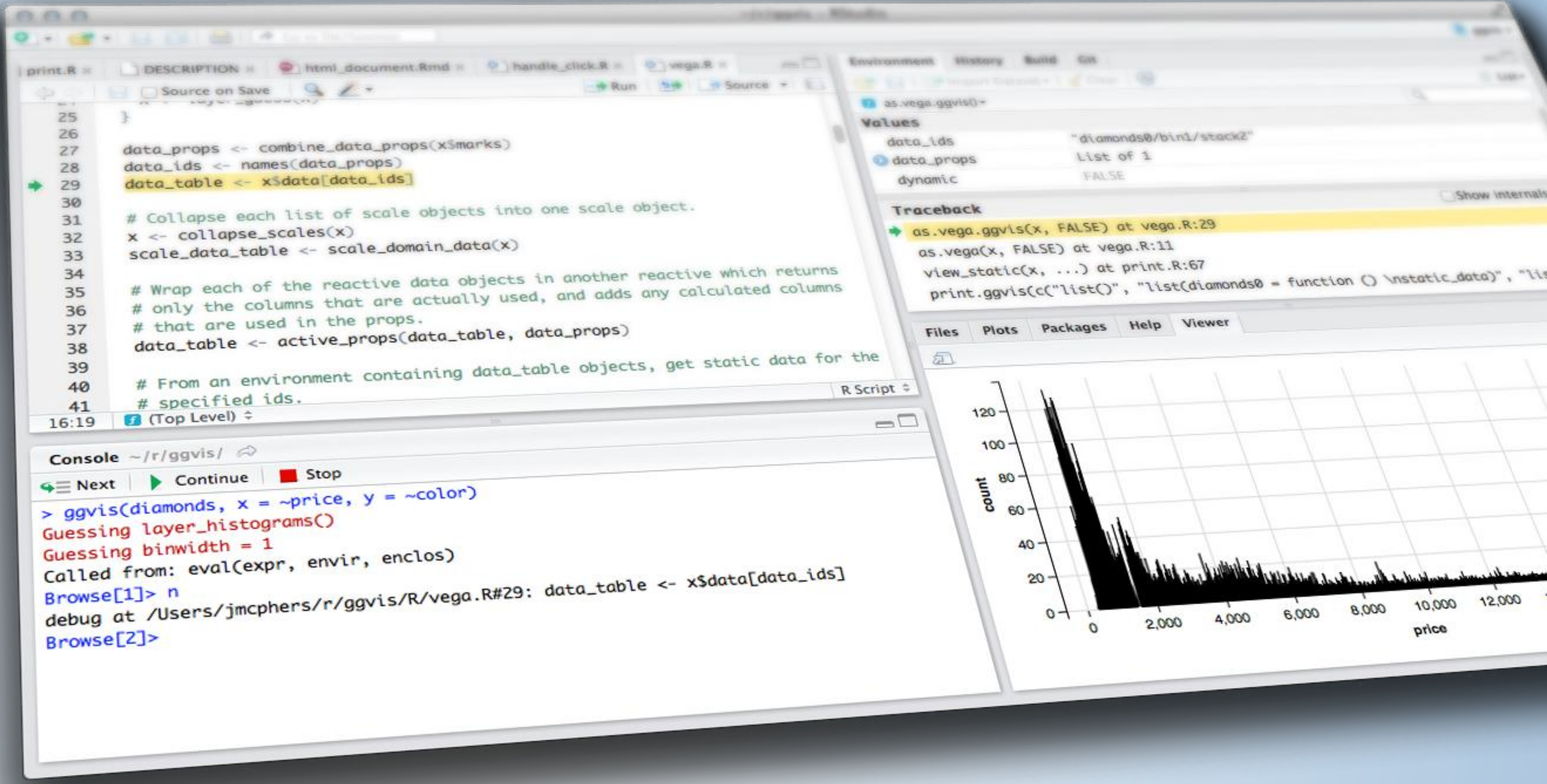
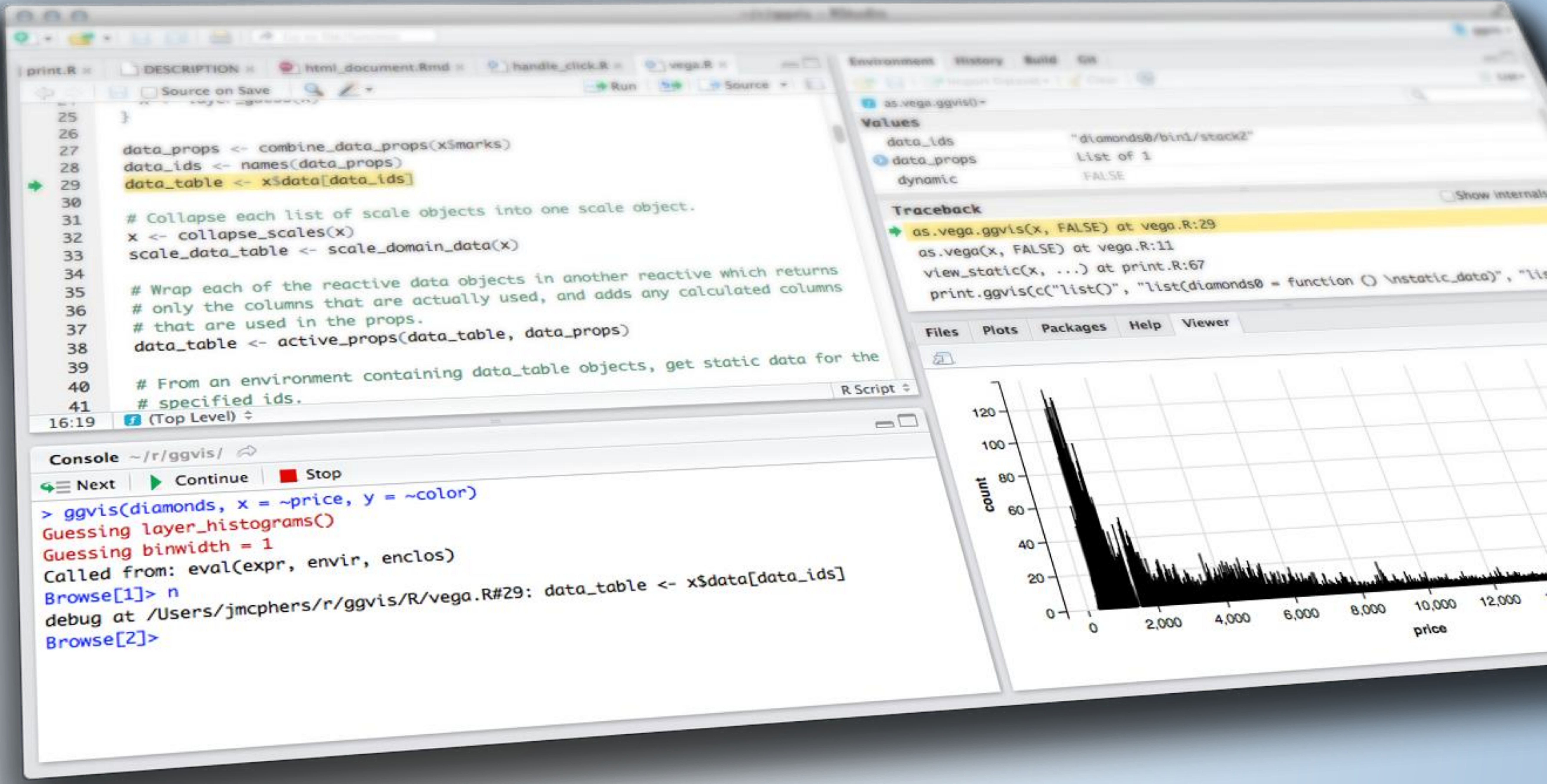


VISUALIZATIONS WITH PLOTLY & INTERACTIVE DATA

Shiny from posit

OUTLINE

- Homework 1 Feedback
  - Troubleshooting Tips
- Interactive Visualizations
  - ggplot2
    - Building a plot
    - Refresher
  - Plotly
    - ggplotly
    - plotly standalone
    - plotly in shiny
  - Advanced DT
- Advanced Reactivity
  - Reactivity catalog
  - Reactivity review
  - Checking preconditions
  - Time as a reactive source
  - Limiting rate

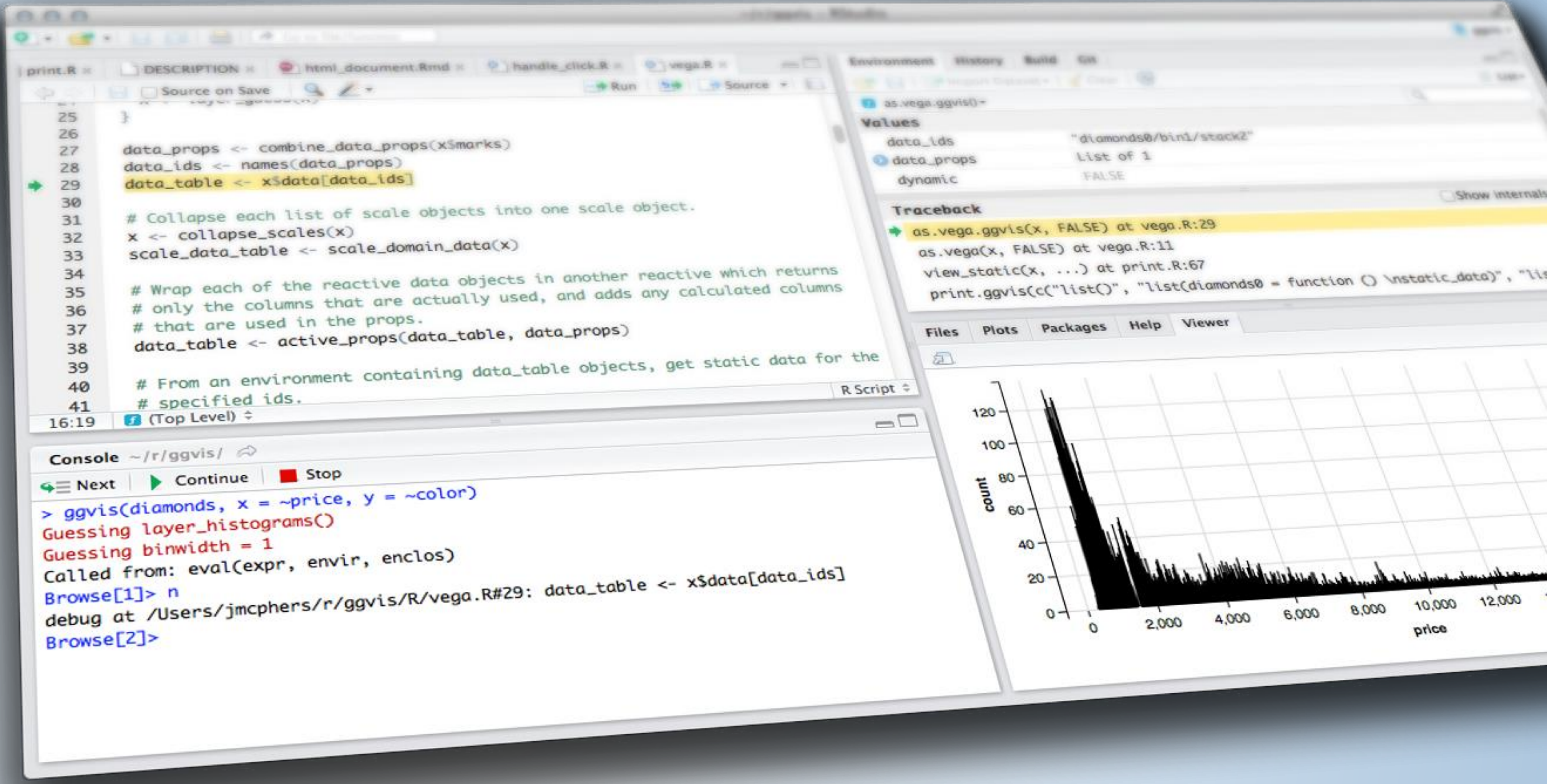


HOMEWORK 1 FEEDBACK

Shiny from posit

FEEDBACK

- Take more ownership of the app - A few submissions were clear repurposing of in-class examples
- You can use <u>one</u> reactive function for multiple figures, and you can have them piggyback off each other. IE, one reactive function can feed into another. Look at the way the melted data is used in the Star Wars dashboard for an example.
- Try to avoid scrolling in your UI (especially with figure specific filters)
- library(dplyr) function mutate() and case_when()
- Please put all code in the app.R file unless written in another language
- If using headers or other HTML in the titlePanel, also set the title

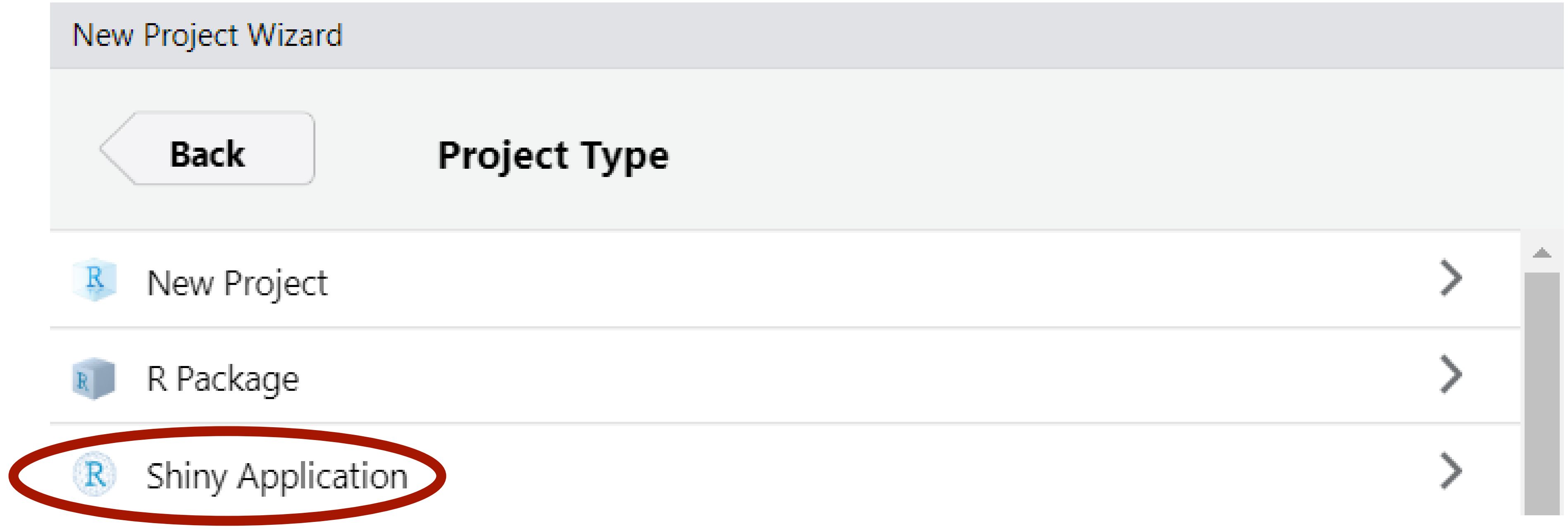


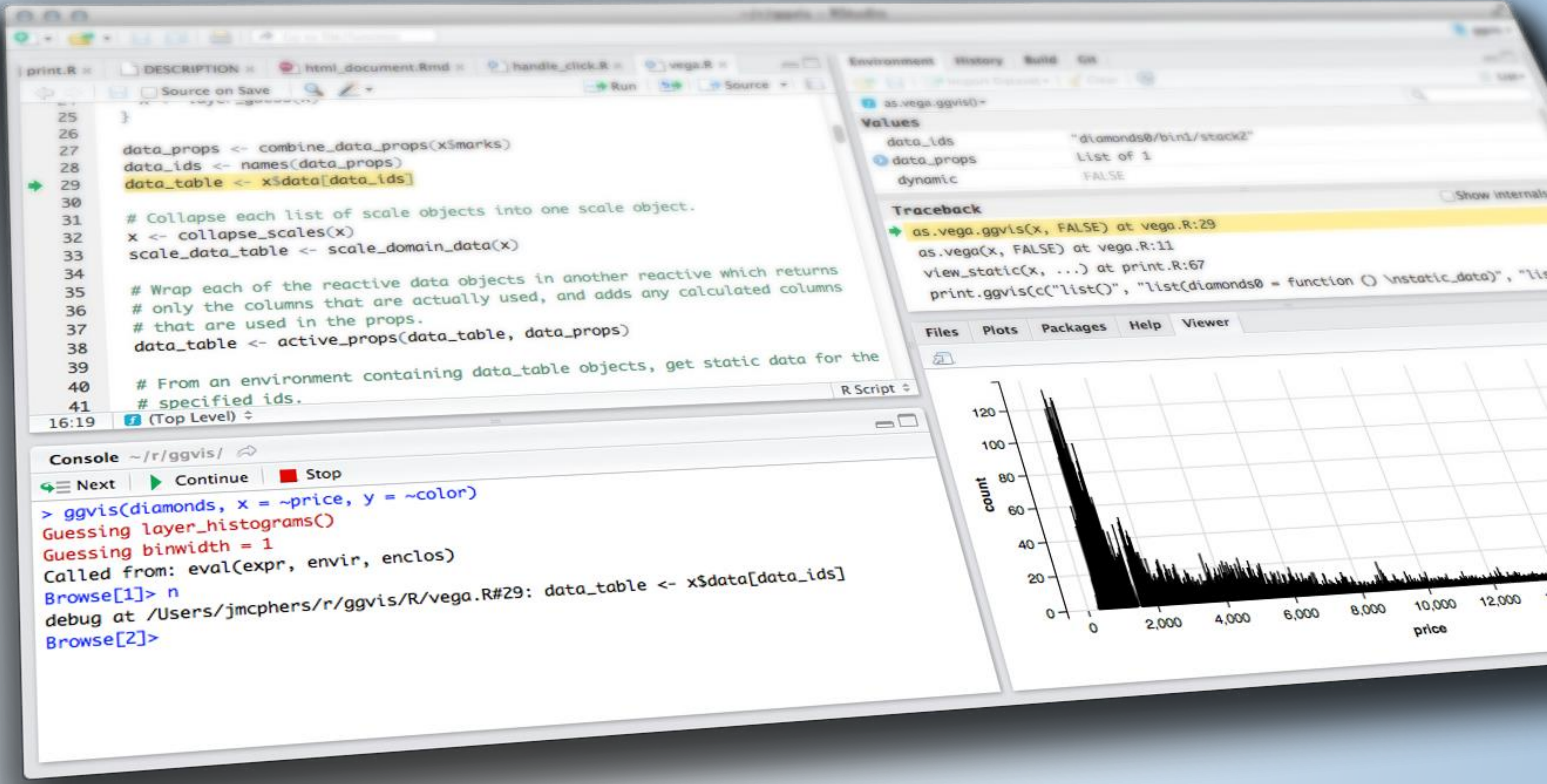
TROUBLESHOOTING SHINY APP TIPS

Shiny from posit

INPUT LIST TESTING

- In the console or separate script, create some dummy inputs to test code before running the entire app
  - `input <- NULL`
  - `input$input_name <- “Some value”`
  - Run portions of your code line by line (skipping `req()` functions)
- Clear your R session before running the app sometimes.
  - Old values might have been removed from your code, and they won’t appear correctly.
- `print()` data.frame summaries and values in your reactive/render functions and make sure they are what you expected
- Start your project as a Shiny Application
  - May need to update RStudio for this option





VISUALIZATIONS WITH PLOTLY & INTERACTIVE DATA

Shiny from posit

“Most of us need to listen to the music to understand how beautiful it is. But often that’s how we present statistics: we just show the notes, we don’t play the music.”

-Hans Rosling

ggplot2 refresh

CREATING A GGPLOT

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and geoms—visual marks that represent data points.

To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.

Complete the template below to build a graph.

ggplot (data = <DATA>) + — required
<GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
stat = <STAT>, position = <POSITION>) +
<COORDINATE_FUNCTION> +
<FACET_FUNCTION> +
<SCALE_FUNCTION> +
<THEME_FUNCTION>

Not required, sensible defaults supplied

ggplot(data = mpg, **aes(**x = cty, y = hwy**))** Begins a plot that you finish by adding layers to. Add one geom function per layer.

aesthetic mappings | data | geom

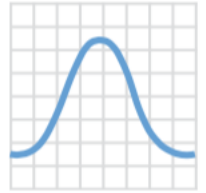
qplot(x = cty, y = hwy, data = mpg, geom = "point"**)** Creates a complete plot with given data, geom, and mappings. Supplies many useful defaults.

TYPES OF PLOTS

ONE VARIABLE continuous

c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)

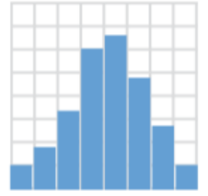
c + geom_area(stat = "bin")
x, y, alpha, color, fill, linetype, size



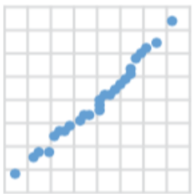
c + geom_density(kernel = "gaussian"**)**
x, y, alpha, color, fill, group, linetype, size, weight

c + geom_dotplot()
x, y, alpha, color, fill

c + geom_freqpoly() x, y, alpha, color, group, linetype, size



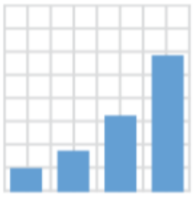
c + geom_histogram(binwidth = 5**)** x, y, alpha, color, fill, linetype, size, weight



c2 + geom_qq(aes(sample = hwy)**)** x, y, alpha, color, fill, linetype, size, weight

discrete

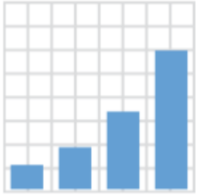
d <- ggplot(mpg, aes(fl))



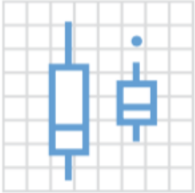
d + geom_bar()
x, alpha, color, fill, linetype, size, weight

discrete x , continuous y

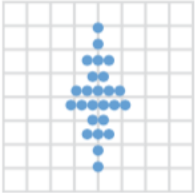
f <- ggplot(mpg, aes(class, hwy))



f + geom_col(), x, y, alpha, color, fill, group, linetype, size



f + geom_boxplot(), x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

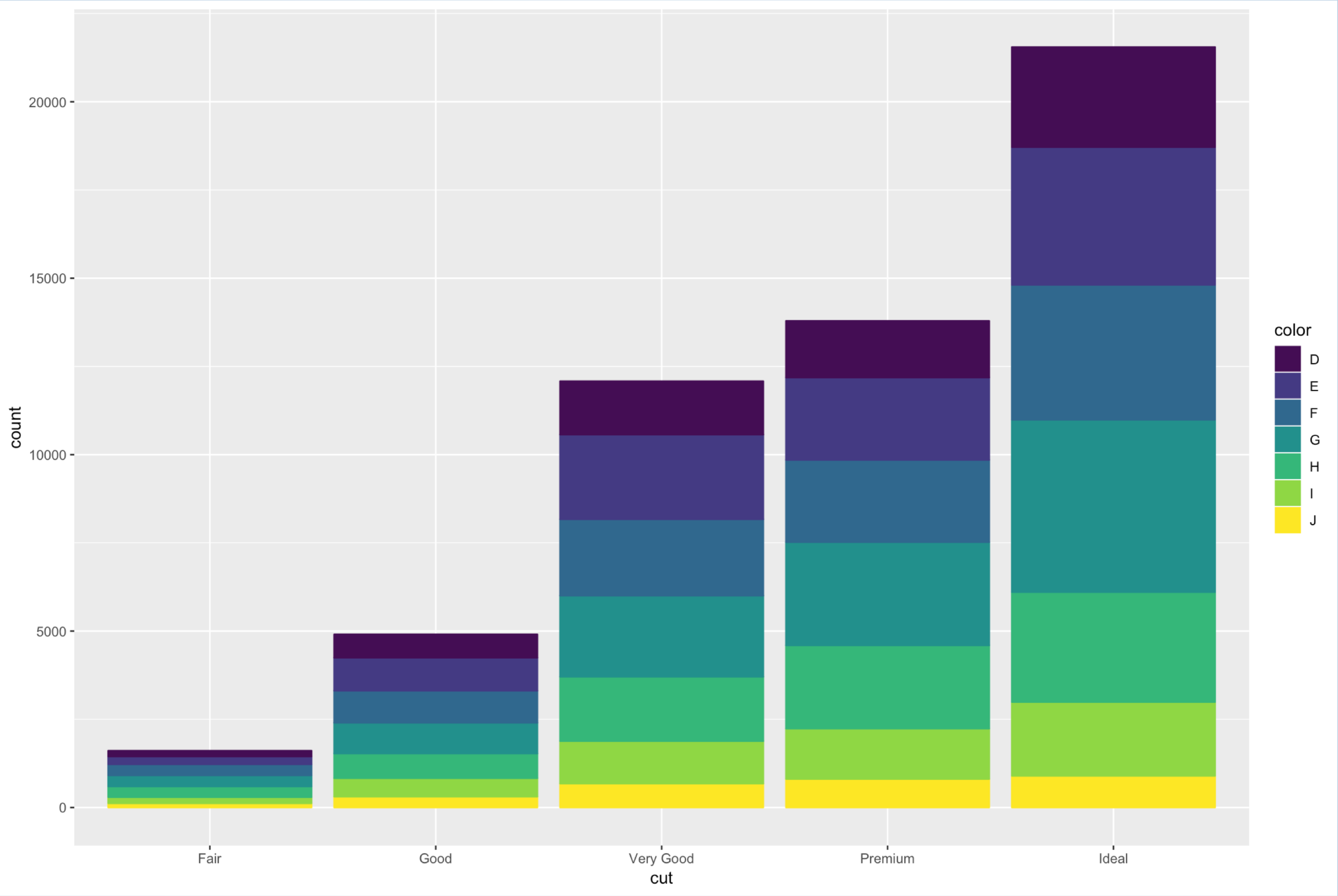


f + geom_dotplot(binaxis = "y", stackdir = "center"**)**, x, y, alpha, color, fill, group

f + geom_violin(scale = "area"**)**, x, y, alpha, color, fill, group, linetype, size, weight

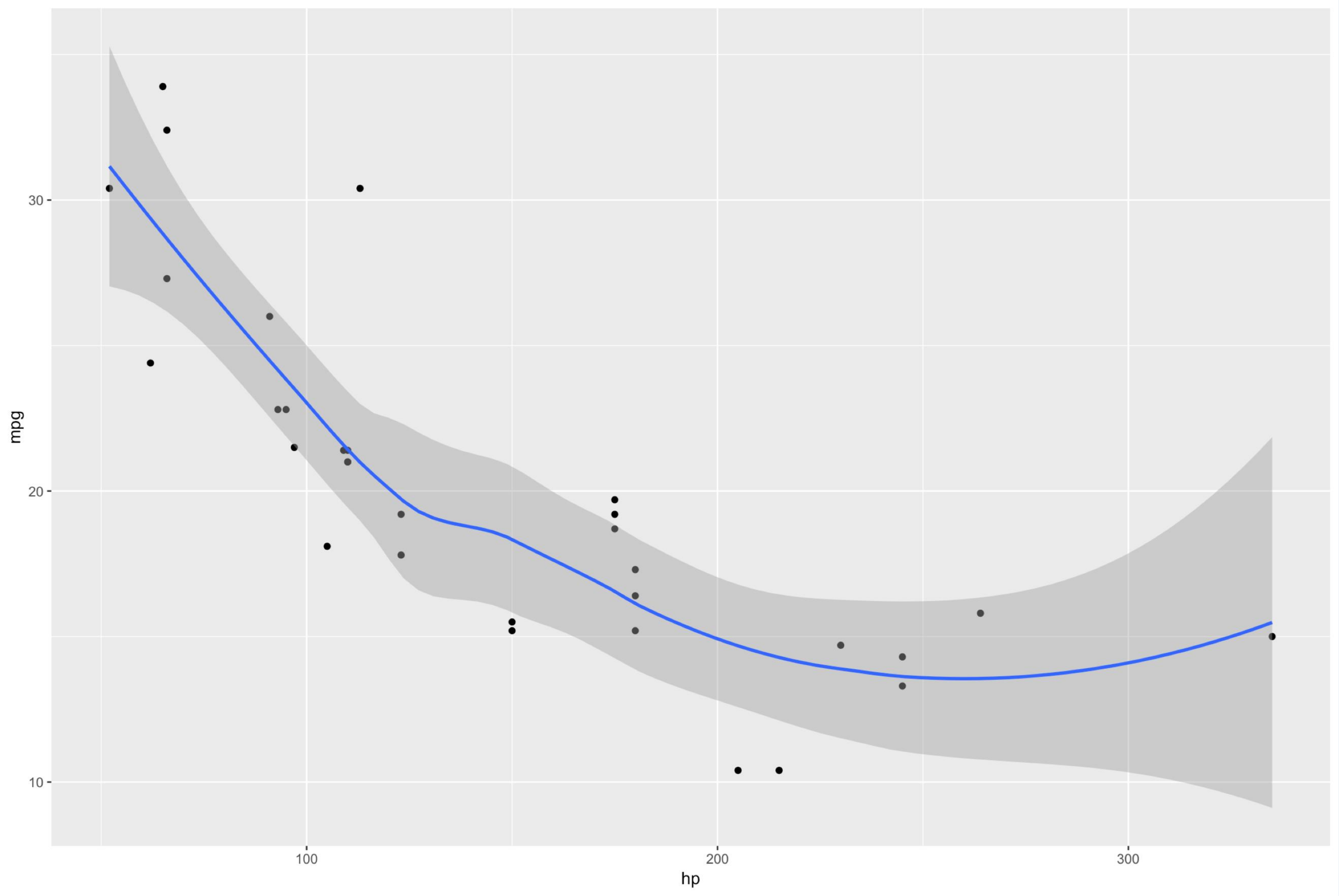
DEMO

```
ggplot(diamonds, aes(x=cut)) +
  geom_bar()
```


20000
15000
10000
5000
0
count
Fair
Good
Very Good
Premium
Ideal
cut
color
D
E
F
G
H
I
J

DEMO

```
ggplot(mtcars, aes(x = hp, y = mpg)) +
  geom_point() +
  geom_smooth()
```



30
20
10
mpg
100
200
300
hp

EXERCISE

- Build any plot
  - Use any dataset

3m 00s

DEMO

Are there any other plots from the markdown file you would like to see?

https://colorbrewer2.org/

plotly

ggplotly

DEMO

```
ggplotly(
    ggplot(mtcars, aes(x = hp, y = mpg)) +
      geom_point() +
      geom_smooth()
 )
```

DEMO

```r
ggplotly(
    ggplot(mtcars, aes(x = hp, y = mpg) +
      geom_point() +
      ggtitle("Car Miles per Gallon by Horse Power") +
      xlab("Horse Power") +
      ylab("MPG"),
    tooltip = c("x", "y")
)
```

DEMO

```
ggplotly(
    ggplot(mtcars, aes(x = hp, y = mpg, text = paste("<b>", rowname, "</b><br>",
      "MPG:", mpg, "<br>Horse Power:", hp))) +
      geom_point() +
      ggtitle("Car Miles per Gallon by Horse Power") +
      xlab("Horse Power") +
      ylab("MPG"),
    tooltip = "text"
)
```

EXERCISE

- Wrap your previous ggplot
  - Create a tooltip
  - Run the code and mess around with the user options in plotly

5m 00s

OTHER FEATURES IN PLOTLY

- Users can save their output by clicking the camera
- You can set tooltip popup type ahead of time
  - ie: `%>% layout(hovermode = 'compare')`
- You can even combine two plots
  - ie: `subplot(plot1, plot2, nrows = 2, shareX = TRUE, heights = c(.7, .3))`
- Plotly will recognize most ggplot arguments, but some don't always come over easily.

plotly standalone

Line Plots

```
plot_ly (
  x = c( 1, 2, 3 ),
  y = c( 5, 6, 7),
  type = 'scatter' ,
  mode = 'lines' )
```

Bubble Charts

```
plot_ly (
  x = c( 1, 2, 3 ),
  y = c( 5, 6, 7 ),
  type = 'scatter' ,
  mode = 'markers' ,
  size = c( 1, 5, 10 ),
  marker = list(
     color = c( 'red', 'blue' ,
     'green' )))
```

Scatter Plots

```
plot_ly (
  x = c( 1, 2, 3 ),
  y = c( 5, 6, 7 ),
  type = 'scatter' ,
  mode = 'markers' )
```

Heatmaps

```
plot_ly (
  z = volcano ,
  type = 'heatmap' )
```

Bar Charts

```
plot_ly (
  x = c( 1, 2, 3),
  y = c( 5, 6, 7),
  type = 'bar' ,
  mode = 'markers' )
```

Area Plots

```
plot_ly (
  x = c( 1, 2, 3 ),
  y = c( 5, 6, 7 ),
  type = 'scatter' ,
  mode = 'lines' ,
  fill = 'tozeroy' )
```

Histograms

```
x <- rchisq ( 100, 5, 0 )
  plot_ly (
  x = x ,
  type = 'histogram' )
```

Box Plots

```
plot_ly (
  y = rnorm( 50 ) ,
  type = 'box' ) %>%

add_trace( y = rnorm( 50, 1 ))
```

2D Histogram

```
plot_ly (
  x = rnorm( 1000, sd = 10 ) ,
  y = rnorm( 1000, sd = 5 ) ,
  type = 'histogram2d' )
```

3D Surface Plots

```
# Using a dataframe:
plot_ly (
  type = 'surface' ,
  z = ~volcano )
```

3D Line Plots

```
plot_ly (
  type = 'scatter3d' ,
  x = c( 9, 8, 5, 1 ) ,
  y = c( 1, 2, 4, 8 ) ,
  z = c( 11, 8, 15, 3 ) ,
  mode = 'lines' )
```

3D Scatter Plots

```
plot_ly (
  type = 'scatter3d' ,
  x = c( 9, 8, 5, 1 ) ,
  y = c( 1, 2, 4, 8 ) ,
  z = c( 11, 8, 15, 3 ) ,
  mode = 'markers' )
```

plotly

in an app

DEMO

Go to old_faithful.R

EXERCISE

- Go to movies_20.R
  - Edit the UI and Server functions so that scatter plot uses plotly
  - Create a useful tooltip

5m 00s

SOLUTION

- Check your app against movies_21.R
- Add library(plotly)
  - Change plotOutput to plotlyOutput
    - Change renderPlot to renderPlotly
    - Wrap the plot in ggplotly()
      - Place tooltip as a paste function in aes() and set tooltip = "text"
      - Or set tooltip = c("x", “y")

Shiny from posit

DT

Advanced

DT OPTIONS AND FUNCTIONS

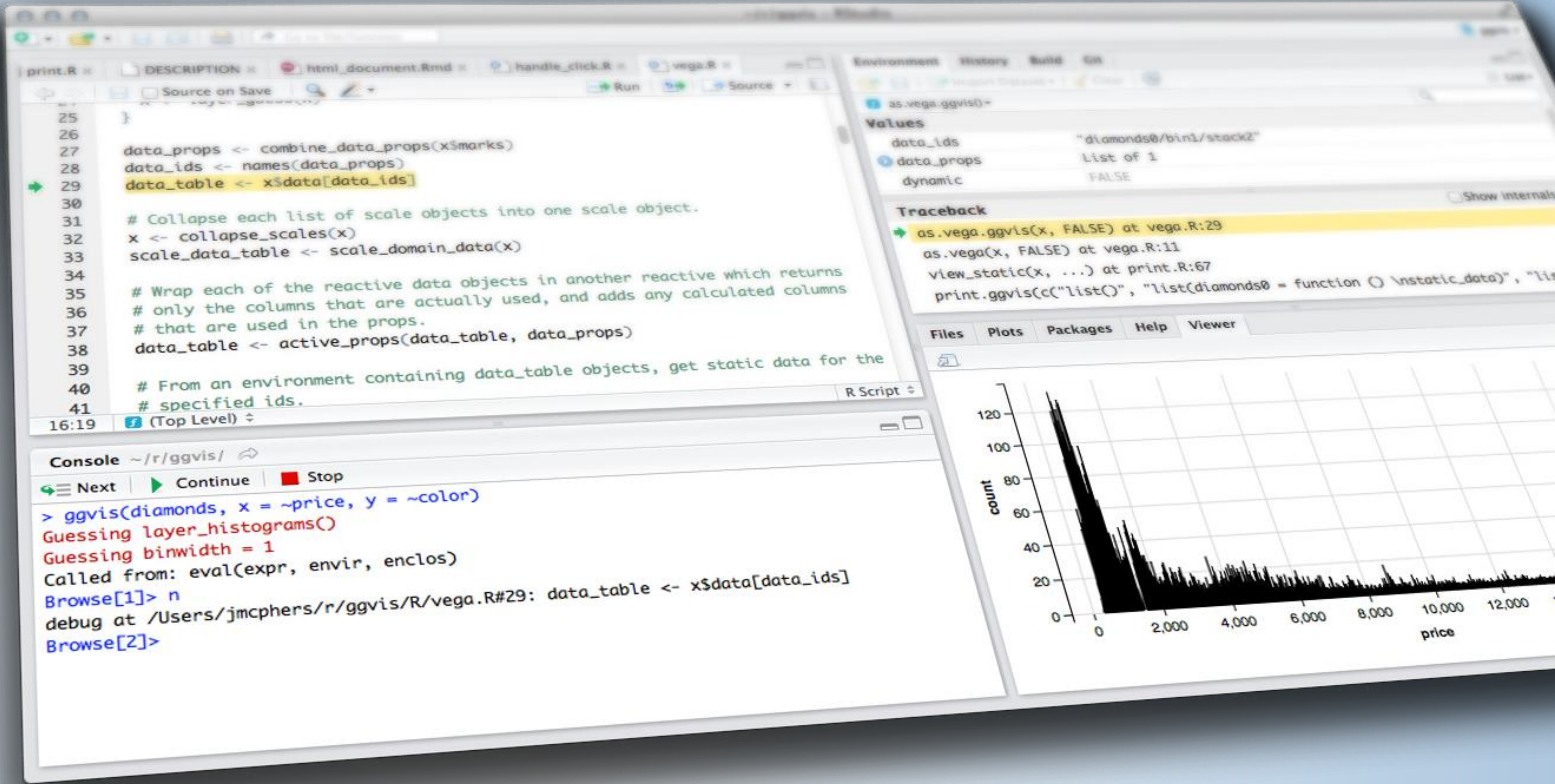
- https://rstudio.github.io/DT/options.html
  - Determine which elements of the table to display (dom)
  - Scrolling
  - Selection
  - Pages
  - etc.
- https://rstudio.github.io/DT/functions.html
  - Format Rows based off values
  - Set row format
  - etc.

DT EXTENSIONS

- https://rstudio.github.io/DT/extensions.html
- Buttons
  - Download data (only works on data displayed)
- Fixed column/header
  - Keeps certain columns or header from scrolling
- Scroller

DEMO

Go to movies_22.R



ADVANCED REACTIVITY

Shiny from posit

Reactivity

catalog

REACTIVITY CATALOG

- Store values: `reactiveValues` / `input` / `makeReactiveBinding`
- Calculate values: `reactive` / `eventReactive`
- Execute tasks: `observe` / `observeEvent`
- Preventing reactivity: `isolate`
- Checking preconditions: `req`
- Time (as a reactive source): `invalidateLater` / ~~`reactiveTimer`~~ *(invalidateLater is a safer and simpler alternative)*
- Rate-limiting: `debounce` / `throttle`
- Live data: `reactiveFileReader` / `reactivePoll`

(Pretty sure this is just the beginning...)

REACTIVITY CATALOG

- Store values: reactiveValues / input / makeReactiveBinding
- Calculate values: reactive / eventReactive
- Execute tasks: observe / observeEvent
- Preventing reactivity: isolate
- Checking preconditions: req
- Time (as a reactive source): invalidateLater / reactiveTimer *(invalidateLater is a safer and simpler alternative)*
- Rate-limiting: debounce / throttle
- Live data: reactiveFileReader / reactivePoll

(Pretty sure this is just the beginning…)

Highlighted functions are fundamental, all others are built on top.

Reactivity

review

REVIEW: REACTIVE EXPRESSIONS

- Use to calculate new values based on reactive values and other reactive expressions.
- Caches its return value, until notified of reactive dependencies being out-of-date.
- Lazily executes — Shiny wants to avoid running these whenever possible. For this reason, meaningful side effects are prohibited from reactive expressions.
- Call it like a function when you want to read its value.

REVIEW: REACTIVE EXPRESSIONS

```
# Declare
movies_subset <- reactive({
  movies %>% filter(title_type %in% input$type)
})

# Read
output$scatterplot <- renderPlot({
  ggplot(movies_subset(), aes(...)) + geom_point()
})
```

REVIEW: OBSERVERS

- Use to execute actions based on changing reactive values and other reactive expressions.
- Doesn't return a value. So performing side effects is usually the only reason you'd want to create one of these.
- Eagerly executed by Shiny.

```
observe({
  print(paste("The value of x is", input$x))
})

## [1] The value of x is 10
## [1] The value of x is 16
## [1] The value of x is 9
```

REACTIVE EXPRESSIONS VS. OBSERVERS

reactive()	observer()
Callable	Not callable
Returns a value	No return value
Lazy	Eager
Cached	N/A
No side effects	Only for side effects

REACTIVE EXPRESSIONS VS. OBSERVERS VS. FUNCTIONS

reactive()	observer()	function()
Callable	Not callable	Callable
Returns a value	No return value	Returns a value
Lazy	Eager	Lazy
Cached	N/A	Not cached
No side effects	Only for side effects	Side effects optional

OBSERVEEVENT VS. EVENTREACTIVE

- Every reactive expression or reactive value read by a `reactive()` or `observe()` block automatically becomes a reactive dependency of that reactive expression/observer.
- `observeEvent` and `eventReactive` give us finer control.

```
observeEvent(input$save_button, {
  write.csv(movies_subset(), "movies.csv")
})
```

"When the save_button button is clicked, write the value of `movie_subset` **to disk."** (Don't write to disk automatically when `movie_subset` changes.)

OBSERVEEVENT AND EVENTREACTIVE

- observeEvent is for event handling
- eventReactive is for delayed computation

```
observeEvent(when_this_changes, {
  do_this
})

r <- eventReactive(when_this_changes, {
  recalculate_this
})
```

Use these functions when you want to **explicitly name your reactive dependencies**, as opposed to letting reactive/observe implicitly depend on anything they read.

EXERCISE

- Open `cranlogs.R` and run it. This app has several problems:
  - We get an error right off the bat — the plot is running before the user has specified any packages.
  - Unless you're a very fast typist, typing package names will cause the cranlogs server to be queried with many incomplete queries.
  - Add an "Update" actionButton to the UI, and make sure nothing happens until it's clicked.

5m 00s

SOLUTION

See cranlogs-solution.R

```
packages <- reactive({
  strsplit(input$packages, " *, *")[[1]]
})
```

```
packages <- eventReactive(input$update, {
  strsplit(input$packages, " *, *")[[1]]
})
```

REVIEW: REACTIVE VALUES

- Read/write versions of input.
- Try not to use this to store *calculated* values. But in some cases, it's unavoidable.

```
# Create
rv <- reactiveValues(x = 10)

# Read
rv$x

# Write
rv$x <- 20
```

EXERCISE

- Open the file `counter.R`. It has three action buttons:
  - Increment: Increase the value by 1
  - Decrement: Decrease the value by 1
  - Reset: Set the value to 0
- Unfortunately, it doesn't work. See if you can implement the server side.

5m 00s

SOLUTION

See counter-solution.R

```
rv <- reactiveValues(count = 0)

observeEvent(input$increment, {
  rv$count <- rv$count + 1
})
observeEvent(input$decrement, {
  rv$count <- rv$count - 1
})
observeEvent(input$reset, {
  rv$count <- 0
})

output$value <- renderText({
  rv$count
})
```

WHEN TO USE REACTIVEVALUES

- Don't use `reactiveValues` when you're calculating a value based on other values and calculations that are already available to you.
- Do use `reactiveValues` to store state that otherwise would be lost from your graph of reactive objects.

REACTIVEVALUES EXAMPLE 1

(1) A calculation over the history of something reactive:

```
observeEvent(input$add, {
  rv$total <- rv$total + input$x
})
```

(Or a more elegant way to do the same, using hadley/shinySignals:)

```
total <- shinySignals::reducePast(reactive(input$x), `+`, 0)
```

REACTIVEVALUES EXAMPLE 2

(2) Tracking which of several events happened most recently:

```
observeEvent(input$editMode, {
  rv$mode <- "edit"
})

observeEvent(input$previewMode, {
  rv$mode <- "preview"
})

output$page <- renderUI({
  if (rv$mode == "edit") {
    ...
  } else if (rv$mode == "preview") {
    ...
  }
})
```

REACTIVEVALUES EXAMPLE 3

(3) To change rules of reactivity:

- Normally, as soon as reactive expressions are invalidated (before they have recalculated) they invalidate everyone downstream who depends on them.
- But sometimes recalculating will end up giving us the same value as the previous anyway, and any downstream recalculations might have been wasted work.

```
dedupeReactive <- function(rexpr, priority = 10) {
  rv <- reactiveValues(value = NULL)

  observe({
    rv$value <- rexpr()  # TODO: Handle errors
  }, priority = priority)

  reactive(rv$value)
}
```

PREVENTING REACTIVITY WITH ISOLATE

- Use `isolate` from inside a reactive expression or observer, to ignore the implicit reactivity of a piece of code.
- Wrap it around expressions or a whole code block.

EXERCISE

- Determine when r1, r2, and r3 update:

```
r1 <- reactive({
  input$x * input$y
})


r2 <- reactive({
  input$x * isolate({ input$y })
})


r3 <- reactive({
  isolate({ input$x * input$y })
})
```

SOLUTION

```r
# Updates every time input$x or input$y change
r1 <- reactive({
  input$x * input$y
})

# Updates only when input$x changes
r2 <- reactive({
  input$x * isolate({ input$y })
})

# Never updates; it will always have its original value
r3 <- reactive({
  isolate({ input$x * input$y })
})
```

Checking preconditions

CHECKING PRECONDITIONS WITH REQ

- Cancel the current output (or observer) if a condition isn't met.
  - `req(input$text)`: Ensure the user has provided a value for the "text" input
  - `req(input$button)`: Ensure the button has been pressed at least once
  - `req(x %% 2 == 0)`: Ensure that x is an even number
  - `req(FALSE)`: Unconditionally cancel the current reactive, observer, or output

CHECKING PRECONDITIONS WITH REQ

- `req(cond)` is similar to:
  - `stopifnot(cond)`
  - `if (!cond) stop()`
  - `assertthat::assert_that(cond)`
- but with these differences:
  - Errors during output rendering show up with bold red text in the UI; `req` just makes the output blank
  - Rather than verifying that `cond` is *true*, `req` verifies that `cond` is *truthy* (see `?isTruthy`)
    - Feels unnatural to be so arbitrary and nebulous, but this definition is just too practical for UI programming
  - Most importantly, `req` is like an error in that it "infects" the downstream elements of the reactive graph (if a reactive throws an error, then any other reactive/observer/output that tries to access it will also throw an error)

EXERCISE

- Open `dynamic.R` and run it.
- It has lots of errors in the browser and the R console — ignore those for the moment.
- From the app, upload the `diamonds.csv` file found in the same directory. Now everything looks good.
- See if you can figure out why these errors appear when the app first comes up, and how you can get them to go away (first without `req`, and then, if you have time and can figure out how, using `req`).

5m 00s

SOLUTION

- Antisolution:dynamic-antisolution.R.
  - This is how you used to have to do it: check for missing values yourself, and return(NULL).
  - You had to do this in every reactive, observer, or output that could have a missing value, plus all of the reactives, observers, and outputs that are downstream!
- Solution: dynamic-solution.R.
  - Now you can use req in the reactives, observers, and outputs that directly use potentially-missing inputs, and everything downstream can just not worry about it.

Time as a reactive source

EXERCISE

- What will this prouduce?

```
ui <- basicPage( verbatimTextOutput(“text") )

server <- function(input, output){

  r <- reactive({ Sys.time() })
  output$text <- renderPrint({ r() })

}

shinyApp(ui, server)
```

An app that reports Sys.time() at the time of first launch, and then doesn’t update it

EXERCISE

- What will this prouduce?

```
ui <- basicPage( verbatimTextOutput(“text") )

server <- function(input, output){

  r <- reactive({
    invalidateLater(1000)
    Sys.time()
  })
  output$text <- renderPrint({ r() })

}

shinyApp(ui, server)
```

An app updates reported Sys.time() every second

Limiting

rate

DEBOUNCE AND THROTTLE

- If a reactive value or expression changes too fast for downstream calculations to keep up, you can end up with a bad user experience (laggy experience, wasted work).
  - debounce and throttle take a reactive expression object as input, and return a rate-limited version of that reactive expression.

```
# A reactive that updates as often as every 50 milliseconds
fast_reactive <- reactive({ ... })

# A reactive that updates no more often than every 2000 milliseconds
throttled_reactive <- throttle(fast_reactive, 2000)

# A reactive that doesn't update until fast_reactive has stopped
# changing for at least 1000 milliseconds
debounced_reactive <- debounce(fast_reactive, 1000)
```

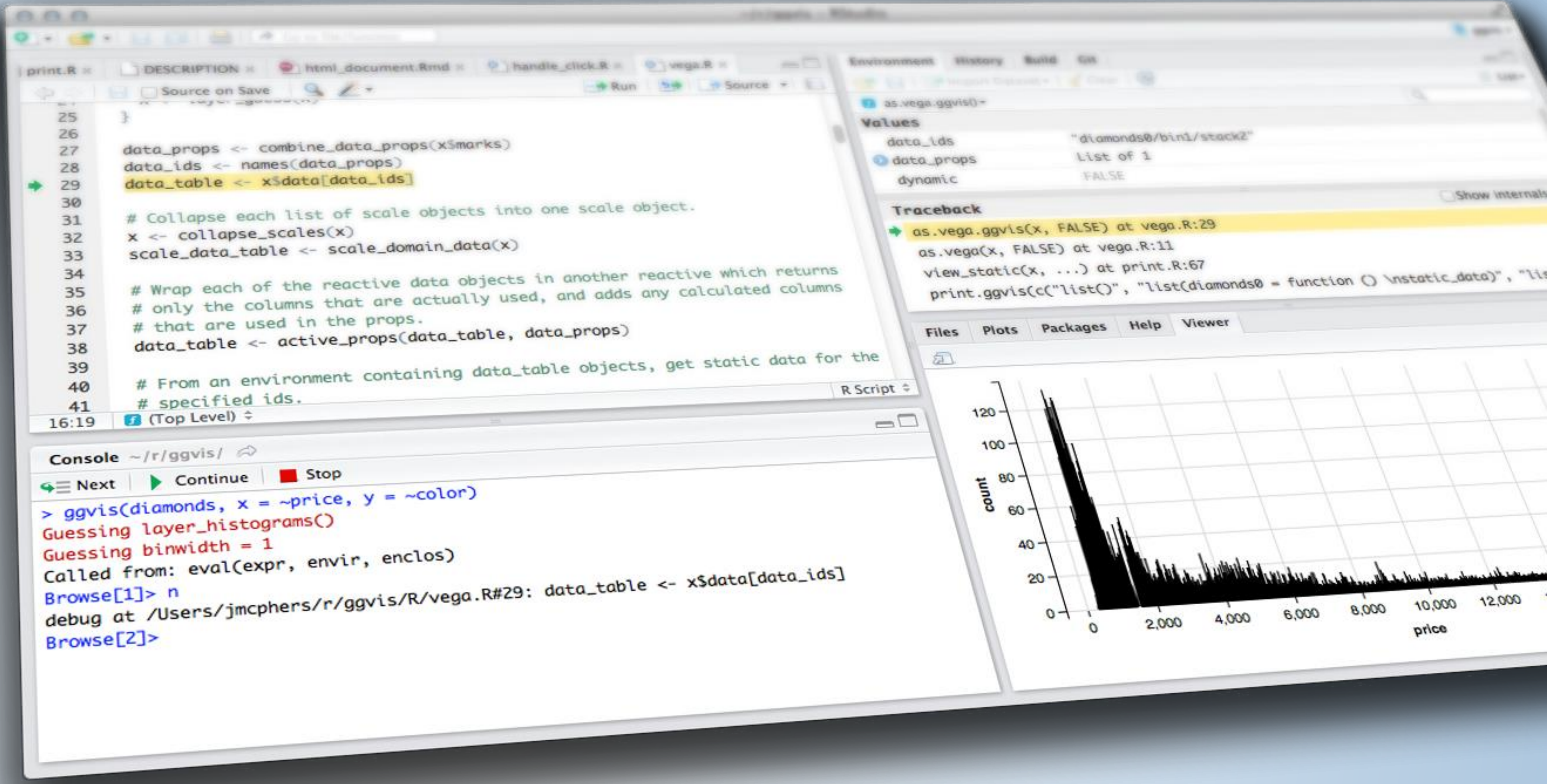
EXERCISE

- Open and run `points.R`. Click on the plot a few times to create points. Notice the annoying laggy behavior — this is due to a (simulated) expensive summary output.
- Use `debounce` or `throttle` to prevent the summary output from running so often.

5m 00s

SOLUTION

See points-solution.R



ADVANCED REACTIVITY

Shiny from posit