Lesson 12: Advanced Shiny

Dr. Kam Tin Seong
Assoc. Professor of Information Systems

School of Computing and Information Systems,
Singapore Management University

13 July 2021

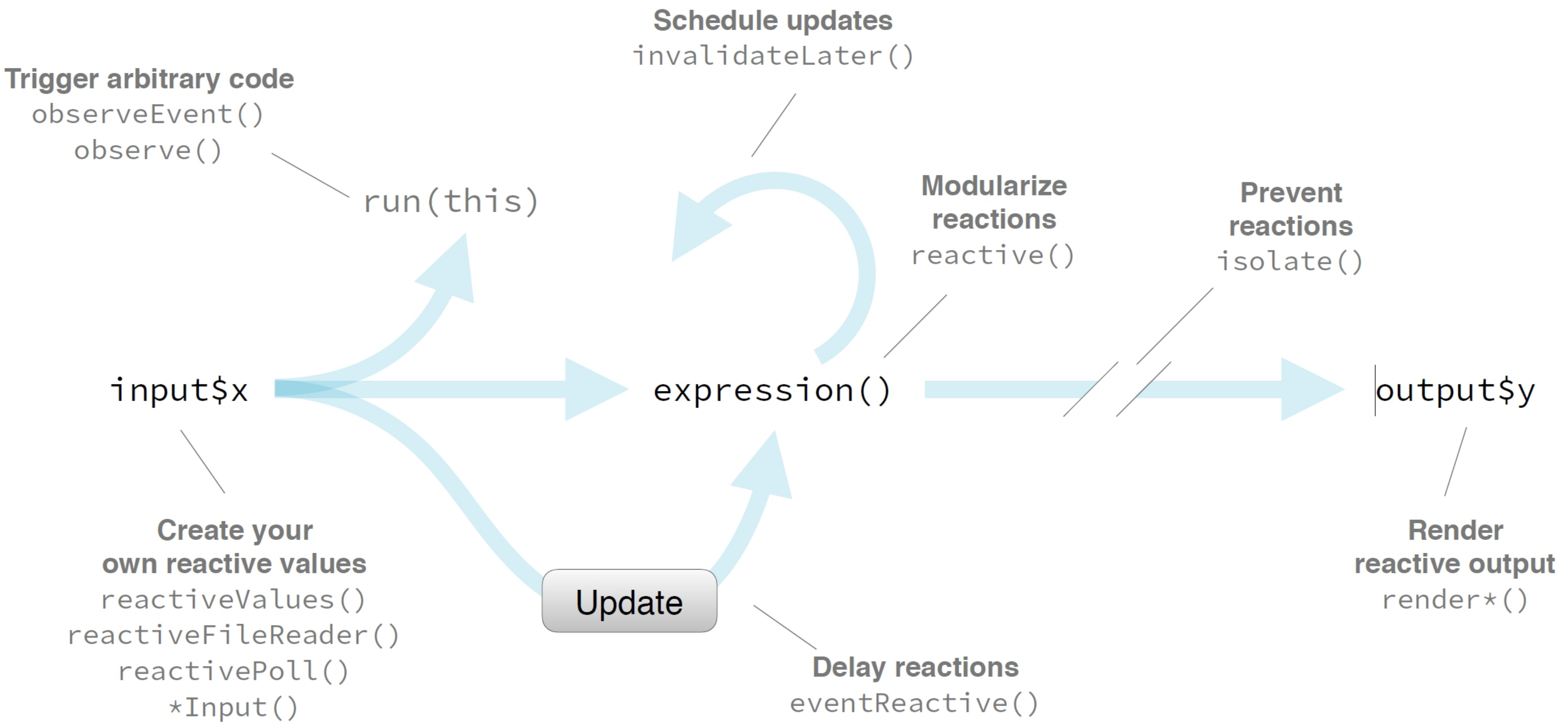
Overview

In this lesson, selected advanced methods of Shiny will be discussed. You will also gain hands-on experiences on using these advanced methods to build Shiny applications.

By the end of this lesson, you will be able to:

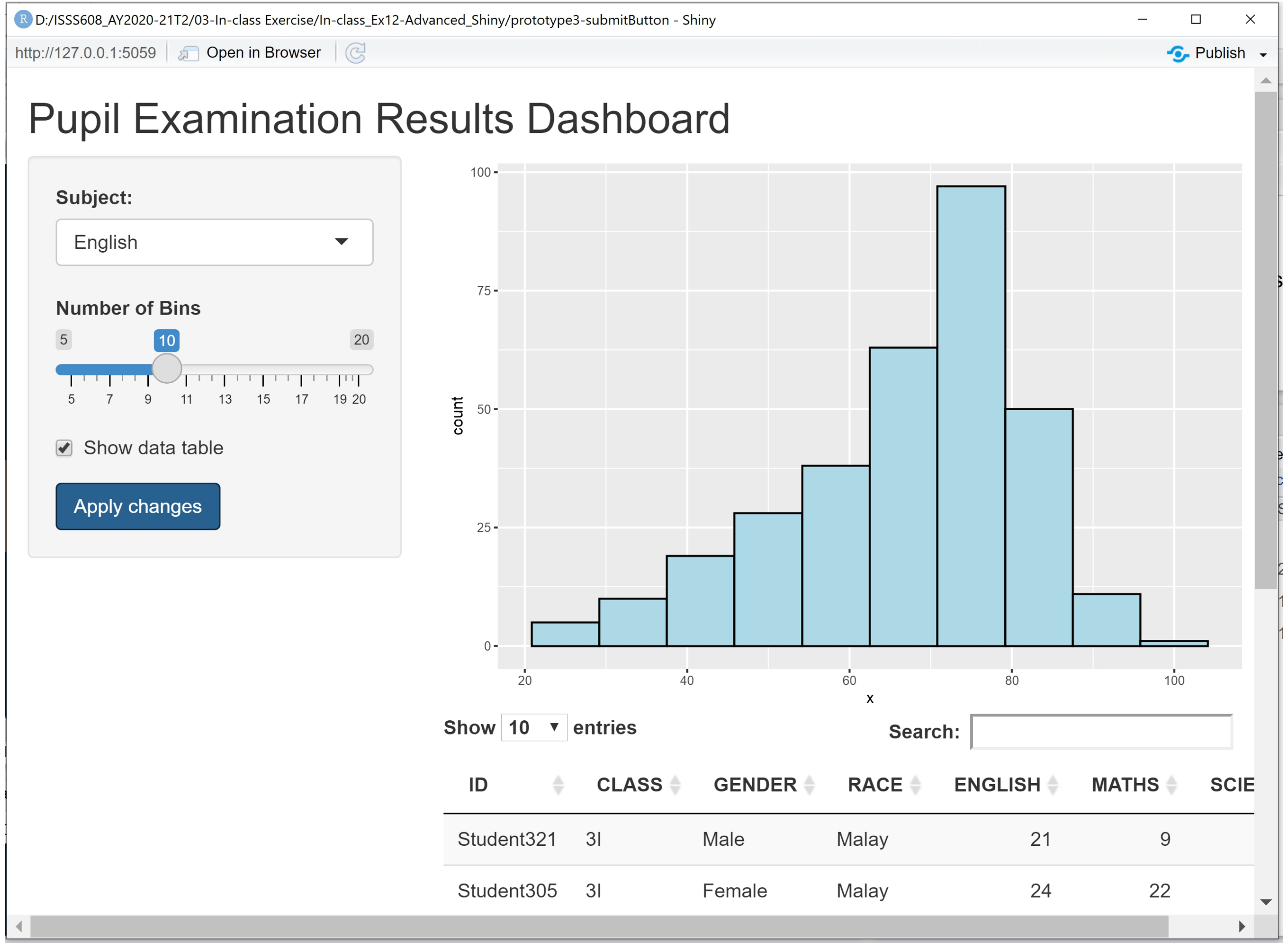
- gain further understanding of the reactive feature of Shiny and Shiny's functions that support reactive flow,
- build interactive Shiny application by using plotly R,
- plot both static and interactive thematic maps by using tmap,
- improve the productivity of Shiny applications development by using related built-in functions of Shiny for debugging and extension package.

Reactive Flow



Build Reactivity

Working with *submitButton()*



Build Reactivity

Working with *submitButton()*

- *submitButton()* is used when you want to delay a reaction.

- Edit the code as shown below:

```
checkboxInput(inputId = "show_data",
                          label = "Show data table",
                          value = TRUE),
            submitButton("Apply changes")
```

Note: The use of *submitButton* is generally discouraged in favor of the more versatile *actionButton()*

Reference: https://shiny.rstudio.com/reference/shiny/latest/submitButton.html

Build Reactivity

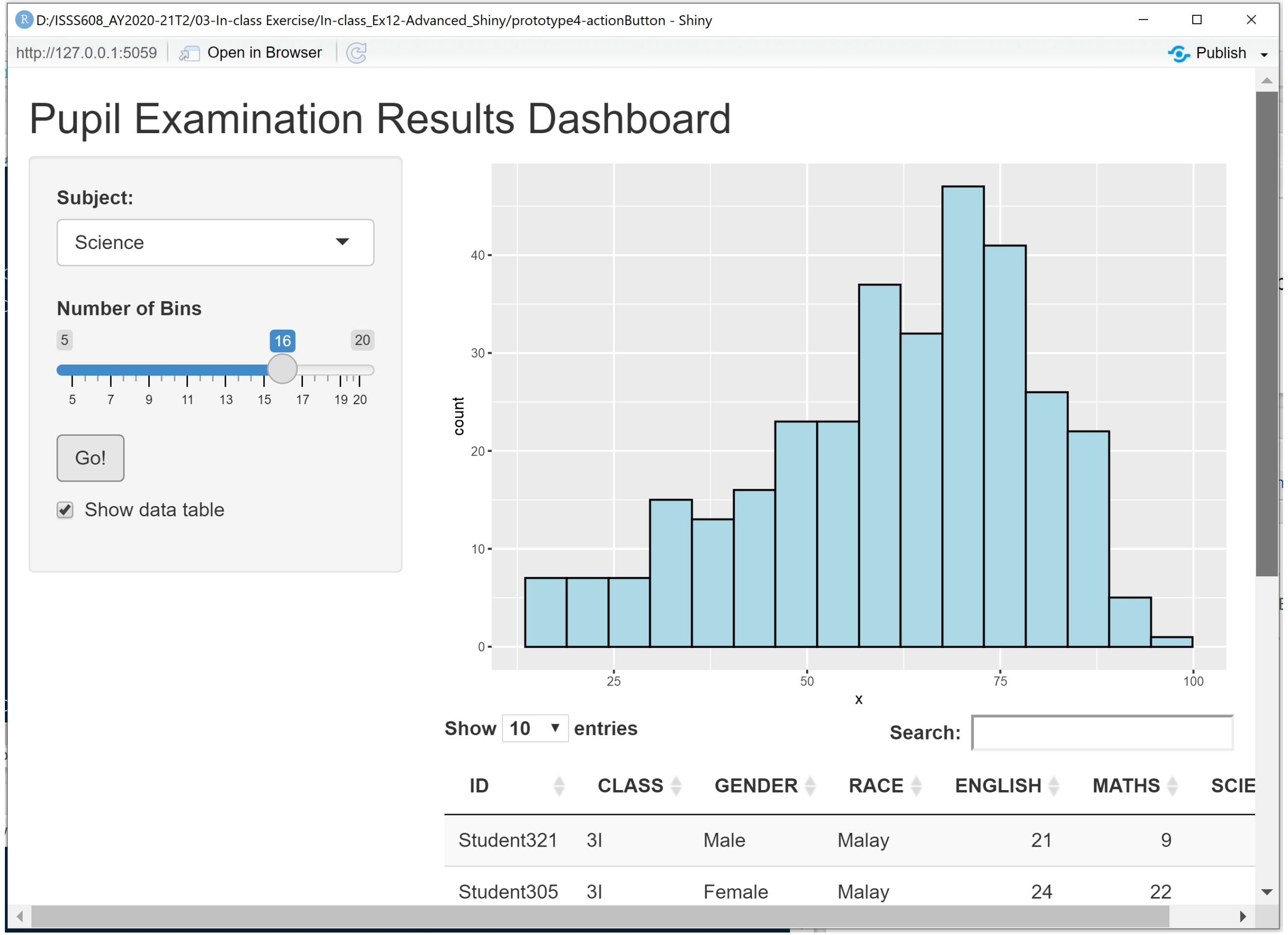
Working with *isolate()* and *actionButton()*

- Sometimes it's useful for an observer/endpoint to access a reactive value or expression, but not to take a dependency on it.
- The *actionButton()* includes some JavaScript code that sends numbers to the server. When the web browser first connects, it sends a value of 0, and on each click, it sends an incremented value: 1, 2, 3, and so on.
- In the server function, there are two items to note. First, *output$distPlot* will take a dependency on *input$goButton*, simply by accessing it. When the button is clicked, the value of *input$goButton* increases, and so *output$distPlot* re-executes.
- The second is that the access to *input$obs* is wrapped with *isolate()*. This function takes an R expression, and it tells Shiny that the calling observer or reactive expression should not take a dependency on any reactive objects inside the expression.

Reference: https://shiny.rstudio.com/articles/isolation.html

Build Reactivity

Working with *isolate()* and *actionButton()*



Build Reactivity

Working with *isolate()* and *actionButton()*

- At the ui, edit the code as shown below:

```
sliderInput(inputId = "bin",
                        label = "Number of Bins",
                        min = 5,
                        max = 20,
                        value = c(10)),
            actionButton("goButton", "Go!"),
            checkboxInput(inputId = "show_data",
                          label = "Show data table",
                          value = TRUE)
```

Build Reactivity

Working with *isolate()* and *actionButton()*

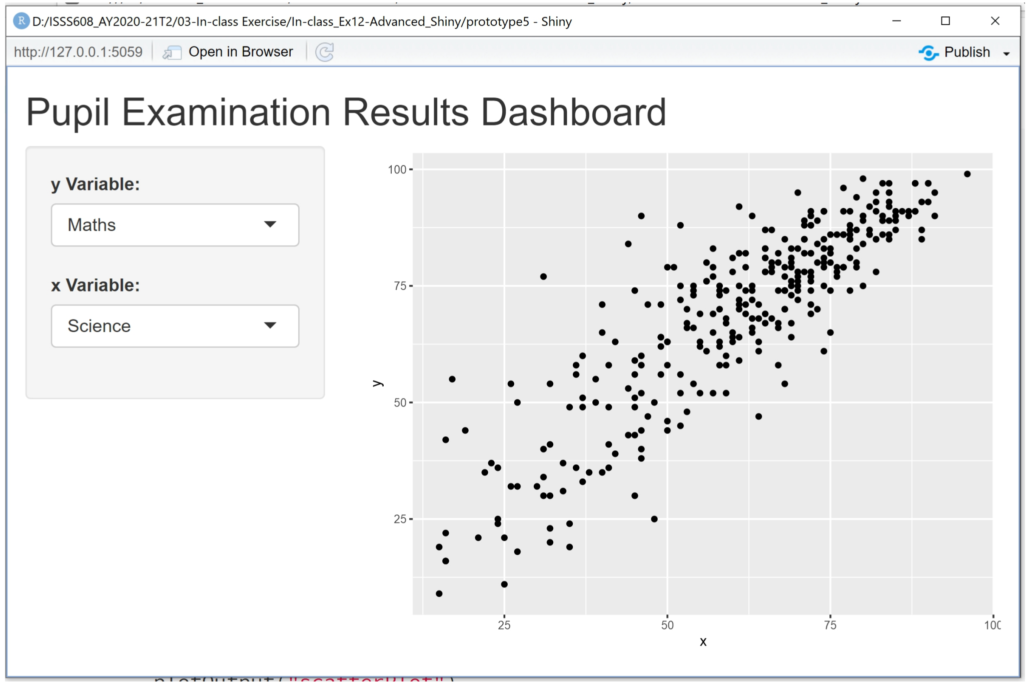
- At the server side, edit the codes as shown below:

```
server <- function(input, output){
    output$distPlot <- renderPlot({
        input$goButton

        x <- unlist(exam[,input$variable])

        isolate({
        ggplot(exam, aes(x)) +
            geom_histogram(bins = input$bin,
                           color="black",
                           fill="light blue")
        })
    })
```

Building Interactive Graphics in Shiny

The plotly way

- Plot the basic visualisation using **ggplots**
- Wrap the visualisation object into plotly object using *ggplotly()*.



Reference:

- Plotly R Open Source Graphing Library (https://plotly.com/r/)
- Interactive web-based data visualization with R, plotly, and shiny (https://plotly-r.com/index.html)

In-class Exercise: Building a static scatter plot in Shiny

```
library(shiny)
library(tidyverse)
exam <- read_csv("data/Exam_data.csv")
ui <- fluidPage(
    titlePanel("Pupil Examination Results Dashboard"),
    sidebarLayout(
        sidebarPanel(
            selectInput(inputId = "yvariable",
                        label = "y Variable:",
                        choices = c("English" = "ENGLISH",
                                    "Maths" = "MATHS",
                                    "Science" = "SCIENCE"),
                        selected = "MATHS"),
            selectInput(inputId = "xvariable",
                        label = "x Variable:",
                        choices = c("English" = "ENGLISH",
                                    "Maths" = "MATHS",
                                    "Science" = "SCIENCE"),
                        selected = "SCIENCE")
        ),
        mainPanel(
            plotOutput("scatterPlot")
        )
    )
```

In-class Exercise: Building a static scatter plot in Shiny

```
server <- function(input, output){
    output$scatterPlot <- renderPlot({
        x <- unlist(exam[,input$xvariable])
        y <- unlist(exam[,input$yvariable])

        ggplot(exam, aes(x, y)) +
            geom_point()
    })
}

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

In-class Exercise:

Building an interactive scatter plot in Shiny using Plotly

- Install plotly R package if it has yet to be install in RStudio.
- include a new line as shown below to launch plotly library.

```
library(shiny)
library(plotly)
library(tidyverse)
```

- At UI, edit the code as shown below

```
        mainPanel(
            plotlyOutput("scatterPlot")
```

- edit the server section of the Shiny app as shown below.

```
server <- function(input, output){
    output$scatterPlot <- renderPlotly({
        x <- unlist(exam[,input$xvariable])
        y <- unlist(exam[,input$yvariable])

        p <- ggplot(exam, aes(x, y)) +
            geom_point(color="grey 10",
                       size=1)

        ggplotly(p)
    })
}
```

Plotting static map in Shiny - *renderPlot()* method

- Setting up

```r
library(shiny)
library(sf)
library(tmap)
library(tidyverse)
```

- Importing the geospatial data

```r
mpsz <- st_read(dsn = "data/geospatial",
                layer = "MP14_SUBZONE_WEB_PL")
```

Plotting static map in Shiny - *renderPlot()* method

The UI codes

```
ui <- fluidPage(
    titlePanel("A simple map display"),
    sidebarLayout(
        sidebarPanel(
            checkboxInput(inputId = "show_data",
                          label = "Show data table",
                          value = TRUE)
        ),
        mainPanel(
            plotOutput("mapPlot"),
            DT::dataTableOutput(outputId = "szTable")
        )
    )
)
```

Plotting static map in Shiny - *renderPlot()* method

The Server codes

```
server <- function(input, output){
    output$mapPlot <- renderPlot({
        tm_shape(mpsz)+
            tm_fill() +
            tm_borders(lwd = 0.1,  alpha = 1)
    })

    output$szTable <- DT::renderDataTable({
        if(input$show_data){
            DT::datatable(data = mpsz %>% select(1:7),
                          options= list(pageLength = 10),
                          rownames = FALSE)
        }
    })
}
```

Important, don't miss out this line

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

Plotting interactive map in Shiny - *renderPlot()* method

At the UI, edit the code as shown below:

```
mainPanel(
            tmapOutput("mapPlot"),
            DT::dataTableOutput(outputId = "szTable")
        )
```

At the server, edit the code as shown below:

```
server <- function(input, output){
    output$mapPlot <- renderTmap({
        tm_shape(mpsz)+
            tm_fill() +
            tm_borders(lwd = 0.1,
                       alpha = 1)
    })
```

Building a choropleth mapping application

Edit the code as shown below:

```
mpsz <- st_read(dsn = "data/geospatial",
                layer = "MP14_SUBZONE_WEB_PL")

popagsex <- read_csv("data/aspatial/respopagsex2000to2018.csv")
```

Building a choropleth mapping application

Edit the codes as shown below:

```
popagsex2018_male <- popagsex %>%
    filter(Sex == "Males") %>%
    filter(Time == 2018) %>%
    spread(AG, Pop) %>%
    mutate(YOUNG = `0_to_4`+`5_to_9`+`10_to_14`+
             `15_to_19`+`20_to_24`) %>%
    mutate(`ECONOMY ACTIVE` = rowSums(.[9:13])+
             rowSums(.[15:17]))%>%
    mutate(`AGED`=rowSums(.[18:22])) %>%
    mutate(`TOTAL`=rowSums(.[5:22])) %>%
    mutate(`DEPENDENCY` = (`YOUNG` + `AGED`)
           /`ECONOMY ACTIVE`) %>%
    mutate_at(.vars = vars(PA, SZ),
              .funs = funs(toupper)) %>%
    select(`PA`, `SZ`, `YOUNG`,
           `ECONOMY ACTIVE`, `AGED`,
           `TOTAL`, `DEPENDENCY`) %>%
    filter(`ECONOMY ACTIVE` > 0)
mpsz_agemale2018 <- left_join(mpsz,
                              popagsex2018_male,
                              by = c("SUBZONE_N" = "SZ"))
```

Building a choropleth mapping application

At the UI, edit the codes as shown below:

```
ui <- fluidPage(
    titlePanel("Choropleth Mapping"),
    sidebarLayout(
        sidebarPanel(
            selectInput(inputId = "classification",
                        label = "Classification method:",
                        choices = list("fixed" = "fixed",
                                       "sd" = "sd",
                                       "equal" = "equal",
                                       "pretty" = "pretty",
                                       "quantile" = "quantile",
                                       "kmeans" = "kmeans",
                                       "hclust" = "hclust",
                                       "bclust" = "bclust",
                                       "fisher" = "fisher",
                                       "jenks" = "jenks"),
                        selected = "pretty"),
```

Building a choropleth mapping application

At the UI, continue edit the codes as shown below:

```
            sliderInput(inputId = "classes",
                        label = "Number of classes",
                        min = 6,
                        max = 12,
                        value = c(6)),
            selectInput(inputId = "colour",
                        label = "Colour scheme:",
                        choices = list("blues" = "Blues",
                                       "reds" = "Reds",
                                       "greens" = "Greens",
                                       "Yellow-Orange-Red" = "YlOrRd",
                                       "Yellow-Orange-Brown" = "YlOrBr",
                                       "Yellow-Green" = "YlGn",
                                       "Orange-Red" = "OrRd"),
                        selected = "YlOrRd")
        ),
```

Building a choropleth mapping application

At the server, edit the codes as shown below:

```
server <- function(input, output){
    output$mapPlot <- renderPlot({
        tm_shape(mpsz_agemale2018)+
            tm_fill("DEPENDENCY",
                    n = input$classes,
                    style = input$classification,
                    palette = input$colour) +
            tm_borders(lwd = 0.1,
                       alpha = 1)
        })
}
```

A gentle reminder,

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

Standard R debugging tools

- Tracing
  - print()/cat()/str()
  - renderPrint eats messages, must use cat(file = stderr(), ""...)
  - Also consider shinyjs package's logjs, which puts messages in the browser's JavaScript console
- Debugger
  - Set breakpoints in RStudio
  - browser()
  - Conditionals: if (!is.null(input$x)) browser()

Common errors

"Object of type 'closure' is not subsettable"

- You forgot to use () when retrieving a value from a reactive expression
  plot(userData) should be *plot(userData())*

Common errors

"Unexpected symbol"

"Argument xxx is missing, with no default"

- Missing or extra comma in UI.
- Sometimes Shiny will realise this and give you a hint, or use RStudio editor margin diagnostics.

Common errors

"Operation not allowed without an active reactive context. (You tried to do something that can only be done from inside a reactive expression or observer.)

- Tried to access an input or reactive expression from directly inside the server function. You must use a reactive expression or observer instead.
- Or if you really only care about the value of that input at the time that the session starts, then use isolate().

Testing

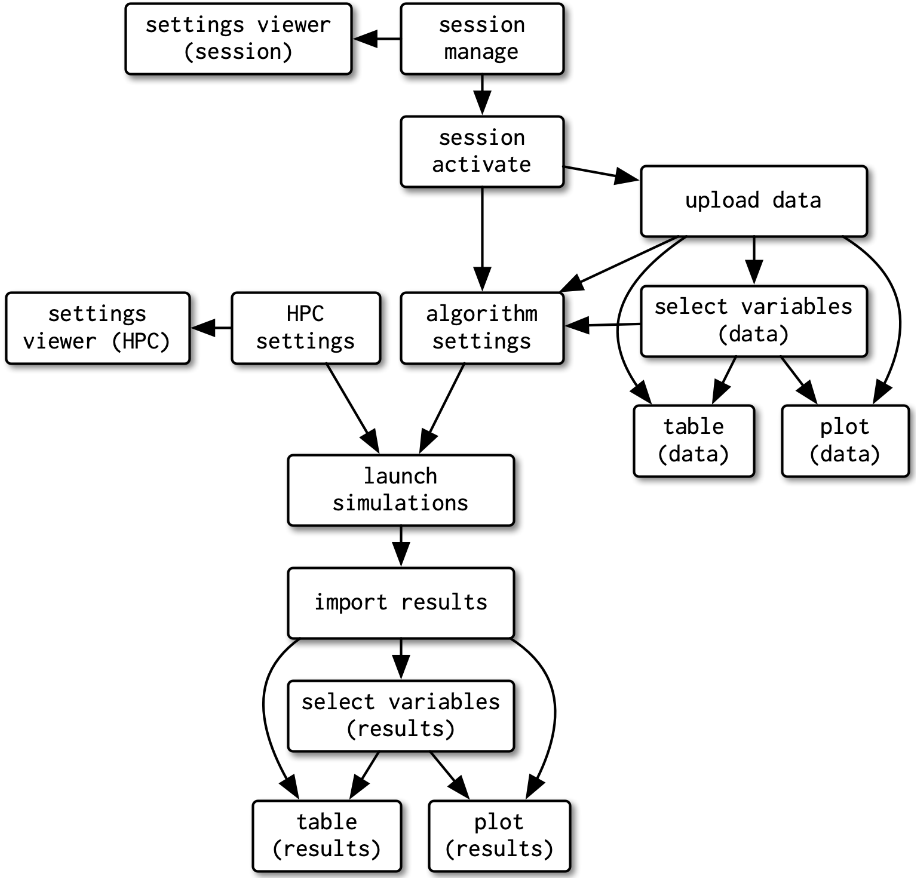
- There are many possible reasons for an application to stop working. These reasons include:
  - An upgraded R package has different behavior. (This could include Shiny itself!)
  - You make modifications to your application.
  - An external data source stops working, or returns data in a changed format.
- Automated tests can alert you to these kinds of problems quickly and with almost zero effort, after the tests have been created.

shinytest

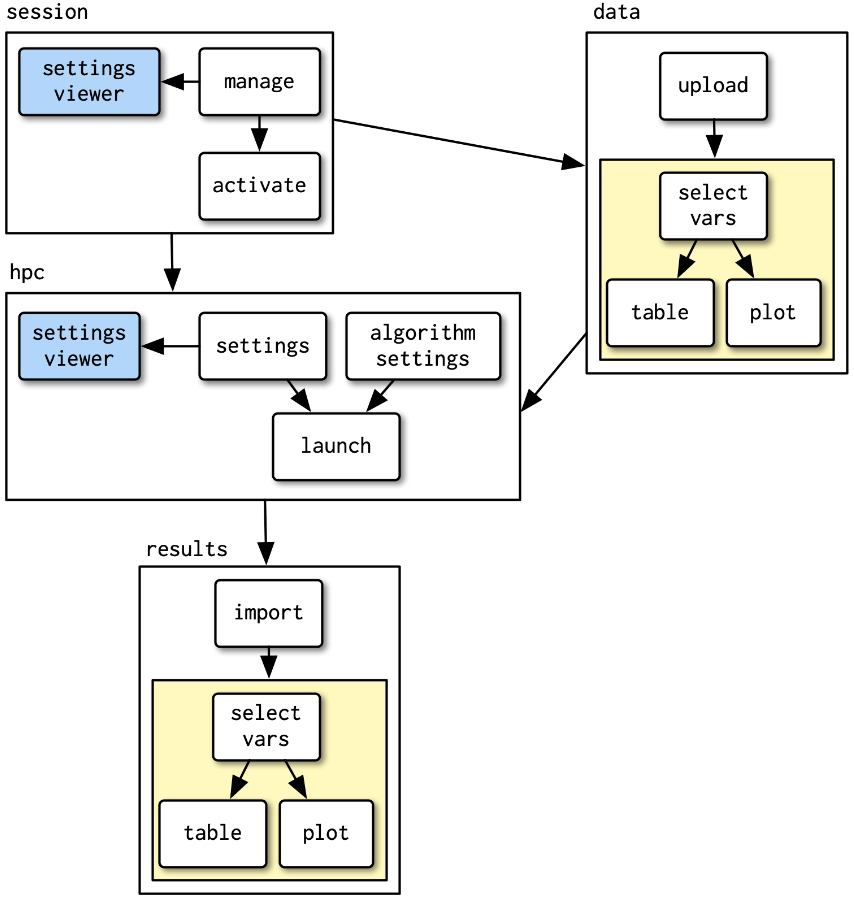
- Shinytest uses snapshot-based testing strategy.
- The first time it runs a set of tests for an application, it performs some scripted interactions with the app and takes one or more snapshots of the application's state.
- These snapshots are saved to disk so that future runs of the tests can compare their results to them.

Introducing Shiny Modules

An example of a large and complex Shiny application diagram.



An example of modulerised Shiny application.



Module basics

A module is very similar to an app. Like an app, it’s composed of two pieces:

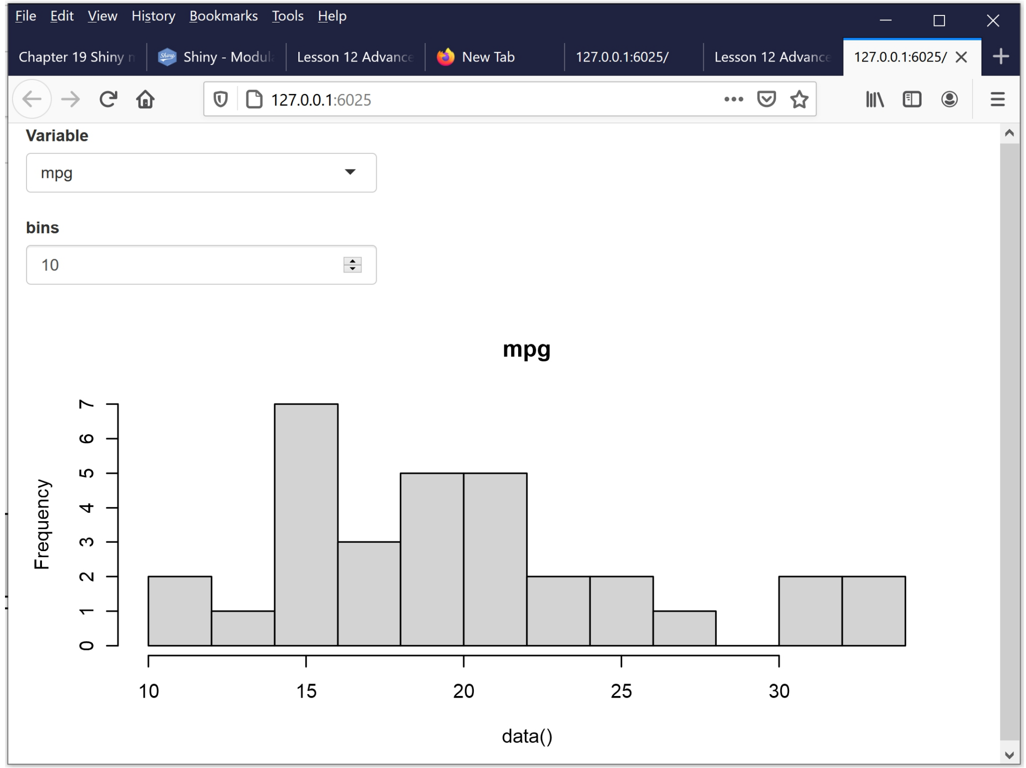
- The **module UI** function that generates the *ui* specification.
- The **module server** function that runs code inside the *server* function.

The two functions have standard forms. They both take an *id* argument and use it to namespace the module. To create a module, we need to extract code out of the app UI and server and put it in to the module UI and server.

The original Shiny application codes

In order to understand the basics of Shiny modules, let us consider a simple Shiny application codes to plot a histogram shown below.

```r
ui <- fluidPage(
  selectInput("var",
              "Variable",
              names(mtcars)),
  numericInput("bins",
               "bins",
               10,
               min = 1),
  plotOutput("hist")
)
server <- function(input,
                   output,
                   session) {
  data <- reactive(mtcars[[input$var]])
  output$hist <- renderPlot({
    hist(data(),
         breaks = input$bins,
         main = input$var)
  }, res = 96)
}
```



Module UI

We’ll start with the module UI. There are two steps:

- Put the UI code inside a function that has an id argument.
- Wrap each existing ID in a call to NS(), so that (e.g.) "var" turns into NS(id, "var").

```
histogramUI <- function(id) {
  tagList(
    selectInput(NS(id, "var"), "Variable", choices = names(mtcars)),
    numericInput(NS(id, "bins"), "bins", value = 10, min = 1),
    plotOutput(NS(id, "hist"))
  )
}
```

Here we have returned the UI components in a *tagList()*, which is a special type of layout function that allows you to bundle together multiple components without actually implying how they will be laid out. It is the responsibility of the person calling *histogramUI()* to wrap the result in a layout function like *column()* or *fluidRow()* according to their needs.

Module server

Next we tackle the server function. This gets wrapped inside another function which must have an id argument. This function calls *moduleServer()* with the *id*, and a function that looks like a regular server function:

```
histogramServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    data <- reactive(mtcars[[input$var]])
    output$hist <- renderPlot({
      hist(data(), breaks = input$bins, main = input$var)
    }, res = 96)
  })
}
```

Note that *moduleServer()* takes care of the namespacing automatically: inside of *moduleServer(id)*, *input$var* and *input$bins* refer to the inputs with names *NS(id, "var")* and *NS(id, "bins")*.]

Revised Shiny Application

Now that we have the ui and server functions, it's good practice to write a function that uses them to generate an app which we can use for experimentation and testing:

```
ui <- fluidPage(
    histogramUI("hist")
    )

server <- function(input, output, session) {
    histogramServer("hist")
    }

shinyApp(ui, server)
```

Note that, like all Shiny control, you need to use the same *id* in both UI and server, otherwise the two pieces will not be connected.

In-class Exercise: Function to import csv file

Module UI function

```r
# Module UI function
csvFileUI <- function(id, label = "CSV file") {
    # `NS(id)` returns a namespace function, which was save as `ns` and will
    # invoke later.
    ns <- NS(id)

    tagList(
        fileInput(ns("file"), label),
        checkboxInput(ns("heading"), "Has heading"),
        selectInput(ns("quote"), "Quote", c(
            "None" = "",
            "Double quote" = "\"",
            "Single quote" = "'"
        ))
    )
}
```

Module server function

```
csvFileServer <- function(id, stringsAsFactors) {
    moduleServer(
        id,
        function(input, output, session) {
            userFile <- reactive({
                validate(need(input$file, message = FALSE))
                input$file
            })
            dataframe <- reactive({
                read.csv(userFile()$datapath,
                         header = input$heading,
                         quote = input$quote,
                         stringsAsFactors = stringsAsFactors)
            })
            observe({
                msg <- sprintf("File %s was uploaded", userFile()$name)
                cat(msg, "\n")
            })
            return(dataframe)
        }
    )
}
```

The Shiny app

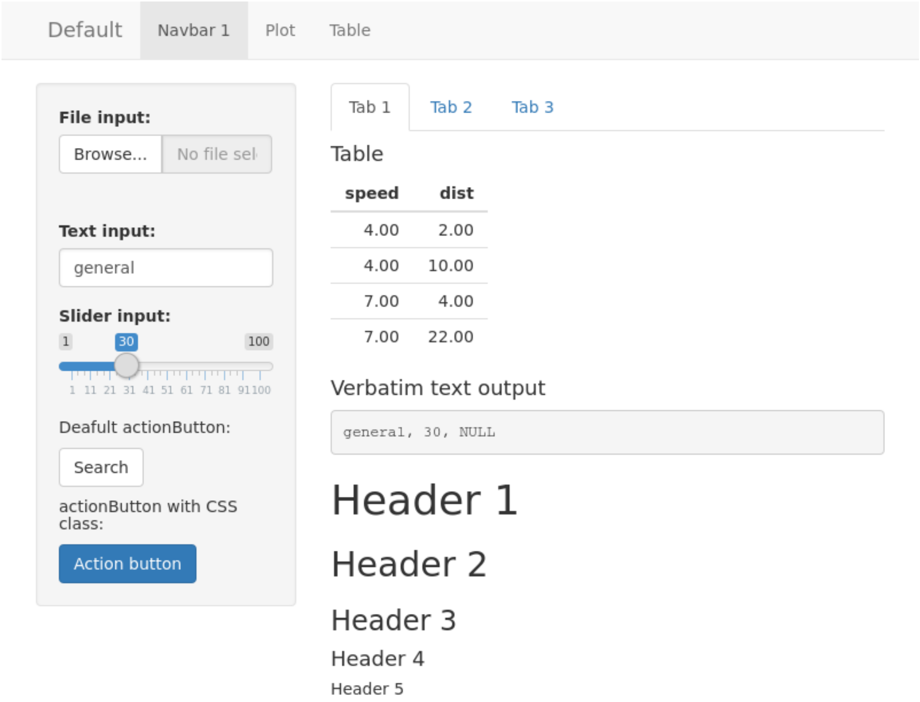
```r
ui <- fluidPage(
    sidebarLayout(
        sidebarPanel(
            csvFileUI("datafile", "User data (.csv format)")
        ),
        mainPanel(
            dataTableOutput("table")
        )
    )
)

server <- function(input, output, session) {
    datafile <- csvFileServer("datafile", stringsAsFactors = FALSE)

    output$table <- renderDataTable({
        datafile()
    })
}

shinyApp(ui, server)
```

Introducing Shiny Module

- As Shiny applications grow larger and more complicated, modules are used to manage the growing complexity of Shiny application code.
- Functions are the fundamental unit of abstraction in R, and we designed Shiny to work with them.
- We can write UI-generating functions and call them from our app, and we can write functions to be used in the server function that define outputs and create reactive expressions.

shinythemes: Themes for Shiny

- It includes several Bootstrap themes from https://bootswatch.com/, which are packaged for use with Shiny applications.



- For detail themes and getting started, refer to the online document.

References

Shiny Module

- Chapter 19 Shiny modules of Mastering Shiny.
- Modularizing Shiny app code, online article
- Communication between modules. This is a relatively old article, some functions have changed.
- Shiny Modules
- Shiny Modules (part 1) : Why using modules?
- Shiny Modules (part 2): Share reactive among multiple modules
- Shiny Modules (part 3): Dynamic module call