Shiny III (User Interface, Lecture 12)

Peter Ganong and Maggie Shi

November 19, 2024

Table of contents I

Conditional UI

Dynamic UI: COVID Example

UI Layout: Basic Dashboard

UI Layout: Multi-page

Recap and Motivation

- Up to now, we've covered much of the basics to get your Shiny app to work: UI + server structure, render decorators, reactive functions
- One of the main benefits of a dashboard is that it can show off the analysis in an aesthetically-pleasing, user-friendly way!
- Today, we'll focus more on the **form** as opposed to the **function** of your app

Conditional UI

Conditional Panels: Intro

- Shiny dashboards can quickly become cluttered and inundate the user with TMI
- One way to reduce clutter and simplify the UI is to give the user the option to show or hide certain panels
- We can do so using **conditional panels**: wrapper around other UI functions that only display if some condition is true

Conditional Panels: Roadmap

- Example 1: add a basic conditional panel to our basic COVID app
- Introduce **Javascript conditions**, which are used to define conditions under which a panel appears
- Example 2: add a more complicated conditional panel to our COVID app using Javascript conditions

COVID App from Last Class

- Let's return to the result of the do-pair-share from last class in the folder: `apps_for_class/covid_plus`
- Recall that this app:
  - Reads in national COVID data
  - Subsets to a state
  - Allows user to choose outcome: Cases or Deaths
  - Plots time series and displays data table

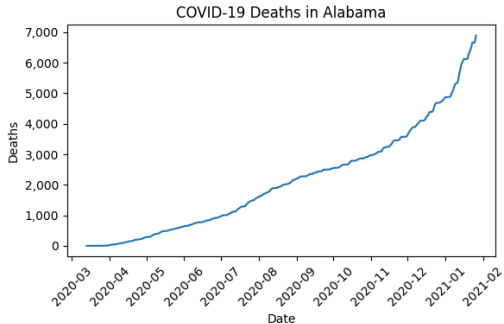
COVID App from Last Class

Choose a state:

Alabama

Choose an outcome:

- Cases
- Deaths



date	state	fips	cases	deaths
2020-03-13	Alabama	1	6	0
2020-03-14	Alabama	1	12	0

Conditional Panels

- Displaying the data table makes the app a bit messy, and perhaps only some users are interested in seeing the raw data
- We can clean this up by giving users the option to show or not show the data with a check box
- Full code for app: `apps_for_class/covid-conditional-basic/`
- All of the changes will be on the **UI-side**

Preview the end goal for Example 1

```
$ cd <your_details>/apps_for_class/
```

```
$ shiny run covid-conditional-basic/app.py
```

Note: we get some warnings related to `set_ticklabels()`. Ignore these.

Example 1: Basic Conditional Panel

- We will need 2 new UI elements

1. Check-box with option to “Show Data” or not
2. Panel with the data table that only shows up *if* “Show Data” is checked

Example 1: Adding a Check Box

- Documentation for check box (link):

ui.input_checkbox

```
ui.input_checkbox(id, label, value=False, *, width=None)
```

Create a checkbox that can be used to specify logical values.

Example 1: Adding a Check Box

- Documentation for check box (link):

ui.input_checkbox

```
ui.input_checkbox(id, label, value=False, *, width=None)
```

Create a checkbox that can be used to specify logical values.

- On **UI-side**, we will add:

```
ui.input_checkbox(id = "show", label = "Show Data", value = FALSE)
```

- `id` parameter is `"show"`, which will store a boolean (i.e., `TRUE/FALSE` logical value), depending on whether it is checked or not
- `value` parameter is `FALSE`, which is what will be stored in `"show"` when the app is initially loaded
- Syntax: to reference `"show"` again on the *UI-side*, we have to call it `input.show` – note the *lack* of parentheses

Example 1: Adding a Conditional Panel

- Now we can move on to adding our conditional panel
- Syntax for conditional panel is:

```
ui.panel_conditional(
    [condition to display panel],
    [what you want to display if condition is true]
)
```

- So for our example, it would be:

```
ui.panel_conditional(
        "input.show",
        ui.output_table("subsetted_data_table")
),
```

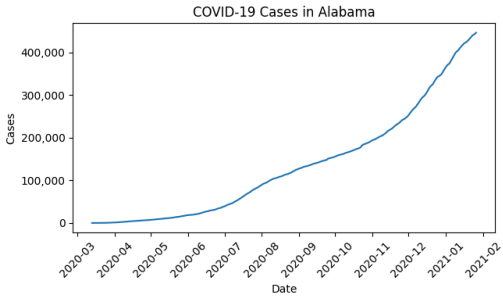
Example 1: Conditional Panel: Result

Choose a state:

Alabama

Choose an outcome:

- (•) Cases
- ( ) Deaths



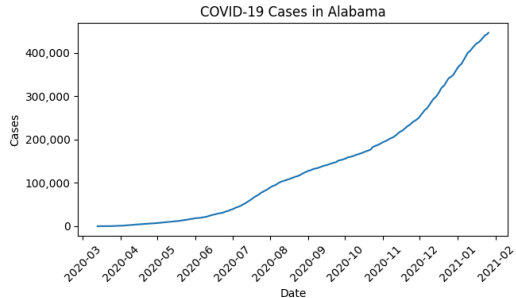
[ ] Show Data

Choose a state:

Alabama

Choose an outcome:

- (•) Cases
- ( ) Deaths



[x] Show Data

date	state	fips	cases	deaths
2020-03-13	Alabama	1	6	0
2020-03-14	Alabama	1	12	0

Javascript Conditions

- Say we want to use more complicated conditions rather than just whether a single input was true or false
- We can operationalize this using **Javascript conditions**
- The syntax is different from Python, but it follows the same logic

- And: e.g., `input.show1 && input.show2`
- Or: e.g., `input.condition_to_show1 || input.condition_to_show2`
- Not: e.g., `!input.my_condition`
- Check if a list (e.g., of selected choices) is non-zero: `input.myselectedvalues.length > 0`

Example 2: A More Complicated Conditional Panel

Preview the end state `$ shiny run covid-conditional-advanced/app.py`

- Say that on the server-side we have:
  - A `render.plot` called `ts` that plots a time series of one state
  - Another `render.plot` called `ts2` that plots two states at a time
- We want to develop the parallel **UI-side**

Let's map the conditions at the whiteboard

1. Has a switch that toggles between showing 1 or 2 states in time series
2. If user chooses to show one state: plot `ts` and asks if they want to show data
3. If a user chooses to show another state and wants to show data: plot `ts` and show `subsetted_data_table`
4. If user chooses to show two states: let user choose the second state and plot the `ts2` for both states.

Example 2: add a toggle switch

- Toggle switch documentation (link)

```
ui.input_switch("anotherstate", "Two States", False)
```

Example 2: add a toggle switch

- Toggle switch documentation (link)

```
ui.input_switch("anotherstate", "Two States", False)
```

- Question: if I want to reference the resulting True/False value of the switch, what would I call it?

Example 2: first conditional panel

- First conditional panel: if user chooses to show one state: plot time series for one state

```
ui.panel_conditional(
     "!input.anotherstate",
     ui.output_plot("ts"),
     ui.input_checkbox("show", "Show Data")
 )
```

Example 2: second conditional panel

- Second conditional panel: if user chooses to show one state *and* wants to show data table

```
ui.panel_conditional(
      "!input.anotherstate && input.show",
      ui.output_table("subsetted_data_table")
),
```

Example 2: third conditional panel

- Third conditional panel: if user chooses to show another state, let user choose the second state and plot the `ts2` for both states.

```
ui.panel_conditional(
    "input.anotherstate",
    ui.input_select(id = 'state2', label = 'Choose a state:',
    choices = ["Alabama", "Alaska", "Arizona", "Arkansas",...]),
    ui.output_plot("ts2")
)
```

Example 2: putting it all together

```
app_ui = ui.page_fluid(
    ui.input_select(id = 'state', label = 'Choose a state:',
    choices = ["Alabama", "Alaska", "Arizona", "Arkansas" ...]),
    ui.input_switch("anotherstate", "Two States", False),
    ui.input_radio_buttons(id = 'outcome', label = 'Choose an outcome:', choices =
↪   ["Cases", "Deaths"]),
    ui.panel_conditional(
        "!input.anotherstate",
        ui.output_plot("ts"),
        ui.input_checkbox("show", "Show Data")),
    ui.panel_conditional(
         "!input.anotherstate && input.show",
         ui.output_table("subsetted_data_table")),
    ui.panel_conditional(
        "input.anotherstate",
        ui.input_select(id = 'state2', label = 'Choose a state:',
        choices = ["Alabama", "Alaska", "Arizona", "Arkansas" ...]),
        ui.output_plot("ts2"))
)
```

Example 2: Result

Choose a state:

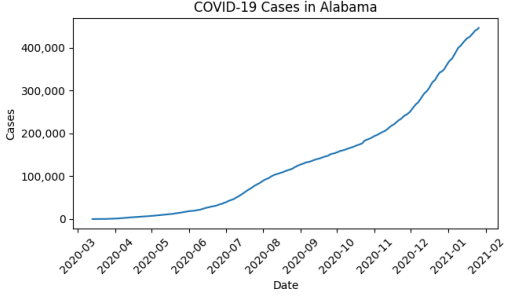
Alabama

Two States

Choose an outcome:

Cases

Deaths



Show Data

date	state	fips	cases	deaths
2020-03-13	Alabama	1	6	0
2020-03-14	Alabama	1	12	0

Choose a state:

Alabama

Two States

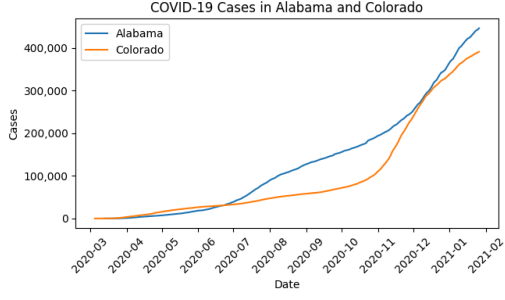
Choose an outcome:

Cases

Deaths

Choose a state:

Colorado



Conditional UI: Summary

- We can use conditional panels to allow users to "turn on and off"
- Syntax for defining conditional panel:

```
ui.panel_conditional(
    [condition to display panel],
    [what you want to display if condition is true]
)
```

- To define a condition:
  - Referencing user input stored as `"name_of_ui_value"`: `input.name_of_ui_value`
  - More complicated conditions involve Javascript conditions

Dynamic UI: COVID Example

Dynamic UI: Intro and Roadmap

- Recap hard-coding of state list is generated in Shiny
- Define dynamic UI
- Introduce `@reactive.effect` and defining functions with no name
- Show how this can be used to avoid hard-coding the state list

Dynamic UI: Recap status quo

- Recall how we've been defining the state selection list in our apps (e.g., see `covid-conditional-basic` or `covid-conditional-advanced`)
- **UI-side** looks like:

```
    ui.input_select(id = 'state', label = 'Choose a state:',
    choices = ["Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado",
↪   "Connecticut", "Delaware", "Florida", "Georgia", "Hawaii", "Idaho", "Illinois",
↪   "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
↪   "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana",
↪   "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York",
↪   "North Carolina", "North Dakota", "Ohio", "Oklahoma", "Oregon", "Pennsylvania",
↪   "Rhode Island", "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah",
↪   "Vermont", "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"])
```

- Generally we want to avoid hard coding:
  - sensitive to typos
  - not robust to large choice sets
  - *(makes your code ugly!)*

Dynamic UI: Auto-generate state list

- Instead of hard coding, we'd like to generate the state list based on the data
  - Read in the data
  - Get list of unique states and sort
  - And use this list as the choices instead of `["Alabama", "Alaska", "Arizona", "Arkansas", "California",...`
- Importantly, want this to happen automatically (i.e., without requiring the user to input anything)

Discussion questions:

1. Prior examples: which side "acts" first: UI or server?
2. How will it need to differ here?

Dynamic UI: Reactive effects

- Because generating the state list requires some `pandas` calculations, we will first add to the **server-side**:
- Syntax for a reactive effect is the following:

```
@reactive.effect
def _():
    # code for the whatever we want to take place automatically
```

- Functions decorated with `@reactive.effect` don't have a name. No one calls them and they don't wait to be called. They just wait in the shadows. They execute if one of their dependencies change.

Dynamic UI: Reactive effect to auto-generate state list

```
app_ui = ui.page_fluid(
    ui.input_select(             #1
        id = 'state',
        label = 'Choose a state:',
        choices = []             #2
    )
    ...
)

def server(input, output, session):
    @reactive.effect
    def _():
        states = full_data()['state']          #4
        state_list = unique(states).tolist()   #5
        state_list = sorted(state_list)        #6
        ui.update_select("state", choices=state_list)              #7
```

- Lines #4-6 generate a sorted list of unique state names called `state_list`
- Line #7 updates the UI element for `"state"`
- Line #1: where `"state"` UI element is defined
- Line #2: the default for `"state"` is that there is no initial choice set: `choices = []`

Dynamic UI: Reactive effect to auto-generate state list

```
app_ui = ui.page_fluid(
    ui.input_select(             #1
        id = 'state',
        label = 'Choose a state:',
        choices = []             #2
    )
    ...
)

def server(input, output, session):
    @reactive.effect
    def _():
        states = full_data()['state']          #4
        state_list = unique(states).tolist()   #5
        state_list = sorted(state_list)        #6
        ui.update_select("state", choices=state_list)              #7
```

- Lines #4-6 generate a sorted list of unique state names called `state_list`
- Line #7 updates the UI element for `"state"`
- Line #1: where `"state"` UI element is defined
- Line #2: the default for `"state"` is that there is no initial choice set: `choices = []`

- Since `full_data()` is read *immediately*, the `@reactive.effect` also runs *immediately*. So the choices are filled in (almost) immediately, and we should never see the blank choice set from line #2

Updating UI from server side

- Every UI element has an equivalent "update" method which can be used from the **server-side**
- `ui.update_select()` updates `ui.input_select()`
- Other examples: `ui.update_checkbox()`, `ui.update_switch()`, `ui.update_radio_buttons()`, `ui.update_numeric()`, etc.

Comparing this reactive effect to previous examples

```
def server(input, output, session):
    @reactive.calc
    def subsetted_data():
        df = full_data()
        return df[df['state'] ==
        ↪  input.state()]
```

- From last lecture
- Runs when `subsetted_data()` is called or `full_data()` is updated.

```
def server(input, output, session):
    @reactive.effect
    def _():
        states = full_data()['state']
        ...
```

- From this lecture
- Does not need to be called
- Runs when `full_data()` is updated (basically immediately)

Dynamic UI: Summary

- `@reactive.effect` + `def _():` creates a reactive element that runs automatically
- All UI elements have an equivalent "update" method that can be used on the *server* side

UI Layout: Basic Dashboard

Customizing UI: Intro

- Shiny offers a multitude of ways to change the look and feel of your UI
- All controlled on the **UI-side**: doesn't include writing more Python

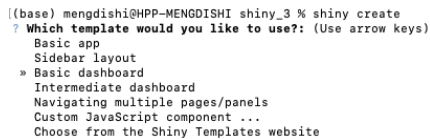
- Shiny comes pre-loaded with some basic templates
- When building your app, can be useful to start from these as a "skeleton" as opposed to coding directly from scratch

UI Layout Basic Dashboard: Roadmap

- sidebar
- columns layout
- cards
- icons

Back to Lecture 1

- Going back to our first Shiny lecture: `cd` to your desired directory, then `$ shiny create`



- This time, navigate down (using arrow keys) to "Basic dashboard"
- Say no to Shiny Express
- Give your folder a name: `basic-dash`
- You may need to `$ pip install faicons` and `$ pip install seaborn`
- Deploy: `$ shiny run --reload basic-dash/app.py`

Basic Dashboard

Penguins dashboard

Filter controls

Mass

2,000 6,000

Species

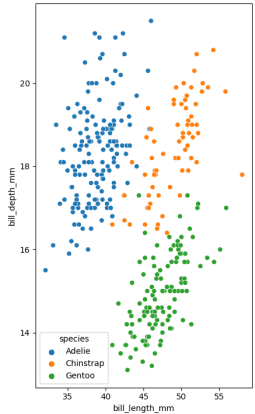
- Adelie
- Gentoo
- Chinstrap

Number of penguins
338

Average bill length
43.8 mm

Average bill depth
17.2 mm

Bill length and depth



Penguin data

species	island	bill_length_mm	bill_depth_mm	body_m
Adelie	Torgersen	39.1	18.7	3750
Adelie	Torgersen	39.5	17.4	3800
Adelie	Torgersen	40.3	18	3250
Adelie	Torgersen	36.7	19.3	3450
Adelie	Torgersen	39.3	20.6	3650
Adelie	Torgersen	38.9	17.8	3625
Adelie	Torgersen	39.2	19.6	4675
Adelie	Torgersen	34.1	18.1	3475
Adelie	Torgersen	42	20.2	4250
Adelie	Torgersen	37.8	17.1	3300
Adelie	Torgersen	37.8	17.3	3700
Adelie	Torgersen	41.1	17.6	3200
Adelie	Torgersen	38.6	21.2	3800
Adelie	Torgersen	34.6	21.1	4400
Adelie	Torgersen	36.6	17.8	3700
Adelie	Torgersen	38.7	19	3450
Adelie	Torgersen	42.5	20.7	4500
Adelie	Torgersen	34.4	18.4	3325
Adelie	Torgersen	46	21.5	4200
Adelie	Biscoe	37.8	18.3	3400

Viewing rows 1 through 21 of 338

Basic Dashboard: Collapsible Sidebar

Basic Dashboard: Collapsible Sidebar

```
app_ui = ui.page_sidebar(
    ui.sidebar(
        ui.input_slider("mass", "Mass", 2000, 6000, 6000),
        ui.input_checkbox_group(
            "species",
            "Species",
            ["Adelie", "Gentoo", "Chinstrap"],
            selected=["Adelie", "Gentoo", "Chinstrap"],
        ),
        title="Filter controls",
    ),
    # rest of UI code
)
```

- `ui.page_sidebar()` wraps around *all* content in the page, even what is not in the sidebar
- `ui.sidebar()` wraps around content in the sidebar
- `title="Filter controls"` is a parameter for `ui.sidebar()`
- Within `ui.sidebar()`, other two components are stacked in the order they appear in code

Basic Dashboard: Columns

Penguins dashboard

Filter controls

Mass

2,000 6,000

Species

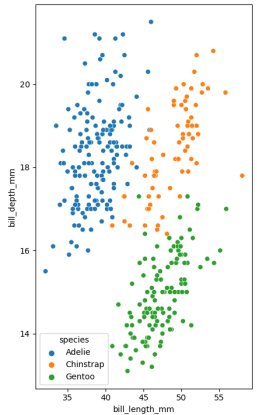
- Adelie
- Gentoo
- Chinstrap

Number of penguins
338

Average bill length
43.8 mm

Average bill depth
17.2 mm

Bill length and depth



Penguin data

species	island	bill_length_mm	bill_depth_mm	body_m
Adelie	Torgersen	39.1	18.7	3750
Adelie	Torgersen	39.5	17.4	3800
Adelie	Torgersen	40.3	18	3250
Adelie	Torgersen	36.7	19.3	3450
Adelie	Torgersen	39.3	20.6	3650
Adelie	Torgersen	38.9	17.8	3625
Adelie	Torgersen	39.2	19.6	4675
Adelie	Torgersen	34.1	18.1	3475
Adelie	Torgersen	42	20.2	4250
Adelie	Torgersen	37.8	17.1	3300
Adelie	Torgersen	37.8	17.3	3700
Adelie	Torgersen	41.1	17.6	3200
Adelie	Torgersen	38.6	21.2	3800
Adelie	Torgersen	34.6	21.1	4400
Adelie	Torgersen	36.6	17.8	3700
Adelie	Torgersen	38.7	19	3450
Adelie	Torgersen	42.5	20.7	4500
Adelie	Torgersen	34.4	18.4	3325
Adelie	Torgersen	46	21.5	4200
Adelie	Biscoe	37.8	18.3	3400

Viewing rows 1 through 21 of 338

Basic Dashboard: Columns

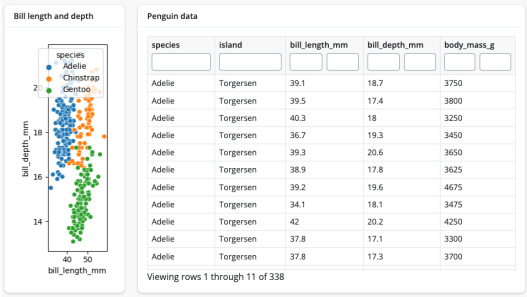
```
ui.layout_columns(
    ui.card(
        ui.card_header("Bill length and depth"),
        ui.output_plot("length_depth"),
        full_screen=True,
    ),
    ui.card(
        ui.card_header("Penguin data"),
        ui.output_data_frame("summary_statistics"),
        full_screen=True,
    ),
),
```

- `ui.layout_columns`: responsive, column-based grid layout
- Underpinned by a 12-column grid
- If no widths are defined (like in this example), will make each component equally-spaced – each takes up 6 columns
- `ui.cards` are a general-purpose container for grouping related UI elements

Basic Dashboard: Columns

```
ui.layout_columns(
    ui.card(
        ui.card_header("Bill length and depth"),
        ui.output_plot("length_depth"),
        full_screen=True,
    ),
    ui.card(
        ui.card_header("Penguin data"),
        ui.output_data_frame("summary_statistics"),
        full_screen=True,
    ),
    col_widths=(3, 9)
),
```

- Alternatively, setting `col_widths = (3,9)` gives 3 columns to plot, 9 to the data

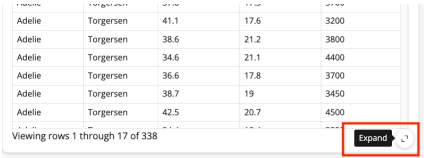


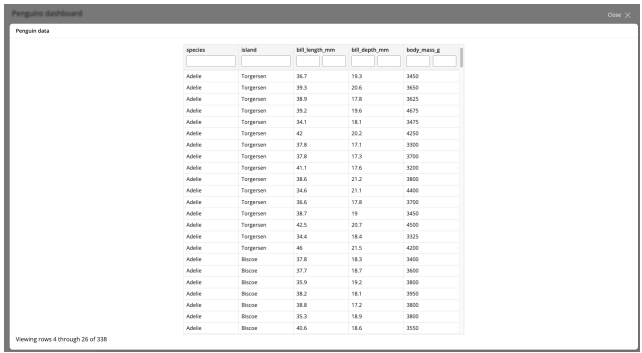
Basic Dashboard: Cards

```
ui.card(
    ui.card_header("Bill length and
↪   depth"),
    ui.output_plot("length_depth"),
    full_screen=True,
    )
```

- `full_screen = True`: gives user option to expand card to full-screen if they hover over it

Basic Dashboard: Fullscreen





UI Layout Basic Dashboard: Summary

- Shiny has basic templates that you can automatically load
- Ways to organize your UI:
  - `ui.page_sidebar()`: collapsible sidebar
  - `ui.layout_columns()`: 12-column grid
  - `ui.card()`: all-purpose container

UI Layout: Multi-page

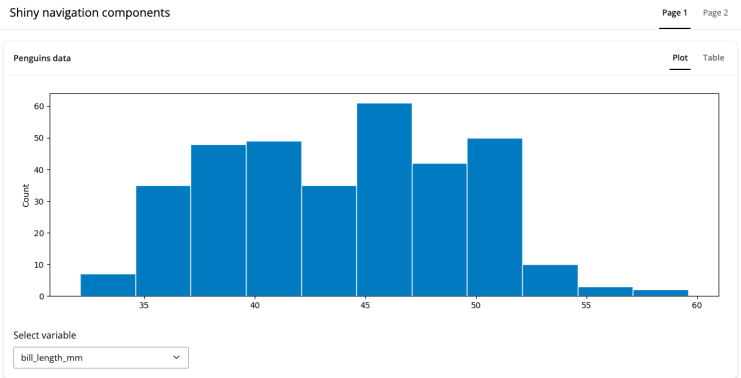
UI Layout Multi-page: Roadmap

- Walk through built-in multi-page dashboard example
- Demonstrate nesting of UI elements

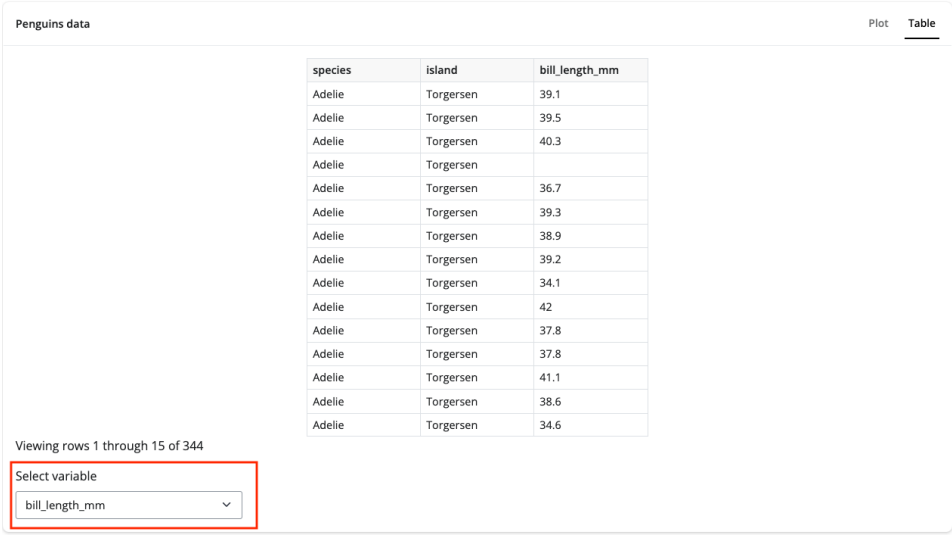
Multi-Page Example

- Now let's look at the multi-page dashboard example

```
(base) mengdishi@HPP-MENGDISHI fall2024 % shiny create
? Which template would you like to use?: (Use arrow keys)
   Basic app
   Sidebar layout
   Basic dashboard
   Intermediate dashboard
 » Navigating multiple pages/panels
   Custom JavaScript component ...
   Choose from the Shiny Templates website
```



Multi-Page Example



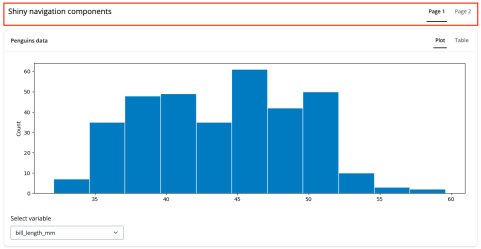
Multi-Page Example: Structure

```
page1 = ui.navset_card_underline(
    ui.nav_panel("Plot", ui.output_plot("hist")),
    ui.nav_panel("Table",
↪   ui.output_data_frame("data")),
    footer=ui.input_select(
        id = "var",
        label = "Select variable",
        choices = ["bill_length_mm",
↪   "body_mass_g"]
    ),
    title="Penguins data",
)

app_ui = ui.page_navbar(
    ui.nav_spacer(),
    ui.nav_panel("Page 1", page1),
    ui.nav_panel("Page 2", "This is the second
↪   'page'."),
    title="Shiny navigation components",
)
```

- First thing to note: `page1` is defined *outside* of `app_ui`
- It is then referenced within `app_ui`: `ui.nav_panel("Page 1", page1)`
- You can do this to make the code within `app_ui` easier to read

Multi-Page Example: Structure

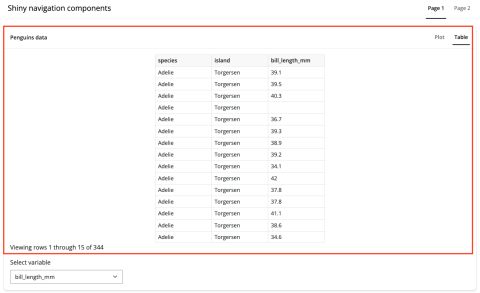
```
page1 = ui.navset_card_underline(
    ui.nav_panel("Plot", ui.output_plot("hist")),
    ui.nav_panel("Table",
↪   ui.output_data_frame("data")),
   footer=ui.input_select(
        id = "var",
        label = "Select variable",
        choices = ["bill_length_mm",
↪   "body_mass_g"]
    ),
    title="Penguins data",
)

app_ui = ui.page_navbar(
    ui.nav_spacer(),
    ui.nav_panel("Page 1", page1),
    ui.nav_panel("Page 2", "This is the second
↪   'page'."),
    title="Shiny navigation components",
)
```



- `ui.page_navbar` wraps around *all* content to define the page
- `ui.page_navbar` defines `"Page 1"` and `"Page 2"`

Multi-Page Example: Structure

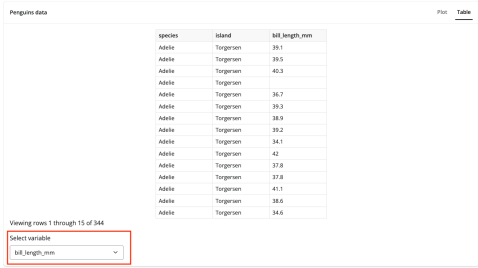
```
page1 = ui.navset_card_underline(
    ui.nav_panel("Plot", ui.output_plot("hist")),
    ui.nav_panel("Table",
 ↪  ui.output_data_frame("data")),
    footer=ui.input_select(
        id = "var",
        label = "Select variable",
        choices = ["bill_length_mm",
 ↪  "body_mass_g"]
    ),
    title="Penguins data",
)

app_ui = ui.page_navbar(
    ui.nav_spacer(),
    ui.nav_panel("Page 1", page1),
    ui.nav_panel("Page 2", "This is the second
 ↪  'page'."),
    title="Shiny navigation components",
)
```



- Within page1, there is another set of `ui.nav_panel` to switch between `"Plot"` and `"Table"`

Multi-Page Example: Structure

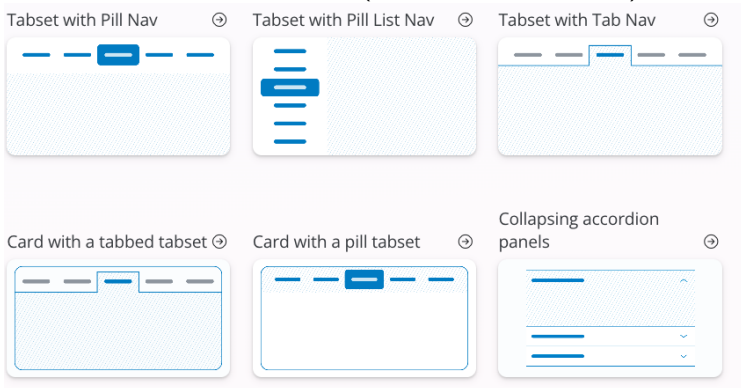
```
page1 = ui.navset_card_underline(
    ui.nav_panel("Plot", ui.output_plot("hist")),
    ui.nav_panel("Table",
↪   ui.output_data_frame("data")),
    footer=ui.input_select(
        id = "var",
        label = "Select variable",
        choices = ["bill_length_mm",
↪   "body_mass_g"]
    ),
    title="Penguins data",
)

app_ui = ui.page_navbar(
    ui.nav_spacer(),
    ui.nav_panel("Page 1", page1),
    ui.nav_panel("Page 2", "This is the second
↪   'page'."),
    title="Shiny navigation components",
)
```



- The drop-down menu to select a variable is defined in the *footer* of page1
- So it remains in place when you switch from `"Plot"` to `"Table"`

Multi-Page Example

- `ui.page_navbar` is a page container
- `navset_card_underline()` is a pre-defined navigation/tab panel layout
- Some other common layouts (link to Shiny Layouts)



Multi-Page Example: Summary

- Tools to create multi-page layouts: `ui.nav_panel`, `ui.page_navbar`, and `navset_card_underline()`
- UI elements can be nested to clean them up

Exercise

- **Exercise**: add to page2 to the multi-page example that looks like the following:

1. UI-side: define a new UI element called `page2`, based on `page1`
2. UI-side: add a `ui.card()` with title ("Penguins sorted by body mass")
3. Server-side: add a `render` function that returns `df[["species", "island", "body_mass_g"]].sort_values(by=[`

Whole Lecture Summary

- We've covered 3 ways to make your dashboard more user-friendly and look better
- **Conditional UI**: hide content depending on user selection
- **Dynamic UI**: let UI "react" to user selection or server-side input
- **Custom UI layouts**: improve readability and make your app easier to navigate