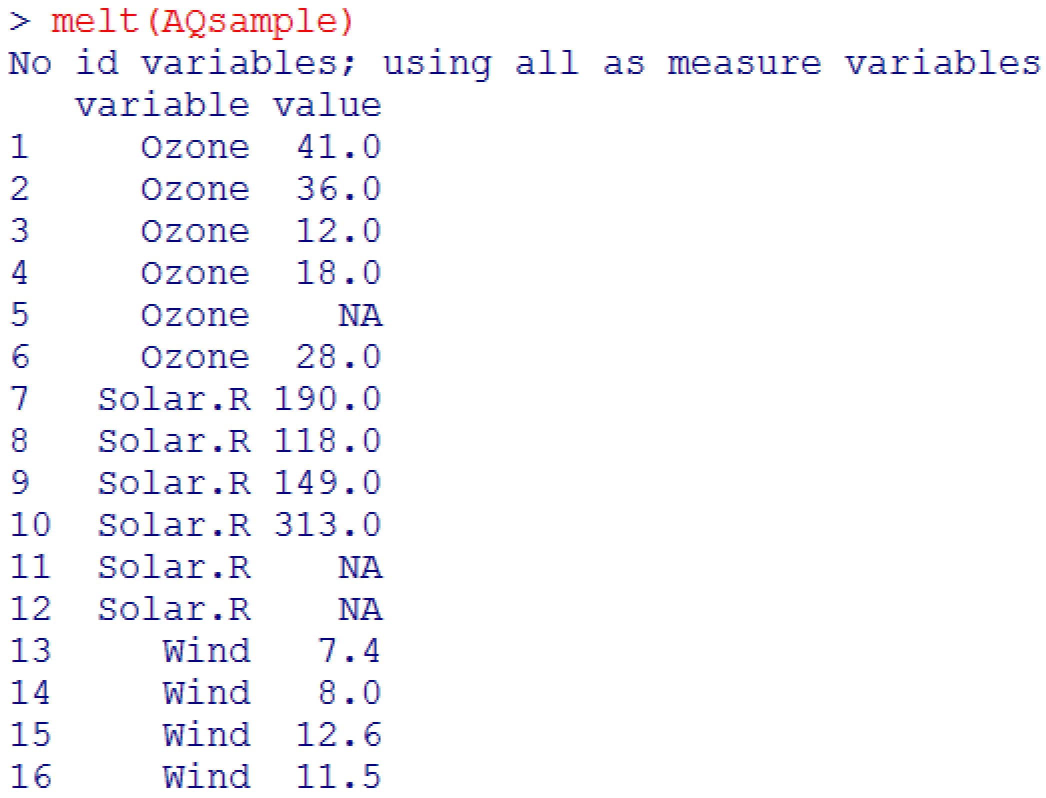
id	time	variable	value
1	1	x1	5
1	2	x1	3
2	1	x1	6
2	2	x1	2
1	1	x2	6
1	2	x2	5
2	1	x2	1
2	2	x2	4

MELTING DATA TO LONG FORMAT

- Example 1 : The `melt()` function.
- We have considered “airquality” dataset
- By default settings for melt funciton
- Command is `melt(AQsample)`



```
> AQsample
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

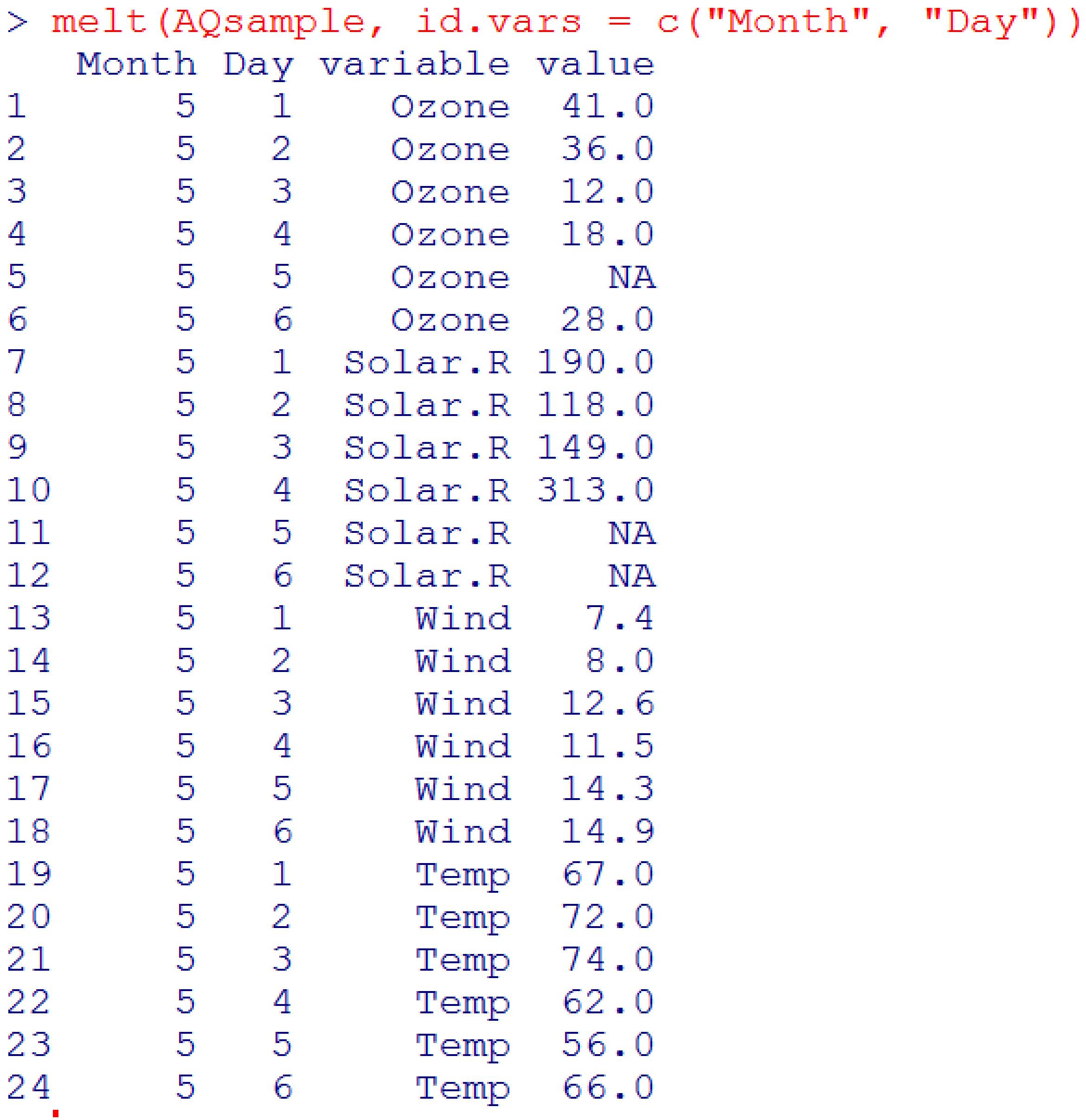


```
> melt(AQsample)
No id variables; using all as measure variables
   variable value
1     Ozone  41.0
2     Ozone  36.0
3     Ozone  12.0
4     Ozone  18.0
5     Ozone    NA
6     Ozone  28.0
7   Solar.R 190.0
8   Solar.R 118.0
9   Solar.R 149.0
10  Solar.R 313.0
11  Solar.R    NA
12  Solar.R    NA
13     Wind   7.4
14     Wind   8.0
15     Wind  12.6
16     Wind  11.5
```

- By default, melt has assumed that all columns with numeric values are variables with values

Melting Data To Long Format

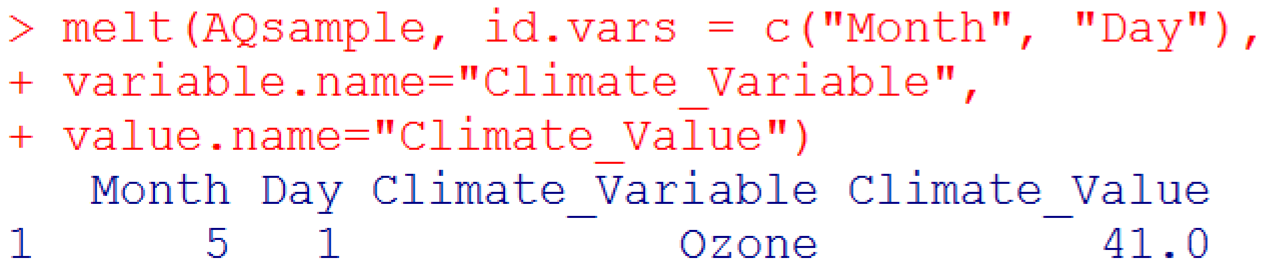
- Example 1: The `melt()` function
- Applying some more arguments
- Based on Identifier variables, the whole dataset is reshaped.
- Here, id.vars are “Month” and “Day” and the remaining variables are treated as measure.vars.
- Data is never lost while reshaping



```
> melt(AQsample, id.vars = c("Month", "Day"))
   Month Day variable value
1      5   1    Ozone  41.0
2      5   2    Ozone  36.0
3      5   3    Ozone  12.0
4      5   4    Ozone  18.0
5      5   5    Ozone    NA
6      5   6    Ozone  28.0
7      5   1  Solar.R 190.0
8      5   2  Solar.R 118.0
9      5   3  Solar.R 149.0
10     5   4  Solar.R 313.0
11     5   5  Solar.R    NA
12     5   6  Solar.R    NA
13     5   1     Wind   7.4
14     5   2     Wind   8.0
15     5   3     Wind  12.6
16     5   4     Wind  11.5
17     5   5     Wind  14.3
18     5   6     Wind  14.9
19     5   1     Temp  67.0
20     5   2     Temp  72.0
21     5   3     Temp  74.0
22     5   4     Temp  62.0
23     5   5     Temp  56.0
24     5   6     Temp  66.0
```

Melting Data To Long Format

- Example 1: The `melt()` function.
- If you want to change the default names of “variable” and “value”, following command is used.



```
> melt(AQsample, id.vars = c("Month", "Day"),
+ variable.name="Climate_Variable",
+ value.name="Climate_Value")
  Month Day Climate_Variable Climate_Value
1     5   1            Ozone          41.0
```

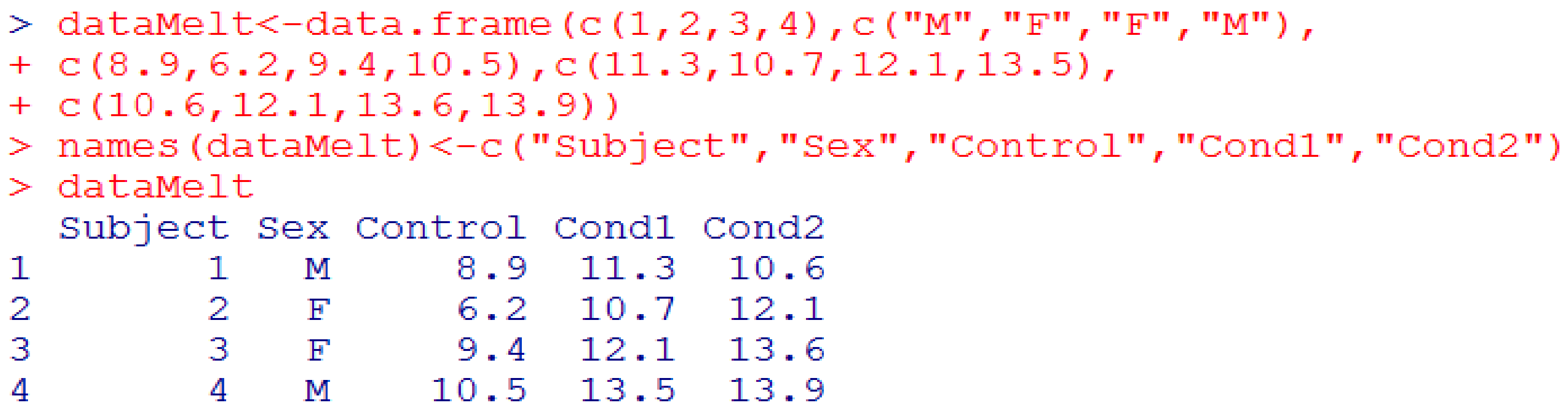
- **Syntax**

melt(data, id.vars, measure.vars, variable.name = "variable", na.rm = FALSE, value.name = "value”)

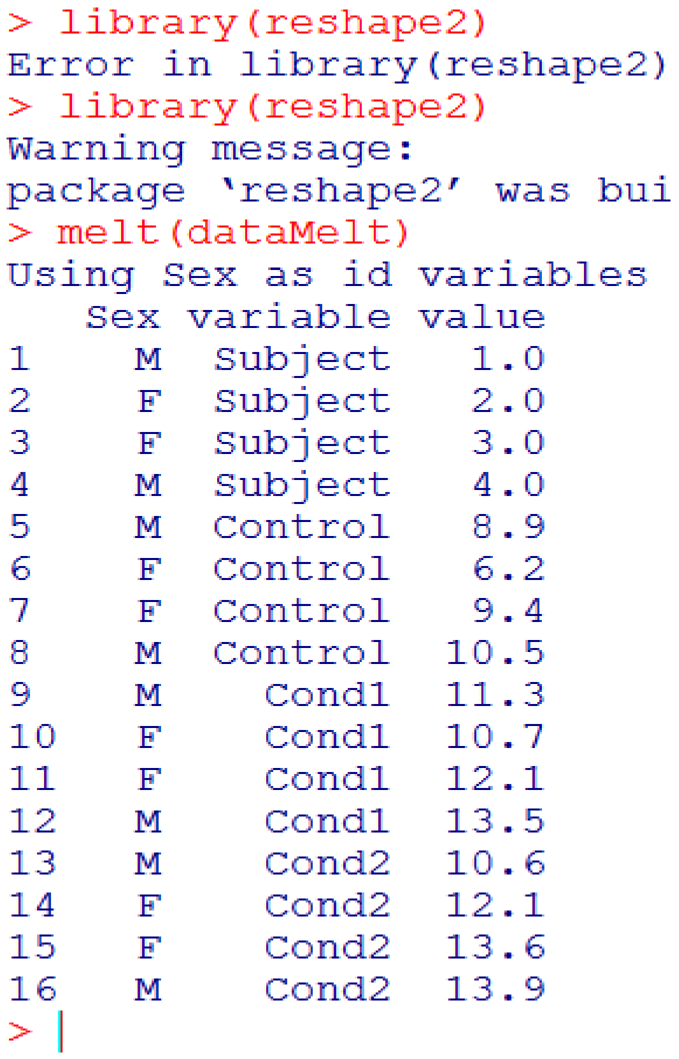
Melting Data To Long Format

- **Example 2** : The `melt()` function

1. Here, data frame, named dataMelt is created.
2. Here, `melt()` function, explicitly specifies ID variables, source columns, destination columns and the measurement column



```
> dataMelt<-data.frame(c(1,2,3,4),c("M","F","F","M"),
+ c(8.9,6.2,9.4,10.5),c(11.3,10.7,12.1,13.5),
+ c(10.6,12.1,13.6,13.9))
> names(dataMelt)<-c("Subject","Sex","Control","Cond1","Cond2")
> dataMelt
  Subject Sex Control Cond1 Cond2
1       1   M     8.9  11.3  10.6
2       2   F     6.2  10.7  12.1
3       3   F     9.4  12.1  13.6
4       4   M    10.5  13.5  13.9
```

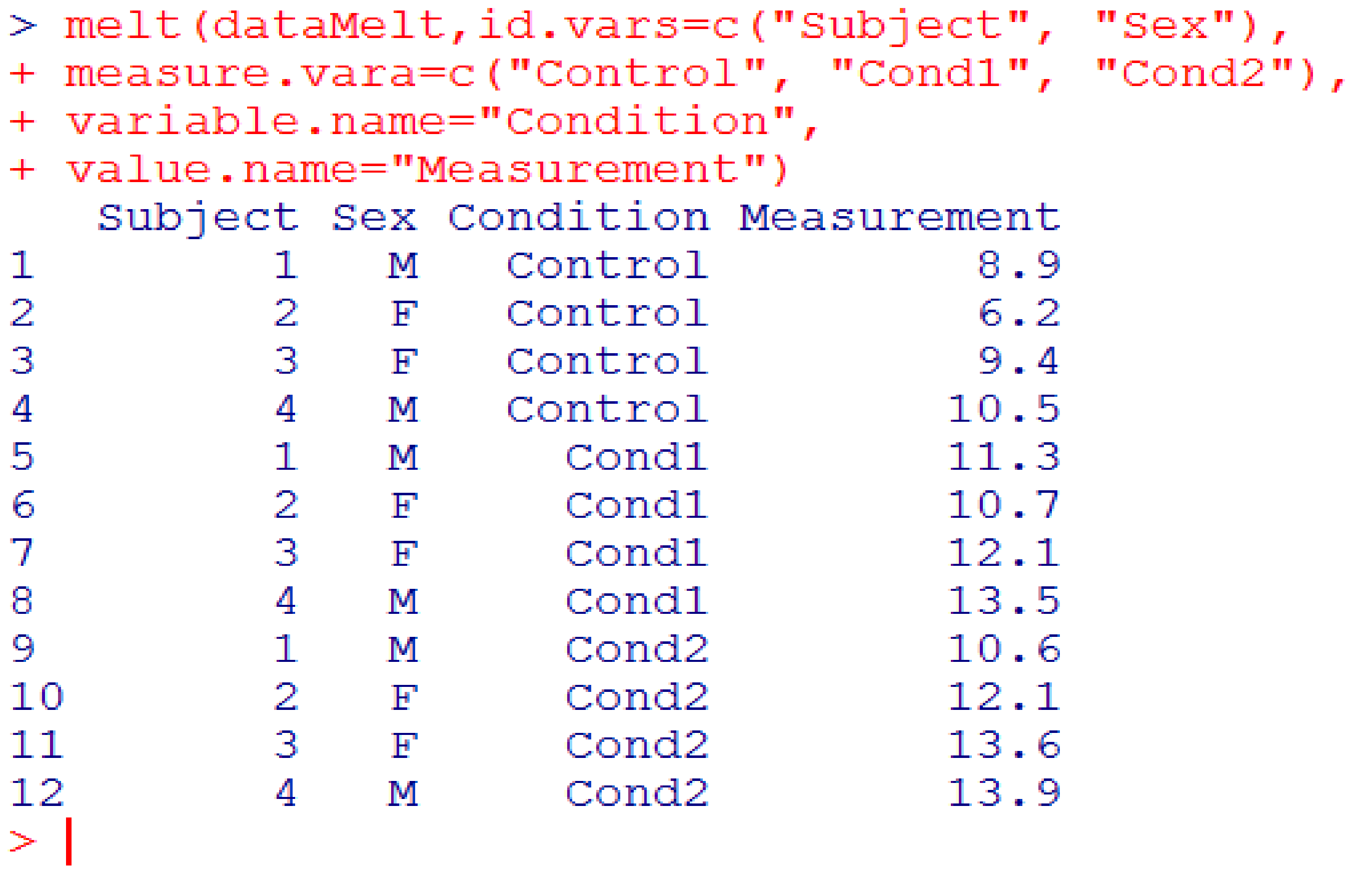


```
> library(reshape2)
Error in library(reshape2)
> library(reshape2)
Warning message:
package 'reshape2' was bui
> melt(dataMelt)
Using Sex as id variables
   Sex variable value
1    M  Subject   1.0
2    F  Subject   2.0
3    F  Subject   3.0
4    M  Subject   4.0
5    M  Control   8.9
6    F  Control   6.2
7    F  Control   9.4
8    M  Control  10.5
9    M    Cond1  11.3
10   F    Cond1  10.7
11   F    Cond1  12.1
12   M    Cond1  13.5
13   M    Cond2  10.6
14   F    Cond2  12.1
15   F    Cond2  13.6
16   M    Cond2  13.9
> 
```

Melting Data To Long Format

- The `melt()` function, by default, considers all categorical variables into identifier variables.
- We can also change the default settings

Here, we are applying additional parameters to specify the identifiers, Measurement Variable & Value names



```
> melt(dataMelt,id.vars=c("Subject", "Sex"),
+ measure.vara=c("Control", "Cond1", "Cond2"),
+ variable.name="Condition",
+ value.name="Measurement")
   Subject Sex Condition Measurement
1        1   M   Control         8.9
2        2   F   Control         6.2
3        3   F   Control         9.4
4        4   M   Control        10.5
5        1   M     Cond1        11.3
6        2   F     Cond1        10.7
7        3   F     Cond1        12.1
8        4   M     Cond1        13.5
9        1   M     Cond2        10.6
10       2   F     Cond2        12.1
11       3   F     Cond2        13.6
12       4   M     Cond2        13.9
> 
```

Casting Data To Wide Format

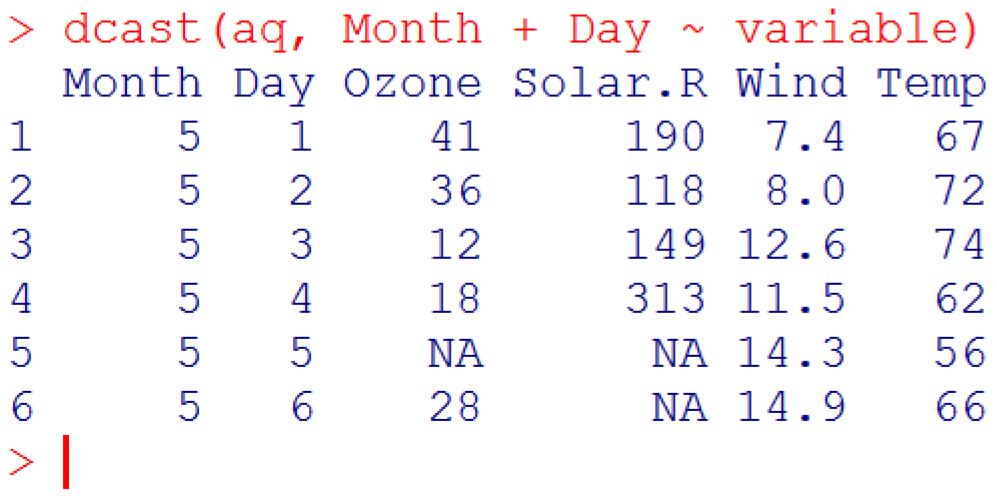
- In reshape2 there are multiple cast functions.
  - Since you will most commonly work with data.frame objects, the dcast() function is used here.
  - There is also acast() to return a vector, matrix, or array.
- The dcast() fucntion uses a formula to describe the shape of the data.
- The arguments on the left side of the formula refers to the "id.vars" and the arguments on the right side of the formula refers to the "measure.vars".
- Here, we are using long data format of Airquality dataset

Casting Data To Wide Format

- Exampe 1: The `dcast()` function
- Dataset used : Long data format of Airquality dataset.
- Here, we need to dcast the “Month” and “Day” (which are again id.vars) and remaining are variable is the measures.vars.

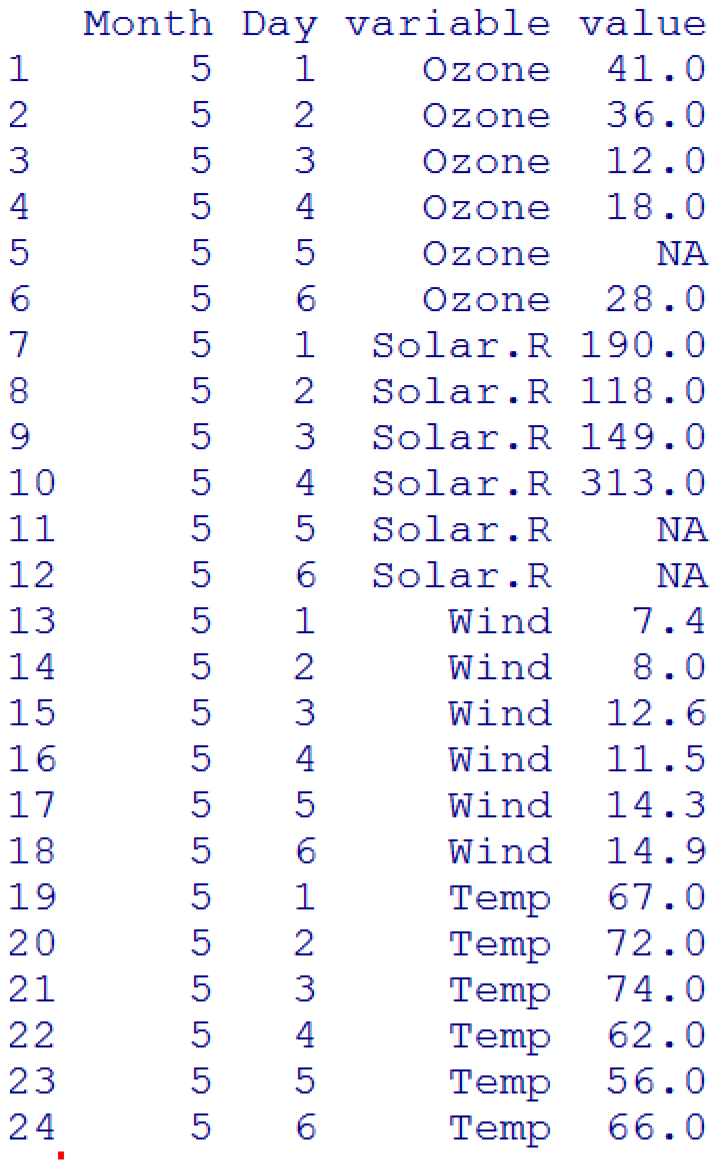
Casting Data To Wide Format

- Exampe 1: The `dcast()` function



```
> dcast(aq, Month + Day ~ variable)
  Month Day Ozone Solar.R Wind Temp
1     5   1    41     190  7.4   67
2     5   2    36     118  8.0   72
3     5   3    12     149 12.6   74
4     5   4    18     313 11.5   62
5     5   5    NA      NA 14.3   56
6     5   6    28      NA 14.9   66
> 
```

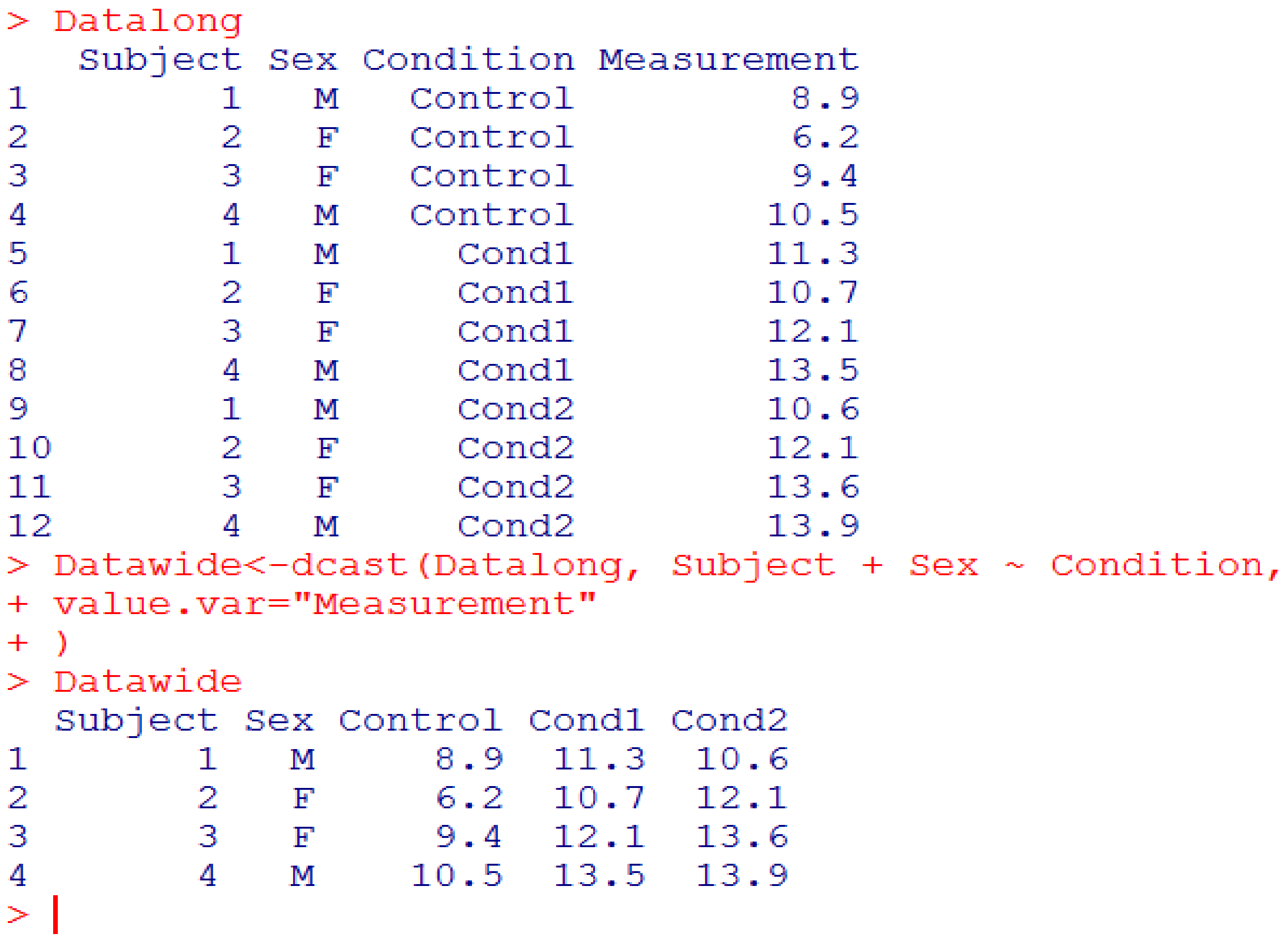
- Check with the following formula month ~ variable



```
   Month Day variable value
1      5   1    Ozone  41.0
2      5   2    Ozone  36.0
3      5   3    Ozone  12.0
4      5   4    Ozone  18.0
5      5   5    Ozone    NA
6      5   6    Ozone  28.0
7      5   1  Solar.R 190.0
8      5   2  Solar.R 118.0
9      5   3  Solar.R 149.0
10     5   4  Solar.R 313.0
11     5   5  Solar.R    NA
12     5   6  Solar.R    NA
13     5   1     Wind   7.4
14     5   2     Wind   8.0
15     5   3     Wind  12.6
16     5   4     Wind  11.5
17     5   5     Wind  14.3
18     5   6     Wind  14.9
19     5   1     Temp  67.0
20     5   2     Temp  72.0
21     5   3     Temp  74.0
22     5   4     Temp  62.0
23     5   5     Temp  56.0
24     5   6     Temp  66.0
```

Casting Data To Wide Format

- Example 2 : Sample dataset
- Formula here is `Subject + Sex ~ Condition`
- The id.vars are Subject and Sex
- The measure.vars are Condition



```
> Datalong
   Subject Sex Condition Measurement
1        1   M   Control         8.9
2        2   F   Control         6.2
3        3   F   Control         9.4
4        4   M   Control        10.5
5        1   M     Cond1        11.3
6        2   F     Cond1        10.7
7        3   F     Cond1        12.1
8        4   M     Cond1        13.5
9        1   M     Cond2        10.6
10       2   F     Cond2        12.1
11       3   F     Cond2        13.6
12       4   M     Cond2        13.9
> Datawide<-dcast(Datalong, Subject + Sex ~ Condition,
+ value.var="Measurement"
+ )
> Datawide
  Subject Sex Control Cond1 Cond2
1       1   M     8.9  11.3  10.6
2       2   F     6.2  10.7  12.1
3       3   F     9.4  12.1  13.6
4       4   M    10.5  13.5  13.9
> |
```

The tidyr package

- The tidyr is new package that makes it easy to “tidy” your data
- **Main Features (Fucntions)**
  - Gather and Spread
  - Unite and Separate
- **To install**

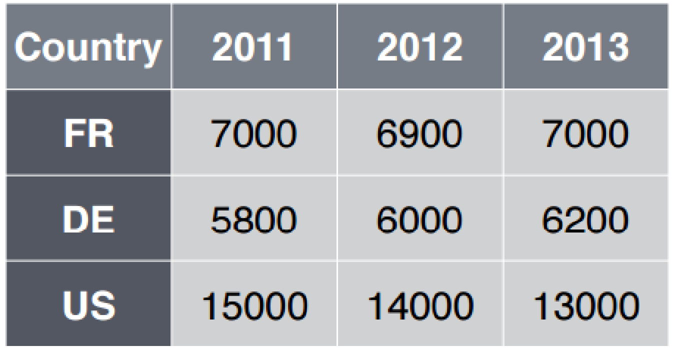
```
Install.packages(“tidyr”)
```

- **To load**

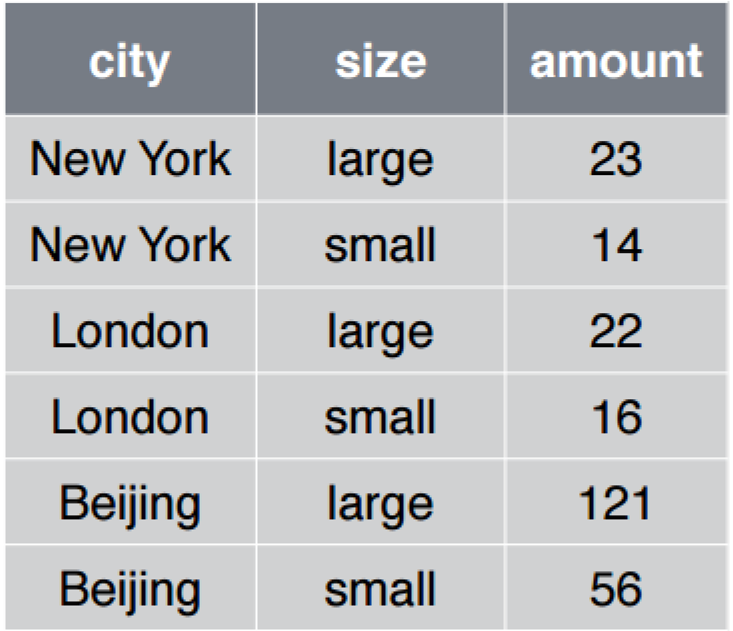
```
librarty(“tidyr”)
```

- **Help**

```
help(package=“tidyr")
```



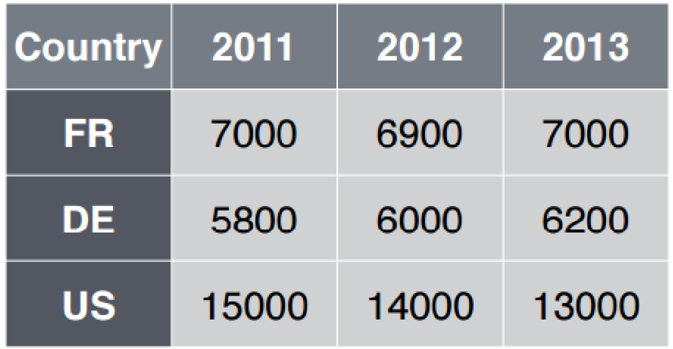
Country	2011	2012	2013
FR	7000	6900	7000
DE	5800	6000	6200
US	15000	14000	13000



city	size	amount
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

The tidyr package

- **The gather() function**
- The gather() function takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed.
- The gather() function can be used when the columns are not variables.
- Example :
  - Dataset used is TB data. Number of TB cases in 3 different countries
- Here, 3 rows and 4 columns.
- Column names [2:4] are simple numbers.
- So, we can apply gather to these columns under one column (For example : Year)



Country	2011	2012	2013
FR	7000	6900	7000
DE	5800	6000	6200
US	15000	14000	13000

THE TIDYR PACKAGE

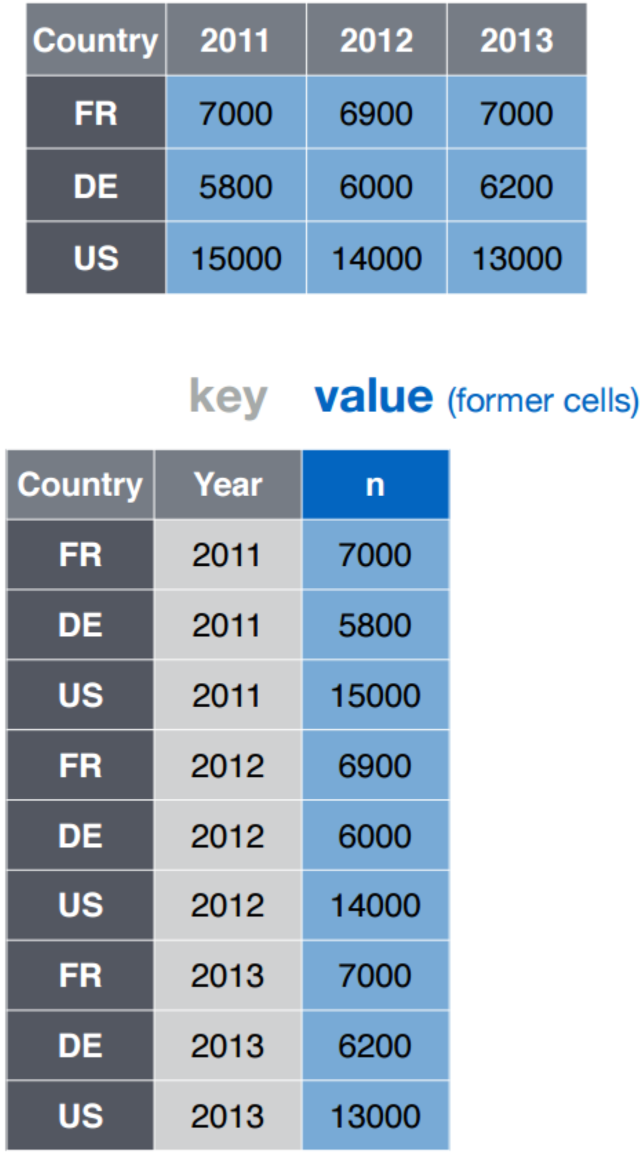
- **The gather() function**
- **Syntax**

```
gather(data, key, value, …, na.rm = FALSE/TRUE
```

- Example : The gather() function
- The following command is used to convert the data.

```
gather(cases, “Year", "n", 2:4)
```

  - cases : Dataset Name
  - Year: Key
  - n: value
  - 2:4 : Specifications of columns (from 2nd column to 4th column, the values should be gathered)



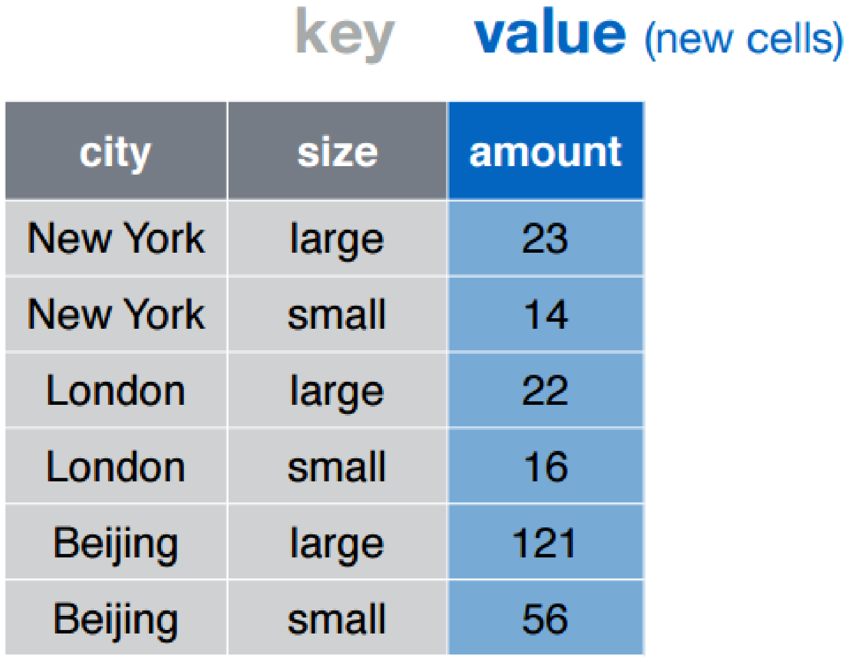
Country	2011	2012	2013
FR	7000	6900	7000
DE	5800	6000	6200
US	15000	14000	13000

key value (former cells)

Country	Year	n
FR	2011	7000
DE	2011	5800
US	2011	15000
FR	2012	6900
DE	2012	6000
US	2012	14000
FR	2013	7000
DE	2013	6200
US	2013	13000

The tidyr package

- **The spread() function**
- The spread() function spreads a key-value pair across multiple columns.
- Dataset used here is the pollution data, which has 6 rows and 3 columns.
- We can spread the values (amount) in two different columns (For example: Large and Small)


key
value (new cells)

city	size	amount
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

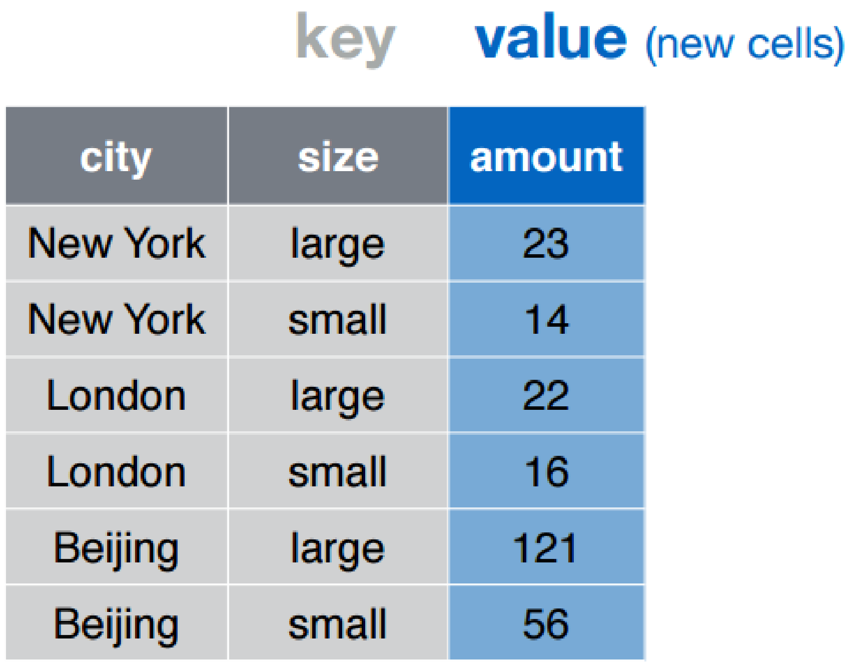
THE TIDYR PACKAGE

- **The spread() function**
- **Syntax** spread(data, key, value)

- Example : The spread() function
- Command is follows

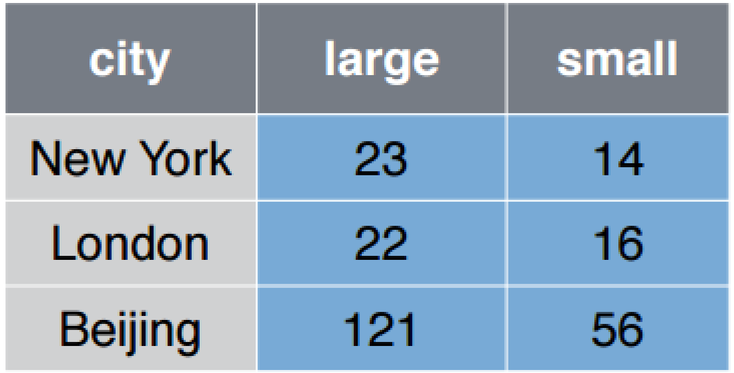
```
spread(pollution, size, amount)
```

  - Pollution : data
  - Size: key
  - Amount : value



key value (new cells)

city	size	amount
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56



city	large	small
New York	23	14
London	22	16
Beijing	121	56

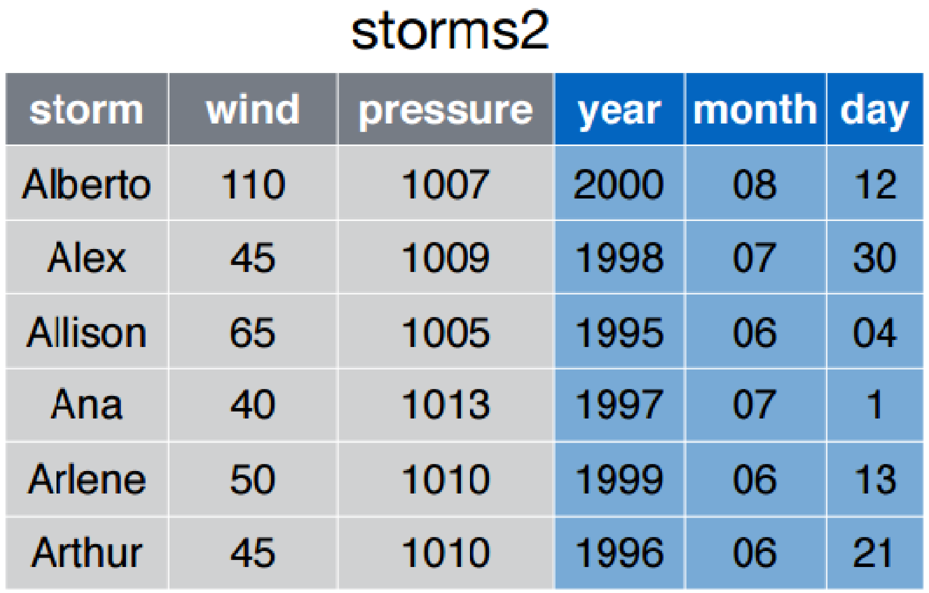
The tidyr package

- The **unite()** function
- It is convenience function to paste together multiple columns into one.
- **Syntax**

unite(data, col, …, sep = "_“)

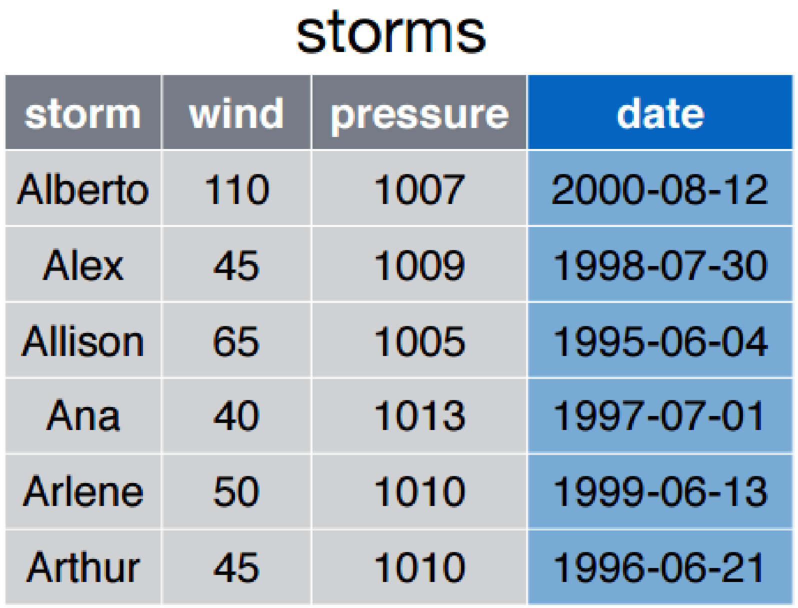
- **Example**

```
unite(storms2, "date", year, month,
day, sep = "-")
```



storms2

storm	wind	pressure	year	month	day
Alberto	110	1007	2000	08	12
Alex	45	1009	1998	07	30
Allison	65	1005	1995	06	04
Ana	40	1013	1997	07	1
Arlene	50	1010	1999	06	13
Arthur	45	1010	1996	06	21



storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

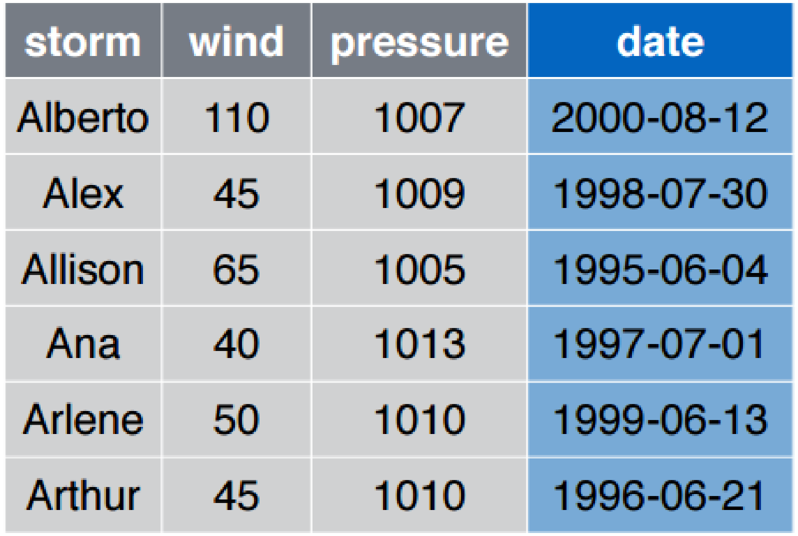
THE TIDYR PACKAGE

- The **separate()** function
  - It turns a single character column into multiple columns.
- **Syntax**

separate(data, col, into, sep = “_/:/;/grep”)

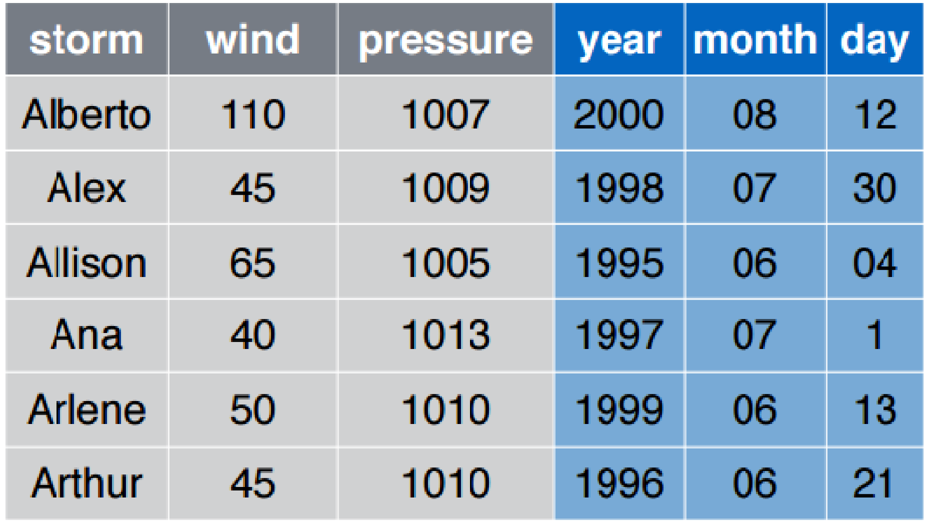
- **Example**
- `separate(storms, date, c("year", "month", "day"), sep = "-")`

storms



storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storms2



storm	wind	pressure	year	month	day
Alberto	110	1007	2000	08	12
Alex	45	1009	1998	07	30
Allison	65	1005	1995	06	04
Ana	40	1013	1997	07	1
Arlene	50	1010	1999	06	13
Arthur	45	1010	1996	06	21

The sqldf package

- Many business users had to dealt to RDBMS previously.
- In R, there is a package called "**sqldf**" for running sql statements and data manipulation in R
- To install

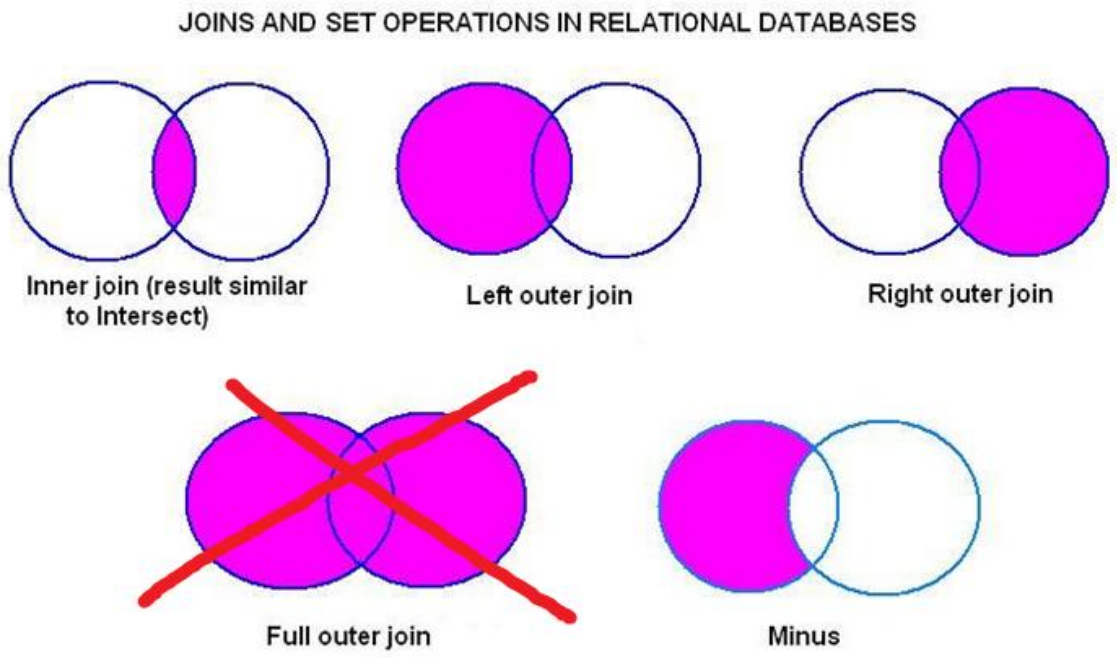
```
install.packages("sqldf")
```

- To load :

```
 library(sqldf)
```

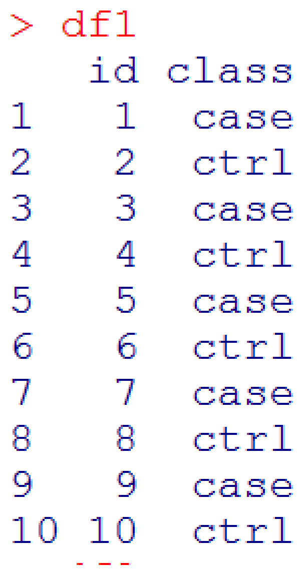
The sqldf Package

- Performing joins is more common in SQL.
- **Left joins** : Returns all left table.
- **Right joins** : Returns all right table.
- **Inner joins** : Returns only rows which are matching data for common variables.
- **Full outer join**: Returns all rows from all tables, if rows are not matching.

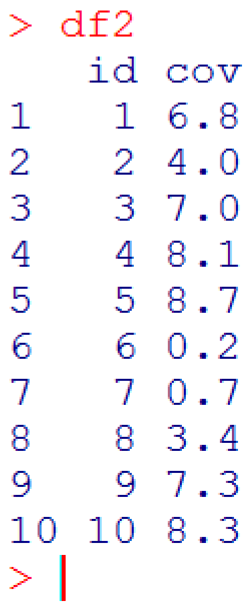

JOINS AND SET OPERATIONS IN RELATIONAL DATABASES
Inner join (result similar to Intersect)
Left outer join
Right outer join
Full outer join
Minus

THE SQLDF PACKAGE

- Example 1: **select()** function
- The following two datasets are used



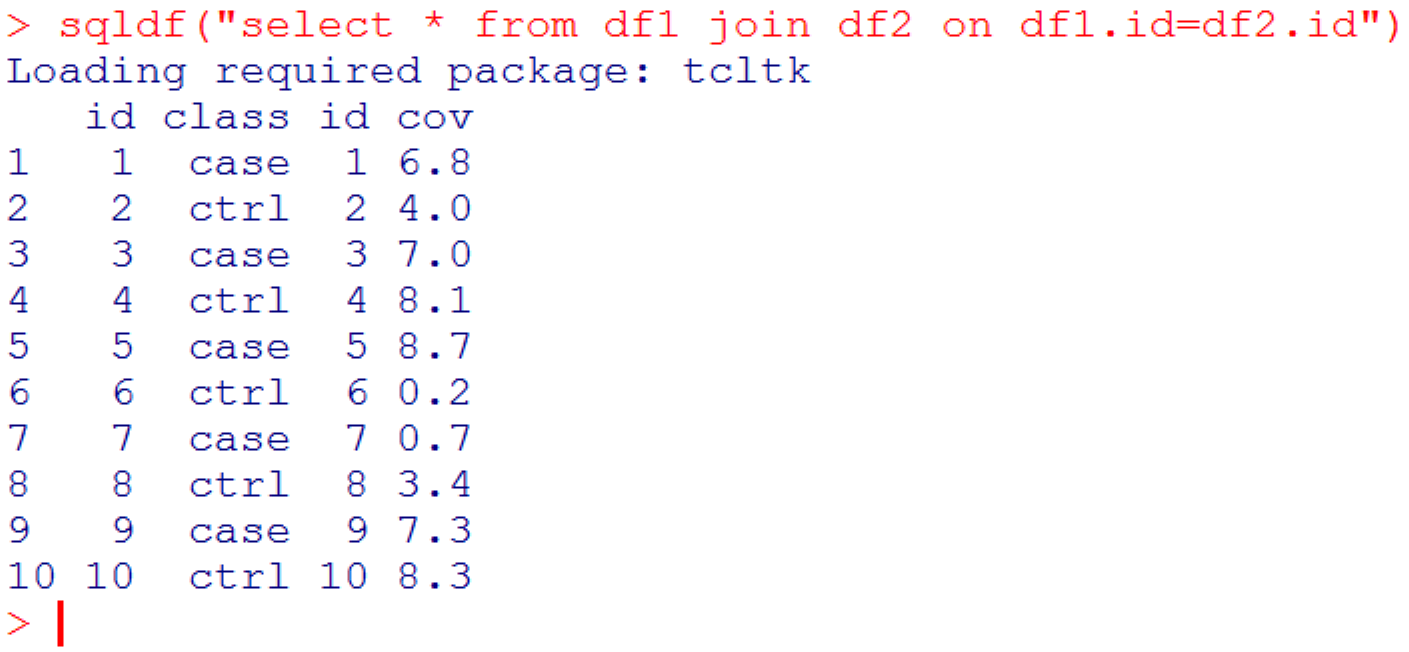
```
> df1
   id class
1   1  case
2   2  ctrl
3   3  case
4   4  ctrl
5   5  case
6   6  ctrl
7   7  case
8   8  ctrl
9   9  case
10 10  ctrl
```



```
> df2
   id cov
1   1 6.8
2   2 4.0
3   3 7.0
4   4 8.1
5   5 8.7
6   6 0.2
7   7 0.7
8   8 3.4
9   9 7.3
10 10 8.3
> |
```

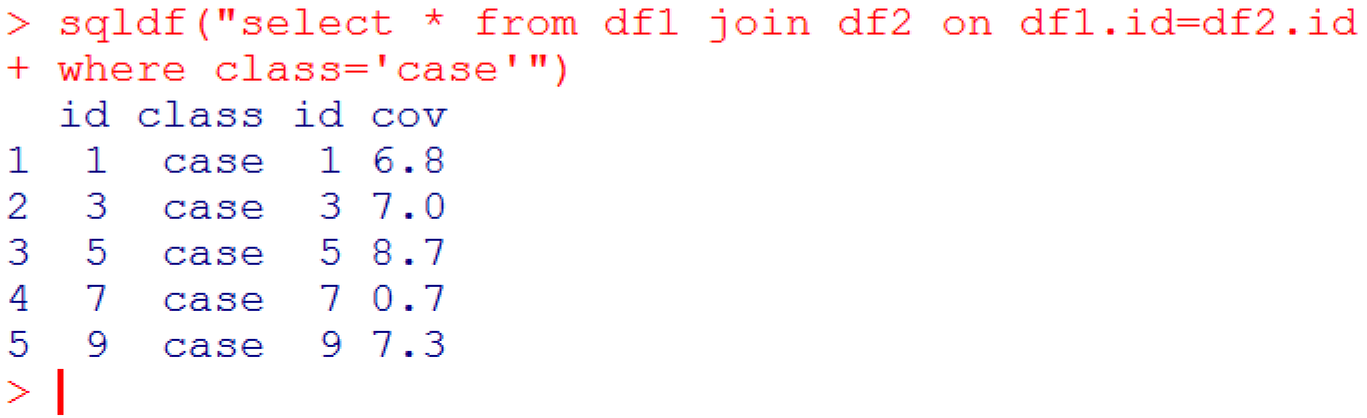
THE SQLDF PACKAGE

- Example : sqldf package
- Performing **Inner Join**



```
> sqldf("select * from df1 join df2 on df1.id=df2.id")
Loading required package: tcltk
   id class id cov
1   1  case  1 6.8
2   2  ctrl  2 4.0
3   3  case  3 7.0
4   4  ctrl  4 8.1
5   5  case  5 8.7
6   6  ctrl  6 0.2
7   7  case  7 0.7
8   8  ctrl  8 3.4
9   9  case  9 7.3
10 10  ctrl 10 8.3
> 
```

- Performing **Inner Join** and **where** clause in it



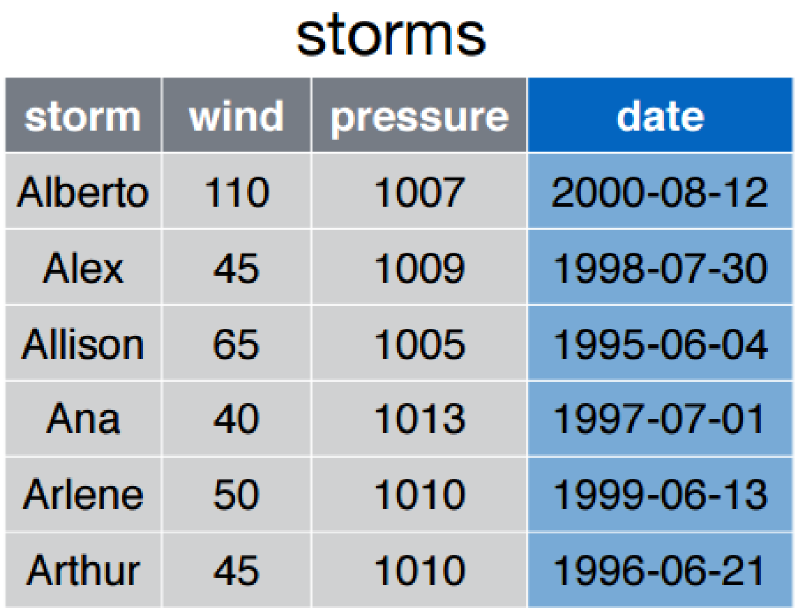
```
> sqldf("select * from df1 join df2 on df1.id=df2.id
+ where class='case'")
  id class id cov
1  1  case  1 6.8
2  3  case  3 7.0
3  5  case  5 8.7
4  7  case  7 0.7
5  9  case  9 7.3
> 
```

- Sub setting the data

  `sqldf("select id from df1“)`

The dplyr package

- A package that transforms tabular data.
- Functions in dplyr package
  - Select
  - Filter
  - Mutate
  - Arrange
  - Group_by and
  - Summarise
- Data set used is **storms** data



storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

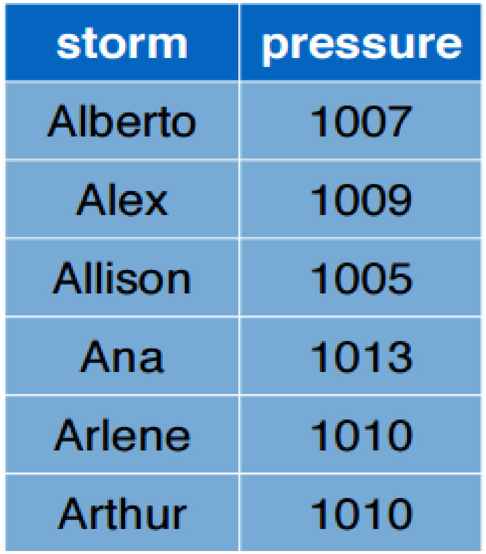
THE DPLYR PACKAGE

- Example : The select() function
- The select() function keeps only the variables you mention.
- **Syntax**

Select(data, …)

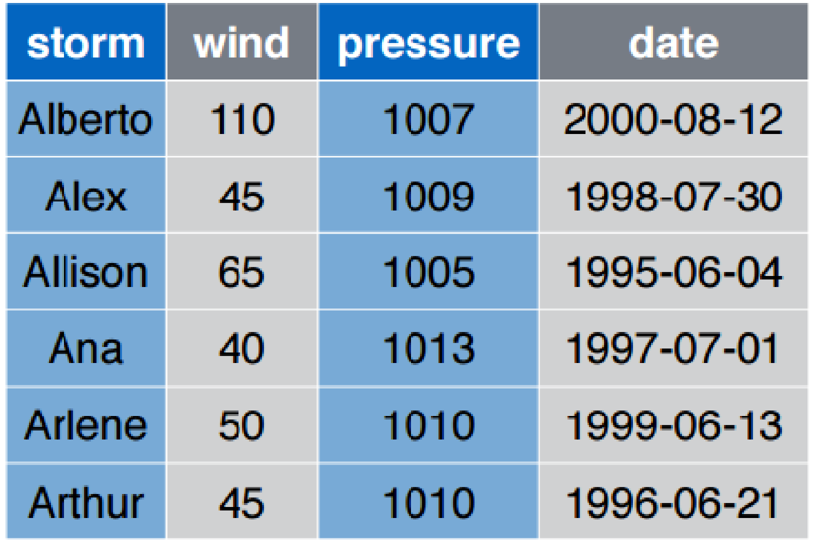
- The command used for the following output

```
select(storms, storm, pressure)
select(storms, -storm)
```

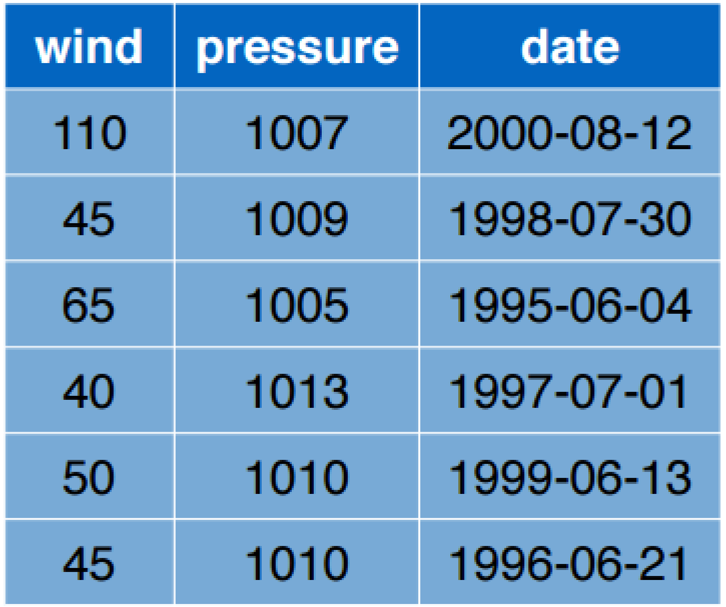


storm	pressure
Alberto	1007
Alex	1009
Allison	1005
Ana	1013
Arlene	1010
Arthur	1010

storms



storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



wind	pressure	date
110	1007	2000-08-12
45	1009	1998-07-30
65	1005	1995-06-04
40	1013	1997-07-01
50	1010	1999-06-13
45	1010	1996-06-21

The dplyr package

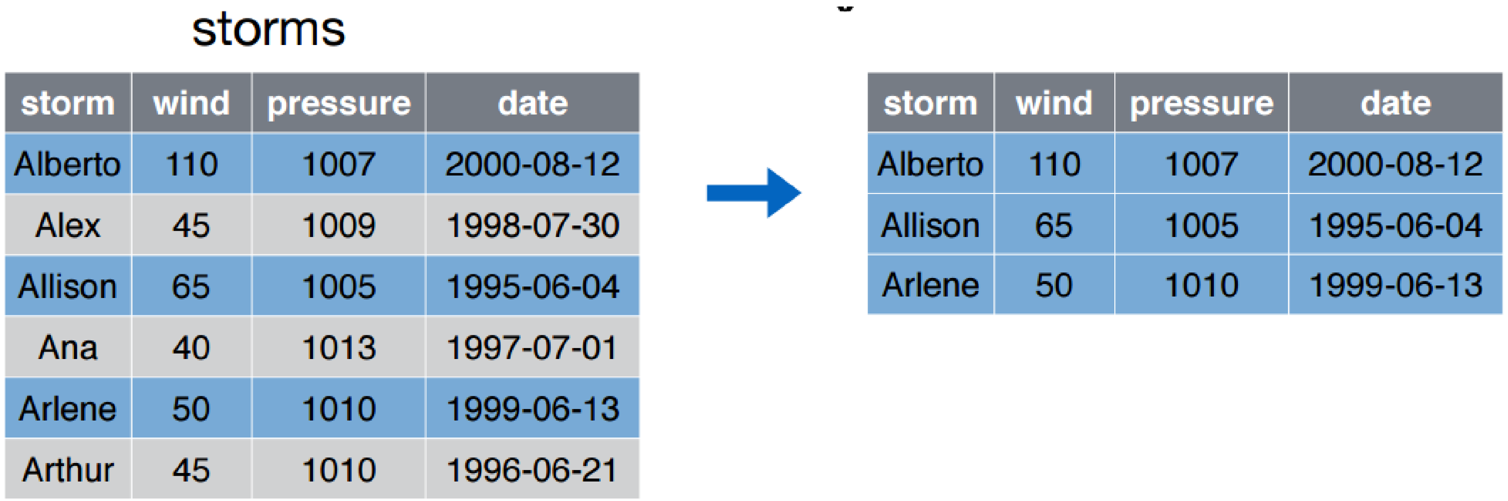
- Example : The filter() function
- The filter() function return rows with matching conditions.
- **Syntax**

filter(data, ...)

- **Command**

```
filter(storms, wind >= 50)
```

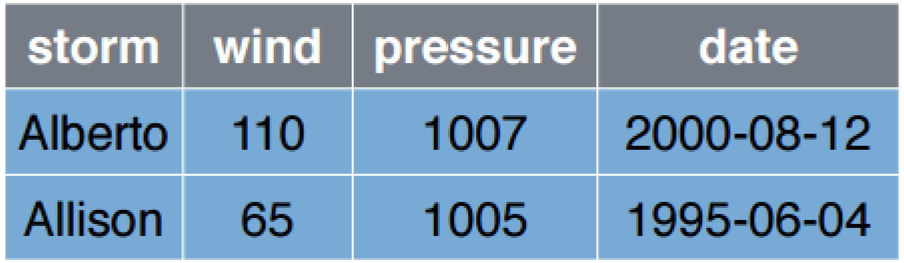
```
filter(storms, wind >= 50,
 storm %in% c("Alberto",
 "Alex", "Allison"))
```



storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Allison	65	1005	1995-06-04
Arlene	50	1010	1999-06-13

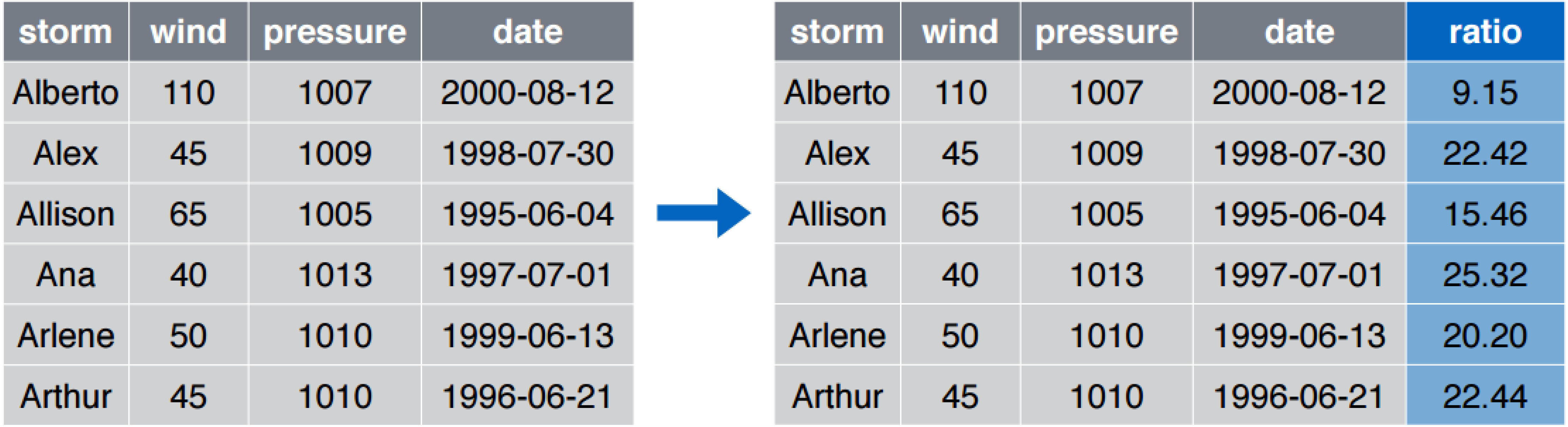


storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Allison	65	1005	1995-06-04

THE DPLYR PACKAGE

- Example : The mutate() function
- The mutate() function Derive new variables from existing variables.
- **Syntax** **mutate(data, ...)**
- **Command**

```
mutate(storms, ratio = pressure /wind)
```



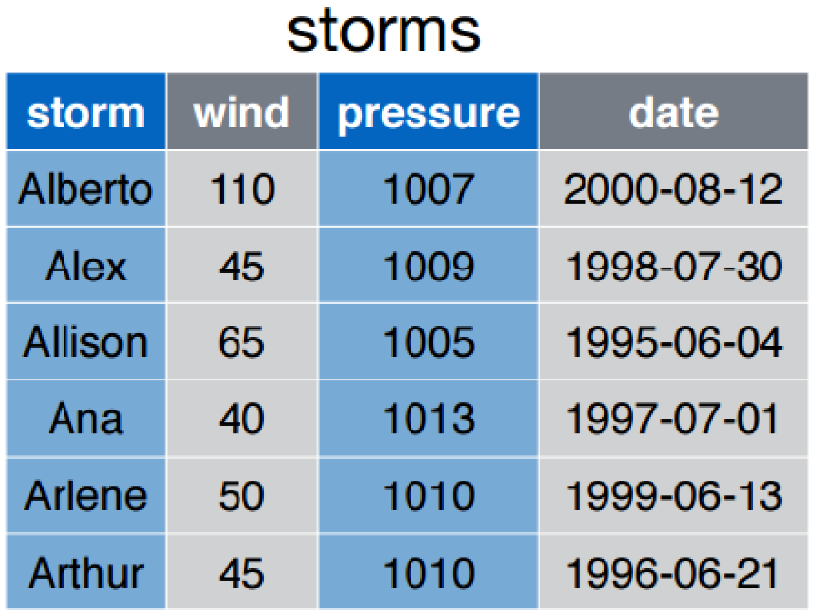
storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	wind	pressure	date	ratio
Alberto	110	1007	2000-08-12	9.15
Alex	45	1009	1998-07-30	22.42
Allison	65	1005	1995-06-04	15.46
Ana	40	1013	1997-07-01	25.32
Arlene	50	1010	1999-06-13	20.20
Arthur	45	1010	1996-06-21	22.44

The dplyr package

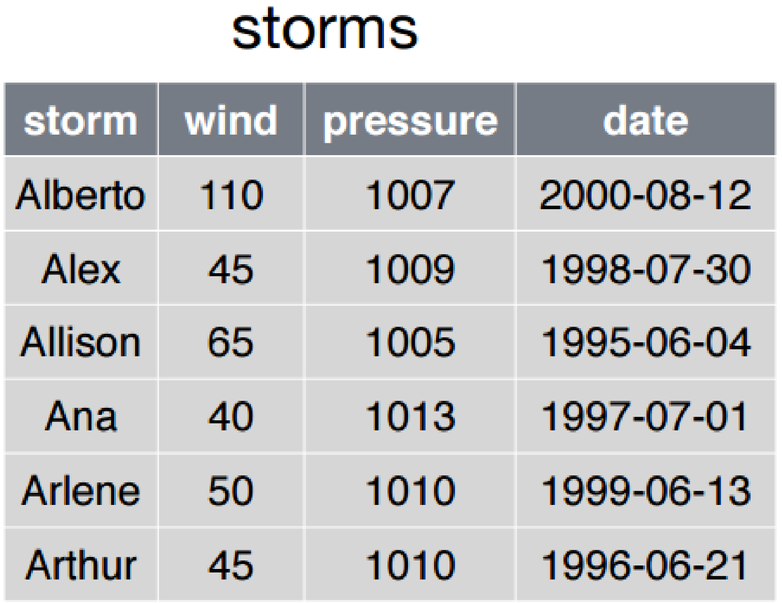
- Example : The arrange() function
- The arrange() function Arrange rows by variables
- **Syntax** **arrange(data, ...)**
- **Command**

```
arrange(storms, wind)
arrange(storms, desc(wind))
```



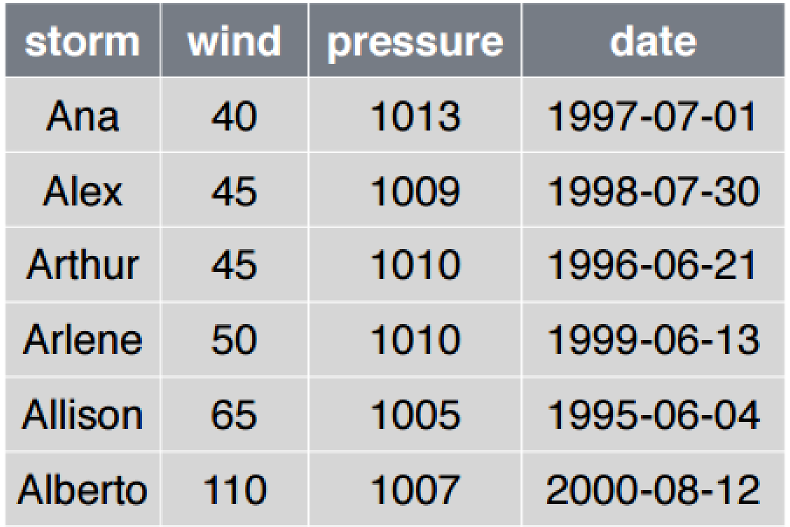
storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



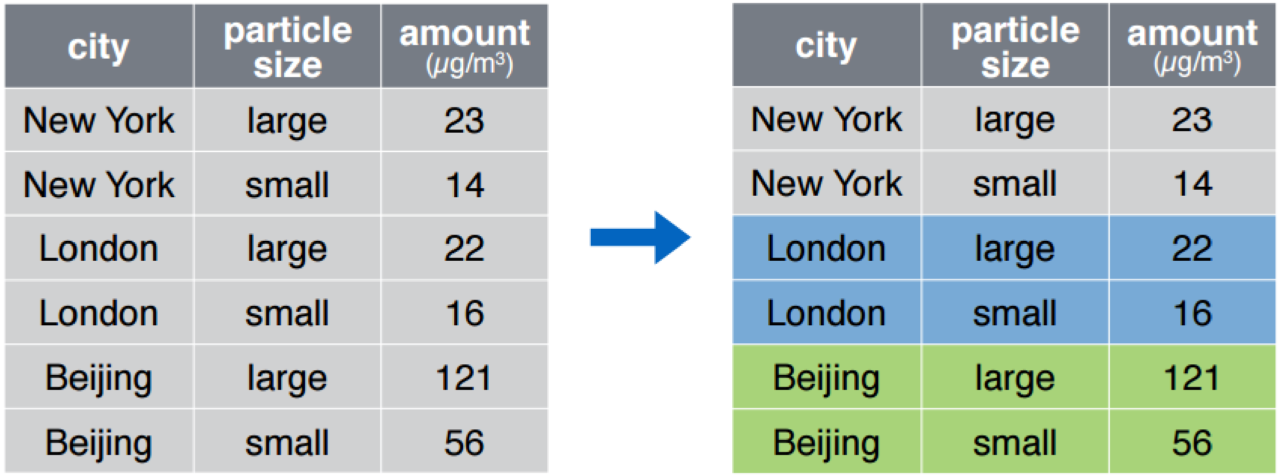
storm	wind	pressure	date
Ana	40	1013	1997-07-01
Alex	45	1009	1998-07-30
Arthur	45	1010	1996-06-21
Arlene	50	1010	1999-06-13
Allison	65	1005	1995-06-04
Alberto	110	1007	2000-08-12

The dplyr package

- Example : The group_by() function
- The group_by() function Group a table by one or more variables.
- The group_by() function takes an existing table and converts it into a grouped table where operations are performed "by group".
- **Syntax**
- **Command**

group_by(data, ...)

```
pollution %>% group_by(city)
```



city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

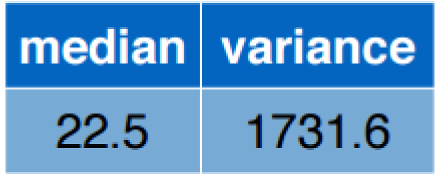
The dplyr package

- Example : The summarise() function.
- The summarise() funciton Summarises multiple values to a single value.
- **Syntax**

summarise(data, ...)

  - data : Data frame or Table
  - ... : Name-value pairs of summary functions like **min(), mean(), max()** etc.
- Applying various summary functions on **Pollution data**
- **Command**

```
pollution %>% summarise(median = median(amount),
                        variance = var(amount))
```



median	variance
22.5	1731.6

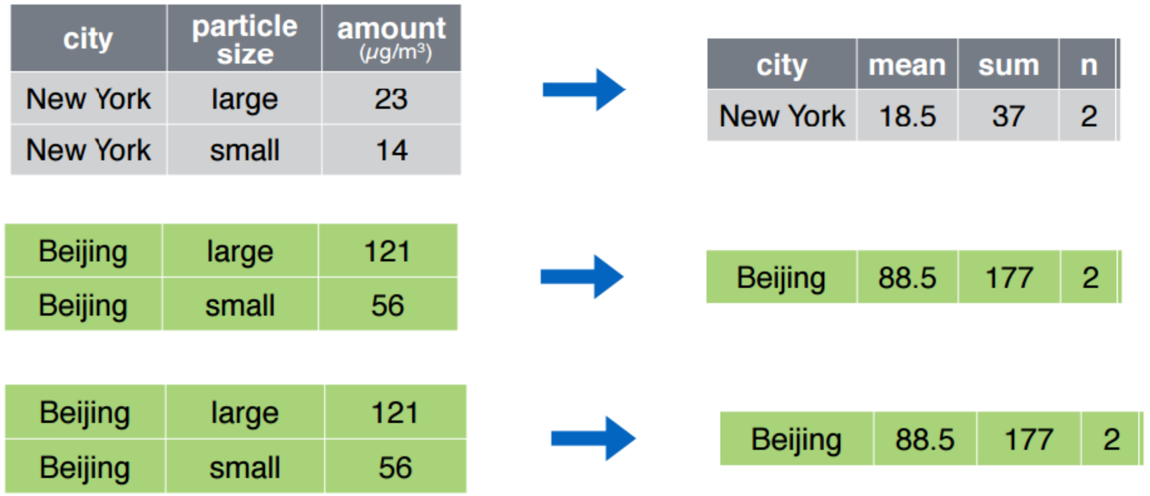
THE DPLYR PACKAGE

- Applying various summary functions on Pollution data

```
pollution %>% summarise(mean = mean(amount), sum
              = sum(amount), n = n())
```

mean	sum	n
42	252	6

```
pollution %>% group_by(city) %>%
summarise(mean = mean(amount), sum
     = sum(amount), n = n())
```



city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
Beijing	large	121
Beijing	small	56
Beijing	large	121
Beijing	small	56

city	mean	sum	n
New York	18.5	37	2
Beijing	88.5	177	2
Beijing	88.5	177	2

THE DPLYR PACKAGE

- Example : The bind() function
- The bind() efficiently bind multiple data frames by row and column.
- It has two functions under this
  - The bind_cols() and bind_rows() function
- The bind_cols() efficiently bind multiple data frames by columns
- The bind_rows() efficiently bind multiple data frames by columns
- **Syntax of bind_cols**

```
bind_cols(x, ...)
```

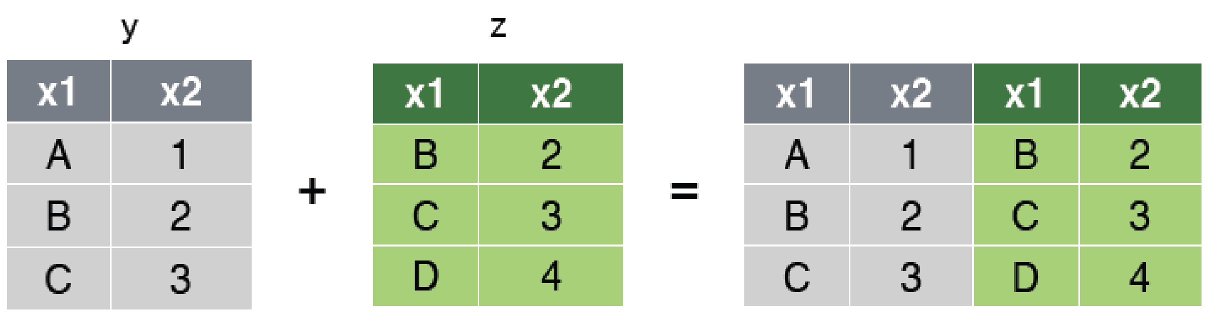
- **Syntax of bind_rows**

```
bind_rows(x, ...)
```

The dplyr package

- Example : The bind() functions
- Commands for bind_rows() and bind_cols()

```
bind_cols(y, z)
```



y

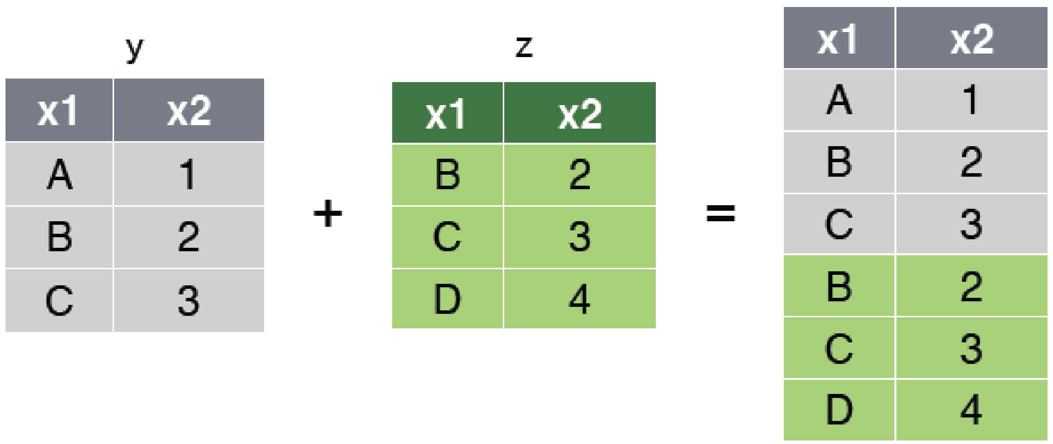
x1	x2
A	1
B	2
C	3

z

x1	x2
B	2
C	3
D	4

x1	x2	x1	x2
A	1	B	2
B	2	C	3
C	3	D	4

```
bind_rows(y, z)
```



y

x1	x2
A	1
B	2
C	3

z

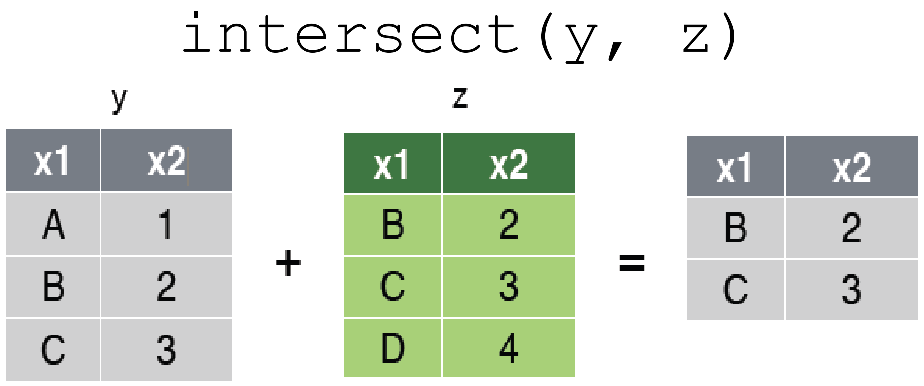
x1	x2
B	2
C	3
D	4

x1	x2
A	1
B	2
C	3
B	2
C	3
D	4

THE DPLYR PACKAGE

- Example : **Set Operations**
- There are four functions under Set Operations in dplyr package
  - The intersect( ) function
  - The union( ) function
  - The setdiff( ) function
  - The setequal( ) function
- **Syntax's**

```
intersect(x, y, ...)
union(x, y, ...)
setdiff(x, y, ...)
setequal(x, y, ...)
```



`intersect(y, z)`

y

x1	x2
A	1
B	2
C	3

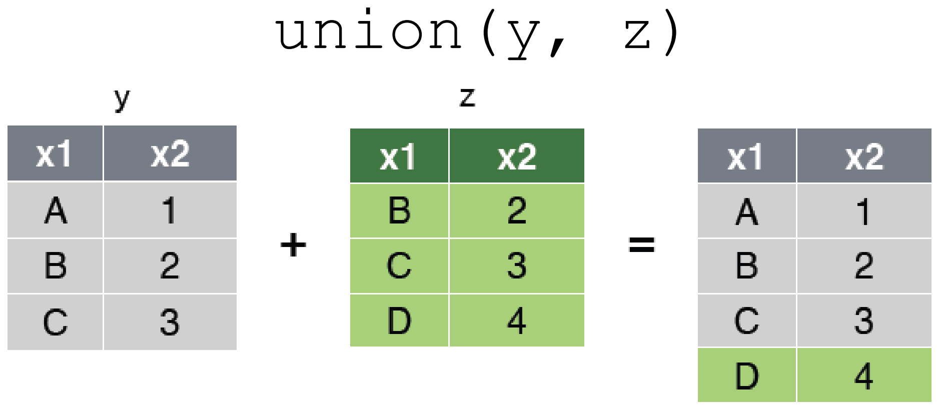
+

z

x1	x2
B	2
C	3
D	4

=

x1	x2
B	2
C	3



`union(y, z)`

y

x1	x2
A	1
B	2
C	3

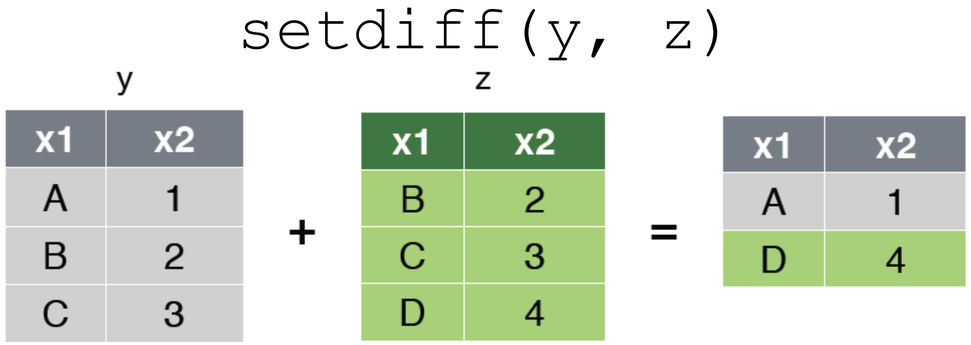
+

z

x1	x2
B	2
C	3
D	4

=

x1	x2
A	1
B	2
C	3
D	4



`setdiff(y, z)`

y

x1	x2
A	1
B	2
C	3

+

z

x1	x2
B	2
C	3
D	4

=

x1	x2
A	1
D	4

The dplyr package

- Example : **The join operations**
- Types of joins in the dplr package along with the syntax
  - 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)

The dplyr package

- Example1 : Left join

```
left_join(songs, artists, by = "name")
```

songs

song	name
Across the Universe	John
Come Together	John
Hello, Goodbye	Paul
Peggy Sue	Buddy

+

artists

name	plays
George	sitar
John	guitar
Paul	bass
Ringo	drums

=

song	name	plays
Across the Universe	John	guitar
Come Together	John	guitar
Hello, Goodbye	Paul	bass
Peggy Sue	Buddy	<NA>

- Example2 : Left join

```
left_join(songs2, artists2, by = c("first", "last"))
```

songs2

song	first	last
Across the Universe	John	Lennon
Come Together	John	Lennon
Hello, Goodbye	Paul	McCartney
Peggy Sue	Buddy	Holly

+

artists2

first	last	plays
George	Harrison	sitar
John	Lennon	guitar
Paul	McCartney	bass
Ringo	Starr	drums
Paul	Simon	guitar
John	Coltranee	sax

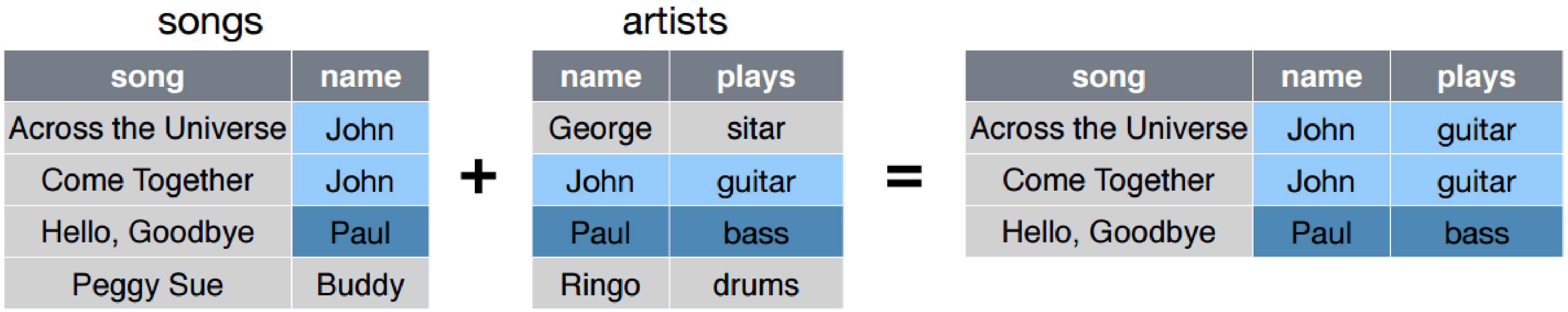
=

song	first	last	plays
Across the Universe	John	Lennon	guitar
Come Together	John	Lennon	guitar
Hello, Goodbye	Paul	McCartney	bass
Peggy Sue	Buddy	Holly	<NA>

The dplyr package

- Example : The inner join

```
inner_join(songs, artists, by = "name")
```



songs

song	name
Across the Universe	John
Come Together	John
Hello, Goodbye	Paul
Peggy Sue	Buddy

+

artists

name	plays
George	sitar
John	guitar
Paul	bass
Ringo	drums

=

song	name	plays
Across the Universe	John	guitar
Come Together	John	guitar
Hello, Goodbye	Paul	bass

Transformations

Transformations : Reassigning Variable

- **Reassigning Variables:**
- It's also possible to make other changes to data frames.
- For example, suppose that we wanted to define a new column (midpoint variable that is the mean of the high and low price.)
- We can add this variable with the same notation:

```
> dow30$mid <- (dow30$High + dow30$Low)/2
> names(dow30)
[1] "symbol" "Date" "Open" "High" "Low"
[6] "Close" "Volume" "Adj.Close" "mid"
```

Transformations

- The transform() function : Function used for changing the number of variables in a data frame
- **Syntax:**

transform(data, ...)

- To use transform, you specify a data frame (as the first argument) and a set of expressions that use variables within the data frame.

- The transform function applies each expression to the data frame and then returns the final data frame.

```
> dow30.transformed <- transform(dow30,
  Date=as.Date(Date), mid = (High + Low)/2)
```

Applying a Function to Each Element of an Object

- Transforming data is applying a common function to set of objects and returning a new set of transformed objects.
- The base R library includes set of different functions for doing this.

- **Applying a function to an array or matrix**
- To apply a function to parts of an array (or matrix), use the `apply` function:

```
apply(X, MARGIN, FUN, ...)
```

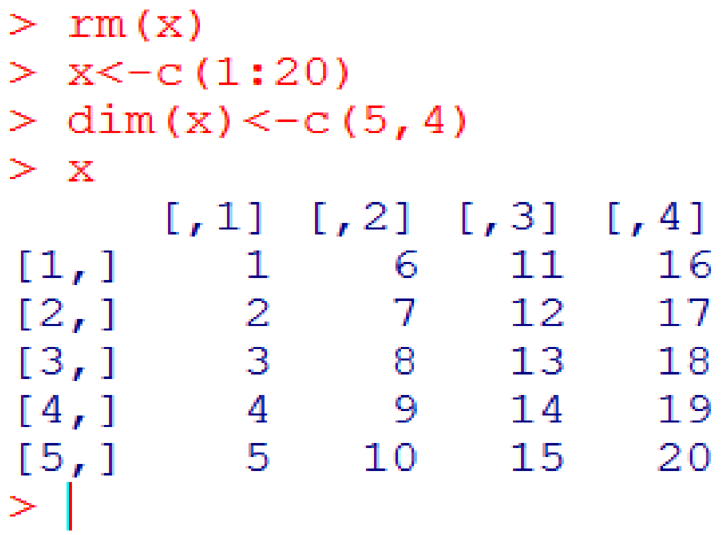
  - **X** is an array (or matrix) to which function is applied
  - **FUN** is the function that is applied
  - **MARGIN** Dimensions of the array to which you would like to apply a function

Applying a function to an array

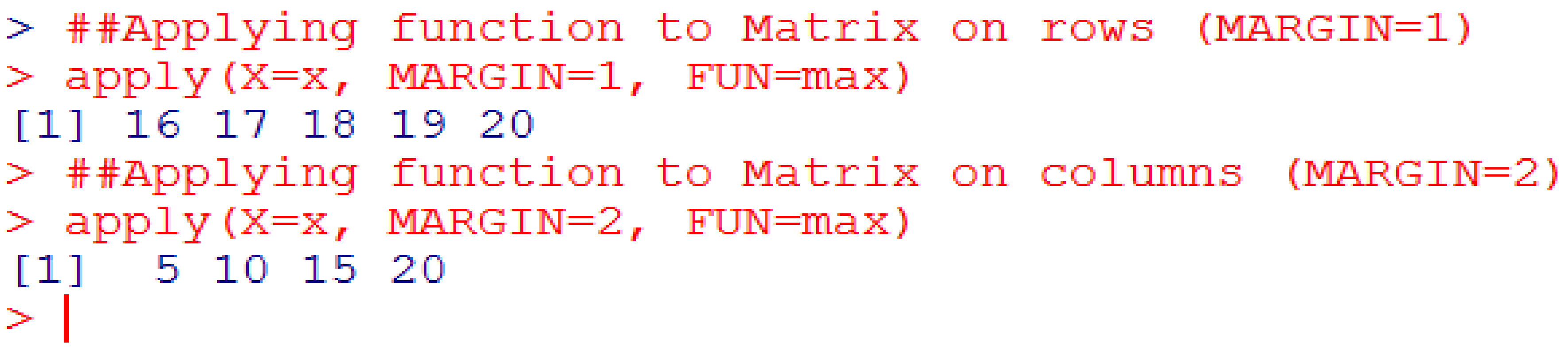
- Sample example for applying a function to an array or matrix

Here, we have created the matrix called as "x" with dimensions 5 rows and 4 columns

Now lets show how **apply** works.
We will use function **max** to get the highest numbers in the matrix



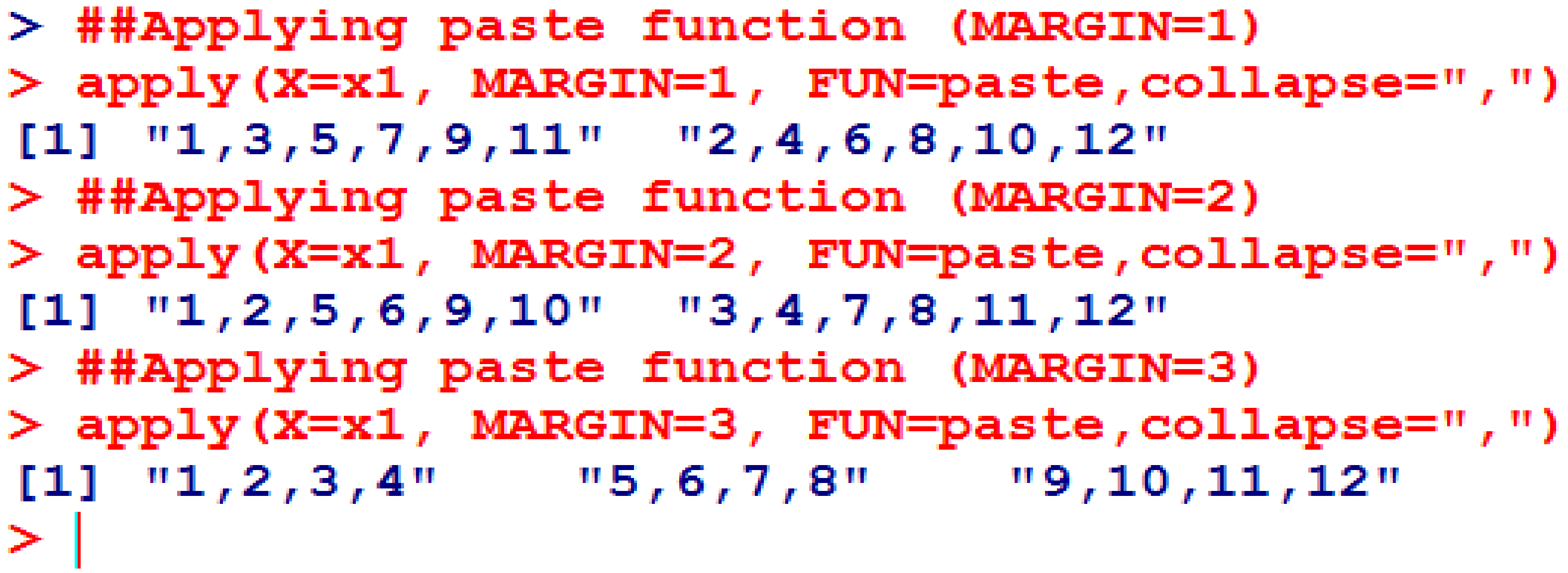
```
> rm(x)
> x<-c(1:20)
> dim(x)<-c(5,4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
> 
```



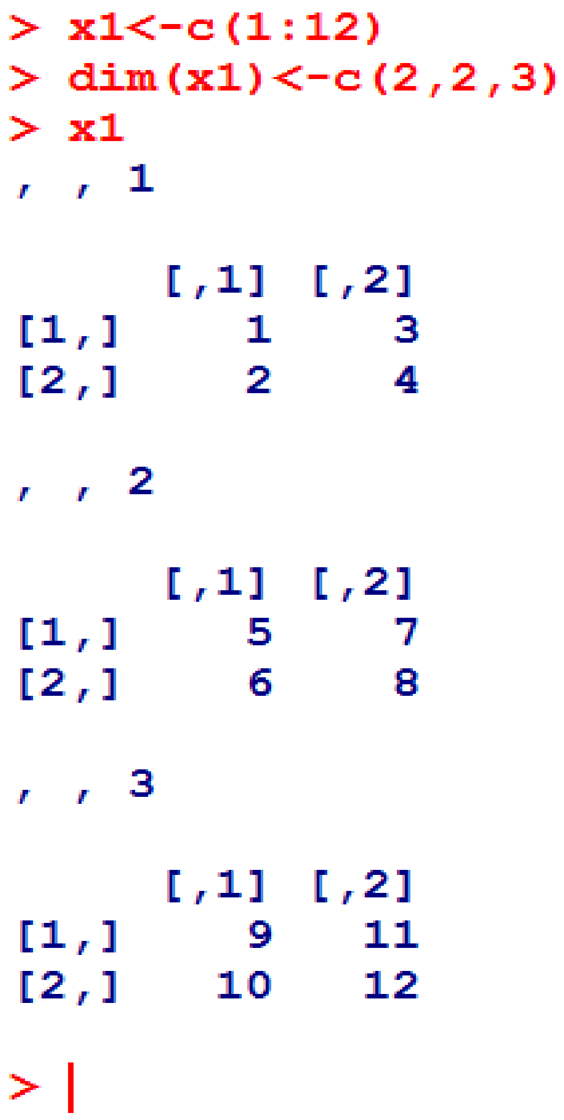
```
> ##Applying function to Matrix on rows (MARGIN=1)
> apply(X=x, MARGIN=1, FUN=max)
[1] 16 17 18 19 20
> ##Applying function to Matrix on columns (MARGIN=2)
> apply(X=x, MARGIN=2, FUN=max)
[1]  5 10 15 20
> 
```

Applying a function to an array

- One more example on **apply** function
- In addition to `MARGIN=1` and `MARGIN=2`, we can also use MARGIN over multiple dimensions
- Let us create a 3-D matrix and **apply** function on it.



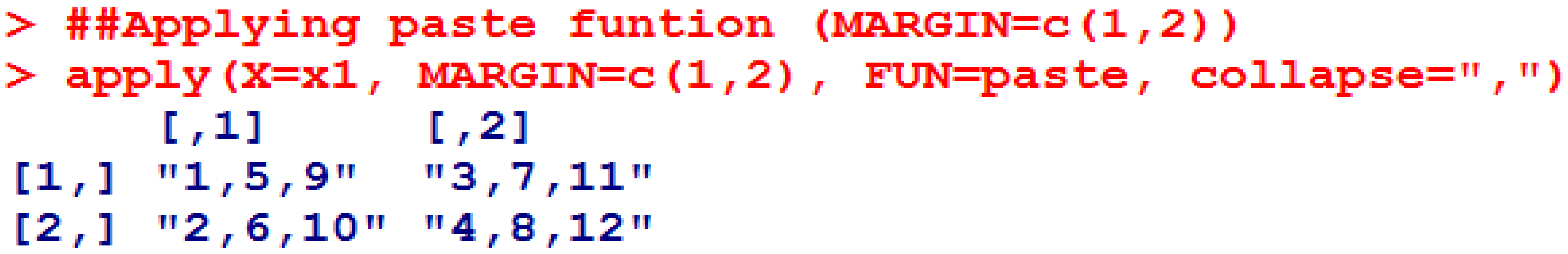
```
> ##Applying paste function (MARGIN=1)
> apply(X=x1, MARGIN=1, FUN=paste,collapse=",")
[1] "1,3,5,7,9,11"  "2,4,6,8,10,12"
> ##Applying paste function (MARGIN=2)
> apply(X=x1, MARGIN=2, FUN=paste,collapse=",")
[1] "1,2,5,6,9,10"  "3,4,7,8,11,12"
> ##Applying paste function (MARGIN=3)
> apply(X=x1, MARGIN=3, FUN=paste,collapse=",")
[1] "1,2,3,4"     "5,6,7,8"     "9,10,11,12"
> 
```



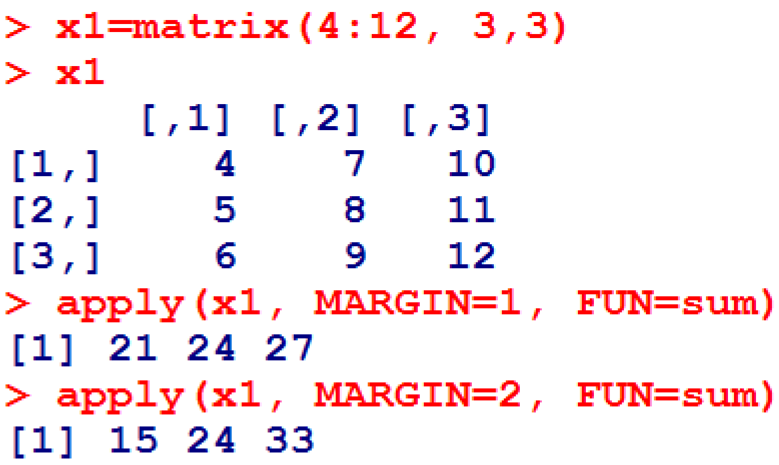
```
> x1<-c(1:12)
> dim(x1)<-c(2,2,3)
> x1
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12

> 
```

Applying a function to an array

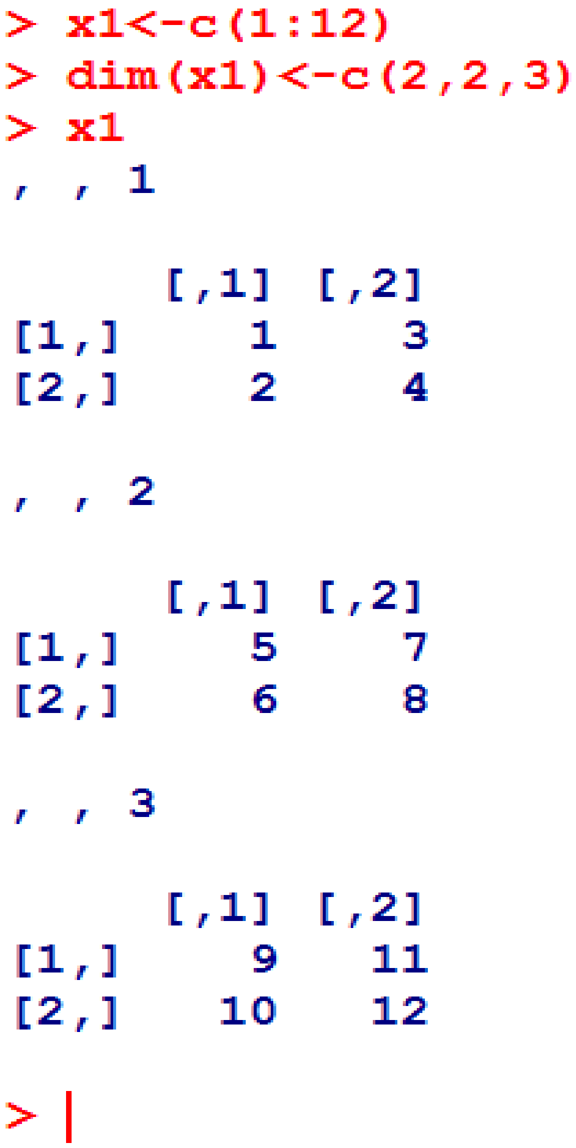
- One more example with `MARGIN=c(1,2)`



```
> ##Applying paste funtion (MARGIN=c(1,2))
> apply(X=x1, MARGIN=c(1,2), FUN=paste, collapse=",")
     [,1]      [,2]
[1,] "1,5,9"   "3,7,11"
[2,] "2,6,10"  "4,8,12"
```



```
> x1=matrix(4:12, 3,3)
> x1
     [,1] [,2] [,3]
[1,]    4    7   10
[2,]    5    8   11
[3,]    6    9   12
> apply(x1, MARGIN=1, FUN=sum)
[1] 21 24 27
> apply(x1, MARGIN=2, FUN=sum)
[1] 15 24 33
```



```
> x1<-c(1:12)
> dim(x1)<-c(2,2,3)
> x1
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12

> |
```

Applying a function to List or Vector

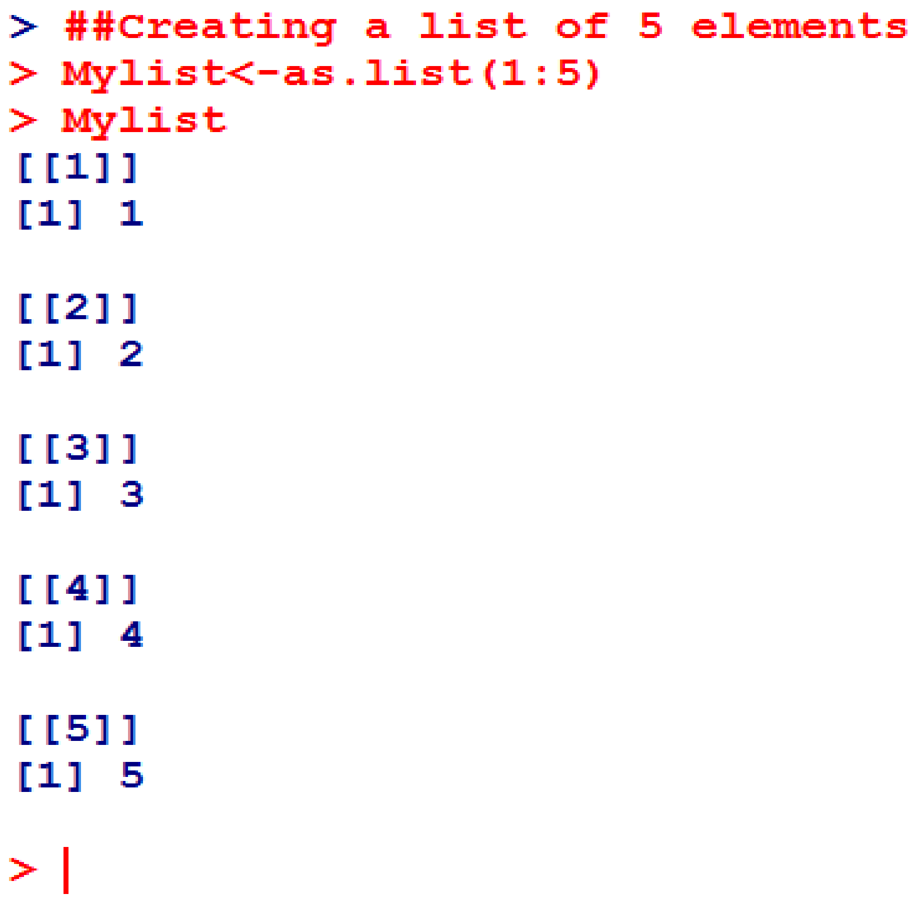
- To apply a function to each element in a vector or a list and return a list, you can use the function lapply
- Syntax

lapply(X, FUNC, ...)

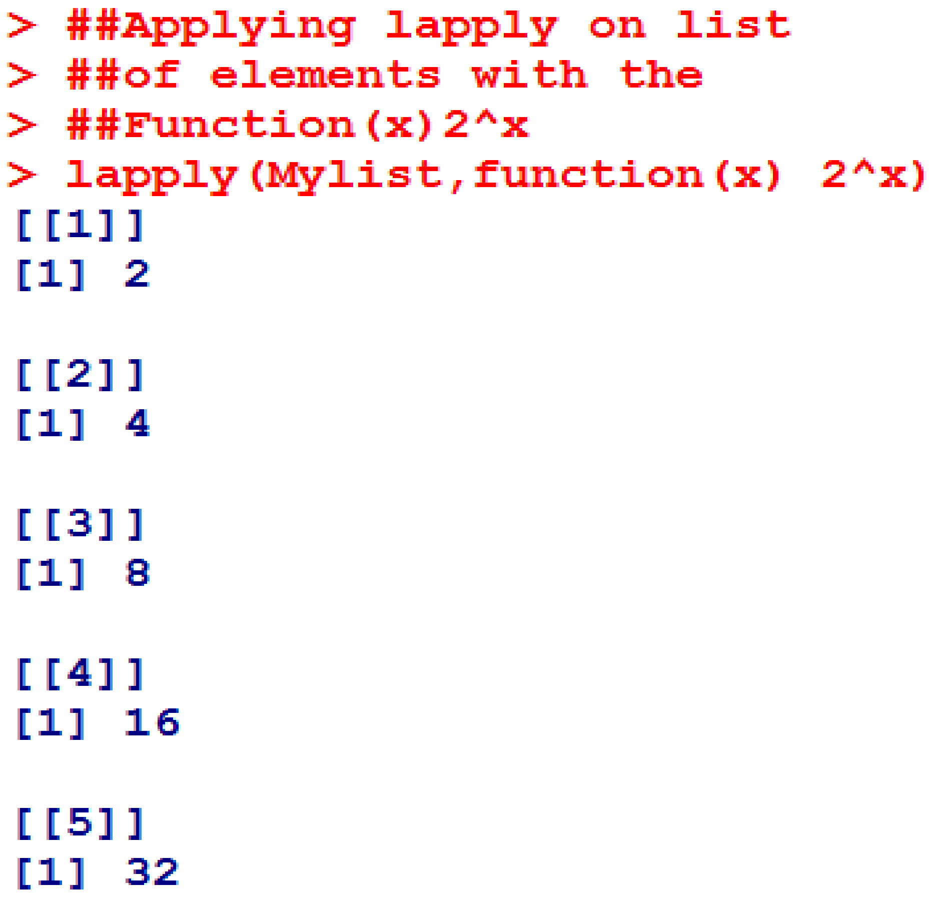
- The function lapply requires two arguments:
  - X : Name of the List or Vector
  - FUNC : Name of the function to be applied on List or Vector
- You may specify additional arguments that will be passed to FUNC.

Applying a function to List or Vector

- Simple example of how to use **lapply**
- Lets create the list of 5 elements and apply some function on the list created.



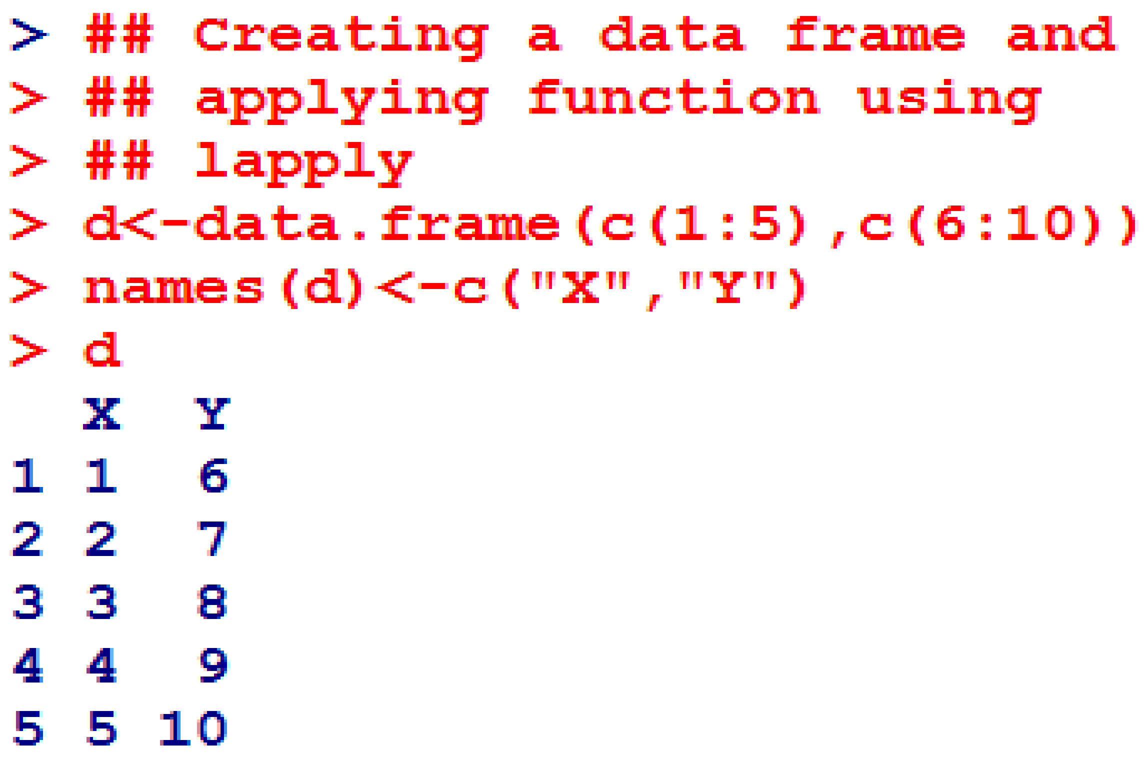
```
> ##Creating a list of 5 elements
> Mylist<-as.list(1:5)
> Mylist
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 4

[[5]]
[1] 5

> |
```



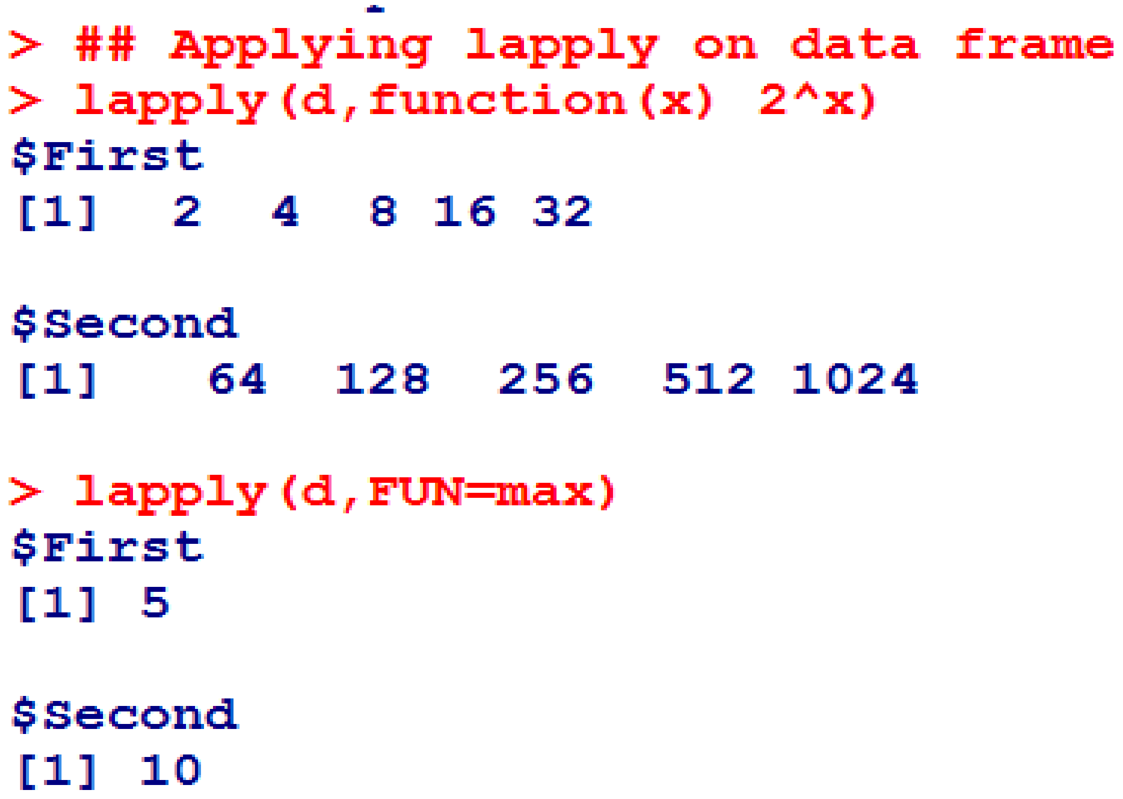
```
> ##Applying lapply on list
> ##of elements with the
> ##Function(x)2^x
> lapply(Mylist,function(x) 2^x)
[[1]]
[1] 2

[[2]]
[1] 4

[[3]]
[1] 8

[[4]]
[1] 16

[[5]]
[1] 32
```

Applying a function To a Data Frame

- You can apply a function to a data frame, and the function will be applied to each vector in the data frame.
- **Example:**



```
> ## Creating a data frame and
> ## applying function using
> ## lapply
> d<-data.frame(c(1:5),c(6:10))
> names(d)<-c("X","Y")
> d
  X  Y
1 1  6
2 2  7
3 3  8
4 4  9
5 5 10
```



```
> ## Applying lapply on data frame
> lapply(d,function(x) 2^x)
$First
[1]  2  4  8 16 32

$Second
[1]   64  128  256  512 1024

> lapply(d,FUN=max)
$First
[1] 5

$Second
[1] 10
```

Binning Data

- Another common data transformation is to group a set of observations into bins (groups) based on value of specific variables.
- **For example**

1. Suppose that you had some time series data where time was measured in days, but you wanted to summarize the data by month.

- There are several functions available for binning numeric data in R.

Binning Data- Cut

- In many data analysis settings, it might be useful to break up a continuous variable such as age into a categorical variable.
- Or, you might want to classify a categorical variable like year into a larger bin, such as 1990-2000.
- The **cut** function in R makes this task simple!

Binning Data- Cut

- The function cut is useful for taking a continuous variable and splitting it into discrete pieces
- Here is the default form of cut for use with numeric vectors:

```
# numeric form
cut(x, breaks)
```

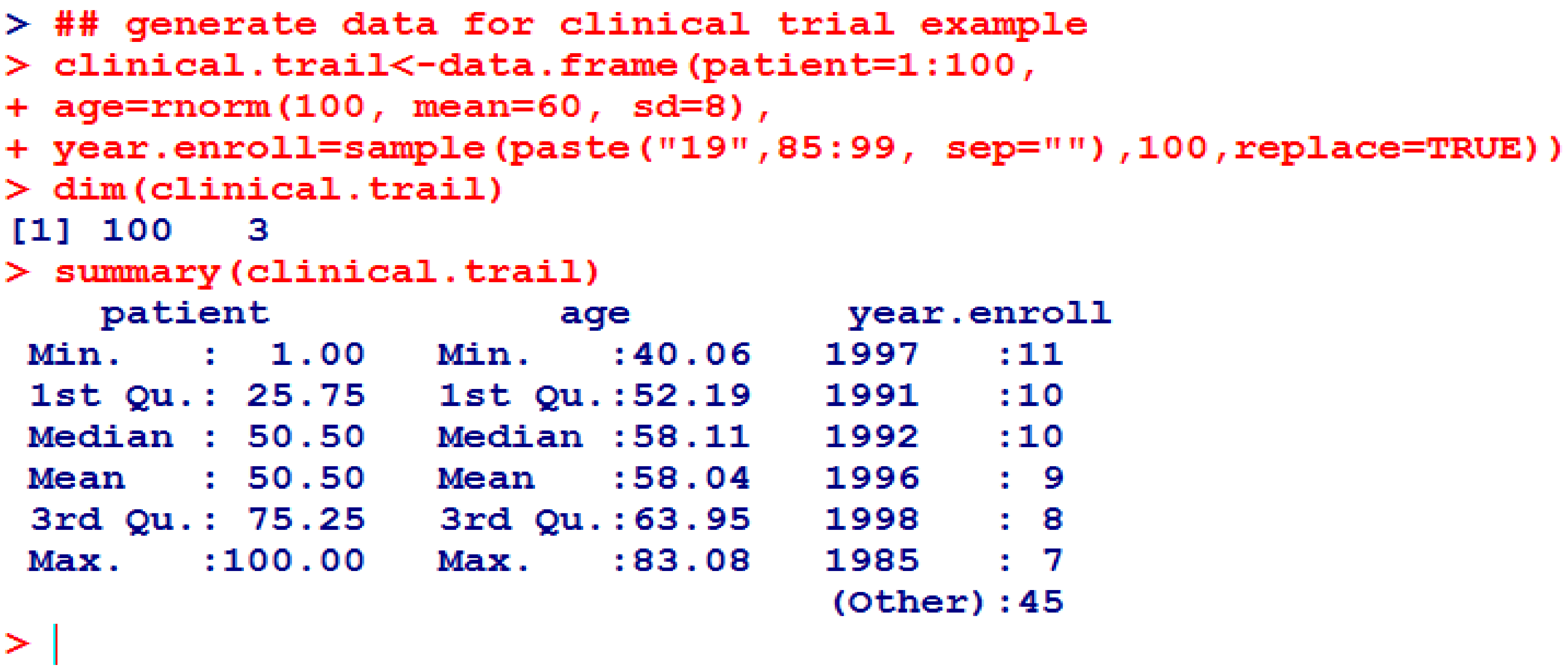
- There is also a version of cut for manipulating Date objects:

```
# Date form
cut(x, breaks, start.on.monday = TRUE)
```

- **The cut function takes a numeric vector as input and returns a factor**

Binning Data- Cut

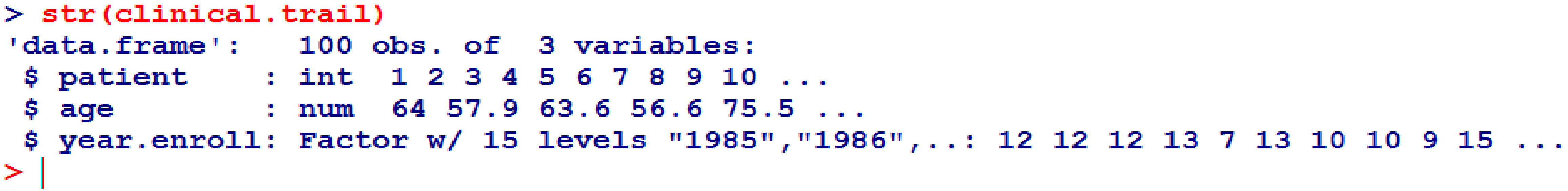
- Example for cut()
- Lets create the hypothetical clinical data set here



```
> ## generate data for clinical trial example
> clinical.trail<-data.frame(patient=1:100,
+ age=rnorm(100, mean=60, sd=8),
+ year.enroll=sample(paste("19",85:99, sep=""),100,replace=TRUE))
> dim(clinical.trail)
[1] 100   3
> summary(clinical.trail)
    patient              age            year.enroll
 Min.   :  1.00   Min.   :40.06   1997   :11  
 1st Qu.: 25.75   1st Qu.:52.19   1991   :10  
 Median : 50.50   Median :58.11   1992   :10  
 Mean   : 50.50   Mean   :58.04   1996   : 9  
 3rd Qu.: 75.25   3rd Qu.:63.95   1998   : 8  
 Max.   :100.00   Max.   :83.08   1985   : 7  
                                  (Other):45  
> 
```

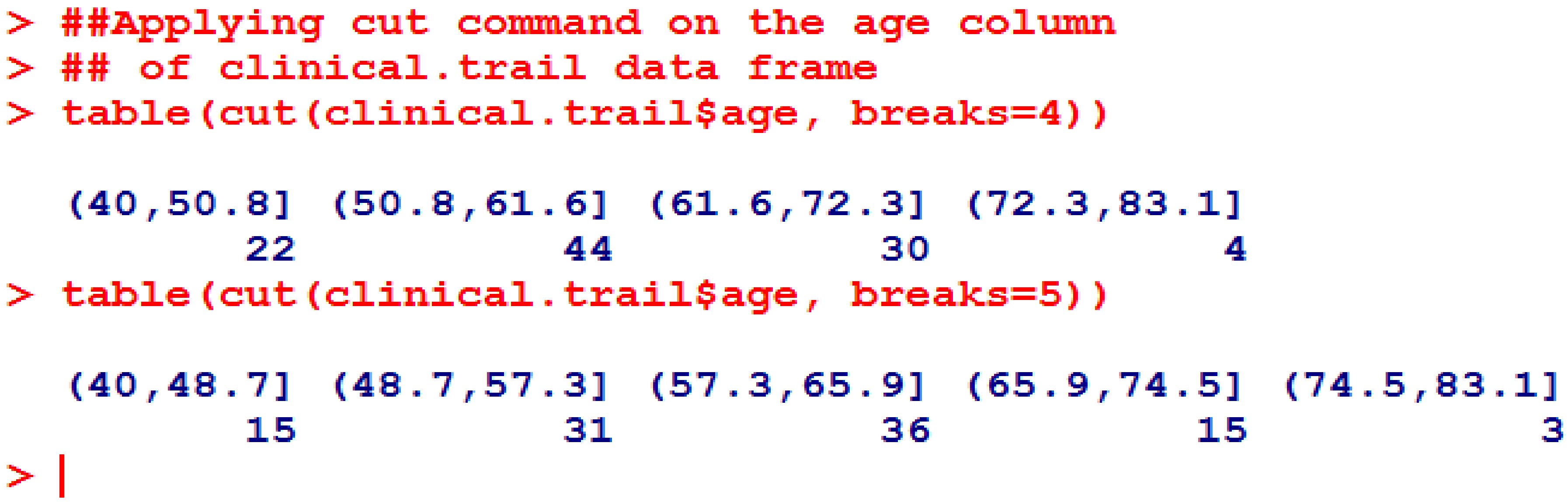
Binning Data- Cut

- We will apply cut command on the **clinical.trail** data frame to make **age a factor (Categorical value).**
- Lets see the structure of the data frame



```
> str(clinical.trail)
'data.frame':   100 obs. of  3 variables:
 $ patient    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ age        : num  64 57.9 63.6 56.6 75.5 ...
 $ year.enroll: Factor w/ 15 levels "1985","1986",..: 12 12 12 13 7 13 10 10 9 15 ...
> 
```

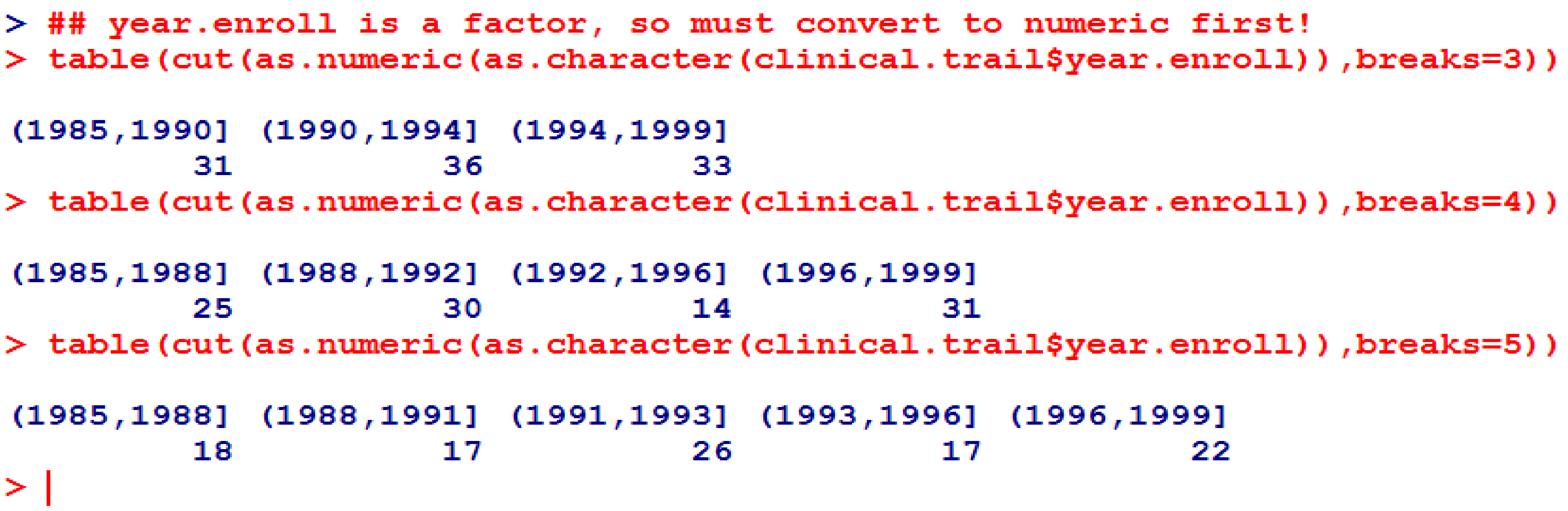
- Applying **cut()** on the **clinical.trial$age (# numeric form)**



```
> ##Applying cut command on the age column
> ## of clinical.trail data frame
> table(cut(clinical.trail$age, breaks=4))

  (40,50.8] (50.8,61.6] (61.6,72.3] (72.3,83.1] 
         22          44          30           4 
> table(cut(clinical.trail$age, breaks=5))

  (40,48.7] (48.7,57.3] (57.3,65.9] (65.9,74.5] (74.5,83.1] 
         15          31          36          15           3 
> 
```

Binning Data- Cut

- Applying **cut()** on the **clinical.trial$year.enroll (#Factor)**
  - Here, **year.enroll** column is a categorical data (CD). So we have to convert CD to numeric data and apply **cut()** command

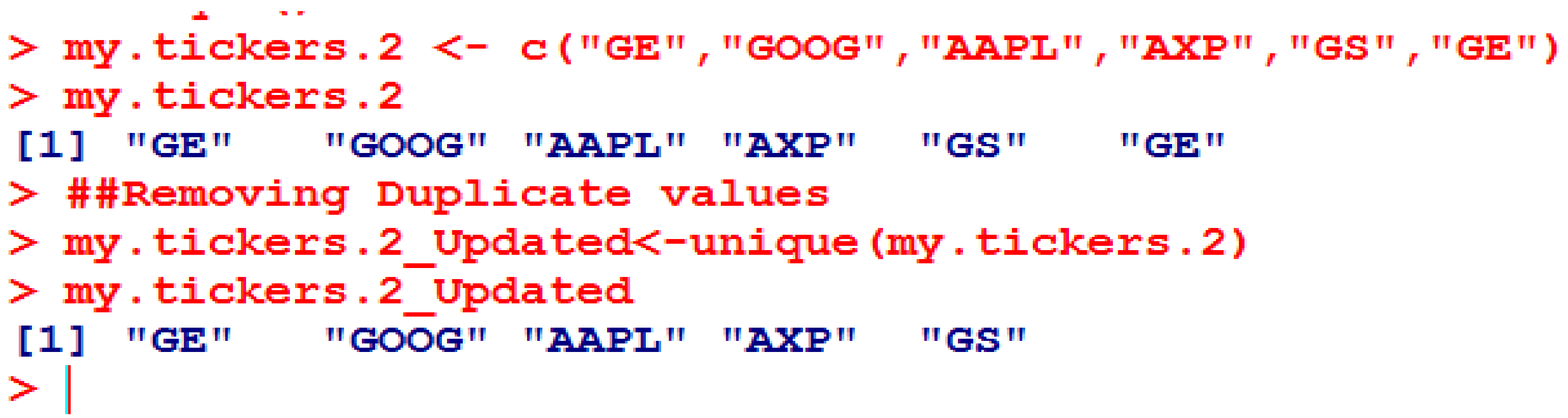


```
> ## year.enroll is a factor, so must convert to numeric first!
> table(cut(as.numeric(as.character(clinical.trail$year.enroll)),breaks=3))

(1985,1990] (1990,1994] (1994,1999] 
         31          36          33 
> table(cut(as.numeric(as.character(clinical.trail$year.enroll)),breaks=4))

(1985,1988] (1988,1992] (1992,1996] (1996,1999] 
         25          30          14          31 
> table(cut(as.numeric(as.character(clinical.trail$year.enroll)),breaks=5))

(1985,1988] (1988,1991] (1991,1993] (1993,1996] (1996,1999] 
         18          17          26          17          22 
> |
```

DATA CLEANING

- Some of the data sets contain values like 997, 998, and 999 which are not actual values there might be duplicate records in the data.

- **Finding and Removing Duplicates**
  - Data sources often contain duplicate values.
  - It's a good idea to check for duplicates in your data
  - R provides some useful functions for detecting duplicate values.



```
> my.tickers.2 <- c("GE","GOOG","AAPL","AXP","GS","GE")
> my.tickers.2
[1] "GE"   "GOOG" "AAPL" "AXP"  "GS"   "GE"
> ##Removing Duplicate values
> my.tickers.2_Updated<-unique(my.tickers.2)
> my.tickers.2_Updated
[1] "GE"   "GOOG" "AAPL" "AXP"  "GS"
> 
```

THANK YOU !!!