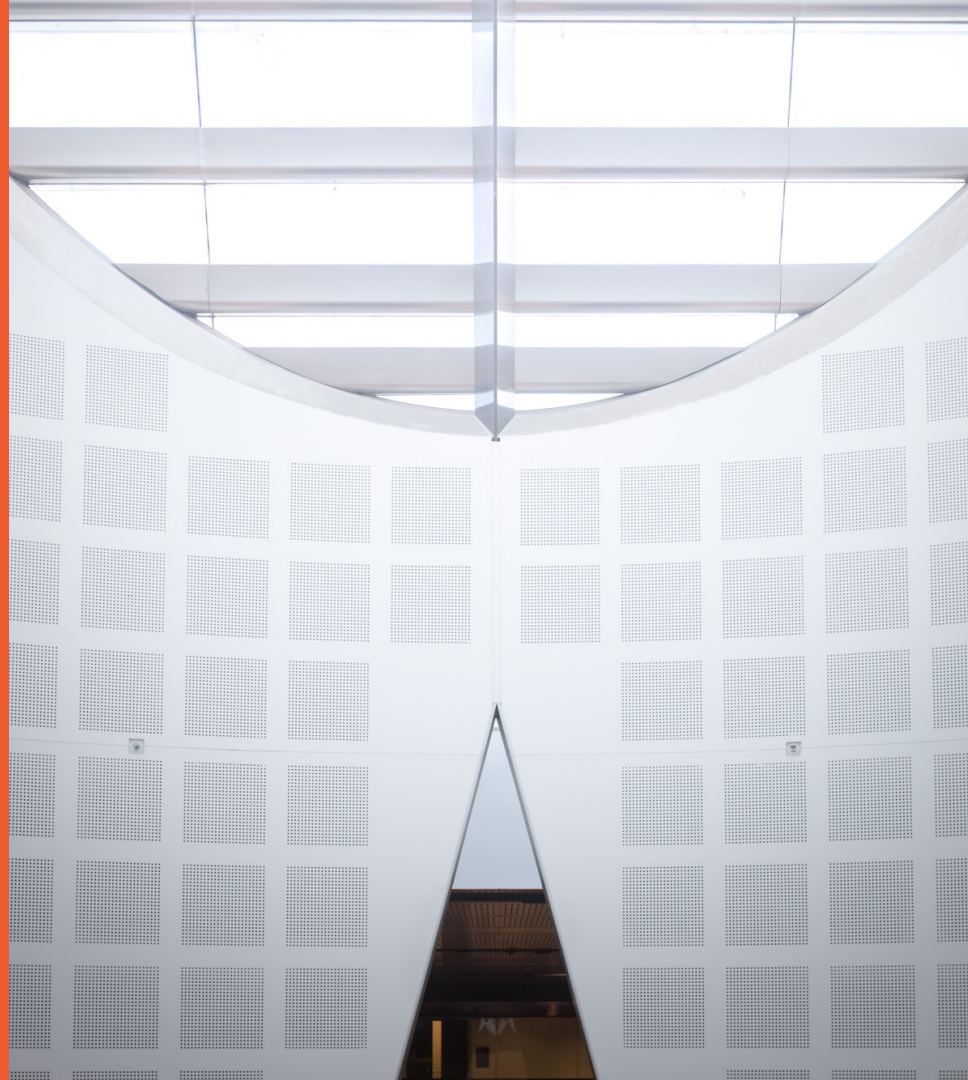
R-Shiny 101

Jazmin Ozsvar, Darya Vanichkina

24th March, 2022

THE UNIVERSITY OF SYDNEY



Welcome to short data science training

- Format: 1.5 hours (45min-1hr + 15-30 min Q&A)
- Upcoming courses: https://tinyurl.com/sihtraining
- Link for suggestions for short courses: https://tinyurl.com/sihshort
- TRAINING MAILING LIST: https://tinyurl.com/sihtraininglist
- Feedback please (at end)

Code of Conduct

- Use welcoming and inclusive language
- Be respectful of different viewpoints and experiences
- "Be nice", and support others' learning
- Full details & how to report issues: https://sydney-informatics-hub.github.io/codeofconduct/

Why (another) Shiny course?

What is Shiny?

- Shiny is an R package for making interactive web apps from within Rstudio
- Based on JavaScript
- Reactive programming (more on this later)
- Used for displaying data (think dashboards) rather than engineering complex apps

https://shiny.rstudio.com/

Shiny gallery

- https://shiny.rstudio.com/gallery/

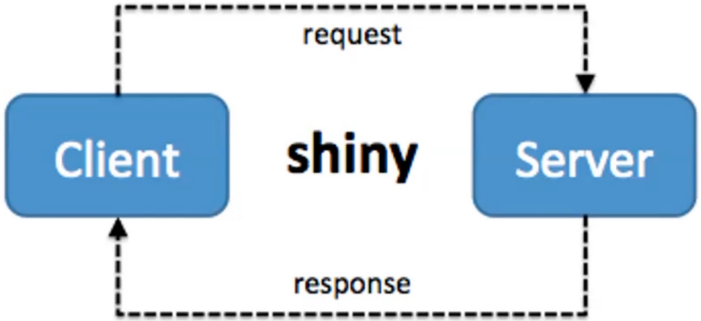
Shiny ecosystem

- Shiny apps are supported by and extended by a huge range of custom packages:
  - https://github.com/nanxstats/awesome-shiny-extensions
- UI/aesthetics: new skins, widgets, D3 based visualisations
- Back end: database connections, API integration
- Deployment: Cloud, Electron
- Development: debugging/testing/prototyping tools

Getting started with Shiny

All Shiny apps have:

- Client side:
  - User interface (UI)
  - Displays output: plots, tables …
  - Widgets for user input
- Server side
  - Responds to input from client side
  - Executes reactive logic
  - Renders outputs



App skeleton (app.R)

```r
# Load package
library(shiny)

# Create UI
ui <- fluidPage(
  "My first app!"
)

# Create server
server <- function(input, output, session) {
}

# Puts the UI and server together
shinyApp(ui, server)
```

Creates a blank page with this text

Defines the server. This is where reactive events will go later.

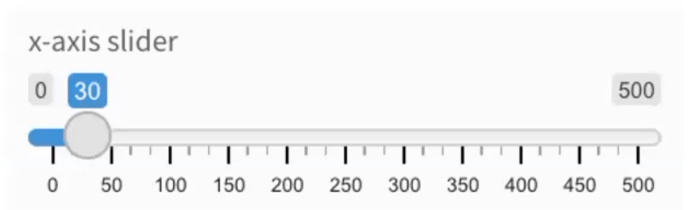
UI elements: inputs

- Input widgets on the client side are denoted with “input” in the name:
  - `sliderInput()`, `numericInput()`, `selectInput()`
- The arguments will always include an InputID and label, the others are dependent on the function:

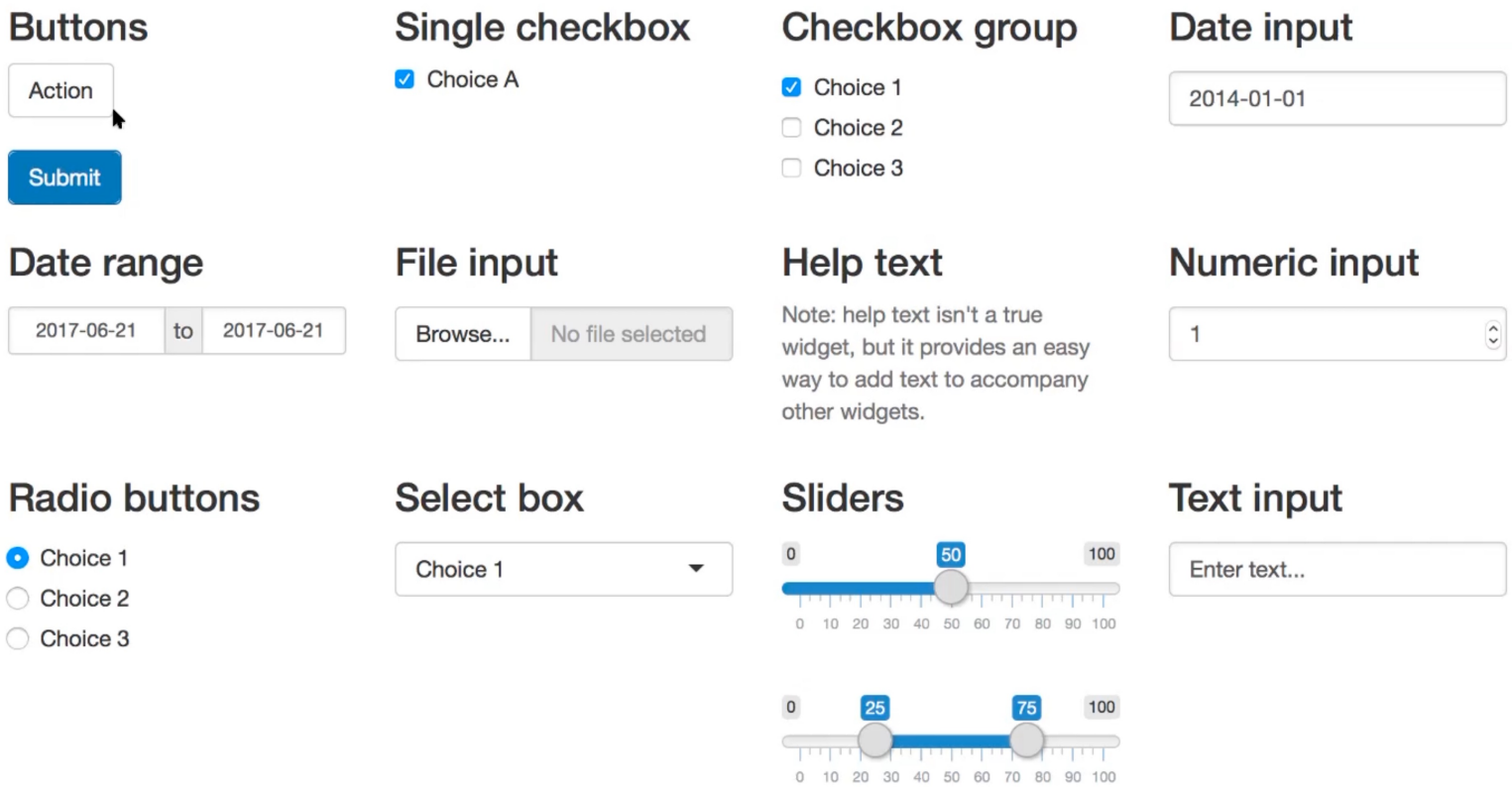
UI code

```
sliderInput(
  inputId = "x_axis_slider",
  label = "x-axis slider",
  min = 0,
  max = 500,
  value = 30
)
```

Interface

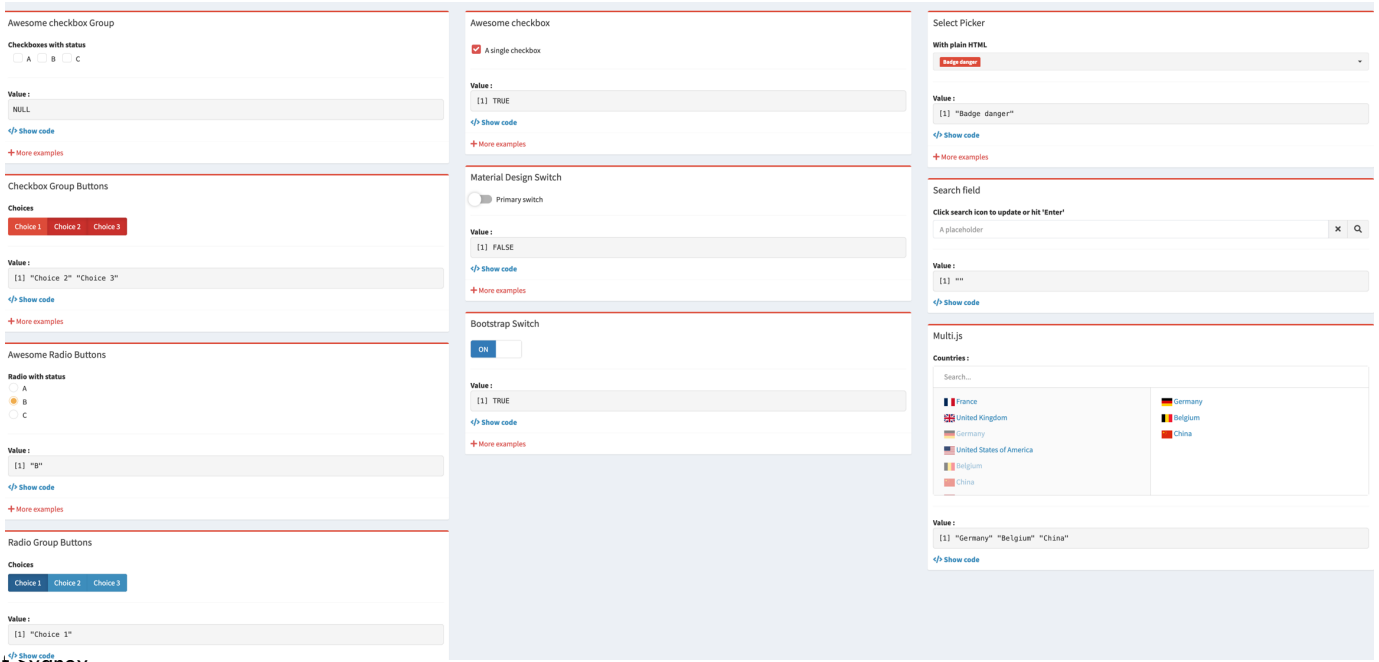


Base Shiny inputs (widgets)



If you want to use more/other widget types

- shinyWidgets: http://shinyapps.dreamrs.fr/shinyWidgets/
  - Progress bars, sliders for text range (months)



Server side: rendering

- Outputs on the server side are defined using the following:

output$object_name ← renderTypeOfElement({})

```
output$reactiveTrajPlot <- renderPlot({

  rv$selected_frames %>%
    filter(
      TF_number >= input$trajectory_TF_number_slider[1],    # Filter data using inputs
      TF_number <= input$trajectory_TF_number_slider[2]
    ) %>%
    trajectory_explorer_plot()    # Plot with a custom function

})
```

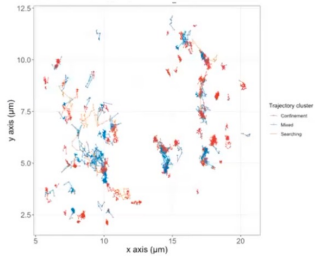
UI elements: outputs

- Outputs on the client side are denoted with “output” in the name
- `plotOutput()`, `textOutput()`, `tableOutput()`, `DT::dataTableOutput()`
- Type of output depends on the render function used in the server
- The output function will always refer to the output object rendered in the server, the other arguments are dependent on the function

UI code

```
plotOutput("reactiveTrajPlot",
           height = "500px",
           width = "600px")
```

Interface



Putting it together

```r
library(shiny)

##### UI #####
ui <- fluidPage(
  titlePanel("A Basic App"),

  sidebarLayout(
    sidebarPanel(
      sliderInput("range",
                  label = "Range of interest:",
                  min = 0, max = 100, value = c(0, 100))
    ),
    mainPanel(
      textOutput("min_max")
    )
  )
)

##### SERVER #####
server <- function(input, output, session) {

  output$min_max <- renderText({
    paste("You have chosen a range that goes from",
          input$range[1], "to", input$range[2])
  })

}

shinyApp(ui, server)
```

Create a slider input labelled "range".

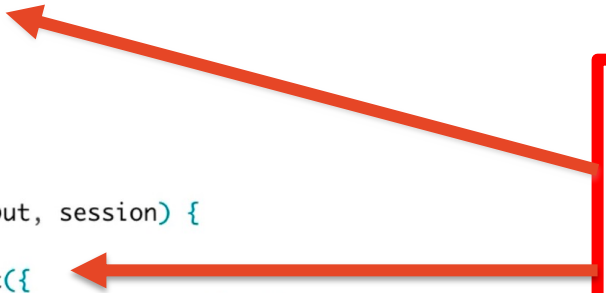
The min and max start as 0 and 100, and the range of possible values is 0 – 100.

```r
library(shiny)

##### UI #####
ui <- fluidPage(
  titlePanel("A Basic App"),

  sidebarLayout(
    sidebarPanel(
      sliderInput("range",
                  label = "Range of interest:",
                  min = 0, max = 100, value = c(0, 100))
    ),
    mainPanel(
      textOutput("min_max")
    )
  )
)

##### SERVER #####
server <- function(input, output, session) {

  output$min_max <- renderText({
    paste("You have chosen a range that goes from",
          input$range[1], "to", input$range[2])
  })

}

shinyApp(ui, server)
```

Our “range” input is referenced as input$range in the server side.

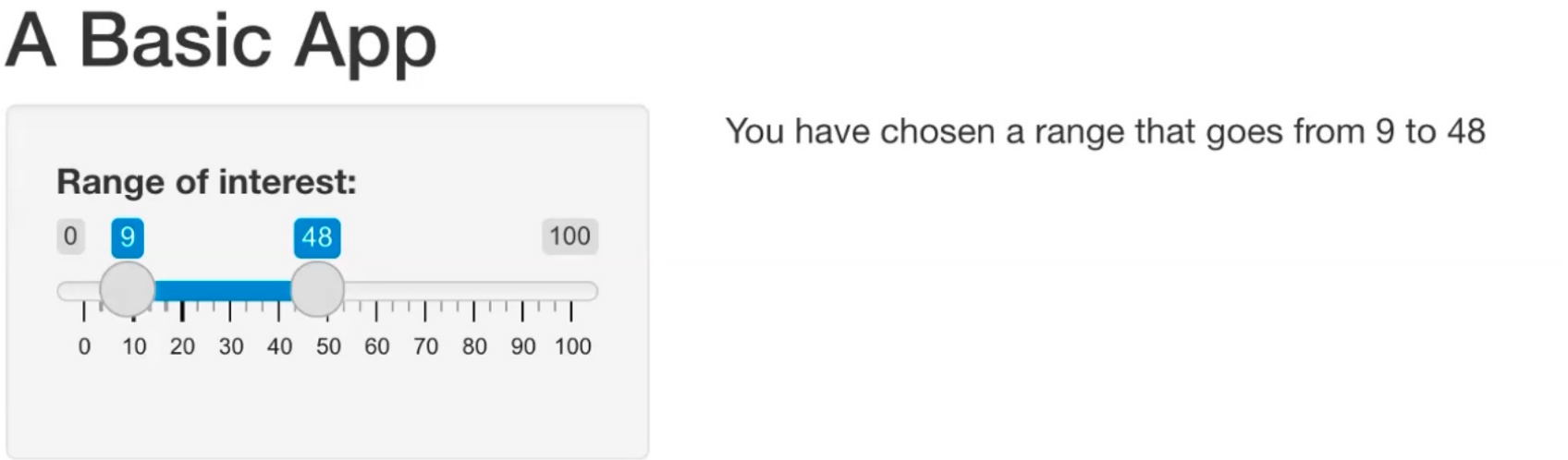
Here we grab the minimum and maximum values from “range” as defined on the UI side.

```r
library(shiny)

##### UI #####
ui <- fluidPage(
  titlePanel("A Basic App"),

  sidebarLayout(
    sidebarPanel(
      sliderInput("range",
                  label = "Range of interest:",
                  min = 0, max = 100, value = c(0, 100))
    ),
    mainPanel(
      textOutput("min_max")
    )
  )
)

##### SERVER #####
server <- function(input, output, session) {

  output$min_max <- renderText({
    paste("You have chosen a range that goes from",
          input$range[1], "to", input$range[2])
  })

}

shinyApp(ui, server)
```



The min_max output that relies on "range" is a text object. We render the text on the server side and display it as output on the client side.

The UI of our app



Reactivity

Non-reactive variables

- A non-reactive variable is one that is not automatically updated
- Non-reactive variables can be updated over time, but this is done through manual assignment or through recalculating the variable using a function
- For example, consider the conversion b/w Celsius & Farenheit

```
temp_c <- 10
temp_f <- (temp_c * 9 / 5) + 32
temp_f
#> [1] 50
```

Changing temp_c does not automatically update temp_f

From Mastering Shiny

Non-reactive variables - functions

- What if we tried to update temp_f using a function?
- We hit a similar problem: computation of temp_f() is not minimized – temp_c might still have the same value

```
temp_c <- 10
temp_f <- function() {
  message("Converting")
  (temp_c * 9 / 5) + 32
}
temp_f()
#> Converting
#> [1] 50
```

Reactive programming

- Allows the AUTOMATIC recalculation of variables on the conditions that their dependencies change

Reactive values

- For data to be handled reactively, it needs to be made reactive first.
- There are two functions for coercing data to become reactive:
  - `reactiveVal()` – creates a single reactive value
  - `reactiveValues()` – creates a container that can hold multiple reactive values

reactiveVal()

- Creates a single reactive value

```
an_arbitrary_number <- reactiveVal({1000})
```

- This reactive value is globally accessible within the Shiny app

reactiveValues()

- Creates a list-like object that stores multiple reactive values

```
reactive_container <- reactiveValues({
  threshold = 1000,
  data = my_data_frame,
  user_list = c("John", "Mary", "Pusheen")
})
```

- The values can be called like the columns of a data frame using $
- If you want to access columns: `reactive_container$data$col1`

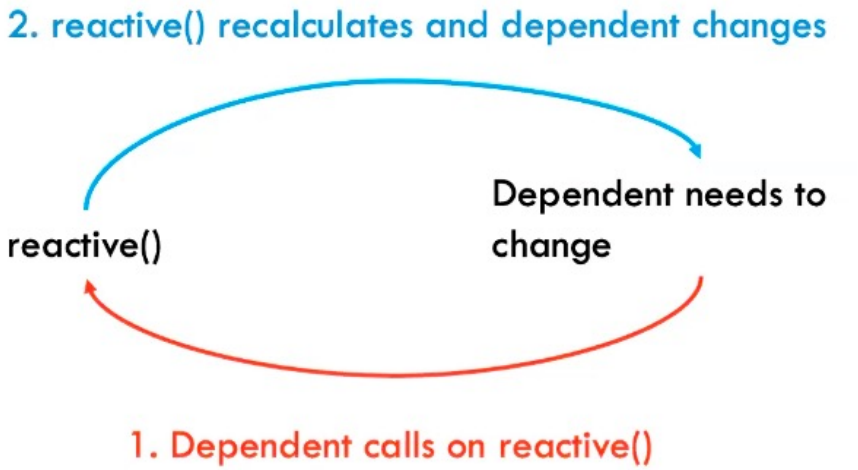
Evaluating a reactive expression

- Evaluates and expression and saves it as a reactive function

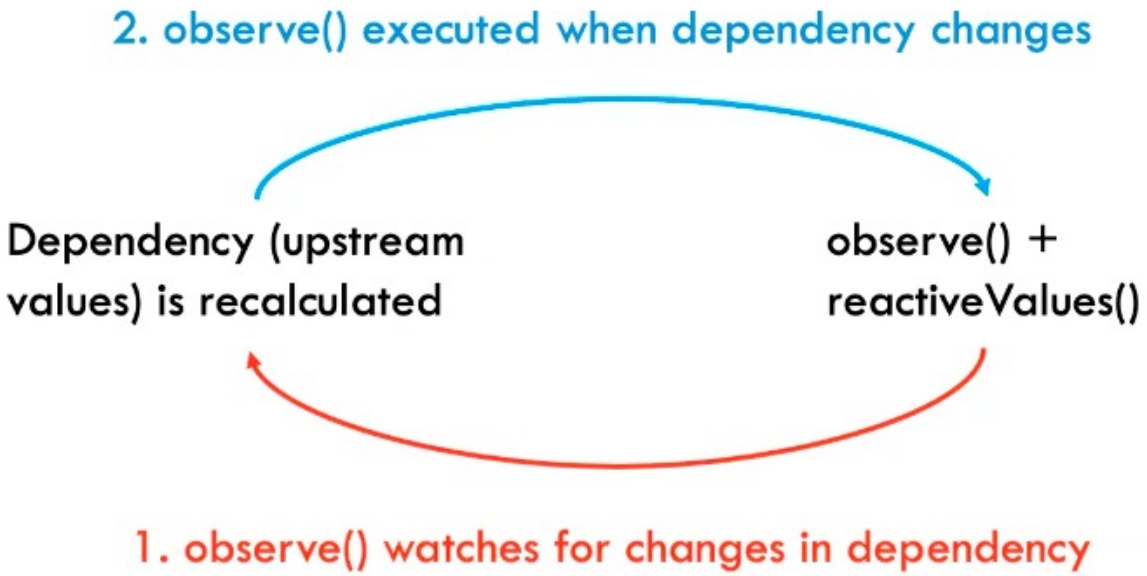
```
filter_dataframe <- reactive({
  my_data_frame %>%
    filter(column1 == FALSE)
})
```

- In most cases, you can use the reactive expression by calling it with parenthesis: filter_dataframe(). However, some modules (e.g. selectize modules) may require it without parentheses.

Lazy evaluation



Contrast with – Eager evaluation



What if my data isn't reactive

- Any calculations conducting on non-reactive data will only be evaluated within the context in which it is called. The changes will not be global and will not be passed to other reactive values

The observe() family

- Reactive values and expressions can be updated or are evaluated through a number of expressions:
- `observe()` – executes code when any upstream dependency of the reactive values in the expression within the observe statement changes
- `observeEvent()` – triggers evaluation of a value on specific inputs
- `eventReactive()` - triggers evaluation of a reactive expression on specific inputs

The observe() family

observe() watches for any changes to the reactive values within its environment
(can retrieve values, can't assign values)

```
observe({
  reactive_data %>%
    select() %>%
    filter()
})
```

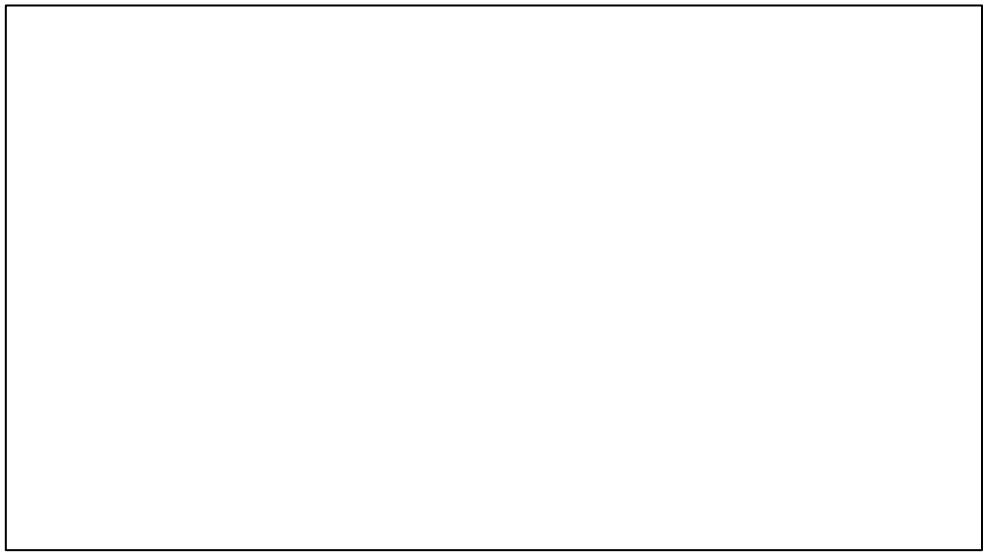
observeEvent() triggered by specific inputs for expression to be evaluated (can both retrieve and assign new variables)

```
observeEvent(input$button, {
  new_variable <- reactive_data %>%
    select() %>%
    filter()
})
```

UI elements

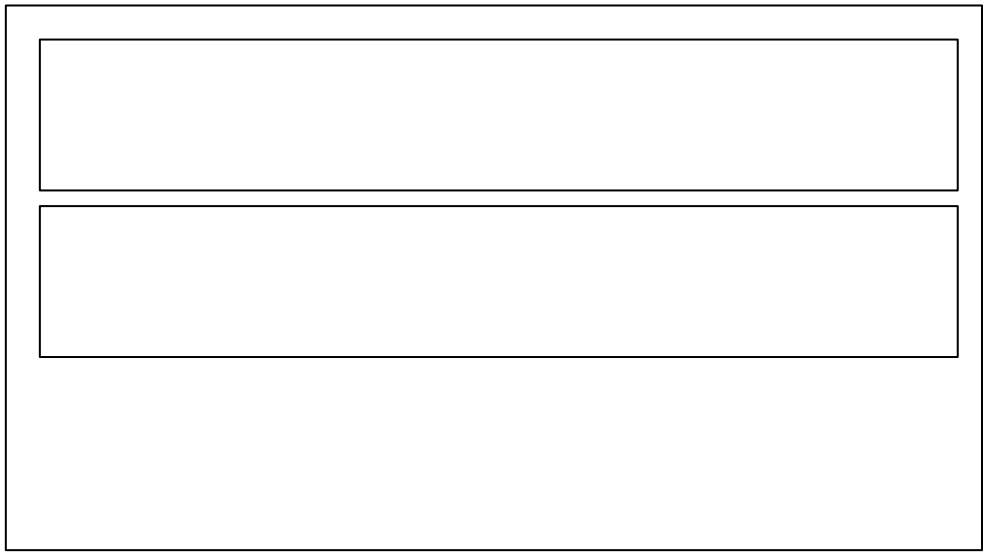
Grid UI

- R Shiny uses a grid (similar to Bootstrap) to build interfaces
- Define the UI using `fluidPage()`



Grid UI

- R Shiny uses a grid (similar to Bootstrap) to build interfaces
- Define the UI using `fluidPage()`

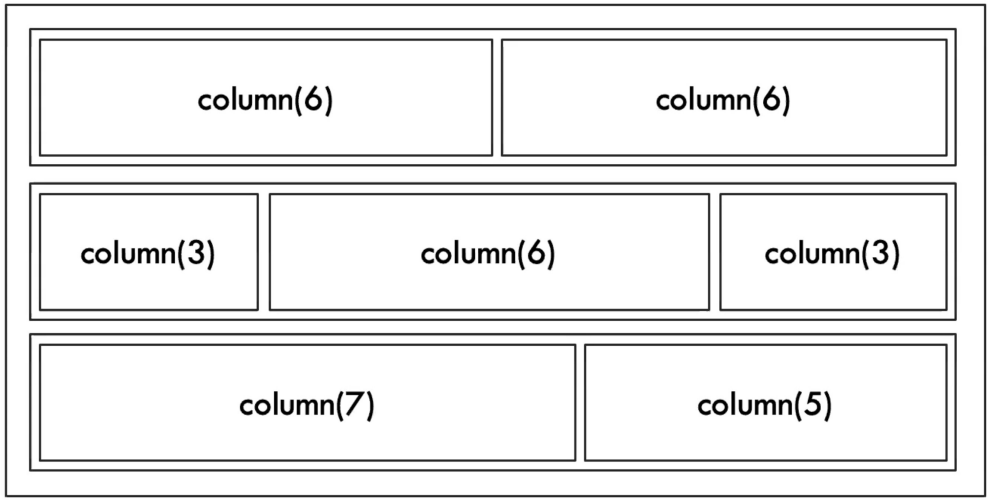


Define some rows within the page using `fluidRow()`.

These will stack on top of each other and expand as required by the objects within them.

Grid UI

- R Shiny uses a grid (similar to Bootstrap) to build interfaces
- Define the UI using `fluidPage()`



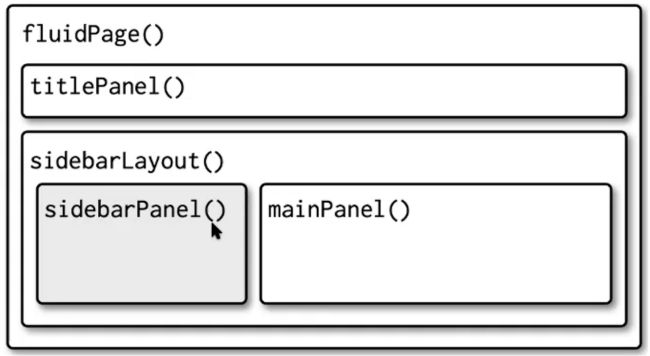
Define columns within the row using `column()`.

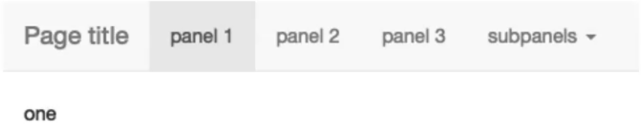
The width can be specified, ex `column(3)`.

Max column width is 12.

Other UI elements

- Shiny contains many off the shelf UI elements to use:





https://mastering-shiny.org/action-layout

UI elements: text

- Text can be added with HTML-derived tags

HTML	Shiny
"This is a paragraph."	p("This is a paragraph.")
"This is a header."	h3("This is a header.")

Deployment

Deployment

- Can be hosted on web pages, deployed locally or embedded in RMarkdown
- Dockerisation of apps simplifies local deployment
- Can host on VM or server
- ShinyApps.io

How to decide if Shiny is right for you?

Why Shiny?

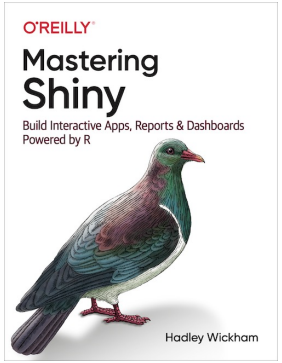
1. Part of the R ecosystem – can process, analyse and visualise data however you want using R
2. Plays well with ggplotly (enhances interactivity of graphics)
3. Shiny apps are simple to share – can download from GitHub and run locally, or deploy to a web page, can also dockerise
4. Lots of packages have been made to enhance Shiny dashboards (shinywidgets, shinyjs)
5. Can make custom widgets provided you know JavaScript
6. Can be gussied up with some CSS
7. Free and open source!! ☺

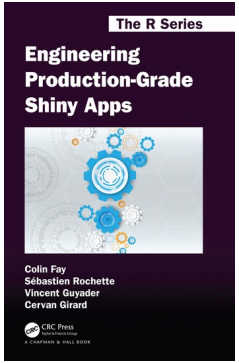
Why NOT Shiny?

1. Steep learning curve (coupled with things getting messy very quickly)
2. The reactive workflows can turn into a nightmare i.e. reactions can negate reactions, or even create a never ending event loop! – a good mental model is essential
3. Troubleshooting can take ages!! ☹
4. Shiny can be laggy if your code/data structure is not optimised, or if you've overengineered a workflow, or if you're trying to display lots of data in a ggplotly visual
5. Not recommended for BI solutions – a lot of time is spent developing the reactivity of the Shiny app relative to an ETL/analysis
6. Not as simple to get started with as a tool like Power BI, which allows drag-and-drop UI development and where all the interactivity is taken care for you

Resources

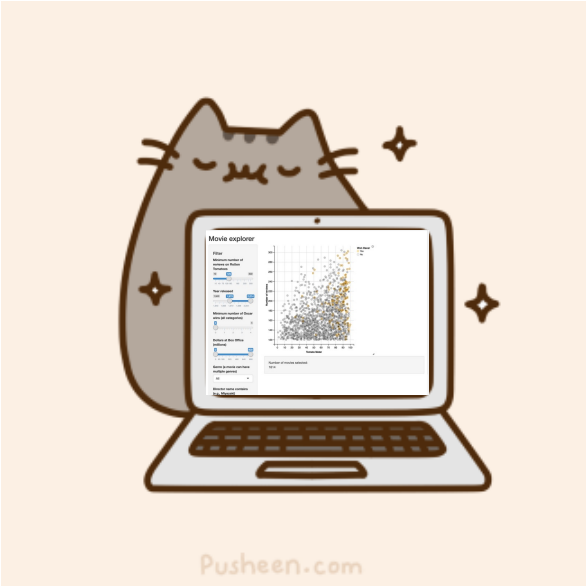
- https://shiny.rstudio.com/tutorial/
- https://mastering-shiny.org/
- https://stackoverflow.com/
- https://engineering-shiny.org/index.html





Questions?

When your Shiny code works



When your Shiny app breaks