

Building Web-enabled Visual Analytics Applications with Shiny: Shiny basic

Dr. Kam Tin Seong
Assoc. Professor of Information Systems
School of Computing and Information Systems,
Singapore Management University

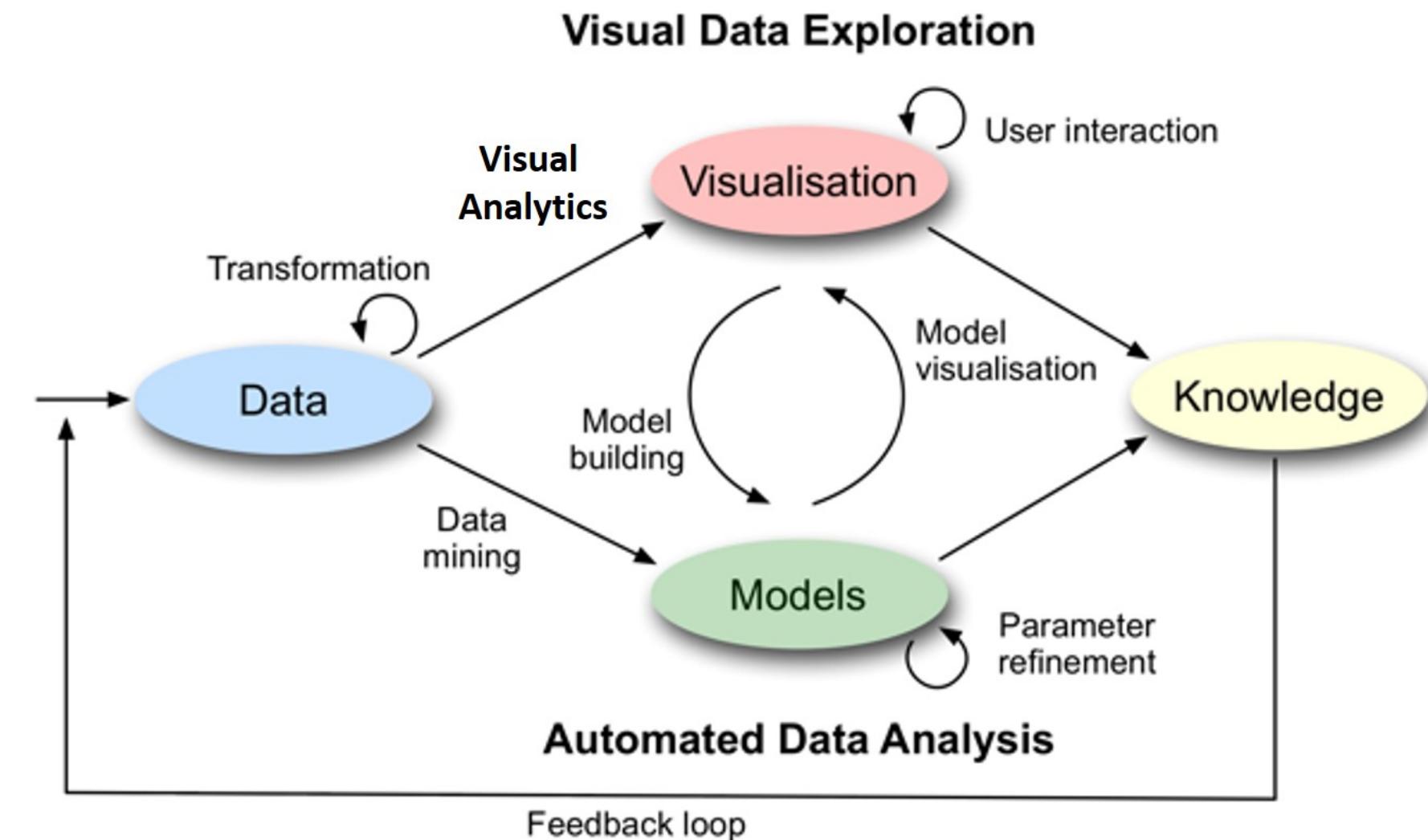
07 Jan 2023

Content

- What is a Web-enabled Visual Analytics Application?
- Why building Web-enabled Visual Analytical Application?
- Evolution of web-based Technology
- Getting to Know Shiny

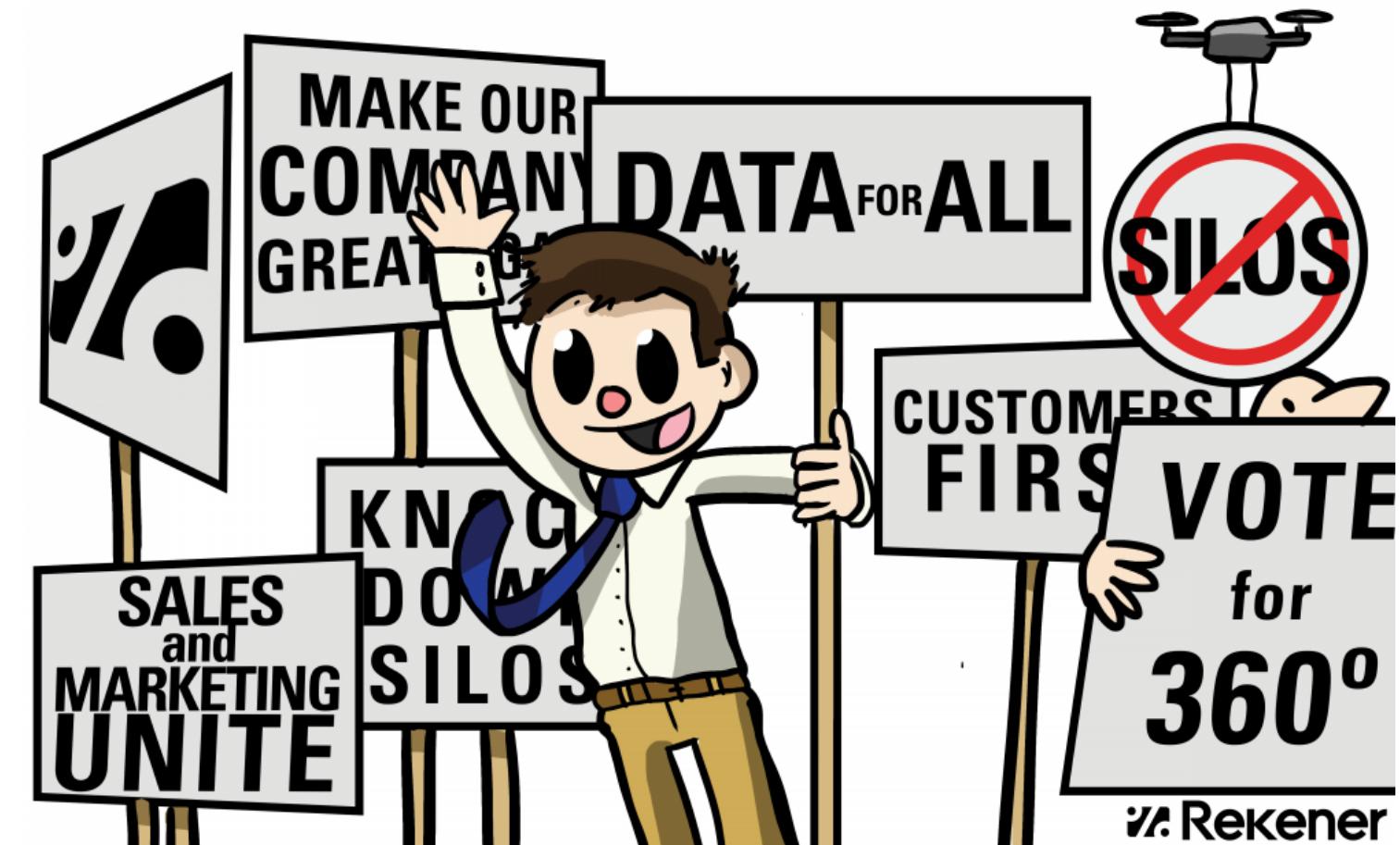
What is a Web-enabled Visual Analytics Application?

- Focuses and emphasises on **interactivity** and effective integration of techniques from **data analytics, visualization and human-computer interaction (HCI)**.



Why building a Web-enabled Visual Analytics Application?

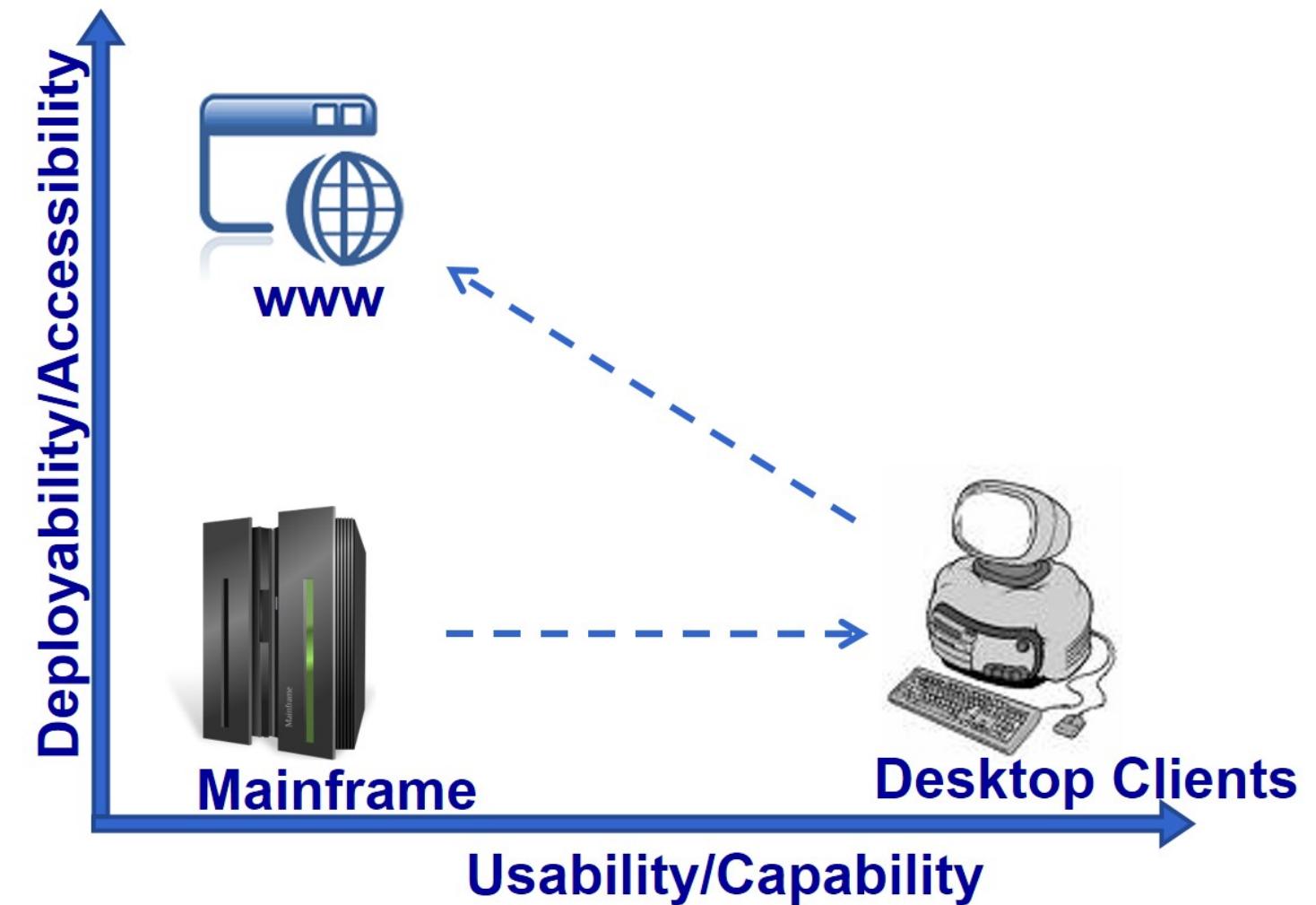
- To explore how the best of these different but related domains can be combined such that the sum is greater than the parts.
- To democratise data and analytics through web-based analytical applications for data exploration, visualisation analysis and modelling.



Source: [Democratize data analytics customer data platform](#)

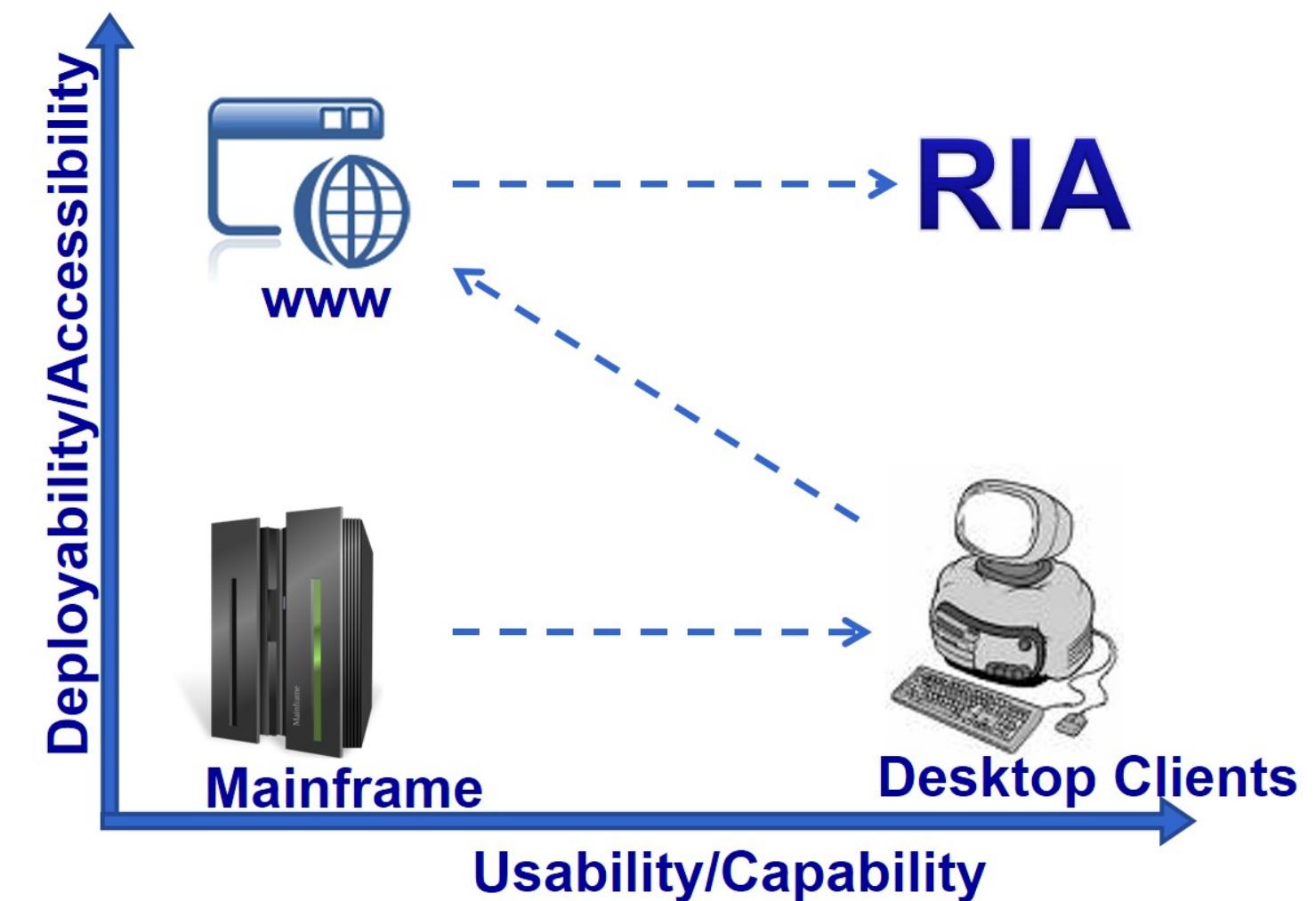
Technology Challenges

- Mainframe computing tend to have low usability and low accessibility.
- Desktop computing tend to have high usability but low accessibility.
- Web-based computing (including mobile computing) are highly accessible but with relatively low capability.



Web-based data visualisation

- The breakthrough is Rich Internet Applications (RIA)



Reference: [Rich Internet Applications](#)

Development of RIA

First generation RIA data visualisation (2000~)

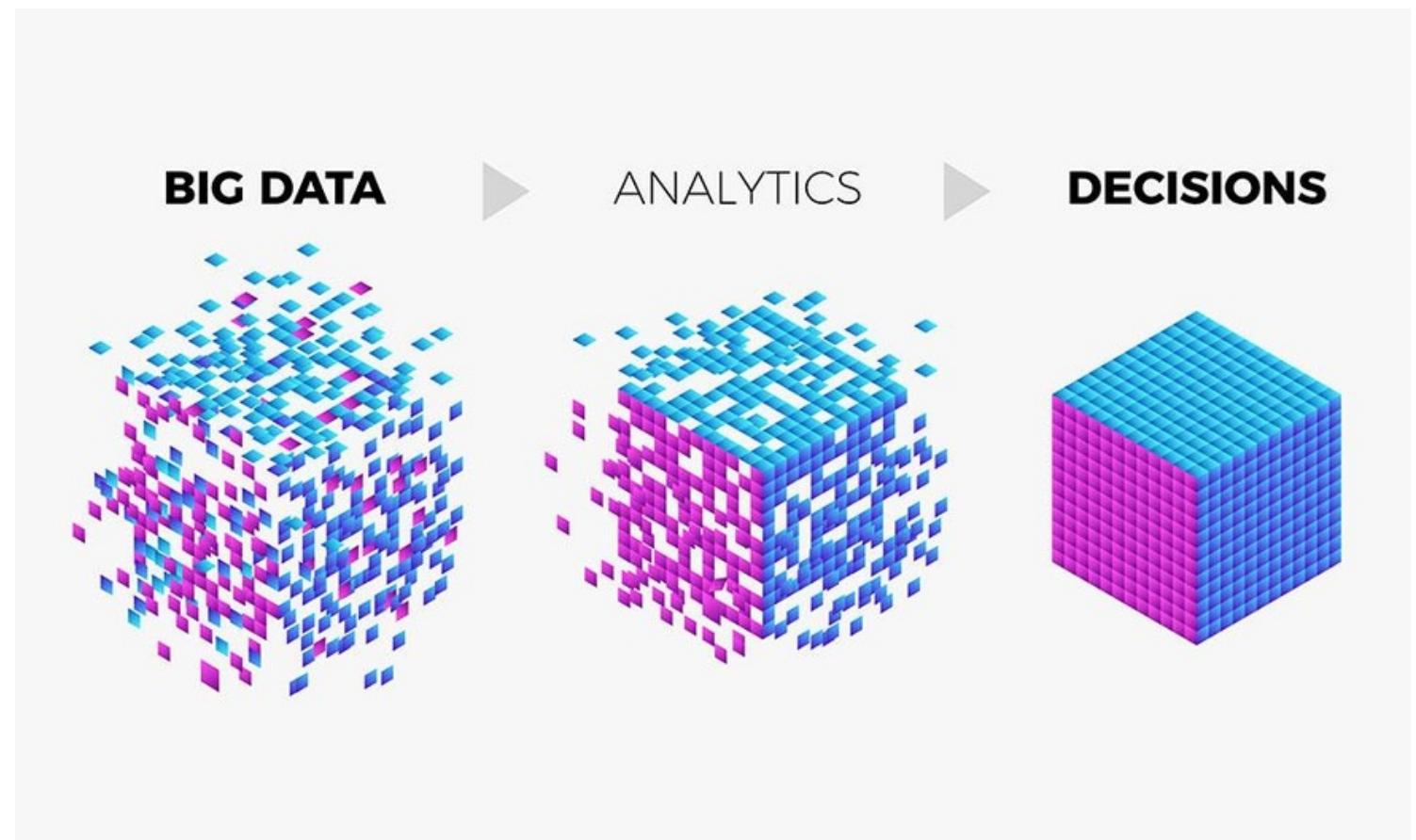
- Adobe Flex Builder
 - [Flare](#)
- Microsoft Silverlight
- JavaFX

Second generation RIA data visualisation (2010~)

- HTML 5 + JavaScript + SVG + CSS
 - Client-side rendering
 - No plug-in is required
 - Mobile computing enabled
- [D3.js - Data Driven Document](#)

Methodological Challenges

- Lack of analysis functions.
- Not reproducible.
- Not extendable.
- Require to learn multiple technologies and methods.



Getting to Know Shiny

Shiny: Overview

- Shiny is an open source package from [Posit](#) (formally called RStudio).
- It provides a **web application framework** to create interactive web applications (visualization) called “Shiny apps”.
- To learn more about Shiny, visit its [homepage](#)



Getting to Know Shiny

What is so special about Shiny?

It allows R users:

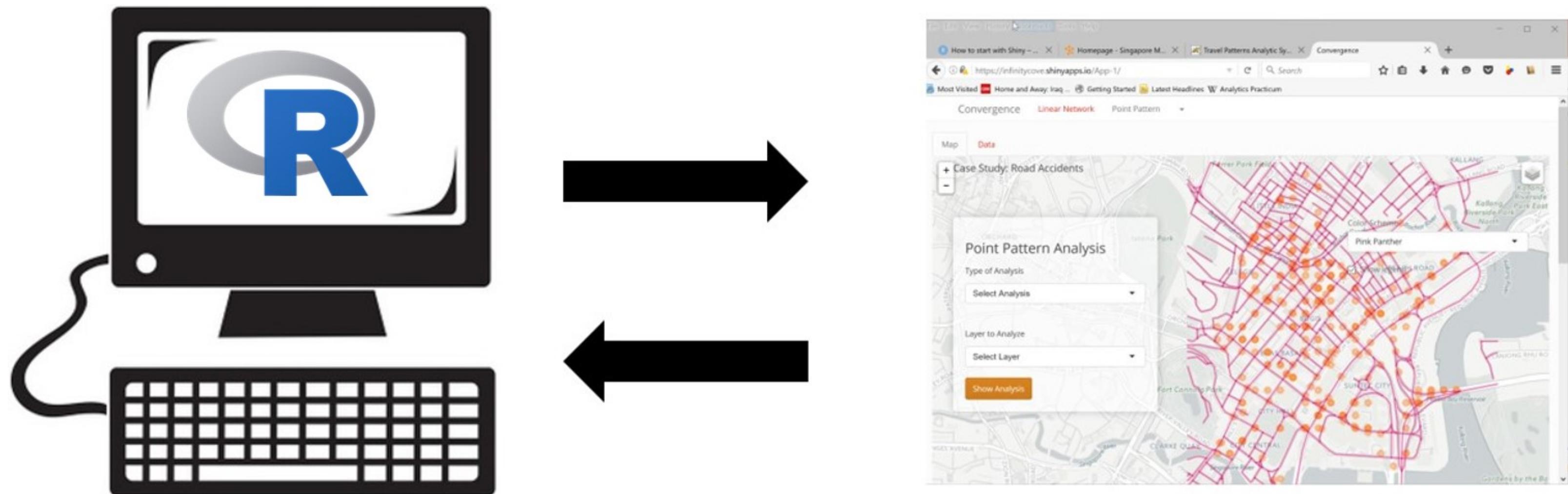
- to build and share highly interactive web-enabled applications without having to invest significant amount of time and efforts to master core web design technologies such as html5, Javascript and CSS.
- to integrate the analytical and visualisation packages of R without having to change from one programming language to another.



Getting to Know Shiny

Understanding the architecture

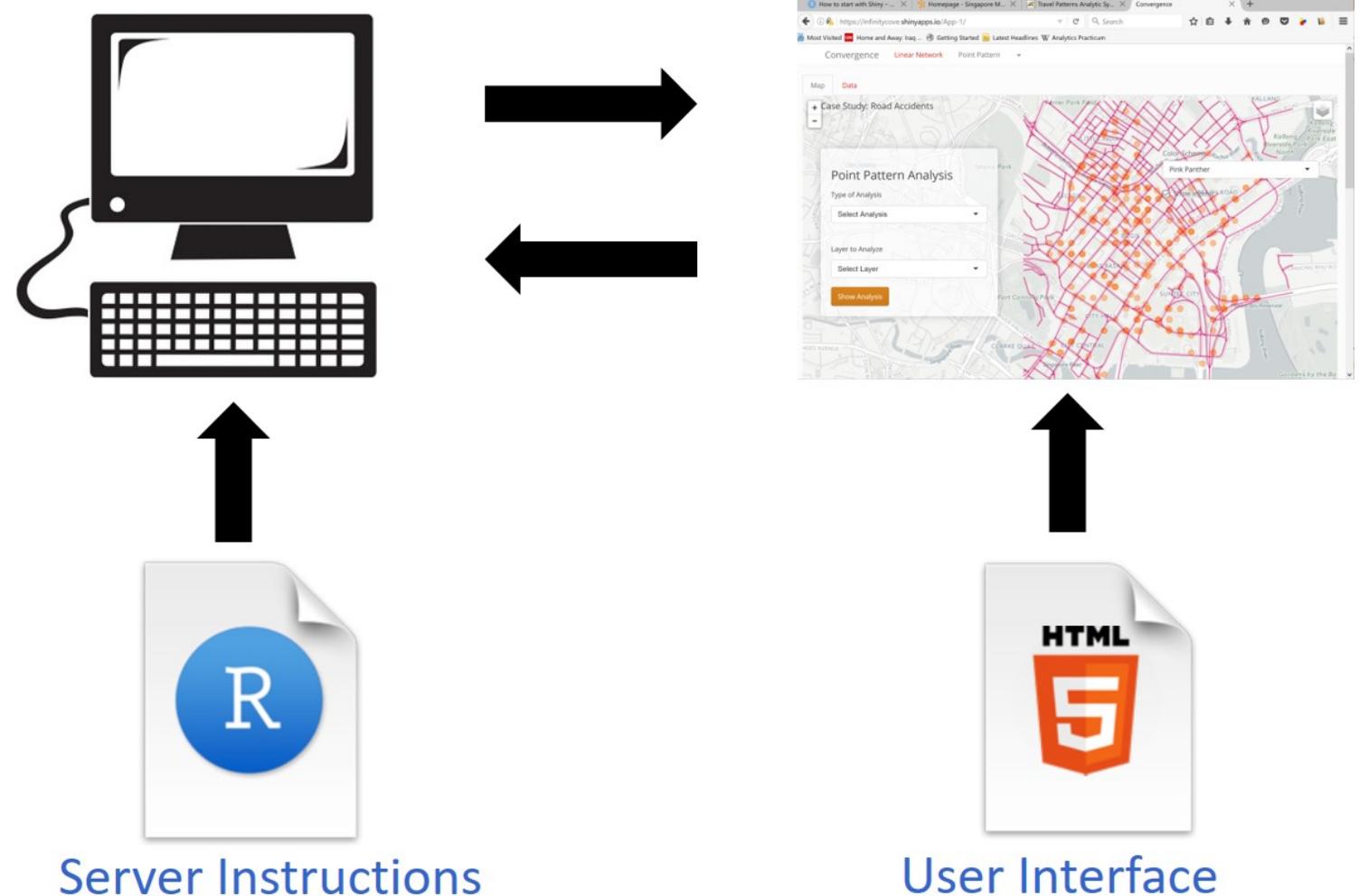
- Every Shiny app is maintained by a computer running R.



Getting to Know Shiny

The Structure of a Shiny app

- A Shiny app comprises of two components, namely:
 - a user-interface script, and
 - a server script.



Getting to Know Shiny

Shiny's user-interface, *ui.R*

- The *ui.R* script controls the layout and appearance of a shiny app.
 - It is defined in a source script name *ui.R*.
 - Actually, *ui* is a web document that the user gets to see, it is based on the famous Twitter bootstrap framework, which makes the look and layout highly customizable and fully responsive.
 - In fact, you only need to know R and how to use the shiny package to build a pretty web application. Also, a little knowledge of HTML, CSS, and JavaScript may help.

Getting to Know Shiny

Shiny's server *server.R*

- The *server.R* script contains the instructions that your computer needs to build your Shiny app.
- You are expected to:
 - know how to programme with R.
 - familiar with Tidyverse, specifically dplyr, tidyr and ggplot2

Getting to Know Shiny

Shiny Examples

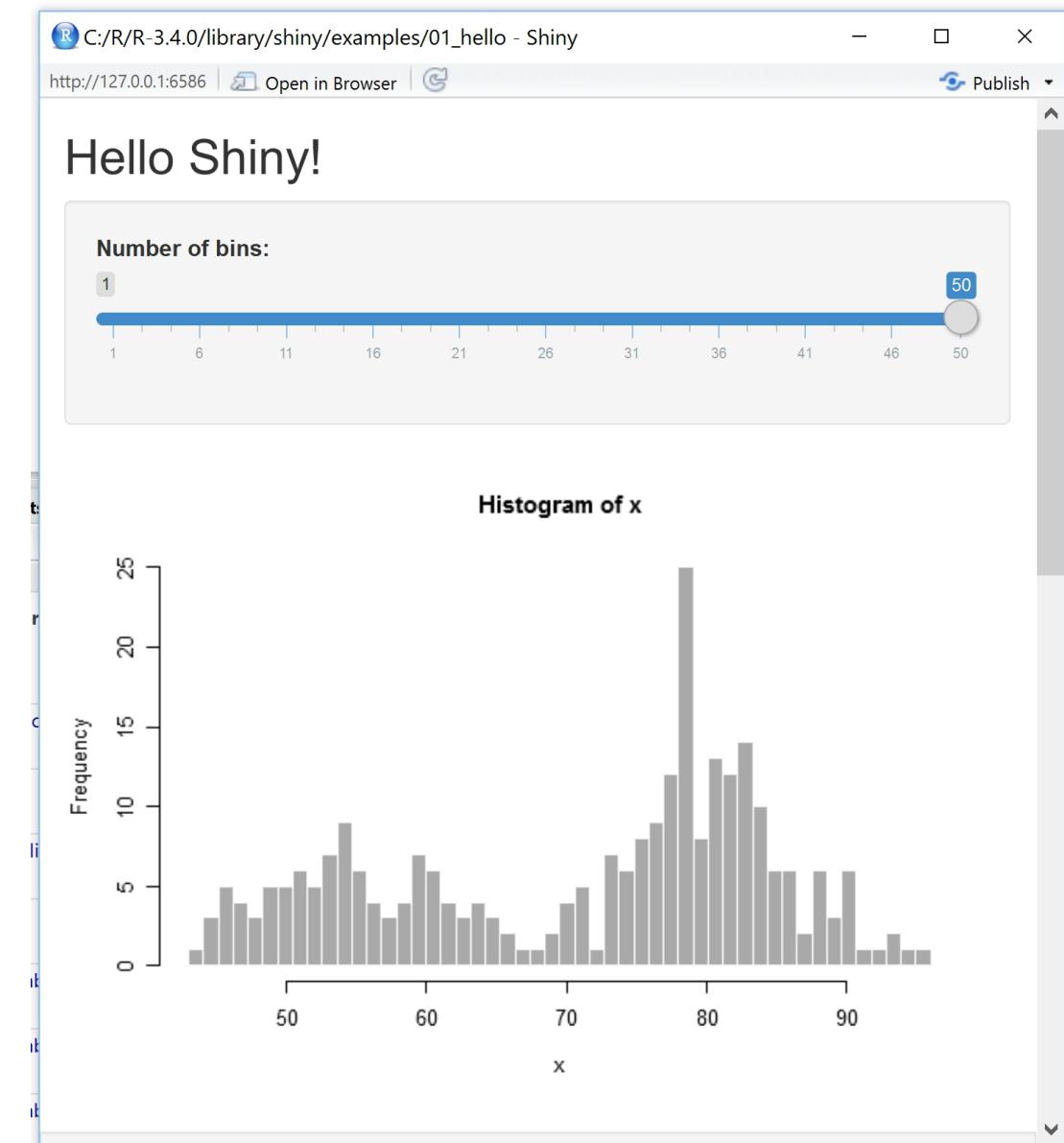
The Shiny package has eleven built-in examples that each demonstrates how Shiny works.

```
runExample("01_hello") # a histogram  
runExample("02_text") # tables and data frames  
runExample("03_reactivity") # a reactive expression  
runExample("04_mpg") # global variables  
runExample("05_sliders") # slider bars  
runExample("06_tabssets") # tabbed panels  
runExample("07_widgets") # help text and submit buttons  
runExample("08_html") # Shiny app built from HTML  
runExample("09_upload") # file upload wizard  
runExample("10_download") # file download wizard  
runExample("11_timer") # an automated timer
```

Getting to Know Shiny

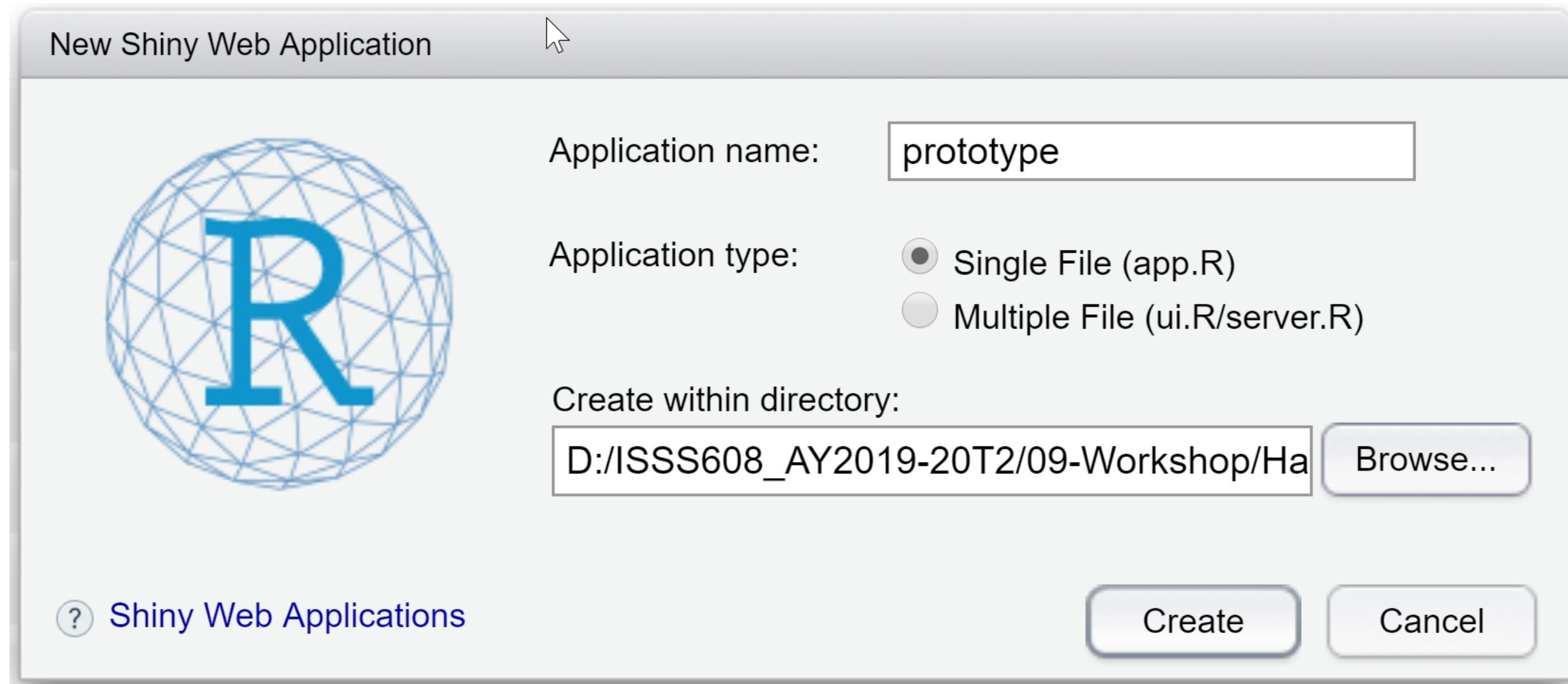
Running Shiny example

```
1 library(shiny)  
2 runExample("01_hello")
```



Building a Shiny app

- A Shiny app can be in a form of a single file called *app.R*.
- Alternatively, a Shiny app can be also created using separate *ui.R* and *server.R* files.
- The separate files way is preferred when the app is complex and involves more codes.



A basic Shiny app script

```
library(shiny)  
ui <- fluidPage()  
  
server <- function(input, output) {}  
  
shinyApp(ui = ui, server = server)
```

User interface

controls the layout and appearance of app

Server function

contains instructions needed to build app

shinyApp()

Creates the Shiny app object

Important tips of Shiny app file



- It is very important that the name of the file is *app.R*, otherwise it would not be recognized as a Shiny app.
- You should not have any R code after the `shinyApp(ui = ui, server = server)` line. That line needs to be the last line in your file.
- It is good practice to place this app in its own folder, and not in a folder that already has other R scripts or files, unless those other files are used by your app.

Loading the dataset

```
1 library(shiny)
2 library(tidyverse)
3
4 exam <- read_csv("data/Exam_data.csv")
5
6 ui <- fluidPage()
7 server <- function(input, output) {}
8 shinyApp(ui=ui, server=server)
```

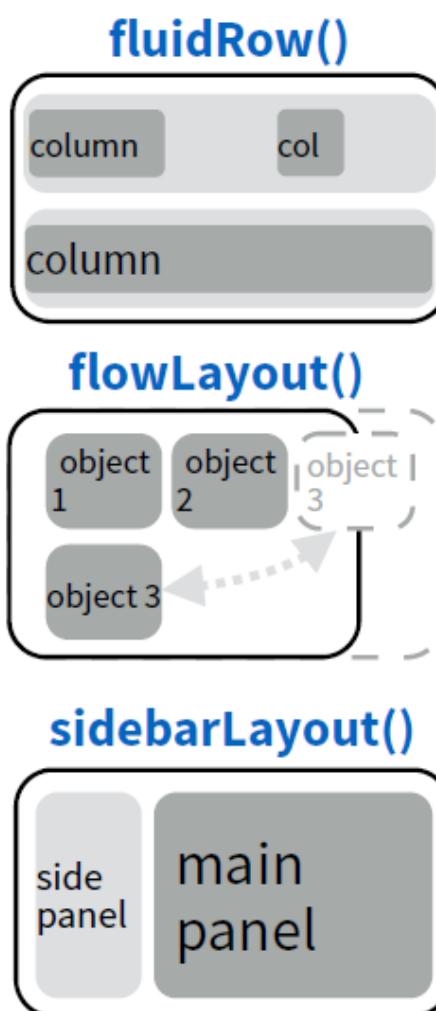
- Make sure that the data file path and file name are correct.
- To check if the dataset has been added correctly, you can add a `print()` argument after reading the data.

Shiny Layout

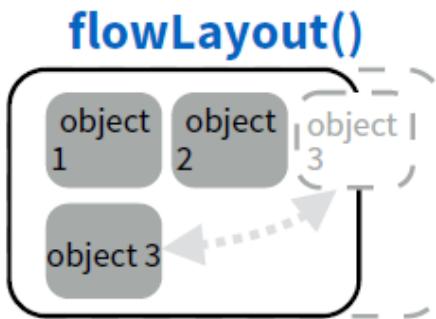
- Shiny ui.R scripts use the function *fluidPage* to create a display that automatically adjusts to the dimensions of your user's browser window.
- You lay out your app by placing elements in the *fluidPage* function.
- *titlePanel* and *sidebarLayout* are the two most popular elements to add to *fluidPage*. They create a basic Shiny app with a sidebar.

Shiny Layout Panels

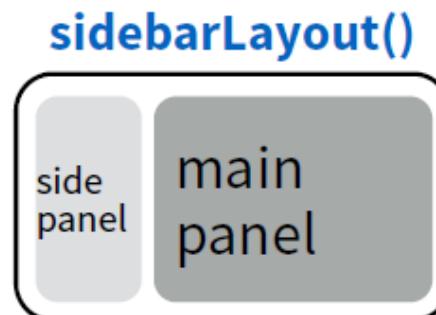
- Panels are used to group multiple elements into a single element that has its own properties.
- Especially important and useful for complex apps with a large number of inputs and outputs such that it might not be clear to the user where to get started.



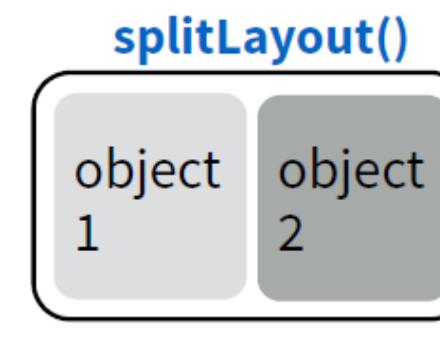
```
ui <- fluidPage(
  fluidRow(column(width = 4),
            column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```



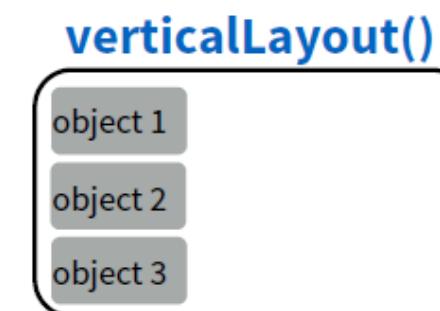
```
ui <- fluidPage(
  flowLayout(# object 1,
            # object 2,
            # object 3)
)
```



```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
)
)
```



```
ui <- fluidPage(
  splitLayout(# object 1,
             # object 2)
)
```

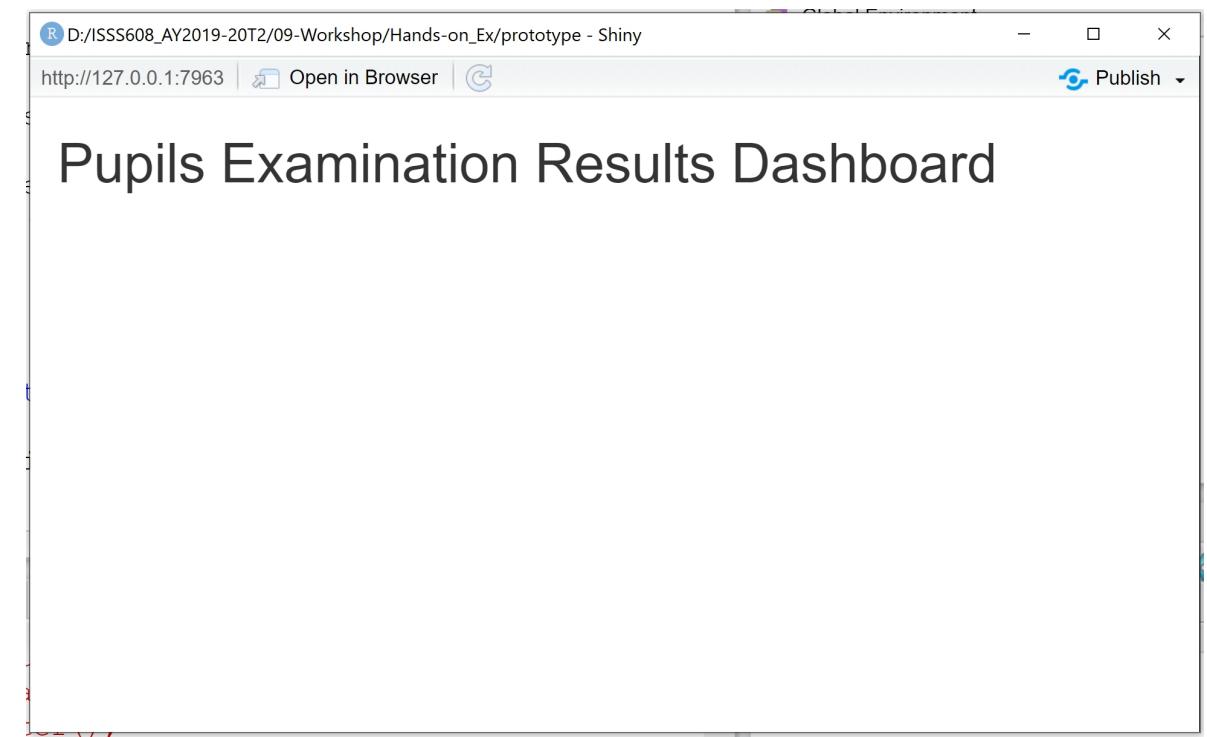


```
ui <- fluidPage(
  verticalLayout(# object 1,
                # object 2,
                # object 3)
)
```

Working with titlePanel

- `titlePanel()` is used to add the application title.

```
1 library(shiny)
2 library(tidyverse)
3
4 exam <- read_csv("data/Exam_data.csv")
5
6 ui <- fluidPage( #<<
7   titlePanel("Pupils Examination Results Dashboard") #<<
8 ) #<<
9
10 server <- function(input, output) { }
11
12 shinyApp (ui=ui, server=server)
```



Shiny Layout Panel : sidebarLayout()

- sidebarLayout() always takes two arguments:
 - sidebarPanel() function output
 - mainPanel() function output
- These functions place content in either the sidebar or the main panels.
- The sidebarPanel() will appear on the left side of your app by default. You can move it to the right side by giving sidebarLayout() the optional argument position = “right”.

```
library(shiny)

# Define UI with default width sidebar
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel("Usually inputs go here"),
    mainPanel("Usually outputs go here")
  )
)

# Define server fn that does nothing :)
server <- function(input, output) {}

# Create the app object
shinyApp(ui = ui, server = server)
```

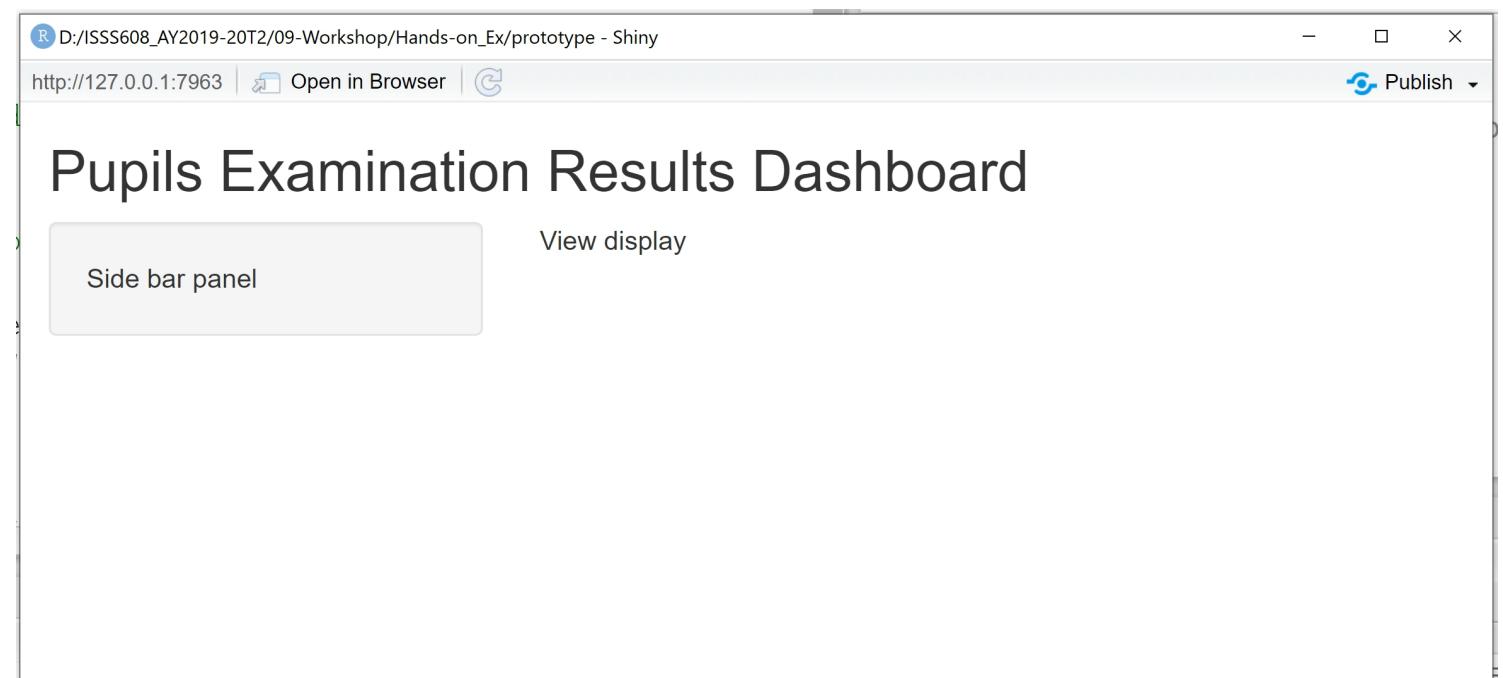


Hands-on Exercise: Working with sidebarLayout()

Lets add the highlighted codes into the original code chunk.

```
1 library(shiny)
2 library(tidyverse)
3
4 exam <- read_csv("data/Exam_data.csv")
5
6 ui <- fluidPage(
7   titlePanel("Pupils Examination Results Dashboard"),
8   sidebarLayout(
9     sidebarPanel("Side bar panel"),
10    mainPanel("View display")
11  )
12 )
13
14 server <- function(input, output) {}
15 shinyApp (ui=ui, server=server)
```

Refresh Shiny App and your screen should look similar to the figure below.



Note

Note that in a fluid design your sidebar and other elements may “collapse” if your browser view is not wide enough.

Shiny Inputs

An overview of Shiny Inputs

- Inputs are what gives users a way to interact with a Shiny app.
- Shiny provides many input functions to support many kinds of interactions that the user could have with an app.

Inputs

collect values from the user

Access the current value of an input object with `input$<inputId>`. Input values are **reactive**.

Action

`actionButton(inputId, label, icon, ...)`

Link

`actionLink(inputId, label, icon, ...)`

Choice 1
 Choice 2
 Choice 3

 Check me

`checkboxGroupInput(inputId, label, choices, selected, inline)`

`checkboxInput(inputId, label, value)`



`dateInput(inputId, label, value, min, max, format, startview, weekstart, language)`



`dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)`

`fileInput(inputId, label, multiple, accept)`

`numericInput(inputId, label, value, min, max, step)`

.....

`passwordInput(inputId, label, value)`

Choice A
 Choice B
 Choice C

`radioButtons(inputId, label, choices, selected, inline)`

Choice 1 ▾
Choice 1
Choice 2

`selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())`

0 5 10

`sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)`

`submitButton(text, icon)`
(Prevents reactions across entire app)

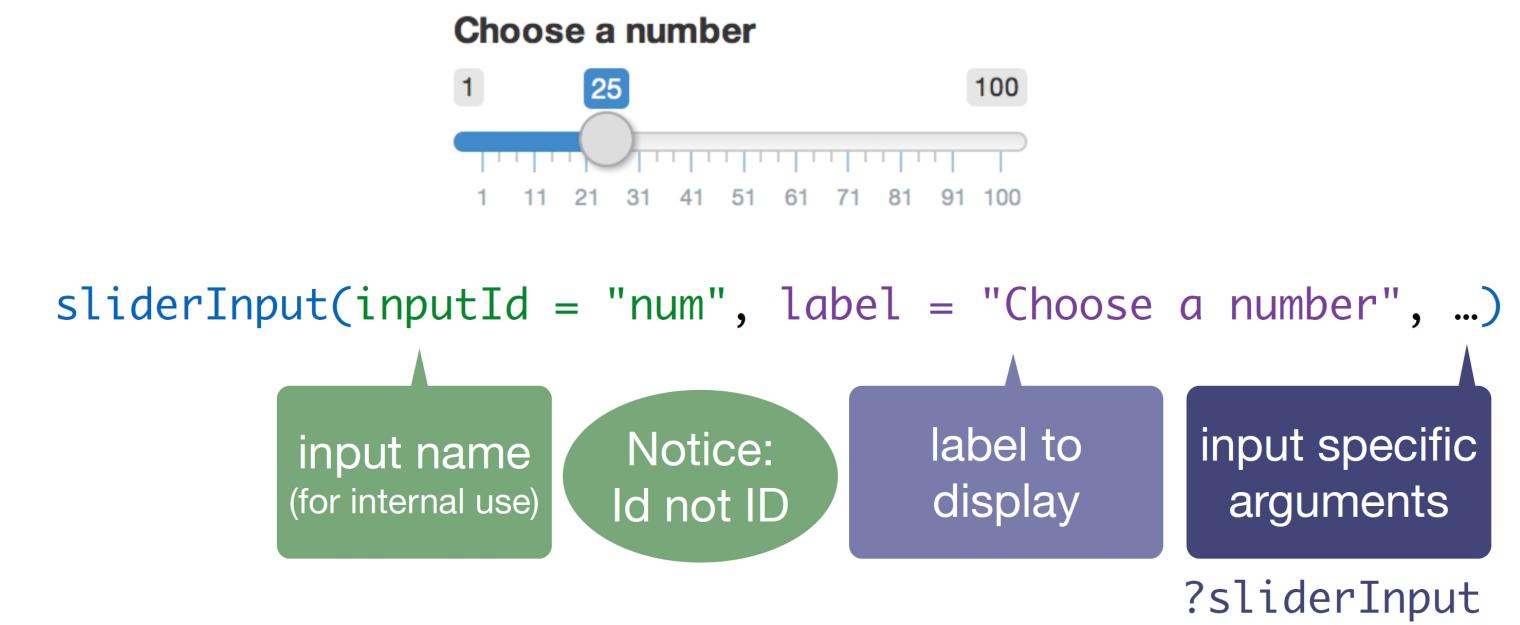
`textInput(inputId, label, value)`

Reference: Refer to [2 Basic UI](#) of Master Shiny to learn more about Shiny UI usage and arguments.

Shiny Inputs

Inputs syntax

- All input functions have the same first two arguments: `inputId` and `label`.
- The `inputId` will be the name that Shiny will use to refer to this input when you want to retrieve its current value.
- It is important to note that every input must have a unique `inputId`.
- The `label` argument specifies the text in the display label that goes along with the input widget.
- Every input can also have multiple other arguments specific to that input type.



Hands-on Exercise: Adding inputs

Adding inputs to the UI by using
`selectInput()` and `sliderInput()`.

```
1 ui <- fluidPage(  
2   titlePanel("Pupils Examination Results Dashboard"),  
3   sidebarLayout(  
4     sidebarPanel(  
5       selectInput(inputId = "variable",  
6                     label = "Subject:",  
7                     choices = c("English" = "ENGLISH",  
8                               "Maths" = "MATHS",  
9                               "Science" = "SCIENCE"),  
10                  selected = "ENGLISH"),  
11       sliderInput(inputId = "bins",  
12                     label = "Number of Bins",  
13                     min = 5,  
14                     max = 20,  
15                     value= 10)  
16     ),  
17   mainPanel()
```

Shiny Output()

An overview of Shiny Output()

- After creating all the inputs, we should add elements to the UI to display the outputs.
- To display output, add it to `fluidPage()` with an `Output()` function.

`plotOutput("hist")`

the type of output
to display

name to give to the
output object

© CC 2015 RStudio, Inc

Note

- Similarly to the input functions, all the output functions have a `outputId` argument that is used to identify each output, and this argument must be unique for each output.
- Each output needs to be constructed in the server code later.

Shiny Output()

Shiny `Output()` options

- Outputs can be any object that R creates and that we want to display in our app - such as a plot, a table, or text.

Function	Inserts
dataTableOutput()	an interactive table
htmlOutput()	raw HTML
imageOutput()	image
plotOutput()	plot
tableOutput()	table
textOutput()	text
uiOutput()	a Shiny UI element
verbatimTextOutput()	text

Hands-on Exercise:

Adding `plotOutput()`

```
1 ui <- fluidPage(  
2   titlePanel("Pupils Examination Results Dashboard"),  
3   sidebarLayout(  
4     sidebarPanel("Side bar panel"),  
5     mainPanel(  
6       plotOutput("distPlot")  
7     )  
8   )  
9 )
```

Shiny server.R

Building an output

There are three rules to build an output in Shiny, they are:

- Save the output object into the output list (remember the app template - every server function has an output argument).
- Build the object with a `render()` function, where `is` the type of output.
- Access input values using the input list (every server function has an input argument)

 Note

The third rule is only required if you want your output to depend on some input.

Shiny server.R

A generic Shiny `render()` syntax

```
renderPlot({ hist(rnorm(100)) })
```

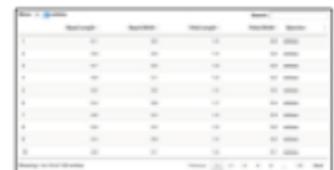
type of object to
build

code block that builds
the object

Shiny server.R

Shiny `render()`

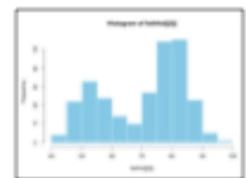
Outputs - `render*()` and `*Output()` functions work together to add R output to the UI



`DT::renderDataTable(expr, options, callback, escape, env, quoted)`



`renderImage(expr, env, quoted, deleteFile)`



`renderPlot(expr, width, height, res, ..., env, quoted, func)`

`'data.frame': 3 obs. of 2 variables:`
`$ Sepal.Length: num 5.1 4.9 4.7`
`$ Sepal.Width : num 3.5 3 3.2`

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.10	3.50	1.60	0.20	Iris-setosa
2	4.90	3.00	1.40	0.20	Iris-setosa
3	4.70	3.20	1.30	0.20	Iris-setosa
4	5.00	3.00	1.50	0.20	Iris-versicolor
5	5.00	3.40	1.40	0.20	Iris-versicolor
6	4.90	3.10	1.30	0.20	Iris-versicolor

`foo`

A screenshot of a Shiny application interface showing a slider input labeled 'Choose a number' and a histogram titled 'Histogram of Random Normal Values'.

`renderPrint(expr, env, quoted, func, width)`

`renderTable(expr, ..., env, quoted, func)`

`renderText(expr, env, quoted, func)`

`renderUI(expr, env, quoted, func)`

works
with

`dataTableOutput(outputId, icon, ...)`

`imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)`

`plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)`

`verbatimTextOutput(outputId)`

`tableOutput(outputId)`

`textOutput(outputId, container, inline)`

`& uiOutput(outputId, inline, container, ...)`

`htmlOutput(outputId, inline, container, ...)`

Hands-on Exercise: Building a basic output

Let's first see how to build a very basic output using `renderPlot()`. We'll create a plot and send it to the `distPlot` output.

```
1 server <- function(input, output) {  
2   output$distPlot <- renderPlot({  
3     ggplot(exam, aes(ENGLISH)) +  
4       geom_histogram(bins = 20,  
5                         color="black",  
6                         fill="light blue")  
7   })  
8 }
```

Note

- This simple code shows the first two rules: we're creating a plot inside the `renderPlot()` function, and assigning it to `distPlot` in the output list.
- Remember that every output created in the UI must have a unique ID, now we see why. In order to attach an R object to an output with ID x, we assign the R object to `output$distPlot`.
- Since `distPlot` was defined as a `plotOutput`, we must use the `renderPlot()` function, and we must create a plot inside the `renderPlot()` function.

Hands-on Exercise: Building a basic output

First revision.

```
1 server <- function(input, output){  
2   output$distPlot <- renderPlot({  
3     x <- unlist(exam[,input$variable])  
4     ggplot(exam, aes(x)) +  
5       geom_histogram(bins = input$bin,  
6                         color="black",  
7                         fill="light blue")  
8   })  
9 }
```

Hands-on Exercise: Building a basic output

Using `aes_string()`.

```
1 server <- function(input, output){  
2   output$distPlot <- renderPlot({  
3     ggplot(exam,  
4       aes_string(x = input$variable)) +  
5     geom_histogram(bins = input$bin,  
6                     color="black",  
7                     fill="light blue")  
8   })  
9 }
```

Note

- Notice that `aes_string()` instead of `aes()` of `ggplot2` is used. To understand the differences between `aes_string()` and `aes()`, please refer to this [link](#).

The shinyApp()

! Important

- It is important to add *shinyApp()* at the end of your Shiny application.

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

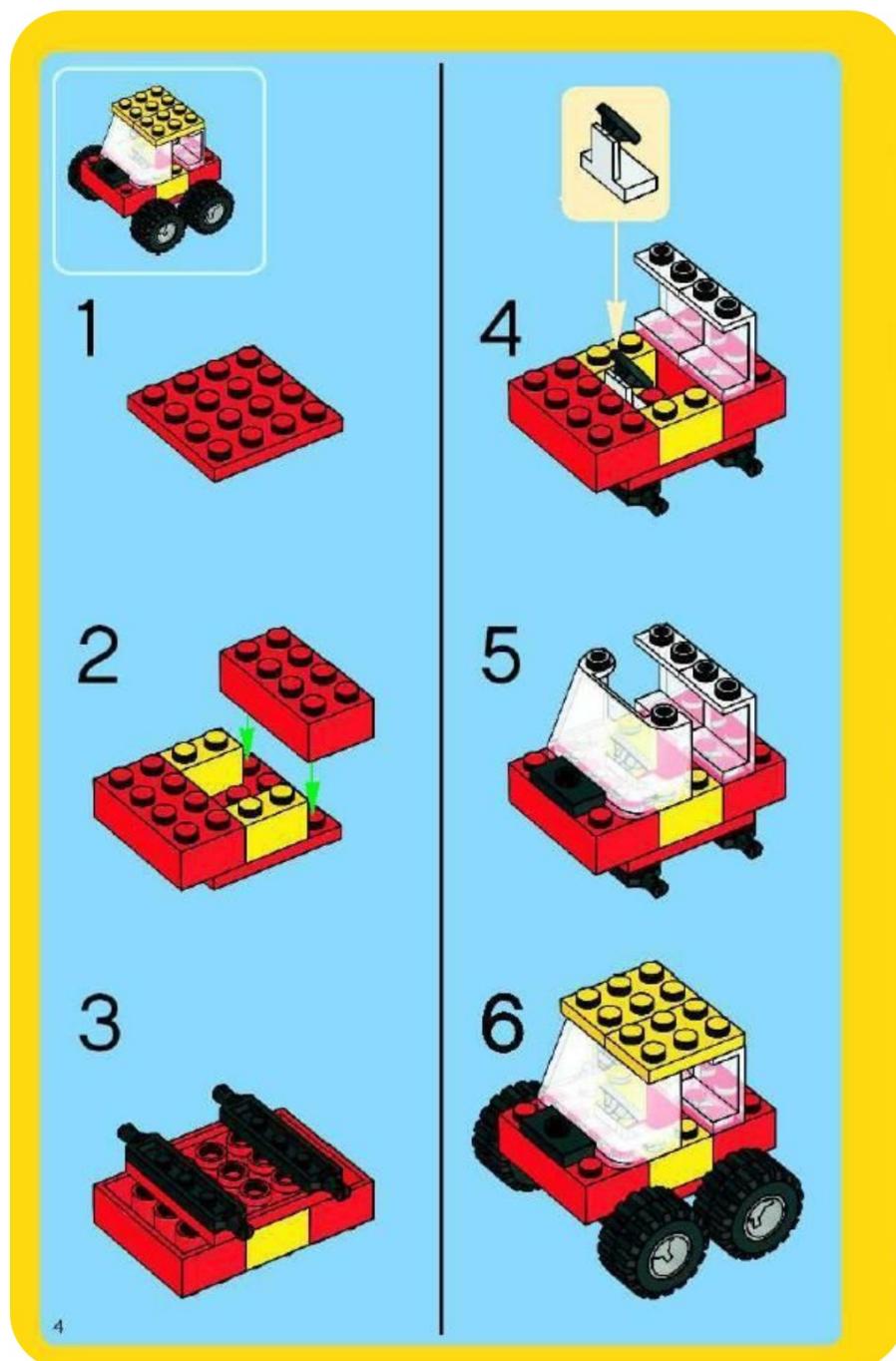
Programming Shiny: Survival Tip!



- Always run the entire script, not just up to the point where you're developing code.
- Sometimes the best way to see what's wrong is to run the app and review the error.
- Watch out for commas!

Building Shiny Application: Survival Tip!

What can we learn from Lego?



- Sketch the storyboard
- Building the app incrementally
 - Using prototyping approach
 - Start as simple as possible
 - Adding features one at a time
- Save -> Run App

References

- Hadley Wickham (2021) [Mastering Shiny](#), O'Reilly Media. This is a highly recommended book.
- [Building Web Applications with Shiny](#), especially Module 1 and 2.
- [Shiny Three Parts Tutorial](#).
- [Online Function reference](#)
- [The basic parts of a Shiny app](#)
- [How to build a Shiny app](#)
- [The Shiny Cheat sheet](#)

Beyond Uncle Google! Last but not least, when you need help

- [How to get help](#)

