L07 – Shiny

24. Mai 2021

Contents

Intro

Shiny ist ein Framework zur Erstellung interaktiver Web-Apps mit R. Hier lernen wir die Basics von Shiny. Ausführlichere Informationen findet man

- in den entsprechenden Funktionsdokumentationen
- in Wickham (2021) - Mastering Shiny: https://mastering-shiny.org/
- und unter https://shiny.rstudio.com/.
- Siehe auch Shiny Cheat Sheet https://github.com/rstudio/cheatsheets/raw/master/shiny.pdf

Hello World

Eine minimale Shiny-App kann folgendermaßen erstellt werden.

```
library(shiny)
ui <- fluidPage("Hello, world!")
server <- function(input, output) {}
app <- shinyApp(ui, server)
```

Eine Shiny-App besteht aus einem User-Interface (UI, angezeigte Elemente) und einer Server-Funktion (beschreibt Berechnungen). Der Aufruf von `shinyApp()` erzeugt aus den beiden Komponenten ein `shiny.appobj`-Objekt.

```
class(app)
## [1] "shiny.appobj"
```

Durch (ggf implizites) `print()` wird die App ausgeführt.

```
app # implicit print()
```

In der Konsole erscheint `Listening on http://127.0.0.1:XYZ`. Die URL `http://127.0.0.1:XYZ` (mit dem entsprechenden Port statt `XYZ`) kann in einem Browser geöffnet werden, da der Aufruf von `print()` die App als Webpage (mit HTML, CSS, JavaScript) verfügbar macht.

Falls die App mit einem Fehler abbricht, muss sie ggf explizit beendet werden. Drücke dazu das rote Stop-Symbol in der Konsole rechts oben.

UI

Wir erzeugen das UI mit `fluidPage()`. Die Funktion nimmt eine beliebige Anzahl an Argumenten. Die Argumente sind die Elemente des UI. Die UI-Elemente gliedern sich in Inputs, Outputs und nicht-interaktive Elemente.

Inputs und Outputs werden mit entsprechenden Funktionen erzeugt. Die Ausgaben dieser Funktionen und von `fluidPage()` sind im Wesentlichen der HTML-Code, der das UI als Webpage beschreibt.

```
library(shiny)

numIn <- numericInput("idOfInput", "A Number", value=42) # an input
c(typeof(numIn), class(numIn))
## [1] "list"      "shiny.tag"
print(numIn) # HTML-Code
## <div class="form-group shiny-input-container">
##   <label class="control-label" id="idOfInput-label" for="idOfInput">A Number</label>
##   <input id="idOfInput" type="number" class="form-control" value="42"/>
## </div>

ui <- fluidPage(
  numIn,
  textOutput("idOfOutput"), # an Output
  "<b>nicht fett</b>", # plain text, text '<b>' will be shown
  HTML("<b>fett</b>") # HTML, <b> marks text as bold
)
c(typeof(ui), class(ui))
## [1] "list"           "shiny.tag.list" "list"
print(ui) # HTML-Code
## <div class="container-fluid">
##   <div class="form-group shiny-input-container">
##     <label class="control-label" id="idOfInput-label" for="idOfInput">A Number</label>
##     <input id="idOfInput" type="number" class="form-control" value="42"/>
##   </div>
##   <div id="idOfOutput" class="shiny-text-output"></div>
##   &lt;b&gt;nicht fett&lt;/b&gt;
##   <b>fett</b>
## </div>
```

Um das so definierte UI anzuzeigen, führe folgendes aus.

```
shinyApp(ui, function(input, output){})
```

Inputs

Die ersten beiden Argumente der Inputs sind `inputId` und `label`. `inputId` ist ein String. Dieser wird in der Server-Funktion genutzt, um auf den Wert des UI-Input-Elementes zuzugreifen. Das `label` wird im UI als Beschriftung des UI-Elementes angezeigt. Inputs sind **reaktiv**: Sobald ein Input verändert wird, werden die Outputs neu gezeichnet.

Führe folgenden Code aus, um die verschiedenen Inputs kennenzulernen. Beachte, dass Veränderungen in den Inputs unten angezeigt werden.

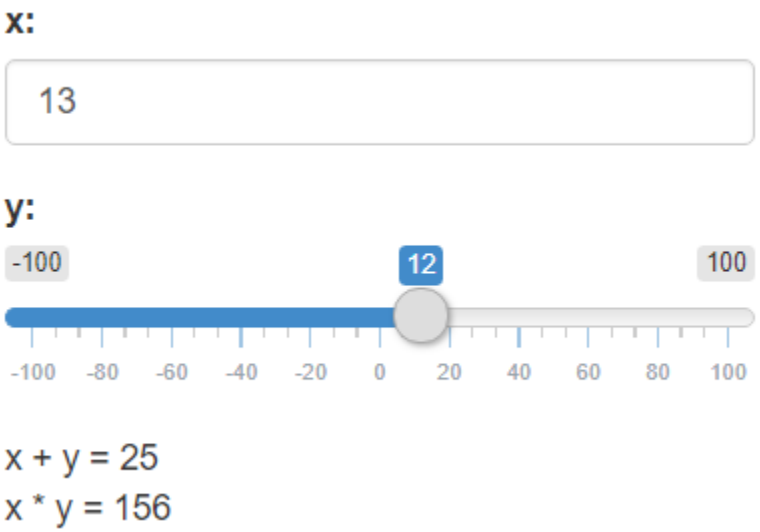
```
library(shiny)
ui <- fluidPage(
  actionButton("a", "Button"),
  checkboxInput("b", "CheckBox", value=TRUE),
  radioButtons("c", "RadioButtons", choices=LETTERS[1:3]),
  numericInput("d", "A Number", value=42),
  selectInput("e", "DropDown", choices=LETTERS[1:3]),
  textInput("f", "Insert Joke"),
  sliderInput("g", "sliding", min=0, max=100, value=50),
  textOutput("z"),
  textOutput("y"),
  textOutput("x"),
  textOutput("w"),
  textOutput("v"),
  textOutput("u"),
  textOutput("t")
)
server <- function(input, output) {
  output$z <- renderText(input$a)
  output$y <- renderText(input$b)
  output$x <- renderText(input$c)
  output$w <- renderText(input$d)
  output$v <- renderText(input$e)
  output$u <- renderText(input$f)
  output$t <- renderText(input$g)
}
shinyApp(ui, server)
```

Aufgabe 1:

Erstelle eine Shiny-App, die 2 verschiedene Inputs für Zahlen hat und sowohl Summe als auch Produkt der Zahlen jeweils in einem `textOutput(id)` mittels `renderText(string)` anzeigt.

Hinweis: Die Werte der Inputs müssen im Argument von `renderText()` abgefragt werden, zB `renderText(paste("x + y =", input$x + input$y))`.

Screenshot der Shiny-App:



Outputs

Zu jeder `...Output()` -Funktion, gehört eine `render...()`-Funktion, die das Output-UI-Element mit Werten befüllt. Die wichtigsten Outputs sind:

- Text: `textOutput()`, `verbatimTextOutput()`; `renderText()`, `renderPrint()`
- statische Tabelle: `tableOutput()`; `renderTable()`
- dynamische Tabelle: `dataTableOutput()`; `renderDataTable()`
- Plot: `plotOutput()`; `renderPlot()`

Das erste Argument einer Output-Funktion ist `outputId`, die — analog zur `inputId` — den Zugriff auf Outputs in der Server-Funktion erlaubt.

Das erste Argument einer render-Funktion ist `expr`. Dieser Ausdruck wird ausgewertet, wenn der Output neu gezeichnet werden muss.

Führe folgenden Code aus, um die verschiedenen Outputs kennenzulernen.

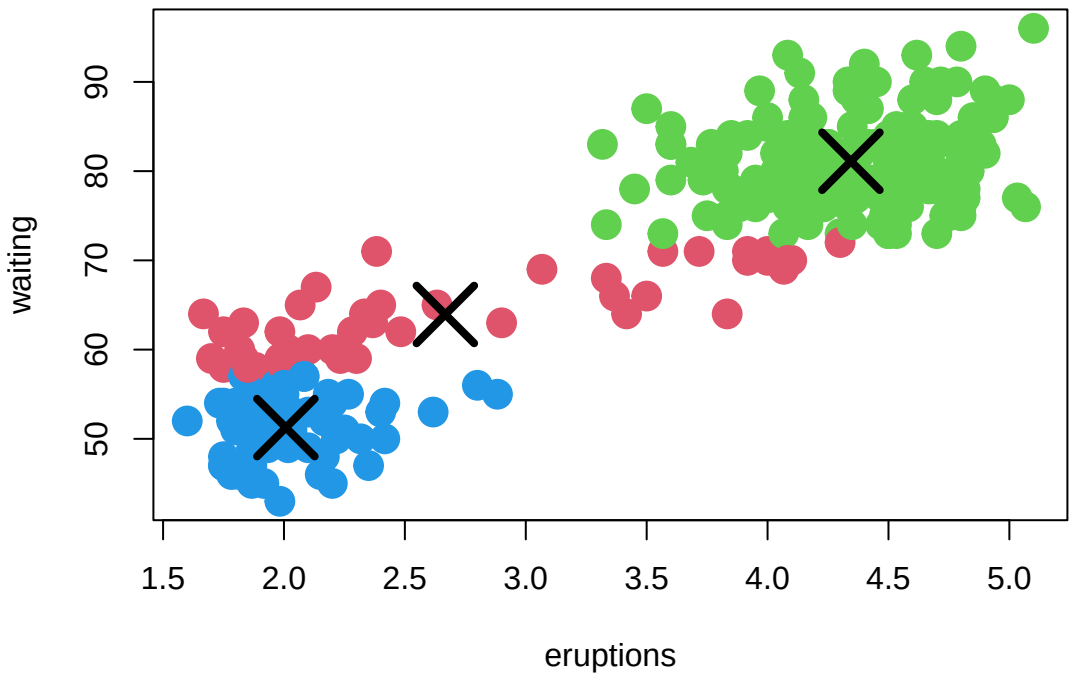
```
library(shiny)
ui <- fluidPage(
  h2("textOutput"), #erzeugt HTML: <h2>Überschrift (Größe 2)</h2>
  textOutput("text1"),
  textOutput("text2"),
  h2("verbatimTextOutput"),
  verbatimTextOutput("verb_text1"),
  verbatimTextOutput("verb_text2"),
  h2("plotOutput"),
  plotOutput("plot"),
  h2("tableOutput"),
  tableOutput("static_table"),
  h2("dataTableOutput"),
  dataTableOutput("dyn_table")
)
server <- function(input, output) {
  output$text1 <- renderText({
    "Hello friend!"
  })
  output$text2 <- renderPrint({
```

```
    "Hello friend!"
  })
  output$verb_text1 <- renderText({
    "Hello friend!"
  })
  output$verb_text2 <- renderPrint({
    "Hello friend!"
  })
  output$plot <- renderPlot({
    plot(sinpi)
    abline(0, 1, col="red")
  })
  output$static_table <- renderTable({
    head(iris)
  })
  output$dyn_table <- renderDataTable({
    iris
  })
}
shinyApp(ui, server)
```

Aufgabe 2:

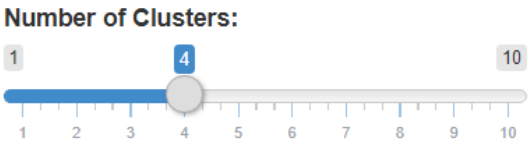
Wandle folgenden Code in eine Shiny-App mit einem Input-Slider für `num_clusters` und zwei Output-Elementen – eines für den Plot, eines für die Summary.

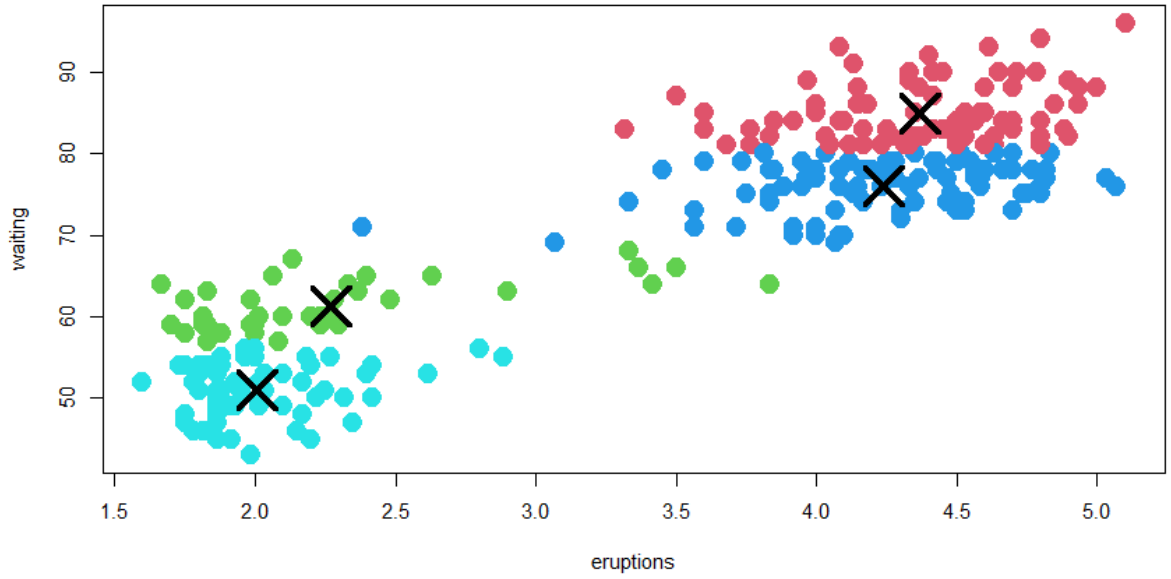
```
num_clusters <- 3

clusters <- kmeans(faithful, num_clusters)
plot(faithful, col = clusters$cluster+1, pch = 20, cex = 3)
points(clusters$centers, pch = 4, cex = 4, lwd = 4)

summary(faithful)
##    eruptions        waiting
##  Min.   :1.600   Min.   :43.0
##  1st Qu.:2.163   1st Qu.:58.0
##  Median :4.000   Median :76.0
##  Mean   :3.488   Mean   :70.9
##  3rd Qu.:4.454   3rd Qu.:82.0
##  Max.   :5.100   Max.   :96.0
```



Hinweis: Alle interaktiven Berechnungen müssen in den Argumenten der render-Funktionen stattfinden.

Screenshot der Shiny-App:






```
  eruptions        waiting
Min.   :1.600   Min.   :43.0
1st Qu.:2.163   1st Qu.:58.0
Median :4.000   Median :76.0
Mean   :3.488   Mean   :70.9
3rd Qu.:4.454   3rd Qu.:82.0
Max.   :5.100   Max.   :96.0
```

Layout

Bemerkung:

Ohne weitere Angaben werden die UI-Elemente untereinander angeordnet. Mit Layouts lässt sich diese Anordnung anpassen. Siehe https://mastering-shiny.org/action-layout.html

Server and Reactivity

Befehle in der Server-Funktion, die Inputs nutzen, dürfen nur in einer sogenannten *reactive expression* genutzt werden, d.h. im ersten Argument einer render-Funktion.

Will man Inputs außerhalb einer `render`-Funktion nutzen, kann man eine *reactive expression* mit `reactive()` erzeugen. Die zurückgegeben Funktion kann wieder nur in einem *reactive contexti* (`render...()`, `reactive()`, ...) benutzt werden.

```
# will not work:
server <- function(input, output) {
```

```
  t <- as.double(input$text)
  output$plot <- renderPlot({
    plot(function(x) sin(t*x))
  })
}

# will work:
server <- function(input, output) {
  get_t <- reactive({
    as.double(input$text)
  })
  output$plot <- renderPlot({
    plot(function(x) sin(get_t()*x))
  })
}
```

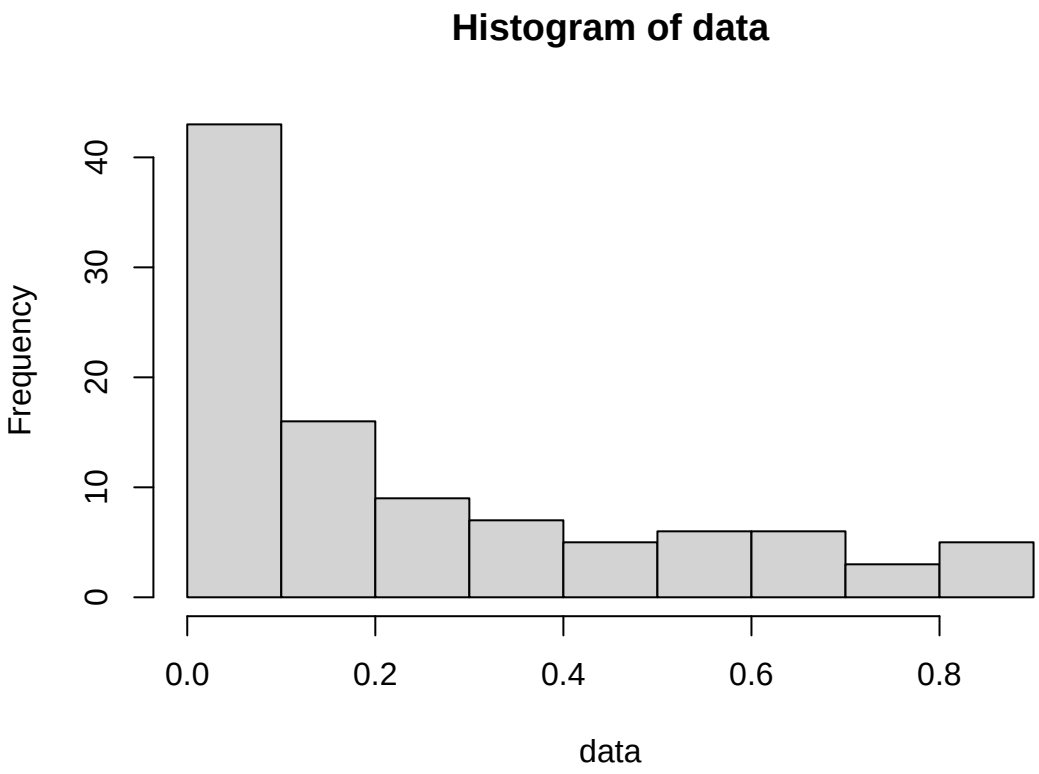
`server()` wird nur einmal ausgeführt. Interaktion wird dadurch möglich, dass render-Funktionen und `reactive()` ihre Argumente als Ausdrücke speichern. Diese Ausdrücke werden ausgewertet, sobald sich etwas in den Inputs ändert, von denen die Ausdrücke abhängen.

Werden Berechnungen durchgeführt, deren Ergebnisse von mehreren Outputs verwendet werden, sollten diese Berechnungen mittels `reactive()` in einer reactive expression durchgeführt werden, damit sie nicht öfter als nötig durchgeführt werden.

Aufgabe 3:

Wandle folgenden Code in eine Shiny-App mit Inputs für `n` (ganze Zahl ≥ 1), `shape1` ($\{0.1, 0.2, \ldots, 10\}$), `shape2` ($\{0.1, 0.2, \ldots, 10\}$) und Outputs für `hist()` und `summary()`.

```
n <- 100
shape1 <- 0.5
shape2 <- 2

data <- rbeta(n, shape1, shape2)

hist(data)

summary(data)
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0000022 0.0384128 0.1335174 0.2417039 0.3948022 0.8765602
```



Hinweis: `data` muss als *reactive expression* (wie oben `t` bzw `get_t()`) verwendet werden.

Screenshot der Shiny-App:

Number of obs

100

shape1

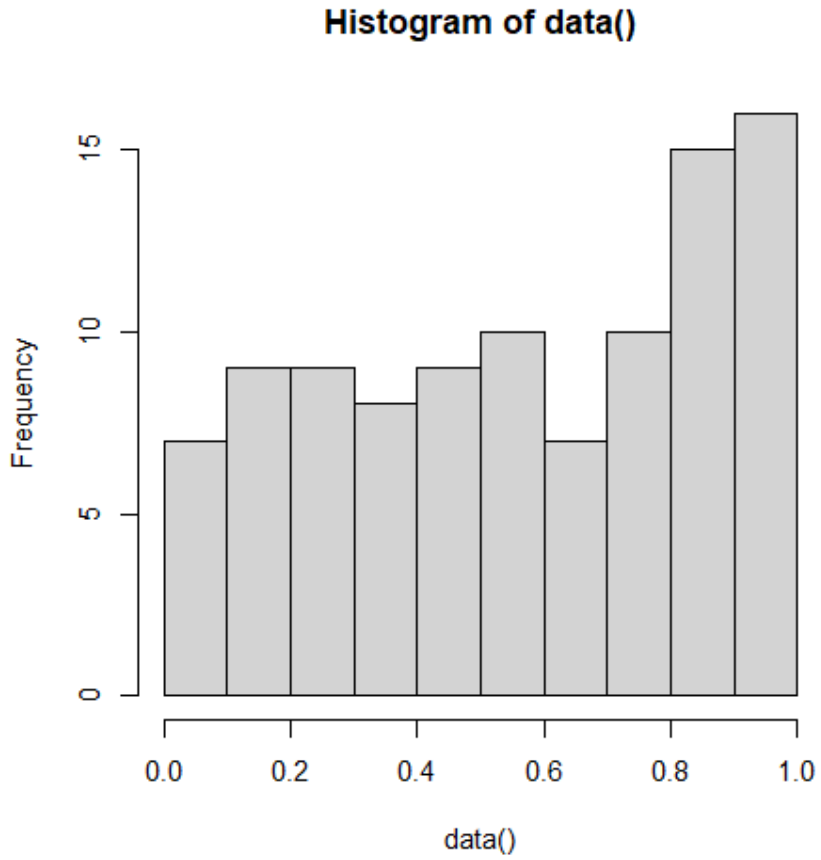
0.1 1 10

0.1 1.1 2.1 3.1 4.1 5.1 6.1 7.1 8.1 9.1 10

shape2

0.1 1 10

0.1 1.1 2.1 3.1 4.1 5.1 6.1 7.1 8.1 9.1 10




```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.02063 0.30992 0.56829 0.56217 0.85304 0.99826
```

Plot Interaction

`plotOutput(click="my_click")` erzeugt ein Input-Element `my_click`, welches angibt, wohin in einem Plot geklickt wurde. Siehe auch in `?plotOutput` die Argumente `click`, `brush`, `hover`, `dblclick`.

Wir möchten in einem Plot durch Klicken weitere Datenpunkte hinzufügen. Die Daten werden in einem Tibble gespeichert. Da sich die Daten während der Ausführung der App ändern (alte Werte werden beibehalten und neue hinzugefügt) und Outputs auf diese Änderung reagieren sollen, müssen die Daten als *reactive values* markiert werden. Dies geschieht mit `reactiveValues()`.

Die Änderung der Daten ist wiederum eine Reaktion auf einen Klick in den Plot. Mit `observeEvent(eventExpr, handlerExpr)` wird `handlerExpr` ausgeführt, sobald eine Änderung einer reactive expression in `eventExpr` passiert. Hier setzen wir `eventExpr` auf `input$my_click`, was einen Klick

Führe folgenden Code aus und klicke mehrere Male an verschiedene Positionen im Plot.

```
library(tibble)
library(ggplot2)

ui <- fluidPage(
  plotOutput("plot", click = "my_click")
)

server <- function(input, output) {

  values <- reactiveValues(d = tibble(x=0, y=0))

  output$plot <- renderPlot({
    ggplot(values$d, aes(x, y)) +
      geom_point() + geom_smooth(method="lm", formula=y~x, se=FALSE) +
      xlim(-1, 1) + ylim(-1, 1)
  })

  observeEvent(input$my_click, {
    values$d <- rbind(
      values$d,
      c(x=input$my_click$x, y=input$my_click$y)
    )
  })

}

shinyApp(ui, server)
```

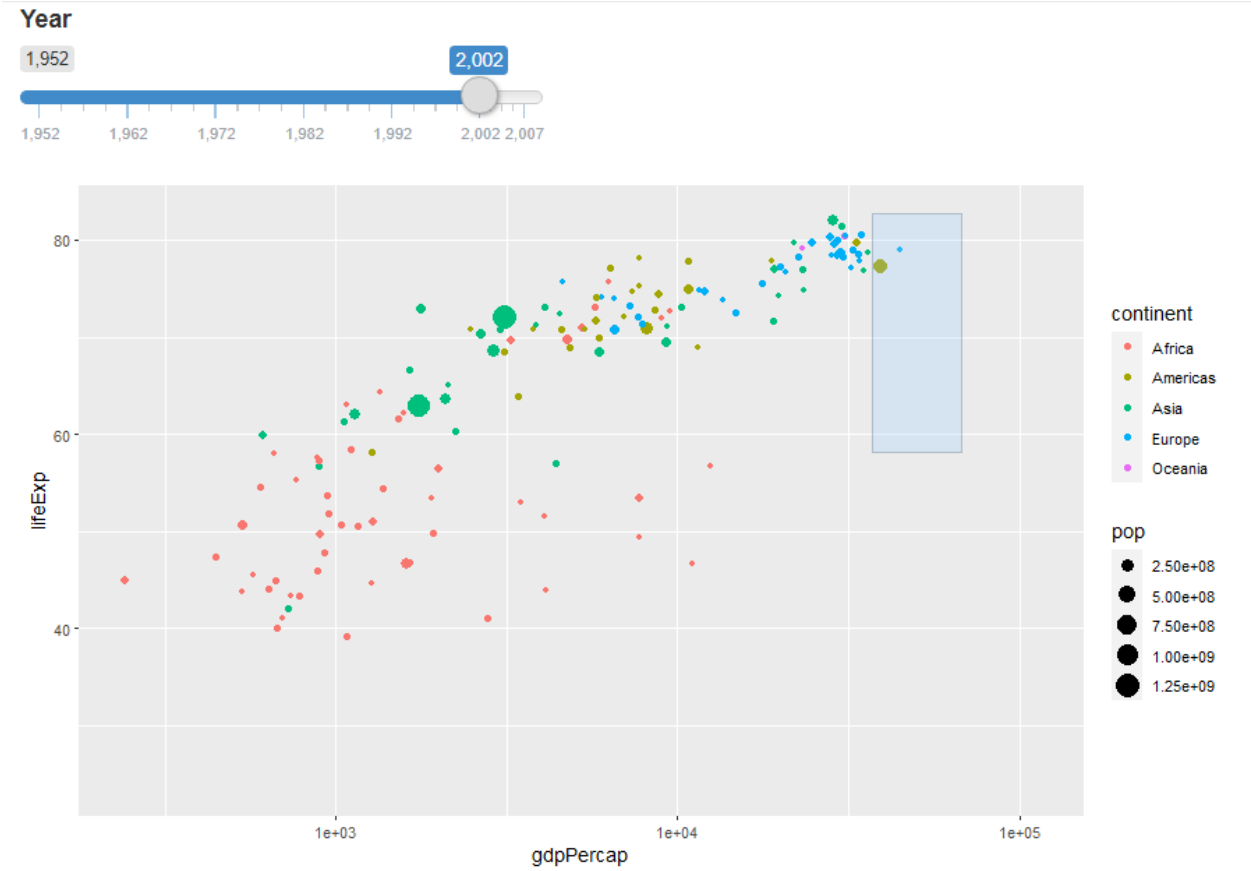
Bemerkung: `reactive()` gibt eine Funktion zurück. Auf darin verwendete reaktive Elemente wird reagiert. Andere reaktive Elemente reagieren wiederum auf diese Änderung. `reactiveValues()` gibt eine Liste zurück. Die Liste ändert sich nicht automatisch, sondern muss von anderen Elementen verändert werden. Andere reaktive Elemente reagieren wiederum auf diese Änderung. `observeEvent()` hat keine Rückgabe. Es sorgt dafür, dass ein Ausdruck bei Änderungen bestimmter Elemente ausgeführt wird.

Aufgabe 4:

Die folgende Shiny-App zeigt Plots *GDP per Capita* und *life expectancy* für verschiedene Länder in verschiedenen Jahren. Aus den Plots können wir allerdings nicht direkt erkennen, welcher Punkt zu welchem Land korrespondiert. Lies in `?plotOutput` über das Argument `brush`. Füge der App ein Tabellen-Output hinzu, das die Zeilen aus `gapminder` anzeigt, die mittels brush-Auswahl (Kasten mit der Maus ziehen) markiert wurden.

```r
library(shiny)
library(ggplot2)
library(gapminder)
library(dplyr)

ui <- fluidPage(
  sliderInput("year", "Year", 1952, 2007, 2002, 5),
  plotOutput("plot")
)

server <- function(input, output) {
  output$plot <- renderPlot({
    gapminder %>%
      filter(year == input$year) %>%
      ggplot(aes(gdpPercap, lifeExp, color=continent, size=pop)) +
      geom_point() +
      scale_x_log10(limits = range(gapminder$gdpPercap)) +
      lims(y = range(gapminder$lifeExp))
  })
}

shinyApp(ui, server)
```

Screenshot der Shiny-App:



country	continent	year	lifeExp	pop	gdpPercap
Norway	Europe	2002	79.05	4535591	44683.98
United States	Americas	2002	77.31	287675526	39097.10

Bonus

Hier noch zwei weitere interessante Funktionalitäten von Shiny:

- Timer: Neuberechnung durch Timer auslösen, zB jede Sekunde einmal neu ausführen.
- On Click: Führe Neuberechnung nicht bei jeder Änderung aus, sondern erst bei Klick auf Button.

Führe folgende Code-Blöcke aus.

```
# a function to plot a histogram, will be used later
histogram <- function(x1, x2, binwidth = 0.1, xlim = c(-3, 3)) {
  df <- data.frame(
    x = c(x1, x2),
    g = c(rep("x1", length(x1)), rep("x2", length(x2)))
  )
  ggplot(df, aes(x, fill = g)) +
    geom_histogram(binwidth = binwidth) +
    coord_cartesian(xlim = xlim)
}
```

Timer

```
ui <- fluidPage(
  numericInput("lambda1", label = "lambda1", value = 3),
  numericInput("lambda2", label = "lambda2", value = 3),
  numericInput("n", label = "n", value = 100, min = 0),
  plotOutput("hist")
)

server <- function(input, output, session) {
  timer <- reactiveTimer(500) # timer changes every 500ms

  x1 <- reactive({
    timer() # reacts on timer()
    rpois(input$n, input$lambda1)
  })
  x2 <- reactive({
    timer() # reacts on timer()
    rpois(input$n, input$lambda2)
  })

  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40)) # reacts on x1() and x2()
  }, res = 96)
}

shinyApp(ui, server)
```

On Click

```
ui <- fluidPage(
  numericInput("lambda1", label = "lambda1", value = 3),
  numericInput("lambda2", label = "lambda2", value = 3),
  numericInput("n", label = "n", value = 1e4, min = 0),
  actionButton("simulate", "Simulate!"),
  plotOutput("hist"))
)

server <- function(input, output, session) {
  x1 <- eventReactive(input$simulate, { # does not react on input$n or input$lambda1, but on input$simu
    rpois(input$n, input$lambda1)
  })
  x2 <- eventReactive(input$simulate, { # does not react on input$n or input$lambda2, but on input$simu
    rpois(input$n, input$lambda2)
  })
  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40)) # reacts on x1() and x2()
  }, res = 96)
}

shinyApp(ui, server)
```