EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

Shiny:
Beyond the Basics

Workshop 2 at the EARL Conference

Date: 11 September 2018
Time: 10am – 1pm
Room: Bridge 2

@ earl-team@mango-solutions.com
☎ +44 (0)1249 705 450
mango-solutions.com

Chapter 1
Understanding Reactivity

1.1 Introduction to the Training

1.1.1 Course Aims

This course has been designed to help you learn shiny's intermediate level features. It is assumed you already have basic knowledge of shiny and have built some simple apps before.

1.1.2 Course Materials

Items appearing in this material are sometimes given a special appearance to set them apart from regular text. Here's how they look:

```
> This is a section of code                # This is a comment
```

🔴	A warning, typically describing non-intuitive aspects of the R language
🟠	A tip: additional features of R or "shortcuts" based on user experience
🟢	Exercises to be performed during (or after) the training course

1.1.3 Course Script and Exercise Answers

A great deal of code will be executed within R during the delivery of this training. This includes the answers to each exercise, as well as other code written to answer questions that arise. You can access the scripts for the course here:

https://github.com/MangoTheCat/shiny_beyond_the_basics

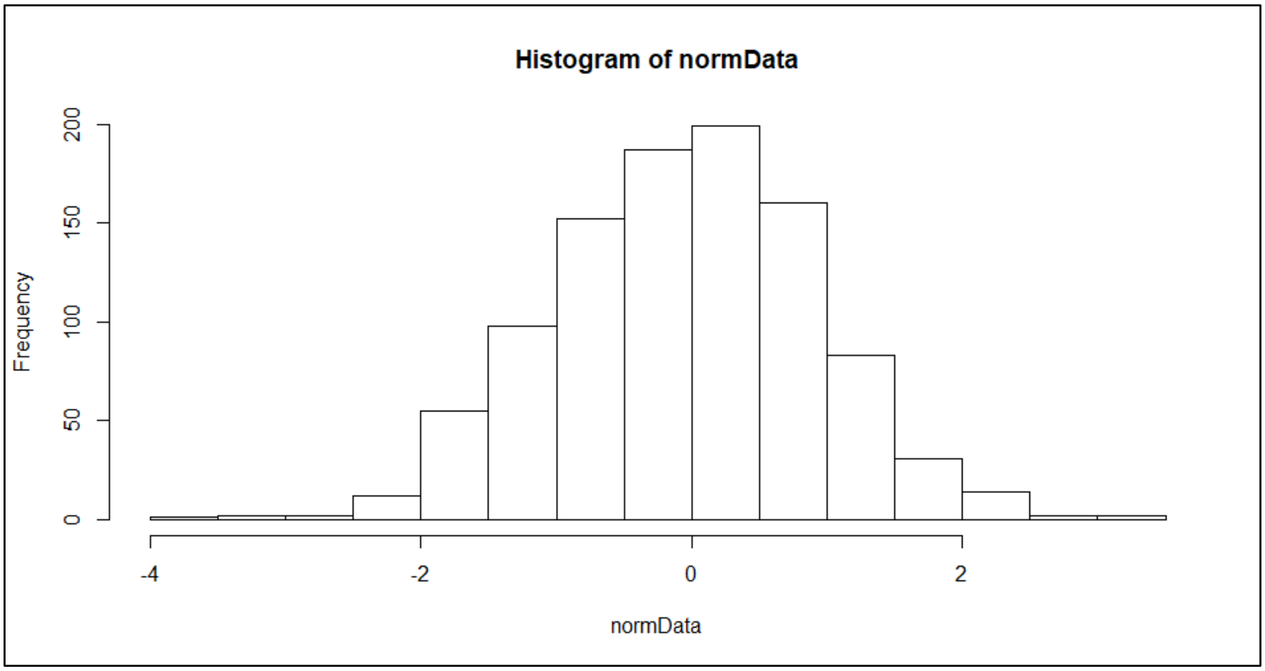
1.2 Overview

When we program in **shiny** we need to think differently about how we write our programs as we no longer have complete control over the flow of exectution -- **shiny** chooses what parts of our code to run and when. In this chapter we look at how this reactivity works and how we can control it.

1.2.1 Reactive vs. Non-reactive Programming

In normal, or "non-reactive" programming we have complete control over when code runs and in what order. For example, if we wanted to simulate 1,000 random numbers and visualise them in a histogram we run the following lines:

```
n <- 1000
hist(rnorm(n))
```



What would happen to the plot if we added a new line to update `n`?

```
n <- 1000
hist(rnorm(n))
n <- 2000
```

The answer is nothing happens to the plot. If we want to re-draw the plot based on the updated `n` we need to run the `hist` code again.

Reactive programming is different. With reactive programming we create dependencies between objects so that when one is updated the other responds. Here's the same functionality in a shiny app where the user can update `n` using an input slider.

```
library(shiny)
ui <- fluidPage(
  sliderInput("n", "Generate n randoms",
              min = 100, max = 2000, value = 1000, step = 100),
  plotOutput("histogram")
)

server <- function(input, output) {
  output$histogram <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui, server)
```

The dependency is created in the server part of the script when we refer to `input$n` within our description for how to render the `output$histogram`.

●

1. The script below is a non-reactive script for plotting a histogram of R's built-in faithful dataset of times between erruptions of the Old Faithful geyser. Convert this script into a reactive version using shiny, allowing the user to specify the number of bins for the histogram.
2. Add a `selectInput` to allow the user to change the histogram colour from three colour options.

```
# Exercise: Convert this script to a reactive version using shiny
library(ggplot2)
nBins <- 10
qplot(faithful$eruptions, bins = nBins, fill = I("orange"))
```

1.3 Reactivity

1.3.1 Shiny Works to Keep Outputs Up-to-date

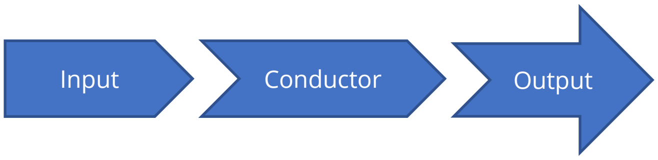
In our simplest apps, **shiny** connects our inputs (e.g. `sliderInput`, `selectInput`) to our outputs (e.g. `renderPlot`, `renderTable`) to create interactivity. When an output refers to an input as part of its code, we say the output takes a "dependency" on that input.

As our apps become more complicated, these dependencies form a network and it is shiny's job to manage this, deciding which code to run and when. Shiny works to keep all of the app's outputs up-to-date, using the dependency network to calculate what needs to run.

When a user changes an input, any objects that depend on it are notified. Shiny works to keep all of the app's outputs (e.g. plots, tables) up-to-date, so will re-run any dependencies that are out-of-date before re-rendering its outputs.

1.3.2 The Three Types of Shiny Object

The three types of objects are Inputs, Outputs, and Conductors. The most common Inputs are the input widgets (`selectInput`, `sliderInput`, etc.) and the most common Outputs are the `render*` group of functions (`renderPlot`, `renderTable`, etc.). Conductors sit inbetween Inputs and Outputs to produce intermediate calculations, with the most common being reactive expressions (created with the `reactive` function).

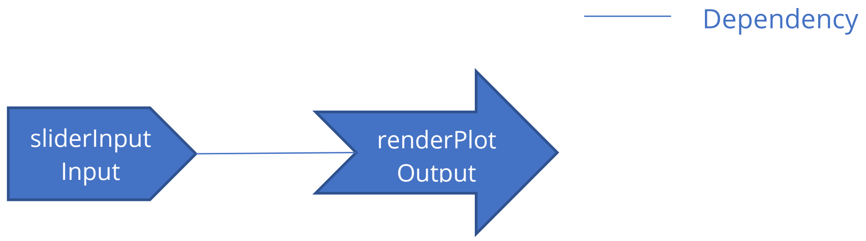


It is shiny's job to keep the Outputs up to date in response to changing Inputs. Shiny will try to be as efficient as possible, running only the code that is necessary for our apps' Outputs.

In our histogram app above we had only one Input, the slider, and one Output, the plot.

```
library(shiny)
ui <- fluidPage(
  sliderInput("n", "Generate n randoms",
              min = 100, max = 2000, value = 1000, step = 100),
  plotOutput("histogram")
)

server <- function(input, output) {
  output$histogram <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui, server)
```

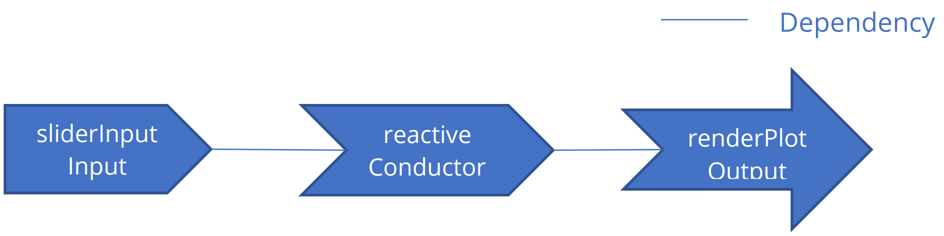
Shiny creates a dependency between the input and the output because we have referred to `input$n` as part of our definition for how to render the histogram.



Since shiny is ultimately in charge of deciding when to run our code, when we write our server code, e.g. describing how to render a plot, it is useful to think in terms of writing a *recipe* for our output. We are describing *how* to create the plot when necessary, but not *when*.

1.3.3 Adding a Conductor

Conductors are used as intermediate processing steps between Inputs and Outputs and are created using the `reactive` function. Conductors take Input values, perform a calculation with them, and then pass the results on to one or more Outputs.



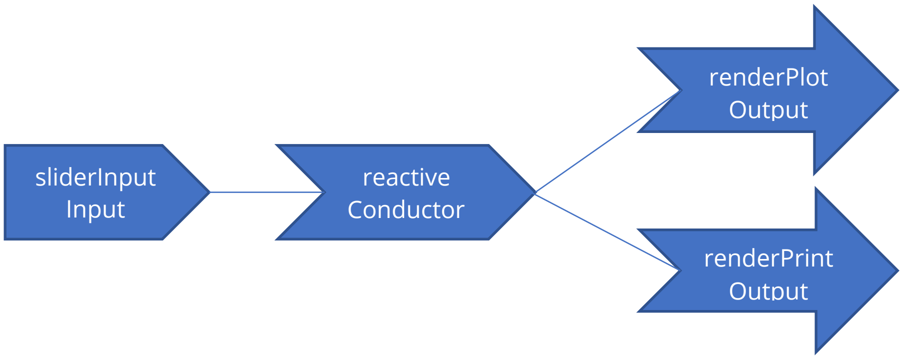
Conductors created with the `reactive` function cache their output values. When outputs request values from a reactive expression, if none of the conductor's inputs have changed it will yield the same value to the calling output.

Typically Conductors are useful when:

- We want to share the results of a calculation between multiple Outputs.
- There is a process in our app that is time consuming and we want to re-run it as little as possible.

1.3.3.1 Using a conductor to share calculations between outputs

The example below extends our histogram app to share our simulated random numbers between two Outputs: a histogram and a text summary. Both outputs use the same Conductor, defined using the `reactive` function, for the source of their data.



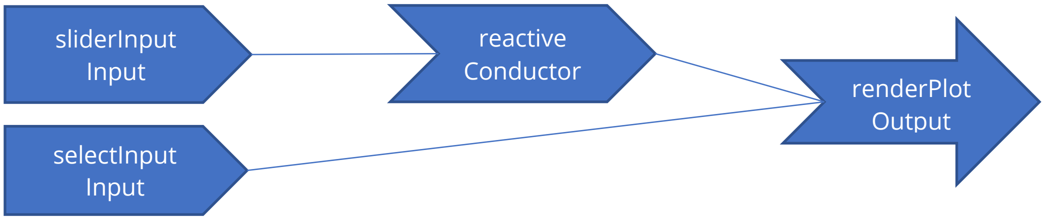
```
library(shiny)
ui <- fluidPage(
  # Input
  sliderInput("n", "Generate n randoms",
              min = 100, max = 2000, value = 1000, step = 100),
  plotOutput("histogram"),
  verbatimTextOutput("textSummary")
)

server <- function(input, output) {

  # Conductor (a reactive expression)
  normData <- reactive({
    rnorm(input$n)
  })

  # Outputs
  output$histogram <- renderPlot({
    hist(normData())
  })
  output$textSummary <- renderPrint({
    summary(normData())
  })
}
shinyApp(ui, server)
```

Both Outputs have a dependency on the `normData` reactive expression, which in turn has a dependency on the slider Input. When the user changes the Input, shiny flags it as invalidated, which in turn invalidates the conductor. The Outputs then re-run their code in order to get the updated values of their dependencies.

Since reactive expressions cache their results, when the second Output requests the latest `normData` it receives the cached copy, so both outputs are rendered using the same values.

1.3.3.2 Using a conductor to save processing

Another reason to use a Conductor in our app is to cache time-consuming calculations. The app below simulates a time-consuming calculation called `normDataSlow`, which generates random numbers as before, but much more slowly.



```
ui <- fluidPage(
  sliderInput("n", "Generate n randoms",
              min = 100, max = 2000, value = 1000, step = 100),
  selectInput("colour", "Select a colour",
              choices = c("orange", "blue", "green")),

  plotOutput("histogram")
)
server <- function(input, output) {
  # Conductor
  normDataSlow <- reactive({
    Sys.sleep(3)  # sleep for 3 seconds (simulating a slow process)
    rnorm(input$n)
  })
  # Output
  output$histogram <- renderPlot({
    hist(normDataSlow(), col = input$colour)
  })
}
shinyApp(ui, server)
```

When the user changes the colour selector, the colour of the histogram will respond immediately because the cached values from `normDataSlow` will be used. The code in `normDataSlow` will only re-run when we change the sliderInput on which it depends.

1. Open the “MPG_Model” app. This app allows the user to build a predictive model for miles per gallon (`mpg`) using their choice of predictor variables from the `mtcars` dataset and see a plot of the prediction accuracy. The app is currently slow to use as a new model is created every time the user interacts with the app, even if just changing the plotting symbol or title.
2. Add a reactive expression in order to cache the results of the model-building process. This will make the plot more responsive if the user just changes the plot title or plotting symbol.
3. Add another Output to the app that displays the residuals of the model as printed text. The residuals are the actual values minus the predicted values and can be accessed using `residuals(model)`.

Extension:

4. In the same style as the reactive network diagrams above, draw the dependency network of your app indicating which objects are Inputs, Conductors and Outputs.

1.4 Making Our Own Outputs

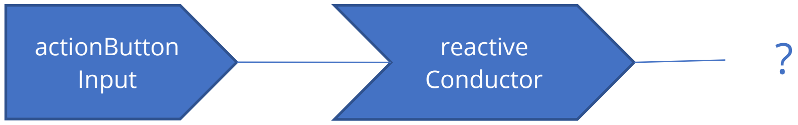
So far, we have only seen one set of output types, the `render*` family (e.g. `renderPlot`, `renderText`, etc.). Shiny will do its best to resolve any dependencies in order to service any renderable outputs in our app. However, we may want our app to produce more than just rendered results, e.g. we may want to write to a file, output a web request, or run another R script.

For a simple example, let’s say we want to output some text on the console whenever a user clicks a button. We are not rendering anything in the app, just outputting a message to the console. What happens when we click the Go button?

```
ui <- fluidPage(
  actionButton("go", "Go!")
)
server <- function(input, output) {
  reactive({
    message(paste("The Go button has the value", input$go))
  })
}
shinyApp(ui, server)
```

The answer is that nothing happens, we get no message printed to the console.

The reason is that in this app we have only an Input and a Conductor, but we have no Output. Shiny will keep the app updated to service the Outputs, so with no outputs it does nothing!



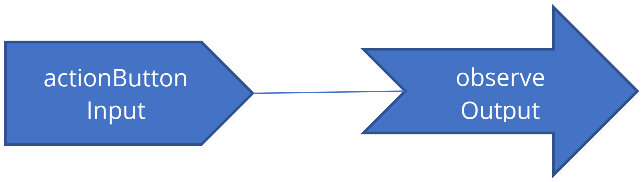
Action buttons are simple input widgets created using `actionButton(inputId, label)`. When clicked their value increments by one. The actual value the action buttons takes is not important, they just give us a mechanism to trigger reactive events.

1.4.1 Making Our Own Outputs with observe

We can define our own Outputs using the `observe` function. We could consider the `render` family of functions to be a kind of observer – they are watching for when their dependencies change, and updating themselves accordingly.

The `observe` function follows the same syntax as `reactive`, but an observer is an Output, not a Conductor. We can fix the previous example by using an `observe` instead of `reactive`.

```
ui <- fluidPage(
  actionButton("go", "Go!")
)
server <- function(input, output) {
  observe({
    message(paste("The Go button has the value", input$go))
  })
}
shinyApp(ui, server)
```



Shiny will attempt to keep the observer Output up to date with respect to its dependencies, so the message will be printed to the console every time the button is clicked.

1.4.2 When to Use Observers vs. Reactive Expressions

A common mistake in **shiny** app design is to confuse reactive expressions (`reactive`) with observers (`observe`), especially as their syntax is very similar. The key to understanding the difference is to consider *side effects*.
When we write a reactive recipe, we do it for one of two reasons:

1. We want it to perform a calculation and return the result
2. We want it to have some other effect (e.g. create a file, generate a graphic.)

In the second case, where we are not interested in a direct return value, we call this a side effect. We should:

- Use reactive expressions for performing calculations and returning the results.
- Use observers for creating side effects.

The table below shows some of the key differences between reactive expressions and observers.

Reactive Expression	Observer
Callable e.g. `buildModel()`	Not callable
Returns a value	No return value
Lazy	Eager
Cached	N/A

Using observers when we should use reactive expressions will tend to make our apps slower, as code can run more often than it needs to. Using reactive expressions when we should use observers could mean our code does not run when we want it to, like in the example app above.

1. For the following scenarios which would be better to use, a reactive expression or an observer?

Scenario	Reactive	Observer
Output a message to console on button press.		*x*
Write a dataset to disk when the user clicks “Save”.		
Return a dataset with its missing data imputed.		
Apply a user-defined filter to a dataset.		
Build a model when the user clicks “Build”.		

2. Open the Excel_to_CSV app. This app should let a user upload an Excel file, convert it to CSV format, and write it to the app’s directory as “data.csv”. However, there are a few things wrong with its reactivity.
3. When the user uploads an Excel file the conversion and writing to disk part of the app does not execute. Test this using the example Excel file within the Excel_to_CSV directory.

Extension:

4. Adjust the app’s reactivity so that the uploaded file is converted and written to disk as soon as it is uploaded.
5. Draw the dependency network of your updated app indicating which objects are Inputs, Conductors or Outputs.

1.5 Summary

- In reactive programming shiny manages dependencies between our app's objects in order to keep our app's outputs up-to-date
- The three types of reactive object are Input, Conductor and Output
- Conductors, made with the `reactive` function, appear in the reactive network between Inputs and Outputs to share calculations between Outputs and to cache time-consuming operations.
- We can create our own Outputs using `observe` in order for our app to create side effects.

Chapter 2

Controlling Reactive Depencencies

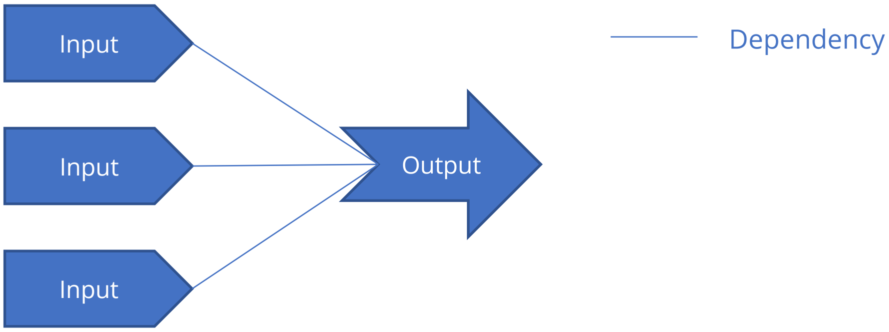
2.1 Overview

Shiny works to keep our app's outputs up to date by calculating which reactive elements need to be run based on their dependency relationships. In this chapter we look at how to exercise greater control over creating dependencies between reactive objects.

2.2 Preventing Dependencies with isolate

Whenever we refer to an input as part of our recipe to produce a particular output, we introduce a dependency relationship between the two. The output will update itself whenever the input changes. For example, in the app below we have one output histogram that depends on multiple inputs.

```
ui <- fluidPage(
  textInput("title", "Title:", value = "Random normals"),
  selectInput("colour", "Select a colour",
              choices = c("orange", "blue", "green")),
  numericInput("sampleSize", "Select size of data:",
               min = 10, max = 500, value = 100),
  plotOutput("histogram")
)
server <- function(input, output){
  output$histogram <- renderPlot(
    hist(x = rnorm(input$sampleSize),
         main = input$title,
         col = input$colour)
  )
}
```

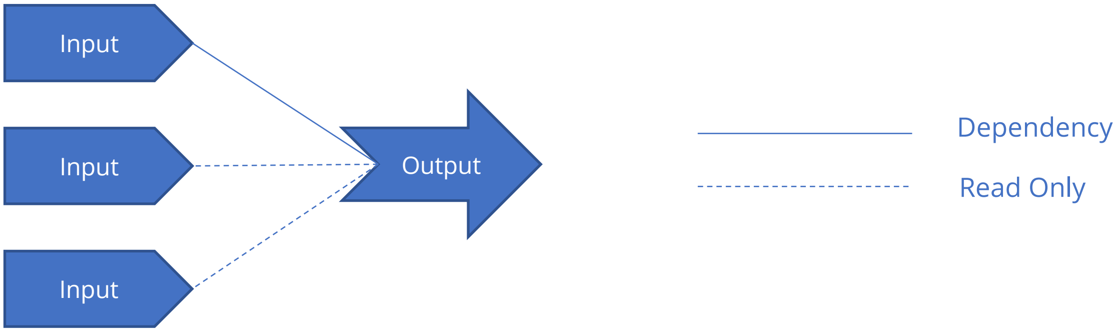


Every time any one of the inputs change the histogram will be redrawn.

We can use `isolate` to prevent these dependency relationships being created. For example, if we want the output to only trigger a refresh when the sample size input changes, we can isolate the other two input elements when we refer to them inside `renderPlot`.

```
server <- function(input, output){
  output$histogram <- renderPlot(
    hist(x = rnorm(input$sampleSize),
         main = isolate(input$title),
         col = isolate(input$colour))
  )
}
```

The plot will still use the latest values from every input widget, but the reactivity will only trigger when the non-isolated input (`input$sampleSize`) changes.



2.3 Defining Our Own Dependencies

So far we have let shiny automatically construct the dependency relationships. If we mention a reactive Input or Conductor within another reactive, a dependency will be formed.

So one approach to controlling dependencies is to allow shiny to form dependencies automatically, then turn the undesired ones off using the `isolate` function.

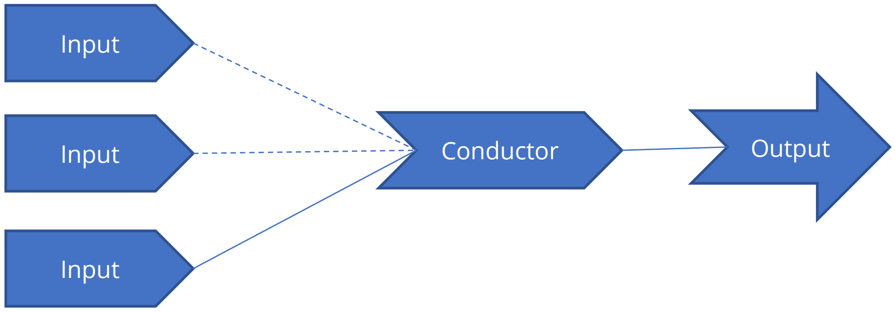
The opposite approach is to tell shiny to form no dependencies automatically and we will define the dependencies ourselves. Both functions `reactive` and `observe` have alternative versions that create dependencies in this way: `eventReactive` and `observeEvent`.

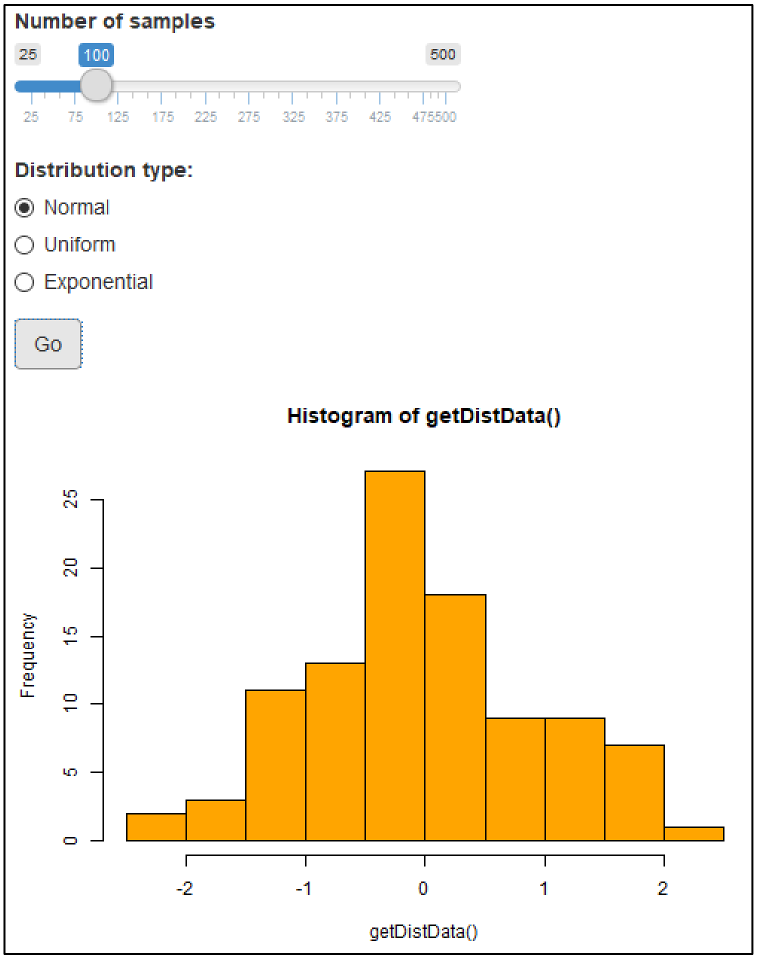
2.3.1 Defining Dependencies with eventReactive

The `eventReactive` function works identically to the `reactive` function, returning a reactive expression. The difference is that with `eventReactive` we state its dependencies explicitly in the function's first argument.

A common use case is to only have reactions trigger on a certain event, for example a button press. The app below has several inputs to a plot, but the reactive only responds when the Go button is clicked:

```
ui <- fluidPage(
  sliderInput("n", "Number of samples", min = 25, max = 500, step =
25, value = 100),
  radioButtons("dist", "Distribution type:",
               c("Normal" = "norm",
                 "Uniform" = "unif",
                 "Exponential" = "exp")),
  actionButton("go", "Go"),
  plotOutput("distPlot")
)
server <- function(input, output) {
  getDistData <- eventReactive(input$go, {
    dist <- switch(input$dist,
                   norm = rnorm,
                   unif = runif,
                   exp = rexp,
                   rnorm)
    dist(input$n)
  })
  output$distPlot <- renderPlot({
    hist(getDistData())
  })
}
```





Number of samples

25 100 500

Distribution type:

● Normal

○ Uniform

○ Exponential

In the example above, if `getDistData()` was defined using `reactive`, it would take dependencies on all of the input widgets mentioned in its recipe. Since `getDistData()` is defined using `eventReactive`, the only dependency formed is the first argument to `eventReactive`, i.e. just the Go button.

2.3.2 Defining Dependencies with observeEvent

Just as we have different ways to define the dependency graph with `reactive` and `eventReactive`, we have analogous functions for observers, namely `observe` and `observeEvent`. The `observe` function will automatically take dependencies on any reactives mentioned in its recipe, whilst `observeEvent` will only take the dependencies we state explicitly in its first argument.

The `observeEvent` function is commonly used in combination with an `actionButton` since we often only want a single dependency taken on the button.

1. Open the Dataset_Filter app and upload a test CSV file (there is an example CSV within the app's folder). This app allows the user to filter the dataset, then write the filtered data to disk as a new CSV file.
2. Enter a filter condition in the app and notice the app tries to apply it to the dataset after every keypress, breaking the app.
3. Update the `getFilteredData` reactive so that it only takes a dependency on the `applyFilter` button. The user should then be able to type a full filter condition, and have it applied only after they click "Apply filter".
4. Currently the app saves the filtered version of the CSV to "filteredData.csv" whenever the filter changes. Add a "Save" button to the app to save "filteredData.csv" only when the user clicks it.

Extension:

5. Draw a depencency diagram of your app, indicating which of reactives are Inputs, Conductors, or Outputs.

2.4 Summary

The table below summarised the differences between the different reactive functions in this chapter.

Function	Dependency Defined	Reactive Type
`observe`	Automatically	Output
`reactive`	Automatically	Conductor
`observeEvent`	Explicitly	Output
`eventReactive`	Explicitly	Conductor

- Apps usually consist of Inputs, Conductors, and Outputs, related by a network of dependencies.
- Shiny decides what code to run and when, based on the dependencies, trying to run as little as possible in order to keep the Outputs up to date.
- Most reactive objects will take dependencies on any reactives mentioned in their recipe, but we can turn off by exception using `isolate`.
- To define dependencies explicitly we can use `eventReactive` and `observeEvent`.

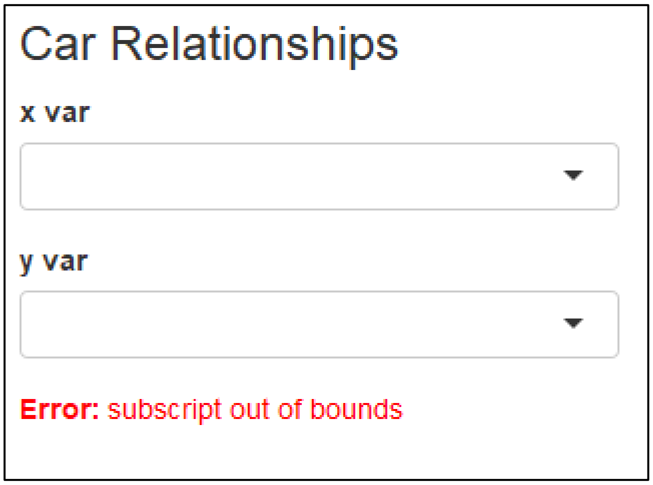
Chapter 3
Validating Inputs

3.1 Checking Input Has Been Provided

It is often the case that our app requires particular inputs to be completed before it can meaningfully produce its outputs, for example when we want to output a plot of a dataset, but the user has yet to specify the variables to plot.

Usually **shiny** will try its best to run all the reactive expressions and attempt to output something. Any errors will be presented directly to the user, usually in worrying red text. For example, here is an app that creates scatterplots of user-chosen variables where the x and y variables default to an empty string when the app is first started:

```
library(shiny)
library(ggplot2)
ui <- fluidPage(
  h3("Car Relationships"),
  selectInput("x", "x var", c("", colnames(mtcars))),
  selectInput("y", "y var", c("", colnames(mtcars))),
  plotOutput("scatter")
)
server <- function(input, output) {
  output$scatter <- renderPlot({
    ggplot(mtcars, aes_string(x = input$x, y = input$y)) +
    geom_point() + geom_smooth()
  })
}
shinyApp(ui, server)
```


Car Relationships

x var

y var

Error: subscript out of bounds

Here the recipe for rendering `output$scatter` runs and tries to make a `ggplot` with `x` and `y` of an empty string. We see the resulting error where the `plotOutput` should be.

We can fix this issue by explicitly telling the rendering function what it requires before **shiny** should attempt to run it. Here we want to require both `input$x` and `input$y` to be non-empty before shiny attempts the render. We do this using the `req` function (short for "require").

```
…
server <- function(input, output) {
  output$scatter <- renderPlot({
    req(input$x, input$y)
    ggplot(mtcars, aes_string(x = input$x, y = input$y)) +
      geom_point() + geom_smooth()
  })
}
shinyApp(ui, server)
```

The scatterplot rendering function now does not attempt to render the plot until the user chooses a value for `input$x` and `input$y`.

3.1.1 Missing Values for req

The following values count as missing when evaluated by `req`:

- `FALSE`
- `NULL`
- `""`
- An empty vector
- A vector that contains only missing values
- A logical vector that contains all FALSE or missing values
- An object of class "try-error" (see `?try`)
- A value that represents an unclicked `actionButton`

● You can use logical expressions within req if you have different criteria for missingness, e.g. `req(input$dropdown != "none")`.

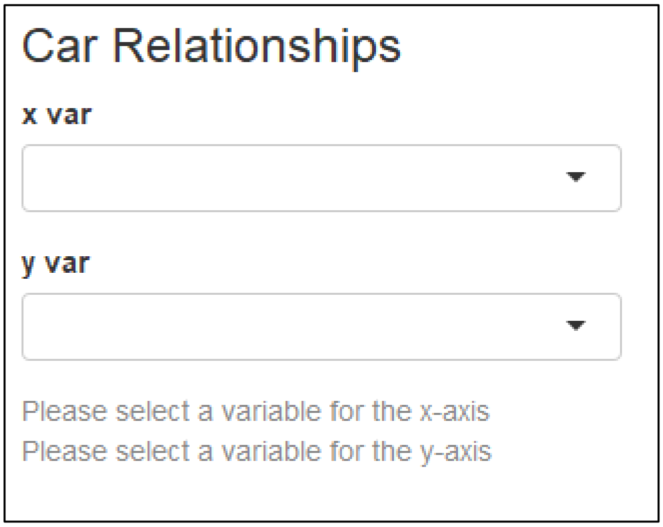
3.2 Displaying User-friendly Errors

We have seen how `req` will silently halt our reactive outputs until their requirements have been met. Sometimes it is not obvious to the user what is wrong or what they should do to fix a problem. We can use two **shiny** functions `validate` and `need` to enforce requirements as before, but this time with more informative messages.

```
…
server <- function(input, output) {
  output$scatter <- renderPlot({
    validate(
      need(input$x, "Please select a variable for the x-axis"),
      need(input$y, "Please select a variable for the y-axis")
    )
    ggplot(mtcars, aes_string(x = input$x, y = input$y)) +
      geom_point() + geom_smooth()
  })
}
shinyApp(ui, server)
```

The `validate` function accepts a list of conditions, and will prevent any outputs from executing until they are met. The conditions are each provided with a call to the `need` function, where we can specify the required reactive value, and also an informative message that the user will see if the requirement is not met.

For example, the application above will intitially look like the following:



Car Relationships

Please select a variable for the x-axis
Please select a variable for the y-axis

● The validation functions don't just apply to the `renderX` family of functions, they can be used in reactive expressions too (e.g. in a `reactive` definition).

●

1. Run the "Dataset_Pairs" app and test uploading a CSV file. Note that the app displays error messages in the outputs when no file is uploaded.
2. Fix the output text summary so that it does not display anything until a file is uploaded.
3. Fix the output plot so that it displays a "Please upload a CSV file." message until a file is uploaded.

Extension:

4. Add a further validation check to ensure the uploaded filename ends with ".csv".

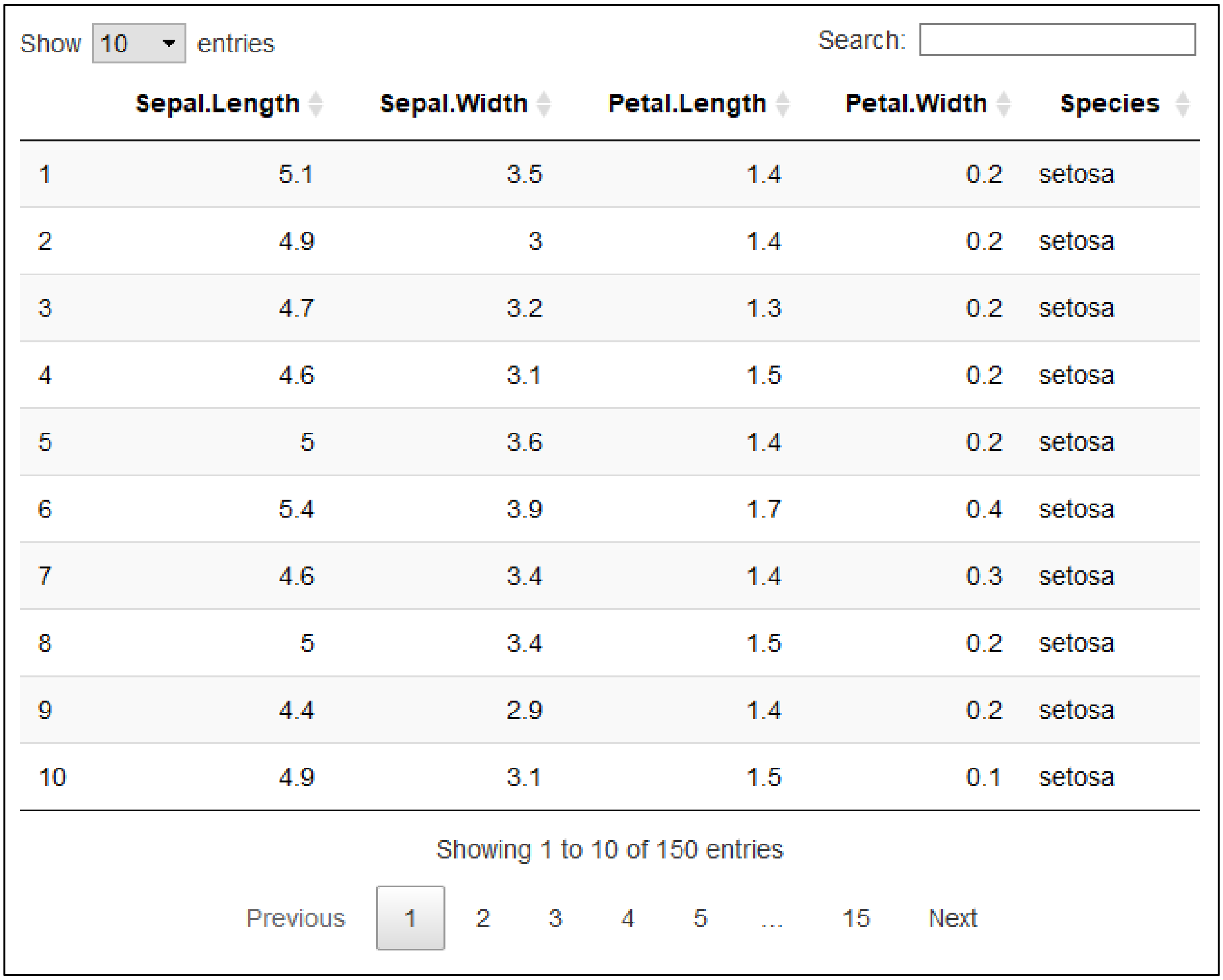
3.3 Summary

- Use `req` to prevent the reactivity of parts of your app until certain requirements are met.
- If you need to provide prompts to the user on what is missing use the `validate` and `need` functions.

Chapter 4
Customising Tables with DT

4.1 Introduction

Presenting data in tables is an extremely common task for app developers and web designers alike. Standard web libraries have grown up around this need, and the “DataTables” JavaScript library has become an extremely popular solution for presenting feature-rich tables on the web, providing features such as highlighting, sorting and searching.



Show 10 entries

Search:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa

Showing 1 to 10 of 150 entries

The **DT** package provides an R wrapper around DataTables, allowing R developers to make use of its features without needing to be a JavaScript programmer. The **DT** package also provides the usual `renderDataTable` and `dataTableOutput` functions for integration into a **shiny** app.

● The full details of the DataTable JavaScript library are available at https://datatables.net/

4.1.1 Shiny Already Has Data Tables

Shiny includes its own built-in wrapper to DataTables which you may have used already. The main functions are `renderDataTable` and `dataTableOutput`, i.e. the exact same names as provided by **DT**. However, the **DT** package provides a more up-to-date implementation of DataTables, with more features, and is the recommended way of using DataTables in your **shiny** apps.

4.1.1.1 Avoiding Name Clashes

Since both **shiny** and **DT** include `renderDataTable` and `dataTableOutput`, a common mistake is to try to use a feature only supported in **DT** but forget to load the package, or to forget to use to the fully-qualified function name, e.g. `DT::renderDataTable`. For this reason, from version 0.4, **DT** now includes the convenience functions `renderDT` and `DTOutput`.

For clarity we will use `renderDT` and `DTOutput` throughout this chapter. If you have an older version of **DT** you can use `DT::renderDataTable` or `DT::dataTableOutput` instead.

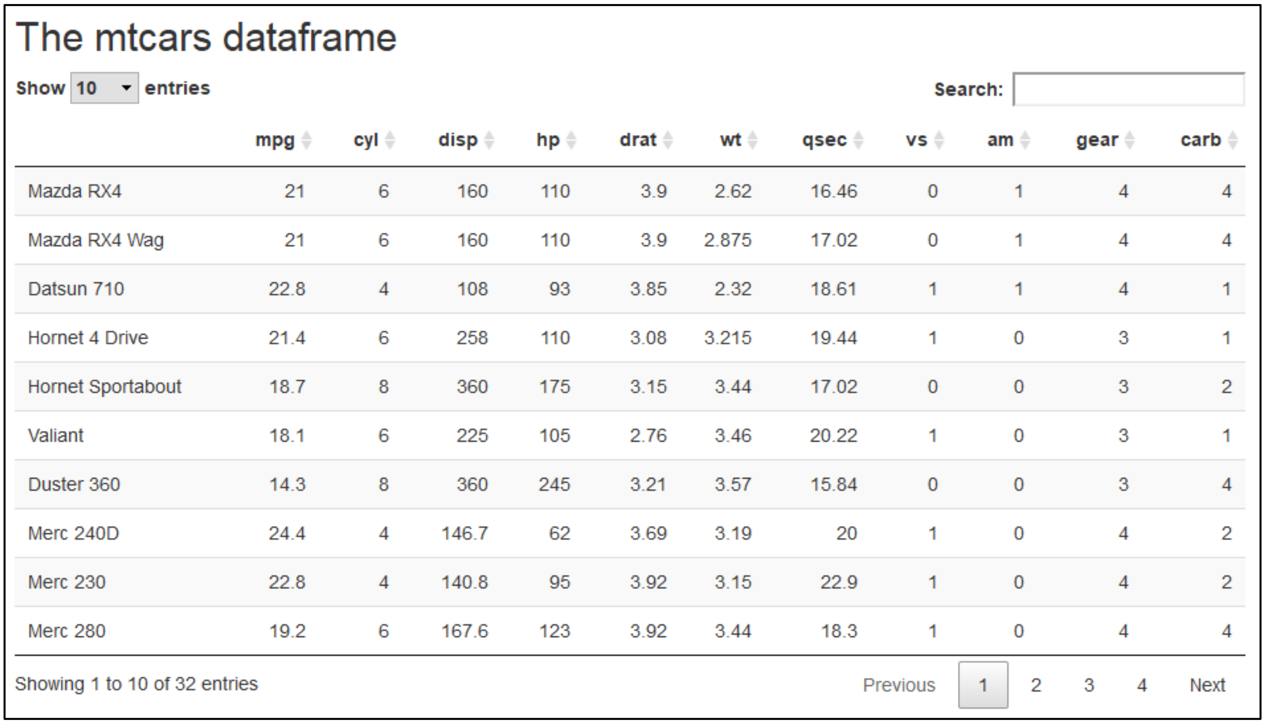
4.2 Customising Output Tables

DT provides many options for customisation, we will look at the general principles for how to customise the presentation of our tables, and illustrate this with a few examples.

4.2.1 A Minimal Table

Below is an example of a minimal shiny app showing a table output from one of R's built-in datasets.

```
library(shiny)
library(DT)

ui <- fluidPage(
  h2("The mtcars dataframe"),
  DTOutput("cars")
)
server <- function(input, output) {
  output$cars <- renderDT({ mtcars })
}
shinyApp(ui, server)
```



The mtcars dataframe

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
Duster 360	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4

Showing 1 to 10 of 32 entries

The standard data table provides pagination, column sorting (use Shift-click to sort by multiple columns), and search.

4.2.2 The datatable Function

The actual creation of the data table widget is carried out by the `datatable` function, which is called implicitly by `renderDT` when we provide a `data.frame` as the last statement in our render function. Alternatively, we can call `datatable` directly within the render function. The following two lines are equivalent:

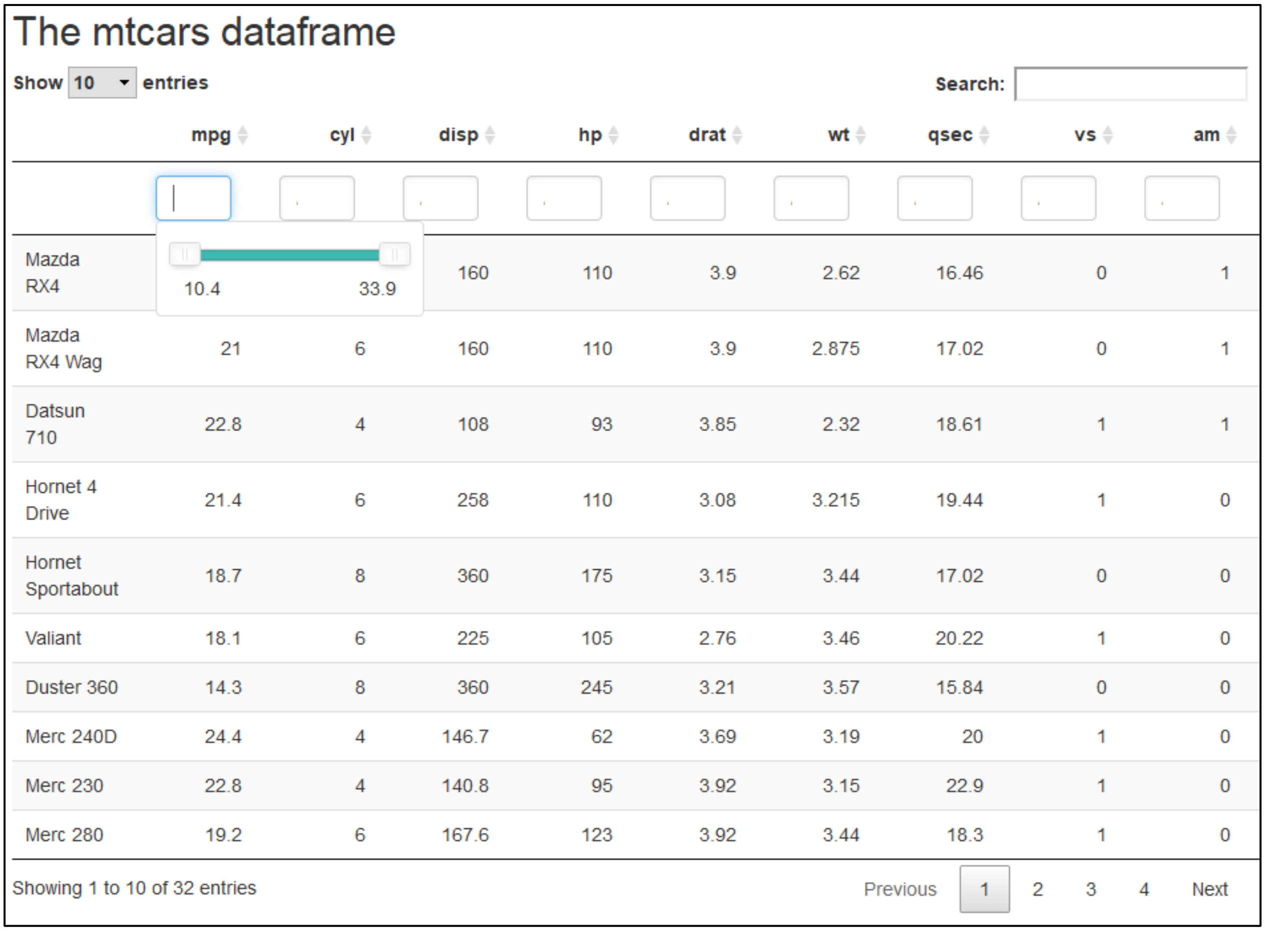
```
output$cars1 <- renderDT({ mtcars })
output$cars2 <- renderDT({ datatable(mtcars) })
```

Calling `datatable` directly is often clearer, as this is where we provide additional arguments to control the data table features. The `datatable` manual page contains the full list of configuration arguments, and we will now look at some of the more common ones.

4.2.2.1 Turning on Column Filtering

DT provides a useful feature that is turned off by default which allows the user to filter the table by entering search criteria in each column. We can turn this on using the `filter` argument:

```
library(DT)
ui <- fluidPage(
  h2("The mtcars dataframe"),
  DTOutput("cars")
)
server <- function(input, output) {
  output$cars <- renderDT({
    datatable(mtcars, filter = "top")
  })
}
shinyApp(ui, server)
```



The mtcars dataframe

Show 10 entries

Search:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am
Mazda RX4			160	110	3.9	2.62	16.46	0	1
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0	0
Valiant	18.1	6	225	105	2.76	3.46	20.22	1	0
Duster 360	14.3	8	360	245	3.21	3.57	15.84	0	0
Merc 240D	24.4	4	146.7	62	3.69	3.19	20	1	0
Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0
Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0

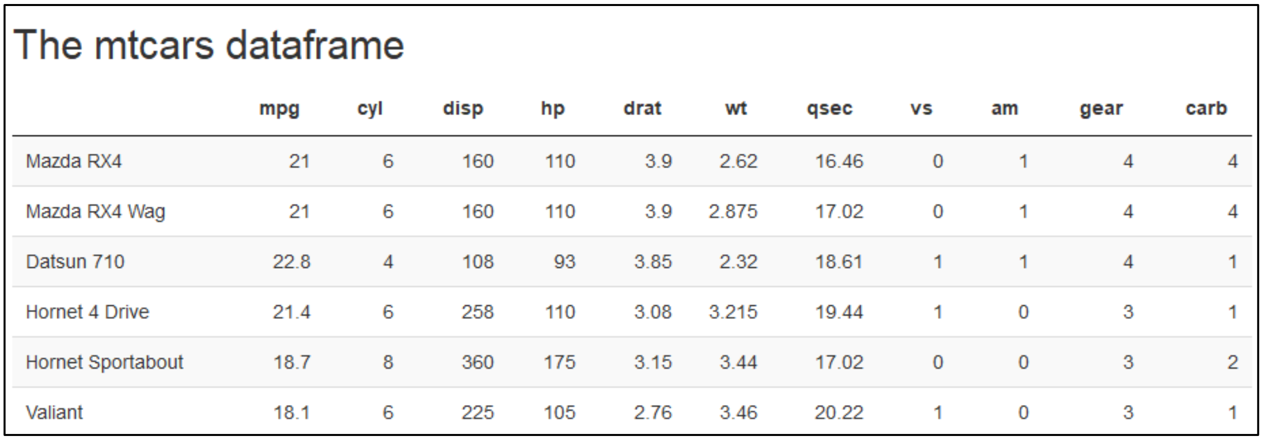
Showing 1 to 10 of 32 entries

We can control the positioning of the column filters by setting filter to “top” or “bottom”, with the default being “none”. Numeric columns display a range slider to use as a filter and text or categorical columns display a drop-down list.

4.2.2.2 Making a Simple Table

For smaller table outputs it can be useful to turn off all of the default options to present a simple table without search or pagination. Many of the configuration options are set via the `options` argument to `datatable`, where we can provide a named list of options and values.

```
server <- function(input, output) {
  output$cars <- renderDT({
    datatable(mtcars, options = list(
      paging = FALSE,
      ordering = FALSE,
      searching = FALSE,
      info = FALSE
    ))
  })
}
```



The mtcars dataframe

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1

4.2.2.3 Common Options:

Option	Description
`paging = FALSE`	Turns off pagination
`ordering = FALSE`	Prevents user from sorting by column
`searching = FALSE`	Turns off the search box
`info = FALSE`	Turns off the “Showing x entries” information
`scrollX = TRUE`	Turns on a horizontal scrollbar
`scrollY = TRUE`	Turns on a vertical scrollbar
`pageLength = 5`	Set number of records per page
`search = list(regex = TRUE)`	Turn on regular expression matching in search box

If your app has several tables with the same options you can set them once globally using `options(DT.options = list(…))` at the top of your app.R script.

1. Create an "Iris_Table" app that outputs the `iris` dataset as a data table using **DT**.
2. Turn off the options for pagination, and the ability to sort by columns.
3. Turn on column filtering so the filters appear at the top of the table (remember this is a separate argument to `datatable`).

For more `datatable` options see https://rstudio.github.io/DT/options.html

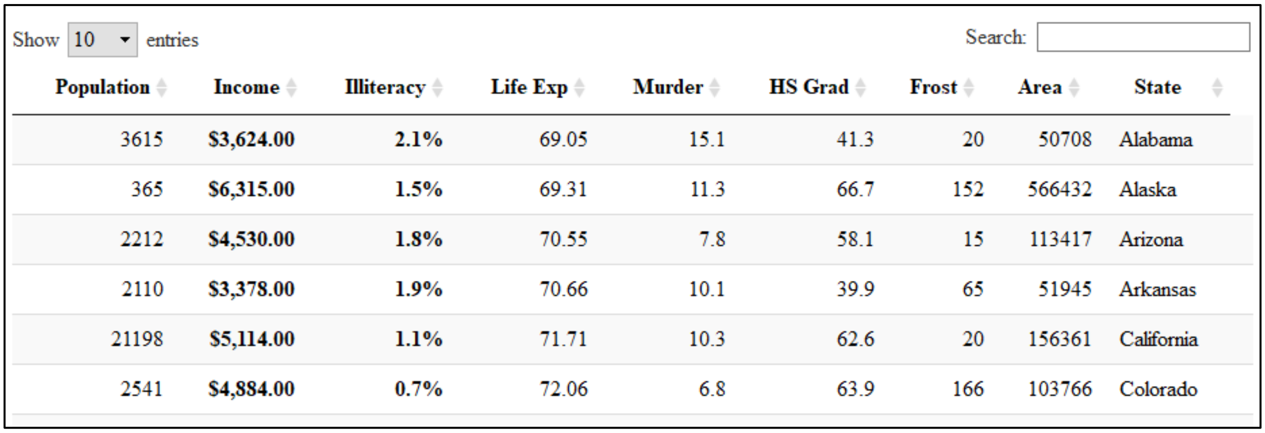
4.2.3 Formatting Columns

The DT package contains several helper functions to format columns, for example to render the data with a currency symbol, or to a certain number of decimal places. The full list of functions is as follows.

Format function	Description
`formatCurrency`	Add currency symbol, interval marks, rounding
`formatString`	Add prefix or suffix to strings
`formatPercentage`	Add percentage symbol, interval marks, rounding
`formatRound/formatSignif`	Round values to decimal places or significant figures
`formatDate`	Format dates

We can apply these formatting functions in a chain of operations, starting with the `datatable` object and piping it through each formatting function with the `%>%` pipe operator. For example:

```
library(magrittr)
stateInfo <- as.data.frame(state.x77, row.names = FALSE)
stateInfo$State <- row.names(state.x77)
stateInfo$Illiteracy <- stateInfo$Illiteracy / 100
datatable(stateInfo, rownames = FALSE) %>%
  formatCurrency("Income", currency = "$") %>%
  formatPercentage("Illiteracy", digits = 1)
```



Show 10 entries

Search:

Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	State
3615	**$3,624.00**	**2.1%**	69.05	15.1	41.3	20	50708	Alabama
365	**$6,315.00**	**1.5%**	69.31	11.3	66.7	152	566432	Alaska
2212	**$4,530.00**	**1.8%**	70.55	7.8	58.1	15	113417	Arizona
2110	**$3,378.00**	**1.9%**	70.66	10.1	39.9	65	51945	Arkansas
21198	**$5,114.00**	**1.1%**	71.71	10.3	62.6	20	156361	California
2541	**$4,884.00**	**0.7%**	72.06	6.8	63.9	166	103766	Colorado

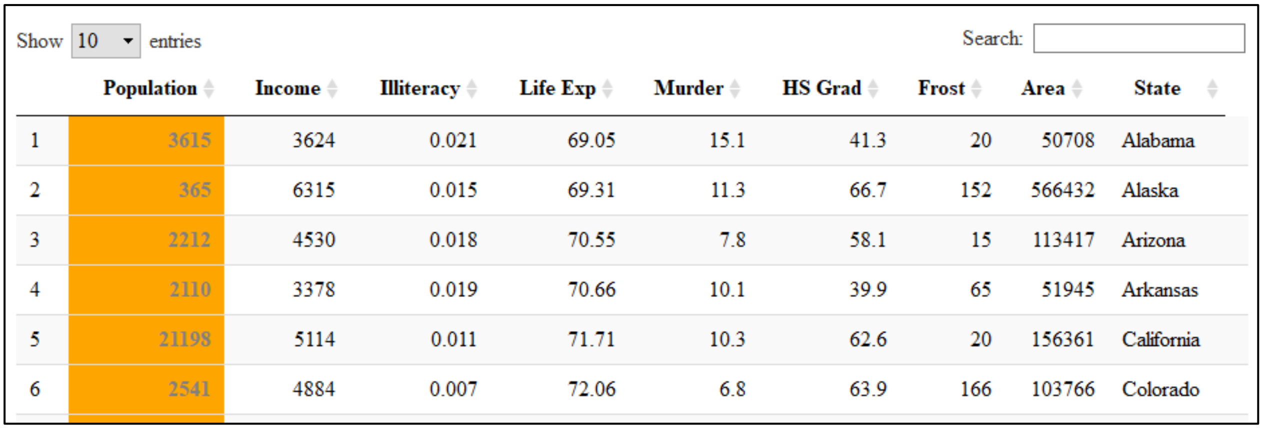
Note the `datatable` object is an html widget, not a `data.frame`, so any calculations on the data need to be carried out before converting it with `datatable` for presentation.

Calling `datatable` from the console will render the table in RStudio's Viewer tab. This is a quick way of testing your formatting functions without running your **shiny** app.

4.2.4 Styling Columns

In addition to formatting columns, DT provides a formatStyle function for applying styling to columns. For example, we can change text colour, font weight, and cell background colour:

```
datatable(stateInfo) %>%
  formatStyle("Population", color = "grey",
              fontWeight = "bold",
              backgroundColor = "orange")
```



Show 10 entries

Search:

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	State
1	3615	3624	0.021	69.05	15.1	41.3	20	50708	Alabama
2	365	6315	0.015	69.31	11.3	66.7	152	566432	Alaska
3	2212	4530	0.018	70.55	7.8	58.1	15	113417	Arizona
4	2110	3378	0.019	70.66	10.1	39.9	65	51945	Arkansas
5	21198	5114	0.011	71.71	10.3	62.6	20	156361	California
6	2541	4884	0.007	72.06	6.8	63.9	166	103766	Colorado

We can provide more than one column name to formatStyle, e.g. `c("Population", "Illiteracy")` to apply the same formatting to multiple columns. Use the `names` function to apply formatting to every column, for example:

```
datatable(stateInfo) %>%
  formatStyle(names(stateInfo), color = "grey")
```

The formatStyle function applies standard CSS styles to the table elements. For a full list of style options consult the excellent CSS tutorial at www.w3schools.com/css/.

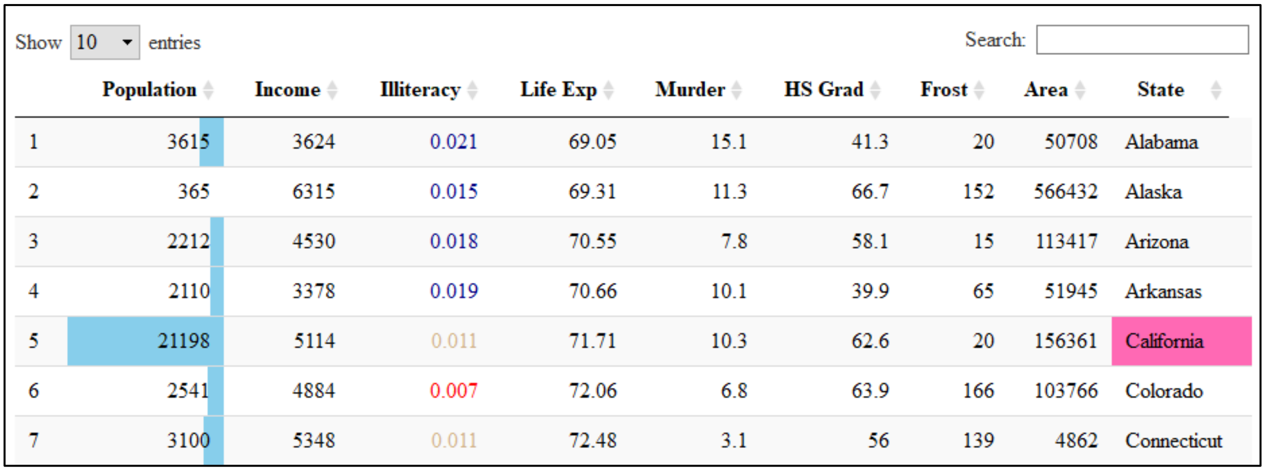
4.2.4.1 Applying Conditional Formatting

A powerful feature of data tables is to format cells according to their contents. DT provides several helper functions to do this, used in conjunction with `formatStyle`:

Conditional Format function	Cell Style
`styleInterval(cuts, values)`	According to their interval membership
`styleEqual(levels, values)`	One-to-one mapping of values to styles
`styleColorBar(data, color, angle)`	Background colour barchart. "data" provides the range of values to scale the bars.

We select the columns and the style property we wish to set using `formatStyle` as before, but we use one of the `style` functions to calculate the values:

```
datatable(stateInfo) %>%
  formatStyle("Illiteracy",
           color = styleInterval(cuts = c(0.008, 0.011),
                                 values = c("red","tan","navy"))) %>%
  formatStyle("Population",
           background = styleColorBar(data = stateInfo$Population,
                                      color = "skyblue")) %>%
  formatStyle("State",
           backgroundColor = styleEqual("California", "hotpink"))
```



Show 10 entries

Search:

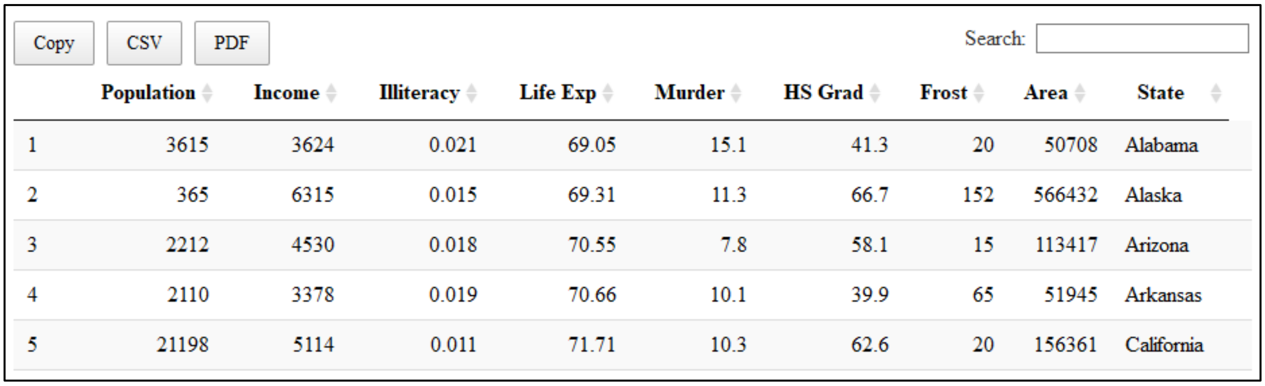
	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	State
1	3615	3624	0.021	69.05	15.1	41.3	20	50708	Alabama
2	365	6315	0.015	69.31	11.3	66.7	152	566432	Alaska
3	2212	4530	0.018	70.55	7.8	58.1	15	113417	Arizona
4	2110	3378	0.019	70.66	10.1	39.9	65	51945	Arkansas
5	21198	5114	0.011	71.71	10.3	62.6	20	156361	California
6	2541	4884	0.007	72.06	6.8	63.9	166	103766	Colorado
7	3100	5348	0.011	72.48	3.1	56	139	4862	Connecticut

4.2.5 *Extensions*

The DT package contains several extensions to DataTable where contributors have added particularly useful extended functionality. However, not all extensions work well together, hence them not being included as standard in default data tables.

Extensions can be activated for a table using the `extensions` argument, with each extension usually having configuration settings passed as usual through the `options` list. For example we can change our table to add the "Buttons" extension that allows users to copy the table to the clipboard, or download it in a number of formats:

```
datatable(stateInfo,
          extensions = "Buttons",
          options = list(
            dom = "Bfrtip",
            buttons = c("copy","csv", "pdf"))
)
```



	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	State
1	3615	3624	0.021	69.05	15.1	41.3	20	50708	Alabama
2	365	6315	0.015	69.31	11.3	66.7	152	566432	Alaska
3	2212	4530	0.018	70.55	7.8	58.1	15	113417	Arizona
4	2110	3378	0.019	70.66	10.1	39.9	65	51945	Arkansas
5	21198	5114	0.011	71.71	10.3	62.6	20	156361	California

You may find that the buttons do not work in RStudio's Viewer, but will work in your **shiny** app when it is opened in a full browser.

For more extensions see https://rstudio.github.io/DT/extensions.html.

1. Create a "Swiss_Table" app that outputs the `swiss` dataset as a data table using **DT**.
2. Format each column as a percentage, rounded to one decimal place.
3. Add conditional formatting to all the numeric columns so that they show a barchart in the background, scaled to the range of the whole dataset. Hint: use `range(swiss)` to get the min and max values.

Extension:

4. Add an extension to allow the user to copy the table to the clipboard easily and download the table as a PDF.

4.3 Using Tables as Inputs

It can sometimes be useful to use data tables to collect information from users, rather than just display it. For example, a user could click on a row of data to prompt the app to show further information about it, or click on individual cells to mark them as outliers.

Usually our input widgets are accessible from our server by accessing the input object, e.g. `input$sampleSize` to obtain the value of an `inputSlider`. Data tables are no different, they provide a number of `input` objects that we can respond to in the server. For a DT created using `DTOutput(outputId = "tableId")` the following inputs are available:

Input object	Description
`input$tableId_columns_selected`	The currently selected columns
`input$tableId_rows_selected`	The currenctly selected rows
`input$tableId_cells_selected`	The currently selected cells
`input$tableId_cell_clicked`	Information about the cell being clicked
`input$tableId_rows_current`	Indices of rows on the current page
`input$tableId_rows_all`	Indices of rows on all pages (after filtering)
`input$tableId_search`	The search string
`input$tableId_search_columns`	Vector of applied column filters

4.3.1 Controlling Selection Options

Note a data table does not allow all types of selection by default, for example you can select rows but not columns. Selection behaviour can be turned on and off via the selection argument to datatable, for example to turn on column selection:

```
renderDT({
  datatable(state.x77, selection = list(target = "column")) })
```

To turn on both row and column selection we can use:

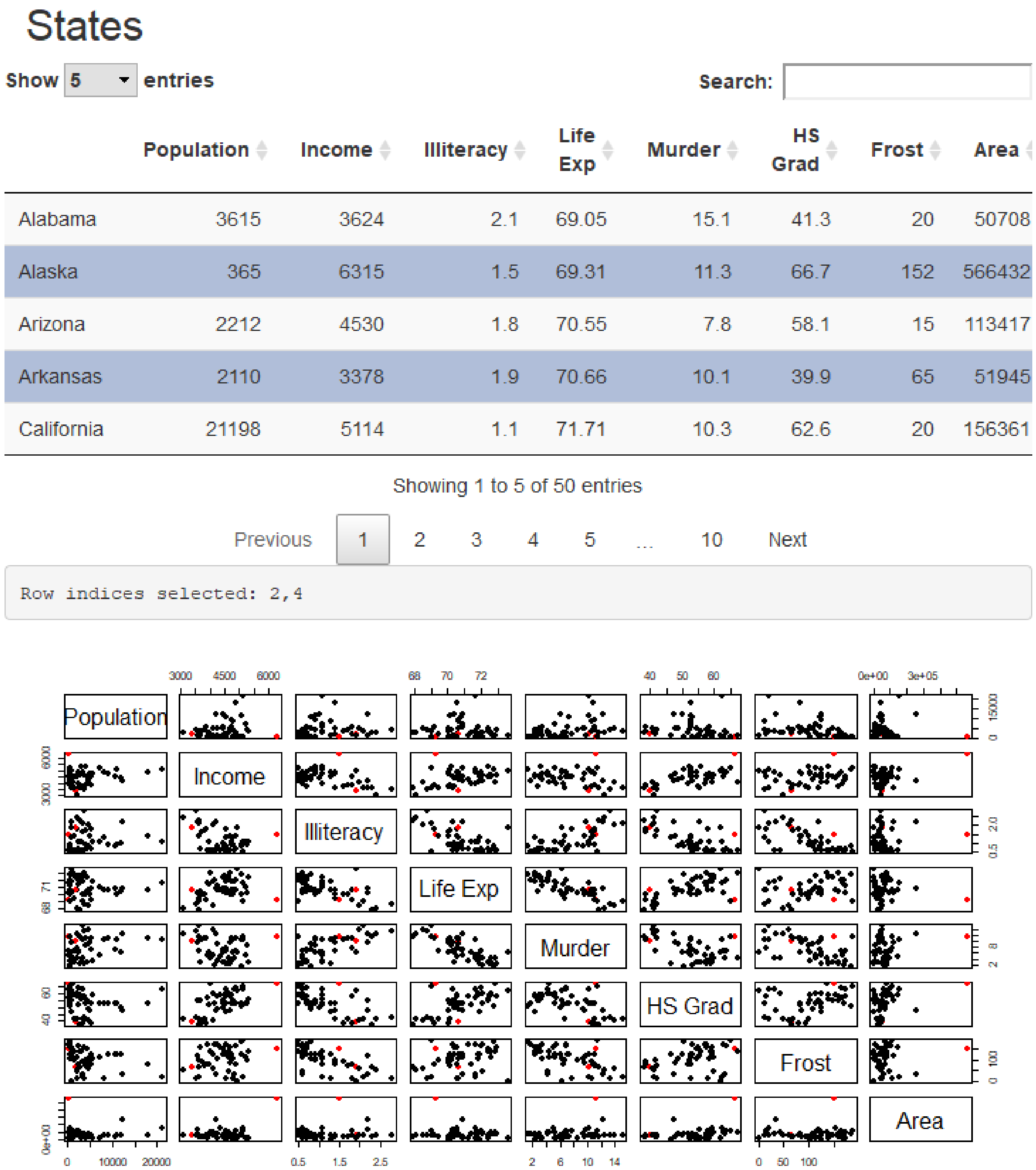
```
renderDT({
  datatable(state.x77, selection = list(target = "row+column")) })
```

4.3.2 Using Datatable Selections

An important detail to remember about the data provided by the DT `input` objects on the server is usually they are providing *indices* of the dataset, not the data themselves. For example, `input$mytable_rows_selected` might take the values `c(1, 2, 3)` if the first three rows of the table were selected, it would not return the row data itself.

We will now make use of some of the table inputs in **shiny** application to explore the `states.x77` dataset. We will use both selected columns and selected rows to control what is output on a scatterplot.

```
ui <- fluidPage(
  titlePanel(title = "States"),
  fluidRow(DTOutput("states")),
  fluidRow(verbatimTextOutput("info")),
  fluidRow(plotOutput("plot"))
)
server <- function(input, output) {
  output$states <- renderDT({
    datatable(state.x77, selection = list(target = "row"))
  })
  output$info <- renderPrint({
    cat("Row indices selected: ")
    cat(input$states_rows_selected, sep = ",")
  })
  output$plot <- renderPlot({
    cols <- rep("black", nrow(state.x77))
    cols[input$states_rows_selected] <- "red"
    pairs(state.x77, col = cols, pch = 19)
  })
}
```



States

Show 5 entries

Search:

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Alabama	3615	3624	2.1	69.05	15.1	41.3	20	50708
Alaska	365	6315	1.5	69.31	11.3	66.7	152	566432
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417
Arkansas	2110	3378	1.9	70.66	10.1	39.9	65	51945
California	21198	5114	1.1	71.71	10.3	62.6	20	156361

Showing 1 to 5 of 50 entries

Previous 1 2 3 4 5 … 10 Next

```
Row indices selected: 2,4
```

1. Create a “Swiss_Relationships” app that outputs the `swiss` dataset as a data table using **DT**.
2. Turn on column and row selection for the data table.
3. Add an output scatter plot that shows the relationship between the first two columns of the dataset (Fertility and Agriculture).
4. Allow the user to select rows of the table and see those provinces highlighted on the plot

Extension:

5. Allow the user to pick any variable to plot (not just Fertility and Agriculture) by selected two columns in the data table. Hint, if the user selects more than two columns, just use the first two for the scatter plot.