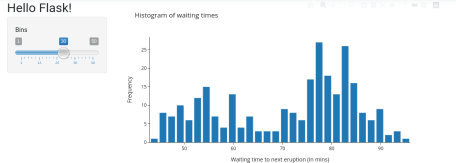
**Spoiler**

Shiny and flask are different tools with different strengths. If you have a simple shiny application, shiny is by far the quicker tool to get the job done and requires less knowledge about web based technologies. However, flask gives us much greater flexibility than could easily be achieved in shiny.

I’m hoping that this will turn into a series of blog posts about flask for data science applications.

With that in mind, lets treat this as an exercise in creating a simple flask application for visualisation of some data with a little interactivity.



**What is Flask?**

Flask is a micro web framework written in Python with a wealth of extensions for things like authentication, form validation, ORMs. It provides the tools to build web based applications. Whilst it is very light, combined with the extensions it is very powerful. Meaning that your web application might be a simple API to a machine learning model that you have created, or a full web application. Pinterest and LinkedIn use flask for example.

**Set up your app**

Create a directory structure in which to house your project, we can quickly create a project root and other necessary directories. We will explain the utility of each of the directories shortly.

mkdir -p app\_example/{static,templates,data} && cd app\_example

I highly recommend for all Python projects to set up a virtual environment. There are a number of tools for managing virtual environments in Python, but I tend to use virtualenv. We can create a new virtual environment for this project in the current directory and activate it.

In 2020 it should also go without saying that we are using Python 3 for this activity, specifically 3.7.3.

virtualenv venv

source venv/bin/activate

**Other directories**

**data:**

For our project, we will also want somewhere to house the data. Since we are talking about a very small tabular dataset of 2 variables and 272 cases, any sort of database would be overkill. So we will just read from a csv on disk. We can create this data set via

Rscript -e "readr::write\_csv(faithful, 'data/faithful.csv')"

**templates:**

The visual elements of the flask application will be web pages rendered from html templates. The name templates is chosen specifically here as it is the default directory that your flask app will look for when trying to render web pages.

**static:**

This will be our place to store any static assets, like CSS style sheets and JavaScript code.

**Packages**

For this project we will need some python packages

* **flask** (obviously)
* **pandas** – useful for data manipulation, but in this case just  
  used to read data from disk
* **numpy** – we will use for calculating the histogram bins and  
  frequencies
* **plotly** – my preferred graphics library in Python at the moment  
  and well suited to web based applications

pip install flask pandas plotly numpy

**Choosing your editor**

My editor of choice for anything that is not R related is VScode, which I find particularly suitable for applications that are created using a mixture of different languages. There are lots of plugins for Python, HTML, CSS and JavaScript for the purposes of code completion, snippets, linting and terminal execution which means I can write, test and run all the parts of my application from the comfort of one place.

**Hello Flask**

With everything set up we can start upon our Flask application. One of the things that I really like about flask is the simple syntax for adding URL endpoints to our site. We can create a “hello world” style example with the following python code (saved in this case in app.py)

# required imports

from flask import Flask

# instantiate the application object

app = Flask(\_\_name\_\_)

# create an endpoint with a decorator

@app.route("/")

def hello():

return "Hello World"

if \_\_name\_\_ == "\_\_main\_\_":

app.run()

Back in the terminal we could run this app with

python app.py

and view at the default URL of localhost:5000. I think the interesting part of the above code snippet is the route decorator

@app.route("/)

Routes refer to the URL patterns of our web application. The "/" is effectively the root route. i.e what you would see at a web address like “[mycoolpage.com](http://mycoolpage.com)”. The decorator here allows us to specify a Python function that should run when a user navigates to a particular URL within our domain name (or a handler).

**What our app needs**

We are creating an application here that allows users to choose an input via a slider, which causes a histogram to redraw. For this, our app will need two routes

* A route for generating the histograms
* A html page that the user will see

**Creating a histogram**

We could write a function which will draw a histogram using **plotly** fairly easily.

# imports

from pandas import read\_csv

import plotly.express as px

# read data from disk

faithful = read\_csv('./data/faithful.csv')

def hist(bins = 30):

# calculate the bins

x = faithful['waiting']

counts, bins = np.histogram(x, bins=np.linspace(np.min(x), np.max(x), bins+1))

bins = 0.5\* (bins[:-1] + bins[1:])

p = px.bar(

x=bins, y=counts,

title='Histogram of waiting times',

labels={

'x': 'Waiting time to next eruption (in mins)',

'y': 'Frequency'

},

template='simple\_white'

)

return p

This is the sort of thing we might create outside of the web application context for visualising this data. If you want to see the plot you might do something like

plot = hist()

plot.show()

However we want to make some modifications for use in our web application.

1. We want to turn our work into a flask application. We can start by adding the required imports and structure to our app.py with the hist function in it.
2. from flask import Flask
3. # other imports
4. # instantiate app
5. app = Flask(\_\_name\_\_)
6. ...
7. # At the end of our script
8. if \_\_name\_\_ == '\_\_main\_\_':

app.run()

1. We want to take the number of bins from a request to our webserver. We could achieve this by, instead of taking the number of bins from the argument to our function, taking it from the argument in the request from the client. When a Flask application handles a request object, it creates a Request object which can be accessed via the request proxy. Arguments can then be obtained from this context
2. from flask import request
3. def hist():
4. bins = int(request.args['bins'])

...

1. We want the function to be available at a route. The request context only really makes sense within a request from a client. Since the client is going to ask our application for the histogram to be updated dependent on their input we decorate our function with a route decorator
2. @app.route('/graph')
3. def hist():

...

1. Return JSON to send to the client. Instead of returning a figure object, we return some JSON that we can process with JavaScript on the client side
2. import json
3. from plotly.utils import PlotlyJSONEncoder
4. @app.route('/graph')
5. def hist():
6. ...

return json.dumps(p, cls=PlotlyJSONEncoder)

If you were to rerun your Flask server now and navigate your browser to localhost:5000/graph?bins=30 you would see the fruit of your labour. Although not a very tasty fruit at the moment, as all you will see is all of the JSON output for your graph. So let’s put the user interface together.

**Creating the user interface**

We will want to grab a few front end dependencies. For brevity they are included here by linking to the CDN. The shiny app we are mimicking uses bootstrap for it’s styling, which we will use too. Similarly the sliderInput() function in **shiny** uses the **ion-rangslider** JS package, so we will too. We will take the Plotly js library (for which the plotly python package is a wrapper). We will not need to know how to create these plots in JavaScript, but will use it to take the plot returned from our flask server and render it client side in the browser.

The head of our HTML file in templates/index.html then looks like

The {{}} notation at the bottom here is jinja syntax. Flask makes use of jinja templating for creating web pages that your users will consume. url\_for is a function that is automatically available when you render a template using flask, used to generate URLs to views instead of having to write them out manually. [Jinja templating](https://jinja.palletsprojects.com/) is a really neat way to blend raw markup with your python variables and functions and some logic statements like for loops. We haven’t written any style yet, but we will create the file ready for later, we will also create somewhere to contain our JavaScript

mkdir -p static/{css,js} && touch static/css/app.css static/js/hist.js

With all of dependencies in place it is relatively easy to create a simple layout. We have a 1/3 to 2/3 layout of two columns for controls and main content respectively, somewhere to contain our input elements and an empty container for our histogram to begin with. The

of our index.html then is

Hello Flask!

Top of Form

Bins



Bottom of Form

We will go back to our app.py and add the route for this view

@app.route('/')

def home():

return render\_template('index.html')

Our full app.py is then

# app.py

from flask import Flask, render\_template, request

from pandas import read\_csv

import plotly.express as px

from plotly.utils import PlotlyJSONEncoder

import json

import numpy as np

faithful = read\_csv('./data/faithful.csv')

app = Flask(\_\_name\_\_)

@app.route('/graph', methods=['GET'])

def hist():

# calculate the bins

x = faithful['waiting']

counts, bins = np.histogram(x, bins=np.linspace(np.min(x), np.max(x), int(request.args['bins'])+1))

bins = 0.5\* (bins[:-1] + bins[1:])

p = px.bar(

x=bins, y=counts,

title='Histogram of waiting times',

labels={

'x': 'Waiting time to next eruption (in mins)',

'y': 'Frequency'

},

template='simple\_white'

)

return json.dumps(p, cls=PlotlyJSONEncoder)

@app.route('/')

def home():

return render\_template('index.html')

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

Running the server and viewing our work still won’t look very impressive, but we are almost there. At the end of our page we are including a JavaScript file, this is to initialise our ion range slider and use it to send the chosen value from client to server to ask for the updated plot.

We can use AJAX (Asynchronous JavaScript and XML) to send the data from the slider to our /graph URL route, and on response, draw a new plotly plot into the div element with the histogram id. We want this function to run when we first load the page and every time a user moves the slider

// hist.js

const updatePlot = (data) => {

$.ajax({

url: 'graph',

type: 'GET',

contentType: 'application/json;charset=UTF-8',

data: {

'bins': data.from

},

dataType: 'json',

success: function(data){

Plotly.newPlot('histogram', data)

}

});

}

$('.js-range-slider').ionRangeSlider({

type: 'single',

skin: 'big',

min: 1,

max: 50,

step: 1,

from: 30,

grid: true,

onStart: updatePlot,

onFinish: updatePlot

});

Now we are getting somewhere. Run your app and navigate to localhost:5000 to see the control and the output plot. As you drag the slider, the plot will redraw.

To finish up we will add a little styling, just to get us closer to our shiny example target. In our app.css file under static/css we add the styling around the input controls and make the title stand out a little more.

.title {

font-size: 2rem;

}

.well {

background-color: #f5f5f5;

padding: 20px;

border: 1px solid #e3e3e3;

border-radius: 4px;

}

Rerun our application with

python app.py

And voila, at localhost:5000 we have something that fairly closely matches our target. I really like flask as a tool for creating web applications and APIs to data and models. There is an awful lot of power and flexibility available in what can be created using the toolset explored here.