

Shiny : : CHEAT SHEET



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 an app. Wrap with `runApp()` if calling from a sourced script or inside a function.

SHARE YOUR APP - in three ways:

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

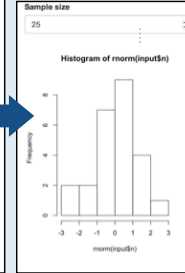
1. Refer to outputs with `output$<id>`
2. Refer to inputs with `input$<id>`
3. Wrap code in a `render*()` function before saving to output

```
library(shiny)

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

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

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

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

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

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

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

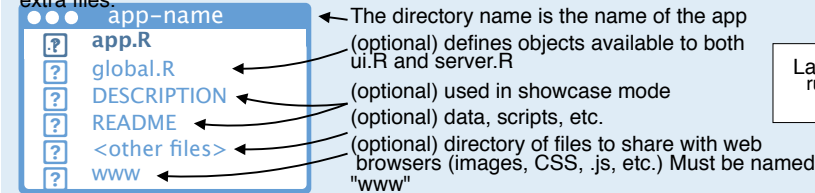
shinyApp(ui = ui, server = server)
```

`ui.R` contains everything you would save to `ui`.

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

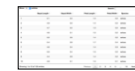
No need to call `shinyApp()`.

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



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

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



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

`renderImage(expr, env, quoted, deleteFile)`

`renderPlot(expr, width, height, res, ..., env, quoted, func)`

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

`renderTable(expr, ..., env, quoted, func)`

`renderText(expr, env, quoted, func)`

`renderUI(expr, env, quoted, func)`

`dataTableOutput(outputId, icon, ...)`

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

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

`verbatimTextOutput(outputId)`

`tableOutput(outputId)`

`textOutput(outputId, container, inline)`

`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` `checkboxGroupInput(inputId, label, choices, selected, inline)`

☒ `Choice 2` `checkboxGroupInput(inputId, label, choices, selected, inline)`

☐ `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` `radioButtons(inputId, label, choices, selected, inline)`

☐ `Choice B` `radioButtons(inputId, label, choices, selected, inline)`

☐ `Choice C` `radioButtons(inputId, label, choices, selected, inline)`

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

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

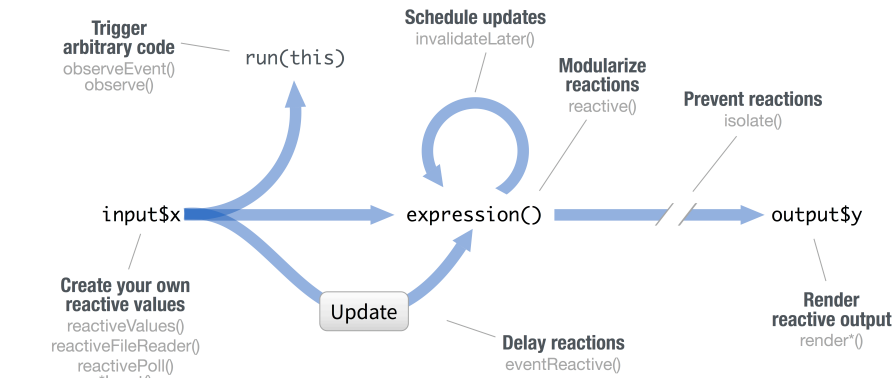
`sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)`

`submitButton(text, icon) (Prevents reactions across entire app)`

`Enter text` `textInput(inputId, label, value)`

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

```
# example snippets

ui <- fluidPage(
  textInput("a", "", "A")
)

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

*Input() functions
(see front page)

reactiveValues(...)

Each input function
creates a reactive value
stored as
input$<inputid>

reactiveValues() creates
a list of reactive values
whose values you can
set.
```

PREVENT REACTIONS

```
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.

MODULARIZE REACTIONS

```
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)
Creates a reactive
expression that
• caches its value to reduce
computation
• can be called by other code
• notifies its dependencies
when it has been invalidated
Call the expression with
function syntax, e.g. re()

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>

TRIGGER ARBITRARY CODE

```
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)
Runs code in 2nd
argument when reactive
values in 1st argument
change. See observe() for alternative.

DELAY REACTIONS

```
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)
Creates reactive
expression with code in
2nd argument that only
invalidates when
reactive values in 1st
argument change.

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.

tags\$a	tags\$data	tags\$h6	tags\$nav	tags\$span
tags\$abbr	tags\$datalist	tags\$head	tags\$noscript	tags\$strong
tags\$address	tags\$dd	tags\$headertags\$object	tags\$style	tags\$sub
tags\$area	tags\$del	tags\$hr	tags\$optgroup	tags\$tbody
tags\$article	tags\$details	tags\$hr	tags\$optgroup	tags\$tbody
tags\$aside	tags\$dfn	tags\$hr	tags\$optgroup	tags\$tbody
tags\$audio	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$b	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$base	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$bdi	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$bdo	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$blockquote	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$br	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$button	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$canvas	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$caption	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$code	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$col	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody
tags\$colgroup	tags\$div	tags\$hr	tags\$optgroup	tags\$tbody

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

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  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

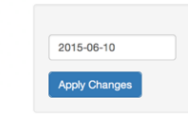
```
tags$head(tags$script(src = "<file name>"))
```

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

Layouts

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

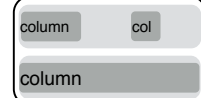
```
wellPanel(dateInput("a", ""),
  submitButton())
```



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

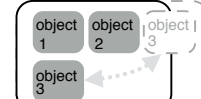
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

fluidRow()



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

flowLayout()



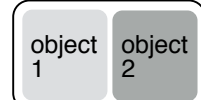
```
ui <- fluidPage(
  flowLayout( # object 1,
    # object 2,
    # object 3
  )
)
```

sidebarLayout()



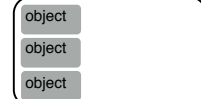
```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

splitLayout()



```
ui <- fluidPage(
  splitLayout( # object 1,
    # object 2
  )
)
```

verticalLayout()



```
ui <- fluidPage(
  verticalLayout( # object 1,
    # object 2,
    # object 3
  )
)
```

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(
  tabsetPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")
  )
)

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

