# Part II

March 27, 2025

## 0.1 Part 2: Interactive Web Applications with Shiny

### 0.1.1 Instructor: Dr. Maryam Movahedifar

```
<img src="Uni_Logo.png" alt="Uni Logo" style="width: 100px; margin-right: 10px;">
<img src="DSC_Logo.png" alt="DSC Logo" style="width: 150px;">
```

## 0.2 Objective:

Learn how to build interactive web applications using the `Shiny` package in R, integrating dynamic user inputs and `ggplot2` visualizations.

## 0.3 Topics Covered:

- **Introduction to Shiny:**
  - Understanding the structure of a Shiny app (UI and server).
  - Creating a basic Shiny app with a simple input-output interaction.
- **Integrating `ggplot2` into Shiny:**
  - Embedding `ggplot2` plots within a Shiny app.
  - Making ggplot2 plots interactive with user inputs.
- **Adding Interactivity and Deployment:**
  - Understanding reactive programming in Shiny (inputs and outputs).
  - Deploying a Shiny app for web access using Shinyapps.io.

## 0.4 1. Introduction to Shiny

In this section, we will explore the basics of `Shiny`, an R package that allows you to build interactive web applications directly from R. You will learn how to create a simple Shiny app with user input and output elements, and understand the core components that make up a Shiny application.

### 0.4.1 Key Concepts:

- **Shiny app structure**: A Shiny app consists of two main parts:
  - **UI (User Interface)**: Defines the layout and appearance of the app, including the elements the user can interact with (like buttons, sliders, text inputs, etc.).
  - **Server**: Handles the logic of the app. It processes inputs from the user and outputs the results (like text, plots, tables).
- **Reactive programming**: The core of Shiny is reactive programming, which automatically updates outputs when inputs change. This makes Shiny ideal for building interactive apps.

---

### 0.4.2 1.1 Create a Basic Shiny App

First, let's write a simple Shiny app where we take user input (a text input) and display it as output.

[1]:
```
#install.packages("shiny")
# Load the shiny package
library(shiny)

# Then define the user interface, the HTML webpage that humans interat with. In␣
↪this case it is a page containing the words "Hello, world!"
ui <- fluidPage(
  "Hello, world!"
)

# then specifies the behaviour of our app by defining a server function. It is␣
↪currently empty, so our app does not do anything.
server <- function(input, output, session) {
}
```

[2]:
```
# Following executes shinyapp to construct and start a shiny application from␣
↪ui and server.
shinyApp(ui, server)
```

```
Listening on http://127.0.0.1:5155
```

[3]:
```
#shinyApp(ui = ui, server = server, options = list(port = 3647, host = "0.0.0.
↪0"))
```

### 0.4.3 Explanation of the Code:

**UI Component:**

- **`fluidPage()`**: This function is used to create a responsive page layout. It adjusts based on the screen size.
- **`textInput("text", "Enter text:", value = "Hello, World!")`**: This line creates a text input box where the user can type. The ID for this input is `"text"`, and the default value is `"Hello, World!"`.
- **`textOutput("output")`**: This specifies where the output will be shown. The ID `"output"` links this element to the `renderText()` output in the server function.

**Server Component:**

- **`server function`**: This function defines the logic of the Shiny app. It specifies how the app should behave based on user inputs.
- **`output$output <- renderText({ input$text })`**: This line connects the input and output. It tells Shiny to take the text entered by the user (`input$text`) and render it as text in the

`"output"` section of the UI.

**`shinyApp()`:**

- This function takes both the UI and server components and combines them to form a Shiny app. Once this function is run, it launches the app in a web browser.

## 0.5 1.2 Running and Stopping

There are a few ways you can run this app:

- Click the **Run App** button in the document toolbar.
- Use the keyboard shortcut: `Cmd/Ctrl + Shift + Enter`.
- If you're not using RStudio, you can source the entire document with `source()` or call `shiny::runApp()` with the path to the directory containing `app.R`.

Before you close the app, go back to RStudio and look at the R console. You'll notice that it says something like: Listening on http://127.0.0.1:3204

This tells you the URL where your app can be found: `127.0.0.1` is a standard address that means "this computer," and `3827` is a randomly assigned port number. You can enter that URL into any compatible web browser to open another copy of your app.

Also, notice that R is busy: the R prompt isn't visible, and the console toolbar displays a stop sign icon. While a Shiny app is running, it "blocks" the R console. This means that you can't run new commands at the R console until the Shiny app stops.

### 0.5.1 Stopping the App

You can stop the app and return access to the console using any one of these options:

- Click the **stop sign** icon on the R console toolbar.
- Click on the console, then press **Esc** (or press **Ctrl + C** if you're not using RStudio).
- Close the Shiny app window.

## 0.6 1.3 Adding UI Controls

Next, we'll add some inputs and outputs to our UI so it's not quite so minimal. We're going to make a very simple app that shows you all the built-in data frames included in the `datasets` package.

Replace your `ui` with this code:

[4]:
```
ui <- fluidPage(
  selectInput("dataset", label = "Dataset", choices = ls("package:datasets")),
  verbatimTextOutput("summary"),
  tableOutput("table")
)
```

This example uses four new functions:

- `fluidPage()` is a layout function that sets up the basic visual structure of the page.

- `selectInput()` is an input control that lets the user interact with the app by providing a value. In this case, it's a select box with the label "Dataset" and lets you choose one of the built-in datasets that come with R.
- `verbatimTextOutput()` and `tableOutput()` are output controls that tell Shiny where to put rendered output (we'll get into the how in a moment). `verbatimTextOutput()` displays code and `tableOutput()` displays tables.

Layout functions, inputs, and outputs have different uses, but they are fundamentally the same under the covers: they're all just fancy ways to generate HTML, and if you call any of them outside of a Shiny app, you'll see HTML printed out at the console. Don't be afraid to poke around to see how these various layouts and controls work under the hood.

[5]:
```
#shinyApp(ui, server)
```

## 0.7 1.4 Adding Behaviour

Next, we'll bring the outputs to life by defining them in the server function.

Shiny uses reactive programming to make apps interactive. Just be aware that it involves telling Shiny how to perform a computation, not ordering Shiny to actually go do it. It's like the difference between giving someone a recipe versus demanding that they go make you a sandwich.

We'll tell Shiny how to fill in the summary and table outputs in the sample app by providing the "recipes" for those outputs. Replace your empty server function with this:

[6]:
```
server <- function(input, output, session) {
  output$summary <- renderPrint({
    dataset <- get(input$dataset, "package:datasets")
    summary(dataset)
  })

  output$table <- renderTable({
    dataset <- get(input$dataset, "package:datasets")
    dataset
  })
}
```

The left-hand side of the assignment operator (`<-`), `output$ID`, indicates that you're providing the recipe for the Shiny output with that ID. The right-hand side of the assignment uses a specific render function to wrap some code that you provide. Each `render{Type}` function is designed to produce a particular type of output (e.g., text, tables, and plots), and is often paired with a `{type}Output` function. For example, in this app, `renderPrint()` is paired with `verbatimTextOutput()` to display a statistical summary with fixed-width (verbatim) text, and `renderTable()` is paired with `tableOutput()` to show the input data in a table.

Run the app again and play around, watching what happens to the output when you change an input.

Notice that the summary and table update whenever you change the input dataset. This dependency is created implicitly because we've referred to `input$dataset` within the output functions.

`input$dataset` is populated with the current value of the UI component with the id `dataset`, and will cause the outputs to automatically update whenever that value changes.

This is the essence of reactivity: outputs automatically react (recalculate) when their inputs change.

# 1 Basic UI

## 1.1 Shiny's Built-in Input Controls

The following sections introduce the various input controls available in Shiny, grouped by their functionality. This serves as a **quick overview** of the available options rather than an exhaustive guide.

For each input type, I'll highlight the **key parameters**, but for a complete list of arguments and advanced configurations, refer to the official documentation.

## 1.2 Free Text

Collect small amounts of text with `textInput()`, passwords with `passwordInput()`, and paragraphs of text with `textAreaInput()`.

[7]:
```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  passwordInput("password", "What's your password?"),
  textAreaInput("story", "Tell me about yourself", rows = 3)
)
```

## 1.3 Numeric Inputs

To collect numeric values, create a constrained text box with `numericInput()` or a slider with `sliderInput()`. If you supply a length-2 numeric vector for the default value of `sliderInput()`, you get a "range" slider with two ends. Sliders in Shiny are **highly customizable**, offering various options to adjust their appearance and behavior. For a detailed guide on available settings and customizations, check out the official documentation:

Shiny Slider Inputs

[8]:
```
ui <- fluidPage(
  numericInput("num", "Number one", value = 0, min = 0, max = 100),
  sliderInput("num2", "Number two", value = 50, min = 0, max = 100),
  sliderInput("rng", "Range", value = c(10, 20), min = 0, max = 100)
)
```

## 1.4 Dates

Collect a single day with `dateInput()` or a range of two days with `dateRangeInput()`. These provide a convenient calendar picker, and additional arguments like `datesdisabled` and `daysofweekdisabled` allow you to restrict the set of valid inputs.

[9]:
```
ui <- fluidPage(
  dateInput("dob", "When were you born?"),
  dateRangeInput("holiday", "When do you want to go on vacation next?")
)
```

### 1.5 Limited Choices

There are two different approaches to allow the user to choose from a prespecified set of options: `selectInput()` and `radioButtons()`.

[10]:
```
animals <- c("dog", "cat", "mouse", "bird", "other", "I hate animals")

ui <- fluidPage(
  selectInput("state", "What's your favourite state?", state.name),
  radioButtons("animal", "What's your favourite animal?", animals)
)
```

### 1.6 Radio Buttons

Radio buttons have two nice features:

- They show all possible options, making them suitable for short lists.
- Using the `choiceNames` and `choiceValues` arguments, they can display options other than plain text.

`choiceNames` determines what is shown to the user, while `choiceValues` determines what is returned in your server function.

[11]:
```
ui <- fluidPage(
  radioButtons("rb", "Choose one:",
    choiceNames = list(
      icon("angry"),
      icon("smile"),
      icon("sad-tear")
    ),
    choiceValues = list("angry", "happy", "sad")
  )
)
```

### 1.7 Dropdowns

Dropdowns created with `selectInput()` take up the same amount of space, regardless of the number of options, making them more suitable for longer lists.

You can also set `multiple = TRUE` to allow the user to select multiple elements.

[12]:
```
ui <- fluidPage(
  selectInput(
```

```
    "state", "What's your favourite state?", state.name,
    multiple = TRUE
  )
)
```

### 1.8 Checkbox Groups

There's no way to select multiple values with radio buttons, but there's an alternative that's conceptually similar: `checkboxGroupInput()`.

[13]:
```
ui <- fluidPage(
  checkboxGroupInput("animal", "What animals do you like?", animals)
)
```

### 1.9 Single Checkbox

If you want a single checkbox for a single yes/no question, use `checkboxInput()`.

[14]:
```
ui <- fluidPage(
  checkboxInput("cleanup", "Clean up?", value = TRUE),
  checkboxInput("shutdown", "Shutdown?")
)
```

### 1.10 File Uploads

Allow the user to upload a file with `fileInput()`.

[15]:
```
ui <- fluidPage(
  fileInput("upload", NULL)
)
```

### 1.11 Action Buttons

Let the user perform an action with `actionButton()` or `actionLink()`.

[16]:
```
ui <- fluidPage(
  actionButton("click", "Click me!"),
  actionButton("drink", "Drink me!", icon = icon("cocktail"))
)
```