Tutorial:

Evaluate values from dynamically produced UI modules

Aaron Clark, Biogen

Agenda

The Problem

A Use Case

Dissect some Code

Conclusions

The Problem

There are a couple things that are difficult about this talk's title:

{Evaluate user inputs from dynamically produced UI Modules}

The big three:
(that I'll cover at least)

- **Generating dynamic UI**
- **Bringing user inputs together in parent module(s)**
- **Evaluating the result**

Our Goal

Let’s learn via an example that accomplishes all three

Use Case

shinyNewColumns

- **Experimental R package available on GitHub**
- **Exports one shiny module for users to include in existing apps**
- **Handy UI to help's users create new variables using simple or advance techniques**
- **Evaluates user inputs into {dplyr} on the fly to create new columns in initial data source**

60 Second Demo

Take it for a spin yourself:

Bit.ly/shinyNewColumns

App Structure

Internal modular structure

app.R

launchModal

newCol

rangeConditions

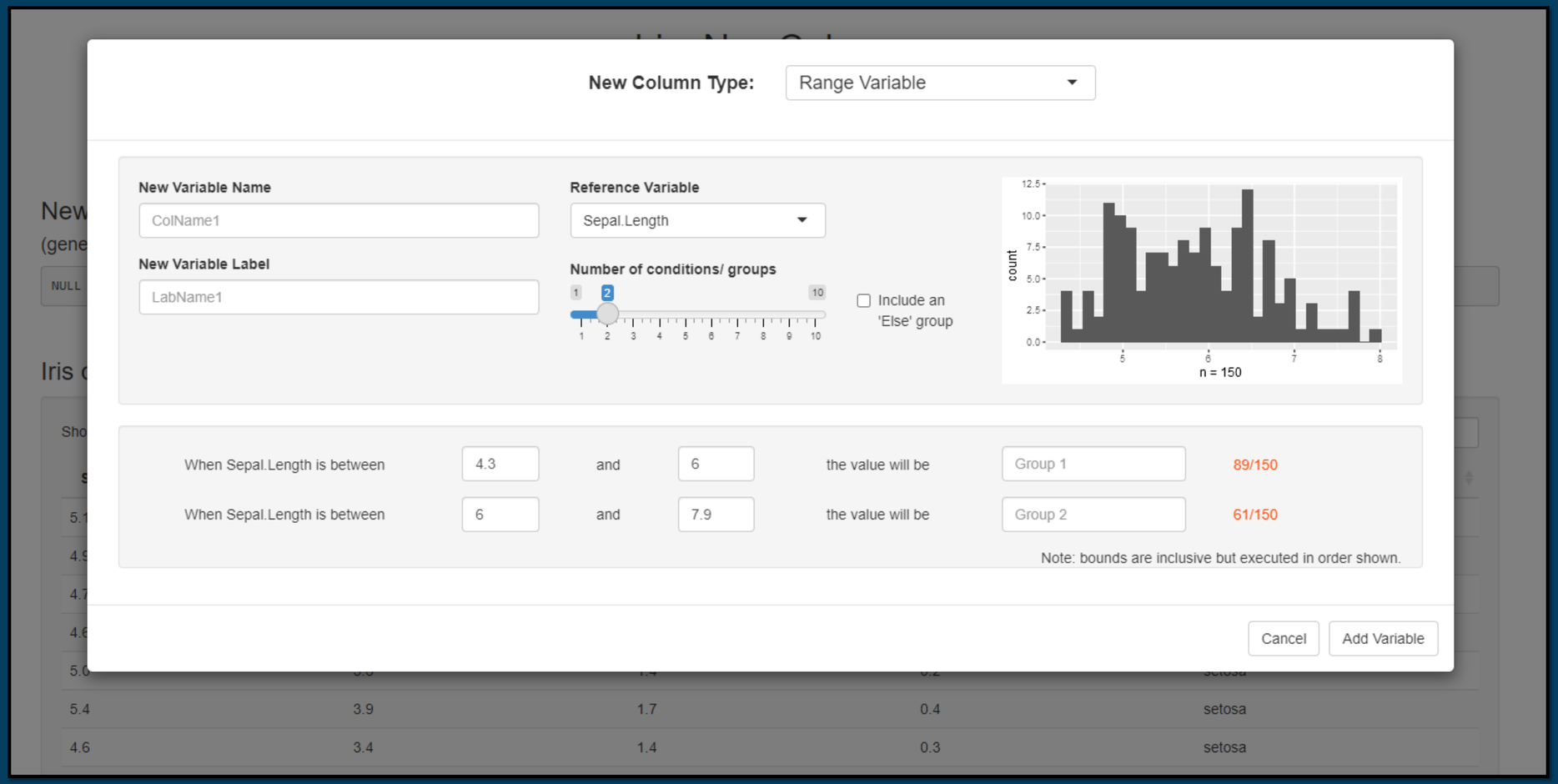
Modules as seen in the app

app.R

launch Modal

newCol

Range Conditions



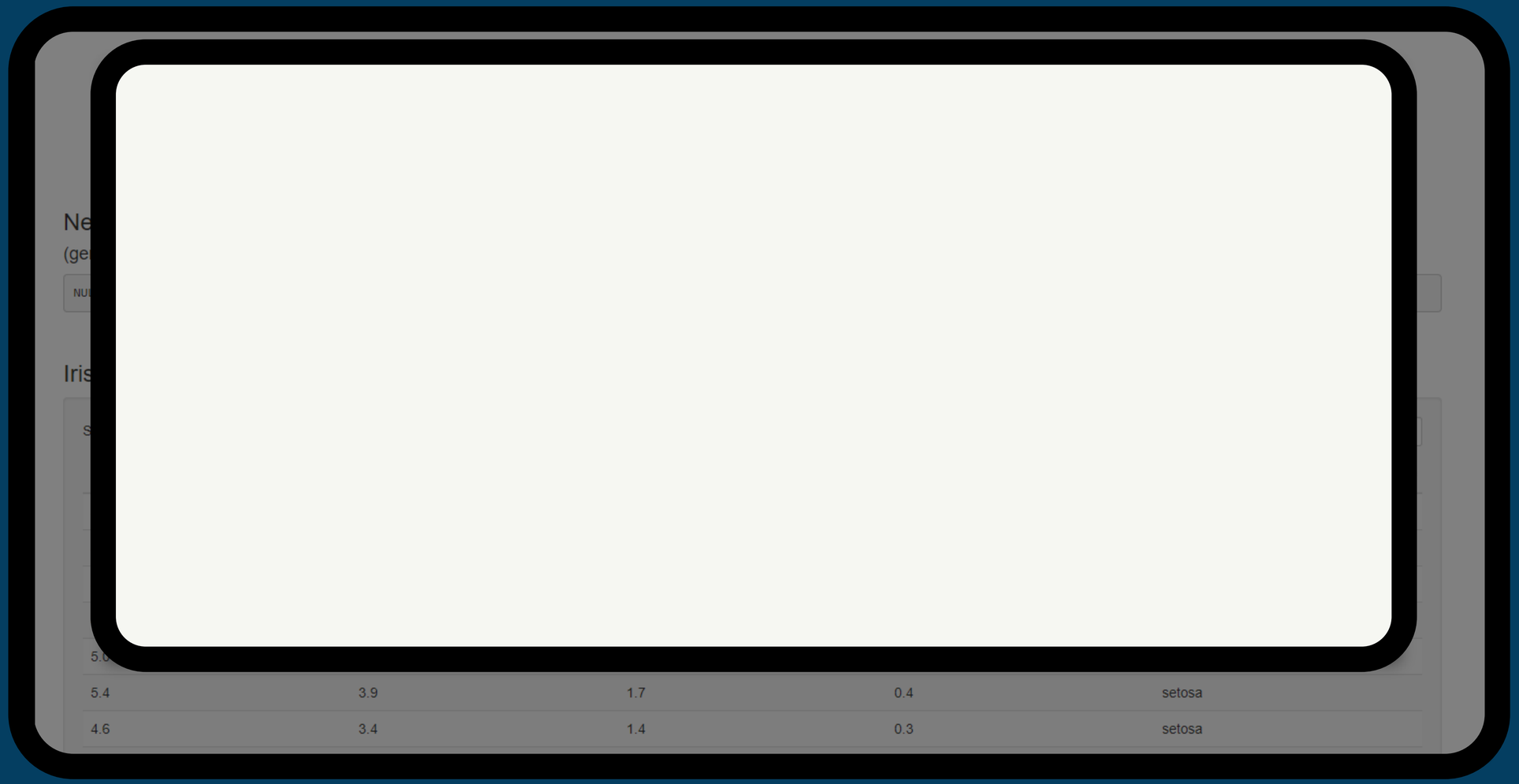
Modules as seen in the app

app.R

launch Modal

newCol

Range Conditions



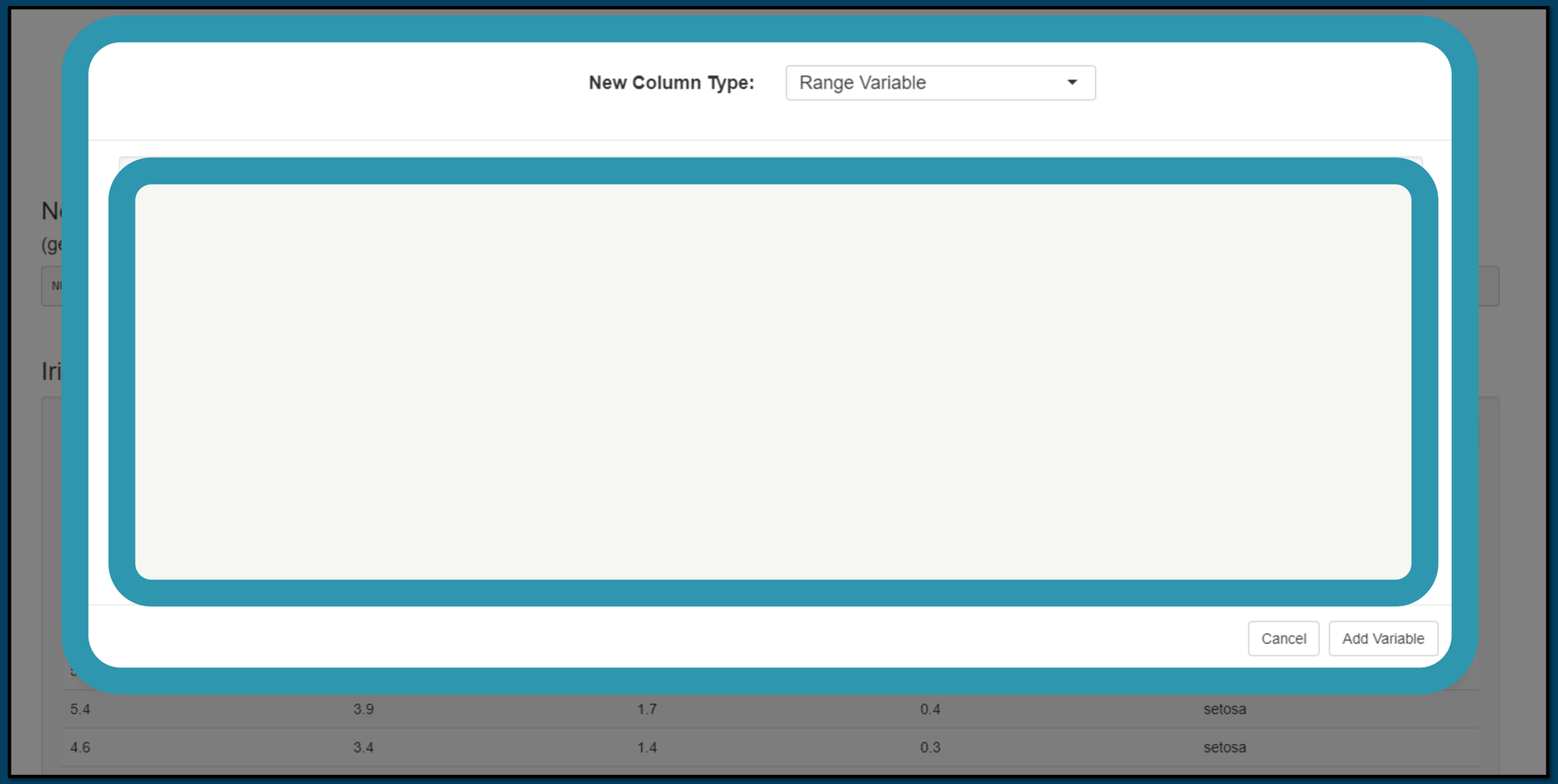
Modules as seen in the app

app.R

launch Modal

newCol

Range Conditions



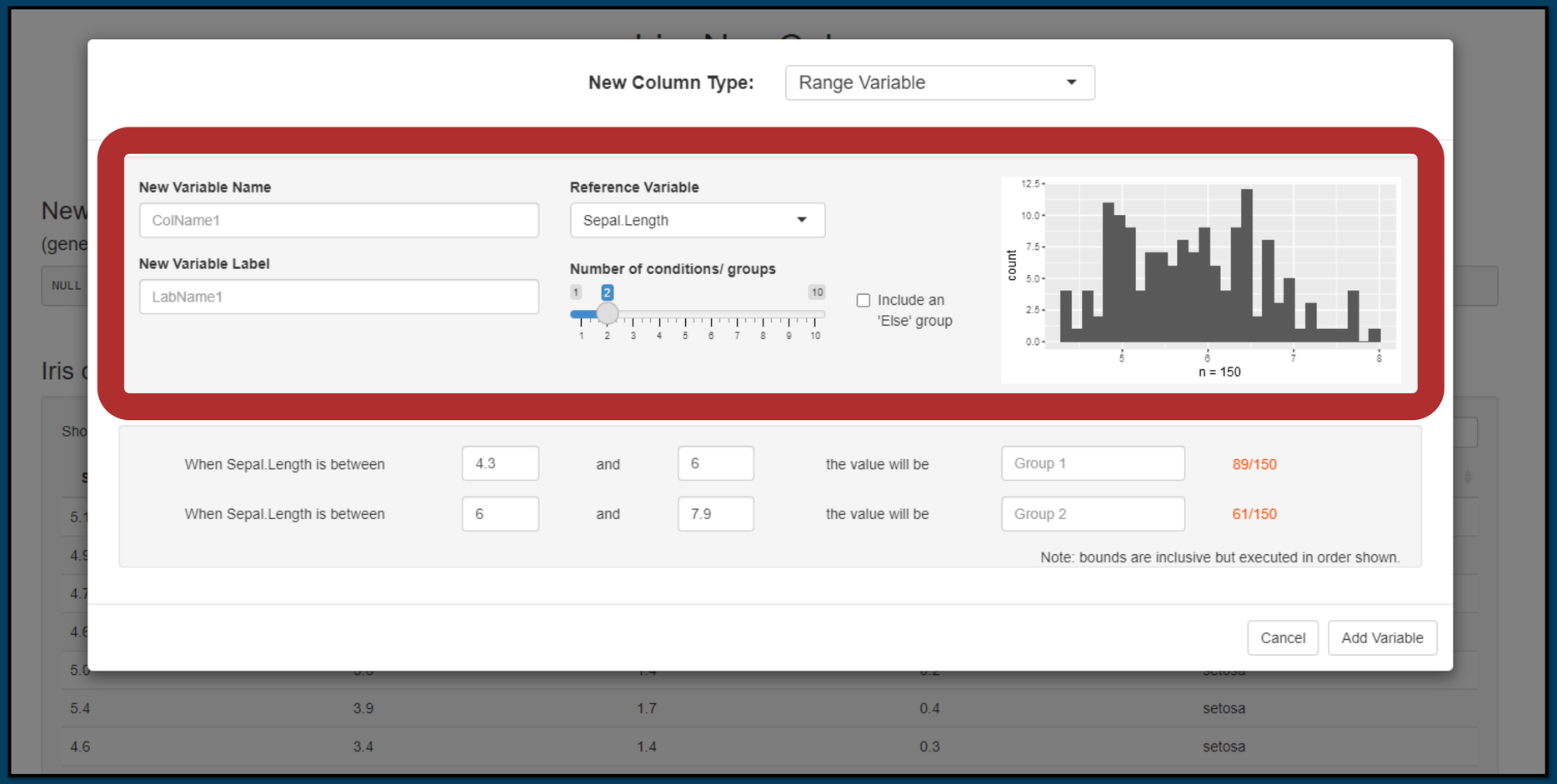
Modules as seen in the app

app.R

launch Modal

newCol

Range Conditions



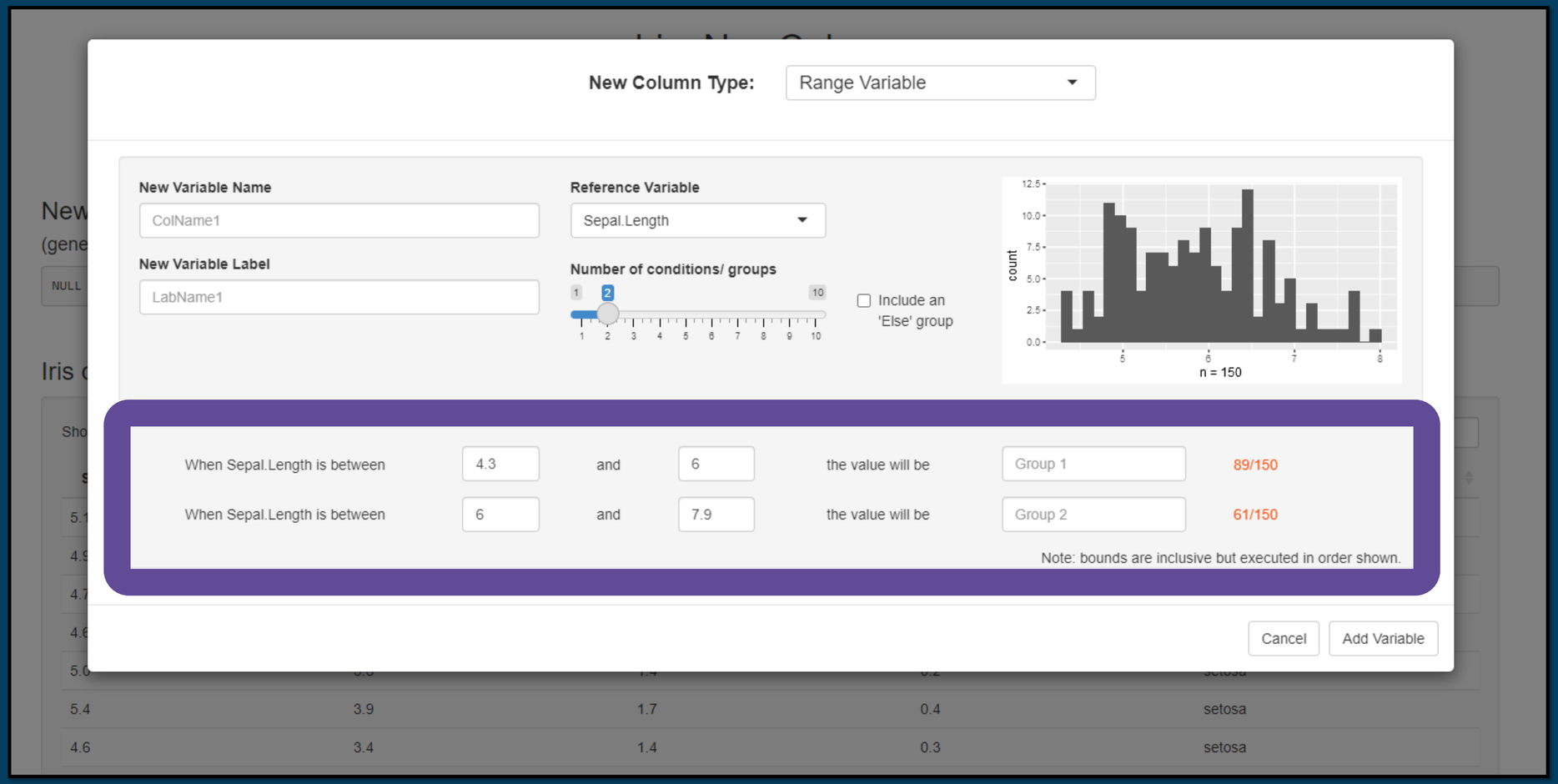
Modules as seen in the app

app.R

launch Modal

newCol

Range Conditions



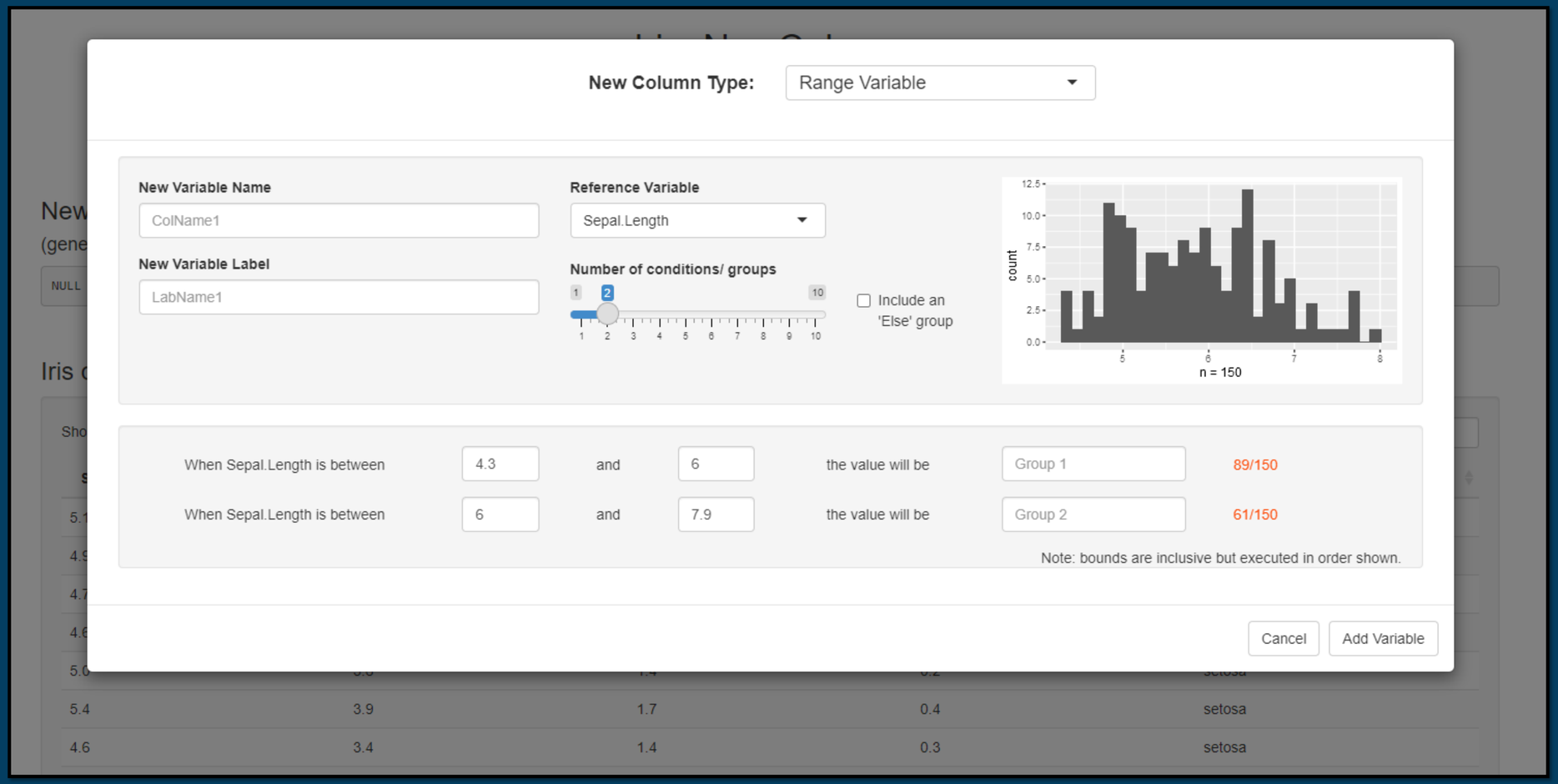
Modules as seen in the app

app.R

launch Modal

newCol

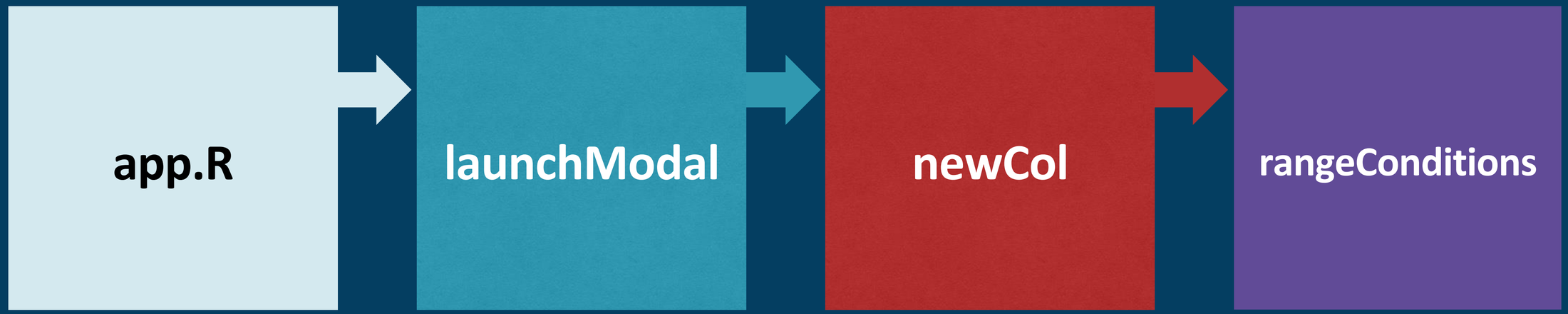
Range Conditions



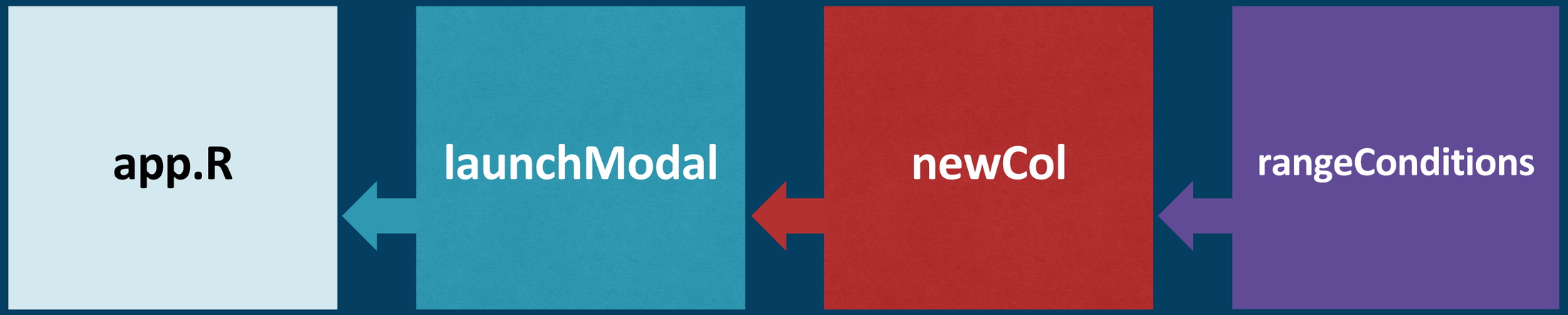
Module Activity

Generating UI, inputs/ outputs, evaluation

Module Activity



Module Activity



What you see:

app.R

library(shinyNewColumns)

UI:

mod_launchModal_ui("snc")

Server:

out <- mod_launchModal_srv("snc", data)

What you see:

app.R

library(shinyNewColumns)

UI:

mod_launchModal_ui("snc")

Server:

out <- mod_launchModal_srv("snc", data)

Call in **global.R** or perhaps as a prefix

shinyNewColumns::mod_launchModal_*

What you see:

app.R

library(shinyNewColumns)

UI:

mod_launchModal_ui("snc")

Server:

out <- mod_launchModal_srv("snc", data)

UI Module simply accepts the 'id' arg and creates a button:

Add New Column

What you see:

app.R

library(shinyNewColumns)

UI:

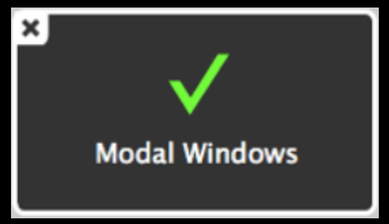
mod_launchModal_ui("snc")

Server:

out <- mod_launchModal_srv("snc", data)

Server Module

launches the modal



Module Activity

app.R

launchModal

newCol

rangeConditions

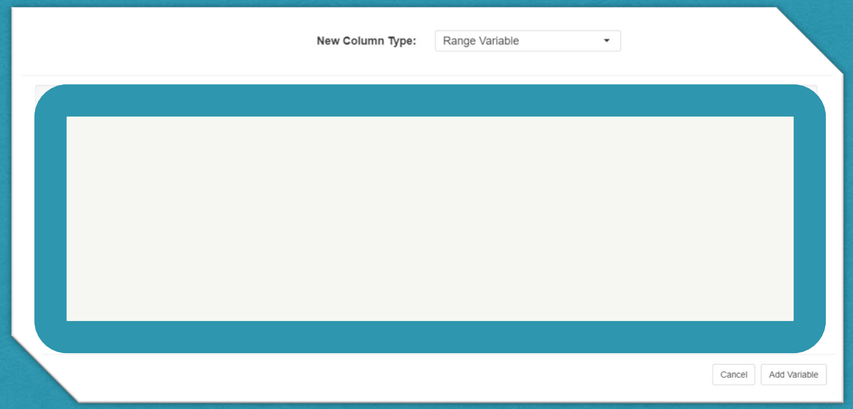
Module Activity

launchModal

UI:

Add New Column

Server:



app.R

newCol

Module Activity

launchModal

Server:

dat (iris)

app.R

```
# initiate reactive values (rv) object to keep track of dplyr
# mutate() expressions throughout the module
rv <- reactiveValues(data = dat, all_mutates = NULL)
```

newCol

Module Activity

launchModal

Server:

dat (iris)

app.R

```
# run 'newCol' module upon 'Create Col' button click, passing data
# and col type. Later, Module returns dplyr mutate expression(s)
observe({
  input$createColBttn
  rv$current_mutate <- mod_newCol_srv(id = "new",
                    dat = reactive(rv$data),
                    colType = reactive(input$createColType))
 })
```

rv$data

New Column Type: Range Variable

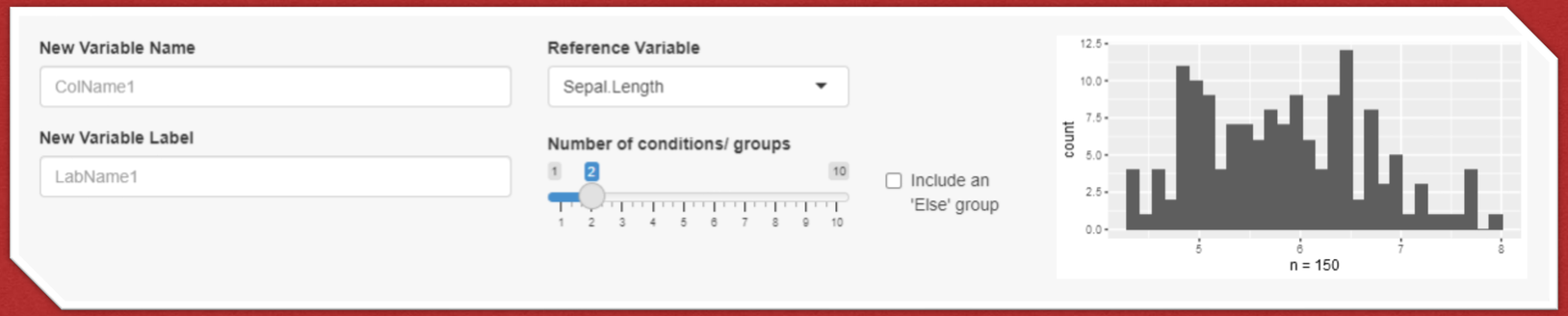
newCol

Module Activity

newCol

UI:

launch Modal



Range Conditions

Module Activity

launch Modal

dat

newCol

Server:

```
# When selected, call rangeConditions module, providing a number of inputs,
#  wrapping them in a reactive context. Save the output as a reactive
moduleExpr <- reactive({
  req(input$numGroups)
    mod_rangeConditions_srv(
      id = "cond1",
      dat = dat,
      grp = reactive(input$numGroups),
      reference_var = reactive(input$reference_var),
      else_group = reactive(input$incl_else),
      else_name = reactive(default_val(input$elseName, else_ph_util)))
})
```

Range Conditions

Module Activity

rangeConditions

UI:

newCol

Module Activity

dat + every UI element from newCol

newCol

rangeConditions

Server: dynamic UI time!

```
# create several vectors of text strings that will be used as
# UI id's that have the same length as there are groups
low <- reactive(paste0("low", seq_len(grp())))
high <- reactive(paste0("high", seq_len(grp())))
then_names <- reactive(paste0("then", seq_len(grp())))
grp_placeholders <- reactive(paste("Group", seq_len(grp())))
```

Step 1: reactive text strings for shiny input ids!

Module Activity

Use purrr::map()

rangeConditions

```
output$casewhens <- renderUI({
  fluidRow(
    column(3, purrr::map(low(), ~ tags$div(class = "add_padding", glue::glue("When {reference_var()} is between")))),
    column(1, purrr::map(low(), ~ numericInput(ns(.x), NULL, value = isolate(input[[.x]]) %||% min(ref_vtr(), na.rm = T), step = resp_step()
    column(1, purrr::map(low(), ~ tags$div(class = "add_padding", "and"))),
    column(1, purrr::map(high(), ~ numericInput(ns(.x), NULL, value = isolate(input[[.x]]) %||% max(ref_vtr(), na.rm = T), step = resp_step(
    column(2, purrr::map(high(), ~ tags$div(class = "add_padding", "the value will be"))),
    column(2, purrr::map2(then_names(), grp_placeholders(), ~  textInput(ns(.x), NULL, value = isolate(input[[.x]]), placeholder = .y) )),
    column(1, purrr::map2(then_names(),  grp_placeholders(), ~ tags$div(class = "add_padding red", glue::glue("{newCol_n()$cnts[newCol_n()$n
  )
})
```

Module Activity

Again, use purrr::map() suite of functions

rangeConditions

```
# get all the range low & high values + string outputs
range_low <- reactive(purrr::map_dbl(low(), ~ default_val(input[[.x]], NA_real_)))
range_high <- reactive(purrr::map_dbl(high(), ~ default_val(input[[.x]], NA_real_)))
range_names <- reactive(purrr::map2_chr(then_names(), grp_placeholders(), ~ default_val(input[[.x]], .y)))
```

```
# create a list of between statements to use in case_when in this module AND parent module
between_expr <- reactive({
  temp <- purrr::pmap(
    list("between", reference_var(), range_low(), range_high(), range_names()),
    build_case_when_formula)

  if (else_group()) append(temp, rlang::expr(TRUE ~ !!else_name())) else  append(temp, rlang::expr(TRUE ~ "NA"))
})
```

Module Activity

We will return this obj later

rangeConditions

```
# create a list of between statements to use in case_when in this module AND parent module
between_expr <- reactive({
  temp <- purrr::pmap(
    list("between", reference_var(), range_low(), range_high(), range_names()),
    build_case_when_formula)

  if (else_group()) append(temp, rlang::expr(TRUE ~ !!else_name())) else  append(temp, rlang::expr(TRUE ~ "NA"))
})
```

```
# dplyr case_when formula
build_case_when_formula <- function(func, value, low, high, string) {
  rlang::expr(!!rlang::call2(func, rlang::sym(value), low, high, .ns = "dplyr") ~ !!string)
}
```

Module Activity

rangeConditions

```
# create a list of between statements to use in case_when in this module AND parent module
between_expr <- reactive({
  temp <- purrr::pmap(
    list("between", reference_var(), range_low(), range_high(), range_names()),
    build_case_when_formula)

  if (else_group()) append(temp, rlang::expr(TRUE ~ !!else_name())) else  append(temp, rlang::expr(TRUE ~ "NA"))
})
```

But for now, we'll start eval in this module to display row counts

```
# Create an expression call using mutate and the between_expr() object above
mutate_expr_call <- reactive({
  colname <- "newCol"
  rlang::call2(
    quote(dplyr::mutate),
    !!colname := rlang::call2(quote(dplyr::case_when),!!!between_expr())
  )
})
```

Enter: call2(quote())

Module Activity

rangeConditions

```r
# Create an expression call using mutate and the between_expr() object above
mutate_expr_call <- reactive({
  colname <- "newCol"
  rlang::call2(
    quote(dplyr::mutate),
    !!colname := rlang::call2(quote(dplyr::case_when),!!!between_expr())
  )
})
```

```r
# Insert that ^^ into a list with the data, and a
# dplyr::select() on our reference variable
all_expressions <- reactive({
  list(
    rlang::expr(dat()),
    rlang::expr(dplyr::select(reference_var())),
    mutate_expr_call()
  )
})
```

Pull all expressions into a list()

Module Activity

```r
# Insert that ^^ into a list with the data, and a
# dplyr::select() on our reference variable
all_expressions <- reactive({
  list(
    rlang::expr(dat()),
    rlang::expr(dplyr::select(reference_var())),
    mutate_expr_call()
  )
})
```

rangeConditions

purrr::reduce helps us pipe each expr,
and base::eval() brings us home

```r
# Create the new column and group by it so we have accurate row
# counts to display next to each condition
mutated_dat <- reactive({
  !any(is.na(range_names()))
  all_expressions() %>% purrr::reduce(~ rlang::expr(!!.x %>% !!.y)) %>% eval()
})

# calculate row counts
newCol_n <- reactive({
  mutated_dat() %>% dplyr::group_by(newCol) %>% dplyr::summarize(cnts = dplyr::n())
})
```

Module Activity

rangeConditions

Time to pass objects back up! Yay!

newCol

```
# create a list of between statements to use in case_when in this module AND parent module
between_expr <- reactive({
  temp <- purrr::pmap(
    list("between", reference_var(), range_low(), range_high(), range_names()),
    build_case_when_formula)

  if (else_group()) append(temp, rlang::expr(TRUE ~ !!else_name())) else  append(temp, rlang::expr(TRUE ~ "NA"))
})
```

Module Activity

newCol

launch Modal

```
# when selected, call rangeConditions module, providing a number of inputs,
#  wrapping them in a reactive context. Save the output as a reactive
moduleExpr <- reactive({
  req(input$numGroups)
    mod_rangeConditions_srv(
      id = "cond1",
      dat = dat,
      grp = reactive(input$numGroups),
      reference_var = reactive(input$reference_var),
      else_group = reactive(input$incl_else),
      else_name = reactive(default_val(input$elseName, else_ph_util)))
})
```

Range Conditions

Module Activity

newCol

launch Modal

```
# construct a call based on inputs (again) & return to parent module
expr_call <-reactive({
  req(moduleExpr())
  colname <- default_val(input$var_name, var_name_ph_util)
  rlang::call2( quote(dplyr::mutate),
    !!colname := rlang::call2(quote(dplyr::case_when),!!!moduleExpr())
  )
})
```

Range Conditions

Module Activity

launchModal

app.R

```
# run 'newCol' module upon 'Create Col' button click, passing data
# and col type. Later, Module returns dplyr mutate expression(s)
observe {
  input$createColBttn
  rv$current_mutate <- mod_newCol_srv(id = "new",
                      dat = reactive(rv$data),
                      colType = reactive(input$createColType))
 })
```

newCol

Module Activity

launchModal

```r
# Upon clicking 'Add Variable' button in modal, combine and evaluate
# dplyr mutate statements in order to modify data
observeEvent(input$addCol, {
  rv$all_mutates <- c(rv$all_mutates, rv$current_mutate)

  # expressions to evaluate on data source
  data_and_expr <- list(
    rlang::expr(rv$data),  # current data
    rv$all_mutates         # current + any other mutates
  )

  # Create the new data frame with mutate(s) applied
  rv$data <- rlang::flatten(data_and_expr) %>%
    purrr::reduce(~rlang::expr(!!.x %>% !!.y)) %>%
    eval()

  removeModal()
})
```

app.R

newCol

Module Activity

launchModal

```
# return the original data, updated with the new
# column. Return the dplyr::mutate() expression
# for fun/ display in app.R.
return( list(
    data = reactive(rv$data),
    expr = reactive(rv$all_mutates)
))
```

app.R

newCol

Module Activity

We did it

shinyNewColumns

Add New Column

New column's code:

(generated by user inputs)

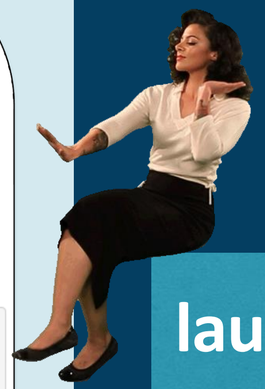
```
[[1]]
dplyr::mutate(SepalLengthCategory = dplyr::case_when(dplyr::between(Sepal.Length,
    4.3, 6) ~ "short", dplyr::between(Sepal.Length, 6, 7) ~ "normal",
    TRUE ~ "long"))
```

Iris data:

Show 25 entries

Search:

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	SepalLengthCategory
5.1	3.5	1.4	0.2	setosa	short
4.9	3.0	1.4	0.2	setosa	short
4.7	3.2	1.3	0.2	setosa	short
4.6	3.1	1.5	0.2	setosa	short
	3.6	1.4	0.2	setosa	short



launch Modal

Internal modular structure

app.R

launchModal

newCol

rangeConditions

What have we learned?

There are a couple things that are difficult about this talk's title:

{Evaluate user inputs from dynamically produced UI Modules}

The big three:
(that I'll covered at least)

- **Generating dynamic UI**
- **Bringing user inputs together in parent module(s)**
- **Evaluating the result**

Generating dynamic UI

Use…

- **reactive id names**
- **purrr::map() suite of functions**

Bringing user inputs together in parent modules

- Combine each iterative UI into it’s own expression
- Put them all in a list
- Return the result

Evaluating those inputs

- Enter {rlang} to save the day: made heavy use of call2(), expr(), sym()
- Base R is great too thank to: quote() & good old fashion eval()

Evaluating those inputs

Steps …

- **Use rlang::call2(quote()) on the returned expressions**
- **Put those in a list**
- **Use purrr::flatten & reduce to pipe each element into eval()**

Check out the project

https://github.com/AARON-CLARK/shinyNewColumns

shinyNewColumns

lifecycle experimental

`shinyNewColumns` is a shiny module used to derive custom columns in R data.frames on the fly. The module simply produces one small UI element: an action button titled 'Create New Column' that can be placed anywhere in an existing shiny application. Upon clicking, a modal containing a user-friendly interface will launch & allow the user to build a new column off an existing data source. Specifically, the user will be able to select a new column type, name, label, and reference column(s) during derivation. Along the way, the module helps the user visualize existing column's distributions and summaries. Upon completion, the module returns the supplied data frame with the new column appended, and optional the `dplyr::mutate()` expressions used to create said column.

Installation

You can install the current development version of `shinyNewColumns` from GitHub using:

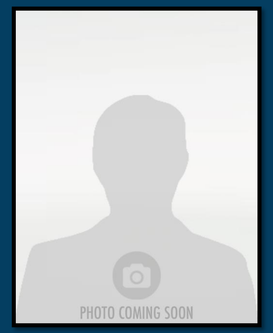
```
remotes::install_github("AARON-CLARK/shinyNewColumns")
```

Example

This is a basic example which shows you how to solve a common problem:

```
library(shinyNewColumns)
library(shiny)
library(DT)
```

A Collaborative Effort



Aaron Clark | **Maya Gans** | **You?**

Open a PR!

Let’s connect

Aaron Clark

clark.aaronchris@gmail.com

{shinyNewColumns}

An open source platform in R shiny to derive custom columns on the fly