Merging / Appending & Aggregating Data

Contents

Data Snapshot

sal_data consist information about Employee's Basic Salary, their ID & full Name

Employee_ID	First_Name	Last_Name	Basic_Salary
E-1001	Mahesh	Joshi	16860
E-1002	Ra		
E-1004	Pr		
E-1005	Sn		
E-1007	R		
E-1008	N		
E-1009	Har		

Columns	Description	Type	Measurement	Possible values
Employee_ID	Employee ID	character	-	-
First_Name	First Name	character	-	-
Last_Name	Last Name	character	-	-
Basic_Salary	Basic Salary	numeric	Rs.	positive values

bonus_data has information of only Bonus given to Employees.

Employee_ID	Bonus
E-1001	16070
E-1003	
E-1004	
E-1006	
E-1008	
E-1010	

"Employee ID" is the common column in both datasets

Columns	Description	Type	Measurement	Possible values
Employee_ID	Employee ID	character	-	-
Bonus	Bonus amount	Numeric	Rs.	Positive values

Merging

Join (Merge) two datasets having one or more common columns with identical names.

merge() function is used to join two data sets. To perform join operations import the data sets.

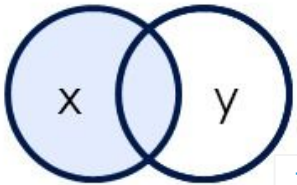
```
# Import sal_data and bonus_data
sal_data <- read.csv("sal_data.csv",header=TRUE)

bonus_data <- read.csv("bonus_data.csv",header=TRUE)
```

Types of Joins

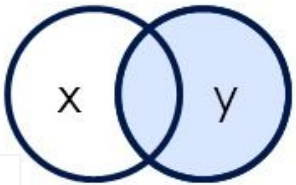
Consider sal_data =x and bonus_data=y

There are 4 types of Joins

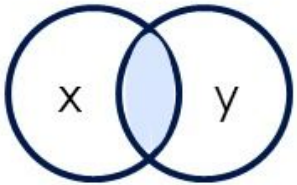
Left Join



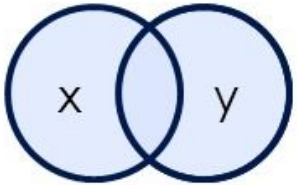
Right Join



Inner Join



Outer Join



Left Join

Left Join returns all rows from the left table, and rows with matching keys from the right table.

```
# Display all the information(including bonus) of Employees from sal_data
leftjoin<-merge(sal_data,bonus_data,by=c("Employee_ID"),all.x=TRUE)
leftjoin
```

```
# Output
  Employee_ID First_Name Last_Name Basic_Salary Bonus
1      E-1001     Mahesh     Joshi        16860 16070
2      E-1002     Rajesh     Kolte        14960    NA
3      E-1004      Priya      Jain        12670 13490
4      E-1005      Sneha     Joshi        15660    NA
5      E-1007        Ram    Kanade        15850    NA
6      E-1008      Nishi    Honrao        15950 15880
7      E-1009     Hameed     Singh        15120    NA
```

- **all.x= true** is specified to include all rows from the data set **x**(i.e. first data frame) and only those from **y**(i.e. second dataframe) that match
- **by=** is used to specify common columns.

Right Join

Right Join returns all rows from the right table, and any rows with matching keys from the left table.

```
# Display all the information of employees who are receiving bonus
rightjoin<-merge(sal_data,bonus_data,by="Employee_ID",all.y=TRUE)
rightjoin
```

```
 # Output
  Employee_ID First_Name Last_Name Basic_Salary Bonus
1      E-1001     Mahesh     Joshi        16860 16070
2      E-1004      Priya      Jain        12670 13490
3      E-1008      Nishi    Honrao        15950 15880
4      E-1003       <NA>      <NA>           NA 15200
5      E-1006       <NA>      <NA>           NA 14200
6      E-1010       <NA>      <NA>           NA 15120
```

all.y= true is specified to keep all rows from the data set **y** and only those from **x** that match

Inner Join

Inner Join returns only the rows in which the x have matching keys in the y.

```
# Display all the information about employees which are common in both
# the tables
innerjoin<-merge(sal_data,bonus_data,by=("Employee_ID"))
innerjoin
```

```
# Output
  Employee_ID First_Name Last_Name Basic_Salary Bonus
1      E-1001     Mahesh     Joshi        16860 16070
2      E-1004      Priya      Jain        12670 13490
3      E-1008      Nishi    Honrao        15950 15880
```

By default **all.x** and **all.y** is equal to **FALSE**

Outer Join

Outer Join returns all rows from x and y, join records from x which have matching keys in the y.

```
# Combine sal_data and bonus_data

outerjoin<-merge(sal_data,bonus_data,by=c("Employee_ID"),all=TRUE)
outerjoin

 # Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1002	Rajesh	Kolte	14960	NA
3	E-1004	Priya	Jain	12670	13490
4	E-1005	Sneha	Joshi	15660	NA
5	E-1007	Ram	Kanade	15850	NA
6	E-1008	Nishi	Honrao	15950	15880
7	E-1009	Hameed	Singh	15120	NA
8	E-1003	<NA>	<NA>	NA	15200
9	E-1006	<NA>	<NA>	NA	14200
10	E-1010	<NA>	<NA>	NA	15120

all= true is specified to include all rows from both data sets.

Package dplyr

- Using **`merge()`** in R on big tables can be time consuming. The join functions in the package **`dplyr`** are much faster.
- **`dplyr`** is the next iteration of package **`plyr`**, developed by Hadley Wickham and Romain Francois, focused on tools for working with data frames (hence the **`d`** in the name).
- It is built to be fast, highly expressive, and open-minded about how your data is stored.
- The package offers six different joins:
    - inner join
    - left join
    - right join
    - full join
    - semi join
    - anti join

inner_join()

- **`inner_join()`:** similar to **`merge()`** with **`all.x=F`** and **`all.y=F`**.
- It returns all rows from x where there are matching values in y, and all columns from x and y.

```
install.packages("dplyr")
library(dplyr)
```

```
inner_join(sal_data,bonus_data,by="Employee_ID")
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1004	Priya	Jain	12670	13490
3	E-1008	Nishi	Honrao	15950	15880

left_join()

- **`left_join():`** similar to **`merge()`** with **`all.x=T`**
- It returns all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns.

```
left_join(sal_data,bonus_data,by="Employee_ID")
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1002	Rajesh	Kolte	14960	NA
3	E-1004	Priya	Jain	12670	13490
4	E-1005	Sneha	Joshi	15660	NA
5	E-1007	Ram	Kanade	15850	NA
6	E-1008	Nishi	Honrao	15950	15880
7	E-1009	Hameed	Singh	15120	NA

right_join()

- **right_join:** similar to **merge()** with **all.y=TRUE**, only difference is that it sorts the result by the common column.
- It returns all rows from y, and all columns from x and y. Rows in y with no match in x will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

```
right_join(sal_data,bonus_data,by="Employee_ID")
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1003	<NA>	<NA>	NA	15200
3	E-1004	Priya	Jain	12670	13490
4	E-1006	<NA>	<NA>	NA	14200
5	E-1008	Nishi	Honrao	15950	15880
6	E-1010	<NA>	<NA>	NA	15120

full_join()

- **`full_join:`** similar to **`merge()`** with **`all=TRUE`**.
- It returns all rows and all columns from both x and y. where there are not matching values, returns NA for the one missing.

```
full_join(sal_data,bonus_data,by="Employee_ID")
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary	Bonus
1	E-1001	Mahesh	Joshi	16860	16070
2	E-1002	Rajesh	Kolte	14960	NA
3	E-1004	Priya	Jain	12670	13490
4	E-1005	Sneha	Joshi	15660	NA
5	E-1007	Ram	Kanade	15850	NA
6	E-1008	Nishi	Honrao	15950	15880
7	E-1009	Hameed	Singh	15120	NA
8	E-1003	<NA>	<NA>	NA	15200
9	E-1006	<NA>	<NA>	NA	14200
10	E-1010	<NA>	<NA>	NA	15120

data is merged keeping the original order of x

semi_join

- **semi_join:** return all rows from x where there are matching values in y, keeping just columns from x.

```
# Keep the sal_data in the same format, but only keep the records that
# also  have a match in the bonus_data
```

```
semi_join(sal_data,bonus_data,by="Employee_ID")
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary
1	E-1001	Mahesh	Joshi	16860
2	E-1004	Priya	Jain	12670
3	E-1008	Nishi	Honrao	15950

With **semi_join()** we get a similar result as with **inner_join()** but the semi_join result contains only the rows originally found in x

anti_join()

- **anti_join**: returns all rows from x where there are not matching values in y, keeping just columns from x.

```
# Display the records of employees who are not receiving bonus
anti_join(sal_data,bonus_data,by="Employee_ID")
```

```
# Output
```

	Employee_ID	First_Name	Last_Name	Basic_Salary
1	E-1002	Rajesh	Kolte	14960
2	E-1005	Sneha	Joshi	15660
3	E-1007	Ram	Kanade	15850
4	E-1009	Hameed	Singh	15120

Appending Data-
Data Snapshot

basic_salary data consist salary of each employee with it's Location & Grade.

Variables

Observations

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070

Columns	Description	Type	Measurement	Possible values
First_Name	First Name	character	-	-
Last_Name	Last Name	character	-	-
Grade	Grade	character	GR1, GR2	2
Location	Location	character	DELHI, MUMBAI	2
ba	Basic Allowance	numeric	Rs.	positive values
ms	Management Supplements	numeric	Rs.	positive values

Appending

- Append means adding cases/observations to a dataset.
- Appending two datasets using **`rbind()`** function requires both the datasets with exactly the same number of variables with exactly the same names.
- If datasets do not have the same number of variables, variables can be either dropped or created so both match.

Appending Data Sets

```
# Import the data sets and append them using rbind() function

salary_1<-read.csv("basic_salary - 1.csv",header=TRUE)
salary_2<-read.csv("basic_salary - 2.csv",header=TRUE)
rbindsalary<-rbind(salary_1,salary_2)
rbindsalary
```

```
# Output
```

	First_Name	Last_Name	Grade	Location	ba	ms
1	Alan	Brown	GR1	DELHI	17990	16070
2	Agatha	Williams	GR2	MUMBAI	12390	6630
3	Rajesh	Kolte	GR1	MUMBAI	19250	14960
4	Ameet	Mishra	GR2	DELHI	14780	9300
5	Neha	Rao	GR1	MUMBAI	19235	15200
6	Sagar	Chavan	GR2	MUMBAI	13390	6700
7	Aaron	Jones	GR1	MUMBAI	23280	13490
8	John	Patil	GR2	MUMBAI	13500	10760
9	Sneha	Joshi	GR1	DELHI	20660	NA
10	Gaurav	Singh	GR2	DELHI	13760	13220
11	Adela	Thomas	GR2	DELHI	13660	6840
12	Anup	Save	GR2	MUMBAI	11960	7880

rbind() combines the vector, matrix or data frame by rows

Quick Recap

In this session, we learnt different ways of joining two data sets using **`merge()`** and package dplyr. Here is the quick recap:

Base function in R : merge()	4 types of joins: • left_join • right join • inner join • outer join
package **`dplyr`**- recommended in case of large data as it is much faster than **`merge()`**	6 types of joins: • **`inner_join()`** • **`left_join()`** • **`right_join()`** • **`full_join()`** • **`semi_join()`** • **`anti_join()`**
Appending Data	• **`rbind()`** combines the vector, matrix or data frame by rows

Aggregating Data - Data Snapshot

basic_salary data consist salary of each employee with it's Location & Grade.

Variables

Observations

First_Name	Last_Name	Grade	Location	ba	ms
Alan	Brown	GR1	DELHI	17990	16070

Columns	Description	Type	Measurement	Possible values
First_Name	First Name	character	-	-
Last_Name	Last Name	character	-	-
Grade	Grade	character	GR1, GR2	2
Location	Location	character	DELHI, MUMBAI	2
ba	Basic Allowance	numeric	Rs.	positive values
ms	Management Supplements	numeric	Rs.	positive values

Introduction to Aggregation

Aggregating data means splitting data into subsets, computing summary statistics on each subset and displaying the results in a conveniently summarised form. Suppose a database has millions of rows. It can be aggregated and summarised on various levels depending on the type of information needed.
Aggregation is necessary to manage complexity of data.

In R, we have a built-in function **`aggregate()`** which carries out the process of taking numerous records and collapsing them into a single summary record.

```
# Import basic_sal data
salary_data<-read.csv("basic_salary.csv",header=TRUE)
```

aggregate()

```
# Calculate sum of variable ‘ms’ by variable ‘Location’
# In this example we are giving one variable and one factor
A<-aggregate(ms~Location,data=salary_data,FUN=sum)
A
```

```
# Output
```

	Location	ms
1	DELHI	45430
2	MUMBAI	75620

- ~ is used to give a formula such as **y ~ x**, where **y** are the numeric type variables to be split into groups according to the grouping **x** variables.
- **FUN=** argument takes the function to compute summary statistics.

```
# The above command can also be written as:
aggregate(salary_data$ms, by=list(salary_data$Location), FUN=sum)
```

by= takes a list of variables by which grouping is to be done

* **aggregate()** by default ignores the missing data values.

aggregate()

```
# Calculate sum of variable ‘ms’ by variables ‘Location’ and ‘Grade’
# In this example we are giving one variable and two factors
C<-aggregate(ms~Location+Grade,data=salary_data,FUN=sum)
C
```

```
# Output
  Location Grade    ms
1    DELHI   GR1 16070
2   MUMBAI   GR1 43650
3    DELHI   GR2 29360
4   MUMBAI   GR2 31970
```

Multiple factors can be added using ‘**+**’ operator.

```
# Calculate sum of variables ‘ba’ and ‘ms’ by variables ‘Location’ and ‘Grade’
# In this example we are giving multiple variables and one factor
B<-aggregate(cbind(ba,ms)~Location,data=salary_data,FUN=sum)
B
```

```
# Output
  Location     ba    ms
1    DELHI  60190 45430
2   MUMBAI 113005 75620
```

cbind() combines vector, matrix or data frame by columns

When aggregated with cbind(), it ignores the rows corresponding to the row having missing values.
So, we recommend not to use cbind()(in case of missing values) with aggregate(), you can aggregate these variables separately.

Aggregating using Package dplyr

- The package dplyr provides a well structured set of functions for manipulating data and performing typical operations with standard, easy to remember syntax. It is also very fast, even with large collections

- Major strength of dplyr is the ability to group the data by a variable or variables and then operate on the data "by group". It uses “split-apply-combine” paradigm approach i.e.split the data into groups, apply some analysis to each group, and then combine the results with the help of **`group_by()`** and **`summarize()`**.

```
# Install and load the package dplyr
install.packages("dplyr")
library(dplyr)
```

Aggregating using Package dplyr

```
# Calculate sum for variable ‘ms’ by variable ‘Location’

loc<-group_by(salary_data,Location)
dplyr_agg<-summarize(loc,sum=sum(ms,na.rm=TRUE))
dplyr_agg
```

```
 # Output
# A tibble: 2 x 2
  Location   sum
  <fct>    <int>
1 DELHI    45430
2 MUMBAI   75620
```

- **group_by()** splits the data into groups upon which some operations can be run. Multiple factors can be specified
  eg.: **group_by**(salary_data, Location, Grade)
- **summarize()** which collapses each group into a single-row summary of that group. **You can also summarise multiple variables at the same time**.
- **sum** is the name of the new column with summarised data which is specified as an argument **sum=**
- **na.rm=TRUE** ignores the missing values. If it is not specified it gives **NA** instead of 45430 because by default **na.rm=FALSE**.

Aggregating using Package data.table

- In case of large data sets, a data.frame can be limiting, the time it takes to do certain things is just too long.
- The package data.table solves this for you by reducing computing time.
- This package provide many features like fast aggregation, fast ordered joins, fast data manipulation, fast file import, etc.
- It is much faster than R's built-in function **`aggregate()`**.

```
# Install and load package data.table
install.packages("data.table")
library(data.table)
```

Aggregating using Package data.table

```
# Calculate sum for variable ‘ms’ by variable ‘Location’
# First convert dataframe salary_data into a datatable using data.table()

salary_data_DT<-data.table(salary_data)
setkey(salary_data_DT,Location)
DT_agg<-salary_data_DT[,.(sumofms=sum(ms,na.rm=TRUE)),by=Location]
DT_agg

 # Output
```

	Location	sumofms
1:	DELHI	45430
2:	MUMBAI	75620

- **setkey()** sorts the rows of a datatable by the columns provided by reference, always in ascending order. It speeds up operations on keyed datatables.
- **by=** takes the factors by which data will be aggregated. Multiple factors can be specified with **list().**
- **na.rm=TRUE** ignores the missing values. It works the same way like in **dplyr.**
- **.()** notation allows you to rename the columns inside datatable.

Both **data.table** and **dplyr** are very efficient when it comes to large data. If you’re looking for pure speed, data.table is the clear winner. However, **dplyr**’s syntax is easier than **data.table**’s.

Quick Recap

In this session, we learnt different methods of aggregating data.

Here is the quick recap:

aggregate()	• It is a built-in function of R, which aggregates the inputted **data.frame** by applying a function specified by the **FUN** argument to the column defined before **~** by the columns defined after **~**
Package **dplyr**	• It provides two functions **groupby()** and **summarise()** for aggregation of data
Package **data.table**	• It provides fast and more efficient aggregation in case of large data sets. • **data.frame** need to be converted to **data.table**