Exercise:10

Types of non-tabular data: This includes data sources such as text, audio, image, video, spatial, and IoT data.

**Time series:** is a series of data points in which each data point is associated with a timestamp. A simple example is the price of a stock in the stock market at different points of time on a given day. Another example is the amount of rainfall in a region at different months of the year. R language uses many functions to create, manipulate and plot the time series data. The data for the time series is stored in an R object called **time-series object**. It is also a R data object like a vector or data frame.

The time series object is created by using the **ts()** function.

Syntax:

The basic syntax for **ts()** function in time series analysis is −

timeseries.object.name <- ts(data, start, end, frequency)

Following is the description of the parameters used −

* **data** is a vector or matrix containing the values used in the time series.
* **start** specifies the start time for the first observation in time series.
* **end** specifies the end time for the last observation in time series.
* **frequency** specifies the number of observations per unit time.

Except the parameter "data" all other parameters are optional.

Example

Consider the annual rainfall details at a place starting from January 2012. We create an R time series object for a period of 12 months and plot it.

[Live Demo](http://tpcg.io/UZraRG)

# Get the data points in form of a R vector.

rainfall <- c(799,1174.8,865.1,1334.6,635.4,918.5,685.5,998.6,784.2,985,882.8,1071)

# Convert it to a time series object.

rainfall.timeseries <- ts(rainfall,start = c(2012,1),frequency = 12)

# Print the timeseries data.

print(rainfall.timeseries)

# Give the chart file a name.

png(file = "rainfall.png")

# Plot a graph of the time series.

plot(rainfall.timeseries)

# Save the file.

dev.off()

When we execute the above code, it produces the following result and chart −

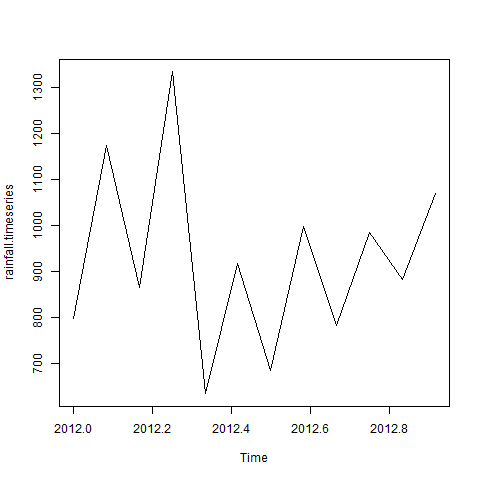
Jan Feb Mar Apr May Jun Jul Aug Sep

2012 799.0 1174.8 865.1 1334.6 635.4 918.5 685.5 998.6 784.2

Oct Nov Dec

2012 985.0 882.8 1071.0

The Time series chart −



Multiple Time Series:

We can plot multiple time series in one chart by combining both the series into a matrix.

[Live Demo](http://tpcg.io/vETvkY)

# Get the data points in form of a R vector.

rainfall1 <- c(799,1174.8,865.1,1334.6,635.4,918.5,685.5,998.6,784.2,985,882.8,1071)

rainfall2 <-

c(655,1306.9,1323.4,1172.2,562.2,824,822.4,1265.5,799.6,1105.6,1106.7,1337.8)

# Convert them to a matrix.

combined.rainfall <- matrix(c(rainfall1,rainfall2),nrow = 12)

# Convert it to a time series object.

rainfall.timeseries <- ts(combined.rainfall,start = c(2012,1),frequency = 12)

# Print the timeseries data.

print(rainfall.timeseries)

# Give the chart file a name.

png(file = "rainfall\_combined.png")

# Plot a graph of the time series.

plot(rainfall.timeseries, main = "Multiple Time Series")

# Save the file.

dev.off()

When we execute the above code, it produces the following result and chart −

Series 1 Series 2

Jan 2012 799.0 655.0

Feb 2012 1174.8 1306.9

Mar 2012 865.1 1323.4

Apr 2012 1334.6 1172.2

May 2012 635.4 562.2

Jun 2012 918.5 824.0

Jul 2012 685.5 822.4

Aug 2012 998.6 1265.5

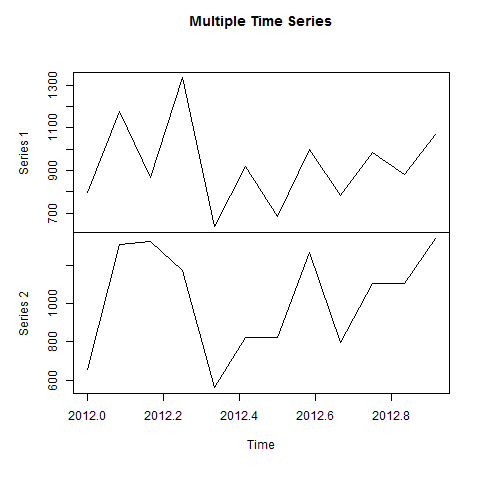
Sep 2012 784.2 799.6

Oct 2012 985.0 1105.6

Nov 2012 882.8 1106.7

Dec 2012 1071.0 1337.8

The Multiple Time series chart −



Spatial analysis is defined as the process of studying entities by examining, assessing, evaluating, and modeling spatial data features such as locations, attributes, and their relationships that reveal the geometric or geographic properties of data.

# Load the sf package

library(sf)

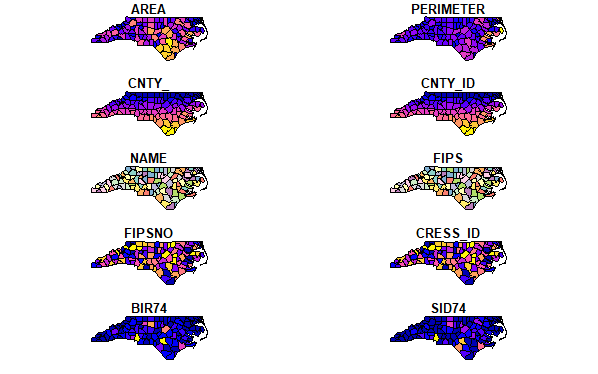
# Read a shapefile containing polygon data

world <- st\_read(system.file("shape/nc.shp", package="sf"))

# Plot the spatial data

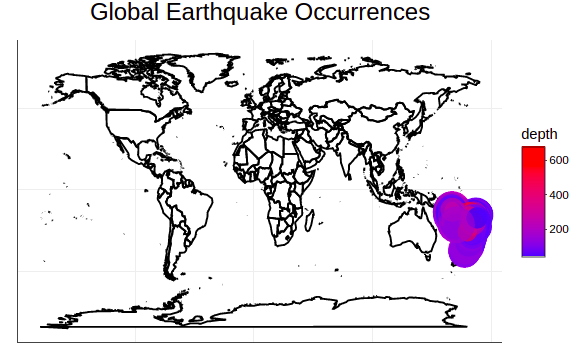
plot(world)

**Output:**



|  |
| --- |
| # Install and load necessary packages if you haven't already  install.packages("ggplot2")  install.packages("maps")  install.packages("plotly")    library(ggplot2)  library(maps)  library(plotly)    # Load earthquake data from the 'quakes' dataset  data(quakes)    # Create a basic map of earthquake occurrences  world\_map <- map\_data("world")  ggplot() +    geom\_polygon(data = world\_map, aes(x = long, y = lat, group = group),                 fill = "white", color = "black") +    geom\_point(data = quakes, aes(x = long, y = lat, size = mag,                                  color = depth), alpha = 0.7) +    scale\_size\_continuous(range = c(1, 10)) +    scale\_color\_gradient(low = "blue", high = "red") +    labs(      title = "Global Earthquake Occurrences",      subtitle = "Magnitude and Depth",      x = "",      y = ""    ) +    theme\_void() +    theme(plot.title = element\_text(hjust = 0.5, size = 18),          plot.subtitle = element\_text(hjust = 0.5, size = 14))    # Make the plot interactive using plotly  earthquake\_plot <- ggplotly()    # Display the interactive plot  earthquake\_plot |

**Output:**



Network Data:

install.packages('igraph')

install.packages('networkD3')

Creating a Network Graph with R

Let's create a simple network graph to demonstrate the basics of network analysis with R. We'll use the igraph package, which provides a simple and flexible set of utilities for network analysis.

library(igraph)

# Create a vector of relationships

relations <- c('A', 'B', 'B', 'C', 'C', 'D', 'D', 'E', 'E', 'F', 'F', 'A')

# Create a graph object

g <- graph(relations)

# Plot the graph

plot(g)

In this code, we first load the igraph library. Then, we create a vector of relationships between various nodes (A, B, C, D, E, F). The graph() function is used to create a graph object from this vector. Finally, we plot the graph using the plot() function.

**Data Transformation in R**

The data transformation in R is mostly handled by the external packages **tidyverse**and **dplyr** . These packages provide many methods to carry out the data simulations. There are a large number of ways to simulate data transformation in R. These methods are widely available using these packages, which can be downloaded and installed using the following command :

install.packages("tidyverse")

**Method 1: Using Arrange() method**

For data transformation in R, we will use The [**arrange() method**](https://www.geeksforgeeks.org/numpy-arrange-in-python/), to create an order for the sequence of the observations given. It takes a single column or a set of columns as the input to the method and creates an order for these.

The arrange() method in the tidyverse package inputs a list of column names to rearrange them in a specified order. By default, the arrange() method arranges the data in ascending order.

# Importing tidyvverse

library(tidyverse)

# Creating a data frame

data\_frame = data.frame(

col1 = c(2,4,1,7,5,3,5,8),

col2 = letters[1:8],

l3 = c(0,1,1,1,0,0,0,0))

# Assigning row names

rownames(data\_frame) <- c("r1",

"r2","r3","r4","r5","r6","r7","r8")

print("Data Frame")

print(data\_frame)

# Arranging a single column in ascending order

arr\_data\_frame <- data\_frame %>% arrange(col1)

print("Arranged Data Frame")

print(arr\_data\_frame)

**Output:**

col1 col2 col3

r1 2 a 0

r2 4 b 1

r3 1 c 1

r4 7 d 1

r5 5 e 0

r6 3 f 0

r7 5 g 0

r8 8 h 0

[1] "Arranged Data Frame"

col1 col2 col3

r3 1 c 1

r1 2 a 0

r6 3 f 0

r2 4 b 1

r5 5 e 0

r7 5 g 0

r4 7 d 1

r8 8 h 0

# Importing tidyverse

library(tidyverse)

# Creating a data frame

data\_frame = data.frame(

col1 = c(2,4,1,7,5,3,5,8),

col2 = letters[1:8],

col3 = c(0,1,1,1,0,0,0,0))

# Assigning row names

rownames(data\_frame) <- c("r1",

"r2","r3","r4","r5","r6","r7","r8")

print("Data Frame")

# Printing data frame

print(data\_frame)

# Arranging column in descending order

arr\_data\_frame <- data\_frame %>%

arrange(desc(col1))

print("Arranged Data Frame")

print(arr\_data\_frame)

**Converting Numeric value to a Factor**

For converting a numeric into a factor we use the **cut()** function. **cut()** divides the range of numeric vector(assume x) which is to be converted by cutting into intervals and codes its value (x) according to which interval they fall. Level one corresponds to the leftmost, level two corresponds to the next leftmost, and so on.

# Creating vectors

age <- c(40, 49, 48, 40, 67, 52, 53)

salary <- c(103200, 106200, 150200, 10606, 10390, 14070, 10220)

gender <- c("male", "male", "transgender",

"female", "male", "female", "transgender")

# Creating data frame named employee

employee<- data.frame(age, salary, gender)

# Creating a factor corresponding to age

# with three equally spaced levels

wfact = cut(employee$age, 3)

table(wfact)

is.factor(wfact)

**Output:**

wfact

(40,49] (49,58] (58,67]

4 2 1

[1] TRUE

# Creating vectors

age <- c(40, 49, 48, 40, 67, 52, 53)

salary <- c(103200, 106200, 150200, 10606, 10390, 14070, 10220)

gender <- c("male", "male", "transgender",

"female", "male", "female", "transgender")

# Creating data frame named employee

employee<- data.frame(age, salary, gender)

# Creating a factor corresponding to age with labels

wfact = cut(employee$age, 3, labels=c('Young', 'Medium', 'Aged'))

table(wfact)

is.factor(wfact)

**Output:**

wfact

Young Medium Aged

4 2 1

[1] TRUE

Date operations:

R programming language provides several [functions](https://www.geeksforgeeks.org/functions-in-r-programming/) that deal with date and time. These functions are used to format and convert the date from one form to another form. R provides a format function that accepts the date objects and also a format parameter that allows us to specify the format of the date we needed. R provides various format specifiers which are mentioned below in Table-

| **Specifier** | **Description** |
| --- | --- |
| %a | Abbreviated weekday |
| %A | Full weekday |
| %b | Abbreviated month |
| %B | Full month |
| %C | Century |
| %y | Year without century |
| %Y | Year with century |
| %d | Day of month (01-31) |
| %j | Day in Year (001-366) |
| %m | Month of year (01-12) |
| %D | Data in %m/%d/%y format |
| %u | Weekday (01-07) Starts on Monday |
| # today date  date<-Sys.Date()  # abbreviated Day  format(date,format="%a")    # full Day  format(date,format="%A")    # weekday  format(date,format="%u") | | |

**Output**

[1] "Sat"  
  
[1] "Saturday"  
  
[1] "6"

# today date

date<-Sys.Date()

# default format yyyy-mm-dd

date

# day in month

format(date,format="%d")

# month in year

format(date,format="%m")

# abbreviated month

format(date,format="%b")

# full month

format(date,format="%B")

# Date

format(date,format="%D")

format(date,format="%d-%b-%y")

**Output**

[1] "2022-04-02"  
[1] "02"  
[1] "04"  
[1] "Apr"  
[1] "April"  
[1] "04/02/22"  
[1] "02-Apr-22"

|  |
| --- |
| # today date  date<-Sys.Date()    # year without century  format(date,format="%y")    # year with century  format(date,format="%Y")    # century  format(date,format="%C") |

**Output**

[1] "22"  
[1] "2022"  
[1] "20"

String parsing:

# R program to split a string

# Given String

gfg < - "naga Venkata lakshmi"

# Using strsplit() method

answer < - strsplit(gfg, " ")

print(answer)

# R program to split a string

# Given String

gfg <- "nagavenkatadurga"

# Using strsplit() method

answer <- strsplit(gfg, split = "[0-9]+")

print(answer)

Exercise:11

Introduction Dirty data problems:

Dirty data can be a result of various factors, including human error, data entry mistakes, software bugs, hardware malfunctions, or problems during data migration and integration

Missing values are represented in R by the NA symbol. NA is a special value whose properties are different from other values. NA is one of the very few reserved words in R

|  |
| --- |
| x<- c(**NA**, 3, 4, **NA**, **NA**, **NA**)  is.na(x) |

**Output:**

[1] TRUE FALSE FALSE TRUE TRUE TRUE

|  |
| --- |
| x <- c(1, 2, **NA**, 3, **NA**, 4)  d <- is.na(x)  x[! d] |

**Output:**

[1] 1 2 3 4

|  |
| --- |
| x <- c(1, 2, 0 / 0, 3, **NA**, 4, 0 / 0)  x  x[! is.na(x)] |

**Output:**

[1] 1 2 NaN 3 NA 4 NaN  
  
[1] 1 2 3 4

|  |
| --- |
| # Create a data frame with 5 rows and 3 columns  data <- data.frame(    A = c(1, 2, **NA**, 4, 5),    B = c(**NA**, 2, 3, **NA**, 5),    C = c(1, 2, 3, **NA**, **NA**)  )    # View the resulting data frame  data |

**Output:**

A B C  
1 1 NA 1  
2 2 2 2  
3 NA 3 3  
4 4 NA NA  
5 5 5 NA

**Find all the missing values in the data**

|  |
| --- |
| # Finding missing values in data.  sum(is.na(data)) |

**Output:**

[1] 5

**Find all the missing values in the columns**

* R

|  |
| --- |
| # Finding missing values column wise  colSums(is.na(data)) |

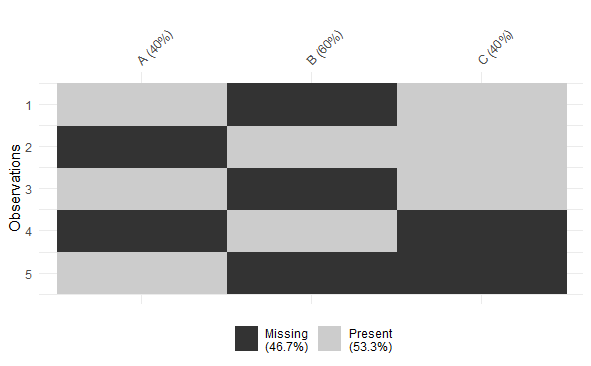
**Output:**

A B C   
1 2 2

**Visualization of missing values of a dataset:**

|  |
| --- |
| # Install and load the 'visdat' package  install.packages("visdat")  library(visdat)    # Create a data frame with missing values  data <- data.frame(    A = c(1, **NA**, 3, **NA**, 5),    B = c(**NA**, 2, **NA**, 4, **NA**),    C = c(1, 2, 3, **NA**, **NA**)  )    # Plot the missing value diagram  vis\_miss(data) |

**Output:**



In order to manipulate the data, R provides a library called dplyr which consists of many built-in methods to manipulate the data. So to use the data manipulation function, first need to import the dplyr package using ***library(dplyr)*** line of code. Below is the list of a few data manipulation functions present in dplyr package.

| **Function Name** | **Description** |
| --- | --- |
| filter() | Produces a subset of a Data Frame. |
| distinct() | Removes duplicate rows in a Data Frame |
| arrange() | Reorder the rows of a Data Frame |
| select() | Produces data in required columns of a Data Frame |
| rename() | Renames the variable names |
| mutate() | Creates new variables without dropping old ones. |
| transmute() | Creates new variables by dropping the old. |
| summarize() | Gives summarized data like Average, Sum, etc. |

**filter() method**

The filter() function is used to produce the subset of the data that satisfies the condition specified in the filter() method. In the condition, we can use conditional operators, logical operators, NA values, range operators etc. to filter out data. Syntax of filter() function is given below-

*filter(dataframeName, condition)*

**Example:**

In the below code we used filter() function to fetch the data of players who scored more than 100 runs from the “stats” data frame.

|  |
| --- |
| # import dplyr package  library(dplyr)    # create a data frame  stats <- data.frame(player=c('A', 'B', 'C', 'D'),                  runs=c(100, 200, 408, 19),                  wickets=c(17, 20, **NA**, 5))    # fetch players who scored more  # than 100 runs  filter(stats, runs>100) |

**Output**

player runs wickets

1 B 200 20

2 C 408 NA

**distinct() method**

The distinct() method removes duplicate rows from data frame or based on the specified columns. The syntax of distinct() method is given below-

*distinct(dataframeName, col1, col2,.., .keep\_all=TRUE)*

**Example:**

Here in this example, we used distinct() method to remove the duplicate rows from the data frame and also remove duplicates based on a specified column.

|  |
| --- |
| # import dplyr package  library(dplyr)    # create a data frame  stats <- data.frame(player=c('A', 'B', 'C', 'D', 'A', 'A'),                  runs=c(100, 200, 408, 19, 56, 100),                  wickets=c(17, 20, **NA**, 5, 2, 17))    # removes duplicate rows  distinct(stats)    #remove duplicates based on a column  distinct(stats, player, .keep\_all = **TRUE**) |

**Output**

player runs wickets

1 A 100 17

2 B 200 20

3 C 408 NA

4 D 19 5

5 A 56 2

player runs wickets

1 A 100 17

2 B 200 20

3 C 408 NA

4 D 19 5

**arrange() method**

In R, the arrange() method is used to order the rows based on a specified column. The syntax of arrange() method is specified below-

*arrange(dataframeName, columnName)*

**Example:**

In the below code we ordered the data based on the runs from low to high using arrange() function.

|  |
| --- |
| # import dplyr package  library(dplyr)    # create a data frame  stats <- data.frame(player=c('A', 'B', 'C', 'D'),                  runs=c(100, 200, 408, 19),                  wickets=c(17, 20, **NA**, 5))    # ordered data based on runs  arrange(stats, runs) |

**Output**

player runs wickets

1 D 19 5

2 A 100 17

3 B 200 20

4 C 408 NA

**select() method**

The select() method is used to extract the required columns as a table by specifying the required column names in select() method. The syntax of select() method is mentioned below-

*select(dataframeName, col1,col2,…)*

**Example:**

Here in the below code we fetched the player, wickets column data only using select() method.

|  |
| --- |
| # import dplyr package  library(dplyr)    # create a data frame  stats <- data.frame(player=c('A', 'B', 'C', 'D'),                  runs=c(100, 200, 408, 19),                  wickets=c(17, 20, **NA**, 5))    # fetch required column data  select(stats, player,wickets) |

**Output**

player wickets

1 A 17

2 B 20

3 C NA

4 D 5

**rename() method**

The rename() function is used to change the column names. This can be done by the below syntax-

*rename(dataframeName, newName=oldName)*

**Example:**

In this example, we change the column name “runs” to “runs\_scored” in stats data frame.

|  |
| --- |
| # import dplyr package  library(dplyr)    # create a data frame  stats <- data.frame(player=c('A', 'B', 'C', 'D'),                  runs=c(100, 200, 408, 19),                  wickets=c(17, 20, **NA**, 5))    # renaming the column  rename(stats, runs\_scored=runs) |

**Output**

player runs\_scored wickets

1 A 100 17

2 B 200 20

3 C 408 NA

4 D 19 5

**mutate() & transmute() methods**

These methods are used to create new variables. The mutate() function creates new variables without dropping the old ones but transmute() function drops the old variables and creates new variables. The syntax of both methods is mentioned below-

*mutate(dataframeName, newVariable=formula)*

*transmute(dataframeName, newVariable=formula)*

**Example:**

In this example, we created a new column avg using mutate() and transmute() methods

|  |
| --- |
| # import dplyr package  library(dplyr)    # create a data frame  stats <- data.frame(player=c('A', 'B', 'C', 'D'),                  runs=c(100, 200, 408, 19),                  wickets=c(17, 20, 7, 5))    # add new column avg  mutate(stats, avg=runs/4)    # drop all and create a new column  transmute(stats, avg=runs/4) |

**Output**

player runs wickets avg

1 A 100 17 25.00

2 B 200 20 50.00

3 C 408 7 102.00

4 D 19 5 4.75

avg

1 25.00

2 50.00

3 102.00

4 4.75

Here mutate() functions adds a new column for the existing data frame without dropping the old ones where as transmute() function created a new variable but dropped all the old columns.

**summarize() method**

Using the summarize method we can summarize the data in the data frame by using aggregate functions like sum(), mean(), etc. The syntax of summarize() method is specified below-

*summarize(dataframeName, aggregate\_function(columnName))*

**Example:**

In the below code we presented the summarized data present in the runs column using summarize() method

|  |
| --- |
| # import dplyr package  library(dplyr)    # create a data frame  stats <- data.frame(player=c('A', 'B', 'C', 'D'),                  runs=c(100, 200, 408, 19),                  wickets=c(17, 20, 7, 5))    # summarize method  summarize(stats, sum(runs), mean(runs)) |

**Output**

sum(runs) mean(runs)

1 727 181.75