Shiny Core Documentation Backup

Note: the documentation has now been moved from Shiny's website, so we are using 'historic backup' of their page. Below document has been cleaned up a bit to make learning easier, but you can access the 'raw' version of the backup following the link below. In there you will find even more materials, especially about the 'Reactivity' of shiny programming and other advanced concepts.

https://web.archive.org/web/20230607090914/https://shiny.rstudio.com/py/docs/overview.html

Also the current official 'list of all the things you can do' in shiny Core is under below link. It's a really good reference when you are building stuff (examples are great), but it is not the easiest way to learn. https://shiny.posit.co/py/api/core/

1.Essentials

Overview

Shiny makes it easy to build web applications with Python code. It enables you to customize the layout and style of your application and dynamically respond to events, such as a button press, or dropdown selection.

If you have experience with the Shiny for R, we recommend starting with the quickstart to learn the main differences between Shiny for R and Shiny for Python.

Shiny applications consist of two parts: the user interface (or UI), and the server function. These are combined using a shiny.App object.

This is shown in the interactive example below.

```
from shiny import App, ui

# Part 1: ui ----
app_ui = ui.page_fluid(
    "Hello, world!",
)

# Part 2: server ----
def server(input, output, session):
    ...

# Combine into a shiny app.
# Note that the variable must be "app".
app = App(app_ui, server)
```

Notice how the UI part defines what visitors will see on their web page. Right now this is just the static text “Hello, world!”. The dynamic parts of our app happen inside the server function, which is currently empty.

Note
You can modify the example code, and then re-run it by pressing the play button on the top-right of the code pane.

Try changing the page text to "Hello, Shiny world!" and re-running the application.

In the next sections we'll modify the UI and server to take input from a user, and display some calculations!

Adding UI inputs and outputs

The first step toward basic interactivity is to add inputs and outputs to the UI.

```
from shiny import App, ui

app_ui = ui.page_fluid(
    ui.input_slider("n", "Choose a number n:", 0, 100, 40),
    ui.output_text_verbatim("txt")
)

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

app = App(app_ui, server)
```

Note the two new UI pieces added:

input_slider() creates a slider.
output_text_verbatim() creates a field to display dynamically generated text. There's no text yet, but we'll add it next.

Adding server logic

Now we can add to the server function. Inside of the server function, we'll define an output function named txt. This output function provides the content for the output_text_verbatim("txt") in the UI.

Try moving the slider below to see the text output automatically change.

LIVE: Shiny editor

```
from shiny import ui, render, App

app_ui = ui.page_fluid(
    ui.input_slider("n", "N", 0, 100, 40),
    ui.output_text_verbatim("txt"),
)
```

```
def server(input, output, session):
    @output
    @render.text
    def txt():
        return f"n*2 is {input.n() * 2}"

# This is a shiny.App object. It must be named `app`.
app = App(app_ui, server)
```

Note that inside of the server function, we do the following:

define a function named txt, whose output shows up in the UI's output_text_verbatim("txt").
decorate it with @render.text, to say the result is text (and not, e.g., an image).
decorate it with @output, to say the result should be displayed on the web page.
(Soon we'll see other kinds of render.* decorators, like render.plot.)

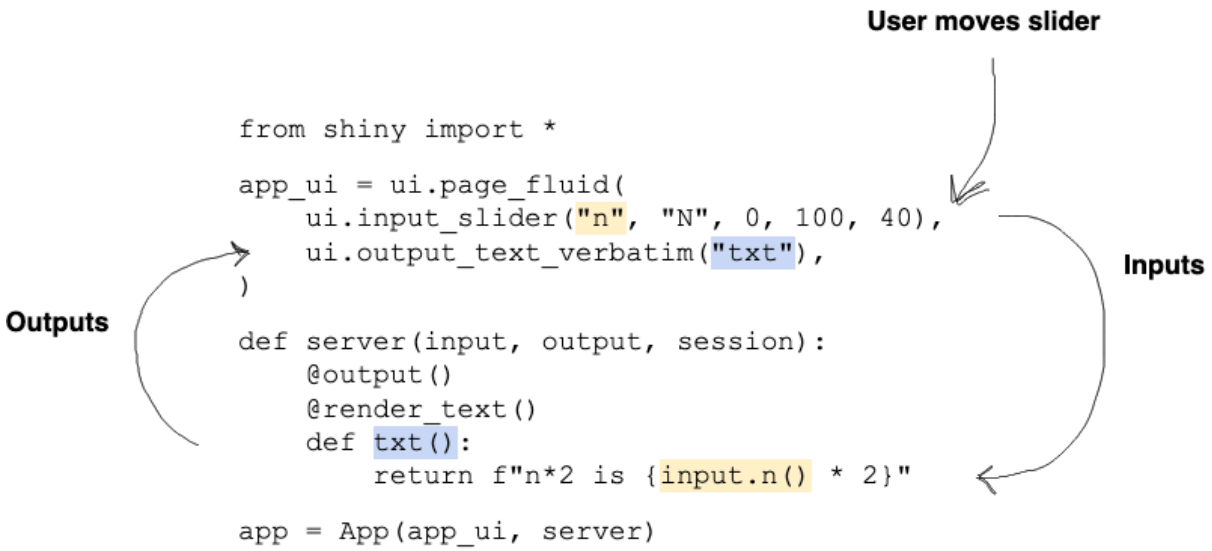
Finally, notice our txt() function takes the value of our slider n, and returns its valued multiplied by 2. To access the value of the slider, we use input.n(). Notice that this is a callable object (like a function) that must be invoked to get the value.

This reactive flow of data from UI inputs, to server code, and back out to UI outputs is fundamental to how Shiny works.

Reactive flow

When you moved the slider in the app above, a series of actions were kicked off, resulting in the output on the screen changing. This is called reactivity.

The diagram below shows how the updated input flows through a Shiny application.



Reactive flow of code through a Shiny application
Inputs, like our slider n, are reactive values: when they change, they automatically cause any the reactive functions that use them (like txt()) to recalculate.

Laying out your UI

So far, our UI has consisted of input and output components. Shiny also has layout components that can contain other components and visually arrange them. Examples of these are sidebar layouts, tab navigation, and cards.

To show this in action, we'll use a common layout strategy for simple Shiny apps, and put input controls in a narrow sidebar on the left. In the following code, we use the function ui.layout_sidebar() to separate the page into two panels. This function takes two arguments: a ui.panel_sidebar and a ui.panel_main, which each can contain whatever components you want to display on the left and right, respectively.

LIVE: Shiny editor

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

# Create some random data
t = np.linspace(0, 2 * np.pi, 1024)
data2d = np.sin(t)[:, np.newaxis] * np.cos(t)[np.newaxis, :]
```

```
app_ui = ui.page_fixed(
    ui.h2("Playing with colormaps"),
    ui.markdown("""
        This app is based on a [Matplotlib example][0] that displays 2D data
        with a user-adjustable colormap. We use a range slider to set the data
        range that is covered by the colormap.

        [0]: https://matplotlib.org/3.5.3/gallery/userdemo/colormap_interactive_adjustment.html
    """),
    ui.layout_sidebar(
        ui.panel_sidebar(
            ui.input_radio_buttons("cmap", "Colormap type",
                dict(viridis="Perceptual", gist_heat="Sequential", RdYlBu="Diverging")
            ),
            ui.input_slider("range", "Color range", -1, 1, value=(-1, 1), step=0.05),
        ),
        ui.panel_main(
            ui.output_plot("plot")
        )
    )
)

def server(input, output, session):
    @output
    @render.plot
    def plot():
        fig, ax = plt.subplots()
        im = ax.imshow(data2d, cmap=input.cmap(), vmin=input.range()[0],
vmax=input.range()[1])
        fig.colorbar(im, ax=ax)
        return fig


app = App(app_ui, server)
```

Notice how Shiny uses nested function calls to indicate parent/child relationships. In this example, ui.input_radio_buttons() is inside of ui.panel_sidebar(), and ui.panel_sidebar() is in ui.layout_sidebar(), and so on.

This example also includes some explanatory text written in a Markdown string literal, and uses the ui.markdown() function to render it to HTML.

Shiny UI is HTML

It’s worth noting at this point that Shiny UI is entirely made up of HTML. Each of the nested function calls in the previous section returns a snippet of HTML.

It’s not important that you know HTML to make good use of Shiny, but if you do know HTML, this fact may be helpful in forming a mental model for how Shiny UI works.

from shiny import ui

ui.input_numeric("n", "N", 0)
Adding a ui.panel_sidebar call around ui.input_numeric simply wraps the control with additional HTML (a and <form>, in this case). The nesting of the generated HTML matches the nesting of the Python function calls in your UI.

```
ui.panel_sidebar(
    ui.input_numeric("n", "N", 0)
)
```

Next Steps

In the following sections we’ll look more deeply into the UI, and then server logic. Then, we’ll put everything together using three example apps. While these three applications are useful on their own, we’ll look next at ways we could improve them using more advanced reactive programming concepts.

Finally, we’ll cover the best ways to run and debug Shiny applications.

2. Input controls

Each input control on a page is created by calling a Python function. All such functions take the same first two string arguments:

id: an identifier used to refer to input’s value in the server code. For example, id="x1" corresponds with input.x1() in the server function. id values must be unique across all input and output objects on a page, and should follow Python variable/function naming rules (lowercase with underscores, alphanumeric characters allowed, cannot start with a number).
label: a description for the input that will appear next to it. Can usually be None if no label is desired.
Note that many inputs take additional arguments. For example, an input_checkbox lets you indicate if it should start checked or not.

In the section below we’ll show the most common input objects. If you’re curious to learn more, see the API Reference on UI Inputs (for example, here’s the page for input_checkbox())

Input examples

Note
In these UI examples there's no server logic, so we're just using None instead of a server function.

Number inputs

ui.input_numeric creates a text box where a number (integer or real) can be entered, plus up/down buttons. This is most useful when you want the user to be able to enter an exact value.

ui.input_slider creates a slider. Compared to a numeric input, a slider makes it easier to scrub back and forth between values, so it may be more appropriate when the user does not have an exact value in mind to start with. You can also provide more restrictions on the possible values, as the min, max, and step size are all strictly enforced.

ui.input_slider can also be used to select a range of values. To do so, pass a tuple of two numbers as the value argument instead of a single number.

LIVE Shiny editor

```
from shiny import ui, App

app_ui = ui.page_fluid(
    ui.input_numeric("x1", "Number", value=10),
    ui.input_slider("x2", "Slider", value=10, min=0, max=20),
    ui.input_slider("x3", "Range slider", value=(6, 14), min=0, max=20)
)

app = App(app_ui, None)
```

Text inputs

Shiny includes three inputs for inputting string values.

Use ui.input_text for shorter, single-line values.

ui.input_text_area displays multiple lines, soft-wraps the text, and lets the user include line breaks, so is more appropriate for longer runs of text or multiple paragraphs.

ui.input_password is for passwords and other values that should not be displayed in the clear. (Note that Shiny does not apply any encryption to the password, so if your app involves passing sensitive information, make sure your deployed app uses https, not http, connections.)

LIVE: Shiny editor

```
from shiny import ui, App

app_ui = ui.page_fluid(
    ui.input_text("x1", "Text", placeholder="Enter text"),
    ui.input_text_area("x2", "Text area", placeholder="Enter text"),
    ui.input_password ("x3", "Password", placeholder="Enter password"),
)

app = App(app_ui, None)
```

Selection inputs

ui.input_selectize and ui.input_select are useful for letting the user select from a list of values. You can choose whether the user can select multiple values or not, using the multiple argument. The difference between the two functions is that ui.input_selectize uses the Selectize JavaScript library, while ui.input_select inserts a standard HTML <select> tag. For most apps, we recommend ui.input_selectize for its better all-around usability, especially when multiple=True.

LIVE Shiny editor

```
from shiny import ui, App
import re

# A list of Python's built-in functions
choices = list(filter(lambda x: re.match(r'[a-z].*', x), dir(__builtins__)))

app_ui = ui.page_fluid(
    ui.input_selectize("x1", "Selectize (single)", choices),
    ui.input_selectize("x2", "Selectize (multiple)", choices, multiple = True),
    ui.input_select("x3", "Select (single)", choices),
    ui.input_select("x4", "Select (multiple)", choices, multiple = True),
)

app = App(app_ui, None)
```

ui.input_radio_buttons and ui.input_checkbox_group are useful for cases where you want the choices to always be displayed. Radio buttons force the user to choose one and only one option, while checkbox groups allow zero, one, or multiple choices to be selected.

LIVE Shiny editor

```
from shiny import ui, App

choices = {"a": "Choice A", "b": "Choice B", "c": "Choice C"}

app_ui = ui.page_fluid(
    ui.input_radio_buttons("x1", "Radio buttons", choices),
    ui.input_checkbox_group("x2", "Checkbox group", choices),
)

app = App(app_ui, None)
```

Toggle inputs

Toggles allow the user to specify whether something is true/false (or on/off, enabled/disabled, included/excluded, etc.).

ui.input_checkbox shows a simple checkbox, while ui.input_switch shows a toggle switch. These are functionally identical, but by convention, checkboxes should be used when part of a form that has a Submit or Continue button, while toggle switches should be used when they take immediate effect.

LIVE: Shiny editor

```
from shiny import ui, App

app_ui = ui.page_fluid(
    ui.input_checkbox("x1", "Checkbox"),
    ui.input_switch("x2", "Switch")
)

app = App(app_ui, None)
```

Date inputs

ui.input_date lets the user specify a date, either interactively or by typing it in. ui.input_date_range is similar, but for cases where a start and end date are needed.

LIVE: Shiny editor

```
from shiny import ui, App

app_ui = ui.page_fluid(
    ui.input_date("x1", "Date input"),
    ui.input_date_range("x2", "Date range input"),
)

app = App(app_ui, None)
```

Action inputs

ui.input_action_button and ui.input_action_link let the user invoke specific actions on the server side. (See Handling events for an introduction to using buttons and links.)

Use ui.input_action_button for actions that feels effectual: recalculating, fetching new data, applying settings, etc. Add class_="btn-primary" to highlight important actions (like Submit or Continue), and class_="btn-danger" to highlight dangerous actions (like Delete).

Use ui.input_action_link for actions that feel more like navigation, like exposing a new UI panel, navigating through paginated results, or bringing up a modal dialog.

LIVE: Shiny editor

```
from shiny import ui, App

app_ui = ui.page_fluid(
    ui.p(ui.input_action_button("x1", "Action button")),
    ui.p(ui.input_action_button("x2", "Action button", class_="btn-primary")),
    ui.p(ui.input_action_link("x3", "Action link")),
)

app = App(app_ui, None)
```

3. Output Controls

Outputs create a spot on the webpage to put results from the server.

At a minimum, all UI outputs require an id argument, which corresponds to the server's output ID.

For example if you create this UI output:

ui.output_text("my_text")

Then you could connect it to the server output using the code below.

```
def server(input, output, session):
    @output
    @render.text
    def my_text():
        return "some text to show"
```

Notice that the name of the function my_text() matches the output ID; this is how Shiny knows how to connect each of the UI's outputs with its corresponding logic in the server.

Notice also the relationship between the names ui.output_text() and @render.text. Shiny outputs generally follow this pattern of ui.output_XXX() for the UI and @render.XXX to decorate the output logic.

Static plot output

Render static plots based on Matplotlib with ui.output_plot() and @render.plot. Plotting libraries built on Matplotlib, like seaborn and plotnine are also supported.

The function that @render.plot decorates typically returns a Matplotlib Figure or plotnine ggplot object, but @render.plot does support other less common return types, and also supports plots generated through side-effects. See the API reference for more details.

Interactive plots
Although ui.output_plot() holds a static plot, it is possible to respond to user input(s) like hovering, clicking, and/or dragging. See here for an example.

LIVE: Shiny editor

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

app_ui = ui.page_fluid(
    ui.output_plot("a_scatter_plot"),
)

def server(input, output, session):
    @output
```

```
    @render.plot
    def a_scatter_plot():
        return plt.scatter([1,2,3], [5, 2, 3])

app = App(app_ui, server)
```

Simple table output

Render various kinds of data frames into an HTML table with ui.output_table() and @render.table.

The function that @render.table decorates typically returns a pandas.DataFrame, but object(s) that can be coerced to a pandas.DataFrame via an obj.to_pandas() method are also supported (this includes Polars data frames and Arrow tables).

Styling tables
For more control over colors, alignment, borders, etc., your server logic may instead return a pandas.Styler object. See the ui.output_table for an example.

LIVE: Shiny editor

```
from pathlib import Path
import pandas as pd
from shiny import ui, render, App

df = pd.read_csv(Path(__file__).parent / "salmon.csv")

app_ui = ui.page_fluid(
    ui.output_table("salmon_species"),
)

def server(input, output, session):
    @output
    @render.table
    def salmon_species():
        return df

app = App(app_ui, server)


## file: salmon.csv
Common,Scientific
Atlantic,Salmo salar
Chinook,Oncorhynchus tshawytscha
Coho,Oncorhynchus kisutch
```

Sockeye,Oncorhynchus nerka

Interactive plots

As you'll learn more about in the section on using ipywidgets, Shiny supports interactive plotting libraries such as plotly, Altair, and more. Here's a basic example using plotly:

LIVE: Shiny editor

```
from shiny import ui, App
from shinywidgets import output_widget, render_widget
import plotly.express as px
import plotly.graph_objs as go

df = px.data.tips()

app_ui = ui.page_fluid(
    ui.div(
        ui.input_select(
            "x", label="Variable",
            choices=["total_bill", "tip", "size"]
        ),
        ui.input_select(
            "color", label="Color",
            choices=["smoker", "sex", "day", "time"]
        ),
        class_="d-flex gap-3"
    ),
    output_widget("my_widget")
)

def server(input, output, session):
    @output
    @render_widget
    def my_widget():
        fig = px.histogram(
            df, x=input.x(), color=input.color(),
            marginal="rug"
        )
        fig.layout.height = 275
        return fig

app = App(app_ui, server)
```

Interactive maps

As you'll learn more about in the section on using ipywidgets, Shiny supports interactive mapping libraries such as ipyleaflet, pydeck, and more. Here's a basic example using ipyleaflet:

LIVE: Shiny editor

```
from shiny import *
from shinywidgets import output_widget, render_widget
import ipyleaflet as L

# note: ignore the printed error "Unhandled error: can't start new thread" if you see one

basemaps = {
  "Satellite": L.basemaps.Gaode.Satellite,
  "OpenStreetMap": L.basemaps.OpenStreetMap.Mapnik
}

app_ui = ui.page_fluid(
    ui.input_select(
        "basemap", "Choose a basemap",
        choices=list(basemaps.keys())
    ),
    output_widget("map")
)

def server(input, output, session):
    @output
    @render_widget
    def map():
        basemap = basemaps[input.basemap()]
        map = L.Map(basemap=basemap, center=[57.2, -3.9], zoom=6)
        edinburgh = (55.953251, -3.188267)
        marker = L.Marker(location=edinburgh, draggable=False)
        map.add(marker)
        return map


app = App(app_ui, server)
```

Other interactive widgets

See the section on using ipywidgets to learn how to effectively use any ipywidgets package inside Shiny.

Text output

Use ui.output_text() / @render.text to render a block of text. Your server logic should return a Python string. You may not use HTML markup or Markdown; see the section on HTML and UI instead.

LIVE: Shiny editor

```
from shiny import ui, render, App

app_ui = ui.page_fluid(
    ui.output_text("my_cool_text")
)

def server(input, output, session):
    @output
    @render.text
    def my_cool_text():
        return "hello, world!"

app = App(app_ui, server)
```

Code output

ui.output_text_verbatim / @render.text (note: not @render.text_verbatim) is similar to text output, but renders text in a monospace font, and respects newlines and multiple spaces (unlike ui.output_text(), which collapses all whitespace into a single space, in accordance with HTML's normal whitespace rule).

```
from shiny import ui, render, App

app_ui = ui.page_fluid(
    ui.output_text_verbatim("a_code_block"),
    # The p-3 CSS class is used to add padding on all sides of the page
    class_="p-3"
)

def server(input, output, session):
    @output
```

```
    @render.text
    def a_code_block():
        # This function should return a string
        return ui.page_navbar.__doc__

app = App(app_ui, server)
```

LIVE: Shiny editor

HTML and UI output

ui.output_ui() / @render.ui are used to render HTML and UI from the server. These are the same exact objects you use in your app_ui UI already, and whenever possible, you should put HTML/UI directly into your app_ui. However, you'll need to use output_ui if you want to render HTML/UI dynamically–that is, if you want the HTML to change as inputs and other reactives change.

LIVE: Shiny editor

```
from shiny import ui, render, App

app_ui = ui.page_fluid(
    ui.input_text("message", "Message", value="Hello, world!"),
    ui.input_checkbox_group("styles", "Styles", choices=["Bold", "Italic"]),
    # The class_ argument is used to enlarge and center the text
    ui.output_ui("some_html", class_="display-3 text-center")
)

def server(input, output, session):
    @output
    @render.ui
    def some_html():
        x = input.message()
        if "Bold" in input.styles():
            x = ui.strong(x)
        if "Italic" in input.styles():
            x = ui.em(x)
        return x

app = App(app_ui, server)
```

When using @render.ui, your output function can return any of the following:

A plain string (which will be rendered as plain text, even if it contains HTML markup)
Any HTML tag object (like ui.tags.div())
Any Shiny UI object, including layouts, inputs, and outputs

You can also write raw HTML or Markdown.

A string wrapped in ui.HTML(), which will be treated as raw HTML markup
A string containing Markdown content, wrapped in ui.markdown(), which will be rendered into HTML
Caution
Be careful not to use ui.HTML or ui.markdown with strings that may be malicious (e.g. based on input from a malicious user, or from a database whose contents could have been influenced by a malicious user), as you could easily introduce a security vulnerability called Cross Site Scripting (XSS). Whenever possible, try to stick to Shiny's tag and UI functions, which are immune to such attacks.

Generate UI components based on data (eg. dropdown only with options which are available in data)

LIVE: Shiny editor

```
from shiny import ui, render, App
import pandas as pd

app_ui = ui.page_fluid(
    ui.output_ui("dropdown_drom_data"),
    ui.output_table("fruits_table"),
)

def server(input, output, session):
    # this data could come from file of url
    global fruits_df
    fruits_df = pd.DataFrame({"color":["red","purple","red"],
                              "name":["apple","grape","strawberry"]})

    @output
    @render.ui
    def dropdown_drom_data():
        global fruits_df
        # here get choices from yoru data, like df['fruit'].unique()
        color_choices = list(fruits_df["color"].unique())
        # for extra effect, here we can add "ALL" option
        color_choices = ["all"] + color_choices
        return ui.input_select("choose_fruit", "choose fruit",
                               choices=color_choices )
```

```
    @output
    @render.table
    def fruits_table():
        global fruits_df
        # here we will use whatever is in the ui.select which we created above
        if input.choose_fruit() == "all":
            return fruits_df
        else:
            return fruits_df[ fruits_df['color'] == input.choose_fruit() ]

app = App(app_ui, server)
```

4. Server Logic

Server logic
In Shiny, the server logic is defined within a function which takes three arguments: input, output, and session. It looks something like this:

```
def server(input, output, session):
    # Server code goes here
    ...
```

All of the server logic we'll discuss, such as using inputs and define outputs, happens within the server function.

Each time a user connects to the app, the server function executes once — it does not re-execute each time someone moves a slider or clicks on a checkbox. So how do updates happen? When the server function executes, it creates a number of reactive objects that persist as long as the user stays connected — in other words, as long is their session is active. These reactive objects are containers for functions that automatically re-execute in response to changes in inputs.

Accessing inputs

Input values are accessed via input.x(), where x is the name of the input. If you need to access an input by a name that is not known until runtime, you can do that with []:

```
input_name = "x"
input[input_name]()  # Equivalent to input["x"]() or input.x()
```

Shiny for Python compared to R

In Python, input.x() is equivalent to input$x in Shiny for R. Unlike in R, () is necessary to retrieve the value. This aligns the reading of inputs with the reading of reactive values and reactive expressions. It also makes it easier to pass inputs to module server functions.

You can't read input values at the top level of the server function. If you try to do that, you'll get an error that says RuntimeError: No current reactive context. The input values are reactive and, as the error suggests, are only accessible within reactive code. We'll learn more about that in the upcoming sections.

Defining outputs

To define the logic for an output, create a function with no parameters whose name matches a corresponding output ID in the UI. Then apply a render decorator and the @output decorator.

Normally, @output matches the function name to output name in the UI. There are times when this doesn't work, like if you have a variable with the same name, or if the output name is a reserved keyword in Python. In these cases, you can use @output(id="txt") and def _():

Here's server function that defines a single text output named txt. Inside of that output function, it reads the value of input.enable().

```
def server(input, output, session):
    @output
    @render.text
    def txt():
        if input.enable():
            return "Yes!"
        else:
            return "No!"
```

When you define an output function, Shiny makes it reactive, and so it can be used to access input values.

Now we can put together the UI we created earlier with the server function to create an App object. When we do that, the resulting object must be named app in order for the application to run.

LIVE: Shiny editor

```
from shiny import App, render, ui

app_ui = ui.page_fluid(
    ui.input_checkbox("enable", "Enable?"),
```

```
    ui.h3("Is it enabled?"),
    ui.output_text_verbatim("txt"),
)

def server(input, output, session):
    @output
    @render.text
    def txt():
        if input.enable():
            return "Yes!"
        else:
            return "No!"

app = App(app_ui, server)
```

5. Putting it together

Putting it together
The basics of the user interface (UI) inputs and outputs enables you to start building dynamic dashboards and applications. On this page, we'll walk through 2 examples.

example	covers
CSV parser	Enter text, see the resulting pandas DataFrame
Plot traits for dog breeds (e.g. drooliness)	Select dog breeds and traits to compare

Note
Since the examples load the pandas library, it may take some time to load the apps on this page.

Example 1: parsing and displaying CSV

In this example, we'll parse CSV data users paste in, and display it as a table.

Note that to do this, we need to use a little bit of code to allow the pandas to read a string of text:

```
import pandas as pd
from io import StringIO

csv_text = """
a,b,c
1,2,3
"""
```

pd.read_csv(StringIO(csv_text))
The key is that .read_csv() expects a file. StringIO() takes the text and makes it into something file-like that .read_csv() understands.

Below, we include this snippet in a Shiny application, to allow users to input text.

LIVE: Shiny editor

file: demo.csv

```
source,fruit,color
file_in_project,banana,yellow
file_in_project,kiwi,green
```

file: app.py

```
# in this file we will load csv in 4 ways:
# 0. from what user typed in
# 1. from a string
# 2. from a file included in shiny interface
# 3. from url - this one is hardest, since it includes
# a need for 'waiting' for the result (using async)

import pandas as pd
from io import StringIO
from pathlib import Path
import pyodide.http # this is needed for async url fetching


from shiny import App, render, ui

import pandas as pd
from io import StringIO

app_ui = ui.page_fluid(
    #  this is how you would 'load' a csv from a string, and pre-fill it
    ui.input_text_area("csv_text", "CSV Text - change it to see table change",
                       value="source,fruit,color\nuser_input,orange,amber\nuser_input,plum,purple"
                       ,rows=5
                      ),
    ui.panel_title("data from user input"),
    ui.output_table("parsed_data_from_user_input"),
    ui.panel_title("data from string"),
    ui.output_table("parsed_data_from_string"),
    ui.panel_title("data from attached file"),
    ui.output_table("parsed_data_from_file"),
    ui.panel_title("data from online file"),
```

```
    ui.output_table("parsed_data_from_url"),

)

def server(input, output, session):

    @output
    @render.table
    def parsed_data_from_user_input():
        file_text = StringIO(input.csv_text())
        data = pd.read_csv(file_text)
        return data

    @output
    @render.table
    def parsed_data_from_string():

        # we can 'pretend' to have a csv, eg for testing
        # notice tripple " quote, which is a 'block string' and allows enters
        csv_text = """
        source,fruit,color
        string,apple,green
        string,cherry,maroon
        """
        # if you wanted a non-block normal string you could say
        # csv_text = "source,fruit,color\nstring,apple,green\nstring,cherry,maroon"

        # StringIO pretends the text came from a file, so that pd knows what to do
        file_text = StringIO(csv_text)
        data = pd.read_csv(file_text)
        return data

    @output
    @render.table
    def parsed_data_from_file():
        # actul local file
        infile = Path(__file__).parent / "demo.csv"
        data = pd.read_csv(infile)
        return data

    @output
    @render.table
    async  def parsed_data_from_url():
        print("starting")
        # online file
        file_url =
"https://raw.githubusercontent.com/drpawelo/data/main/random/fruits_source.csv"
        response = await pyodide.http.pyfetch(file_url)
```

```
    data = await response.string()
    loaded_df = pd.read_csv(StringIO(data))
    print(loaded_df)
    # notice await - it means that the function which follows
    # is 'alowed' to take time (async) and we are fine with (a)waiting for it
    print("done")
    return loaded_df


# This is a shiny.App object. It must be named `app`.
app = App(app_ui, server)
```

Example 2: Visualizing dog traits

In this example, we'll visualize dog traits by breed. For example, the drooling level of German Shepherds.

Viewers will be able to select the following:

which dog breeds to see traits for.
which traits to plot ratings for.
Here is a preview of the data:

breed trait rating
Retrievers (Labrador) Affectionate With Family
5
German Shepherd Dogs Affectionate With Family
5
Bulldogs Affectionate With Family
4
Retrievers (Labrador) Good With Young Children
5
German Shepherd Dogs Good With Young Children
5
Bulldogs Good With Young Children
3
We'll take inputs for breed and trait, and use them to subset the data.

LIVE: [Shiny editor]

```
## file: app.py

from shiny import App, render, ui
import pandas as pd
import seaborn as sns

from pathlib import Path

sns.set_theme()

long_breeds = pd.read_csv(Path(__file__).parent / "dog_traits_long.csv")

options_traits = long_breeds["trait"].unique().tolist()
options_breeds = long_breeds["breed"].unique().tolist()

app_ui = ui.page_fluid(
   ui.input_selectize("traits", "Traits", options_traits, multiple=True, selected=["Drooling
Level", "Barking Level"]),
   ui.input_selectize("breeds", "Breeds", options_breeds, multiple=True,
selected=["Poodles","Bulldogs"]),
   ui.output_plot("barchart")
)


def server(input, output, session):
   @output
   @render.plot
   def barchart():
      # note that input.traits() refers to the traits selected via the UI
      indx_trait = long_breeds["trait"].isin(input.traits())
      indx_breed = long_breeds["breed"].isin(input.breeds())
      # subset data to keep only selected traits and breeds
      sub_df = long_breeds[indx_trait & indx_breed]
      print(sub_df)

      sub_df["dummy"] = 1

      # plot data. we use the same dummy value for x, and use hue to set
      # the bars next to eachother
      g = sns.catplot(
         data=sub_df, kind="bar",
         y="rating", x="dummy", hue="trait",
         col="breed", col_wrap=3,
      )

      # remove labels on x-axis, which is on the legend anyway
      g.set_xlabels("")
      g.set_xticklabels("")
```

```
        g.set_titles(col_template="{col_name}")

        return g


app = App(app_ui, server)
```

```
## file: dog_traits_long.csv
breed,trait,rating
Retrievers (Labrador),Affectionate With Family,5
French Bulldogs,Affectionate With Family,5
German Shepherd Dogs,Affectionate With Family,5
Retrievers (Golden),Affectionate With Family,5
Bulldogs,Affectionate With Family,4
Poodles,Affectionate With Family,5
Beagles,Affectionate With Family,3
Rottweilers,Affectionate With Family,5
Retrievers (Labrador),Good With Young Children,5
French Bulldogs,Good With Young Children,5
German Shepherd Dogs,Good With Young Children,5
Retrievers (Golden),Good With Young Children,5
Bulldogs,Good With Young Children,3
Poodles,Good With Young Children,5
Beagles,Good With Young Children,5
Rottweilers,Good With Young Children,3
Retrievers (Labrador),Good With Other Dogs,5
French Bulldogs,Good With Other Dogs,4
German Shepherd Dogs,Good With Other Dogs,3
Retrievers (Golden),Good With Other Dogs,5
Bulldogs,Good With Other Dogs,3
Poodles,Good With Other Dogs,3
Beagles,Good With Other Dogs,5
Rottweilers,Good With Other Dogs,3
Retrievers (Labrador),Shedding Level,4
French Bulldogs,Shedding Level,3
German Shepherd Dogs,Shedding Level,4
Retrievers (Golden),Shedding Level,4
Bulldogs,Shedding Level,3
Poodles,Shedding Level,1
Beagles,Shedding Level,3
Rottweilers,Shedding Level,3
Retrievers (Labrador),Coat Grooming Frequency,2
French Bulldogs,Coat Grooming Frequency,1
German Shepherd Dogs,Coat Grooming Frequency,2
Retrievers (Golden),Coat Grooming Frequency,2
Bulldogs,Coat Grooming Frequency,3
```

Poodles,Coat Grooming Frequency,4
Beagles,Coat Grooming Frequency,2
Rottweilers,Coat Grooming Frequency,1
Retrievers (Labrador),Drooling Level,2
French Bulldogs,Drooling Level,3
German Shepherd Dogs,Drooling Level,2
Retrievers (Golden),Drooling Level,2
Bulldogs,Drooling Level,3
Poodles,Drooling Level,1
Beagles,Drooling Level,1
Rottweilers,Drooling Level,3
Retrievers (Labrador),Openness To Strangers,5
French Bulldogs,Openness To Strangers,5
German Shepherd Dogs,Openness To Strangers,3
Retrievers (Golden),Openness To Strangers,5
Bulldogs,Openness To Strangers,4
Poodles,Openness To Strangers,5
Beagles,Openness To Strangers,3
Rottweilers,Openness To Strangers,3
Retrievers (Labrador),Playfulness Level,5
French Bulldogs,Playfulness Level,5
German Shepherd Dogs,Playfulness Level,4
Retrievers (Golden),Playfulness Level,4
Bulldogs,Playfulness Level,4
Poodles,Playfulness Level,5
Beagles,Playfulness Level,4
Rottweilers,Playfulness Level,4
Retrievers (Labrador),Watchdog/Protective Nature,3
French Bulldogs,Watchdog/Protective Nature,3
German Shepherd Dogs,Watchdog/Protective Nature,5
Retrievers (Golden),Watchdog/Protective Nature,3
Bulldogs,Watchdog/Protective Nature,3
Poodles,Watchdog/Protective Nature,5
Beagles,Watchdog/Protective Nature,2
Rottweilers,Watchdog/Protective Nature,5
Retrievers (Labrador),Adaptability Level,5
French Bulldogs,Adaptability Level,5
German Shepherd Dogs,Adaptability Level,5
Retrievers (Golden),Adaptability Level,5
Bulldogs,Adaptability Level,3
Poodles,Adaptability Level,4
Beagles,Adaptability Level,4
Rottweilers,Adaptability Level,4
Retrievers (Labrador),Trainability Level,5
French Bulldogs,Trainability Level,4
German Shepherd Dogs,Trainability Level,5
Retrievers (Golden),Trainability Level,5
Bulldogs,Trainability Level,4

Poodles,Trainability Level,5
Beagles,Trainability Level,3
Rottweilers,Trainability Level,5
Retrievers (Labrador),Energy Level,5
French Bulldogs,Energy Level,3
German Shepherd Dogs,Energy Level,5
Retrievers (Golden),Energy Level,3
Bulldogs,Energy Level,3
Poodles,Energy Level,4
Beagles,Energy Level,4
Rottweilers,Energy Level,3
Retrievers (Labrador),Barking Level,3
French Bulldogs,Barking Level,1
German Shepherd Dogs,Barking Level,3
Retrievers (Golden),Barking Level,1
Bulldogs,Barking Level,2
Poodles,Barking Level,4
Beagles,Barking Level,4
Rottweilers,Barking Level,1
Retrievers (Labrador),Mental Stimulation Needs,4
French Bulldogs,Mental Stimulation Needs,3
German Shepherd Dogs,Mental Stimulation Needs,5
Retrievers (Golden),Mental Stimulation Needs,4
Bulldogs,Mental Stimulation Needs,3
Poodles,Mental Stimulation Needs,5
Beagles,Mental Stimulation Needs,4
Rottweilers,Mental Stimulation Needs,5

Note that:

We use input_selectize() with multiple=True to let users select multiple options from a dropdown.
We figure out the options for each input before hand, using the pandas .unique().tolist() methods.
Next up
On this page, we've looked at two examples of simple applications that react to changes in inputs. This just scratches the surface of what Shiny can do. In the next section, we'll go through more advanced tools for reactive programming in Shiny.

from:
https://web.archive.org/web/20230607090914/https://shiny.rstudio.com/py/docs/overview.html