Shiny Basics

Antonio Vetrò

Version 1.0 - April 2021

License

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License (http://creativecommons.org/licenses/by-sa/4.0/).

- You are free to:
  - Share - copy and redistribute the material in any medium or format
  - Adapt - remix, transform, and build upon the material

  for any purpose, even commercially.

  The licensor cannot revoke these freedoms as long as you follow the license terms.
- Under the following terms:
  - **Attribution** - You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **ShareAlike** - If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Table of contents

- Introduction
- User interface: input
- User interface: output
- Reactivity
- Layouts

Introduction

What is Shiny?

Shiny is a framework for creating interactive web applications using R code, by exposing your work in R via a web browser.

https://shiny.rstudio.com/ (https://shiny.rstudio.com/)

Enabling features

- A set of user interface functions to generate the HTML, CSS, and JavaScript code needed for common visualization tasks.
- Automatic track of the data-code dependencies (reactive programming).

Installation and loading

As for any other package in R:

- Installation: `install.packages("shiny")`
- Version check: `packageVersion("shiny")`
- Loading: `library(shiny)`

Key components for every Shiny app

1. The **data** to visualize
2. The **user interface** (UI), which defines the *view* of the app (layout and appearance)
3. The **server function** defines the *logic* of the app
4. **Reactive expressions** keep the view **consistent** with the data

Basic working mechanism

1. The UI is a web page connected to the Server
2. The server runs a live R session
3. When the users manipulate the UI:
   - the details are sent to the server
   - the server updates the presentation, by running the corresponding R code
   - the server sends back the updated presentation to the UI
4. Each connection to the Shiny app makes a unique and independent session

Shiny app template structure

1. Import Shiny: `library(shiny)`
2. User interface object: `ui <- fluidPage(){ }`
   - it assembles an HTML user interface
3. Server function: `server <- function(input, output, session) { }`
   - it defines how to build (and rebuild) the R objects defined in the UI
4. App composition: `shinyApp(ui=ui, server=server)`
   - The shinyApp function creates app objects given a pair UI/server

Create a new app on RStudio

- Click on File –> New Project,
  - then selecting New Directory and
  - *Shiny Web application*.
- A new directory and an **app.R** file is created
  - Note: each app will have its own directory

Hello World

```
shinyApp(
  ui <- fluidPage(
      "Hello, world!"
  ),

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

Hello, world!

Run the Hello World app

- In RStudio:
  - click the Run App button in the document toolbar
  - use a keyboard shortcut: `Cmd/Ctrl` + `Shift` + `Enter`
- In R console:
  - use `(source())` – mind the extra `()`! – on the whole document

Run the Hello World app

- From a script of a function:
  - call `shiny::runApp(<path to directory w/ app.R.>)`
  - you may use the argument `display.mode = "showcase"` to view an app in showcase mode, i.e. displaying your app alongside its code

```
runApp("MyApp", display.mode = "showcase")`
```

Reload the Hello World app

- In RStudio: through the reload app button in the toolbox
- In R console: `Cmd`/`Ctrl` + `Shift` + `Enter`

Note: a Shiny app blocks the R console while it is running, so you will not be able to run any R commands in the meanwhile.

Stop the Hello World app

- In Rstudio:
  - Use the stop button on the R console toolbar
  - Press `Esc` while in the console
- In R console:
  - Press `Ctrl` + `C`
  - Close the Shiny app window

The app directory structure

- The directory name corresponds to the name of the app
- `global.R` : optional, it defines objects available to both ui.R and server.R when they are in separate files
- `DESCRIPTION`: optional, it contains text to display in showcase mode
- `README` : optional, used in showcase mode
- `www`: optional, it is a directory containing files to share with web browsers (images, CSS, .js, etc.)
- *other files*: optional, data, scripts and other files needed for running the app

User interface: input

Input control

- Shiny provides a large collection of input functions to access the current value of an input object
- Input values are reactive
- Input objects have a unique id: ids connect the front-end with the back-end

  Constraints:

  1. unique name
  2. only letters, numbers and underscores (as any R variable)

Input object

- Input objects reflect what happens in the browser: for this reason, they are read-only
- The programmer specify inputs id in the UI

```
ui <- fluidPage( .... sliderInput("samples", ..) )
```

- The server function accesses it with `input$<inputId>`

```
server <- function(input, output) {
  ...
  dist <- rnorm(input$samples)
}
```

Input id: other parameters

- **label** : a human-readable label for the control.
- **value** : the default value
- other parameters are input type-specific

```
sliderInput("samples","Number of samples:",
                   min = 1, max = 100,value = 6))
```

It is possible to validate data at server side with `validate()`: it applies to output rendering function (see them later)

https://rdrr.io/cran/shiny/man/validate.html (https://rdrr.io/cran/shiny/man/validate.html)

Free text input

- `textInput()` : for short text
- `passwordInput()` for passwords
- `textAreaInput()` for paragraphs of text

Free text input: example

```
fluidPage(
     textInput("email", "Please type your email"),
     passwordInput("password", "Password"),
     textAreaInput("reminder", "Write any comment", rows =
)
```

Please type your email

Password

Write any comment

Numeric input

- `numericInput()` : a constrained text box
- `sliderInput()` : a slider with one or two ends

Numeric input: example

```
fluidPage(
  sliderInput("age", "Age", value = 30, min = 16, max = 75
  numericInput("num", "People", value= 2, min= 0, max= 10)
  sliderInput("salary", "Range", value = c(1000, 5000),
                                      min= 0, max= 25000
)
```

Age

16 30 75

16 22 28 34 40 46 52 58 64 70 75

People

2

Range

1,000 5,000 25,000

0 2,500 5,000 7,500 10,000 15,000 20,000 25,000

Dates input

- `dateInput()` it collects a single day
- `dateRangeInput()` it collects a range of two days

```
fluidPage(
  dateInput("job", "First job day"),
  dateRangeInput("studies", "Duration of master degree stu
)
```

First job day

2021-05-07

Duration of master degree studies

2021-05-07 to 2021-05-07

Limited choices: select

`selectInput()`: choice of one or multiple (`multiple = TRUE`) elements from a drop-down menu

```
judgements <- c("good","average","bad")
fluidPage(
    selectInput("score", "Evalute ", judgements,
                selected = "average")
)
```

Evalute

Limited choices: radio buttons

`radioButtons()` : choice(s) over a complete list

```
judgements <- c("good","average","bad")
judg.short <- c("1","0","-1")

fluidPage(
  radioButtons("score", "Evaluation",
               choiceNames=judgements,
               choiceValues=judg.short)
)
```

Evaluation
good
average
bad

Limited choices: checkboxes

- `checkboxInput()`: single checkbox for a single yes/no question
- `checkboxGroupInput()`: multiple choices over a list

Limited choices: checkboxes - example

```
judgements <- c("humanities","engineering","economics")
judg.short <- c("1","0","-1")

fluidPage(title = "Filter subscriptions",
  checkboxInput("typeS", "Academic", value = TRUE),
  checkboxGroupInput("freqS", "Degrees", judgements)
)
```

Academic

Degrees
humanities
engineering
economics

Other input types

- `fileInput()`: to upload a file; it requires special handling on the server side
- `actionButton()` and `actionLink()`: let the user perform an action; paired at server side with `observeEvent()` or `eventReactive()` (see later)

User interface: output

Outputs

- Three types of output:
  - text,
  - table,
  - plot
- Each **output** function on the front end is coupled with a **render** function in the back end
  - e.g., `plotOutput()` works with `renderPlot()` and viceversa

Outputs

- Output objects are accessed through the same ids defined in the input objects
- Output objects are atomic: they are either executed or not as a whole
- You get an error if you
  - forget the render function
  - attempt to read from an output

Output: text

- `textOutput()` is usually paired with `renderText()`
  - it puts text in <div> or <span>;
  - `renderText()` pastes text into a single string, like `cat()`.
- `verbatimTextOutput()` is usually paired with `renderPrint()`
  - it puts fixed-width text in a <pre>.
  - `renderPrint()` prints the result as if you were in the console, like `print()`;

Output: text - example

```
shinyApp(
ui <-  fluidPage(
    textOutput("text"),
    verbatimTextOutput("code_output")),

server <- function(input, output) {
    output$text <- renderText("Summary")
    output$code_output <- renderPrint(summary(1:10)) })
```

Summary

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   1.00    3.25    5.50    5.50    7.75   10.00 
```

Output: tables

- `tableOutput()` is paired with `renderTable()` to show a static table of data
  - it shows all the data at once
  - creates a simple HTML table
- `dataTableOutput()` is paired with `renderDataTable()` to show a dynamic table
  - user interaction allows to decide which rows are visible
  - uses the *DataTables* Javascript library to create the interactive table

Output: tables - example with tableOutput

```
shinyApp(
ui <- fluidPage( tableOutput("static")),

server <- function(input, output) {
  output$static <- renderTable(head(mtcars)) }
)
```

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

Table example w/dataTableOutput

```
shinyApp(
ui <- fluidPage(dataTableOutput("dynamic")),
server <- function(input, output) {
  output$dynamic <- renderDataTable(mtcars,
                          options = list(pageLength = 4))})
```

Show 10 entries Search:

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21	6	160	110	3.9	2.62	16.46	0	1	4	4
21	6	160	110	3.9	2.875	17.02	0	1	4	4
22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb

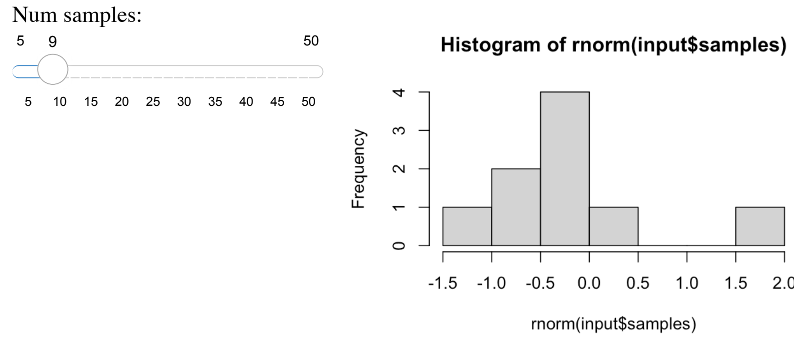
Showing 1 to 4 of 32 entries Previous 1 2 3 4 5 … 8 Next

Output: Plots

- `plotOutput()` is paired with `renderPlot()` to show any type of R graphic (base, ggplot2, etc)
- Plots can also act as inputs to enable interactivity: we'll see this later

Output: Plots - example

```
shinyApp(
ui <- flowLayout(
  sliderInput("samples", "Num samples:",
                         min= 5, max= 50, value= 9),
  plotOutput("distPlot",  width = 350, height = 250) ),
server <- function(input,output){
  output$distPlot <- renderPlot({hist(rnorm(input$samples)
```



Outputs: more

- `imageOutput()` paired with `renderImage()`
- `htmlOutput()` and `uiOutput()` paired with `renderUI()`

Wrap up exercise

- Create your own app by copying the code of the histogram app in a new directory
- Modify visual elements and input parameters to get used to the commands
- Customize the histogram with ggplot2
- Try to add a table with the summary of the distribution: what happens?

Reactivity

Reactive programming: premises

- Key idea: specify a graph of dependencies to automatically update outputs when input change
- Mostly within the server function, where input is read and used to generate the output
- Input can be read (i.e. it produces reactive values) only if placed within a **reactive context** that automatically tracks what inputs the output uses
- A reactive context is created by reactive functions (`render()`, `reactive()`, etc)

Reactive programming as declarative programming

- Reactive programming is a form of declarative programming:
  - the developer defines the application behavioral requirements, norms and goals
  - this is the style of programming in Shiny
- It is antithetic to traditional programming, a.k.a. imperative programming:
  - the developer issues commands to get immediately a result
  - this is the style of programming in R scripts

Simple example of reactivity

```
shinyApp(
  ui <- fluidPage(
    numericInput("age", "What's your age?", value = 15),
    textOutput("sentence")
  ),
  server <- function(input, output) {
      output$sentence <- renderText({
        paste0("You are ", input$age, " years old")
      })
  }
)
```

What's your age?

15

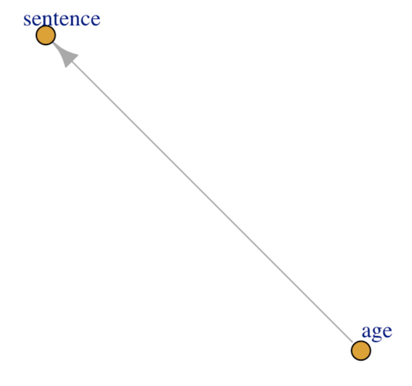
You are 15 years old

Reactive graph

- It describes which inputs are connected to which outputs
- In the former example, **sentence** has a reactive dependency on **age**
    - this means that **sentence** has to be recomputed every time **age** changes
- Execution order is defined by the reactive graph, not by the order of lines

Reactive graph

- Package `reactlog` can help building the reactive graph to keep track of connections



Reactive expressions

- Reactive expressions take inputs and produce outputs
- They combine features of both inputs and outputs:
  - they automatically cache results, and only update when their inputs change
  - the results of a reactive expression can be used in an output
- In the reactive graph, they stay between input and output
  - The term **producers** refer to reactive inputs and expressions
  - The term **consumers** to refer to reactive expressions and outputs

Reactive expressions: example

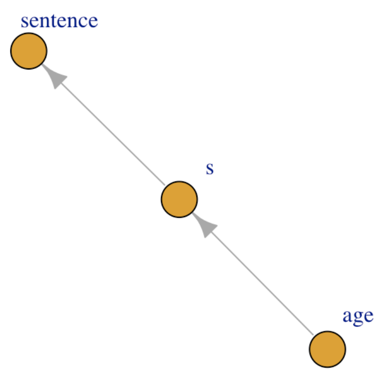
```
shinyApp(
    ui <- fluidPage(
      numericInput("age", "What's your age?", value = 15),
      textOutput("sentence")
    ),

  server <- function(input, output) {
    s <- reactive( paste0("You are ", input$age, " years o
    output$sentence <- renderText(s())
  })
```

What's your age?

15

You are 15 years old

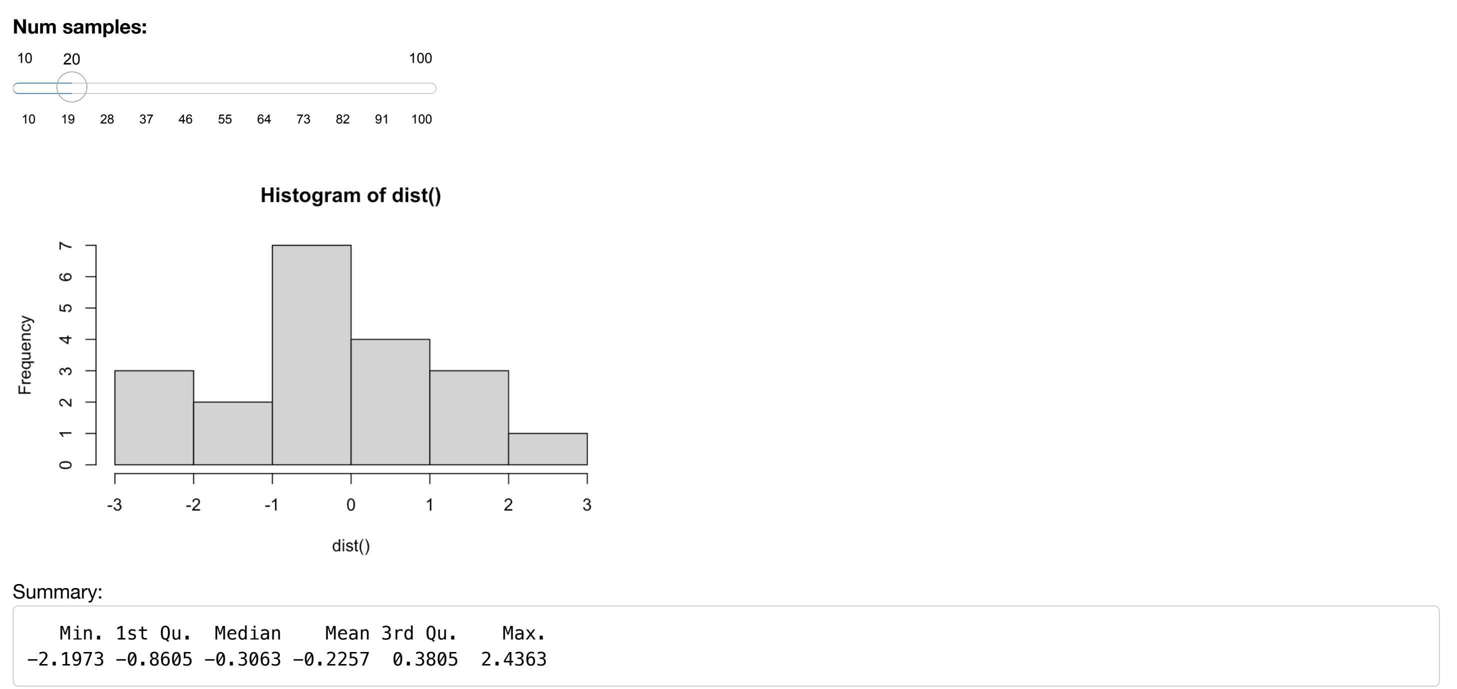
Reactive graph of the example



Histogram exercise revisited 1/2

```
shinyApp(
  ui <- fluidPage(
      sliderInput("samples", "Num samples:",
                              min= 10, max= 100, value= 20
      plotOutput("distPlot", width = 450, height = 300),
      textOutput("summary_text"),
      verbatimTextOutput("summary_data")
  ),
  server <- function(input, output) {
      dist <- reactive( rnorm(input$samples) ) ## reactive

      output$distPlot <- renderPlot({ hist(dist())})
      output$summary_text <- renderText("Summary:")
      output$summary_data <- renderPrint(summary(dist()))
  })
```

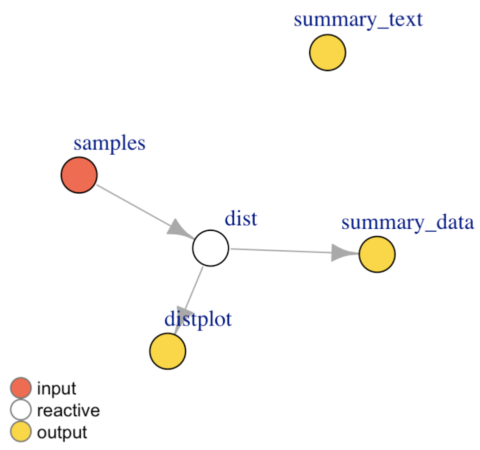

Histogram exercise revisited 2/2



Exercise

Draw the reactive graph of the histogram app

Solution



Controlling timing of evaluation: timed evaluation

- `reactiveTimer()` is triggered by the passage of time

```
timer <- reactiveTimer(1000)
shinyApp(
  ui <- fluidPage(verbatimTextOutput("time")),
  server <- function(input, output) {
    output$time <- renderText({
          timer()
          format(Sys.time(), "%a %b %d %X %Y")
})  })
```

```
Fri May 07 14:36:40 2021
```

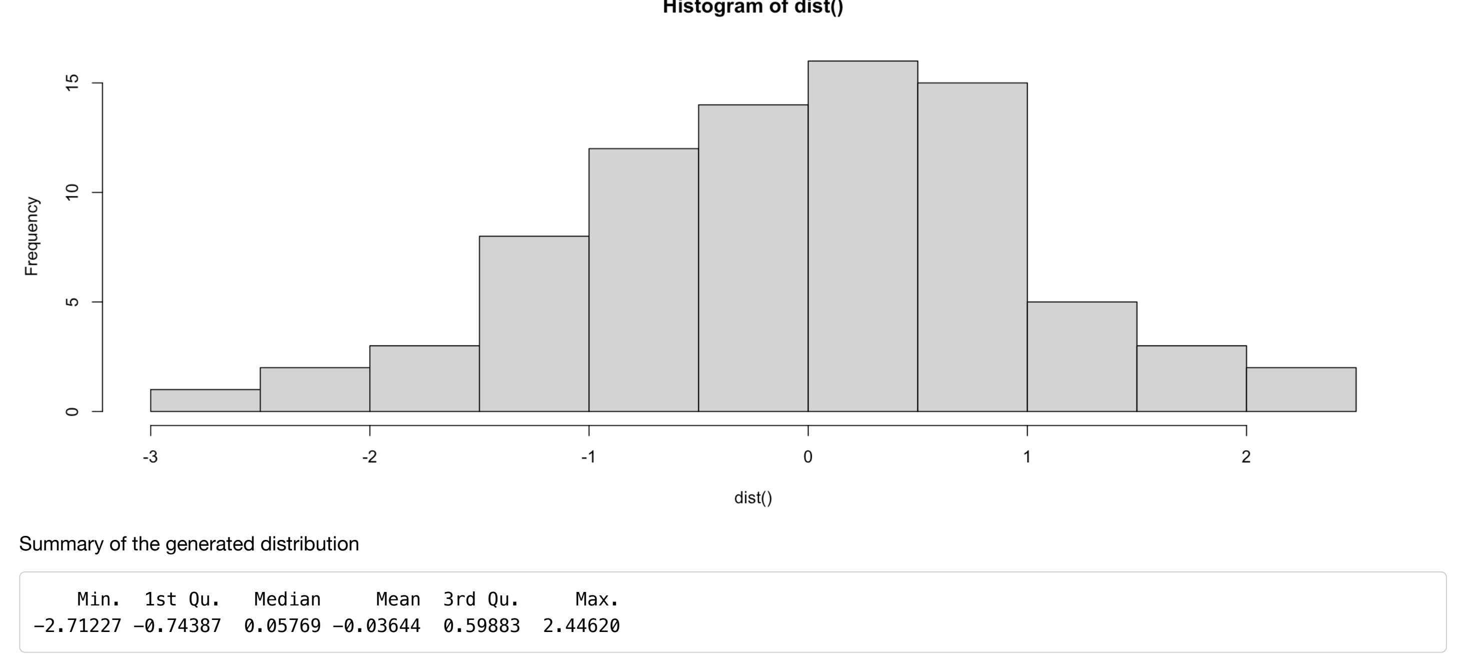

Another sample for the usage of timer() 1/2

Compute distributions at regular intervals

```
timer <- reactiveTimer(2000)
shinyApp(
  ui <- fluidPage(
      plotOutput("distPlot"),
      tags$p("Summary of the generated distribution"),
      verbatimTextOutput("summary_data")),
  server <- function(input, output) {
      dist <- reactive({
            timer(); rnorm( sample.int(n=100,size = 1) ) }
      output$distPlot <- renderPlot({ hist( dist())})
      output$summary_data <- renderPrint(summary(dist()))}
)
```

Another sample for the usage of timer() 2/2

Compute distributions at regular intervals

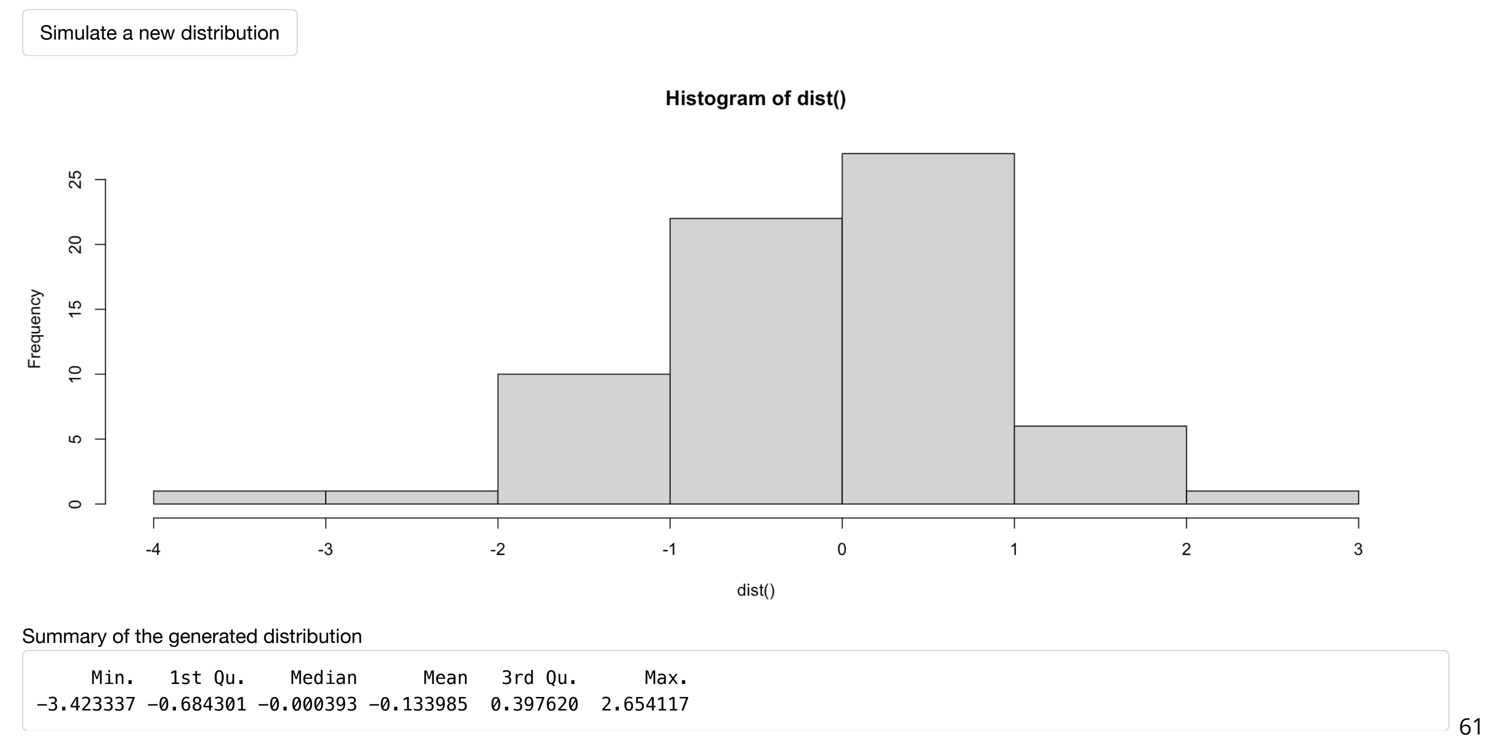


Controlling timing with actionButton

- To avoid useless computations server side with timer()
- Action button adds a new dependency

```
shinyApp(
  ui <- fluidPage(
    actionButton("simulate", "Simulate a new distribution"
    plotOutput("distPlot"),
    "Summary of the generated distribution",
    verbatimTextOutput("summary_data") ),
  server <- function(input, output) {
    dist <- reactive({
          input$simulate; rnorm( sample.int(n=100,size = 1
    output$distPlot <- renderPlot({ hist( dist()) })
    output$summary_data <- renderPrint(summary(dist()))}
)
```

Controlling timing of evaluation: actionButton 2/2



Controlling timing of evaluation: eventReactive()

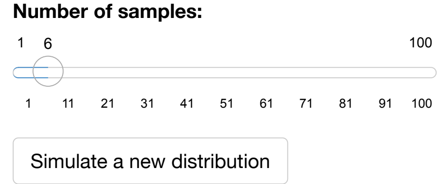
- It makes possible to separate the dependencies from the values used to compute the result
    - As a result, it creates a calculated value that only updates in response to an event
- Usage:
    - the first argument specifies what to take a dependency on
    - the second argument specifies what to compute

Histogram app with eventReactive() 1/2

The histogram is updated only when the button is clicked

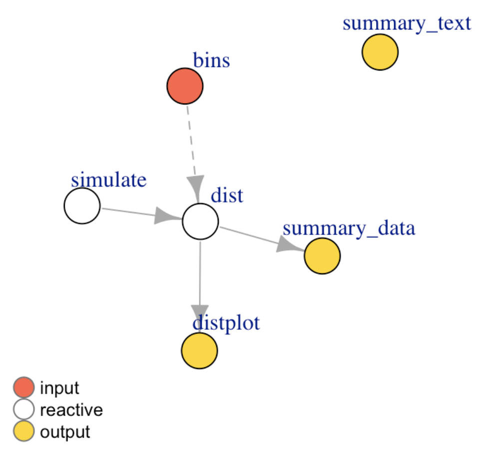
```
shinyApp(
  ui <- fluidPage(
      sliderInput("samples", "Number of samples:",
                             min = 1,max = 100, value = 6)
      actionButton("simulate", "Simulate a new distributio
      plotOutput("distPlot", width = 600, height = 300),
      "Summary of the generated distribution",
      verbatimTextOutput("summary_data")
  ),
  server <- function(input, output) {
      dist <- eventReactive(input$simulate, { rnorm(input$
      output$distPlot <- renderPlot({ hist(dist()) })
      output$summary_data <- renderPrint(summary(dist()))
  }
)
```


Histogram app with eventReactive() 2/2



Summary of the generated distribution

Reactive graph for the modified histogram app



Controlling timing of evaluation: observer

- Needed for actions whose reach and/or side effects are outside the app. Examples:
  - debug messages on console
  - sending data to a web API
  - updating a database
  - …

Controlling timing of evaluation: observer

- Multiple ways of creating an observer: here we focus only on `observeEvent()`:
  - it responds to reactive inputs, values, and expressions
  - the first argument (`eventExpr`) is the input or expression to take a dependency on, as simple as `input$click` or a complex expression
  - the second argument (`handlerExpr`) is the code that will be run (it should be a side-effect-producing action, since the return value is ignored)

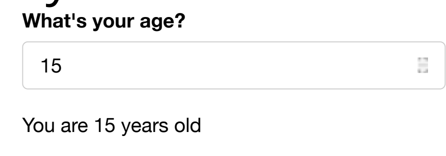
ObserverEvent example with the age app

Try it and check the console

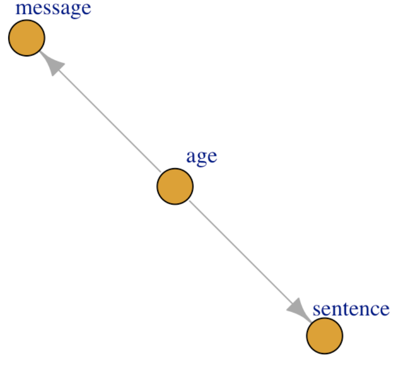
```
shinyApp(
  ui <- fluidPage(
      numericInput("age", "What's your age?", value = 15),
      textOutput("sentence") ) ,

  server <- function(input, output) {
      s <- reactive( paste0("You are ", input$age, " years
      output$sentence <- renderText(s())
      # Added observeEvent
      observeEvent(input$age, { message("Age inserted") })
  }
)
```

ObserverEvent example with the age app

Try it and check the console



Reactive graph for the revisited age app



Guidelines for observeEvent() and eventReactive()

- Use `observeEvent()` whenever you want to perform an action in response to an event
  - a value recalculation is not an action: use `eventReactive()` for that
  - the result of `observeEvent()` is never assigned to a variable
  - it is not possible to refer to the result of `observeEvent()` from other reactive consumers
  - as a consequence, in terms of reactive graph, it is located on the outputs’ side

Guidelines for observeEvent() and eventReactive()

- Use `eventReactive()` to create a calculated value that only updates in response to an event
  - remind that it will ignore all invalidations coming from its reactive dependencies

Layouts

Pages

Page functions set up the HTML, CSS, and JavaScript needed by Shiny

- `fluidPage()` layout is composed of rows. Each row include 12 columns
  - components fill all available browser width in realtime
  - it is possible to use in combination to higher-level layout functions

Pages

- `fixedPage()` works like `fluidPage()` but has a fixed maximum width
  - 940 pixels on a typical display,
  - 724px on smaller displays,
  - 1170px larger ones
  - all layout must be done with `fixedRow()` and `column()`

Pages

- `fillPage()` fills the full height of the browser
  - useful to get the whole screen occupied by a plot
  - if the page content's height is smaller than the browser window, white space is left at the bottom
  - if the page content's height is larger than the window, it is scrolled
  - N.B.: use functions that are designed for fill layouts to avoid unexpected appearance

Layouts funtions within fluidPage()

- Layout functions allow to visually organize the elements of the app
- Layout embeds a hierarchy of function calls that matches the hierarchy in the generated HTML
- At the lowest level of the hierarchy are panel functions and single visual elements

Example structure

```
fluidPage(
  titlePanel(
    # app title/description
  ),
  sidebarLayout(
    sidebarPanel(
      # inputs
    ),
    mainPanel(
      # outputs
    )
  )
)
```

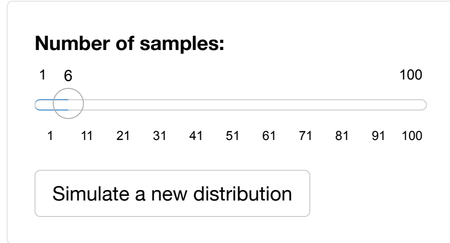
The histogram app with layout 1/2

```
.ui <- fluidPage(
    titlePanel("Normal distribution generator"),
    sidebarLayout(
      sidebarPanel(
        sliderInput("samples", "Number of samples:",
                        min = 1,max = 100, value = 6),
        actionButton("simulate", "Simulate a new distribut
      ),
      mainPanel(
        plotOutput("distPlot", height=300, width=420),
        "Summary of the generated distribution",
        verbatimTextOutput("summary_data")
      )
    )
   )
```


The histogram app with layout 1/2

```
shinyApp(ui = .ui,
  server = function(input, output) {
        dist <- eventReactive(input$simulate,
                              { rnorm(input$samples)})
        output$distPlot <- renderPlot({ hist(dist()) })
        output$summary_data <- renderPrint(summary(dist())
  }
)
```

The histogram app with layout 2/2



Summary of the generated distribution

Flexible layout

- With `fluidRow()` and `column()` it is possible to create a flexible multi-row layout within `fluidPage()`
  - the first argument of `column()` gives how many of the row units to occupy (up to 12 available).

Flexible layout

```
fluidPage(
  fluidRow(
    column(3,
      ...
    ),
    column(9,
      ...
    )
  ),
  fluidRow(
    column(4,
      ...
    ),
    column(6,
      ...
    ),
    column(2,
      ...
    )
  )
)
```


Exercise

Re-design the histogram app in such a way that:

- the plot is at the bottom of the page, centered
- the summary is next to the slider

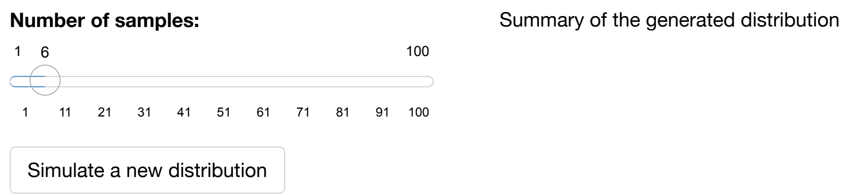
Solution 1/2

```
.ui <- fluidPage(
    titlePanel("Normal distribution generator"),
    fluidRow(     #First row contains slider and summary d
      column(4,
          sliderInput("samples", "Number of samples:",
                                  min=1,max =100, value = 6),
          actionButton("simulate", "Simulate a new distrib
       column(8,
           "Summary of the generated distribution" ,
           verbatimTextOutput("summary_data") ) ),
     fluidRow(  #second row contains only the plot
       column(12, align="center", plotOutput("distPlot",
                                             height=300,wi
   ) )
```

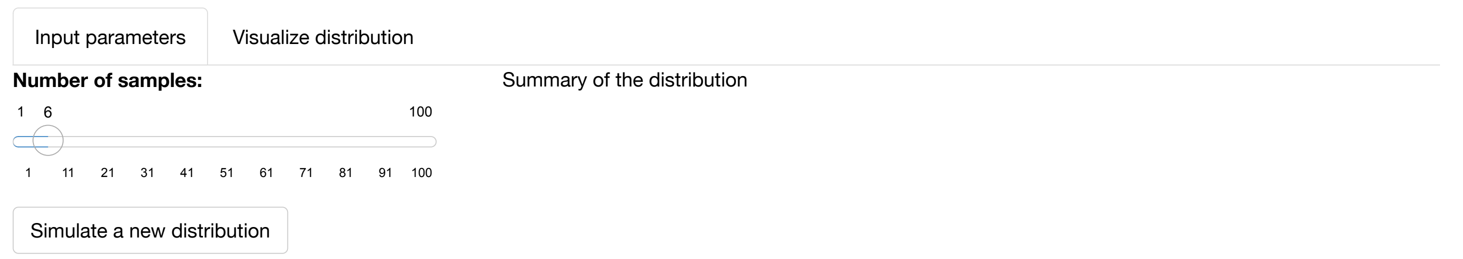

Solution 1/2

```
shinyApp(ui = .ui,
  server = function(input, output) {
        dist <- eventReactive(input$simulate,
                              { rnorm(input$samples)})
        output$distPlot <- renderPlot({ hist(dist()) })
        output$summary_data <- renderPrint(summary(dist())
)
```

Solution 2/2



Histogram app with tab panels 2/2



Tab panles

```
shinyApp(
  ui = fluidPage(
      tabsetPanel(  #set up tabset
         # first tab is for input parameters and for summary data
         tabPanel("Input parameters",
            fluidRow(
              column(4,sliderInput("samples", "Number of samples:",
                                   min=1,max =100, value = 6),
                     actionButton("simulate", "Simulate a new distrib
              column(8,"Summary of the distribution",
                     verbatimTextOutput("summary_data"))
             ), #end fluidRow
          ), #end first tab
         # second panel is for visualizing the histogram
         tabPanel("Visualize distribution", plotOutput("distPlot"))
      ) #end tabsetPanel
  ),  #end ui page
  server = function(input, output) {
      dist <- eventReactive(input$simulate, { rnorm(input$samples)})
      output$distPlot <- renderPlot({ hist(dist()) })
      output$summary_data <- renderPrint(summary(dist()))
  }
)
```


Navlists and navbars

- The function `navlistPanel()` shows tabs titles vertically in a sidebar.
- Tabs can also be used within a `navbarPage()`
    - it is a page that contains a top level navigation bar
    - it runs the tab titles horizontally
    - a navigation bar can be used to navigate through a set of tabPanel() elements.
- The `navbarMenu()` function creates an embedded menu within the navbar page

Histrogram app with navbarPage() 1/2

```
shinyApp(
  ui <- navbarPage(title="Normal distribution navigator",
      tabPanel("Input parameters",
        fluidRow(
          column(4, sliderInput("samples", "Number of samples:",
                                 min=1,max =100,value = 6),
                  actionButton("simulate", "Simulate a new distributic
          column(8, "Summary of the distribution",
                      verbatimTextOutput("summary_data") ), ),
        ),
      navbarMenu("More",
          tabPanel("Visualize distribution", plotOutput("distPlot") ),
          tabPanel("See the data",verbatimTextOutput("generatedData")
        )
      ), #end navbarPage
  server <- function(input, output) {
          dist <- eventReactive(input$simulate, { rnorm(input$samples)
          output$distPlot <- renderPlot({ hist(dist()) })
          output$summary_data <- renderPrint(summary(dist()))
          output$generatedData <- renderPrint(dist()) }
)
```


Histrogram app with navbarPage() 2/2

Number of samples:

1 6 100

1 11 21 31 41 51 61 71 81 91 100

Simulate a new distribution

Summary of the distribution

Modifying the page with HTML

- It is possible to add static HTML elements with tags:
  - e.g. `tags$form`, `tags$span`, `tags$caption`
  - unnamed arguments are passed into the tag; named arguments become tag attributes
- Shiny provides wrappers for the most common HTML elements
  - e.g. , `a(href="", "link")`, `h1("Title abcdef")`
- Raw code can be inserted, too: -

```
HTML("<h1> Data curation process </h1> <p> Here we describe how data curation has been performed<p>")
```


Become familiar with html tags

Use the HTML tags to add text within the tabs of the histogram app

Useful resources

- Layout guidelines and themes:
  - https://shiny.rstudio.com/articles/layout-guide.html (https://shiny.rstudio.com/articles/layout-guide.html)
  - https://mastering-shiny.org/action-layout.html#bootstrap (https://mastering-shiny.org/action-layout.html#bootstrap)
- Debugging Shiny applications:
  - https://mastering-shiny.org/action-workflow.html#debugging (https://mastering-shiny.org/action-workflow.html#debugging)
- Extension packages:
  - https://github.com/nanxstats/awesome-shiny-extensions (https://github.com/nanxstats/awesome-shiny-extensions)
  - https://shiny.rstudio.com/gallery/ (https://shiny.rstudio.com/gallery/)
- Learning more on Shiny
  - https://shiny.rstudio.com/tutorial/ (https://shiny.rstudio.com/tutorial/)
  - https://shiny.rstudio.com/articles/ (https://shiny.rstudio.com/articles/)
  - https://mastering-shiny.org/ (https://mastering-shiny.org/)