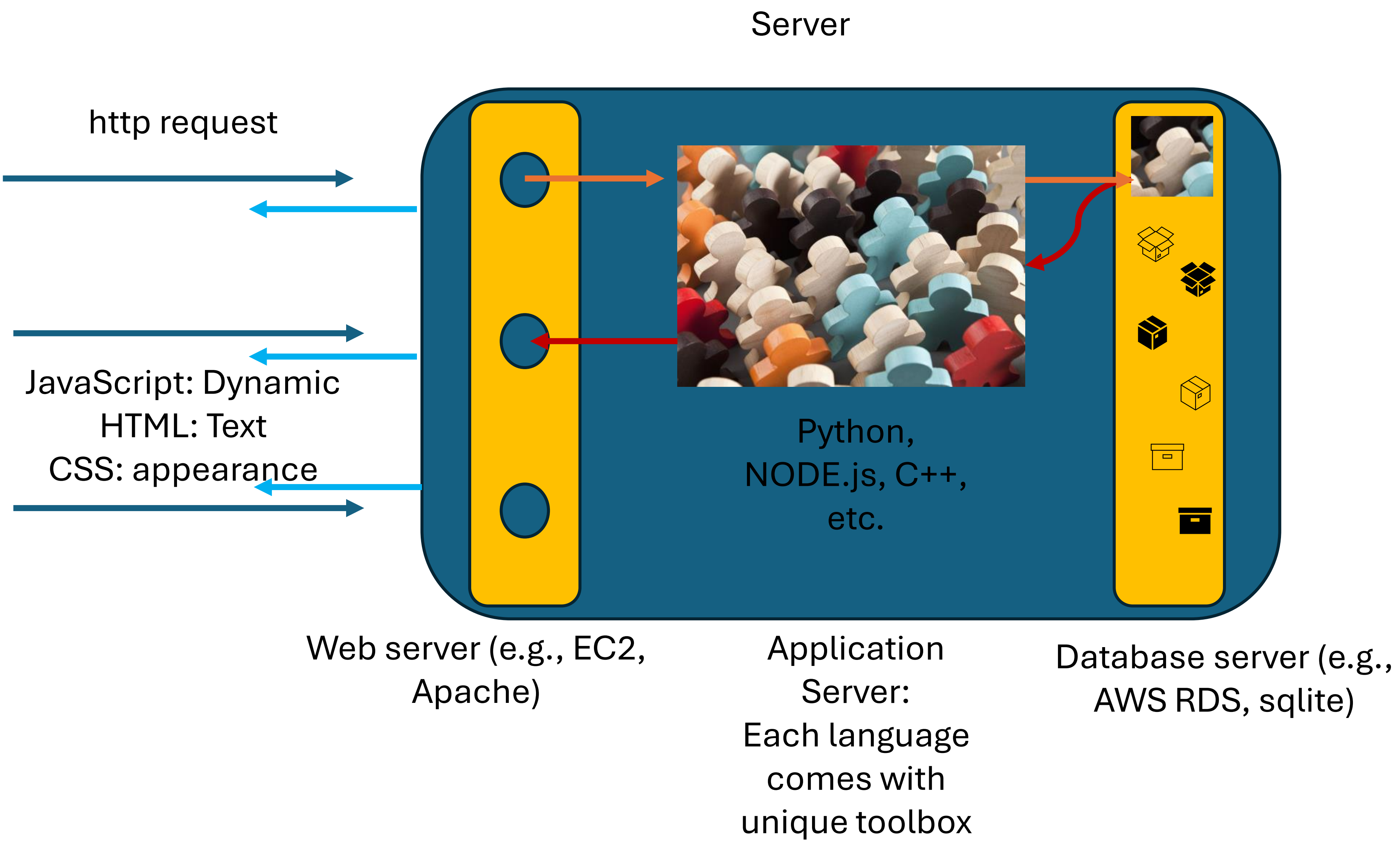
Minimal tutorial for contributing to web-app

See: https://github.com/Ben-Choat/PrecipFreq_WebApp

Prompt: “web dev hacker”

1. System Architecture for Web Application (free)

https://www.youtube.com/watch?v=Z3AjqMkTq38


Server
http request
JavaScript: Dynamic
HTML: Text
CSS: appearance
Python,
NODE.js, C++,
etc.
Web server (e.g., EC2, Apache)
Application Server: Each language comes with unique toolbox
Database server (e.g., AWS RDS, sqlite)

Web Server:
- Receives and sends http requests from/to clients (e.g., web browsers)

Application Server:
- Business logic (e.g., computation)

Database Server:
- Data storage and retrieval
- CRUD (create, read, update, delete)

So many tools and frameworks!

plotly

Panel

plotly | Dash

Shiny for python™

Streamlit

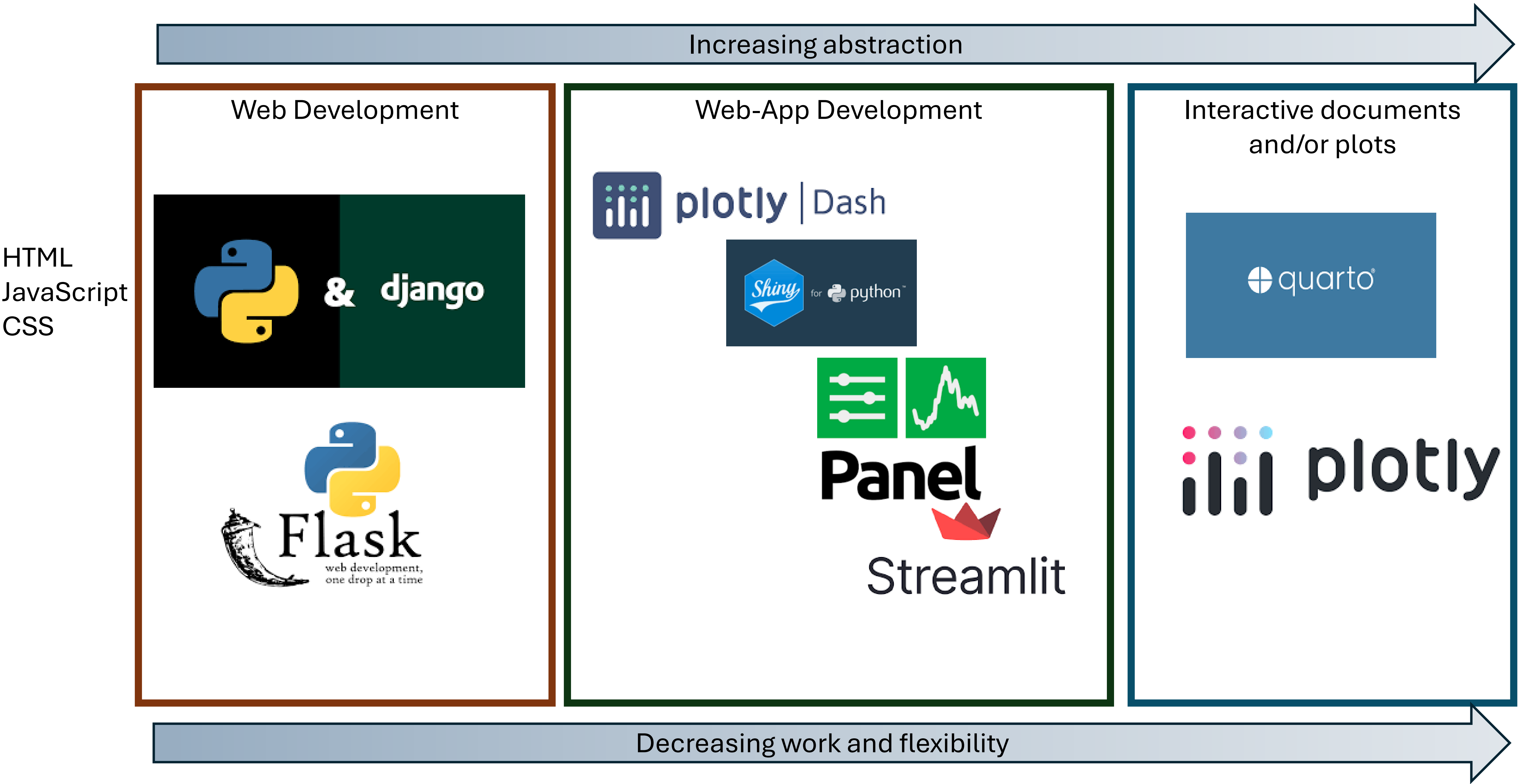
& django

Flask
web development,
one drop at a time

quarto®

Etc.....

Ordered by abstraction



For Oregon website – I think Flask + Dash is the way to go

Increasing abstraction

HTML
JavaScript
CSS

Web Development

- Most flexible
- Most upfront decisions
- Steepest learning curve
- Requires server for deploying

- Flask is lighter weight
- Easier for "simpler websites"
- Less "out-of-the-box"
- More flexible (good and bad)

Web-App Development

- Range of flexibility
- Many frameworks
- Typically no free authentication
- Requires server for deploying

- Plotly Dash is built on Flask
- Easy yet flexible
- Good for smaller to larger applications

Interactive documents and/or plots

- Least flexible
- Specific applications
- Render stand-alone HTMLs
- Minimal interactivity
- No server-side computation
- No server-side data integration
- Easily sharable

Decreasing work and flexibility

Motivation

- Staying as Python-based as possible/reasonable
- Staying flexible (not only for current application, but any future application)
- Can still use JavaScript as needed for enhanced flexibility

Full Stack

- Plotly
  - wraps plotly.js for interactive plots
- Dash
  - uses 'Callbacks' enabling JavaScript-based interactivity
  - Dash-Leaflet enables interactive mapping
- Flask
  - Full web framework with useful tools

Full Stack

Basic intro to Plotly, Dash, and Flask (minimum needed)

Files/Folders

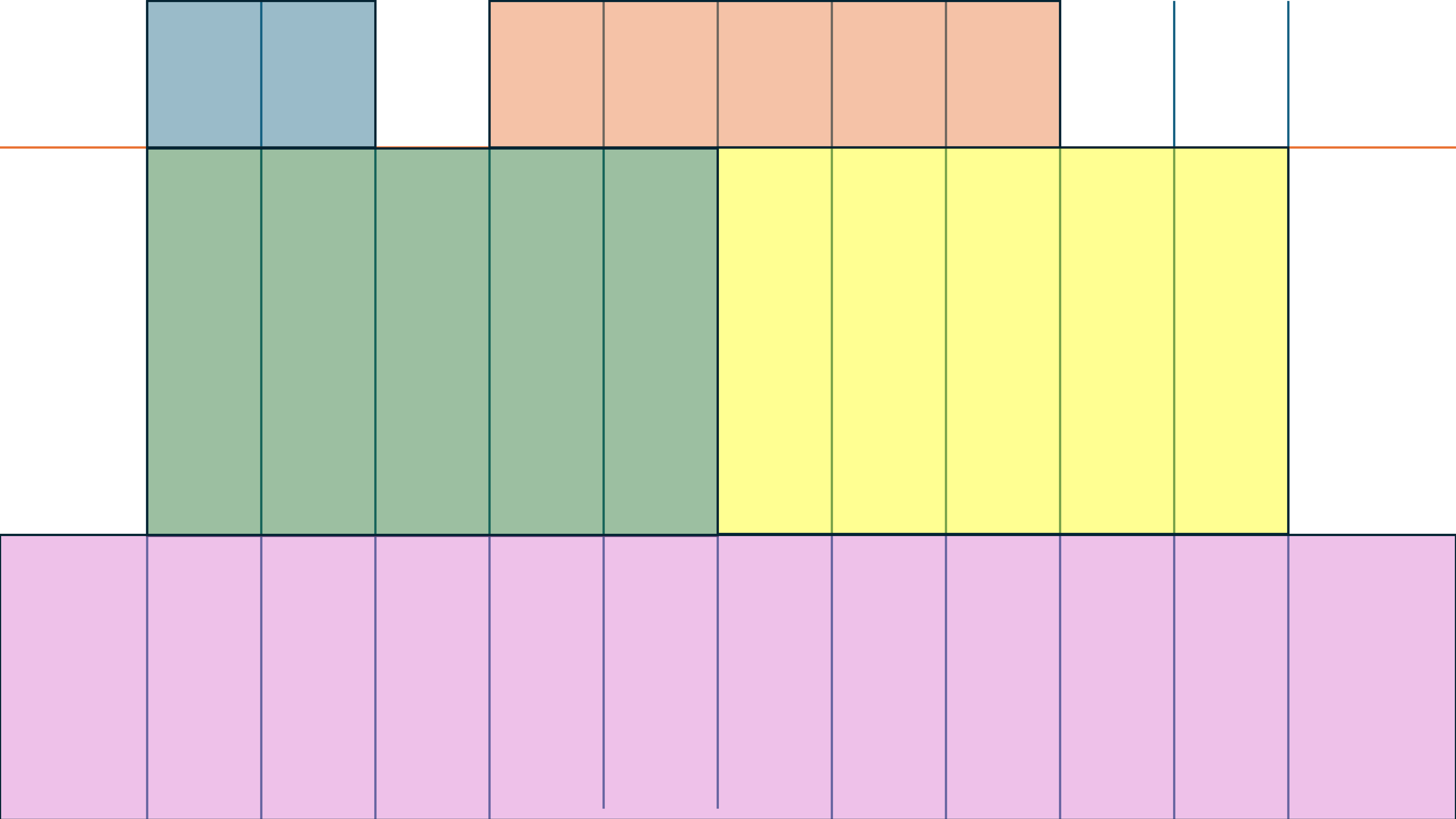
1. A_pure_html.html
2. B_css_html.html
3. C_PlotlyPlots.py
4. D_DashBasic.py
5. E_DashInteractive.py
6. F_DashBootstrapComp.py
7. G_FlaskBasic.py
8. H_FlaskApp1
9. I_FlaskApp_base
10. ~~J_FlaskApp_bp~~

A_pure_html.html

1. General Intro to layout and components
2. Inspect in Browser

B_css_html.html

1. CSS intro
2. Flex-box
3. Grid (12 columns w/defined rows)



C_PlotlyPlots.py

1. Plotly.express (think Seaborn)
2. Plotly.graph_objects (think matplotlib)

3. For a graph gallery, see:
    1. https://plotly.com/python/plotly-express/
    2. Graph objects in Python (plotly.com)
4. Modify hovertext and template
5. Html in pop-ups
6. Modify figure object after created

D_DashBasic.py

1. Layout
   1. App definition
   2. Layout
   3. (no interactivity yet)
2. Dash-core-components
3. Dash-html-components
4. Dash-bootstrap-components
5. See file for useful links

6. Run from terminal
7. Debug=True
8. Styling
   1. Flex-column vs flex-row

E_DashInteractive.py

1. Basic Layout
    1. App definition
    2. Layout
    3. ADD INTERACTIVITY:
    4. Callback(s)
    5. Function tied to each callback
    6. Command to execute
2. JS and events (and callbacks)
    1. See precip map lat-long as example (look at browser console)
    2. ‘id’
3. Debug troubleshooting in dash (icon on app)
4. Show div for width control, html.P’s for labels

F_DashBootstrapComp.py

1. Bootstrap
    1. Enables ‘easier’ control for variety of screen sizes (e.g., can more easily make mobile friendly – we are not currently using this fully)
2. Container->Row ->Col
3. Can easily change themes (but we will rely on local .css files for the most part
4. Using print for troubleshooting (if hasn’t come up yet)

G_FlaskBasic.py

1. Multiple ways to define and run a flask app
2. Most basic shown here
3. Return text or html

H_FlaskApp1

src (e.g., H_FlaskApp1)

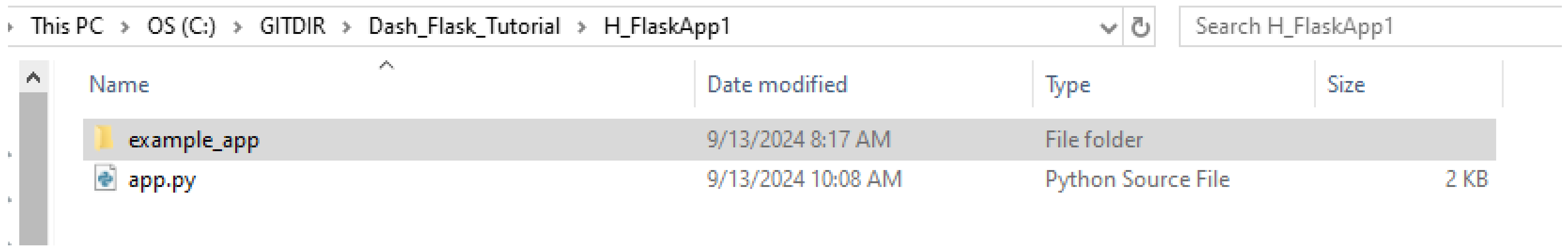
- app.py (or however you name it, I think I used application.py for OWRD app)
  This is the file you are currently looking at.
  It calls the app factory to run the app.

app_home_folder (e.g., example_app)

- __init__.py (where app factory lives)
- static (holds resources such as .png, .csv, etc.)
  Flask expects 'static', we redefine it to be 'assets’ because that is what Dash uses.
- Templates
  Holds .html files (so page templates)
- .py files
  any files where we use python for computation, or whatever else
  These could live in another folder, but I've been keeping them here.

1. User guide: https://flask.palletsprojects.com/en/3.0.x/#user-s-guide
2. intro tutorial: https://flask.palletsprojects.com/en/3.0.x/tutorial/
3. setting up application factory: https://flask.palletsprojects.com/en/3.0.x/tutorial/factory/

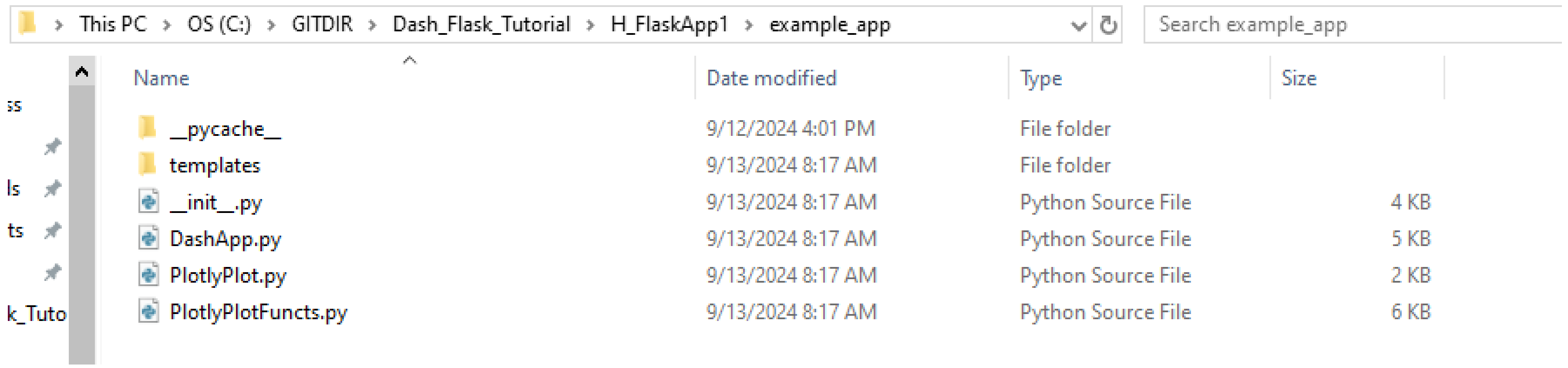
H_FlaskApp1



This PC › OS (C:) › GITDIR › Dash_Flask_Tutorial › H_FlaskApp1

Search H_FlaskApp1

Name	Date modified	Type	Size
example_app	9/13/2024 8:17 AM	File folder	
app.py	9/13/2024 10:08 AM	Python Source File	2 KB



This PC › OS (C:) › GITDIR › Dash_Flask_Tutorial › H_FlaskApp1 › example_app

Search example_app

Name	Date modified	Type	Size
__pycache__	9/12/2024 4:01 PM	File folder	
templates	9/13/2024 8:17 AM	File folder	
__init__.py	9/13/2024 8:17 AM	Python Source File	4 KB
DashApp.py	9/13/2024 8:17 AM	Python Source File	5 KB
PlotlyPlot.py	9/13/2024 8:17 AM	Python Source File	2 KB
PlotlyPlotFuncts.py	9/13/2024 8:17 AM	Python Source File	6 KB

H_FlaskApp1 (app.y, __init__.py)

1. We make the app a python package by including __init__.py in app_name folder (e.g., example_app)
2. In app.py (can have other names) we simply import and run the app
3. __init__.py
   1. Define App
   2. Define static_folder to match Dash expectations
   3. Secret key: signs cookies (and some other stuff) to help secure site
   4. Can define configuration info in a config.py file (e.g., environmental variables
   5. Define routes (i.e., app.route; assuming app is named ‘app’)
      1. Return html directly, redirect, render directly, render template
   6. Create dash app

H_FlaskApp1 (PloltyPlot.py)

1. Passing info between html and python code
   A. request
      a) get: retrieve data
      b) put: updating data
      c) post: create new data
2. Use PlotlyPlot.py, template/plot.html, and app.route('/plot') in __init__.py to see this in action
3. PlotlyPlot.py
   1. Render_template and pass variables

H_FlaskApp1 (plot.html)

1. plot.html
   A. Html layout (see flex-column)
   B. Form (post) + Select + option
   C. Jinja2 {% code %} ; {{ variable }}
   D. Use of JS to update when value changes (onchange=…)
   E. Displaying plot, using | safe
      A. When passing html, use ‘safe’ so the raw html is rendered
      B. Otherwise jinja2 escapes html for security reasons
   F. hold

H_FlaskApp1 (DashApp.py, dashView.html, __init__.py)

1. Dash App:
    1. Create the dash app (as seen before)
    2. To allow friendly cooperation with Flask, we define the app as:

```
app = Dash(__name__, server=server, url_base_pathname='/dashApp/’, …)
```

1. Wrapping in a function (server) (we will pas server to it in __init__.py)
2. Using two rows and columns to control
3. dashView.html to show iframe

I_FlaskApp_base

Very much the same as H_FlaskApp1, but using base to create more fluid experience

- Base.html creates a base html layout that other html templates build on.
    - This is used to create some attributes that are consistent (e.g., navbar) across pages.
    - Can use, e.g.,

```
<section>
  {% block content %}{% endblock %}
</section>
```

  - In base.html. Then in otherFile.html, place new html content within {% block_content %} <h1>other html content</h1> {% endblock %} and the other html content will show up within the <section> section of base.html
- Using dashView.html and plot.html as examples

J_FlaskApp_bp

Doesn’t yet exist.
Only difference between this and I_FlaskApp_base, is the use of ‘blueprints’ in place of app.routes().

Either approach is valid, but blueprints are considered ‘best practice’ by flask.

They enable more modularity and better organization (especially for larger apps)

For example, this allows you to specify different ‘static_folders’ and ‘external_stylesheets’ for each blueprint

Currently, the precip web app uses blueprints