RStudio

250 Northern Ave, Boston, MA 02210
Phone: 844-448-1212

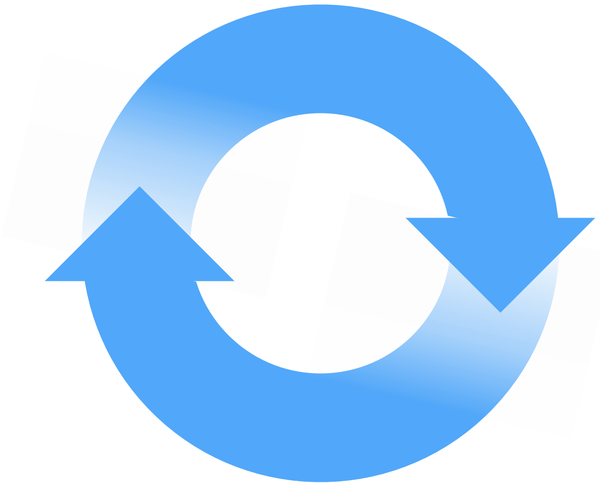
Email: info@rstudio.com
Web: http://www.rstudio.com

All Training materials are provided "as is" and without warranty and RStudio disclaims any and all express and implied warranties including without limitation the implied warranties of title, fitness for a particular purpose, merchantability and noninfringement.

RStudio
250 Northern Ave, Boston, MA 02210
Phone: 844-448-1212
Email: info@rstudio.com
Web: http://www.rstudio.com

How to start with Shiny, Part 2

How to customize reactions



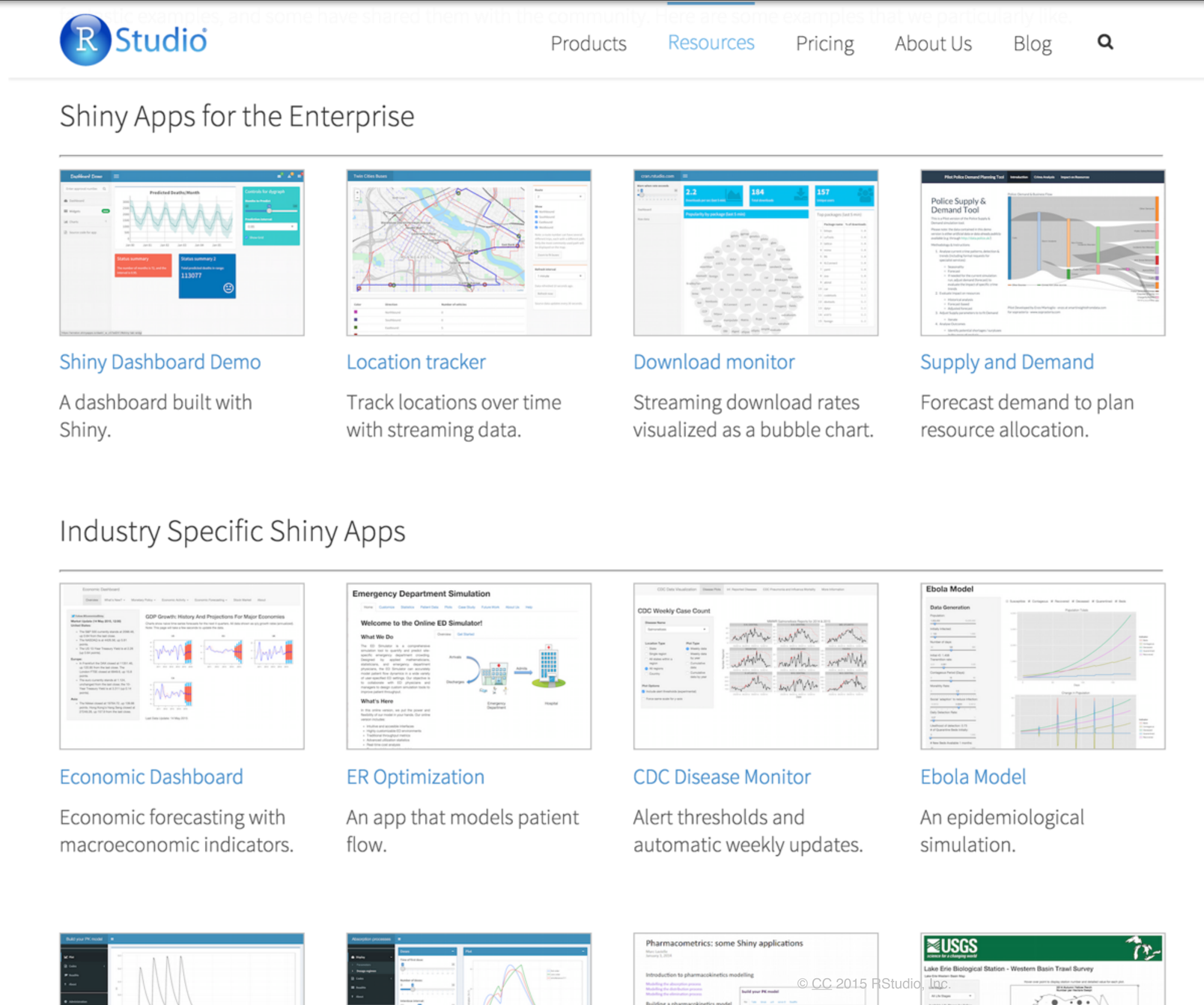
Garrett Grolemund
Data Scientist and Master Instructor
May 2015
Email: garrett@rstudio.com
Twitter: @StatGarrett

Code and slides at:

bit.ly/shiny-quickstart-2

Shiny Showcase

www.rstudio.com/products/shiny/shiny-user-showcase/



How to start with Shiny

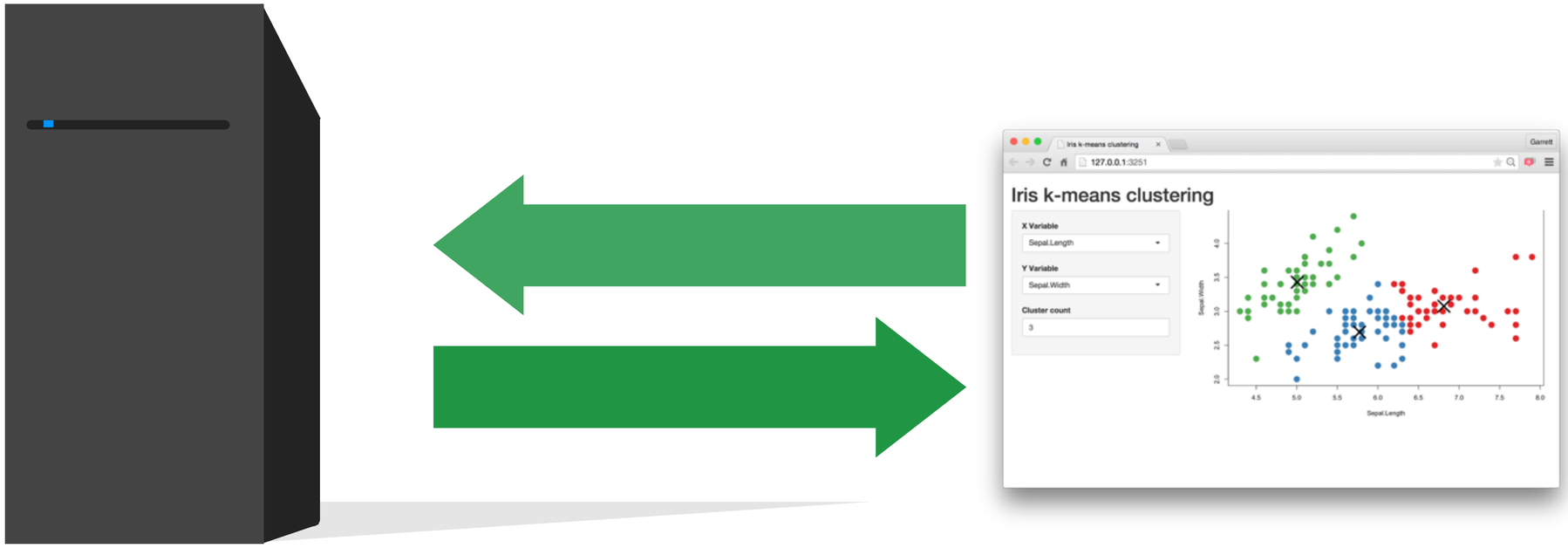
1. How to build a Shiny app (www.rstudio.com/resources/webinars/)

2. How to customize reactions (Today)

3. How to customize appearance (June 3)

The story so far

Every Shiny app is maintained by a computer running R



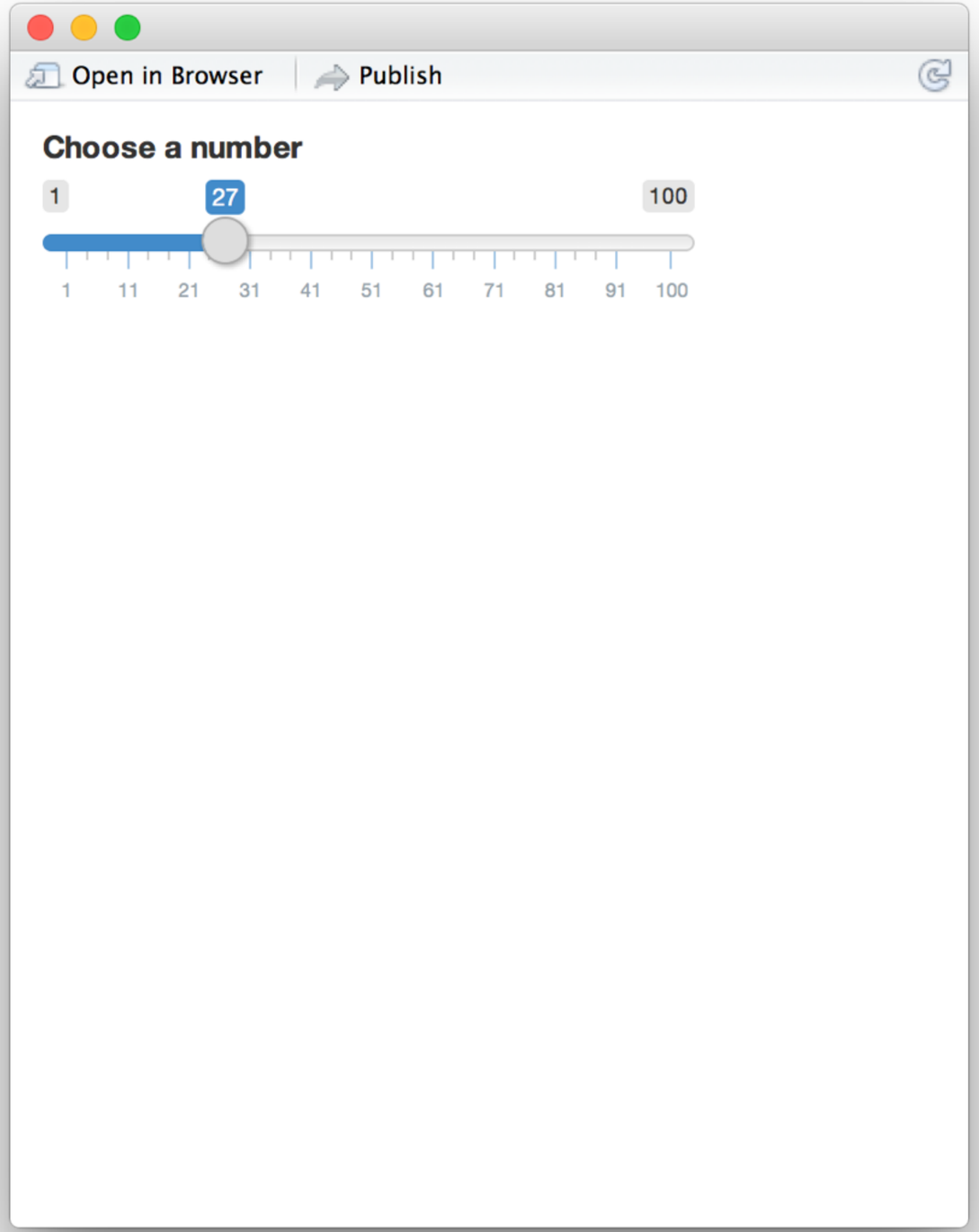
App template

The shortest viable shiny app

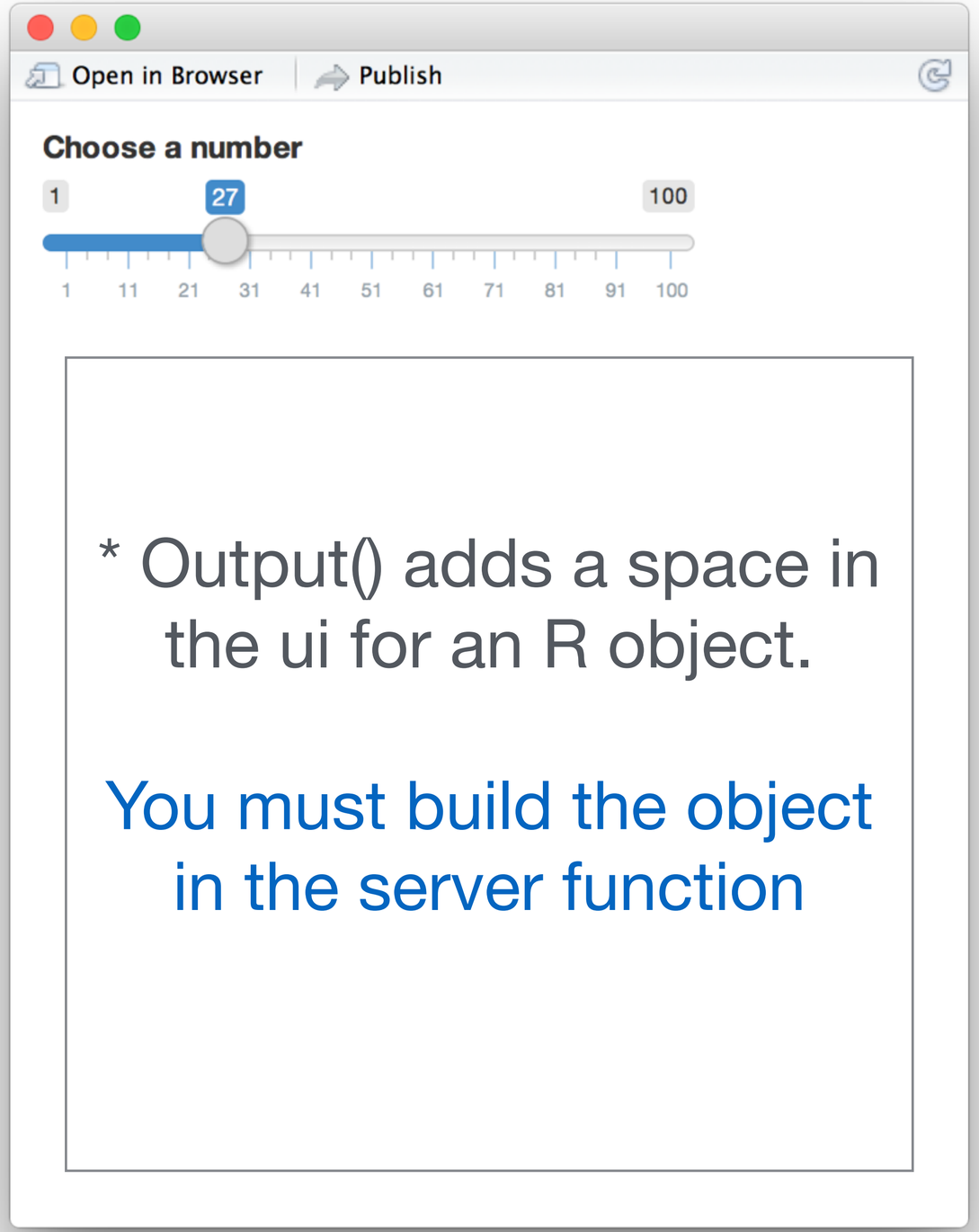

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

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

```
library(shiny)

ui <- fluidPage(




)

server <- function(input, output) {



}

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

```
library(shiny)

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

)

server <- function(input, output) {



}

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) {




}

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

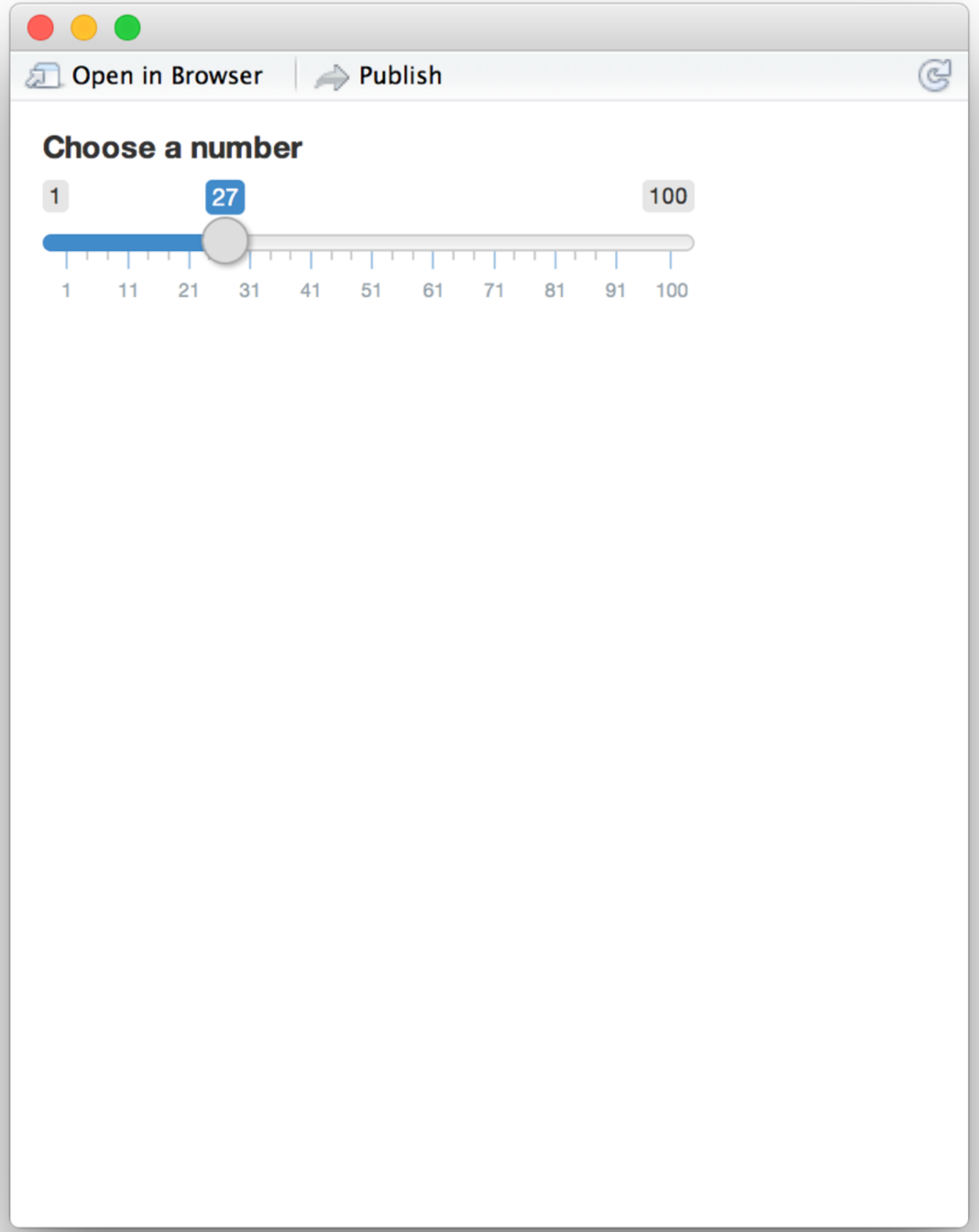


* Output() adds a space in the ui for an R object.

You must build the object in the server function

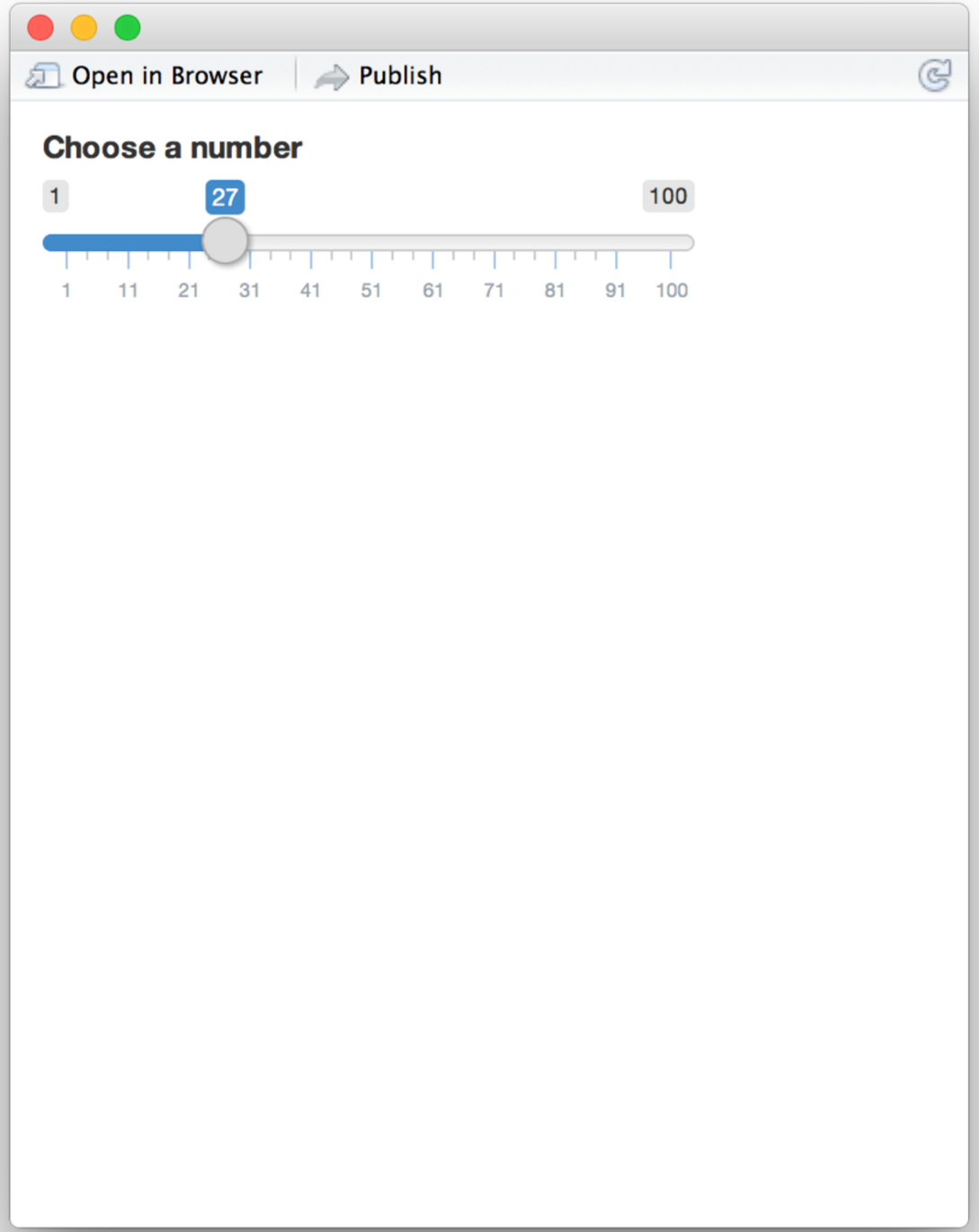
```
library(shiny)

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

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


}

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

1



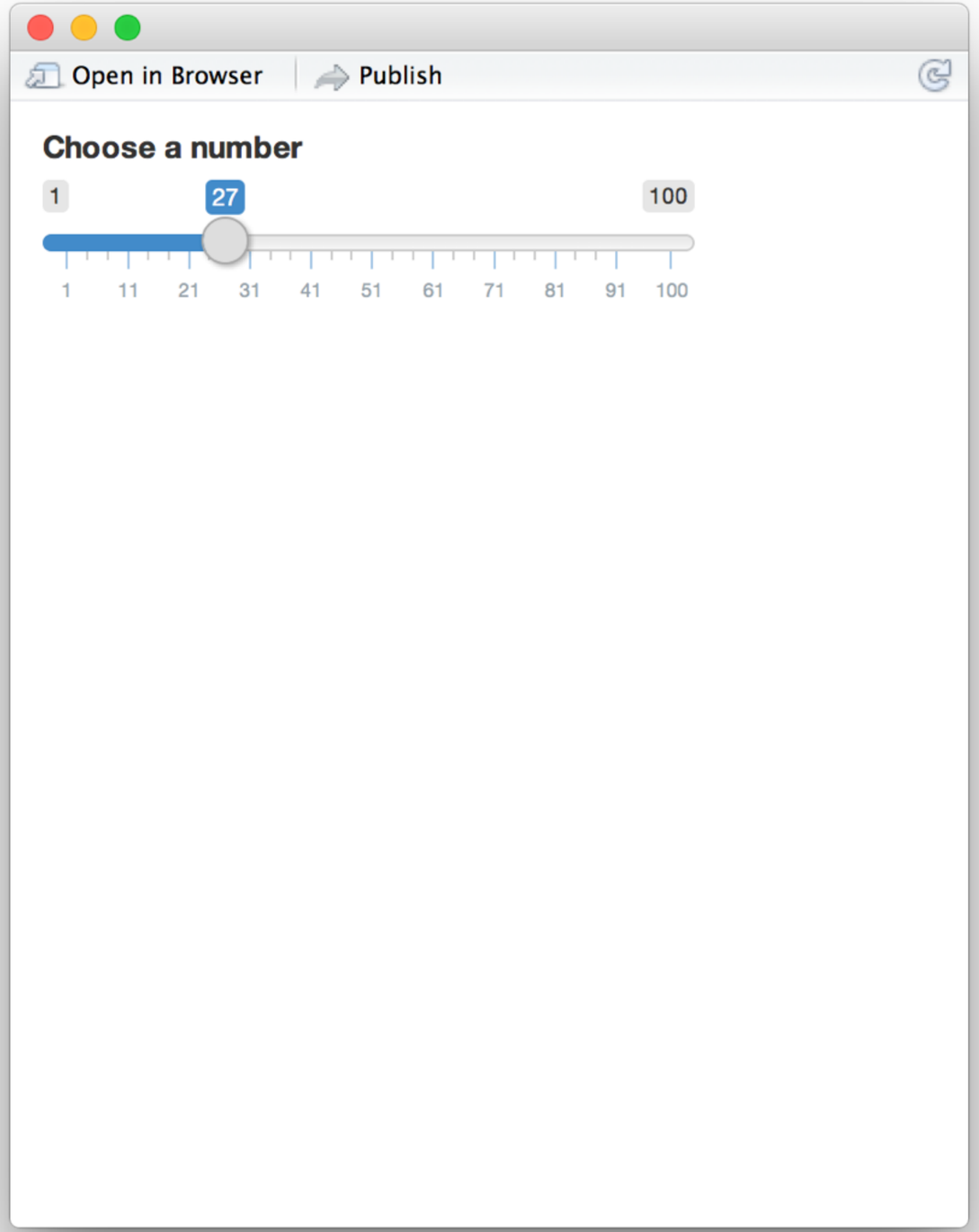
```
library(shiny)

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

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

  })
}

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

2

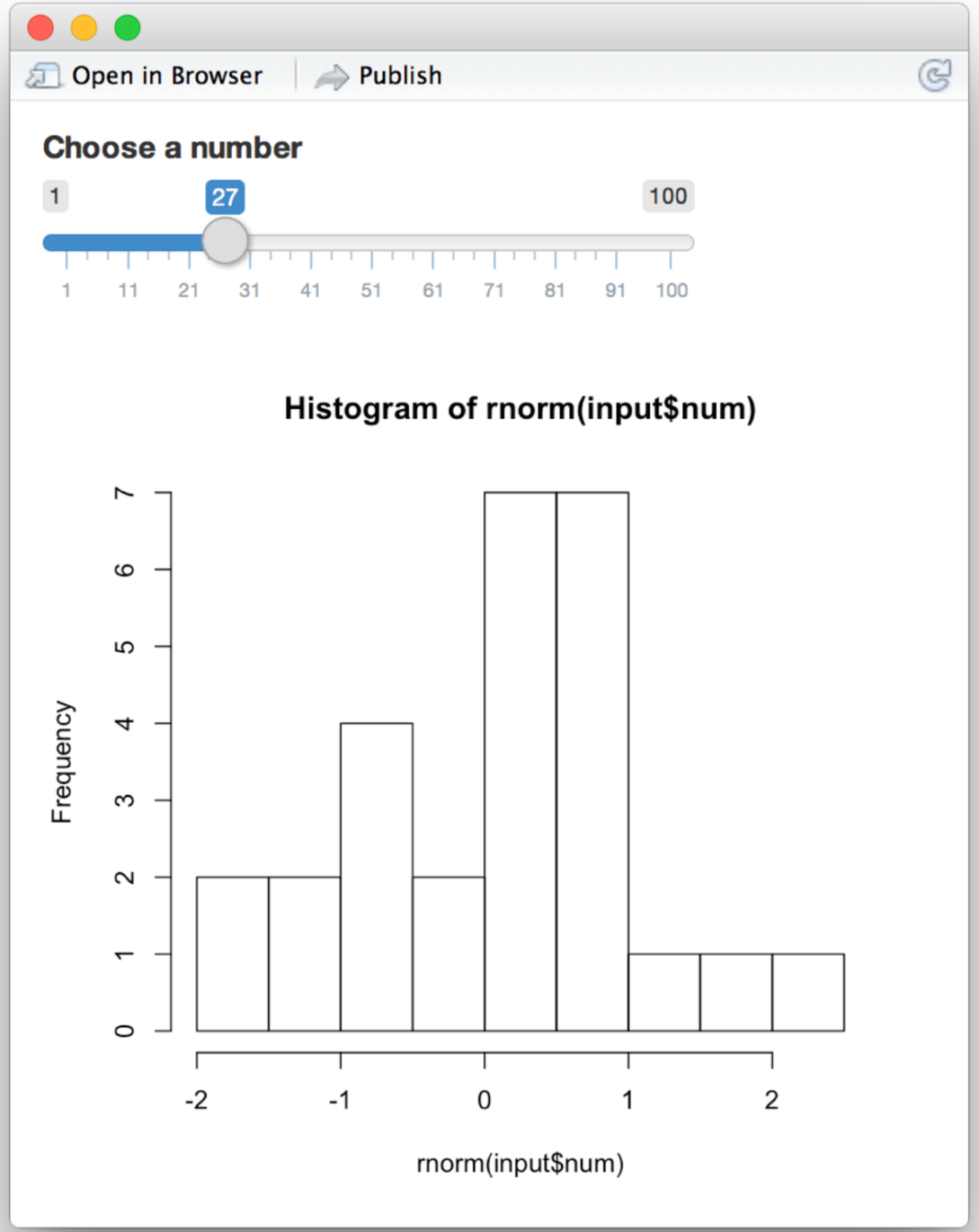


```
library(shiny)

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)
```

3



```
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)
```



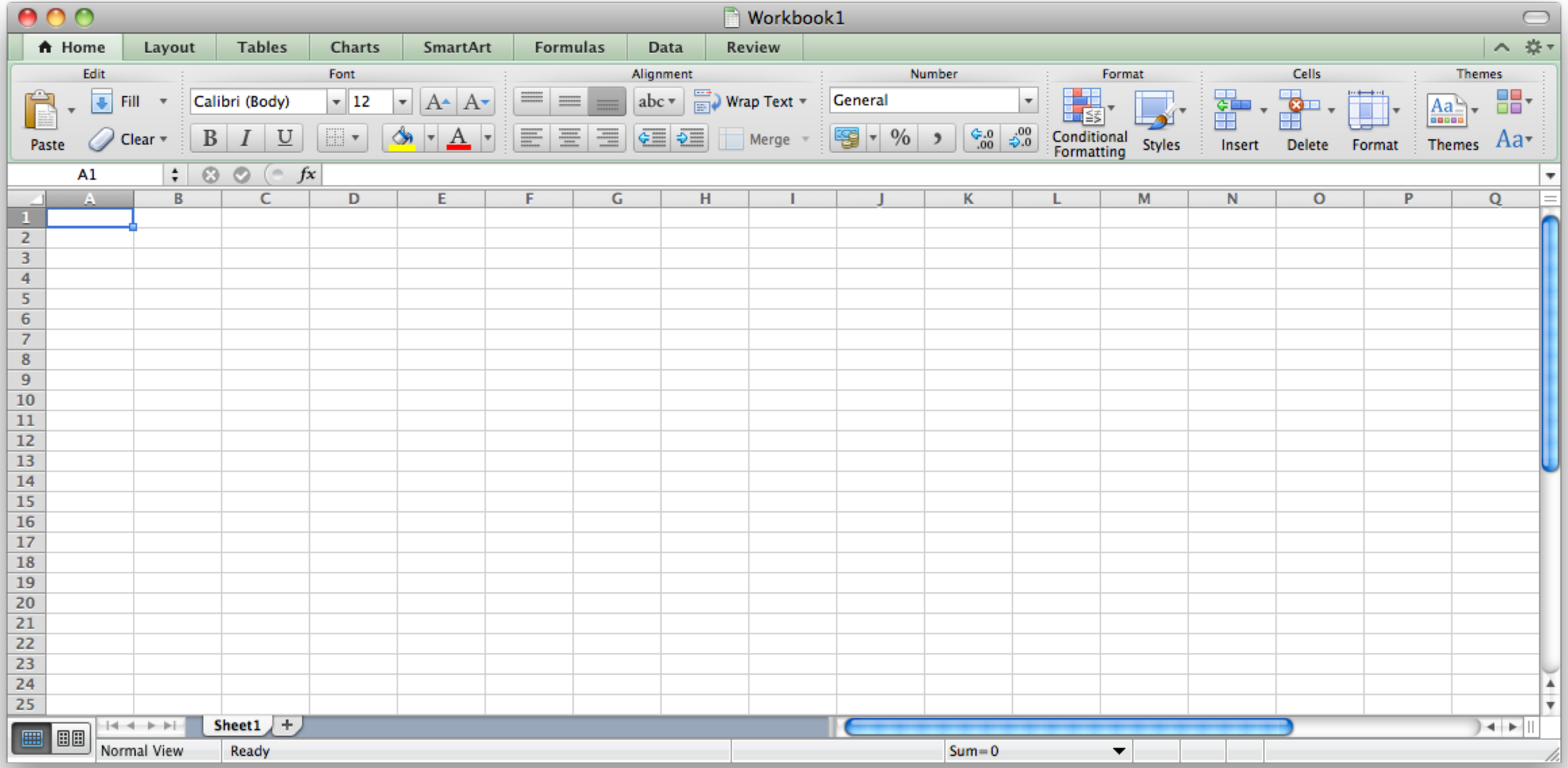
Sharing apps

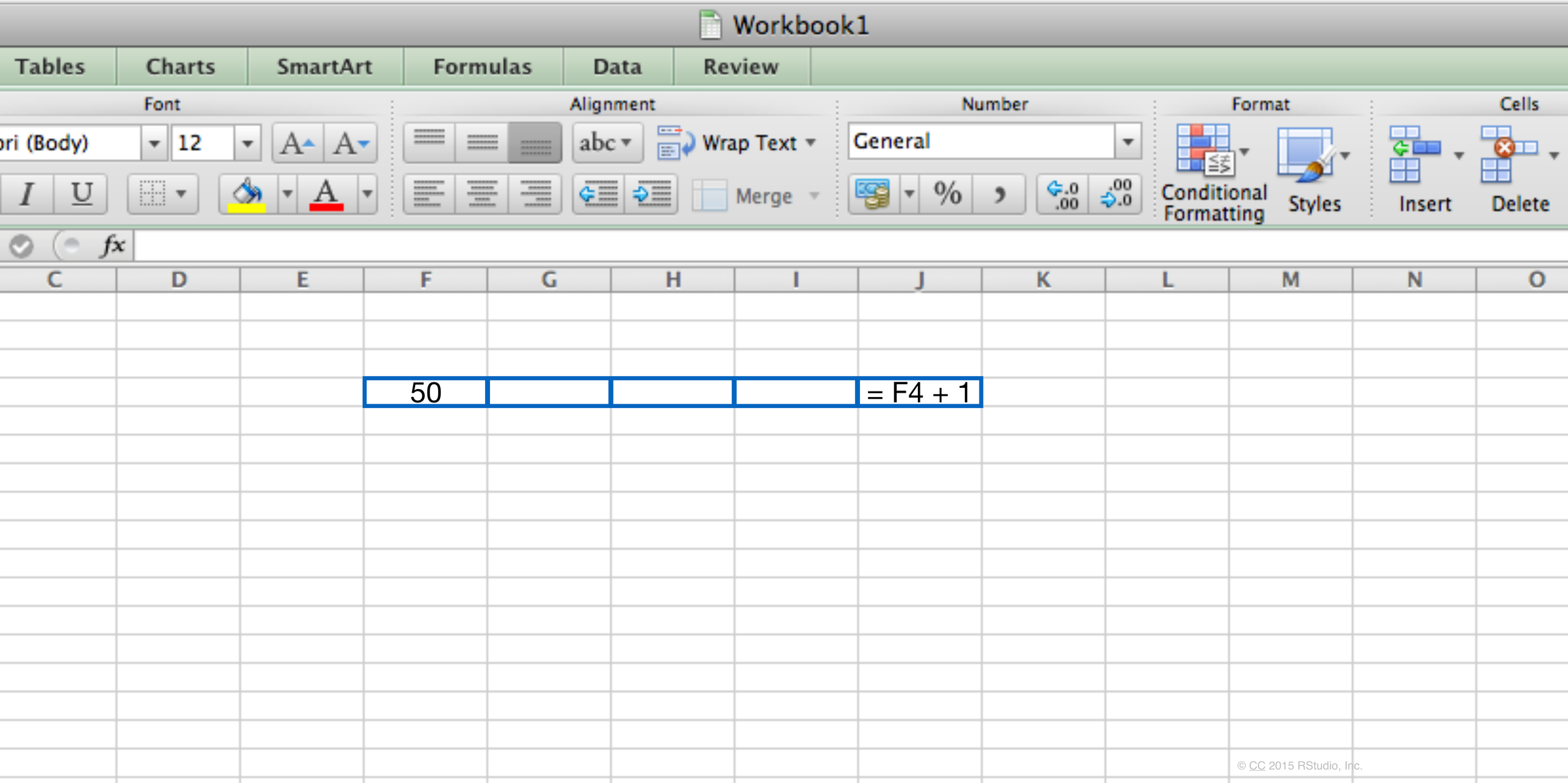
Shiny Server (Pro)

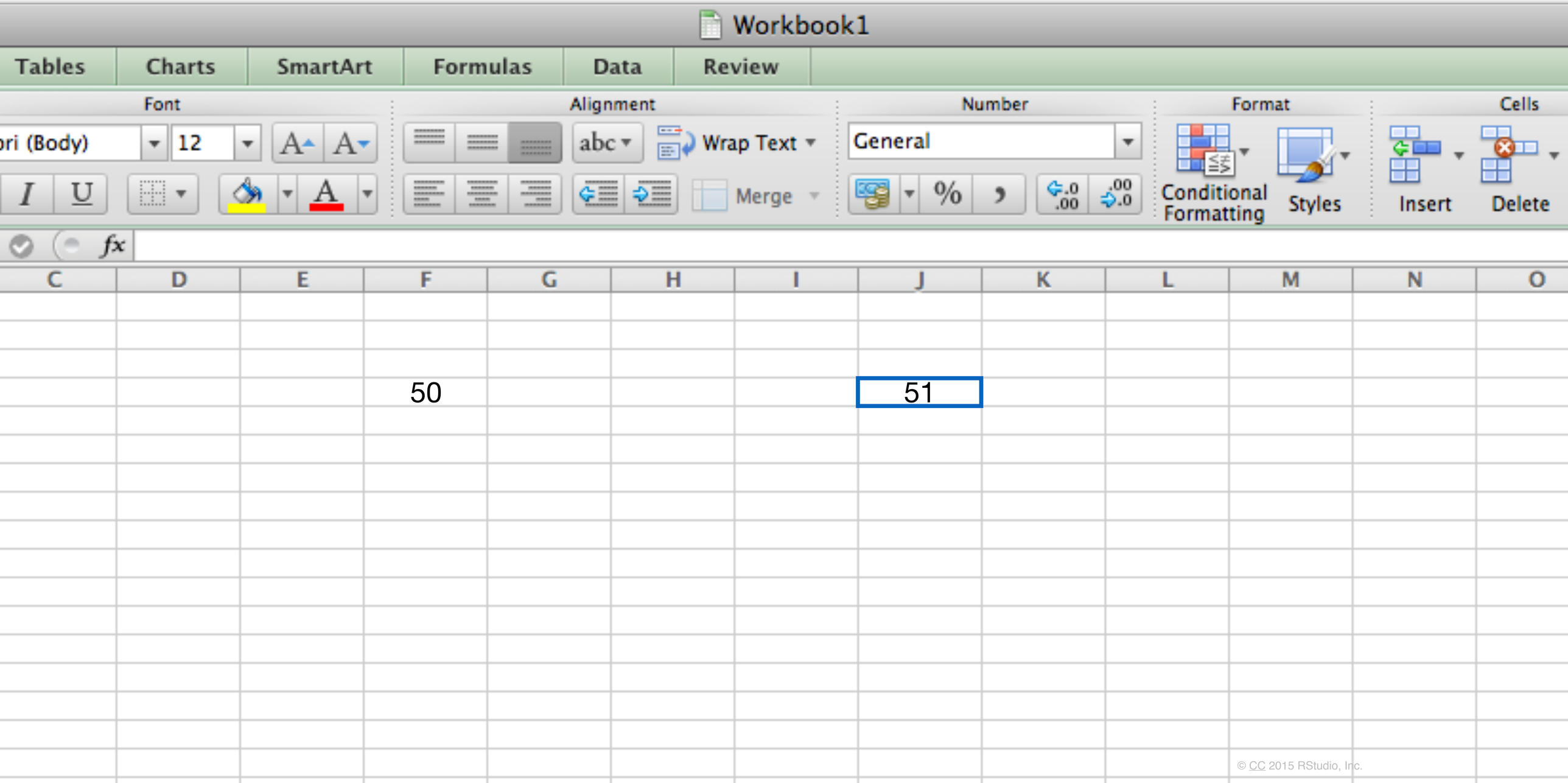
http://www.rstudio.com/resources/webinars/

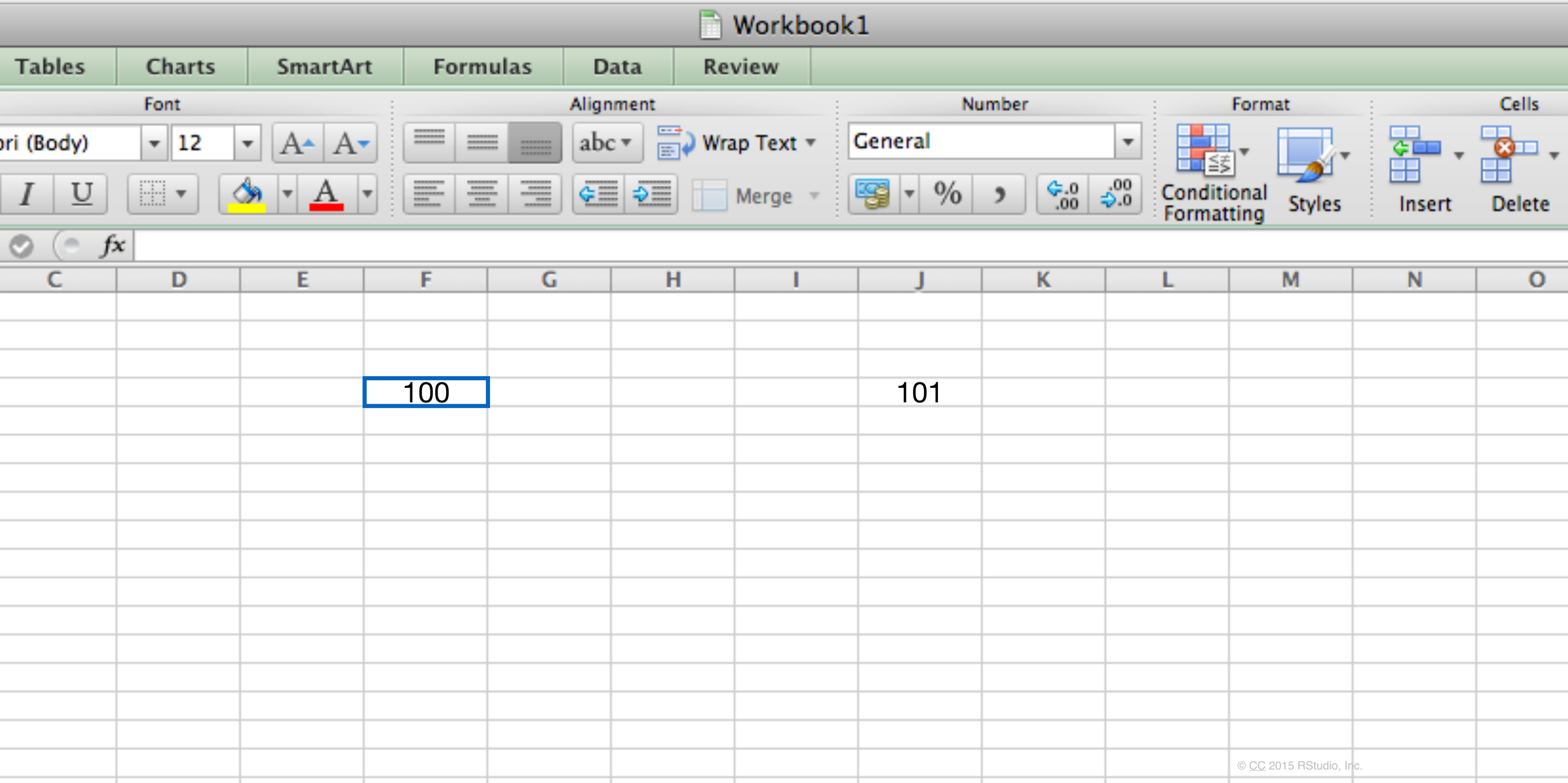
What is Reactivity?

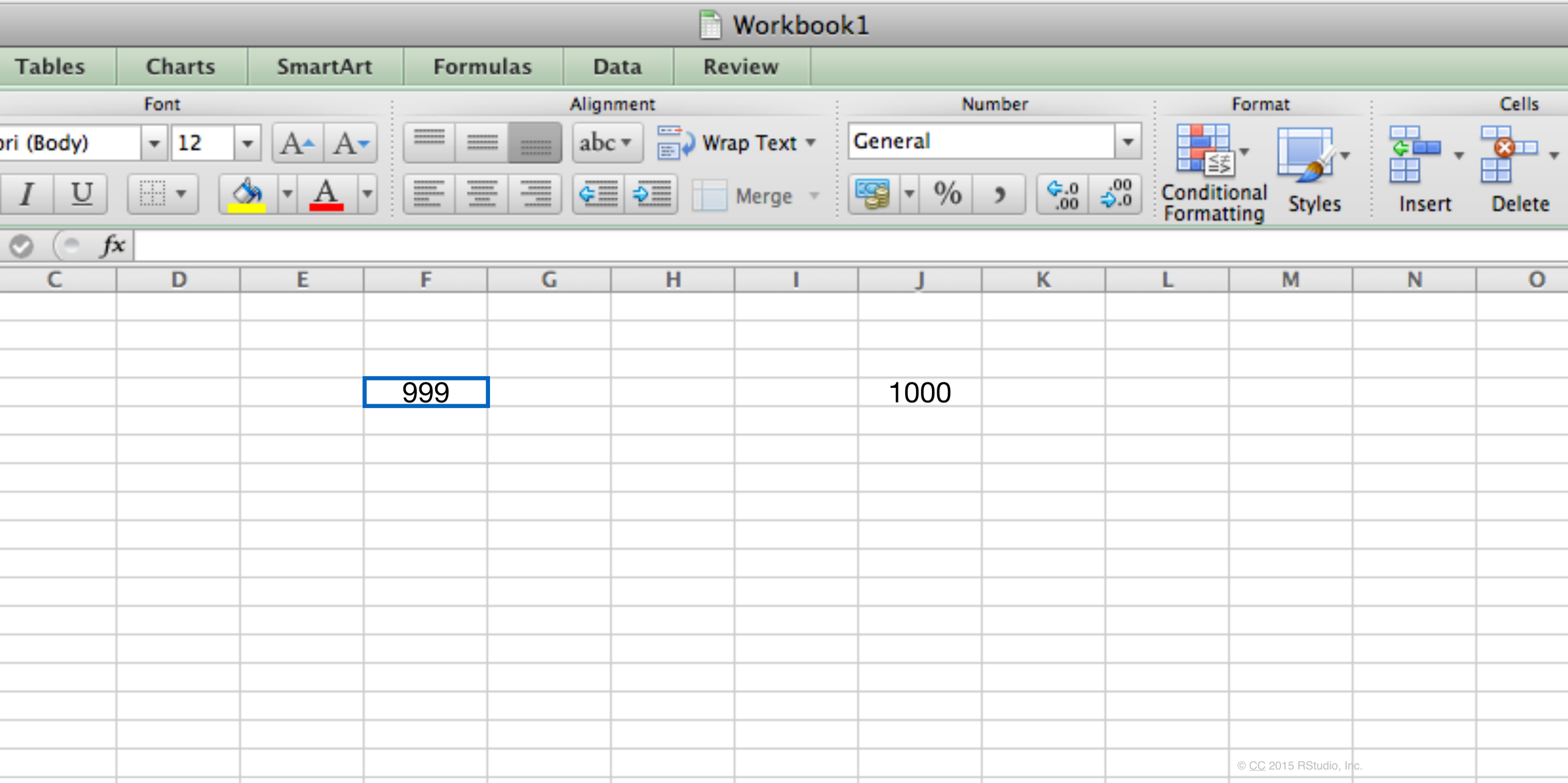
Think Excel.

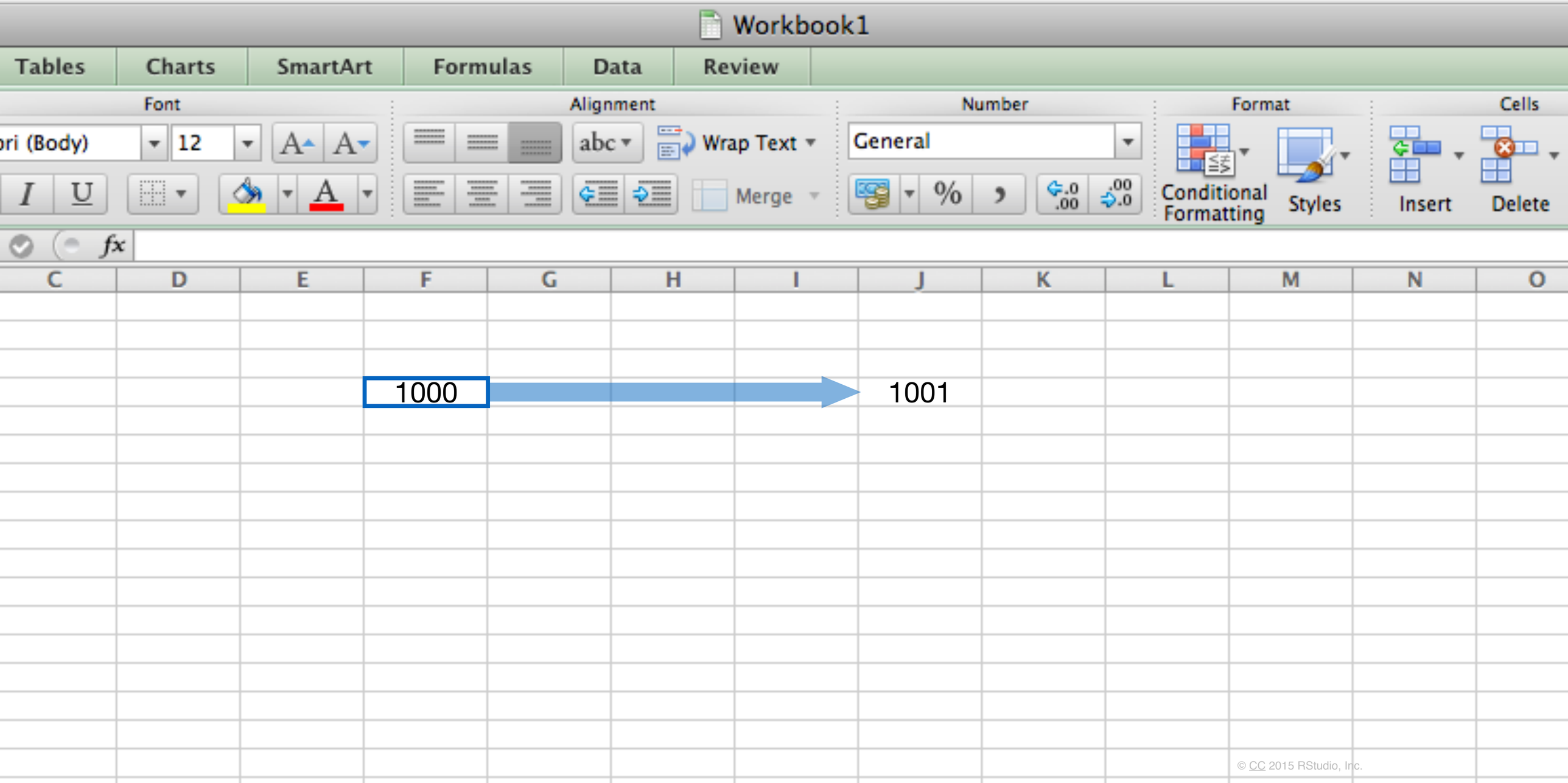


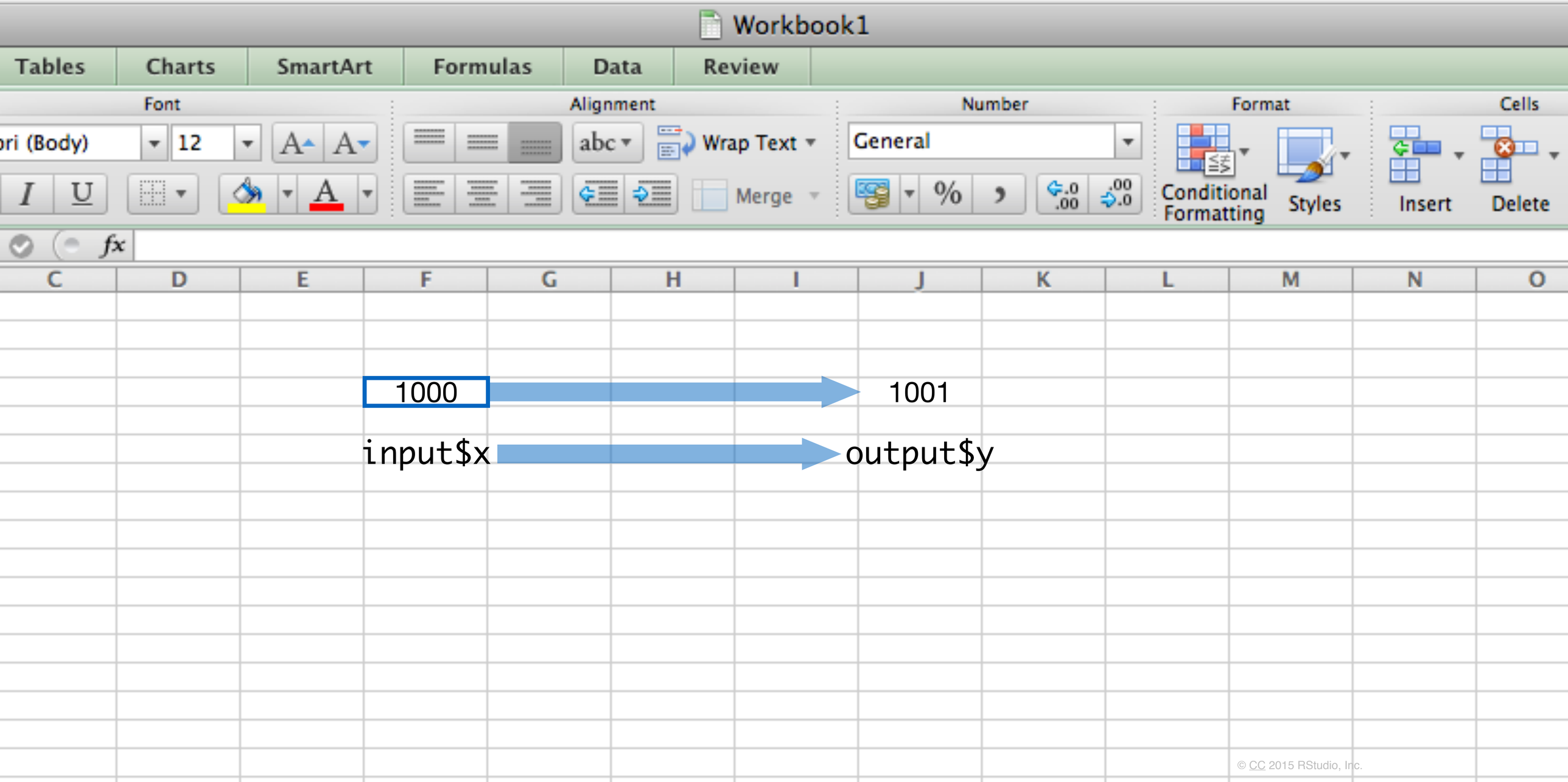


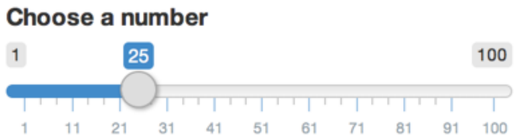


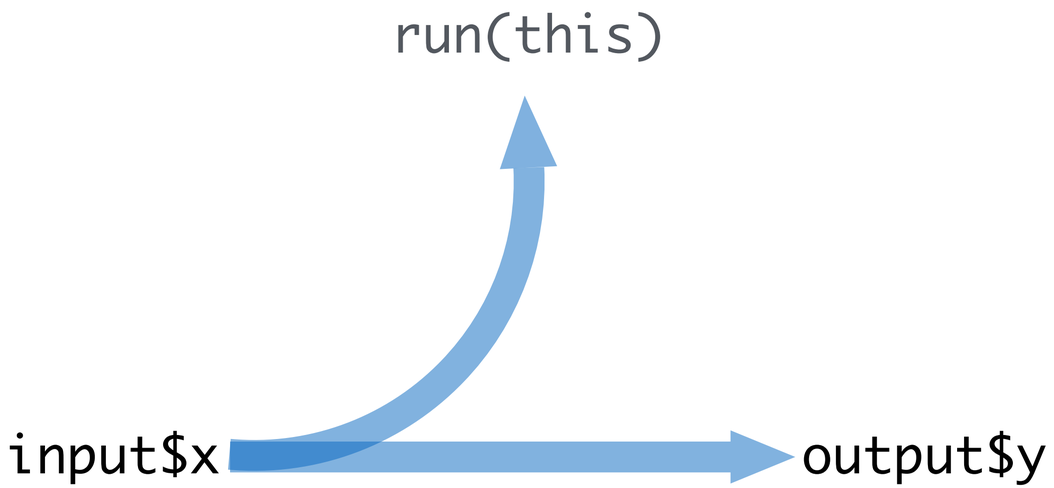


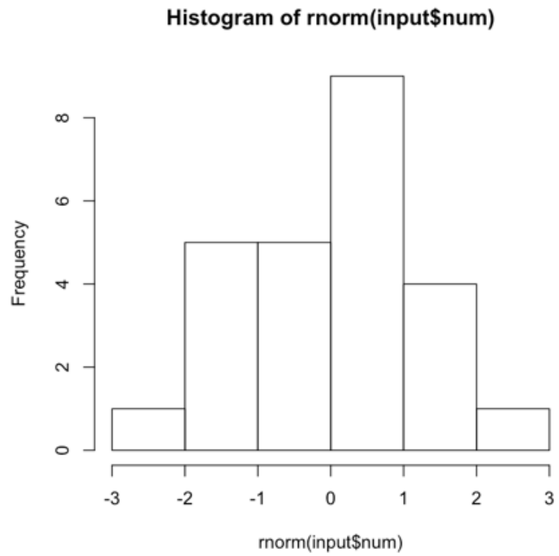


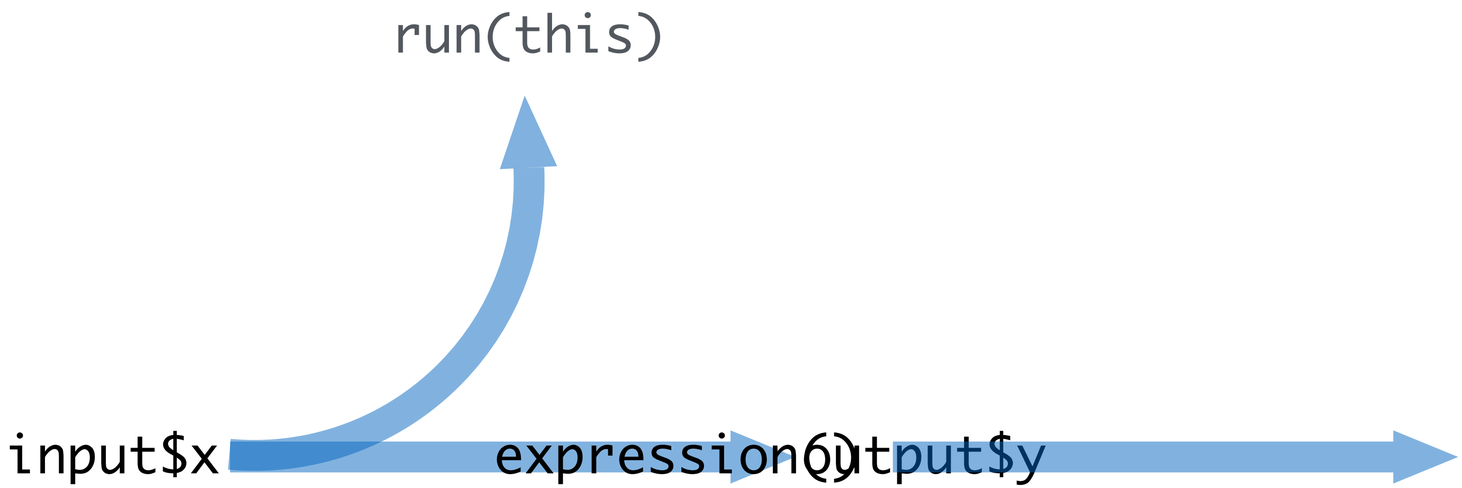


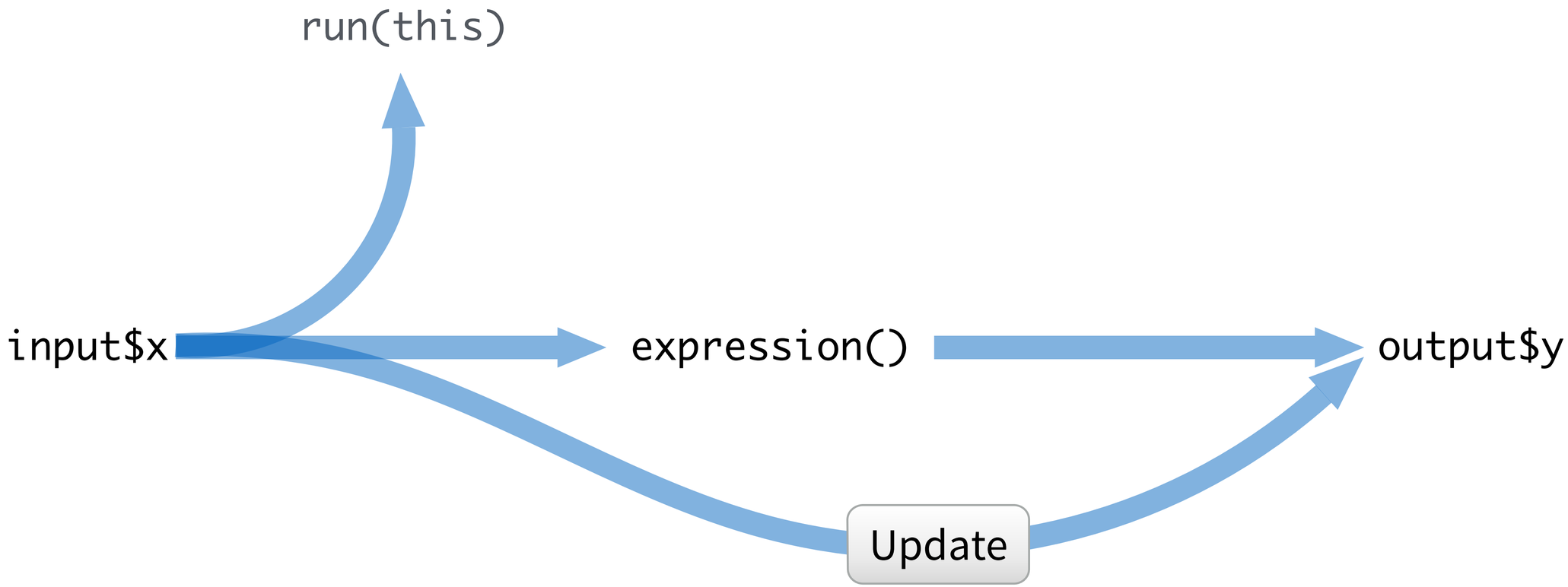






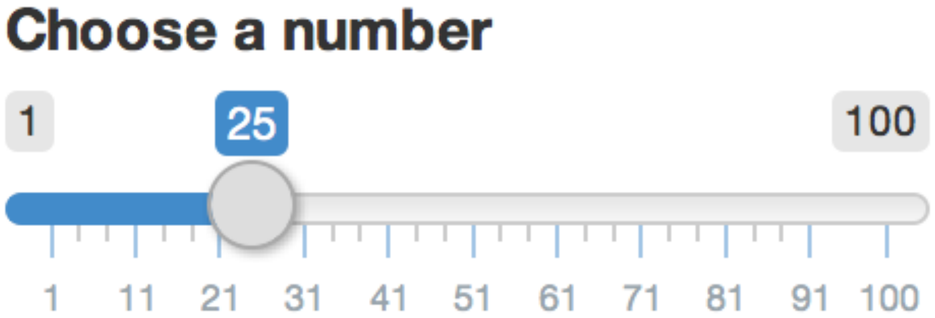






Begin with
reactive values

Syntax

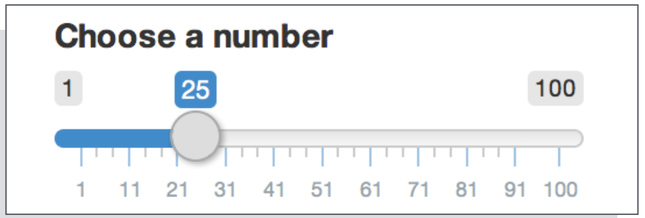



```
sliderInput(inputId = "num", label = "Choose a number", …)
```

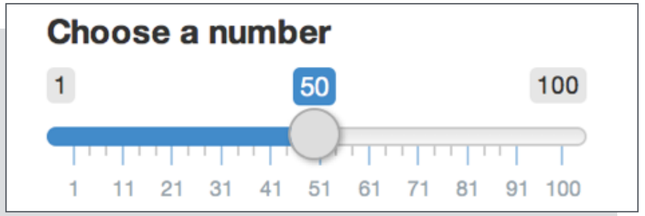
this input will provide a value saved as **input$num**

Input values

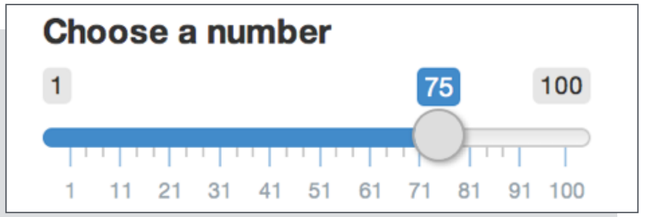
The input value changes whenever a user changes the input.




```
input$num = 25
```




```
input$num = 50
```



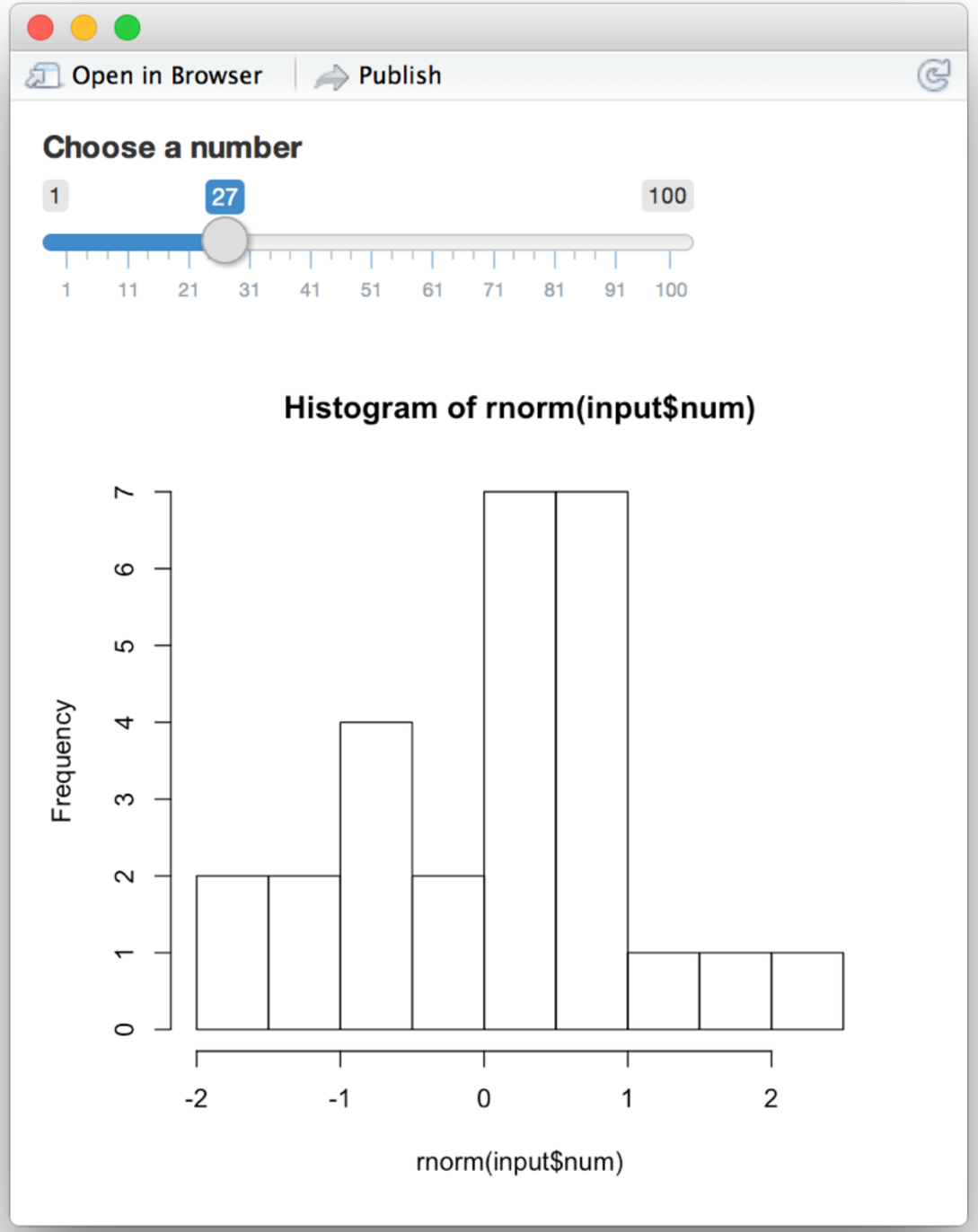

```
input$num = 75
```

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

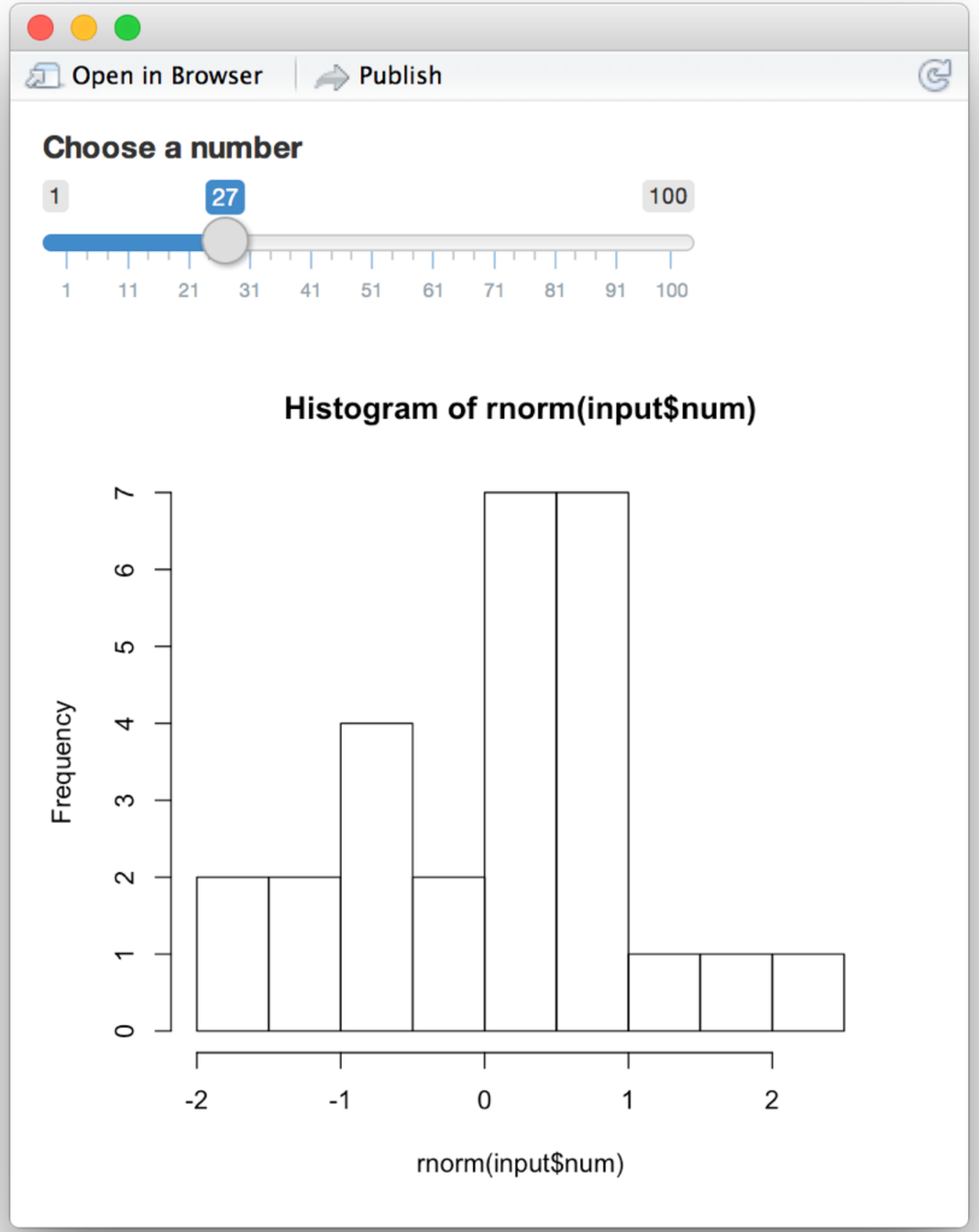
```
renderPlot({ hist(rnorm(100, input$num)) })
```

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

```
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)
```

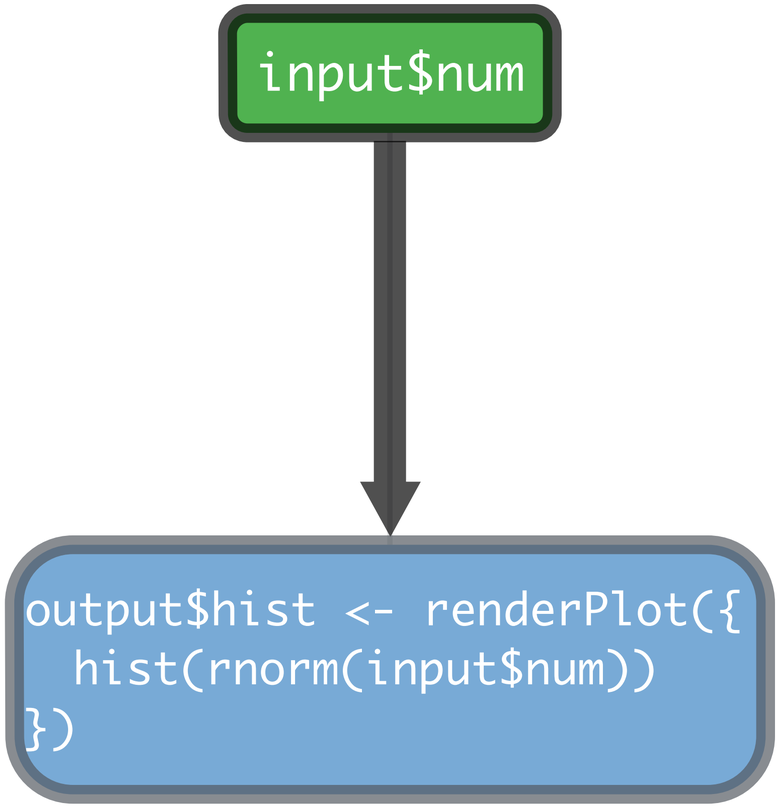


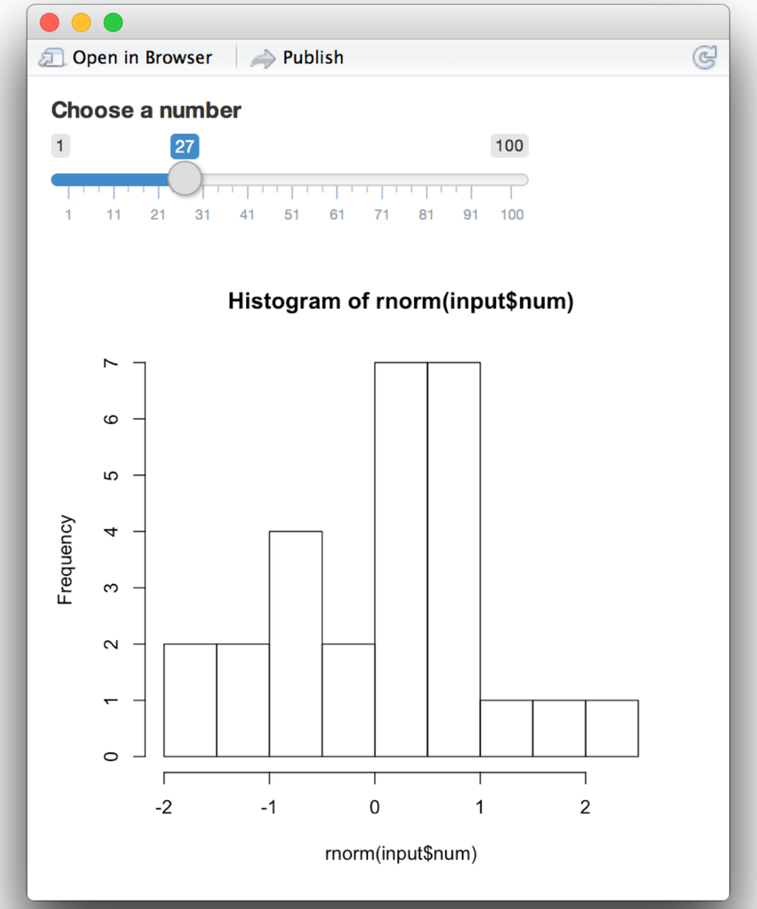
```
library(shiny)

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

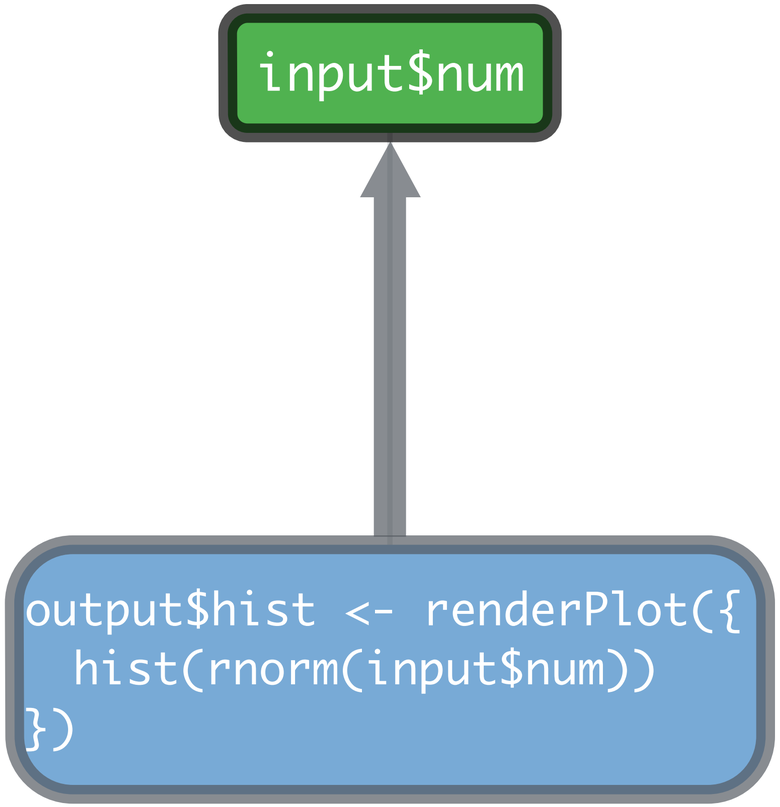


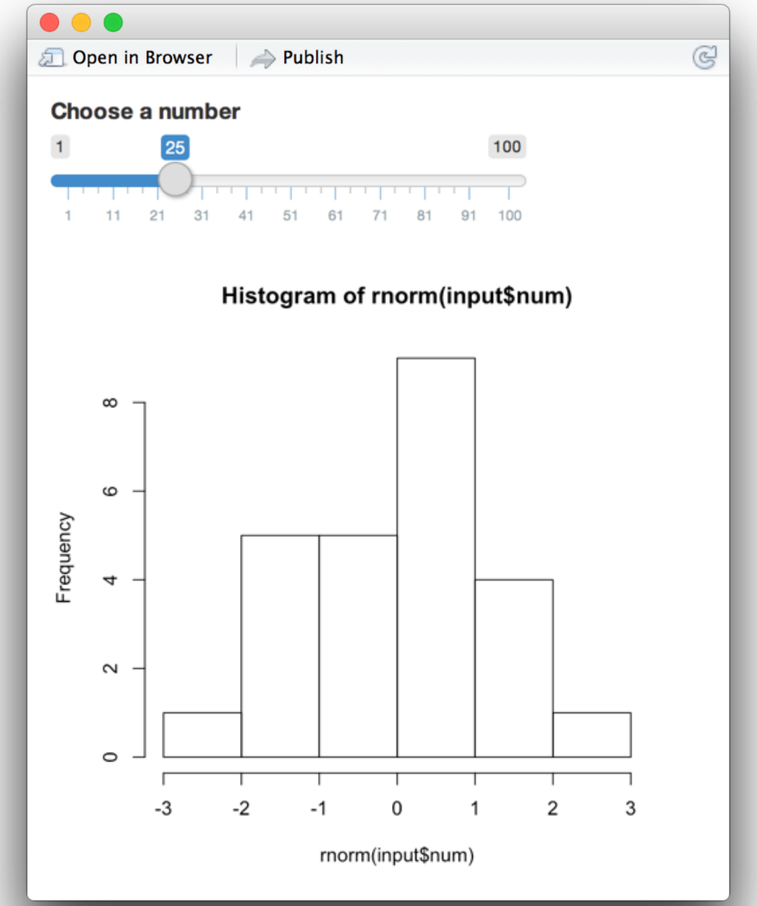


Think of reactivity in R as a two step process

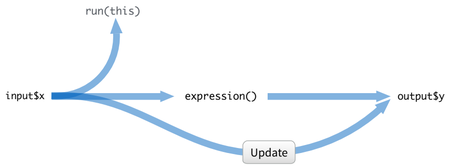
1 **Reactive values notify**
the functions that use them
when they become invalid

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





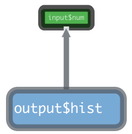
Recap: Reactive values



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.**

Reactive toolkit

(7 indispensible functions)

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*()

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

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

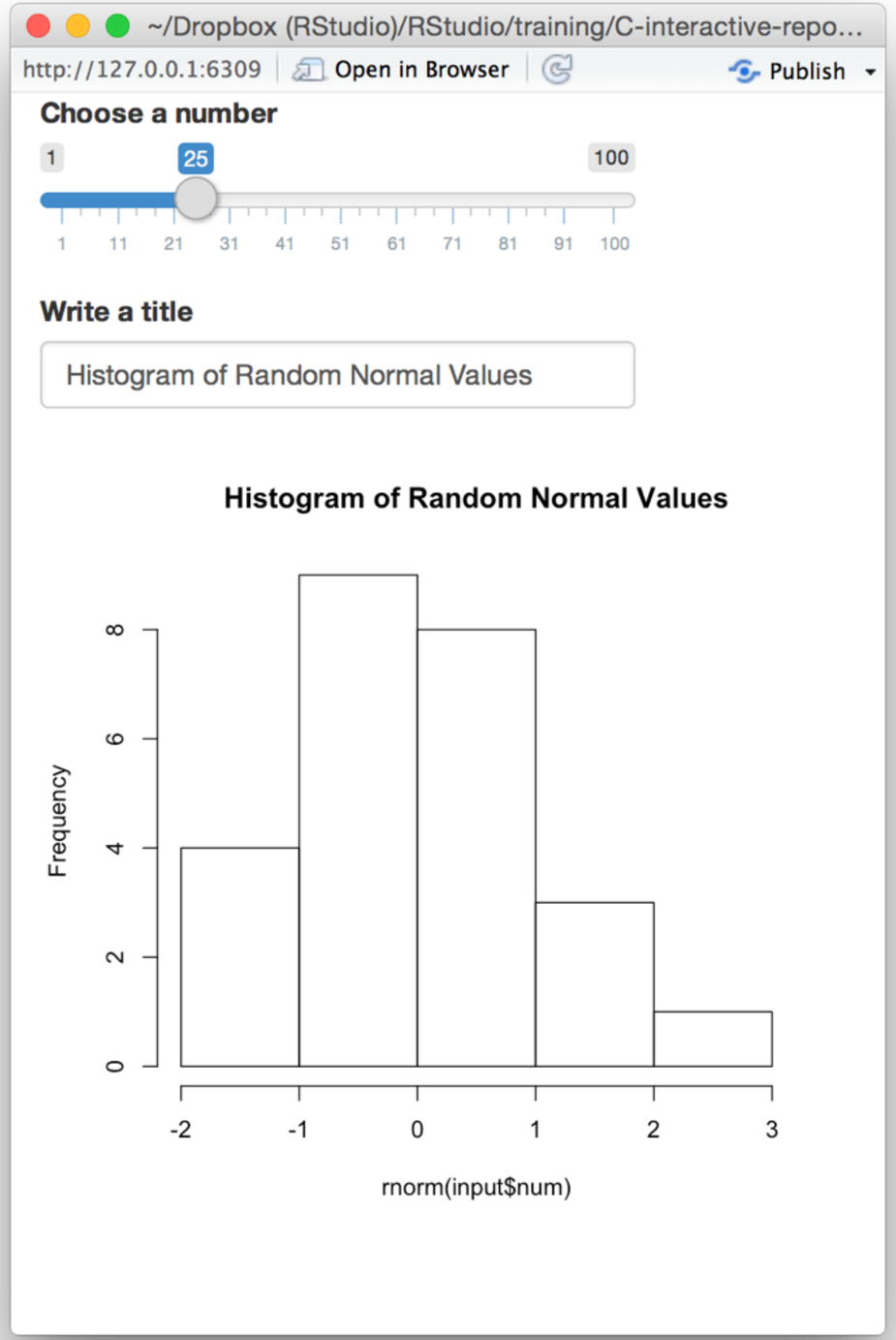
render*()

Builds reactive output to display in UI

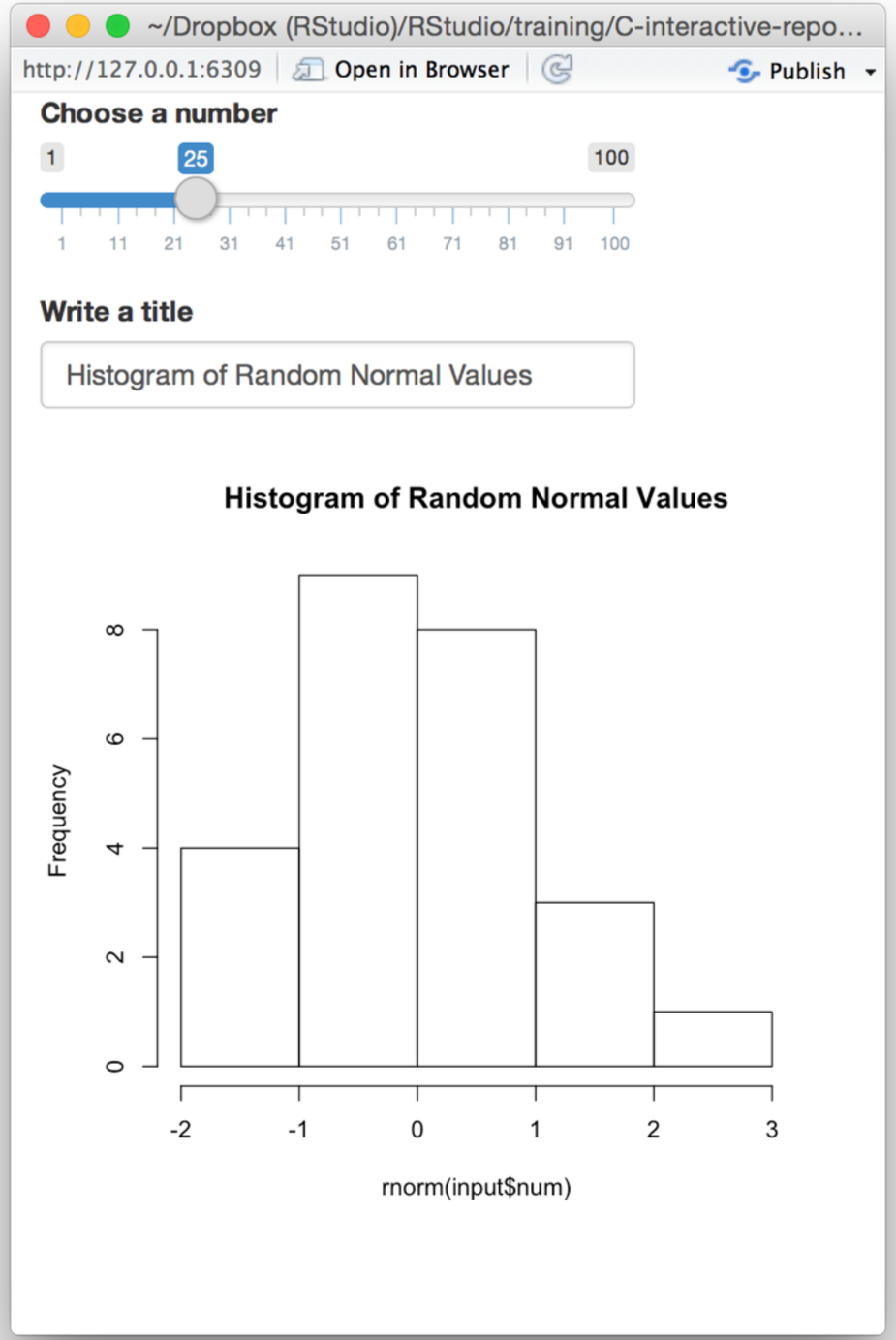
```
renderPlot(  { hist(rnorm(input$num)) })
```

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)
```



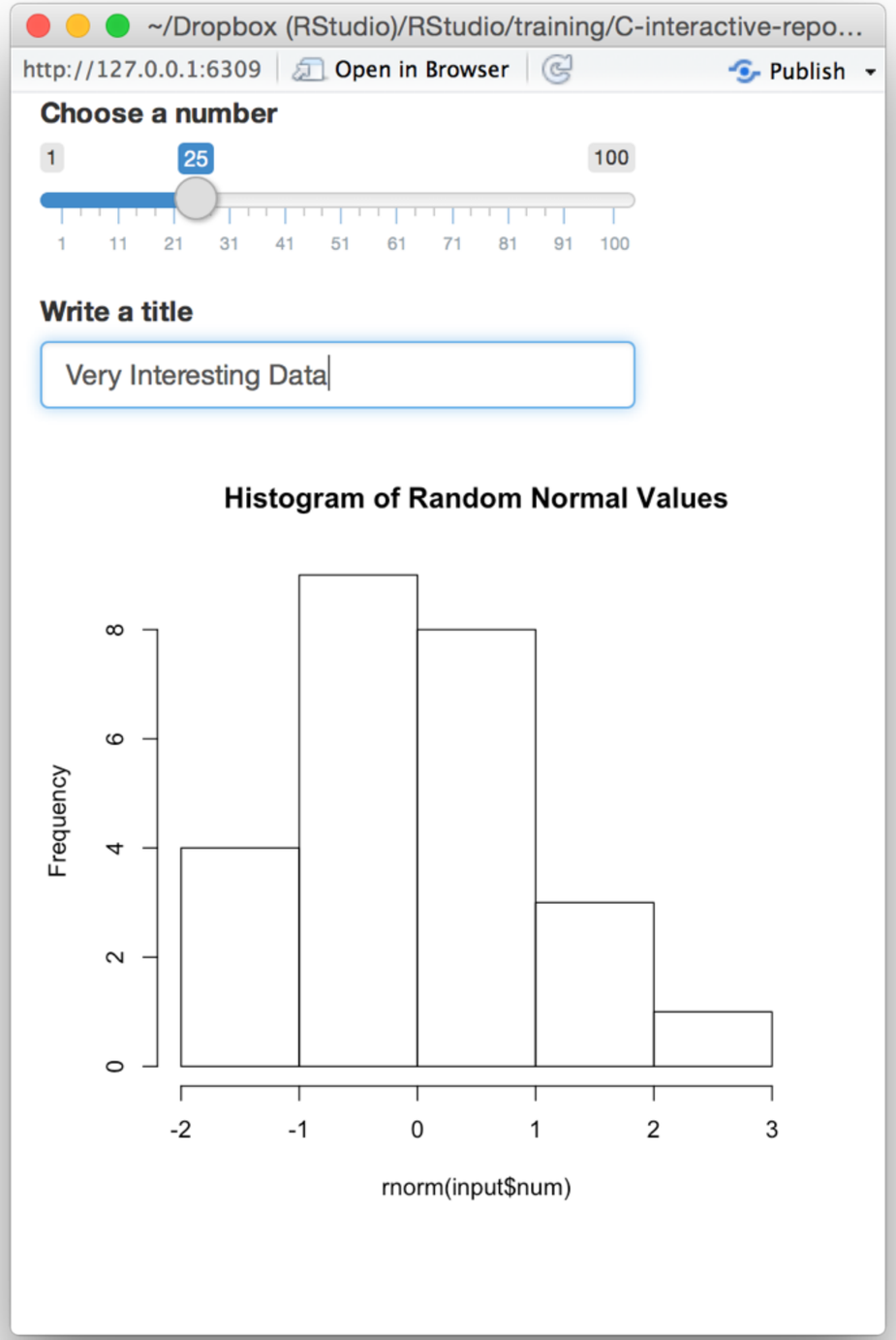
```
# 01-two-inputs

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

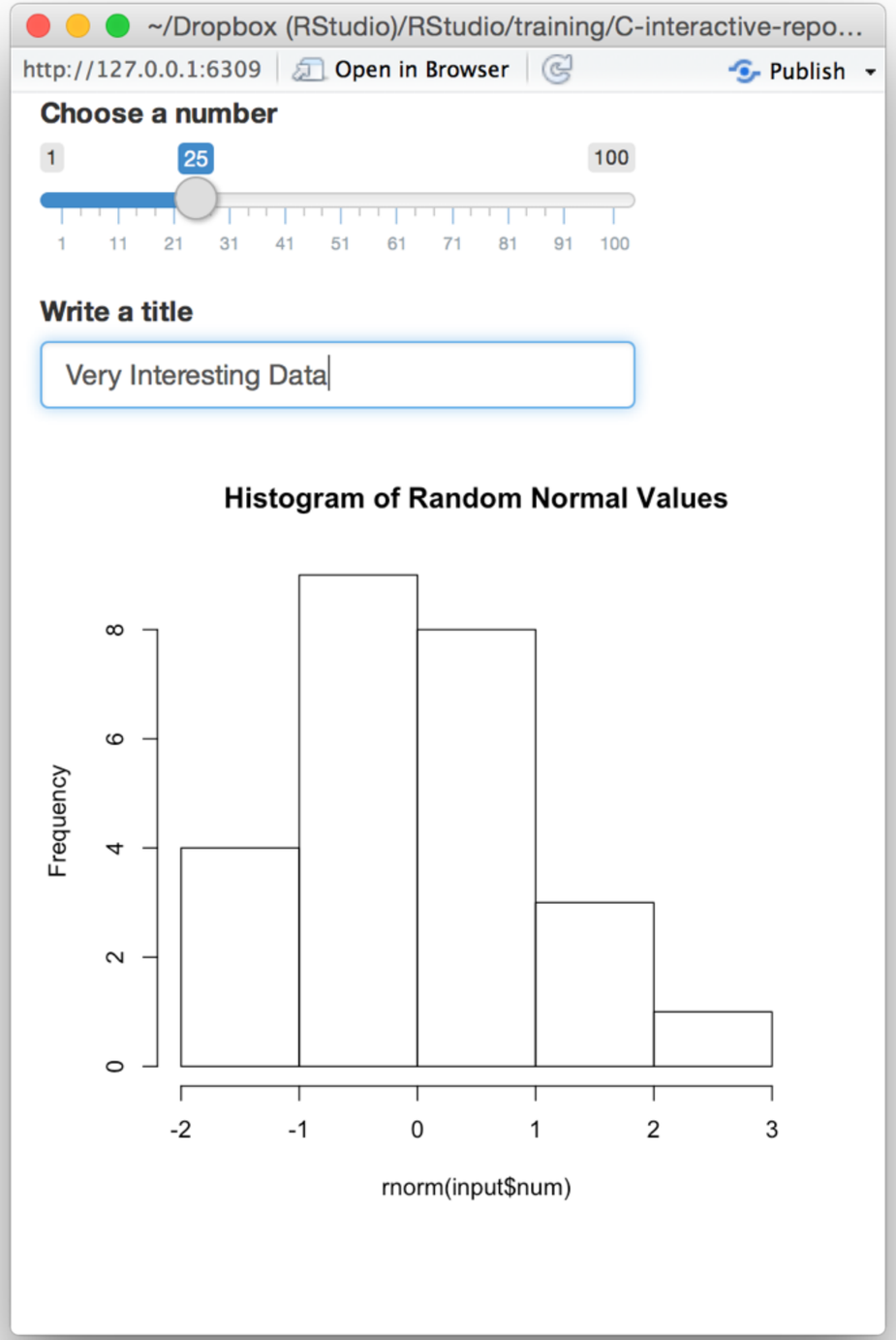
```
output$hist <- renderPlot({
  hist(rnorm(input$num),
    main = input$title)
})
```



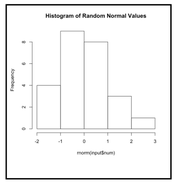
input$num

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)
})
```

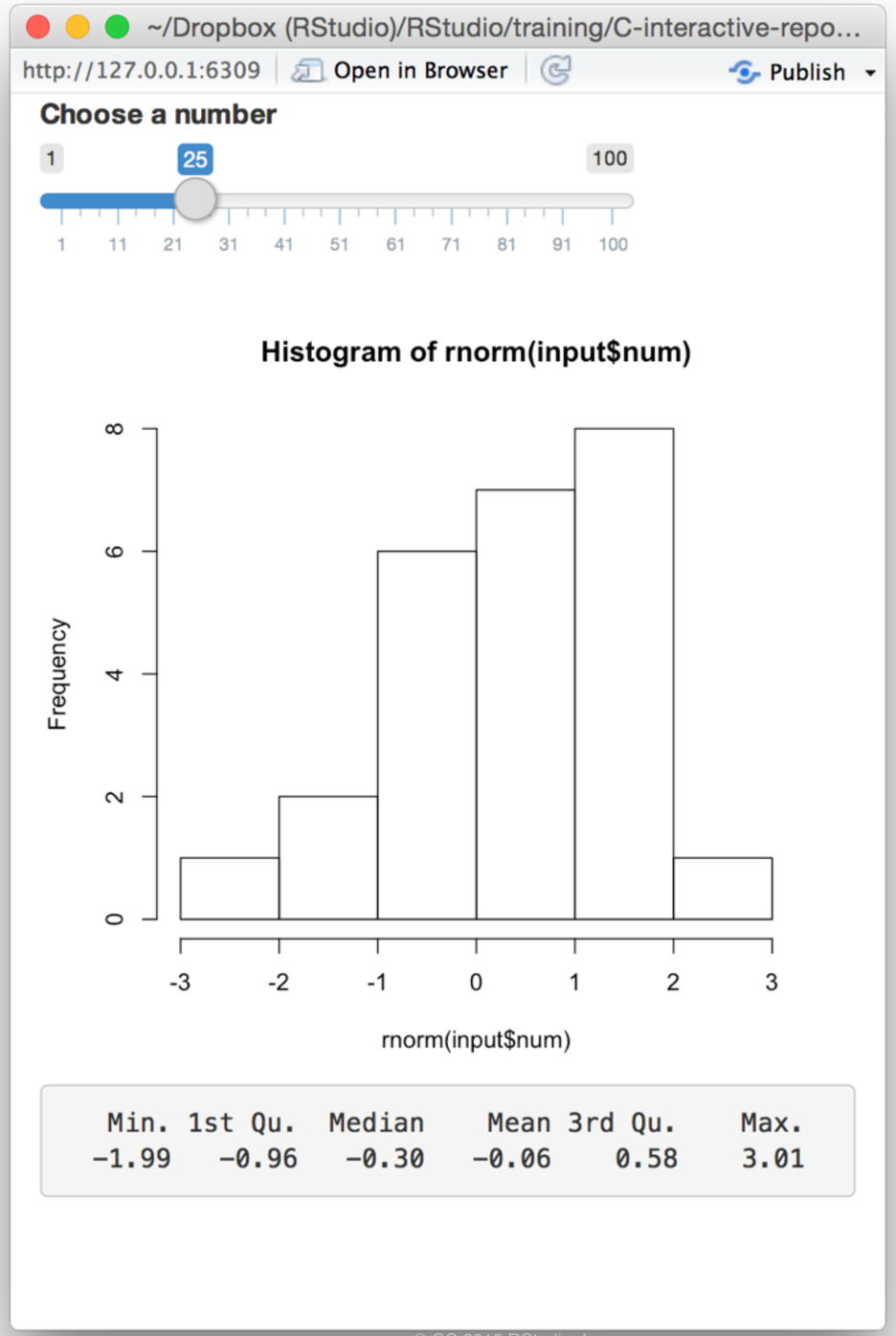
render*() makes an observer object that has a **block of code** associated with it

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

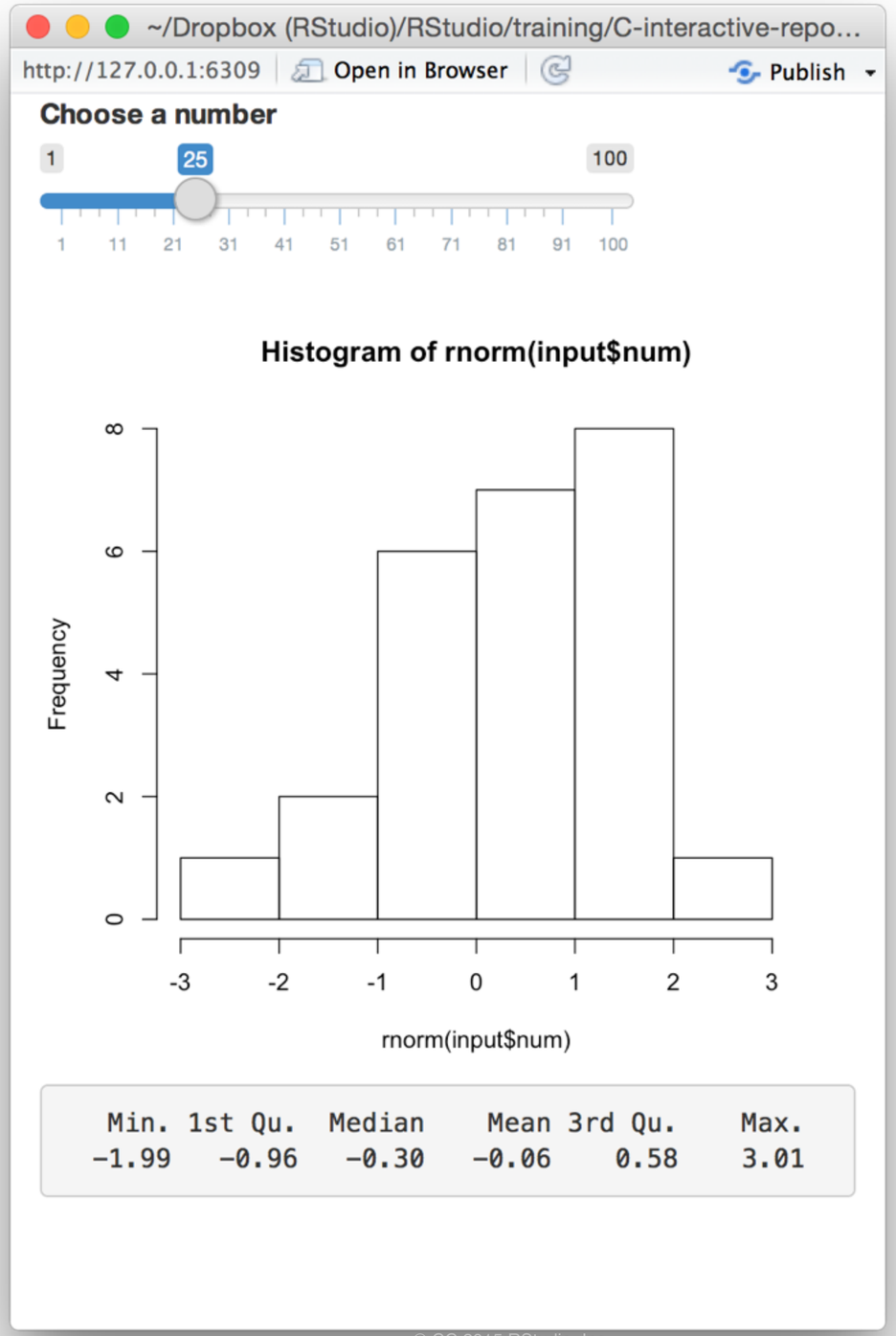
The object will **rerun the entire code block** to update itself whenever it is invalidated

Modularize code with reactive()

```
# 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)
```



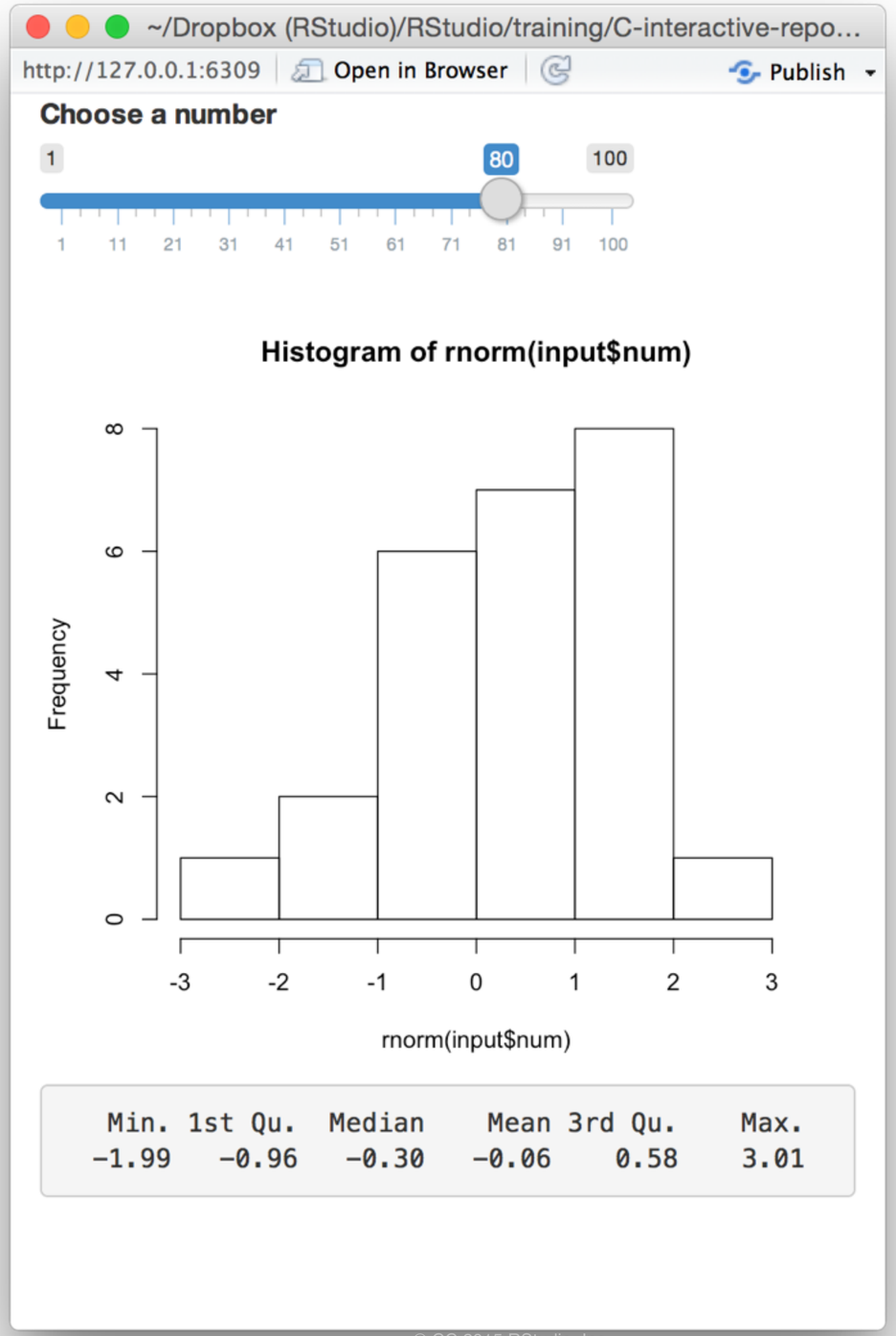
```
# 02-two-outputs

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))
  })
```

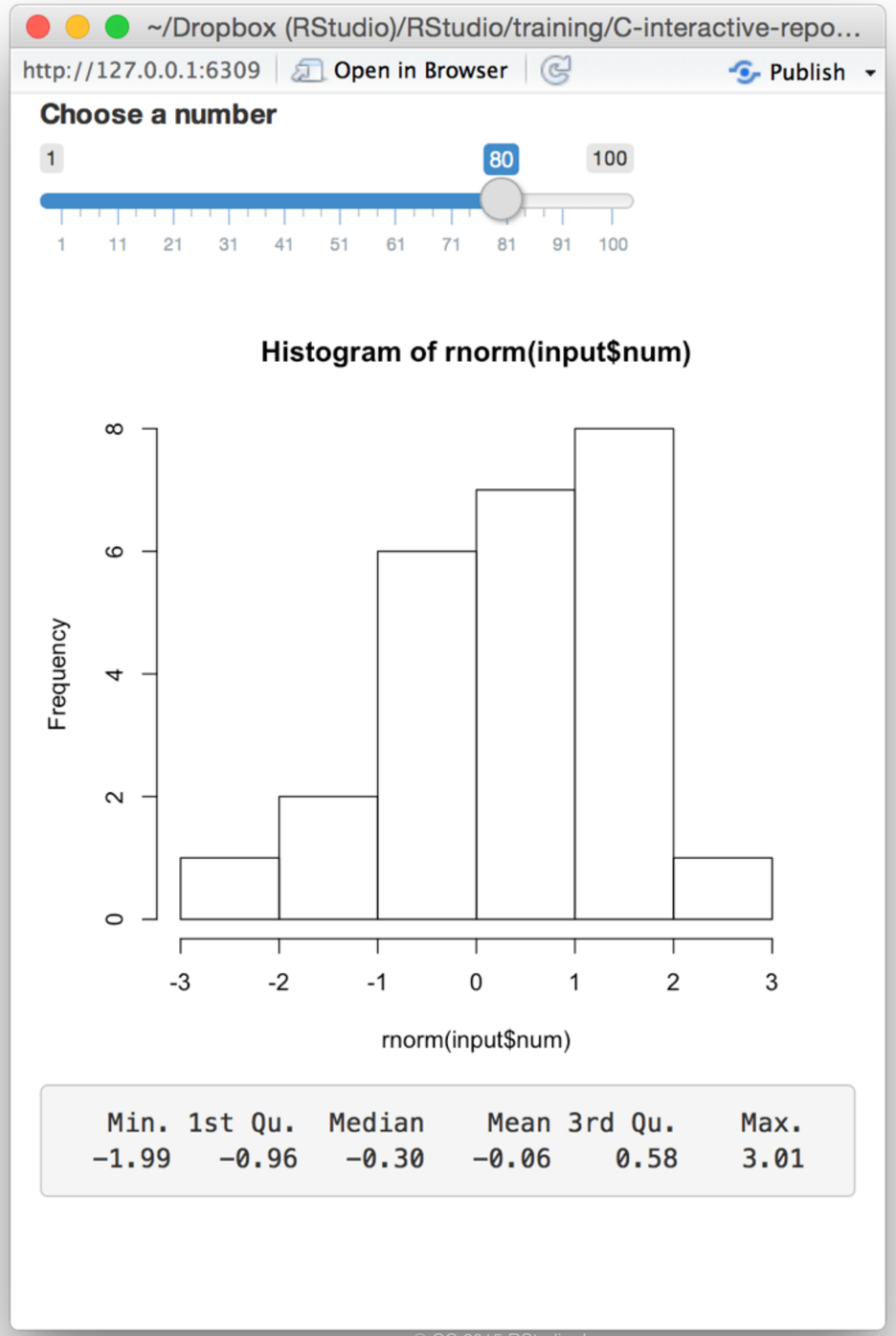
```
output$stats <-
  renderPrint({
    summary(rnorm(input$num))
  })
```



input$num

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

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

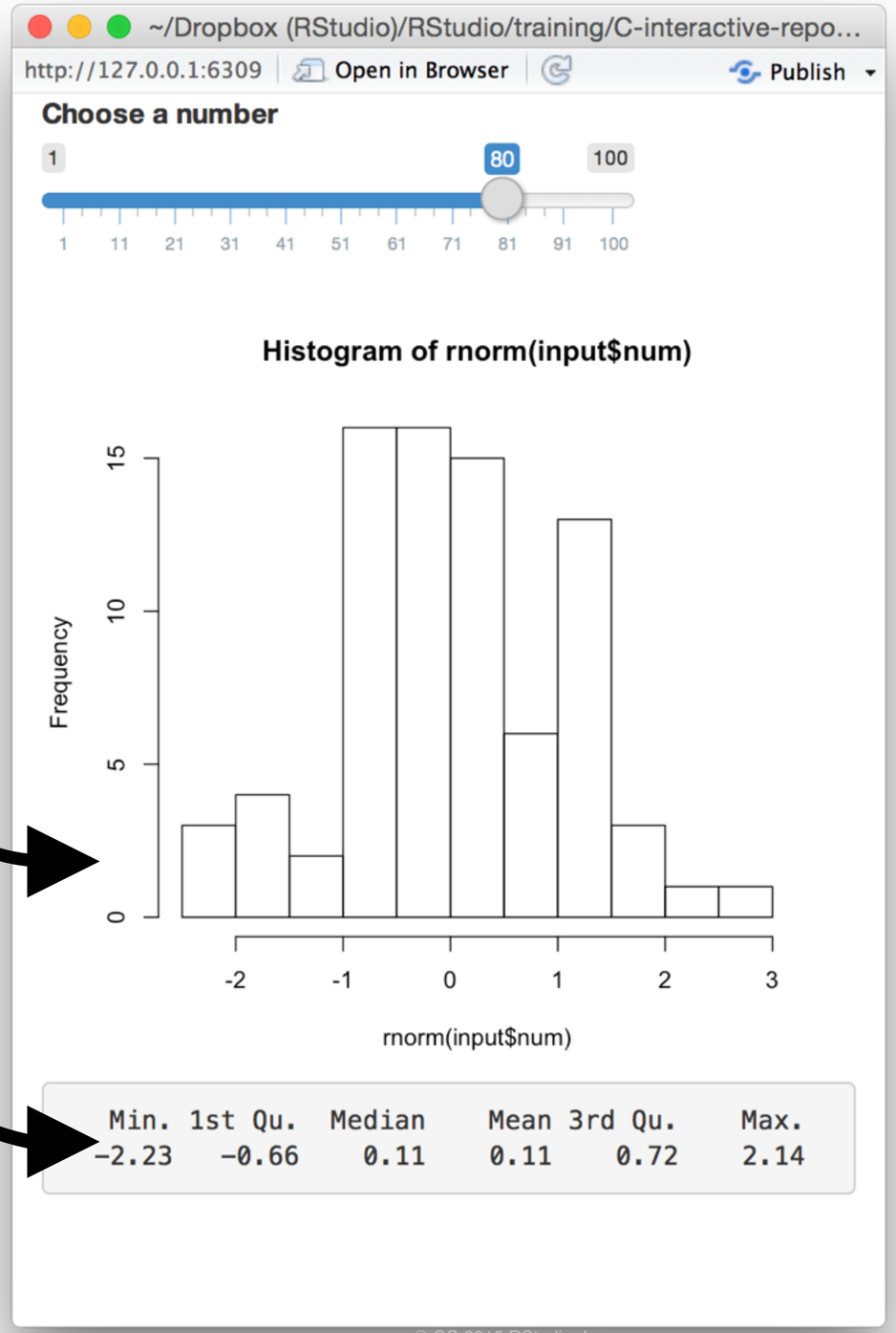


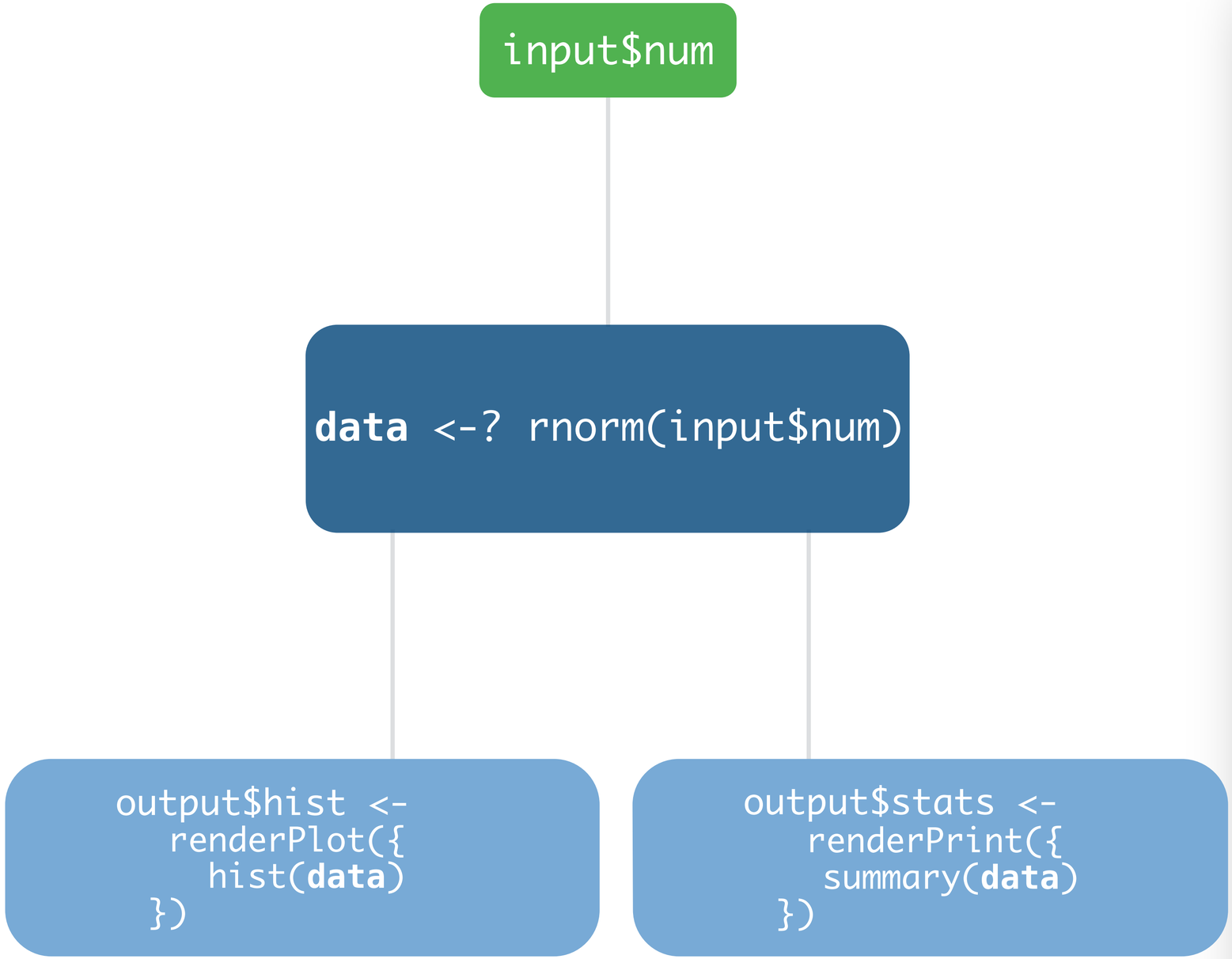
input$num

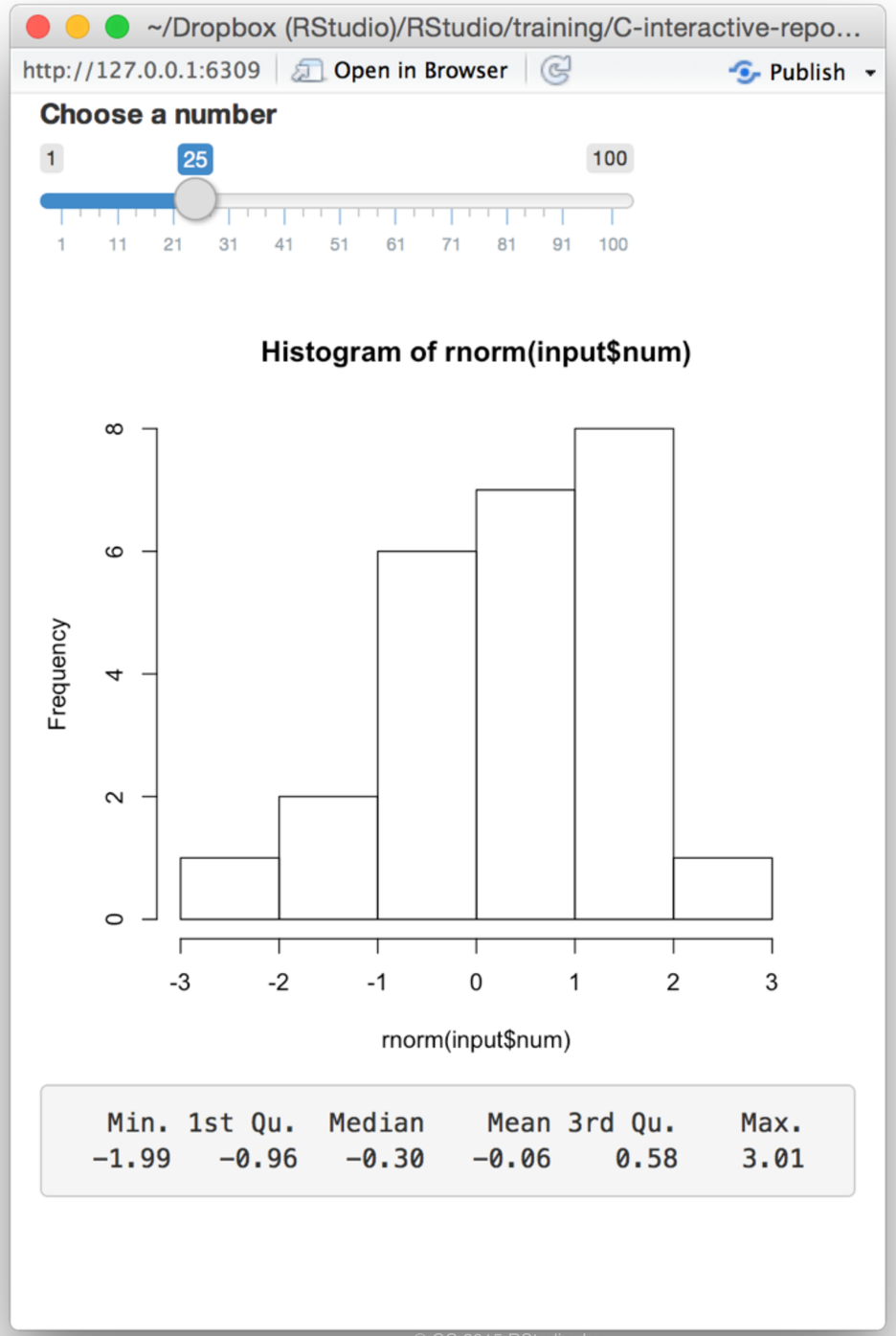
Can these describe the same data?

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

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







reactive()

Builds a reactive object (reactive expression)

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

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

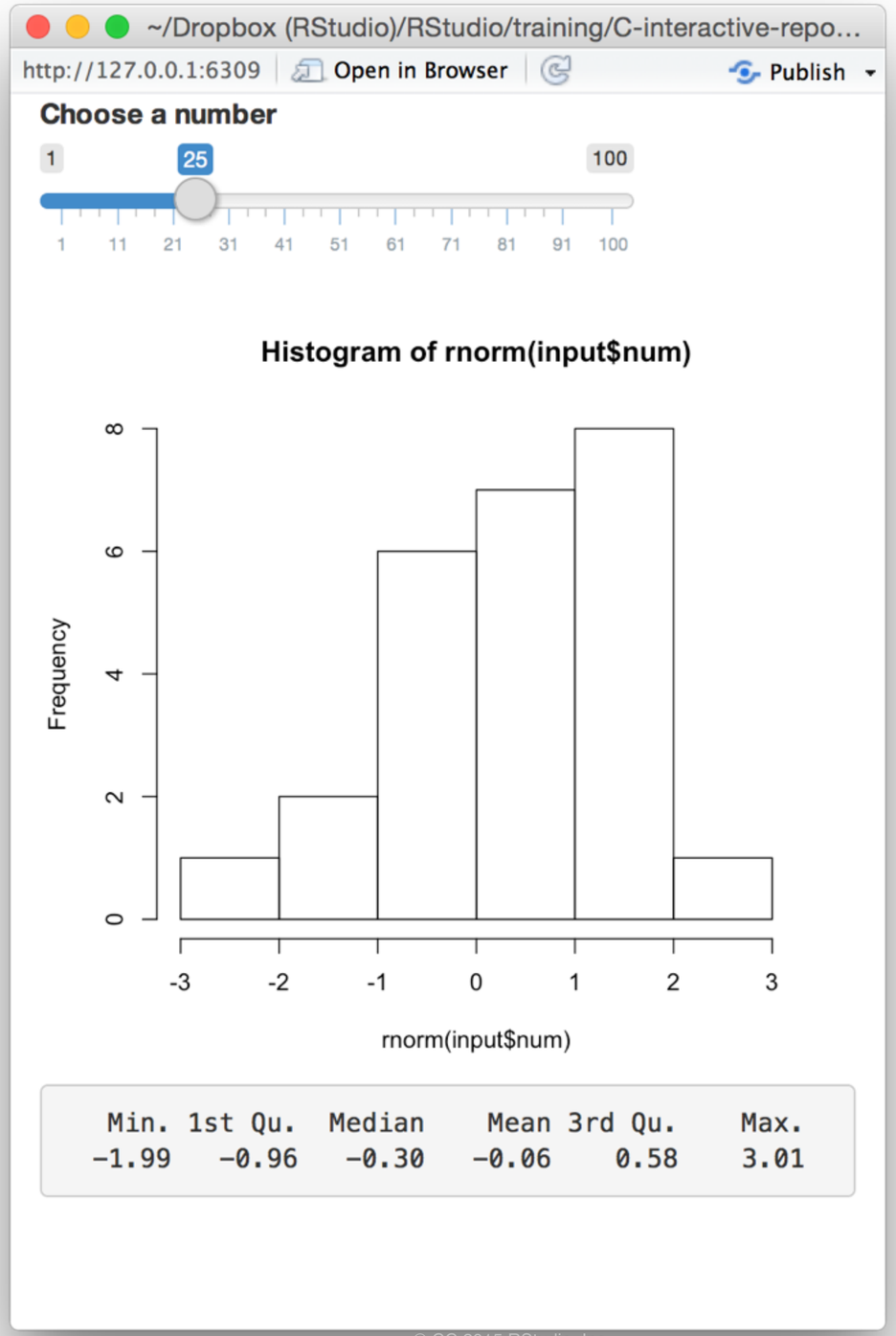
code used to build (and rebuild) object

A reactive expression is special in two ways

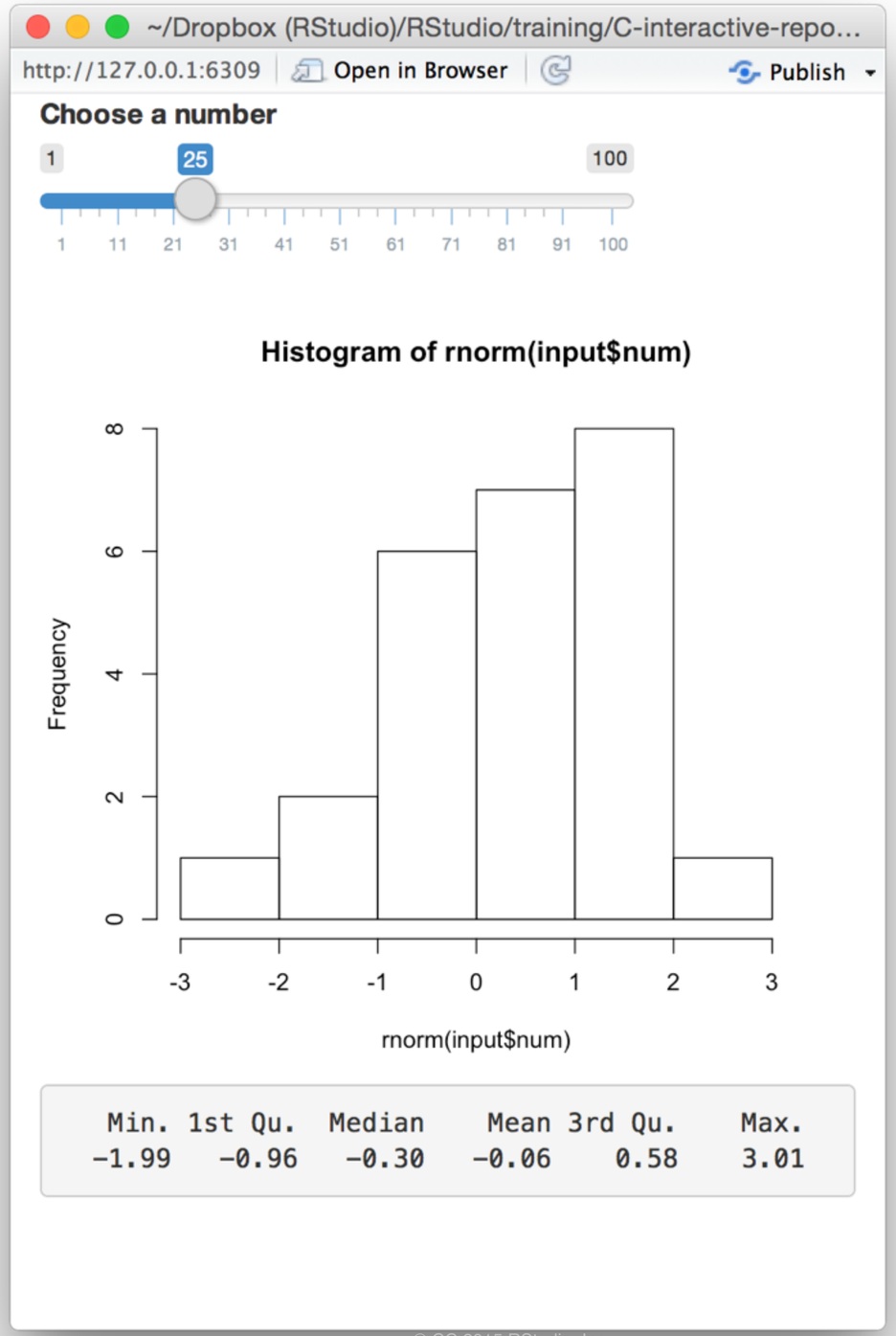
```
data()
```

1. You call a reactive expression like a function

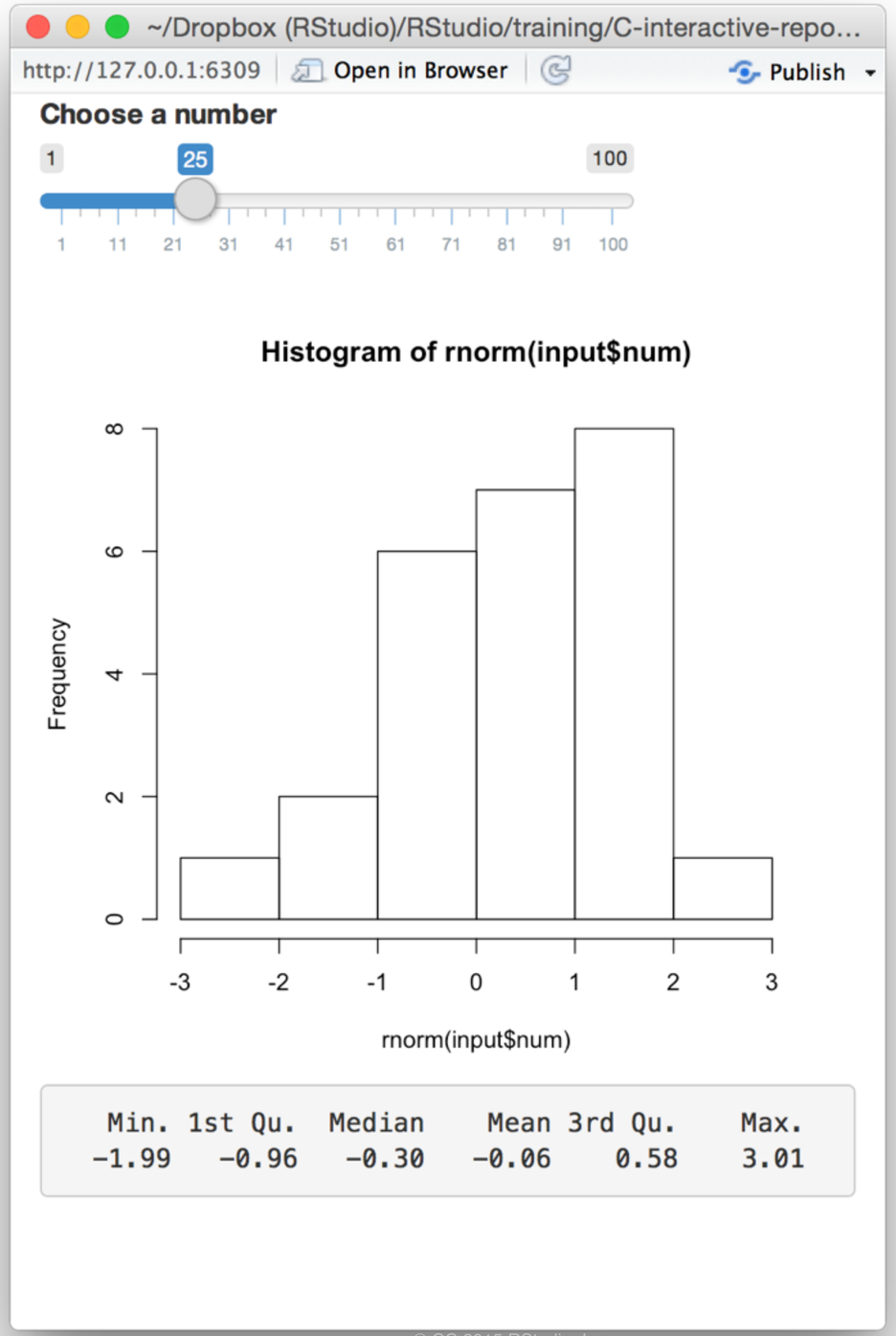
```
# 02-two-outputs

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)
```

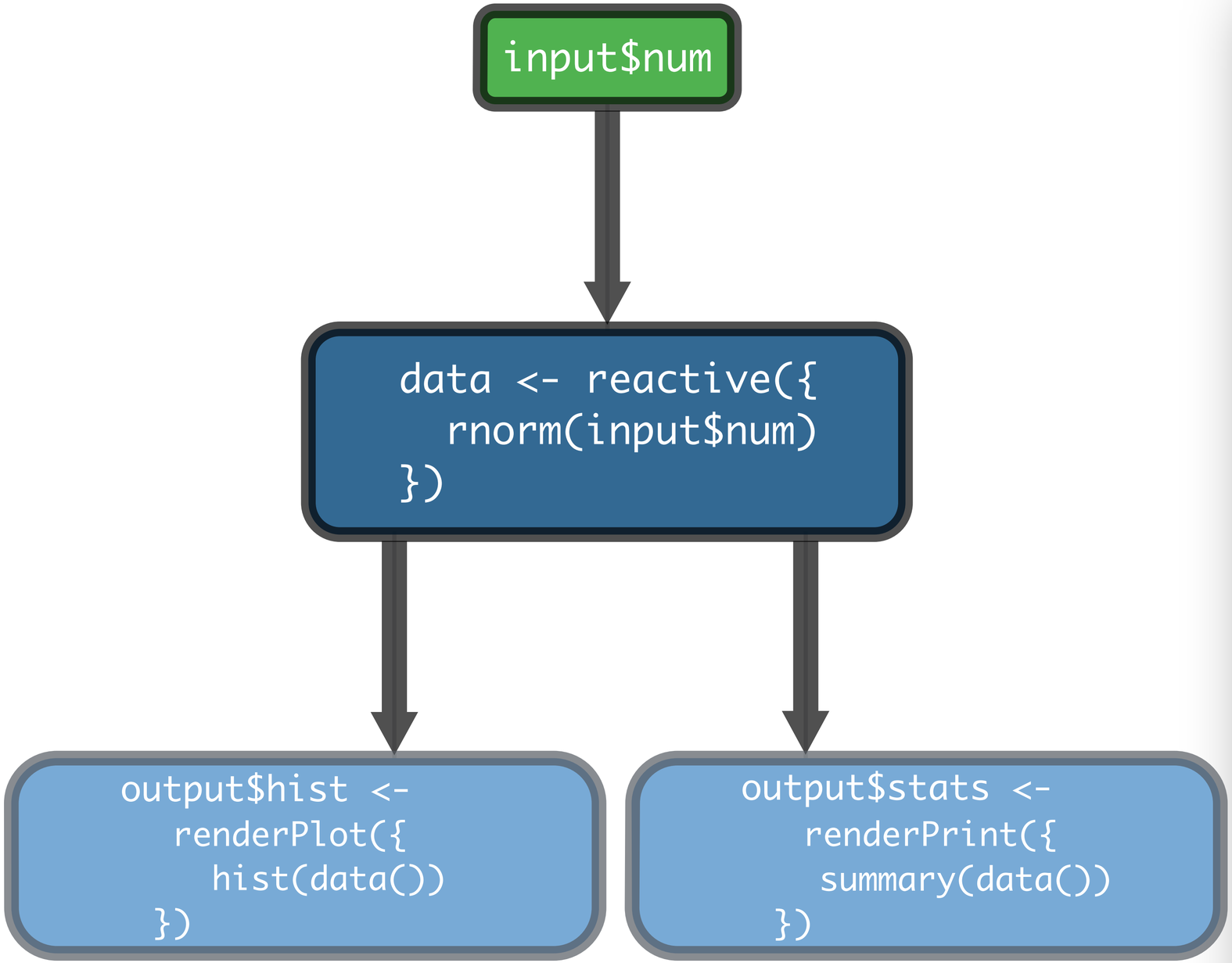


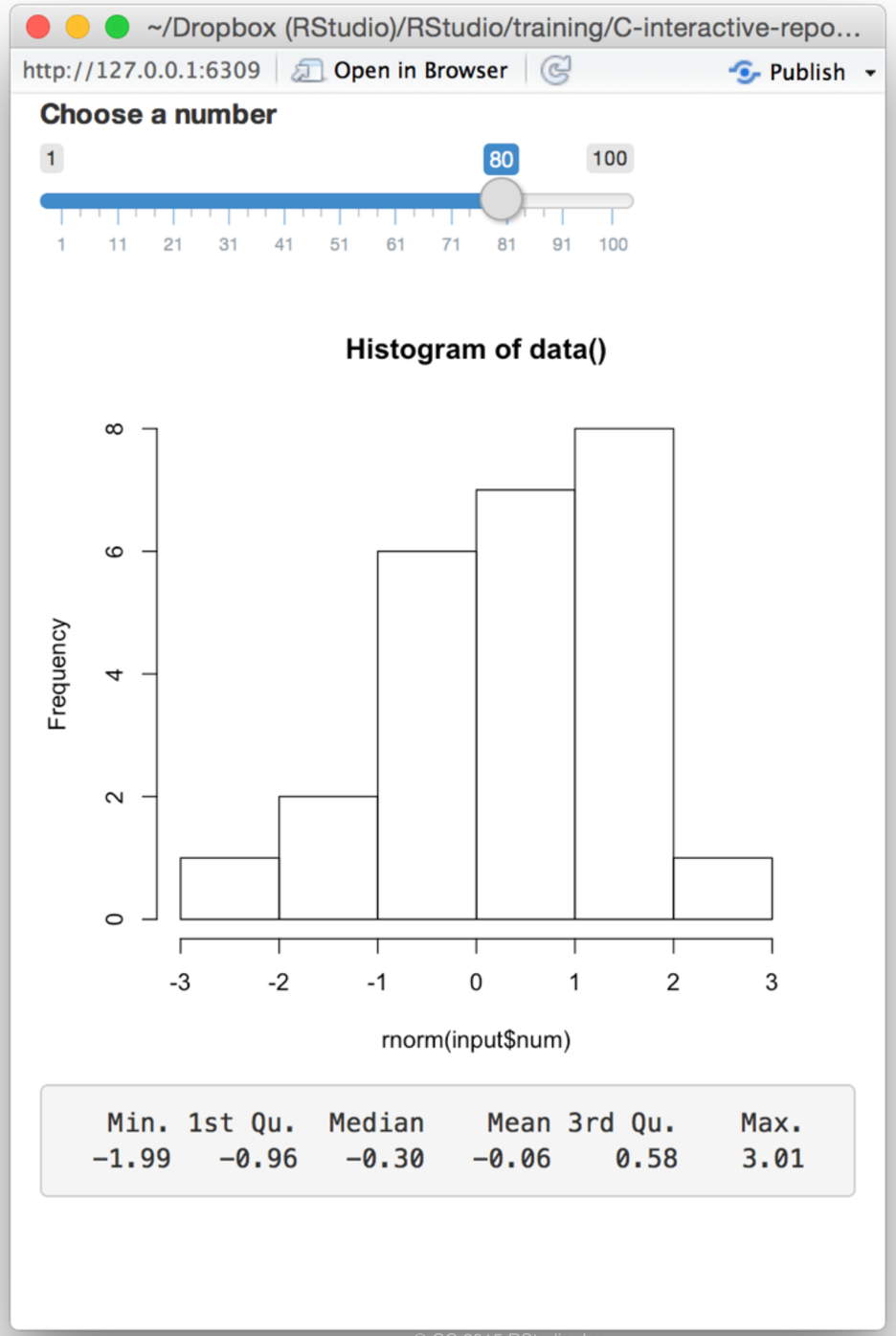
```r
# 02-two-outputs

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 = server)
```

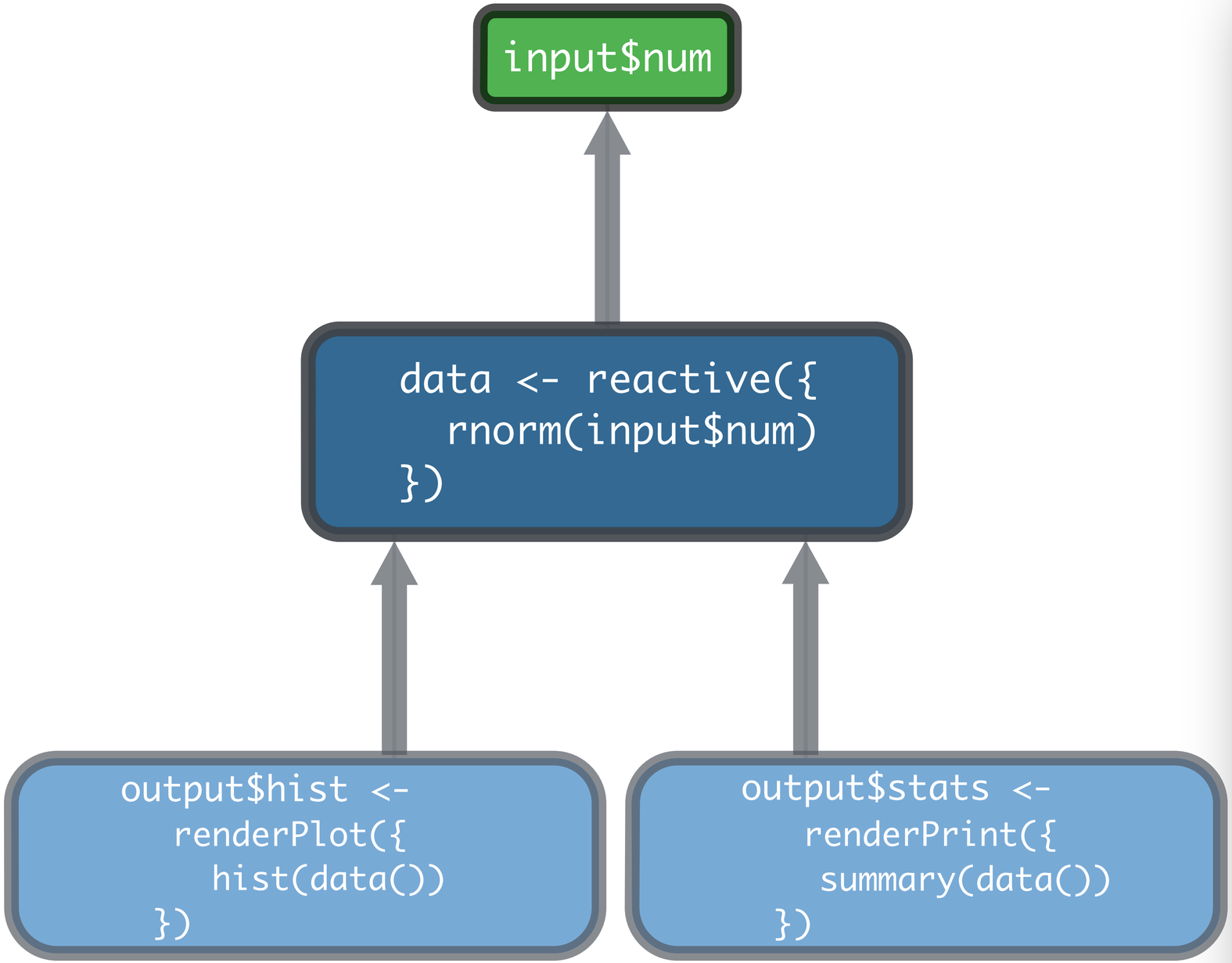


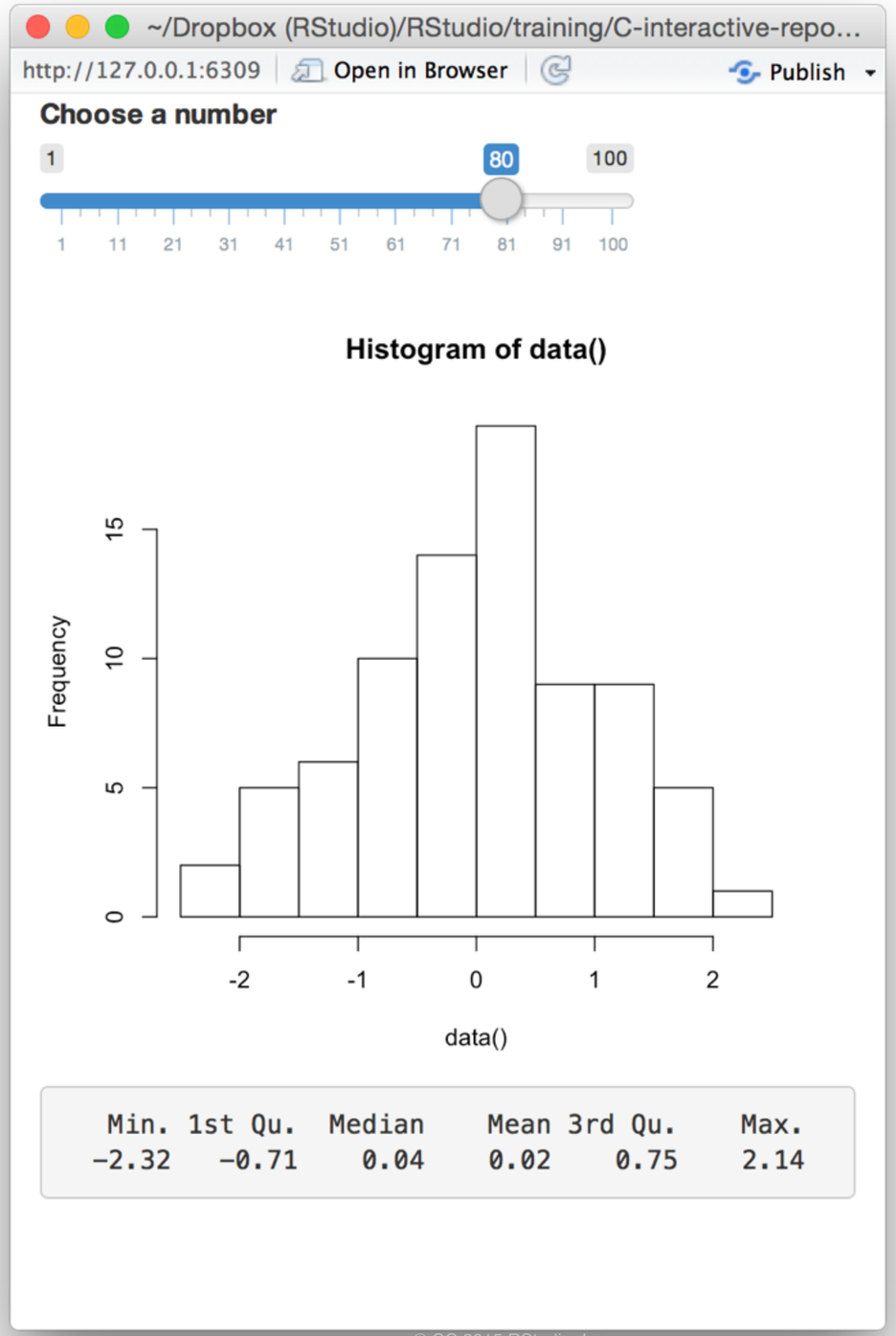
```
# 03-reactive

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 = server)
```











A reactive expression is special in two ways

```
data()
```

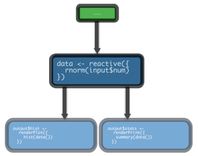
1 You call a reactive expression like a function

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

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**

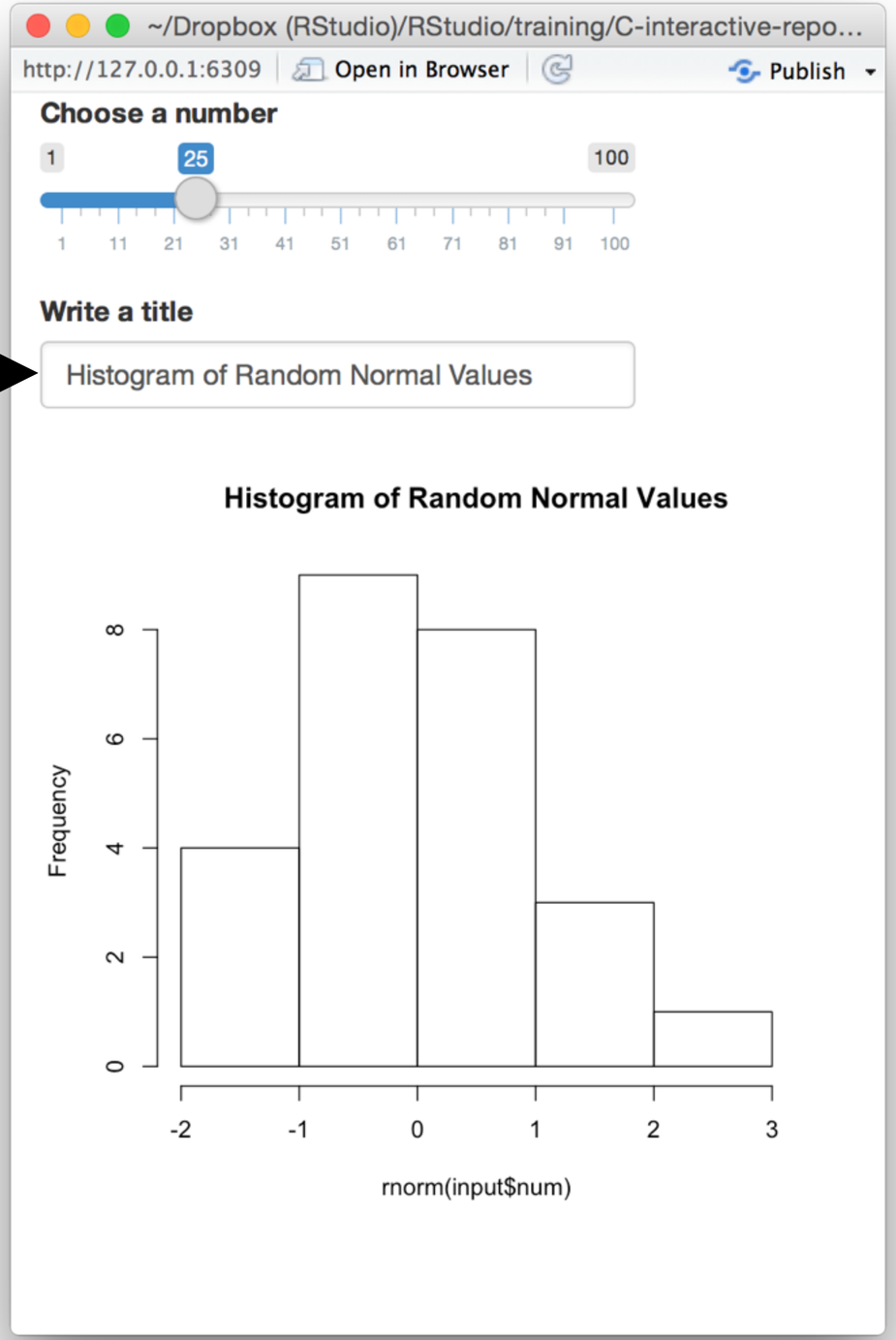
2

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

Prevent reactions with isolate()

```
# 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()

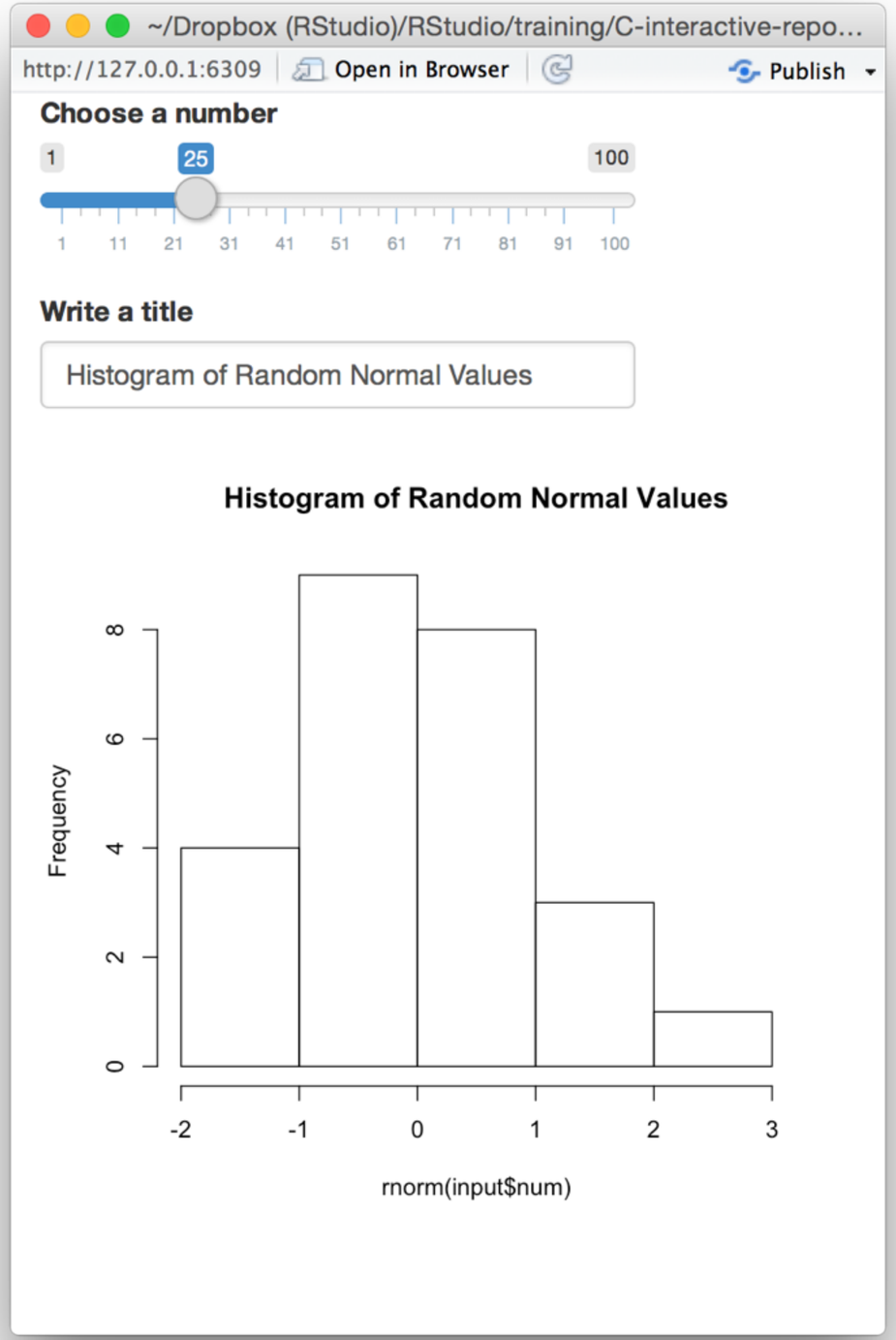
Returns the result as a non-reactive value

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

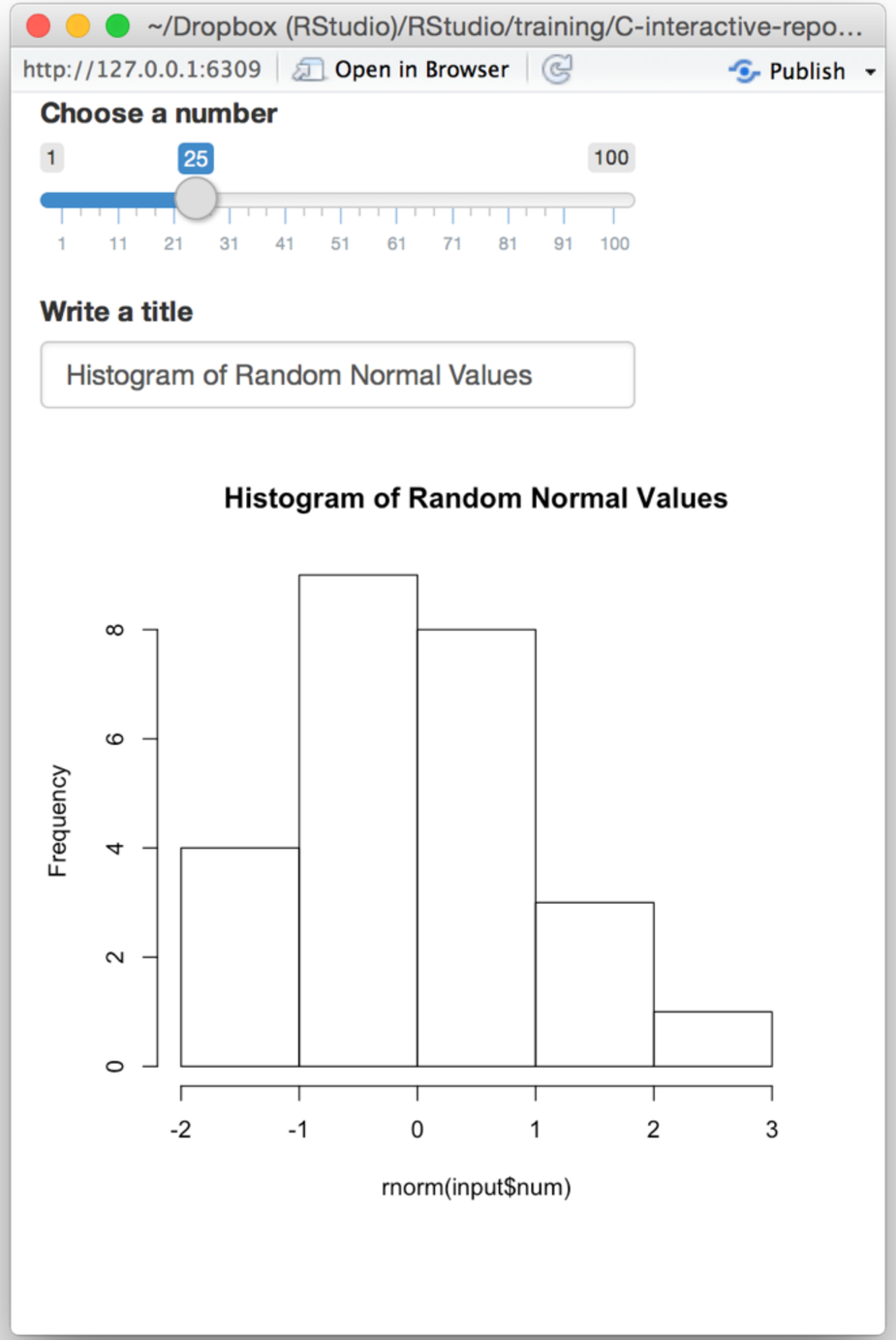
object will NOT respond to *any reactive value in the code*

code used to build object

```
# 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)
```



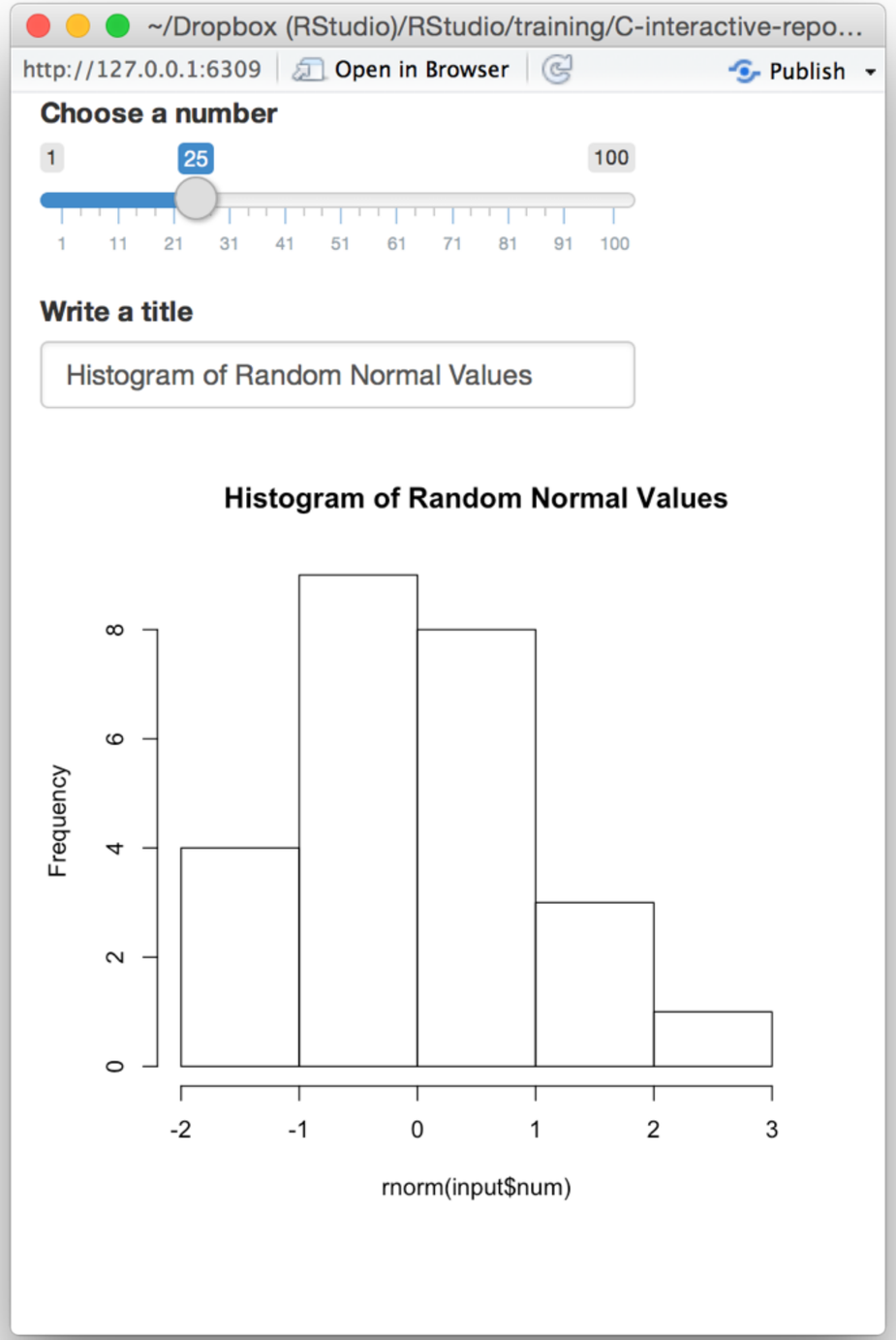
```
# 04-isolate

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

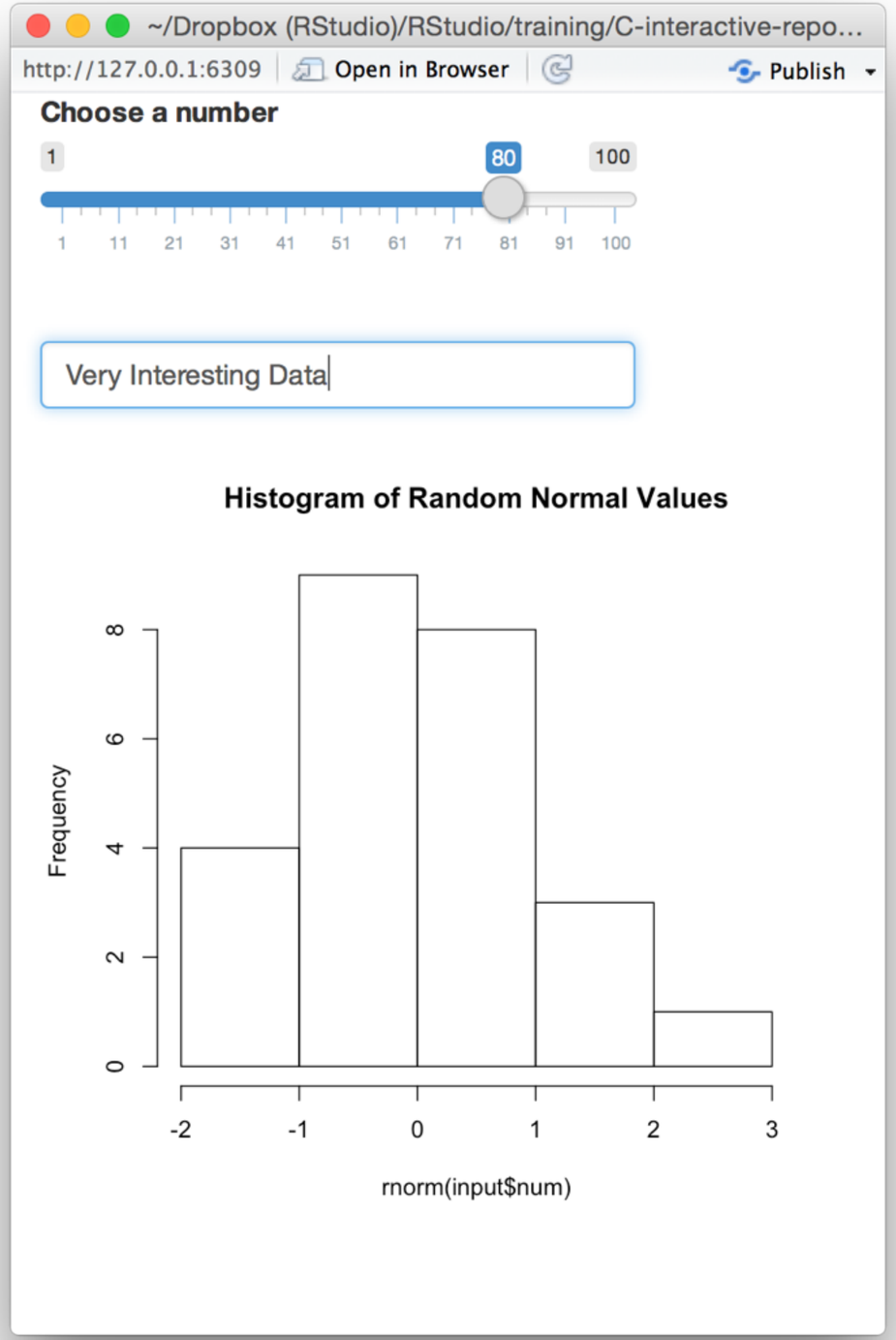
```
output$hist <- renderPlot({
  hist(rnorm(input$num),
       main = isolate(input$title))
})
```

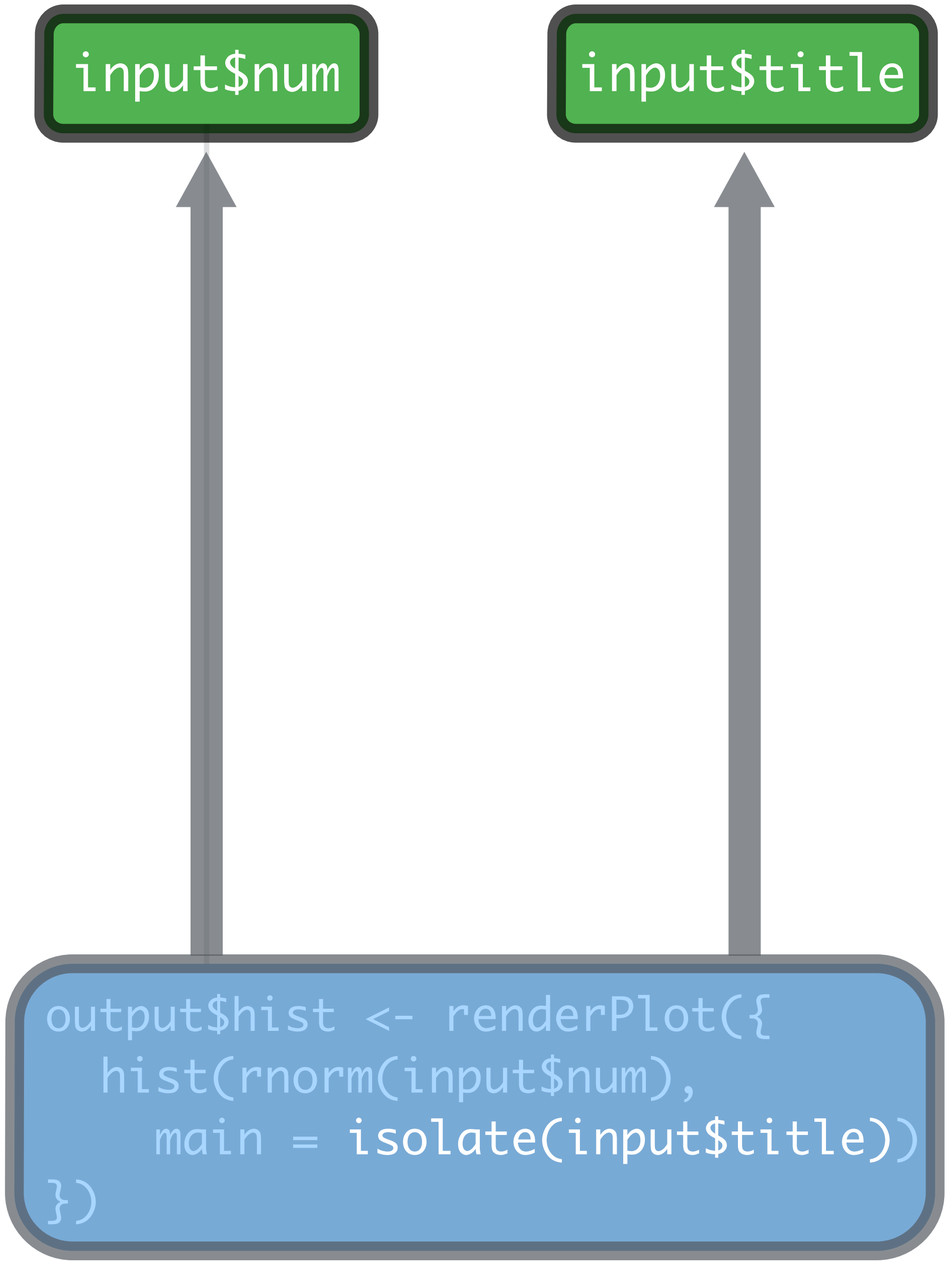


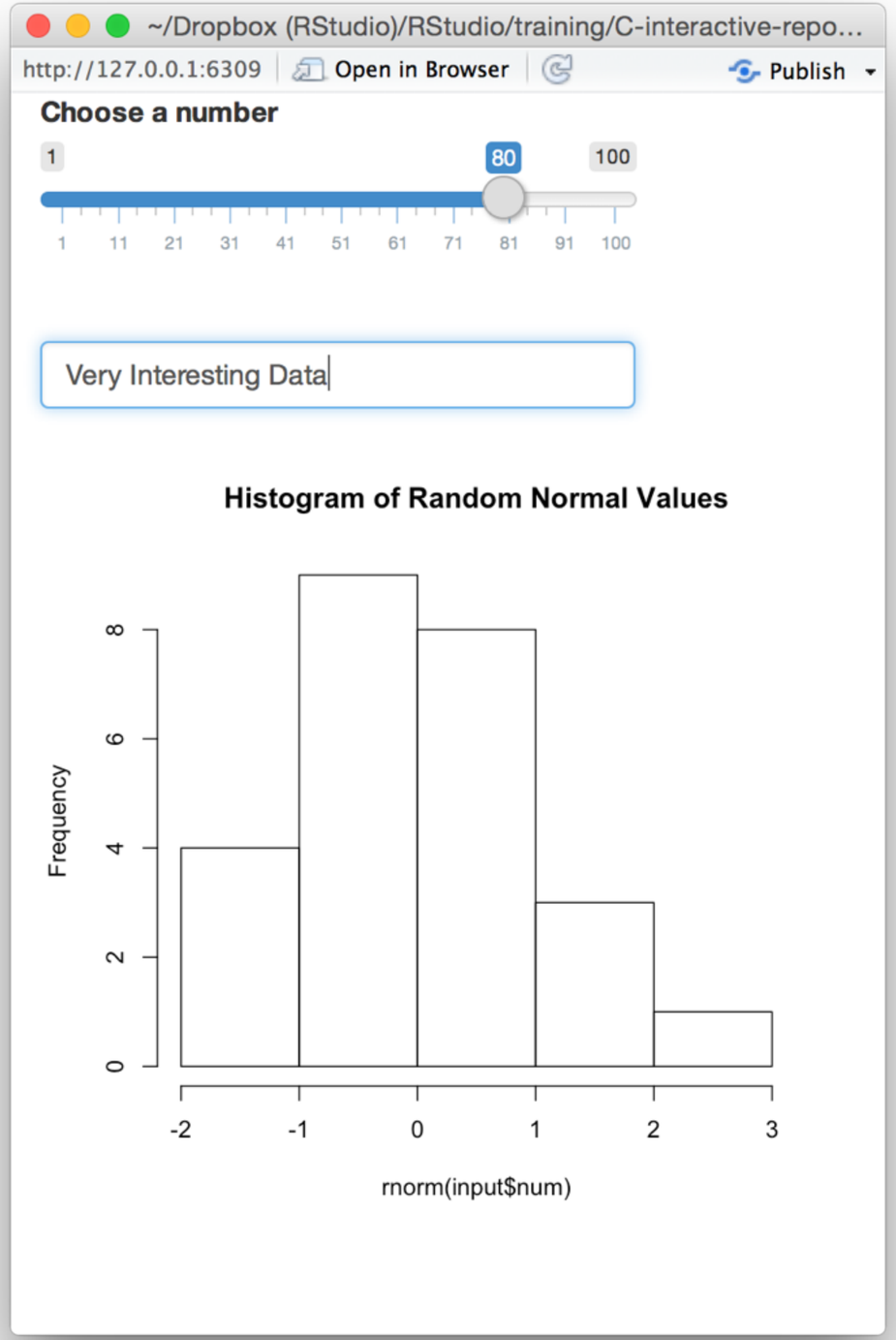
input$num

input$title

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





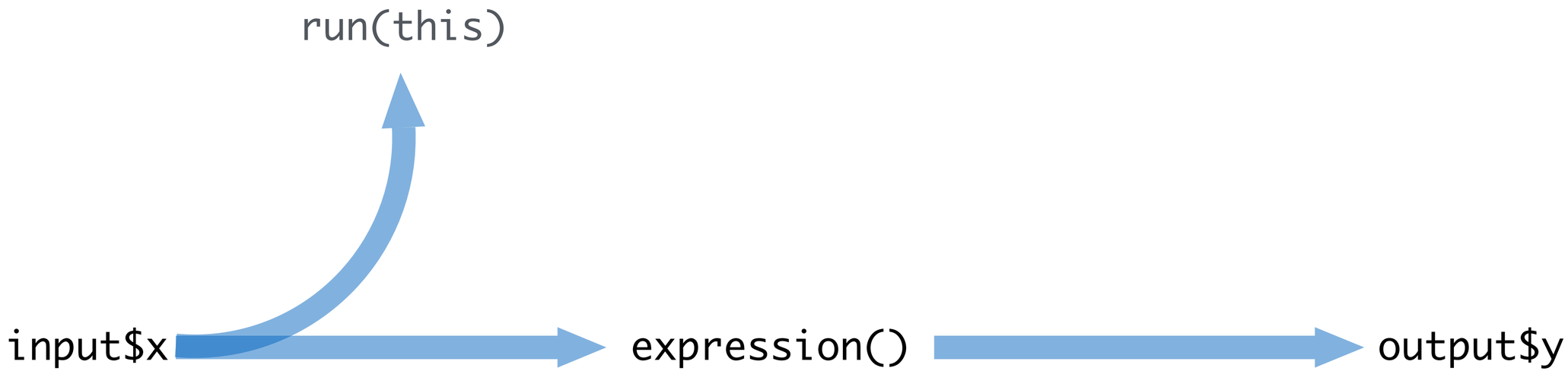


Recap: isolate()

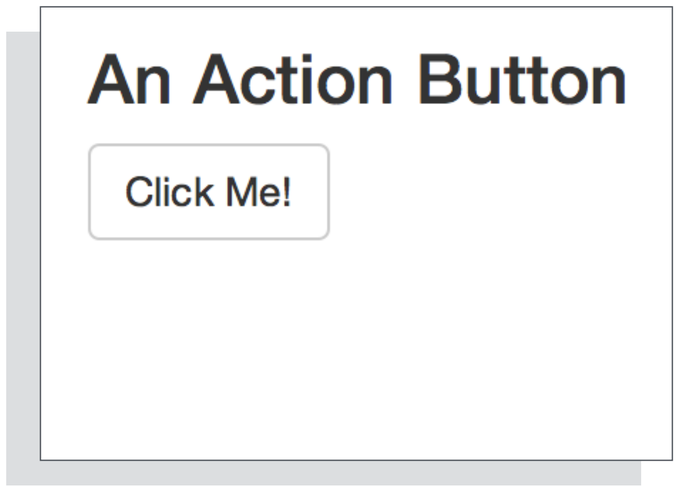
isolate() makes an **non-reactive object**

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

Trigger code with observeEvent()



Action buttons



input function

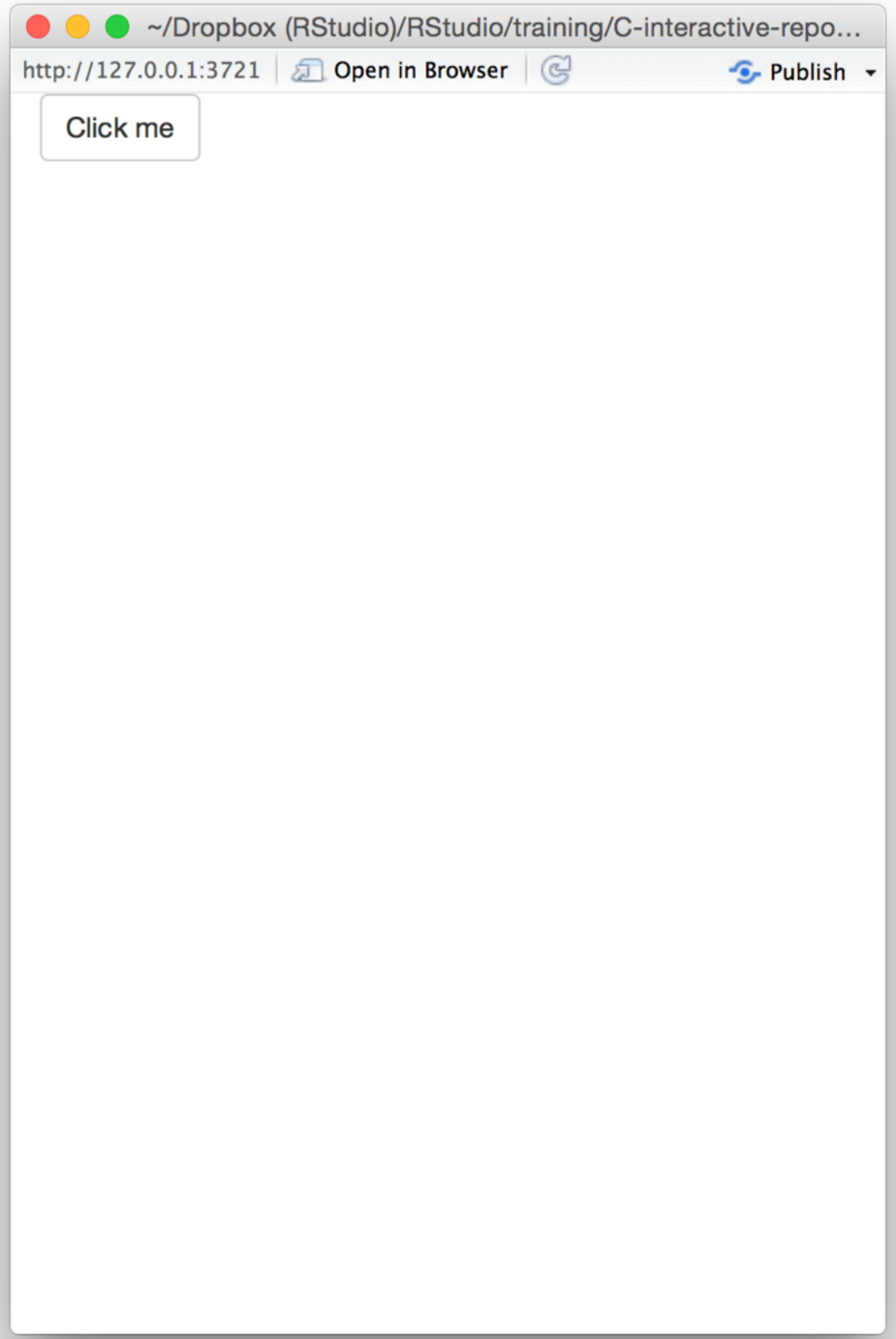
input name (for internal use)

label to display

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

Notice: Id not ID

```
# 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) })
```

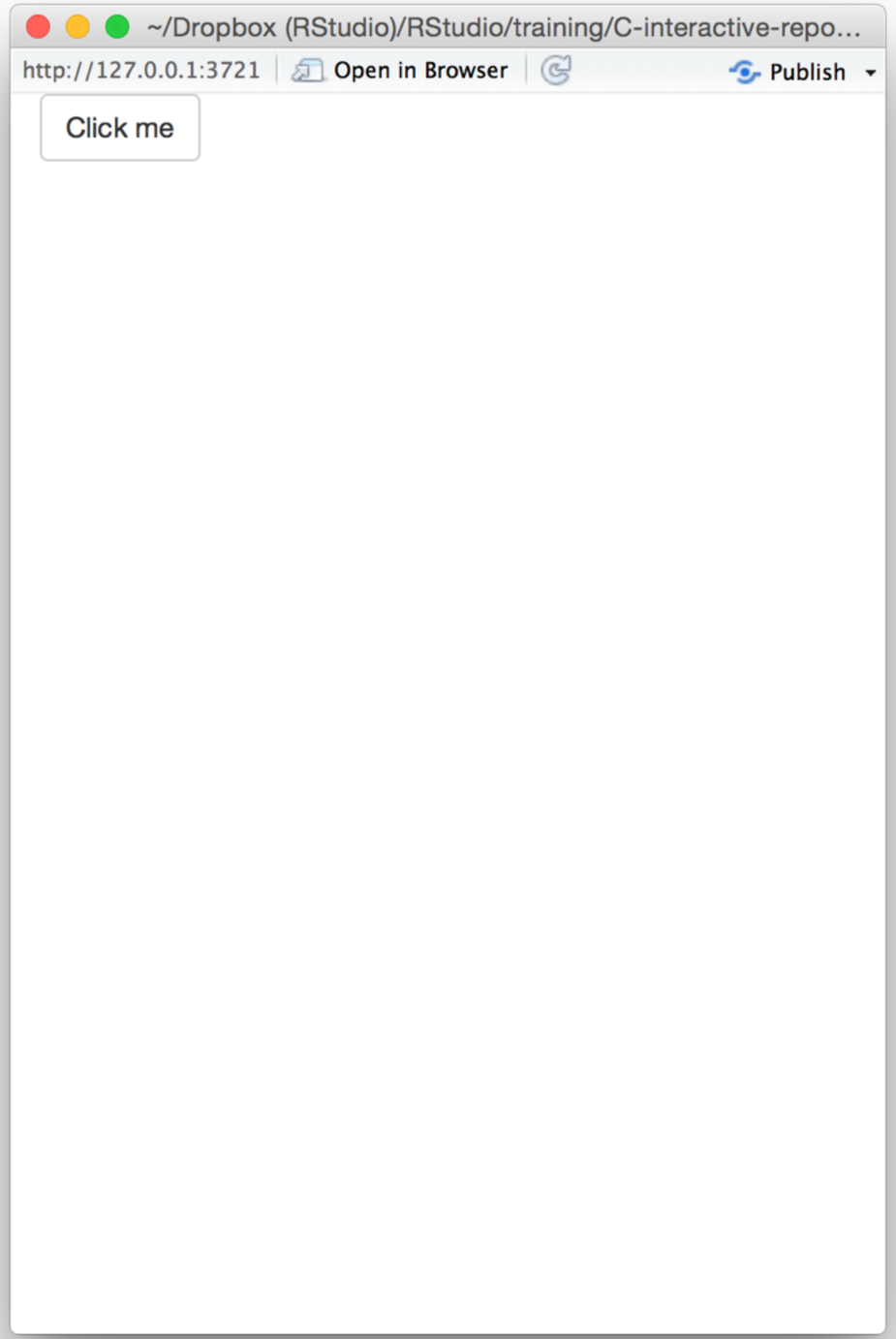
reactive value(s) to respond to

(observer invalidates ONLY when this value changes)

code block to run whenever observer is invalidated

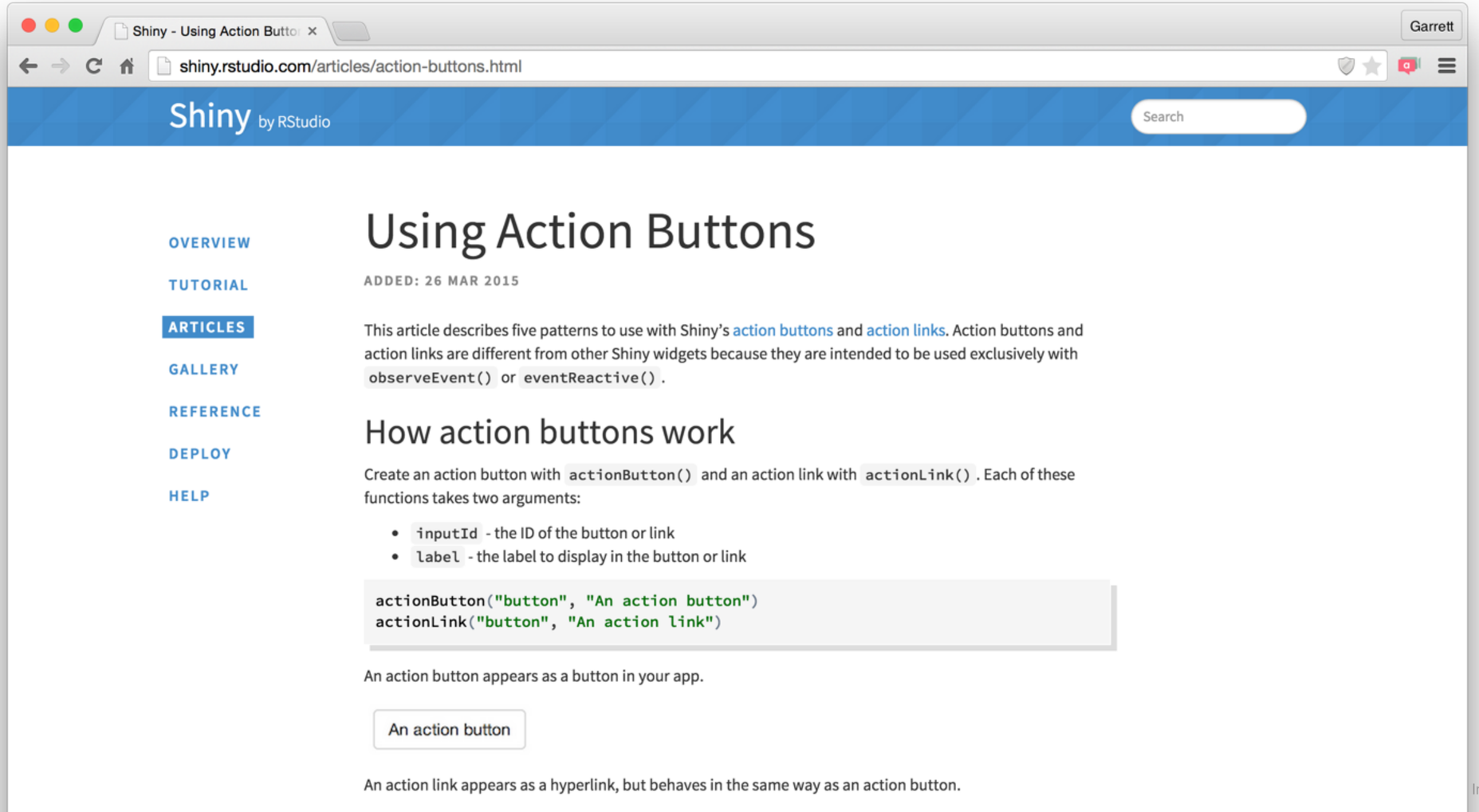
note: observer treats this code as if it has been isolated with isolate()

```
# 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



observe()

Also triggers code to run on server.

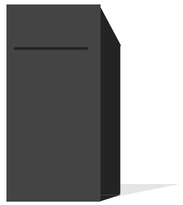
Uses same syntax as render*(), reactive(), and isolate()

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

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

code block to run whenever observer is invalidated

Recap: observeEvent()



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

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

reactive value(s) to respond to

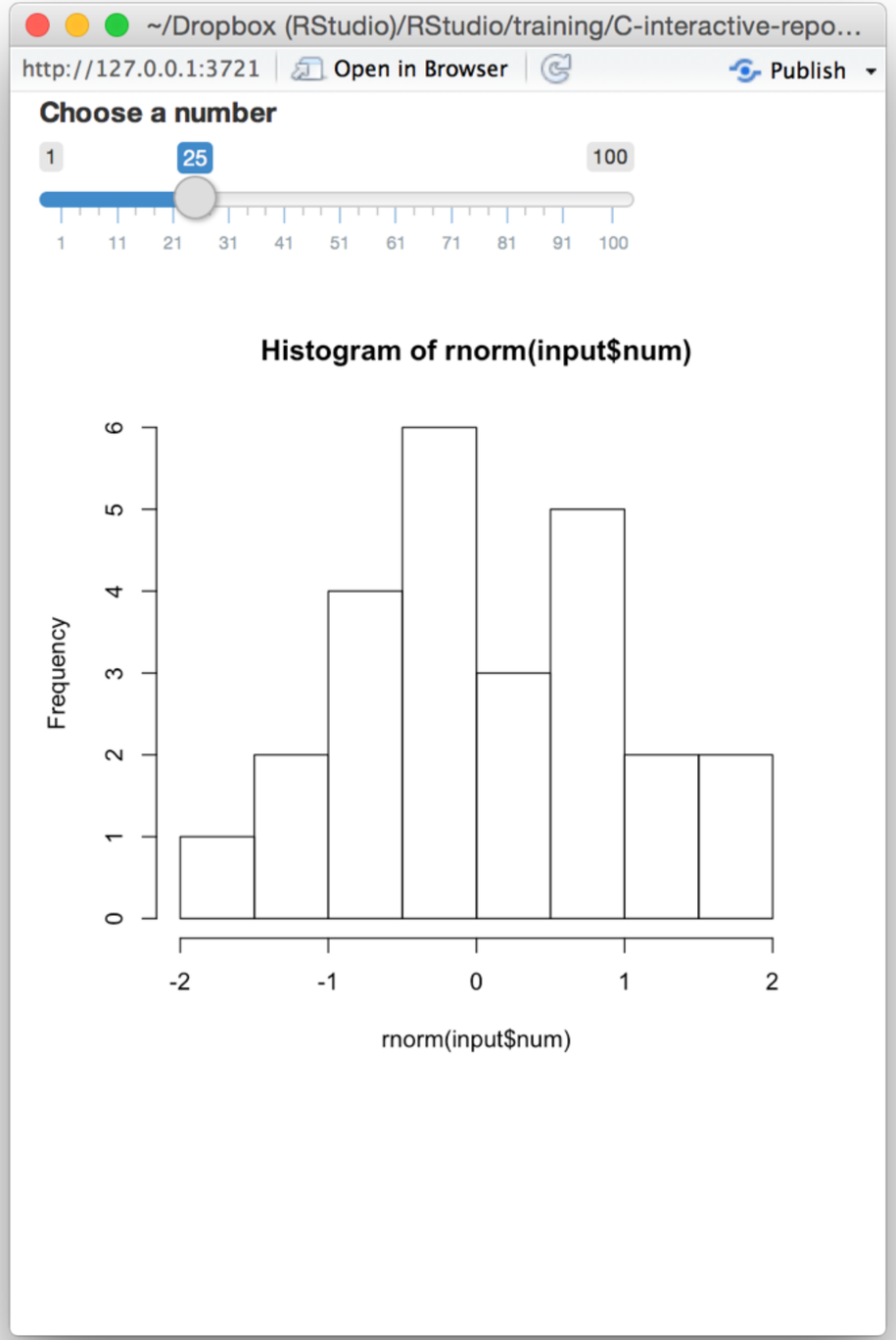
Specify **precisely** which reactive values should invalidate the observer

`observe()`

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

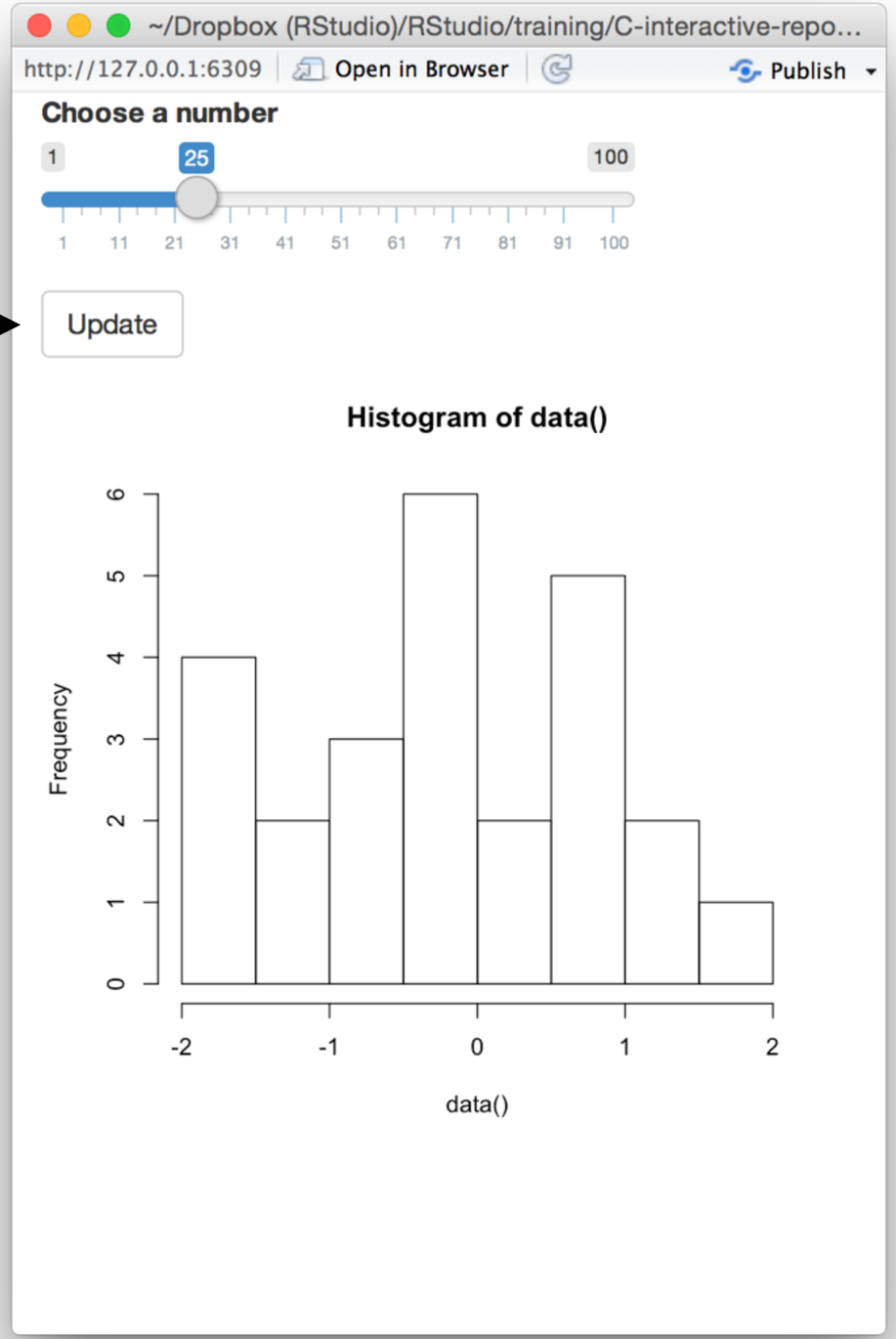
Delay reactions
with eventReactive()

```
# 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) })
```

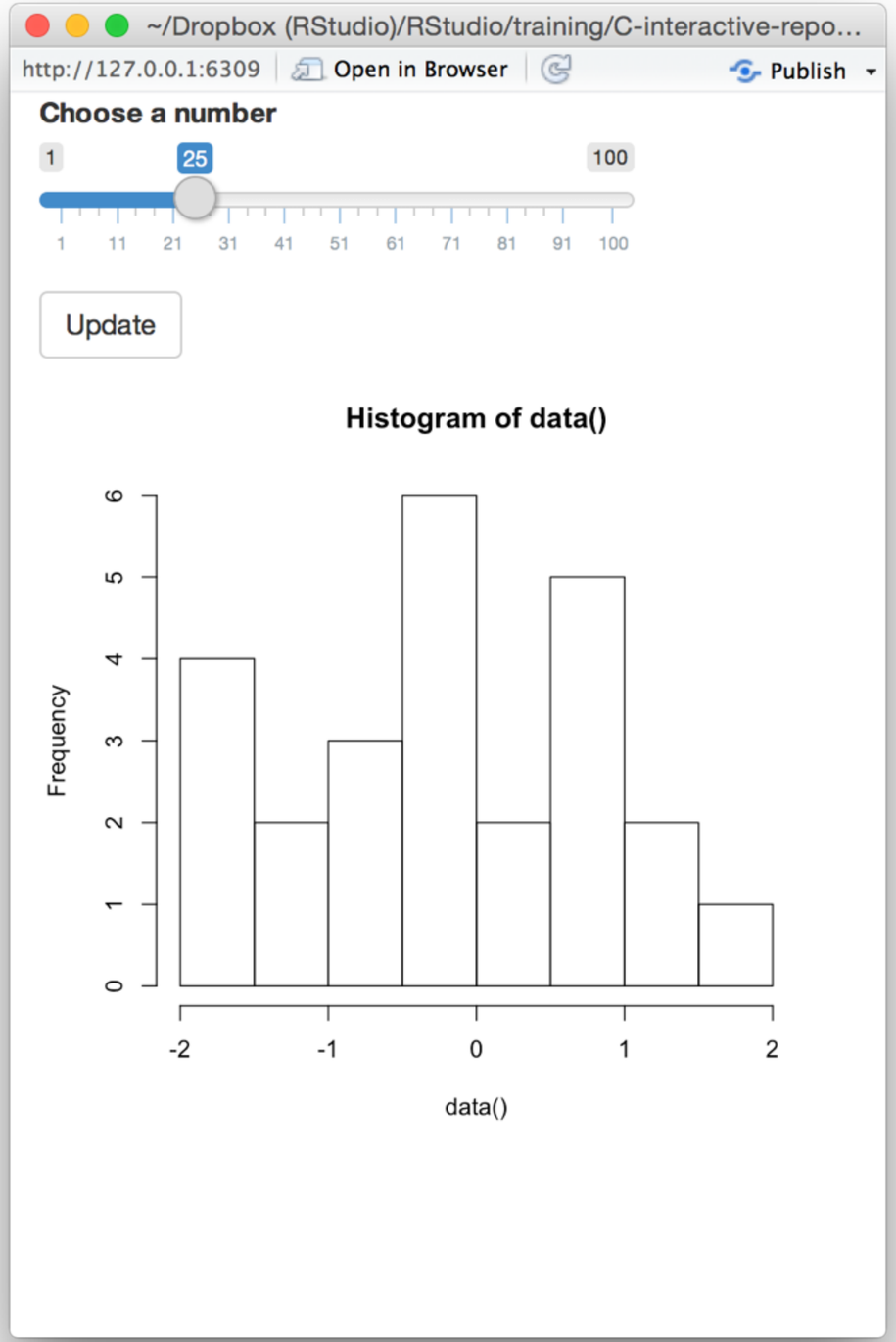
reactive value(s) to respond to

(expression invalidates ONLY when this value changes)

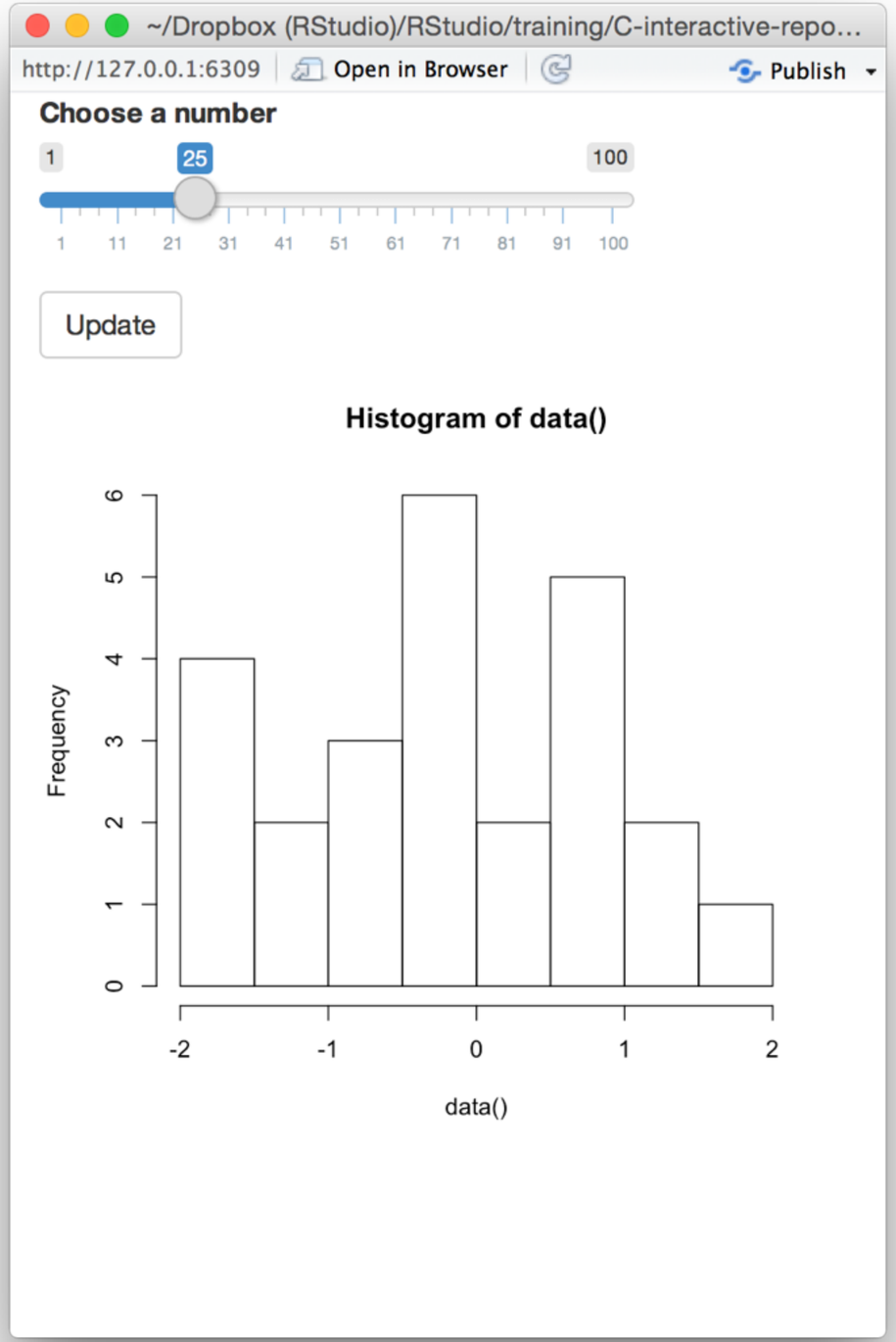
code used to build (and rebuild) object

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

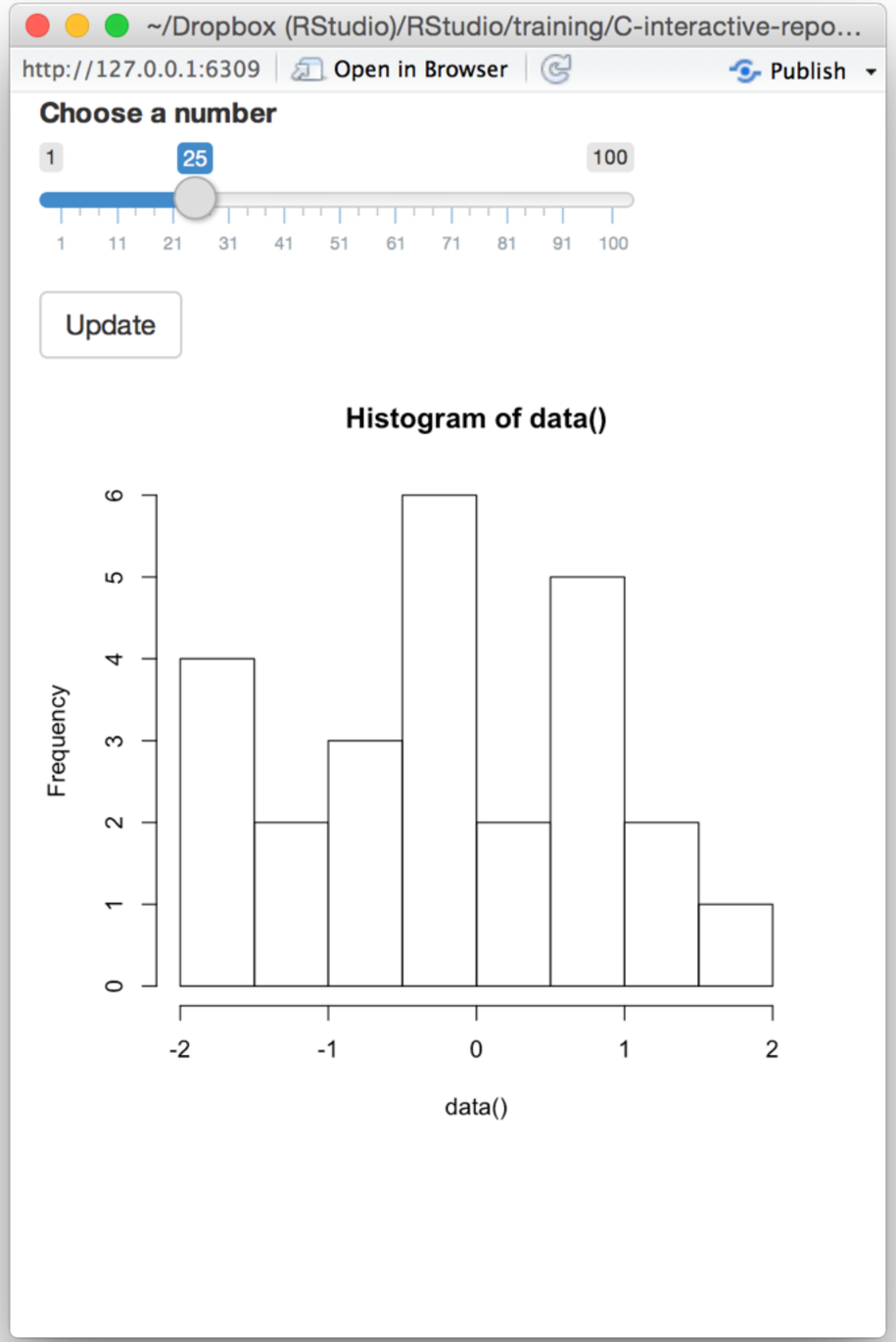
```
# 07-eventReactive

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)
```



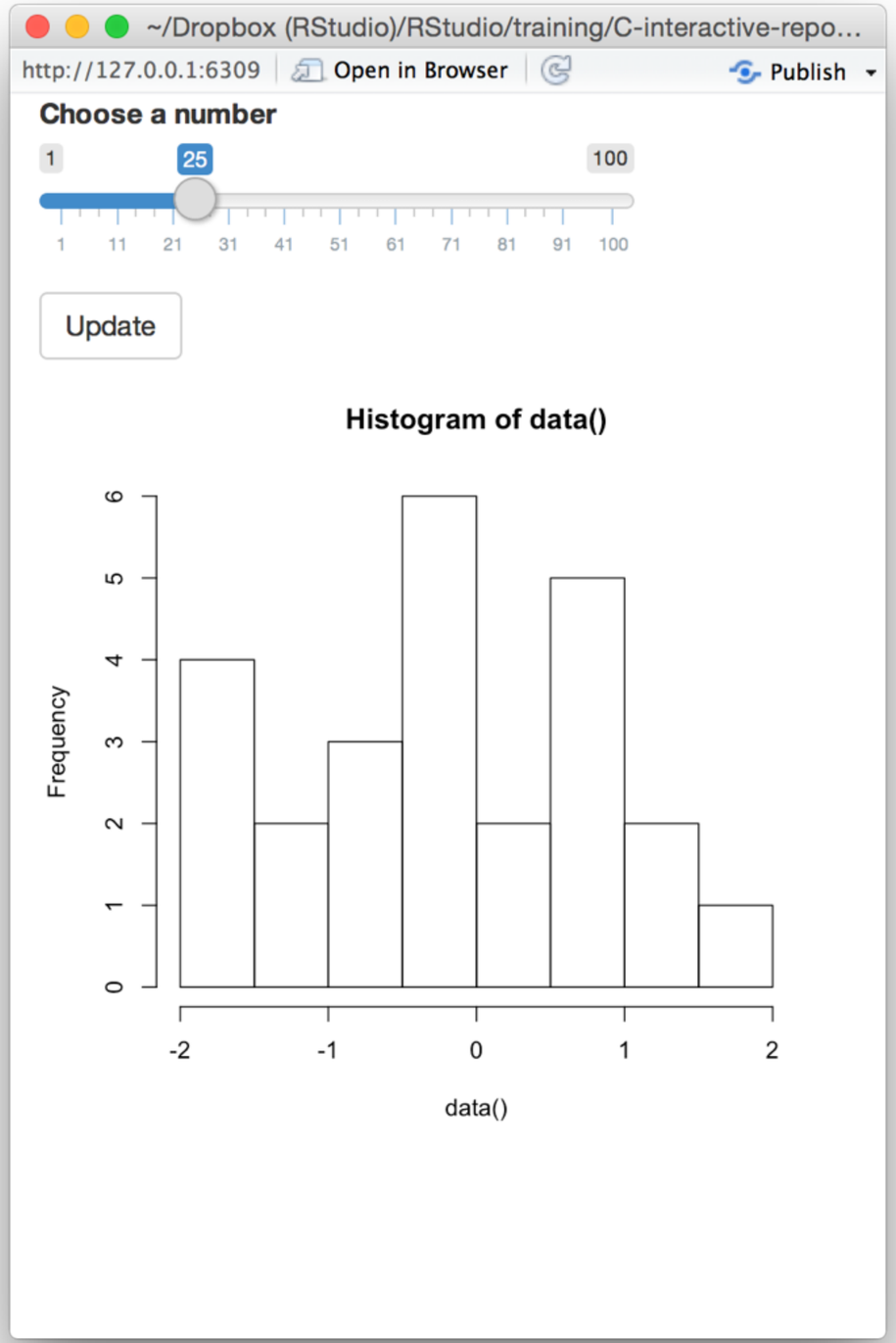
```r
# 07-eventReactive

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))
  })
}

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



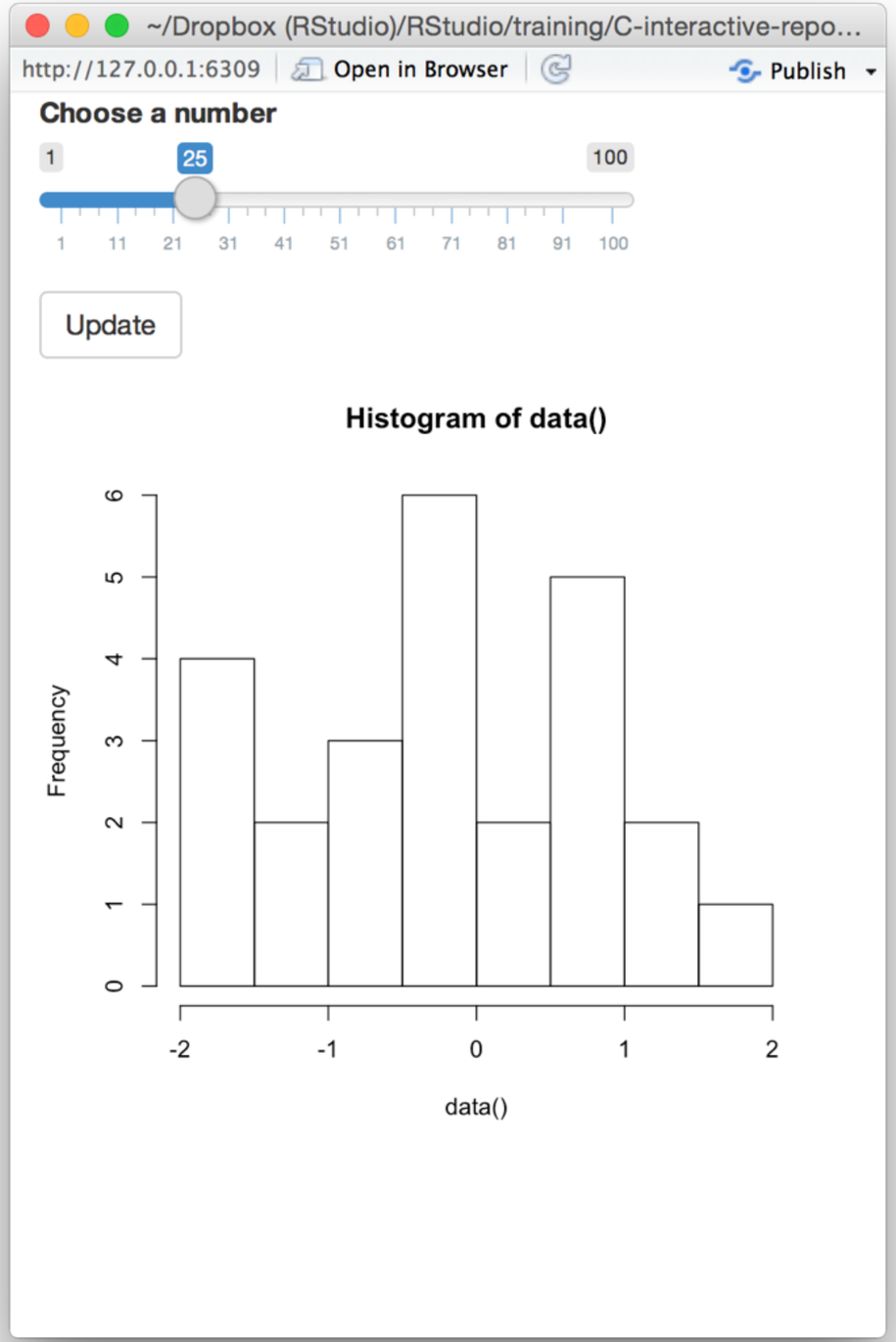
```
# 07-eventReactive

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))
  })
}

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$hist <- renderPlot({
    hist(rnorm(input$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)
```

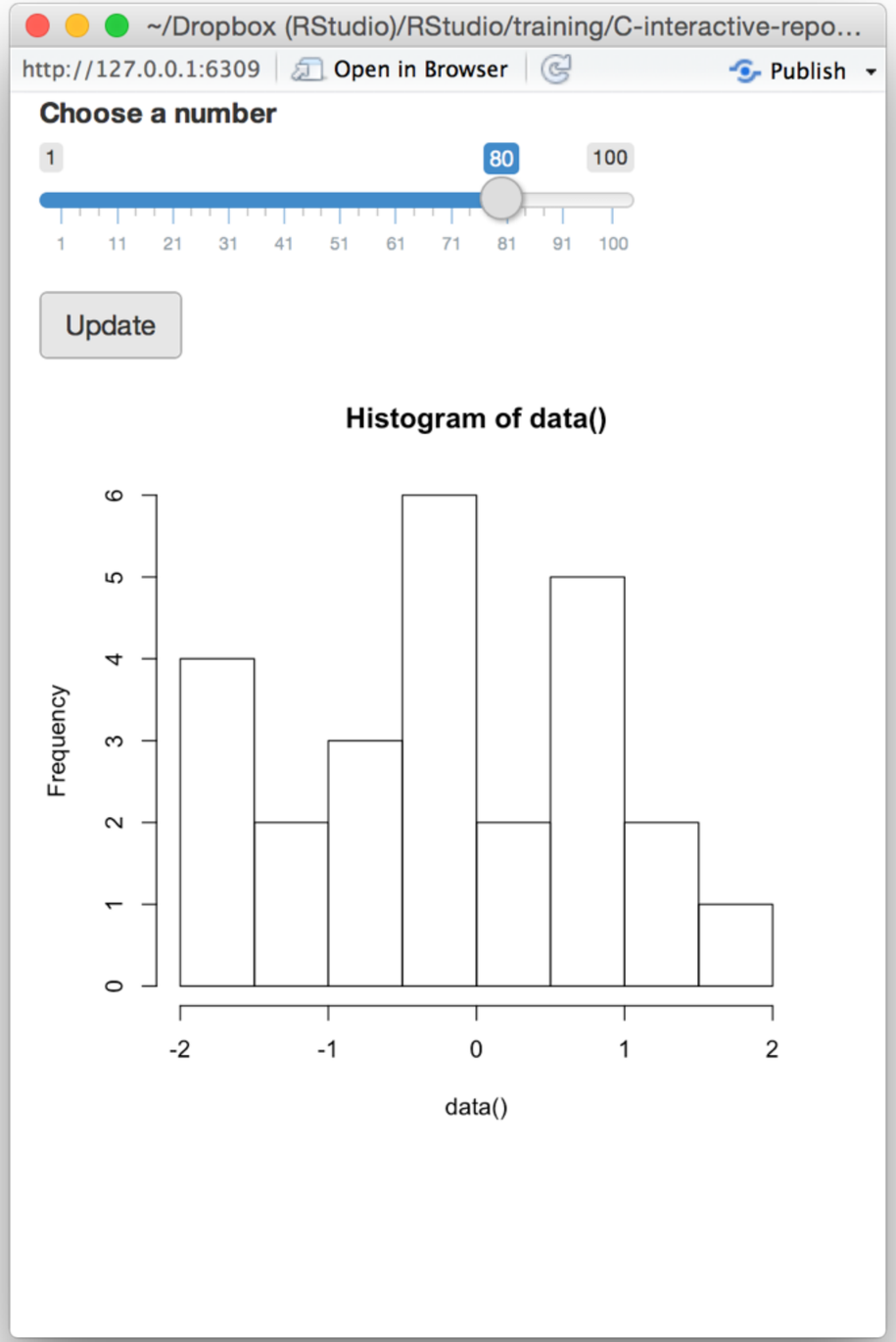


input$num

input$go

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

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

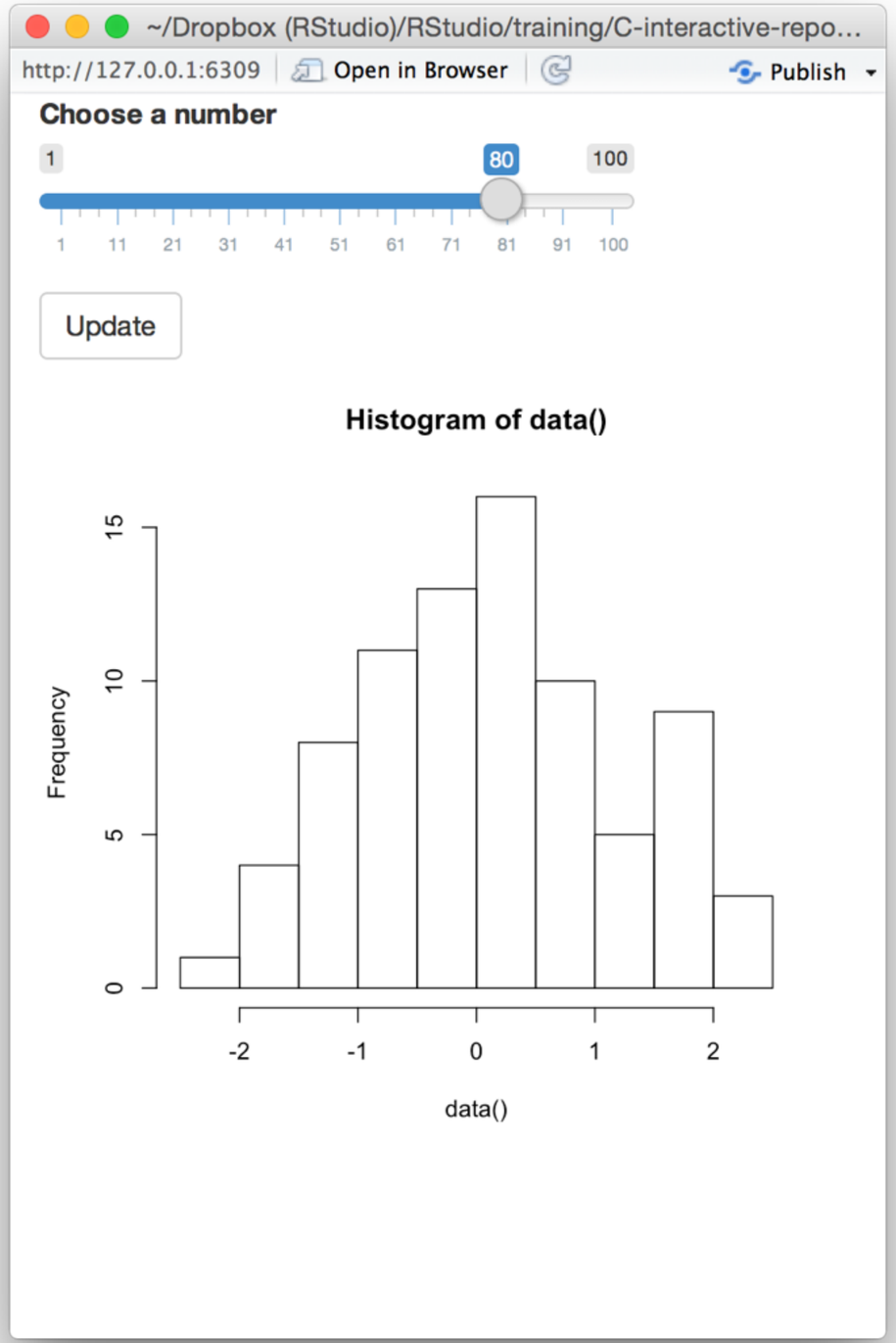


input$num

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**

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

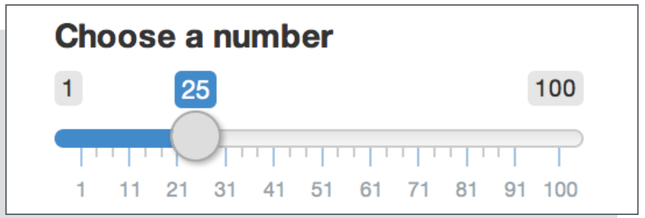
reactive value(s) to respond to

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

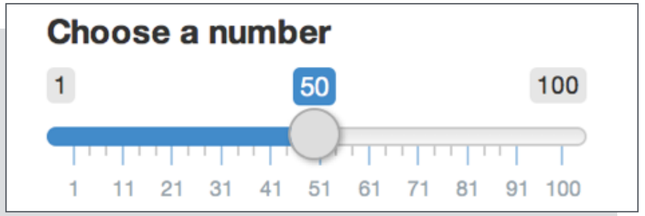
Manage state
with reactiveValues()

Input values

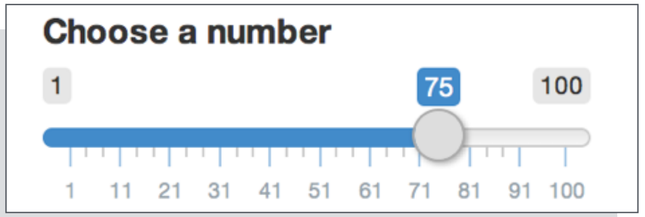
The input value changes whenever a user changes the input.




```
input$num = 25
```




```
input$num = 50
```




```
input$num = 75
```

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

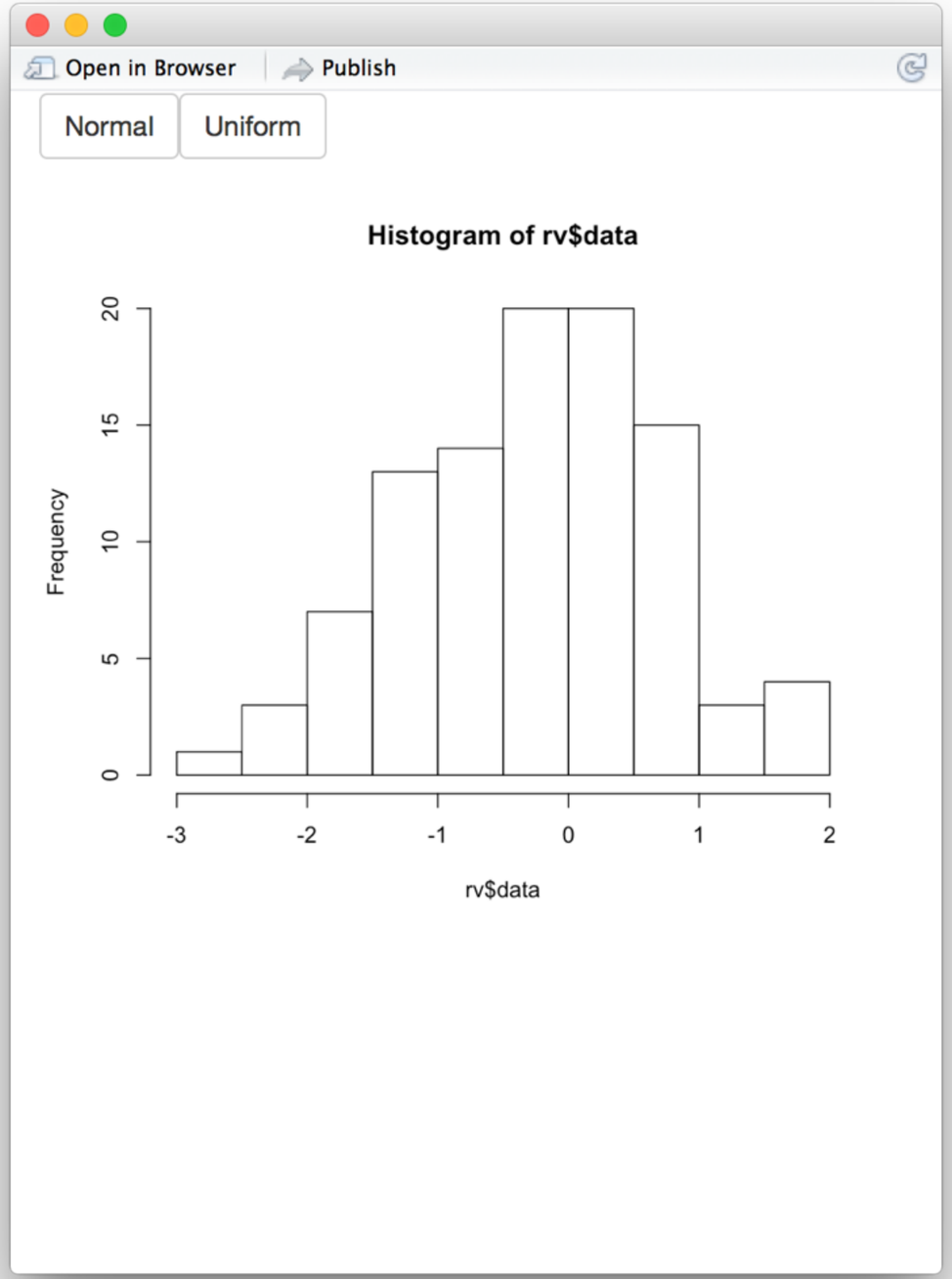
reactiveValues()

Creates a list of reactive values to manipulate programmatically

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

(optional) **elements** to add to the list

```
# 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)
```

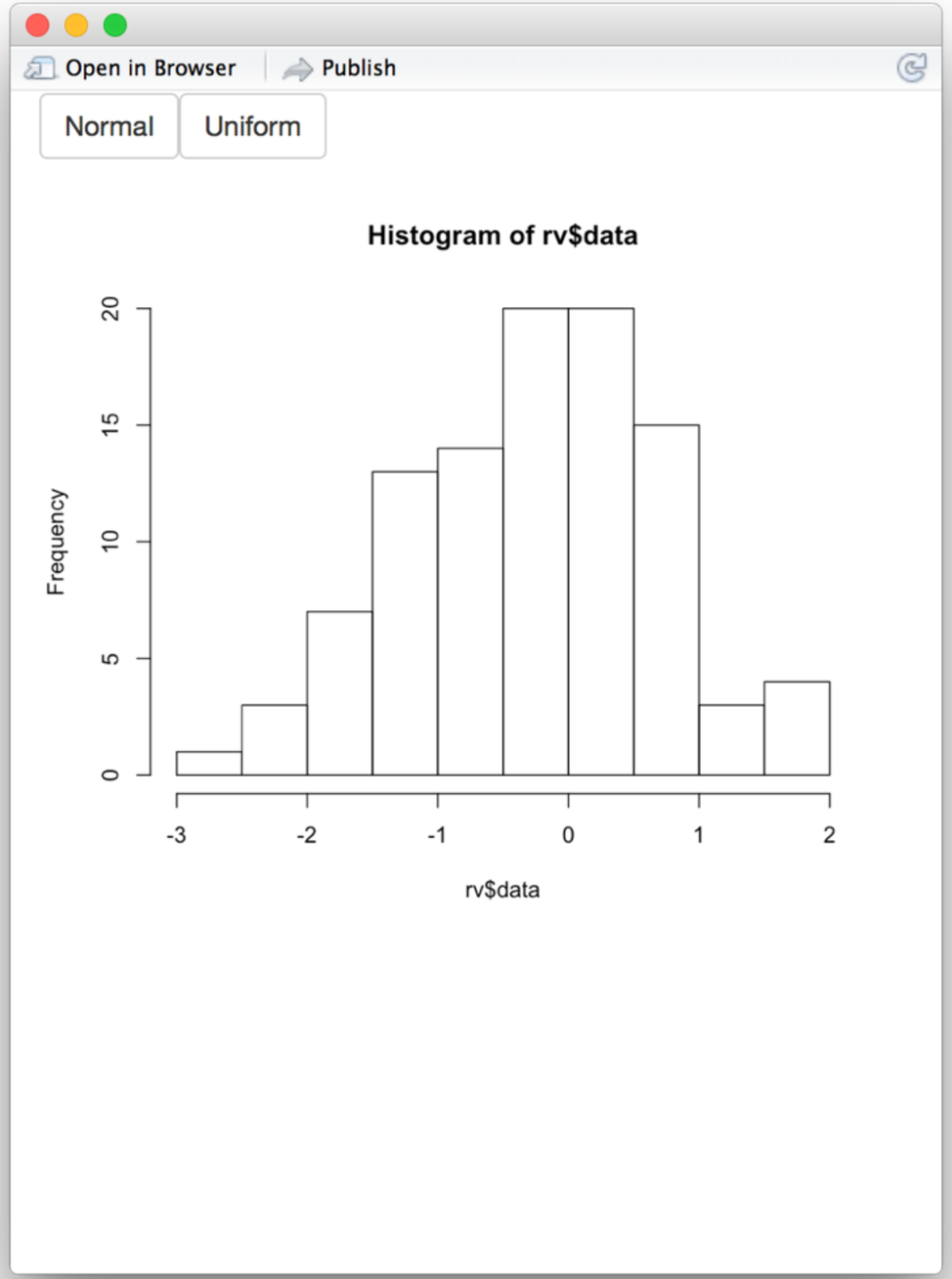


input$norm

rv$data
rnorm(100)

input$unif

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

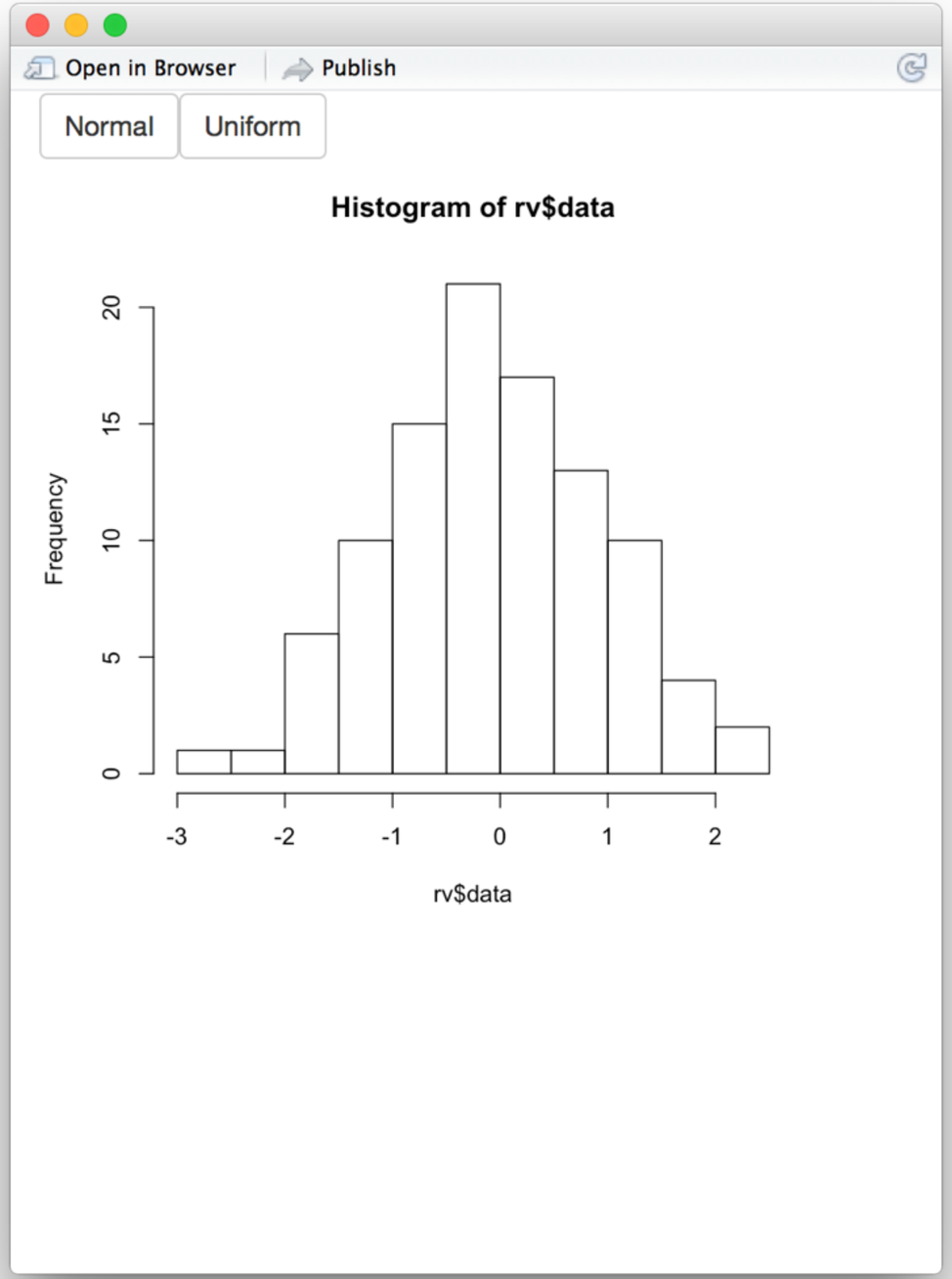


input$norm

rv$data
rnorm(100)

input$unif

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

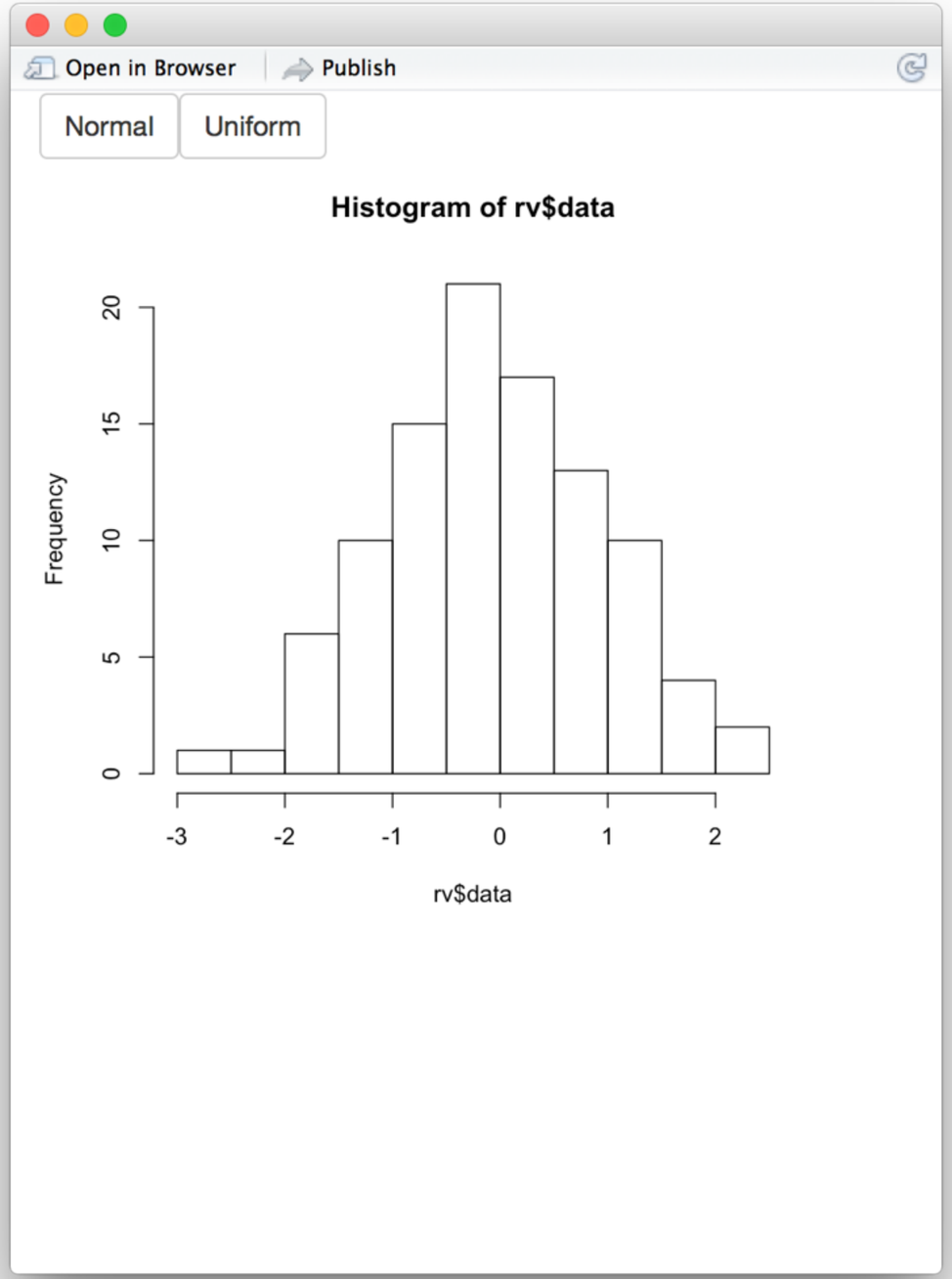


input$norm

rv$data
runif(100)

input$unif

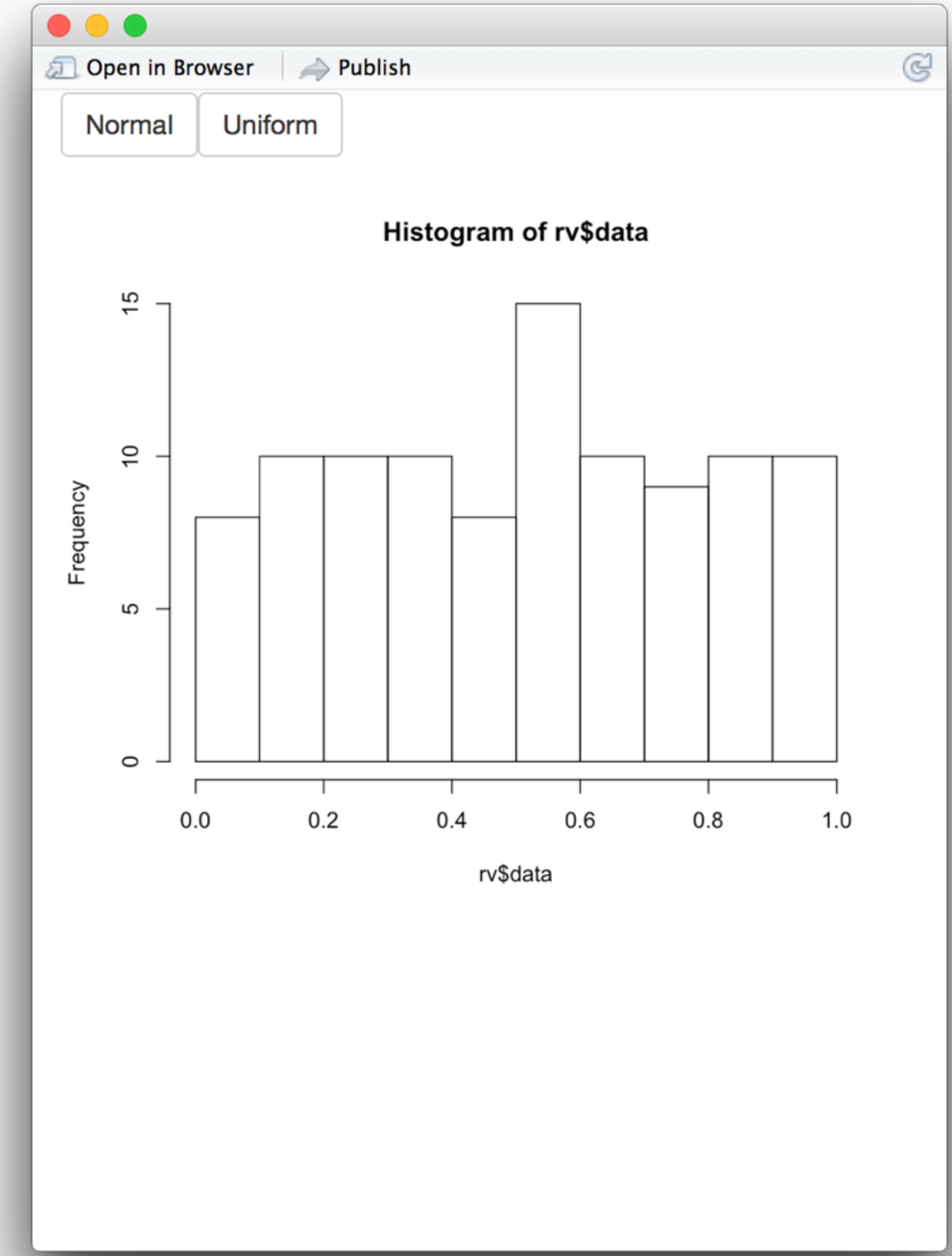
```
output$hist <-
  renderPlot({
    hist(rv$data)
})
```



input$norm

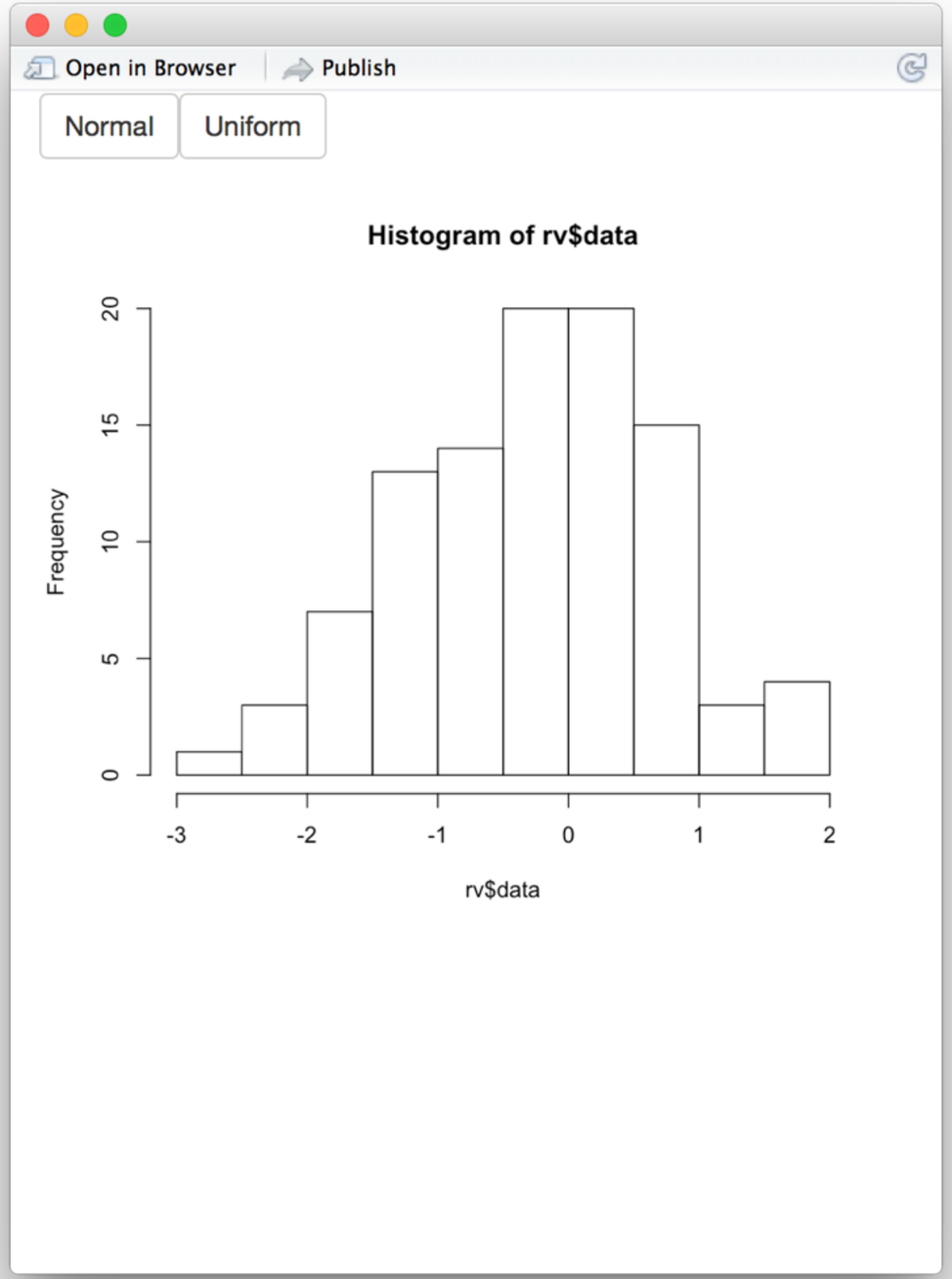
rv$data
runif(100)

input$unif

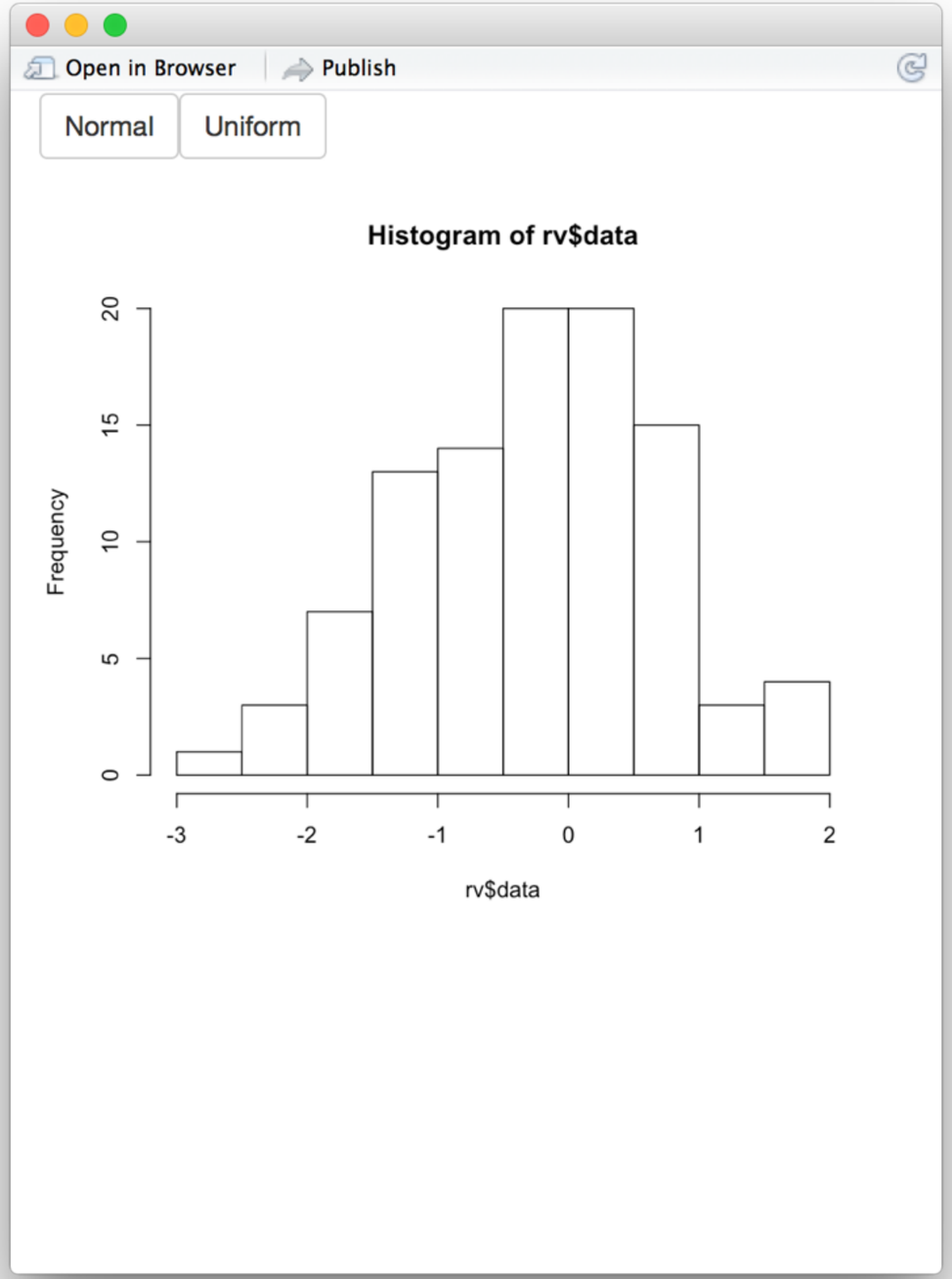



```
output$hist <-
  renderPlot({
    hist(rv$data)
})
```

```
# 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)
```



```
# 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)
```



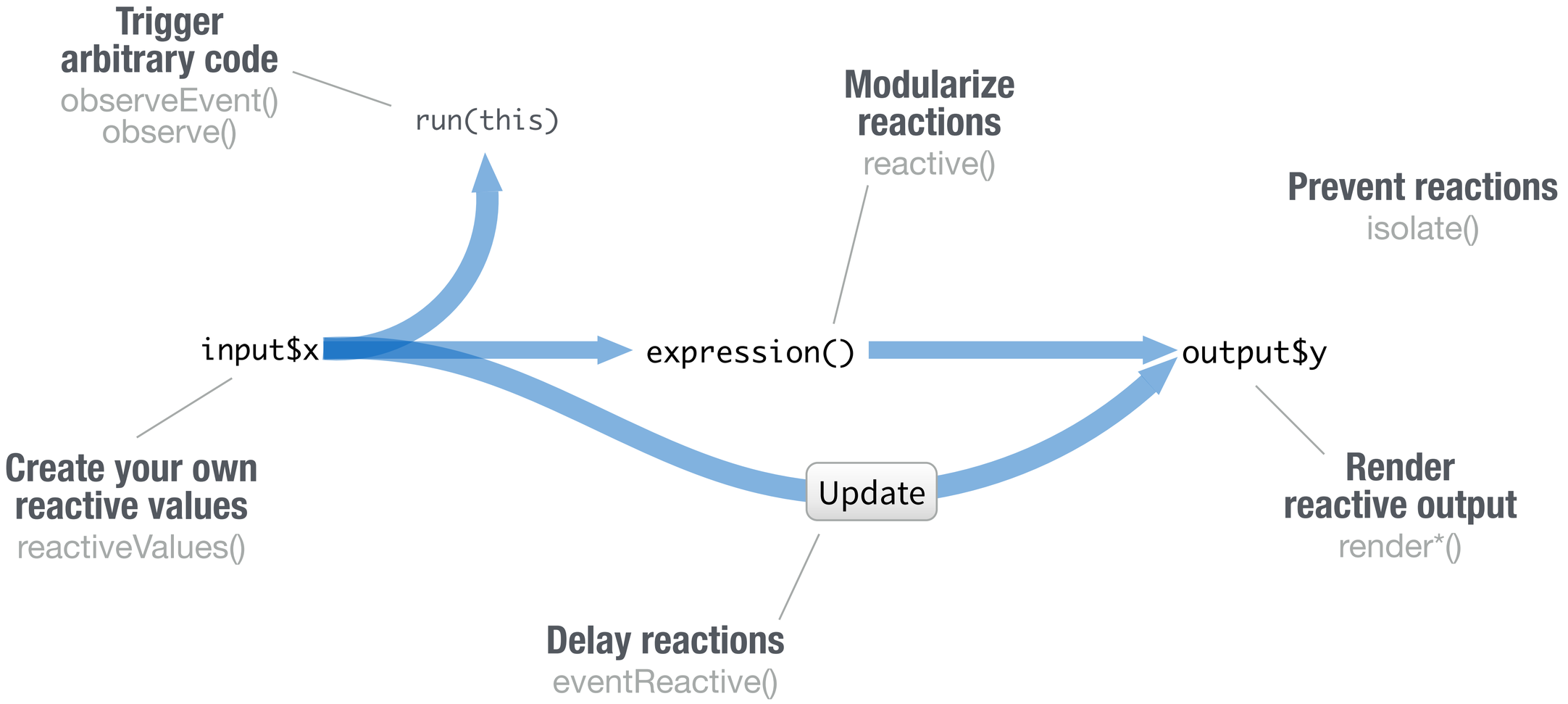
Recap: reactiveValues()

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

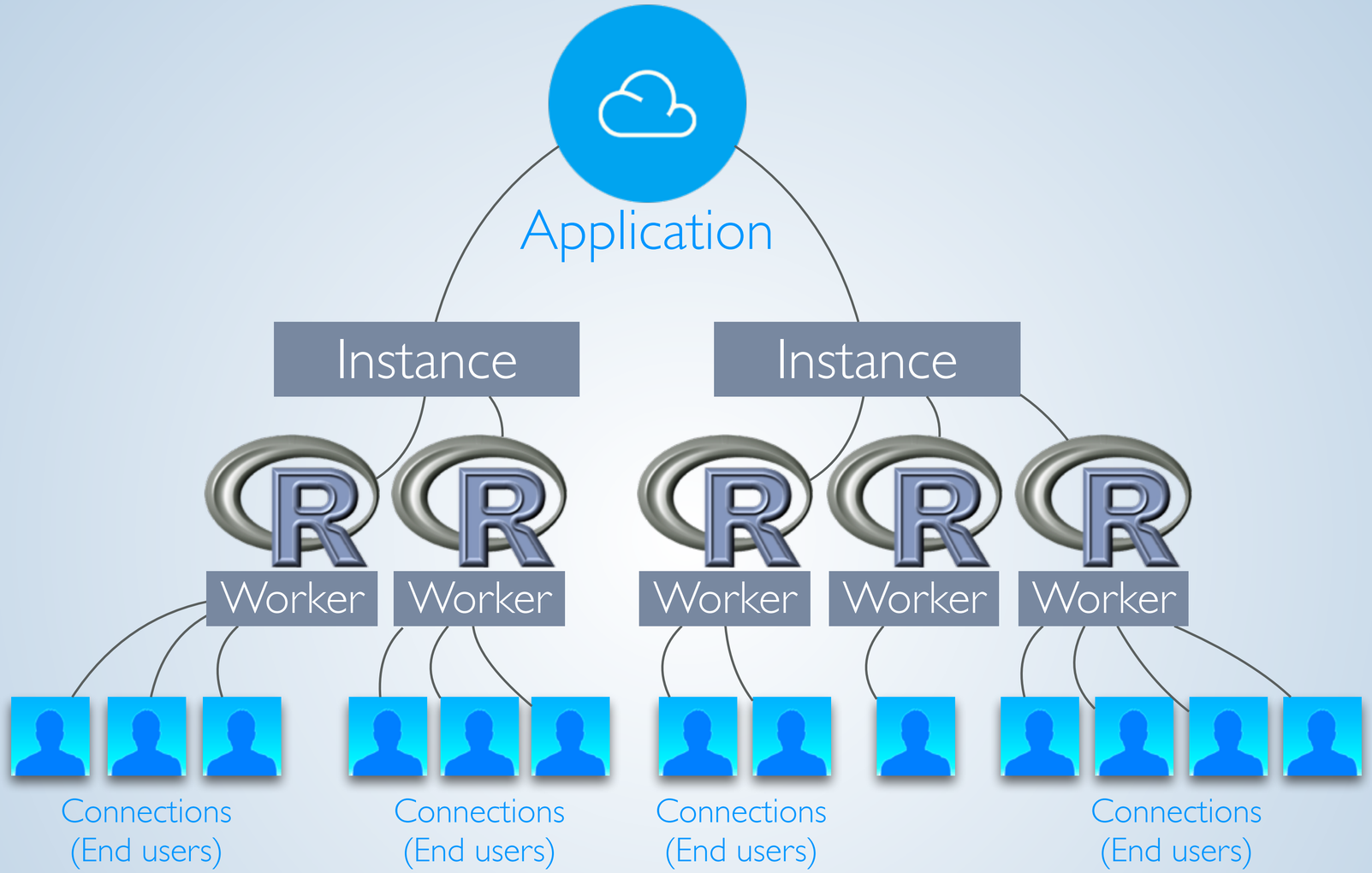
```
rv$data <-
```

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

You now how to



Parting tips


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)
```

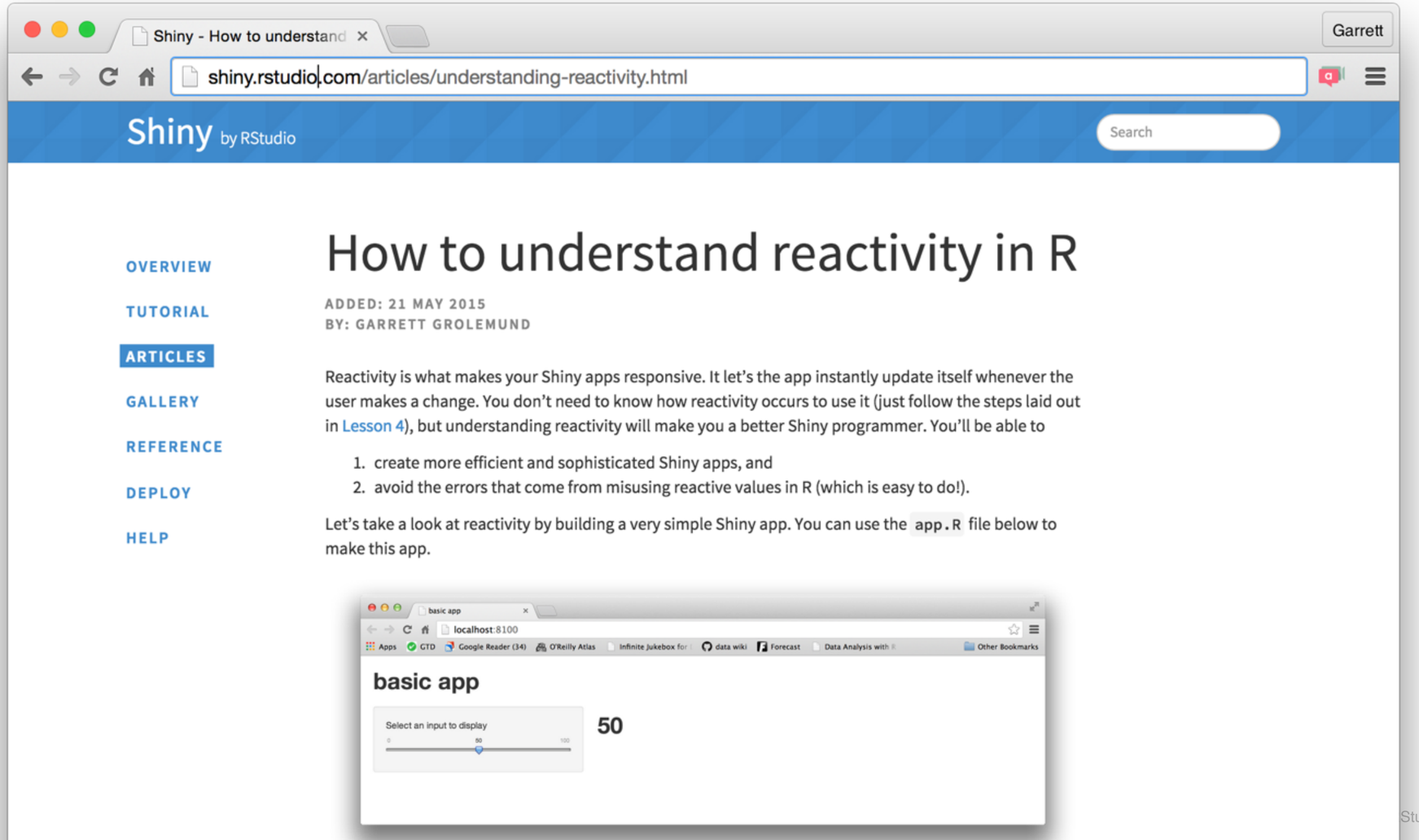
Code outside the server function will be run once per R session (worker)

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)

How can R possibly implement reactivity?

http://shiny.rstudio.com/articles/understanding-reactivity.html



Learn more

How to start with Shiny

1. How to build a Shiny app (www.rstudio.com/resources/webinars/)

2. How to customize reactions (Today)

3. How to customize appearance (June 3)

The Shiny Development Center

shiny.rstudio.com