Shiny Training Cheat Sheet

Create a new Shiny app

File > New File > Shiny Web App... Enter name for app, set as single file app (if not already selected) and choose directory. Finally, click Create.

Input elements

In `{shiny}` input elements follow a pattern like below, where the ellipsis `...` means any other arguments, and these arguments will depend on the type of input.

```
{typeOfInput}Input(inputId, label, ...)
```

For example

```
textInput("nameOfInput", "Label for input")
```

Any element you create in `{shiny}` is translated to HTML.

```
print(textInput("nameOfInput", "Label for input"))
```

```
## <div class="form-group shiny-input-container">
##   <label class="control-label" id="nameOfInput-label" for="nameOfInput">Label for input</label>
##   <input id="nameOfInput" type="text" class="form-control" value=""/>
## </div>
```

Adding an input

To add an input element, you just need to call the desired input function in your `ui` function, supplying as a minimum the `inputId` and `label` arguments. If you want to check what arguments are available for a specific input, remember to use the `?` in front of a function or press `F1` with the input function selected in your code.

A simple example is

```
ui <- fluidPage(
  textInput("myInput", "I'm a text input!")
)

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

}

shinyApp(ui, server)
```

Output elements

Output elements in `{shiny}` follow a pattern like

```
{typeOfOutput}Output(outputId, ...)
```

Just like inputs, the additional arguments vary with the output type.

For example

```
textOutput("nameOfOutput")
```

As mentioned for inputs, an output element becomes HTML when the app is run.

```
print(textOutput("nameOfOutput"))
```

```
## <div id="nameOfOutput" class="shiny-text-output"></div>
```

Accessing an input

To access, or refer to, an input, we normally use the syntax

```
input$nameOfInput
```

Shiny adds all inputs you declare to an internal named list called `input`. Because it is a list, you can also access inputs like

```
input[["nameOfInput"]]
```

Adding an output

Just as inputs are added to the `ui` function, so too with outputs. Continuing the simple app already shown, we could add an output like below.

```
ui <- fluidPage(
  textInput("myInput", "I'm a text input!"),
  textOutput("myOutput")
)

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

}

shinyApp(ui, server)
```

However, running this app would look just the same whether the output is there or not. This is because we have not told `{shiny}` just *what* the output should display.

Accessing an output

As you might expect, outputs are handled similarly to inputs; you access them like

```
output$nameOfOutput
```

or

```
output[["nameOfOutput"]]
```

Adding content to an output

For an output to actually display anything, you must add some content to it. This is done by using one of the family of `render` functions. They follow a pattern like

```
render{typeOfElement}(expr, ...)
```

For example

```
renderText("I'll appear on the page when the app is run")
```

To assign this to an output we use syntax like

```
output$myOutput <- renderText({"Some output"})
```

These output assignments belong in the `server` function of the app, like below

```
ui <- fluidPage(
  textInput("myInput", "I'm a text input!"),
  textOutput("myOutput")
)

server <- function(input, output, session) {
  output$myOutput <- renderText({"...and I'm a text output!"})
}

shinyApp(ui, server)
```

Reactivity

In `{shiny}` we have a powerful technique for writing reactive apps. Using reactive expressions we can create expressions whose output depends on their input. This allows for dynamic behaviour, as a user makes choices with the inputs of the app.

To create a reactive expression use the syntax

```
reactiveData <- reactive({

  return_data %>%
  ...
})
```

To use this expression, we use the same syntax as we do for calling a function with no arguments, `reactiveData()`

For example

```
output$table <- renderDT({

  returnData() %>%
  ...
})
```

Note that reactive expressions can only be used in a **reactive context**. This includes the `render` family of functions.

Continuing the simple app, we could have

```
ui <- fluidPage(
  textInput("myInput", "I'm a text input!"),
  textOutput("myOutput")
)

server <- function(input, output, session) {
  reactiveText <- reactive({
    upper(input$myInput)
  })
  output$myOutput <- renderText({
    reactiveText() # Reference a reactive expression - it is not a function!
  })
}

shinyApp(ui, server)
```

Debugging

As Hadley Wickham says many times in his numerous books and blogs

> Debugging Shiny code is hard

This is often due to the complexities introduced by reactivity.

Some tips for debugging include

- Start simple
- Make sure your code works outside of Shiny! One way of writing Shiny apps is known as **markdown first**; you write your code in RMarkdown initially, and only once everything (or some logical unit of the whole code) is working, only then create (or add to) a `{shiny}` app
- Use the `cat()` or `print()` functions
- Breakpoints and `browser()`
- Run `options(shiny.reactlog=TRUE)`, run the app and press `Ctrl+F3` for an interactive summary of reactive activity