Shiny for Python : : CHEAT SHEET

Build an App

A **Shiny** app is an interactive web page (**ui**) powered by a live Python session run by a **server** (or by a browser with Shinylive).

Users can manipulate the UI, which will cause the server to update the UI's displays (by running Python code).

Save your app as **app.py** in a directory with the files it uses.

```
●●● app-name
  app.py
  www/
  <other>/
```

The directory name is the app name.

Place images, CSS, etc. to share with the browser in a folder named "**www.**"

Run `shiny create .` in the terminal to generate a template app.py file

Launch apps with `shiny run app.py --reload`

Nest Python functions to build an HTML interface

Add Inputs with ui.input_*() functions

Add Outputs with ui.ouput_*() functions

Designate output functions with the @output decorator

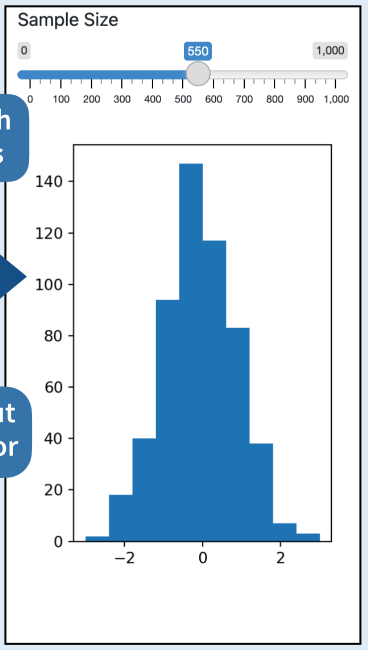
For each output, define a function that generates the output

Call the values of UI inputs with input.<id>()

Layout the UI with Layout Functions

Specify the type of output with a @render. decorator

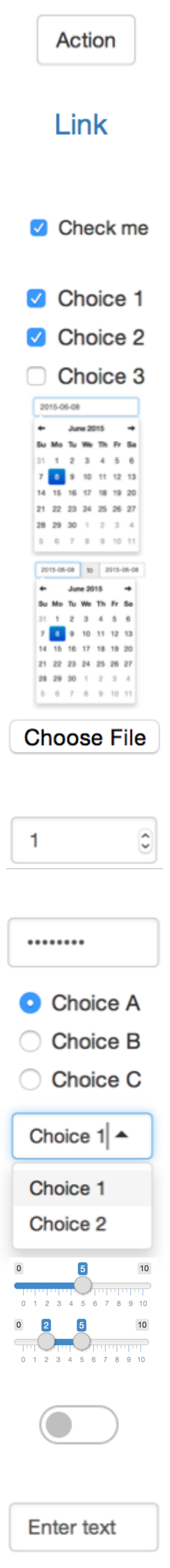
```
from shiny import App, render, ui
import matplotlib.pyplot as plt
import numpy as np

app_ui = ui.page_fluid(
  ui.input_slider(
    "n", "Sample Size", 0, 1000, 20
  ),
  ui.output_plot("dist")
)

def server(input, output, session):
  @output
  @render.plot
  def dist():
    x = np.random.randn(input.n())
    plt.hist(x, range=[-3, 3])

app = App(app_ui, server)
```



Call App() to combine app_ui and server into an interactive app

Inputs

Use a ui. function to make an input widget that saves a value as **<id>**. Input values are *reactive* and need to be called as **<id>()**.



ui.input_action_button(id, label, icon, width, …**)**

ui.input_action_link(id, label, icon, …**)**

ui.input_checkbox(id, label, value, width**)**

ui.input_checkbox_group(id, label, choices, selected, inline, width**)**

ui.input_date(id, label, value, min, max, format, startview, weekstart, language, width, autoclose, datesdisabled, daysofweekdisabled**)**

ui.input_date_range(id, label, start, end, min, max, format, startview, weekstart, language, separator, width, autoclose**)**

ui.input_file(id, label, multiple, accept, width, buttonLabel, placeholder, capture**)**

ui.input_numeric(id, label, value, min, max, step, width**)**

ui.input_password(id, label, value, width, placeholder**)**

ui.input_radio_buttons(id, label, choices, selected, inline, width**)**

ui.input_select(id, label, choices, selected, multiple, selectize, width, size**)**
Also **ui.input_selectize()**

ui.input_slider(id, label, min, max, value, step, ticks, animate, width, sep, pre, post, timeFormat, timezone, dragRange**)**

ui.input_switch(id, label, value, width**)**

ui.input_text(id, label, value, width, placeholder, autocomplete, spellcheck**)**
Also **ui.input_text_area()**

Share

Share your app in three ways:

1. **Host it on shinyapps.io**, a cloud based service from Posit. To deploy Shiny apps:

   Create a free or professional account at **shinyapps.io**

   Use the reconnect-python package to publish with **rsconnect deploy shiny <path to directory>**

2. **Purchase Posit Connect**, a publishing platform for R and Python.

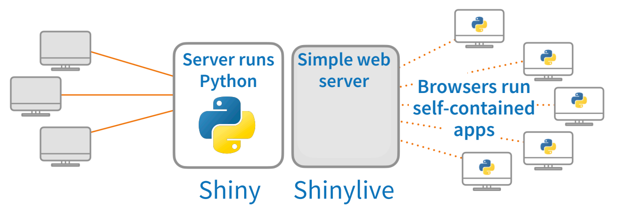
   posit.co/connect

3. **Use open source deployment options**

   shiny.posit.co/py/docs/deploy.html

Shinylive

Shinylive apps use WebAssembly to run entirely in a browser–no need for a special server to run Python.



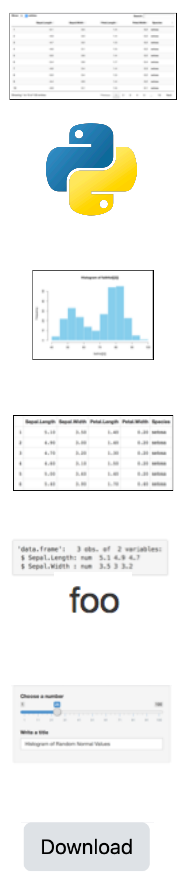
- Edit and/or host Shinylive apps at **shinylive.io**
- Create a Shinylive version of an app to deploy with `shinylive export myapp site` Then deploy to a hosting site like Github or Netlify
- Embed Shinylive apps in Quarto sites, blogs, etc.

~~~
---
filters:
 - shinylive
---
An embedded Shinylive app:
```{shinylive-python}
#| standalone: true
# [App.py code here…]
```
~~~

To embed a Shinylive app in a Quarto doc, include the bold syntax.

Outputs

Match **ui.output_*** functions to **@render.*** decorators to link Python output to the UI.



ui.output_data_frame(id**)**
@render.data_frame

ui.output_image(id, width, height, click, dblclick, hover, brush, inline**)**
@render.image

ui.output_plot(id, width, height, click, dblclick, hover, brush, inline**)**
@render.plot

ui.output_table(id**)**
@render.table

ui.output_text_verbatim(id, …**)**
ui.output_text(id, container, inline**)**
@render.text

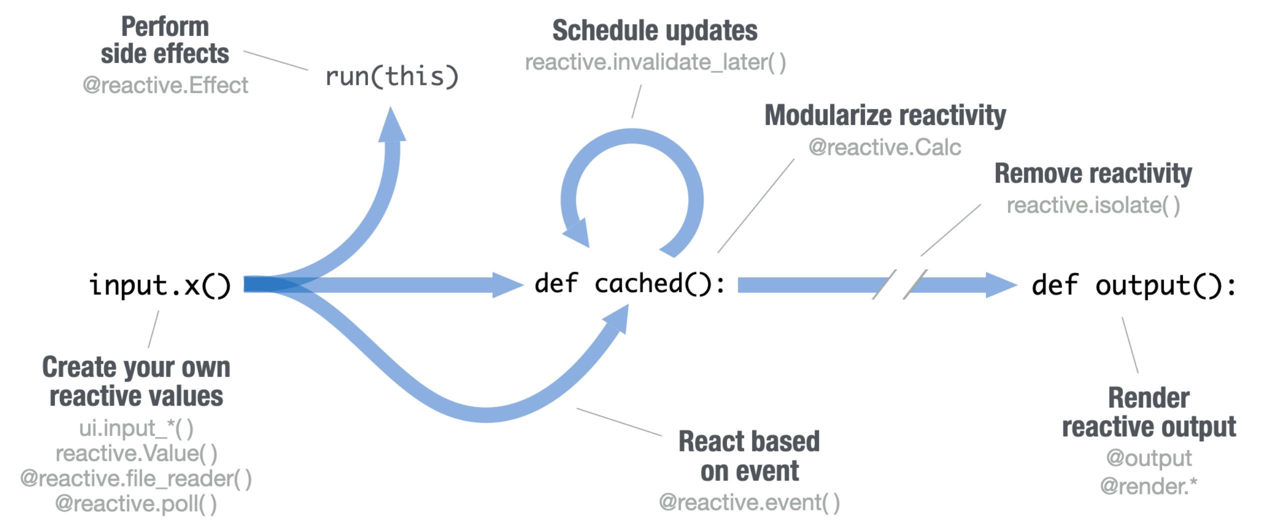
ui.output_ui(id, inline, container, …**)**
ui.output_html(id, inline, container, …**)**
@render.ui

ui.download_button(id, label, icon, …**)**
@session.download

posit™

Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error `No current reactive context.`



CREATE YOUR OWN REACTIVE VALUES

```
# ...
app_ui = ui.page_fluid(
 ui.input_text("a", "A")
)

def server(
 input, output, session
):
 rv = reactive.Value()
 rv.set(5)
# ...
```

ui.input_*() makes an input widget that saves a reactive value as **input.<id>()**.

reactive.value( )
Creates an object whose value you can set.

DISPLAY REACTIVE OUTPUT

```
app_ui = ui.page_fluid(
 ui.input_text("a", "A"),
 ui.output_text("b"),
)
def server(
 input, output, session
):
 @output
 @render.text
 def b():
  return input.a()
```

ui.output_*() adds an output element to the UI.

@output
@render.*
Decorators to identify and render outputs

def <id>(): Code to generate the output

CREATE REACTIVE EXPRESSIONS

```
# ...
def server(
 input, output, session
):
 @reactive.Calc
 def re():
  return input.a() + input.b()
# ...
```

@reactive.Calc
Makes a function a reactive expression. Shiny notifies functions that use the expression when it becomes invalidated, triggering recomputation. Shiny caches the value of the expression while it is valid to avoid unnecessary computation.

PERFORM SIDE EFFECTS

```
# ...
def server(
 input, output, session
):
 @reactive.Effect
 @reactive.event(input.a)
 def print():
   print("Hi")
# ...
```

@reactive.Effect
Reactively trigger a function with a side effect. Call a reactive value or use @reactive.event to specify when the function will rerun.

REACT BASED ON EVENT

```
# ...
def server(
 input, output, session
):
 @reactive.Calc
 @reactive.event(input.a)
 def re():
  return input.b()
# ...
```

@reactive.event( )
Makes a function react *only when* a specified value is invalidated, here input.a.

REMOVE REACTIVITY

```
# ...def server(
 input, output, session
):
 @output
 @render.text
 def a():
  with reactive.isolate():
   return input.a()
# ...
```

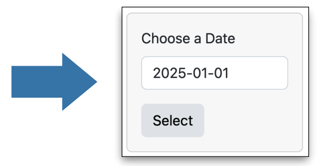
reactive.isolate()
Create non-reactive context within a reactive function. Calling a reactive value within this context will *not* cause the calling function to re-execute should the value become invalid.

Layouts

Combine multiple elements into a "single element" that has its own properties with a panel function:

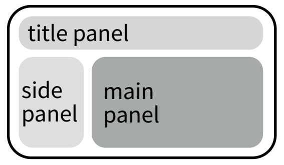
ui.panel_absolute()	ui.panel_sidebar()
ui.panel_conditional()	ui.panel_title()
ui.panel_fixed()	ui.panel_well()
ui.panel_main()	ui.row() / ui.column()

```
ui.panel_well(
 ui.input_date(...),
 ui.input_action_button(...)
)
```



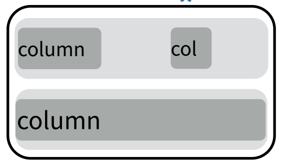
Layout panels with a layout function. Add elements as arguments of the layout functions.

ui.layout_sidebar()



```
app_ui = ui.page_fluid(
 ui.panel_title( ),
 ui.layout_sidebar(
  ui.panel_sidebar( ),
  ui.panel_main( ),
 ))
```

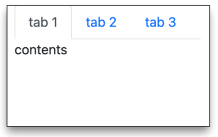
ui.row()



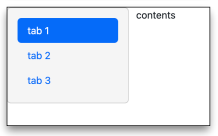
```
app_ui = ui.page_fluid(
 ui.row(
  ui.column(width = 4),
  ui.column(width = 2, offset = 3),
 ),
 ui.row(ui.column(width = 12)))
```

Layer **ui.nav( )**s on top of each other, and navigate between them, with:

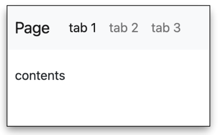
```
ui.page_fluid(ui.navset_tab(
  ui.nav("tab 1", "contents"),
  ui.nav("tab 2", "contents"),
  ui.nav("tab 3", "contents")))
```



```
ui.page_fluid(ui.navset_pill_list(
  ui.nav("tab 1", "contents"),
  ui.nav("tab 2", "contents"),
  ui.nav("tab 3", "contents")))
```



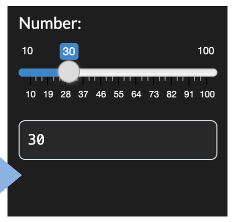
```
ui.page_navbar(
  ui.nav("tab 1", "contents"),
  ui.nav("tab 2", "contents"),
  ui.nav("tab 3", "contents"),
  title = "Page")
```



Themes

Use the **shinyswatch** package to add existing bootstrap themes to your Shiny app ui.

```
import shinyswatch

app_ui = ui.page_fluid(
 shinyswatch.theme.darkly(),
 ...
)
```



Shiny for R Comparison

Shiny for Python is quite similar to Shiny for R with a few important differences:

1. Call inputs as **input.<id>()**	input$x	input.x()
2. Use **decorators** to create and render outputs. Define outputs as functions **def <id>():**	output$y <- renderText(z())	@output @renderText def y(): return z()
3. To create a reactive expression, use **@reactive.Calc**	z <- reactive({ input$x + 1 })	@reactive.Calc def z(): return input.x()+1
4. To create an observer, use **@reactive.Effect**	a <- observe({ print(input$x) })	@reactive.Effect def a(): print(input.x())
5. Combine these with **@reactive.event**	b <- eventReactive(input$goCue, {input$x + 1})	@reactive.Calc @reactive.event(input.go_cue) def b(): return input.x()+1
6. Use **reactive.Value()** instead of reactiveVal()	reactiveVal(1)	reactive.Value(1)
7. Use **nav_*()** instead of *Tab()	insertTab() appendTab() etc.	nav_insert() nav_append() etc.
8. Functions are intuitively organized into submodules	dateInput() textInput() etc.	ui.input_date() ui.input_text() etc.

posit™