

# Shiny : : CHEAT SHEET



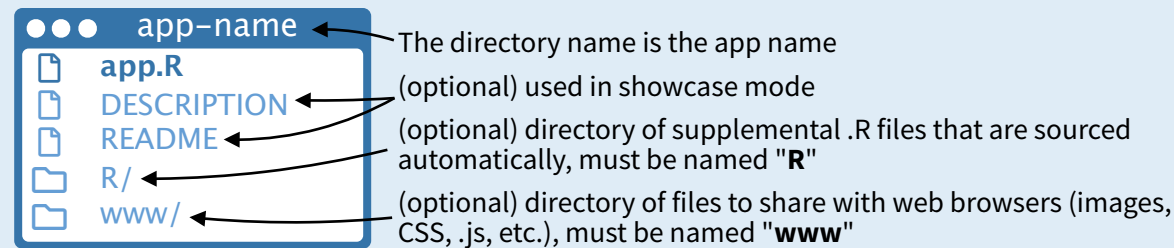
## Building an App

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).

Save your template as **app.R**. Keep your app in a directory along with optional extra files.



Launch apps stored in a directory with **runApp(<path to directory>)**.

To generate the template, type **shinyapp** and press **Tab** in the RStudio IDE or go to **File > New Project > New Directory > Shiny Application**

```
# app.R
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

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

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

Customize the UI with **Layout Functions**

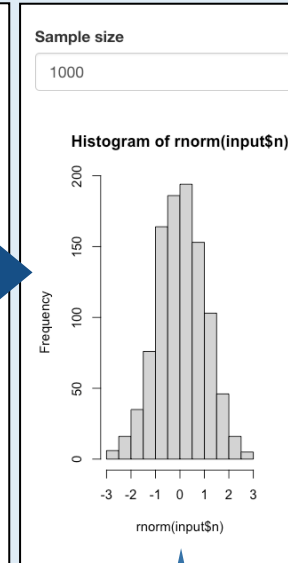
Add Inputs with **\*Input()** functions

Add Outputs with **\*Output()** functions

Wrap code in **render\*()** functions before saving to output

Refer to UI inputs with **input\$<id>** and outputs with **output\$<id>**

Call **shinyApp()** to combine **ui** and **server** into an interactive app!



See annotated examples of Shiny apps by running **runExample(<example name>)**. Run **runExample()** with no arguments for a list of example names.

## 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, width, ...)

Link

**actionLink**(inputId, label, icon, ...)

☒ Choice 1

**checkboxGroupInput**(inputId, label, choices, selected, inline, width, choiceNames, choiceValues)

☒ Choice 2

☐ Choice 3

**checkboxInput**(inputId, label, value, width)

☒ Check me

2015-05-08

**dateInput**(inputId, label, value, min, max, format, startview, weekstart, language, width, autoclose, datesdisabled, daysofweekdisabled)

June 2015

2015-05-08

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

2015-05-08

2015-05-08

Choose File

**fileInput**(inputId, label, multiple, accept, width, buttonLabel, placeholder)

1

**numericInput**(inputId, label, value, min, max, step, width)

\*\*\*\*\*

**passwordInput**(inputId, label, value, width, placeholder)

☒ Choice A

**radioButtons**(inputId, label, choices, selected, inline, width, choiceNames, choiceValues)

☐ Choice B

☐ Choice C

Choice 1

**selectInput**(inputId, label, choices, selected, multiple, selectize, width, size) Also **selectizeInput()**

Choice 1

Choice 2

0 1 2 3 4 5 6 7 8 9 10

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

0 1 2 3 4 5 6 7 8 9 10

0 1 2 3 4 5 6 7 8 9 10

Enter text

**textInput**(inputId, label, value, width, placeholder) Also **textAreaInput()**

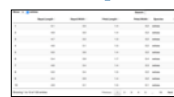
## Share

Share your app in three ways:

1. **Host it on shinyapps.io**, a cloud based service from Posit. To deploy Shiny apps:
  - Create a free or professional account at **shinyapps.io**
  - Click the Publish icon in RStudio IDE, or run: **rsconnect::deployApp("<path to directory>")**
2. **Purchase Posit Connect**, a publishing platform for R and Python. **posit.co/products/enterprise/connect/**
3. **Build your own Shiny Server** **posit.co/products/open-source/shinyserver/**

## Outputs

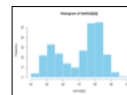
**render\*()** and **\*Output()** functions work together to add R output to the UI.



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



**renderImage**(expr, env, quoted, deleteFile, outputArgs)



**renderPlot**(expr, width, height, res, ..., alt, env, quoted, execOnResize, outputArgs)

data: Frame[1] 3 obs. of 3 variables:  
\$ Sepal.Length: num 5.1 4.9 4.7  
\$ Sepal.Width: num 5.2 5.2

**renderPrint**(expr, env, quoted, width, outputArgs)

Height.Length	Height.Width	Height.Length	Height.Width
5.1	5.2	5.2	5.2
4.9	5.2	5.2	5.2
4.7	5.2	5.2	5.2

**renderTable**(expr, striped, hover, bordered, spacing, width, align, rownames, colnames, digits, na, ..., env, quoted, outputArgs)

foo

**renderText**(expr, env, quoted, outputArgs, sep)



**renderUI**(expr, env, quoted, outputArgs)

**dataTableOutput**(outputId)

**imageOutput**(outputId, width, height, click, dblclick, hover, brush, inline)

**plotOutput**(outputId, width, height, click, dblclick, hover, brush, inline)

**verbatimTextOutput**(outputId, placeholder)

**tableOutput**(outputId)

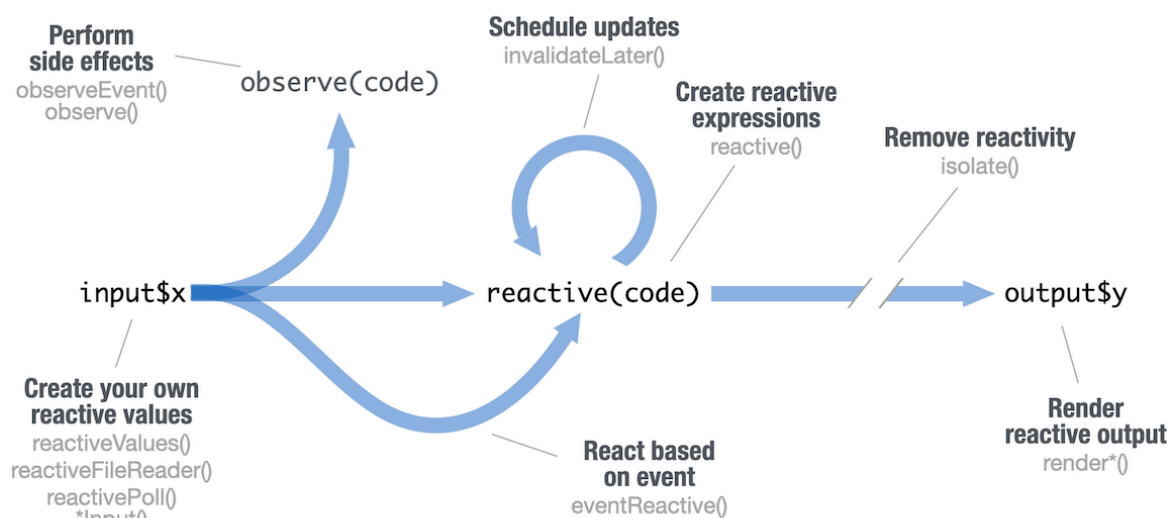
**textOutput**(outputId, container, inline)

**uiOutput**(outputId, inline, container, ...) **htmlOutput**(outputId, inline, container, ...)

These are the core output types. See **htmlwidgets.org** for many more options.

# Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error **Operation not allowed without an active reactive context.**



## CREATE YOUR OWN REACTIVE VALUES

```
# *Input() example
ui <- fluidPage(
  textInput("a", "", "A")
)
```

**\*Input() functions**  
Each input function creates a reactive value stored as **input\$<inputid>**.

```
# reactiveVal example
server <-
function(input, output){
  rv <- reactiveVal()
  rv$number <- 5
}
```

**reactiveVal(...)**  
Creates a single reactive values object.  
**reactiveValues(...)**  
Creates a list of names reactive values.

## CREATE REACTIVE EXPRESSIONS

```
ui <- fluidPage(
  textInput("a", "", "A"),
  textInput("z", "", "Z"),
  textOutput("b")
)

server <-
function(input, output){
  re <- reactive({
    paste(input$a, input$z)
  })
  output$b <- renderText({
    re()
  })
}
shinyApp(ui, server)
```

**reactive(x, env, quoted, label, domain)**  
**Reactive expressions:**  
• cache their value to reduce computation  
• can be called elsewhere  
• notify dependencies when invalidated  
Call the expression with function syntax, e.g. **re()**.

## REACT BASED ON EVENT

```
ui <- fluidPage(
  textInput("a", "", "A"),
  actionButton("go", "Go"),
  textOutput("b")
)

server <-
function(input, output){
  re <- eventReactive(
    input$go, {input$a}
  )
  output$b <- renderText({
    re()
  })
}
shinyApp(ui, server)
```

**eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, ..., label, domain, ignoreNULL, ignoreInit)**  
Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

## RENDER REACTIVE OUTPUT

```
ui <- fluidPage(
  textInput("a", "", "A"),
  textOutput("b")
)

server <-
function(input, output){
  output$b <-
    renderText({
      input$a
    })
}
shinyApp(ui, server)
```

**render\*() functions**  
Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.  
Save the results to **output\$<outputid>**.

## PERFORM SIDE EFFECTS

```
ui <- fluidPage(
  textInput("a", "", "A"),
  actionButton("go", "Go")
)

server <-
function(input, output){
  observeEvent(
    input$go, {
      print(input$a)
    }
  )
}
```

**observe(x, env)**  
Creates an observer from the given expression.  
**observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, ..., label, suspended, priority, domain, autoDestroy, ignoreNULL, ignoreInit, once)**  
Runs code in 2nd argument when reactive values in 1st argument change.

## REMOVE REACTIVITY

```
ui <- fluidPage(
  textInput("a", "", "A"),
  textOutput("b")
)

server <-
function(input, output){
  output$b <-
    renderText({
      isolate({input$a})
    })
}
shinyApp(ui, server)
```

**isolate(expr)**  
Runs a code block. Returns a **non-reactive** copy of the results.

# UI

An app's UI is an HTML document.

Use Shiny's functions to assemble this HTML with R.

```
fluidPage(
  textInput("a", "")
)
## <div class="container-fluid">
## <div class="form-group shiny-input-container">
## <label for="a"></label>
## <input id="a" type="text"
##   class="form-control" value="">
## </div>
## </div>
```

Returns HTML

**HTML** Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. **tags\$a()**. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

Run **names(tags)** for a complete list.  
**tags\$h1("Header")** -> **<h1>Header</h1>**

The most common tags have wrapper functions. You do not need to prefix their names with **tags\$**

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  br(),
  p(strong("bold")),
  p(em("italic")),
  p(code("code")),
  a(href="http://", "link"),
  HTML("<p>Raw html</p>")
)
```

**Header 1**  
**bold**  
**italic**  
**code**  
**link**  
Raw html

**CSS** To include a CSS file, use **includeCSS()**, or  
1. Place the file in the **www** subdirectory  
2. Link to it with:

```
tags$head(tags$link(rel = "stylesheet",
  type = "text/css", href = "<file name>"))
```

**JS** To include JavaScript, use **includeScript()** or  
1. Place the file in the **www** subdirectory  
2. Link to it with:

```
tags$head(tags$script(src = "<file name>"))
```

**IMAGES** To include an image:  
1. Place the file in the **www** subdirectory  
2. Link to it with **img(src="<file name>")**

# Themes

Use the **bslib** package to add existing themes to your Shiny app ui, or make your own.

```
library(bslib)
ui <- fluidPage(
  theme = bs_theme(
    bootswatch = "darkly",
    ...
  )
)
```



**bootswatch\_themes()** Get a list of themes.

# Layouts

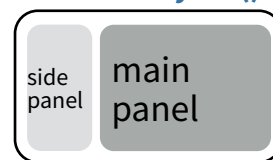
Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

```
wellPanel(
  dateInput("a", ""),
  submitButtons()
)
```

**absolutePanel()**  
**conditionalPanel()**  
**fixedPanel()**  
**headerPanel()**  
**inputPanel()**  
**mainPanel()**  
**navlistPanel()**  
**sidebarPanel()**  
**tabpanel()**  
**tabsetPanel()**  
**titlePanel()**  
**wellPanel()**

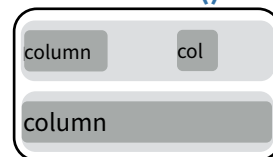
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

## sidebarLayout()



```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

## fluidRow()



```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

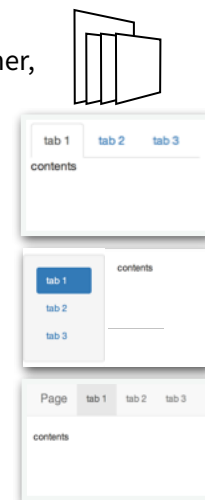
Also **flowLayout()**, **splitLayout()**, **verticalLayout()**, **fixedPage()**, and **fixedRow()**.

Layer **tabPanels** on top of each other, and navigate between them, with:

```
ui <- fluidPage(
  tabsetPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")
  )
)
```

```
ui <- fluidPage(
  navlistPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")
  )
)
```

```
ui <- navbarPage(title = "Page",
  tabPanel("tab 1", "contents"),
  tabPanel("tab 2", "contents"),
  tabPanel("tab 3", "contents")
)
```



Build your own theme by customizing individual arguments.

**bs\_theme(bg = "#558AC5", fg = "#F9B02D", ...)**

**?bs\_theme** for a full list of arguments.

**bs\_themer()** Place within the server function to use the interactive theming widget.

