

# Shiny : : CHEAT SHEET



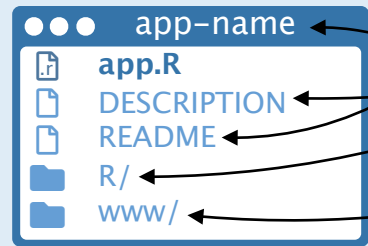
## Building an App

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.



The directory name is the app name  
 (optional) used in showcase mode  
 (optional) directory of supplemental .R files that are sourced automatically, must be named "R"  
 (optional) directory of files to share with web browsers (images, CSS, .js, etc.), must be named "www"

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

In **ui** nest R functions to build an HTML interface

Tell the **server** how to render outputs and respond to inputs with R

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

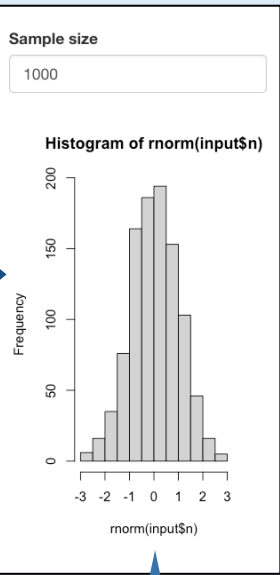
Add Inputs with **\*Input()** functions

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.

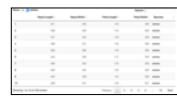
## Share

Share your app in three ways:

- Host it on shinyapps.io**, a cloud based service from RStudio. To deploy Shiny apps:
  - Create a free or professional account at [shinyapps.io](https://shinyapps.io)
  - Click the Publish icon in RStudio IDE, or run: **rsconnect::deployApp("<path to directory>")**
- Purchase RStudio Connect**, a publishing platform for R and Python. [rstudio.com/products/connect/](https://rstudio.com/products/connect/)
- Build your own Shiny Server** [rstudio.com/products/shiny/shiny-server/](https://rstudio.com/products/shiny/shiny-server/)

## Outputs

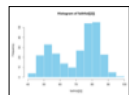
**render\*()** and **\*Output()** functions work together to add R output to the UI.



**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)



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



**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)

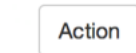
**uiOutput**(outputId, inline, container, ...)  
**htmlOutput**(outputId, inline, container, ...)

These are the core output types. See [htmlwidgets.org](https://htmlwidgets.org) for many more options.

## Inputs

Collect values from the user.

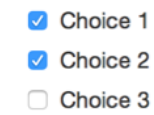
Access the current value of an input object with **input\$<inputId>**. Input values are **reactive**.



**actionButton**(inputId, label, icon, width, ...)



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



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



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



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

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



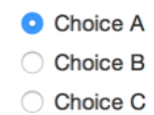
**fileInput**(inputId, label, multiple, accept, width, buttonLabel, placeholder)



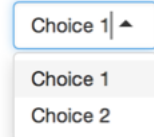
**numericInput**(inputId, label, value, min, max, step, width)



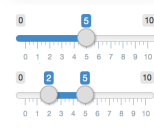
**passwordInput**(inputId, label, value, width, placeholder)



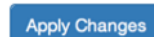
**radioButtons**(inputId, label, choices, selected, inline, width, choiceNames, choiceValues)



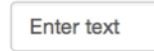
**selectInput**(inputId, label, choices, selected, multiple, selectize, width, size)  
 Also **selectizeInput()**



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



**submitButton**(text, icon, width)  
 (Prevent reactions for entire app)

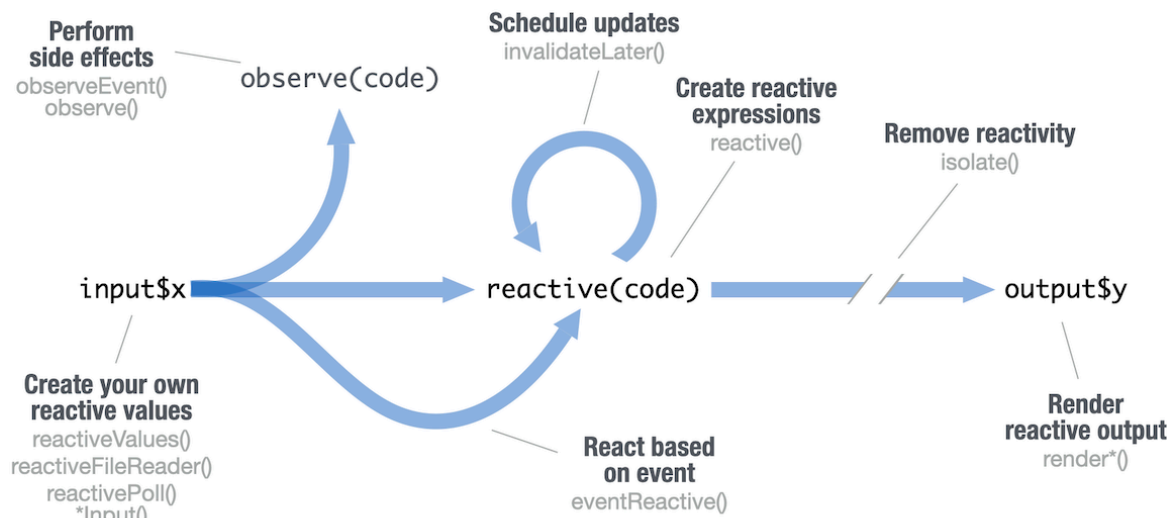


**textInput**(inputId, label, value, width, placeholder) Also **textAreaInput()**



# 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** (see front page)  
Each input function creates a reactive value stored as **input\$<inputid>**.

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

**reactiveValues(...)**  
Creates a list of reactive values whose values you can set.

## CREATE REACTIVE EXPRESSIONS

```
library(shiny)
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

```
library(shiny)
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

```
library(shiny)
ui <- fluidPage(
  textInput("a", "", "A"),
  textOutput("b")
)
server <-
function(input,output){
  output$b <-
  renderText({
    input$a
  })
}
shinyApp(ui, server)
```

**render\*() functions** (see front page)  
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

```
library(shiny)
ui <- fluidPage(
  textInput("a", "", "A"),
  actionButton("go", "Go")
)
server <-
function(input,output){
  observeEvent(input$go, {
    print(input$a)
  })
}
shinyApp(ui, server)
```

**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. See **observe()** for alternative.

## REMOVE REACTIVITY

```
library(shiny)
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>")
)
```

**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.

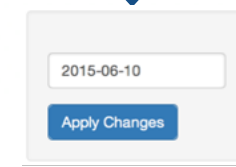
```
library(bslib)
ui <- fluidPage(
  theme = bs_theme(
    bootswatch = "darkly",
    ...
  )
)
```

**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.

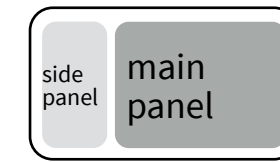
```
wellPanel(
  dateInput("a", "",
    submitButtons())
)
```



- absolutePanel()
- conditionalPanel()
- fixedPanel()
- headerPanel()
- inputPanel()
- mainPanel()
- navlistPanel()
- sidebarPanel()
- tabPanel()
- tabsetPanel()
- titlePanel()
- wellPanel()

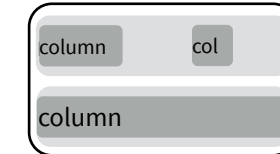
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

## sidebarLayout()



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

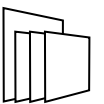
## fluidRow()



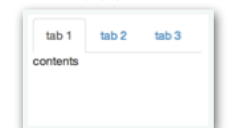
```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

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")
)
```

