1) Explain the process of plotting a graph using different layers of ggplot2 in detail

- ggplot2 is an R package used for statistical computing and data representation using data visualization.
- It follows underlying graphics called **Grammar of Graphics** which includes certain rules and independent components which can be used to represent data in various formats.
- **Data:** The element is the data set itself
- **Aesthetics:** The data is to map onto the Aesthetics attributes such as x-axis, y-axis, color, fill, size, labels, alpha, shape, line width, line type
- **Geometrics:** How our data being displayed using point, line, histogram, bar, boxplot
- **Facets:** It displays the subset of the data using Columns and rows
- **Statistics:** Binning, smoothing, descriptive, intermediate
- **Coordinates:** the space between data and display using Cartesian, fixed, polar, limits

  **Themes:** Non-data link

**Data Layer:**

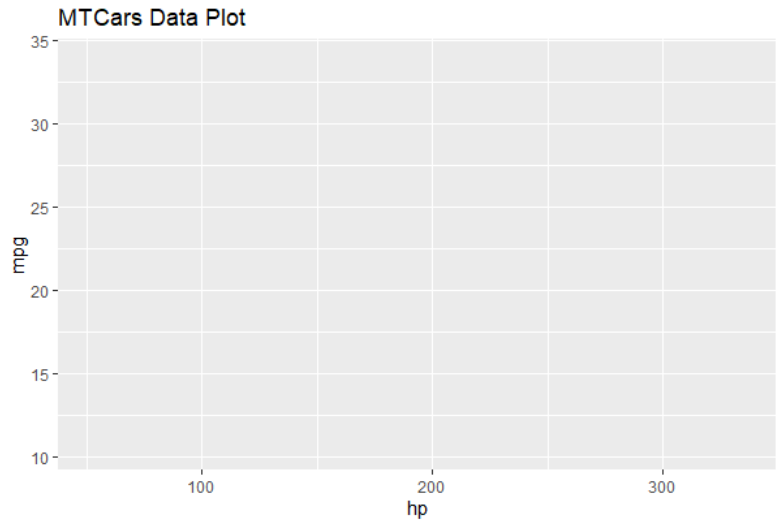
In the data Layer we define the source of the information to be visualize, let's use the mtcars dataset in the ggplot2

```
library(ggplot2)
library(dplyr)

ggplot(data = mtcars) +
  labs(title = "MTCars Data Plot")
```



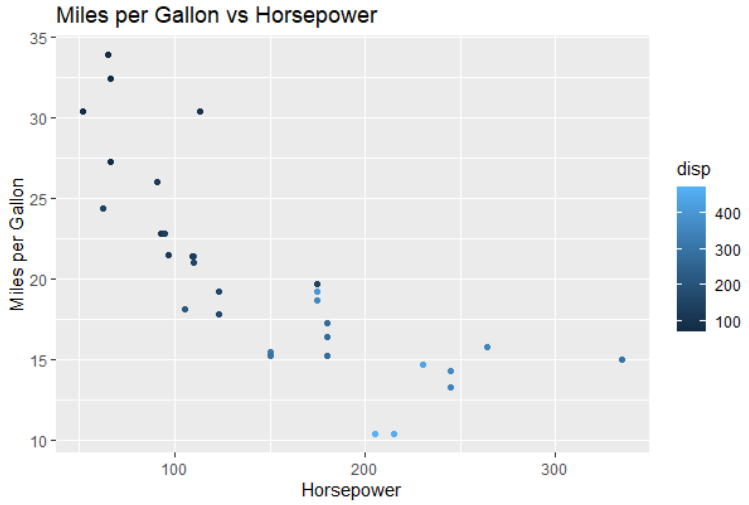

**Aesthetic Layer:**

```
# Aesthetic Layer
ggplot(data = mtcars, aes(x = hp, y = mpg, col = disp))+
 labs(title = "MTCars Data Plot")
```



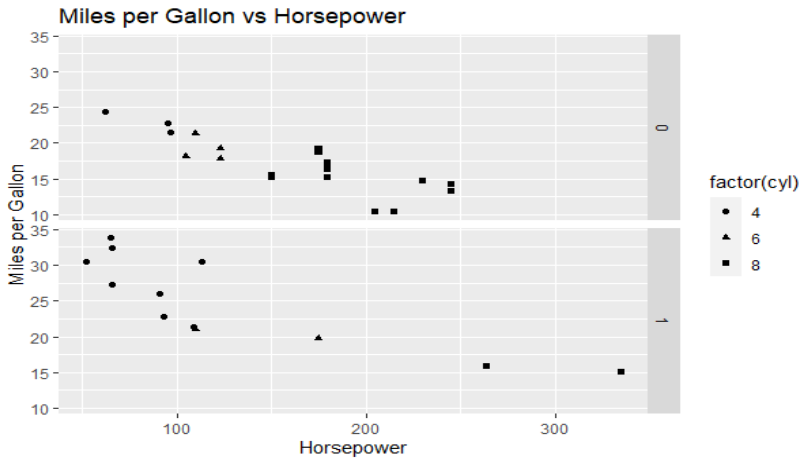

**Geometric layer**

```
# Geometric layer
ggplot(data = mtcars, aes(x = hp, y = mpg, col = disp)) +
  geom_point() +
  labs(title = "Miles per Gallon vs Horsepower",
       x = "Horsepower",
       y = "Miles per Gallon")
```

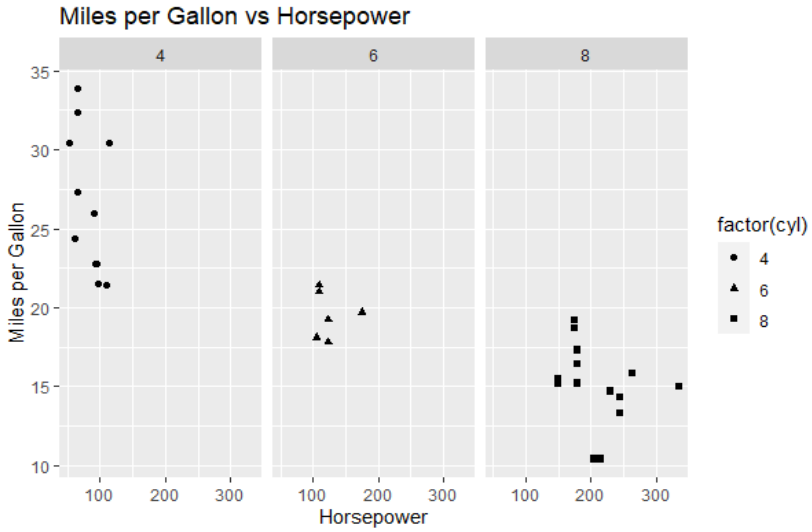



**Facet Layer:**

```
# Facet Layer
# Separate rows according to transmission type
p <- ggplot(data = mtcars, aes(x = hp, y = mpg, shape = factor(cyl))) + geom_point()

p + facet_grid(am ~ .) +
  labs(title = "Miles per Gallon vs Horsepower",
       x = "Horsepower",
       y = "Miles per Gallon")
```

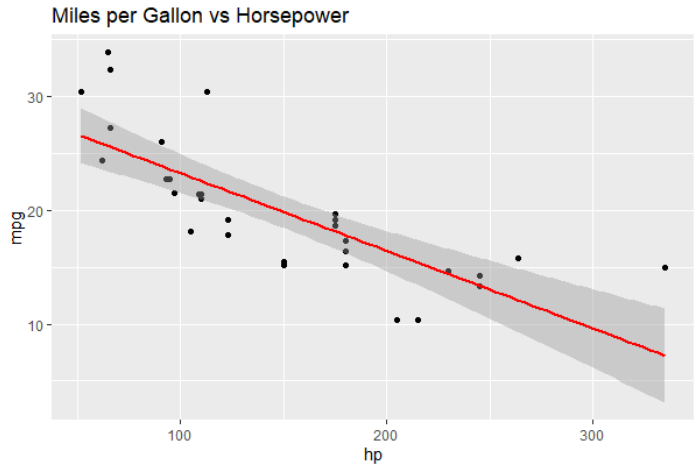



```
# Separate columns according to cylinders
p <- ggplot(data = mtcars, aes(x = hp, y = mpg, shape = factor(cyl))) + geom_point()

p + facet_grid(. ~ cyl) +
  labs(title = "Miles per Gallon vs Horsepower",
       x = "Horsepower",
       y = "Miles per Gallon")
```



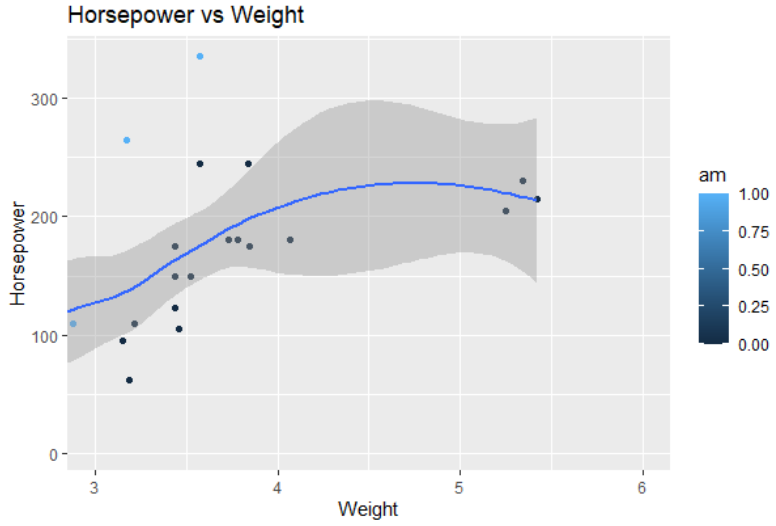

## Statistics layer

```
ggplot(data = mtcars, aes(x = hp, y = mpg)) +
  geom_point() +
  stat_smooth(method = lm, col = "red") +
  labs(title = "Miles per Gallon vs Horsepower")
```



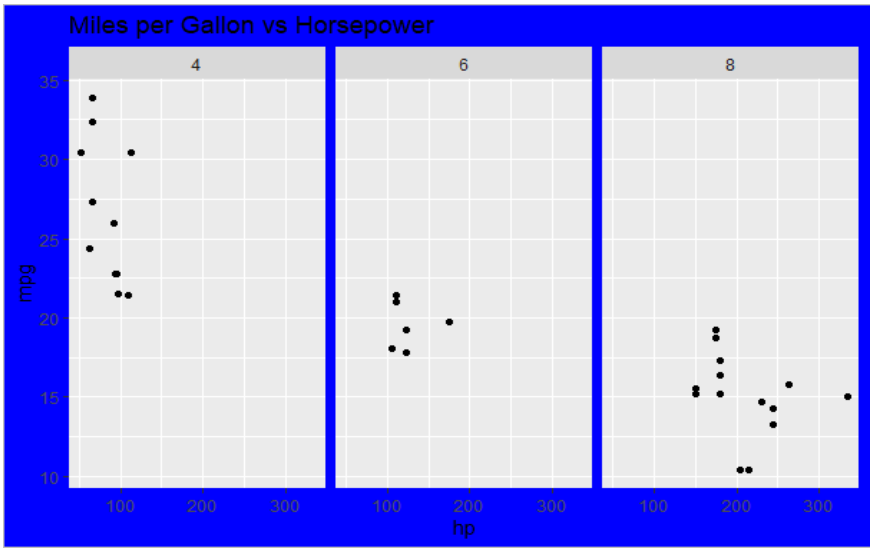

**Coordinates layer:**

```
# Add coord_cartesian() to proper zoom in
ggplot(data = mtcars, aes(x = wt, y = hp, col = am)) +
                geom_point() + geom_smooth() +
                coord_cartesian(xlim = c(3, 6))
```




**Theme Layer:**

```
gplot(data = mtcars, aes(x = hp, y = mpg)) +
  geom_point() +
  facet_grid(. ~ cyl) +
  theme(plot.background = element_rect(fill = "blue", colour = "gray")) +
  labs(title = "Miles per Gallon vs Horsepower"
```


Miles per Gallon vs Horsepower
4
6
8
mpg
35
30
25
20
15
10
100
200
300
hp

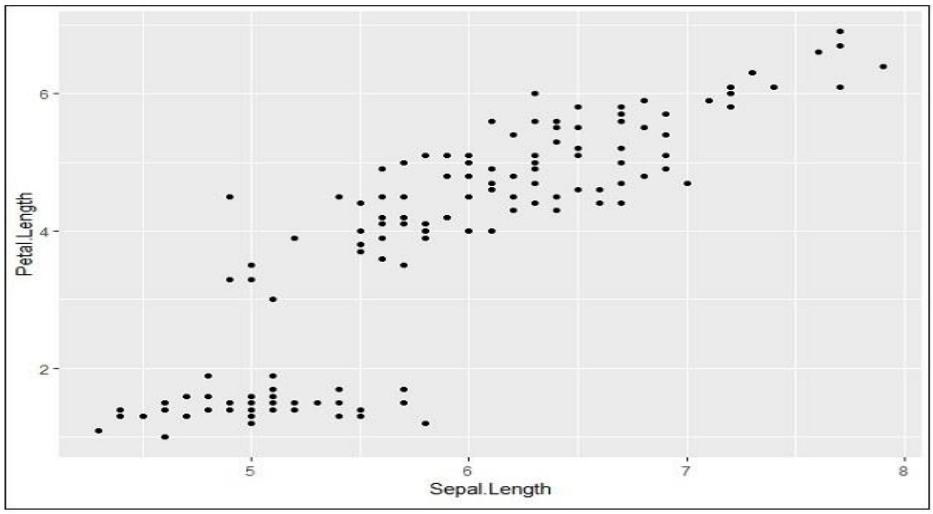
2) Explain how to create following plots using ggplot2 with example
a) jitter plot b) scatter plot

A)

- Scatter Plots are similar to line graphs which are usually used for plotting. The scatter plots show how much one variable is related to another.
- The relationship between variables is called as correlation which is usually used in statistical methods.
- We will use the same dataset called "Iris" which includes a lot of variation between each variable. This is famous dataset which gives measurements in centimeters of the variables sepal length and width with petal length and width for 50 flowers from each of 3 species of iris.

Creating Basic Scatter Plot

# Basic Scatter Plot
> ggplot(iris, aes(Sepal.Length, Petal.Length)) + geom_point()




Jitter Plots

- A **jitterplot** is a type of scatter plot used to display the distribution of a set of numerical data points.
- Jitter is nothing but a random value that is assigned to dots to separate them
- This variation helps prevent symbols from overlapping and makes it easier to see the distribution of data points in cases there is high density of points in certain areas of the plot.
- It is a useful way of handling overplotting .
- If you have a densely populated plot, a jitterplot can make your visualization easier to understand

The syntax to draw a ggplot jitter in R Programming is

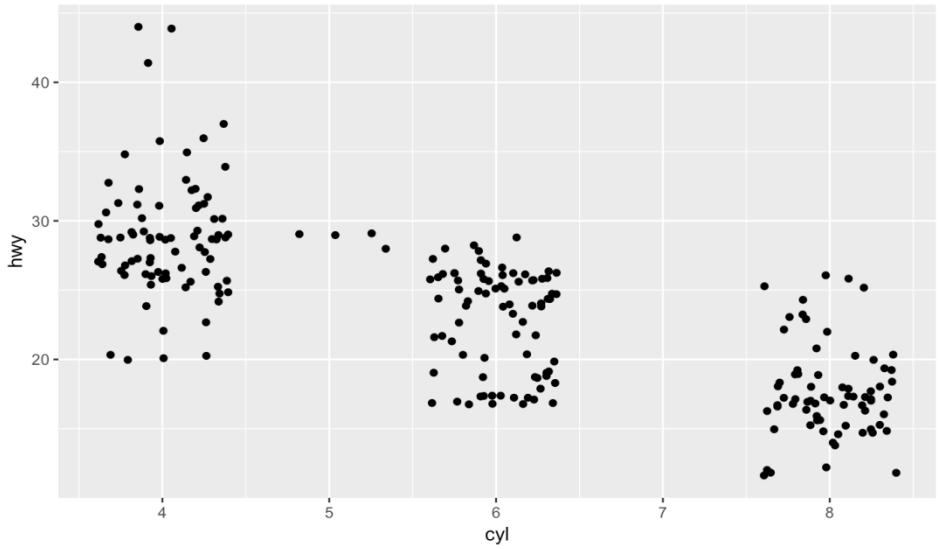
**geom_jitter(data = NULL, width = NULL, height = NULL)**

and the complex syntax behind this Jitter is:

**geom_jitter(mapping = NULL, data = NULL, stat = "identity",**
**width = NULL, height = NULL, position = "jitter", ...,**
**na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)**

### Create R ggplot2 Jitter

**ggplot(mpg, aes(cyl, hwy)) + geom_jitter()**




3) Explain the process of generating the Word Clouds in R

**Word Cloud** is a data visualization technique used for representing text data in which the size of each word indicates its frequency or importance

### The 5 main steps to create word clouds in R

### Step 1: Create a text file

- Copy and paste the text in a plain text file (e.g : ml.txt)

- Save the file

## Step 2 : Install and load the required packages

# Install

install.packages("tm")  # for text mining

install.packages("SnowballC")  # for text stemming

install.packages("wordcloud")  # word-cloud generator

install.packages("RColorBrewer")  # color palettes

# Load

library("tm")

library("SnowballC")

library("wordcloud")

library("RColorBrewer")

**Step 3 : Text mining**

**load the text**

The text is loaded using **Corpus()** function from **text mining** (tm) package. Corpus is a list of a document (in our case, we only have one document).

1. **We start by importing the text file created in Step 1**

To import the file saved locally in your computer, type the following R code. You will be asked to choose the text file interactively.

text <- readLines(file.choose())

**# Read the text file from internet**

filePath <- "http://www.sthda.com/sthda/RDoc/example-files/martin-luther-king-i-have-a-dream-speech.txt"

text <- readLines(filePath)

2. **Load the data as a corpus**

```
# Load the data as a corpus
docs <- Corpus(VectorSource(text))
```

3. **Inspect the content of the document**

```
inspect(docs)
```

*Text transformation*

Transformation is performed using **tm_map()** function to replace, for example, special characters from the text.
Replacing "/", "@" and "|" with space:

```
toSpace <- content_transformer(function (x , pattern ) gsub(pattern, " ", x
))
docs <- tm_map(docs, toSpace, "/")
docs <- tm_map(docs, toSpace, "@")
docs <- tm_map(docs, toSpace, "\\|")
```

**Cleaning the text**

the **tm_map()** function is used to remove unnecessary white space, to convert the text to lower case, to remove common stopwords like 'the', "we".

The R code below can be used to clean your text :

```
# Convert the text to lower case
docs <- tm_map(docs, content_transformer(tolower))
# Remove numbers
docs <- tm_map(docs, removeNumbers)
# Remove english common stopwords
docs <- tm_map(docs, removeWords, stopwords("english"))
# Remove your own stop word
# specify your stopwords as a character vector
docs <- tm_map(docs, removeWords, c("blabla1", "blabla2"))
# Remove punctuations
docs <- tm_map(docs, removePunctuation)
# Eliminate extra white spaces
docs <- tm_map(docs, stripWhitespace)
# Text stemming
# docs <- tm_map(docs, stemDocument
```

## Step 4 : Build a term-document matrix

Document matrix is a table containing the frequency of the words. Column names are words and row names are documents. The function *TermDocumentMatrix()* from **text mining** package can be used as follow :

```
dtm <- TermDocumentMatrix(docs)
m <- as.matrix(dtm)
v <- sort(rowSums(m),decreasing=TRUE)
d <- data.frame(word = names(v),freq=v)
head(d, 10)
             word freq
will         will   17
```

```
freedom   freedom   13
ring         ring   12
day           day   11
dream       dream   11
let           let   11
every       every    9
able         able    8
one           one    8
together together    7
```

### Step 5 : Generate the Word cloud

The importance of words can be illustrated as a **word cloud** as follow :

```
set.seed(1234)
wordcloud(words = d$word, freq = d$freq, min.freq = 1,
          max.words=200, random.order=FALSE, rot.per=0.35,
          colors=brewer.pal(8, "Dark2"))
```

Output:


4) Write a short note on:

a) waffle charts in R

b) word cloud in R

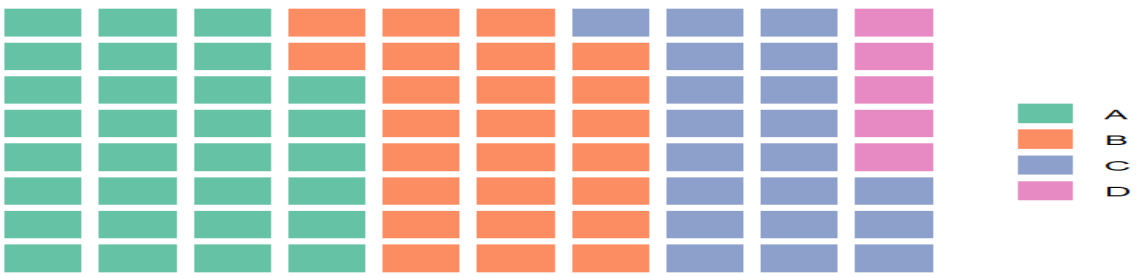
- A waffle chart shows progress towards a target or a completion percentage.
- Waffle Charts are a great way of visualizing data in relation to a whole, to highlight progress against a given threshold, or when dealing with populations too varied for pie charts.

- A lot of times, these are used as an alternative to the pie charts. It also has a niche for showing parts-to-whole contribution. It doesn't misrepresent or distort a data point (which a pie chart is sometimes guilty of doing).
- Waffle Charts are mainly used when composing parts of a whole, or when comparing progress against a goal
- Waffle charts are also known as Squared Pie Charts.
- The waffle package function contains a function of the same name that can be used to create waffle charts, also known as square pie charts or gridplots, based on ggplot2.
- For example, you might want a Waffle Chart when plotting how the expenses of a company are composed by each type of expense, or when classifying percentages of a population at a given moment.

**Most basic waffle chart**

To create a basic waffle plot **pass a vector containing the count for each group to the function**. The number of rows of the plot can be selected with rows (defaults to 10). Choose a value according to your data.

```
# install.packages("waffle", repos = "https://cinc.rud.is")
library(waffle)
# Vector
x <- c(30, 25, 20, 5)
# Waffle chart
waffle(x, rows = 8)
```




**Use a named vector to change the legend names**

5) Explain Important Functions of plotly package in R.

**Important Functions:**

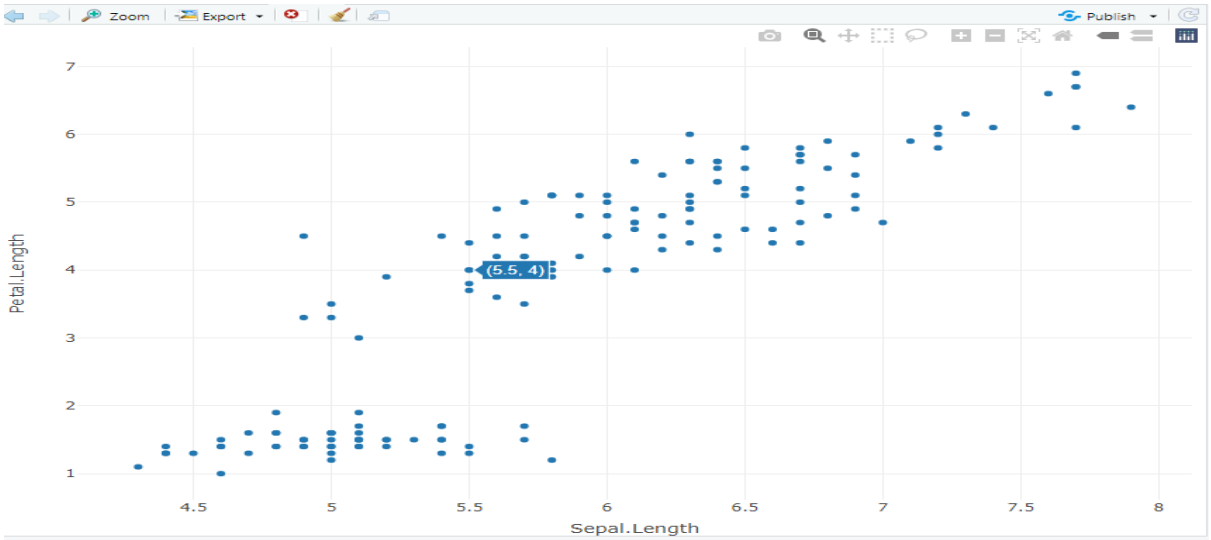
**1) plot_ly():**

- It basically initiates a plotly visualization.
- This function maps R objects to plotly.js, an (MIT licensed) web-based interactive charting library.
- It provides abstractions for doing common things and sets some different defaults to make the interface feel more 'R-like' (i.e., closer to plot() and ggplot2::qplot()).

***Syntax:***
*plot_ly(data = data.frame(), …, type = NULL, name, color, colors = NULL, alpha = NULL, stroke, strokes = NULL, alpha_stroke = 1, size, sizes = c(10, 00), span, spans = c(1, 20), symbol, symbols = NULL, linetype, linetypes = NULL, split, frame, width = NULL, height = NULL, source = "A")*

Basic plots

```
#import the iris data for example
data(iris)
fig <- plot_ly(data = iris, x = ~Sepal.Length, y = ~Petal.Length)
fig
```

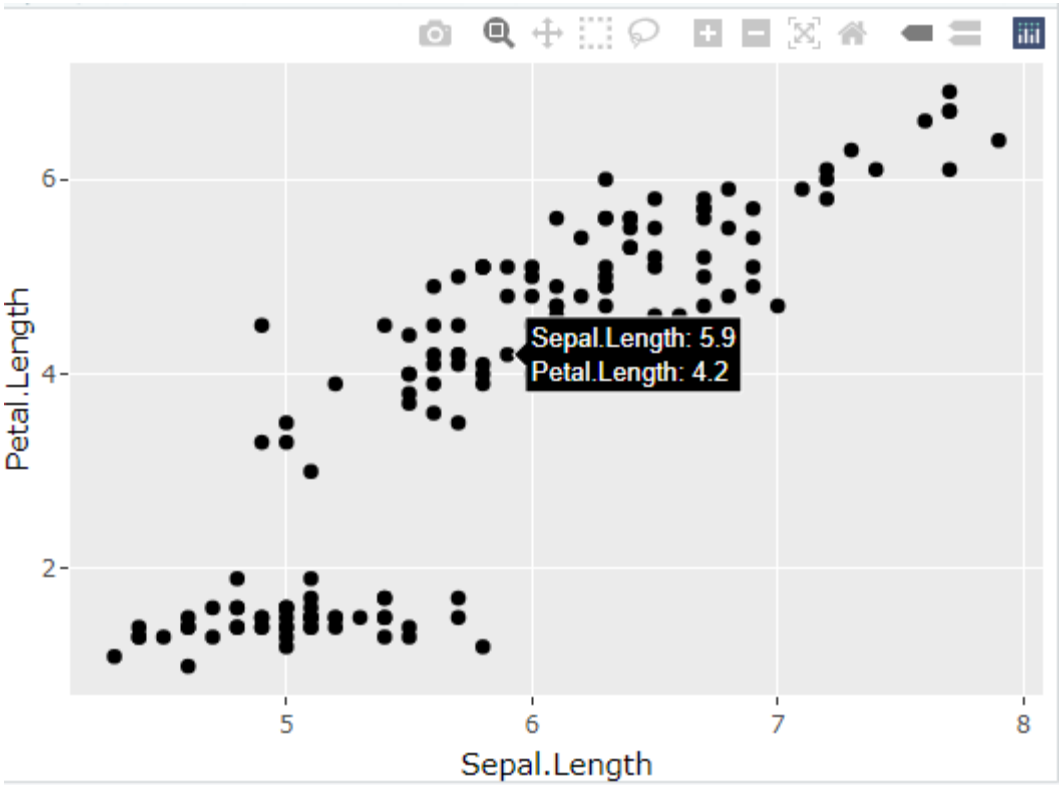



You can see that with the plot you can scroll over the data point and see the value appear in the box next to the point. You can customize what information you want to appear in these pop-up boxes and how it will appear.

With ggplot2

This same plot can also be created in ggplot2 and saved as an object. Then the plot can be made interactive using the ggplotly function. (Note that **not all features will work** when using ggplotly so you sometimes have to do some trial and error)

```
fig2<- ggplot(data=iris, aes(x=Sepal.Length, y=Petal.Length))+
 geom_point()


ggplotly(fig2)
```




2) **plotly_build()**

- **This generic function creates the list object sent to plotly.js for rendering.**
- **Using this function can be useful for overriding defaults or for debugging rendering errors.**

***Syntax:*** *plotly_build(p, registerFrames = TRUE)*

**Arguments**

**P:** a ggplot object, or a plotly object, or a list.

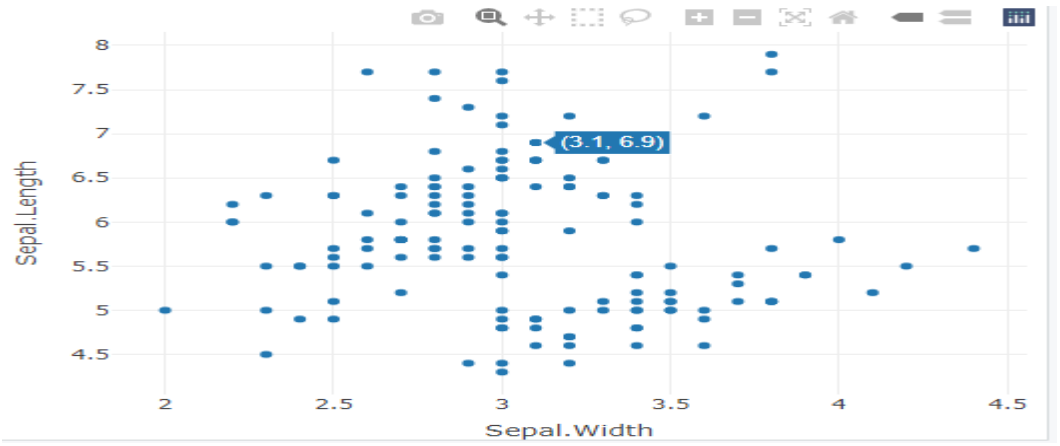
**registerFrames:** should a frame trace attribute be interpreted as frames in an animation?

```
# import plotly library
library(plotly)

# create plotly visualisation
p <- plot_ly(iris, x = ~Sepal.Width,
             y = ~Sepal.Length)

# structure of plotly using
# plotly_build
str(plotly_bulid(p, registerFrames = TRUE))
```

3) **layout():** Modify the layout of a plotly visualization

***Syntax:*** *layout(p, …, data = NULL)*

```
# import plotly library
library(plotly)

# create plotly visualisation
p <- plot_ly(iris, x = ~Sepal.Width,
             y = ~Sepal.Length)
layout(p, data = NULL)
```

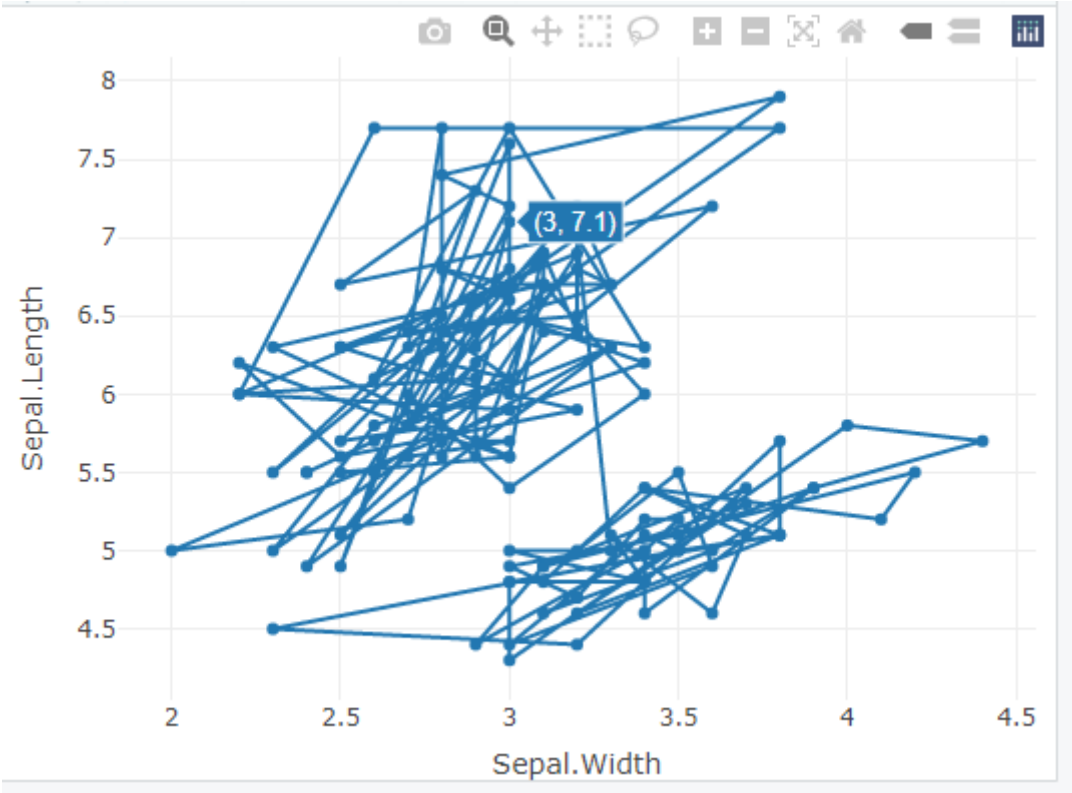



4) **add_trace:** Add trace(s) to a plotly visualization

***Syntax:*** *add_trace(p, …, data = NULL, inherit = TRUE)*

```
# import plotly library
library(plotly)
```

```
# create plotly visualisation
p <- plot_ly(iris, x = ~Sepal.Width,
                   y = ~Sepal.Length)

# adding trace (lines) to plotly
# visualisation
add_trace(p, type = "scatter",
          mode = "markers+lines")
```




5) **animation_opts()**

- Provides animation configuration options. Animations can be created by either using the frame argument in plot_ly() or frame ggplot2 aesthetic in ggplotly().
- By default, animations populate a play button and slider component for controlling the state of the animation

**Syntax:**

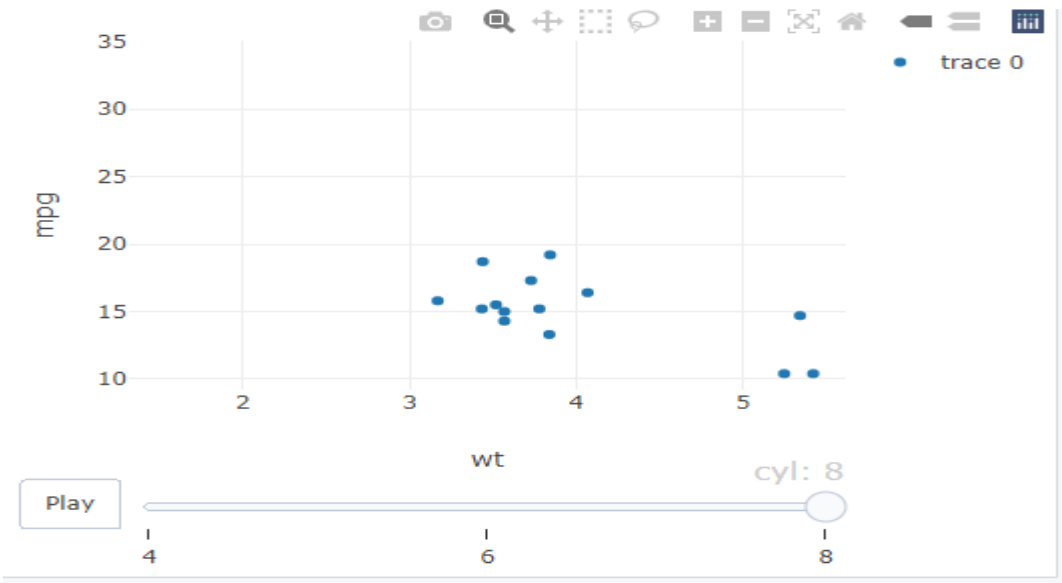
animation_opts(p, frame = 500, transition = frame, easing = “linear”, redraw = TRUE, mode = “immediate”)

animation_slider(p, hide = FALSE, …)

animation_button(p, …, label)

**Example:**

```
# import plotly library
library(plotly)

plot_ly(mtcars, x = ~wt, y = ~mpg,
        frame = ~cyl) %>%
animation_opts(transition = 0)
```



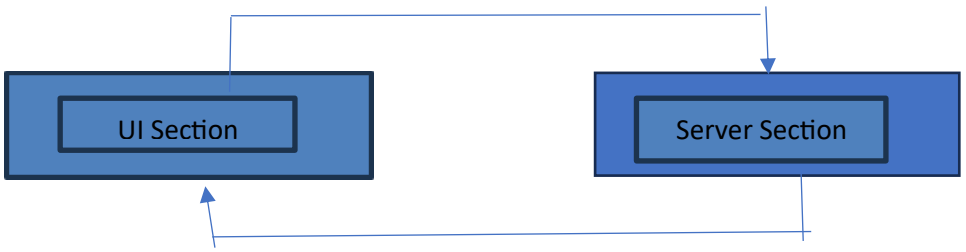
6) Explain the structure of shiny app in detail

## Structure of a Shiny App

- Shiny apps are contained in a single script called app.R.
- The script app.R lives in a directory (for example, newdir/) and the app can be run with runApp("newdir").

**app.R has three components:**

- a user interface object(ui.R)
- a server function(server.R)
- a call to the shinyApp function




```
library(shiny)

ui <-
 fluidPage(

 )

server <-
 function(input, output) {

 }

shinyApp(ui = ui, server = server)
```

## UI Section:

- The user interface (ui) object controls the layout and appearance of your app.
- The UI section of your shiny application is where you define what your application will look like.
- This is where you write your shiny functions and R code, which shiny will then take care of translating into html and CSS code that today's modern web browsers can read and render into a beautiful web application.
- For example, the function `shiny::fluidPage` is often used in the UI section to define the outer most layer of the UI.
- It creates a web page containing both rows and columns which is fluid, meaning it scales to fill all available browser width.
- `fluidpage()` is R code, but once you initiate the rendering process from the console with the `runApp()` function or by clicking the "Run App" button in the RStudio IDE, it will get converted into HTML and CSS code that defines the layout of your application.

Layout Options

**Sidebar Layout:**

One of the most common application layouts is the Sidebar Layout, which has a skinny sidebar on the left hand-side (by default) where input controls are usually added and a main panel on the right-hand side which typically displays the results.

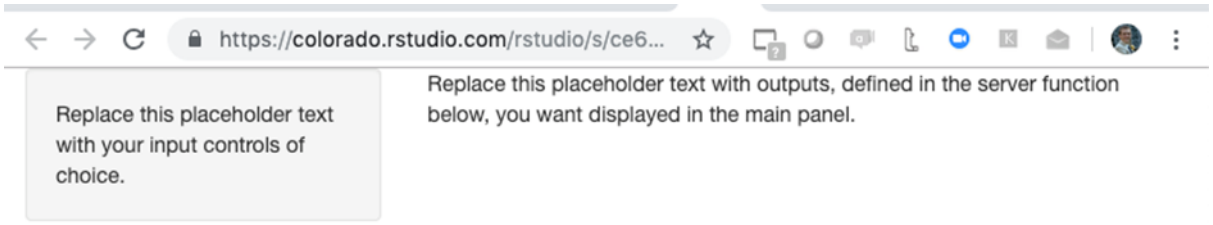
The `sidebarLayout` functions contains 2 required arguments, `sidebarPanel` and `mainPanel`, which are functions you can use to define the input controls for the sidebar and the output content you went to display in the main panel, respectively:

**Grid Layout:**

This type of layout gives you a bit more control of how to arrange your page. First, use the function `fluidRow` to add rows to your application and then the function `column` to create columns within those rows. These functions allow you to create any grid layout you like, including even grids within grids

```r
# Minimum viable example of a shiny app with a sidebar layout

library(shiny)

# Define UI
ui <- fluidPage(
  # Sidebar Layout
  sidebarLayout(
    # Sidebar Layout Elements
    sidebarPanel = sidebarPanel("Replace this placeholder text with your input controls of choice."),
    mainPanel = mainPanel(
      "Replace this placeholder text with outputs, defined in the server function below,
                    you want displayed in the main panel."
    )
      ))
# Server function
server <- function(input, output) {
 }
# Run the application
shinyApp(ui = ui, server = server)
```

**code generates the application below. As you can see, it is made up of a small sidebar on the left and a main panel on the right:**



## Server Section:

- The server section of your shiny application is where the computation takes place.
- The server function, the 2nd part of your Shiny application after the UI, is where most of your computation takes place
- It is in the server function where you will use the values users inputted via the control widgets to do things like wrangle your data and create outputs like plots
- These outputs will be saved to a special object called `output` which you can later reference in your UI in order to display them in your Shiny application.
- Any code you write for transforming data or creating visualizations will most likely be in this section.

It is useful to remember the following 4 things when creating a new application output:

1. In the server function, access user input values via the variable `input$inputID`
2. In the server function, build your output (e.g. plot, table, text, etc.) using one of the `render` functions (i.e. `renderPlot`, `renderDataTable`, `renderText`, etc.), choosing the function that matches your output type. Make sure to assign that output to the `output` object.
3. In the UI section, access your output via the variable `output$output_name`
4. In the UI section, display your output using one of the corresponding *Output functions based on your output's type (i.e. `plotOutput`, `tableOutput`, `textOutput`, etc.)

7) Explain the user interface section and server section of shiny app in detail.

8) Give the syntax for following functions of shiny package in R:

a)actionButton() b)checkboxGroupInput()

c)textInput() d**) textOutput()**

**1) actionButton():** It creates an action button or a link. Their initial value is zero, and increments by one each time it is pressed.

***Syntax:***

*actionButton(inputId, label, icon = NULL, width = NULL, ...)*
*actionLink(inputId, label, icon = NULL, ...)*

**2)checkboxGroupInput():**

It creates a group of checkboxes that can be used to toggle multiple choices independently. The server will receive the input as a character vector of the selected values.

***Syntax:***

*checkboxGroupInput(inputId, label, choices = NULL, selected = NULL, inline = FALSE, width = NULL, choiceNames = NULL, choiceValues = NULL)*

3) **textInput():** It creates a text input label.

***Syntax:***

*textInput(inputId, label, value = "", width = NULL, placeholder = NULL)*

4) **textOutput():**

It creates a text output element. Render a reactive output variable as text within an application page.

***Syntax:***

*textOutput(outputId, container = if (inline) span else div, inline = FALSE)*

**Example Program**

```
library(shiny)

# Define UI ----

ui <- fluidPage(

 titlePanel(" widgets"),

  fluidRow(
```

```
      column(3,
        h3("Buttons"),
        actionButton("action", "Action"),
        br(),
        br(),
        submitButton("Submit")),
      column(3,
        checkboxGroupInput("checkGroup",
                           h3("Checkbox group"),
                           choices = list("Choice 1" = 1,
                                          "Choice 2" = 2,
                                          "Choice 3" = 3),
                           selected = 1)),
      column(3,
        textInput("text", h3("Text input"),
                  value = "Enter text..."))
 ) )
 # Define server logic ----
server <- function(input, output) {
 }
# Run the app ----
shinyApp(ui = ui, server = server)
```

**9) a)** Write and explain the different packages used for creating maps in R.

b) Explain Plotting Simple Features (sf) with Plot

There are plenty of packages in R that can be used to make maps, like

- **Leaflet**:
- **Tmap**:
- **Mapview:**

- **Maps:**
- **ggplot:**
- **spplot,** etc.

Each of the packages has its own advantages and disadvantages. But all of them have the common goal of making it easy to create maps and visualize geospatial data.

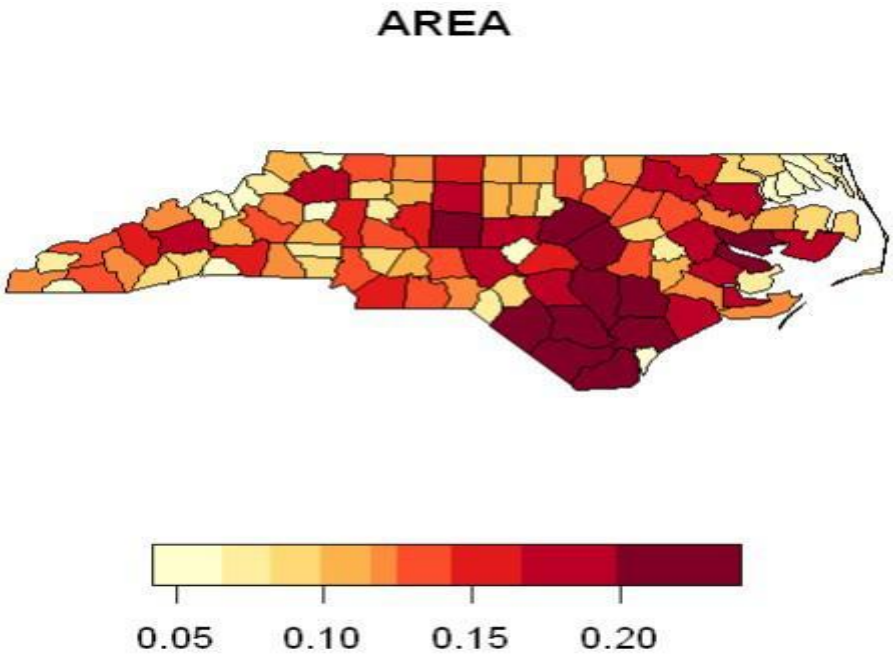
To create a map with R, the first thing you need to have is data that includes geospatial information, such as latitude and longitude coordinates or shapefiles then from that you can use packages like **rgdal** to read and manipulate these data, and finally then use one of the mapping packages to visualize the data.

Another equally important notion in making maps with R is the usage of projections. A projection is a method to represent the earth's surface on a flat map and there are many different projections that can be used for different purposes. Some projection preserves area or angles while others preserve distances or directions. While choosing a projection for a map it's important to consider what aspect of the earth's surface you want to emphasize or preserve.

**Plotting Simple Features (sf) with Plot**

- We've used the "sf" and "RColorBrewer" libraries.
- Load a shapefile "nc" which contains information on the counties in North Carolina
- plot a simple feature using the "plot" function from the "sf" package.
- We specifically plotted the "AREA" attribute of the "nc" dataset,
- Title is given by using 'main' argument (main= "AREA" )and styled the plot with color palette using the "brewer.pal" function from the "RColorBrewer" library
- plot is then divided into 9 bins using the "quantile" option in the "breaks" argument,

```
library(sf)
library(RColorBrewer)

# Load data
nc <- st_read(system.file("shape/nc.shp", package="sf"))

# Plotting simple features (sf) with plot
plot(nc["AREA"], main = "AREA", breaks = "quantile",
     nbreaks = 9, pal = brewer.pal(9, "YlOrRd"))
```

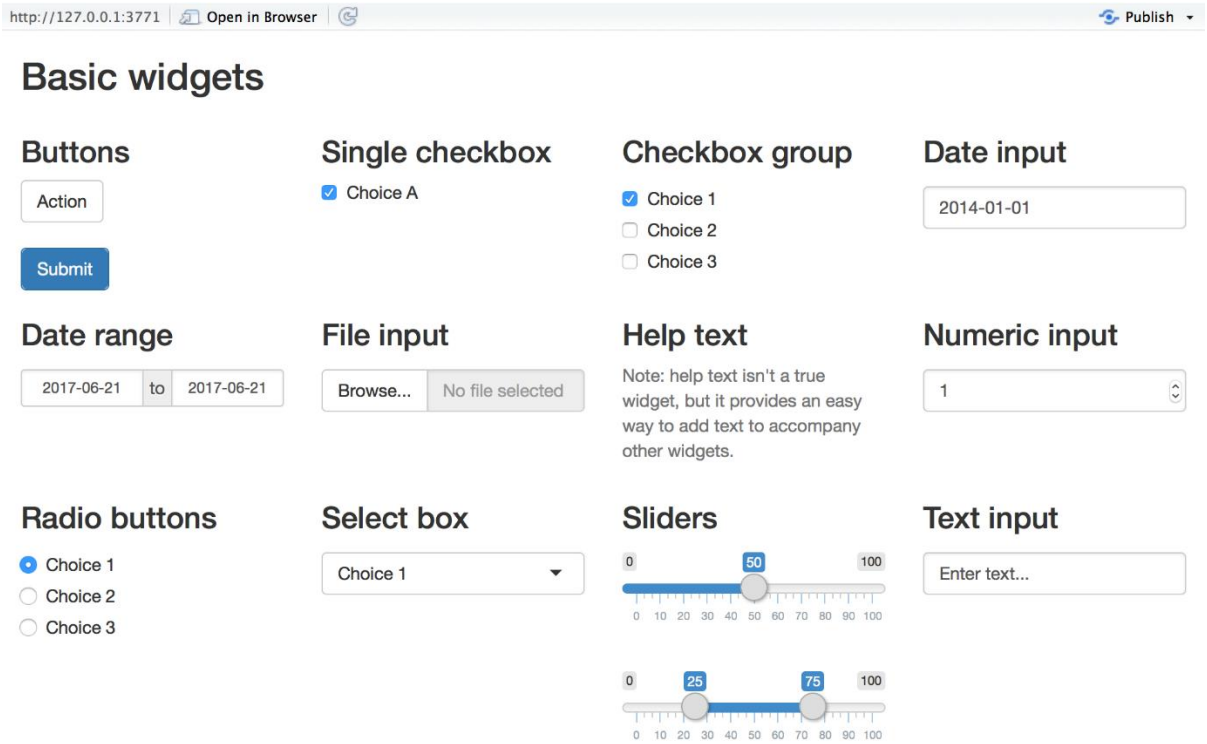



**10)a)** List the standard Shiny widgets.

b) Give the features of plotly package in R.

)Shiny widgets collect a value from your user. When a user changes the widget, the value will change as well.

2) Shiny widgets can also be invoked directly from the console (useful during authoring) and show their output within the RStudio Viewer pane or an external web browser




Shiny comes with a family of pre-built widgets, each created with a transparently named R function. For example, Shiny provides a function named `actionButton` that creates an Action Button and a function named `sliderInput` that creates a slider bar.

## The standard Shiny widgets are:

| function | widget |
|---|---|
| actionButton | Action Button |
| checkboxGroupInput | A group of check boxes |
| checkboxInput | A single check box |
| dateInput | A calendar to aid date selection |
| dateRangeInput | A pair of calendars for selecting a date range |
| fileInput | A file upload control wizard |
| helpText | Help text that can be added to an input form |
| numericInput | A field to enter numbers |
| radioButtons | A set of radio buttons |
| selectInput | A box with choices to select from |

| function | widget |
|---|---|
| sliderInput | A slider bar |
| submitButton | A submit button |
| textInput | A field to enter text |

Some of these widgets are built using the Twitter Bootstrap project, a popular open source framework for building user interfaces.

**Adding widgets**

- To add a widget to your app, place a widget function in sidebarPanel or mainPanel in your ui object.
- Each widget function requires several arguments. The first two arguments for each widget are

- a **name for the widget**: The user will not see this name, but you can use it to access the widget's value. The name should be a character string.
- a **label**: This label will appear with the widget in your app. It should be a character string, but it can be an empty string "".

In this example, the name is "action" and the label is "Action":

```
actionButton("action", label = "Action")
```

- The remaining arguments vary from widget to widget, depending on what the widget needs to do its job. They include things like initial values, ranges, and increments.

**Features of Plotly**

Lets take a look at some of the features that you can use when looking at the plot created with plotly.

- Tooltip "hover" info
- Zoom in and out of graphs
- Users can export graphs as an image
- you can hover over the different points to get more information about them

**Some of the more complex ones are:**

- Integrating multiple graphs together

- Template hover info
- Animations and moving graphics
- Integration with enterprise coding environments