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

Introduction to Shiny

How to build an interactive app

Garrett Grolemund

Data Scientist and Professional Educator
April 2016
Email: garrett@rstudio.com

Shiny Showcase

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

RStudio Products Resources Pricing About Us Blog

Shiny Apps for the Enterprise

Shiny Dashboard Demo

A dashboard built with Shiny.

Location tracker

Track locations over time with streaming data.

Download monitor

Streaming download rates visualized as a bubble chart.

Supply and Demand

Forecast demand to plan resource allocation.

Industry Specific Shiny Apps

Economic Dashboard

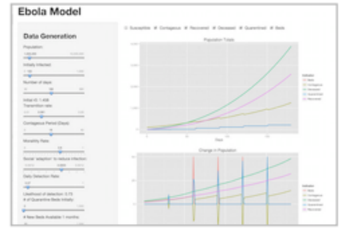
Economic forecasting with macroeconomic indicators.

ER Optimization

An app that models patient flow.

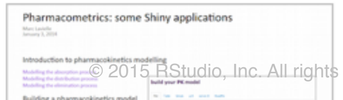
CDC Disease Monitor

Alert thresholds and automatic weekly updates.



Ebola Model

An epidemiological simulation.



Outline

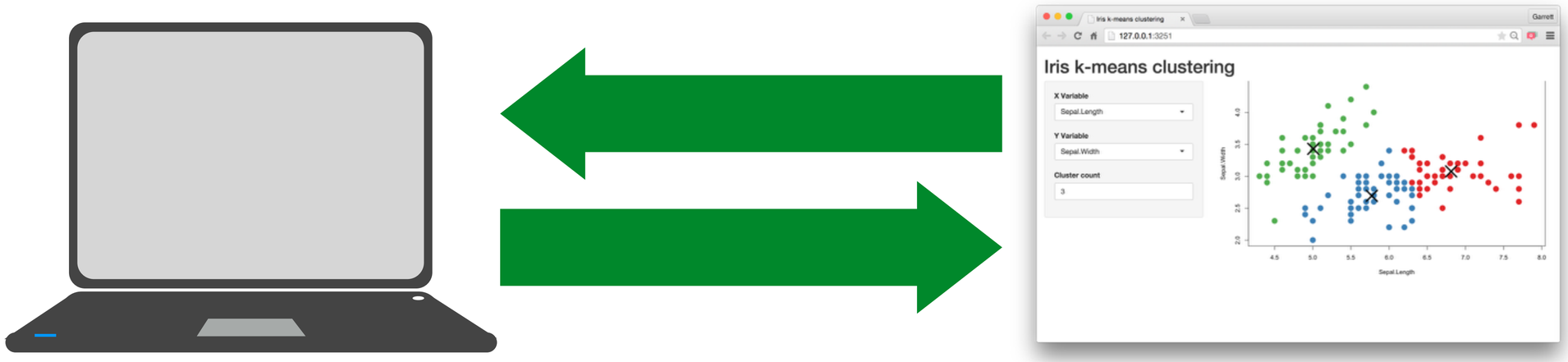
1. **How to build** a Shiny App
2. **Reactive Programming**
3. The **User Interface** of your apps

How to build an app

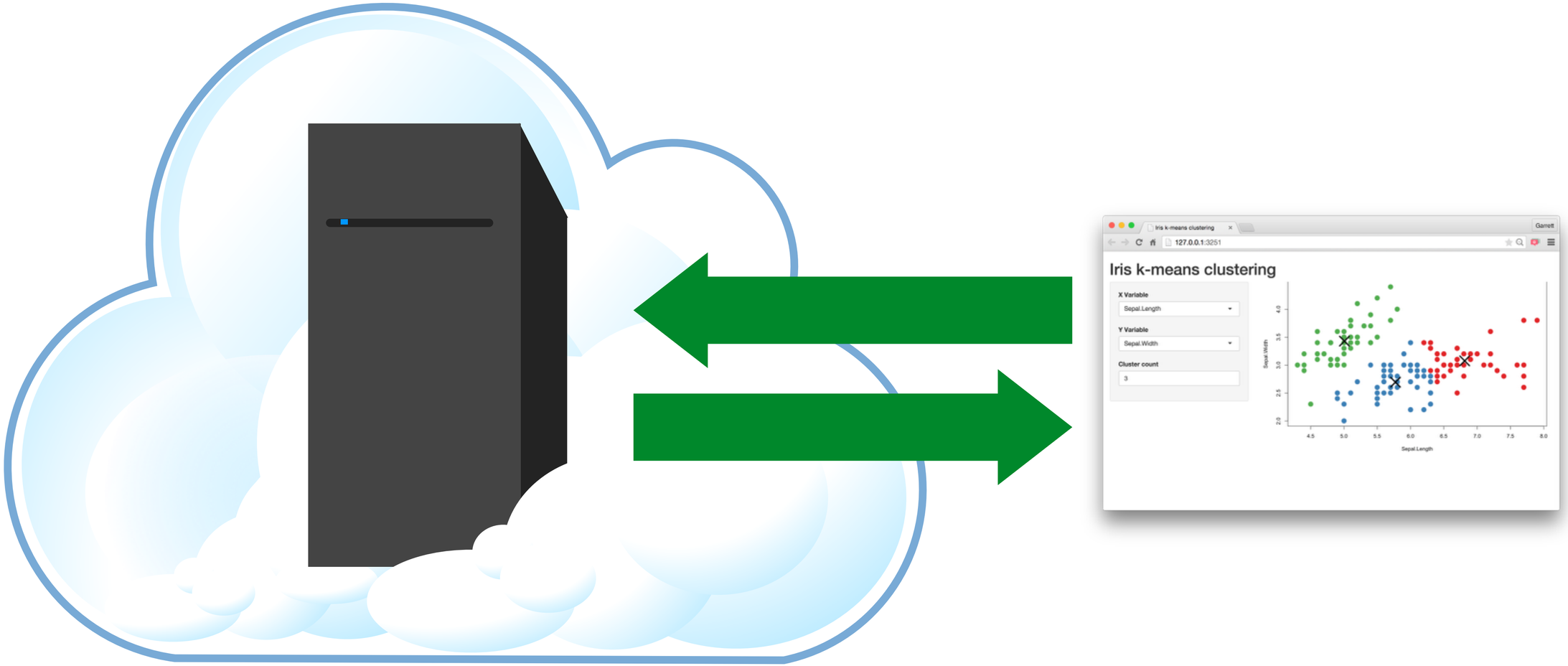
Outline

1. Components of an app
    a. Inputs and Outputs
    b. Server function
2. File Structure
3. Sharing

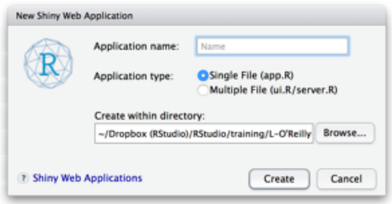
Every Shiny app is maintained by a computer running R



Every Shiny app is maintained by a computer running R



Recap



Open a new Shiny app with
File ▶ New File ▶ Shiny Web App…

Launch the app by opening app.R and clicking **Run App**

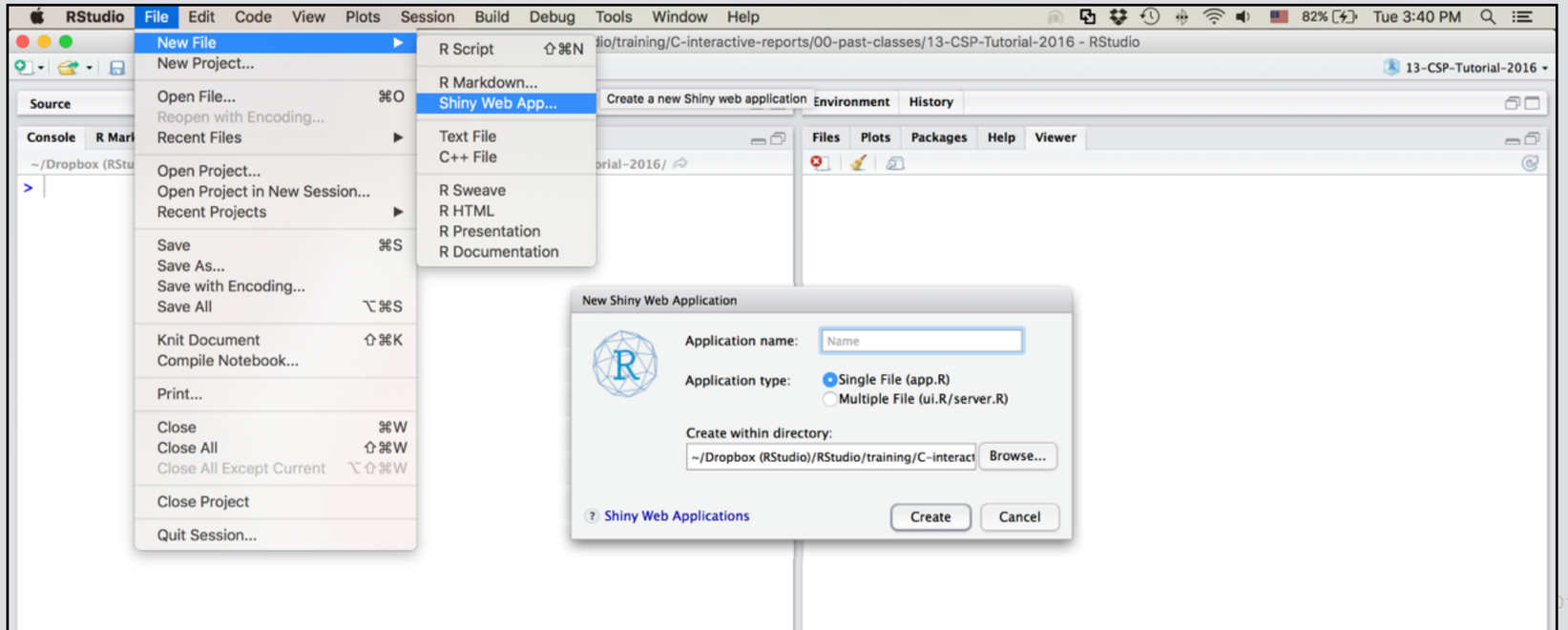
Close app by clicking the stop sign icon

Select view mode in the drop down menu next to Run App

Warm up

1. Go to File ▶ New File ▶ Shiny Web App ▶ Single File

2. Click Run App at the top of the file to launch your first Shiny App.



Components of an app

Server computer
running R and Shiny

A Web browser

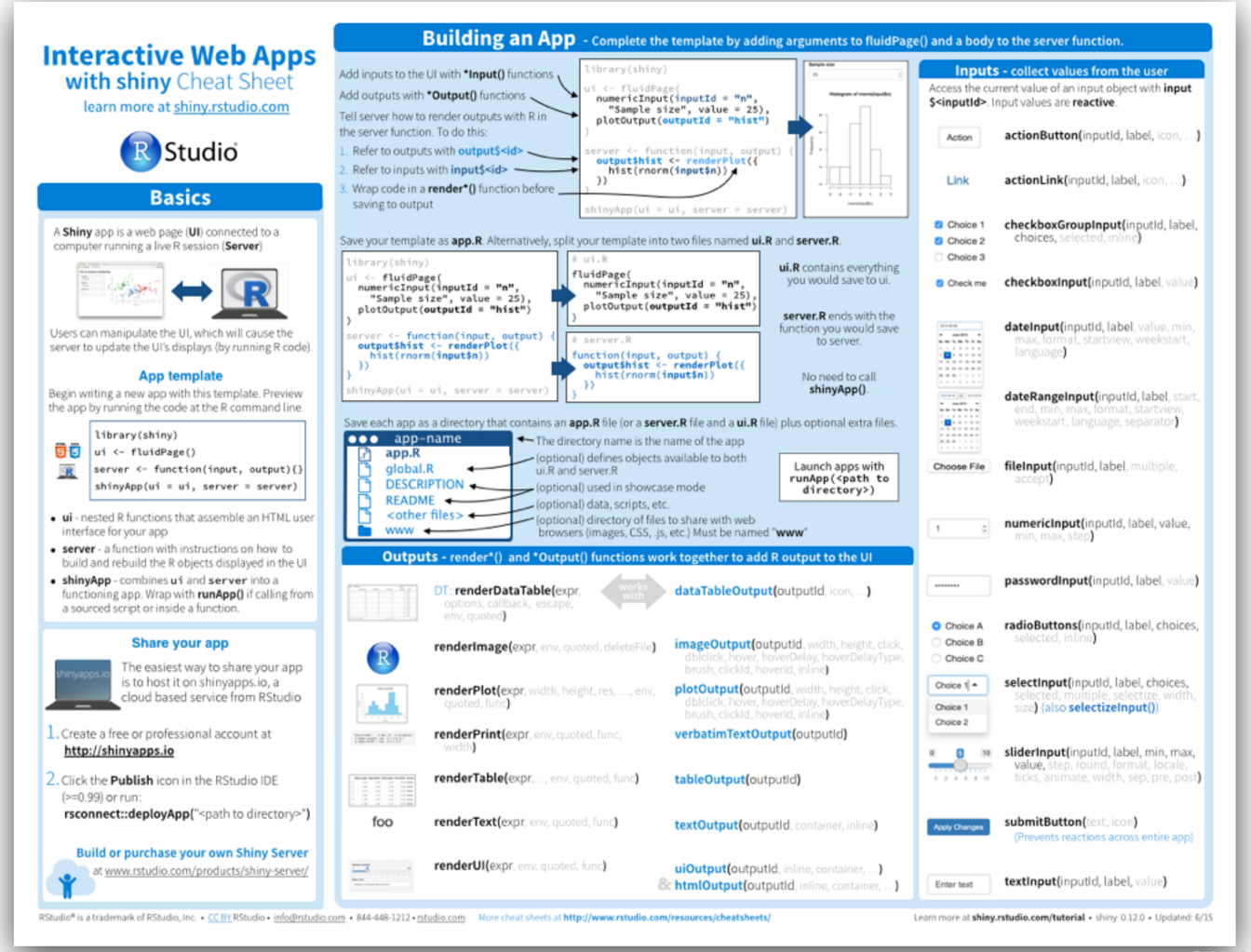
App template

The shortest viable shiny app

```
library(shiny)

ui <- fluidPage()


server <- function(input, output) {}


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

The Shiny Cheat Sheet

www.rstudio.com/resources/cheatsheets/



Interactive Web Apps with shiny Cheat Sheet

learn more at shiny.rstudio.com

RStudio

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)

Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

App template

Begin writing a new app with this template. Preview the app by running the code at the R command line.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines **ui** and **server** into a functioning app. Wrap with **runApp()** if calling from a sourced script or inside a function.

Share your app

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE (>=0.99) or run:
   rsconnect::deployApp("<path to directory>")

Build or purchase your own Shiny Server at www.rstudio.com/products/shiny-server/

Building an App - Complete the template by adding arguments to fluidPage() and a body to the server function.

Add inputs to the UI with ***Input()** functions

Add outputs with ***Output()** functions

Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

```
# server.R
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
```

ui.R contains everything you would save to ui.

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that contains an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.

app-name
- app.R — The directory name is the name of the app
- global.R — (optional) defines objects available to both ui.R and server.R
- DESCRIPTION — (optional) used in showcase mode
- README — (optional) data, scripts, etc.
- <other files> — (optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"
- www

Launch apps with **runApp(<path to directory>)**

Outputs - render*() and *Output() functions work together to add R output to the UI

	works with
DT::**renderDataTable**(expr, options, callback, escape, env, quoted)	**dataTableOutput**(outputId, icon, ...)
renderImage(expr, env, quoted, deleteFile)	**imageOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)
renderPlot(expr, width, height, res, ..., env, quoted, func)	**plotOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)
renderPrint(expr, env, quoted, func, width)	**verbatimTextOutput**(outputId)
renderTable(expr, ..., env, quoted, func)	**tableOutput**(outputId)
renderText(expr, env, quoted, func)	**textOutput**(outputId, container, inline)
renderUI(expr, env, quoted, func)	**uiOutput**(outputId, inline, container, ...) & **htmlOutput**(outputId, inline, container, ...)

Inputs - collect values from the user

Access the current value of an input object with **input $<inputId>**. Input values are **reactive**.

Action	**actionButton**(inputId, label, icon, ...)
Link	**actionLink**(inputId, label, icon, ...)
Choice 1, Choice 2, Choice 3	**checkboxGroupInput**(inputId, label, choices, selected, inline)
Check me	**checkboxInput**(inputId, label, value)
	dateInput(inputId, label, value, min, max, format, startview, weekstart, language)
	dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)
Choose File	**fileInput**(inputId, label, multiple, accept)
1	**numericInput**(inputId, label, value, min, max, step)
•••••••	**passwordInput**(inputId, label, value)
Choice A, Choice B, Choice C	**radioButtons**(inputId, label, choices, selected, inline)
Choice 1, Choice 1, Choice 2	**selectInput**(inputId, label, choices, selected, multiple, selectize, width, size) (also **selectizeInput()**)
0, 5, 10	**sliderInput**(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)
Apply Changes	**submitButton**(text, icon) (Prevents reactions across entire app)
Enter text	**textInput**(inputId, label, value)

Your Turn

Trim your app down to the template.

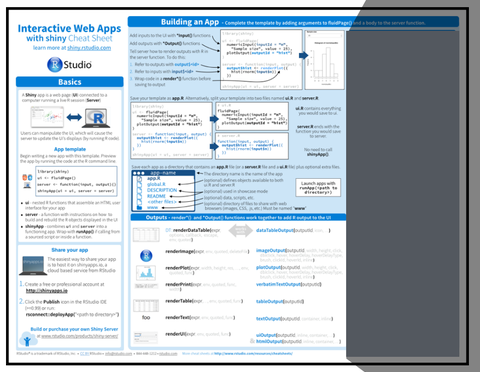
Then run the app.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

Build your app around

Inputs...

Inputs



Action — **actionButton**(inputId, label, icon, …)

Link — **actionLink**(inputId, label, icon, …)

Choice 1
Choice 2
Choice 3 — **checkboxGroupInput**(inputId, label, choices, selected, inline)

Check me — **checkboxInput**(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)

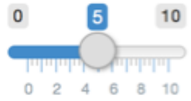
Choose File — **fileInput**(inputId, label, multiple, accept)

1 — **numericInput**(inputId, label, value, min, max, step)

•••••••• — **passwordInput**(inputId, label, value)

Choice A
Choice B
Choice C — **radioButtons**(inputId, label, choices, selected, inline)

Choice 1
Choice 1
Choice 2 — **selectInput**(inputId, label, choices, selected, multiple, selectize, width, size) (also **selectizeInput()**)

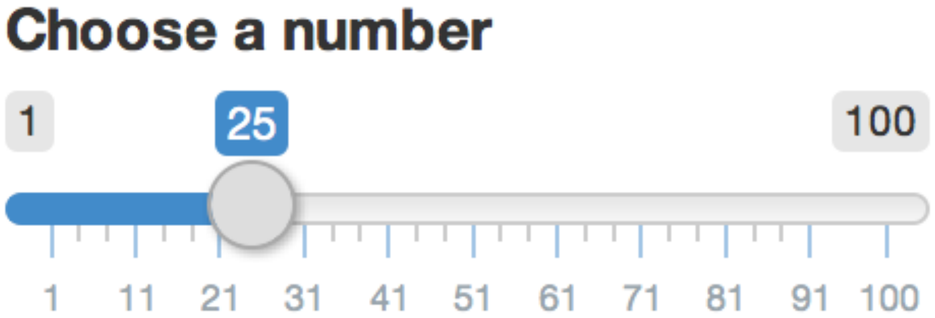


sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)

Apply Changes — **submitButton**(text, icon)
(Prevents reactions across entire app)

Enter text — **textInput**(inputId, label, value)

Syntax




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

input name
(for internal use)

Notice:
Id not ID

label to display

input specific arguments

```
?sliderInput
```

Your Turn

Add to your app:

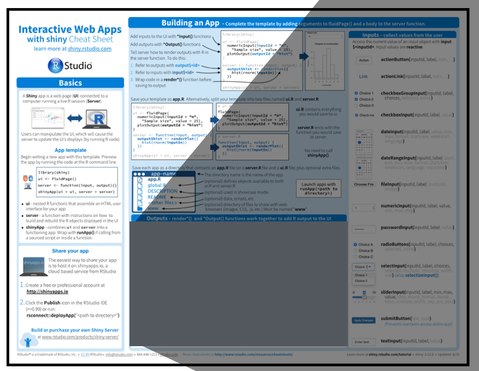
1. A slider that goes from 1 to 100

Then run the app. What happens?

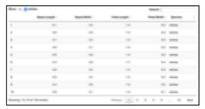
. . . and Outputs

Outputs

display output from R.



Outputs - render*() and *Output() functions work together to add R output to the UI



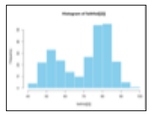
DT::**renderDataTable**(expr, options, callback, escape, env, quoted)

works with

dataTableOutput(outputId, icon, …)

renderImage(expr, env, quoted, deleteFile)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)



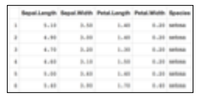
renderPlot(expr, width, height, res, …, env, quoted, func)

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

```
'data.frame':  3 obs. of  2 variables:
 $ Sepal.Length: num  5.1 4.9 4.7
 $ Sepal.Width : num  3.5 3 3.2
```

renderPrint(expr, env, quoted, func, width)

verbatimTextOutput(outputId)



renderTable(expr,…, env, quoted, func)

tableOutput(outputId)

foo

renderText(expr, env, quoted, func)

textOutput(outputId, container, inline)

renderUI(expr, env, quoted, func)

uiOutput(outputId, inline, container, …)
& **htmlOutput**(outputId, inline, container, …)

*Output()

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

```
plotOutput(outputId = "plot")
```

the type of output to display

name to give to the output object

Your Turn

Add to your app:

1. A slider that goes from 1 to 100
2. A plot named "hist"

Then run the app. What happens?

Tell the

server

how to assemble
inputs into outputs

Outputs

display output from R.

```
library(shiny)
ui <- fluidPage(
  sliderInput("num", "", 1, 100, 25),
  plotOutput("hist")
)
server <- function(input, output) {}




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

Build outputs in 3 steps:

1. Add a *Output() function to ui (places output)

Outputs

display output from R.

```
library(shiny)
ui <- fluidPage(
  sliderInput("num", "", 1, 100, 25),
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <-



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

Build outputs in 3 steps:

1. Add a *Output() function to ui (places output)

2. Have server save object as output (save to output$)

Match names

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

```
output$hist
```

Outputs

display output from R.

```
library(shiny)
ui <- fluidPage(
  sliderInput("num", "", 1, 100, 25),
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
     hist(iris$Sepal.Length)
  })
}
shinyApp(ui = ui, server = server)
```

Build outputs in 3 steps:

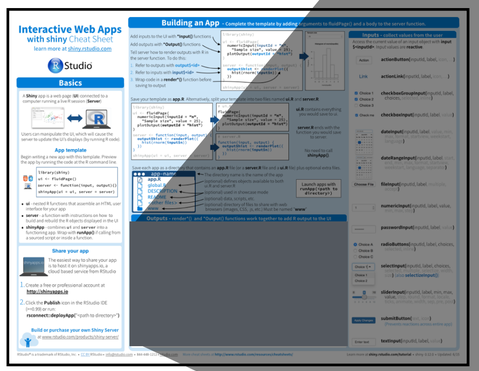
1. Add a *Output() function to ui (places output)

2. Have server save object as output (save to output$)

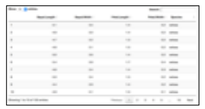
3. Make with render*() function in server (builds output)

Outputs

display output from R.



Outputs - render*() and *Output() functions work together to add R output to the UI



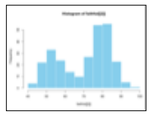
DT::**renderDataTable**(expr, options, callback, escape, env, quoted)

works with

dataTableOutput(outputId, icon, …)

renderImage(expr, env, quoted, deleteFile)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)



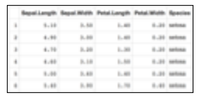
renderPlot(expr, width, height, res, …, env, quoted, func)

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

```
'data.frame':  3 obs. of  2 variables:
 $ Sepal.Length: num  5.1 4.9 4.7
 $ Sepal.Width : num  3.5 3 3.2
```

renderPrint(expr, env, quoted, func, width)

verbatimTextOutput(outputId)



renderTable(expr,…, env, quoted, func)

tableOutput(outputId)

foo

renderText(expr, env, quoted, func)

textOutput(outputId, container, inline)

renderUI(expr, env, quoted, func)

uiOutput(outputId, inline, container, …)
& **htmlOutput**(outputId, inline, container, …)

render*()

Builds reactive output to display in UI

```
renderPlot({ hist(iris$Sepal.Length) })
```

type of object to build

code that builds the object

Your Turn

Add to your app:

1. A slider that goes from 1 to 100
2. A plot that displays the results of

```
hist(rnorm(100))
```

Then run the app.

Create Reactivity

Reactions

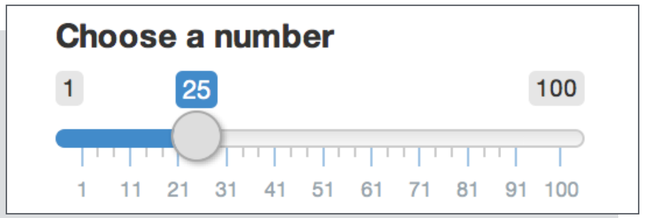
The **input$** list stores the current value of each input object under its name.

```
sliderInput(inputId = "num",…)
```

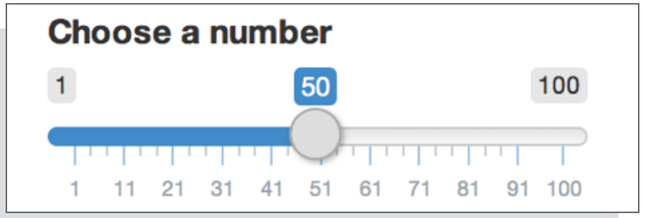
```
input$num
```

Input values

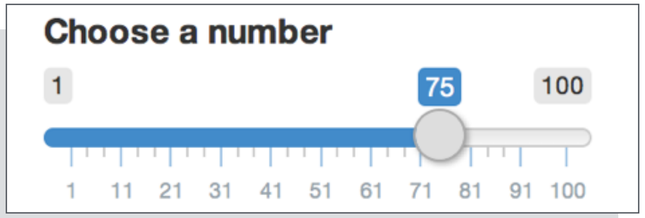
The input value changes whenever a user changes the input.




```
input$num = 25
```




```
input$num = 50
```




```
input$num = 75
```

Reactivity 101

Reactivity automatically occurs whenever you use an input value to render an output object

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

Reactions

```
library(shiny)
ui <- fluidPage(
  sliderInput("num", "", 1, 100, 25),
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
     hist(rnorm(input$num))
  })
}
shinyApp(ui = ui, server = server)
```

Reactivity automatically occurs whenever you use an input value to render an output object

4. An input value

Reactions

```
library(shiny)
ui <- fluidPage(
  sliderInput("num", "", 1, 100, 25),
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
     input$num
     hist(rnorm(100))
  })
}
shinyApp(ui = ui, server = server)
```

Reactivity automatically occurs whenever you use an input value to render an output object

4. An input value

Your Turn

Change your app to plot a histogram of *n* random normal values, where *n* is the value of the slider.

Run the app and ensure that the histogram reacts when you move the slider.

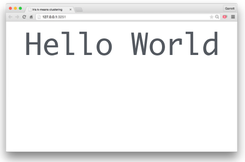
How to build an app

Recap

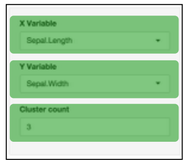
Recap: Shiny

```
library(shiny)
ui <- fluidPage()
server <- function(input, output) {}
shinyApp(ui = ui, server = server)
```

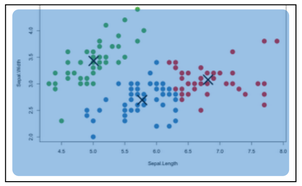
Begin each app with the template



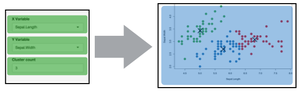
Add elements as arguments to **fluidPage()**



Create reactive inputs with an ***Input()** function



Display R results with an ***Output()** function



Use the server function to assemble inputs into outputs

Recap: Shiny

Use the server function to assemble inputs into outputs. Follow 3 rules:

`output$hist <-`

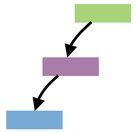
1. Save the output that you build to **output$**

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

2. Build the output with a **render*()** function

`input$num`

3. Access input values with **input$**



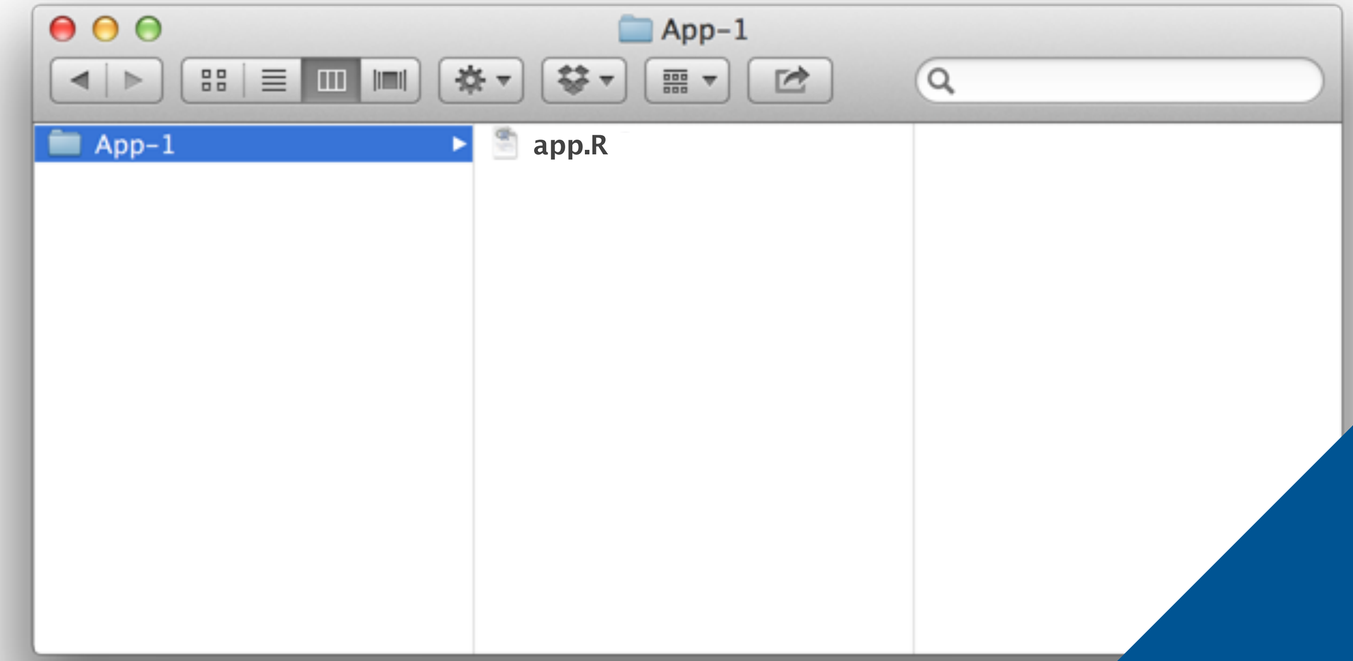
Create reactivity by using **Inputs** to build **rendered Outputs**

File Structure

How to save your app

One directory with every file the app needs:

- app.R *(your script which ends with a call to shinyApp())*
- datasets, images, css, helper scripts, etc.

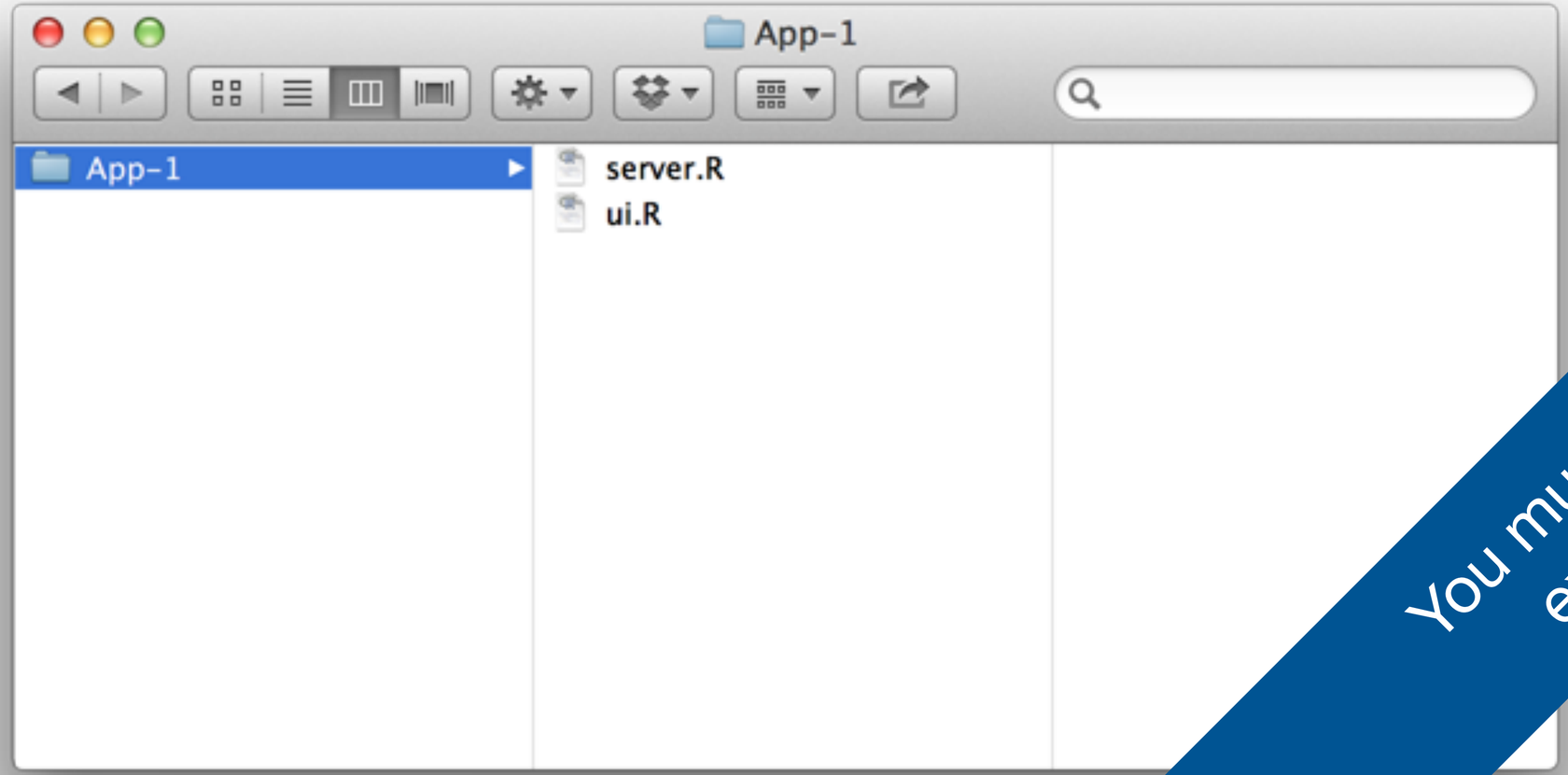


You must use this exact name (**app.R**)

Two file apps

One directory with two files:

- server.R
- ui.R

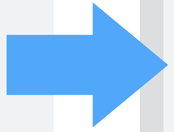


You must use these exact names

Two file apps

```
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)
```

```
# ui.R
library(shiny)
fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)
```



```
# server.R
library(shiny)
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}
```

runApp

You can launch any app from the command line with `runApp`

```
runApp("~/Documents/App-1")
```

File path to app directory.

R will append the file path to the working directory, if path does not begin at the home directory

Share

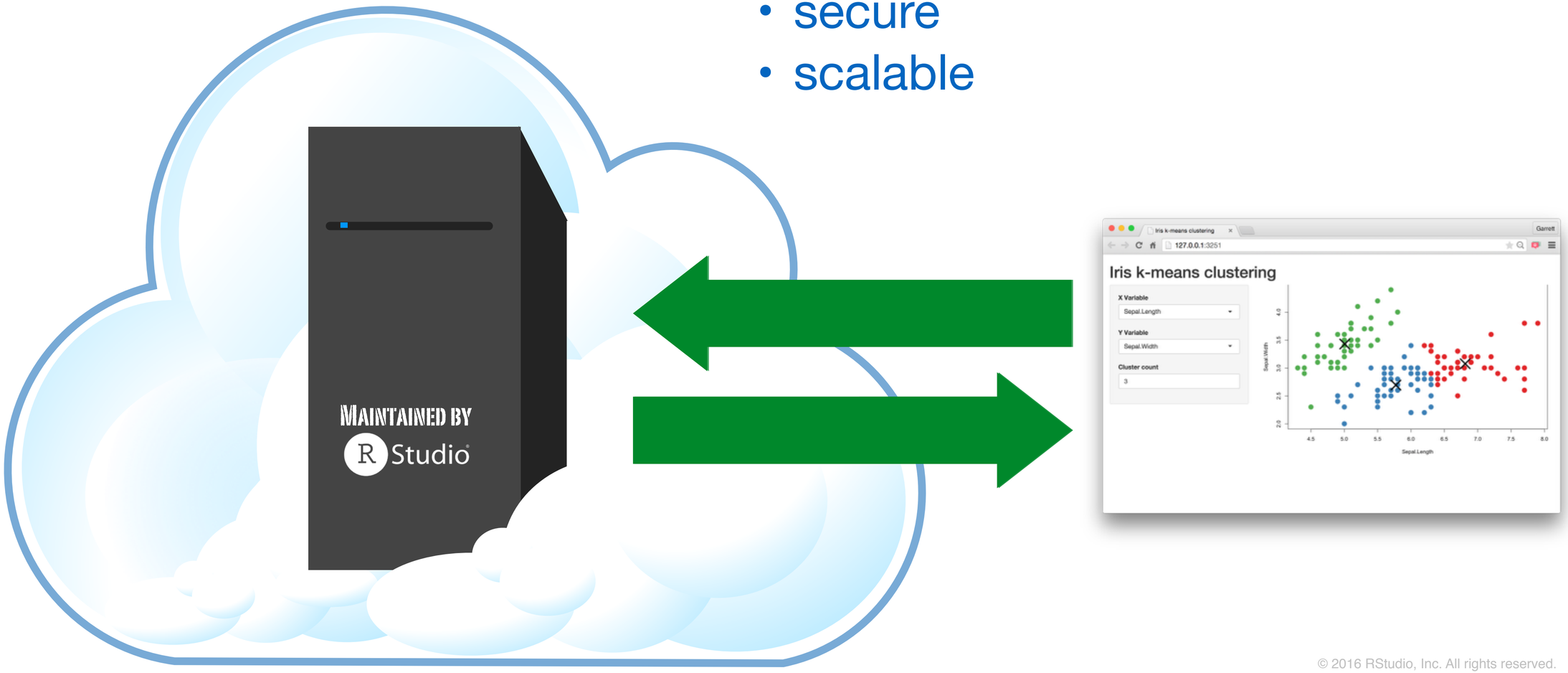
your app

shinyapps.io

Shinyapps.io

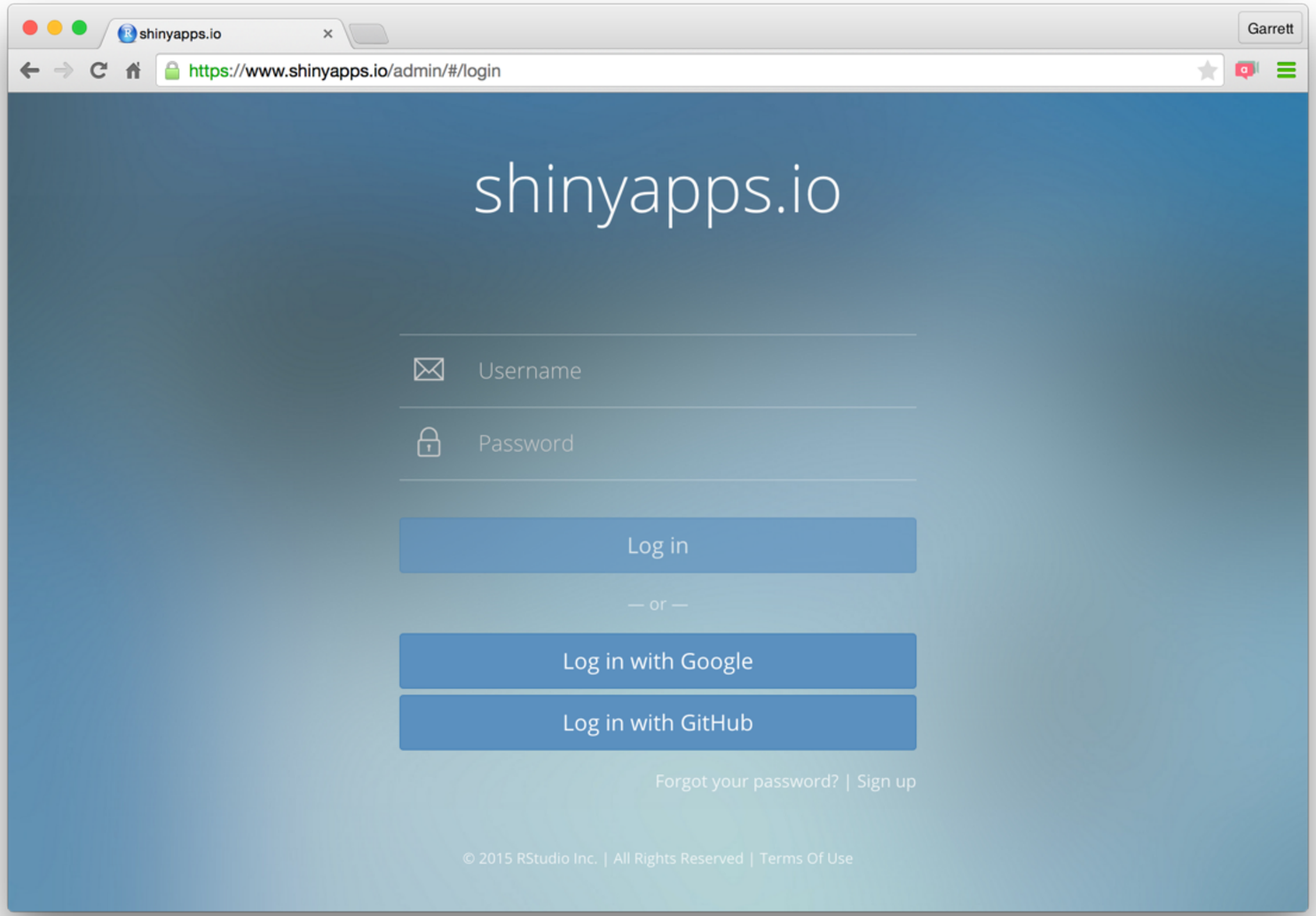
A server maintained by RStudio

- easy to use
- secure
- scalable



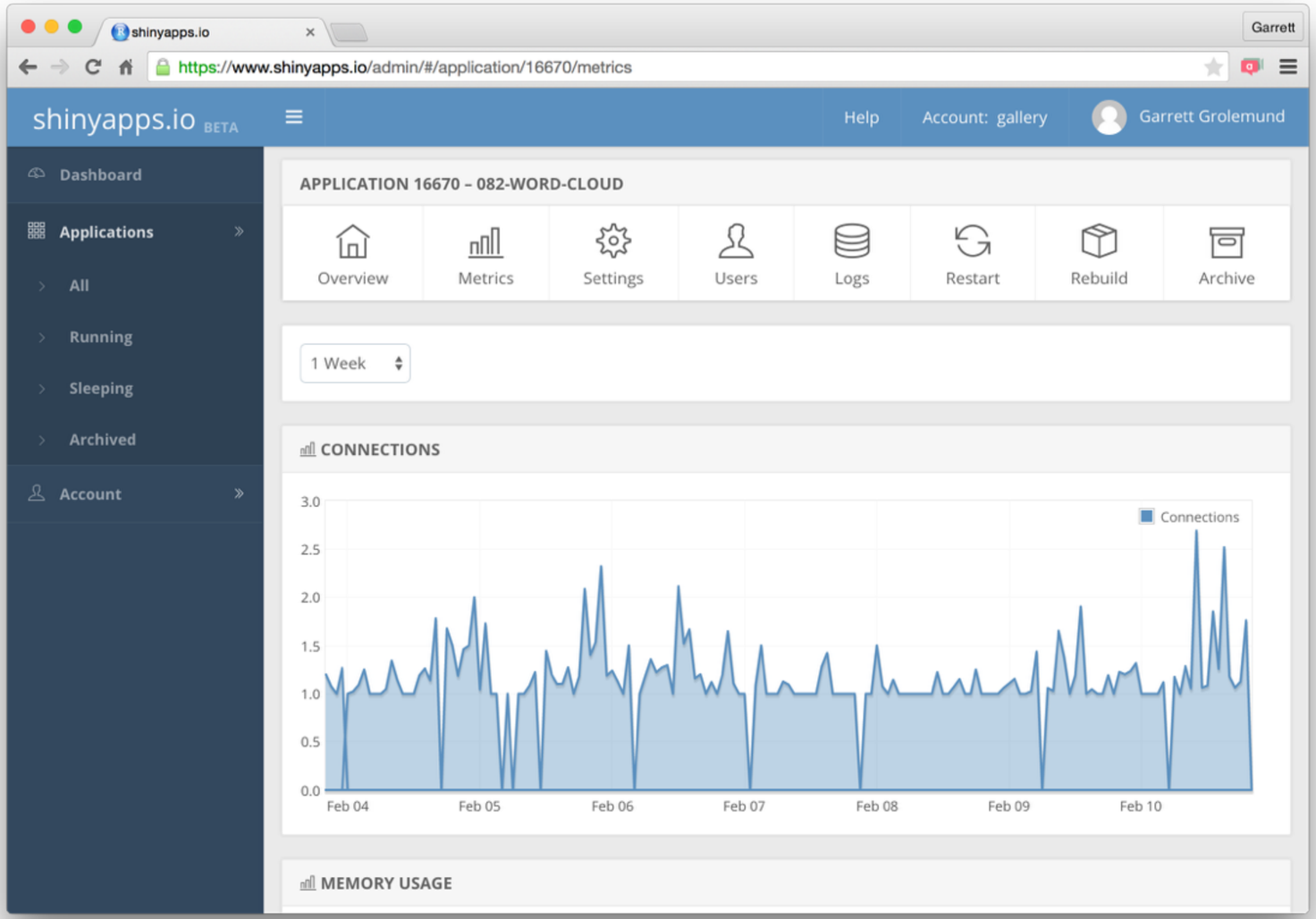
Hassle-free cloud hosting for Shiny

www.shinyapps.io



Hassle-free cloud hosting for Shiny

shinyapps.io



RStudio

FREE

$0 /month

New to Shiny? Deploy your applications to the cloud for FREE. Perfect for teachers and students or those who want a place to learn and play. No credit card required.

- **5** Applications
- **25** Active Hours
- Community Support
- RStudio Branding

BASIC

$39 /month

(or $440/year)

Take your users' experience to the next level. shinyapps.io Basic lets you scale your application performance by adding R processes dynamically as usage increases.

- **Unlimited** Applications
- **250** Active Hours
- Multiple Instances
- Email Support

STANDARD

$99 /month

(or $1,100/year)

Need password protection? shinyapps.io Standard lets you authenticate your application users.

- **Unlimited** Applications
- **1000** Active Hours
- Authentication
- Multiple Instances
- Email Support

PROFESSIONAL

$299 /month

(or $3,300/year)

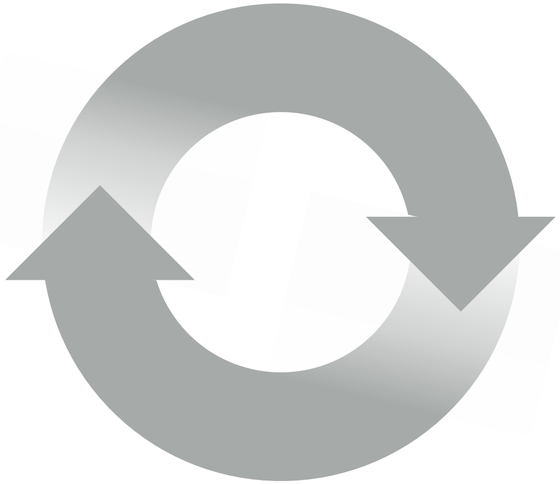
shinyapps.io Professional has it all. Share an account with others in your business or change your shinyapps.io domain into a URL of your own.

- **Unlimited** Applications
- **5000** Active Hours
- Authentication
- Multiple Users
- Multiple Instances
- Custom Domains*
- Email Support

Reactive Programming

Shiny

+



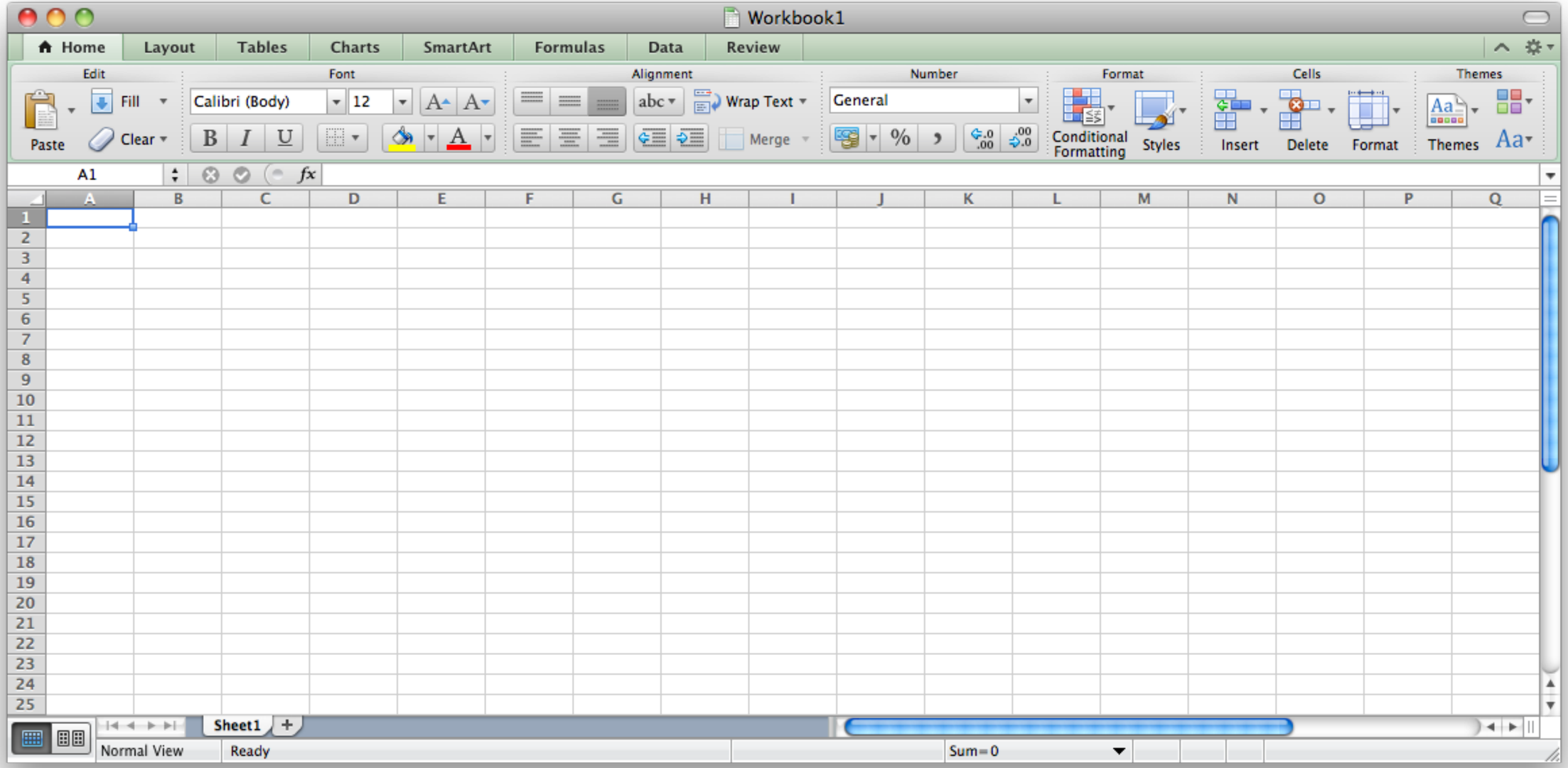
Reactive
Programming

+

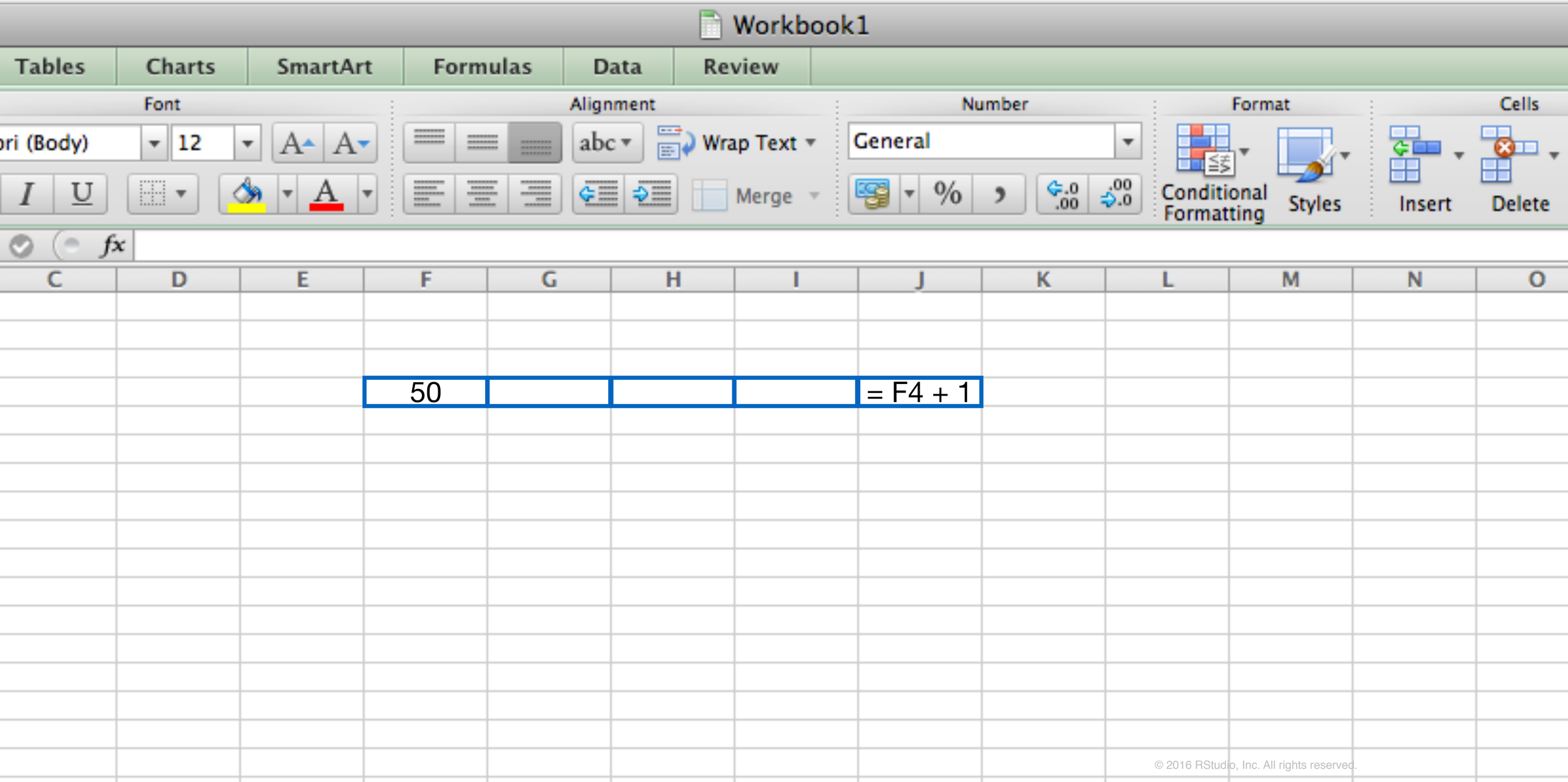
Web based
User Interface

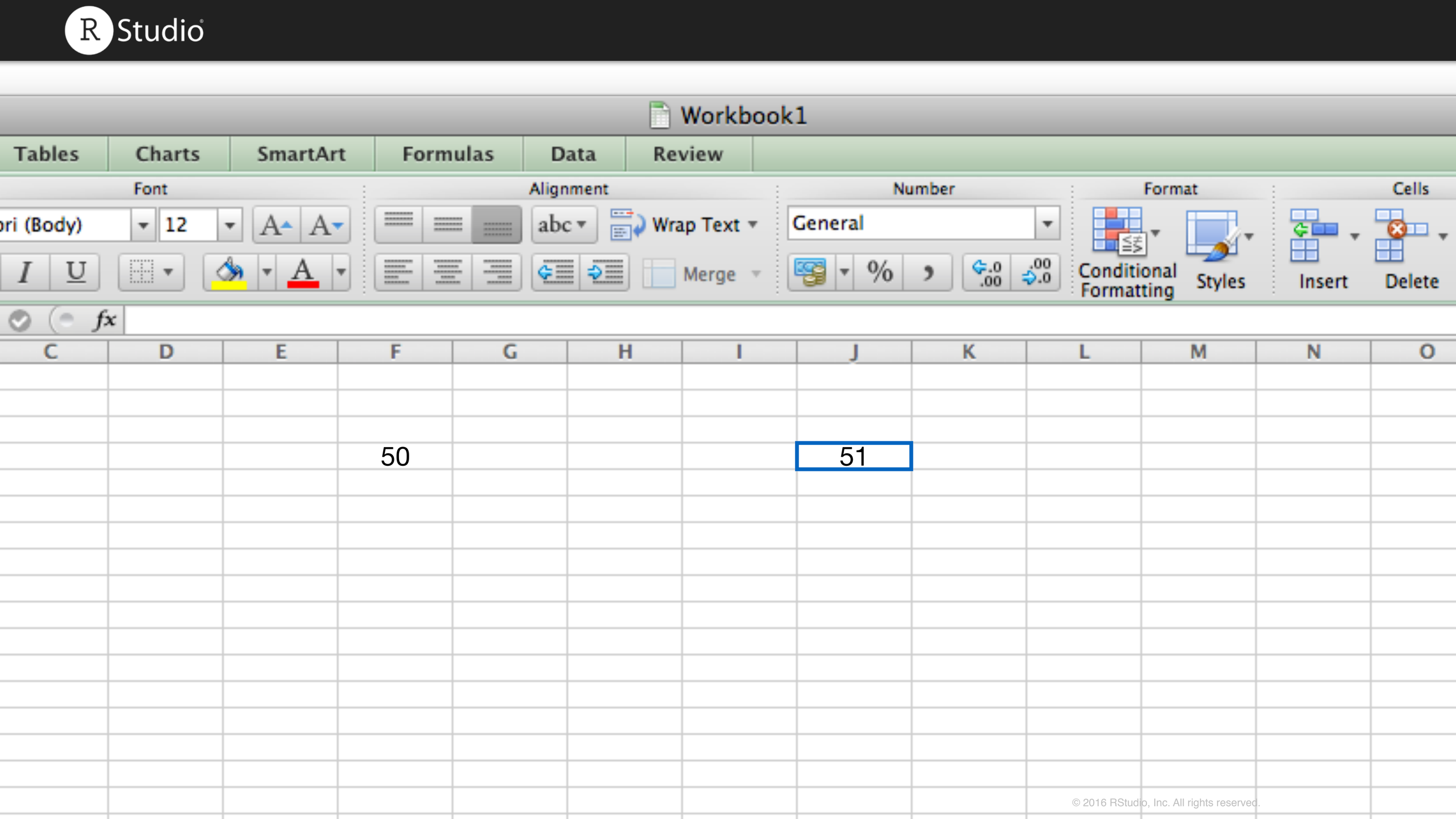
RStudio

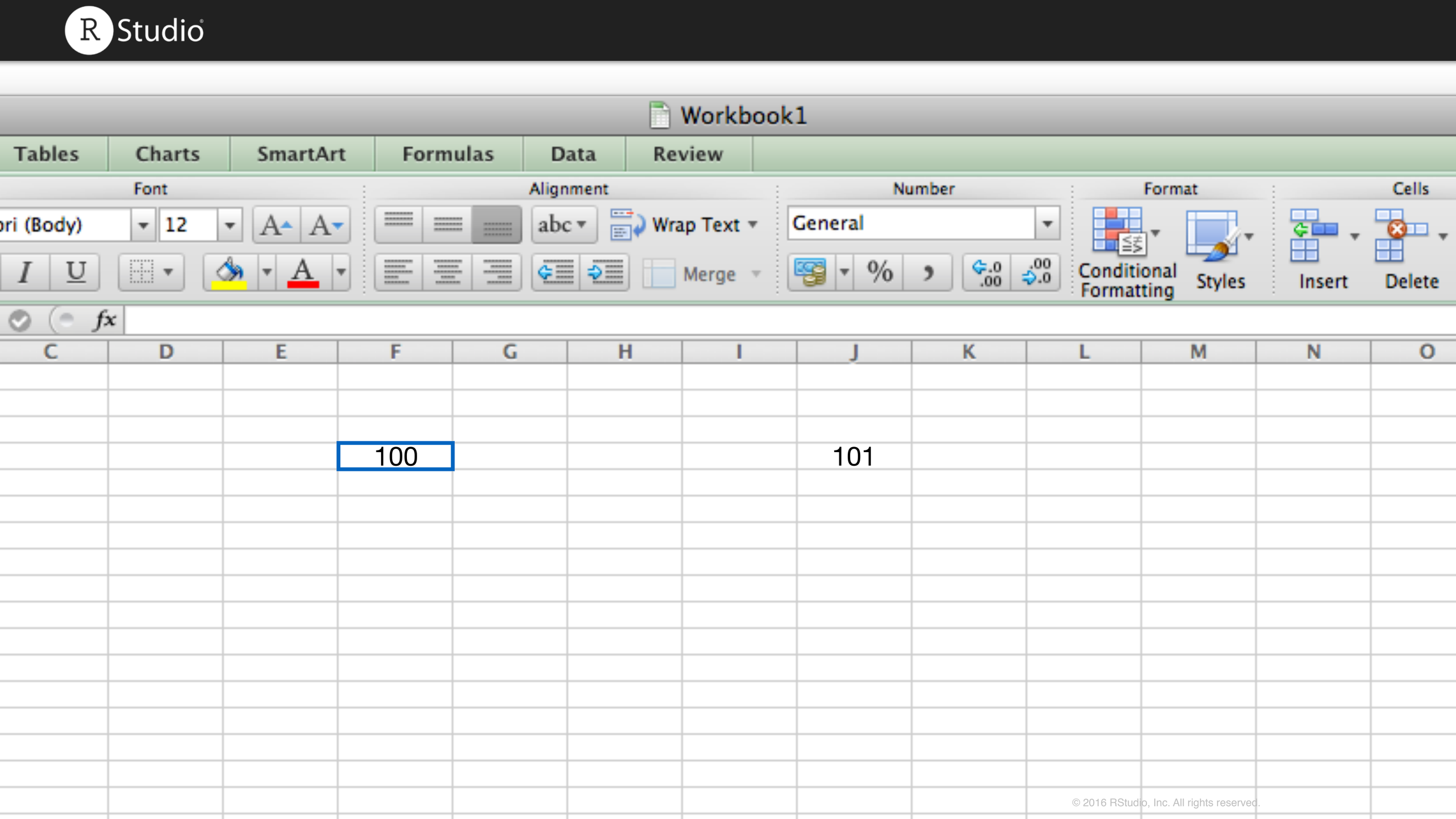
Think Excel.

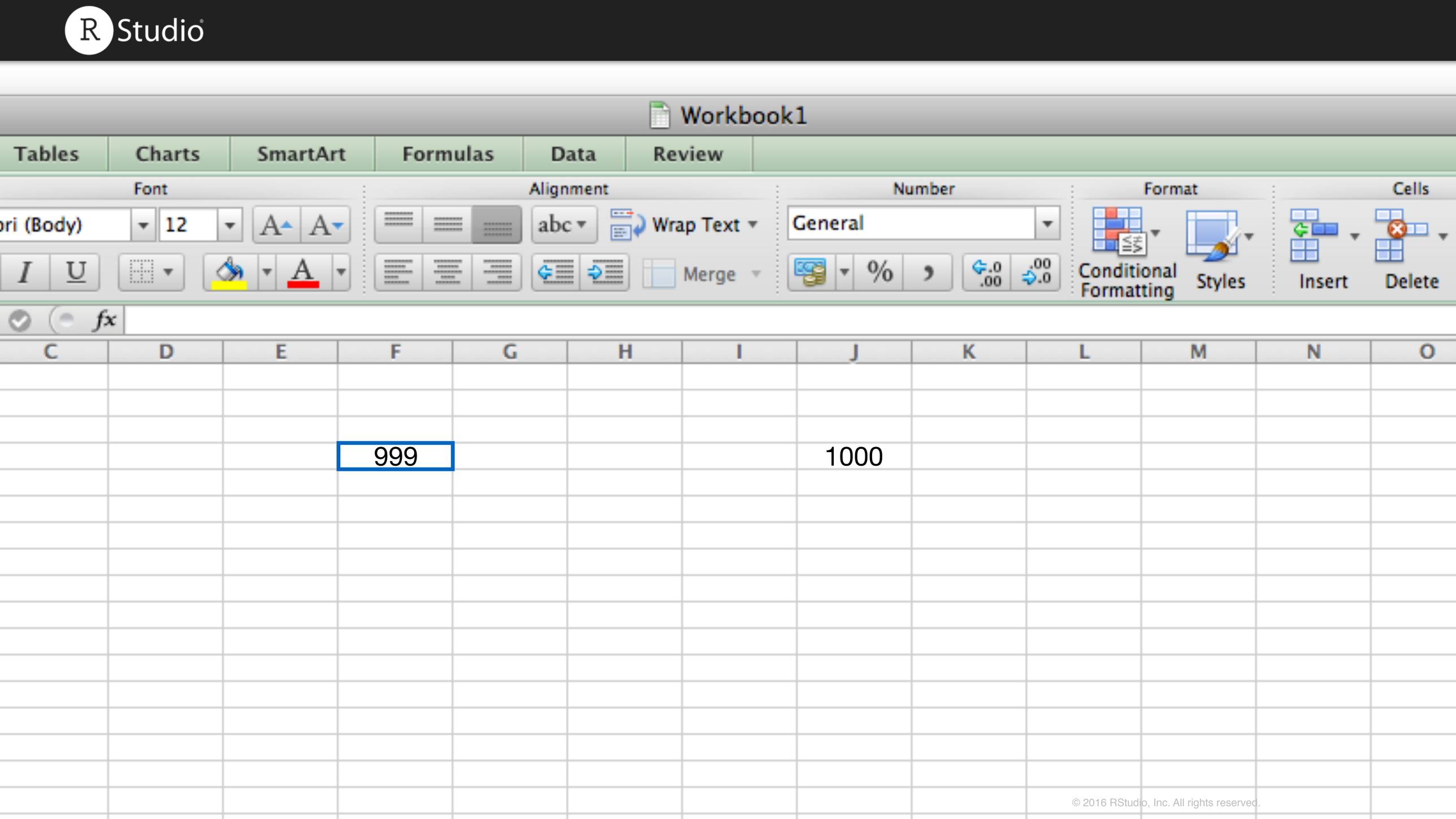


RStudio

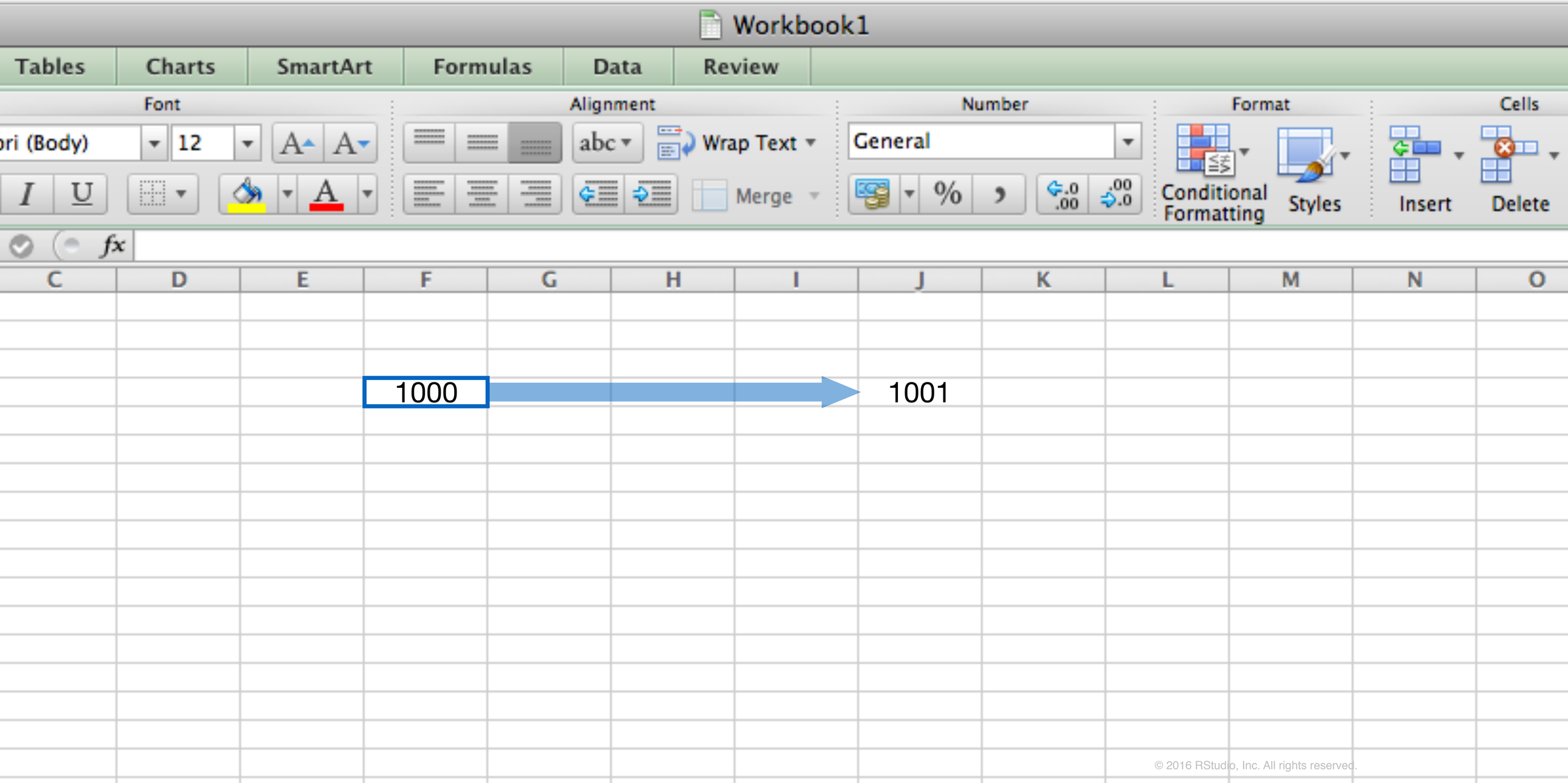




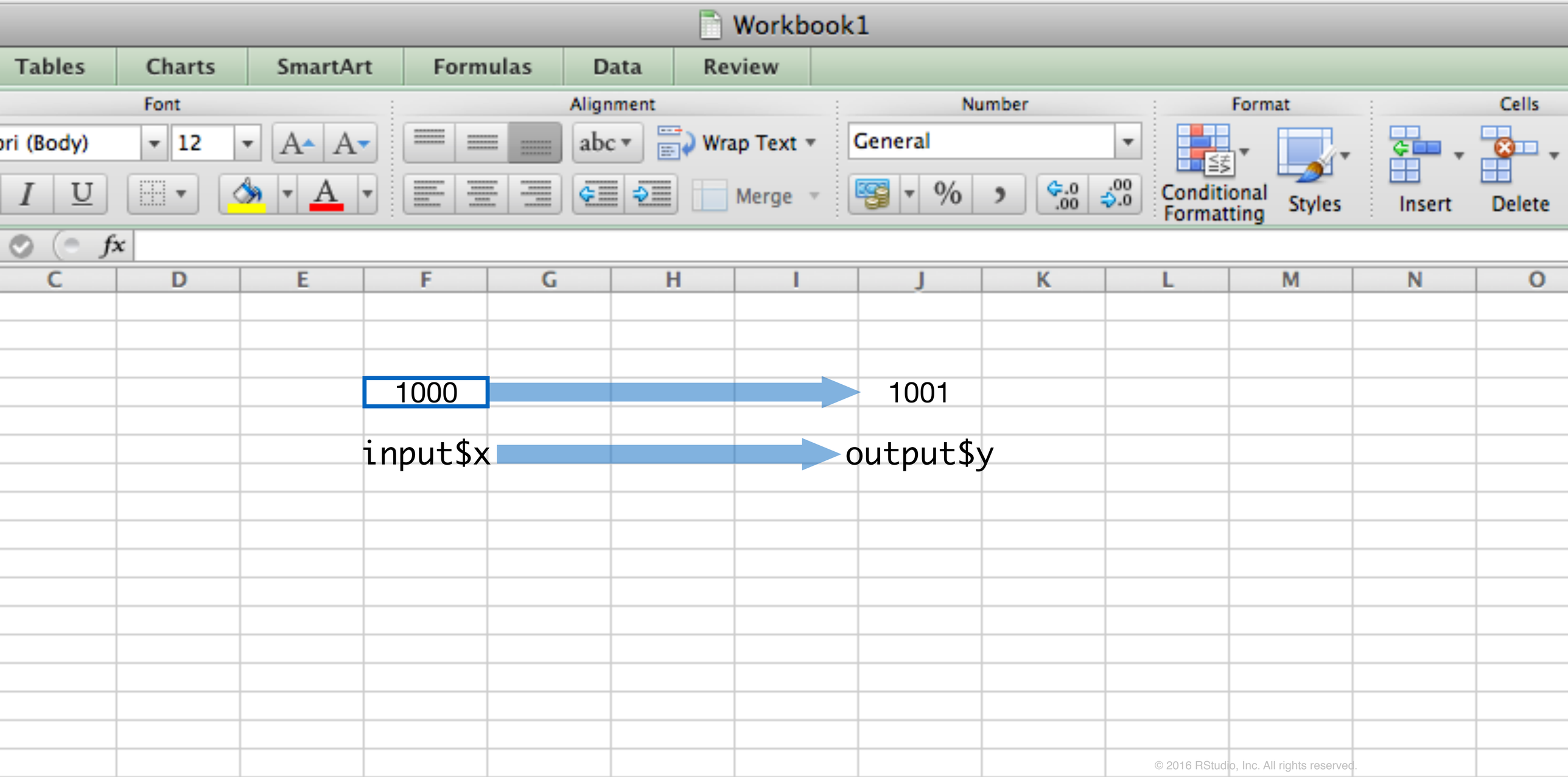


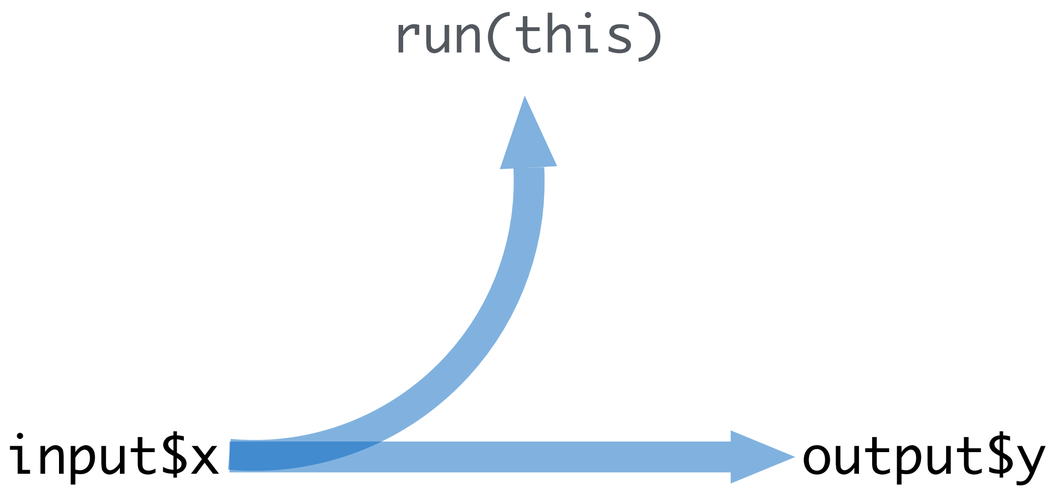


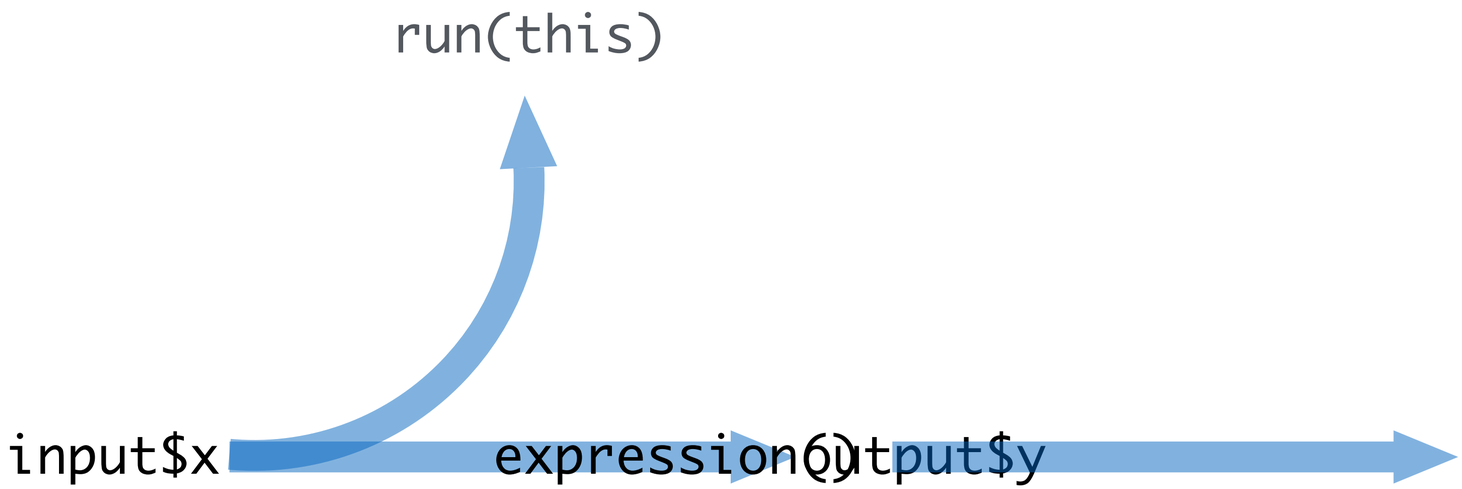
RStudio

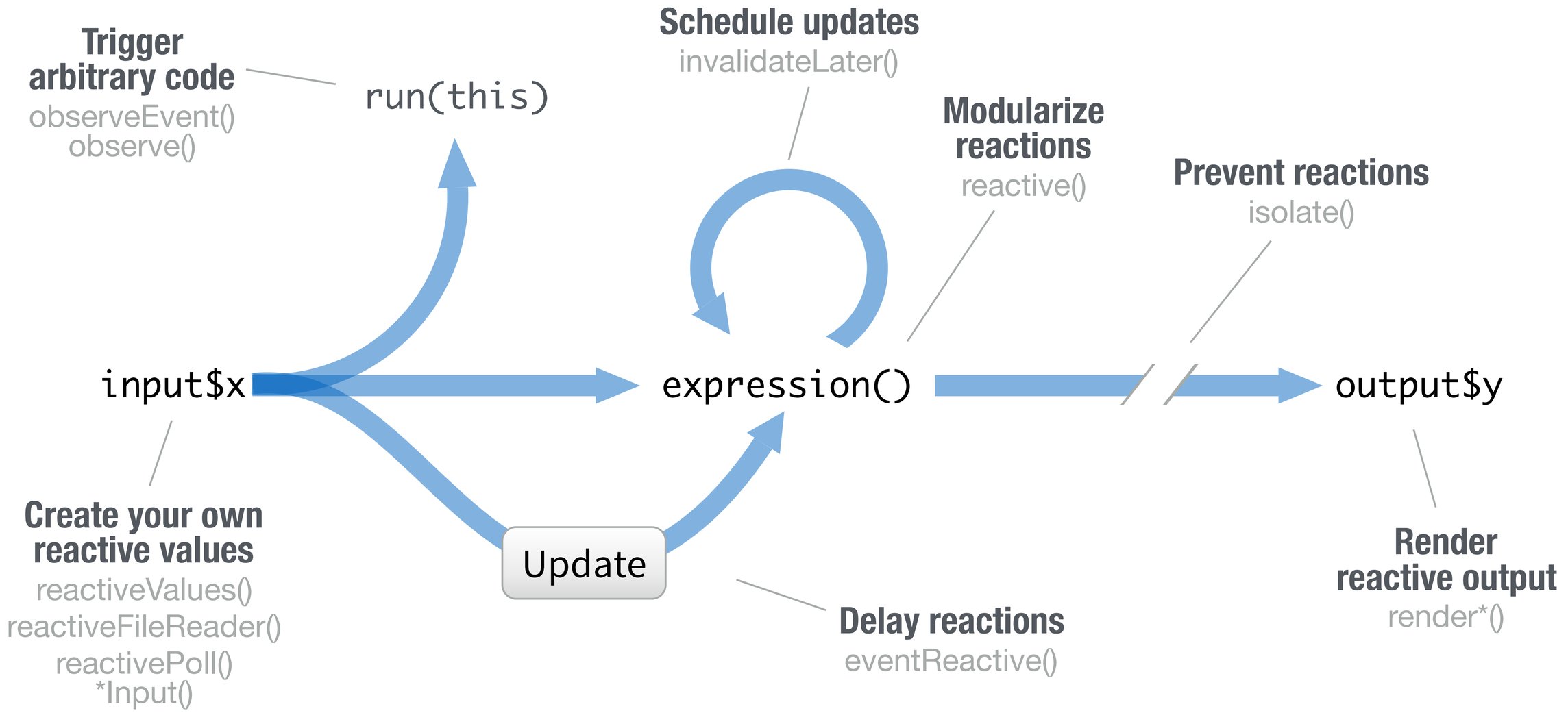


RStudio





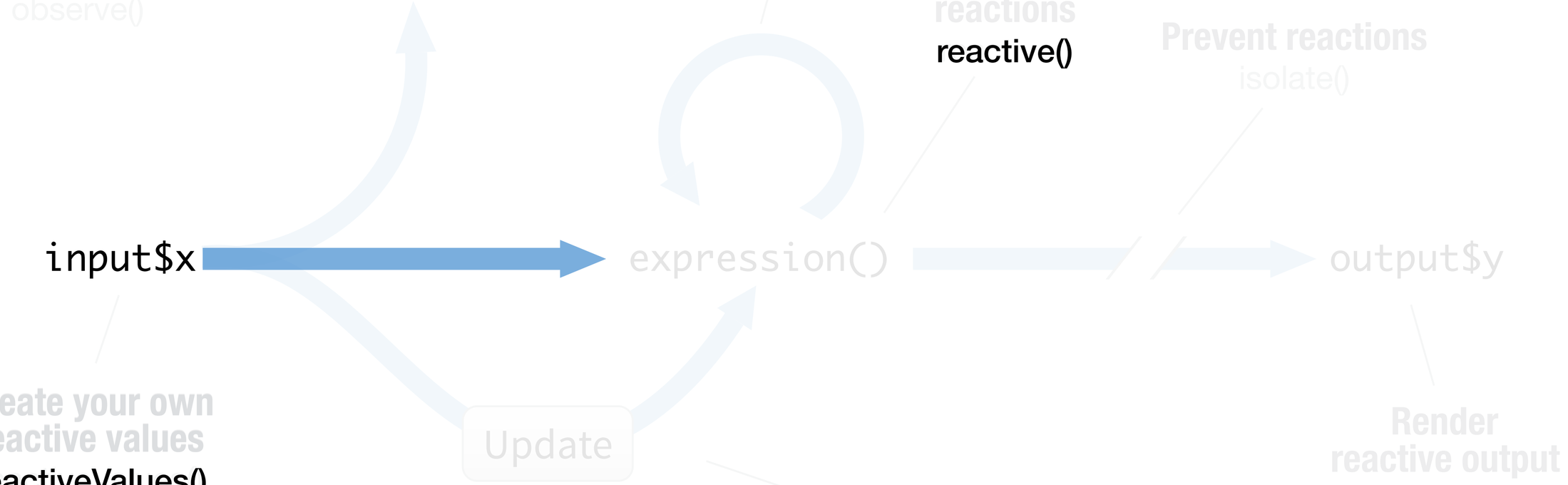




Vocabulary

Reactive Values

Values that notify objects downstream when they change

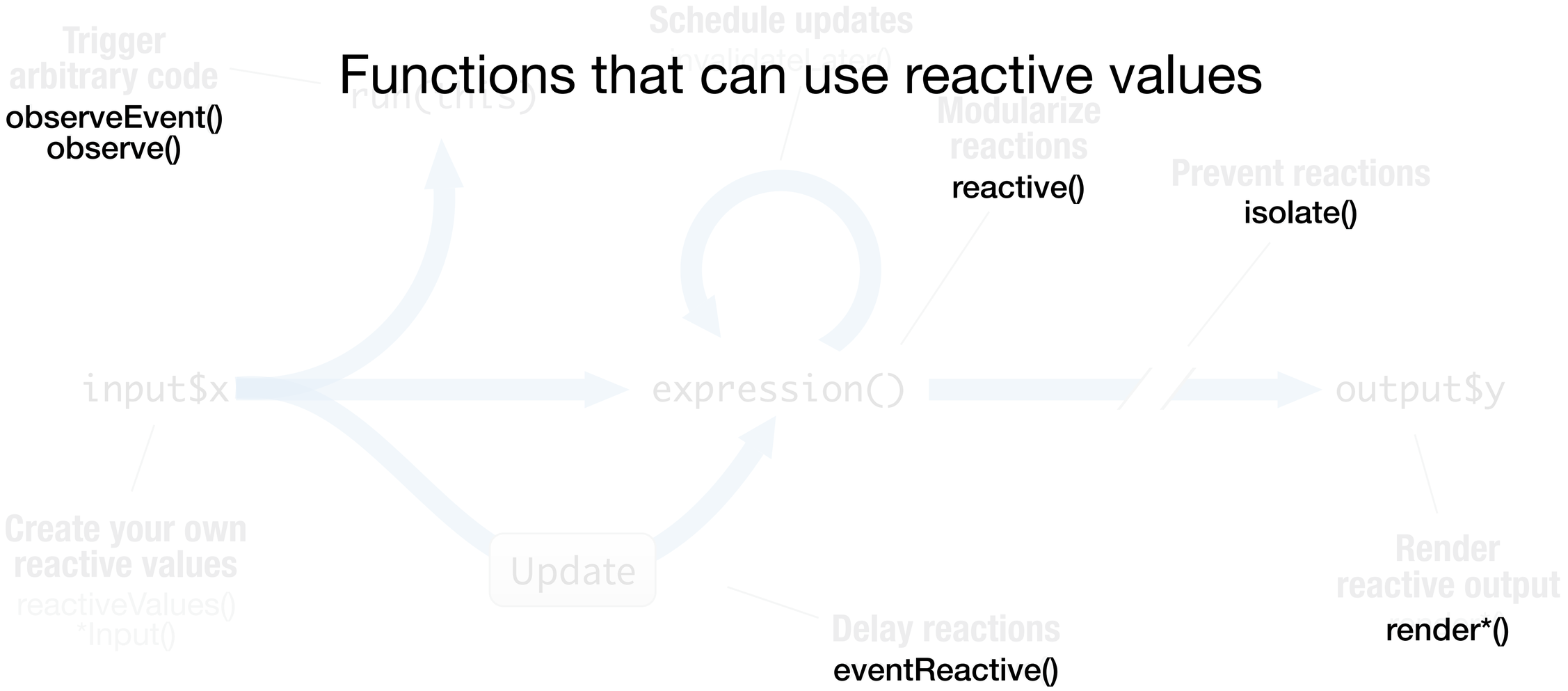


reactive()

reactiveValues()
reactiveFileReader()
reactivePoll()
*Input()

Reactive Functions

Functions that can use reactive values

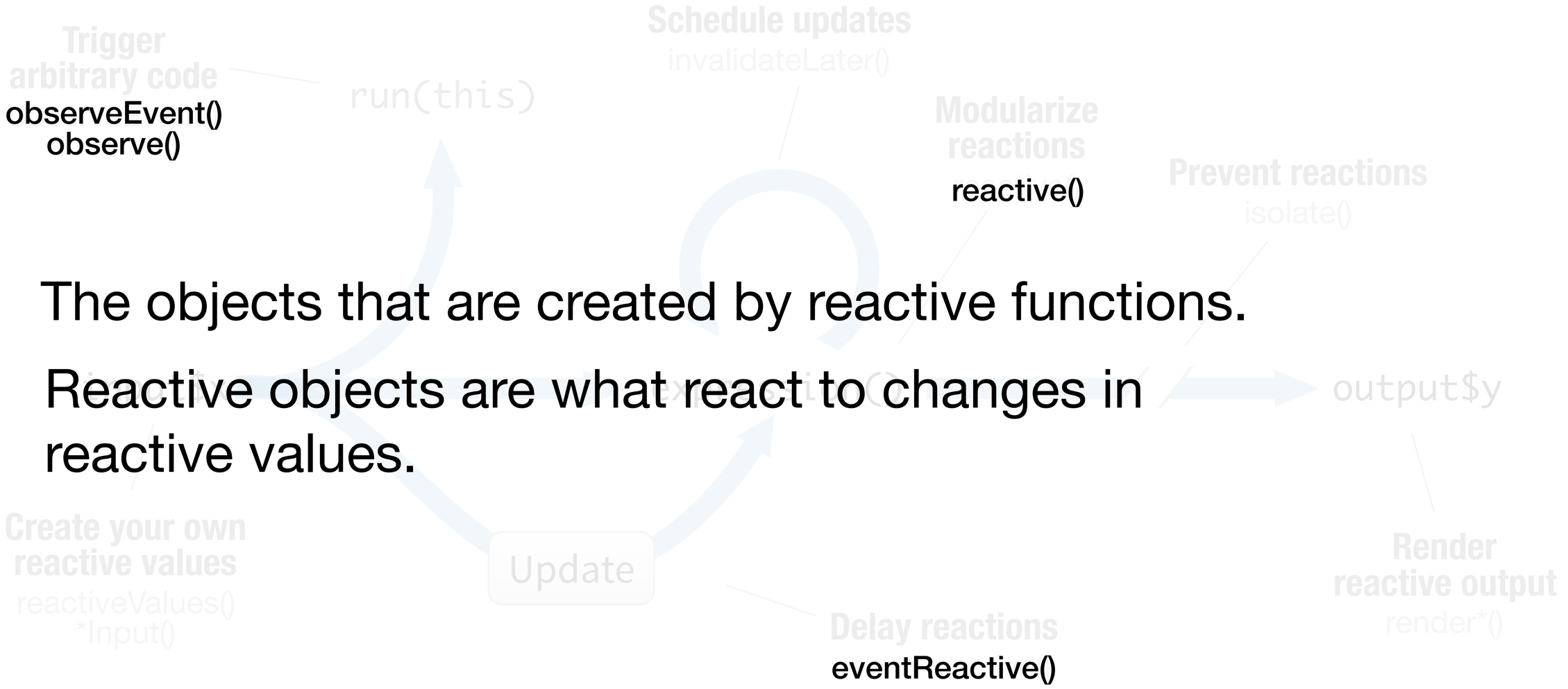


You cannot call an **input value** (reactive value) from outside of a **reactive function**.

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

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

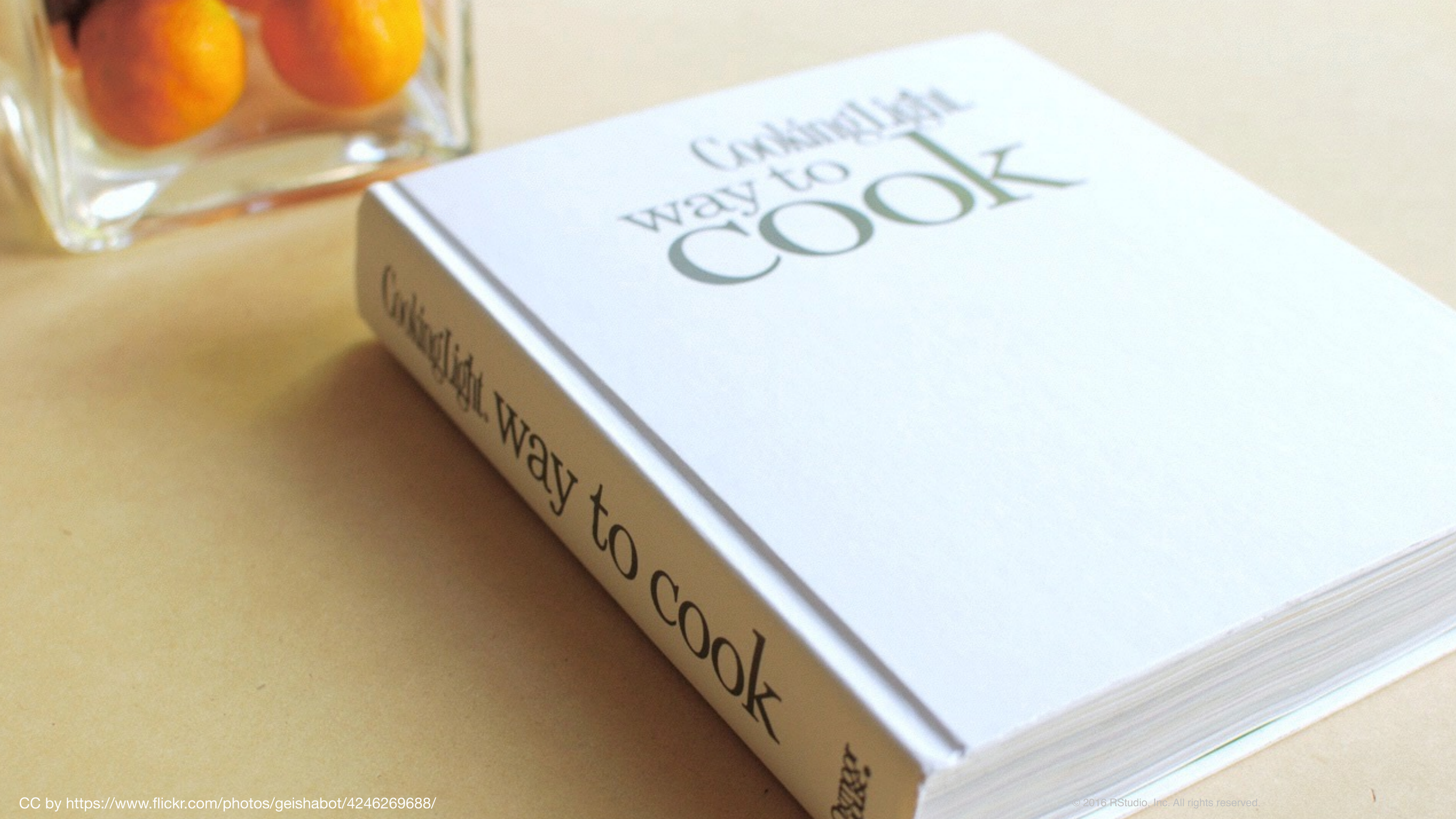
Reactive objects



The objects that are created by reactive functions.

Reactive objects are what react to changes in reactive values.

```
objects <- functions()values
```



Think of reactivity in R as a four step process

A reactive function

1 creates a reactive object

and gives it a set of instructions to execute later.

The reactive object

2 takes a dependency

on one or more reactive values

3 Reactive values notify

the objects that depend on them when they change

4 Reactive objects respond

when notified by executing their instructions

```
input$x
```

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

```
> hist(rnorm(input$x))
```

Outline

1. Reactive programming in Shiny
   a. rendered output
   b. reactive expressions
   c. isolate
   d. eventReactive
   e. observeEvent, observe
   f. reactiveValues
   g. invalidateLater
2. Advice

Reactive objects

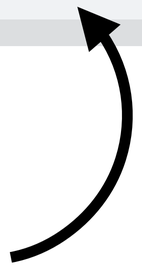
Different reactive functions create different types of reactive objects. Pay attention to:

1. Which reactive values will an object take a dependency on?
2. How will the object respond when invalidated?

Build display output with render*()

render*() (output objects)

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

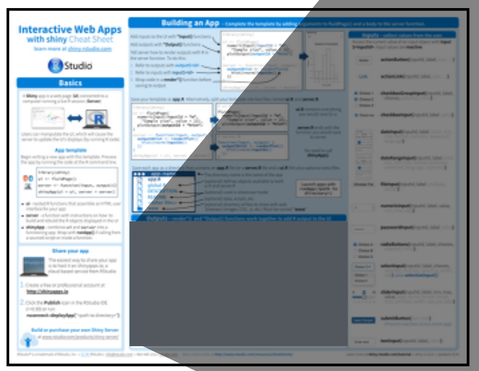


- Takes a dependency on every reactive value in code chunk*
- Reruns entire code chunk when invalidated

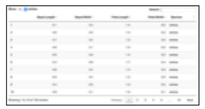
**Unless you use isolate()*

Outputs

display output from R.



Outputs - render*() and *Output() functions work together to add R output to the UI



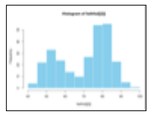
DT::**renderDataTable**(expr, options, callback, escape, env, quoted)

works with

dataTableOutput(outputId, icon, …)

renderImage(expr, env, quoted, deleteFile)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

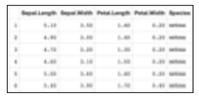


renderPlot(expr, width, height, res, …, env, quoted, func)

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

renderPrint(expr, env, quoted, func, width)

verbatimTextOutput(outputId)



renderTable(expr,…, env, quoted, func)

tableOutput(outputId)

foo

renderText(expr, env, quoted, func)

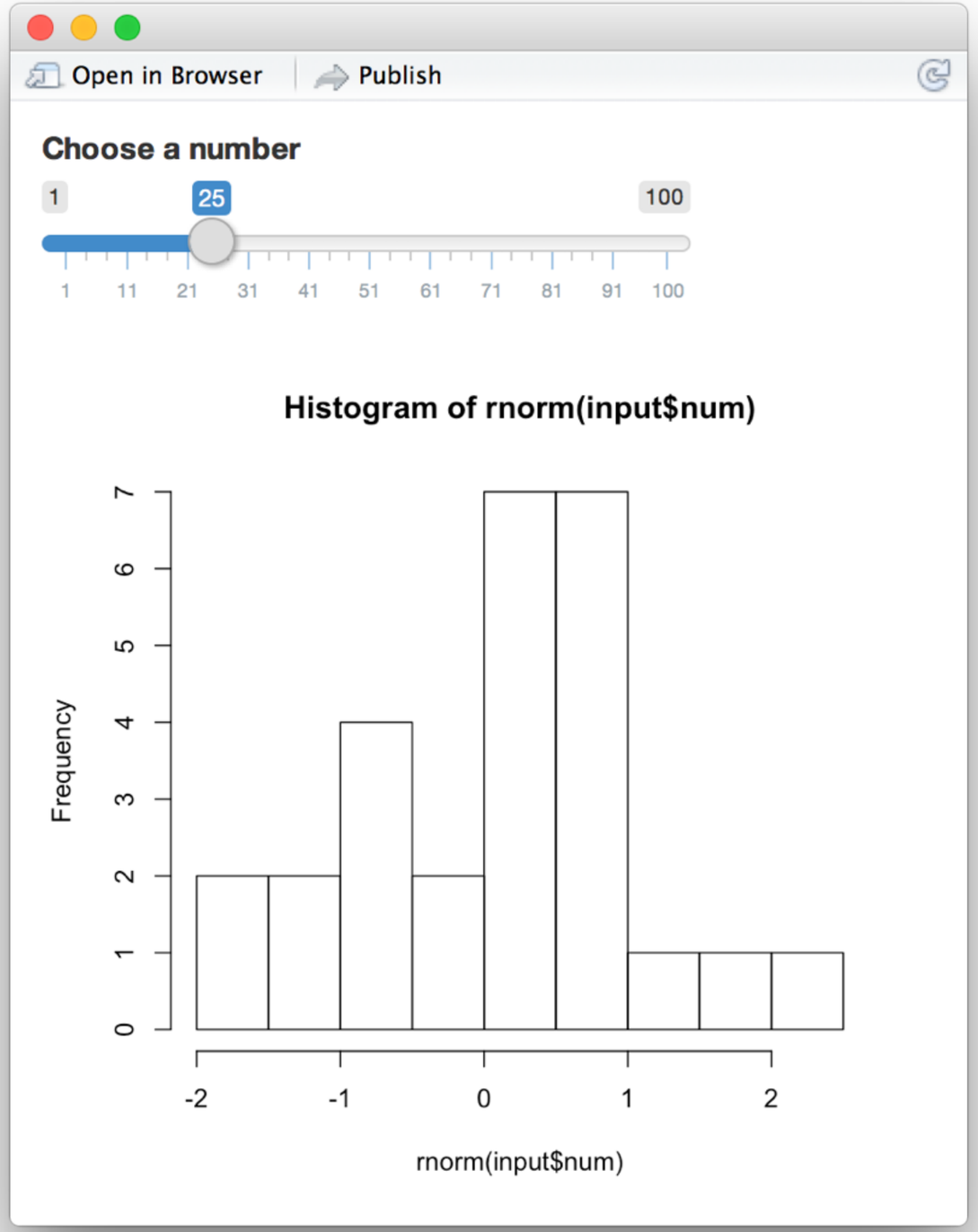
textOutput(outputId, container, inline)

renderUI(expr, env, quoted, func)

uiOutput(outputId, inline, container, …)
& **htmlOutput**(outputId, inline, container, …)

input$num

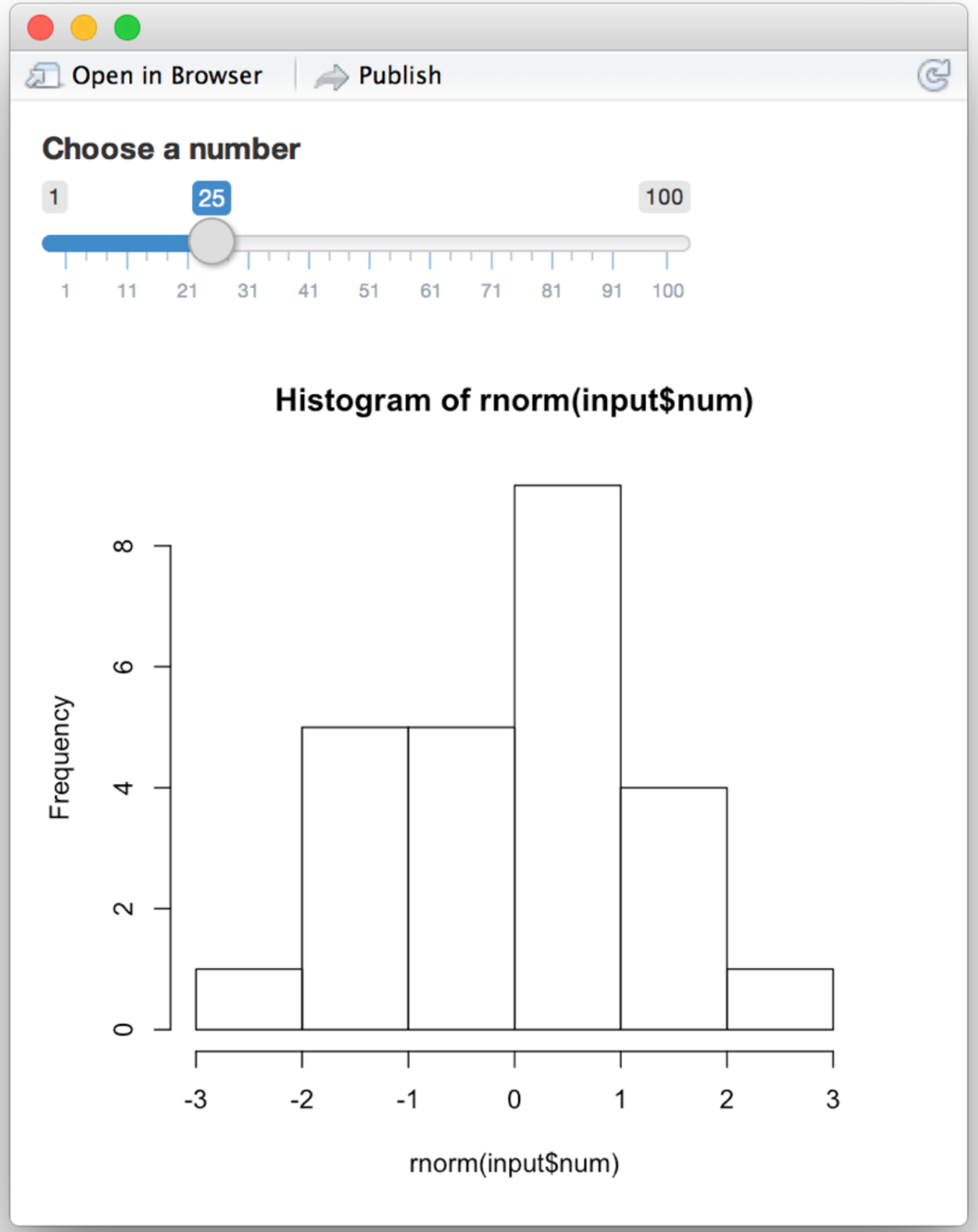
```
output$hist <- renderPlot({
  hist(rnorm(input$num))
})
```



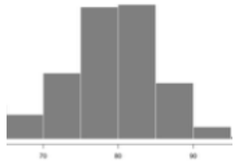
input$num

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

```
> hist(rnorm(input$num))
```



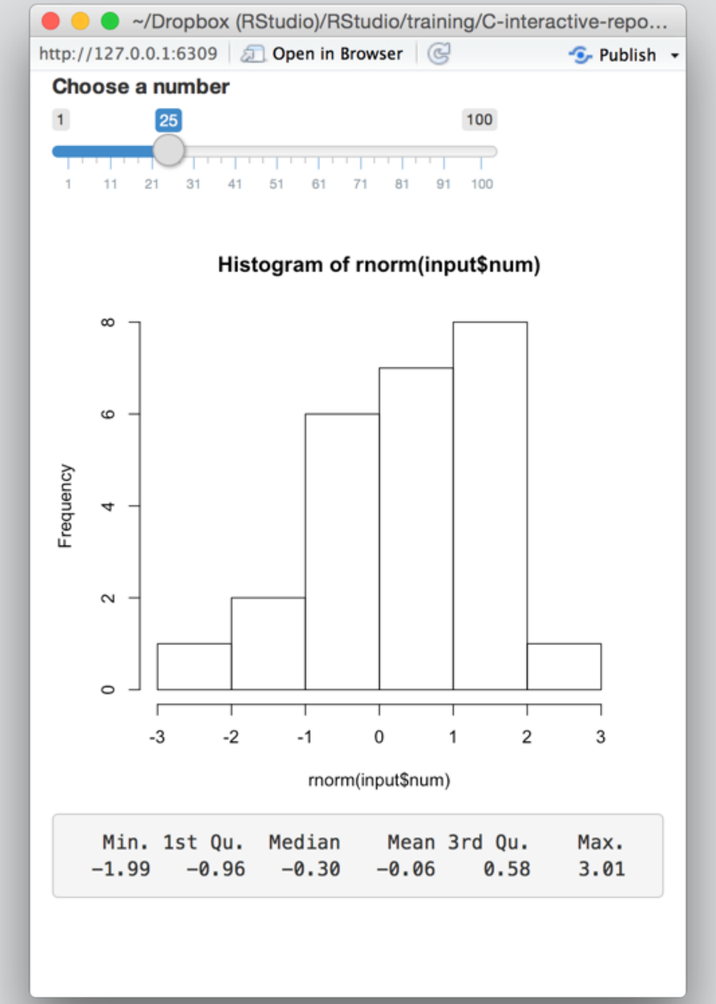
Use…



render() to make an **object to display** in the UI.

`input$x` → `output$y`

Your Turn



Use **renderPrint()** and **verbatimTextOutput()** to add the output of

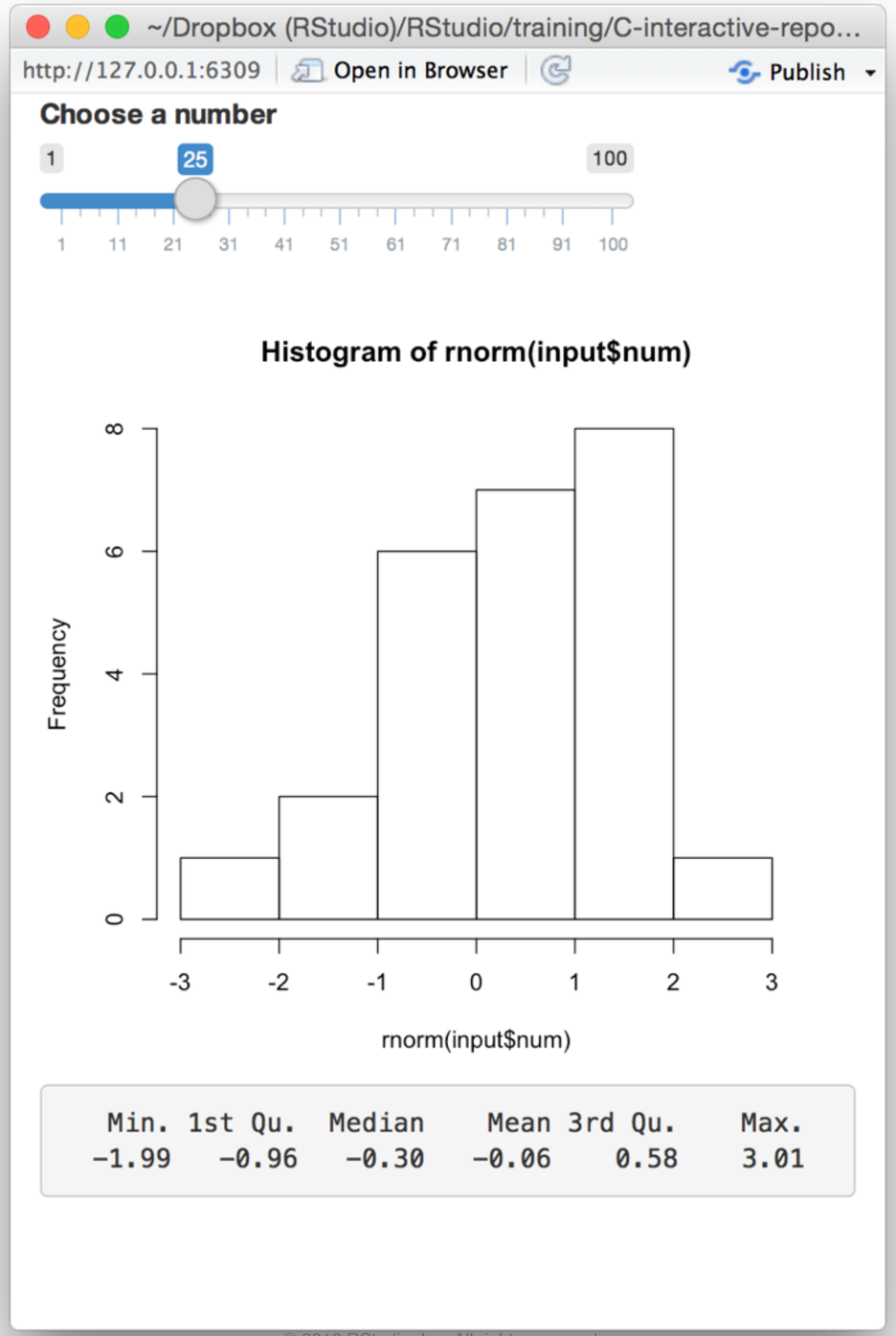
```
summary(rnorm(input$num))
```

to your single file histogram app.

05:00

Build reusable objects with reactive()

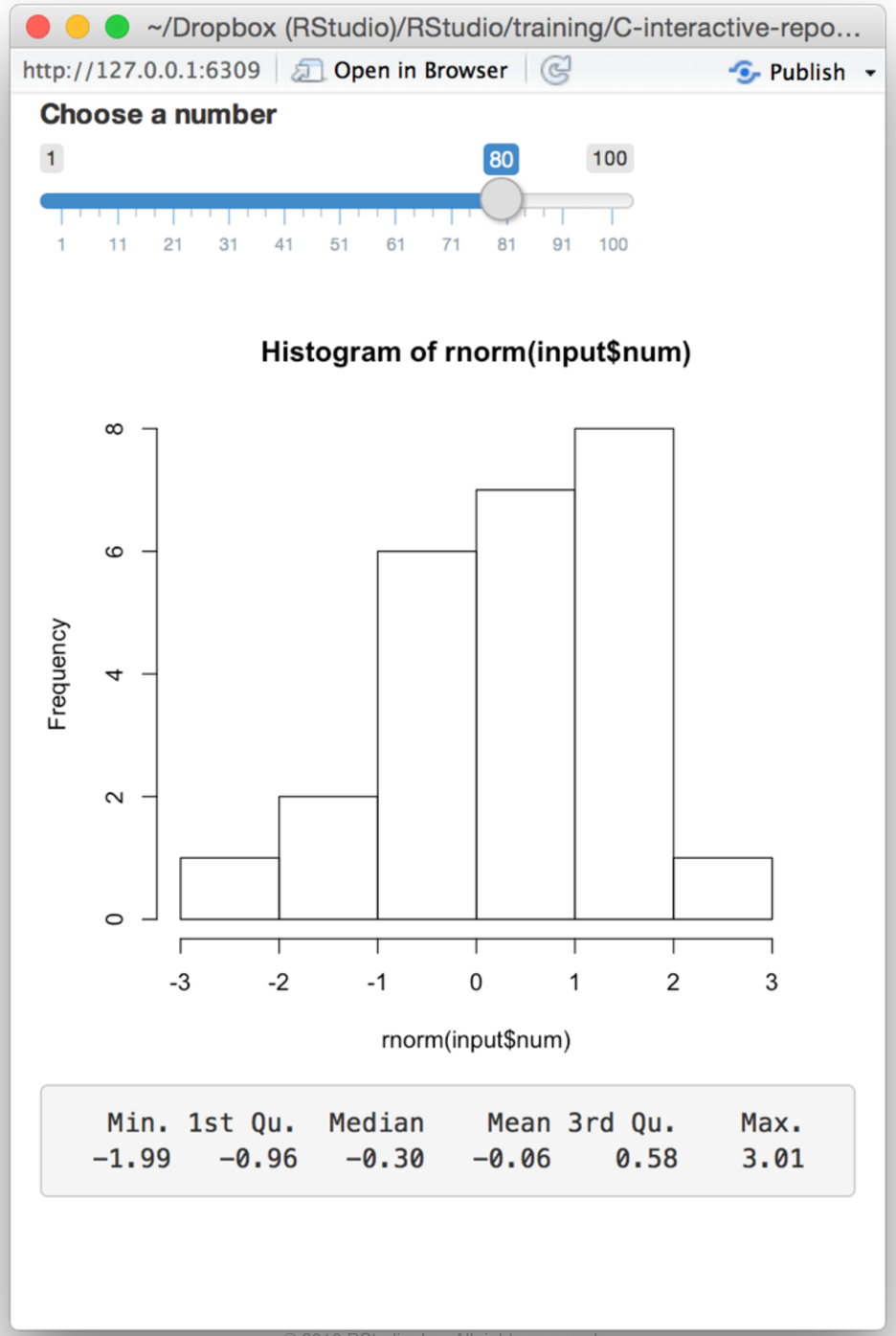
```
ui <- fluidPage(
  sliderInput("num", "Slide Me", 1, 100, 50),
  plotOutput("hist"),
  verbatimTextOutput("sum")
)
server <- function(input, output) {
   output$hist <- renderPlot({
     hist(rnorm(input$num))
   })
   output$sum <- renderPrint({
     summary(rnorm(input$num))
   })
}
shinyApp(ui = ui, server = server)
```



input$num

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

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



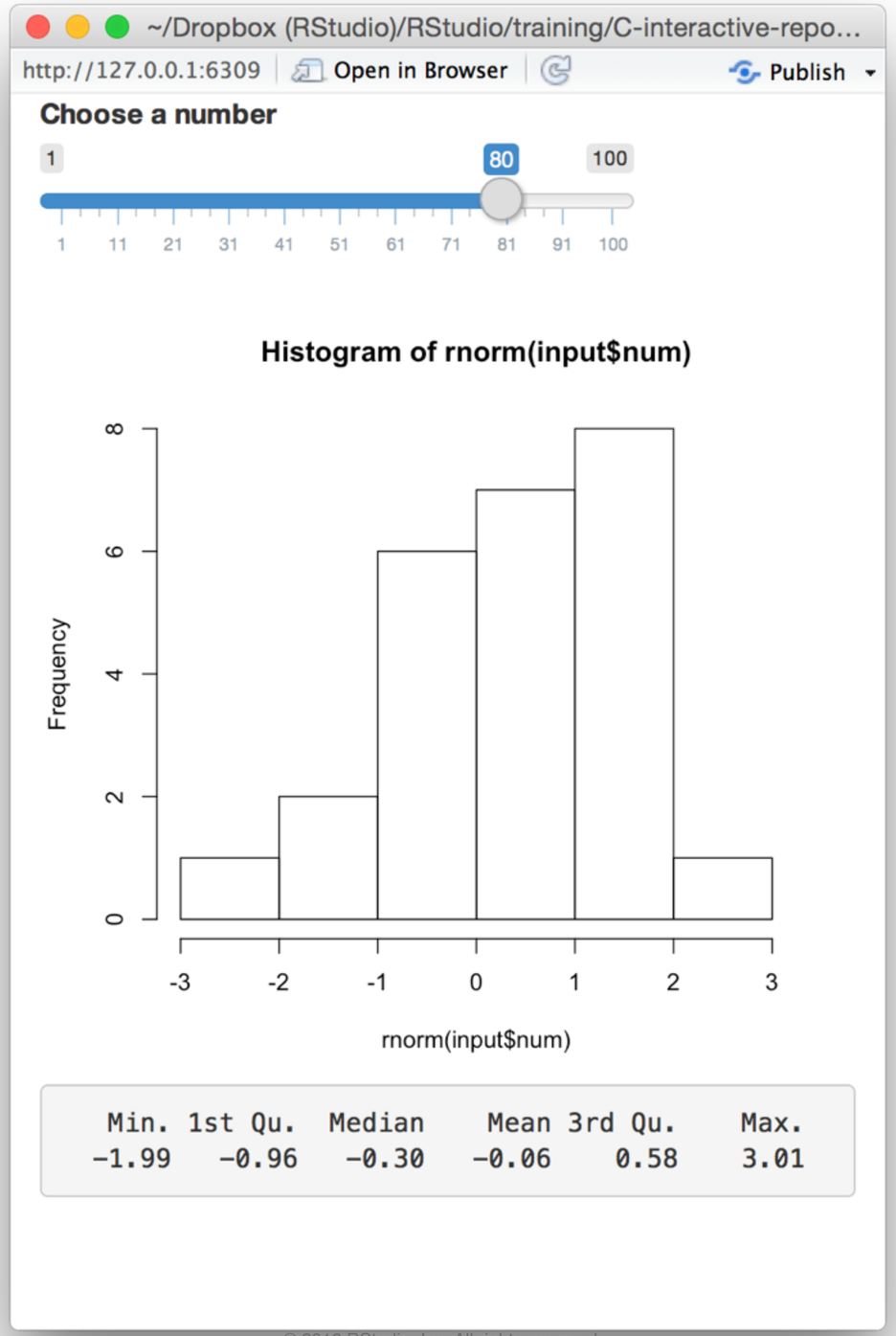
input$num

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

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

```
> hist(rnorm(input$num))
```

```
> summary(rnorm(input$num))
```



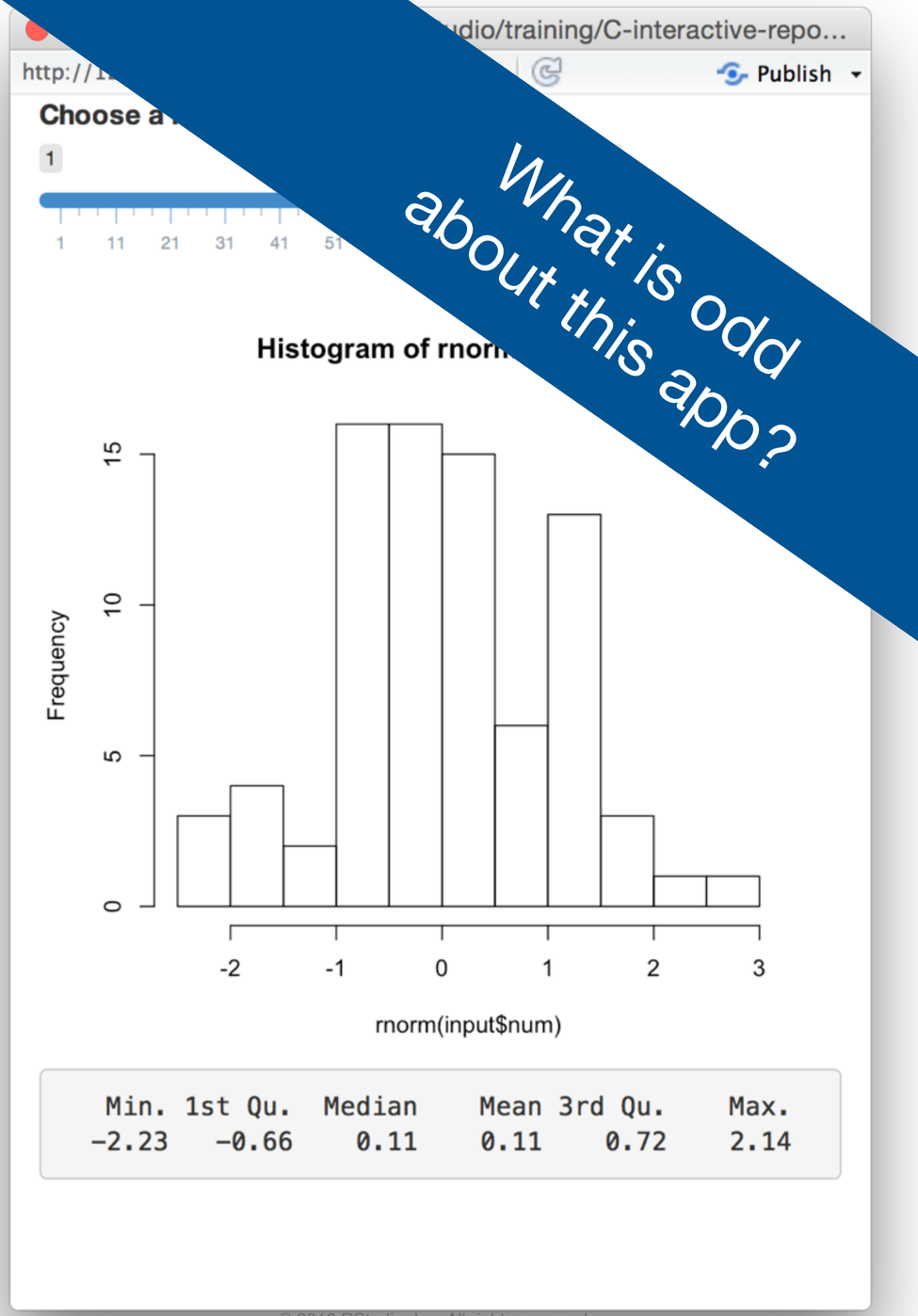
input$num

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

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

```
> hist(rnorm(input$num))
```

```
> summary(rnorm(input$num))
```



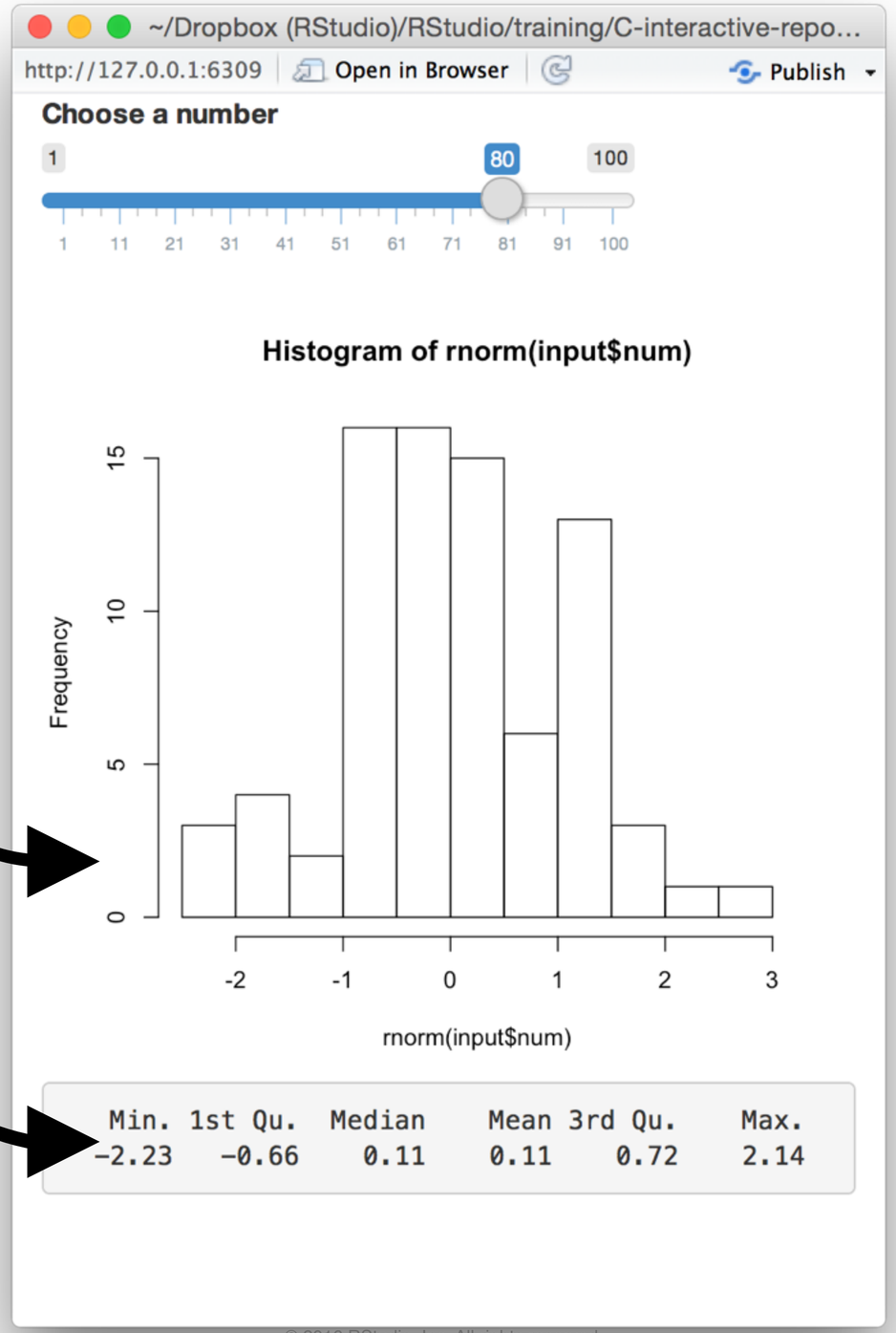
What is odd about this app?

```
input$num
```

Can these describe the same data?

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

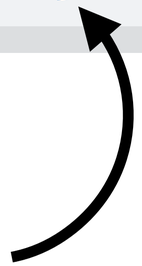
```
output$sum <-
  renderPrint({
   summary(rnorm(input$num))
  })
```



reactive()

Makes a reactive object that you can use in downstream code.

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



- Takes a dependency on every reactive value in code chunk*
- Only notifies downstream objects when invalidated

**Unless you use isolate()*

A reactive expression is special in two ways

```
data()
```

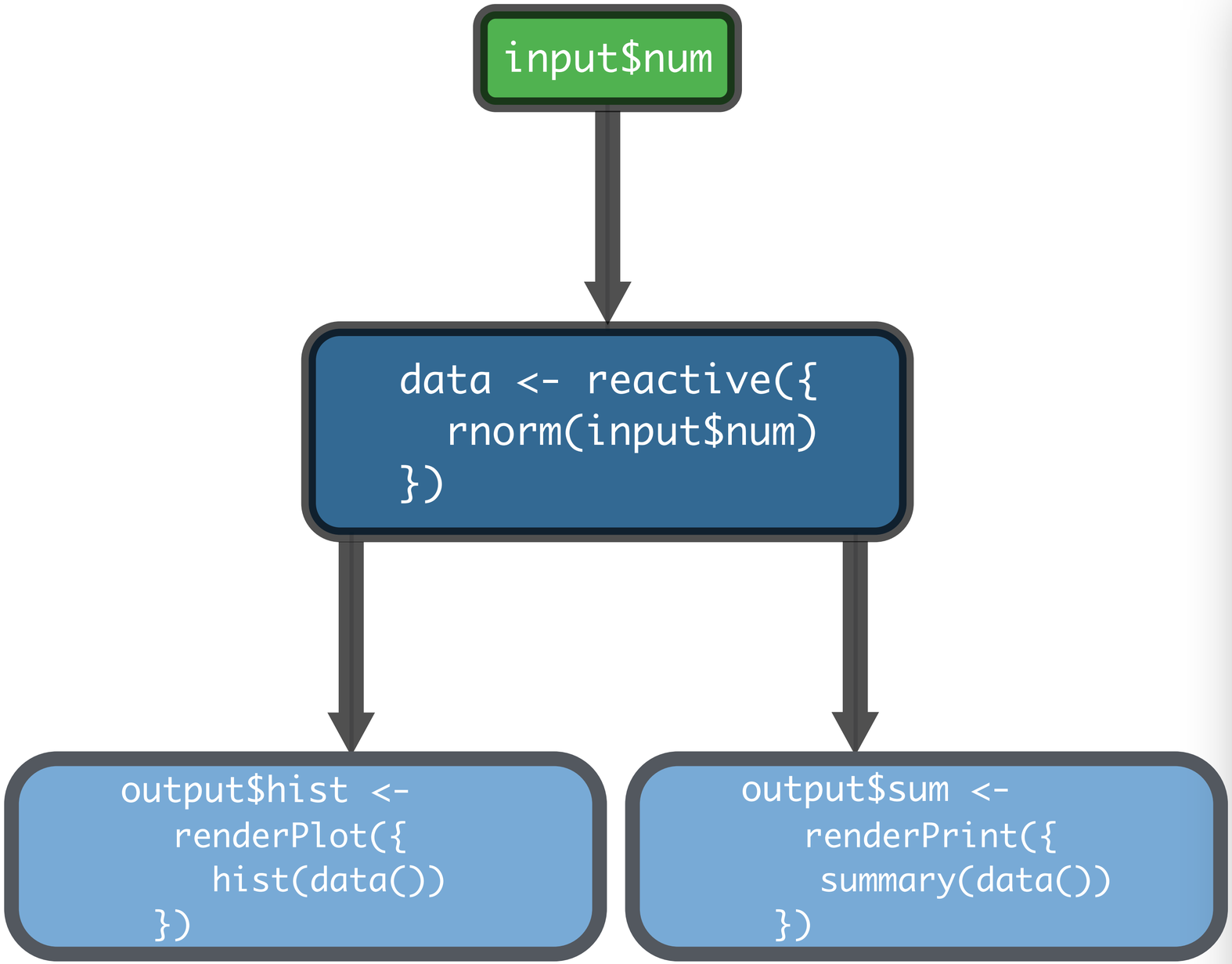
1. You call a reactive expression like a function

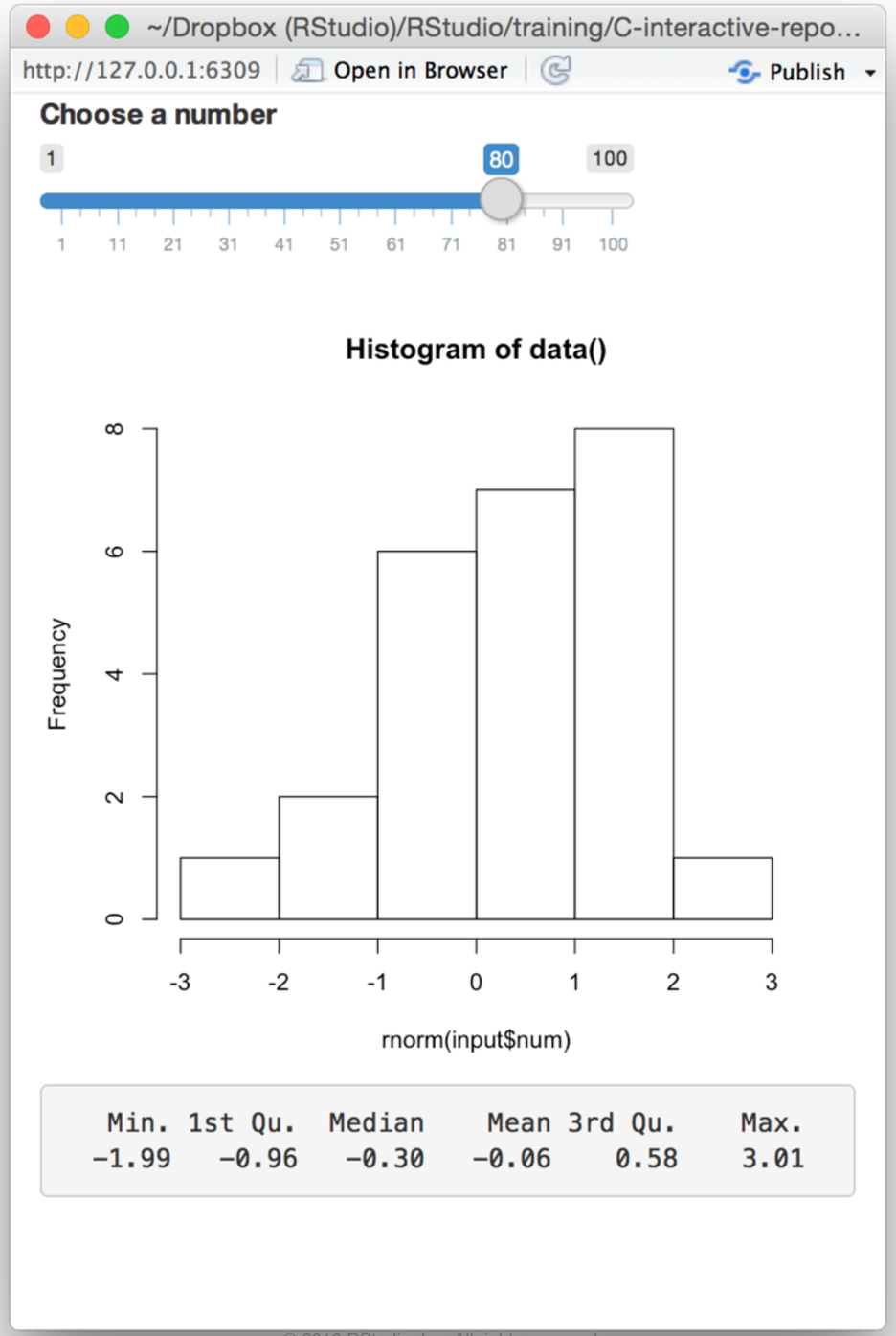
A reactive expression is special in two ways

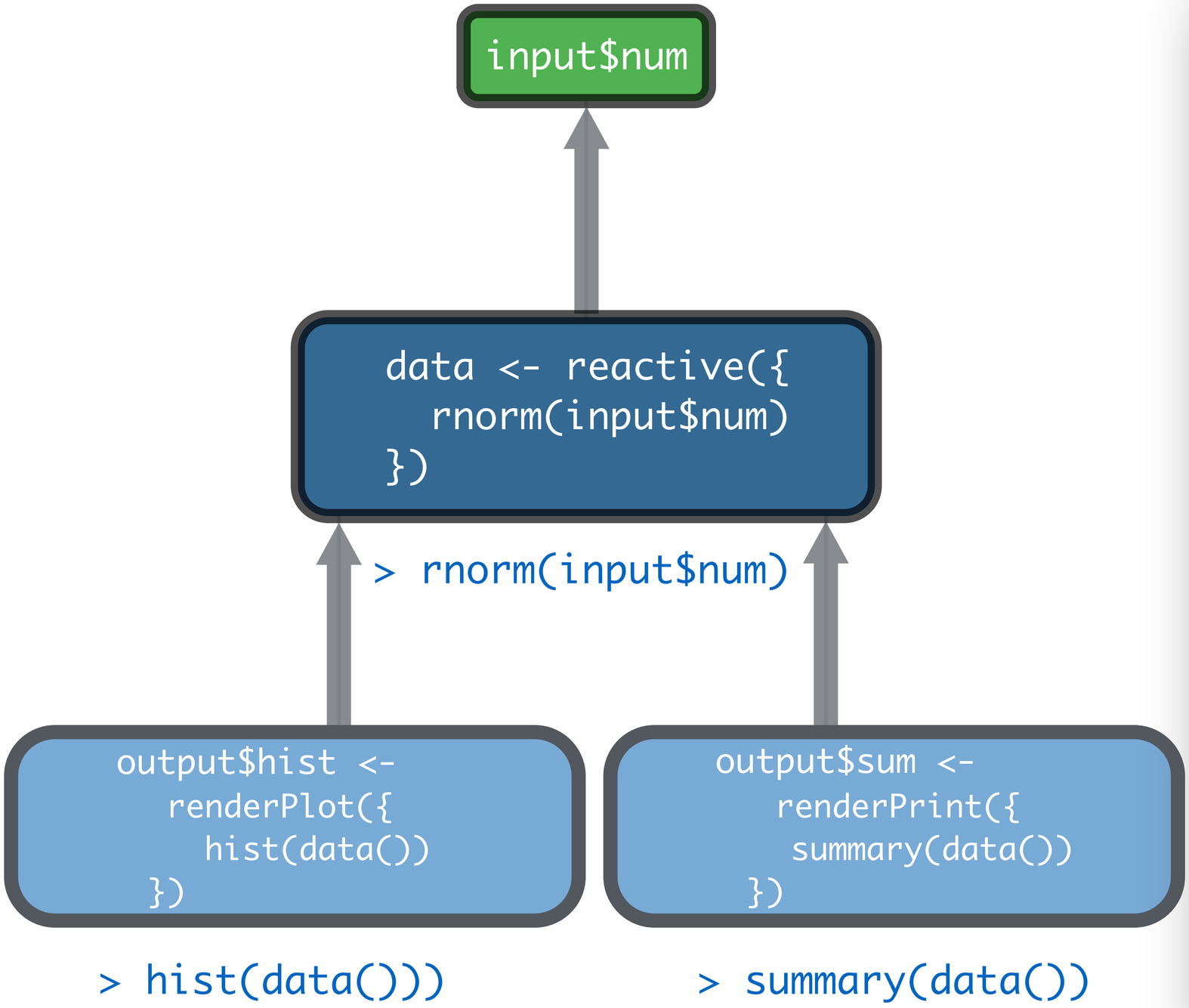
```
data()
```

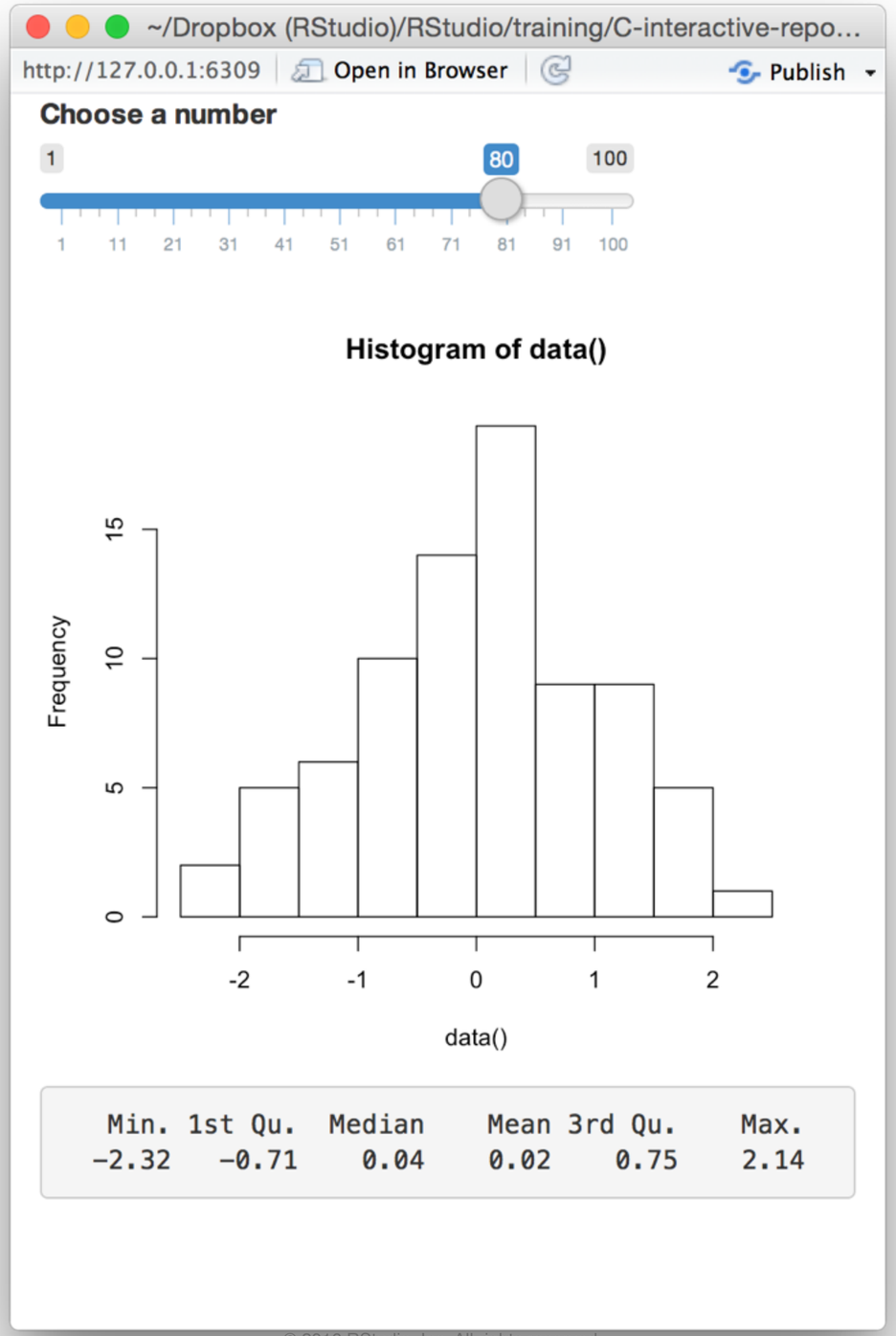
1 You call a reactive expression like a function

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

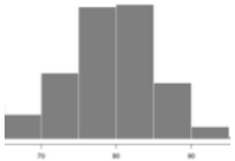


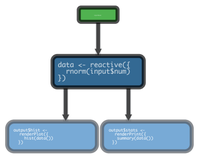






Use…

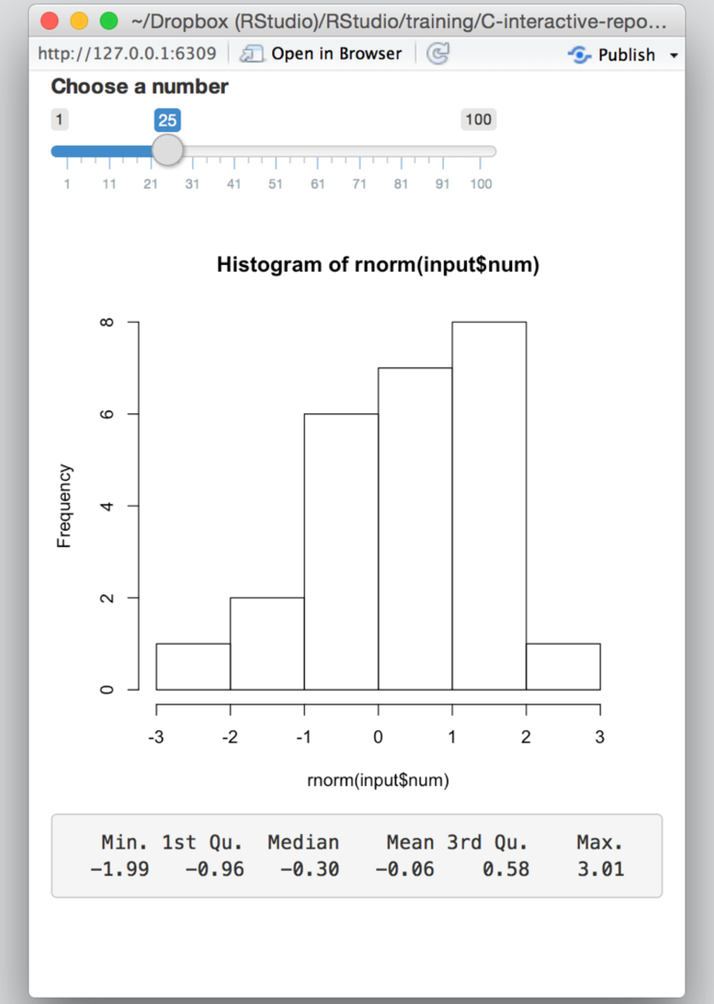




render() to make an **object to display** in the UI.

reactive() to make an **object to use** in downstream code.

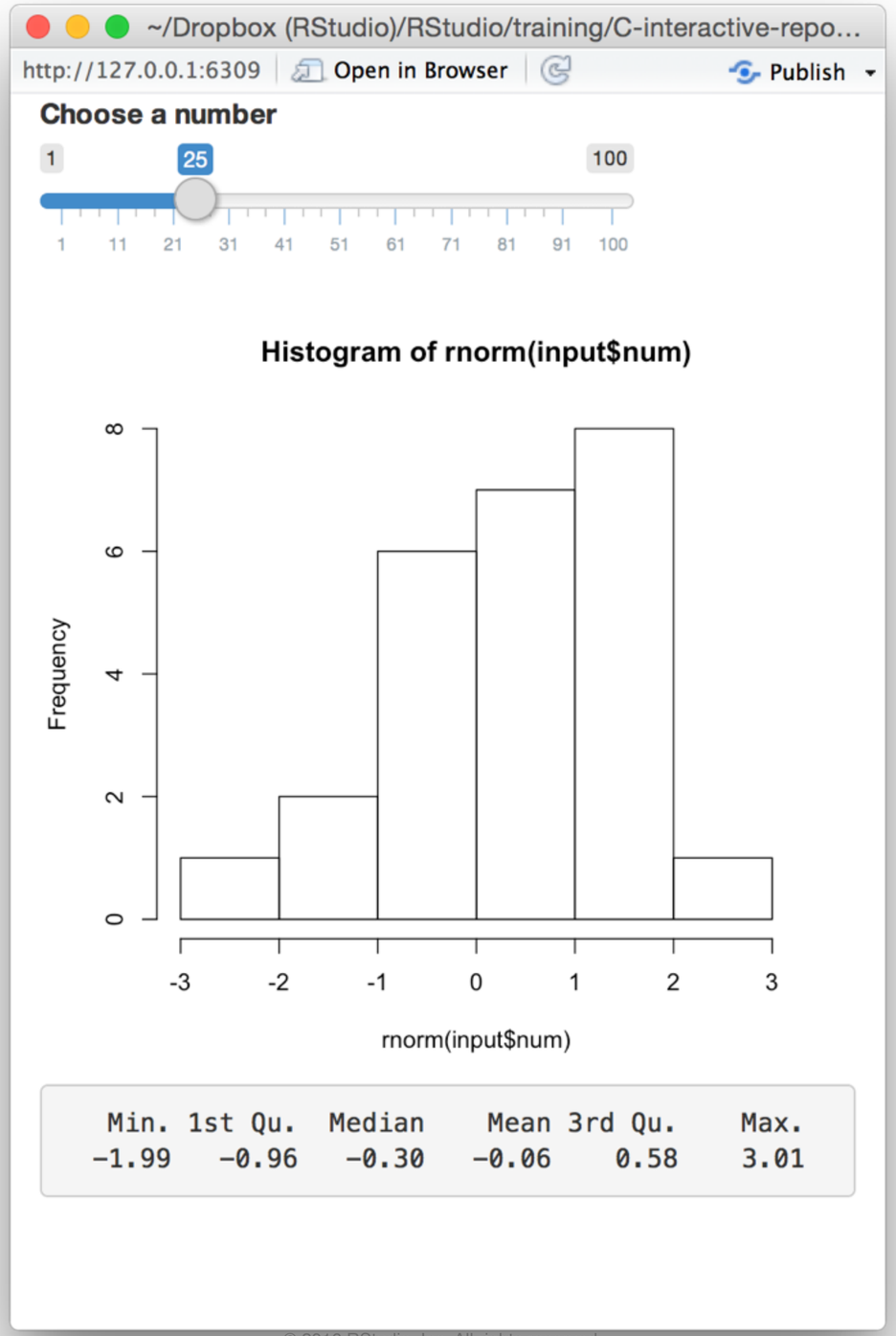
Your Turn



Use **reactive()** to pass the same data to the histogram and the summary.

Ensure that you can predict how the app will work.

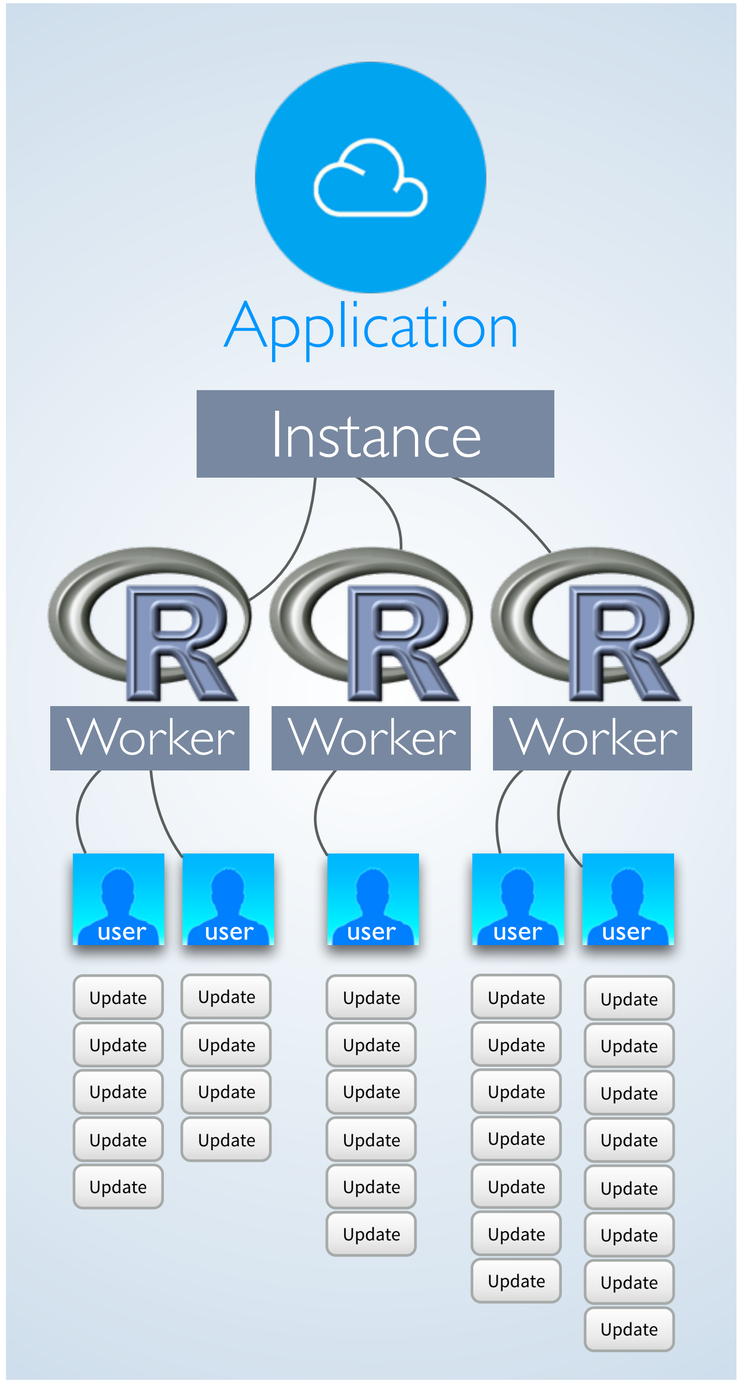
```
ui <- fluidPage(
  sliderInput("num", "Slide Me", 1, 100, 50),
  plotOutput("hist"),
  verbatimTextOutput("sum")
)
server <- function(input, output) {
   data <- reactive({ rnorm(input$num) })
   output$hist <- renderPlot({
     hist(data())
   })
   output$sum <- renderPrint({
     summary(data())
   })
}
shinyApp(ui = ui, server = server)
```



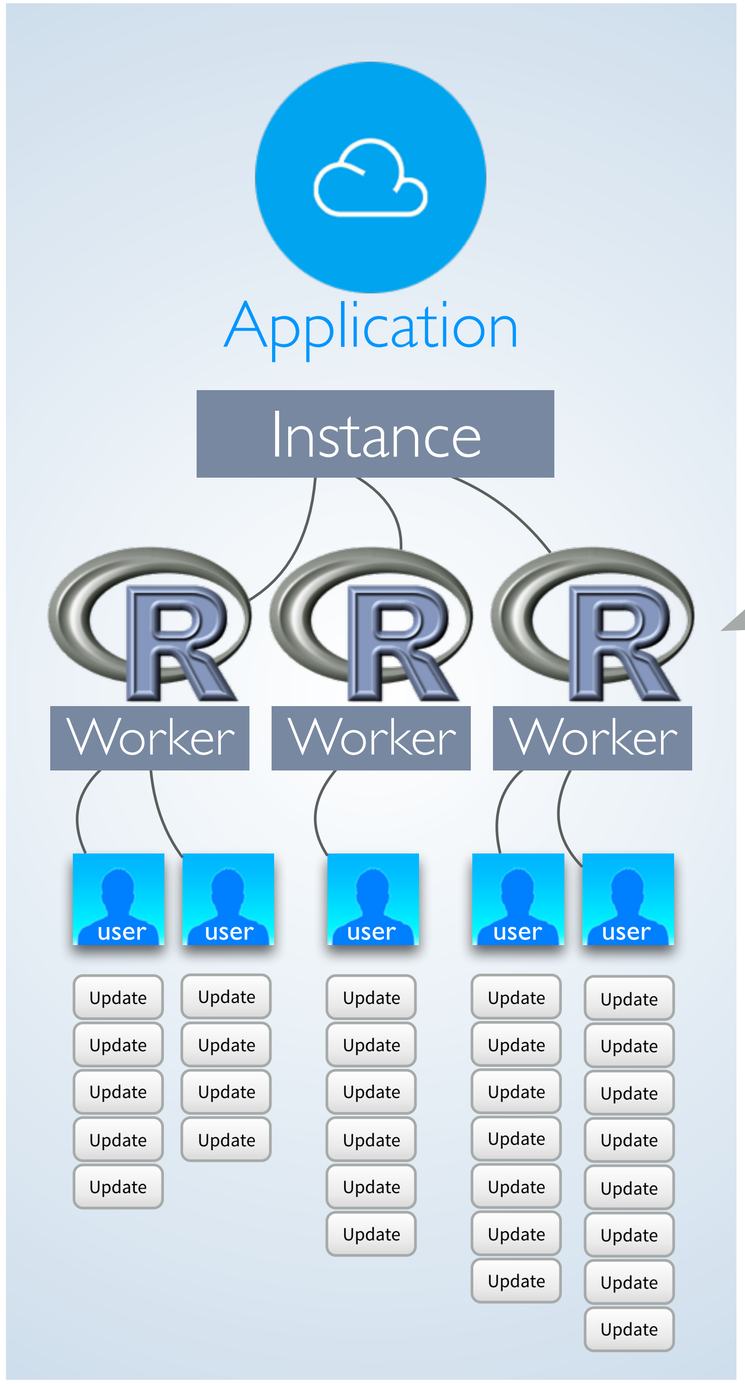
Avoid Repetition

Avoid Unnecessary Repetition

Where you place your code determines how often it will be run.

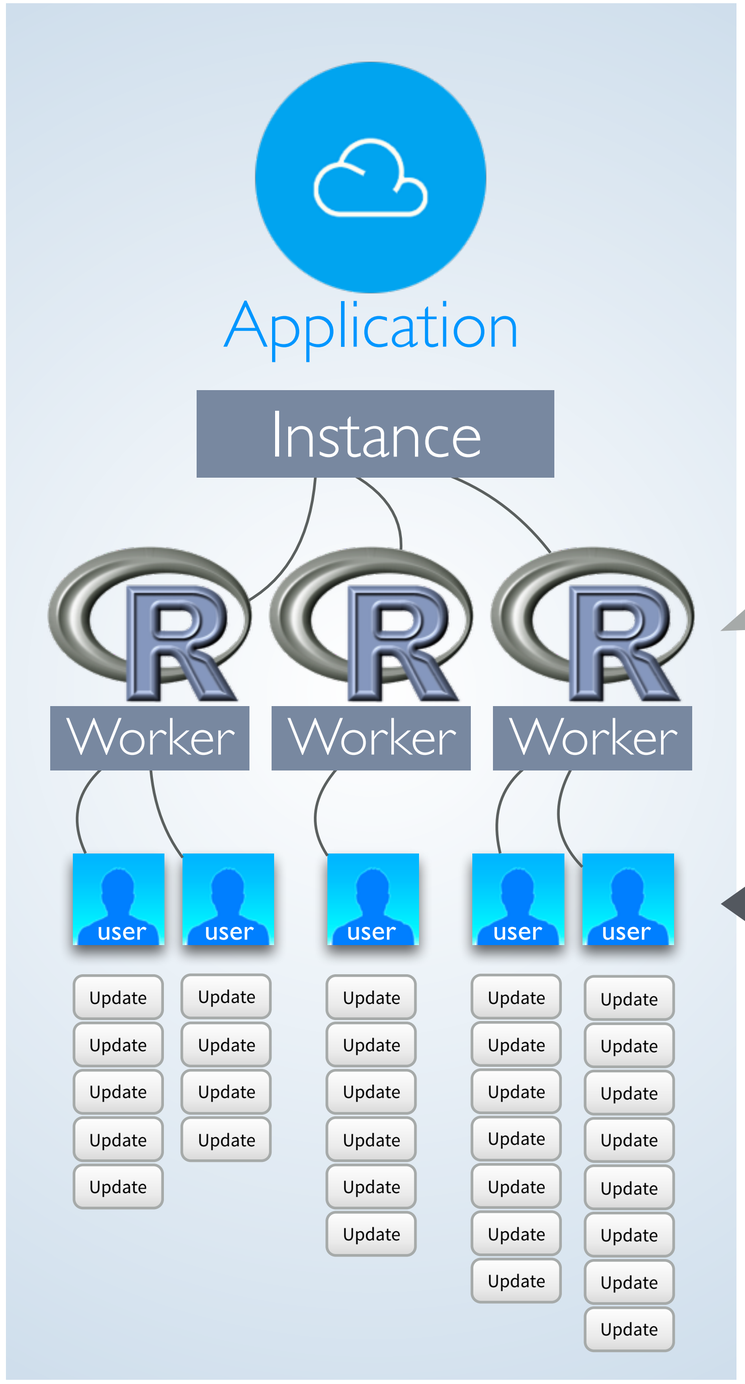



```
library(shiny)

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 worker

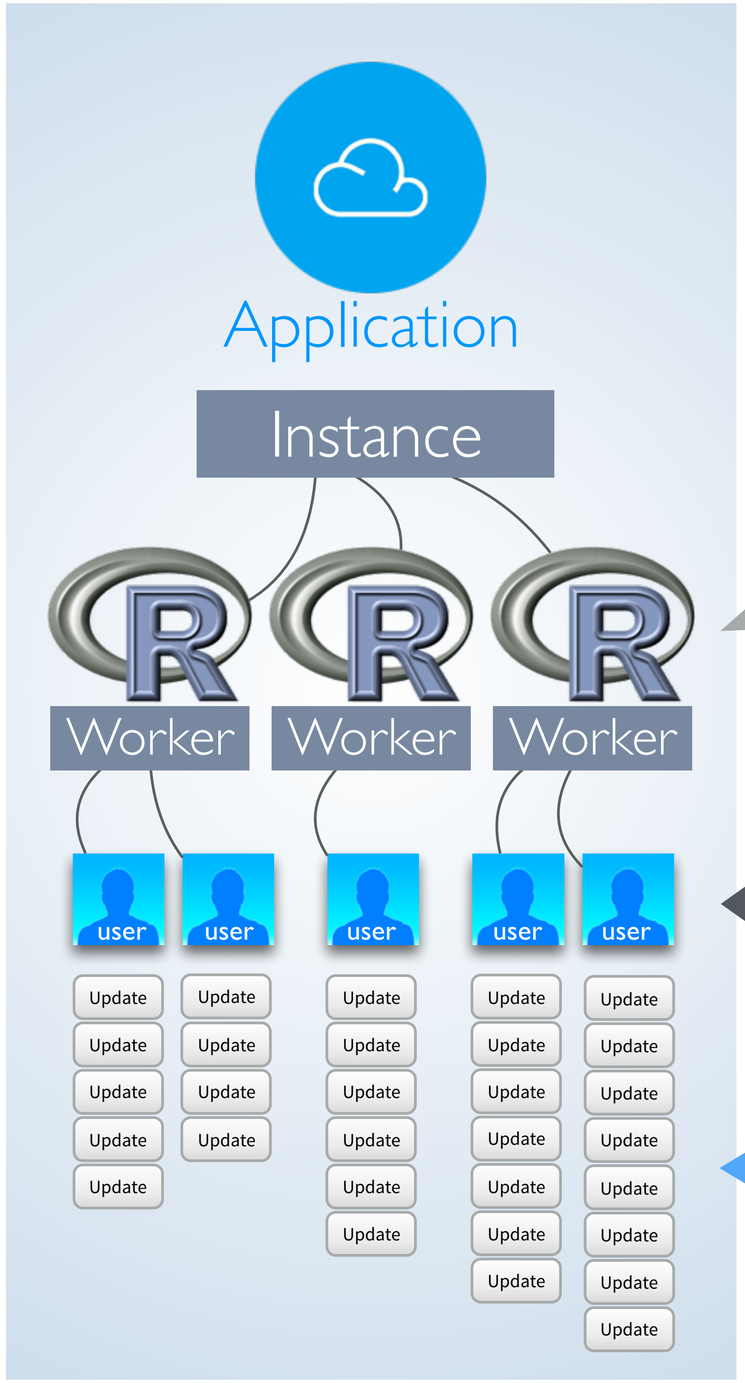
```
library(shiny)

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 worker

Code inside the server function will be run once per connection

```
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 worker

Code inside the server function will be run once per connection

Code inside of a reactive function will be run once per reaction

```
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)
```