Shiny Advanced

Antonio Vetrò

Version 1.1 - April 2021

License

Table of contents

Interactive graphs

Interactivity mechanism

- Interactivity in Shiny is natively supported on plots (`plotOutput()`) and images (`plotImage()`)
- Mouse-based interaction: click and double-click, hovering, brushing
  - Touch events support is under development
- When an interaction event occurs, the mouse coordinates are sent to the server as input$ variables as specified by the action corresponding arguments `click`, `dblclick`, `hover`, `brush`

Coordinates

- In `plotOutput()`:
  - coordinates are scaled to the data space when using base graphics and ggplot2,
  - otherwise raw pixel coordinates are used
- In `imageOutput()`: raw pixel coordinates are used

Interactivity: basic flow

1. The underneath JavaScript captures the mouse event
2. The event triggers invalidation of the corresponding input in Shiny
3. All the downstream reactive consumers are recomputed
4. `plotOutput()` generates a new png image and sends it to the browser

Notes on interactivity with ggplot2

- The code in `renderPlot()` should return a ggplot object, otherwise the coordinates are not properly scaled to the data space (e.g., when using `print(plot)`).
- The arguments `clickId` and `hoverId` do not work for ggplot2 (and for any other grid-based graphics, such as those in library lattice and grid)

Events: click

- Argument `click` is set to NULL by default
- It can be set to:
  - a value,
  - the object returned by `clickOpts(id, clip = TRUE)`
  - a custom function

Events: click - set by value

- When using a value (e.g., “plot_click”), the plot sends the coordinates of the clicked point
- The value is accessed via `input$plot_click`, which gives a named list with x and y elements
  - hence you access them via `input$plot_click$x` and `input$plot_click$y`

Events: click - set by clickOpts function

- `clickOpts(id="plot_click")` is equivalent to setting `click = "plot_click"`
  - with `clip = TRUE` (default), the click area is clipped to the plotting area
  - with `clip = FALSE`, the server receives click events also when the mouse is outside the plotting area (but within the image borders)

Examples with built-in dataset in R

- R has several built-in datasets (https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html)
  - check library `datasets`
- In our examples we will use the Motor Trend Car Road Tests (https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html) dataset
  - fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models, from the 1974 Motor Trend US magazine)
- To get used to it, you can use `data("mtcars")`, `View(mtcars)`, `class(mtcars)`, `?mtcars` and then leverage your R skills
- Alternatively, you may look at this presentation

mtcars data in a nutshell

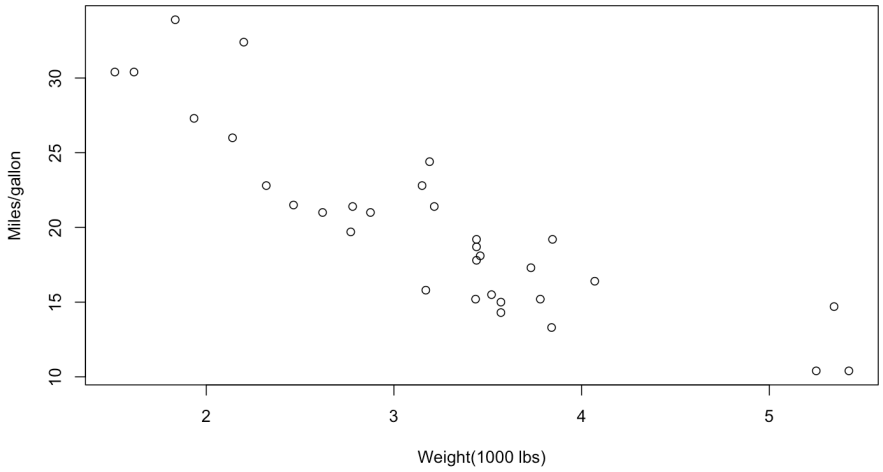
A data frame with 32 observations on 11 (numeric) variables.

1. mpg: Miles/(US) gallon
2. cyl: Number of cylinders
3. disp: Displacement (cu.in.)
4. hp: Gross horsepower
5. drat: Rear axle ratio
6. wt: Weight (1000 lbs)
7. qsec: 1/4 mile time
8. vs: Engine (0 = V-shaped, 1 = straight)
9. am: Transmission (0 = automatic, 1 = manual)
10. gear: Number of forward gears
11. carb: Number of carburetors

Events: click - Example 1/2

```
shinyApp( ui = basicPage(
    plotOutput("plotCars", click = "point_click",
               width = 650, height = 400),
    verbatimTextOutput("info")),

  server = function(input, output) {
    output$plotCars <- renderPlot({
      plot(mtcars$wt, mtcars$mpg,
      xlab="Weight(1000 lbs)", ylab="Miles/gallon")
    })

    output$info <- renderText({
      paste0("x=", input$point_click$x, "\ny=", input$poin
    })
  })
```

Events: click example 2/2




```
x=
y=
```


Events: click - contextual information with nearpoints()

- `nearPoints()` returns a dataframe containing the points near the click event (it works with brush, too)
  - the rows are sorted by distance to the event
  - by default it will return all data within 5 pixels of the mouse event
- For plots created with ggplot2, all necessary information is automatically derived from the plot specification (e.g., `xvar` and `yvar`), unless you use computed columns (e.g., `aes(meters\*1000, minutes\*60)`).

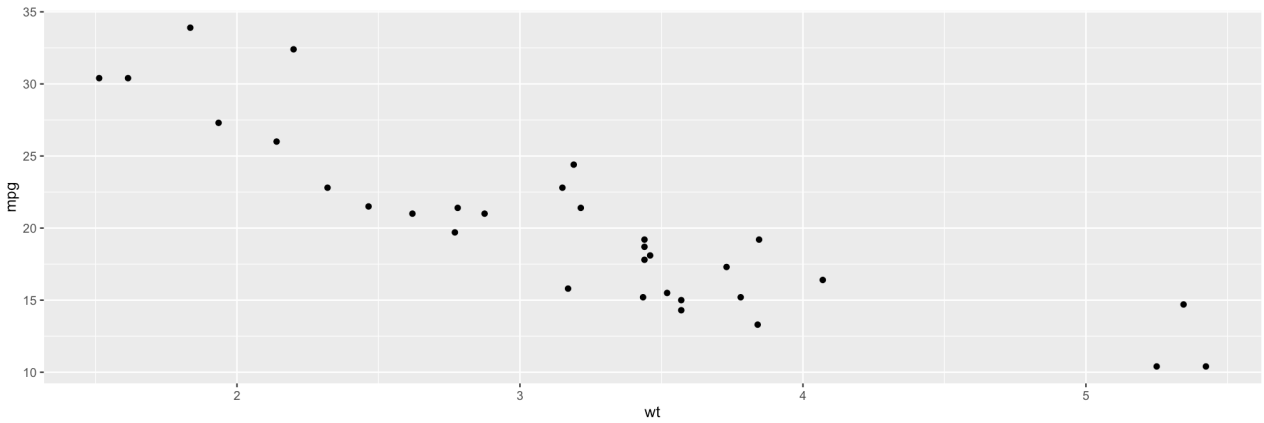
Additional settings for nearpoints()

- It might be useful to set additional arguments:
  - `allRows`: default set to FALSE; if TRUE, the input data frame will have a new column, `selected_`, which indicates whether the row was selected or not.
  - `threshold`: a maximum distance (in **pixels**) to the pointer location.
  - `maxpoints`: maximum number of rows to return.
  - `addDist`: if TRUE, add a column named `dist_` that contains the distance from the coordinate to the point (in **pixels**)

Example with nearpoints() 1/2

```
shinyApp(
  ui = fillPage(

         plotOutput("plot", click = "plot_click",
                    width = 900, height = 300),
         tableOutput("data")),

  server =function(input, output) {
              output$plot <- renderPlot({
                   ggplot(mtcars, aes(wt, mpg)) + geom_poi
            output$data <- renderTable({
              req(input$plot_click)
              nearPoints(mtcars, input$plot_click,
                         threshold = 20)})
  })
```

Example with nearpoints() 2/2



Events: dblClick

- `dblClick` behaves similarly to `click`
- Additional control through `dblclickOpts()`
- `dblclickOpts()`: all `clickOpts()` arguments + `delay`
    - `delay` is maximum time (in ms) between a pair clicks to be counted as a double-click

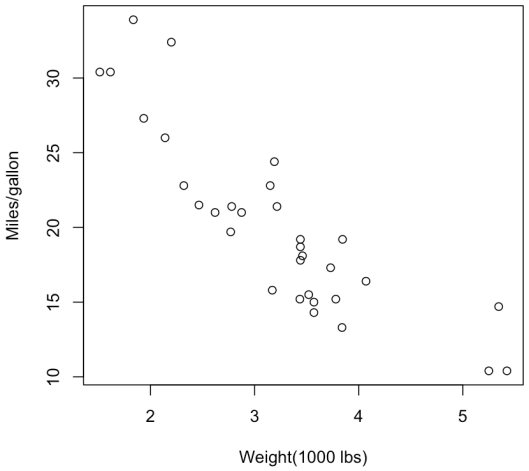
Events: hover

- Also `hover` behaves similarly to `click`
- `hoverOpts()` has more arguments
  - `delay`: how many ms before sending the mouse location to the server, according to `delayType`
  - `delayType`:
    - if “throttle”, one event every `delay` ms
    - if “debounce”, event after cursor still for `delay` ms
- `nullOutside`: whether set the value to NULL when the mouse exits the plotting area

Example with hover 1/2

```
shinyApp(
  ui = fluidPage(
    fluidRow(
      column(6,
        plotOutput("plotCars",hover=
            hoverOpts(id="h",delayType="debounce",delay=10
            width = 400, height = 400)) ,
    column(5, tags$h2("Hover mouse for 1 second to get the
    verbatimTextOutput("info")))),

  server = function(input, output) {
    output$plotCars <- renderPlot({
      plot(mtcars$wt, mtcars$mpg, xlab="Weight(1000 lbs)",

    output$info <- renderText({ paste0("x=", input$h$x, "\
```

Example with hover 2/2



Hover mouse for 1 second to get the point coordinates

x=
y=

Events: brushing

- Event to define a rectangular selection on the graph with the mouse
- It is managed with the `brush` argument and the `brushedPoints()` helper
- Behavior is very similar to `click` and `nearPoints()`

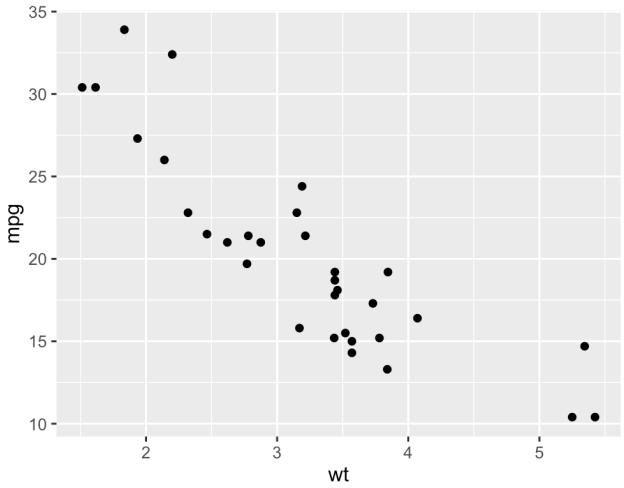
Brushing: customization

- With the `brushOpts()` function, it is possible to get further customization on the aestethics and the behavior, e.g.:
  - `fill`: the color of the brush
  - `stroke`: outline color of the brush
  - `opacity`: opacity of the brush
  - `direction`: if “xy”, the brush can be drawn and moved in both x and y directions. If “x”, or “y”, the brush will only work horizontally or vertically (useful for timeseries).
  - `resetOnNew`: whether to reset the brush when image is updated/replaced

Events: brushing - example 1/2

```
shinyApp(
  ui = fluidPage(plotOutput("plot",
           brush = brushOpts(id="sel", fill="orange", stro
                width = 450, height = 350),
  verbatimTextOutput("info")),

  server = function(input, output) {
     output$plot <- renderPlot({
       ggplot(mtcars, aes(wt, mpg)) + geom_point()}, res =

    output$info <- renderPrint({
       req(input$sel)
       brushedPoints(mtcars, input$sel) })
})
```

Events: brushing - example 2/2



A note on req() 1/2

- In the previous example, `req(input$sel)` avoids that `output$info` is fed with no data: it will proceed with following computations only if the user has supplied a value
- To satisfy `req()`, supplied values must be *truthy*, i.e. when they are not
  - FALSE
  - NULL
  - ""
  - an empty atomic vector
  - and a few other cases: see the documentation

A note on req() 2/2

- If values are *falsy* , `req()` throws a “silent” exception (neither logged nor displayed) that stops the execution of all downstream reactives and outputs, by putting them in an invalidation state
  - by setting `cancelOutput=FALSE`, instead of clearing the output, it will display the last valid value (if any).
- If values are *truthy*, the object is returned
  - in the example, `req(input$sel)` will return `input$sel`

Plot modification with reactiveVal

- `reactiveVal()` is similar to `reactive()`, with an important difference: it is possible to update the reactive value and all reactive consumers that refer to it
- Example:
  - `val <- reactiveVal(10)`: `val()` will return 10
  - `val(1000)`: `val()` will return 1000
  - `val(val() / 5)`: `val()` will return 200

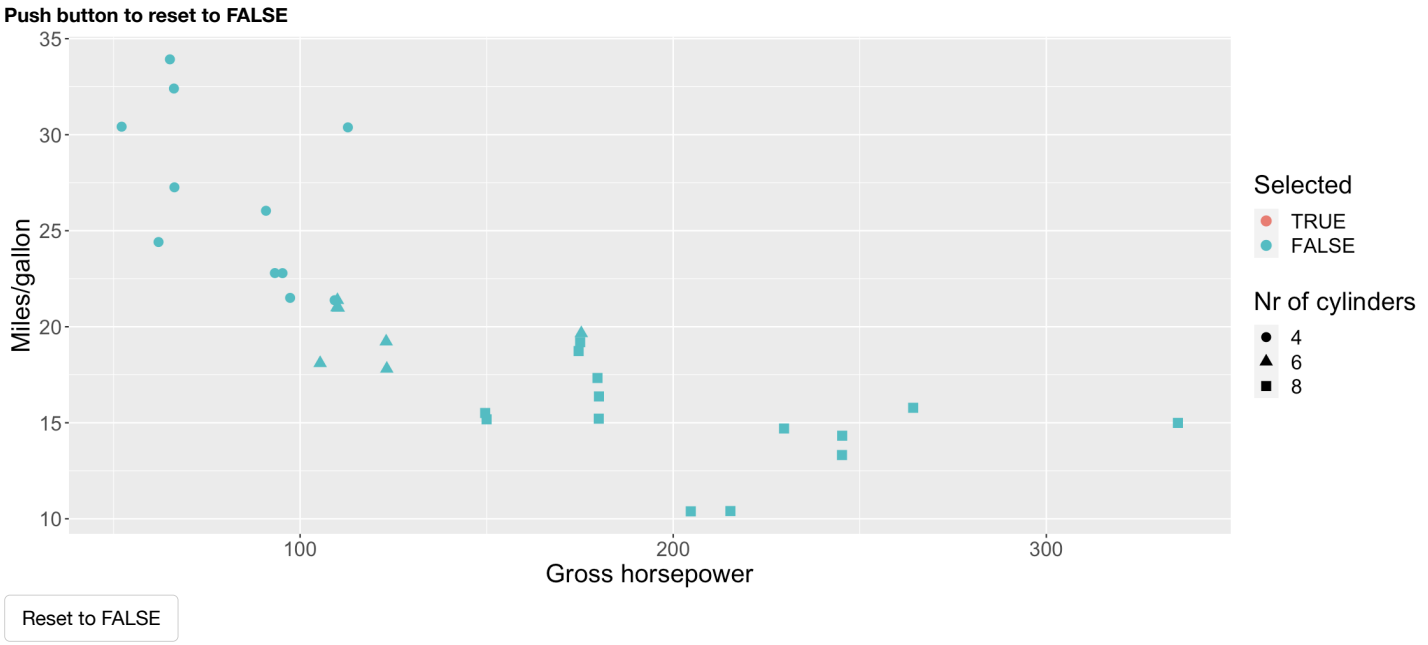
Plot modification with reactiveVal: example 1/3

```
.ui <- fluidPage(
        tags$strong("Push button to reset to FALSE"),
        plotOutput("plot",
                   brush = "point_brush",
                   hover="plot_hover"),
        actionButton(inputId = "b_reset",
                     label= "Reset to FALSE"),
        verbatimTextOutput("avg")
      )
```


Plot modification with reactiveVal: example 2/3

```
.server = function(input, output) {
 selected <- reactiveVal(rep(FALSE, nrow(mtcars)))
 observeEvent(input$point_brush, { brushed = brushedPoints(mtcars, inp
 selected(brushed | selected()) })

 observeEvent(input$b_reset, {selected(rep(FALSE, nrow(mtcars))) })
 output$plot <- renderPlot({mtcars$sel <- selected()
      ggplot(mtcars, aes(hp, mpg)) +  geom_point(aes(colour=sel, shape
      scale_colour_discrete(name="Selected", limits = c("TRUE", "FALSE
      scale_shape_discrete(name="Nr of cylinders") +
      xlab("Gross horsepower") + ylab("Miles/gallon") +
      theme(text = element_text(size=18))  })

   output$avg <- renderPrint( {
      req(input$point_brush)
      mtcars$sel <- selected()
      indexes <- mtcars$sel == TRUE
      paste0("Average consumption of selected: " ,
         round( sum(mtcars[indexes,]$mpg)/sum(indexes==TRUE) , 2) )
      } )}
```

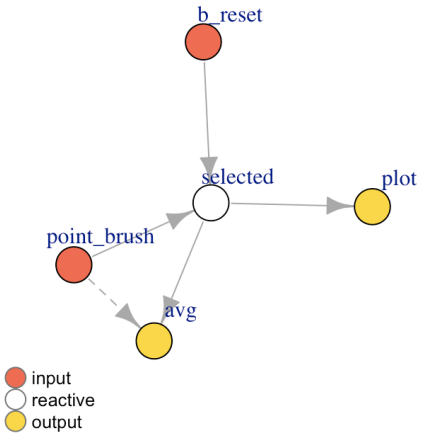
Plot modification with reactiveVal: example 2/2



Exercise

Draw the reactive graph of the previous application

Solution: reactive graph



ReactiveValues()

- `reactiveValues()` creates a list of reactive values

```
reactiveConsole(TRUE)
r <- reactiveValues(x = 10, y=20)
r$x
```

```
## [1] 10
```

```
r$y
```

```
## [1] 20
```

```
r$x <- (r$x + r$y)*2
r$x
```

```
## [1] 60
```

ReactiveValues() - important notes

- As for `reactiveVal()`:
  - when you read a value from it, the calling reactive expression takes a reactive dependency on that value;
  - when you write a value to it, it notifies any reactive functions that depend on that value.
- The `reactiveValues` object itself is not reactive.
- Unlike other R objects, modifying any copy of a `reactiveValues` object modifies all copies:

```
b1 <- b2 <- reactiveValues(times = 150) ;
b1$times <- b2$times * 2
b2$times
```

```
## [1] 300
```


Dynamic graph building

Higher levels of interactivity

- Shiny allows interactvity at a higher level, where ther user can:
    - set the properties of displayed graphs
    - change the data underneath the graphs
    - building her own UI and dashboard
        - the latter aspect will not be covered here

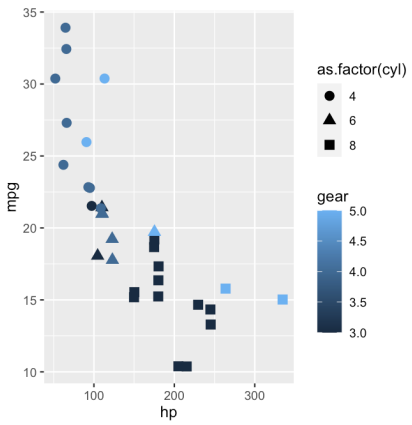
Dynamic plot size

- Plot size can be put in a reactive context **server side**
- Providing zero-argument functions to the width and height arguments of `renderPlot()` that return the desired size in pixels.
- As a result, the size of the plot becomes dynamic.

Dynamic plot size: example 1/2

```
shinyApp(

    ui = fluidPage(
      sidebarLayout(
        sidebarPanel(
          sliderInput("h", "height", min = 150, max = 600, value = 300
           sliderInput("w", "width", min = 150, max = 600, value = 300

        mainPanel(plotOutput("plot")))
    ),

    server = function(input, output) {
      output$plot <- renderPlot(
        width = function() input$w,
        height = function() input$h,
        { ggplot(mtcars, aes(hp, mpg)) +
          geom_point(aes(colour=gear,
            shape=as.factor(cyl)), size=3, position="jitter")} )
    }
)
```


Dynamic plot size: example 2/2



Data selection with tidy evaluation

- To work with `tidyverse` functions in Shiny, an *indirection* mechanism is needed to use data frame variables (used by `dplyr`) embedded in environment variables (that store the user-defined input)
- The type of *indirection* mechanism to be used depends on whether the function at hand makes a “data-masking” operation or a “tidy-selection” operation

Data masking

- Frequently used in ggplot2 in `aes()`
  - also in many dplyr functions, such as `filter()`, `mutate()`, etc.
- Indirection is made with `.data` or `.env` inside data-masking functions in combination with `[[` (instead of `$`) to dynamically identify the variables within a dataframe

```
min <- 15
var <- "mpg"
mtcars %>% filter(.data[[var]] < .env$min)
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs am gear c
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3
```

Data masking: example with filter 1/2

```
shinyApp(

  ui = fluidPage(
    selectInput("var", "Variable", choices = names(mtcars)
    numericInput("min", "Minimum value", value=6),
    tableOutput("output")
  ),

  server = function(input, output, session) {

    data <- reactive(mtcars %>%
                       filter(.data[[input$var]] > .env$input$min
    output$output <- renderTable(data())}
)
```


Data masking: example with filter 2/2

Variable

mpg

Minimum value

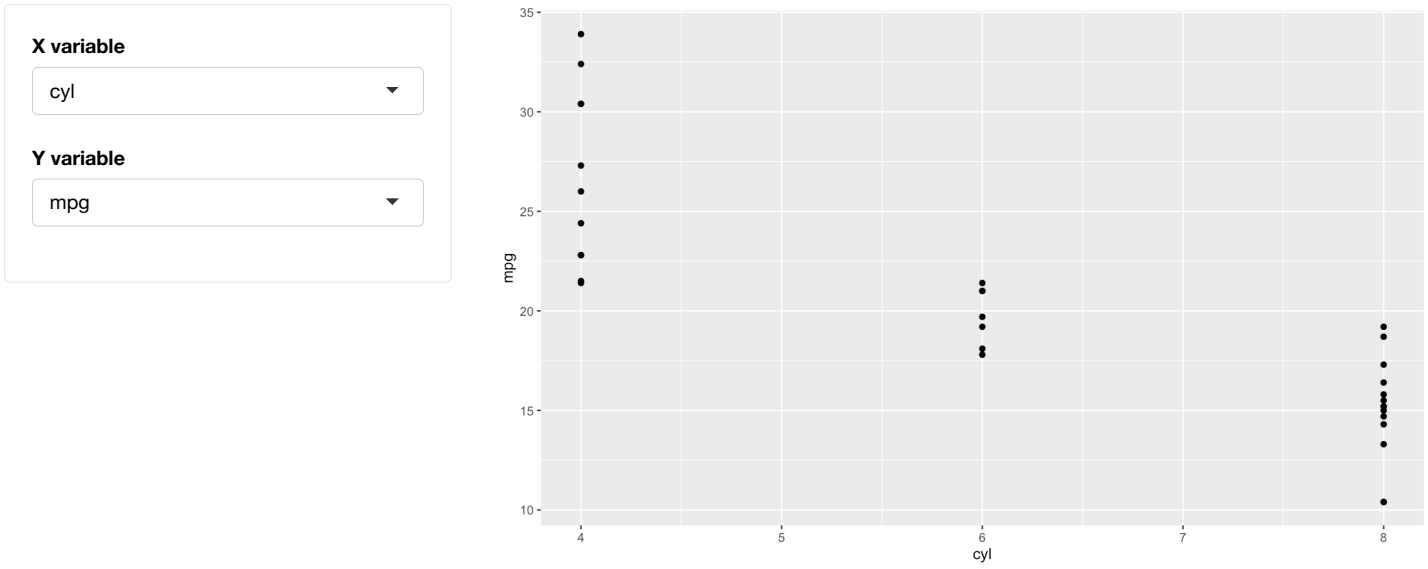
6

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.00	6.00	160.00	110.00	3.90	2.62	16.46	0.00	1.00	4.00	4.00
21.00	6.00	160.00	110.00	3.90	2.88	17.02	0.00	1.00	4.00	4.00
22.80	4.00	108.00	93.00	3.85	2.32	18.61	1.00	1.00	4.00	1.00
21.40	6.00	258.00	110.00	3.08	3.21	19.44	1.00	0.00	3.00	1.00
18.70	8.00	360.00	175.00	3.15	3.44	17.02	0.00	0.00	3.00	2.00
18.10	6.00	225.00	105.00	2.76	3.46	20.22	1.00	0.00	3.00	1.00
14.30	8.00	360.00	245.00	3.21	3.57	15.84	0.00	0.00	3.00	4.00
24.40	4.00	146.70	62.00	3.69	3.19	20.00	1.00	0.00	4.00	2.00
22.80	4.00	140.80	95.00	3.92	3.15	22.90	1.00	0.00	4.00	2.00
19.20	6.00	167.60	123.00	3.92	3.44	18.30	1.00	0.00	4.00	4.00
17.80	6.00	167.60	123.00	3.92	3.44	18.90	1.00	0.00	4.00	4.00

Data masking: example with ggplot 1/2

```
shinyApp(
  ui = fluidPage(
     sidebarLayout(
        sidebarPanel(
          selectInput("x", "X variable", choices = names(mtcars),selec
          selectInput("y", "Y variable", choices = names(mtcars),selec
        ),
        mainPanel( plotOutput("plot"))
     )
  ),

  server = function(input, output) {
    output$plot <- renderPlot({
        ggplot(mtcars) +
        geom_autopoint(aes(.data[[input$x]], .data[[input$y]]) )
    })
  }
)
```


Data masking: example with ggplot 2/2



Tidy selection

- Tidy-selection is used to quickly select columns by a variety of characteristics (e.g., name, type), for example in `dplyr::select()`
- Indirection obtained with `any_of()` or `all_of()`
  - both take as argument the data-variables names, in the form of environment variable
  - if a variable name does not exist in the input:
    - `all_of()` throws an error
    - `any_of()` ignore it

Tidy selection - example 1/2

```
shinyApp(
ui = navbarPage("Exploratory analysis",
          tabPanel("Select variables",
            selectInput(inputId="columns", label="Select",
                        choices = names(mtcars), multiple = TRUE),
            verbatimTextOutput("summary") ),

          tabPanel("See the data", plotOutput("plots"))
  ),

server = function(input, output) {

   selection <- reactive({
        req(input$columns)
        mtcars %>% select(all_of(input$columns))})

   output$summary <- renderPrint({summary(selection())})

   output$plots <- renderPlot({
     ggpairs(selection()) +
     theme_bw() })
})
```

Tidy selection - example 2/2

Interaction between graphs: an example (client code)

```
.ui =
  fluidPage(
    fluidRow(
      column(width = 8,
           h4("Select a sample with brush on the left graph and explor
              the impact of number of cylinders on consumption on the
      column(width = 4,
          actionButton(inputId = "reset", label= "Reset"))) ,

      fluidRow(
          column(width = 6,
            plotOutput("plot1", height = 300,
                       brush = brushOpts(id= "brush1",resetOnNew = TRU

            column(width = 6, plotOutput("plot2", height = 300)) ) #er
)
```

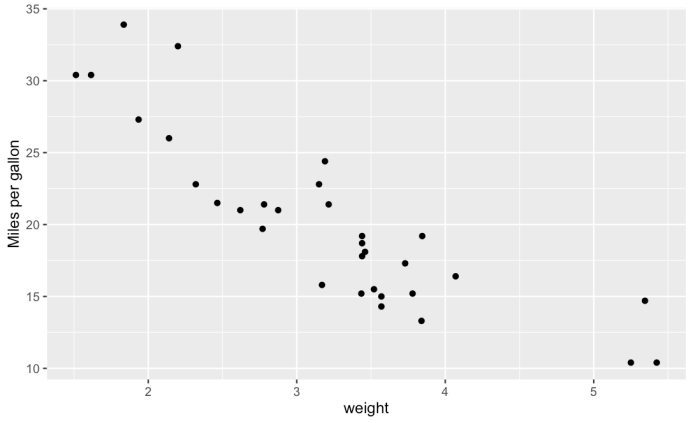
Interaction between graphs: an example (server code)

```
.server = function(input, output) {

    sel <- reactiveVal(NULL)
           observeEvent(input$brush1,{
             sel(brushedPoints(mtcars,input$brush1)) })

    observeEvent(input$reset, sel(NULL))

    output$plot1 <- renderPlot({
           ggplot(mtcars, aes(wt, mpg)) +
           geom_point()+
           xlab("weight") +
           ylab("Miles per gallon") })

    output$plot2 <- renderPlot({
           req(sel())
           ggplot(sel(), aes(x=as.factor(cyl), y=mpg)) +
           geom_boxplot(fill="lightsteelblue1", alpha=0.3) +
           xlab("Number of cylinders") +
           ylab("Miles per gallon") }) }
```


Interaction between graphs:app

Select a sample with brush on the left graph and explore the impact of number of cylinders on consumption on the right graph

Reset

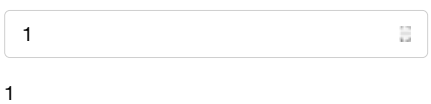


Data validation

Input validation with req()

- `req()` stops a reactive producer until required values are provided
- It is possible to combine `req()` with a logical statement:

```
shinyApp(
  ui = fluidPage( numericInput("num","Insert a number between 1 and 20
                        min=1, max = 20, value=1, step=1),
                  textOutput("echo")),

  server = function(input,output){
      n <- reactive({ a <- input$num
                      req(a<21 && a>0  )
                      a})

         output$echo <- renderText({print(n())}
    )}
)
```



1

User feedback

- The code of the previous example will not make the user aware of the error
- To provide visual feedback to the user, it is possible to rely on the shinyFeedback (https://github.com/merlinoa/shinyFeedback) package
  - `install.packages("shinyFeedback")`
  - `library("shinyFeedback")`

The shinyFeedabck package

- Several error messages available: `feedback()`, `feedbackWarning()`, `feedbackDanger()`, and `feedbackSuccess()`
  - They need to be called at *server side*
- Mandatory arguments:
  - `inputId` to bind the function to the proper input object
  - `show` to set whether or not to show the feedback.
  - `text` the message to display (aestethics customizable: see documentation).
- The function `useShinyFeedback()` should be added to the ui

Revised example with combined use of shinyFeedback and req() 1/2

```
shinyApp(

  ui = fluidPage(
    useShinyFeedback(),
    numericInput("num","Insert a number between 1 and 20",
                 min=1, max = 20, value=1, step=1),
    textOutput("echo")),

  server = function(input,output){
      n <- reactive({ a <- input$num
      ok <- a<21 && a>0
      feedbackWarning(inputId="num", show=!ok,
              text="Remember: number must be between 1 and 20")
      req(ok)
      a
  })
     output$echo <- renderText({ n() }  )}
)
```


Revised example with combined use of shinyFeedback and req() 2/2

1

Output validation with validate()

- Output validation is useful when an invalid state depends on multiple inputs
- It is possible to use `validate(message)` to put an error on the output:
  - it can be called within a reactive or an output
  - it takes any number of arguments, each being a condition to test
  - if any of the conditions represent failure, it signals an error and stops the execution of the following code, displaying the message in any downstream outputs
  - if the error is not handled by application-specific code, it is displayed to the user by Shiny

Example with validate() 1/2

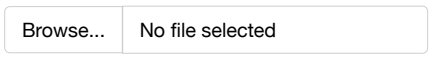
```
shinyApp(

  ui = fluidPage(
    numericInput("num1","Insert a positive number",
                 min=1, max = 20, value=1, step=1),
    numericInput("num2","Insert a positive number",
                 min=1, max = 20, value=1, step=1),
    textOutput("sum")),


  server = function(input,output){
    output$sum <- renderText({

      if( (input$num1< 0 || input$num2<0)){
            validate("You should put only numbers >0")}
      input$num1+input$num2 })
  }
)
```

Example with validate() 2/2

Example of output validation with file upload 1/2

```
shinyApp(

  ui = fluidPage(
    fileInput("file", NULL, accept = c(".csv")),
    dataTableOutput("data")
  ),

  server = function(input, output, session) {
    df <- reactive({
      req(input$file)
      ext <- tools::file_ext(input$file$name)
      if (ext != "csv")
        validate("Only .csv files are admitted ")

      read.csv(input$file$datapath, header = TRUE, sep=",") })

    output$data <- renderDataTable({ df() })
  }
)
```

Example of output validation with file upload 2/2



Useful resources

- Examples of interactive tools built on top of Shiny:
  - The ExPanDaR (https://joachim-gassen.github.io/ExPanDaR/) package
  - The exploratory (https://exploratory.io/) tool
- The dashboard (https://rstudio.github.io/shinydashboard/get_started.html) package to implement dashboards with dashboardPage()
- The book Interactive web-based data visualization with R, plotly, and shiny (https://plotly-r.com/)
- A list (https://www.bigbookofr.com/shiny.html) of other books on Shiny
- All the links provided at the end of Shiny basics presentation