Data Visualisation using RShiny

Laurie Baker, Elaine Ferguson, Will Harvey and Rachel Steenson

Tanzania, August 2019

Course Overview

Day 1:

- 1.1 Getting to know your data
- 1.2 Data subsetting and summarising
- 1.3 Build exploratory plots
- **1.4 Building an interactive plot in RShiny**

Day 2:

- 2.1 Introduction to leaflet
- 2.2 Building a leaflet map in R
- 2.3 Build an interactive map in RShiny

Day 3:

- 3.1 Review
- 3.2 Build your own apps!

Building an interactive plot in RShiny

A shiny app is normally built with 2 sections:

- The **ui** (user interface) contains code for the part of the app that the user sees.
- The **server** contains code for the processing behind the user interface.

These can be contained in one R script, or as multiple R scripts saved in the same folder.

Your turn

Complete section 1.4a of the handout.

Summary - section 1.4a

- The 2 methods are very similar
- It can be easier to separate your code and sections using the ‘multiple file’ layout.

All of our examples and practice sections will use seperate **server.R** and **ui.R** files.

Whenever we show example code, ui code will have a yellow background and server code will have a blue background.

Inputs

- To make the app interactive, you need an **input** that the user can change
- Shiny includes a selection of functions for this called **widgets**.
- These are used in the **ui** to set the options for the user to interact with.

Name	Description
actionButton()	a button click input
checkboxInput()	a single checkbox
checkboxGroupInput()	a set of checkboxes where multiple can be selected
dateInput()	a calendar for date selection
dateRangeInput()	a pair of calendars for start and end date selection
numericInput()	a free-text for numbers
radioButtons()	a set of buttons where only 1 can be selected
selectInput()	a dropdown menu
sliderInput()	a slider bar
textInput()	a free-text for letters/words

Outputs

- An **output** is required to show the user the effects of their **input**
- `render()` is used in the **server** to generate the output
- `output()` is used in the **ui** to display the output to the user

render	output	Produces
renderImage()	imageOutput()	an image
renderPlot()	plotOutput()	a plot/map
renderPrint()	verbatimTextOutput()	any printed output formatted as code
renderTable()	tableOutput()	a table
DT::renderDataTable()	dataTableOutput()	an interactable table
renderText()	textOutput()	any character string formatted to match app
rendeerUI()	uiOutput() or htmlOutput()	any character string formatted with raw HTML code

Using inputs and outputs together

ui

```
# Input
radioButtons(inputId = "radio", label = "Choose a number:", selected = 1,
             choices = list("Choice 1" = 1, "Choice 2" = 2, "Choice 3" = 3))

# Output
verbatimTextOutput("chosen_number")
```

server

```
# Render
output$chosen_number <- renderPrint({
  input$radio
})
```

Using inputs and outputs together

Input: `radioButtons()`

Choose a number:

- (•) Choice 1
- ( ) Choice 2
- ( ) Choice 3

Input: `textOutput()`

```
[1] "1"
```

What if we want the user to select what data is shown?

User defined barplot

Before, we had you create a barplot of cases by sex:

```
ggplot() +
  geom_bar(data=raw_data, aes(x=sex, fill=sex)) +
  theme_classic()
```

And by species:

```
ggplot() +
  geom_bar(data=raw_data, aes(x=species, fill=species)) +
  theme_classic()
```

What part of the code changes to make those two plots?

User defined barplot

- In Shiny, we can let the user define what to plot.
- E.g. rather than showing 2 plots, the user can swap between sex or species using the dropdown menu provided.

```
selectInput(inputId = "xaxis",
            label = h3("Select the x-axis variable:"),
            choices = list("Sex" = "sex", "Species" = "species"),
            selected = 1)
```

User defined barplot

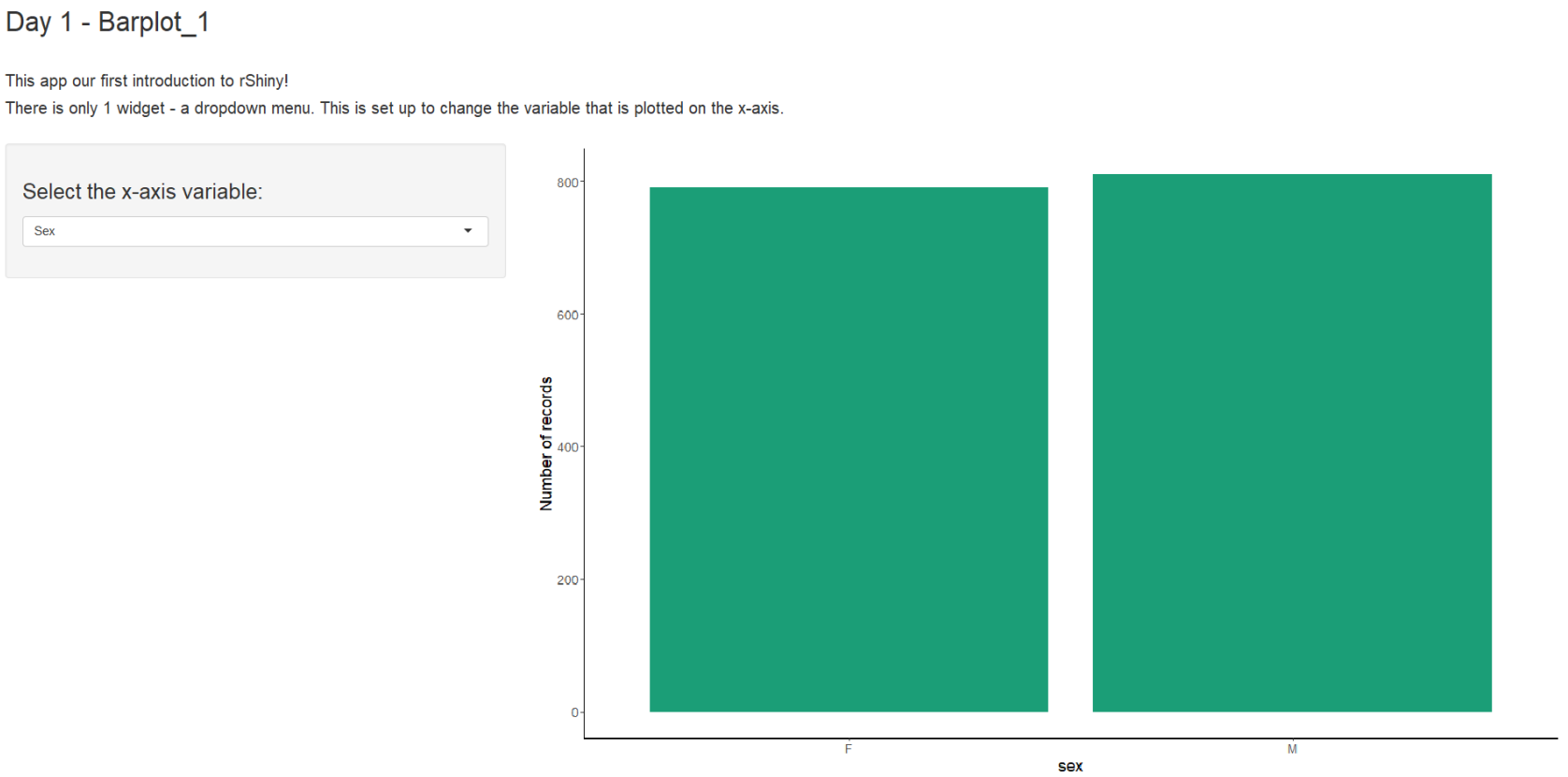
- The plot is made interactive by using the user's selection as an input in our server code.
- We then make the plot visible to the user by adding an output to the ui.

```
output$barPlot <- renderPlot({
  ggplot() +
    geom_bar(data=raw_data, aes_string(x=input$xaxis, fill=input$xaxis)) +
    theme_classic()
})
```

```
plotOutput("barPlot", height=200)
```

Your turn

Read the *App Guide* handout, then complete section 1.4b of the handout.



Summary - section 1.4b ui code

shinyUI Side Panel

```
selectInput(inputId = "xaxis",
            label = h3("Select the x-axis variable:"),
            choices = list("Sex" = "sex", "Species" = "species", "Age" = "age"),
            selected = 1)
```

Summary - section 1.4b server code

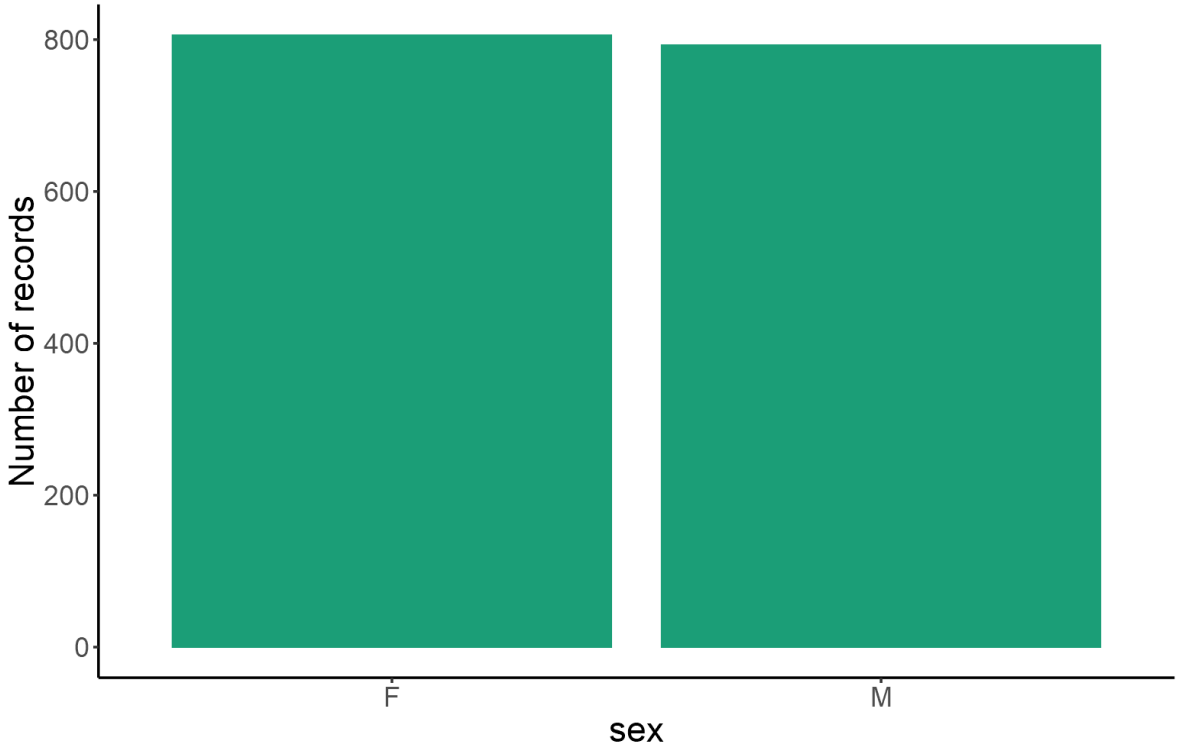
shinyServer Section

```
output$barPlot <- renderPlot({
  ggplot() +
    geom_bar(data=raw_data,
             aes_string(x=input$xaxis),
             fill=col_palette[1]) +
    labs(x=paste0(input$xaxis), y="Number of records") +
    ...
})
```

Summary - section 1.4b app

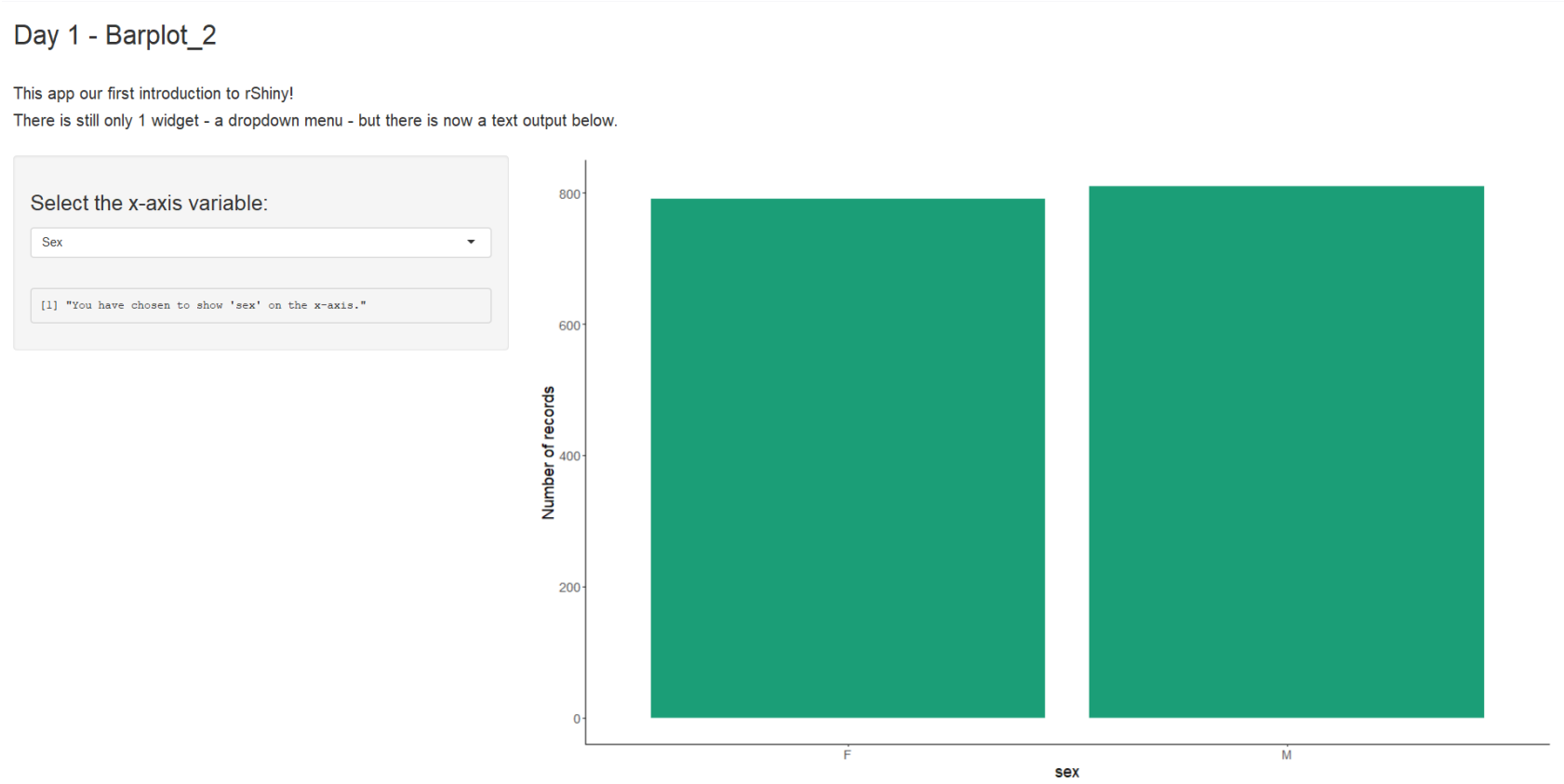
Select the x-axis variable:

Sex



Your turn

Complete section 1.4c of the handout.



Summary - section 1.4c ui code

shinyUI Side Panel

```
# Existing text output
verbatimTextOutput("output_text")

# New text output
verbatimTextOutput("output_values")
```

Summary - section 1.4c server code

shinyServer Section

```
# Existing text output
output$output_text <- renderPrint({
  paste0("You have chosen to show '", input$xaxis, "' on the x-axis.")
})

# New text output
output$output_values <- renderPrint({
  summary(raw_data[[input$xaxis]])
})
```

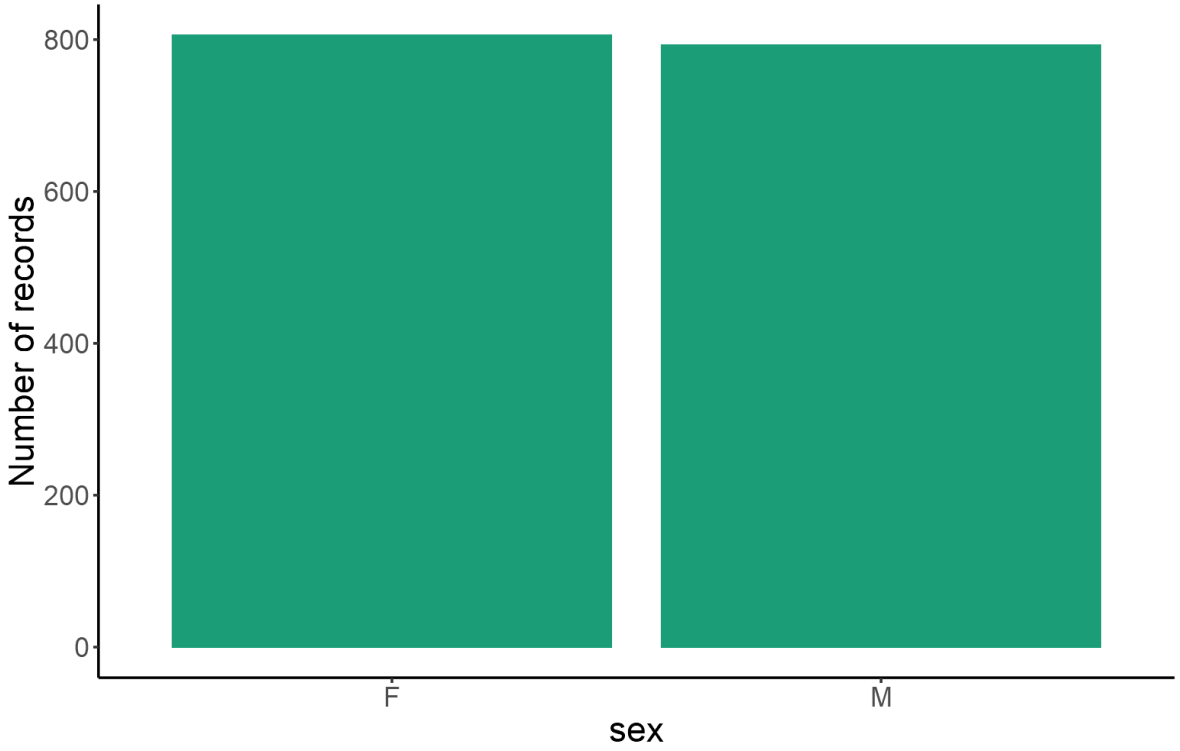
Summary - section 1.4c app

Select the x-axis variable:

Sex

```
[1] "You have chosen to show 'sex'
```

```
   Length     Class      Mode
     1599 character character
```



Making your code 'reactive'

- All code within the server must be contained within a **reactive** element, otherwise the app will fail to run.
- The previous examples used `render()` functions, which are **reactive** but limited to a small amount of code.

```
output$output_text <- renderPrint({
    paste0("You have chosen to show '", input$xaxis, "' on the x-axis.")
    })
```

Making your code ‘reactive’ contd

- If we want to process the data based on user input (e.g. filter, summarise), we need to wrap the code within a **reactive function**.

- `reactive()`
- `eventReactive()`
- `reactiveValues()`
- `observeEvent()`

What if we want the user to subset the data?

The `reactive()` function

- The `reactive()` function is used to carry out data processing in response to one or more user inputs, with the output saved as an object
- Objects created in the function should be done with an = NOT <-
- The last line of the function should be the object you would like to be saved overall (e.g. a data subset)
- When **any** input used in the `reactive` function is changed, the output will automatically update

An example with the `reactive()` function

shinyUI Side Panel

```
radioButtons(inputId = "select_species", label = "Select a Species:",
             choices = sort(unique(raw_data$species))),
```

shinyUI Main Panel

```
tableOutput("chosen_species_info")
```

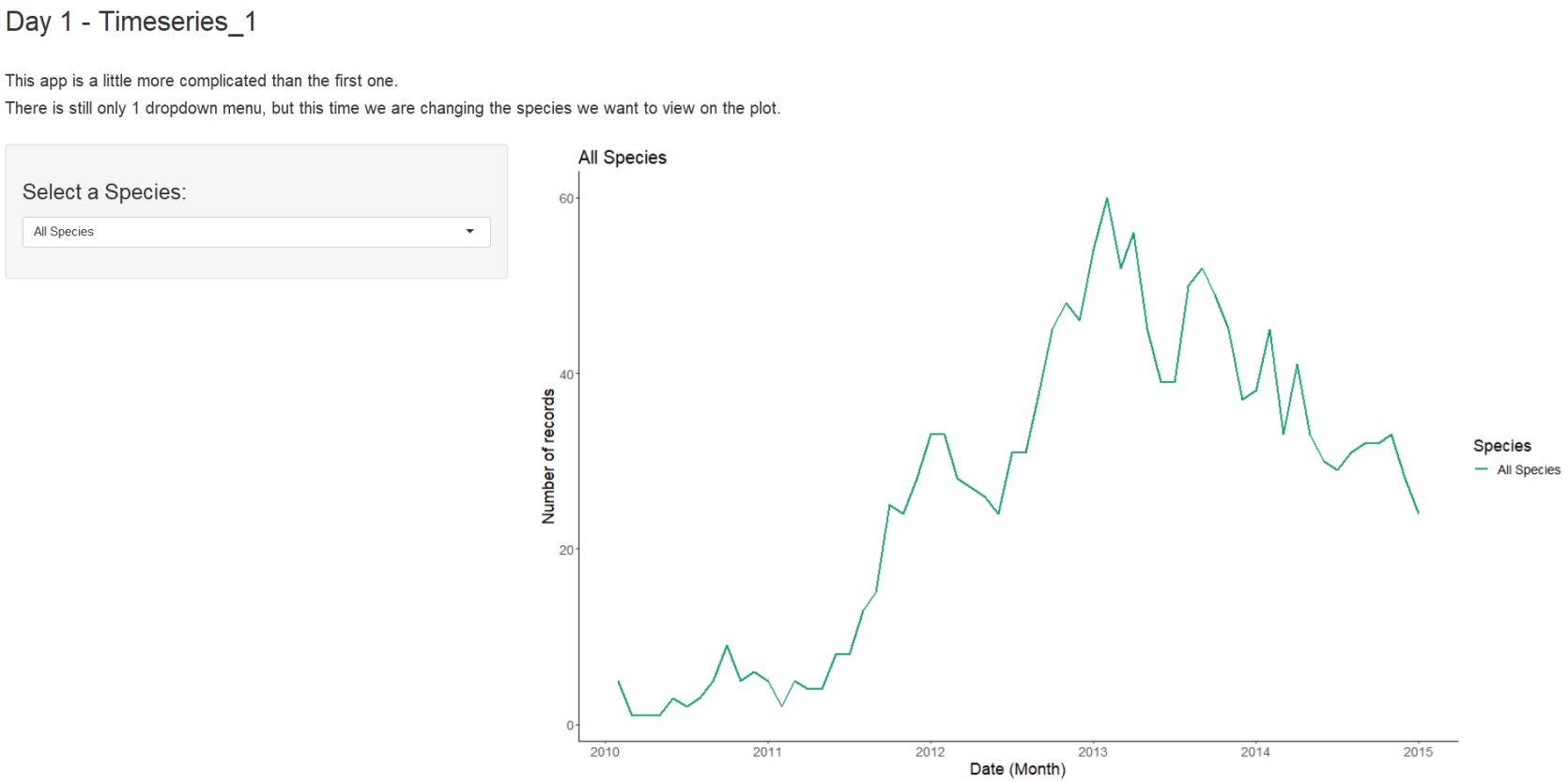
shinyServer Section

```
data_sub <- reactive({
  data_subset = raw_data %>%
    filter(species == input$select_species)
  data_subset = head(data_subset)
  data_subset
})

output$chosen_species_info <- renderTable({
  data_sub()
})
```

An example with the `reactive()` function

Your turn

Complete section 1.4d of the handout.



Summary - section 1.4d ui code

Above shinyUI Section

```
options_list <- c("All Regions", sort(unique(raw_data$region)))
```

shinyUI Side Panel

```
selectInput("select_region", label = h3("Select a Region:"),
            choices = options_list,
            selected = 1)
```

Summary - section 1.4d server code (1/2)

Above shinyServer Section

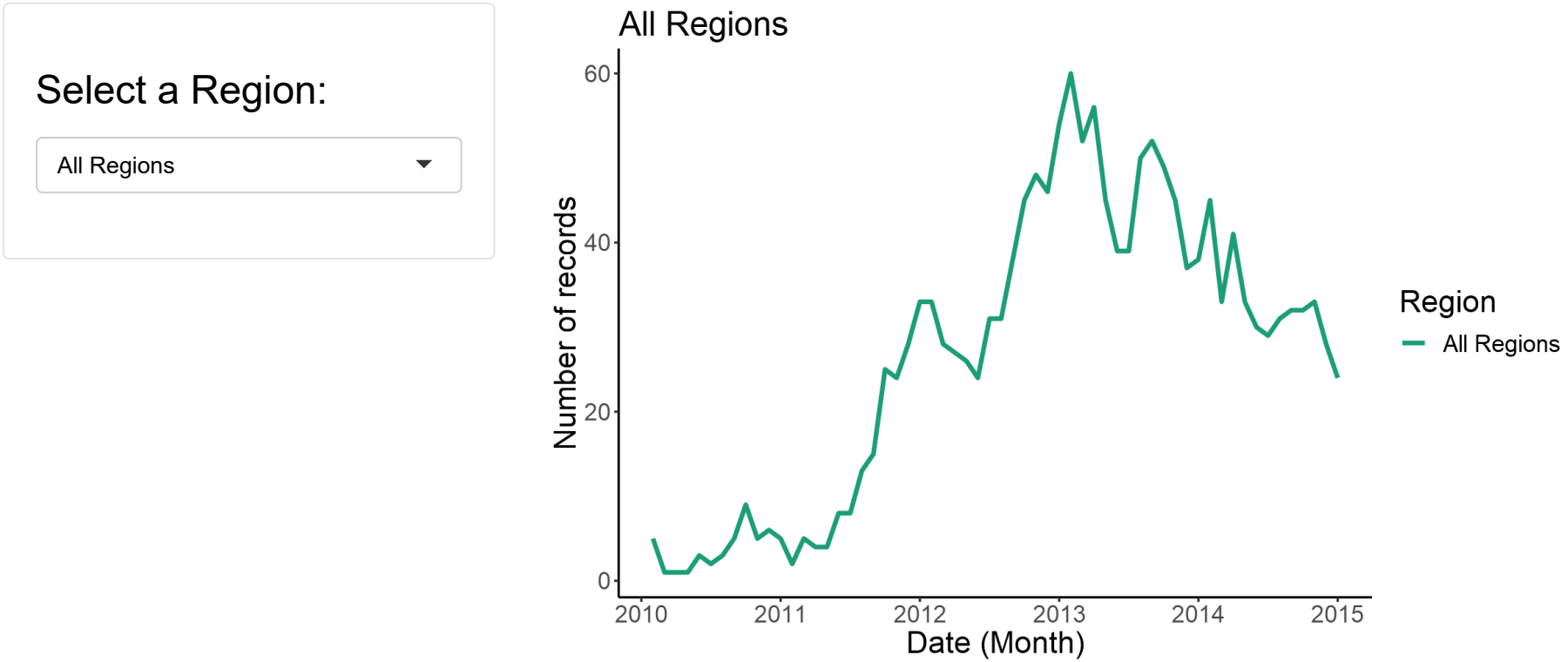
```
overall_summary <- raw_data %>%
  group_by(month) %>%
  summarise(n = length(month)) %>%
  mutate(region = "All Regions")

region_summary <- raw_data %>%
  group_by(month, region) %>%
  summarise(n = length(month))

summary_data <- bind_rows(overall_summary, region_summary)
```

Summary - section 1.4d server code (2/2)

shinyServer Section

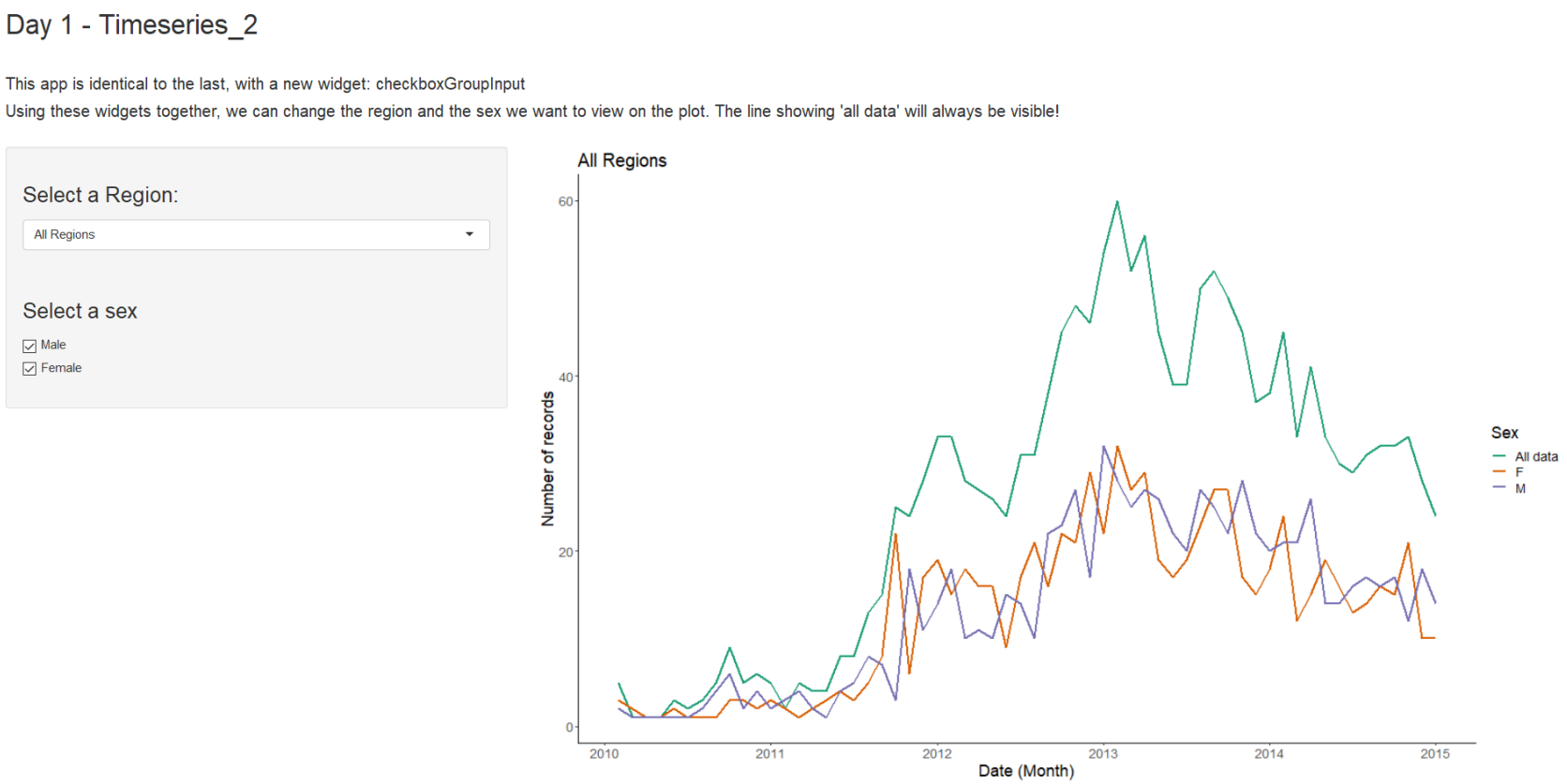
```
data_subset <- reactive({
  data_sub = summary_data %>%
    filter(region==input$select_region | region=="All Regions")
  as.data.frame(data_sub)
})

output$tsPlot <- renderPlot({
   ggplot() +
      geom_path(data=data_subset(), aes(x=month, y=n, color=region), size=1) +
      scale_color_manual(name="Region", values=col_palette) +
      labs(title=input$select_region, x="Date (Month)", y="Number of records") +
    ...
})
```

Summary - section 1.4d app



Your turn

Complete section 1.4e of the handout.



Summary - section 1.4e ui code

shinyUI Side Panel

```
checkboxGroupInput("select_species", label = h3("Select a Species"),
                           choices = list("Cat" = "cat", "Dog" = "dog", "Human"="human",
                                          "Jackal"="jackal", "Lion"="lion"),
                           selected = c("cat", "dog", "human", "jackal", "lion"))
```

Summary - section 1.4e server code (1/2)

Above shinyServer Section

```
overall_summary <- raw_data %>%
  group_by(month) %>%
  summarise(n = length(month)) %>%
  mutate(region="All data",
         species="All data")

region_allspecies_summary <- raw_data %>%
  group_by(month, region) %>%
  summarise(n = length(month)) %>%
  mutate(species="All species")

species_allregions_summary <- raw_data %>%
  group_by(month, species) %>%
  summarise(n = length(month)) %>%
  mutate(region="All Regions")

region_species_summary <- raw_data %>%
  group_by(month, region, species) %>%
  summarise(n = length(month))

summary_data <- bind_rows(overall_summary, region_allspecies_summary,
                          species_allregions_summary, region_species_summary)
```

Summary - section 1.4e server code (2/2)

shinyServer Section

```
data_subset <- reactive({

  data_sub = summary_data %>%
    filter(region==input$select_region | region=="All data")

  if(length(input$select_species)>0){
    data_sub = data_sub %>%
      filter(species %in% input$select_species | species=="All species" | species=="All data")
  } else {
    data_sub = data_sub %>%
      filter(species=="All species" | species=="All data")
  }

  data_sub = data_sub %>%
    group_by(month, region, species) %>%
    summarise(n = sum(n))
  as.data.frame(data_sub)
})
```

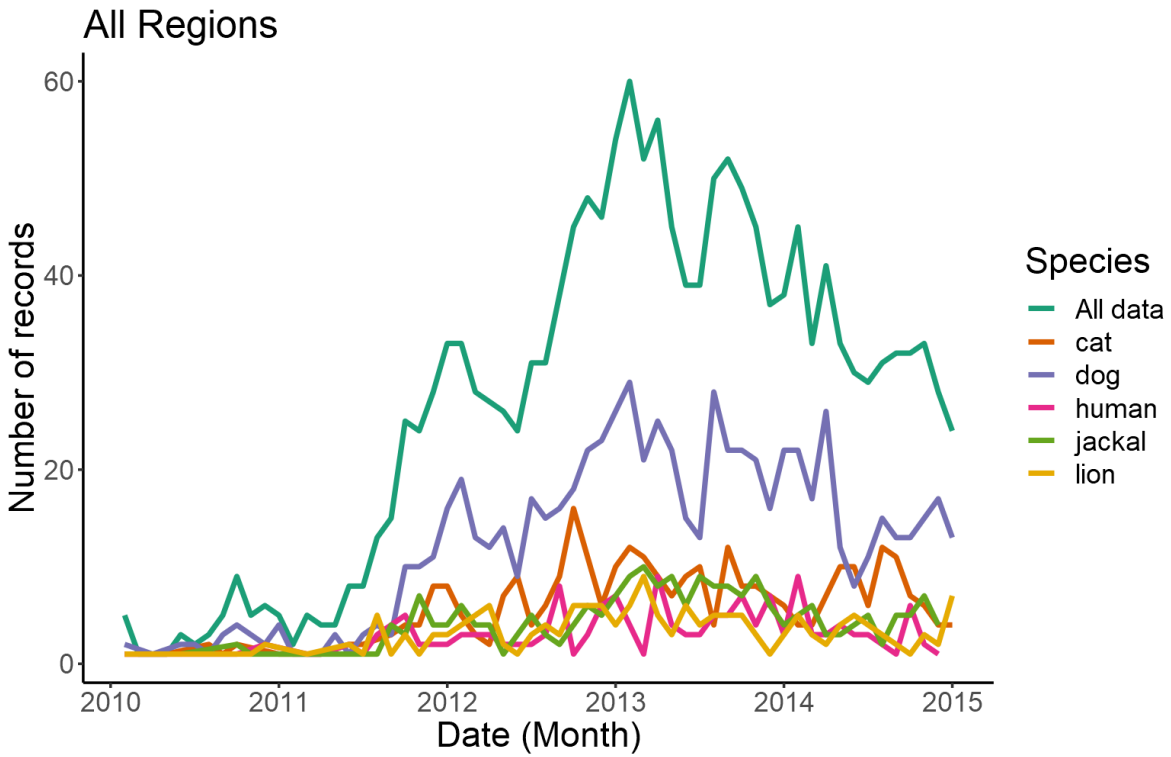
Summary - section 1.4e app

Select a Region:

All Regions

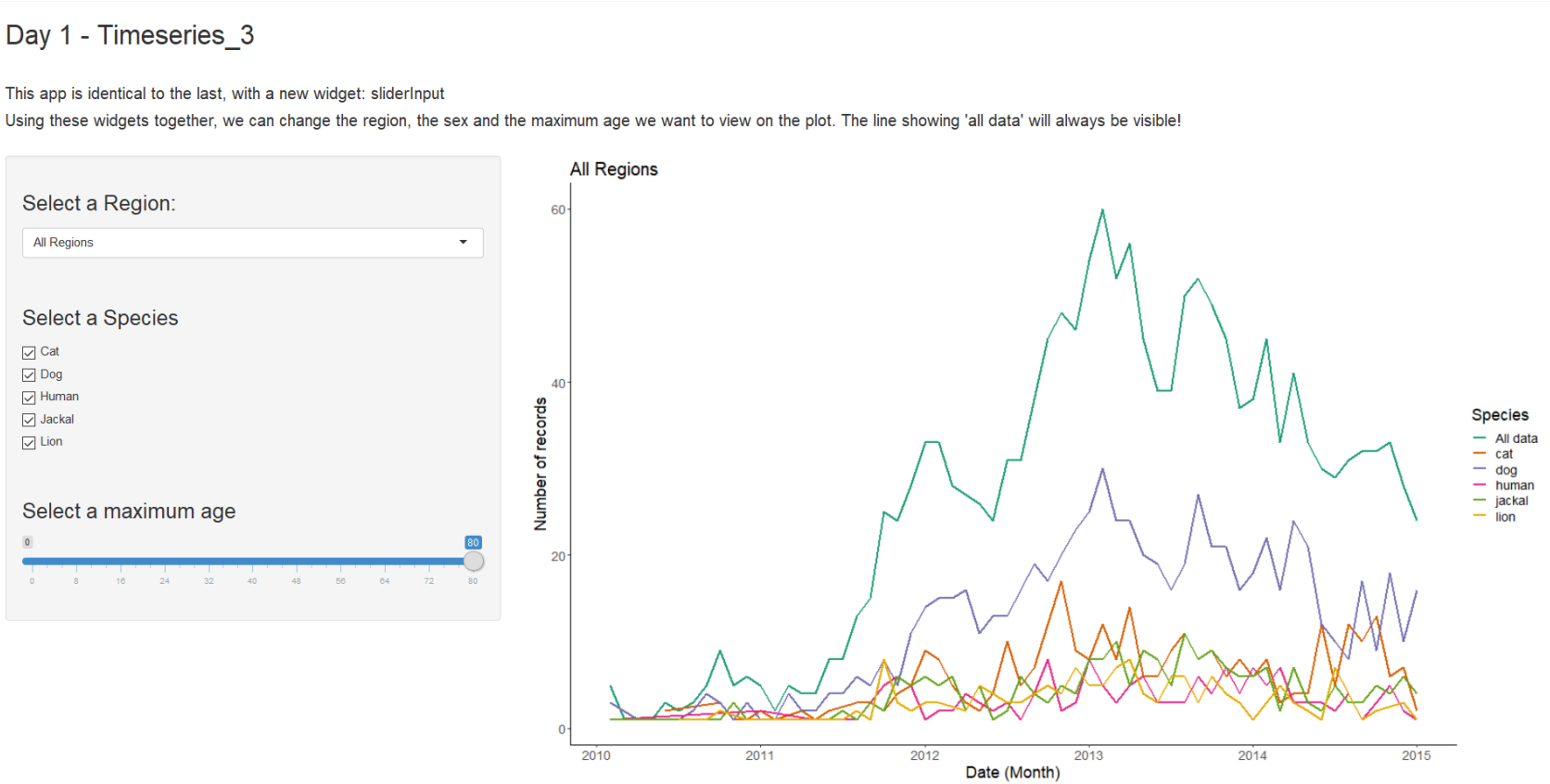
Select a Species

- [x] Cat
- [x] Dog
- [x] Human
- [x] Jackal
- [x] Lion



Your turn

Complete section 1.4f of the handout.



Summary - section 1.4f ui code

shinyUI Side Panel

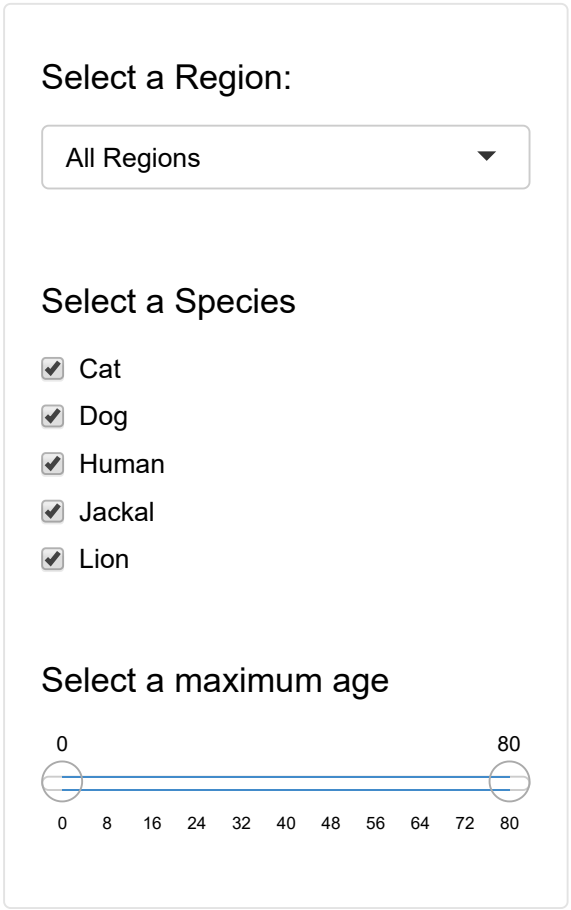
```
sliderInput("age_slider", label = h3("Select a maximum age"),
                  min = min_age, max = max_age, value = c(min_age, max_age))
```

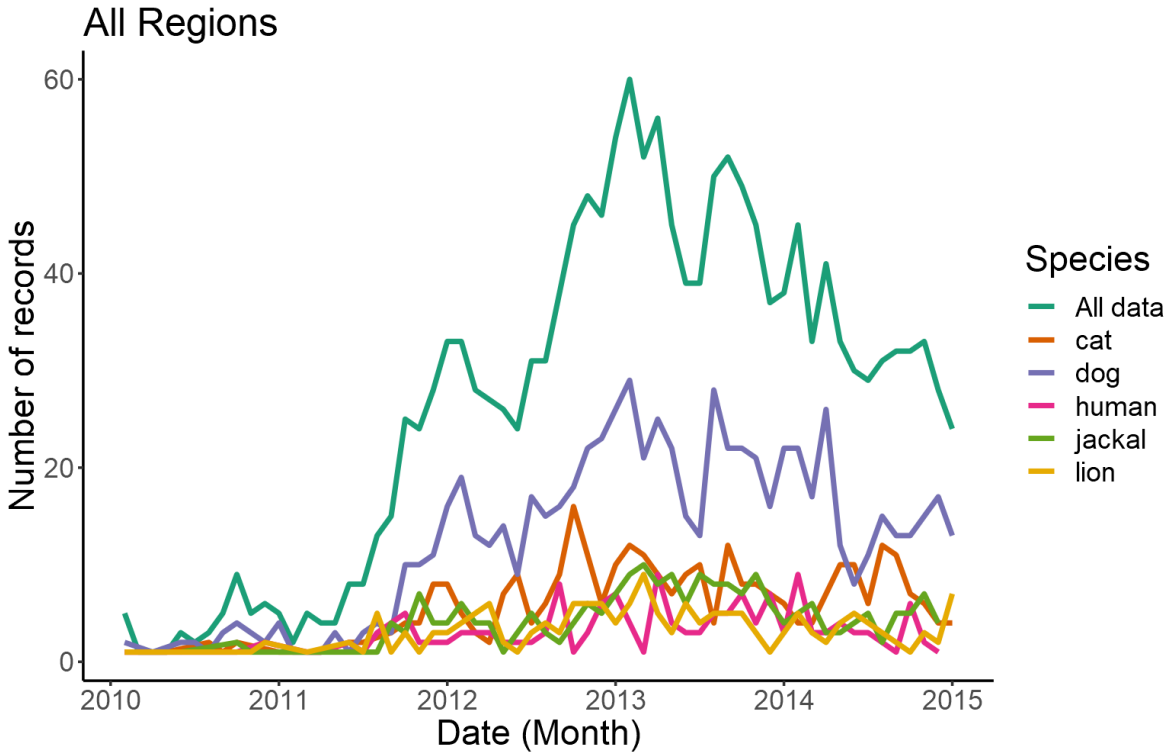
Summary - section 1.4f server code

shinyServer Section

```
data_subset <- reactive({

    data_sub = summary_data %>%
      filter(region==input$select_region | region=="All data")

    if(length(input$select_species)>0){
      data_sub = data_sub %>%
        filter(species %in% input$select_species | species=="All species" | species=="All data")
    } else {
      data_sub = data_sub %>%
        filter(species=="All species" | species=="All data")
    }

    data_sub = data_sub %>%
      filter(age >= input$age_slider[1] & age <= input$age_slider[2] | is.na(age))

    data_sub = data_sub %>%
      group_by(month, region, species) %>%
      summarise(n = sum(n))
    as.data.frame(data_sub)
  })
```

Summary - section 1.4f app





What if we want to delay the app changes?

The `eventReactive()` function

- The `eventReactive` function is very similar to the `reactive()` function - it is used to carry out data processing.
- The difference is that it is triggered by a specific user input
- The input can be set as any of the widgets in your app, but it works well with `actionButton()`
- `eventReactive()` returns `NULL` until the button is pressed, which means any outputs that rely on it are hidden until the button is pressed…

An example with the `eventReactive()` function

shinyUI Side Panel

```
radioButtons(inputId = "select_species", label = "Select a Species:",
             choices = sort(unique(raw_data$species)),
             selected = 1)

actionButton(inputId="go_button", label="Click to generate the plot:")
```

shinyUI Main Panel

```
plotOutput("chosen_species")
```

An example with the `eventReactive()` function

shinyServer Section

```
data_subset <- eventReactive(input$go_button, {
      data_sub = raw_data %>%
        filter(species == input$select_species)
      data_sub
    })

output$chosen_species <- renderPlot({
      ggplot() +
        geom_bar(data=data_subset(), aes(x=age)) +
        ggtitle(unique(data_subset()$species))
    })
```

An example with the `eventReactive()` function

Input: `radioButtons()`

Select a Species:

- (•) cat
- ( ) dog
- ( ) human
- ( ) jackal
- ( ) lion

Input: `actionButton()`

Click to generate the plot:

What if we want to delay the app changes, but still show an output to start with?

What if we want to delay the app changes, but still show an output to start with?

- We can combine 2 functions: `observeEvent()` and `reactiveValues()`

The `observeEvent()` function

- The `observeEvent()` function is very similar to the `eventReactive()` function - it is used to carry out a process in response to a specific user input.
- The difference is `observeEvent()` **does not produce an output** - instead it works by **updating existing outputs**

The `reactiveValues()` function

- The `reactiveValues()` function produces an object that acts like a list. - you can store multiple values or objects to *slots* - *slots* can be accessed from the object with the `$` symbol
- The `reactiveValues` object can be created with values, or values can be added after
- Stored values are updated by reactive functions

observeEvent() and reactiveValues()

eventReactive() - shinyServer Section

```
data_subset <- eventReactive(input$go_button, {
      data_sub = raw_data %>% filter(species == input$select_species)
      data_sub
    })
output$chosen_species <- renderPlot({
      ggplot() + geom_bar(data=data_subset(), aes(x=age)) +
        ggtitle(unique(data_subset()$species))
    })
```

observeEvent() and reactiveValues() - shinyserver Section

```
rv <- reactiveValues(data_sub = filter(raw_data, species=="cat"))
observeEvent(input$go_button, {
      rv$data_sub = raw_data %>%
        filter(species == input$select_species)
    })
output$chosen_species <- renderPlot({
      ggplot() + geom_bar(data = rv$data_sub, aes(x=age)) +
        ggtitle(unique(rv$data_sub$species))
    })
```

observeEvent() and reactiveValues()

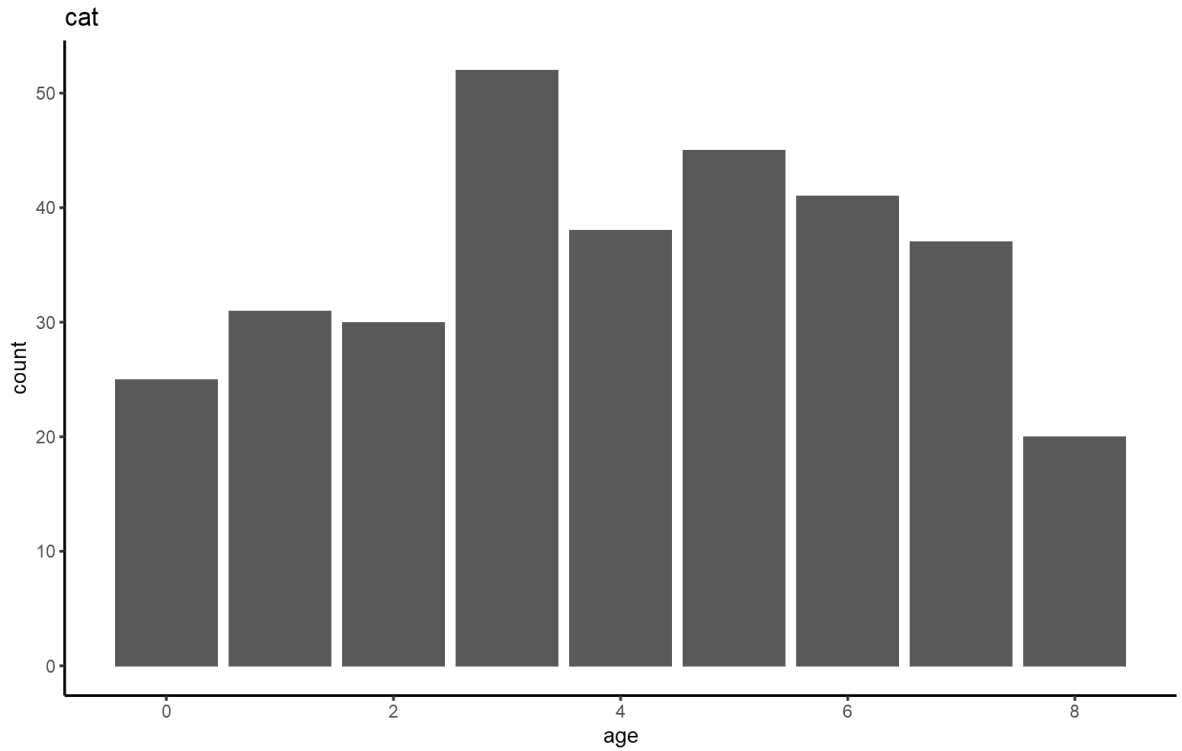
Input: radioButtons()

Select a Species:

- (•) cat
- ( ) dog
- ( ) human
- ( ) jackal
- ( ) lion

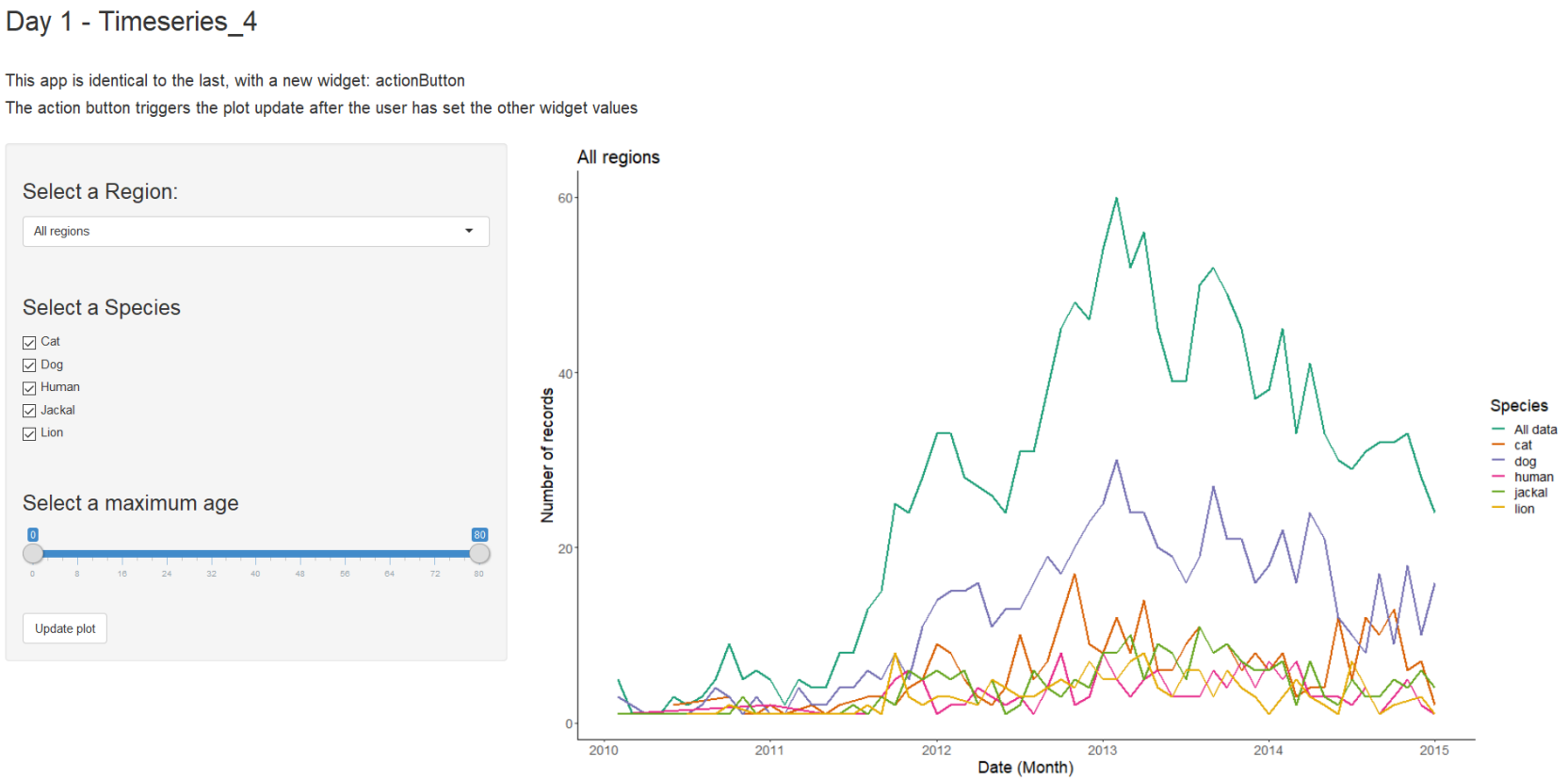
Input: actionButton()

Click to generate the plot:



Your turn

Complete section 1.4g of the handout.



Summary - section 1.4g

What reactive elements have changed in the ui code and the server code?

Summary - section 1.4g

What reactive elements have changed in the ui code and the server code?

shinyUI Side Panel

```
actionButton("action_button", label = "Update plot")

actionButton("reset_button", label = "Reset plot")
```

Summary - section 1.4g

What reactive elements have changed in the ui code and the server code?

Above shinyserver Section

```
start_df <- summary_data %>%
  filter(region=="All regions" | region=="All data") %>%
  group_by(month, region, species) %>%
  summarise(n = sum(n))
```

Summary - section 1.4g

What reactive elements have changed in the ui code and the server code?

shinyserver Section

```
data_subset <- reactiveValues(data = start_df)
```

Summary - section 1.4g

What reactive elements have changed in the ui code and the server code?

shinyserver Section

```
observeEvent(input$action_button, {

  data_sub = summary_data %>%
      filter(region==input$select_region | region=="All data")

    if(length(input$select_species)>0){
      data_sub = data_sub %>%
        filter(species %in% input$select_species | species=="All species" | species=="All data")
    } else {
      data_sub = data_sub %>%
        filter(species=="All species" | species=="All data")
    }

    data_sub = data_sub %>%
      filter(age >= input$age_slider[1] & age <= input$age_slider[2] | is.na(age))

    data_sub = data_sub %>%
      group_by(month, region, species) %>%
      summarise(n = sum(n))
    data_subset$data <- as.data.frame(data_sub)
  })
```

Summary - section 1.4g

What reactive elements have changed in the ui code and the server code?

shinyserver Section

```
output$tsPlot <- renderPlot({
   ggplot() +
     geom_path(data=data_subset$data, aes(x=month, y=n, color=species), size=1) +
     scale_color_manual(name="Species", values=col_palette) +
    ...
  })
```

Summary - section 1.4g

Can you spot any new functions in the shinyServer that we have not introduced yet?

Summary - section 1.4g

Can you spot any new functions in the shinyServer that we have not introduced yet?

```
observeEvent(input$reset_button, {

  updateSelectInput(session, inputId = "select_region",
                    selected = "All regions")

  updateCheckboxGroupInput(session, inputId = "select_species",
                           selected = c("cat", "dog", "human", "jackal", "lion"))

  updateSliderInput(session, inputId = "age_slider",
                    min = min_age, max = max_age, value = c(min_age, max_age))
})
```

Summary - section 1.4g

Can you think of any scenarios where it would be helpful to use an `actionButton()` to trigger a reactive event?

Summary - section 1.4g

Can you think of any scenarios where it would be helpful to use an `actionButton()` to trigger a reactive event?

- Delay an action
  - complicated processing (e.g. model)
  - plotting (e.g. map)
- Select data
  - swap quickly between datasets or variables
- Reset the app
  - replace any user choices with default values

Building an interactive plot in RShiny - Summary

Basic functions

- **inputs** are used in the ui to give the user options to change
  e.g. `selectInput()`
- **render** is used in the server to create the plots, tables etc. that the user sees
  e.g. `renderPlot()`
- **outputs** are used to display the rendered objects in the ui
  e.g. `plotOutput()`

Building an interactive plot in RShiny - Summary

Reactive functions

- `reactive()` is used to carry out processes immediately, such as subsetting and calculations
- If you want to delay the processing in your app, you can use `eventReactive()` to watch for a specific user input and then carry out processes
- If you want to delay the processing in your app, but display a base dataset, you can use `reactiveValues()` to create a stored object and `observeEvent()` to update the value after a specific user input