```r
}

data_props <- combine_data_props(x$marks)
data_ids <- names(data_props)
data_table <- x$data[data_ids]

# Collapse each list of scale objects into one scale object.
x <- collapse_scales(x)
scale_data_table <- scale_domain_data(x)

# Wrap each of the reactive data objects in another reactive which returns
# only the columns that are actually used, and adds any calculated columns
# that are used in the props.
data_table <- active_props(data_table, data_props)

# From an environment containing data_table objects, get static data for the
# specified ids.
```

```
> ggvis(diamonds, x = ~price, y = ~color)
Guessing layer_histograms()
Guessing binwidth = 1
Called from: eval(expr, envir, enclos)
Browse[1]> n
debug at /Users/jmcphers/r/ggvis/R/vega.R#29: data_table <- x$data[data_ids]
Browse[2]>
```

Values

data_ids	"diamonds0/bin1/stack2"
data_props	List of 1
dynamic	FALSE

Traceback

```
as.vega.ggvis(x, FALSE) at vega.R:29
as.vega(x, FALSE) at vega.R:11
view_static(x, ...) at print.R:67
print.ggvis(c("list()", "list(diamonds0 = function () \nstatic_data)", "li
```

MODULES

Shiny from R Studio

OUTLINE

- What is a Shiny module?
- Anatomy of a Shiny module
  - User interface
  - Server function
  - Calling the module
  - Packaging modules
- Exercise: Modularize this!
- Combining modules

What is a Shiny module?

MODULES

- A module is a self-contained, composable component of a Shiny app
  - self-contained like a function
  - can be combined to make an app
- Have their own UI and server (in addition to the app UI and server)

MODULES

- Useful for reusability
  - rather than copy and paste code, you can use modules to help manage the pieces that will be repeated throughout a single app or across multiple apps
  - can be bundled into packages
- Essential for managing code complexity in larger apps

LIMITATIONS TO JUST FUNCTIONALIZING

- It’s possible to write UI-generating functions and call them from your app’s UI, and you write functions for the server that define outputs and create reactive expressions
- However you must make sure your functions generate input and output IDs that don’t collide since input and output IDs in Shiny apps share a global namespace, meaning, each ID must be unique across the entire app
- **Solution:** Namespaces! Modules add namespacing to Shiny UI and server logic

~~*“The thing to keep in mind about defining shiny UI, is that the main tool for managing complexity is the same tool as everywhere else in R:* ***the function****. Since Shiny user interfaces are defined using function calls, take advantage of that fact by reusing and encapsulating UI creation logic into functions. The same can be said for defining server logic--write functions, and when those functions get too complicated, split them into more functions.”*~~

–Joe Cheng, 7/9/2015

“Roughly, hygienic macro expansion is desirable for the same reason as lexical scope: both enable local reasoning about binding so that program fragments compose reliably.”

–Matthew Flatt

Shiny Modules

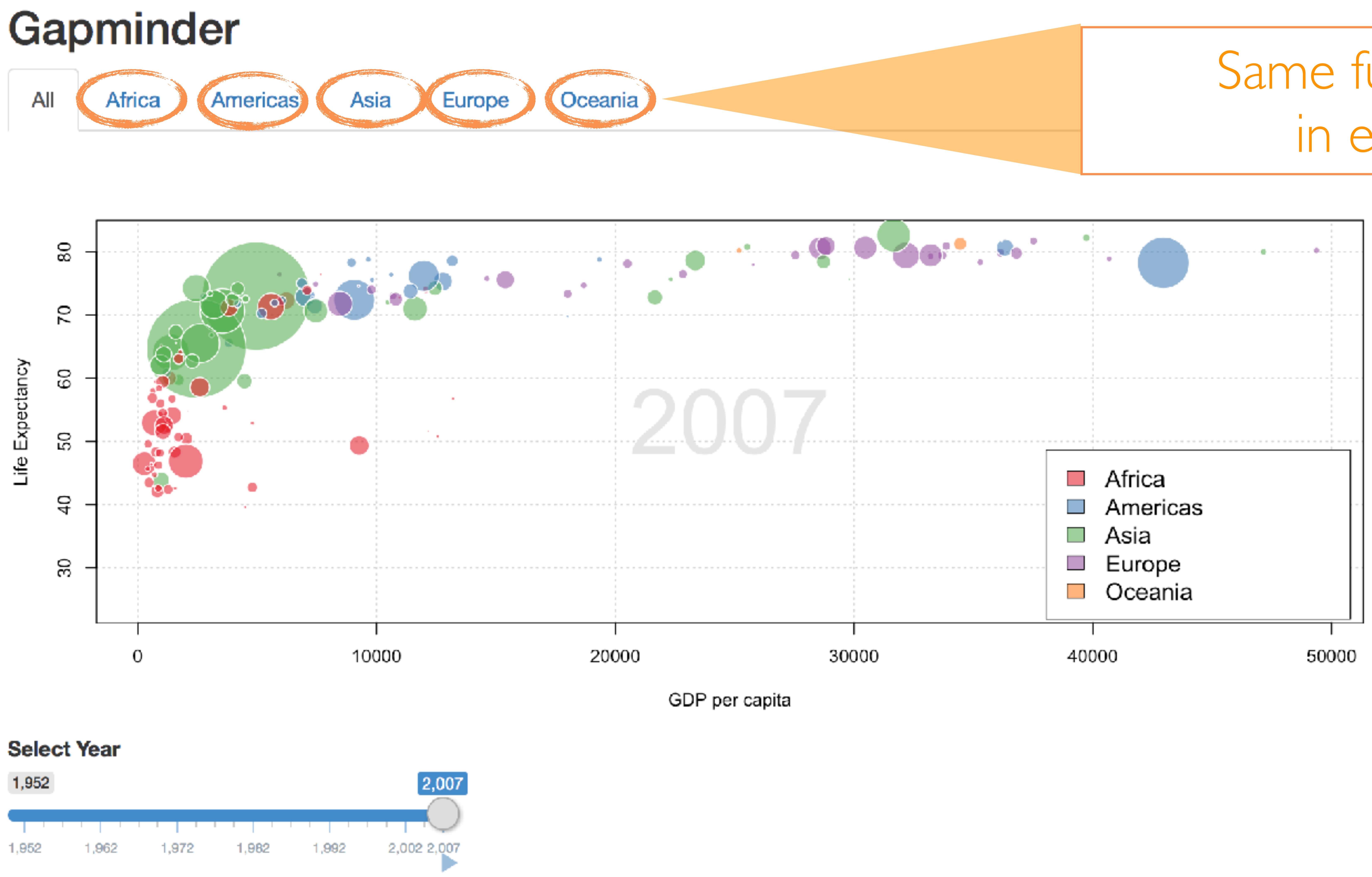
“Roughly, ~~hygienic macro expansion~~ is desirable for the same reason as lexical scope: both enable ***local reasoning*** *about binding so that program fragments* ***compose reliably****.”*

–Matthew Flatt

gapminder_app.R

DEMO

Same functionality in each tab



LADDER OF PROGRESSION

- Level 1. Use modules to break large monolithic apps into manageable pieces
- Level 2. Create reusable modules
- Level 3. Combine/nest modules

Anatomy of a Shiny module

WHAT'S IN A MODULE?

```
library(shiny)
name_of_module_UI <- function(id, …) {
  # Create a namespace function using the provided id
  ns <- NS(id)
  # UI elements go here
  tagList(
    …
  )
}

name_of_module <- function(input, output, session, …) {
  # Server logic goes here
}
```

User interface controls the layout and appearance of module

Server function contains instructions needed to build module

SHINYMOD SNIPPET

- Use this RStudio snippet to do the boilerplate for you: http://bit.ly/2rEEtUy
- Add snippet using: Preferences | Code | Edit Snippets
- Use snippet by typing **`shinymod`** [Tab], then type the name of your new module

User interface

MODULE UI

- A function
- Takes, as input, an id that gets pre-pended to all HTML element ids with a helper function: **NS()**
- Can also have additional parameters

```
name_of_module_UI <- function(id, …) {
  # Create a namespace function using id
  ns <- NS(id)
  # UI elements go here
  tagList(
    …
  )
}
```

MODULE UI

- In module UI, you **must** wrap all input and output IDs with **`ns()`**

```
selectInput("x", "X variable", …)

DT::dataTableOutput("rawdata")

plotOutput("plot", click = "click")
```

```
selectInput(ns("x"), "X variable", …)

DT::dataTableOutput(ns("rawdata"))

plotOutput(ns("plot"), click = ns("click"))
```

- I guarantee you will often forget to do this! If your module doesn't work as expected, double-check this first!

Server function

MODULE SERVER

- Includes the code needed for your module
- Looks almost identical to the app server function, except that you may have additional parameters
- App server function is automatically invoked by Shiny; module server function must be invoked by the app author

```
name_of_module <- function(input, output,
    session, …) {

  # Server logic goes here

}
```

MODULE VS. APP

- Similarities:
  - Inputs in module UI can be accessed in module server with input$
  - Outputs in module UI can be defined in module server with output$
  - Can create reactive expressions, observers
- Differences:
  - Inputs/outputs cannot be directly accessed from outside the module namespace
  - Module server function can take additional parameters, and can return a value to the caller

How it works

HOW IT WORKS

- All UI input/output controls have IDs that are prefixed with "*namespace-*".
- The server module function is passed special clones of **`input`**, **`output`**, and **`session`** that implicitly prefixes all IDs with "*namespace-*".

```
# in module UI
numericInput(ns("rows"), "Row count", 5),
plotOutput(ns("plot"))

# in module server
output$plot <- renderPlot({
  plot(head(data, input$rows))
})
```

HOW IT WORKS

- All UI input/output controls have IDs that are prefixed with "*namespace-*".
- The server module function is passed special clones of **`input`**, **`output`**, and **`session`** that implicitly prefixes all IDs with "*namespace-*".

```
# in module UI
numericInput(ns("rows"), "Row count", 5),
plotOutput(ns("plot"))

# in module server
output$plot <- renderPlot({
  plot(head(data, input$rows))
})
```

"mymod-rows"

"mymod-plot"

output$`mymod-plot`

input$`mymod-rows`

Calling the module

CALLING THE MODULE IN YOUR APP

- In the app UI:
  - Include the module UI with `name_of_module_UI("id", …)`
  - Can also include other UI elements that are not included in the module
- In the app server:
  - Include the module server with `callModule(name_of_module, "id", …)`
    - DO NOT pass `input`, `output`, `session` to `callModule`
  - Can define outputs that are not included in the module
- The id must match and must be unique among other inputs/outputs/modules at the same "scope" (either top-level ui/server, or within a parent Shiny module)

DEMO

gapminder.R

ui:

```
ui <- fluidPage(
  …
  titlePanel("Gapminder"),
  tabsetPanel(id = "continent",
    tabPanel("All", gapModuleUI("all")),
    tabPanel("Africa", gapModuleUI("africa")),
    tabPanel("Americas", gapModuleUI("americas")),
    tabPanel("Asia", gapModuleUI("asia")),
    tabPanel("Europe", gapModuleUI("europe")),
    tabPanel("Oceania", gapModuleUI("oceania"))
  )
)
```

server:

```
server <- function(input, output) {
  callModule(gapModule, "all", all_data)
  callModule(gapModule, "africa", africa_data)
  callModule(gapModule, "americas", americas_data)
  callModule(gapModule, "asia", asia_data)
  callModule(gapModule, "europe", europe_data)
  callModule(gapModule, "oceania", oceania_data)
}
```

Communicating with modules

PASSING INPUT TO MODULES

- Sometimes, modules need to be informed by their callers
  - Access to simple parameter values
  - Access to reactive expressions
  - Access to inputs that are outside the module
- You can accomplish this by adding parameters to your module server function (and callModule calls)

PASSING INPUT TO MODULES

- Example: Passing by value (static)

```r
# Module server function
country_module <- function(input, output, session, country) {
  data <- countries %>% filter(Country == country)
}

# In app server function
callModule(country_module, id = "us", country = "United States")
```

PASSING INPUT TO MODULES

- Example: Passing a reactive expression (changes over time)
  - Pass reactives by reference (no parens) to preserve reactivity!

```
# Module server function
country_module <- function(input, output, session, selected_country) {
  data <- reactive({
    countries %>% filter(Country = selected_country())
  })
}

# In app server function
country <- reactive({ ... })
callModule(country_module, id = "us", selected_country = country)
```

No parens!

PASSING INPUT TO MODULES

- Example: Passing a reactive input
  - Wrap in a new reactive expression

```r
# Module server function
country_module <- function(input, output, session, selected_country) {
  data <- reactive({
    countries %>% filter(Country = selected_country())
  })
}

# In app server function
callModule(country_module, id = "us", country = reactive(input$country))
```

RETURNING OUTPUT FROM MODULE

- Modules can also return (static or reactive) values back to their callers.
- You can take advantage of this to make modules that represent reusable inputs, or reusable calculations/computations.

RETURNING OUTPUT FROM MODULE

- Example: Returning a reactive result

```r
# Module server function that returns a reactive
csv_upload <- function(input, output, session) {
  r <- reactive({
    req(input$file)
    read.csv(input$file$datapath)
  })
  r
}

# In app server function
upload <- callModule(csv_upload, "upload")
output$table <- renderTable({ upload() })
```

RETURNING OUTPUT FROM MODULE

- If a module must return an **`input$value`**, wrap in a reactive
- If a module must return multiple values, return a named list

```
csv_upload <- function(input, output, session) {
  df <- reactive({
    req(input$file)
    read.csv(input$file$datapath)
  })

  list(
    data = df,
    file = reactive(input$file),
    size = reactive(file.info(req(input$file$datapath))$length)
  )
}
```

Packaging modules

PACKAGING MODULES

- **Inline code:** Put the UI and server function code of the module directly in your app
  - If using app.R (single-file) syle, include module code in that file, before the app's UI and server logic
  - If using ui.R / server.R (two-file) style, include module code in the global.R file
  - If defining many modules / modules containing a lot of code, may result in bloated global.R/app.R file.
- **Standalone R file:**
  - Save module code in a separate .R file and then call `source("path-to-module.R")` from app.R or global.R
  - Probably the best approach for modules that won't be reused across applications
- **R package:** Useful if your modules are intended to be reused across apps
  - R package needs to export and document your module's UI and server functions
  - R packages can include more than one module

Modularize this!

EXERCISE

- See `movies_20.R` for a simpler version of the movie browser:
  - only select x, y, and z (for colors) variables
  - and alpha level and size of points
  - three tabs: one for each title type, showing a scatterplot and data table
- Note that this app is created by repeating the plotting and data table code chunks three times each
- Modularize the app using `moviesmodule_template.R`

10m 00s

SOLUTION

- App: `movies_21.R`
- Module: `moviesmodule.R`

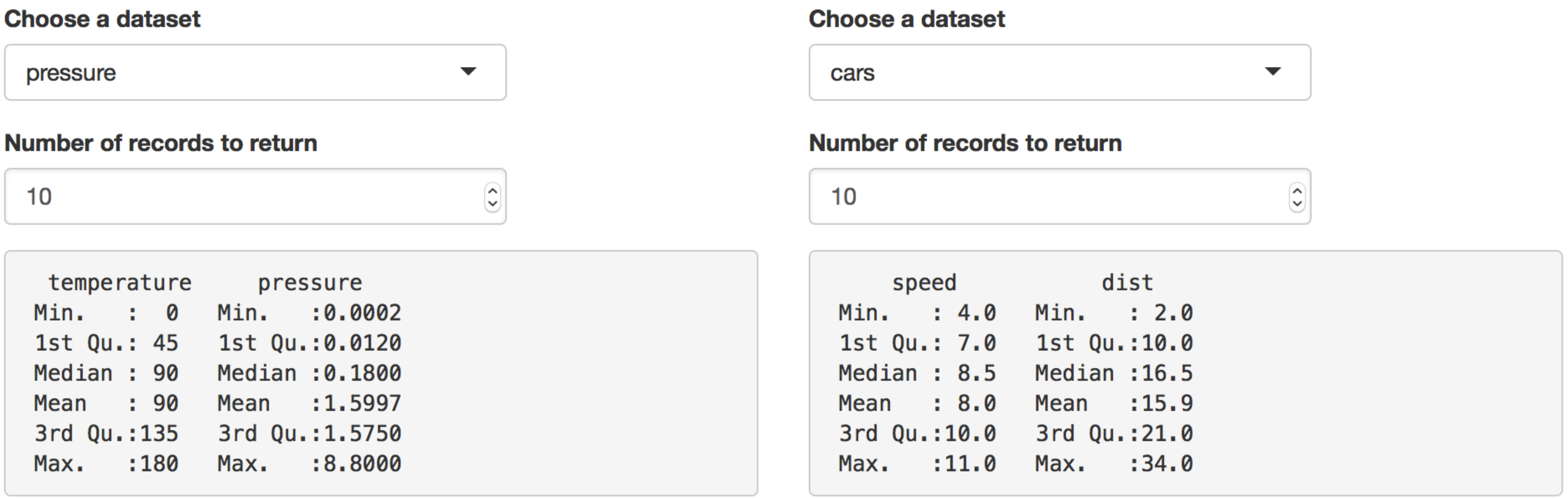
Combining modules

COMBINING MODULES

- When building an app that uses modules that depend on each other, avoid violating the sanctity of the module's namespace (similar to a function's local environment)
- If results of Module 1 will be used as inputs in Module 2, then Module 1 needs to return those results as an output, so that Module 2 does not have to “reach in and grab them”

DEMO

left_right_01.R



Choose a dataset

pressure

Number of records to return

10

```
  temperature      pressure     
 Min.   :  0   Min.   :0.0002  
 1st Qu.: 45   1st Qu.:0.0120  
 Median : 90   Median :0.1800  
 Mean   : 90   Mean   :1.5997  
 3rd Qu.:135   3rd Qu.:1.5750  
 Max.   :180   Max.   :8.8000  
```

Choose a dataset

cars

Number of records to return

10

```
     speed           dist      
 Min.   : 4.0   Min.   : 2.0  
 1st Qu.: 7.0   1st Qu.:10.0  
 Median : 8.5   Median :16.5  
 Mean   : 8.0   Mean   :15.9  
 3rd Qu.:10.0   3rd Qu.:21.0  
 Max.   :11.0   Max.   :34.0  
```

Shiny from RStudio

DEMO

Module 1: Dataset chooser

```r
# Module 1 UI
dataset_chooser_UI <- function(id) {
  ns <- NS(id)

  tagList(
    selectInput(ns("dataset"), "Choose a dataset", c("pressure", "cars")),
    numericInput(ns("count"), "Number of records to return", 10)
  )
}

# Module 1 Server
dataset_chooser <- function(input, output, session) {
  dataset <- reactive({
    req(input$dataset)
    get(input$dataset, pos = "package:datasets")
  })

  return(list(
    dataset = dataset,
    count = reactive(input$count)
  ))
}
```

Why not `dataset()`?

Why wrapped with `reactive()`?

Shiny from RStudio

DEMO

Module 2: Dataset summarizer

```r
# Module 2 UI
dataset_summarizer_UI <- function(id) {
  ns <- NS(id)

  verbatimTextOutput(ns("summary"))
}

# Module 2 Server
dataset_summarizer <- function(input, output, session, dataset, count) {
  output$summary <- renderPrint({
    summary(head(dataset(), count()))
  })
}
```

Shiny from RStudio

App combining the two modules

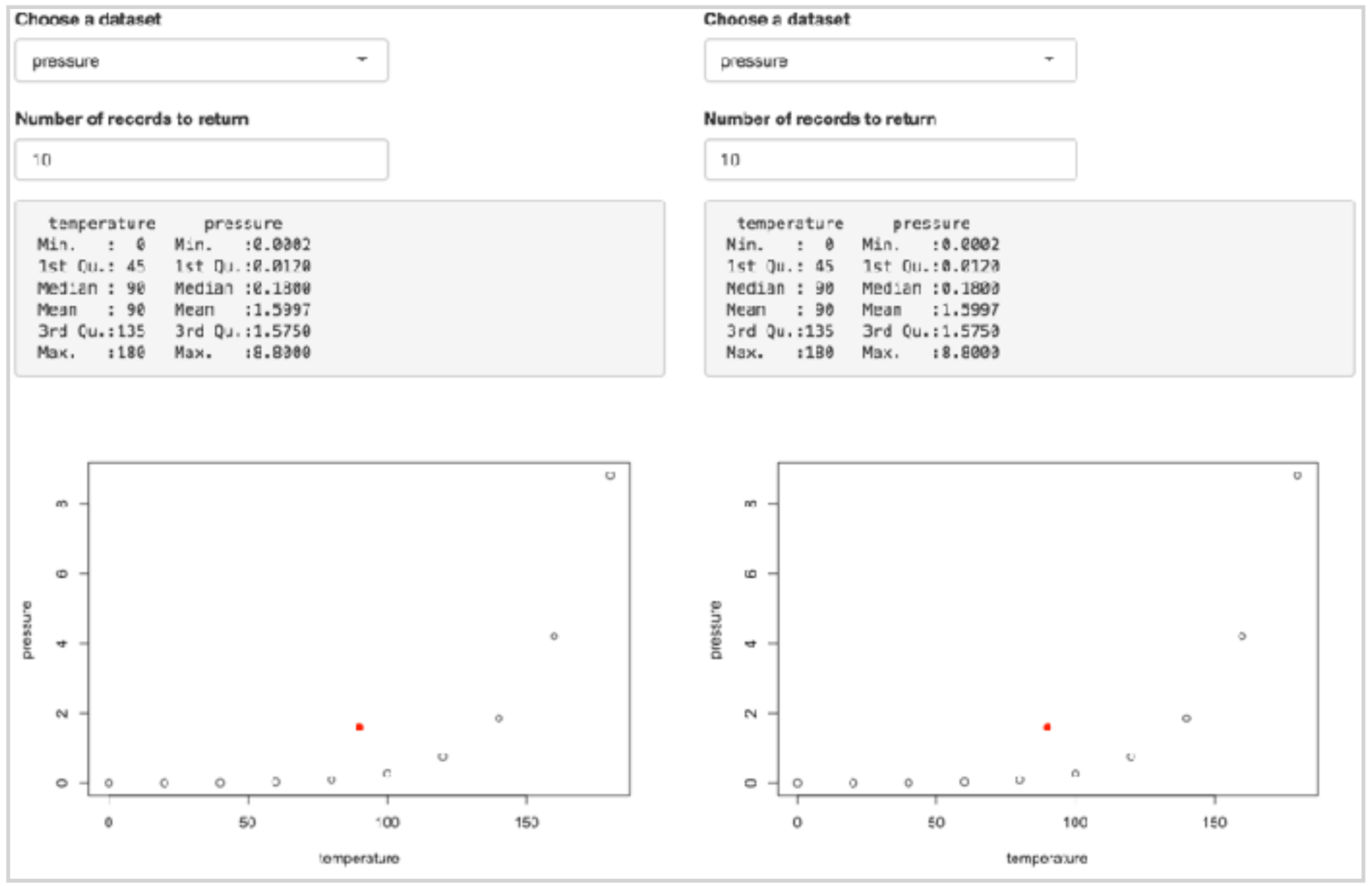
DEMO

```
# App UI
ui <- fluidPage(
  fluidRow(
    column(6,
      dataset_chooser_UI("left_input"),
      dataset_summarizer_UI("left_output")
    ),
    column(6,
      dataset_chooser_UI("right_input"),
      dataset_summarizer_UI("right_output")
    )
  )
)

# App server
server <- function(input, output, session) {
  left_result <- callModule(dataset_chooser, "left_input")
  right_result <- callModule(dataset_chooser, "right_input")

  callModule(dataset_summarizer, "left_output",
             dataset = left_result$dataset, count = left_result$count)
  callModule(dataset_summarizer, "right_output",
             dataset = right_result$dataset, count = right_result$count)
}
```

EXERCISE

- Start with `left_right_01.R`
- Update the 2nd module, dataset summarizer, so that it also returns mean of each of the two variables
- Add a 3rd module, dataset plotter, that makes a scatterplot of the selected dataset, and overlays a red point at (mean of x, mean of y) that it takes from the results of the dataset summarizer
- Add this module into your app such that the plot is printed underneath the summary

Desired outcome:



Choose a dataset: pressure

Number of records to return: 10

```
 temperature     pressure     
 Min.   :  0   Min.   :0.0002  
 1st Qu.: 45   1st Qu.:0.0120  
 Median : 90   Median :0.1800  
 Mean   : 90   Mean   :1.5997  
 3rd Qu.:135   3rd Qu.:1.5750  
 Max.   :180   Max.   :8.8000  
```

Choose a dataset: pressure

Number of records to return: 10

```
 temperature     pressure     
 Min.   :  0   Min.   :0.0002  
 1st Qu.: 45   1st Qu.:0.0120  
 Median : 90   Median :0.1800  
 Mean   : 90   Mean   :1.5997  
 3rd Qu.:135   3rd Qu.:1.5750  
 Max.   :180   Max.   :8.8000  
```

10m 00s

Module 2: Dataset summarizer (updated)

SOLUTION

```
# Module 2 UI
dataset_summarizer_UI <- function(id) {
  ns <- NS(id)

  verbatimTextOutput(ns("summary"))
}

# Module 2 Server
dataset_summarizer <- function(input, output, session, dataset, count) {

  selected_data <- reactive({ head(dataset(), count()) })

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

  mean_x <- reactive({ mean(selected_data()[,1]) })
  mean_y <- reactive({ mean(selected_data()[,2]) })

  return(list(
    mean_x = mean_x,
    mean_y = mean_y
  ))
}
```

Shiny from RStudio

SOLUTION

Module 3: Dataset plotter (new)

```
# Module 3 UI
dataset_plotter_UI <- function(id) {
  ns <- NS(id)

  plotOutput(ns("scatterplot"))
}

# Module 3 Server
dataset_plotter <- function(input, output, session, dataset, count,
                            mean_x, mean_y) {
  output$scatterplot <- renderPlot({
    plot(head(dataset(), count()))
    points(x = mean_x(), y = mean_y(), pch = 19, col = "red")
  })
}
```

Shiny from RStudio

SOLUTION

App combining all three modules

See `left_right_02.R` for details

```
}

data_props <- combine_data_props(x$marks)
data_ids <- names(data_props)
data_table <- x$data[data_ids]

# Collapse each list of scale objects into one scale object.
x <- collapse_scales(x)
scale_data_table <- scale_domain_data(x)

# Wrap each of the reactive data objects in another reactive which returns
# only the columns that are actually used, and adds any calculated columns
# that are used in the props.
data_table <- active_props(data_table, data_props)

# From an environment containing data_table objects, get static data for the
# specified ids.
```

16:19 (Top Level)

```
as.vega.ggvis()
Values
data_ids       "diamonds0/bin1/stack2"
data_props     List of 1
dynamic        FALSE

Traceback
as.vega.ggvis(x, FALSE) at vega.R:29
as.vega(x, FALSE) at vega.R:11
view_static(x, ...) at print.R:67
print.ggvis(c("list()", "list(diamonds0 = function () \nstatic_data)", "li
```

Console ~/r/ggvis/

```
> ggvis(diamonds, x = ~price, y = ~color)
Guessing layer_histograms()
Guessing binwidth = 1
Called from: eval(expr, envir, enclos)
Browse[1]> n
debug at /Users/jmcphers/r/ggvis/R/vega.R#29: data_table <- x$data[data_ids]
Browse[2]>
```

MODULES

Shiny from R Studio