Table des matières

[Further documentation: 2](#_Toc171590259)

[Module 1: Introduction to shiny: 3](#_Toc171590260)

[Getting started with shiny 3](#_Toc171590261)

[2) Example of a custom shiny App 5](#_Toc171590262)

[3) Building your shiny app 6](#_Toc171590263)

[4) Wickham Excerpts 6](#_Toc171590264)

[1 Your first shiny app 6](#_Toc171590265)

[1.4 Adding UI controls 6](#_Toc171590266)

[1.5 Adding behaviour 7](#_Toc171590267)

[1.6 Reducing duplication with rective expressions 8](#_Toc171590268)

[1.7 Summary 9](#_Toc171590269)

[1.8 Exercises 10](#_Toc171590270)

[2 Basic UI 10](#_Toc171590271)

[2.2.1 Common structure 10](#_Toc171590272)

[2.2.2 Free text 11](#_Toc171590273)

[2.2.3 Numeric inputs 11](#_Toc171590274)

[2.2.4 Dates: 12](#_Toc171590275)

[2.2.5 Limited choices 12](#_Toc171590276)

[2.2.6 File uploads 13](#_Toc171590277)

[2.2.7 Action buttons 14](#_Toc171590278)

[2.2.8 Exercises 14](#_Toc171590279)

[2.3 Outputs 15](#_Toc171590280)

[2.3.1 Text 15](#_Toc171590281)

[2.3.2 Tables 16](#_Toc171590282)

[2.3.3 Plots 17](#_Toc171590283)

[2.3.4 Downloads 17](#_Toc171590284)

[3 Basic reactivity 18](#_Toc171590285)

[3.3 Reactive programming 19](#_Toc171590286)

[Uploading Apps to shinyapps.io and R Markdown 21](#_Toc171590287)

[Module 2 Customizing Shiny apps: 22](#_Toc171590288)

[Input and output with shiny 22](#_Toc171590289)

[Using Sliders 25](#_Toc171590290)

[Shiny thems 25](#_Toc171590291)

[Shiny reference: Application layout theme 26](#_Toc171590292)

[Module 3: flexdashboards 27](#_Toc171590293)

[Getting started with flexdashboards 27](#_Toc171590294)

[Making Dashboards 27](#_Toc171590295)

[Intro to flexdashboard 28](#_Toc171590296)

# Further documentation:

Shiny:

<https://mastering-shiny.org/preface.html>

<https://shiny.posit.co/r/getstarted/shiny-basics>

<https://shiny.posit.co/r/articles/build/layout-guide/>

<http://rstudio.github.io/shinythemes/?_ga=2.132961481.679985627.1601843315-1966566028.1601473074>

UI layout functions:

<https://shiny.posit.co/r/reference/shiny/1.5.0/>

Exemples:

<https://shiny.posit.co/r/gallery>

https://shiny.posit.co/r/articles/build/sliders/

Shinyapps deployment through shiny:

<https://shiny.rstudio.com/articles/deployment-web.html>

<https://shiny.rstudio.com/articles/shinyapps.html>

maps:

<https://shiny.posit.co/r/getstarted/shiny-basics/lesson5/>

https://shiny.posit.co/r/gallery/education/ncaa-swim-team-finder/

Cheat sheets:

<https://posit.co/resources/cheatsheets/>

Runing shiny apps in production:

<https://posit.co/resources/videos/shiny-in-production-principles-practices-and-tools/>

<https://posit.co/products/enterprise/connect/>

<https://shiny.posit.co/r/articles/#deployment>

R for data sciences:

<https://r4ds.hadley.nz/>

Advanced R:

<https://adv-r.hadley.nz/>

R packages:

<https://adv-r.hadley.nz/>

Flexdashboard:

<https://rmarkdown.rstudio.com/flexdashboard/index.html>

<https://pkgs.rstudio.com/flexdashboard>

<https://bookdown.org/yihui/rmarkdown/layout.html> (all you need is there)

Shiny within flexdashboard:

<https://pkgs.rstudio.com/flexdashboard/articles/shiny.html>

# Module 1: Introduction to shiny:

## Getting started with shiny

Create a first shiny app:

Tow options:

1: writing **shinyapp** after having called shiny will write the templatefor shiny apps.

2: This way will call a more fulfilled template

Rstudio -> new file -> shiny app -> option to select wether the app needs 1 file (simple app), or two (more complicated app with distinguished files for ui and server sripts)

In our case, we chose a single file to have, this will generate a default file name app.R with a template for an interactive histogram

library(shiny)

# Define UI for application that draws a histogram

ui <- fluidPage(

# Application title

titlePanel("Old Faithful Geyser Data"),

# Sidebar with a slider input for number of bins (left part/input)

sidebarLayout(

sidebarPanel(

sliderInput(

#name of object

"bins",

#object output title

"Number of bins:",

#min and max values for the slider

min = 1,

max = 50,

#Starting value for the slider when opening app

value = 30)

),

# Show a plot of the generated distribution (right part/output)

#plotOutput takes as its argument disPlot defined down below in the server function

mainPanel(

plotOutput("distPlot")

)

)

)

# Define server logic required to draw a histogram

server <- function(input, output) {

#renderPlot function will draw whatever visualisation we config in it

output$distPlot <- renderPlot({

#Selection of the data

x <- faithful[, 2]

# generate bins based on input$bins from ui.R defined above in sliderInput function

bins <- seq(min(x), max(x), length.out = input$bins + 1)

# draw the histogram with the specified number of bins

hist(x, breaks = bins, col = 'darkgray', border = 'white',

xlab = 'Waiting time to next eruption (in mins)',

main = 'Histogram of waiting times')

})

}

# Run the application

shinyApp(ui = ui, server = server)

## 2) Example of a custom shiny App

To create our first customised application, we are going to use the already sawn template.

It is good practice to do so when using new tools in R.

Good practice is to keep data in the same file directory as your shiny code is located.

## 3) Building your shiny app

A Shiny app is built in two parts:

1. User interface
2. Server function

## Wickham Excerpts

Reading of part 1 to 3.3 of

<https://mastering-shiny.org/preface.html>

### 1 Your first shiny app

#### 1.3 Running and stopping

If you’re not using RStudio, you can (source()) the whole document, or call [shiny::runApp()](https://rdrr.io/pkg/shiny/man/runApp.html) with the path to the directory containing app.R.

When you run a shiny app on Rstudio, consol prints out #> Listening on <http://127.0.0.1:3827>

127.0.0.1 meaning app is located on “this computer”, and 3827 is a randomly assigned port number.

When developing a shiny app on Rstudio, good practice is to code, run app and reload app

As much as necessary (noting that running commands isn’t possible while the app is still running)

### 1.4 Adding UI controls

The following shunk of code is the UI part to create a simple app that allows user to navigate through the different built-in R data sets.

ui <- fluidPage(

selectInput("dataset", label = "Dataset", choices = ls("package:datasets")),

verbatimTextOutput("summary"),

tableOutput("table")

)

.fluidPage() is a layout function that set ups 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.

.verbatimTextOutput() and tableOutput() are output controls that tells the Shiny where to put rendered output

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

### 1.5 Adding behaviour

Next let’s bring the outputs to life by defining them in the server function.

ui <- fluidPage(

selectInput("dataset", label = "Dataset", choices = ls("package:datasets")),

verbatimTextOutput("summary"),

tableOutput("table")

)

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

})

}

Each {type}Output of the UI part has its equivalent in the server part as render{Type}

Here for instance, we have tableOutput({}) with the equivalent renderTable({})

Interactive apps belong to the realm of reactive programming. (outputs depends on input)

### 1.6 Reducing duplication with rective expressions

dataset <- get(input$dataset, "package:datasets")

Is declared twice in server part, in every kind of programming, this is poor practice to have duplicated code.

However get methods can’t be assigned to a variable once and being let alone because they are real time process. Neither they can be called within functions for reason sfurther explained in part 13.2.

To conter the roblem, we need a new mechanism; **reactive expressions.**

Those are created by wrapping a block of code in reactive({}) and assigning it to a variable and you use a reactive expression by calling it like a function.

This gives us:

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

# Create a reactive expression

dataset <- reactive({

get(input$dataset, "package:datasets")

})

output$summary <- renderPrint({

# Use a reactive expression by calling it like a function

summary(dataset())

})

output$table <- renderTable({

dataset()

})

}

### 1.7 Summary

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

<https://posit.co/resources/cheatsheets/>

### 1.8 Exercises

# 2 Basic UI

As we saw in the previous chapter, you use functions like [sliderInput()](https://rdrr.io/pkg/shiny/man/sliderInput.html), [selectInput()](https://rdrr.io/pkg/shiny/man/selectInput.html), [textInput()](https://rdrr.io/pkg/shiny/man/textInput.html), and [numericInput()](https://rdrr.io/pkg/shiny/man/numericInput.html) to insert input controls into your UI specification. Now we’ll discuss the common structure that underlies all input functions and give a quick overview of the inputs built into Shiny.

### 2.2.1 Common structure

Front and back end are both connected by inputID; If UI has an input with ID “name”, server function will access it with input$name.

### 2.2.2 Free text

Collect small amounts of text with [textInput()](https://rdrr.io/pkg/shiny/man/textInput.html), passwords with [passwordInput()](https://rdrr.io/pkg/shiny/man/passwordInput.html)3, and paragraphs of text with [textAreaInput()](https://rdrr.io/pkg/shiny/man/textAreaInput.html).

ui <- fluidPage(

textInput("name", "What's your name?"),

passwordInput("password", "What's your password?"),

textAreaInput("story", "Tell me about yourself", rows = 3)

)

If you want to ensure that the text has certain properties you can use [validate()](https://rdrr.io/pkg/shiny/man/validate.html), which we’ll come back to in Chapter [8](https://mastering-shiny.org/action-feedback.html#action-feedback).

### 2.2.3 Numeric inputs

o collect numeric values, create a constrained text box with [numericInput()](https://rdrr.io/pkg/shiny/man/numericInput.html) or a slider with [sliderInput()](https://rdrr.io/pkg/shiny/man/sliderInput.html).

If you supply a length-2 numeric vector for the default value of [sliderInput()](https://rdrr.io/pkg/shiny/man/sliderInput.html), you get a “range” slider with two ends.

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)

)

Generally, I recommend only using sliders for small ranges because greater the ranges are greater the difficulty to select a number is.

Sliders are extremely customisable and there are many ways to tweak their appearance. See [?sliderInput](https://rdrr.io/pkg/shiny/man/sliderInput.html) and <https://shiny.rstudio.com/articles/sliders.html> for more details.

### 2.2.4 Dates:

Collect a single day with [dateInput()](https://rdrr.io/pkg/shiny/man/dateInput.html) or a range of two days with [dateRangeInput()](https://rdrr.io/pkg/shiny/man/dateRangeInput.html). These provide a convenient calendar picker, and additional arguments like datesdisabled and daysofweekdisabled allow you to restrict the set of valid inputs.

ui <- fluidPage(

dateInput("dob", "When were you born?"),

dateRangeInput("holiday", "When do you want to go on vacation next?")

)

Date format, language, and the day on which the week starts defaults to US standards. If you are creating an app with an international audience, set format, language, and weekstart so that the dates are natural to your users.

### 2.2.5 Limited choices

There are two different approaches to allow the user to choose from a prespecified set of options: [selectInput()](https://rdrr.io/pkg/shiny/man/selectInput.html) and [radioButtons()](https://rdrr.io/pkg/shiny/man/radioButtons.html).

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)

)

Dropdowns created with [selectInput()](https://rdrr.io/pkg/shiny/man/selectInput.html) take up the same amount of space, regardless of the number of options, making them more suitable for longer options. You can also set multiple = TRUE to allow the user to select multiple elements.

ui <- fluidPage(

selectInput(

"state", "What's your favourite state?", state.name,

multiple = TRUE

)

)

If you have a very large set of possible options, you may want to use “server-side” [selectInput()](https://rdrr.io/pkg/shiny/man/selectInput.html) so that the complete set of possible options are not embedded in the UI (which can make it slow to load), but instead sent as needed by the server. You can learn more about this advanced topic at <https://shiny.rstudio.com/articles/selectize.html#server-side-selectize>.

There’s no way to select multiple values with radio buttons, but there’s an alternative that’s conceptually similar: [checkboxGroupInput()](https://rdrr.io/pkg/shiny/man/checkboxGroupInput.html).

ui <- fluidPage(

checkboxGroupInput("animal", "What animals do you like?", animals)

)

If you want a single checkbox for a single yes/no question, use [checkboxInput()](https://rdrr.io/pkg/shiny/man/checkboxInput.html):ui <- fluidPage(

checkboxInput("cleanup", "Clean up?", value = TRUE),

checkboxInput("shutdown", "Shutdown?")

)

### 2.2.6 File uploads

fileInput():

ui <- fluidPage(

fileInput("upload", NULL)

)

[fileInput()](https://rdrr.io/pkg/shiny/man/fileInput.html) requires special handling on the server side, and is discussed in detail in Chapter [9](https://mastering-shiny.org/action-transfer.html#action-transfer).

### 2.2.7 Action buttons

Let the user perform an action with [actionButton()](https://rdrr.io/pkg/shiny/man/actionButton.html) or [actionLink()](https://rdrr.io/pkg/shiny/man/actionButton.html):

ui <- fluidPage(

actionButton("click", "Click me!"),

actionButton("drink", "Drink me!", icon = icon("cocktail"))

)

#> This Font Awesome icon ('cocktail') does not exist:

#> \* if providing a custom `html\_dependency` these `name` checks can

#> be deactivated with `verify\_fa = FALSE`

Actions links and buttons are most naturally paired with [observeEvent()](https://rdrr.io/pkg/shiny/man/observeEvent.html) or [eventReactive()](https://rdrr.io/pkg/shiny/man/observeEvent.html) in your server function. You haven’t learned about these important functions yet, but we’ll come back to them in Section [3.5](https://mastering-shiny.org/basic-reactivity.html#controlling-timing-of-evaluation).

You can customise the appearance using the class argument by using one of "btn-primary", "btn-success", "btn-info", "btn-warning", or "btn-danger". You can also change the size with "btn-lg", "btn-sm", "btn-xs". Finally, you can make buttons span the entire width of the element they are embedded within using "btn-block".

The class argument works by setting the class attribute of the underlying HTML, which affects how the element is styled. To see other options, you can read the documentation for Bootstrap, the CSS design system used by Shiny: [<http://bootstrapdocs.com/v3.3.6/docs/css/#buttons>](http://bootstrapdocs.com/v3.3.6/docs/css/#buttons).

### 2.2.8 Exercises

### 2.3 Outputs

Outputs in the UI create placeholder that are later filled by the server function.

Like inputs, outputs take a unique ID as their first argument, if your UI specification creates an output with ID “plot”, you will access it in the server function with output$plot.

Each output function is coupled with a render function in the back end.

Each output function on the front end is coupled with a render function in the back end. There are three main types of output, corresponding to the three things you usually include in a report: text, tables, and plots.

### 2.3.1 Text

ui <- fluidPage(

textOutput("text"),

verbatimTextOutput("code")

)

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

output$text <- renderText({

"Hello friend!"

})

output$code <- renderPrint({

summary(1:10)

})

}

Note that the [{}](https://rdrr.io/r/base/Paren.html) are only required in render functions if need to run multiple lines of code. As you’ll learn shortly, you should do as little computation in your render functions as possible, which means you can often omit them. Here’s what the server function above would look like if written more compactly::

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

output$text <- renderText("Hello friend!")

output$code <- renderPrint(summary(1:10))

}

Note that there are two render functions which behave slightly differently:

* [renderText()](https://rdrr.io/pkg/shiny/man/renderPrint.html) combines the result into a single string, and is usually paired with [textOutput()](https://rdrr.io/pkg/shiny/man/textOutput.html)
* [renderPrint()](https://rdrr.io/pkg/shiny/man/renderPrint.html) prints the result, as if you were in an R console, and is usually paired with [verbatimTextOutput()](https://rdrr.io/pkg/shiny/man/textOutput.html).

We can see the difference in this app toy

ui <- fluidPage(

textOutput("text"),

verbatimTextOutput("print")

)

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

output$text <- renderText("hello!")

output$print <- renderPrint("hello!")

}

This is equivalent to the difference between [cat()](https://rdrr.io/r/base/cat.html) and [print()](https://rdrr.io/r/base/print.html) in base R

### 2.3.2 Tables

There are two options for displaying data frames in tables:

[tableOutput()](https://rdrr.io/pkg/shiny/man/renderTable.html) and [renderTable()](https://rdrr.io/pkg/shiny/man/renderTable.html) render a static table of data, showing all the data at once.

[dataTableOutput()](https://rdrr.io/pkg/shiny/man/renderDataTable.html) and [renderDataTable()](https://rdrr.io/pkg/shiny/man/renderDataTable.html) render a dynamic table, showing a fixed number of rows along with controls to change which rows are visible.

[tableOutput()](https://rdrr.io/pkg/shiny/man/renderTable.html) is most useful for small, fixed summaries (e.g. model coefficients); [dataTableOutput()](https://rdrr.io/pkg/shiny/man/renderDataTable.html) is most appropriate if you want to expose a complete data frame to the user. If you want greater control over the output of [dataTableOutput()](https://rdrr.io/pkg/shiny/man/renderDataTable.html), I highly recommend the [reactable](https://glin.github.io/reactable/index.html) package by Greg Lin.

ui <- fluidPage(

tableOutput("static"),

dataTableOutput("dynamic")

)

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

output$static <- renderTable(head(mtcars))

output$dynamic <- renderDataTable(mtcars, options = list(pageLength = 5))

}

### 2.3.3 Plots

You can display any type of R graphic (base, ggplot2, or otherwise) with [plotOutput()](https://rdrr.io/pkg/shiny/man/plotOutput.html) and [renderPlot()](https://rdrr.io/pkg/shiny/man/renderPlot.html):

ui <- fluidPage(

plotOutput("plot", width = "400px")

)

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

output$plot <- renderPlot(plot(1:5), res = 96)

}

Plots are special because they are outputs that can also act as inputs. [plotOutput()](https://rdrr.io/pkg/shiny/man/plotOutput.html) has a number of arguments like click, dblclick, and hover. If you pass these a string, like click = "plot\_click", they’ll create a reactive input (input$plot\_click) that you can use to handle user interaction on the plot, e.g. clicking on the plot. We’ll come back to interactive plots in Shiny in Chapter [7](https://mastering-shiny.org/action-graphics.html#action-graphics).

### 2.3.4 Downloads

You can let the user download a file with [downloadButton()](https://rdrr.io/pkg/shiny/man/downloadButton.html) or [downloadLink()](https://rdrr.io/pkg/shiny/man/downloadButton.html). These require new techniques in the server function, so we’ll come back to that in Chapter [9](https://mastering-shiny.org/action-transfer.html#action-transfer).

2.3.5 Exercises

### 3 Basic reactivity

#### 3.2 The server function

Each user gets a different session automatically created by Shiny in order to don’t let an user change another user’s UI when interacting with the same app.

##### 3.2.1 Input

Input$ID objects can only be read in server function, those are read-only reactive values.

Those objects can only be read in reactive contexts as in renderText() or reactive().

shinyApp(ui, server)

#> Error: Can't access reactive value 'count' outside of reactive consumer.

#> ℹ Do you need to wrap inside reactive() or observer()?

##### 3.2.2 Output

Output$id in UI must be related to render function in server

ui <- fluidPage(

textOutput("greeting")

)

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

output$greeting <- renderText("Hello human!")

}

Only work in reactive contexts as input objects.

### 3.3 Reactive programming

Interest of an app is to have interactivity by having both input and output objects

ui <- fluidPage(

textInput("name", "What's your name?"),

textOutput("greeting")

)

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

output$greeting <- renderText({

paste0("Hello ", input$name, "!")

})

}

Notice that Shiny automatically updates outputs when inputs change.

We don’t command Shiny how to react, we just give it recipes to follow.

#### 3.3.1 imperative vs declarative programming

The difference between command and recipes I sone of the key differences between two important styles of programming.

.Imperative programming like R or python where you directly command

.declarative programming where you express higher level goals or describe important constraint and rely on someone else to translate in into action.

With imperative code you say “Make me a sandwich”8. With declarative code you say “Ensure there is a sandwich in the refrigerator whenever I look inside of it”. Imperative code is assertive; declarative code is passive-aggressive.

However background process must be understood and will be explain along this book

#### 3.3.2 Laziness

Identifiers typos aren’t detected be aware of that

#### 3.3.3 The reactive graph

Reactive graphs are the best suited way to understand how work a reactive app.

They can be generated thanks to **reactlog.**

#### 3.3.4 Reactive expressions

Allow to avoid duplicating reactive code : reactive()

Reactive expressions take inputs and produce outputs so they have a shape that combines features of both inputs and outputs in reactive graphs.

#### 3.3.5 Execution order

The order in which reactive code is run is determined only by the reactive graph, not by its layout in the server function. However better to write code inright order for better human comprehension.

It works!

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

output$greeting <- renderText(string())

string <- reactive(paste0("Hello ", input$name, "!"))

}

3.3.6 Exercices

## Uploading Apps to shinyapps.io and R Markdown

Sources:

<https://shiny.rstudio.com/articles/deployment-web.html>

<https://shiny.rstudio.com/articles/shinyapps.html>

One way to deploy a shinyApp on the web is through shinyapps.io, and another way to do so I through an R Markdown file.

Shinyapp.io is a free service provided by Rstudio.

Don’t set directory in your script to deploy it efficiently

You must register your csv file in the same directory than your script in both cases.

For Shinyapp.io check out procedure on their website. But mainly run the following code in R console:

library(rsconnect)

rsconnect::deployApp('path/to/your/app')

#To archive an app then being able to delete it in the shinyapp.io GUI at #archived apps

rsconnect::terminateApp("APPNAME")

For deployment through Rmarkdown files first create an Rmarkdown then Shiny within config.

Must click on run document to run app.

Then click on publish only this document through upper right button. Seems that .R file must be called app.R though.

Then if Rstudio account is already linked to shinyapp.io, it can be publish through it (takes a few minutes).

# Module 2 Customizing Shiny apps:

## Input and output with shiny

UI:

Input functions are:

textInput() allow user to write in text

numericInput() lets users type numbers

sliderInput() tells users to select values from a slider

radiobuttons() presents radio buttons

checkboxInput() or checkboxGroupInput() present checkboxes with single or multiple options

Output functions are:

1. textOutput()
2. verbatimTextOutput() : output from the R console
3. tableOutput() : displays a formatted HTML to display data table
4. dataTableOutput() : displays a fancier HTML with additional tools to explore data

**dependent of DT library**

1. plotOutput() : displays plots
2. plotlyOutput (): displays plotly figures (interactive figures)

Output functions are always paired with rendering functions from the server function.

Their list is bellow with the matching index of their output function

Server:

Rendering functions:

1. RenderText()
2. RenderPrint()
3. renderTable
4. renderDataTable
5. renderPlot
6. renderPlotly

Shiny documentation:

Gallery, an extensive tutorial, reference pages, and more.

It might have been overwhelming at first, but now you should have the skills to peruse this with a little more ease.

<https://shiny.rstudio.com/>

Here's the order I would use in digesting this site:

(1) In the "Get Started" page, the video lecture is very useful, but a bit long and duplicative of the videos for this course. You might consider watching the first 40 minutes if you want a deeper review.

(2) I would read the Written Tutorials, lessons 1-7. <https://shiny.rstudio.com/tutorial/written-tutorial/lesson1/>

. These should be almost completely review for you, but it is nice to see similar content in a different format.

(3) If you haven't reviewed the Shiny Gallery yet, definitely do that. I would start with the "Shiny Demos" section. Look at the code for an example like this: <https://shiny.rstudio.com/gallery/telephones-by-region.html>

. Are you able to puzzle out the basics of how this figure works? If not, this is an indication that you need to continue to review the material.

(4) The "Articles" section provides a great deal of information, but you can probably make this a lower priority. I do call out a few individual pages of this section later in our course.

(5) The function reference page is available when you need to refresh yourself on the arguments for specific functions. You don't need to review all of this just yet. Come back to this as needed when you get stuck.

Reading “datasets” library packages is done this way:

library(datasets)

data("WorldPhones")

wp = WorldPhones

head(wp)

https://shiny.posit.co/r/gallery/start-simple/telephones-by-region/

# Rely on the 'WorldPhones' dataset in the datasets

# package (which generally comes preloaded).

library(datasets)

# Use a fluid Bootstrap layout

fluidPage(

# Give the page a title

titlePanel("Telephones by region"),

# Generate a row with a sidebar

sidebarLayout(

# Define the sidebar with one input

sidebarPanel(

selectInput("region", "Region:",

choices=colnames(WorldPhones)),

#html tag to mark a sparation between two elements

hr(),

helpText("Data from AT&T (1961) The World's Telephones.")

),

# Create a spot for the barplot

mainPanel(

plotOutput("phonePlot")

)

)

)

#Rely on the 'WorldPhones' dataset in the datasets

# package (which generally comes preloaded).

library(datasets)

# Define a server for the Shiny app

function(input, output) {

# Fill in the spot we created for a plot

output$phonePlot <- renderPlot({

# Render a barplot

barplot(WorldPhones[,input$region]\*1000,

main=input$region,

ylab="Number of Telephones",

xlab="Year")

})

}

## Using Sliders

<https://shiny.posit.co/r/articles/build/sliders/>

## Shiny thems

<http://rstudio.github.io/shinythemes/?_ga=2.132961481.679985627.1601843315-1966566028.1601473074>

## Shiny reference: Application layout theme

<https://shiny.posit.co/r/articles/build/layout-guide/>

Examples for handling visualisations in columns is available at “Module 2 /multiple visualization handling”

Shiny uses bootstrap a popular HTML & CSS framework

Best way to customize shinyapps is with R’s bslib package

This guide will teach you how to use bslib to build:

* [Sidebars](https://rstudio.github.io/bslib/articles/sidebars/index.html)
* [Multi-page apps that have a navigation bar](https://rstudio.github.io/bslib/articles/dashboards/index.html#multi-page)
* [Multi-page panels, like a tabset](https://rstudio.github.io/bslib/articles/dashboards/index.html#multi-panel)
* [Accordions](https://rstudio.github.io/bslib/reference/accordion.html)
* [Multi-column apps that leverage a grid layout](https://rstudio.github.io/bslib/reference/layout_columns.html)

Create a panel containing an application titles:

titlePanel("App name")

Creates a bottom Panel:

wellPanel()

Allows to set multiple panels:

tabsetPanels(

      tabPanel(title="Text", textOutput("tab\_text")),

      tabPanel(title="Numbers", textOutput("tab\_number"))

)

Allows to wrap input:

sidebarLayout(

sidebarPanel(

Create a dropdown menu:

selectInput(

Using themes:

Eg:

library(shinythemes)

shinyApp(

ui = fluidPage(theme = shinytheme("united"),

...

),

server = function(input, output) { }

)

<https://rstudio.github.io/shinythemes/>

# Module 3: flexdashboards

## Introduction

This part is extracted from coursea courses and section 5.1 of

<https://bookdown.org/yihui/rmarkdown/layout.html>

which contains all necessary information.

Flexdashboard is an R library.

It is just a special way of formatting an RMarkdown document

First of all install.packages(‘flexdashboard’) in console.

From there, create a new Markdown document then choose “Flex Dashboard” template

When clicking on knit, it creates an html file in repertory.

### 1 Config:

**title:** “Desired Title”

**output:**

**storyboard:** true

Enables integrating code chunks for comments related to charts.

If enabled ###whatever will no longer print out title of charts but comments in a block.

**orientation:** columns or rows

With columns, charts will be displayed vertically while with

Rows they are display horizontally.

**vertical\_layout:** fill or scroll

fill matches the all space each time window dimension is used while scroll allows user to select its dashboards.

**theme:** yeti

Multiple themes available in documentation

### 2: Layouts

**Multiple pages:**

**Declaring a page is done as follow:**

Anyname

==========================

**Declaring a page with the title Visualizations associatade with a barplot icon**

**Is done has follow:**

Visualizations {data-icon="fa-signal"}

=====================================

With the name Table and tables icon:

Tables {data-icon="fa-table"}

=====================================

**Declaring pages with different orientations:**

Page 1 {data-orientation=columns}

=====================================

**Story boards:**

**Columns:**

Declaring a column is done with

Column {data-width=**number**}

---------------------------------------

Precising size seems is not mandatory, just using Column or any name with dashes marks below is okay.

Row with Column {data-height=**number**} depending on setted orientation

Must be followed by at less one dash mark.

**Tabsets:**

Tabsets are declared as follow:

Column {.tabset}

Page {.tabset}

======================================

Declared charts within will therefore printed on distinct tabsets of the column/page

**### Will display a chart with title or with comment depending on if storyboard is set to true or not**

**Charts:**

Chart boxes are defined in the Rmarkdown style within columns as follow

###Chart

**Comments:**

###Comments

### 3: Publication

#### Without Shiny:

When no shiny documents incorporated, It is possible to **simply share the html document** generated when knitting script.

Also possible to share those through free service **RPubs.com.**

This platform is like shinyapps.io except it’s for all markdown documents.

Once registered knitted document is related to account and you only have to click o Publish button. However this is not a private document.

#### With Shiny components

Use shinyapps.io

## Shiny within flexdashboards

<https://pkgs.rstudio.com/flexdashboard/articles/shiny.html>

Note that the [shinydashboard](https://rstudio.github.io/shinydashboard/) package provides another way to create dashboards with Shiny.

1. Add runtime: shiny to the options declared at the top of the document (YAML front matter).
2. Add the {.sidebar} attribute to the first column of the dashboard to make it a host for Shiny input controls (note this step isn’t strictly required, but many Shiny based dashboards will want to do this).
3. Add Shiny inputs and outputs as appropriate.
4. When including plots, be sure to wrap them in a call to [renderPlot](https://shiny.rstudio.com/reference/shiny/latest/renderPlot.html). This is important not only for dynamically responding to changes but also to ensure that they are automatically re-sized when their container changes.

Exemple:

---

title: "Old Faithful Eruptions"

output: flexdashboard::flex\_dashboard

runtime: shiny

---

```{r global, include=FALSE}

# load data in 'global' chunk so it can be shared by all users of the dashboard

library(datasets)

data(faithful)

```

Column {.sidebar}

-----------------------------------------------------------------------

Waiting time between eruptions and the duration of the eruption for the

Old Faithful geyser in Yellowstone National Park, Wyoming, USA.

```{r}

selectInput("n\_breaks", label = "Number of bins:",

choices = c(10, 20, 35, 50), selected = 20)

sliderInput("bw\_adjust", label = "Bandwidth adjustment:",

min = 0.2, max = 2, value = 1, step = 0.2)

```

Column

-----------------------------------------------------------------------

### Geyser Eruption Duration

```{r}

renderPlot({

hist(faithful$eruptions, probability = TRUE, breaks = as.numeric(input$n\_breaks),

xlab = "Duration (minutes)", main = "Geyser Eruption Duration")

dens <- density(faithful$eruptions, adjust = input$bw\_adjust)

lines(dens, col = "blue")

})

```

### Loading Data

As described above, you should perform any expensive loading of data within the global chunk, for example:

Note that special handling of the global chunk is a recently introduced feature of the **rmarkdown** package (v1.1 or later) so you should be sure to install the latest version of rmarkdown from CRAN before using it:

**Input:**

| **R Function** | **Input Type** |
| --- | --- |
| [selectInput](https://shiny.rstudio.com/reference/shiny/latest/selectInput.html) | A box with choices to select from |
| [sliderInput](https://shiny.rstudio.com/reference/shiny/latest/sliderInput.html) | A slider bar |
| [radioButtons](https://shiny.rstudio.com/reference/shiny/latest/radioButtons.html) | A set of radio buttons |
| [textInput](https://shiny.rstudio.com/reference/shiny/latest/textInput.html) | A field to enter text |
| [numericInput](https://shiny.rstudio.com/reference/shiny/latest/numericInput.html) | A field to enter numbers |
| [checkboxInput](https://shiny.rstudio.com/reference/shiny/latest/checkboxInput.html) | A single check box |
| [dateInput](https://shiny.rstudio.com/reference/shiny/latest/dateInput.html) | A calendar to aid date selection |
| [dateRangeInput](https://shiny.rstudio.com/reference/shiny/latest/dateRangeInput.html) | A pair of calendars for selecting a date range |
| [fileInput](https://shiny.rstudio.com/reference/shiny/latest/fileInput.html) | A file upload control wizard |

**Render:**

| **R Function** | **Output Type** |
| --- | --- |
| [renderPlot](https://shiny.rstudio.com/reference/shiny/latest/renderPlot.html) | R graphics output |
| [renderPrint](https://shiny.rstudio.com/reference/shiny/latest/renderPrint.html) | R printed output |
| [renderTable](https://shiny.rstudio.com/reference/shiny/latest/renderTable.html) | Data frame, matrix, other table like structures |
| [renderText](https://shiny.rstudio.com/reference/shiny/latest/renderText.html) | Character vectors |

#### Global Sidebar

If you are creating a flexdashboard with [Multiple Pages](https://pkgs.rstudio.com/flexdashboard/articles/using.html#multiple-pages) you may want to use a single sidebar that applies across all pages. In this case you should define the sidebar using a level 1 markdown header (the same as is used to define pages).

For example, this dashboard includes a global sidebar:

---

title: "Sidebar for Multiple Pages"

output: flexdashboard::flex\_dashboard

runtime: shiny

---

Sidebar {.sidebar}

=====================================

```{r}

# shiny inputs defined here

```

Page 1

=====================================

### Chart 1

```{r}

```

Page 2

=====================================

### Chart 2

```{r}

```

### Learning More

The following articles are excellent resources for learning more about Shiny and creating interactive documents:

1. The [Shiny Dev Center](https://shiny.rstudio.com) includes extensive articles, tutorials, and examples to help you learn more about Shiny.
2. The [Introduction to Interactive Documents](https://shiny.rstudio.com/articles/interactive-docs.html) article provides a great resources for getting started with Shiny and R Markdown.
3. The R Markdown website includes additional details on the various options for [deploying interactive documents](https://rmarkdown.rstudio.com/authoring_shiny.html#deployment)

Other advanced features must be seen as module creation for complicated applications

**See Advanced:**

<https://pkgs.rstudio.com/flexdashboard/articles/shiny.html>

### Shiny and plotly in dashboards

See document in Module 3 branch of the certification.

# Shiny and flexdashboard

We cover the basics of integrating Shiny and flexdashboard in the video. Additional and advanced features are explained in the documentation:

[Flexdashboard and Shiny](https://pkgs.rstudio.com/flexdashboard/articles/shiny.html)

Keep in mind that integrating Shiny and a flexdashboard is really not more complex than including Shiny in a R Markdown report. You only need to 4 things, which are laid out very clearly in the video and reading:

(1) Add runtime: shiny to the header at the top of the document

(2) Add {.sidebar} to the first column of dashboard. Put your input functions in this column.

(3) In the next column, add output by using render functions like renderPlot.

This will get you very far towards making a complete dashboard with Shiny integration. You have more options for advanced dashboards, but just getting this together is the foundation.