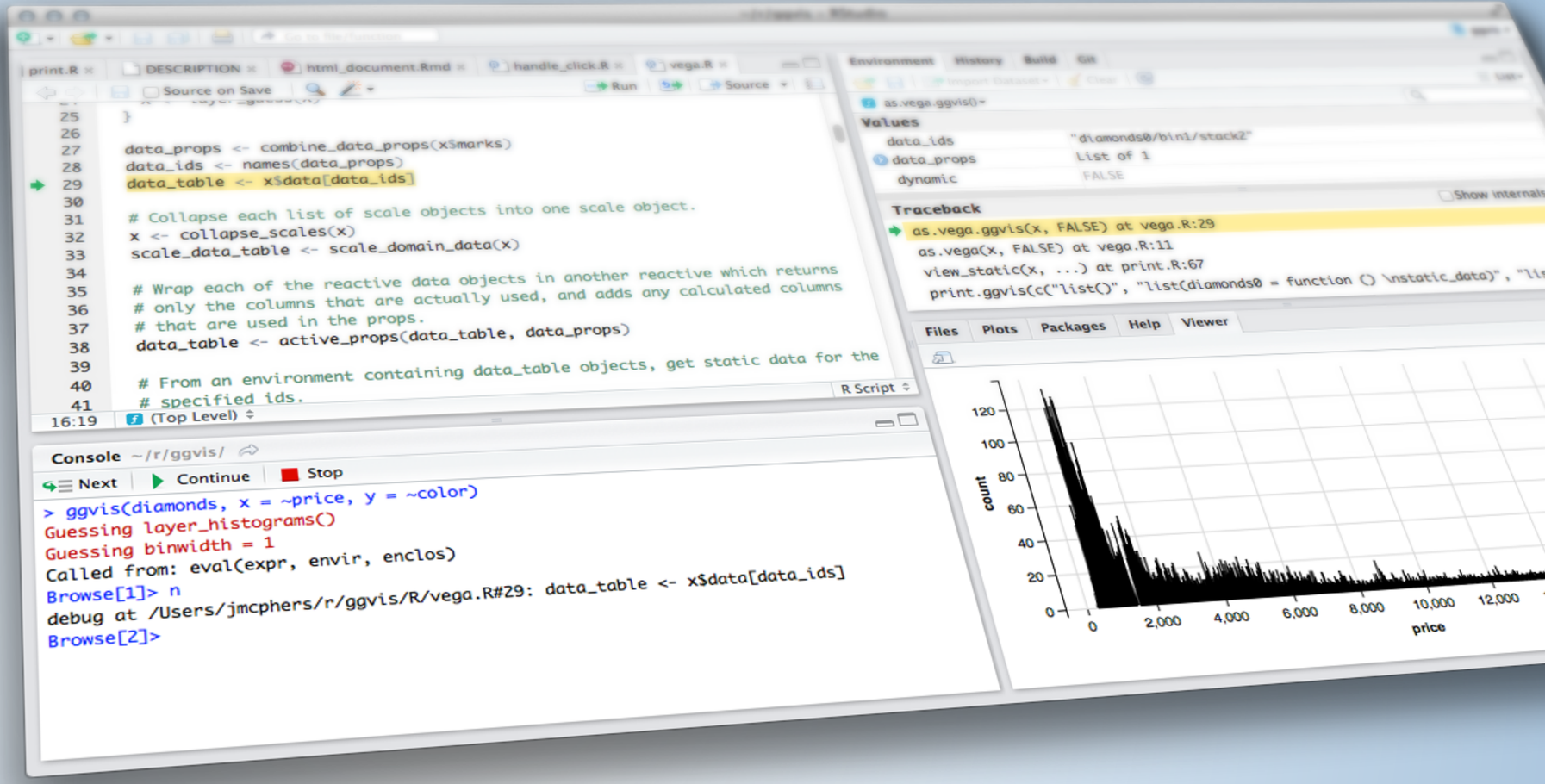


REACTIVE PROGRAMMING

PART 1

Shiny from RStudio™

OUTLINE

- Reactivity 101
- Reactive objects
  - Reactive sources and endpoints
  - Reactive conductors
  - Implementation
  - Observers and side effects
- Render functions

Reactivity

101

REACTIVITY 101

- When value of variable **x** changes, anything that relies on **x** is re-evaluated
- Contrast with regular R:

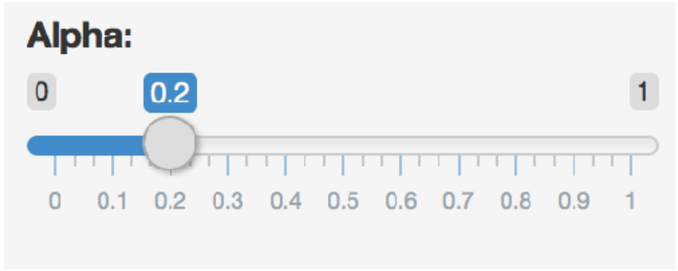
```
x <- 5
y <- x + 1
x <- 10
# What is y? 6 or 11?
```

REACTIONS

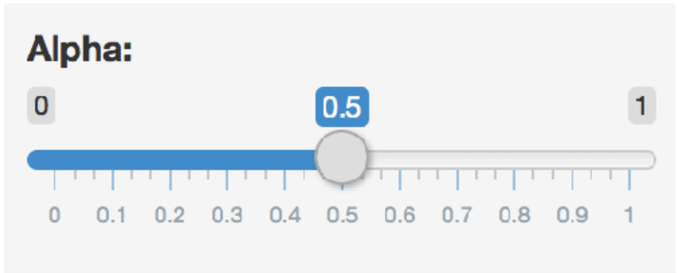
The **input$** list stores the current value of each input object under its name.

```
# Set alpha level
sliderInput(inputId = "alpha",
            label = "Alpha:",
            min = 0, max = 1,
            value = 0.5)
```

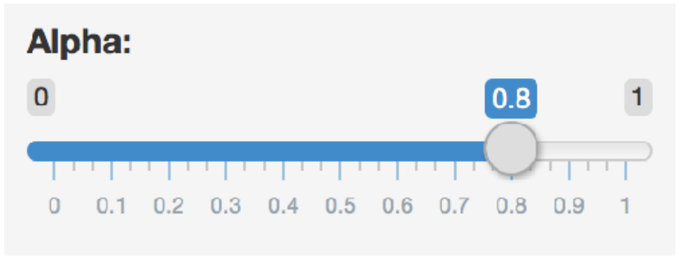
input$alpha



input$alpha = 0.2



input$alpha = 0.5

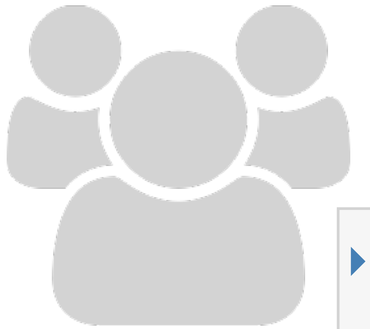


input$alpha = 0.8

REACTIVITY 101

Reactivity automatically occurs when an **`input`** value is used to render an **`output`** object

```
# Define server function required to create the scatterplot
server <- function(input, output) {
   # Create the scatterplot object the plotOutput function is expecting
    output$scatterplot <- renderPlot(
     ggplot(data = movies, aes_string(x = input$x, y = input$y,
                                      color = input$z)) +
       geom_point(alpha = input$alpha)
    )
}
```



EXERCISE

- Go back to the app you built earlier
- Add a new `sliderInput` defining the size of points (ranging from 0 to 5)
- Use this variable in the geom of the `ggplot` function as the size argument
- Run the app to ensure that point sizes react when you move the slider
- Compare your code / output with the person sitting next to / nearby you

3m 00s

SOLUTION

Solution to the previous exercise

`movies_06.R`

Reactive objects

TYPES OF REACTIVE OBJECTS

Reactive source

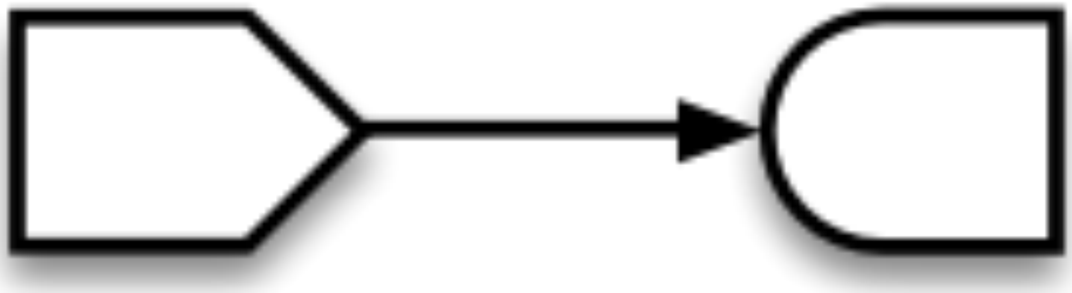
Reactive conductor

Reactive endpoint

Reactive sources and endpoints

SOURCES AND ENDPOINTS

- **Reactive source:** Typically, this is user input that comes through a browser interface
- **Reactive endpoint:** Something that appears in the user's browser window, such as a plot or a table of values

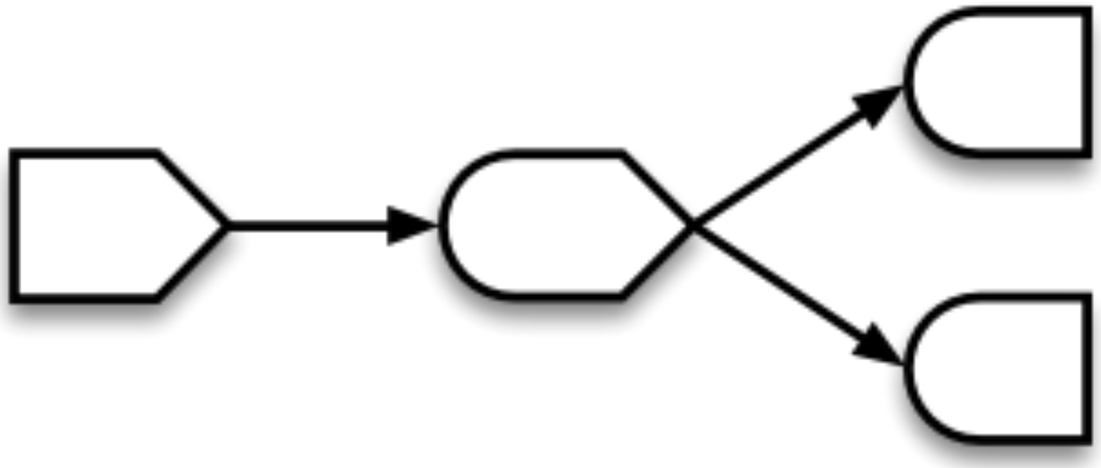


- This is the built-in reactivity discussed in the previous section
- A reactive source can be connected to multiple endpoints, and vice versa

Reactive conductors

CONDUCTORS

- **Reactive conductor:** Reactive component between a source and an endpoint
- A conductor can both be a dependent (consumer) and have dependents (producer)
  - Sources can only be producers (they can have dependents)
  - Endpoints can only be consumers (they can be dependents)



DEMO

Suppose you want the option to plot only certain types of movies as well as report how many such movies are plotted:

1. Add a UI element for the user to select which type(s) of movies they want to plot
2. Filter for chosen title type and save as a new (reactive) expression
3. Use new data frame (which is reactive) for plotting
4. Use new data frame (which is reactive) also for reporting number of observations

DEMO

1. **Add a UI element for the user to select which type(s) of movies they want to plot**

```
# Select which types of movies to plot
checkboxGroupInput(inputId = "selected_type",
                   label = "Select movie type(s):",
                   choices = c("Documentary", "Feature Film", "TV Movie"),
                   selected = "Feature Film")
```

DEMO

2. Filter for chosen title type and save the new data frame as a reactive expression

before app:

```
library(dplyr)
```

server:

```
# Create a subset of data filtering for chosen titl
movies_subset <- reactive({
  req(input$selected_type)
  filter(movies, title_type %in% input$selected_ty
})
```

Creates a **cached expression** that knows it is out of date when input changes

DEMO

3. Use new data frame (which is reactive) for plotting

```
# Create the scatterplot object the plotOutput function is expecting
output$scatterplot <- renderPlot({
  ggplot(data = movies_subset(), aes_string(x =
                                            col

    geom_point(…) +
    …
})
```

Cached - only re-run when inputs change

DEMO

4. Use new data frame (which is reactive) also for printing number of observations

ui:

```
mainPanel(
  …
  # Print number of obs plotted
  uiOutput(outputId = "n"),
  …
  )
```

server:

```
# Print number of movies plotted
output$n <- renderUI({
  types <- movies_subset()$title_type %>%
    factor(levels = input$selected_type)
  counts <- table(types)

  tagList("There are",
    counts,
    paste(input$selected_type, col=", "),
    "movies in this dataset.",
    tags$br())
})
```

DEMO

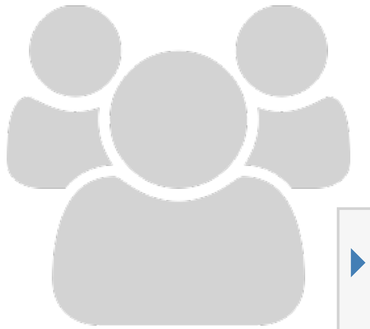
Putting it all together…

`movies_07.R`

(also notice the HTML tags,
added for visual separation, in the `mainPanel`)

WHEN TO USE REACTIVES

- By using a reactive expression for the subsetted data frame, we were able to get away with subsetting once and then using the result twice
- In general, reactive conductors let you
  - not repeat yourself (i.e. avoid copy-and-paste code) which is a maintenance boon)
  - decompose large, complex (code-wise, not necessarily CPU-wise) calculations into smaller pieces to make them more understandable
- These benefits are similar to what happens when you decompose a large complex R script into a series of small functions that build on each other



EXERCISE

- For consistency, in **`movies_07.R`**, there should be at least one more spot on the app where the new **`movies_subset`** dataset should be used, instead of the full **`movies`** dataset
  - *Hint:* Does the data table match the plotted data?
- Find and fix
- Run the app to confirm your fix is working
- Compare your code / output with the person sitting next to / nearby you

3m 00s

SOLUTION

Solution to the previous exercise

`movies_08.R`

REACTIVE CONTEXTS

- Reactive values can only be used inside **reactive contexts**. A reactive context identifies a chunk of code that needs to be rerun if any reactive values change.
- Any reactive consumer (`render*()` or `reactive()`) is a reactive context
- Accessing a reactive value outside of a reactive context is an error

```
server <- function(input, output, session) {
  print(input$x)
}
# ERROR: Operation not allowed without an active reactive context
```

EXERCISE

Suppose we want to plot only a random sample of movies, of size determined by the user. What is wrong with the following?

ui:

```
# Select sample size
numericInput("n_samp", "Sample size:", min = 1, max = nrow(movies), value = nrow(movies))
```

server:

```
# Create a new data frame that is a sample of n_samp observations from movies
movies_sample <- sample_n(movies, input$n_samp)

# Plot the sampled movies
output$scatterplot <- ggplot(data = movies_sample,
                             aes_string(x = input$x, y = input$y, color = input$z)) +
                      geom_point(…)
```

1 m 00 s

SOLUTION

Solution can also be found in **movies_09.R**.

Note that **output$n** and **output$datatable** are also updated in the script.

ui:

```
# Select sample size
numericInput("n_samp", "Sample size:", min = 1, max = nrow(movies), value = 50)
```

server:

```
# Create a new data frame that is n_samp observations from selected type movies
movies_sample <- reactive({
 sample_n(movies_subset(), input$n_samp)
})

# Plot the sampled movies
output$scatterplot <- renderPlot({
  ggplot(data = movies_sample(), aes_string(x = input$x, y = input$y, color = input$z)) +
    geom_point(…)
})
```

Implementation

IMPLEMENTATION OF REACTIVE OBJECTS

- **Reactive values – `reactiveValues()`:** implementation of reactive sources
  - e.g. `input` object is a reactive value, which looks like a list, and contains many individual reactive values that are set by input from the web browser
- **Reactive expressions – `reactive()`:** implementation of reactive conductors
  - Can access reactive values or other reactive expressions, and they return a value
  - Useful for caching the results of any procedure that happens in response to user input
  - e.g. reactive data frame subsets we created earlier
- **Observers – `observe()`:** implementation of reactive endpoints
  - Can access reactive sources and reactive expressions, but they don't return a value; they are used for their **side effects**
  - e.g. `output` object is a reactive observer, which also looks like a list, and contains many individual reactive observers that are created by using reactive values and expressions in reactive functions

REACTIVITY ONLY WORKS WITH REACTIVE OBJECTS

- Only reactive primitives (like the ones on the previous slide) and things built on top of reactive primitives, will elicit reactivity. In particular, do NOT expect changes to "normal" variables to cause reactivity.

```
x <- 10
y <- reactive({ x })

# Much later...
x <- 20
```

REACTIVE VALUES

- Like an R environment object (or what other languages call a hash table or dictionary), but reactive
- Like the `input` object, but not read-only

```
rv <- reactiveValues(x = 10)
rv$x <- 20
rv$y <- mtcars
```

REACTIVE VALUES

- Reading a value from a **`reactiveValues`** object is a reactive operation.
  - The act of reading it means the current reactive conductor or endpoint will be notified the next time the value changes.
- Maybe surprisingly, setting/updating a value on a **`reactiveValues`** object is *not* in itself a reactive operation, meaning no relationship is established between the current reactive conductor or endpoint (if any!) and the **`reactiveValues`** object.

REACTIVE VALUES

- New feature: **`reactiveVal`**, similar to **`reactiveValues`** but for a single value instead of a whole list/environment of values.
- Reading a **`reactiveVal`** is congruent with reading a reactive expression

```
rv <- reactiveVal(10)    # declare/initialize
rv()                     # read
rv(20)                   # write
```

Observers and side effects

EXERCISE

Suppose we want the user to provide a title for the plot. What is wrong with the following, and how would you fix it? See `movies_10.R`.

ui:

```
textInput(inputId = "plot_title",
          label = "Plot title",
          placeholder = "Enter text to be used as plot title"),
```

server:

```
output$pretty_plot_title <- toTitleCase(input$plot_title)

output$scatterplot <- renderPlot({
  ggplot(data = movies_sample(), aes_string(x = input$x, y = input$y, color = input$z)) +
    geom_point(alpha = input$alpha, size = input$size) +
    labs(title = output$pretty_plot_title)
})
```

3m 00s

SOLUTION

Observers do not have dependencies, use reactives instead.
Solution can also be found in `movies_11.R`.

ui:

```
textInput(inputId = "plot_title",
          label = "Plot title",
          placeholder = "Enter text to be used as plot title"),
```

server:

```
pretty_plot_title <- reactive({ toTitleCase(input$plot_title) })

output$scatterplot <- renderPlot({
  ggplot(data = movies_sample(), aes_string(x = input$x, y = input$y, color = input$z)) +
    geom_point(alpha = input$alpha, size = input$size) +
    labs(title = pretty_plot_title())
})
```

REACTIVE EXPRESSIONS VS. OBSERVERS

- Similarities: Both store expressions that can be executed
- Differences:
  - Reactive expressions return values, but observers don't
  - Observers (and endpoints in general) *eagerly* respond to reactives, but reactive expressions (and conductors in general) do not
  - Reactive expressions must not have *side effects*, while observers are *only* useful for their side effects

DEMO

We cheated earlier, let's make it right with an observer!

See `movies_12.R`.

```
server <- function(input, output, session) {
  ...
  # Update the maximum allowed n_samp for selected type movies
  observe({
    updateNumericInput(session, inputId = "n_samp",
                       value = nrow(movies_subset())
                       )
  })
 ...
}
```

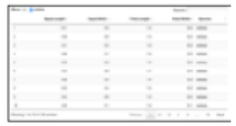
Render functions

RENDER FUNCTIONS

```
render*({ [code_chunk] })
```

- Provide a code chunk that describes how an output should be populated
- The output will update in response to changes in any reactive values or reactive expressions that are used in the code chunk

LIST OF REACTIVE FUNCTIONS



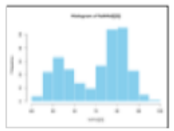
DT::**renderDataTable**(expr, options, callback, escape, env, quoted)

works with

dataTableOutput(outputId, icon, …)

renderImage(expr, env, quoted, deleteFile)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

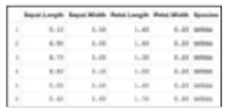


renderPlot(expr, width, height, res, …, env, quoted, func)

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

renderPrint(expr, env, quoted, func, width)

verbatimTextOutput(outputId)



renderTable(expr,…, env, quoted, func)

tableOutput(outputId)

foo

renderText(expr, env, quoted, func)

textOutput(outputId, container, inline)

renderUI(expr, env, quoted, func)

uiOutput(outputId, inline, container, …)

& **htmlOutput**(outputId, inline, container, …)

RECAP

```
render*({ [code_chunk] })
```

- These functions make objects to display
- Results should always be saved to **`output$`**
- They make an observer object that has a block of code associated with it
- The object will rerun the entire code block to update itself whenever it is invalidated

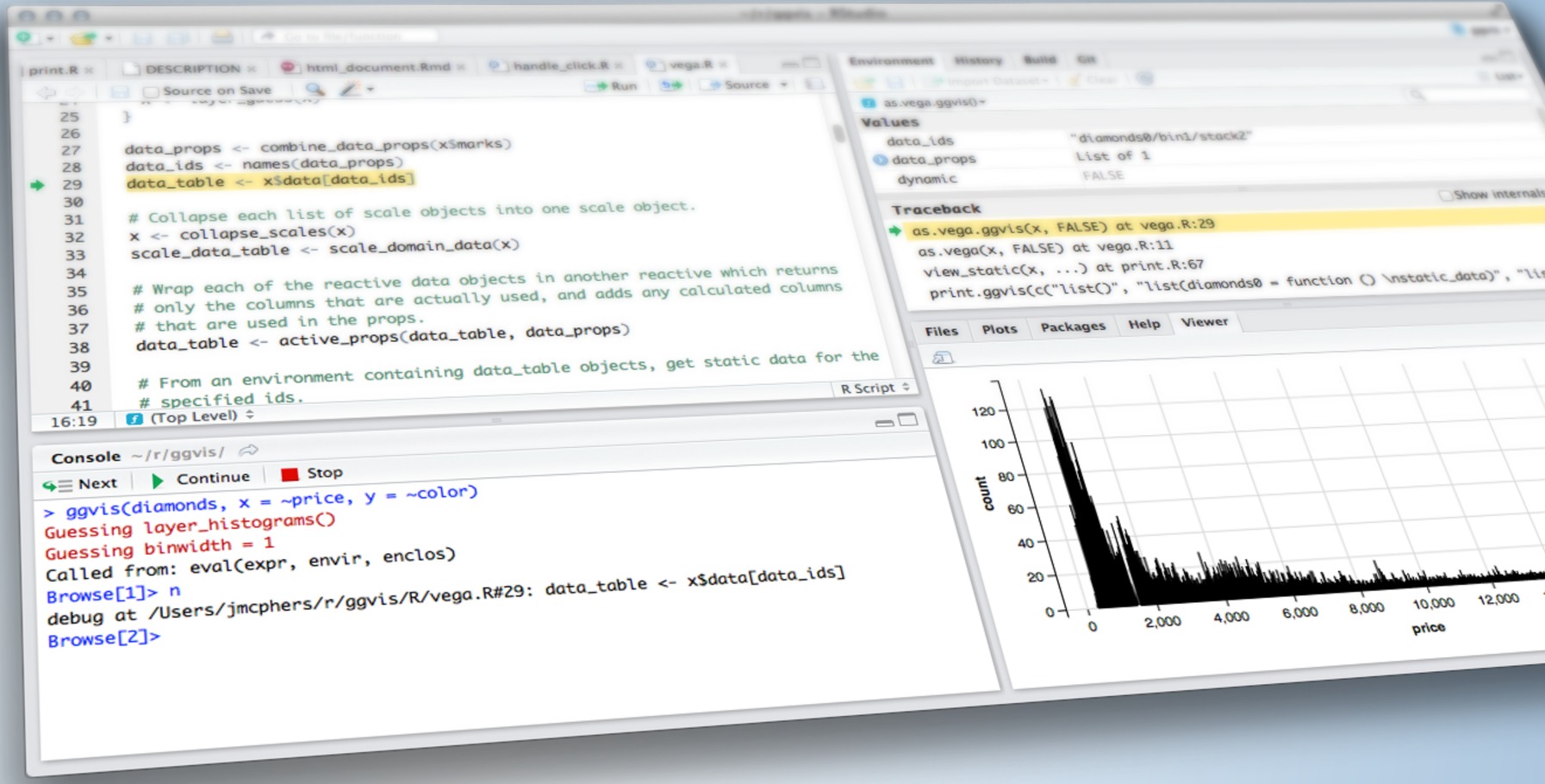
EXERCISE

- Run the app in `movies_12.R`.
- Try entering a few different plot titles and observe that the plot title updates however the sampled data that is being plotted does not.
- Given that the `renderPlot()` function reruns each time `input$plot_title` changes, why does the sample stay the same?

1m 00s

SOLUTION

Because the data frame that is used in the plot is defined as a reactive expression with a code chunk that does not depend on `input$plot_title`.



REACTIVE PROGRAMMING

PART 1

Shiny from RStudio™