

# STAT5003

## Week 1: Shiny interactive graphics

Dr. Justin Wishart



# Overview of a app

```
## — Installed icons ————— icon 0.2.0 —
```

```
## x ionicons          x google_material
## x academicons       x feather_icons
## x simple_icons      x octicons
## x bioicons          ✓ fontawesome      5.15.3
```

```
##
```

```
## Attaching package: 'icons'
```


```
## The following object is masked from 'package:fontawesome':
```

```
##
```

```
##      fa
```

## What is a Shiny app?



- An R package that creates interactive web applications
  - without needing to know HTML, CSS, or JavaScript
- Combines two things
  - Statistical Power of  R
    - Use any analysis that can be coded in R.

# What is reactive output?

- Output of program/interface reacts to user input
- Not a new concept at all
  - Easiest pervasive example of this is MS Excel!
- Exercise for interested reader: Shiny apps equivalent

# Deployment of applet.

- Local deployment: Can run on any machine with R installed and a modern web browser. Syntax below
  - `> shiny::runApp(<path-to-my-app>)`
- Hosted on [shinyapps.io](https://shinyapps.io)
  - RStudio server built to host shiny apps (free and paid options).
- Server side deployment:
  - Can host the app on a server running the Rstudio server software.

# Basic File Format of Applet.

- Two ways
  - Newer: Single file **app.R**
  - Older legacy way: Two Files
    - **server.R** (Analysis code)
    - **ui.R** (Display code)

# Basics of an App

Open apps/intro-01.R

```
library(shiny)

# Define UI
ui <- fluidPage()

# Define server interactions
server <- function(input, output) {}

# Run the application
shinyApp(ui = ui, server = server)
```

- Controls layout and appearance
  - user input area
  - output drawn
  - It really is HTML/CSS/Javascript
- Server needs code to
  - deal with inputs

# Skeleton of standalone app file

```
library(shiny)

# Define UI
ui <- fluidPage()

# Define server interactions
server <- function(input, output) {}

# Run the application
shinyApp(ui = ui, server = server)
```

- Four essential parts



# Skeleton of standalone app file

```
library(shiny)
```

```
# Define UI
```

```
ui <- fluidPage()
```

```
# Define server interactions
```

```
server <- function(input, output) {}
```

```
# Run the application
```

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

- A call of the shiny library.

# Skeleton of standalone app file

```
library(shiny)
```

```
# Define UI
```

```
ui <- fluidPage()
```

```
# Define server interactions
```

```
server <- function(input, output) {}
```

```
# Run the application
```

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

- A definition of the **user interface (ui)**
  - Inputs and where they are
  - Where the outputs should be
  - HTML/Javascript/CSS goes here

# Skeleton of standalone app file

```
library(shiny)

# Define UI
ui <- fluidPage()

# Define server interactions
server <- function(input, output) {}

# Run the application
shinyApp(ui = ui, server = server)
```

- Server code instructions/recipe
  - How the server should create output from user input
  - When to **listen** for input changes (i.e. **reactive**)

# Skeleton of standalone app file

```
library(shiny)

# Define UI
ui <- fluidPage()

# Define server interactions
server <- function(input, output) {}

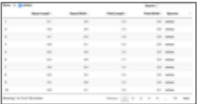

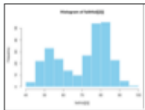

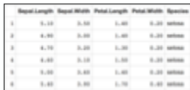

# Run the application
shinyApp(ui = ui, server = server)
```

- A call to run the application.

# Place Output in App - Two main families of functions

- `*Output` set that anchors the output in the UI
- `render*` set that tells Shiny how to make the output
- Segue: R is a sequential language
  - Classic R, each line is run sequentially.
  - Reactive language is not quite the same.
  - Shiny decides when to react.
  - When it reacts, it needs instructions to carry out!
  - You give Shiny instructions to do a set of tasks.
  - Shiny does the tasks, in the order it wants.

# render *and* Output Links

Visual Output	render*	*Output
	<code>DT::renderDataTable(expr,...)</code>	<code>dataTableOutput(outputId)</code>
	<code>renderImage(expr,...)</code>	<code>imageOutput(outputId,...)</code>
	<code>renderPlot(expr,...)</code>	<code>plotOutput(outputId,...)</code>
	<code>renderPrint(expr,...)</code>	<code>verbatimTextOutput(outputId)</code>
	<code>renderTable(expr,...)</code>	<code>tableOutput(outputId)</code>
<code>foo</code>	<code>renderText(expr,...)</code>	<code>textOutput(outputId, ...)</code>
	<code>renderUI(expr,...)</code>	<code>uiOutput(outputId, ...)</code>

# Example: put \*Output call in UI.

```
library(shiny)
```

```
ui <- fluidPage(  
  plotOutput("myShinyPlot")  
)
```

```
server <- function(input, output) {}
```

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

- \*Output(<outputId>) call goes in the UI area
  - e.g. plotOutput("myShinyPlot") shown above

# Example: Linking it to server render

```
library(shiny)

ui <- fluidPage(
  plotOutput("myShinyPlot")
)





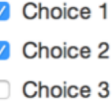
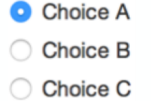

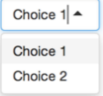

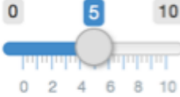

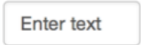
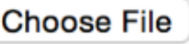
server <- function(input, output) {
  output$myShinyPlot <- renderPlot({
    hist(faithful[, 1])
  })
}

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

- `output$<outputId> <- render*(<output code>)` in server area
  - This is a boring plot that isn't interactive!
  - Need a statement with `input$<inputId>` to make it **reactive**!



# Basic Reactive Inputs

	<code>actionButton(inputId, label)</code>		<code>numericInput(inputId, label, value)</code>
	<code>actionLink(inputId, label)</code>		<code>passwordInput(inputId, label)</code>
	<code>checkboxGroupInput(inputId, label, choices)</code>		<code>radioButtons(inputId, label, choices)</code>
	<code>checkboxInput(inputId, label)</code>		<code>selectInput(inputId, label, choices)</code>
	<code>dateInput(inputId, label)</code>		<code>sliderInput(inputId, label, min, max)</code>
	<code>dateRangeInput(inputId, label)</code>		<code>textInput(inputId, label)</code>
	<code>fileInput(inputId, label)</code>		

# Skeleton Input structure

```
ui <- fluidPage(  
  selectInput(inputId = "myInput",...),  
  plotOutput(outputId = "myOutput")  
)  
  
server <- function(input, output) {  
  output$myOutput <- renderPlot({  
    plot(x = input$myInput)  
  })  
}
```

- input is an R environment
  - \*Input put in the UI area
  - objects accessed/references with `input$<inputId>`
  - `<inputId>` is a character string label of your choice
  - **All** inputs require an `<inputId>`
    - Only the old `submitButton` doesn't need an `<inputId>`

# Input structure

```
ui <- fluidPage(  
  someInput(inputId = "myInput",...),  
  someOutput(outputId = "myOutput")  
)  
  
server <- function(input, output) {  
  output$myOutput <- renderPlot({  
    plot(x = input$myInput)  
  })  
}
```

- The Output is placed in the UI with one of the `*Output` functions
- Server code written which depends on the `input$<inputId>`
  - Can be in the `render*` function
  - Can be in a **reactive** object (more on this later)

# Example: Choosing the dataset.

Open apps/intro-02.R

```
ui <- fluidPage(  
  selectInput(inputId = "datachoice",  
              label = "Choose dataset to view:",  
              choices = c("eruptions", "waiting"),  
              selected = "eruptions"),  
  plotOutput("myShinyOutput")  
)
```

# Exercise

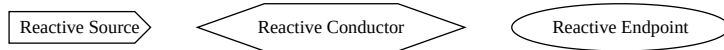
Add another input to the basic faithful histogram app

- Add a slider to control the number of bins with the following arguments
  - **inputId = numBins**
  - **label = "Select number of bins"**
  - **min = 1**
  - **max = 50**
  - **value = 30**
- Use this input variable to control the number of bins as an argument in the **geom\_histogram** function. (see ? geom\_histogram, the bins argument)
- Run the app and see the result

Reactive output in 

# Reactivity

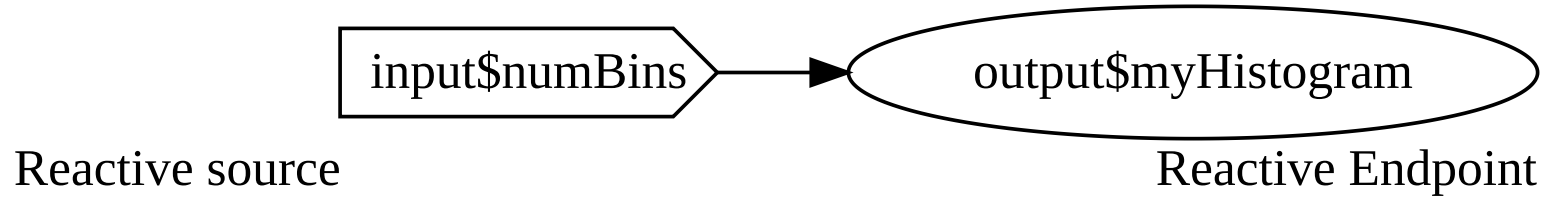
## Visual representation



- What are reactive sources?
  - `input` and `reactiveValues`
- What are reactive conductors?
  - `reactive`
- What are reactive endpoints?
  - output objects (`render*`)
  - More generally an observer
- **NB:** The node shapes shown here differ slightly to the RStudio shapes on their website.

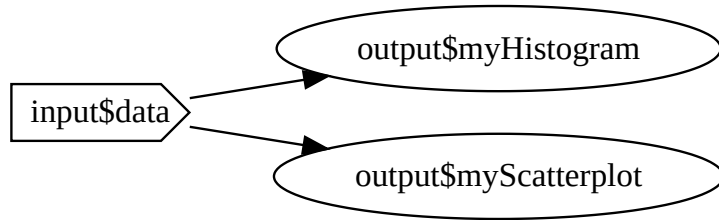
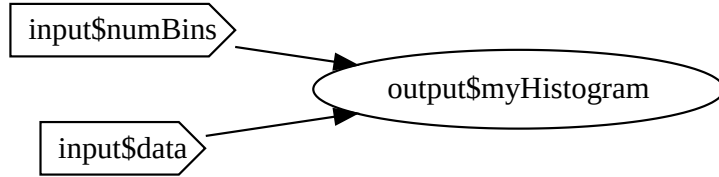
# Simple reactivity

- User input is the reactive source
- User output depends directly on the source





# Multiple sources and outputs

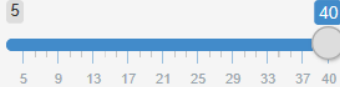
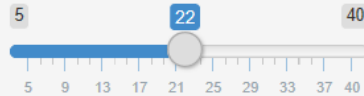


- Sources can impact on multiple outputs.
- Endpoints can depend on multiple sources.

# What is `input$<inputId>`?

- `input` is actually an R **environment**
  - Similar to a **list**
- elements accessed with `input$<inputId>`
- **Updates** each time user changes the input in UI.
- Linked to output in the **server** area.

# Examples of changing input

User Interface	input\$<inputId>
<input checked="" type="checkbox"/> Show 95% confidence bands	→ input\$showBands = TRUE
<input type="checkbox"/> Show 95% confidence bands	→ input\$showBands = FALSE
<div>Sample size</div> <div>128</div>	→ input\$n = 128
<div>Signal to Noise Ratio</div> <div></div>	→ input\$SNRslider = 40
<div>Signal to Noise Ratio</div> <div></div>	→ input\$SNRslider = 22

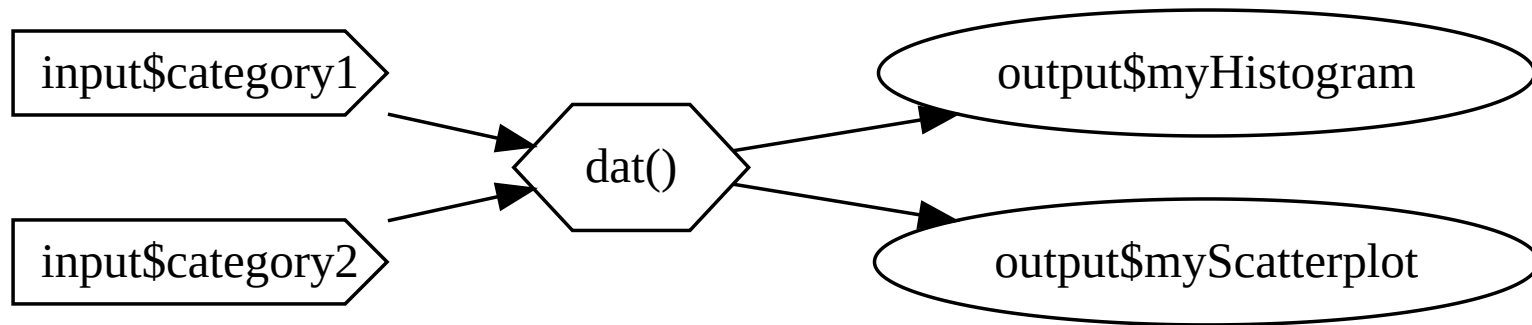
# Reactive Output

```
ui <- fluidPage(  
  sliderInput(inputId = "numBins",  
             label = "Select number of bins",  
             min = 1, max = 50, value = 30),  
  plotOutput("myShinyOutput")  
)  
  
server <- function(input, output) {  
  output$myShinyOutput <- renderPlot({  
    ggplot(faithful, aes_string(faithful[, 1])) +  
    geom_histogram(bins = input$numBins)  
  })  
}
```

- **Reacts:** Each time `input$numBins` updates
  - `output$myShinyOutput` is **invalidated** (out of date)
  - Server re-runs `renderPlot` with supplied *recipe*.

# Reactive Conductor

- Reactive component between source and endpoint.
- Again can have
  - one or more dependencies (parents)
  - one or more dependents (children)



- Useful if computing the common data takes time.

# Reactive object

- A reactive object turns a standard expression into a reactive expression.
- Essentially when its inputs change, it
  - re-evaluates itself
  - invalidates everything that depends on it, so they re-evaluate themselves.
- Typical syntax given below.
  - `my_reactive <- reactive({ input$something; other_reactive(); })`
  - `my_reactive()` returns the current value of `my_reactive`
  - `my_reactive` gives the reference or reactive expression.

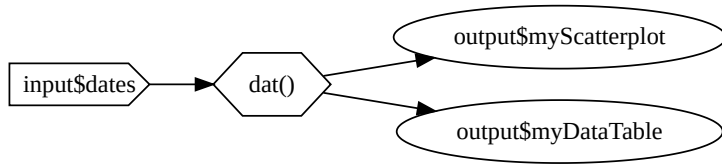
# Example: Reactive Conductor

```
ui <- fluidPage(  
  sliderInput(inputId = "n",  
    label = "Select number of observations to simulate",  
    min = 1, max = 50, value = 30),  
  plotOutput("myShinyOutput"),  
  verbatimTextOutput("mySummary")  
)  
  
server <- function(input, output) {  
  dat <- reactive({  
    rnorm(input$n)  
  })  
  output$myShinyOutput <- renderPlot({  
    boxplot( dat() )  
  })  
  output$mySummary <- renderPrint({ summary( dat() ) })  
}
```

- Reactive data, `dat()`, is computed once
  - Used twice!
  - Don't forget the `()` to use it

## Example:

- Too many observations in your Data
- Suppose you want to filter by time window



1. Add a UI element to select Date range
2. Filter by date range and assign it as a reactive (reactive conductor)
3. Change the output to depend on the reactive object



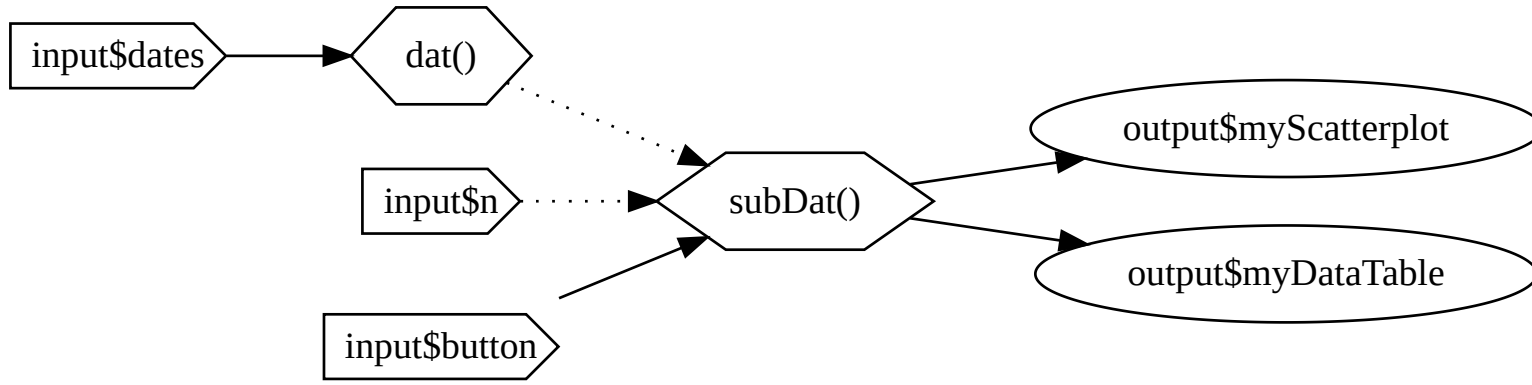
# eventReactive and actionButton

- **eventReactive**(eventExpr, valueExpr): delays a reaction
  - eventExpr is the input to cause reaction
  - valueExpr is the code block to create reactive data
  - Any reactives in valueExpr block are **isolated**

```
mydat <- eventReactive(input$x, {  
  input$y  
  input$z  
  return(something)  
})
```

- I.e. input\$x causes mydat() to update, the other inputs don't
- Simple application, make eventExpr a button!

# Visually

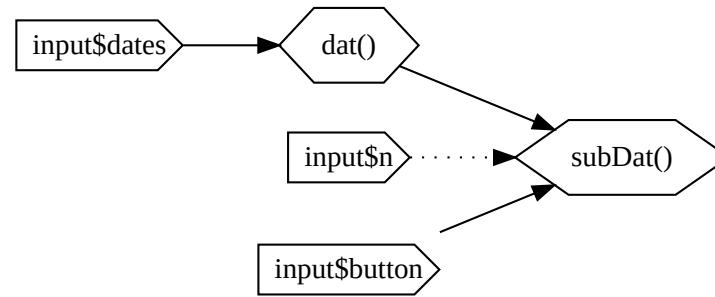


- An updated `input$n` or `dat()` does **not** invalidate `subDat()`
  - Both `input$n` and `dat()` are isolated.
- Clicking `input$button` **does** invalidate `subDat()`

# Alternative reactivity

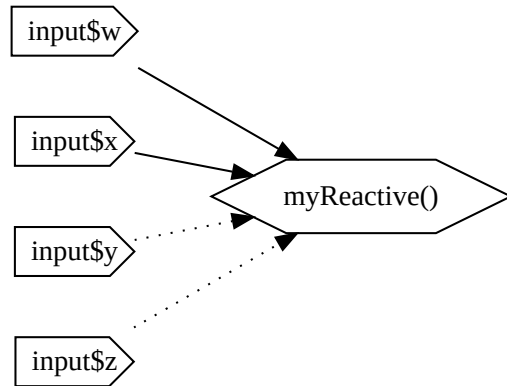
- Suppose you want it to update if either `dat()` or `input$n` updates
  - But **not** when the slider is moved.
- Use a reactive with an `isolate` command.

```
subDat <- reactive({  
  input$sampButton  
  dat() %>% sample_n(isolate(input$n))  
})
```



# Finer control reactions

- An eventReactive will re-run, **every** time the **first** argument changes.
- A reactive will re-run code, **every** time **any** of the input it depends on change.
  - An isolate can be used in a reactive to specify which reactions are needed



```
myReactive <- reactive({  
  input$w  
  input$x  
  isolate({  
    input$y  
    input$z  
    return(<something depending on w, x, y and  
  })  
})
```

# Observers and observeEvent

- Observers **don't** return values in R
  - Can't call an observer
- Useful for making *side effects*
  - E.g. updating the UI.
  - Sending messages to console.
  - Saving data to file
  - plot something

# General observer

- Runs code when input(s)/reactive(s) change

```
observe({  
  input$x  
  input$y  
  <run this code>  
})
```

# observeEvent

- observer equivalent of eventReactive!
- observeEvent(eventExpr, handlerExpr)
  - Runs handlerExpr code when eventExpr updates
- Useful combo, create a save data file button!

```
observeEvent(input$saveButton, {  
  # Something that does a side-effect  
})
```

# Observers vs reactive

	<b>reactive()</b>	<b>observe()</b>	<b>function()</b>
Callable	Yes	No	Yes
Returns values	Yes	No	Yes
Side effects	No	Yes	Optional
Evaluation	Lazy	Eager	Lazy

- Lazy evaluation guide at [Hadley's Advanced R](#)



