Public Health Scotland

Introduction to R Shiny
Day 1

Public Health Scotland

Learning Outcomes – Day 1

1. Introduction: what is R Shiny?
2. Knowledge of packages often relevant to R Shiny work
3. Shiny app breakdown: UI, Server, Global scripts
4. Overall layouts – eg. sidebarLayout (sidebarPanel, mainPanel), navbarPage, tabPanel, fluidRow, column
5. Reading in prepared data
6. Creating charts and tables – ggplot, DT
7. shinyWidgets and Reactivity: radioButtons, drop-down menus, sliders
8. Multi-tab dashboards

Public Health Scotland

What is R Shiny?

- Shiny is an open source R package that allows the creation of interactive web applications straight from R
- Shiny works in any R environment and comes with pre-built and customizable output widgets for displaying plots, tables, and printed output of R objects
- **Key point:** functions associated with Shiny are generally written in CamelCase (unfortunately!)
- Shiny power - check out some PHS Shiny dashboards:
  - ScotPHO Profiles Tool: https://scotland.shinyapps.io/ScotPHO_profiles_tool/
  - Covid Wider Impacts Dashboard: https://scotland.shinyapps.io/phs-covid-wider-impact/

Public Health Scotland

R packages associated with R Shiny work

- Some examples of key packages and their functions with R Shiny:
  - **shiny** – the package that allows all of this to be possible!
  - **shinyWidgets** – for action buttons, sliders, drop-downs, radio buttons and many more, check it out: https://shiny.rstudio.com/gallery/widget-gallery.html
  - **shinyjs** – for bringing basic JavaScript into R eg. enable and disable functions
  - **shinyBS** – create buttons and collapsible panels
  - **shinycssloaders** – for adding icon pictures to tabs
  - **shinymanager** – for password protecting live apps
  - **ggplot2**, **DT** – for creating charts/tables
  - **dplyr** and **readr** – standard packages for data manipulation and reading/writing csv files

Public Health Scotland

Shiny app breakdown

- Shiny apps may either be contained within a single script (not recommended for full dashboard building) or split across multiple scripts.

- Small apps are contained within a single script called **`app.R`** which requires three main components to run the app:
  - **user interface object** (defined as UI) which controls the layout and appearance of your app.
  - **server function** (defined as server) which contains the instructions your computer needs to build the app.
  - A call to the **`shinyApp function`** which creates Shiny app objects from an explicit UI/server pair.

The User Interface (UI)

- The user interface (UI) controls what is displayed on the application page and how the components are laid out.
- Examples:
  - navigation bars
  - text/titles, markdown elements
  - download buttons, plot outputs from server, user input widgets
- **Key point:** Shiny uses the function `fluidPage()` to create a display that automatically adjusts to the dimensions of the users browser window. The above UI elements will generally be placed within this function.

Public Health Scotland

The Server

- The server-side controls everything that happens behind the scenes, for example, the data that will be displayed through the UI.
- This part of the script defines how we generate all the plots and tables seen by the user.
- It also defines how user inputs from our widgets (such as a user selecting from a drop-down menu) affects these plots and tables.
- If necessary, the server section may also be used for minor data wrangling, such as filtering it in such a way that it can be fed into a plot.

Extra: the Global script

- This section becomes important when creating large dashboards.
- Generally used for loading packages and functions, and also prepared data files as named objects.
- May also be used to define other things such as colour palettes and plot parameters.

- **Key point:** adding this script helps to keep your code clean and tidy. When running the app, all packages, functions and data will be read in from the Global script, while the app itself is created from the UI and Server.

Public Health Scotland

Any questions so far?

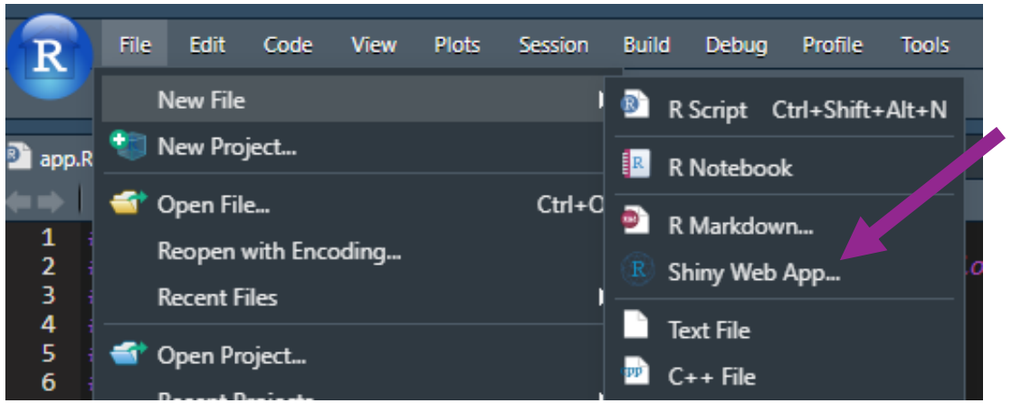
Public Health Scotland

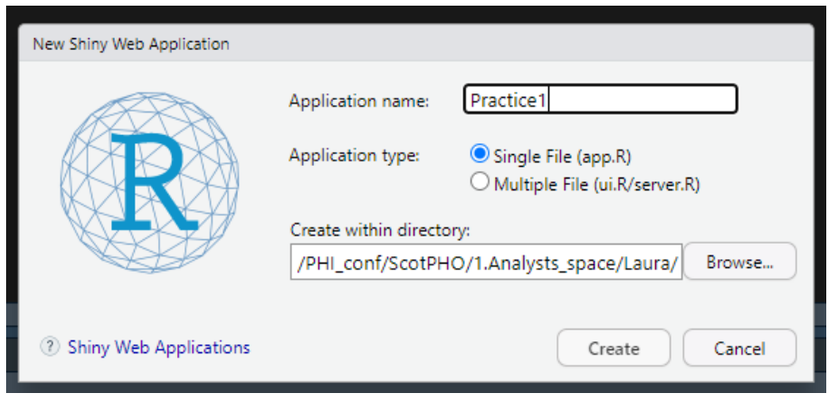
Code Along: the bare bones of a Shiny app

- Open a new session on R Studio server
- Go to File > New File > Shiny Web App…
- You will be asked to give your app a name (pick anything you like eg. “Practice1”) and to set a file path (where you can save your practice scripts).
- Ensure the Application type is set as “Single File (app.R)” as we are beginning with the basics.

- Click “Create” and R should provide you with a basic sample Shiny app created using the classic “Old Faithful Geyser” dataset.

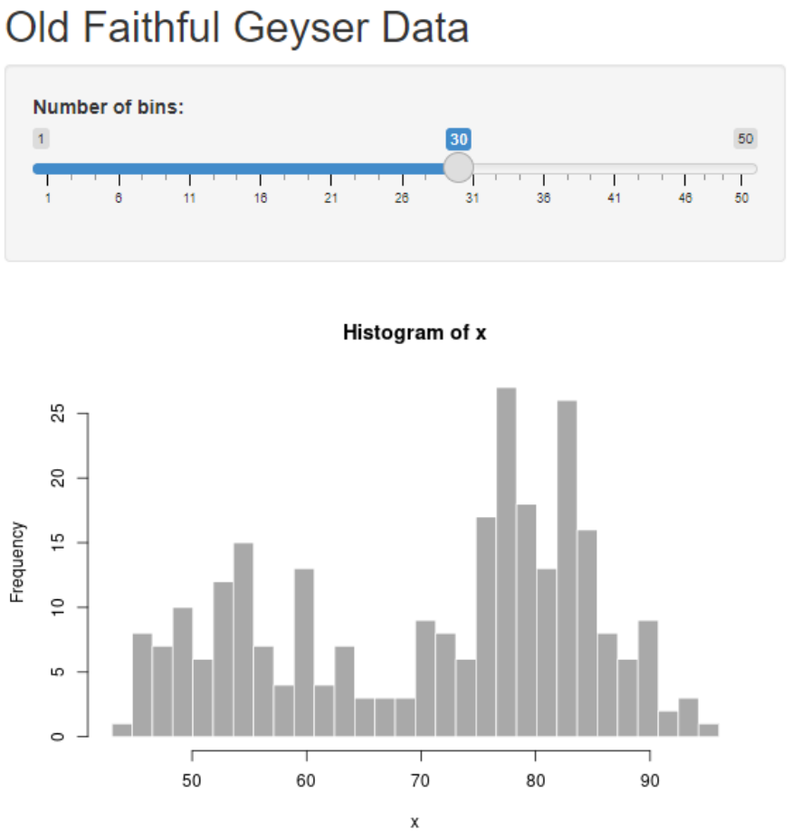
Code Along: the bare bones of a Shiny app





Upon creating the new Shiny Web App you will see some sample code related to the "Old Faithful Geyser" dataset.

Code Along: the bare bones of a Shiny app



Click **“Run App”** at the top right of the script window.

The app may run in a new window, or in the viewer pane (you can change between with the drop-down arrow at “Run App”).

You should see the Old Faithful Geyser app.

Public Health Scotland

Code Along: the bare bones of a Shiny app

The Old Faithful Geyser data isn't exactly the bare bones of a Shiny app, because it contains data and elements such as plots and slider inputs!

Remove all code relating to Old Faithful Geyser, and we are left with literally an empty app.

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {
}

shinyApp(ui = ui, server = server)
```

This can be a good place to begin when starting from scratch. We can build an app up from nothing!

Public Health Scotland

Code Along: the bare bones of a Shiny app

```
ui <- fluidPage(

  titlePanel("Title"),

  sidebarLayout(
     sidebarPanel(
       "Sidebar",
       br(),
       "Some other text in the sidebar"
     ),
     mainPanel(
       "Main section",
       br(),
       br(),
       br(),
       "Some other text in the main section"
      )
   )
)

server <- function(input, output) {
}

shinyApp(ui = ui, server = server)
```

`titlePanel()` function creates text formatted as a title at the top of the app.

`sidebarLayout()` function is a layout option for and requires two arguments: **`sidebarPanel()`** and **`mainPanel()`** functions.

`br()` function is used to separate text onto different lines.

Public Health Scotland

Outcomes

We've seen now seen how the **`sidebarPanel()`** and **`mainPanel()`** functions work within **`sidebarLayout()`** to create different sections of the app.

Now we're going to load some sample data and slowly build up our own app which will include titles, radio buttons, drop-down menus, charts, tables and text. We'll also try to split the outputs across multiple tabs using some Shiny functions you haven't seen yet.

We'll be using the nyc_dogs dataset which comes from the NYC department of health and contains the popularity of dog names in New York City.

Public Health Scotland

Code Along: creating a functioning Shiny app

We'll begin by loading the required packages and reading in our simple nyc_dogs dataset.

```
library(shiny)
library(tidyverse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")
```

Next, we'll work on forming a basic Shiny app from this dataset.

Public Health Scotland

Code Along: creating a functioning Shiny app

First, we'll add a radio button to the UI which allows us to select the gender of the dog and display the output as a table. **Remember that although the presence of the table output is defined in the UI, the table itself must be created in the server.**

```
ui <- fluidPage(

   radioButtons("gender",
     "Male or Female Dogs?",
     choices = c("Male", "Female")),

   tableOutput("table_output")
)
```

The **`radioButtons()`** function requires three arguments here:

`radioButtons(inputId, label, choices = NULL)`

The **`tableOutput()`** function requires only one argument:

`tableOutput(outputId)`

We will define this outputId in the server, where we create our table.

Public Health Scotland

Code Along: creating a functioning Shiny app

Now we'll look at what needs to be in the server. This is where we define what should be included in the table output that will appear on the user interface.

```
server <- function(input, output) {

    output$table_output <- renderTable ({
        nyc_dogs %>%
            filter(gender == input$gender) %>%
         slice(1:10)
   })
}

shinyApp(ui = ui, server = server)
```

Within the server function, we are creating a table output object. In the UI code, the table is labelled as "**`table_output`**" and this must match in the server.

To create this object we use the **`renderTable({})`** function to select our dataset.

We have also included **`slice(1:10)`** so we only see the first 10 rows of the data.

Public Health Scotland

Code Along: creating a functioning Shiny app

Male or Female Dogs?

- (•) Male
- ( ) Female

dog_name	gender	breed	birth	colour	borough
Buddy	Male	Afghan Hound	Jan-00	Brindle	Manhattan
Trouble	Male	Afghan Hound	Jan-03	Blond	Bronx
Sisu	Male	Afghan Hound	Oct-04	Black	Manhattan
Jakie	Male	Afghan Hound	Feb-05	White	Queens
Geo	Male	Afghan Hound	Jan-07	Orange	Bronx
Troy	Male	Afghan Hound	Jul-09	Blond	Staten Island
Nick	Male	Afghan Hound	Nov-09	Black	Queens
Prince	Male	Afghan Hound	Jan-10	Tan	Queens
Bernie	Male	Akita	Jan-99	White	Queens
Jason	Male	Akita	Jan-99	Black	Queens

This is very basic and doesn't give us any real options to sort or search the data.

So let's do that using the **`dataTableOutput()`** and **`renderDataTable()`** functions from the **`DT package`**.

Public Health Scotland

Code Along: creating a functioning Shiny app

```
library(shiny)
library(tidyveryse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")

ui <- fluidPage(

   radioButtons("gender",
       "Male or Female Dogs?",
       choices = c("Male", "Female")),

   DT::dataTableOutput("table_output")
)

server <- function(input, output) {

   output$table_output <- DT::renderDataTable({
       nyc_dogs %>%
     filter(gender == input$gender)
 })
}

shinyApp(ui = ui, server = server)
```

Our code is much the same. However in the UI we are using:

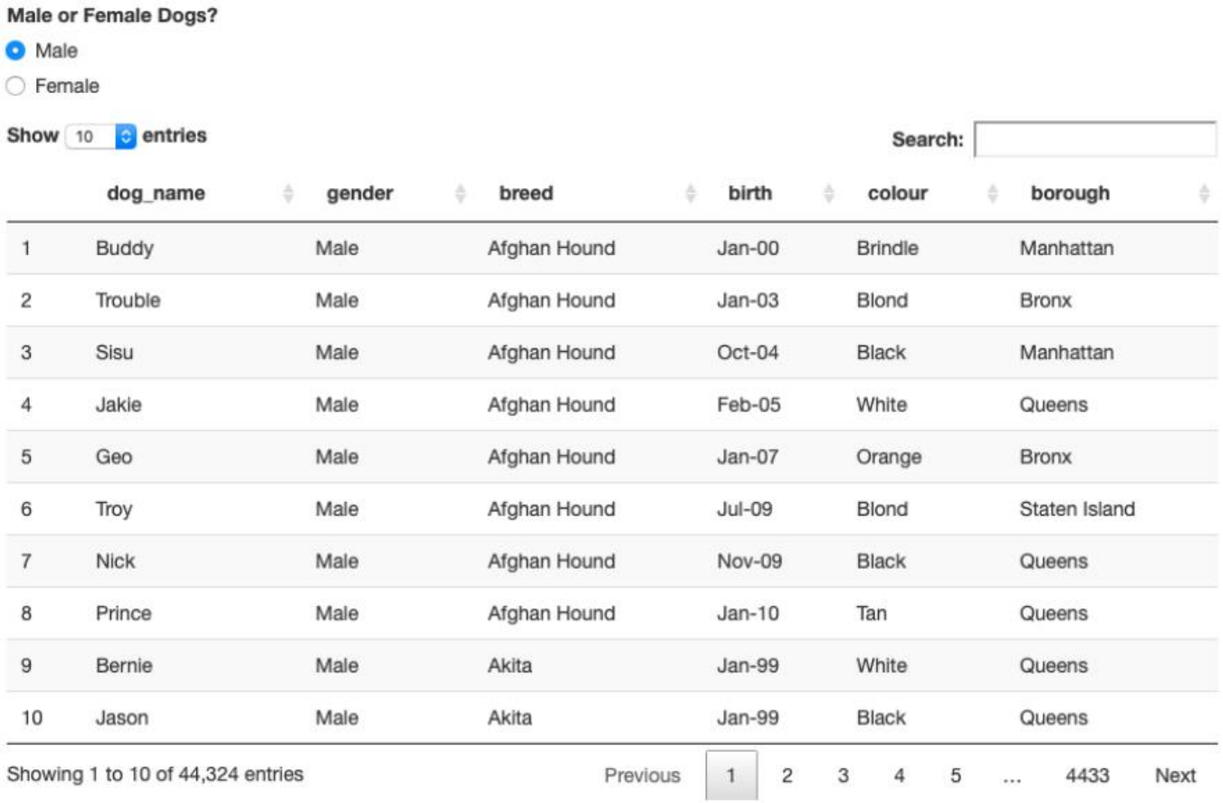
`DT::dataTableOutput("table_output")`
instead of
`tableOutput("table_output")`

And in the server we are using:

`DT::renderDataTable({…})`
instead of
`renderTable({…})`

Public Health Scotland

Code Along: creating a functioning Shiny app



Male or Female Dogs?

- (•) Male
- ( ) Female

Show 10 entries Search:

	dog_name	gender	breed	birth	colour	borough
1	Buddy	Male	Afghan Hound	Jan-00	Brindle	Manhattan
2	Trouble	Male	Afghan Hound	Jan-03	Blond	Bronx
3	Sisu	Male	Afghan Hound	Oct-04	Black	Manhattan
4	Jakie	Male	Afghan Hound	Feb-05	White	Queens
5	Geo	Male	Afghan Hound	Jan-07	Orange	Bronx
6	Troy	Male	Afghan Hound	Jul-09	Blond	Staten Island
7	Nick	Male	Afghan Hound	Nov-09	Black	Queens
8	Prince	Male	Afghan Hound	Jan-10	Tan	Queens
9	Bernie	Male	Akita	Jan-99	White	Queens
10	Jason	Male	Akita	Jan-99	Black	Queens

Showing 1 to 10 of 44,324 entries Previous 1 2 3 4 5 … 4433 Next

Your app will now look something like this, where you can sort the data in the table and search through it

Public Health Scotland

Code Along: creating a functioning Shiny app

```
library(shiny)
library(tidyveryse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")

ui <- fluidPage(

  fluidRow(
    column(3,
      radioButtons("gender",
       "Male or Female Dogs?",
       choices = c("Male", "Female"))
    ),
    column(3,
      selectInput("breed",
      "Which Breed?",
      choices = unique(nyc_dogs$breed))
    )
  ),

   tableOutput("table_output")
)
```

Here, we introduce **`fluidRow()`** which creates a grid-like layout for our app in conjunction with the **`column()`** function.

We've used **`radioButtons()`** before, but now we're also going to add a drop-down menu to select breed using the **`selectInput()`** function.

As with **`radioButtons()`**, the **`selectInput()`** function requires three arguments here:

`selectInput(inputId, label, choices)`

Public Health Scotland

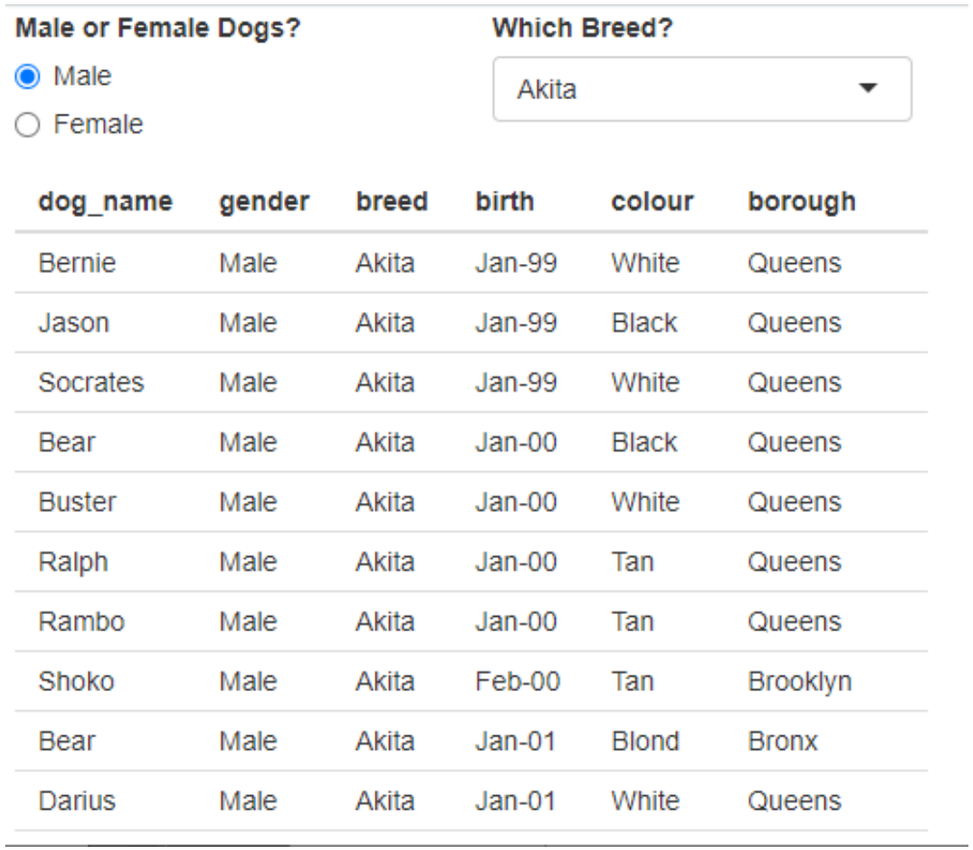
Code Along: creating a functioning Shiny app

```
server <- function(input, output) {

   output$table_output <- renderTable({
       nyc_dogs %>%
       filter(gender == input$gender) %>%
       filter(breed == input$breed)
   })
}

shinyApp(ui = ui, server = server)
```

We also have to add this new breed filter to the server side, if we want it to function on the app!

Public Health Scotland

Code Along: creating a functioning Shiny app



Male or Female Dogs?

- (•) Male
- ( ) Female

Which Breed?

Akita

dog_name	gender	breed	birth	colour	borough
Bernie	Male	Akita	Jan-99	White	Queens
Jason	Male	Akita	Jan-99	Black	Queens
Socrates	Male	Akita	Jan-99	White	Queens
Bear	Male	Akita	Jan-00	Black	Queens
Buster	Male	Akita	Jan-00	White	Queens
Ralph	Male	Akita	Jan-00	Tan	Queens
Rambo	Male	Akita	Jan-00	Tan	Queens
Shoko	Male	Akita	Feb-00	Tan	Brooklyn
Bear	Male	Akita	Jan-01	Blond	Bronx
Darius	Male	Akita	Jan-01	White	Queens

We now have an app which displays our data table with an option to select the gender of dog, and a drop-down menu to filter by breed.

Public Health Scotland

Exercise: creating a functioning Shiny app

- Add two more drop-downs to the app we have which allow you to filter for:
  - Borough
  - Dog colour

- Have a look at https://shiny.rstudio.com/gallery/widget-gallery.html to see the types of widgets available to use in Shiny apps

Public Health Scotland

Answer: creating a functioning Shiny app

```
ui <- fluidPage(

  fluidRow(
    column(3,
      radioButtons("gender",
       "Male or Female Dogs?",
       choices = c("Male", "Female"))
    ),
    column(3,
      selectInput("breed",
      "Which Breed?",
      choices = unique(nyc_dogs$breed))
    ),
    column(3,
      selectInput("borough",
      "Which Borough?",
      choices = unique(nyc_dogs$borough))
    ),
    column(3,
      selectInput("colour",
      "Which Colour?",
      choices = unique(nyc_dogs$colour))
    )
  ),

   tableOutput("table_output")
)
```

← UI

Server →

```
server <- function(input, output) {

   output$table_output <- renderTable({
        nyc_dogs %>%
        filter(gender == input$gender) %>%
        filter(breed == input$breed) %>%
        filter(borough == input$borough) %>%
        filter(colour == input$colour)
  })
}

shinyApp(ui = ui, server = server)
```

Public Health Scotland

Code Along: creating a functioning Shiny app

We're now going to add some charts to our app in order to better display the data!

We'll be using some basic ggplot2 knowledge to create the charts.

Our app should have a radioButton for selecting gender, and a drop-down menu for selecting breed. We will aim to include bar charts for borough and dog colour, instead of displaying the data in a table.

```
library(shiny)
library(ggplot2)
library(dplyr)
library(readr)

nyc_dogs <- read_csv("data/nyc_dogs.csv")
```

Public Health Scotland

Code Along: creating a functioning Shiny app

```
ui <- fluidPage(

  fluidRow(
    column(6,
      radioButtons("gender",
       "Male or Female Dogs?",
       choices = c("Male", "Female"))
    ),
    column(6,
      selectInput("breed",
      "Which Breed?",
      choices = unique(nyc_dogs$breed))
    )
  ),

  fluidRow(
    column(6,
      plotOutput("colour_barchart")
    ),
    column(6,
      plotOutput("borough_barchart")
    )
  )
)
```

← UI

Server →

```
server <- function(input, output) {

   filtered_data <-
     nyc_dogs %>%
       filter(gender == input$gender) %>%
       filter(breed == input$breed)

   output$colour_barchart <- renderPlot({
     ggplot(filtered_data) +
       geom_bar(aes(x = colour))
   })

   output$borough_barchart <- renderPlot({
     ggplot(filtered_data) +
       geom_bar(aes(x = borough))
   })
}

shinyApp(ui = ui, server = server)
```

In the server, we create an object called "**`filtered_data`**" from our dataset, so that we do not have to repeat chunks of filtering code for each bar chart.

But what happens when we run this code?

Reactivity

When we run the previous code, we get an error:

```
Do you need to wrap inside reactive() or observer()?
```

We've tried to use regular R code to filter our data so that it can be used for both plots but we need the filter for this object to be reactive depending on what the user inputs.

Public Health Scotland

Code Along: creating a functioning Shiny app

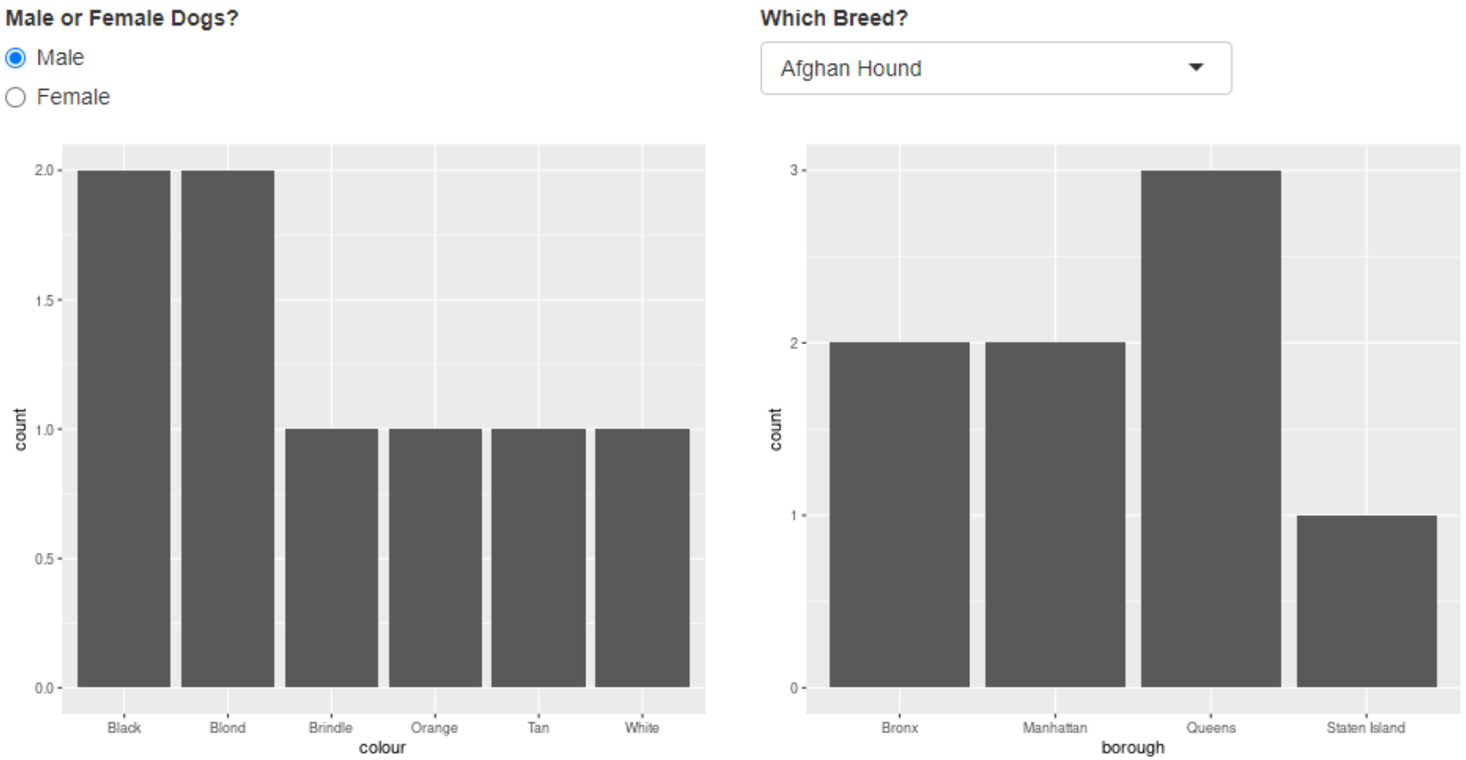
```
server <- function(input, output) {

   filtered_data <- reactive({
     nyc_dogs %>%
       filter(gender == input$gender) %>%
       filter(breed == input$breed)
   })

   output$colour_barchart <- renderPlot({
     ggplot(filtered_data()) +
       geom_bar(aes(x = colour))
   })

   output$borough_barchart <- renderPlot({
     ggplot(filtered_data()) +
       geom_bar(aes(x = borough))
   })
}

shinyApp(ui = ui, server = server)
```

We can fix this by wrapping our filtered data in **`reactive({})`**.

This defines our **`filtered_data`** variable as having a reactive value, and so when used later on in the code for chart creation, it must be followed by brackets:

`filtered_data()`

Code Along: creating a functioning Shiny app



Our app should now look like this, where the bar charts are reactive to selections made by the user either by the radio button or the drop-down menu.

Public Health Scotland

Code Along: creating a functioning Shiny app

What if we want to display our data using both tables and charts? We can create a multi-tabbed dashboard!

This introduces the Shiny function **`navbarPage()`** which creates a navigation bar along the top of the app, and **`tabPanel()`** which is used to create different tabs within the navigation bar.

Public Health Scotland

Including multiple tabs in your dashboard

Multi-tabs requires nesting of functions within other functions to ensure the dashboard displays correctly.

In order to have, for example, charts in one tab and the table in another, we need to wrap each of these code blocks (one for charts and one for tables) inside it's own **`tabPanel()`** function.

These **`tabPanel()`** functions then need to be nested inside the **`navbarPage()`** function which creates the overarching navigation bar along the top of the app.

Code Along: multiple tabs UI

```
ui <- fluidPage(
  titlePanel("NYC DOGS"),
  navbarPage("Navigation Bar",
             tabPanel(title = "Table",
                      fluidRow(
                        column(3,
                           radioButtons("gender",
                                        "Male or Female Dogs?",
                                        choices = c("Male", "Female"))
                        ),
                        column(3,
                            selectInput("breed",
                                        "Which Breed?",
                                        choices = unique(nyc_dogs$breed))
                        ),
                        column(3,
                            selectInput("borough",
                                        "Which Borough?",
                                        choices = unique(nyc_dogs$borough))
                        ),
                        column(3,
                            selectInput("colour",
                                        "Which Colour?",
                                        choices = unique(nyc_dogs$colour))
                        )
                      ),

                      tableOutput("table_output")),
```

Here's the code for the first tab containing a table.

Within our **`fluidPage()`**, I've created a **`navbarPage()`** to introduce a navigation bar, and nested a **`tabPanel()`** within this.

The next **`tabPanel()`** will open directly after the final line here.

Code Along: multiple tabs UI

```
tabPanel(title = "Plot",
         fluidRow(
            column(6,
              radioButtons("gender_chart",
                           "Male or Female Dogs?",
                           choices = c("Male", "Female"))
                ),
            column(6,
              selectInput("breed_chart",
                          "Which Breed?",
                          choices = unique(nyc_dogs$breed))
                )
             ),

         fluidRow(
            column(6,
              plotOutput("colour_barchart")
                ),
            column(6,
              plotOutput("borough_barchart")
                )
             )
         ),
```

Note these input changes for later.

Here's the code for the second tab containing our charts.

The next **`tabPanel()`** for the final tab will open directly after the final line here.

Public Health Scotland

Code Along: multiple tabs UI

```
tabPanel(title = "Information",
          p("This data has been taken from the NYC Department of health and", br(),
            "shows the popularity of dog names in New York City by gender, colour,", br(),
            "breed and borough."),
          p(tags$a("The NYC Department of Health Website", href = "https://www.health.ny.gov/"))
         )
  )
)

server <- function(input, output) {
```

Finally, the code for our final tab. After closing this **`tabPanel()`** we also close the **`navbarPage()`** and **`fluidPage()`** functions.

We can then move onto the server.

Code Along: multiple tabs Server

```
server <- function(input, output) {

   table_data <- reactive({
     nyc_dogs %>%
       filter(gender == input$gender) %>%
       filter(breed == input$breed) %>%
       filter(colour == input$colour) %>%
       filter(borough == input$borough)
   })
   output$table_output <- renderTable({
     table_data()
   })

   plot_data <- reactive({
     nyc_dogs %>%
       filter(gender == input$gender_chart) %>%
       filter(breed == input$breed_chart)
   })
   output$colour_barchart <- renderPlot({
     ggplot(plot_data()) +
       geom_bar(aes(x = colour))
   })
   output$borough_barchart <- renderPlot({
     ggplot(plot_data()) +
       geom_bar(aes(x = borough))
   })
}
shinyApp(ui = ui, server = server)
```

In our server, we're creating a reactive object for the table called "**`table_data`**" including the relevant filters we want, and we're using this for our table output.

Similarly we've created a reactive dataset for the plots called "**`plot_data`**".

We noted in our UI that for our plots **`tabPanel()`**, we had to change the **`inputId`** for our **`radioButton()`** and **`selectInput()`** drop-down. We've also changed this input name in the server.

The end product

NYC DOGS

Navigation Bar | Table | Plot | Information

Male or Female Dogs?
- (•) Male
- ( ) Female

Which colour? Brindle

Which Borough? Manhattan

Which Breed? American Pit Bull Mix / Pit Bull Mix

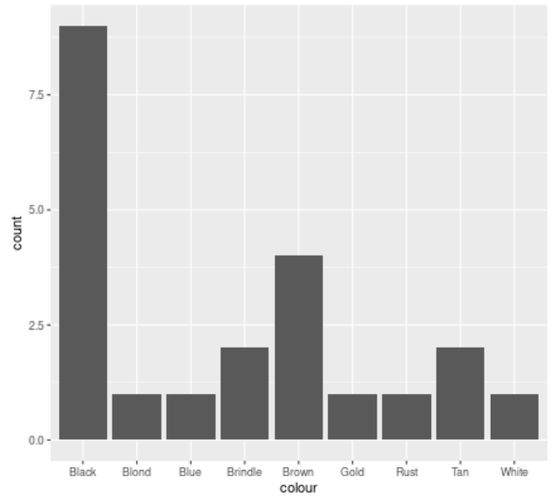
dog_name	gender	breed	birth	colour	borough
Mugsy	Male	American Pit Bull Mix / Pit Bull Mix	May-02	Brindle	Manhattan
Trouble	Male	American Pit Bull Mix / Pit Bull Mix	May-03	Brindle	Manhattan
Oden	Male	American Pit Bull Mix / Pit Bull Mix	Sep-03	Brindle	Manhattan
Honest	Male	American Pit Bull Mix / Pit Bull Mix	Jan-04	Brindle	Manhattan
Diesel	Male	American Pit Bull Mix / Pit Bull Mix	Jun-04	Brindle	Manhattan
Nudge	Male	American Pit Bull Mix / Pit Bull Mix	Jan-09	Brindle	Manhattan
Pavel	Male	American Pit Bull Mix / Pit Bull Mix	Jan-09	Brindle	Manhattan
Max	Male	American Pit Bull Mix / Pit Bull Mix	Apr-09	Brindle	Manhattan
Mitts	Male	American Pit Bull Mix / Pit Bull Mix	Jul-09	Brindle	Manhattan
Roy	Male	American Pit Bull Mix / Pit Bull Mix	Nov-10	Brindle	Manhattan
Lil	Male	American Pit Bull Mix / Pit Bull Mix	Nov-11	Brindle	Manhattan
Seagull	Male	American Pit Bull Mix / Pit Bull Mix	Dec-11	Brindle	Manhattan
Dexter	Male	American Pit Bull Mix / Pit Bull Mix	Jan-12	Brindle	Manhattan
Vader	Male	American Pit Bull Mix / Pit Bull Mix	Feb-12	Brindle	Manhattan

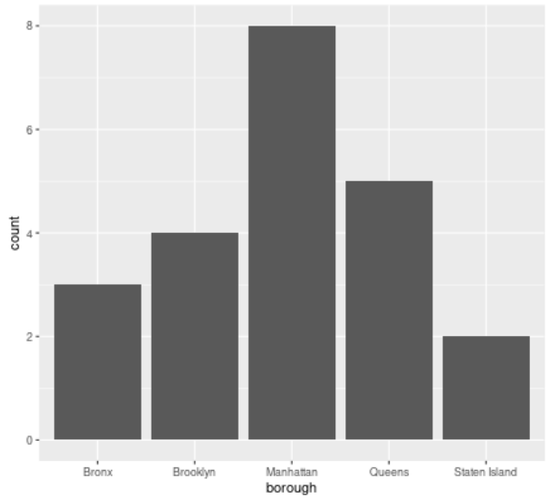
NYC DOGS

Navigation Bar | Table | Plot | Information

Male or Female Dogs?
- (•) Male
- ( ) Female

Which Breed? Akita Crossbreed





NYC DOGS

Navigation Bar | Table | Plot | Information

This data has been taken from the NYC Department of health and shows the popularity of dog names in New York City by gender, colour, breed and borough.

The NYC Department of Health Website

Public Health Scotland

Shiny themes and icons

We can use basic HTML and CSS to set fonts and themes for our dashboard and insert icons in the navigation bar for each section.

Shiny has a variety of ready made themes that can be used in your dashboard, try out the theme selector: https://shiny.rstudio.com/gallery/shiny-theme-selector.html

There are also a variety of icons that can be added to your Shiny dashboard using Font Awesome and Glyphicon: https://fontawesome.com/v5.15/icons?d=gallery&p=2

(remember that some icons may be trapped behind a paywall, but many are free!)

Shiny themes and icons

In order to use themes and icons, we need to load two more packages:

```
> library(shinythemes)
> library(shinycssloaders)
```

```
library(shiny)
library(tidyverse)
library(shinythemes)
library(shinycssloaders)

ui <- fluidPage(

  theme = shinytheme("flatly"),

  titlePanel(tags$h1("NYC DOGS")),
  navbarPage("Navigation Bar",
             tabPanel(title = "Table" icon = icon("table"),
                      fluidRow(
                        column(3,
                           radioButtons('gender',
```

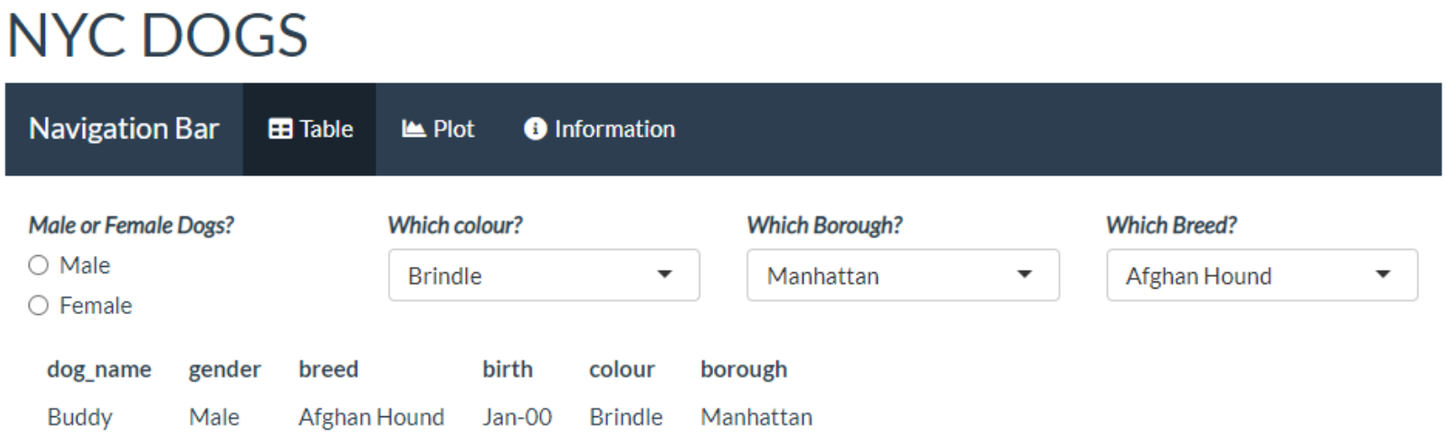
`shinytheme()` function

Wrapping title in **"`tags$h1()`"** is basic HMTL for **HEADER 1**. You could also use something like **"`tags$i("NYC DOGS")`"** to make the text *italic.*

"icon =" is an argument within the tabPanel(). It adds icons to our navigation bar tabs.
See Font Awesome link for icons.

Exercise: Shiny themes and icons

Now my Shiny app looks something like this, note the text formatting, theme and icons:



Take a look at the themes and icons links. Can you make your dashboard stand out? You can also have a play about with basic HTML to make your text bold, italic, underlined or anything else you can think of. Got good ggplot2 knowledge? Play with your chart styles!

Public Health Scotland

Practice makes perfect

Today you've created a Shiny dashboard from scratch which contains tabulated data, charts and information text tabs. You've made the data reactive to whatever the user inputs and you have used different themes, icons and text formatting to make it stand out.

If you have time before our next session, try playing around with this dashboard.

Swap your fluidRow() and column() grid layouts for something else such as sidebarLayout(), sidebarPanel() and mainPanel() to see how the layout changes.

Try adding new tabs with different charts.

Attempt to make a similar dashboard for a whole new dataset.

Public Health Scotland

Next time

- Splitting big dashboards: UI, Server and Global scripts
- Using multiple Public Health Scotland datasets
- Use of specific Public Health Scotland colours and logos
- Modals and help buttons
- Data downloads
- Deploying an app
- The importance of using GitHub

Public Health Scotland

End of day 1 – any questions?