Shiny II (Reactive Programming, Lecture 11)

Peter Ganong and Maggie Shi

November 13, 2024

Table of contents I

First Reactive Example

Case Study: COVID-19 Dashboard I

Case Study: COVID-19 Dashboard II

First Reactive Example

Reactive Functions: Roadmap

- Define reactive functions
- Add a new `my_sumstats()` function to the `normal_distn_app` app from last class
- Seems deceptively easy, but requires debugging
- Solution to our bug will be a reactive function
- Discuss other use cases for reactive functions

Reactive Functions definition

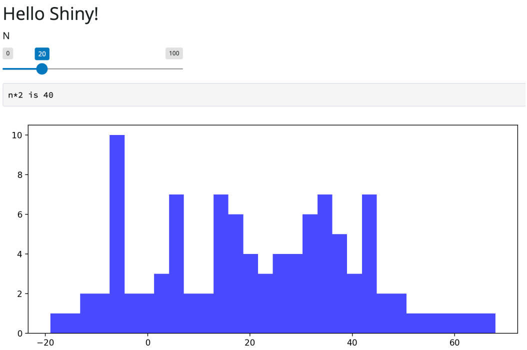
- A “Reactive” function is one that keeps track of **dependencies** (e.g., `input`) and will only re-run a piece of code when it detects one of its dependencies has changed

- This minimizes unnecessary computations by only updating outputs that are affected by dependency changes

- Ordering of reactive functions doesn’t matter; instead they just track their dependencies

Reactive Functions: Normal Distribution Example

- To see the usefulness of reactive functions, let's go back to our example from last class:



Hello Shiny!

```
n*2 is 40
```

- App from last class shared in `student30538/before_lecture/shiny_11/apps_for_class/normal_distn_app/ap`

Reactive Functions: Normal Distribution Example

- Say we want to add some summary statistics to the bottom

- Recall what the **server side** currently looks like:

```
def server(input, output, session):
    # [other server-side code]
    @render_altair
    def my_hist():
        sample = np.random.normal(input.mu(), 20, 100)
        df = pd.DataFrame({'sample': sample})
        return(
            alt.Chart(df).mark_bar().encode(
                alt.X('sample:Q', bin=True),
                alt.Y("count()")
            )
        )
```

Reactive Functions: Normal Distribution Example

- Add in text with minimum, median, and maximum below the graph

```
@render.text
def my_sumstats():
    sample = np.random.normal(input.n(), 20, 100)
    min = np.min(sample)
    max = np.max(sample)
    median = np.median(sample)
    return "Min:" + str(min) + ", Median: " + str(median), ", Max: "
     ↪  + str(max)
```

- Question: are we done now?

Reactive Functions: Normal Distribution Example

- Add in text with minimum, median, and maximum below the graph

```
@render.text
def my_sumstats():
    sample = np.random.normal(input.n(), 20, 100)
    min = np.min(sample)
    max = np.max(sample)
    median = np.median(sample)
    return "Min:" + str(min) + ", Median: " + str(median), ", Max: "
     ↪  + str(max)
```

- Question: are we done now?
- Answer: Nope – still have to add to the **UI side**!

Reactive Functions: Normal Distribution Example

- Add `ui.output_text_verbatim()` to the UI side

- We reference `my_sumstats()` as "`my_sumstats`" on the UI side

```
app_ui = ui.page_fluid(
    ui.panel_title("Histogram of 200 Draws from Normal with mean
 ↪  mu"),
    ui.input_slider("mu", "mean mu", 0, 100, 20),
    ui.output_plot("my_hist"),
    ui.output_text_verbatim("my_sumstats")
)
```

Tech Interlude

I wasn't able to get Altair plots to render alongside `ui.output_text_verbatim`. `Matplotlib` using `@render.plot` works just fine. I have no idea if this is a problem that other people will have, but since the point of this lecture is to learn shiny, not to learn plots, I reverted the dashboards in this lecture to use `Matplotlib`.

In-class exercise

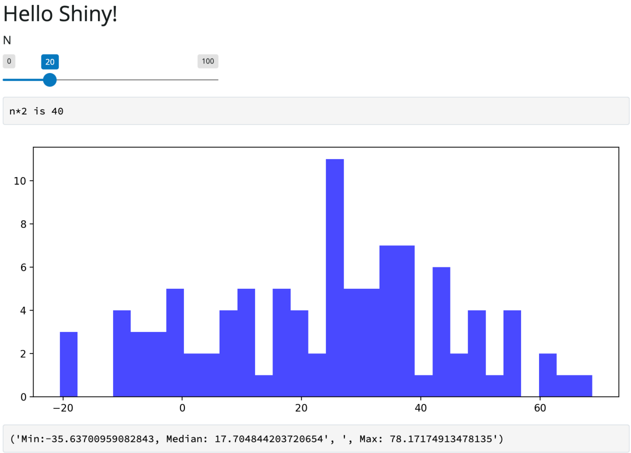
Goal: Update the app from last class using the code above

1. navigate to app folder: `student30538/before_lecture/shiny_11/apps_for_class/normal_distn_app/`
2. In VSCode, modify `app.py` with the material from the prior slides in lecture
   - add the extra line in `app_ui`
   - add the new function `my_sumstats()`
3. In terminal, `shiny run --reload app.py`

Reactive Functions: Normal Distribution Example

- Your updated app should look like the following

- But with a different histogram + summary stats, because it was a random sample



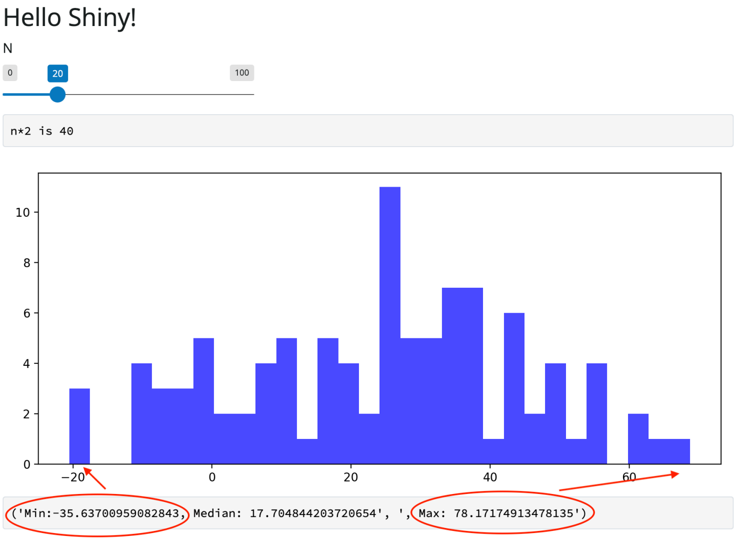
Hello Shiny!

N

```
n*2 is 40
```

```
('Min:-35.63700959082843, Median: 17.704844203720654', ', Max: 78.17174913478135')
```

Reactive Functions: Normal Distribution Example



```
n*2 is 40
```

```
('Min:-35.63700959082843, Median: 17.704844203720654', ', Max: 78.17174913478135')
```

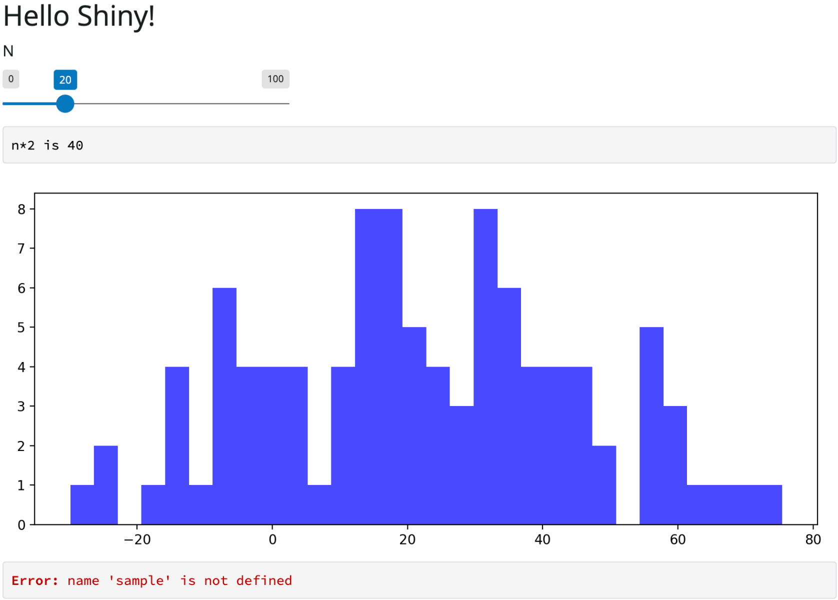
- But wait!

Reactive Functions: Normal Distribution Example

- Issue: `sample` is drawn twice: in `my_hist()` and again in `my_sumstats()`

```
@render.plot
def my_hist():
    sample = np.random.normal(input.n(), 20, 100)
    fig, ax = plt.subplots()
    ax.hist(sample, bins=30, color='blue', alpha=0.7)
    return fig

@render.text
def my_sumstats():
    sample = np.random.normal(input.n(), 20, 100)
    min = np.min(sample)
    max = np.max(sample)
    median = np.median(sample)
    return "Min:" + str(min) + ", Median: " + str(median), ", Max: "
     ↪  +str(max)
```

Reactive Functions: Normal Distribution Example



Hello Shiny!

N

```
n*2 is 40
```

```
Error: name 'sample' is not defined
```

Reactive Functions: Normal Distribution Example

- Recall that `my_sumstats()` is a **function** – so it won't recognize `sample` if it is defined in another function, `my_hist()`
- Typically we would define `sample` outside of the function, and then feed it into `my_sumstats()` as an input
- In Shiny, we do this by making a `sample` into a **reactive function**

Reactive Functions: Normal Distribution Example

- Recall that `my_sumstats()` is a **function** – so it won't recognize `sample` if it is defined in another function, `my_hist()`
- Typically we would define `sample` outside of the function, and then feed it into `my_sumstats()` as an input
- In Shiny, we do this by making a `sample` into a **reactive function**
- First, add to the top:

```
from shiny import App, render, ui, reactive
```

Reactive Functions: Normal Distribution Example

- On the **server side**: define a new reactive function called `sample()`
- `@reactive.calc` decorator: used for functions whose return value depends on inputs or other reactive values

```
@reactive.calc
def sample():
    return(np.random.normal(input.n(), 20, 100))
```

- This function is *reactive* because it will only run if its dependency, `input.n()`, changes

Reactive Functions: Normal Distribution Example

- Then in my_hist() and my_sumstats(), replace every prior instance of sample with sample()

- And remove any prior code that defines sample

```
@render.plot
def my_hist():
    sample = np.random.normal(input.n(), 20, 200)
    fig, ax = plt.subplots()
    ax.hist(sample, bins=30, color='blue', alpha=0.7)
    return fig

@render.text
def my_sumstats():
    min = np.min(sample())
    max = np.max(sample())
    median = np.median(sample())
    return "Min:" + str(min) + ", Median: " + str(median), ", Max: " +str(max)
```

Reactive Functions: Normal Distribution Example

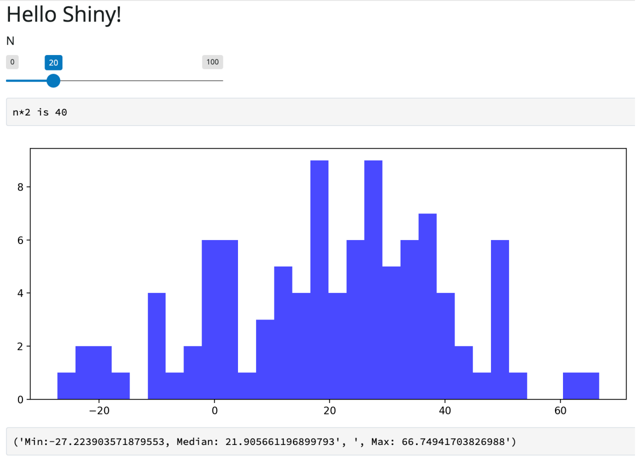
- Question: do we need to go back to UI side and change anything?

Reactive Functions: Normal Distribution Example

- Question: do we need to go back to UI side and change anything?
- Answer: no, because we didn't change anything about what the app displays

Reactive Functions: Normal Distribution Example

- Run it again, and now the summary statistics are correct!



Hello Shiny!

N

```
n*2 is 40
```

```
('Min:-27.223903571879553, Median: 21.905661196899793', ', Max: 66.74941703826988')
```

Reactive Functions: Other Use Cases

So far we have seen `@reactive.calc`. Next lecture, we will also cover `@reactive.effect` and `@reactive.event`. Here are all the things that you ultimately can do using reactive functions

- **Data importing**: import and store data from an external source
- **Reduce run-time**: re-run function *only* when one of the inputs changes
- **Dynamic data filtering**: filter/subset data based on user input
- **Conditional UI**: change or hide specific UI elements (e.g., input fields, dropdown menu values) based on user input *(next class)*

Reactive Functions: Summary

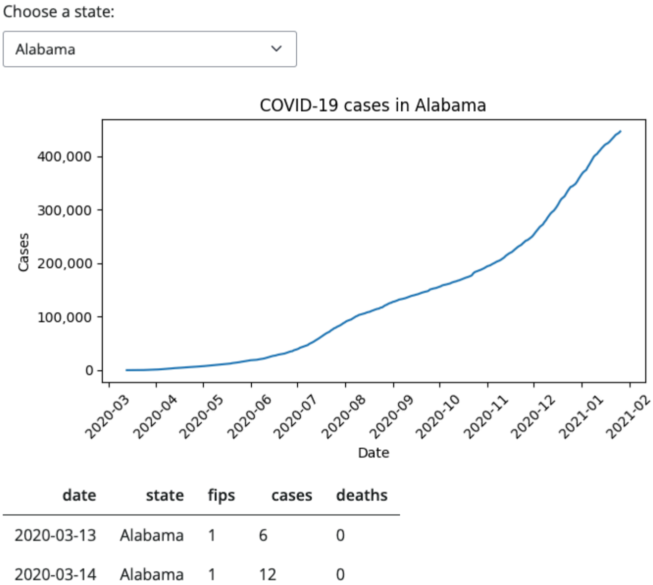
- **Reactive functions** are functions that run when one of their dependencies change
- Ensure values are consistent across different parts of your app
- Minimize redundant and unnecessary re-calculations
- This section used a toy example where it is easy for the computer to re-calculate everything. In the next section, we will turn to a more realistic example.

Case Study: COVID-19 Dashboard I

COVID Data Example: End Goal

Starting with external data on COVID cases and deaths by state, create an app that has:

1. Drop-down list of states
2. Preview of data of user-selected state
3. Time series plot of COVID cases and deaths in selected state (next section)



Choose a state:

Alabama

COVID-19 cases in Alabama

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

COVID Data Example: Roadmap

1. Familiarize ourselves with the data

2. Create skeleton `app.py`

3. Input state as a dropdown list

4. Import data (a reactive calculation)

5. Filter to selected state (another reactive calculation)

6. Display selected state data

COVID Data Example: Data

- Before we work on the app, let's familiarize ourselves with the data: `nyt_covid19_data.csv`

```
df = pd.read_csv("apps_for_class/covid/nyt_covid19_data.csv")
print(df.head())
print("First date: " + str(df['date'].min()))
print("Last date: " + str(df['date'].max()))
```

COVID Data Example: Data

- Before we work on the app, let's familiarize ourselves with the data: `nyt_covid19_data.csv`

```
df = pd.read_csv("apps_for_class/covid/nyt_covid19_data.csv")
print(df.head())
print("First date: " + str(df['date'].min()))
print("Last date: " + str(df['date'].max()))
```

```
         date       state  fips  cases  deaths
0  2020-01-21  Washington    53      1       0
1  2020-01-22  Washington    53      1       0
2  2020-01-23  Washington    53      1       0
3  2020-01-24    Illinois    17      1       0
4  2020-01-24  Washington    53      1       0
First date: 2020-01-21
Last date: 2021-01-26
```

Step 1: Set up basic app structure

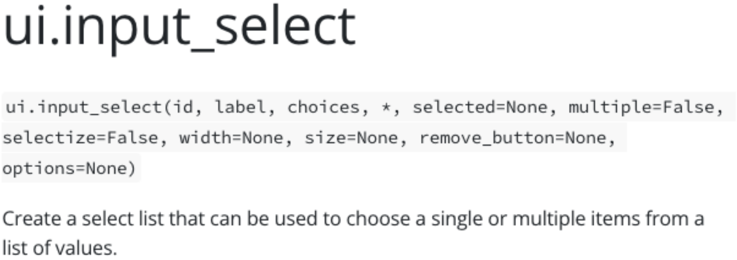
- Your basic app structure (UI + server + call to app) will always be the same
- In covid/app.py:

```
from shiny import App, render, ui, reactive

app_ui = ui.page_fluid(
    # ui code
)
def server(input, output, session):
    # server code

app = App(app_ui, server)
```

Step 2: Drop-down list

- We want to create a drop-down list with every state name

- Documentation for dropdown menu UI



ui.input_select

```
ui.input_select(id, label, choices, *, selected=None, multiple=False,
selectize=False, width=None, size=None, remove_button=None,
options=None)
```

Create a select list that can be used to choose a single or multiple items from a list of values.

Step 2: Drop-down list

- Starting on the **UI side**:

```
from shiny import App, render, ui, reactive

app_ui = ui.page_fluid(
    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"])
)
```

- Hard-coding every state name is not ideal. Next lecture, we'll discuss how to pre-populate this list.

Step 3: Import data

- We will import the data on the **server side** as a reactive function

```
def server(input, output, session):
    @reactive.calc
    def full_data():
        return pd.read_csv("nyt_covid19_data.csv", parse_dates = ['date'])

app = App(app_ui, server)
```

- Question: if we ran the app right now, what would we get?

Step 3: Import data

- We will import the data on the **server side** as a reactive function

```
def server(input, output, session):
    @reactive.calc
    def full_data():
        return pd.read_csv("nyt_covid19_data.csv", parse_dates = ['date'])

app = App(app_ui, server)
```

- Question: if we ran the app right now, what would we get?

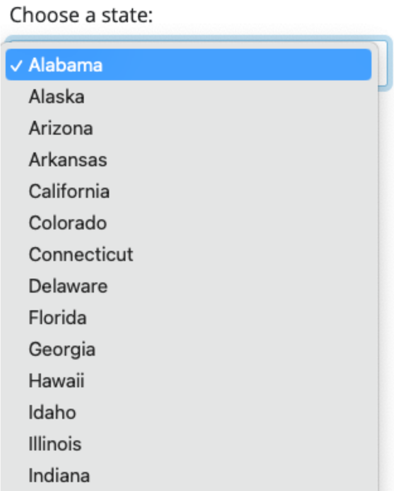
- Answer: it would display a drop-down menu with state names

- And in the background, it would load and store the data as `full_data()`

Run the app

- In terminal:

```
shiny run --reload covid/app.py
```



Step 4: Filter to selected state

- Question: given how we've defined the dropdown menu on the **UI-side** (below), how do we reference the state the user selects on the **server-side**?

```
app_ui = ui.page_fluid(
    ui.input_select(id = 'state', label = 'Choose a state:',
    choices = .... )
)
```

Step 4: Filter to selected state

- Question: given how we've defined the dropdown menu on the **UI-side** (below), how do we reference the state the user selects on the **server-side**?

```
app_ui = ui.page_fluid(
    ui.input_select(id = 'state', label = 'Choose a state:',
    choices = .... )
)
```

- Answer: on the server side, we reference it using `input.state()`

Step 4: Filter to selected state

- On the **server-side**, add a `@reactive.calc` function that returns the subsetted dataframe:

```
def server(input, output, session):
    @reactive.calc
    def full_data():
        return pd.read_csv("nyt_covid19_data.csv", parse_dates = ['date'])

    @reactive.calc  #new function, reacts to input.state()
    def subsetted_data():
        df = full_data()
        return df[df['state'] == input.state()]
```

Step 5: Display selected state data

- Then again on server-side, add a function to make a table (`@render.table()`)

```
def server(input, output, session):
    @reactive.calc
    def full_data():
        return pd.read_csv("nyt_covid19_data.csv", parse_dates = ['date'])

    @reactive.calc
    def subsetted_data():
        df = full_data()
        return df[df['state'] == input.state()]

    @render.table()
    def subsetted_data_table():
        return subsetted_data()
```

- `subsetted_data()`: **reactive** function that does the subsetting
- `subsetted_data_table()`: **render** function that prepares the data for UI

Step 5: Display selected state data

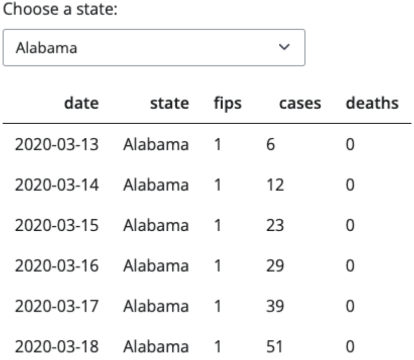
- Back to the **UI-side**, add a UI element for the table of the subsetted data

```
app_ui = ui.page_fluid(
    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"]),
    ui.output_table("subsetted_data_table")
)
```

Run the app again

In Terminal: `shiny run --reload covid/app.py`

Without any user selection, the app will default to the first state

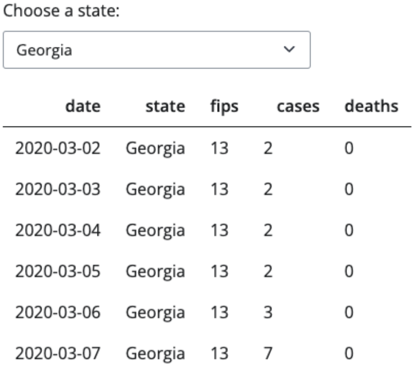


Choose a state:

Alabama

date	state	fips	cases	deaths
2020-03-13	Alabama	1	6	0
2020-03-14	Alabama	1	12	0
2020-03-15	Alabama	1	23	0
2020-03-16	Alabama	1	29	0
2020-03-17	Alabama	1	39	0
2020-03-18	Alabama	1	51	0

But will dynamically update once the user chooses a state



Choose a state:

Georgia

date	state	fips	cases	deaths
2020-03-02	Georgia	13	2	0
2020-03-03	Georgia	13	2	0
2020-03-04	Georgia	13	2	0
2020-03-05	Georgia	13	2	0
2020-03-06	Georgia	13	3	0
2020-03-07	Georgia	13	7	0

COVID Data Example: Summary

- Read in all the data (`full_data()`) and limit to one state(`subsetted_data()`)
- If the state of interest changes, we do not need to read in the data again
- Process tips
  - Develop your apps piece-by-piece and re-run at each step to debug
  - **If the feature takes in user input**: start on UI side, then move to server side

Case Study: COVID-19 Dashboard II

Add a time series plot: roadmap

Now we want to add a timeseries plot to the dashboard

Step 6: Add a time series plot

- **Server side**:

```
def server(input, output, session):
   # [other server components]
   @render.plot
   def ts():
        df = subsetted_data_table()
        fig, ax = plt.subplots(figsize=(6,6))
        ax.plot(df['date'], df['cases'])
        return fig
```

Step 6: Add a time series plot

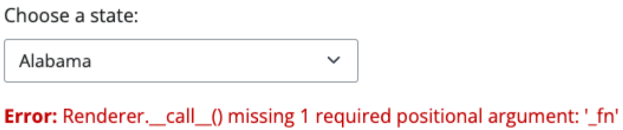
- **Server side**:

```
def server(input, output, session):
   # [other server components]
   @render.plot
   def ts():
        df = subsetted_data_table()
        fig, ax = plt.subplots(figsize=(6,6))
        ax.plot(df['date'], df['cases'])
        return fig
```

- Then on the **UI-side**:

```
app_ui = ui.page_fluid(
    # [other UI components],
    ui.output_plot('ts')
)
```

Try running it...

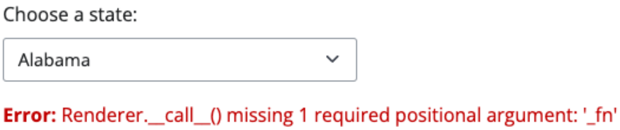


Choose a state:

Alabama

Error: Renderer.__call__() missing 1 required positional argument: '_fn'

Try running it...



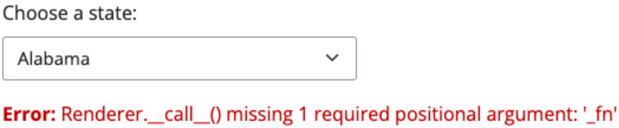
Choose a state:

Alabama

Error: Renderer.__call__() missing 1 required positional argument: '_fn'

- Oops!
- This error message isn't very informative...let's try looking at the terminal output

Try running it...

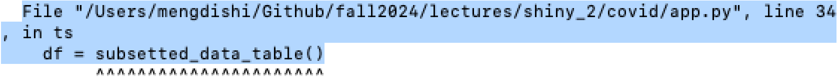


Choose a state:

Alabama

Error: Renderer.__call__() missing 1 required positional argument: '_fn'

- Oops!
- This error message isn't very informative...let's try looking at the terminal output



```
  File "/Users/mengdishi/Github/fall2024/lectures/shiny_2/covid/app.py", line 34
, in ts
    df = subsetted_data_table()
         ^^^^^^^^^^^^^^^^^^^^^^
```

- Question: can you figure out what the issue is here?

Debugging

- Answer: `subsetted_data_table()` is designed to display the data on the dashboard (**render**)

- Instead, we want the output of the **reactive** function `subsetted_data()`

Debugging

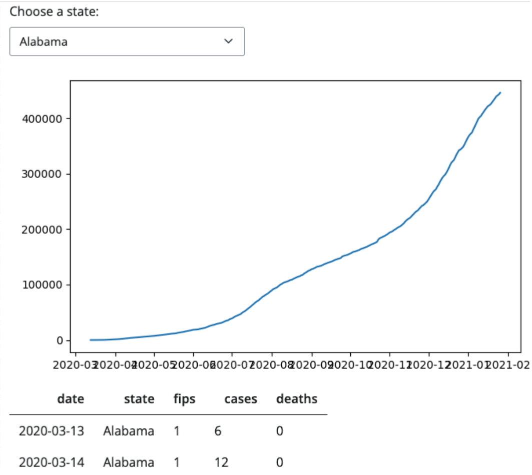
- Answer: `subsetted_data_table()` is designed to display the data on the dashboard (**render**)

- Instead, we want the output of the **reactive** function `subsetted_data()`

```
def server(input, output, session):
    # [other server components]
    @render.plot
    def ts():
        df = subsetted_data()
        fig, ax = plt.subplots(figsize=(6,6))
        ax.plot(df['date'], df['cases'])
        return fig
```

COVID Data Example

- Now when we save and it re-runs, we get what we wanted



Choose a state:

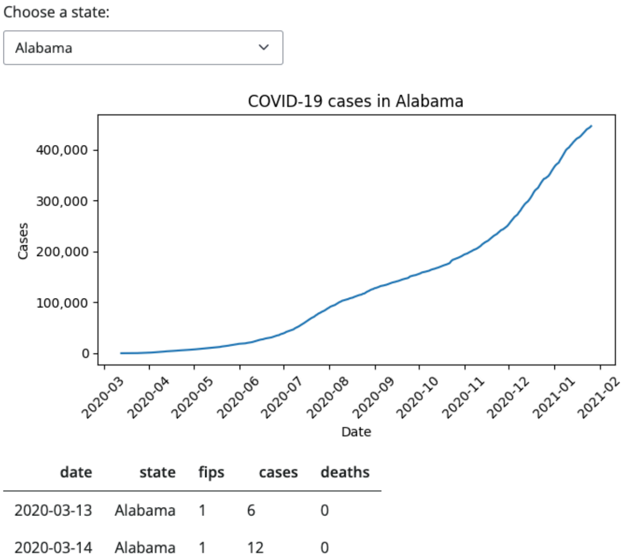
Alabama

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

Improving our Plot

```
def server(input, output, session):
   # [other server components]
   @render.plot
   def ts():
        df = subsetted_data()
        fig, ax = plt.subplots(figsize=(6,6))
        ax.plot(df['date'], df['cases'])
        ax.tick_params(axis = 'x', rotation = 45)
        ax.set_xlabel('Date')
        ax.set_ylabel('Cases')
        ax.set_title(f'COVID-19 cases in {input.st()}')
        ax.set_yticklabels(['{:, .0f}'.format(x) for x in
 ↪  ax.get_yticks()])
        return fig
```

COVID Data Example



Choose a state:

Alabama

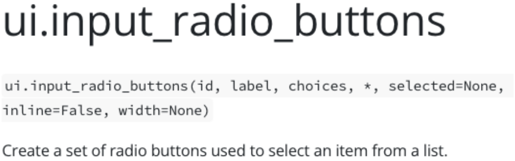
date	state	fips	cases	deaths
2020-03-13	Alabama	1	6	0
2020-03-14	Alabama	1	12	0

COVID Data Example: Summary

- We wrote a new server side function `ts()`
- We added `ui.output_plot('ts')`
- Bug. Useful message only available at terminal
- Use reactive decorated functions to load data (not render decorated functions)

Do-pair-share: adding to the app

- App from last class shared in `student30538/before_lecture/shiny_11/apps_for_class/covid/app.py`
- Now add one more piece to the app: **radio buttons** that allow user to choose if they want to display cases or deaths



ui.input_radio_buttons

```
ui.input_radio_buttons(id, label, choices, *, selected=None,
inline=False, width=None)
```

Create a set of radio buttons used to select an item from a list.

Hints: start with the easier step of adding radio buttons on the UI side. then move to the harder step of modifying the function `ts()` to choose which data to analyze based on the radio button input.

Whole Lecture Summary

- We've covered a core component of dashboards: **reactive programming**
- Reactive functions track of dependencies and will only re-run a piece of code when it detects one of its dependencies has changed
- This allows the dashboard to react and dynamically update based on user input
- Application: app that dynamically filters and plots COVID data based on user-selected state