R markdown and Shiny

"Processing large dataset with R"

R markdown

- Rmarkdown is a comprehensive way of creating reports which involve R.

- R markdown is a format for *literate programming* documents. It is based on *markdown*, a markup language that is widely used to generate html pages.

- Literate programming weaves instructions, documentation and detailed comments in between machine executable code, producing a document that describes the program.

- Unlike a word processor, such as Microsoft Word, where what you see is what you get, with R markdown, you need to *compile* the document into the final report.

How it works

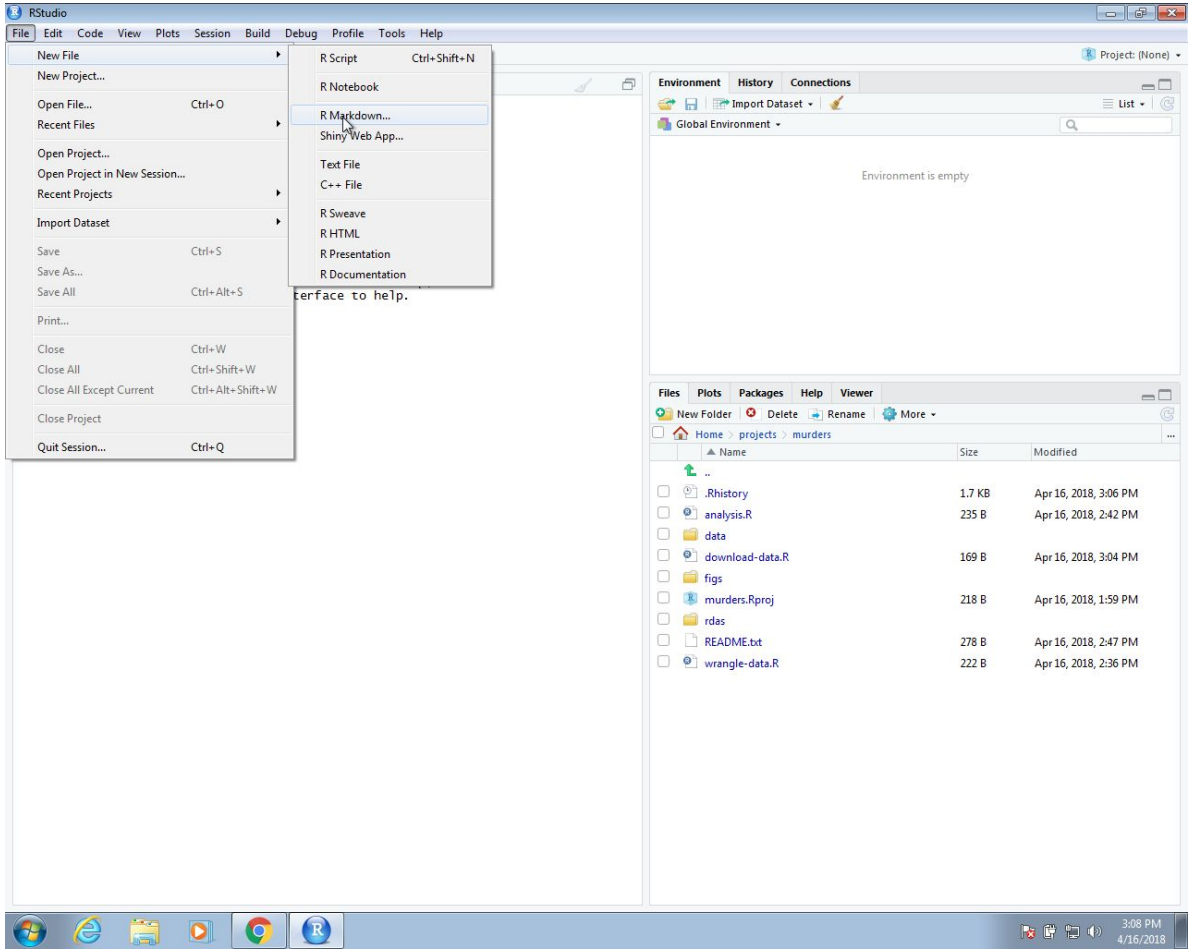
When you run render, R Markdown feeds the .Rmd file to knitr, which executes all of the code chunks and creates a new markdown (.md) document which includes the code and it's output.

The markdown file generated by knitr is then processed by pandoc which is responsible for creating the finished format.

This may sound complicated, but R Markdown makes it extremely simple by encapsulating all of the above processing into a single render function.

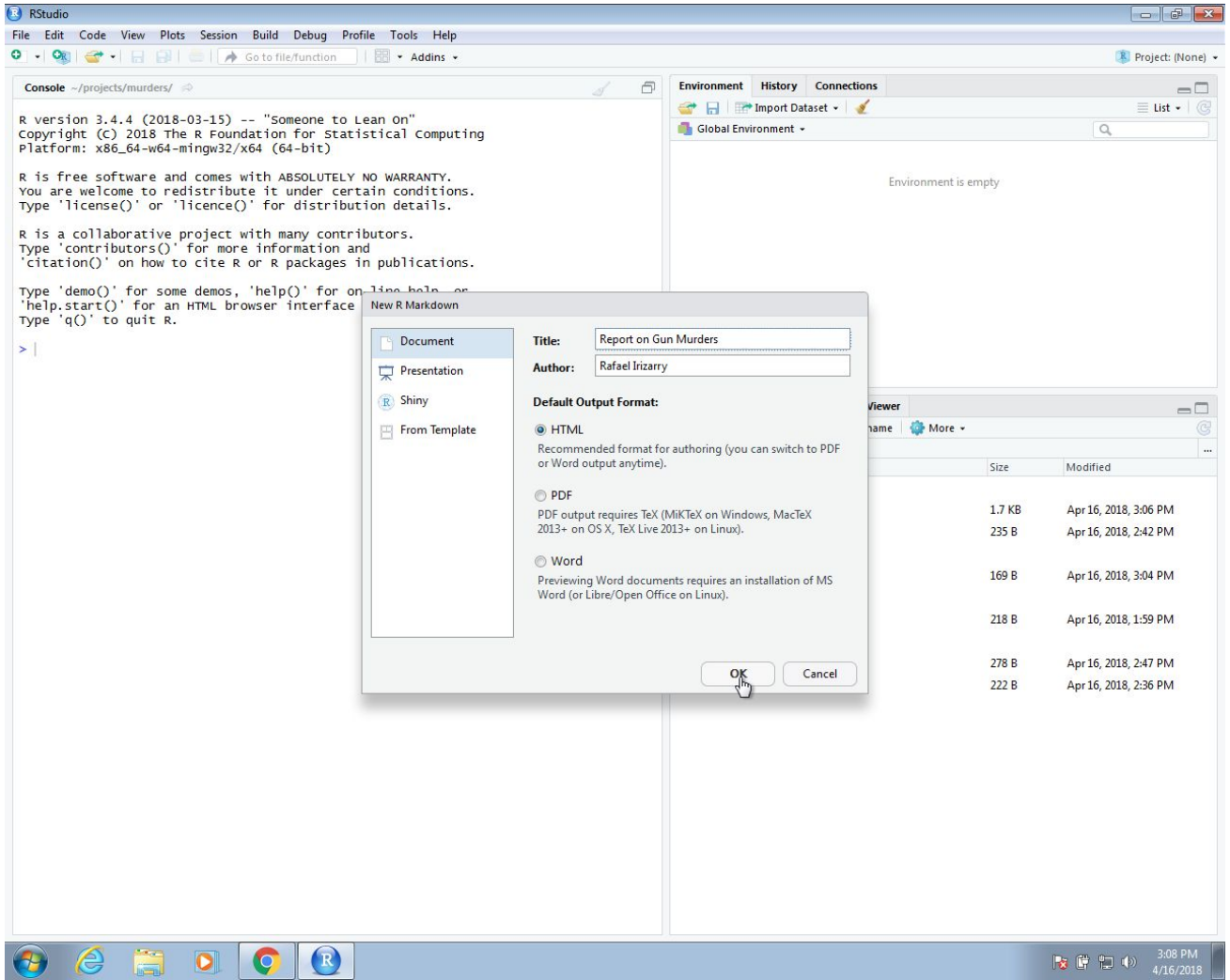
Let’s start

In RStudio, you can start an R markdown document by clicking on *File*, *New File*, the *R Markdown*, like this:



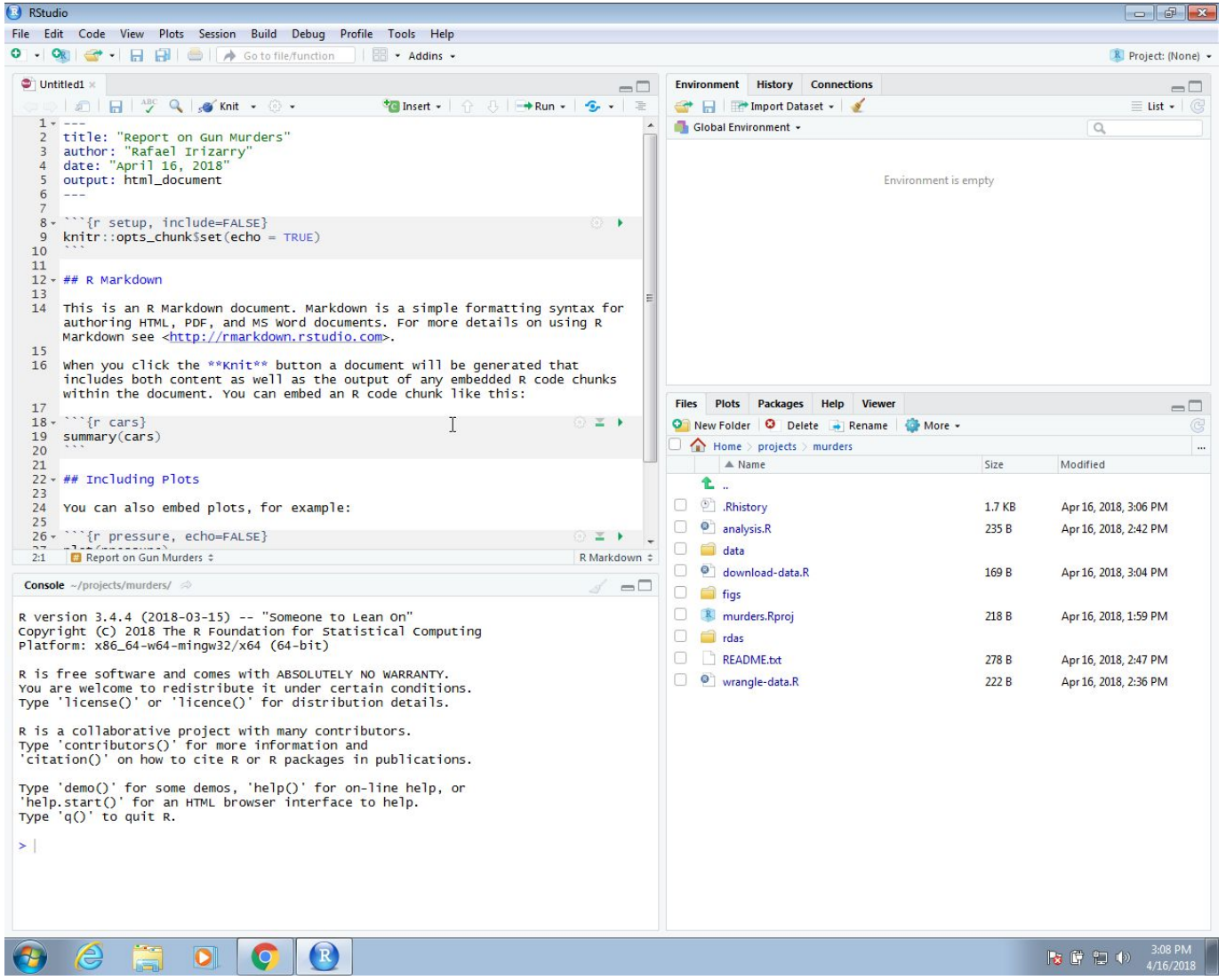
Project initialization

You will then be asked to enter a title and author for your document. Here we hypothesized to prepare a report on gun murders so we give it an appropriate name. You can also decide what format you would like the final report to be in: HTML, PDF, or Microsoft Word.
Later, we can easily change this, but here we select html as it is the preferred format for debugging purposes:



Project initialization

This will generate a template file:



The header

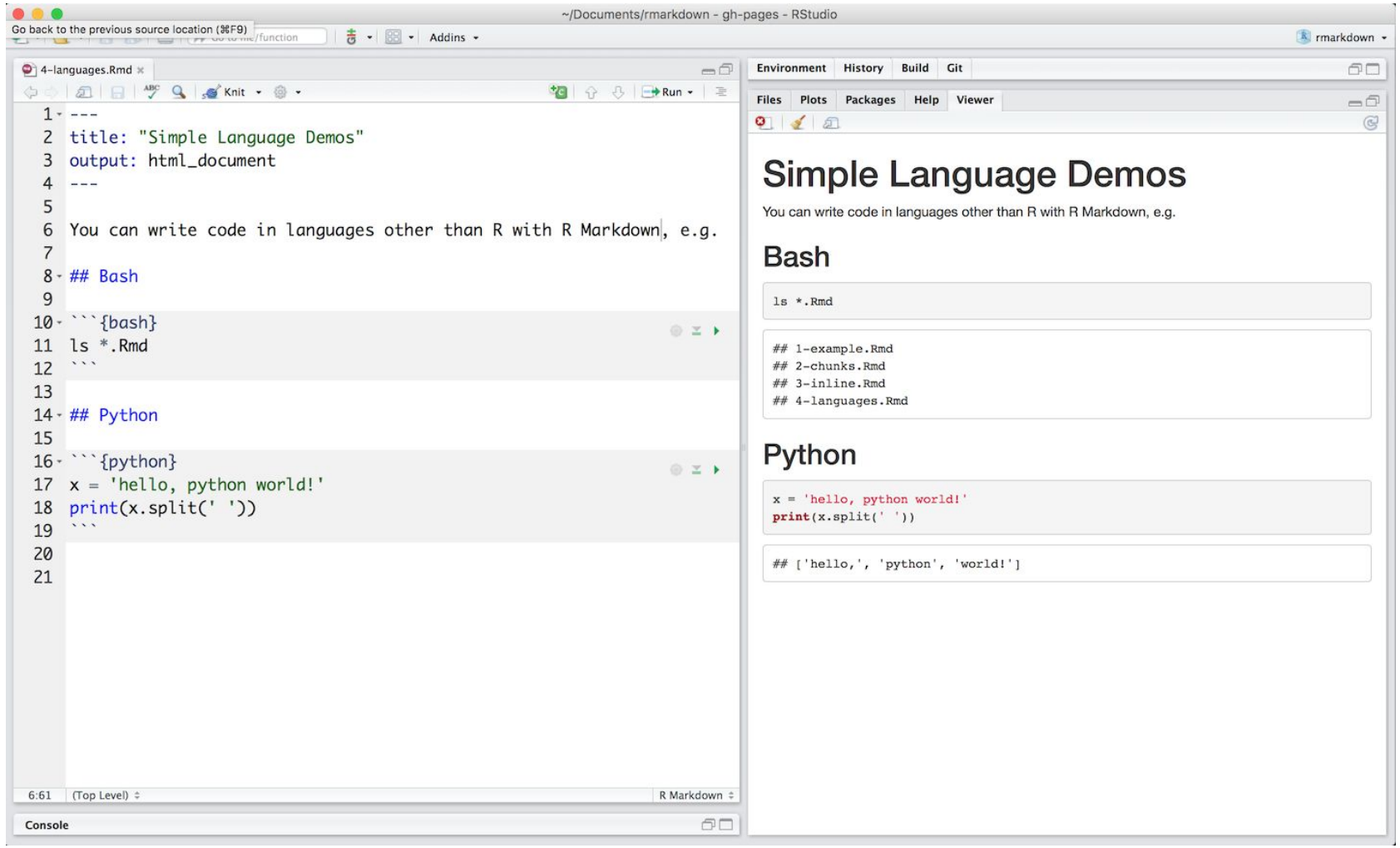
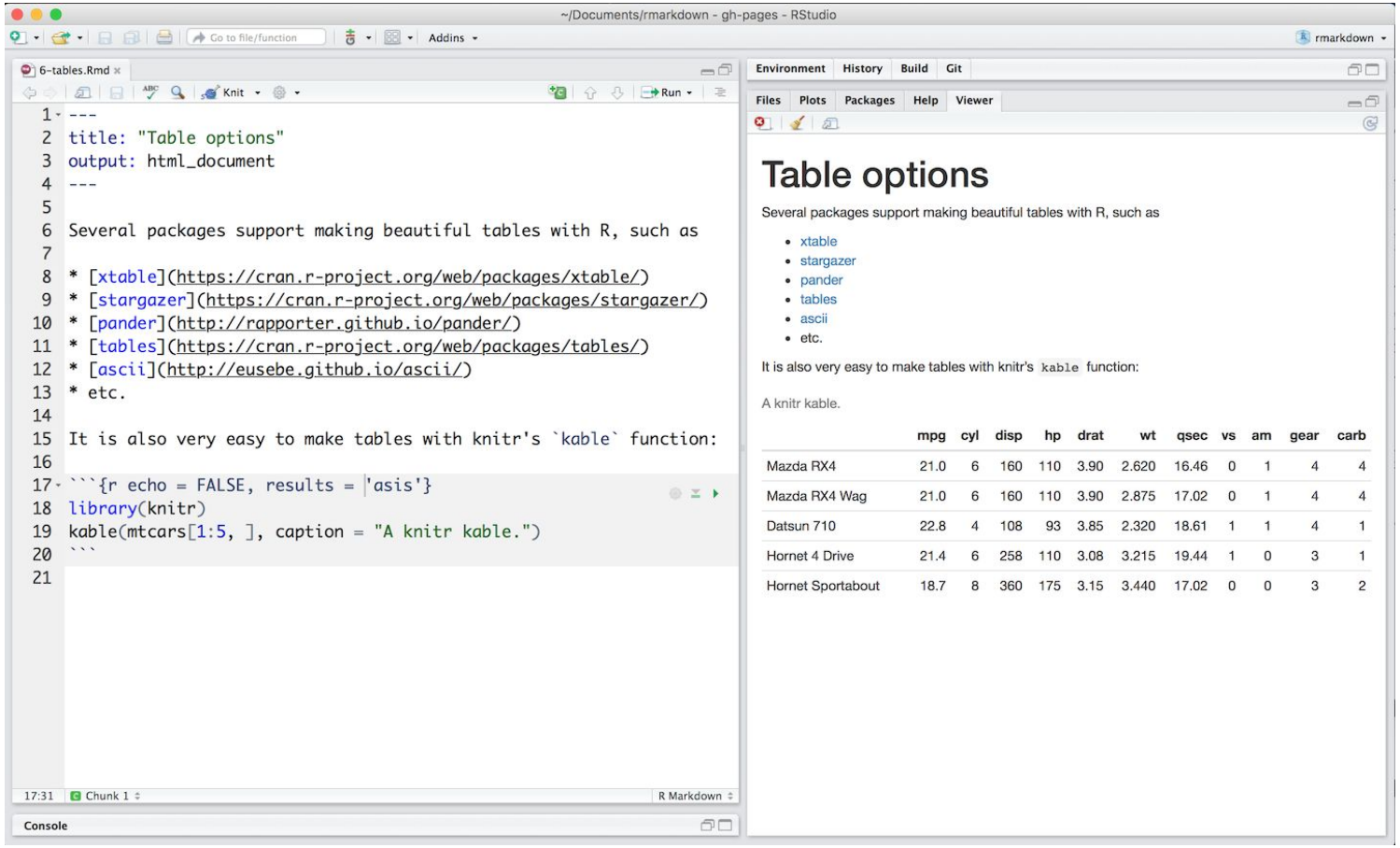
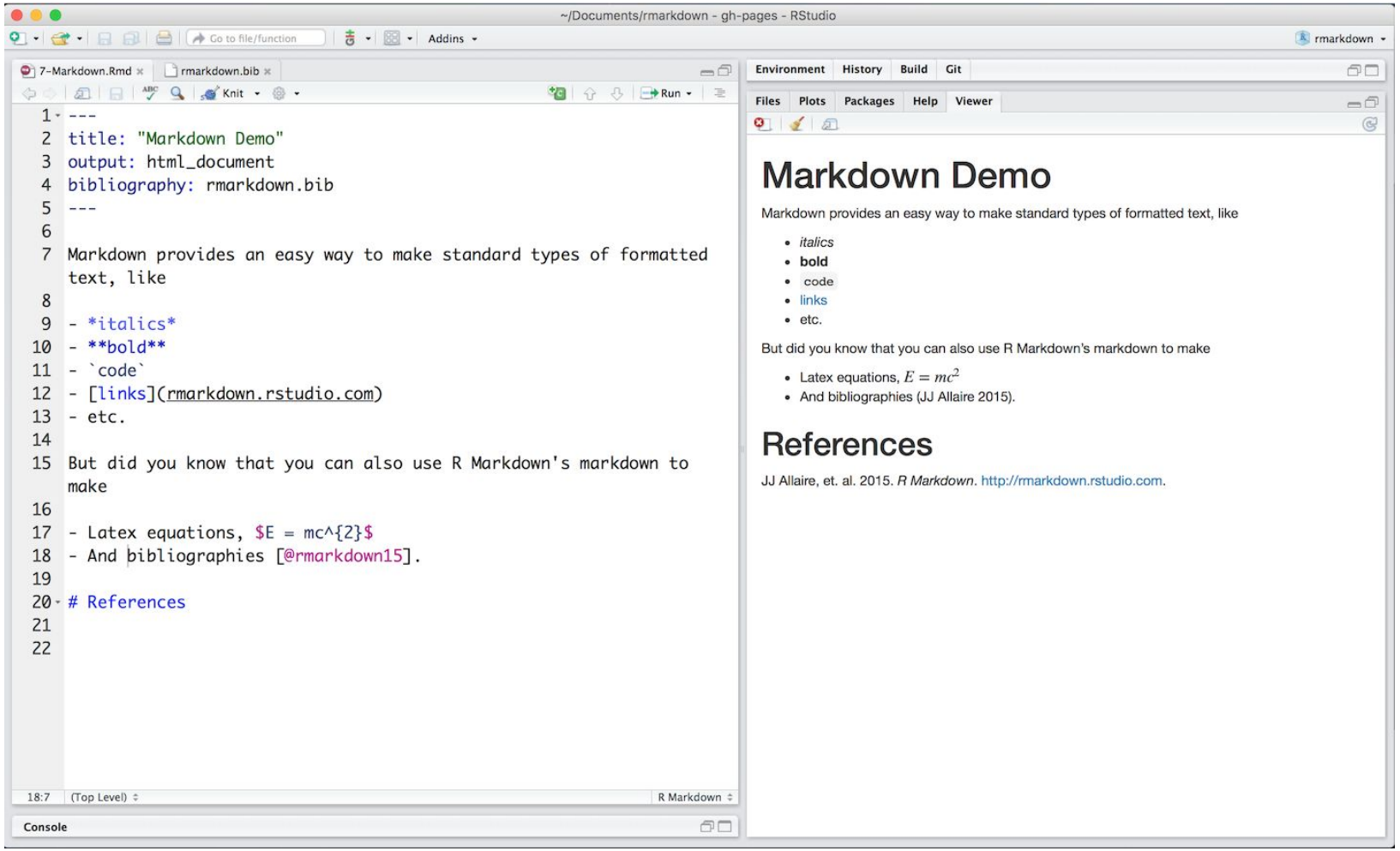
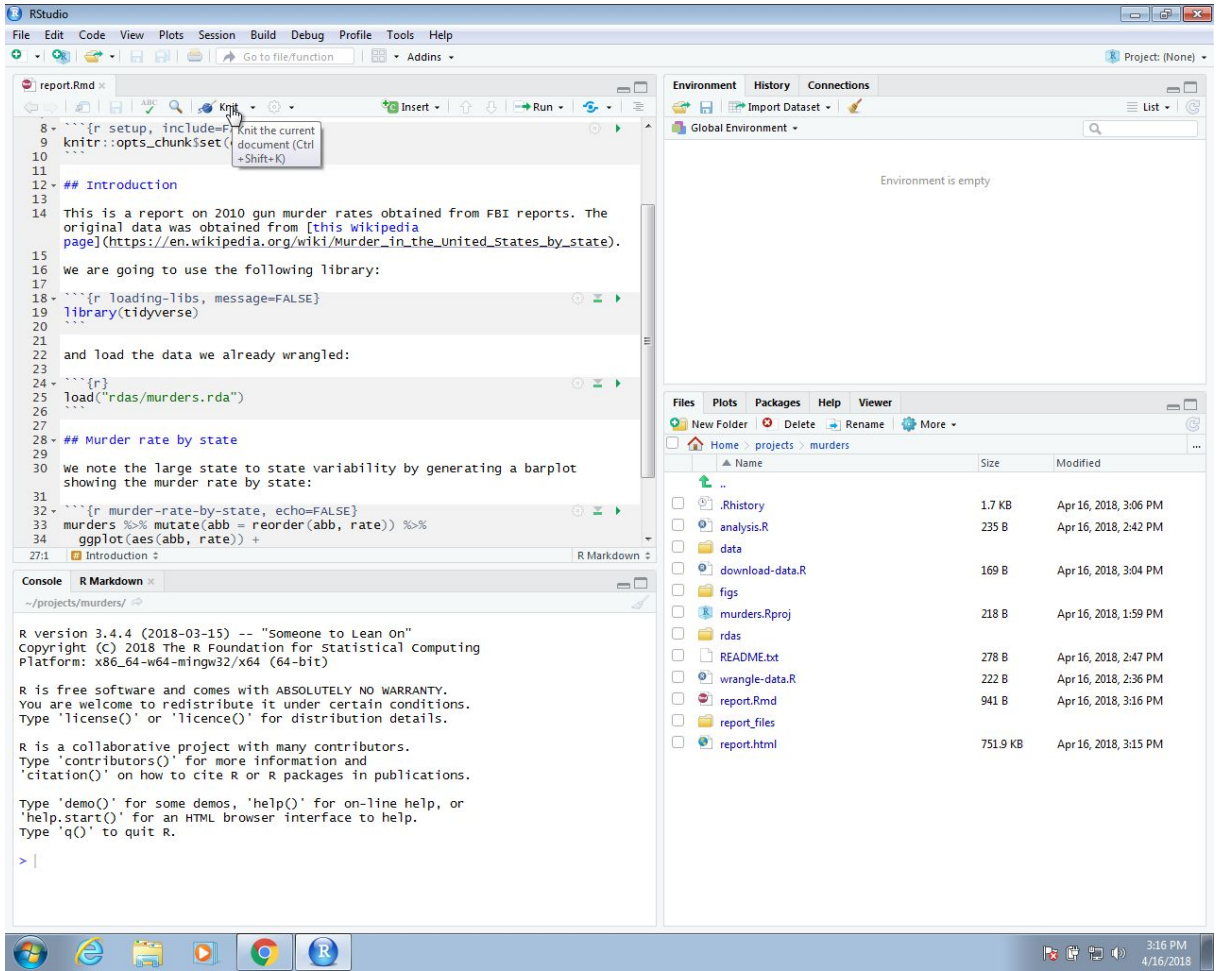
At the top you see:

```
---
title: "Report on Gun Murders"
author: "Rafael Irizarry"
date: "April 16, 2018"
output: html_document
---
```

The things between the --- is the header. We actually don't need a header, but it is often useful. You can define many other things in the header than what is included in the template.
The one parameter that we will highlight is output. By changing this to, say, pdf_document, we can control the type of output that is produced when we compile.

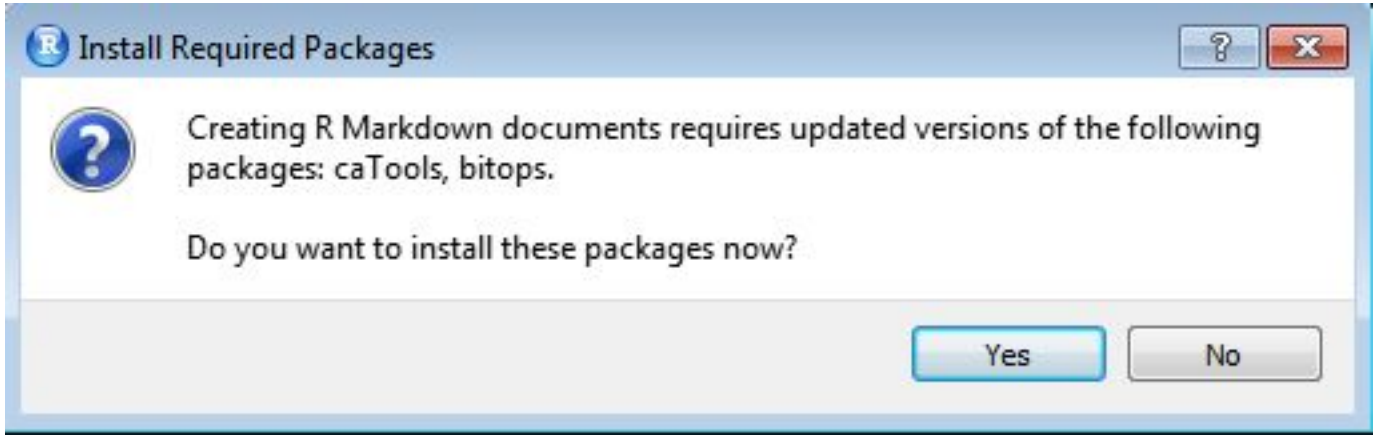
R code chunks

In various places in the document, we see something like this:

```
```{r}
summary(pressure)
```
```

These are the code **chunks**. When you compile the document, the R code inside the chunk, in this case summary(pressure), will be evaluated and **the result included** in that position in the final document.
To add your own R chunks, you can type the characters above quickly with the key binding **command-option-i** on the Mac and **Ctrl-Alt-i** on Windows.

We recommend getting into the habit of **adding a label** to the R code chunks. This will be very useful when debugging, among other situations. You do this by adding a descriptive word like this:

```
```{r – summary pressure}
summary(pressure)
```
```

Chunk Options

- include = FALSE **prevents code and results** from appearing in the finished file. R Markdown still runs the code in the chunk, and the results can be used by other chunks.

- echo = FALSE prevents **code, but not the results** from appearing in the finished file. This is a useful way to embed figures.

- the eval option disables the **execution** of the code.

- message = FALSE prevents **messages** that are generated by code from appearing in the finished file.

- warning = FALSE prevents **warnings** that are generated by code from appearing in the finished.

- fig.cap = "..." adds a **caption** to graphical results.

See the R Markdown Reference Guide for a complete list of knitr chunk options

Code Languages

.Rmd file executes code in bash and python.



- Python
- SQL
- Bash
- Rcpp
- Stan
- JavaScript
- CSS

Tables

By default, R Markdown displays data frames and matrixes as they would be in the R terminal (in a monospaced font). If you prefer that data be displayed with additional formatting you can use the knitr::kable function



Markdown Basics

Format the text in your R Markdown file with [Pandoc's Markdown](), a set of markup annotations for plain text files. When you render your file, Pandoc transforms the marked up text into formatted text in your final file format



- Headers
- Lists
- Links
- Images
- Block quotes
- Latex equations
- Horizontal rules
- Tables
- Footnotes
- Bibliographies and Citations
- Slide breaks
- Italicized text
- Bold text
- Superscripts
- Subscripts
- Strikethrough text

Markdown Basics

- the # allows to organize the document in sections, subsections, … The more you add #, the more you go down into the subsections, subsubsections, …

- the * allows to higlight some part of the text: if you use one *tyty* it produces *tyty*. If you use **tyty**, the text **tyty** will be in bold.

- the > allows to create some “citation / remark box”: > The citation that I would like to highlight!

- It is possible to provide online code using single quote. As an exemple: ls().

knitR

We use the **knitR** package to compile R markdown documents. The specific function used to compile is the knit function, which takes a filename as input. RStudio provides a button that makes it easier to compile the document.



Example

````
--- title: "Report on Gun Murders"
author: "Rafael Irizarry"
date: "`r format(Sys.Date())`"
output: github_document
---
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
## Introduction
This is a report on 2010 gun murder rates obtained from FBI reports. The original data was obtained from [this Wikipedia page](https://en.wikipedia.org/wiki/Murder_in_the_United_States_by_state). We are going to use the following library:
```{r loading-libs, message=FALSE}
library(tidyverse)
```
and load the data we already wrangled:
```{r} load("rdas/murders.rda")
```
## Murder rate by state
We note the large state to state variability by generating a barplot showing the murder rate by state:
```{r murder-rate-by-state, echo=FALSE}
murders %>% mutate(abb = reorder(abb, rate)) %>%
ggplot(aes(abb, rate)) + geom_bar(width = 0.5, stat = "identity", color = "black") + coord_flip()
```
````

Knit button

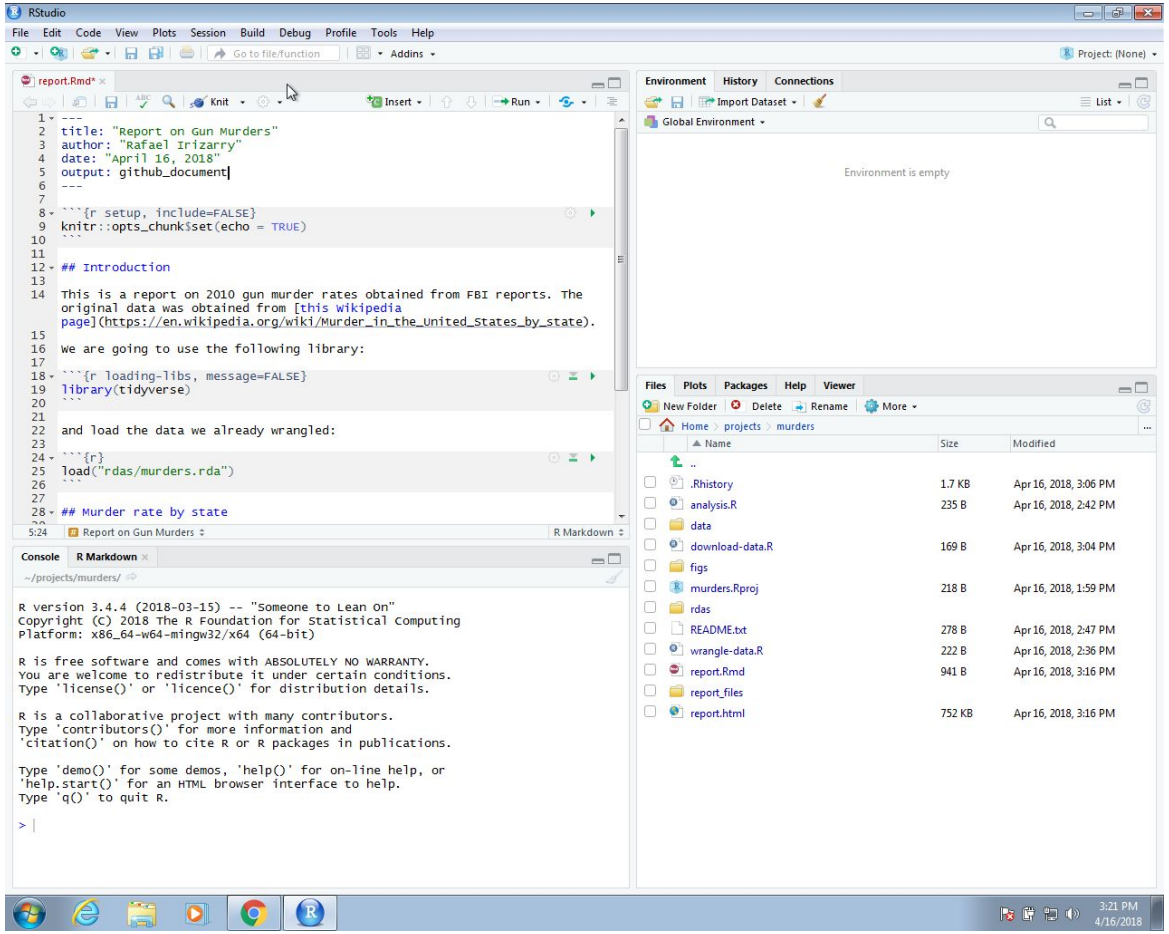
The first time you click on the *Knit* button, a dialog box may appear asking you to **install packages** you need:



Once you have installed the packages, clicking the *Knit* will **compile** your R markdown file and the resulting document will pop-up:
This produces an html document which you can see in your working directory. To view it, open a terminal and list the files. You can open the file in a browser and use this to present your analysis. You can also produce a PDF or Microsoft document by changing:
output: html_document to output: pdf_document or output: word_document.

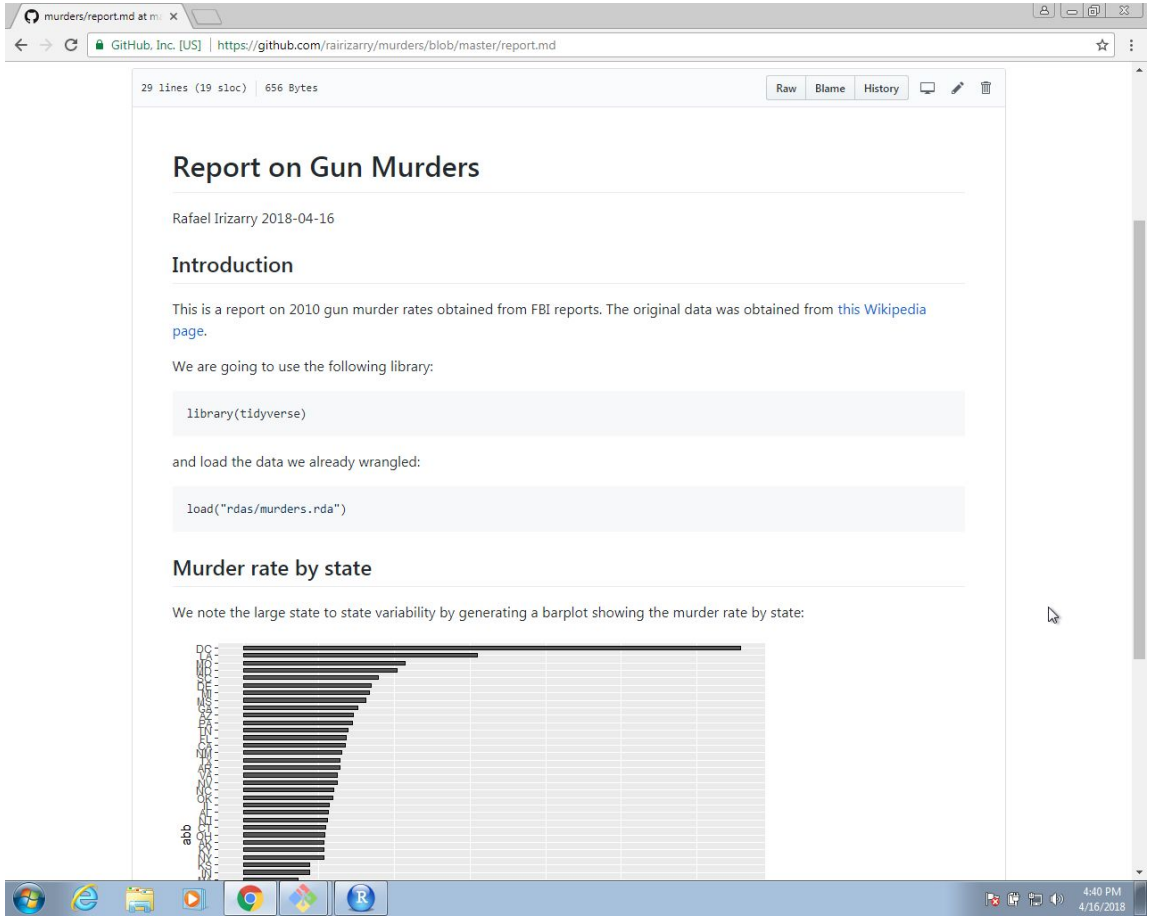
github_document

We can also produce documents that render on GitHub using output: github_document like this:



Web report

This will produce a markdown file, with suffix **md**, that renders in GitHub. Because we have uploaded these files to GitHub, you can click on the **md** file and you will see the report as a webpage:



Slide presentation

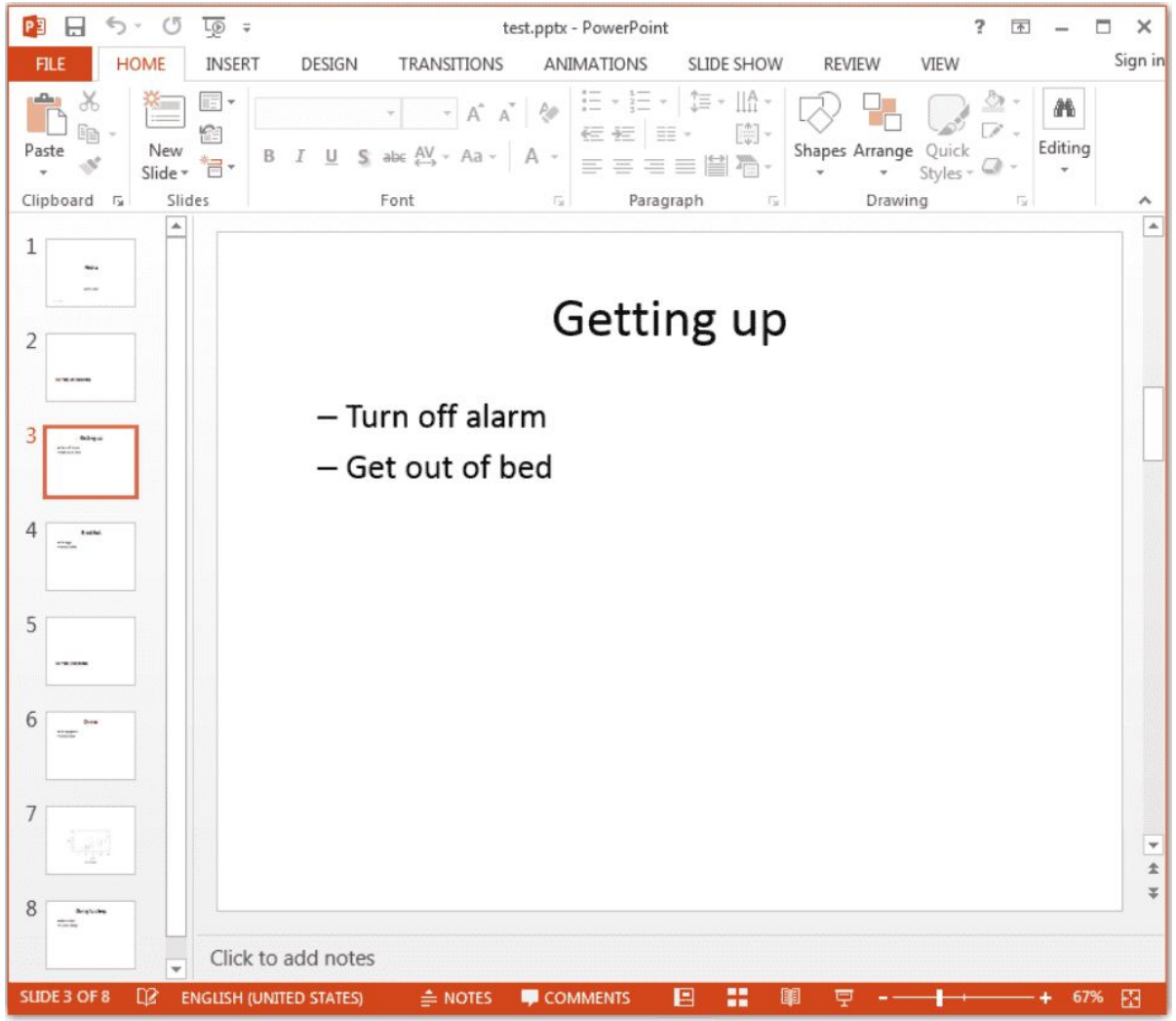
R Markdown renders to four presentation formats:

- beamer_presentation - PDF presentations with beamer
- ioslides_presentation - HTML presentations with ioslides
- slidy_presentation - HTML presentations with slidy
- powerpoint_presentation - PowerPoint presentation
- revealjs::revealjs_presentation - HTML presentations with reveal.js

Each format will intuitively divide your content into slides, with a new slide beginning at each first or second level header.

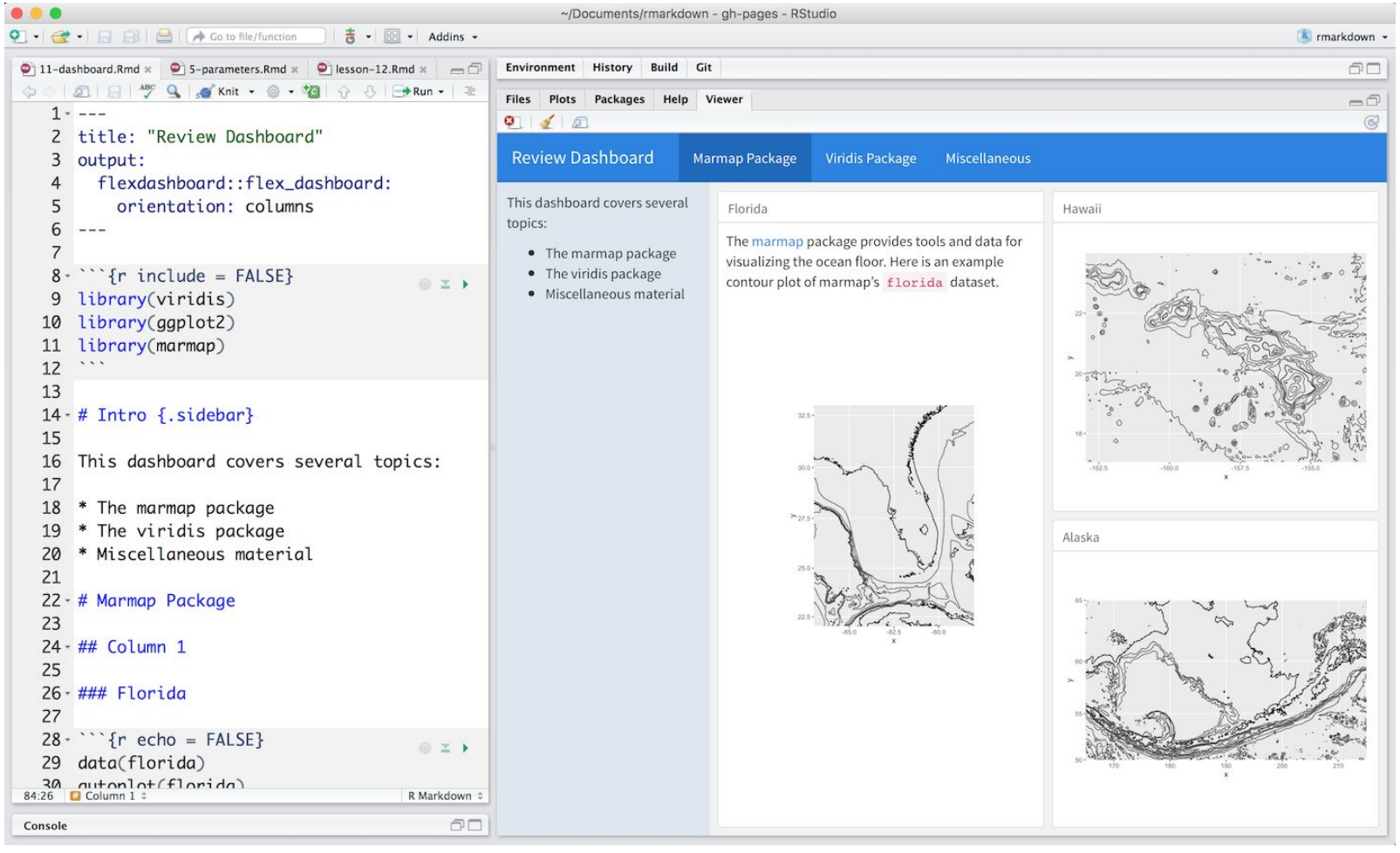
Power Point

```
--- title: "Habits"
author: John Doe
date: March 22, 2005
output: powerpoint_presentation
---
# In the morning
## Getting up
 -    Turn off alarm
 -    Get out of bed
## Breakfast
 -    Eat eggs
 -    Drink coffee
# In the evening
## Dinner
 -    Eat spaghetti
 -    Drink wine
---
```{r, cars, fig.cap="A scatterplot.", echo=FALSE}
plot(cars)
```
Going to sleep
 -    Get in bed
 -    Count sheep
```



Dashboard

Dashboards are a useful way to communicate large amounts of information visually and quickly. Create one with the flexdashboard::flex_dashboard output format



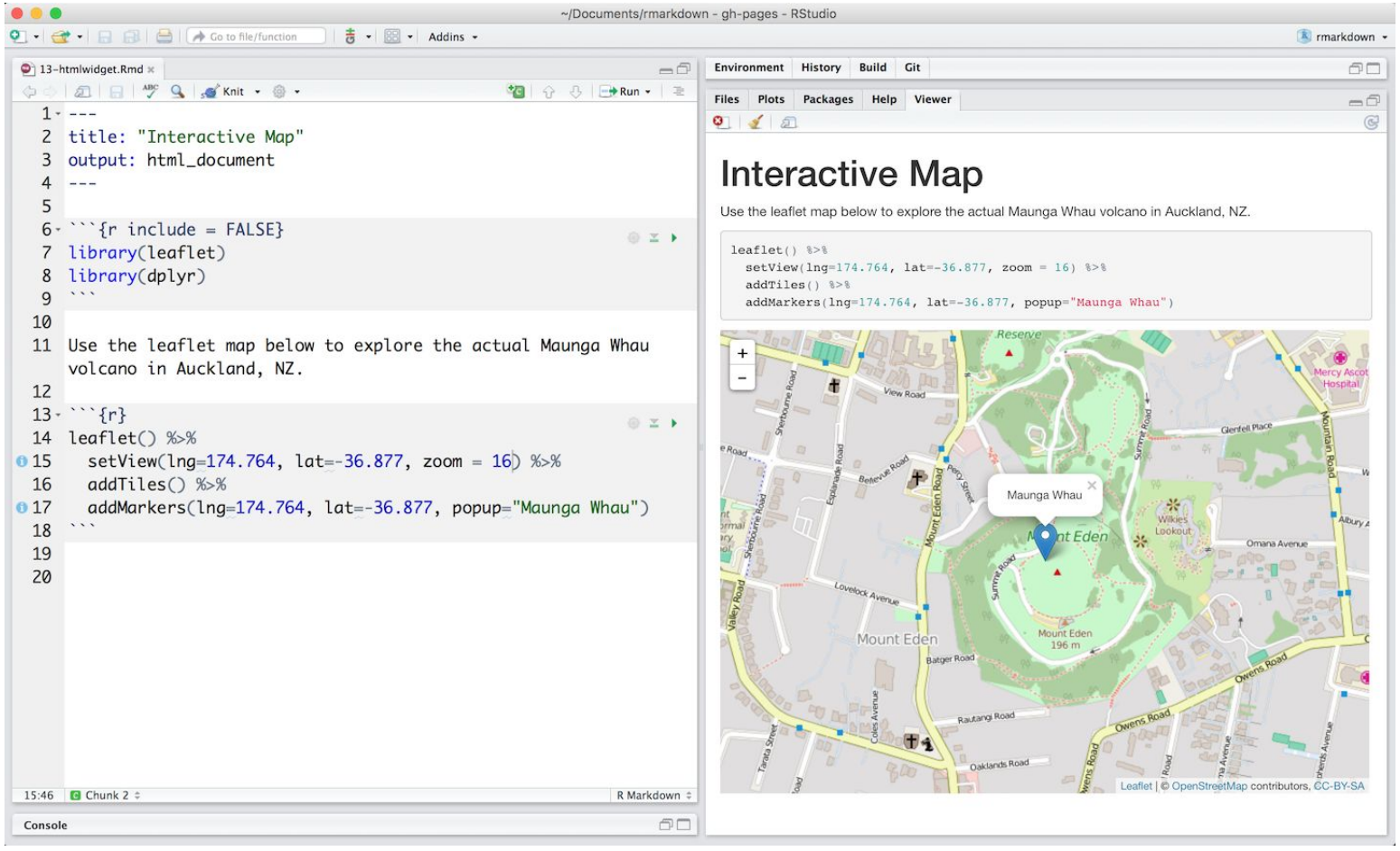
Flexdashboard makes it easy to organize your content into a visual layout:

- Each Level 1 Header (#) begins a new page in the dashboard.
- Each Level 2 Header (##) begins a new column.
- Each Level 3 Header (###) begins a new box.

Interactive documents

Markdown documents are a perfect platform for interactive content. To make your documents interactive, add:

1. Interactive JavaScript visualizations based on [htmlwidgets](#) or
2. Reactive components made with [Shiny](#)



Shiny

Shiny is an R package that makes it easy to build interactive web apps straight from R.
You can host standalone apps on a webpage or embed them in R Markdown documents or build dashboards.
You can also extend your Shiny apps with CSS themes, htmlwidgets, and JavaScript actions.

web apps was for R users:

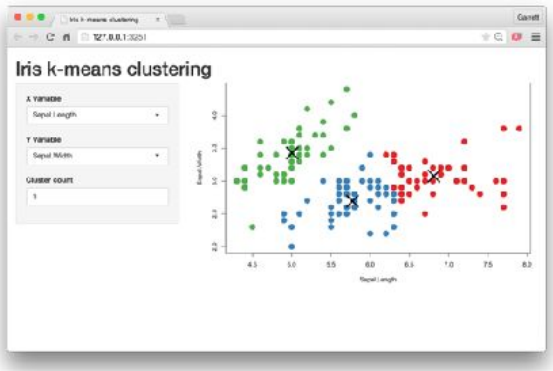
In the past

- You need a deep knowledge of web technologies like HTML, CSS and JavaScript.
- Making complex interactive apps requires careful analysis of interaction flows to make sure that when an input changes, only the related outputs are updated.

With shiny

- Providing a carefully curated set of user interface (**UI** for short) functions that generate the HTML, CSS, and JavaScript that you need for common tasks.
- Introducing a new style of programming called **reactive programming** which automatically tracks the dependencies of a code chunk.

Every Shiny app is maintained by a computer running R



Server Instructions

User Interface (UI)

Introduction

Two key components of every Shiny app:

- the **UI** (short for user interface) which defines how your app ***looks***,
- the **server function** which defines how you app ***works***. Shiny uses reactive programming to **automatically update outputs** when inputs change so we'll finish off the chapter by learning the third important component of Shiny apps: reactive expressions.

```
install.packages("shiny")
library(shiny)
```

Build your app

Add elements to your app as arguments to `fluidPage()`

```
ui <- fluidPage(
  # *Input() functions,
  # *Output() functions
)
```

Inputs

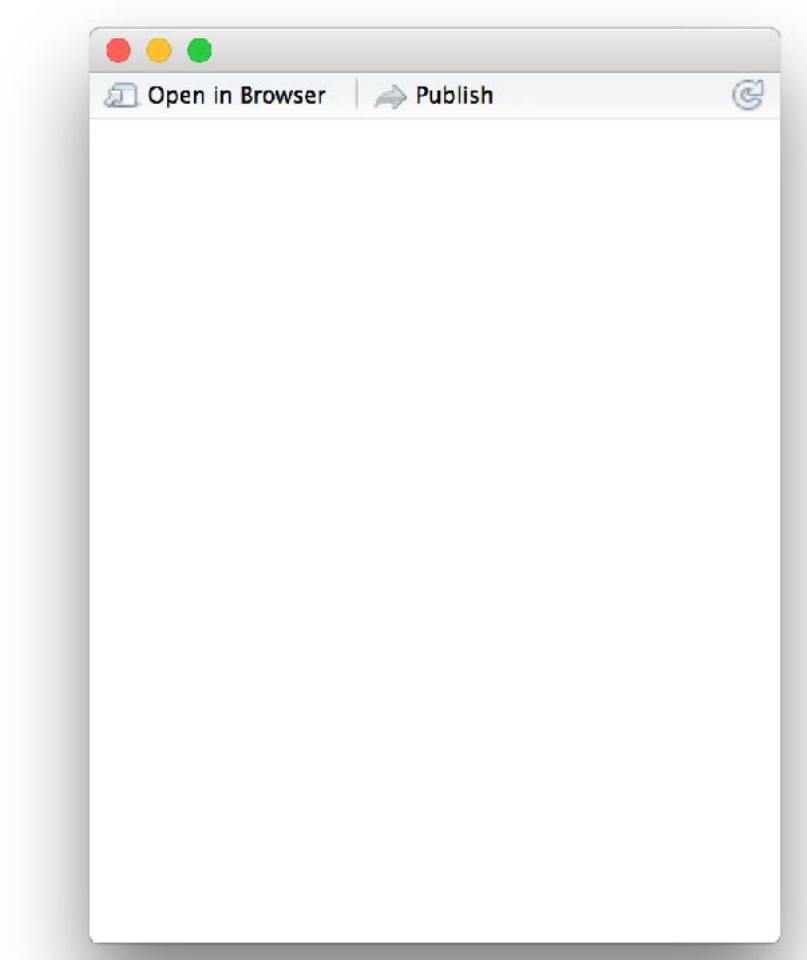
```
library(shiny)
ui <- fluidPage(



)

server <- function(input, output) {}

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

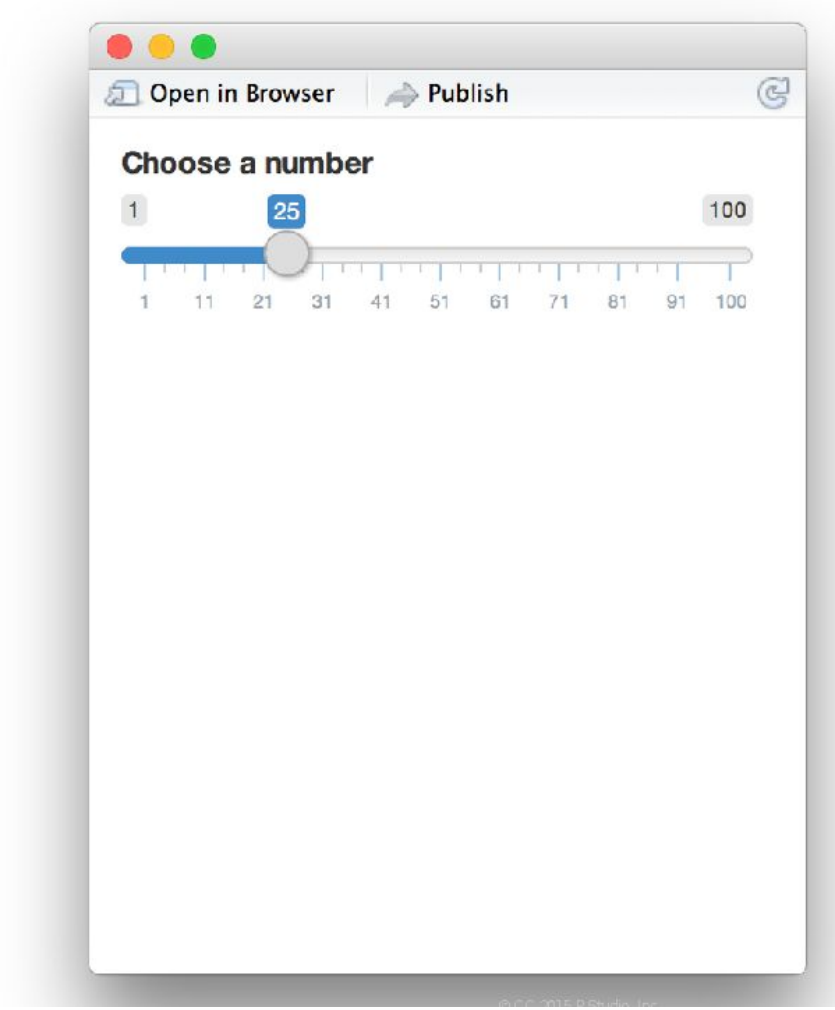


Inputs

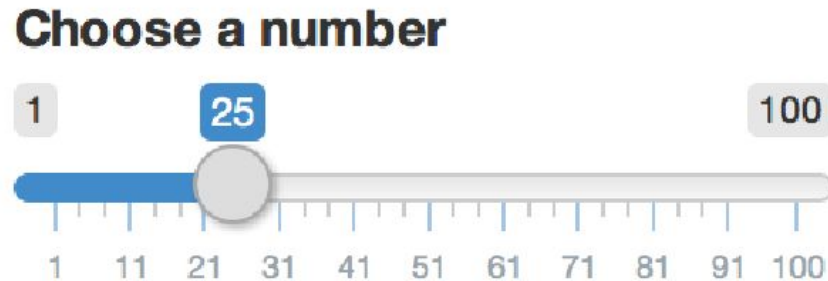
```
library(shiny)
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100)
)

server <- function(input, output) {}

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



Syntax




```
sliderInput(inputId = "num", label = "Choose a number", ...)
```

input name
(for internal use)

Notice:
Id not ID

label to display

input specific arguments

?sliderInput

Other inputs

Buttons

Action

Submit

`actionButton()`
`submitButton()`

Single checkbox

Choice A

`checkboxInput()`

Checkbox group

Choice 1
Choice 2
Choice 3

`checkboxGroupInput()`

Date input

2014-01-01

`dateInput()`

Date range

2014-01-24 to 2014-01-24

`dateRangeInput()`

File input

Choose File No file chosen

`fileInput()`

Numeric input

1

`numericInput()`

Password Input

`passwordInput()`

Radio buttons

Choice 1
Choice 2
Choice 3

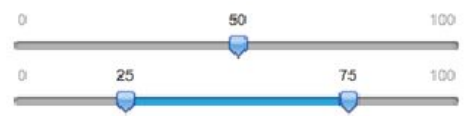
`radioButtons()`

Select box

Choice 1

`selectInput()`

Sliders



`sliderInput()`

Text input

Enter text...

`textInput()`

Outputs

To display output, add it to `fluidPage()` with an `*Output()` function

```
plotOutput("hist")
```

Outputs

Function	Inserts
dataTableOutput()	an interactive table
htmlOutput()	raw HTML
imageOutput()	image
plotOutput()	plot
tableOutput()	table
textOutput()	text
uiOutput()	a Shiny UI element
verbatimTextOutput()	text

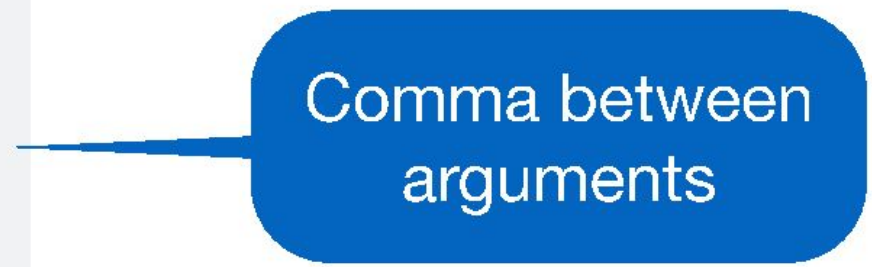
Outputs

```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

server <- function(input, output) {}

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



how to assemble inputs into outputs ?

Server function

1 Save objects to display to output$

```
server <- function(input, output) {
  output$hist <- # code

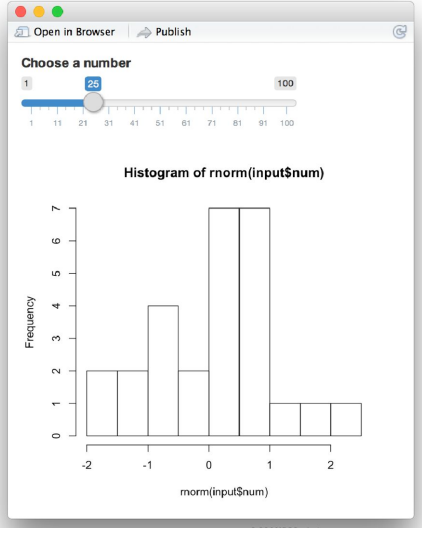

}
```

2 Build objects to display with **render*()**

```
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(100))
  })
}
```

3 Access **input** values with input$

```
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}
```



Render

render*()

Builds reactive output to display in UI

```
renderPlot({ hist(rnorm(100)) })
```

type of object to build

code block that builds the object

Use the **render*()** function that creates the type of output you wish to make.

function	creates
renderDataTable()	An interactive table (from a data frame, matrix, or other table-like structure)
renderImage()	An image (saved as a link to a source file)
renderPlot()	A plot
renderPrint()	A code block of printed output
renderTable()	A table (from a data frame, matrix, or other table-like structure)
renderText()	A character string
renderUI()	a Shiny UI element

Two inputs

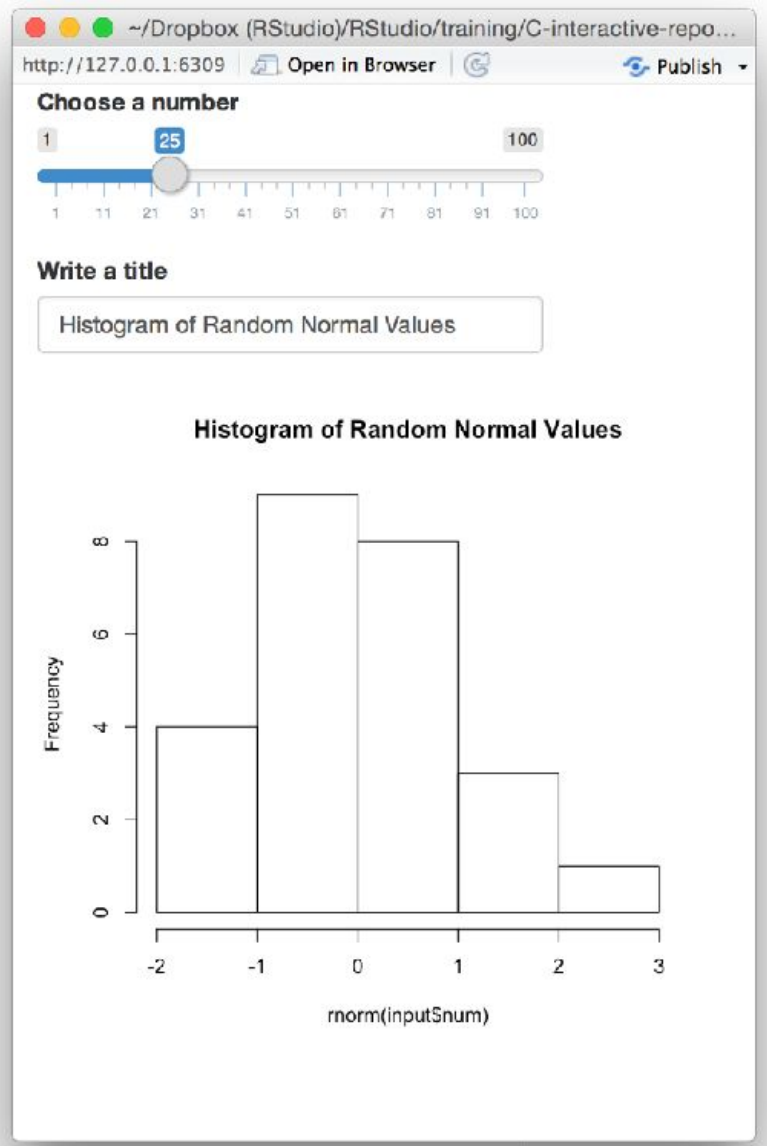
```
# 01-two-inputs

library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  textInput(inputId = "title",
    label = "Write a title",
    value = "Histogram of Random Normal Values"),
  plotOutput("hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num), main = input$title)
  })
}

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



Two outputs

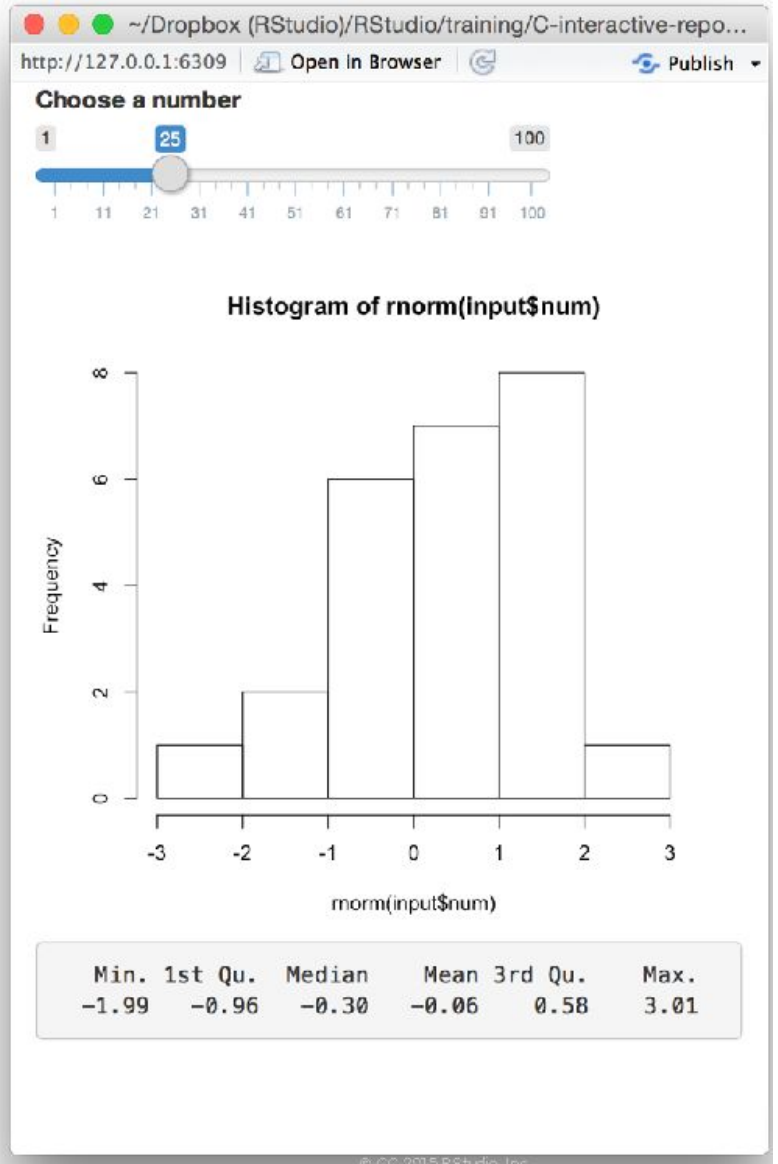
```
# 02-two-outputs

library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist"),
  verbatimTextOutput("stats")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
  output$stats <- renderPrint({
    summary(rnorm(input$num))
  })
}

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



Reactivity

reactive()

Builds a reactive object (reactive expression)

```
data <- reactive({ rnorm(input$num) })
```

object will respond to *every reactive value in the code*

code used to build (and rebuild) object

Two outputs with reactive()

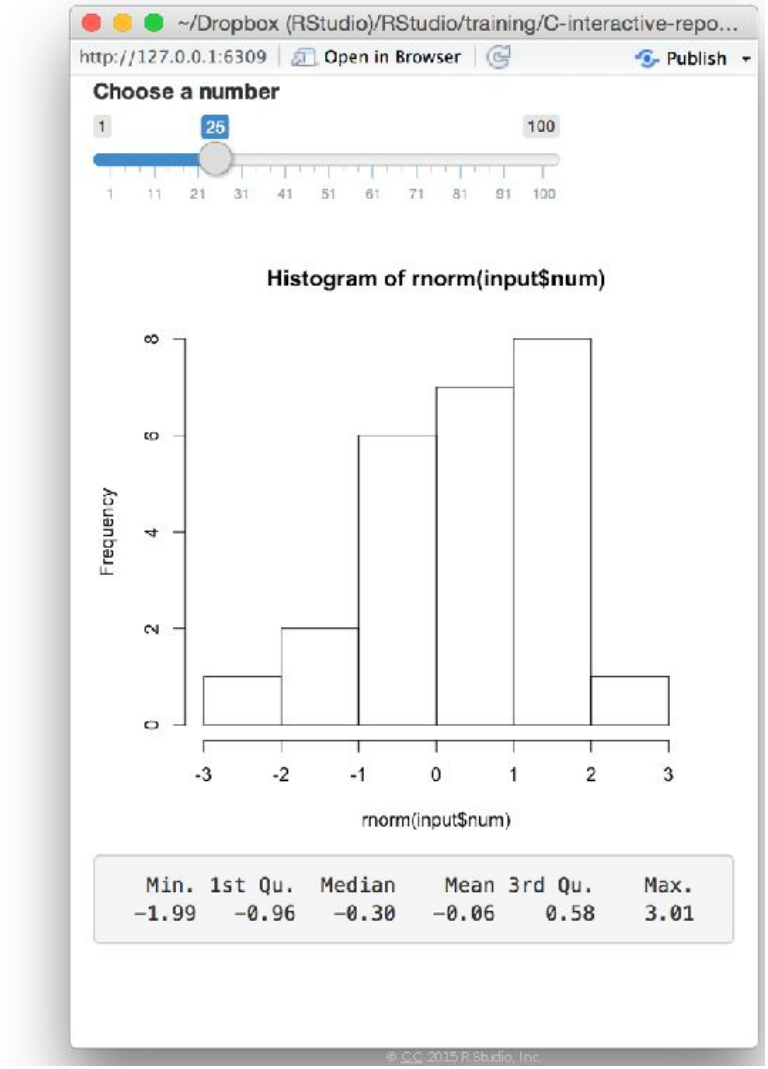
```
# 03-reactive

library(shiny)

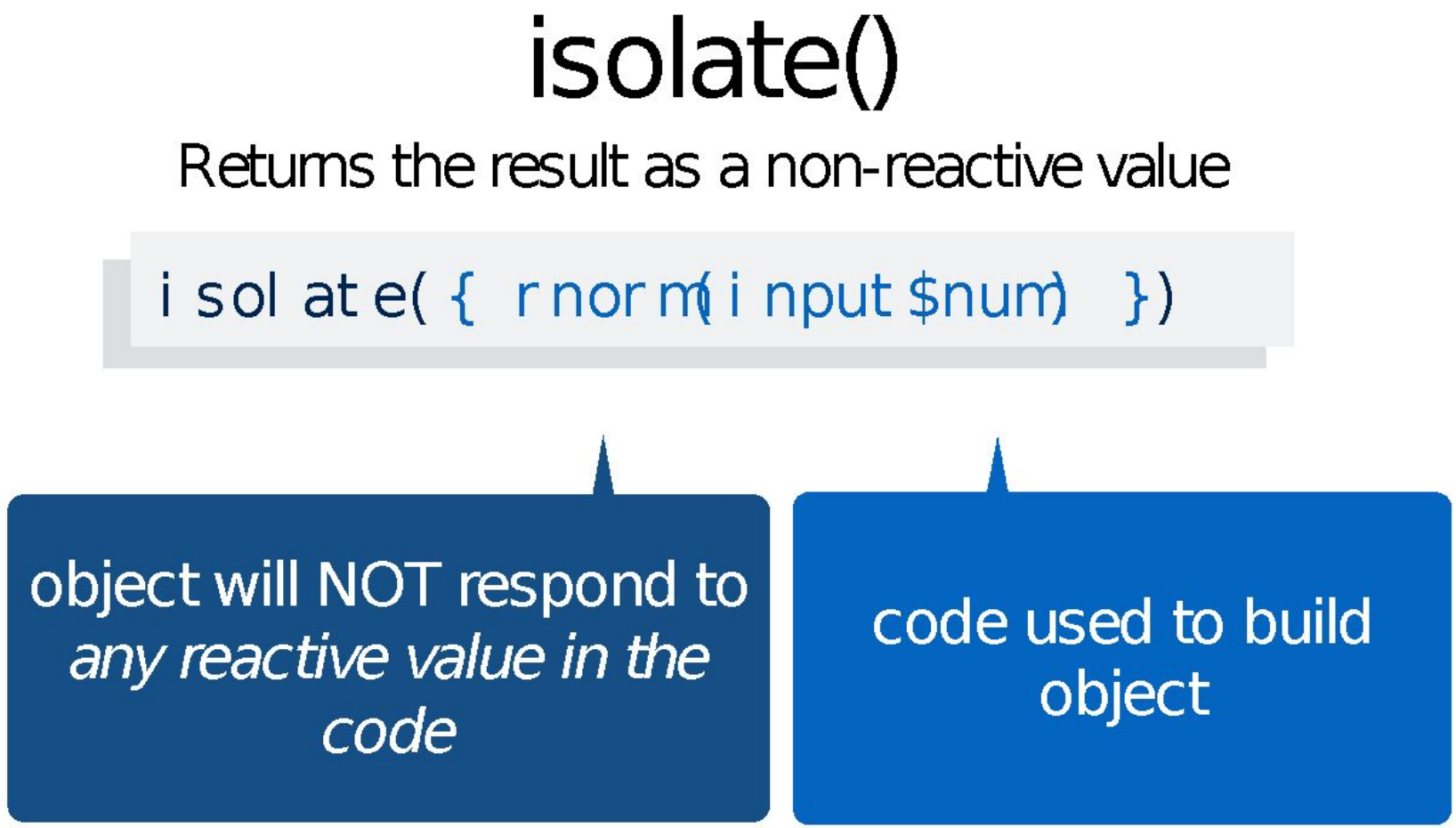
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist"),
  verbatimTextOutput("stats")
)

server <- function(input, output) {
  data <- reactive({
    rnorm(input$num)
  })
  output$hist <- renderPlot({
    hist(data())
  })
  output$stats <- renderPrint({
    summary(data())
  })
}

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



Can we prevent the title field from updating the plot?



Can we prevent the title field from updating the plot?

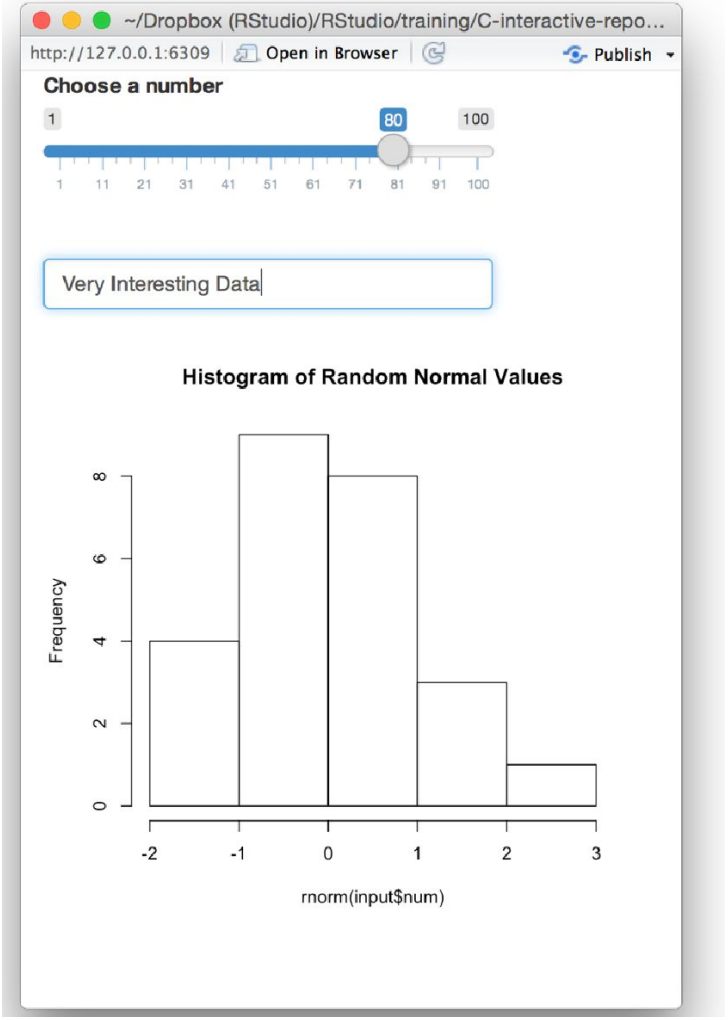
```
# 04-isolate

library(shiny)

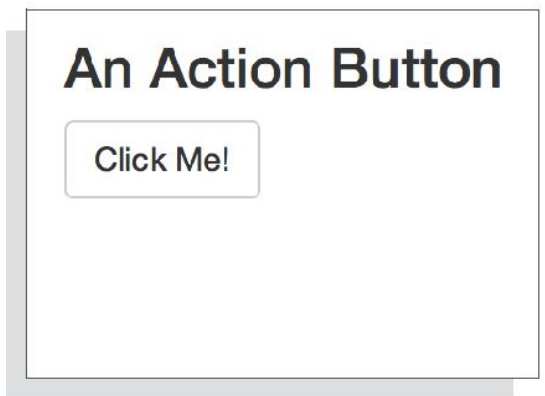
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  textInput(inputId = "title",
    label = "Write a title",
    value = "Histogram of Random Normal Values"),
  plotOutput("hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num),
      main = isolate({input$title}))
  })
}

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



Action buttons



input name
(for internal use)

label to
display

```
actionButton(inputId = "go", label = "Click Me!")
```

Notice:
Id not ID

observeEvent()

Triggers code to run on server

```
observeEvent(input$clicks, { print(input$clicks) })
```

reactive value(s) to respond to

(observer invalidates ONLY when this value changes)

code block to run whenever observer is invalidated

note: observer treats this code as if it has been isolated with isolate()

Action buttons

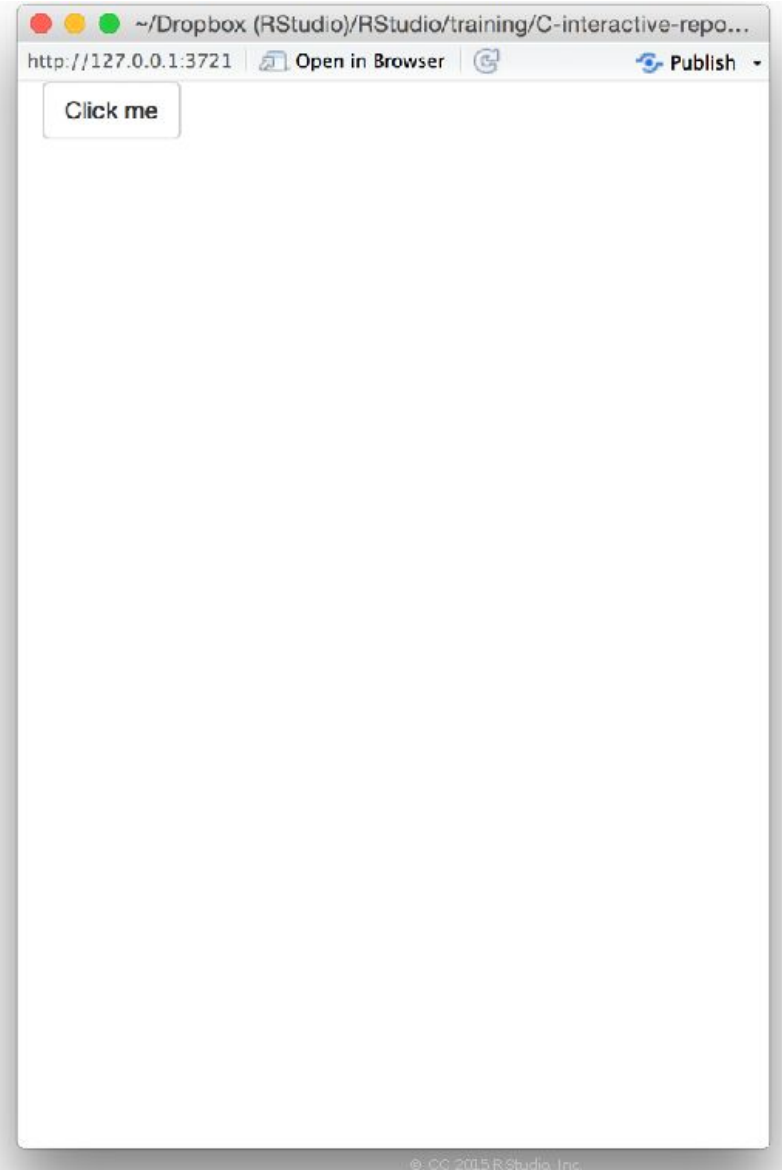
```
# 05-actionButton

library(shiny)

ui <- fluidPage(
  actionButton(inputId = "clicks",
    label = "Click me")
)

server <- function(input, output) {
  observeEvent(input$clicks, {
    print(as.numeric(input$clicks))
  })
}

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



reactiveValues()

Creates a list of reactive values to manipulate programmatically

```
rv <- reactiveValues(data = rnorm(100))
```

(optional) elements to add to the list

```r
# 08-reactiveValues

library(shiny)

ui <- fluidPage(
  actionButton(inputId = "norm", label = "Normal"),
  actionButton(inputId = "unif", label = "Uniform"),
  plotOutput("hist")
)

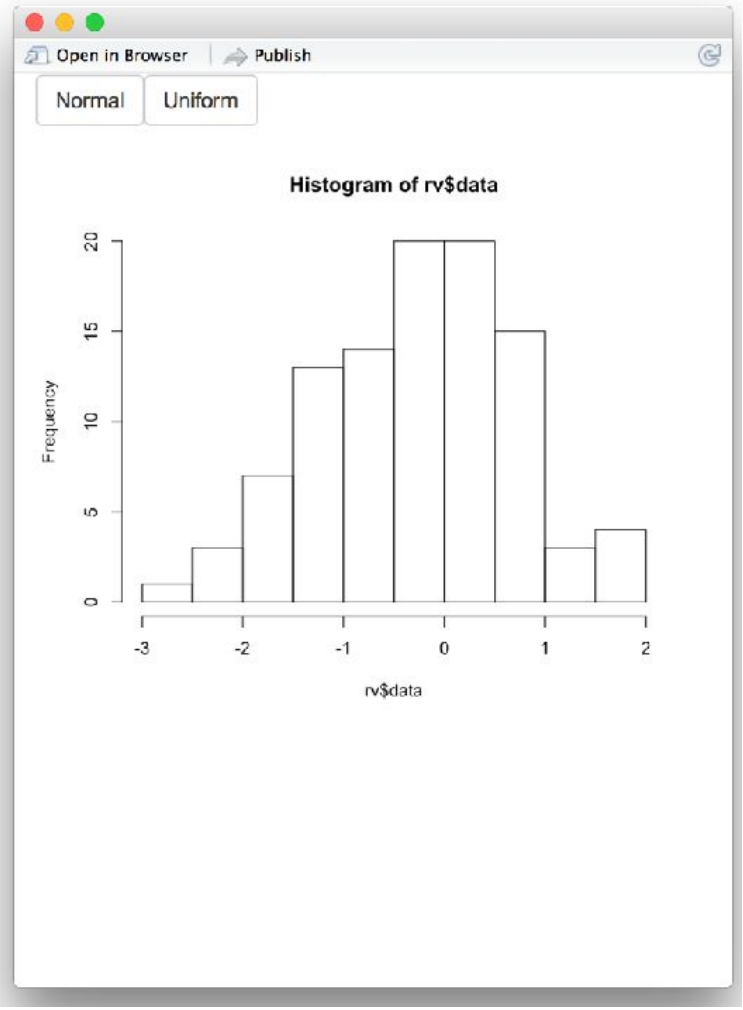
server <- function(input, output) {

  rv <- reactiveValues(data = rnorm(100))

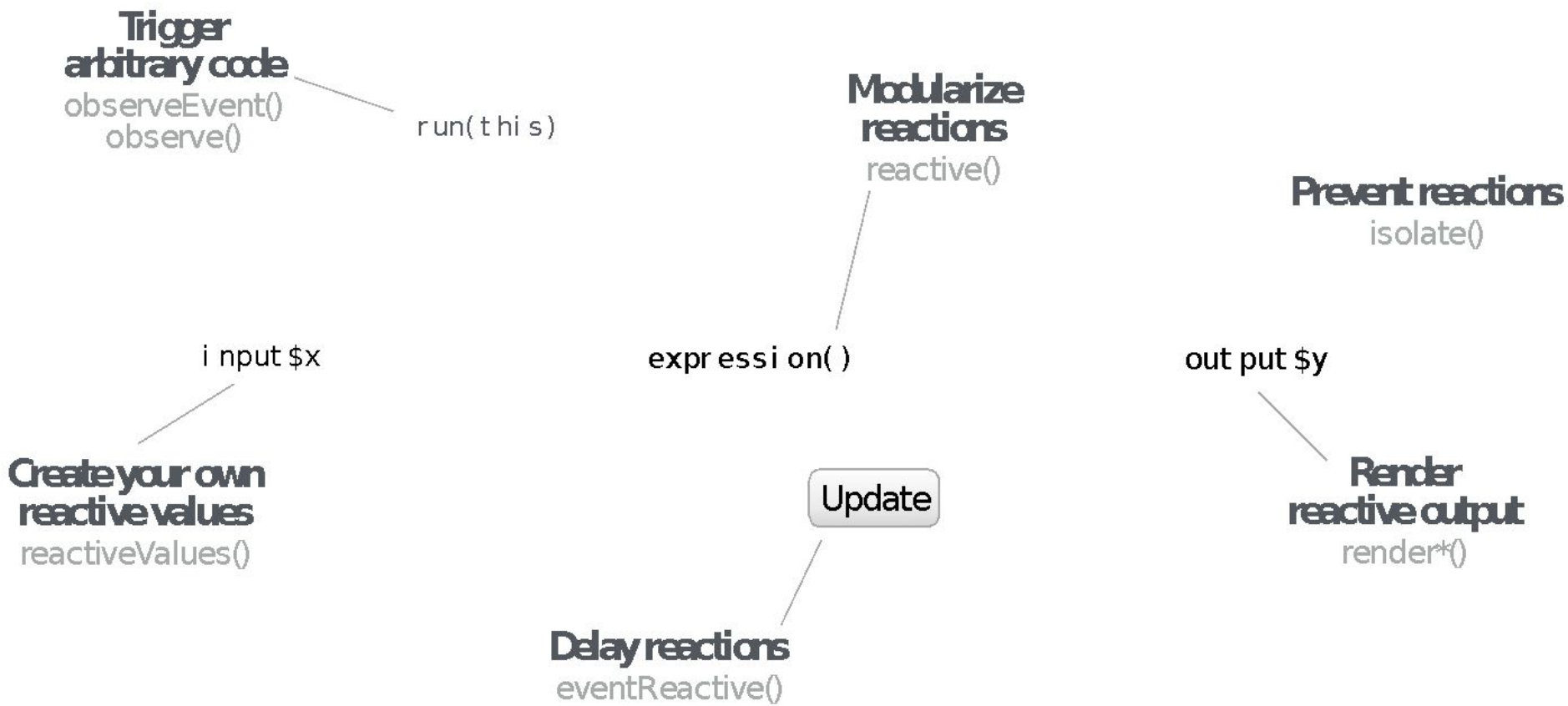
  observeEvent(input$norm, { rv$data <- rnorm(100) })
  observeEvent(input$unif, { rv$data <- runif(100) })

  output$hist <- renderPlot({
    hist(rv$data)
  })
}

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



Summary


Trigger arbitrary code
observeEvent()
observe()
run(this)
Modularize reactions
reactive()
Prevent reactions
isolate()
input $x
expression()
output $y
Create your own reactive values
reactiveValues()
Update
Render reactive output
render*()
Delay reactions
eventReactive()