Shiny for Python

Lecture 22

Dr. Colin Rundel


Shiny
www.rstudio.com

Shiny for Python

Shiny is also available as a Python package, `shiny`. It is a separate implementation written in Python, not a wrapper around the R package.

Almost everything from the last two lectures carries over.

- The same split between a UI and a server function
- The same reactive model with inputs, outputs, reactive expressions, and observers
- The same browser-side code, so apps look and behave the same
- The `bslib` components (cards, value boxes, sidebars, layouts) are built in

What changes is the syntax, and most of today is about mapping what we already know from R onto Python.

Getting started

Add the package to a project with `uv`,

```bash
uv add shiny
```

The app is written as a python script and then run from the command line via:

```bash
uv run shiny run --reload app.py
```

The `--reload` flag watches `app.py` and restarts the app every time the file is saved.

Positron and VS Code have a Shiny extension that adds a run button and shows the app in a viewer pane, much like RStudio does for R.

The button only appears for a file named `app.py`, or one whose name starts with `app_` or ends with `_app`, such as

Core and Express

Shiny for Python has two syntaxes for writing an app.

- Shiny Express mixes UI and server code together in one script. It is compact and reads like a notebook, which makes it nice for small apps. (Competes with streamlit)
- Shiny Core keeps a UI object and a server function separate, just like R.

We will use Core throughout. It maps directly onto the R apps we have written, and what you learn transfers to Express easily.

Some Python syntax

Functions inside functions

A function can be defined inside another function. The inner function can use the variables of the outer one, including its arguments.

```py
def power_func(k):
    def power(x):
        return x**k

    return power
```

```py
square = power_func(2)
square(4)
```

```
16
```

A Shiny server function works the same way. Every output and reactive is a function defined inside `server()`, which is how they can all see `input`.

Decorators

A decorator is a function that takes a function and returns something to use in its place, usually another function.

```py
def shout(f):
    def wrapper(name):
        return f(name).upper()

    return wrapper
```

```py
def greet(name):
    return f"Hello, {name}!"
```

```py
greet("Colin")
```

```
'Hello, Colin!'
```

```py
shout(greet)("Colin")
```

```
'HELLO, COLIN!'
```

```py
@shout
def greet(name):
    return f"Hello, {name}!"
```

```py
greet("Colin")
```

```
'HELLO, COLIN!'
```

Decorators that register

Here `render()` stores each function by name in a dictionary and returns it unchanged.

```py
outputs = {}

def render(f):
    outputs[f.__name__] = f
    return f
```

```py
@render
def plot():
    return "a plot"

@render
def table():
    return "a table"
```

We can then access those functions from the `outputs` dictionary,

```py
list(outputs)
```

```
['plot', 'table']
```

```py
outputs["plot"]()
```

```
'a plot'
```

This is roughly what Shiny does. The name of a decorated function is the id of the output it fills.

Arguments and stacking

A decorator can take arguments. `repeat(2)` is called first and returns the decorator to apply.

```py
def repeat(n):
    def decorator(f):
        def wrapper(name):
            return [f(name) for _ in range(n)]
        return wrapper
    return decorator
```

```py

@repeat(2)
def greet(name):
    return f"Hello, {name}!"
```

```py
greet("Colin")
```

```
['Hello, Colin!', 'Hello, Colin!']
```

```py
@repeat(2)
@shout
def greet(name):
    return f"Hello, {name}!"
```

```py
greet("Colin")
```

```
['HELLO, COLIN!', 'HELLO, COLIN!']
```

Stacked decorators are applied from the bottom up, so the order matters.

Positional and keyword arguments

In a Python call all positional arguments must come before any keyword arguments.

```
f(pos1, pos2, key1=val1, key2=val2)
```

positional — keyword

In shiny the UI functions take their children as *positional arguments*, so the named options that R code often puts first have to move to the end.

R

```r
card(
  full_screen = TRUE,
  card_header("Plot"),
  plotOutput("plot")
)
```

py

```python
ui.card(
    ui.card_header("Plot"),
    ui.output_plot("plot"),
    full_screen=True,
)
```

From R to Python

Anatomy of an app

R

```r
library(shiny)
library(bslib)

ui = page_fluid()

server = function(input, output, session) {

}

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

py

```python
from shiny import App, ui


app_ui = ui.page_fluid()

def server(input, output, session):
    ...


app = App(app_ui, server)
```

- `ui` holds the UI functions and `App` ties the pieces together.
  - Most apps also import `render` for the output renderers and
  - `reactive` for the reactive tools.
- The UI object is called `app_ui` by convention, since the name `ui` is taken by the module.
- `shiny run app.py` looks for an object named `app` by default. Another name can be given as `shiny run app.py:my_app`.

UI function names

Every UI function lives in `ui` and uses `snake_case`. Inputs start with `input_` and outputs start with `output_`.

R	Python
`sliderInput()`	`ui.input_slider()`
`numericInput()`	`ui.input_numeric()`
`selectInput()`	`ui.input_select()`
`actionButton()`	`ui.input_action_button()`
`plotOutput()`	`ui.output_plot()`
`textOutput()`	`ui.output_text()`
`uiOutput()`	`ui.output_ui()`
`h4()`, `p()`, `div()`	`ui.h4()`, `ui.p()`, `ui.div()`
`page_sidebar()`, `sidebar()`	`ui.page_sidebar()`, `ui.sidebar()`
`card()`, `value_box()`	`ui.card()`, `ui.value_box()`

The `bslib` functions keep their names, since they were already in `snake_case` with consistent prefixes.

Translating a UI

R

```r
ui = page_sidebar(
  title = "Beta-Binomial Visualizer",
  sidebar = sidebar(
    h4("Data:"),
    sliderInput(
      "x", "# of heads",
      min = 0, max = 100, value = 7
    ),
    sliderInput(
      "n", "# of flips",
      min = 0, max = 100, value = 10
    )
  ),
  card(
    class = "p-2",
    plotOutput("plot")
  )
)
```

py

```python
app_ui = ui.page_sidebar(
    ui.sidebar(
        ui.h4("Data:"),
        ui.input_slider(
            "x", "# of heads",
            min=0, max=100, value=7,
        ),
        ui.input_slider(
            "n", "# of flips",
            min=0, max=100, value=10,
        ),
    ),
    ui.card(
        ui.output_plot("plot"),
        class_="p-2",
    ),
    title="Beta-Binomial Visualizer",
)
```

- The sidebar is now the first positional argument, and `title` moves to the end.
- `class` is reserved in Python, so the argument is `class_`.

Reading inputs

In R an input is read as `input$x`. In Python it is `input.x()`, with parentheses.

R
```
input$n <- input$x
```

py
```
input.n() <- input.x()
```

Python is being more explicit here. Every reactive value is an object, and calling it both returns the current value and records the dependency. This is the same convention R already uses for reactive expressions.

Leaving off the parentheses is the most common mistake when coming from R. The result is the reactive object and not its value, which usually leads to a weird type error.

py
```
input.n <- input.x
#> TypeError: unsupported operand type(s) for <-: 'Value' and 'Value'
```

Outputs

Outputs are where the decorators from earlier come in. Each output is a decorated function defined inside `server()`.

R

```r
output$summary_title = renderText({
  paste(
    "Summary Statistics for the",
    input$summary_dist
  )
})
```

py

```python
@render.text
def summary_title():
    return (
        "Summary Statistics for the "
        + input.summary_dist()
    )
```

- A `render*()` function wrapped around a block of code becomes a `@render` decorator on a function with no arguments.
- There is no assignment to `output`. The function name is the output id, so `summary_title()` fills `ui.output_text("summary_title")`.
- The function needs an explicit `return` (a Python function without a `return` returns `None`).

Plots and tables

`@render.plot` works with `matplotlib` and anything built on it, such as `seaborn` and `plotnine`. Return the figure or axes, or draw with `pyplot` and return nothing.

```py
@render.plot
def plot():
    ax = sns.lineplot(d(), x="p", y="density", hue="distribution")
    for line in ax.lines:
        ax.fill_between(*line.get_data(), color=line.get_color(), alpha=0.3)
    return ax
```

For tables use `@render.data_frame` with `ui.output_data_frame()`, which shows a `pandas` or `polars` data frame as a sortable grid.

```py
@render.data_frame
def table():
    return d()
```

Reactive expressions

`reactive()` becomes the `@reactive.calc` decorator. The result is used in the same way in both languages, by calling `d()` from another reactive context.

Nothing about the behavior changes. A calc is lazy, it caches its value, and it is invalidated when any input it read changes.

R

```r
d = reactive({
  tibble(
    p = seq(0, 1, length.out = 1001),
    prior = dbeta(p, input$a, input$b)
  )
})
```

py

```python
@reactive.calc
def d():
    p = np.linspace(0, 1, 1001)
    return pd.DataFrame({
        "p": p,
        "prior": stats.beta.pdf(
            p, input.a(), input.b()
        ),
    })
```

Observers

`observe()` becomes `@reactive.effect`. The name reflects what an observer is for, which is running code for its side effects.

R
```r
observe({
  updateSliderInput(
    session, "x", max = input$n
  )
})
```

py
```python
@reactive.effect
def _():
    ui.update_slider(
        "x", max=input.n()
    )
```

- An effect is never called by name, so by convention the function is named _. Reusing _ for different effects is fine. `reactive.calc`s and `render`s need unique names.
- The `update*()` functions are in `ui` as `ui.update_slider()`, `ui.update_numeric()`, and so on. They find the current session on their own, so `session` is not passed.

Events

`bindEvent()` becomes the `@reactive.event()` decorator, which takes the reactive value(s) to listen to.

R

```r
d = reactive({
  ...
}) |>
  bindEvent(input$run)
```

py

```python
@reactive.calc
@reactive.event(input.run)
def d():
    ...
```

- The event is passed as `input.run` without parentheses. We are handing over the reactive object to watch, not reading its current value.
- Multiple events are given as separate arguments, `@reactive.event(input.run, input.reset)`, just like `bindEvent`.
- `@reactive.event()` goes directly above the `def`, underneath `@reactive.calc`, `@reactive.effect`, or `@render.*`. Decorators apply from the bottom up, so the event wraps the function first. The wrong order is an error.
- `ignoreNULL` and `ignoreInit` become `ignore_none` and `ignore_init`.

Dynamic UI

`renderUI()` becomes `@render.ui`, paired with `ui.output_ui()`. A list comprehension takes the place of `map()`, and `*` the place of `!!!`.

R

```r
output$boxes = renderUI({
  boxes = map(
    input$dists,
    \(dist) value_box(
      title = dist,
      value = post_mode(dist)
    )
  )

  layout_column_wrap(
    width = "200px", !!!boxes
  )
})
```

py

```python
@render.ui
def boxes():
    boxes = [
        ui.value_box(
            dist,
            post_mode(dist),
        )
        for dist in input.dists()
    ]

    return ui.layout_column_wrap(
        *boxes, width="200px"
    )
```

Requiring values

`req()` works the same way in both languages: if any argument is missing, the calculation stops quietly and the output stays blank.

R
```r
output$tbl = renderTable({
  req(input$file)
  read.csv(input$file$datapath)
})
```

py
```python
@render.table
def tbl():
    file = req(input.file())
    return pd.read_csv(file[0]["datapath"])
```

- `req()` is imported from `shiny` alongside `App`, `reactive`, `render`, and `ui`. It receives values, so the inputs are called, `input.file()`.
- It returns its first argument, so checking and reading a value can be one line.
- "Missing" means falsy in Python: `None`, `False`, `""`, and empty lists, but also `0`, which R treats as a valid value. A data frame cannot be passed to `req()` at all, since pandas raises an error when asked whether one is true.

Showing a message

There is no `validate()` or `need()`. Instead, raise a `SafeException` and its message is shown in place of the output.

R

```r
output$plot = renderPlot({
  validate(
    need(input$x <= input$n,
         "Too many heads")
  )
  ...
})
```

py

```python
from shiny.types import SafeException

@render.plot
def plot():
    if input.x() > input.n():
        raise SafeException("Too many heads")
    ...
```

- Any exception's message is shown while developing, but deployed apps (e.g. on Posit Connect) usually replace it with a generic error. A `SafeException`'s message is always shown.
- Raised inside a `@reactive.calc`, it reaches every output that uses the calc, just like `validate()`.
- The message is styled as an error, not the gray text `validate()` gives.

Other pieces

A few more tools have direct equivalents in Python.

R	Python
`reactiveVal(0)`	`reactive.value(0)`
`isolate(input$x)`	`with reactive.isolate():`
`invalidateLater(1000)`	`reactive.invalidate_later(1)`
`downloadHandler()`	`@render.download`
`tags$small()`	`ui.tags.small()`
`HTML()`, `markdown()`	`ui.HTML()`, `ui.markdown()`

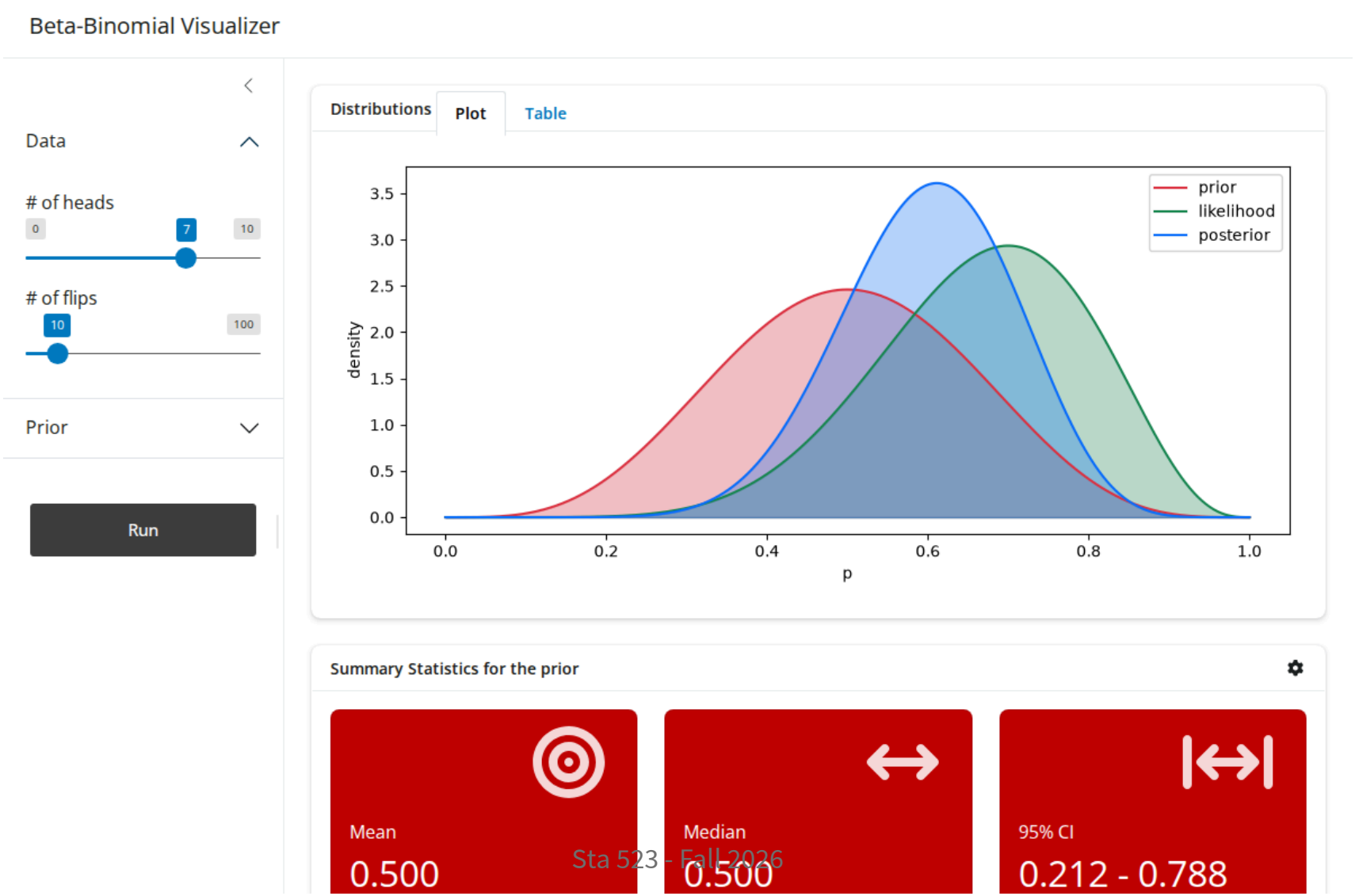
Common mistakes

- Writing `input.x` instead of `input.x()` when reading a value, or `input.run()` instead of `input.run` inside `@reactive.event()`.
- Forgetting to `return` the value from a calc or an output.
- An output function whose name does not match its id in the UI. Nothing fails and the output just stays empty.
- Two calcs or outputs with the same name. The second definition silently replaces the first.
- Putting `@reactive.event()` above `@reactive.calc` or `@render.*` instead of below it.
- Keyword arguments ahead of the children in a UI function, which is a syntax error.
- A missing comma between UI components. The error message often points at the wrong line.

Beta-Binomial app

The app in Python

Our goal is the app from last lecture, without the pieces that depended on `bs_theme()`.



What we need

The statistical pieces change along with the Shiny ones.

R	Python
`seq(0, 1, length.out = 1001)`	`np.linspace(0, 1, 1001)`
`dbeta(p, a, b)`	`stats.beta.pdf(p, a, b)`
`dbinom(x, n, p)`	`stats.binom.pmf(x, n, p)`
`tibble()`	`pd.DataFrame()`
`pivot_longer()`	`.melt()`
`summarize(..., .by = distribution)`	`.groupby("distribution").agg(...)`
`ggplot()` with `geom_line()`	`sns.lineplot()`
`geom_ribbon()`	`ax.fill_between()`
`bsicons::bs_icon()`	`faicons.icon_svg()`

```py
import numpy as np
import pandas as pd
import seaborn as sns
from scipy import stats
from faicons import icon_svg
from shiny import App, reactive, render, ui
```

Example

Translate the Beta-Binomial app from last lecture into Shiny for Python.

Add the packages used by the app and the accompanying examples:

```bash
uv add shiny numpy pandas matplotlib scipy faicons seaborn
```

Publishing

Publishing a Python app

The options are the same as for R, and so is the advice about keeping an app in a single folder with relative paths.

- Posit Connect Cloud publishes Python apps from a GitHub repository or from Positron. It needs a `requirements.txt` listing the app's packages, which `uv` can write.

```bash
uv export --no-hashes --format requirements.txt -o requirements.txt
```

- `shinylive` bundles the app as a static site that runs entirely in the browser using Pyodide. Packages with compiled code need Pyodide-compatible builds; many pure-Python packages also work. `numpy`, `pandas`, `scipy`, and `matplotlib` are available.

```bash
uv add shinylive
uv run shinylive export myapp site
uv run python -m http.server --directory site
```

Summary

R to Python

Concept	R	Python
app object	`shinyApp(ui, server)`	`app = App(app_ui, server)`
run	`runApp()`	`shiny run --reload app.py`
input widget	`sliderInput("x", ...)`	`ui.input_slider("x", ...)`
output placeholder	`plotOutput("plot")`	`ui.output_plot("plot")`
read an input	`input$x`	`input.x()`
output	`output$plot = renderPlot({...})`	`@render.plot` on `def plot():`
reactive expression	`d = reactive({...})`	`@reactive.calc` on `def d():`
observer	`observe({...})`	`@reactive.effect` on `def _():`
event	`bindEvent(input$run)`	`@reactive.event(input.run)`
update an input	`updateSliderInput(session, "x", ...)`	`ui.update_slider("x", ...)`
dynamic UI	`uiOutput()`, `renderUI()`	`ui.output_ui()`, `@render.ui`
splice a list	`!!!boxes`	`*boxes`
HTML class	`class = "p-2"`	`class_="p-2"`

Takeaways

- Shiny for Python is a separate implementation of the same design. The UI and server split, the reactive graph, and the `bslib` components all carry over from R.
- A decorator passes the function below it to another function and rebinds the name to the result. Shiny uses decorators wherever R wraps a block of code in a function call, so `renderPlot({})`, `reactive({})`, and `observe({})` become `@render.plot`, `@reactive.calc`, and `@reactive.effect`.
- An output is matched to the UI by the name of its function. Most output renderers require an explicit `return`; `@render.plot` can also capture pyplot drawing without one.
- Reactive values are read by calling them, as in `input.x()`. They are passed without parentheses to `@reactive.event()`, which sits directly above the `def`.
- UI functions live in `ui` with `snake_case` names. Children come first as positional arguments and named options follow.

Learning more

- Shiny for Python documentation, particularly the pages on reactivity and Express vs Core
- Components and layouts galleries, with code for each
- API reference for the Core syntax
- Shiny for R users, a short comparison of the two versions
- Primer on Python decorators from Real Python
- shinylive for Python and the Posit Connect Cloud docs