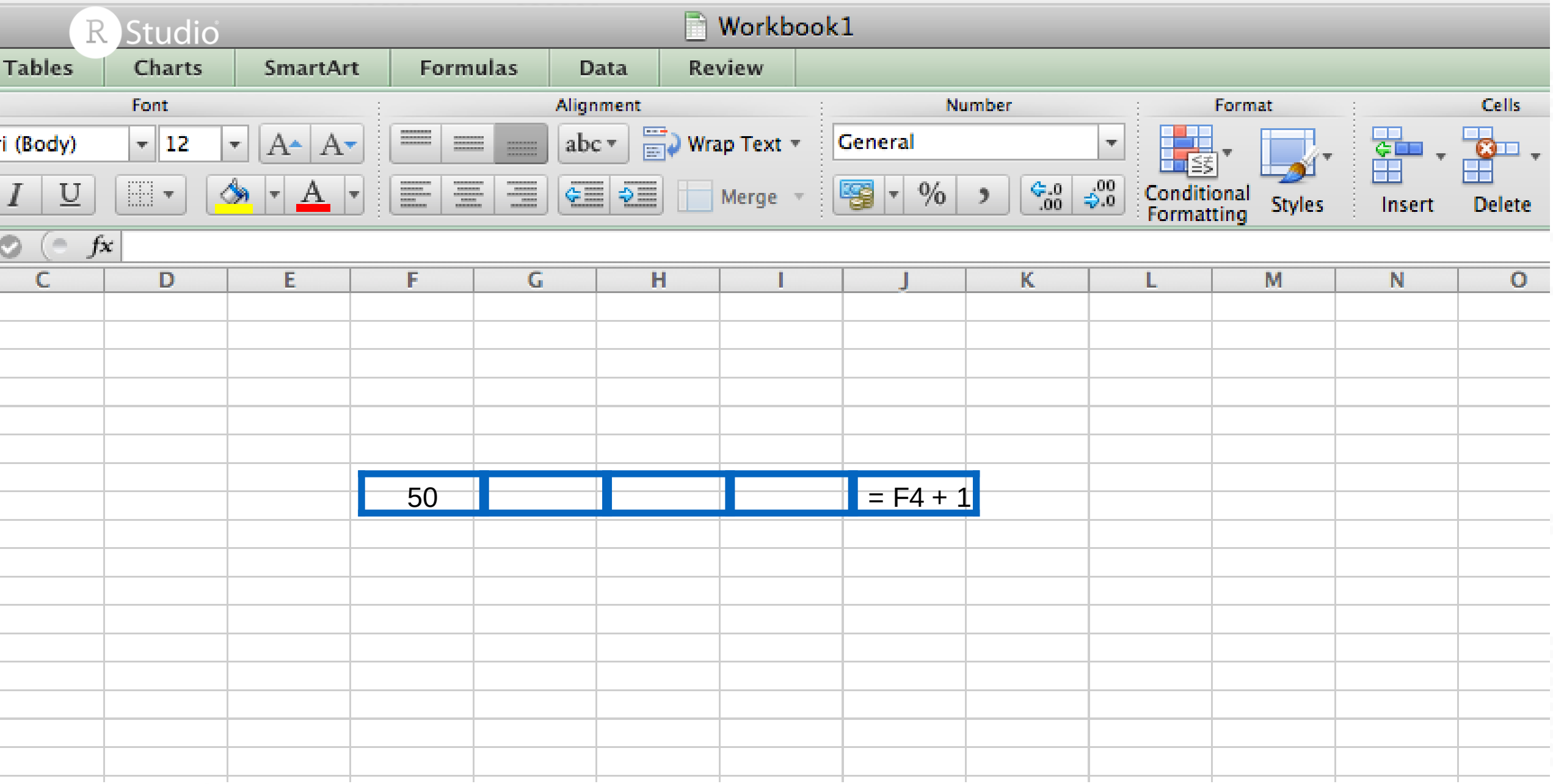
NobleProg

Reactivity

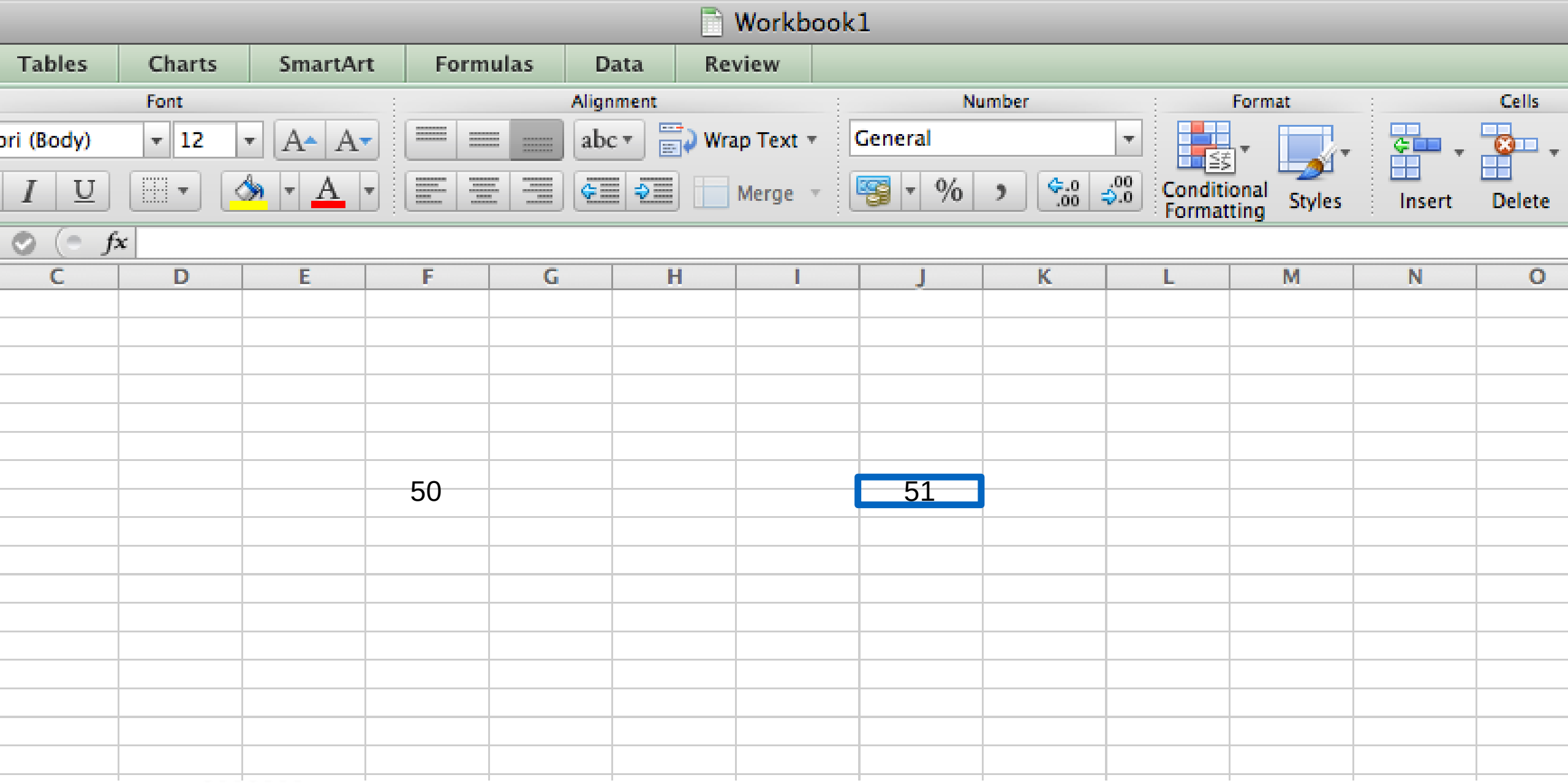
The World's Local Training Provider

What is Reactivity?

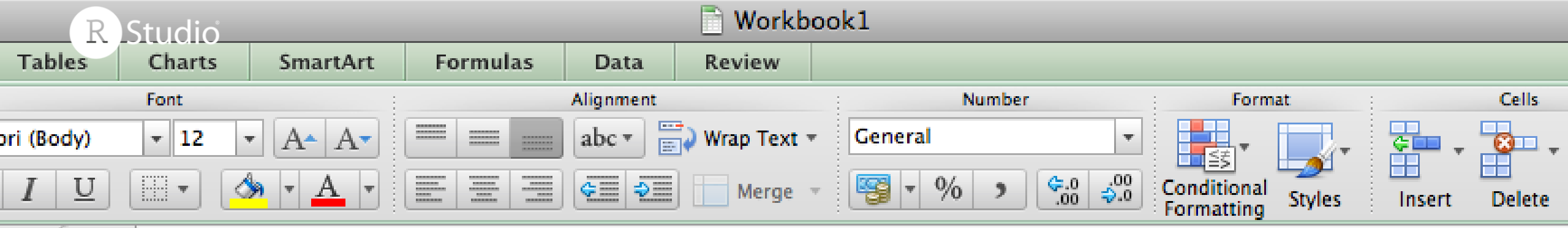
NobleProg



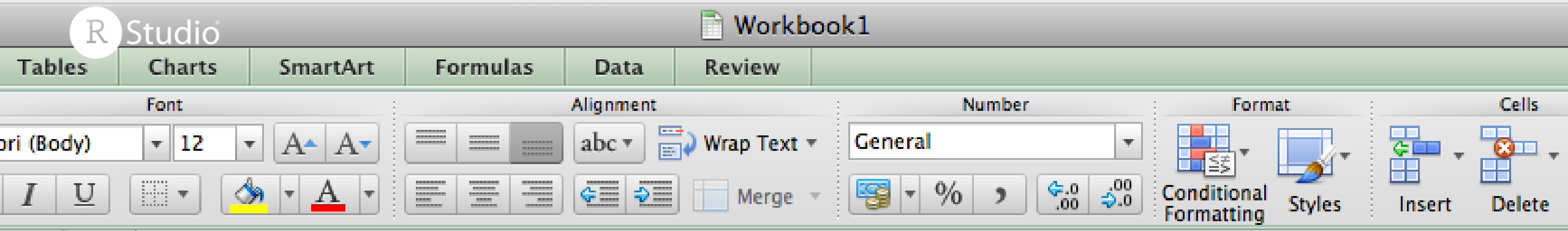
NobleProg



NobleProg

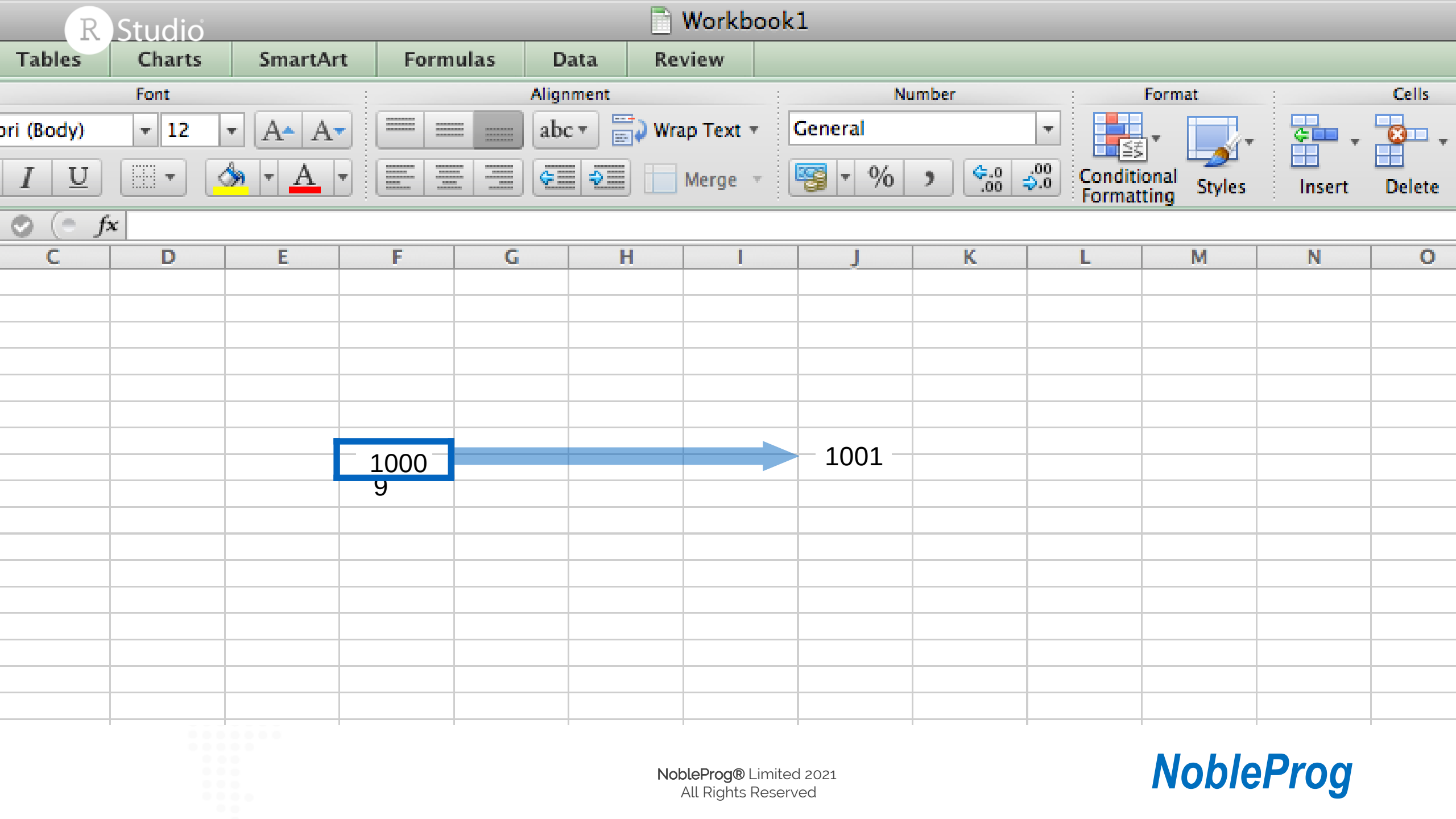


NobleProg

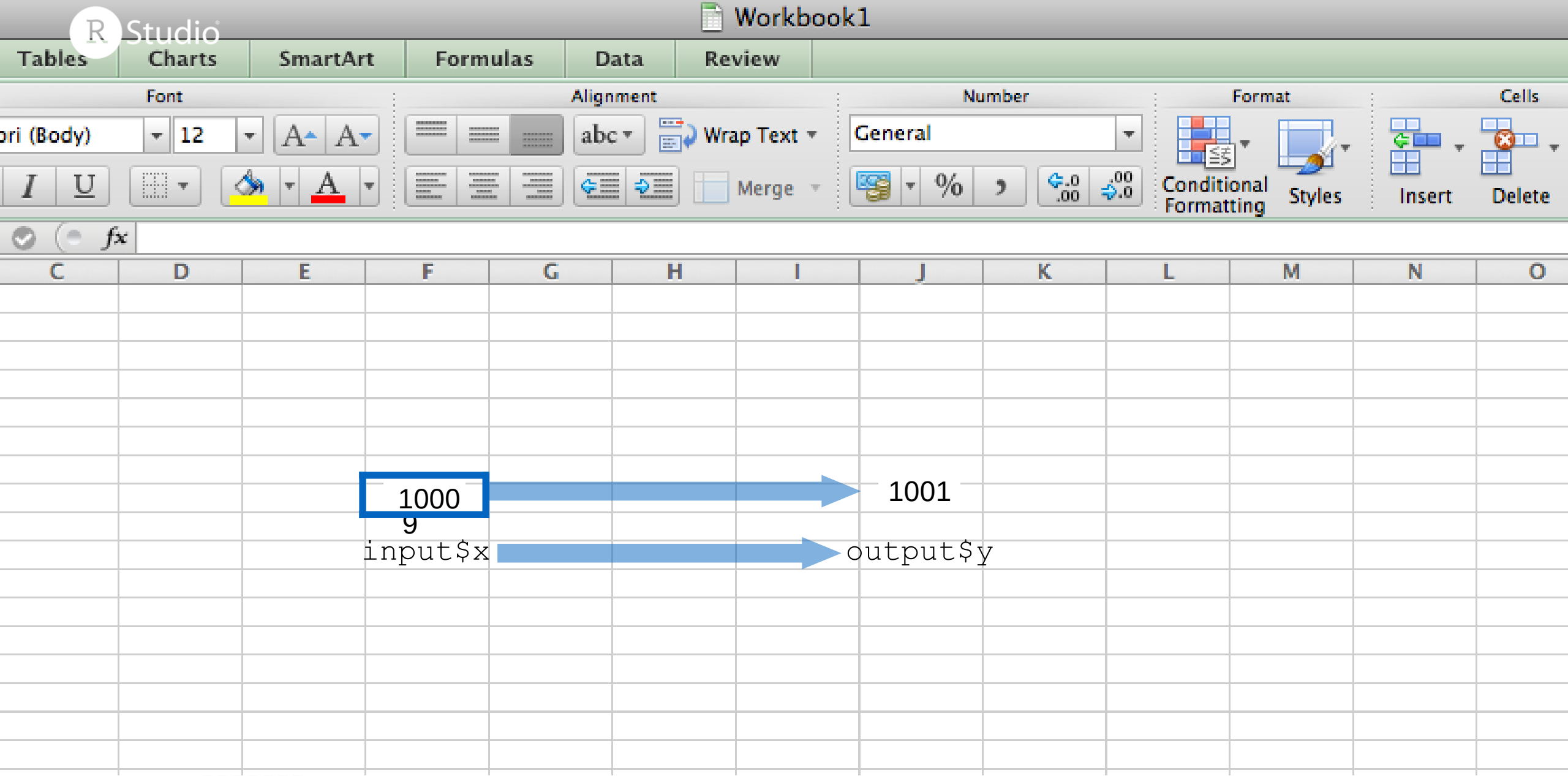


C	D	E	F	G	H	I	J	K	L	M	N	O
			999				1000					
			0				1					

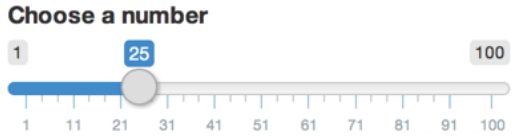
NobleProg

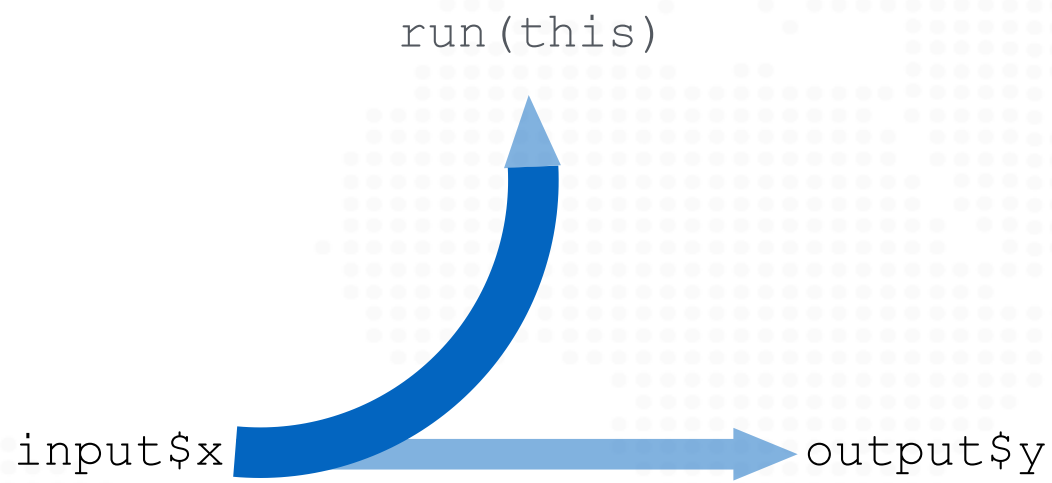


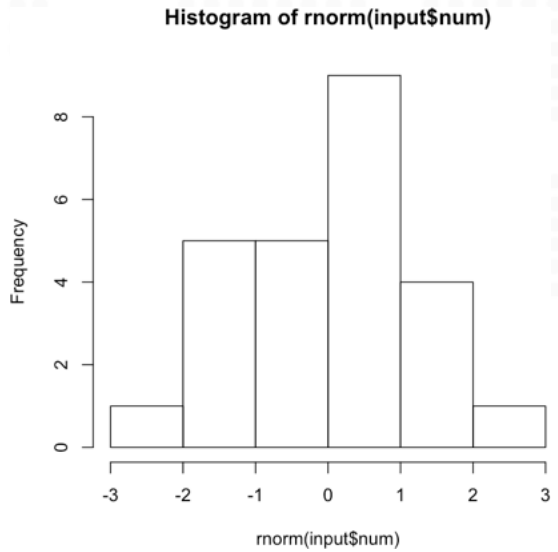
NobleProg



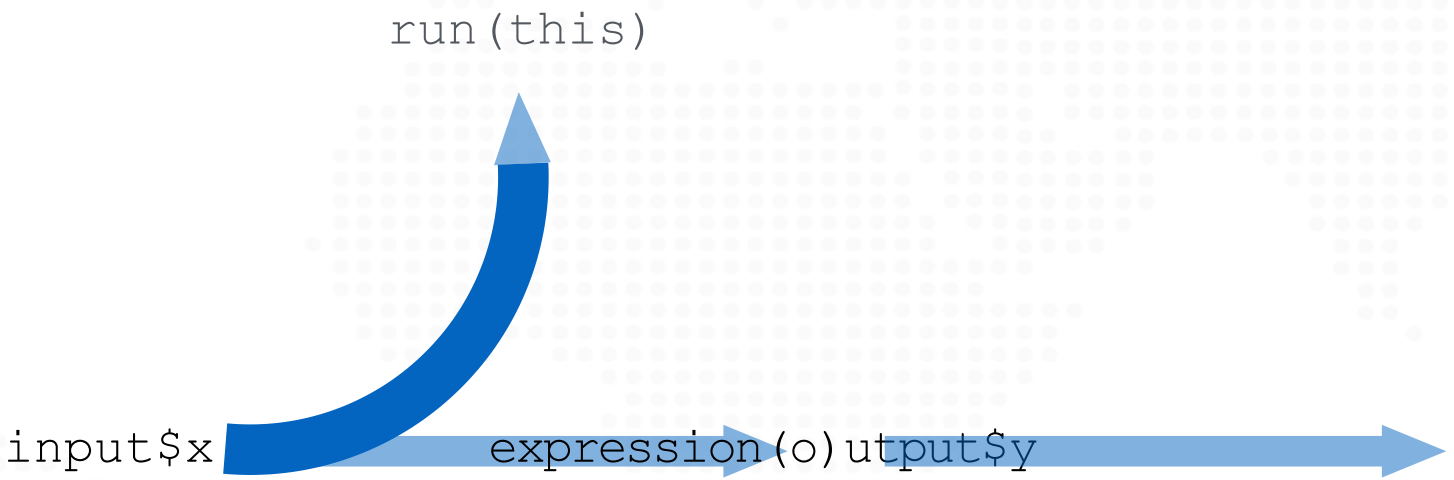
NobleProg



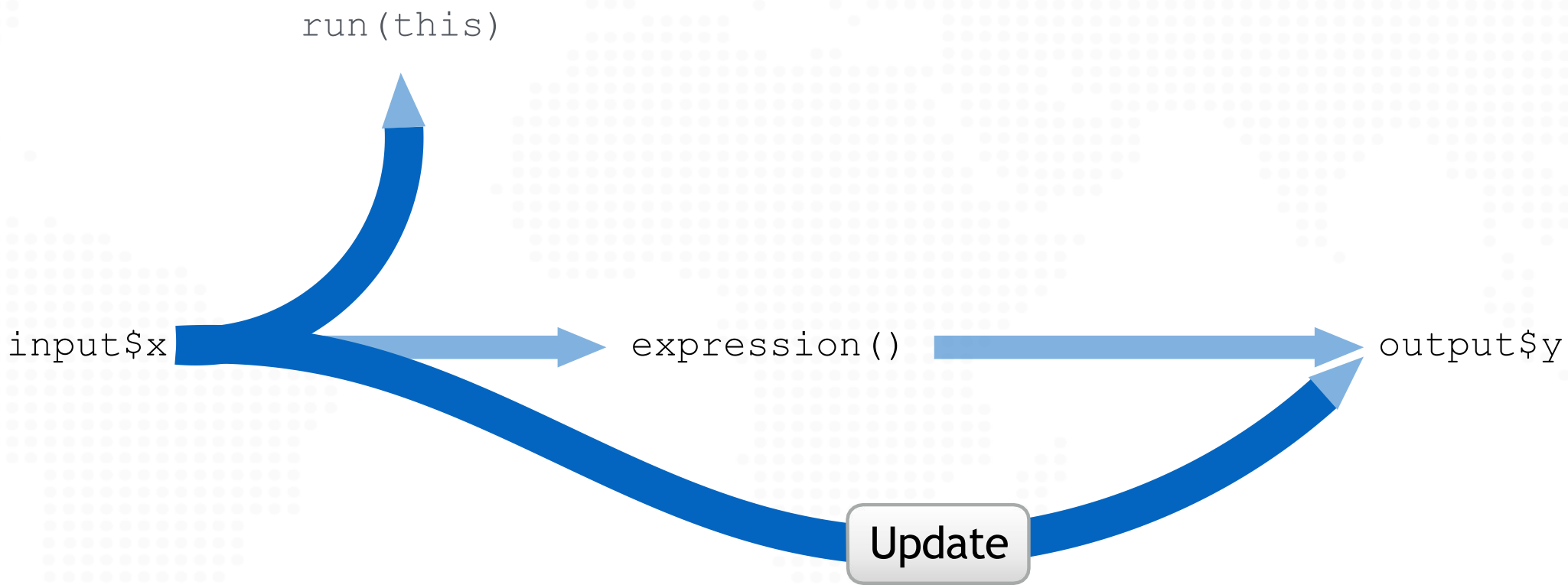




NobleProg

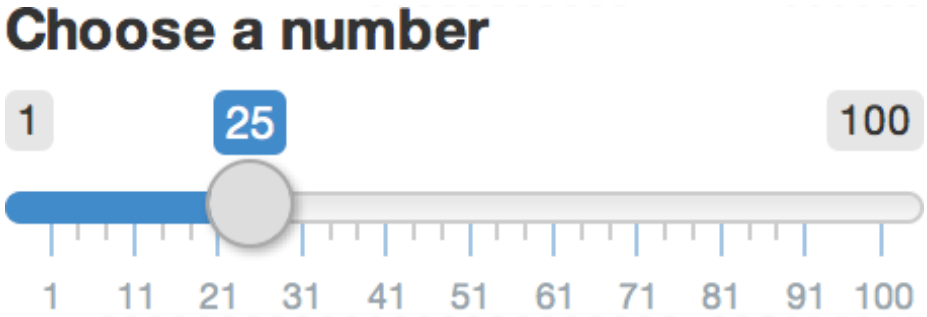


NobleProg



NobleProg

Syntax



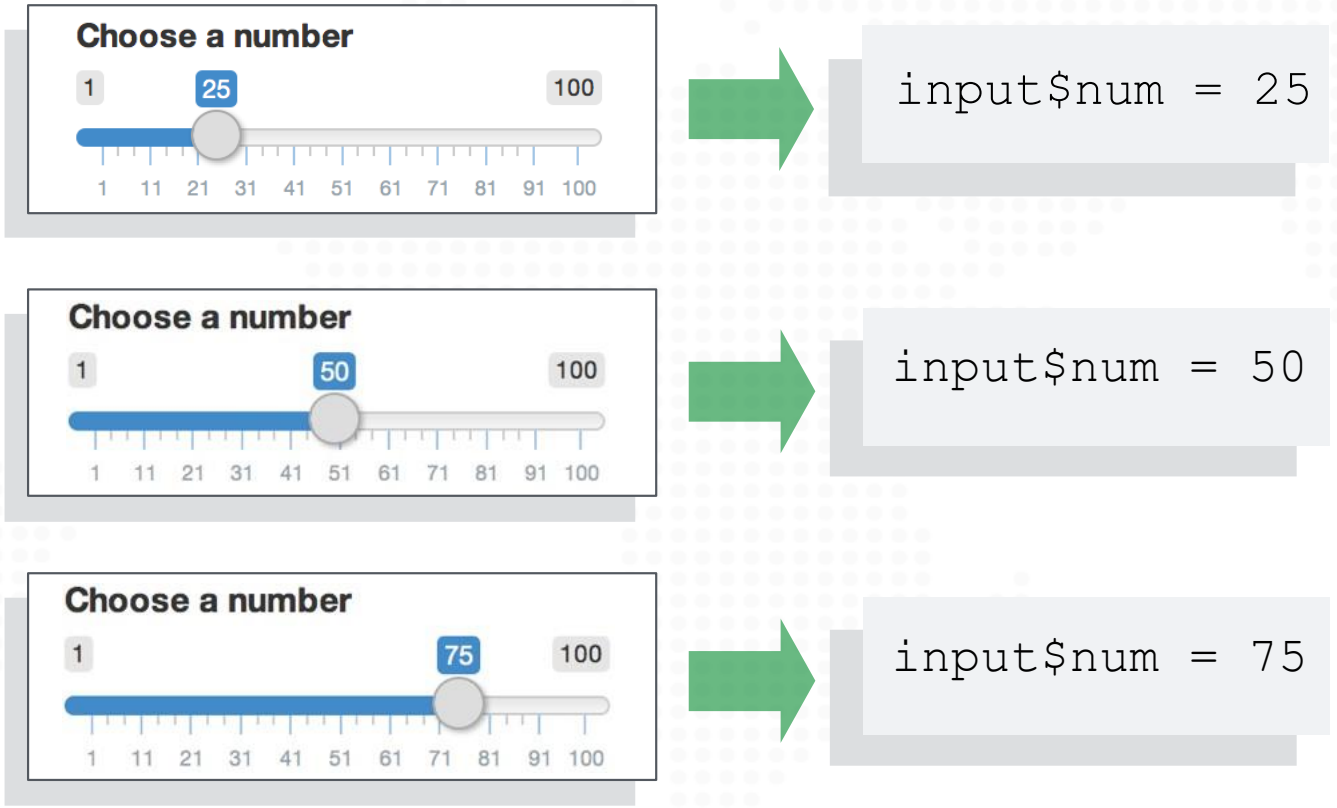

```
sliderInput(inputId = "num", label = "Choose a number", …)
```

this input will provide a value saved as **input$num**

NobleProg

Input values

The input value changes whenever a user changes the input.



NobleProg

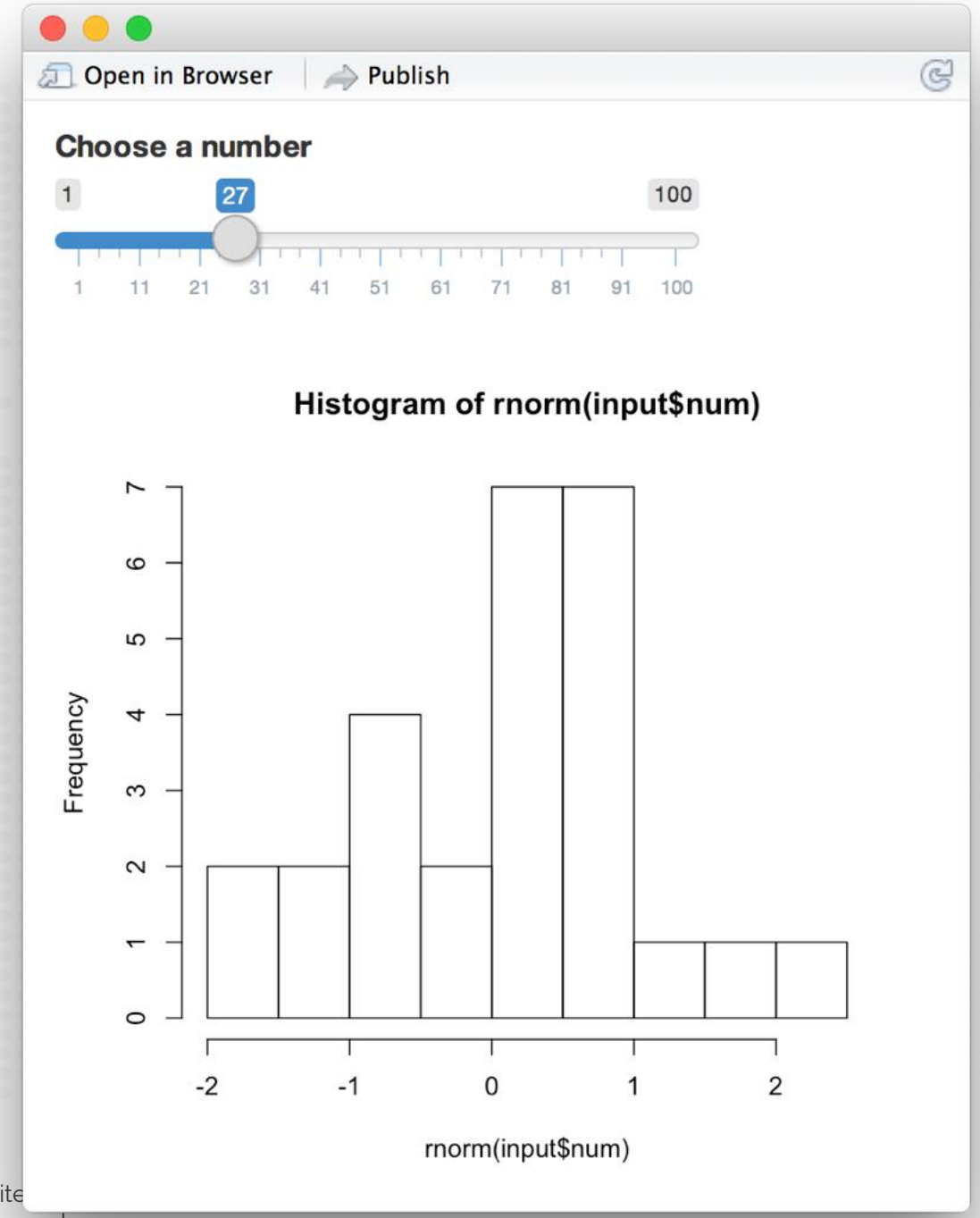
Reactive values work together with reactive functions.
You cannot call a reactive value from outside of one.

```
renderPlot({ hist(rnorm(100, input$num)) })
```

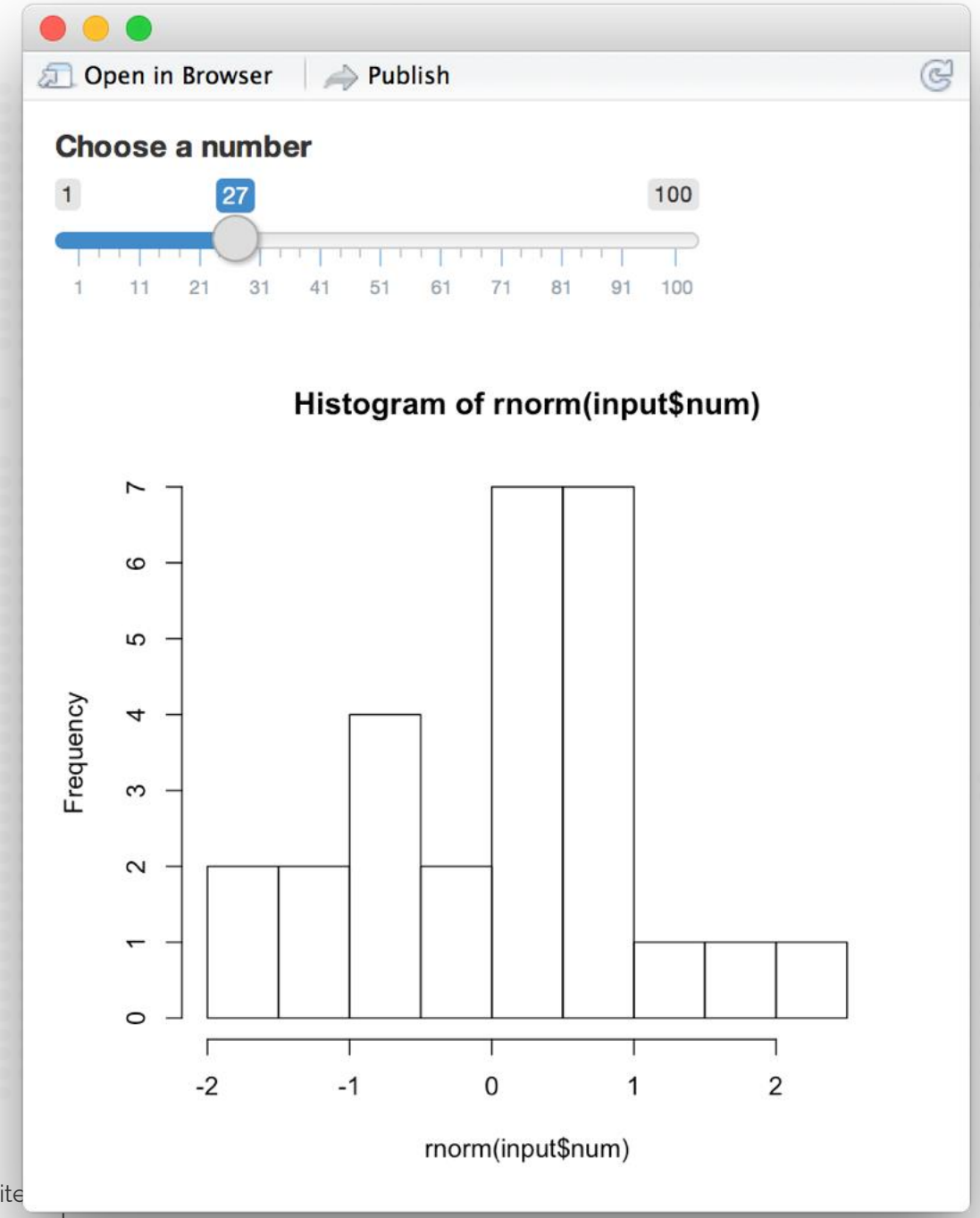
```
hist(rnorm(100, input$num))
```


NobleProg

```
library(shiny)

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

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

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



```
library(shiny)

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

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

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

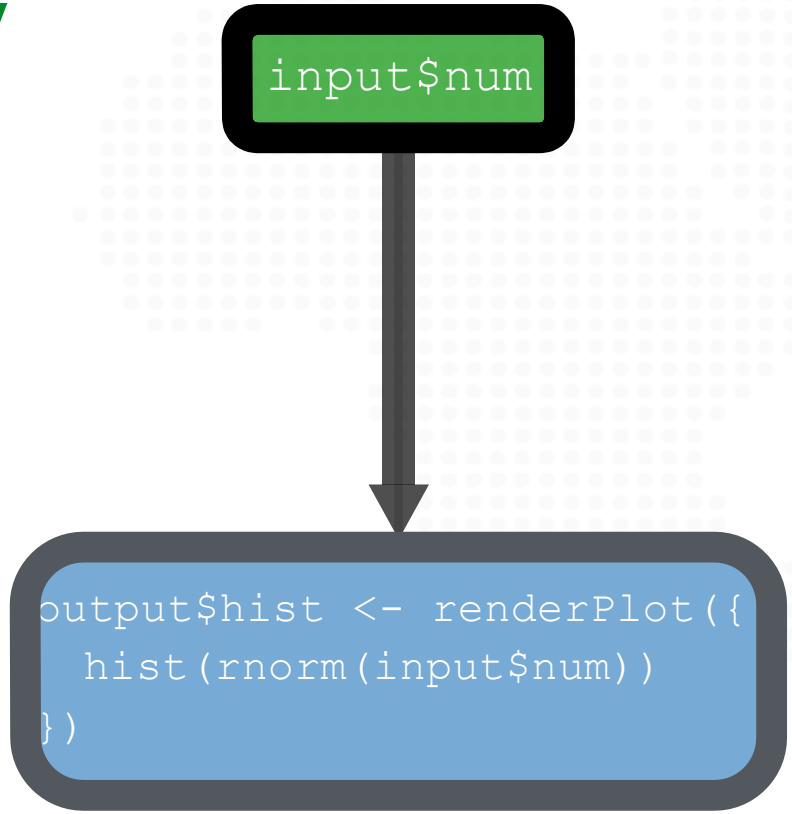


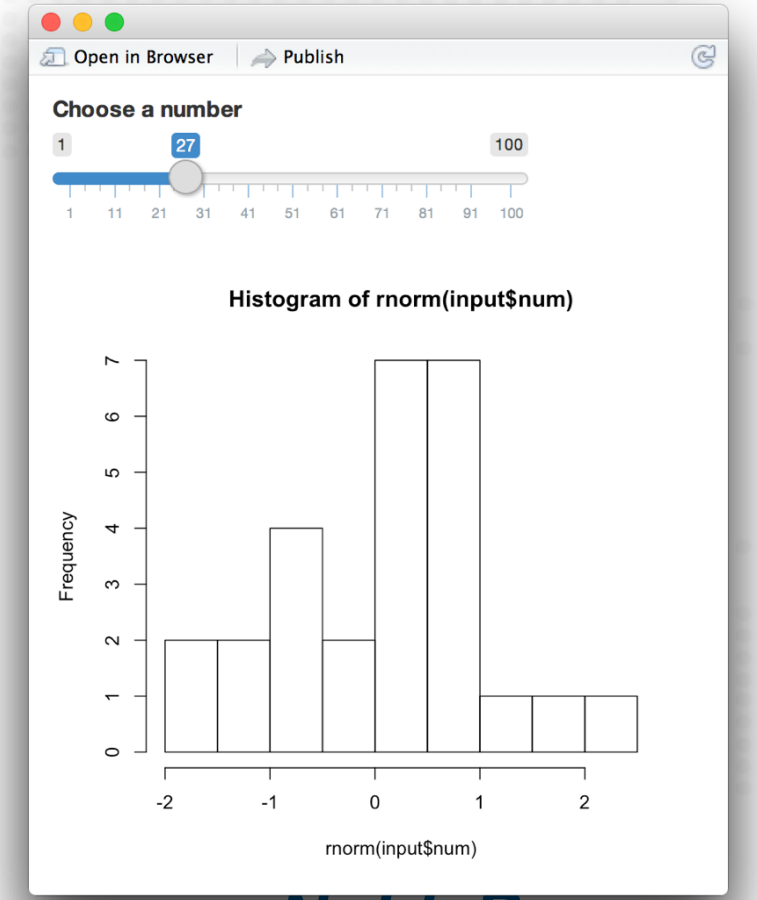
```
library(shiny)

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

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

}

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

```
 Error in .getReactiveEnvironment()
$currentContext() :
  Operation not allowed without an
active reactive context. (You tried
to do something that can only be done
from inside a reactive expression or
observer.)
```

Think of reactivity in R as a two step process

1 **Reactive values notify**
the functions that use them
when they become invalid



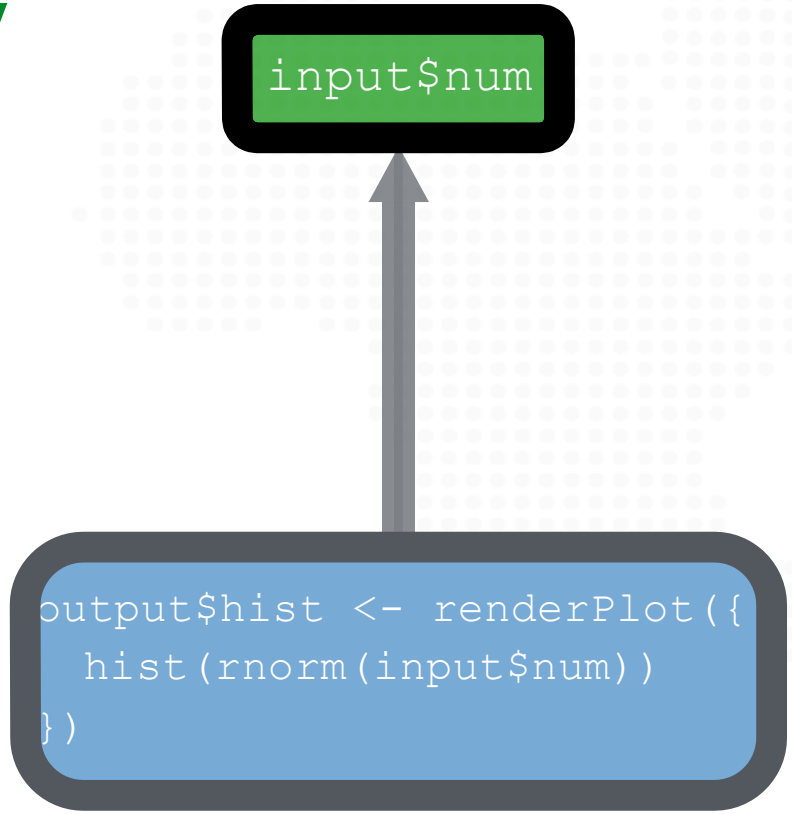


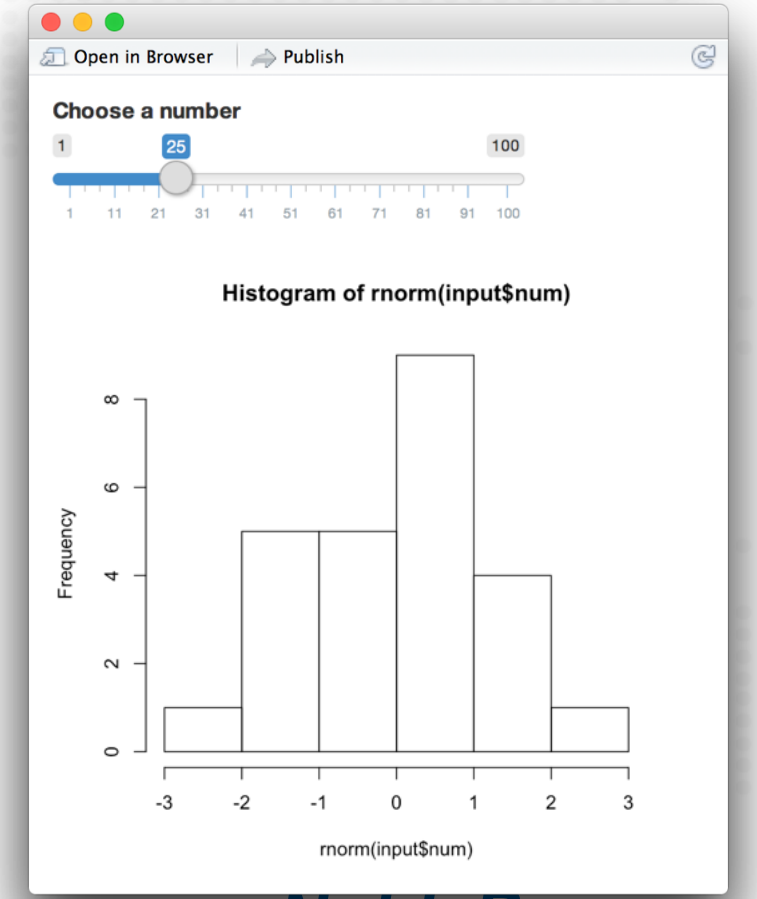
NobleProg

Think of reactivity in R as a two step process

1 **Reactive values notify**
the functions that use them
when they become invalid

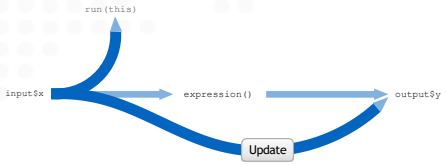
2 The objects created by
reactive functions respond
(different objects respond differently)





NobleProg

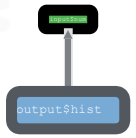
Recap: Reactive values



Reactive values act as the data streams that flow through your app.

The **input** list is a list of reactive values. The values show the current state of the inputs.

You can only call a reactive value from a function that is designed to work with one



Reactive values notify. The objects created by **reactive functions respond.**

NobleProg

Reactive toolkit (7 indispensible functions)

NobleProg

Reactive functions

1. Use a code chunk to build (and rebuild) an object
   - **What code** will the function use?

2. The object will respond to changes in a set of reactive values
   - **Which reactive values** will the object respond to?

Display output with render*()

NobleProg

Render functions build output to display in the app

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

NobleProg

render*()

Builds reactive output to display in UI

```
renderPlot(  { hist(rnorm(input$num)) })
```

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

code used to build (and rebuild) object

NobleProg

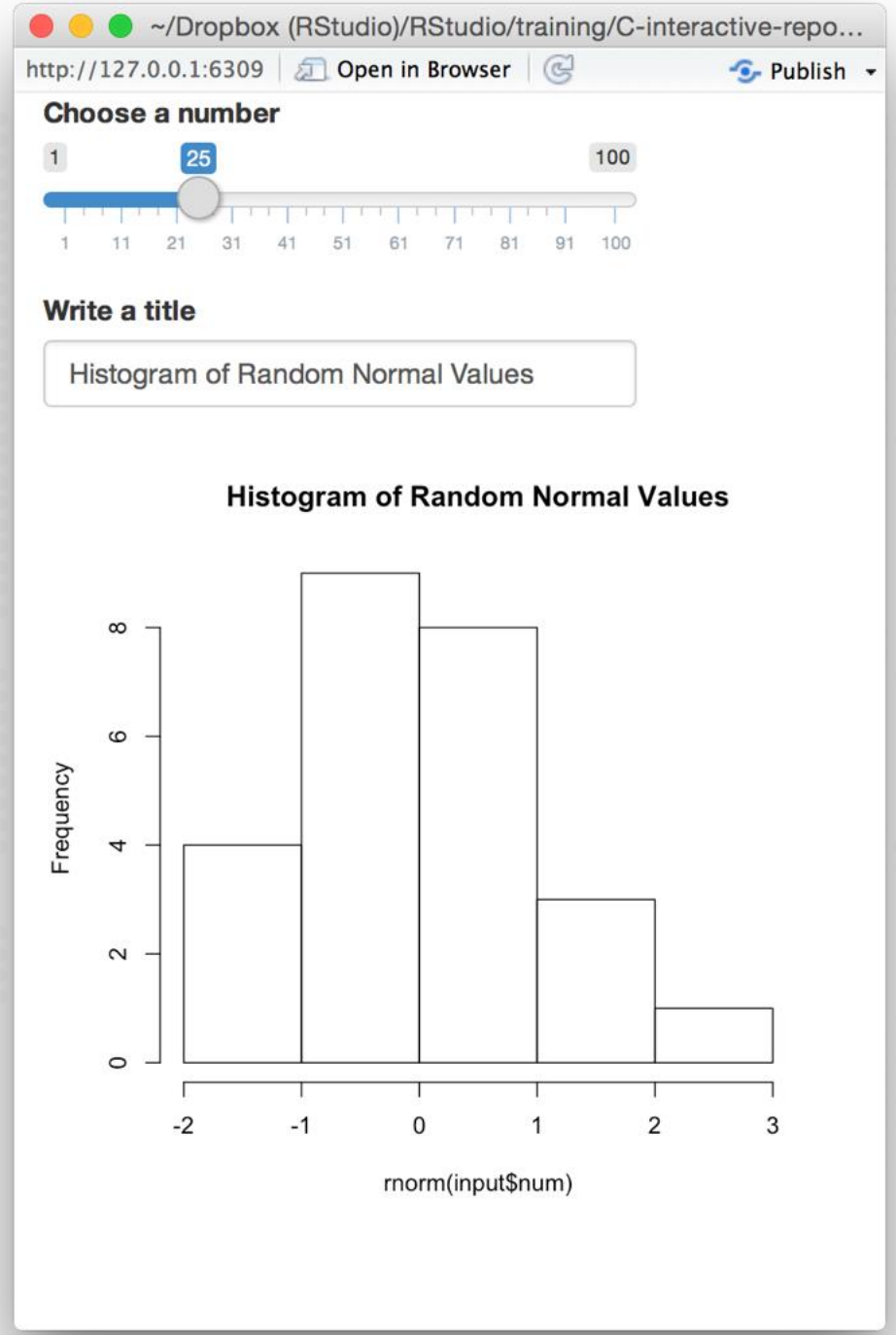
render*()

Builds reactive output to display in UI

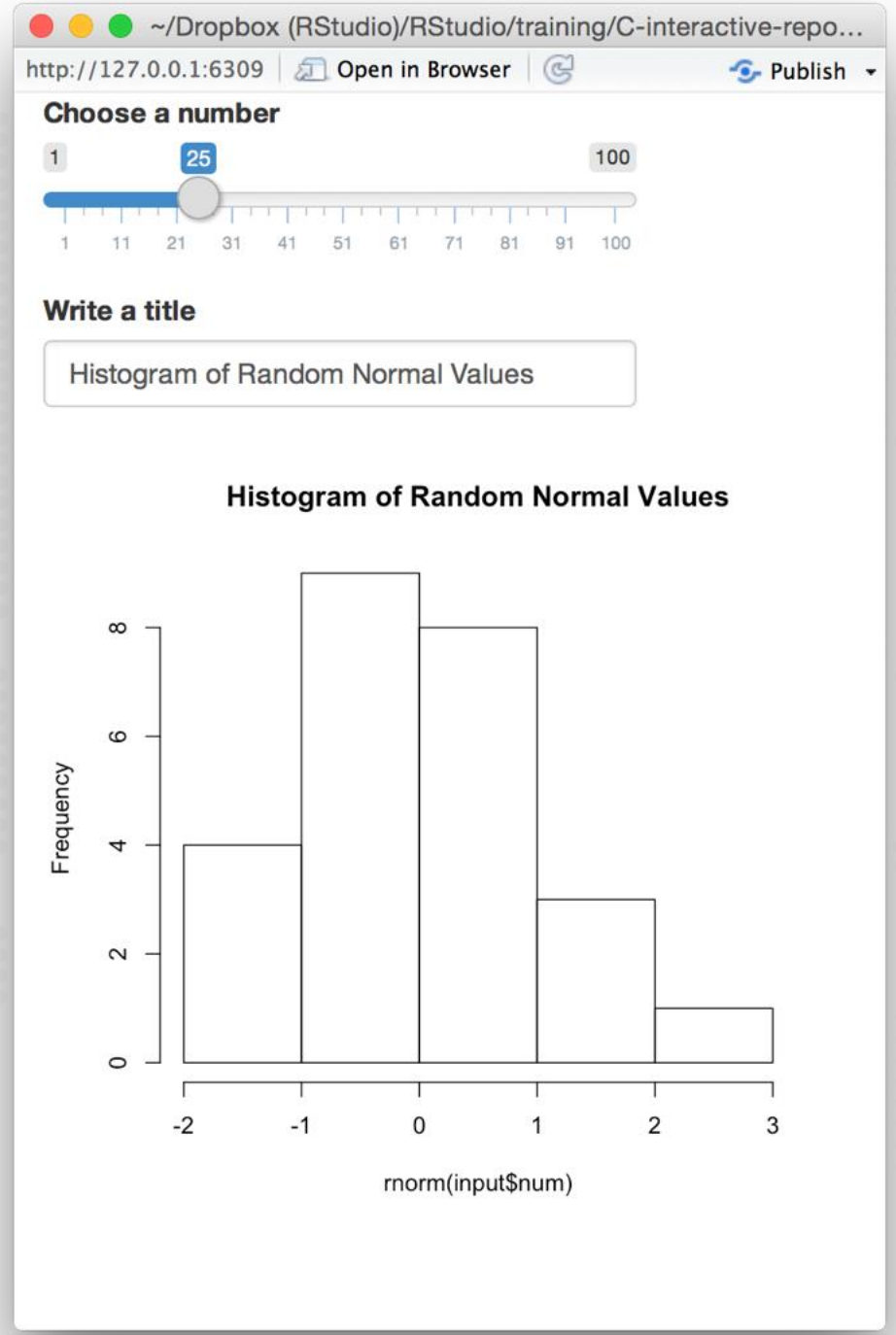
```
renderPlot(  { hist(rnorm(input$num)) })
```

When notified that it is invalid, the object created by a render*() function will rerun the entire block of code associated with it

```
# 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)
```



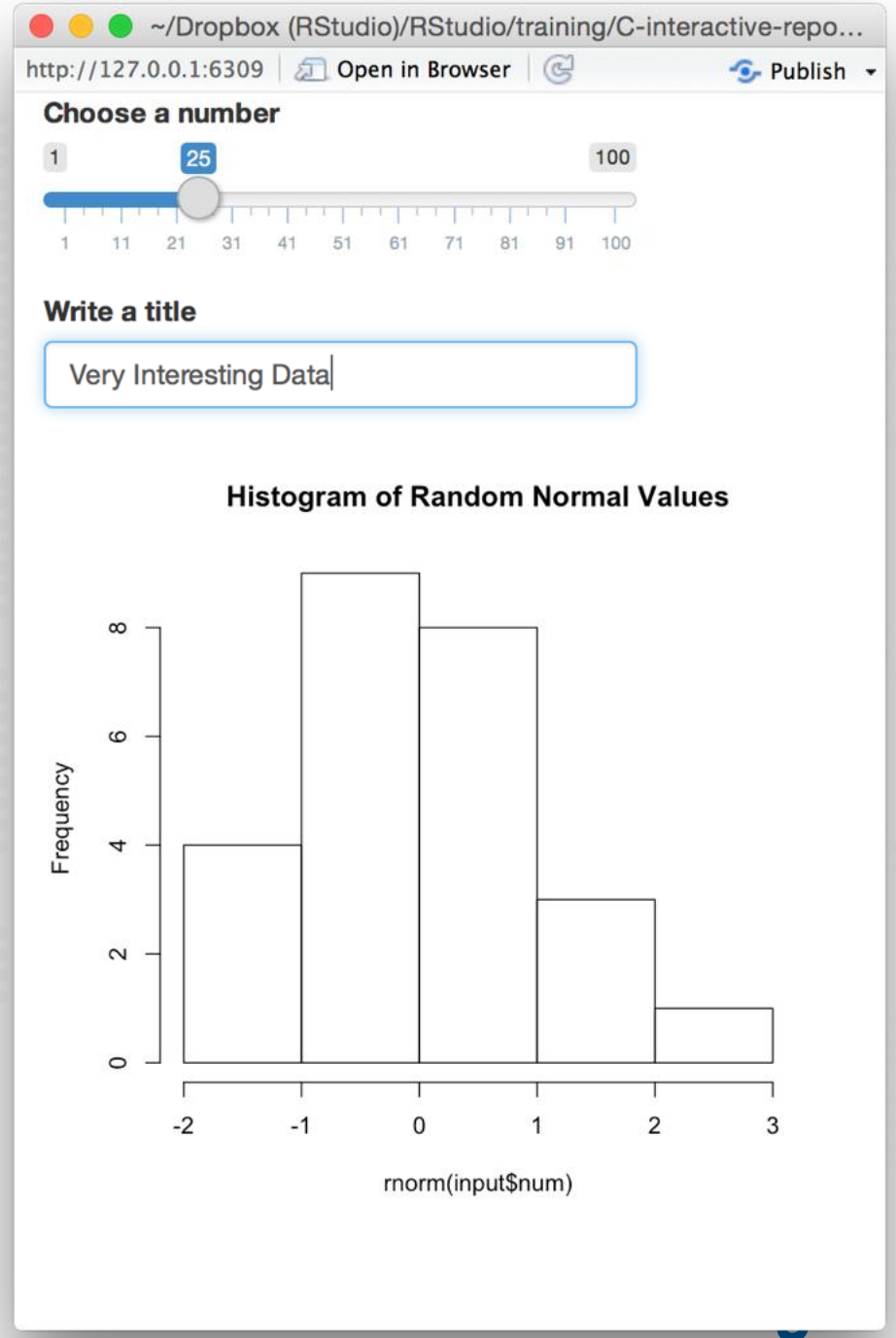
```
# 01-two-inputs

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)
```



input$num

input$title

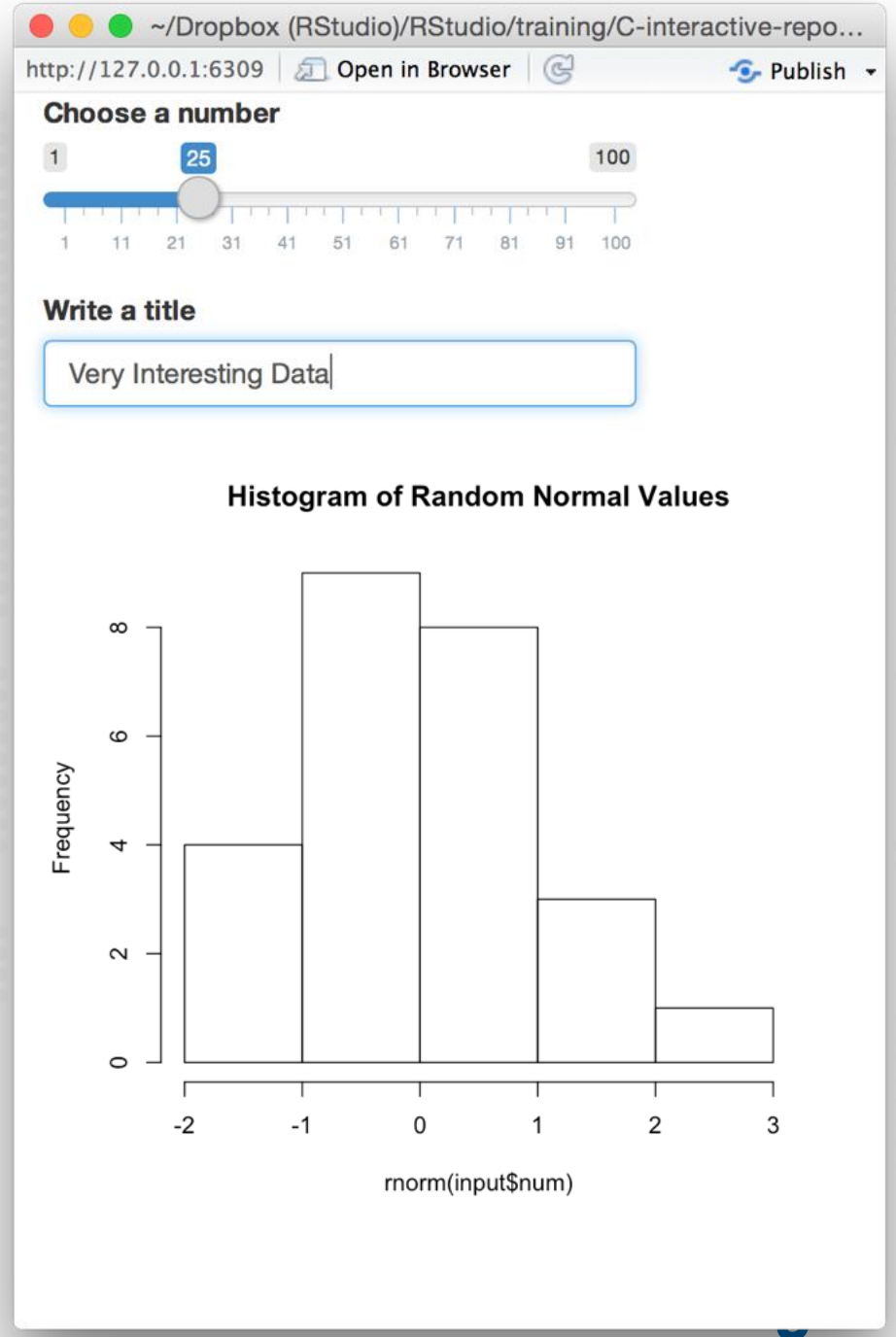
```
output$hist <-
 renderPlot({
 hist(rnorm(input$num),
   main = input$title)
```



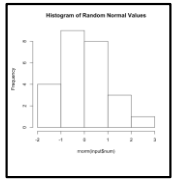
input$num

input$title

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



Recap: render*()



render*() functions make **objects to display**

`output$`

Always save the result to **output$**

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

render*() makes an observer object that has a **block of code** associated with it

```
renderPlot(  { hist(rnorm(input$num)) })
```

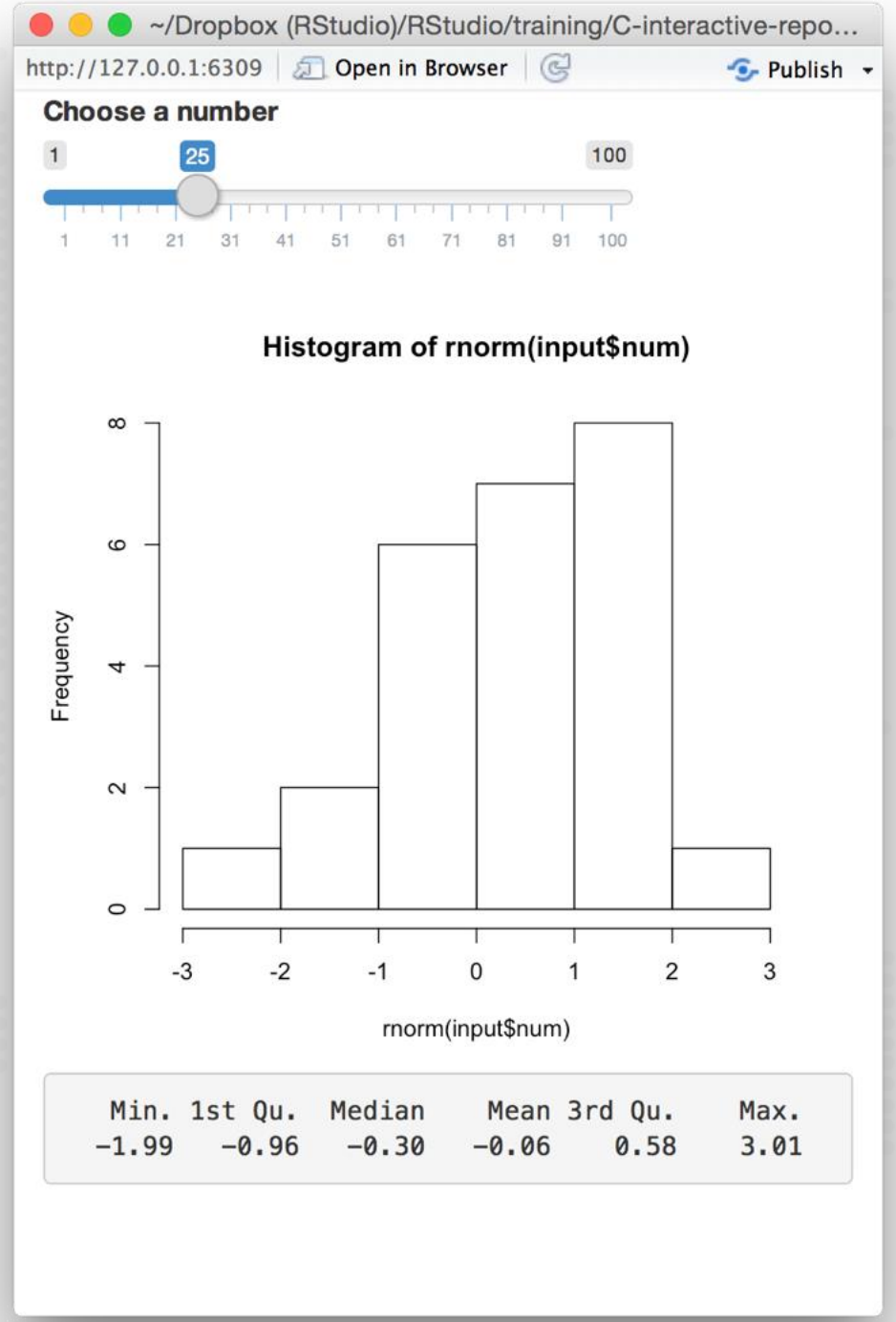
The object will **rerun the entire code block** to update itself whenever it is invalidated

NobleProg

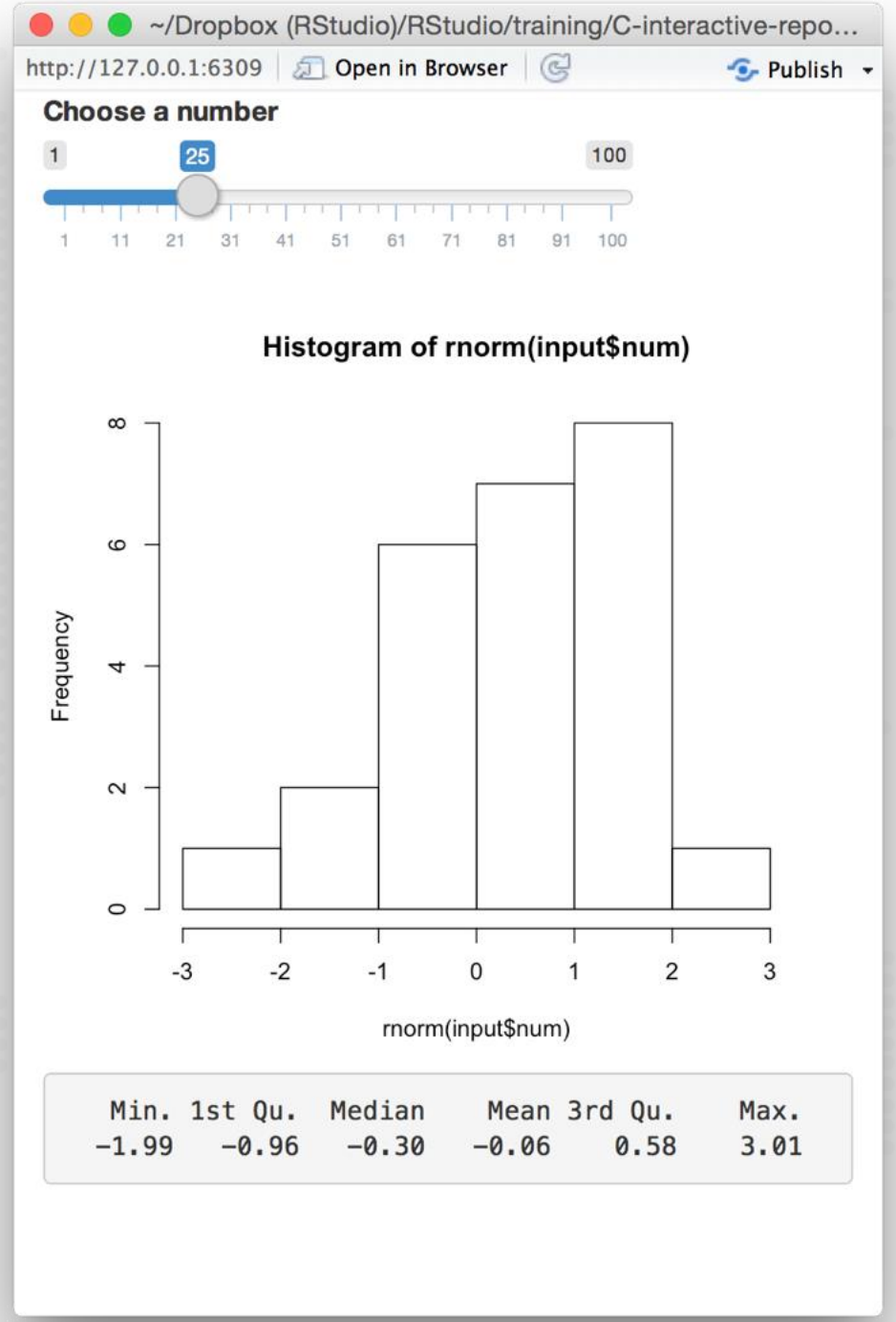
Modularize code with reactive()

NobleProg

```
# 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)
```



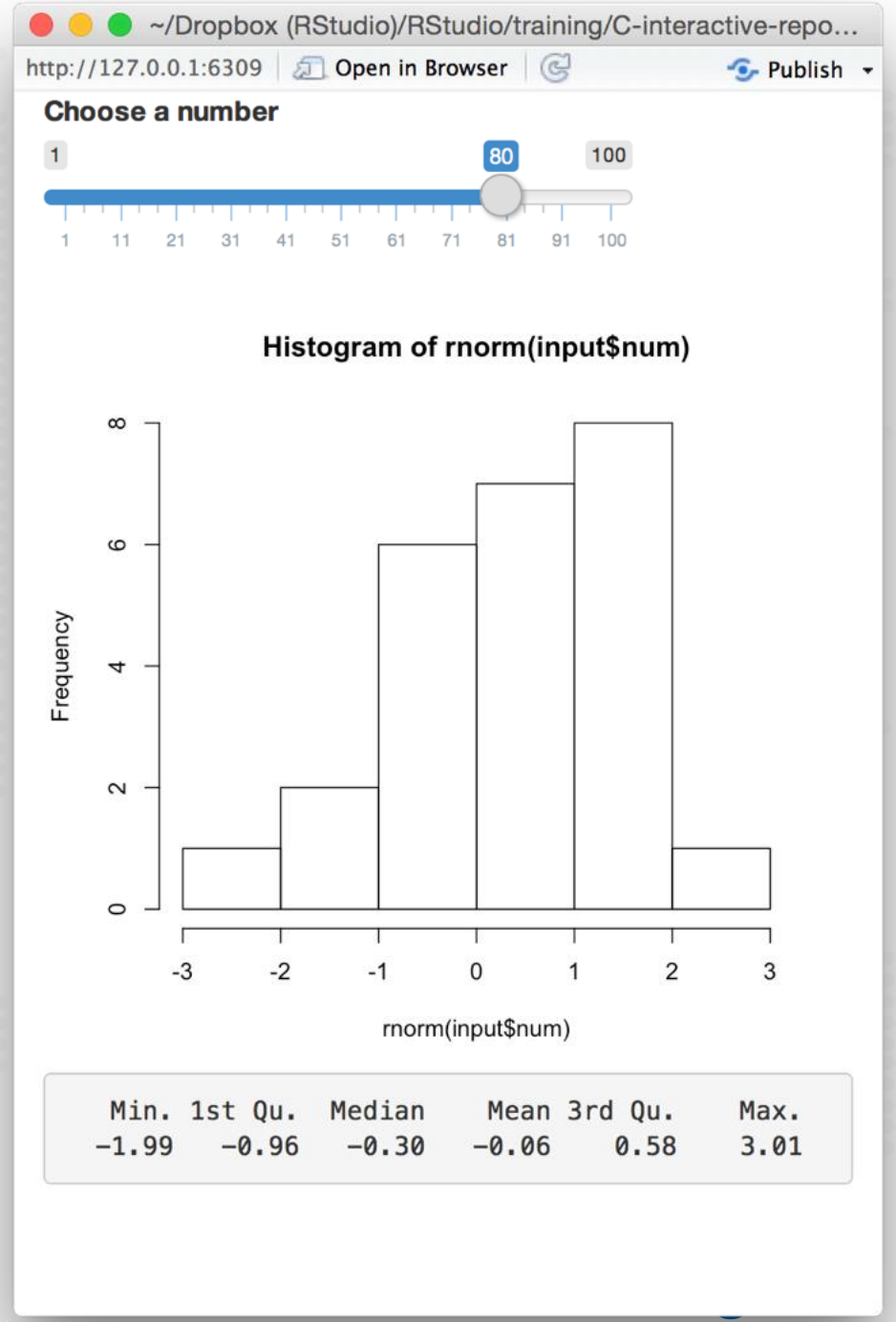
```
# 02-two-outputs

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)
```



input$num

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

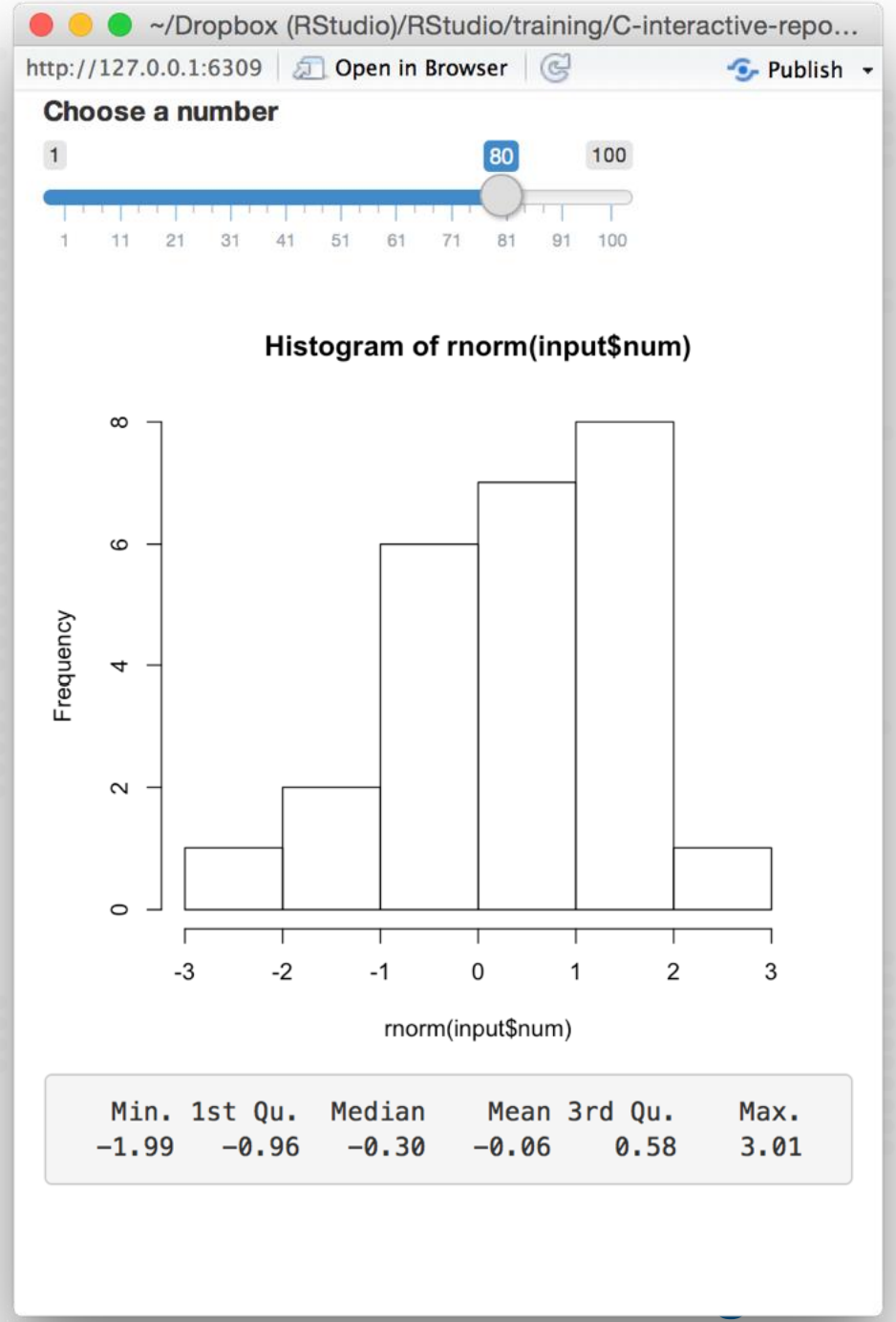
```
output$stats <-
  renderPrint({
  summary(rnorm(input$num))
  })
```



input$num

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

```
output$stats <-
  renderPrint({
  summary(rnorm(input$num))
  })
```

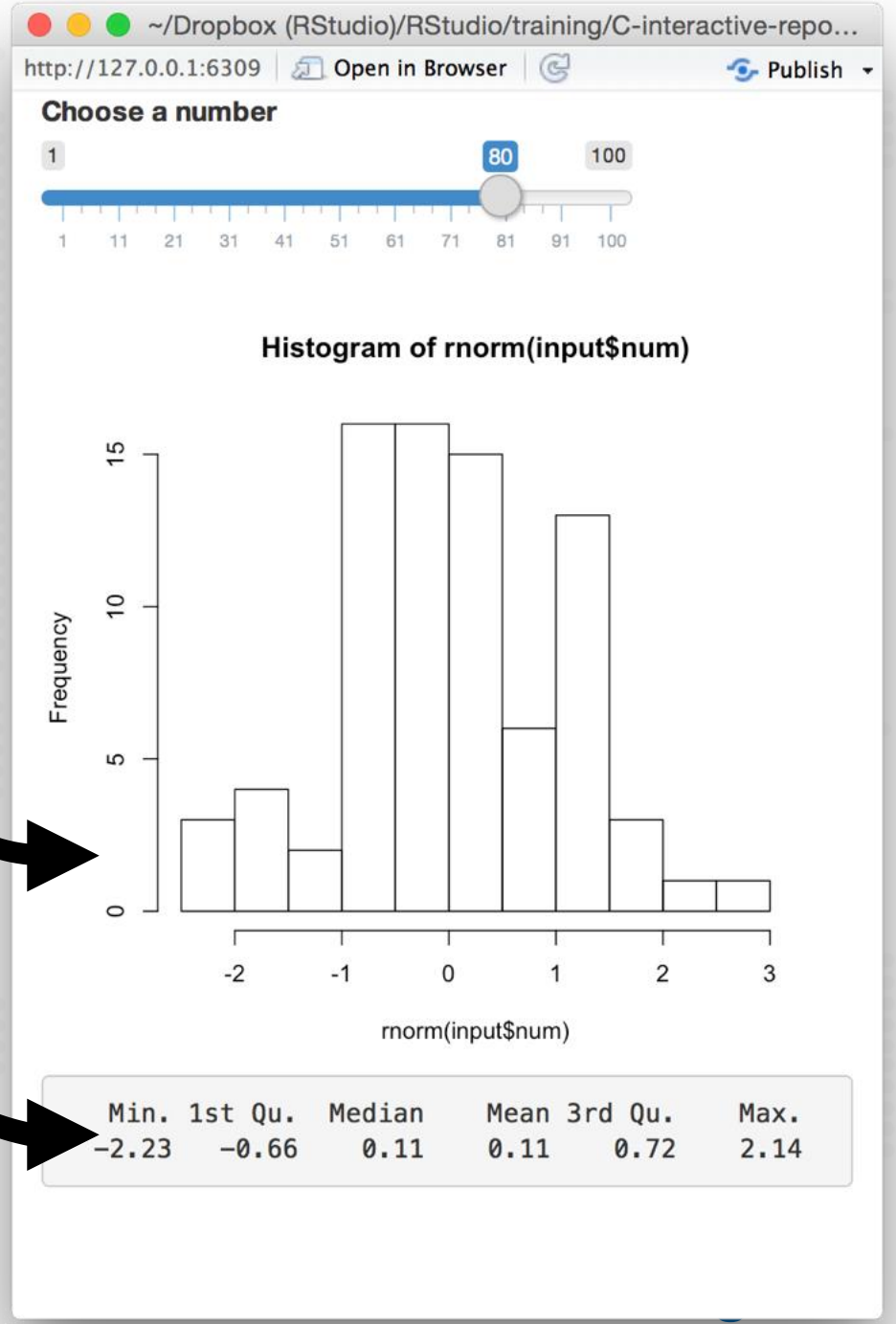


`input$num`

Can these describe the same data?

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

```
output$stats <-
  renderPrint({
  summary(rnorm(input$num))
  })
```

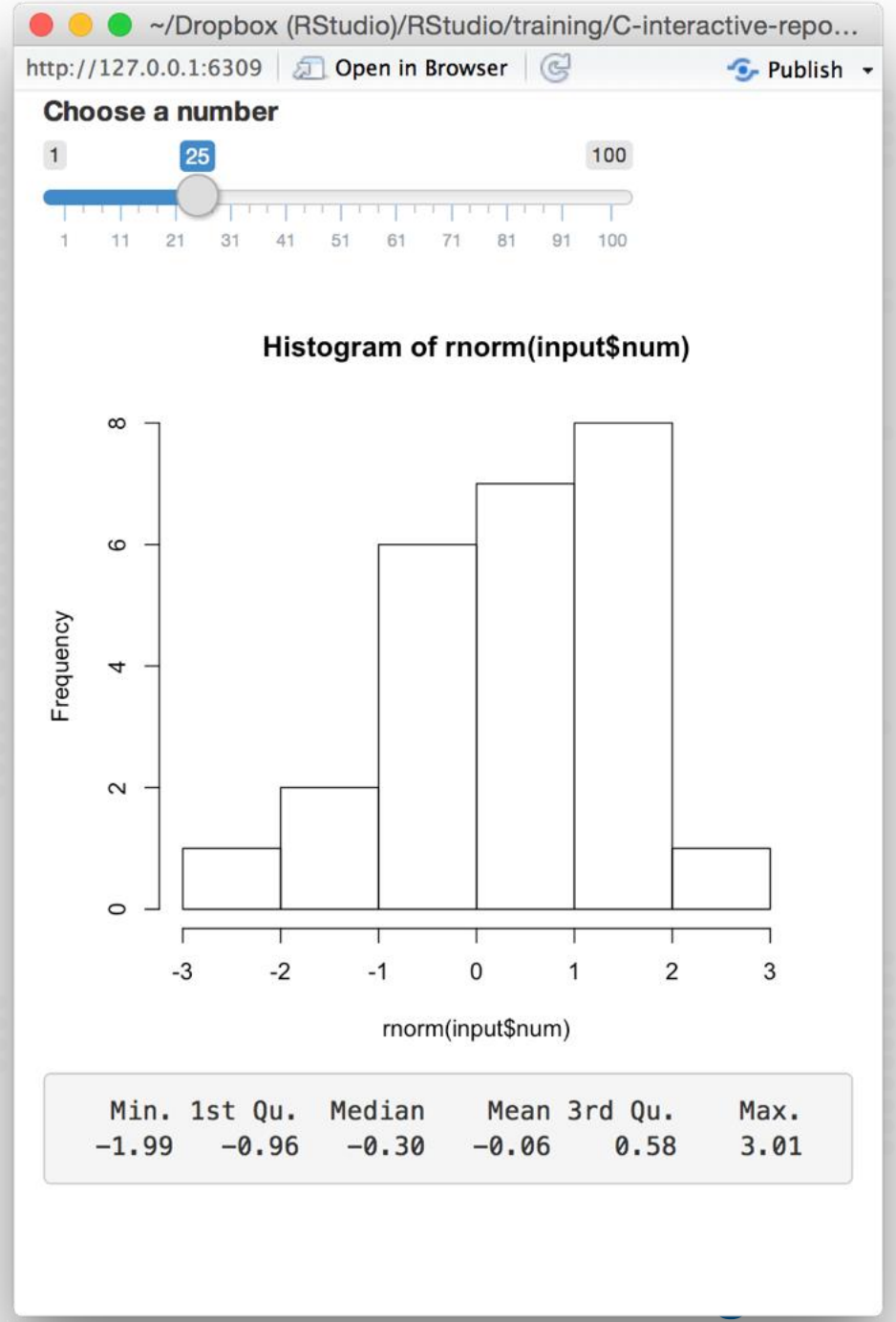


```
input$num
```

```
data <-? rnorm(input$num)
```

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

```
output$stats  <-
   renderPrint({
     summary(data)
  })
```



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

NobleProg

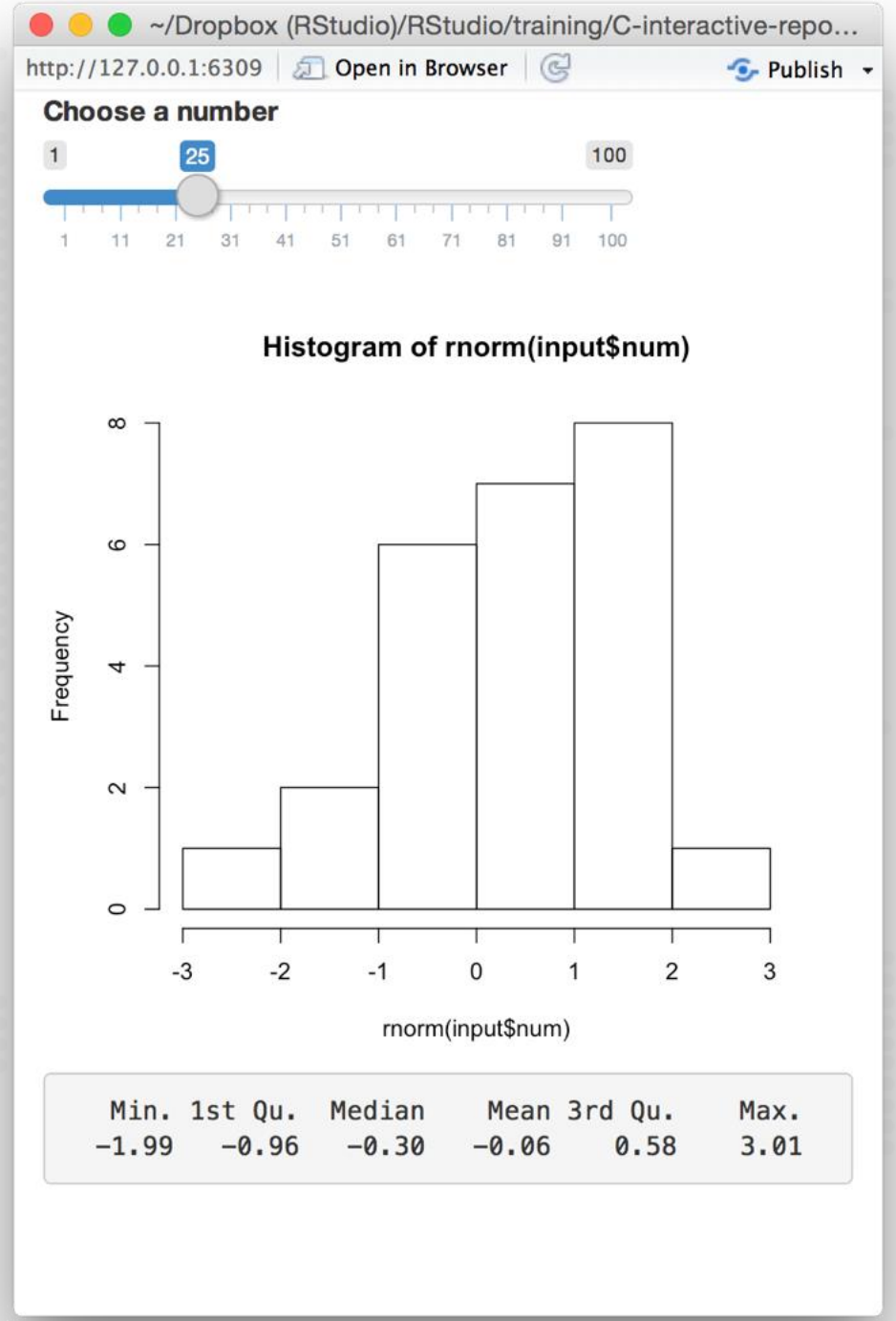
A reactive expression is special in two ways

```
data()
```

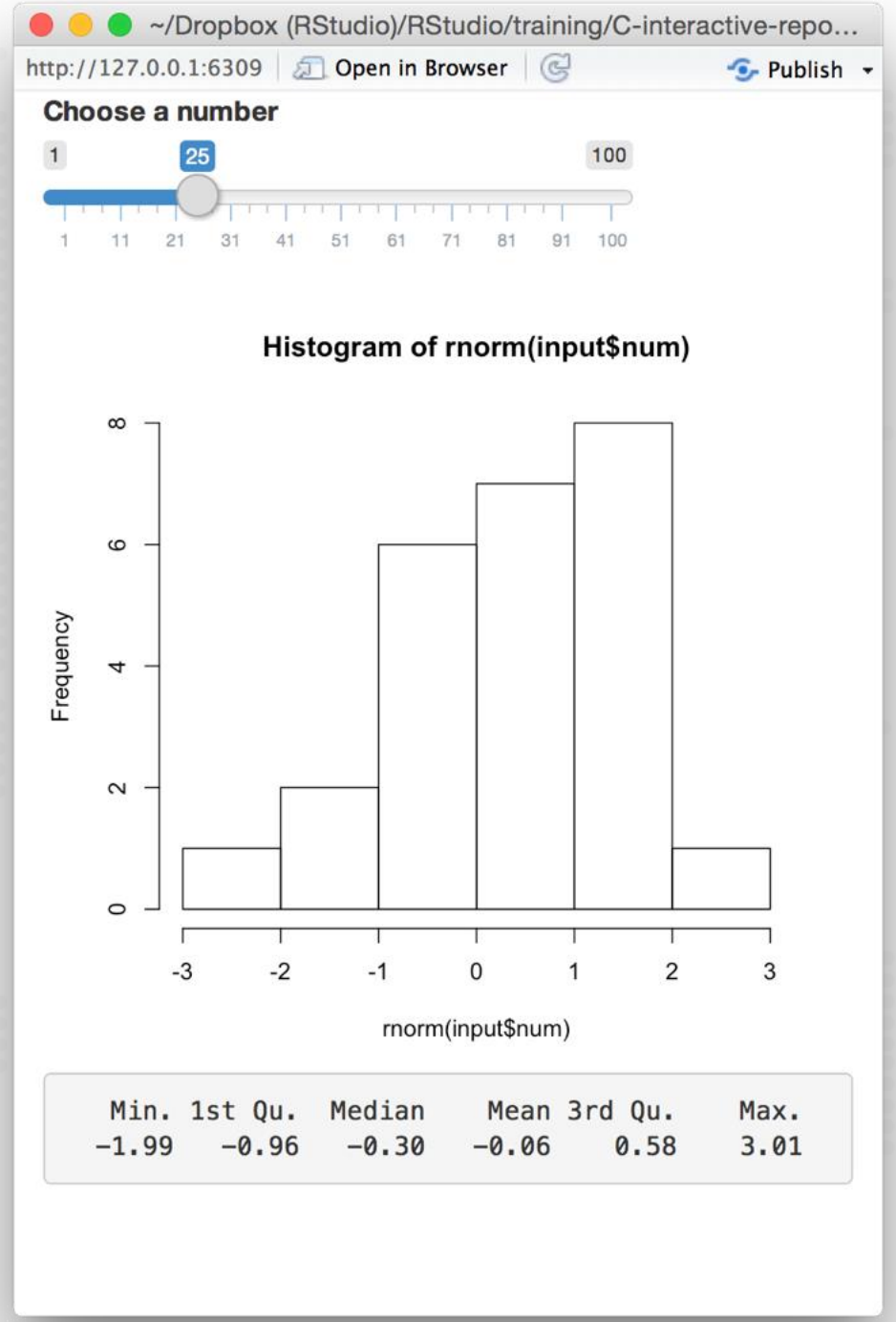
1. You call a reactive expression like a function

NobleProg

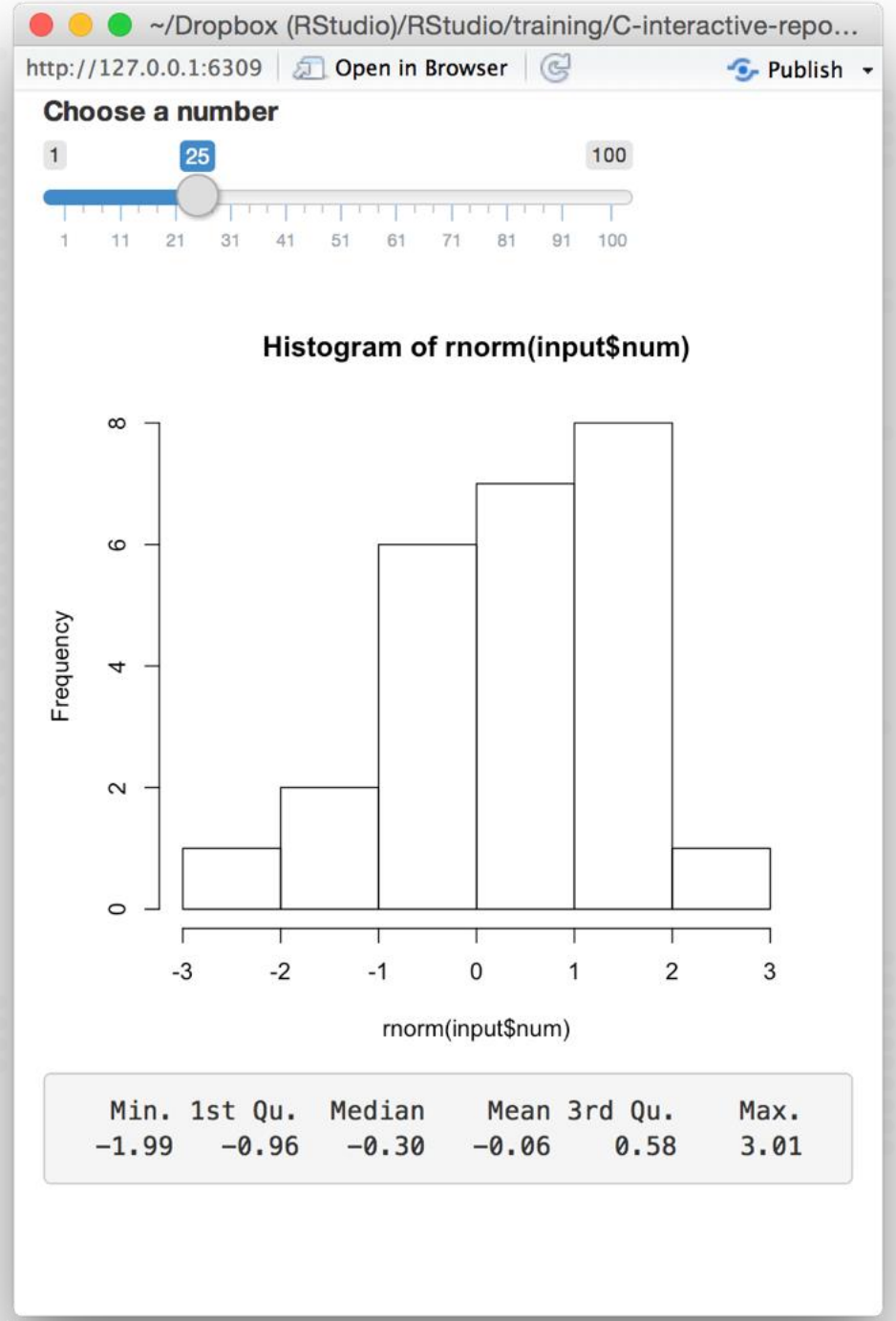
```
# 02-two-outputs

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 =
```

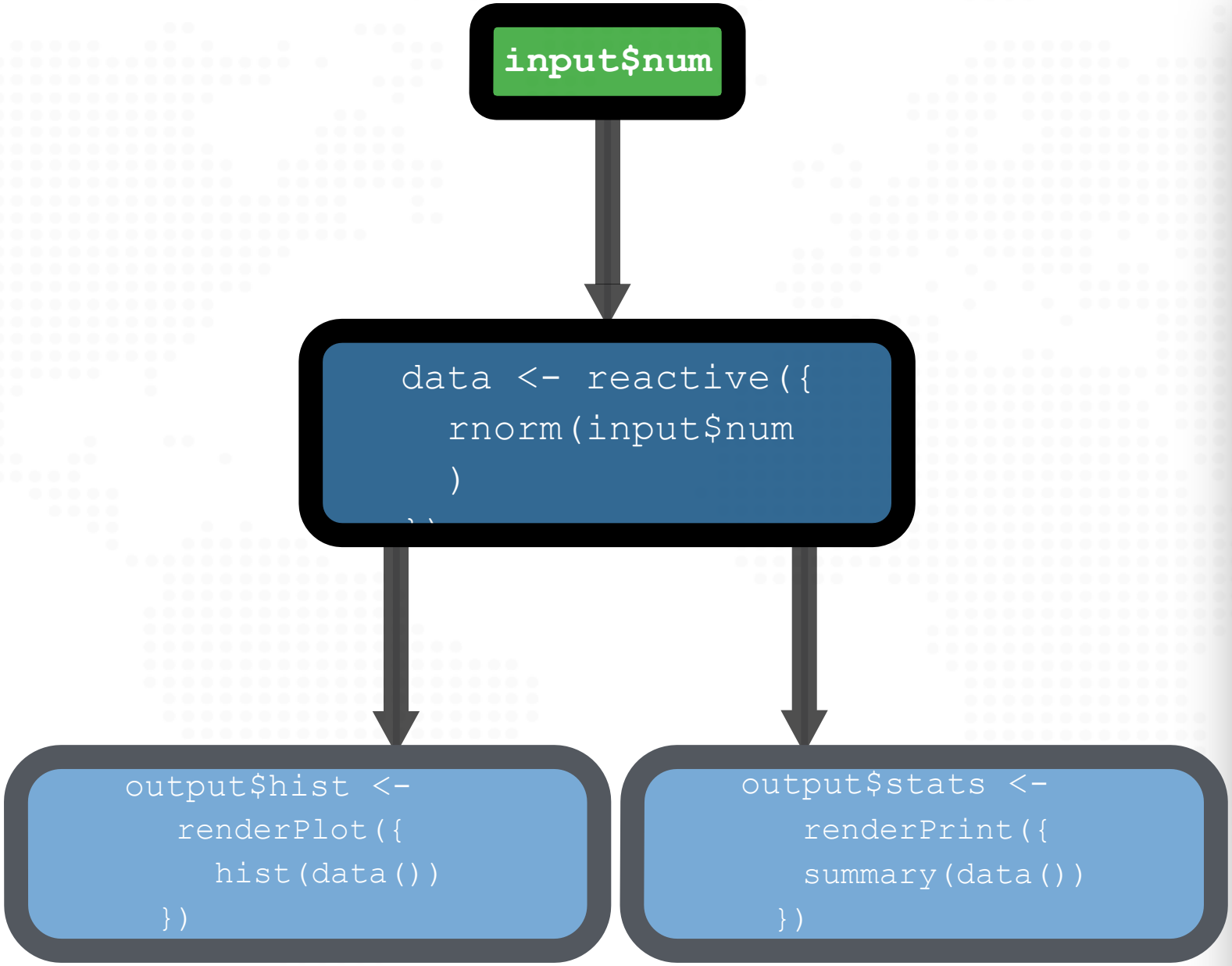


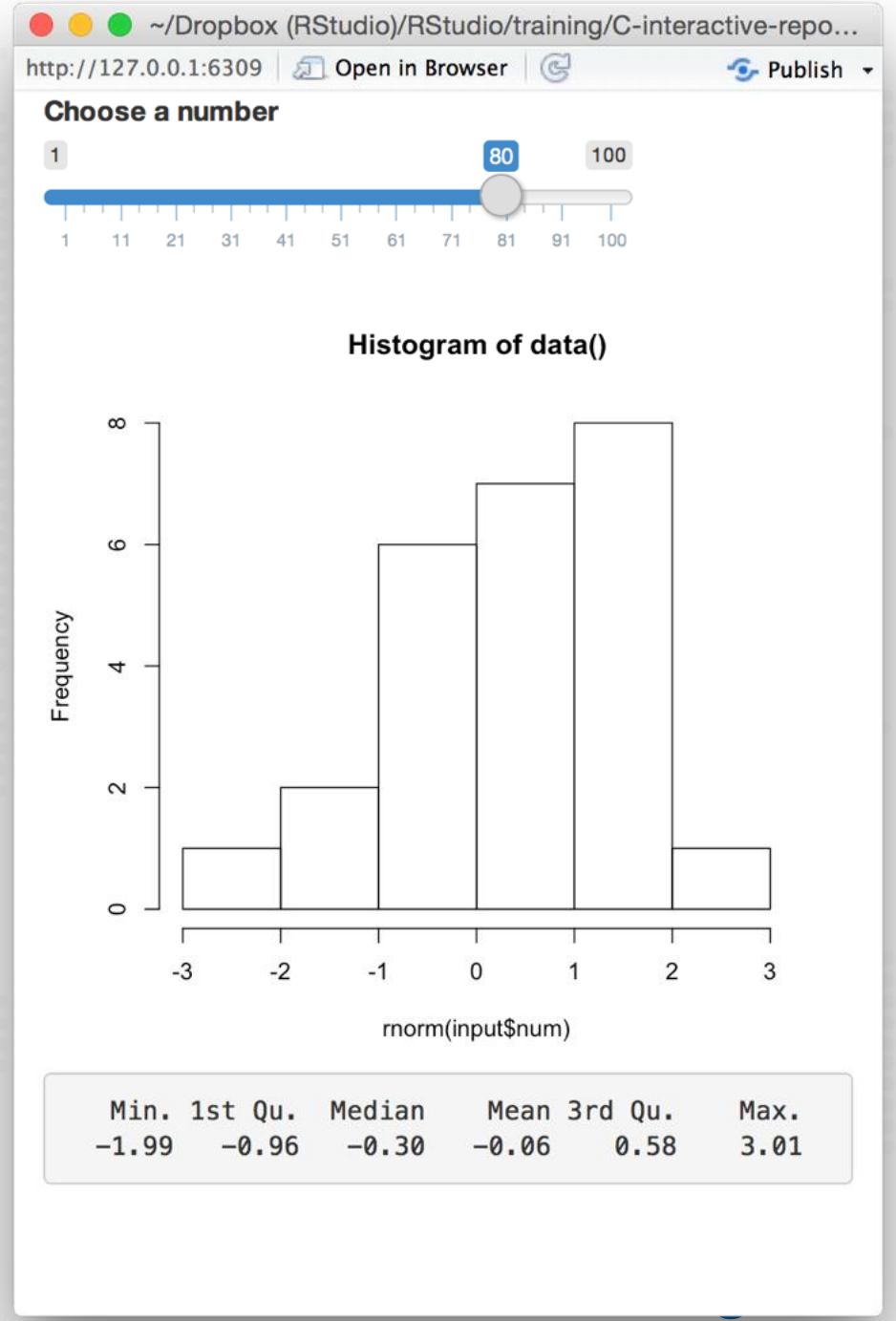
```
# 02-two-outputs

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

shinyApp(ui = ui, server =
```

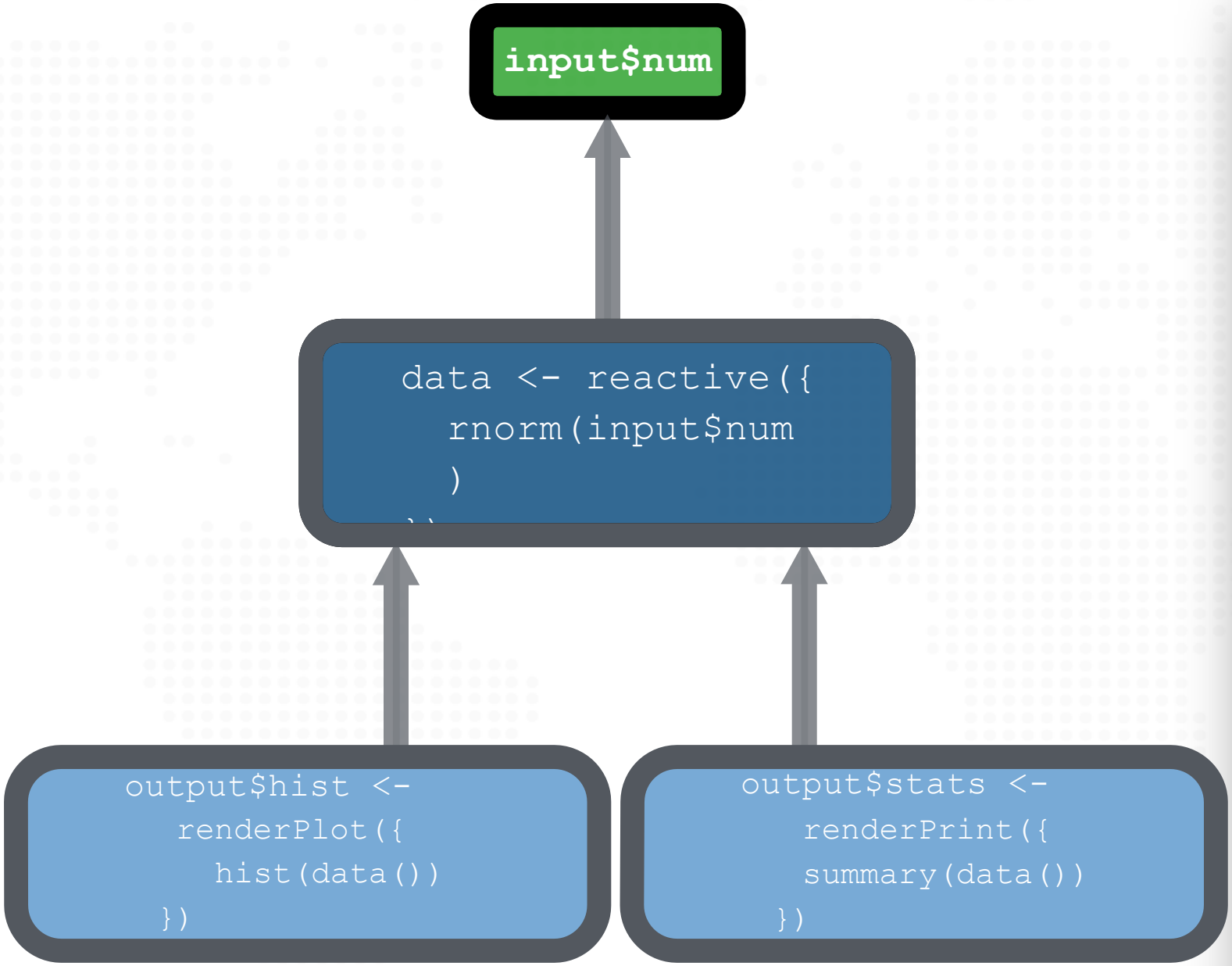


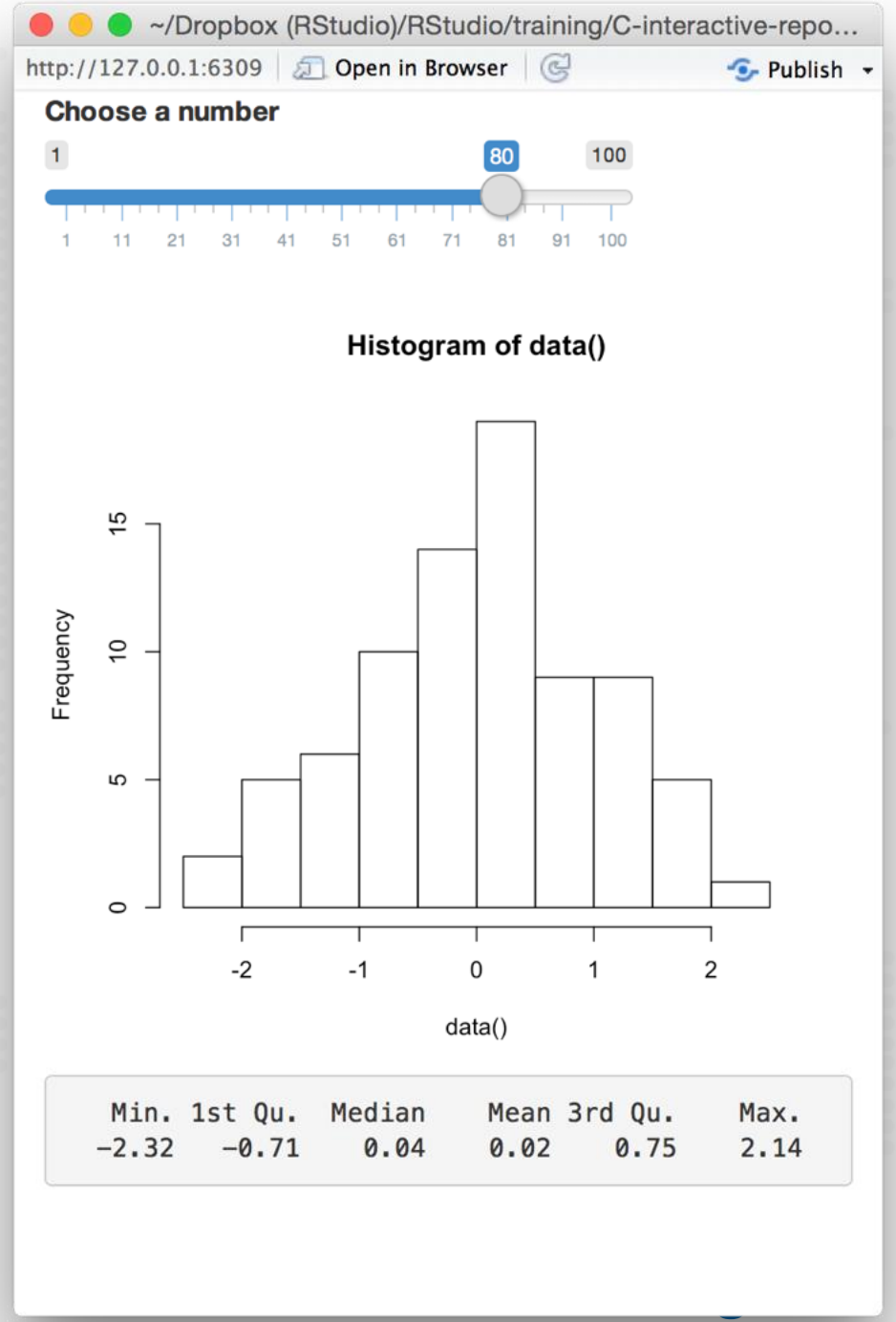
```
# 03-reactive

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 =
```











A reactive expression is special in two ways

```
data()
```

1. You call a reactive expression like a function

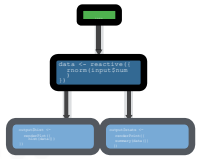
2. Reactive expressions **cache** their values
(the expression will return its most recent value,
unless it has become invalidated)

NobleProg

Recap: reactive()

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

reactive() makes an **object to use** (in downstream code)



Reactive expressions are themselves **reactive**. Use them to modularize your apps.

`data()`

Call a reactive expression like a **function**

2

Reactive expressions **cache** their values to avoid unnecessary computation

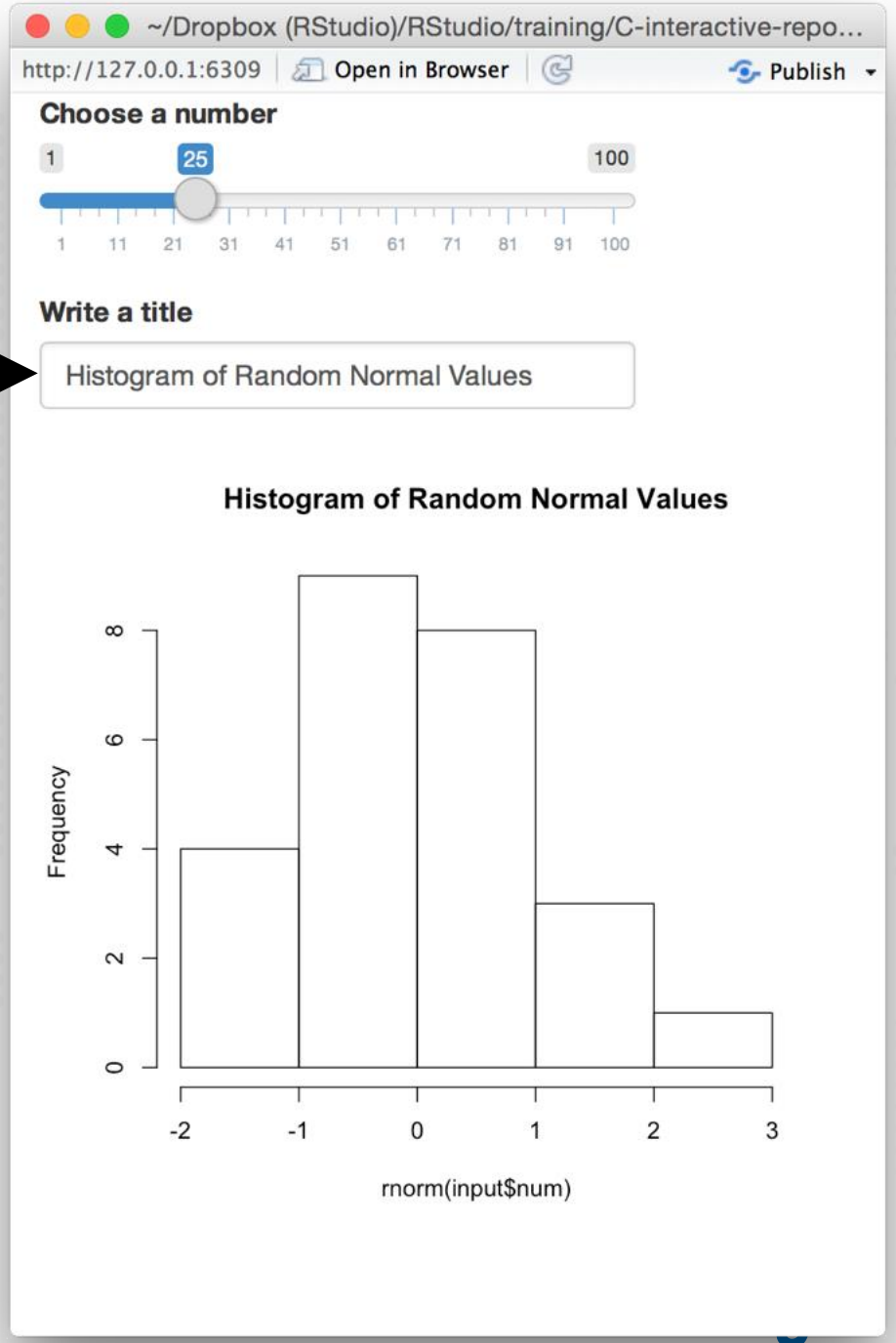
NobleProg

Prevent reactions with isolate()

NobleProg

```
# 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)
```

Can we prevent the title field from updating the plot?



isolate()

Returns the result as a non-reactive value

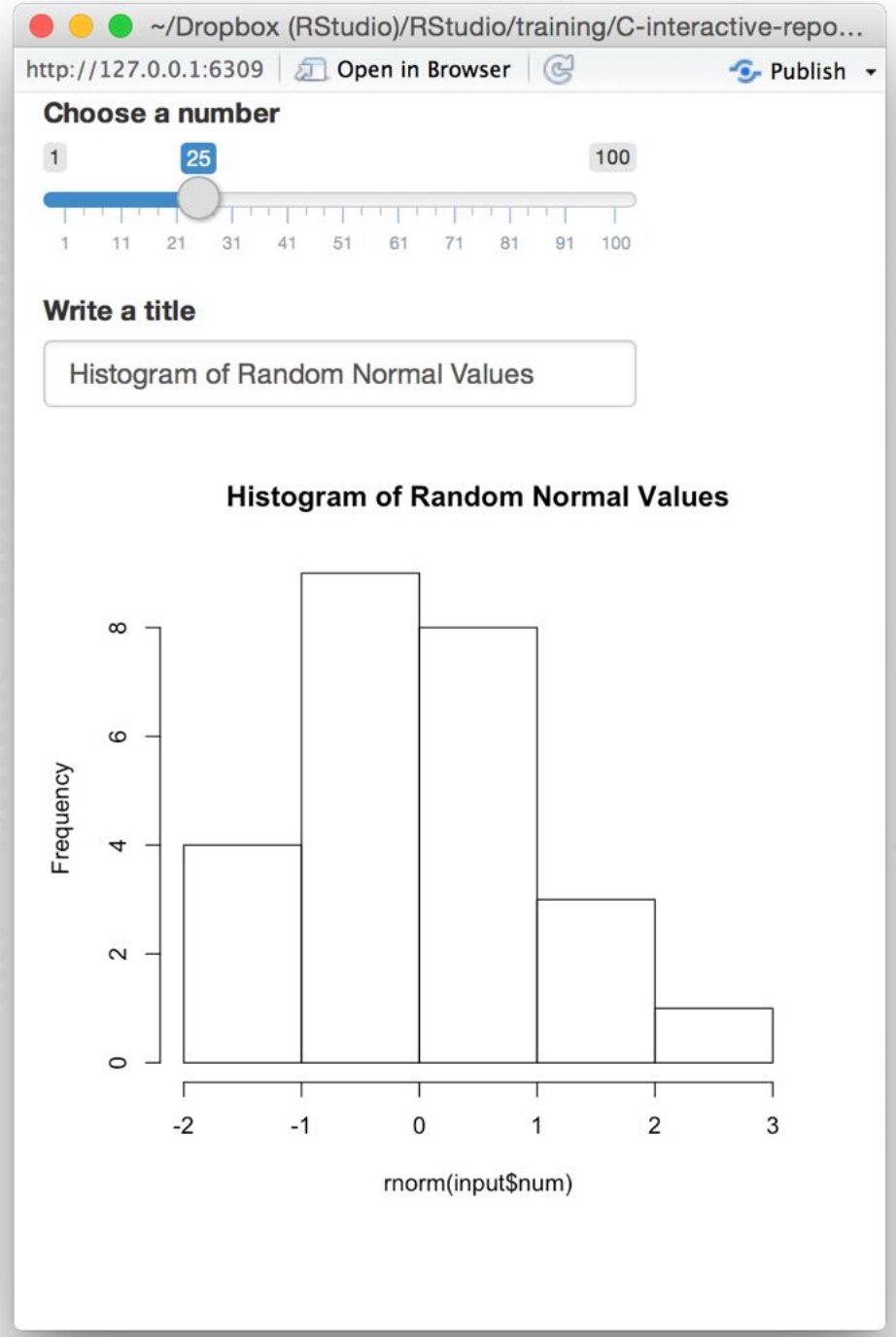
```
isolate({ rnorm(input$num) })
```

object will NOT respond to *any reactive value in the code*

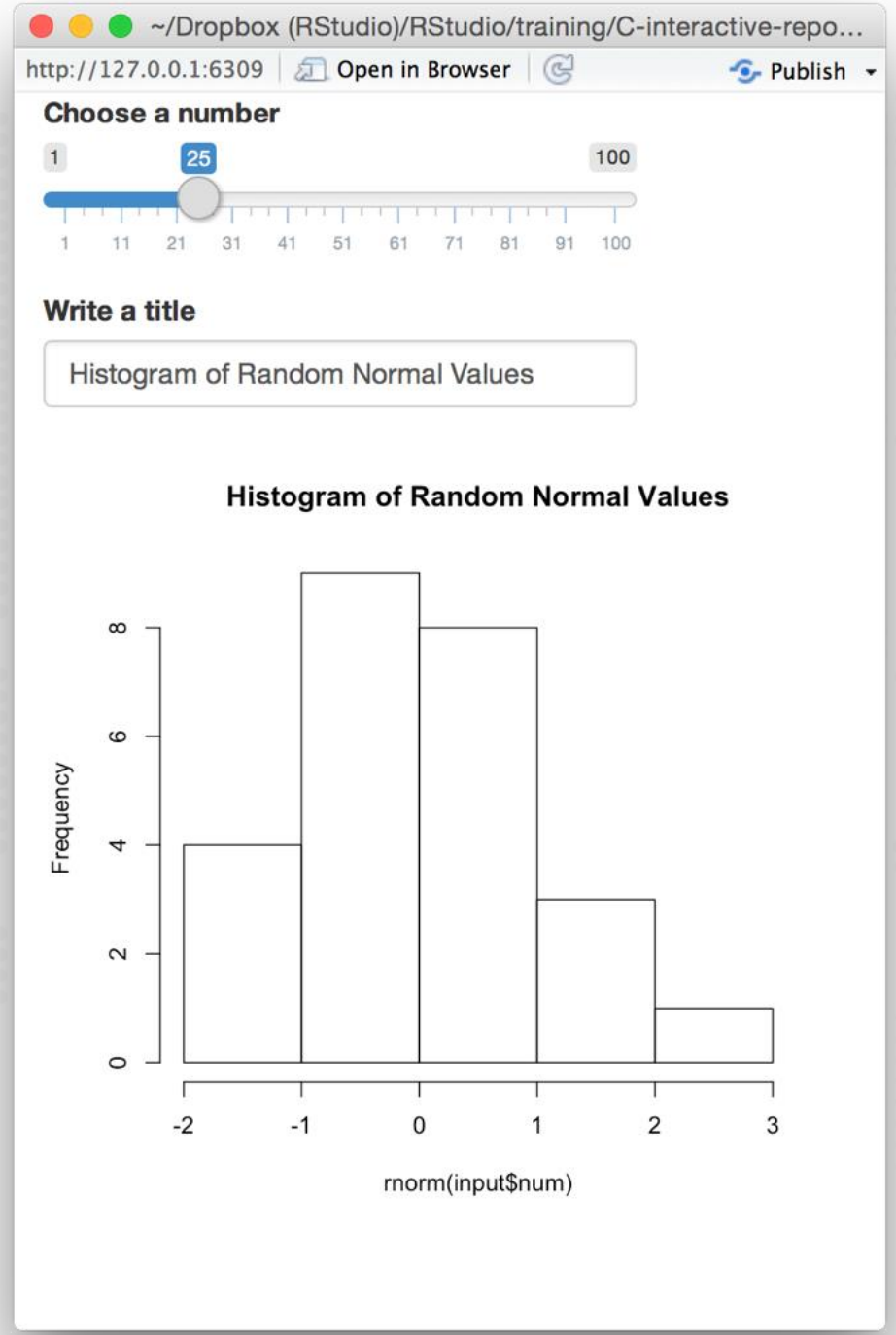
code used to build object

NobleProg

```
# 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)
```



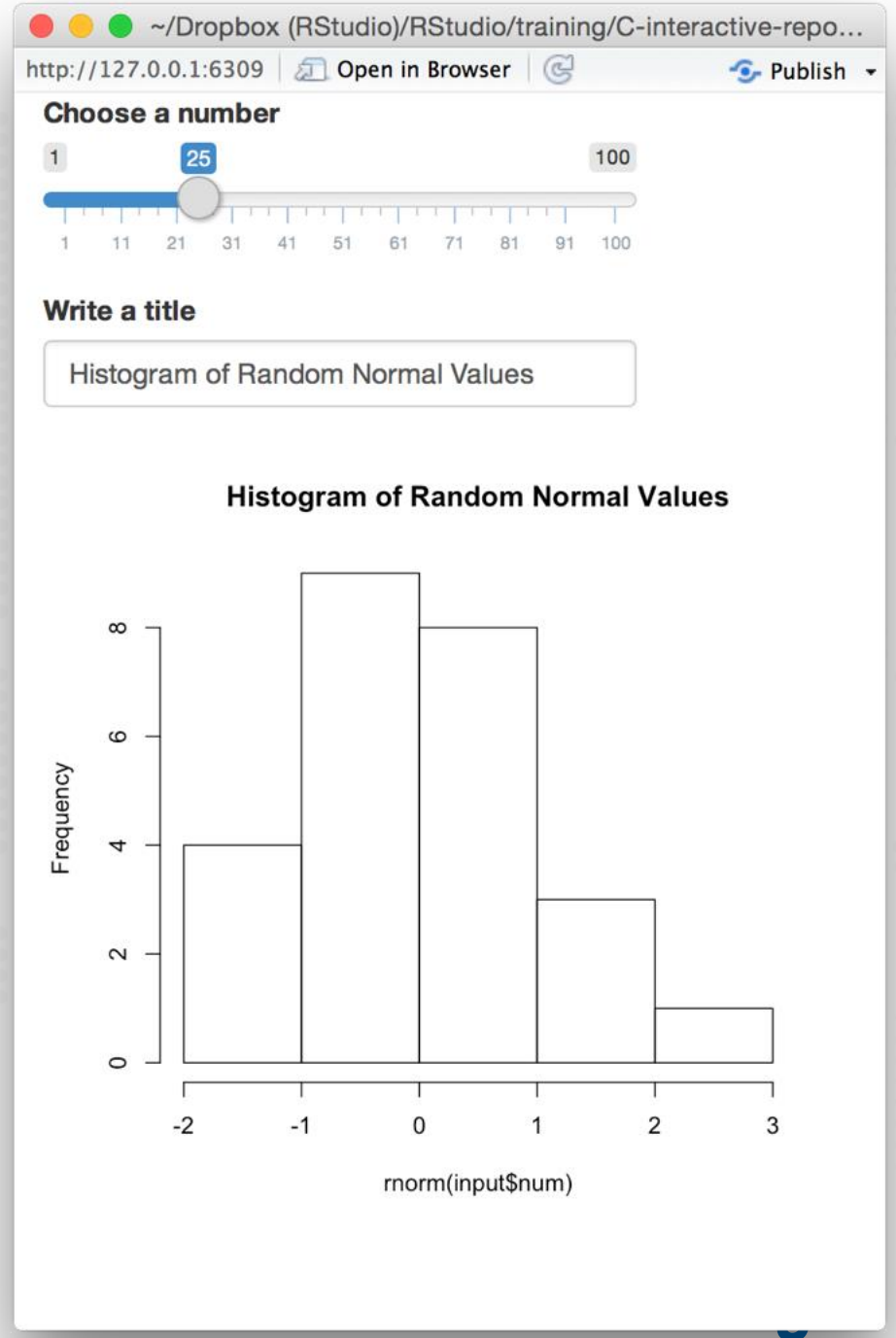
```
# 04-isolate

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)
```



input$num

input$title

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

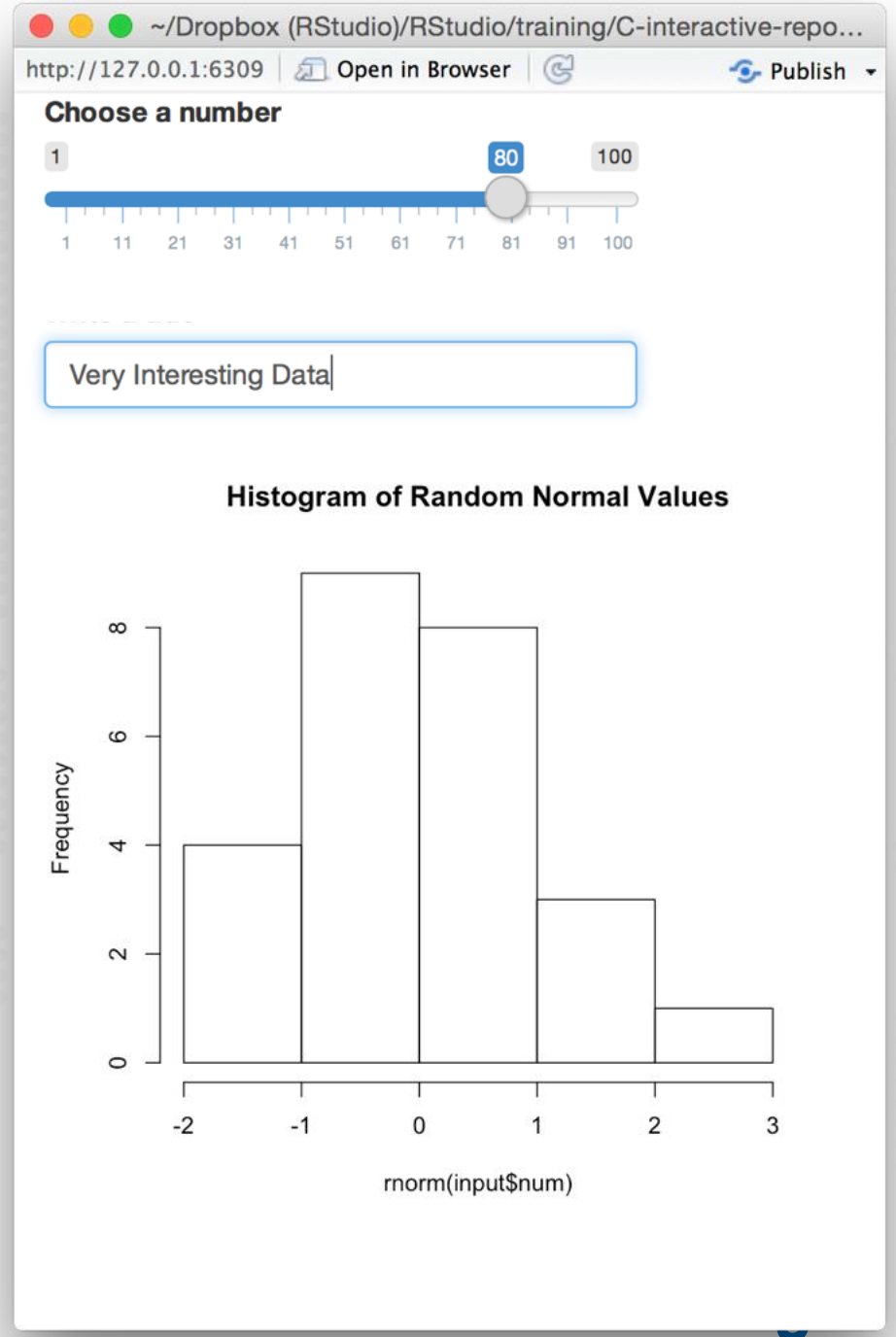


input$num

input$title

```
output$hist <-
  renderPlot({
  hist(rnorm(input$num),
    main =
```

```
})
```

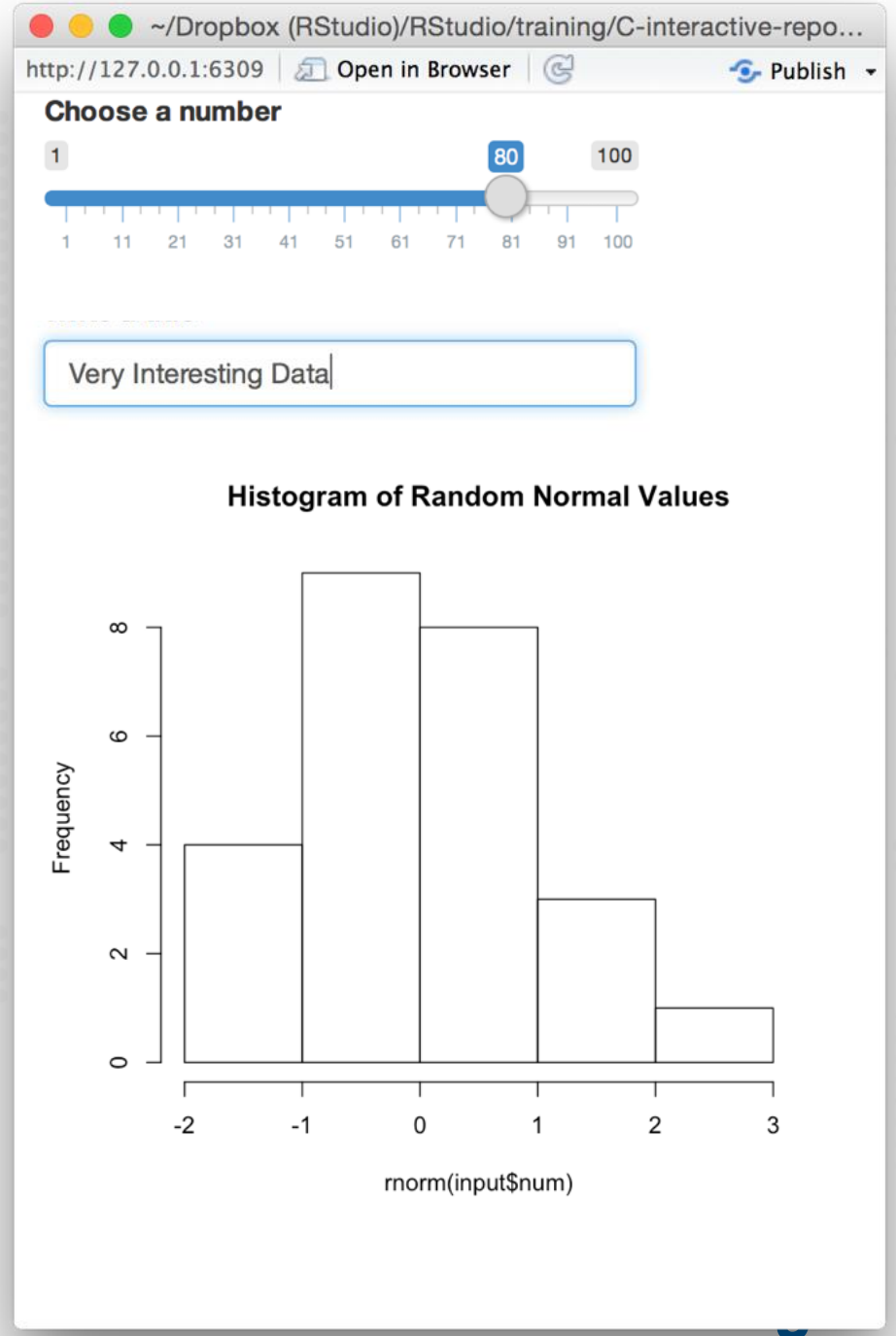


input$num

input$title

```
output$hist <-
  renderPlot({
  hist(rnorm(input$num),
    main =
```

```
})
```



Recap: isolate()

isolate() makes an **non-reactive object**

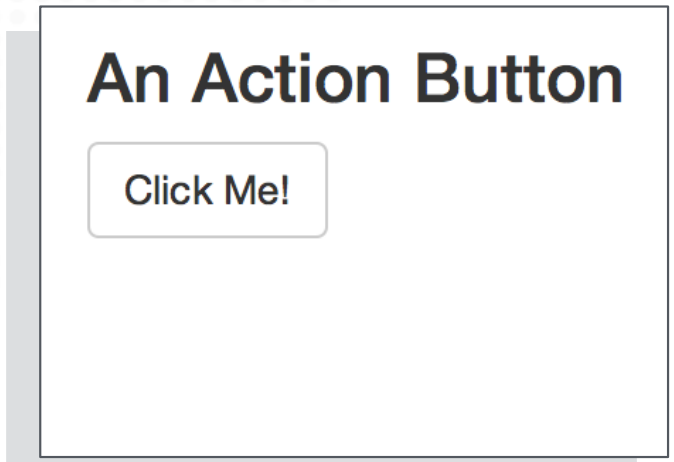
Use isolate() to treat reactive values like normal R values

NobleProg

Trigger code with observeEvent()

NobleProg

NobleProg



input function

input name (for internal use)

label to display

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

Notice: Id not ID

NobleProg

```
# 05-actionButton

library(shiny)

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

server <- function(input, output) {



}

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

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

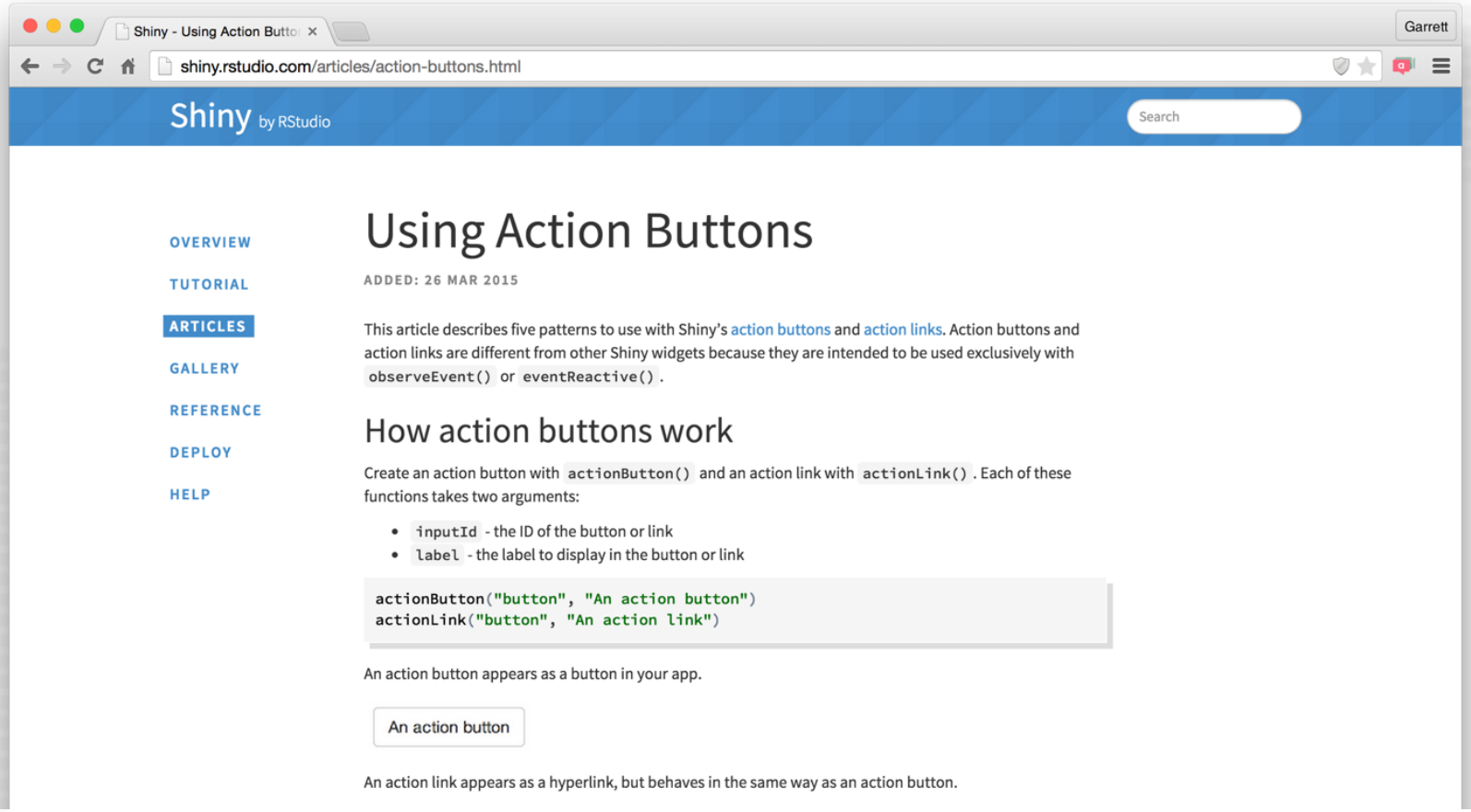
note: observer treats this code as if it has been isolated with isolate()

NobleProg

```
# 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)
```

Action buttons article

http://shiny.rstudio.com/articles/action-buttons.html



NobleProg

observe()

Also triggers code to run on server.

Uses same syntax as render*(), reactive(), and isolate()

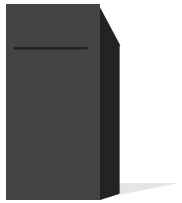
```
observe({ print(input$clicks) })
```

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

code block to run whenever observer is invalidated

NobleProg

Recap: observeEvent()



observeEvent() **triggers code to run** on the server

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

reactive value(s) to respond to

Specify **precisely** which reactive values should invalidate the observer

`observe()`

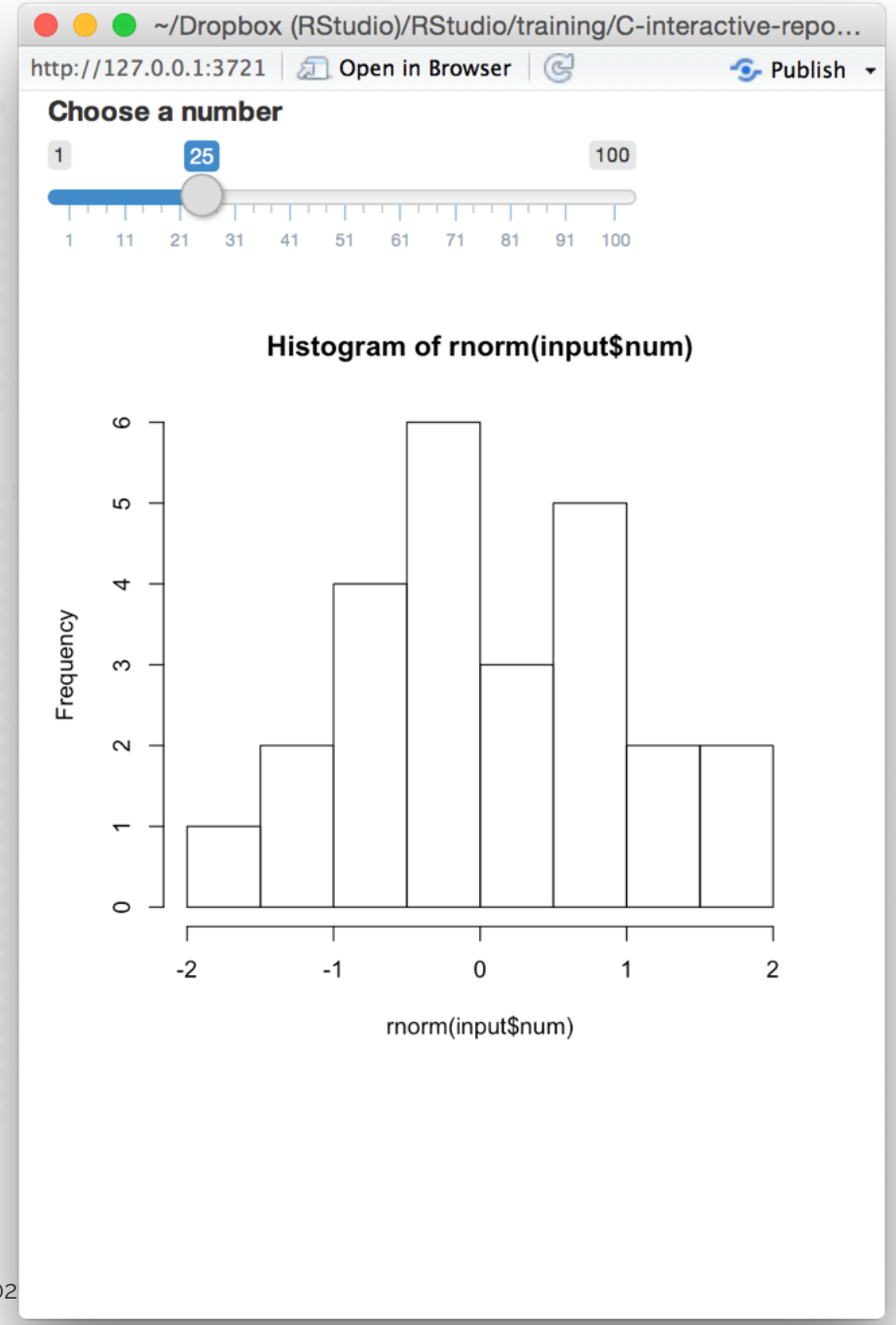
Use **observe()** for a more implicit syntax

NobleProg

Delay reactions with eventReactive()

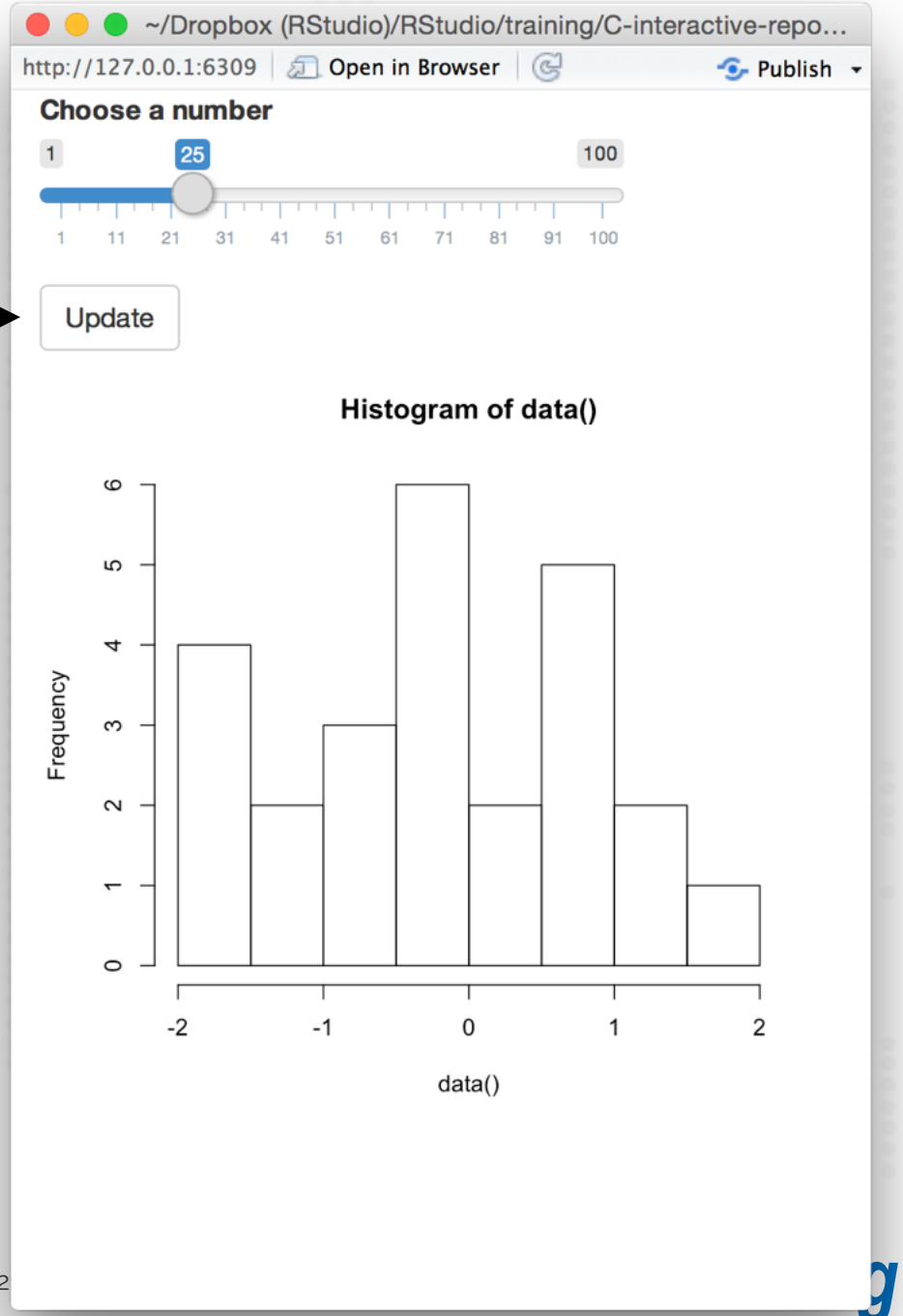
NobleProg

```
# 07-eventReactive

library(shiny)

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

server <- function(input, output)
{



  output$hist <-
    renderPlot({
    hist(rnorm(input$num))
  })
}

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



```
# 07-eventReactive

library(shiny
)
ui <- fluidPage(
  sliderInput(inputId =
  "num",
    label = "Choose a number",
    value = 25, min = 1, max =
    100),
  actionButton(inputId =
    "go", label = "Update"),
)  plotOutput("hist")
server <- function(input, output)
{



  output$hist <-
    renderPlot({
    hist(rnorm(input$num))
  })
}


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

Can we prevent the graph from updating until we hit the button?



eventReactive()

A reactive expression that only responds to specific values

```
data <- eventReactive(input$go,  { rnorm(input$num) })
```

reactive value(s) to respond to

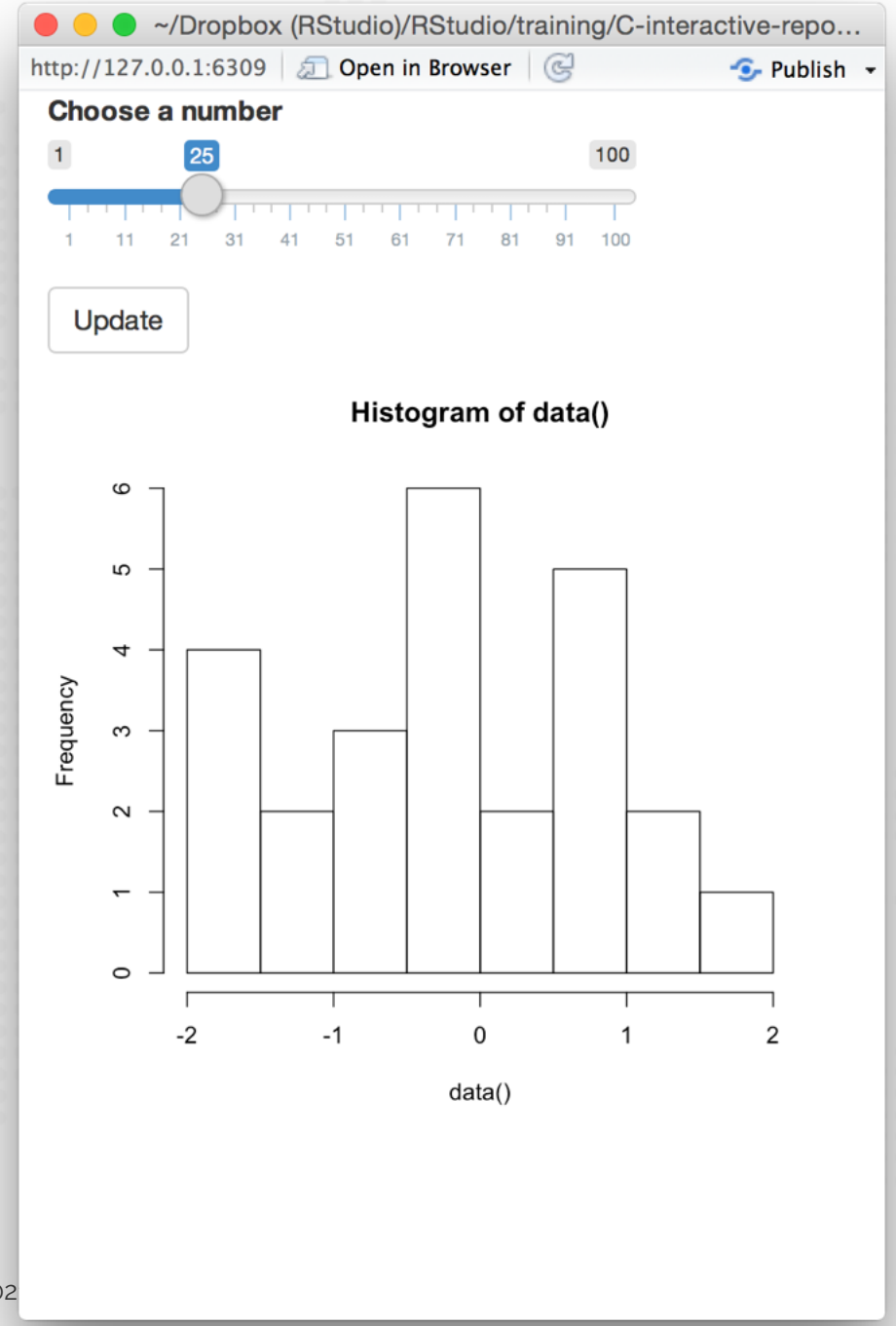
(expression invalidates ONLY when this value changes)

code used to build (and rebuild) object

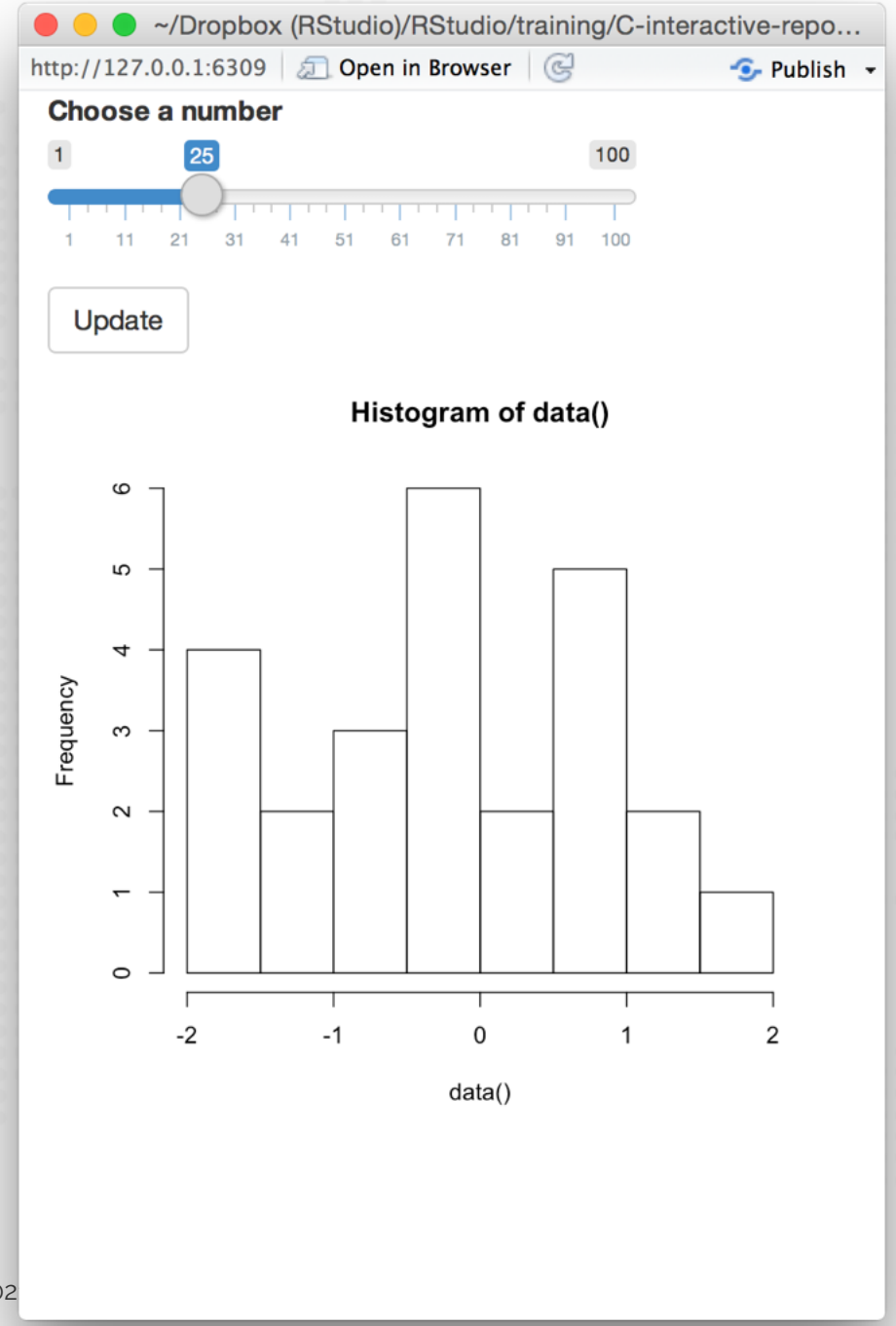
note: expression treats this code as if it has been isolated with isolate()

NobleProg

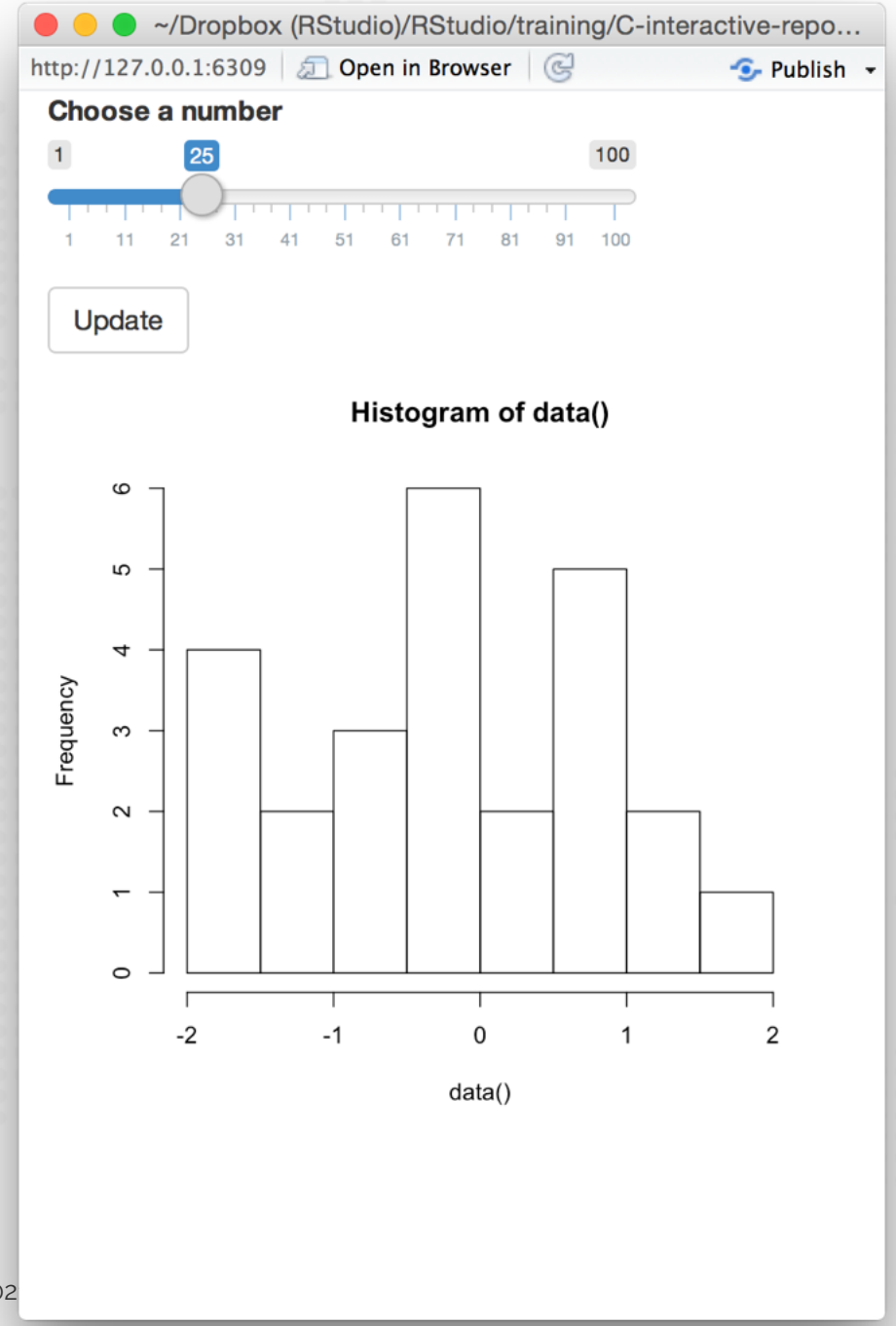
```
# 07-eventReactive

library(shiny)

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

server <- function(input, output) {

  output$hist <-
    renderPlot({
    hist(rnorm(input$num))
  })
}

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



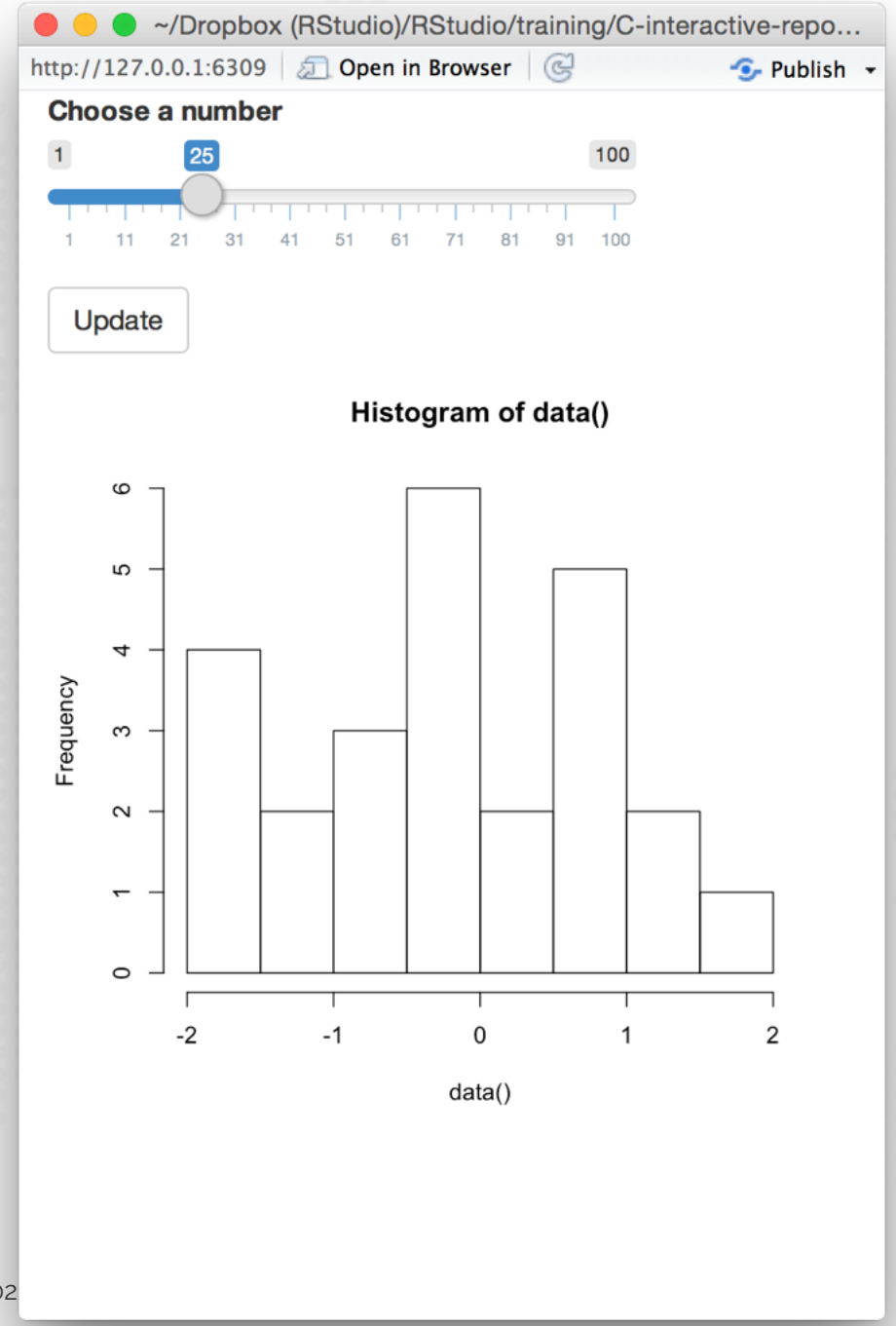
```
# 07-eventReactive

library(shiny)

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


  server <- function(input, output)
                                    {
  data <- eventReactive(input$go, {

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



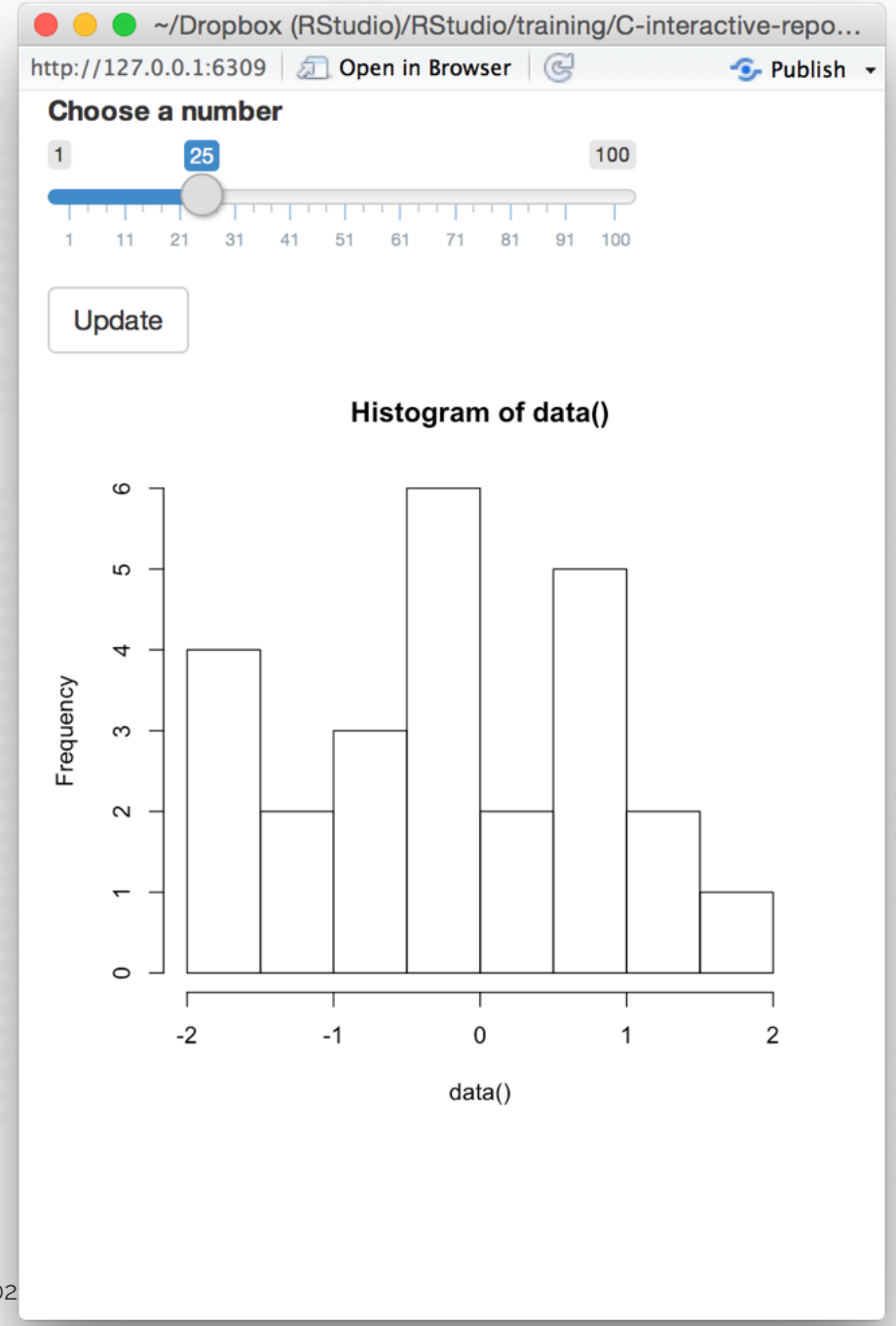
```
# 07-eventReactive

library(shiny)

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


  server <- function(input, output)
                                    {
  data <- eventReactive(input$go, {

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



```
# 07-eventReactive

library(shiny)

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

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

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



```
# 07-eventReactive

library(shiny)

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

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

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

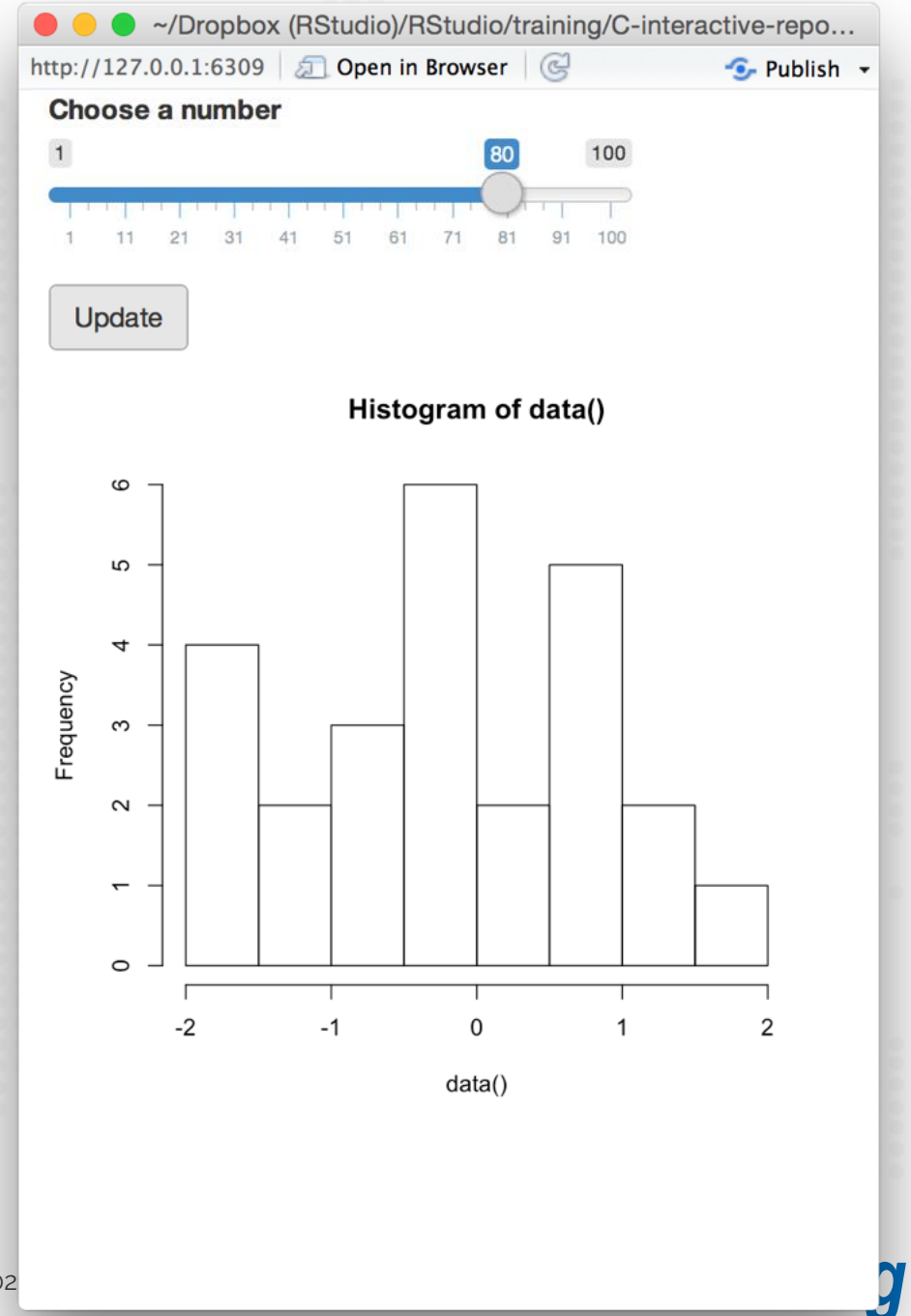


`input$num`

`input$go`

```
data <- eventReactive(input$go, {
  rnorm(input$num)
})
```

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

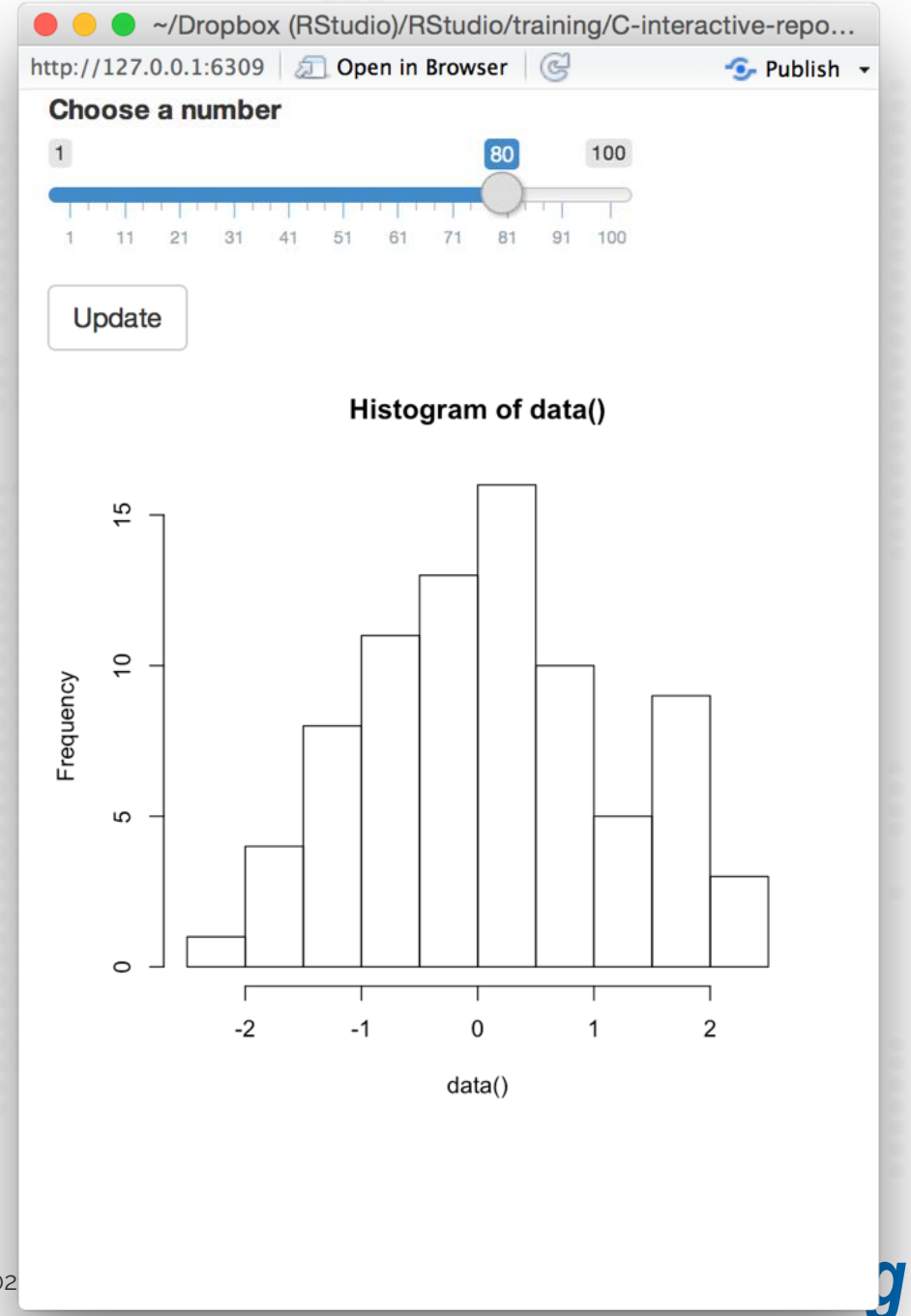


input$num

input$go

```
data <- eventReactive(input$go, {
  rnorm(input$num)
})
```

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



Recap: eventReactive()

Update

Use eventReactive() to **delay reactions**

`data()`

eventReactive() creates a **reactive expression**

You can specify **precisely** which reactive values should invalidate the expression

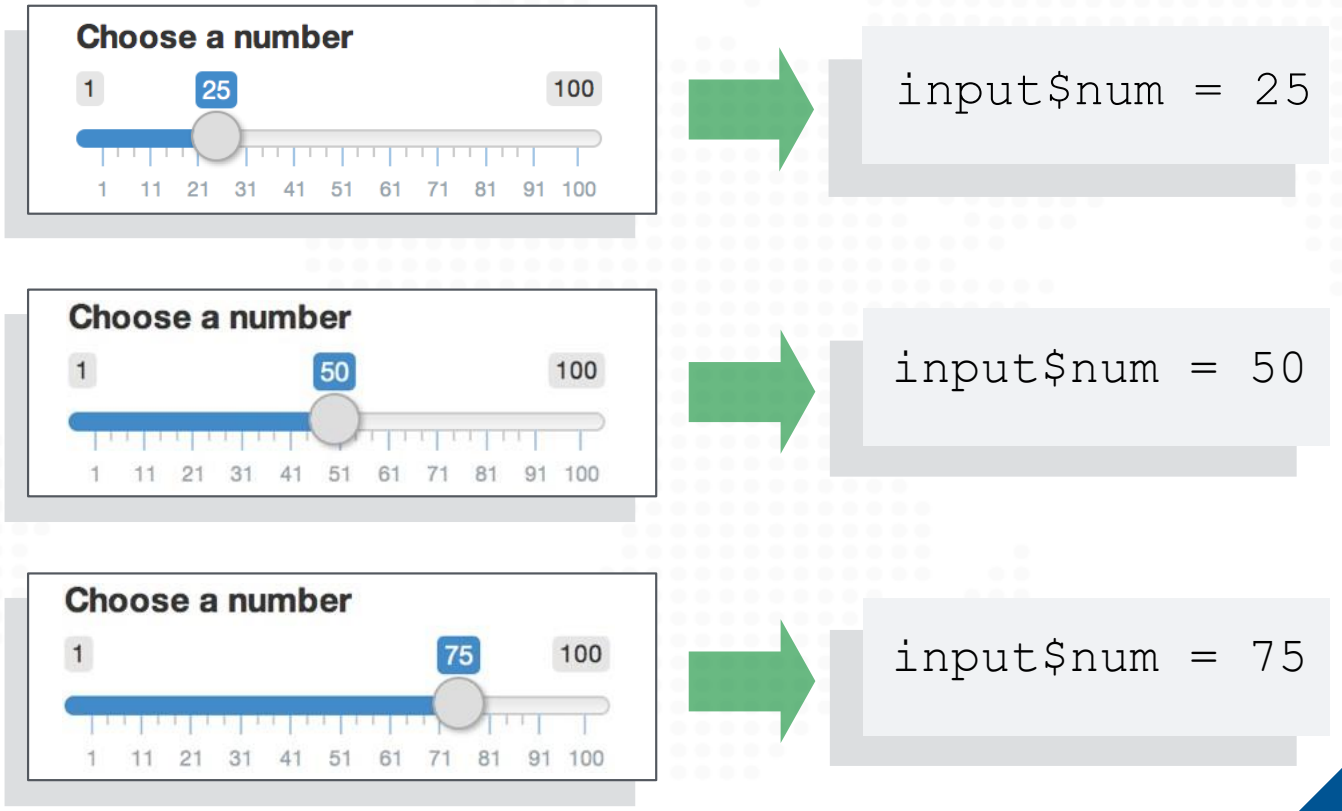
NobleProg

Manage state with reactiveValues()

NobleProg

Input values

The input value changes whenever a user changes the input.



You **cannot** set these values in your code

reactiveValues()

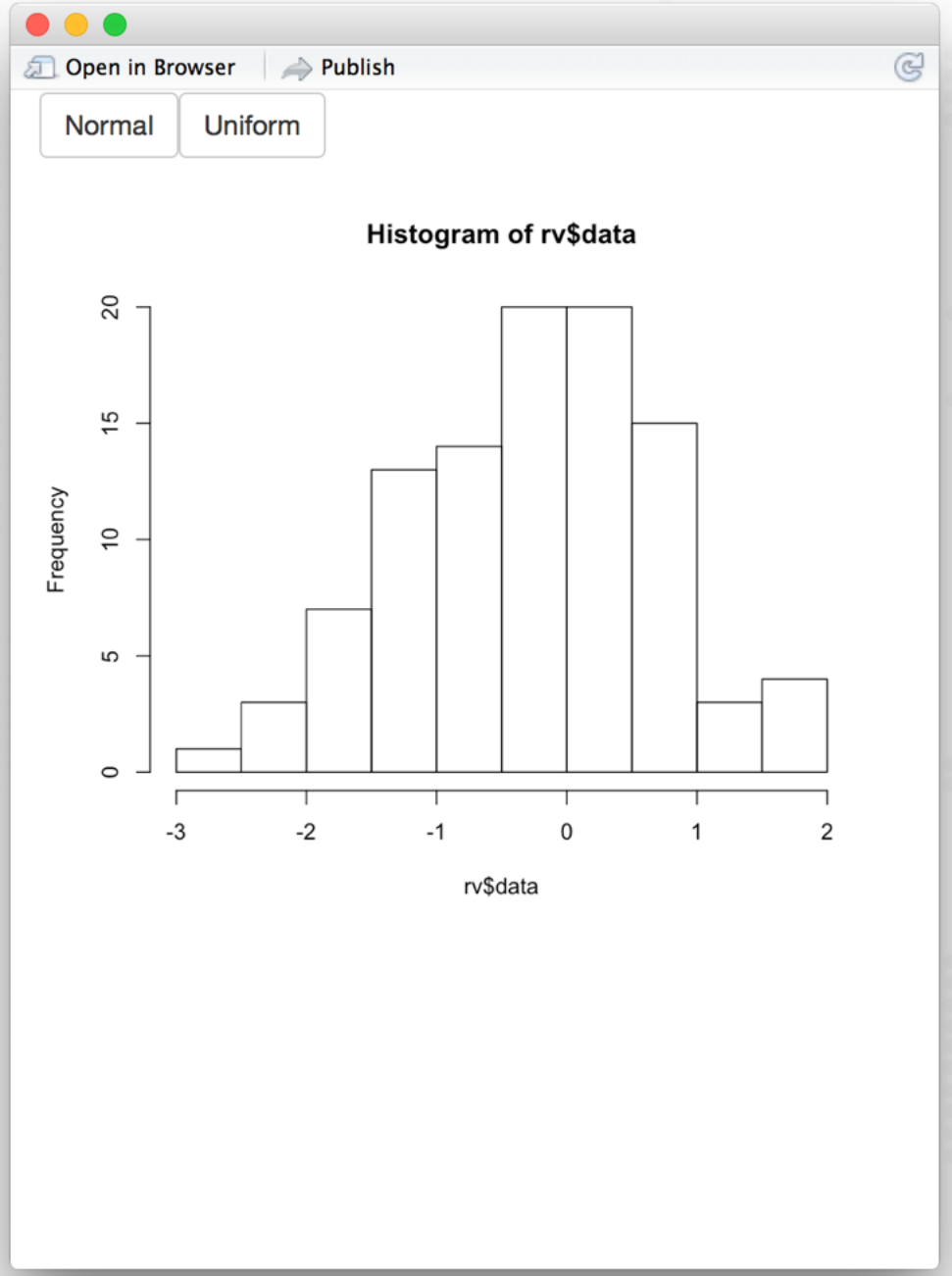
Creates a list of reactive values to manipulate programmatically

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

(optional) elements to add to the list

NobleProg

```
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)
```

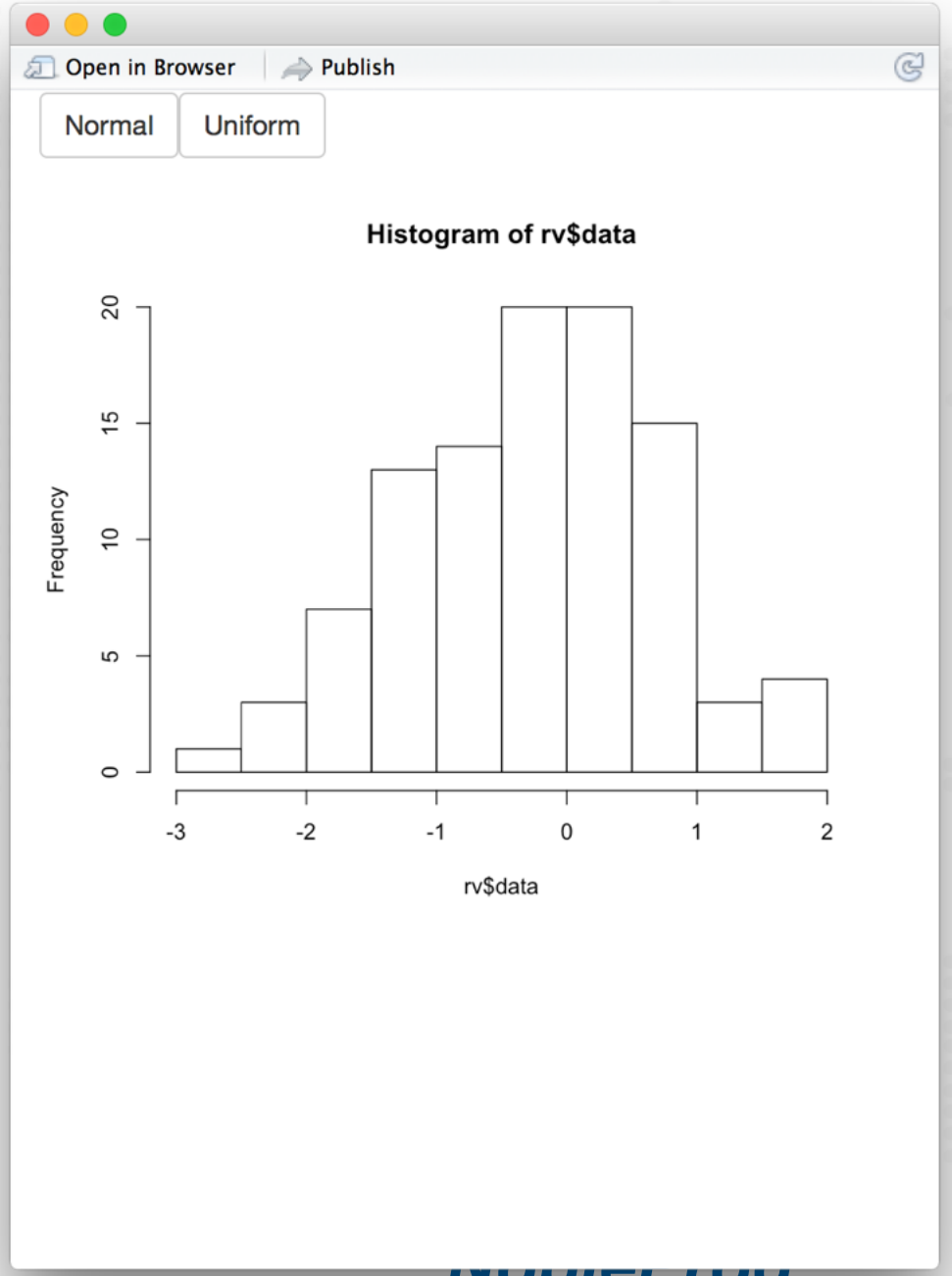


input$norm

rv$data
rnorm(100)

input$unif

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

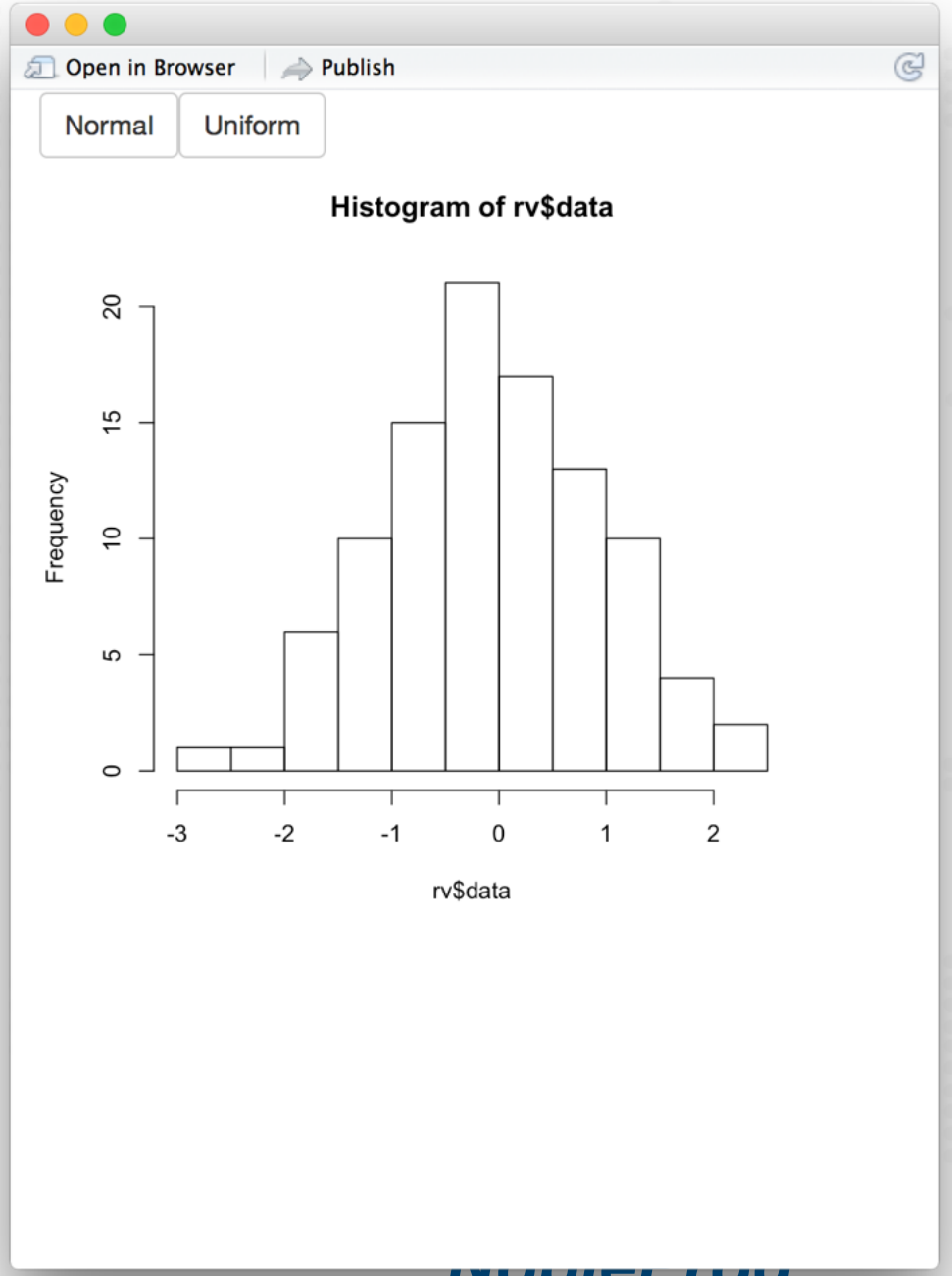


input$norm

rv$data
rnorm(100)

input$unif

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

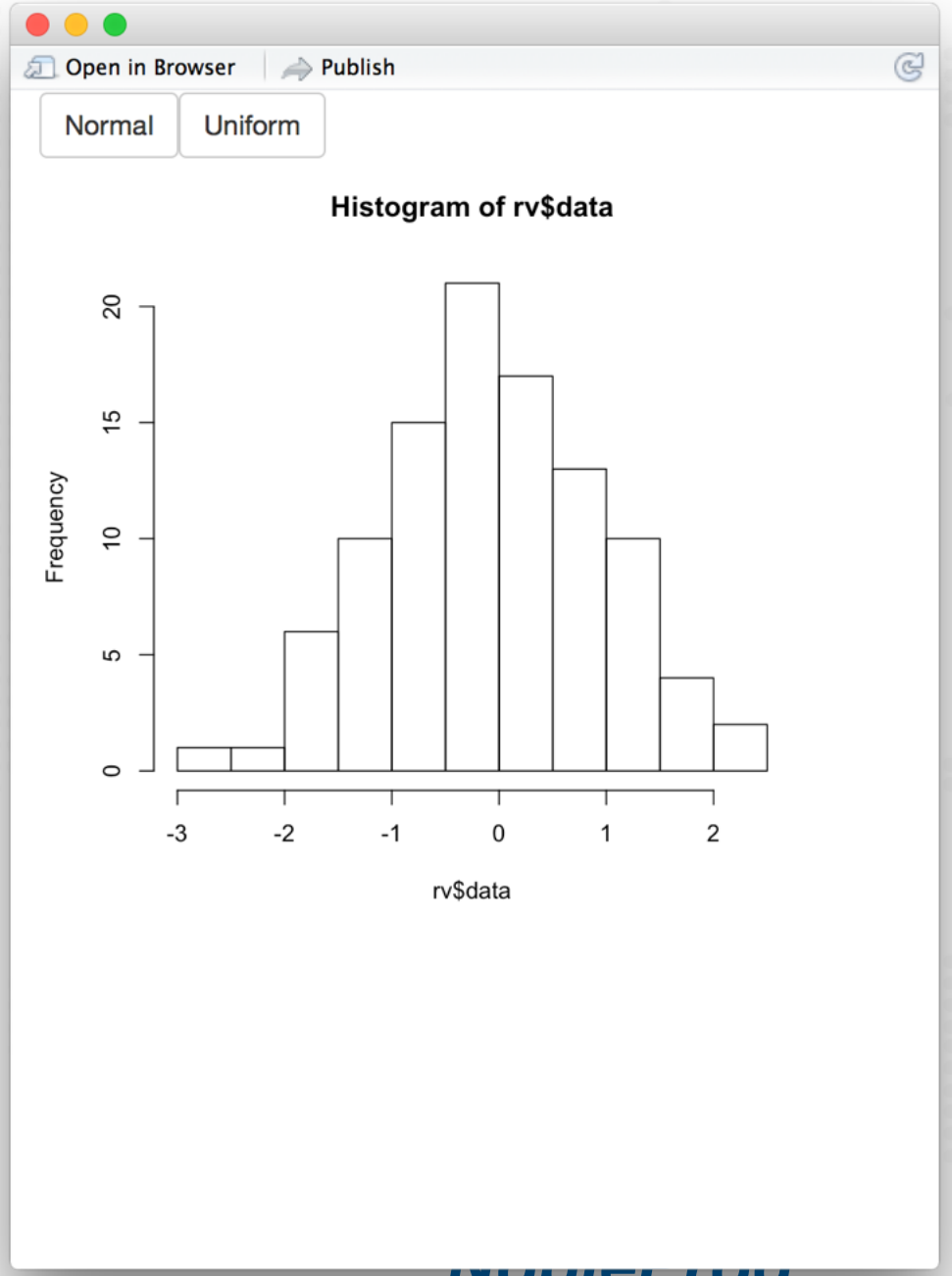


input$norm

rv$data
runif(100)

input$unif

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

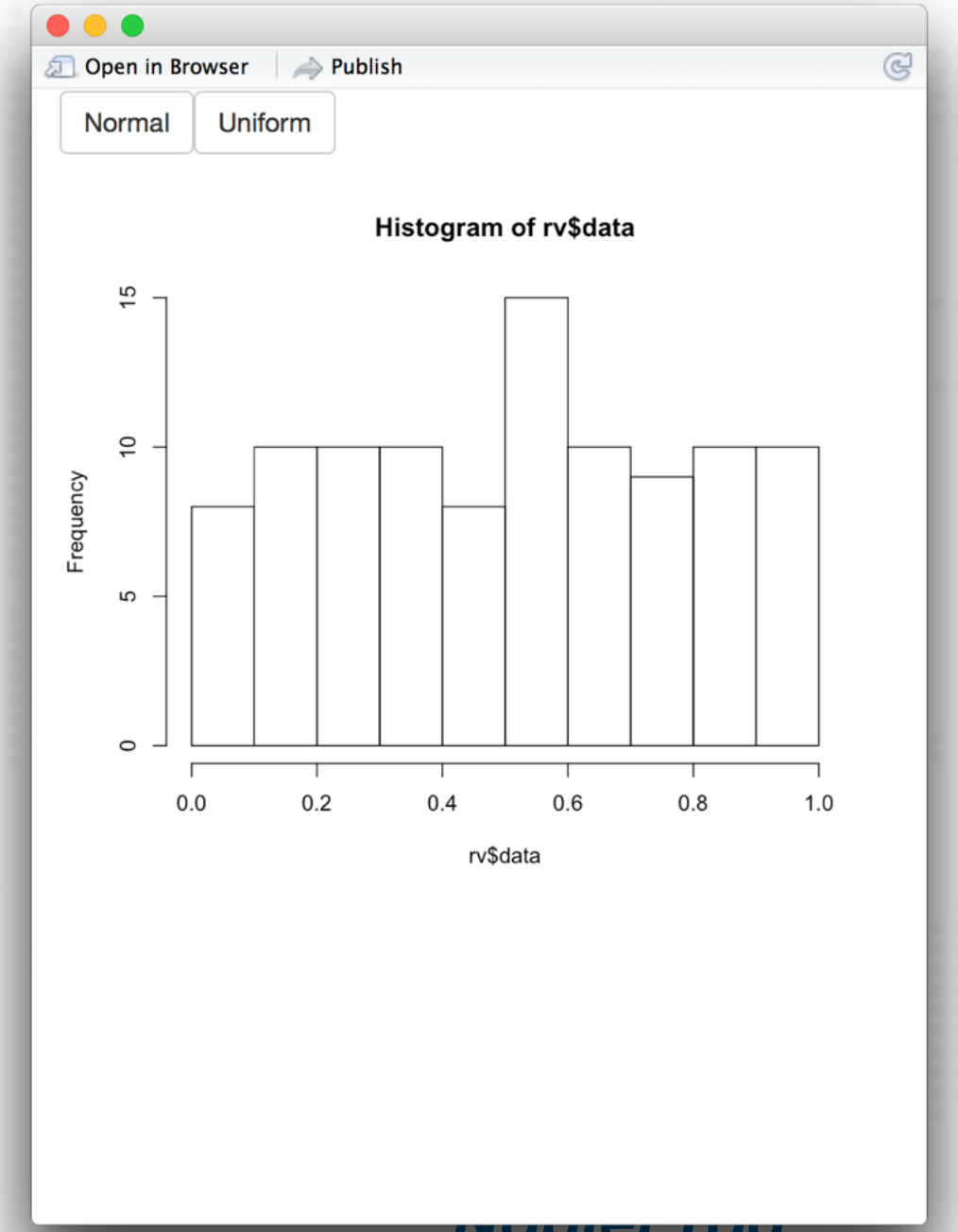


input$norm

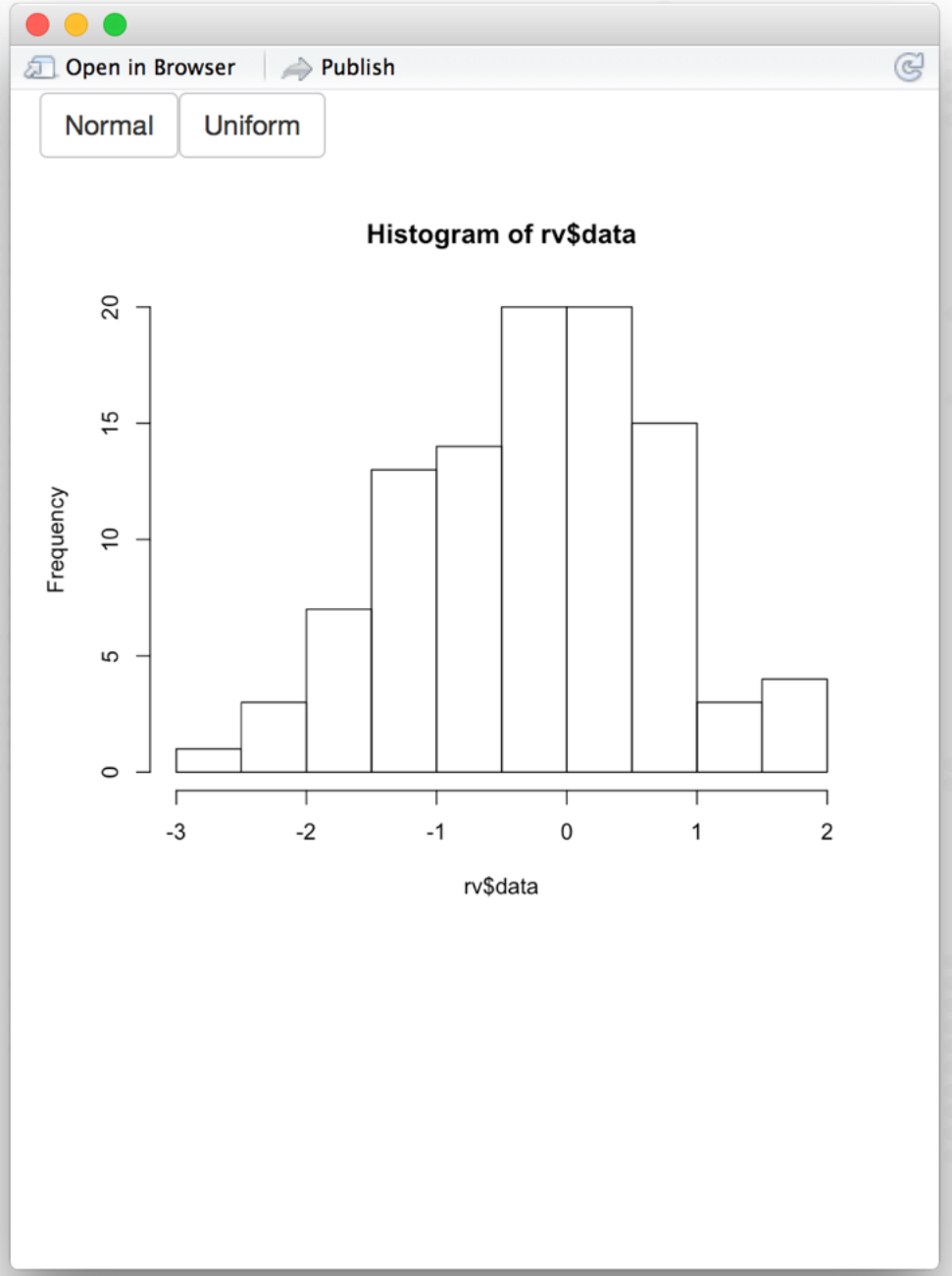
rv$data
runif(100)

input$unif

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



```
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)
```



Recap: reactiveValues()

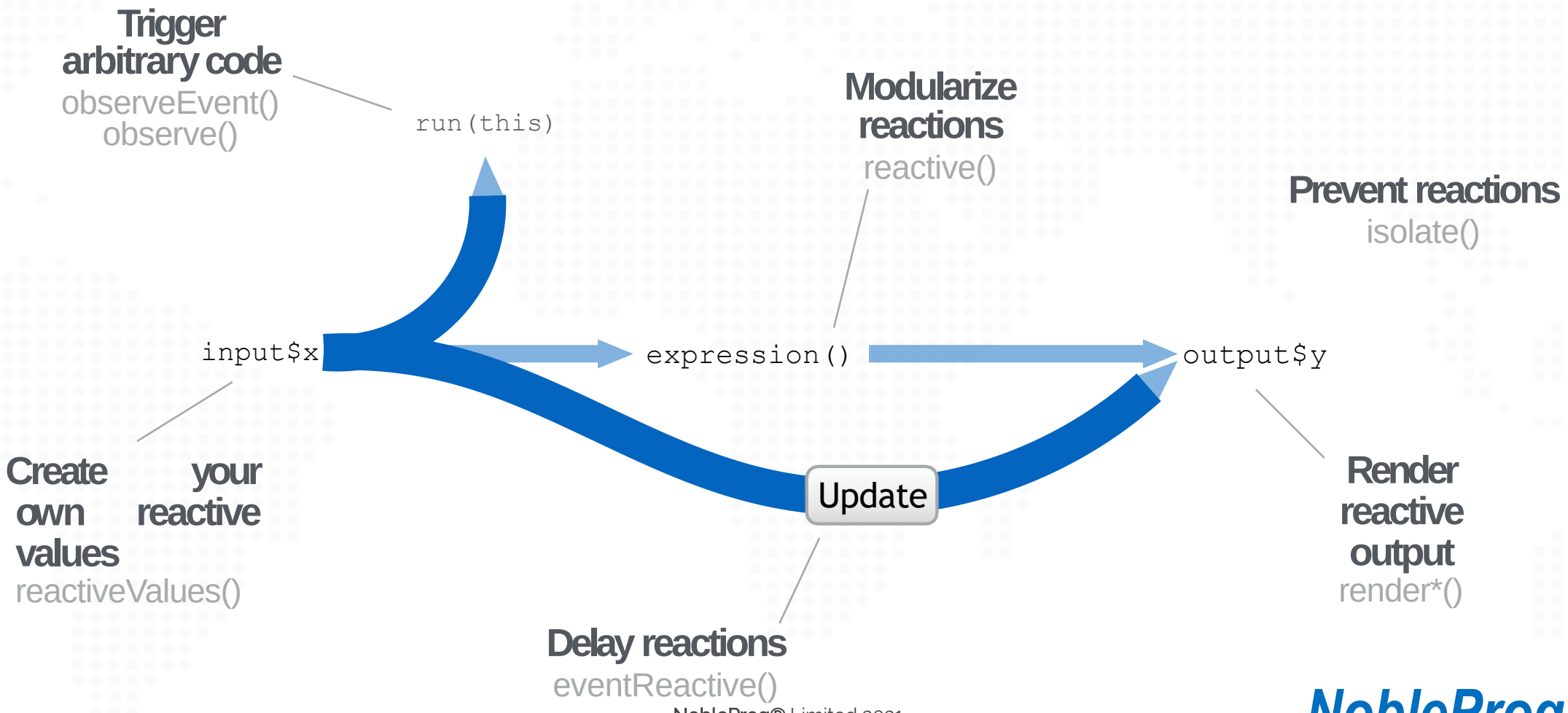
reactiveValues() creates a list of **reactive values**

```
rv$data <-
```

You can manipulate these values (usually with observeEvent())

NobleProg

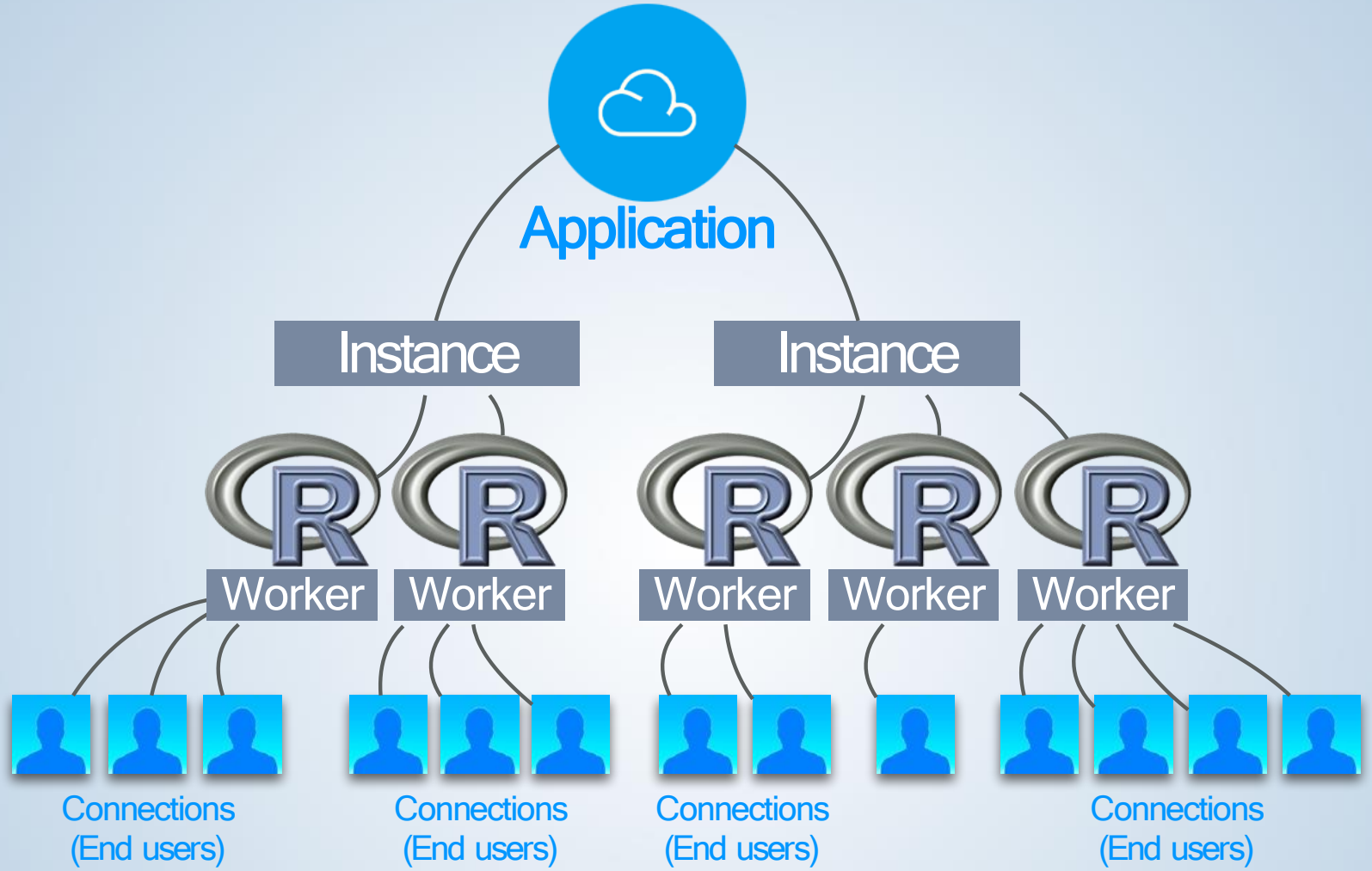
You now how to



NobleProg

Performance Tips

NobleProg


Application
Instance
Instance
Worker
Worker
Worker
Worker
Worker
Connections
(End users)
Connections
(End users)
Connections
(End users)
Connections
(End users)

RStudio

Reduce repetition

Place code where it will be re-run as little as necessary

```
library(shiny)

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

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

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

Code outside the server function will be run once per R session (worker)

Code inside the server function will be run once per end user (connection)

Code inside a reactive function will be run once per reaction (e.g. many times)

NobleProg

Dynamic UI

```
ui <- fluidPage(
  numericInput("n", "Simulations", 10),
  actionButton("simulate", "Simulate")
)

server <- function(input, output, session) {
  observeEvent(input$n, {
    label <- paste0("Simulate ", input$n, " times")
    updateActionButton(inputId = "simulate", label = label)
  })
}
```


NobleProg

Exercise

- Create a small application that allows the user to:
  - Select age range and gender
  - User clicks a „Submit“ button.
  - Returns the corresponding injuries from „injuries.tsv“ file.

- Add a file upload with a dynamic UI:
  - User uploads a file („injuries.tsv“).
  - Shiny reads the values of „gender“ and „age“ and generates the UI.
  - „Submit“ button + display values as above.