Table of Contents

**[Lesson 1: Web Development 1](#_Toc32581)**

[Introduction to Web Development 1](#_Toc32267)

[Lesson Overview 2](#_Toc31789)

[The World Wide Web 3](#_Toc4446)

[Components of a Web App 3](#_Toc26165)

[Introduction to HTML 3](#_Toc8413)

[Div and Span 4](#_Toc8711)

[IDs and Classes 4](#_Toc13589)

[CSS 5](#_Toc31355)

[Bootstrap Library 9](#_Toc10249)

[Javascript 9](#_Toc31179)

[Plotly 11](#_Toc14897)

[The Backend 12](#_Toc18735)

[Getting Started with Flask 13](#_Toc6165)

[Flask + Pandas 15](#_Toc12918)

[Flask with Plotly and Pandas Part 1 16](#_Toc7687)

[Flask with Plotly and Pandas Part 2 17](#_Toc31585)

[Flask with Plotly and Pandas Part 3 18](#_Toc21148)

[Flask with Plotly and Pandas Part 4 19](#_Toc20674)

[Deployment 20](#_Toc5566)

[Lesson Summary 23](#_Toc26544)

**[Lesson 2: Portfolio Exercise: Deploy a Data Dashboard 24](#_Toc12696)**

[Introduction 24](#_Toc27989)

[Troubleshooting Possible Errors 26](#_Toc9136)

[Congratulations 26](#_Toc25112)

[APIs [advanced version] 27](#_Toc26593)

[The World Bank API [advanced version] 28](#_Toc9016)

[World Bank Data Dashboard [advanced version] 29](#_Toc7187)

Web Development with Flask

# Lesson 1: Web Development

## Introduction to Web Development

<https://www.youtube.com/watch?v=-PGMIIXFCgg&feature=emb_logo>

### Why should a data scientist learn web development?

In this course, you are going to use Flask to build a data dashboard. You might be thinking that you already have good tools for visualizing data such as matplotlib, seaborn, or Tableau.

However, the web development skills you'll learn in this lesson will prepare you for building other types of data science applications. Data scientists are increasingly being asked to deploy their work as an application in the cloud.

For example, consider a project where you build a model that classifies disaster relief messages into categories. With your web development skills, you could turn that model into a web app where you would input a message and display the resulting message category.

As another example, consider a system that recommends movies based on a user's preferences. Part of the recommendation engine could include a web application that displays recommended products based on a userid. What you learn in this course will set you up for building the web app portion of the recommendation engine.

## Lesson Overview

<https://www.youtube.com/watch?v=9WQF-CCNdJ8&feature=emb_logo>

### How to Think about This Lesson

The lesson first gives an overview of the three base languages for web development: html, css, and JavaScript. You could take an entire course just on each of these languages. The goal is for you to get comfortable writing at least some code in each language so that you understand the web template files at the end of the lesson. This lesson goes through a lot of information to get you up to speed.

To work with the web template and make a data dashboard, you will only need to write Python code. If you want to customize the dashboard, you can do so with just a few changes to the html code. But the underlying technologies of data dashboard will be css, html, JavaScript, and Python.

### Lesson Outline

* Basics of a web app
  + html
  + css
  + javascript
* Front-end libraries
  + boostrap
  + plotly
* Back-end libraries
  + flask
* Deploy a web app to the cloud

### Lesson Files

All of the lesson's exercises are contained in classroom workspaces. You'll even deploy a web app from the classroom workspace; however, if you prefer to work locally, you can find the lesson files in this **[data scientist nanodegree GitHub repo](https://github.com/udacity/DSND_Term2/tree/master/lessons/WebDevelopment" \t "_blank)**.

## The World Wide Web

<https://www.youtube.com/watch?v=Rxn-zCyg_iA&feature=emb_logo>

## Components of a Web App

<https://www.youtube.com/watch?v=2aJf5sO2ox4&feature=emb_logo>

## Introduction to HTML

<https://www.youtube.com/watch?time_continue=1&v=G7fBus1JSc0&feature=emb_logo>

### HTML Document Example

Here is an example of HTML code

**<!DOCTYPE html>**

<html>

<head>

<title>Page Title</title>

</head>

<body>

<h1>A Photo of a Beautiful Landscape</h1>

<a href="https://www.w3schools.com/tags">HTML tags</a>

<p>Here is the photo</p>

<img src="photo.jpg" alt="Country Landscape">

</body>

</html>

### Explanation of the HTML document

As you progress through the lesson, you'll find that the <head> tag is mostly for housekeeping like specifying the page title and adding meta tags. Meta tags are in essence information about the page that web crawlers see but users do not. The head tag also contains links to javascript and css files, which you'll see later in the lesson.

The website content goes in the <body> tag. The body tag can contain headers, paragraphs, images, links, forms, lists, and a handful of other tags. Of particular note in this example are the link tag <a> and the image tag <img>.

Both of these tags link to external information outside of the html doc. In the html code above, the link <a> tag links to an external website called w3schools. The href is called an attribute, and in this case href specifies the link.

The image <img> tag displays an image called "photo.jpg". In this case, the jpg file and the html document are in the same directory, but the documents do not have to be. The src attribute specifies the path to the image file relative to the html document. The alt tag contains text that gets displaced in case the image cannot be found.

### Full List of Tags and How to Use Them

This is a link to one of the best references for html. Use this website to look up html tags and how to use them. **[W3Schools HTML Tags](https://www.w3schools.com/tags/default.asp" \t "_blank)**

In fact, the **[W3Schools website](https://www.w3schools.com/" \t "_blank)** has a lot of free information about web development syntax.

### Checking your HTML

It's a good idea to check the validity of your HTML. Here is a website that checks your HTML for syntax errors: **[W3C Validator](https://validator.w3.org/" \l "validate_by_input" \t "_blank)**. Try pasting your HTML code here and running the validator. You can read through the error messages and fix your HTML.

## Div and Span

<https://www.youtube.com/watch?time_continue=43&v=cbKA_dvthcY&feature=emb_logo>

### Summary of Div and Span Elements

You can use div elements to split off large chunks of html into sections. Span elements, on the other hand, are for small chunks of html. You generally use span elements in the middle of a piece of text in order to apply a specific style to that text. You'll see how this works a bit later in the CSS portion of the lesson.

<div>

<p>This is an example of when to use a div elements versus a span element. A span element goes around <span>a small chunk of html</span></p>

</div>

## IDs and Classes

<https://www.youtube.com/watch?v=jnfDqdxDbO4&feature=emb_logo>

### Example HTML

<div id="top">

<p class="first\_paragraph">First paragraph of the section</p>

<p class="second\_paragraph">Second paragraph of the section</p>

</div>

<div id="bottom">

<p class="first\_paragraph">First paragraph of the section</p>

<p class="second\_paragraph">Second paragraph of the section</p>

</div>

## CSS

<https://www.youtube.com/watch?v=s_sdzHR9cs0&feature=emb_logo>

### CSS and this Lesson

To build the data dashboard at the end of this lesson, you won't need to actually write any CSS. Instead, you'll use libraries that take care of the CSS for you. In this that, that would be the **[Bootstrap library](https://getbootstrap.com/" \t "_blank)**.

But if you are interested in understanding what Bootstrap is doing under the hood, then you need to understand how to style a website with CSS. This page has a summary of some important aspects of CSS programming.

### What is the Purpose of CSS?

In most professional websites, css is kept in a separate stylesheet. This makes it easier to separate content (html) from style (css). Code becomes easier to read and maintain.

If you're interested in the history of css and how it came about, here is an interesting link: **[history of css](https://www.w3.org/Style/CSS20/history.html" \t "_blank)**.

CSS stands for cascading style sheets. The "cascading" refers to how rules trickle down to the various layers of an html tree. For example, you might specify that all paragraphs have the same font type. But then you want to override one of the paragraphs to have a different font type. How does a browser decide which rules apply when there is a conflict? That's based on the cascade over. You can read more about that **[here](https://www.lifewire.com/what-does-cascade-mean-3466872" \t "_blank)**.

### Different ways to write CSS

As discussed in the video, there are essentially two ways to write CSS: **inline** or with a **stylesheet**.

Inline means that you specify the CSS directly inside of an html tag like so:

<p style="font-size:20px;">This is a paragraph</p>

Alternatively, you can put the CSS in a stylesheet. The stylesheet can go underneath an html head tag like so:

...

<head>

<style>

p {font-size: 20px;}

</style>

</head>

Or the css can go into its own separate css file (extension .css). Then you can link to the css file within the html head tag like so:

<head>

<link rel="stylesheet" type"text/css" href="style.css">

</head>

where style.css is the path to the style.css file. Inside the style.css file would be the style rules such as

p {

color:red;

}

### CSS Rules and Syntax

CSS is essentially a set of rules that you can use to stylize html. The **[W3 Schools CSS Website](https://www.w3schools.com/css/default.asp" \t "_blank)** is a good place to find all the different rules you can use. These including styling text, links, margins, padding, image, icons and background colors among other options.

The general syntax is that you:

1. select the html element, id, and/or class of interest
2. specify what you want to change about the element
3. specify a value, followed by a semi-colon

For example

a {

text-decoration:none;

}

where a is the element of interest, text-decoration is what you want to change, and none is the value. You can write multiple rules within one set of brackets like:

a {

text-decoration:none;

color:blue;

font-weight:bold;

}

You can also select elements by their class or id.

To select by class name, you use a dot like so:

.class\_name {

color: red;

}

To select by id name, you use the pound sign:

**#id\_name** {

color: red;

}

You can make more complex selections as well like "select paragraphs inside the div with id "div\_top" . If your html looks like this,

<div id="div\_top">

<p>This is a paragraph</p>

</div>

then the CSS would be like this:

div**#div\_top** p {

color: red;

}

### Margins and Padding

The difference between margin and padding is a bit tricky. Margin rules specify a spatial buffer on the outside of an element. Padding specifies an internal spatial buffer.

These examples below show how this works. They use a div element with a border. Here is the div without any margin or padding:

<div style="border:solid red 1px;">

Box

</div>

Box

#### Margin

In this case, the div has a margin of 40 pixels. This creates a spatial buffer on the outside of the div element.

<div style="border:solid red 1px;margin:40px;">

Box

</div>

Box

#### Padding

This next case has a padding of 40px. In the case of padding, the spatial buffer is internal.

<div style="border:solid red 1px;padding:40px;">

Box

</div>

Box

#### Margin and Padding

In this case, the div element has both a margin of 40 pixels and a padding of 40 pixels.

<div style="border:solid red 1px;margin:40px;padding:40px;">

Box

</div>

Box

### Specifying Size: Pixels versus Percent versus EM Units

In CSS there are various ways to define sizes, widths, and heights. The three main ones are pixels, percentages, and em units.

When you use px, you're defining the exact number of pixels an element should use in terms of size. So

<p style="font-size: 12px;">

means the font-size will be exactly 12 pixels.

The percent and em units have a similar function. They dynamically change sizing based on a browser's default values. For example

<p style="font-size: 100%">

means to use the default browser font size. 150% would be 1.5 times the default font size. 50% would be half. Similarly, 1em unit would be 1 x default\_font. So 2em would be 2 x default font, etc. The advantage of using percents and em is that your web pages become dynamic. The document adapts to the default settings of whatever device someone is using be that a desktop, laptop or mobile phone.

As an aside, percentages and em units are actually calculating sizes relative to parent elements in the html tree. For example, if you specify a font size in a body tag , then the percentages will be relative to the body element:

<body style="font-size: 20px">

<p style="font-size:80%">This is a paragraph</p>

...

</body>

Because different browsers might render html and CSS differently, there isn't necessarily a right or wrong way to specify sizes. This will depend on who will use your website and on what type of devices. You can read more **[here](https://www.w3schools.com/html/html_responsive.asp" \t "_blank)**. You won't need to worry about all of this because in the web app, you're going to use a CSS framework that takes care of all of this for you.

### Practice Writing CSS

In the next exercise, you'll practice writing CSS.

## Bootstrap Library

<https://www.youtube.com/watch?time_continue=1&v=KsrqjguHWUI&feature=emb_logo>

### Documentation References

Here are some key parts of the Bootstrap documentation for your reference:

* **[Starter Template](https://getbootstrap.com/docs/4.0/getting-started/introduction/" \l "starter-template" \t "_blank)**
* **[Column Grid Explanation](https://getbootstrap.com/docs/4.0/layout/grid/" \t "_blank)**
* **[Containers and Responsive Layout](https://getbootstrap.com/docs/4.0/layout/overview/" \t "_blank)**
* **[Images](https://getbootstrap.com/docs/4.0/content/images/" \t "_blank)**
* **[Navigation Bars](https://getbootstrap.com/docs/4.0/components/navbar/" \t "_blank)**
* **[Font Colors](https://getbootstrap.com/docs/4.0/utilities/colors/" \t "_blank)**

### Why Bootstrap?

Bootstrap is one of the easier front-end frameworks to work with. Bootstrap eliminates the need to write CSS or JavaScript. Instead, you can style your websites with HTML. You will be able to design sleek, modern looking websites more quickly than if you were coding the CSS and JavaScript directly.

## Javascript

<https://www.youtube.com/watch?time_continue=1&v=vgXUKgsT_48&feature=emb_logo>

### JavaScript and this Lesson

To build the data dashboard at the end of this lesson, you won't need to write any JavaScript at all. That's because you'll use libraries (**[Bootstrap](https://getbootstrap.com/" \t "_blank)** and **[Plotly](https://plot.ly/" \t "_blank)**) that take care of the JavaScript for you.

You won't need to get into the details of JavaScript syntax, but it's good to have at least an idea of what is happening under the hood.

### What is JavaScript?

* JavaScript is a high-level language like Python, PHP, Ruby, and C++. It was specifically developed to make the front-end of a web application more dynamic; however, you can also use javascript to program the back-end of a website with the JavaScript runtime environment **[node](https://nodejs.org/en/" \t "_blank)**.
* Java and javaScript are two completely different languages that happen to have similar names.
* JavaScript syntax, especially for front-end web development, is a bit tricky. It's much easier to write front-end JavaScript code using a framework such as **[jQuery](http://api.jquery.com/" \t "_blank)**.

### Basic JavaScript Syntax

Here are a few rules to keep in mind when writing JavaScript:

* a line of code ends with a semi-colon ;
* () parenthesis are used when calling a function much like in Python
* {} curly braces surround large chunks of code or are used when initializing dictionaries
* [] square brackets are used for accessing values from arrays or dictionaries much like in Python

Here is an example of a JavaScript function that sums the elements of an array.

**function** **addValues**(x) {

**var** sum\_array = 0;

**for** (**var** i=0; i < x.length; i++) {

sum\_array += x[i];

}

**return** sum\_array;

}

addValues([3,4,5,6]);

### What is jQuery?

Jquery is a JavaScript library that makes developing the front-end easier. JavaScript specifically helps with manipulating html elements. The reason we are showing you Jquery is because the Bootstrap library you'll be using depends on Jquery. But you won't need to write any Jquery yourself.

Here is a link to the documentation of the core functions in jquery: **[jQuery API documentation](https://api.jquery.com/" \t "_blank)**

Jquery came out in 2006. There are newer JavaScript tools out there like **[React](https://reactjs.org/" \t "_blank)** and **[Angular](https://angularjs.org/" \t "_blank)**.

As a data scientist, you probably won't need to use any of these tools. But if you work in a startup environment, you'll most likely hear front-end engineers talking about these tools.

### jQuery Syntax

The jQuery library simplifies JavaScript quite a bit. Compare the syntax. Compare these two examples from the video for changing the h1 title element when clicking on the image.

This is pure JavaScript code for changing the words in the h1 title element.

**function** **headFunction**() {

document.getElementsByTagName("h1")[0].innerHTML =

"A Photo of a Breathtaking View";

}

This code searches the html document for all h1 tags, grabs the first h1 tag in the array of h1 tags, and then changes the html. Note that the above code is only the function. You'd also have to add an onClick action in the image html tag like so:

<img src="image.jpg" onclick="headFunction()">

The jQuery code is more intuitive. Once the document has loaded, the following code adds an onclick event to the image. Once the image is clicked, the h1 tag's text is changed.

$(document).ready(**function**(){

$("img").click(**function**(){

$("h1").text("A Photo of a Breathtaking View");

});

});

The dollar sign $ is jQuery syntax that says "grab this element, class or id". That part of the syntax should remind you somewhat of CSS. For example $("p#first") means find the paragraph with id="first". Or $("#first") would work as well.

Javascript has something called callback function, which can make learning javascript a bit tricky. Callback functions are essentially functions that can be inputs into other functions. In the above code, there is the ready() function that waits for the html document to load. Then there is another function being passed into the ready function. This section function adds an on-click event to an image tag. Then there's another function passed into the click() function, which changes the h1 text.

### Exercise

In the next exercise, you'll write a bit of jQuery just so that you can see how it works and what it does. This is the only time in the lesson you'll actually write any JavaScript.

## Plotly

[https://www.youtube.com/watch?time\_continue=1&v=QsmOW1jNeio&feature=emb\_logo](https://www.youtube.com/watch?time_continue=57&v=QsmOW1jNeio&feature=emb_logo)

### Chart Libraries

There are many web chart libraries out there for all types of use cases. When choosing a library, you should consider checking whether or not the library is still being actively developed.

**[d3.js](https://d3js.org/" \t "_blank)** is one of the most popular (and complex!) javascript data visualization libraries. This library is still actively being developed, which you can tell because the latest commit to the **[d3 GitHub repository](https://github.com/d3/d3" \t "_blank)** is fairly recent.

Other options include **[chart.js](https://classroom.udacity.com/nanodegrees/nd009t/parts/59b40204-e44a-4d2e-9542-742120b99f8b/modules/362a2065-0966-4854-b4cd-8a58666d0493/lessons/3737ebcb-984e-4959-bf2a-95fe13de4916/concepts/ww.chartjs.org/" \t "_blank)**, **[Google Charts](https://developers.google.com/chart/" \t "_blank)**, and **[nvd3.js](http://nvd3.org/" \t "_blank)**, which is built on top of d3.js

### Why Plotly

For this lesson, we've chosen **[plotly](https://plot.ly/" \t "_blank)** for a specific reason: Plotly, although a private company, provides open source libraries for both JavaScript and Python.

Because the web app you're developing will have a Python back-end, you can use the Python library to create your charts. Rather than having you learn more JavaScript syntax, you can use the Python syntax that you already know. However, you haven't built a back-end yet, so for now, you'll see the basics of how Plotly works using the JavaScript library. The syntax between the Python and Javascript versions is similar.

Later in the lesson, you'll switch to the Python version of the Plotly library so that you can prepare visualizations on the back-end of your web app. Yet you could write all the visualization code in JavaScript if you wanted to. Watch the screencast below to learn the basics of how Plotly works, and then continue on to the Plotly exercise.

Here are a few links to some helpful parts of the plotly documentation:

* **[javascript examples](https://plot.ly/javascript/" \t "_blank)**
* **[getting started](https://plot.ly/javascript/getting-started/" \t "_blank)**
* **[linking to the plotly library](https://plot.ly/javascript/getting-started/" \l "plotlyjs-cdn" \t "_blank)**

## The Backend

<https://www.youtube.com/watch?time_continue=1&v=Esl0NL63S2c&feature=emb_logo>

In this next part of the lesson, you'll build a backend using Flask. Because Flask is written in Python, you can use any Python library in your backend including pandas and scikit-learn.

In this part of the lesson, you'll practice:

* setting up the backend
* linking the backend and the frontend together
* deploying the app to a server so that the app is available from a web address

### What is Flask?

**[Flask](http://flask.pocoo.org/" \t "_blank)**. A web framework takes care of all the routing needed to organize a web page so that you don't have to write the code yourself!

When you type "**http://www.udacity.com"** into a browser, your computer sends out a request to another computer (ie the server) where the Udacity website is stored. Then the Udacity server sends you the files needed to render the website in your browser. The Udacity computer is called a server because it "serves" you the files that you requested.

The HTTP part of the web address stands for Hypter-text Transfer Protocol. HTTP defines a standard way of sending and receiving messages over the internet.

When you hit enter in your browser, your computer says "get me the files for the web page at **www.udacity.com"**: except that message is sent to the server with the syntax governed by HTTP. Then the server sends out the files via the protocol as well.

There needs to be some software on the server that can interpret these HTTP requests and send out the correct files. That's where a web framework like Flask comes into play. A framework abstracts the code for receiving requests as well as interpreting the requests and sending out the correct files.

### Why Flask?

* First and foremost, you'll be working with Flask because it is written in Python. You won't need to learn a new programming language.
* Flask is also a relatively simple framework, so it's good for making a small web app.
* Because Flask is written in Python, you can use Flask with any other Python library including pandas, numpy and scikit-learn. In this lesson, you'll be deploying a data dashboard and pandas will help get the data ready.

Continue to start building the backend.

## Getting Started with Flask

<https://www.youtube.com/watch?time_continue=2&v=i_U3O-7cymk&feature=emb_logo>

### Using Flask in the Classroom Workspace

In the next part of the lesson, you'll see a classroom workspace. The classroom workspace already has Flask set up for you. So for now, all you need to do to run the Flask app is to open a Terminal and type.

python worldbank.py

That assumes you are in the default workspace directory within Terminal. That will get the server running.

### Seeing your App in the Workspace

Once the server is running, open a new terminal window and type

env | grep WORK

This command will return the Linux environmental variables that contain information about your classroom workspace. The env command will list all the environmental variables. The | symbol is a pipe for sending output from one command to another. The grep command searches text, so grep WORK will search for any text containing the word WORK.

The command should return two variables:

WORKSPACEDOMAIN=udacity-student-workspaces.com

WORKSPACEID=viewc7f3319f2

Your WORKSPACEID variable will be different but the WORKSPACEDOMAIN should be the same. Now, open a new web browser window, and type the following in the address bar:

http:*//WORKSPACEID-3001.WORKSPACEDOMAIN*

In this example, that would be: **[https://viewc7f3319f2-3001.udacity-student-workspaces.com/](https://viewc7f3319f2-3001.udacity-student-workspaces.com/" \t "_blank)**

DON'T FORGET TO INCLUDE -3001. You should be able to see the web app. The number 3001 represents the port for accessing your web app.

### Creating New Pages

To create a new web page, you first need to specify the route in the routes.py as well as the name of the html template.

@app.route('/new-route')

**def** **render\_the\_route**():

**return** render\_template('new\_route.html')

The route name, function name, and template name do not have to match; however, it's good practice to make them similar so that the code is easier to follow.

The new\_route.html file must go in the templates folder. Flask automatically looks for html files in the templates folder.

### What is @app.route?

To use Flask, you don't necessarily need to know what @app.route is doing. You only have to remember that the path you place inside of @app.route() will be the web address. And then the function you write below @app.route is used to render the correct html template file for the web address.

In Python, the @ symbol is used for decorators. Decorators are a shorthand way to input a function into another function. Take a look at this code. Python allows you to use a function as an input to another function:

**def** **decorator**(input\_function):

**return** input\_function

**def** **input\_function**():

print("I am an input function")

decorator\_example = decorator(input\_function)

decorator\_example()

Running this code will print the string:

I am an input function

Decorators provide a short-hand way of getting the same behavior:

**def** **decorator**(input\_function):

print("Decorator function")

**return** input\_function

@decorator

**def** **input\_function**():

print("I am an input function")

input\_function()

This code will print out:

Decorator function  
I am an input function

Instead of using a decorator function, you could get the same behavior with the following code:

input\_function = decorator(input\_function)

input\_function()

Because @app.route() has the . symbol, there's an implication that app is a class (or an instance of a class) and route is a method of that class. Hence a function written underneath @app.route() is going to get passed into the route method. The purpose of @app.route() is to make sure the correct web address gets associated with the correct html template. This code

@app.route('/homepage')

**def** **some\_function**()

**return** render\_template('index.html')

is ensuring that the web address '**[www.website.com/homepage`](http://www.website.com/homepage%60" \t "_blank)** is associated with the index.html template.

If you'd like to know more details about decorators and how @app.route() works, check out these tutorials:

* **[how @app.route works](https://ains.co/blog/things-which-arent-magic-flask-part-1.html" \t "_blank)**
* **[general decorators tutorial](https://realpython.com/primer-on-python-decorators/" \t "_blank)**

## Flask + Pandas

<https://www.youtube.com/watch?v=L_M_8UVY42k&feature=emb_logo>

### Code from the Screencast

Here is the code from the routes.py file before refactoring.

The data set comes from this link at the World Bank's data repository: **[link to dataset](https://data.worldbank.org/indicator/SP.RUR.TOTL.ZS?view=chart" \t "_blank)**

**from** worldbankapp **import** app

**from** flask **import** render\_template

**import** pandas **as** pd

df = pd.read\_csv('data/API\_SP.RUR.TOTL.ZS\_DS2\_en\_csv\_v2\_9948275.csv', skiprows=4)

*# Filter for 1990 and 2015, top 10 economies*

df = df[['Country Name','1990', '2015']]

countrylist = ['United States', 'China', 'Japan', 'Germany', 'United Kingdom', 'India', 'France', 'Brazil', 'Italy', 'Canada']

df = df[df['Country Name'].isin(countrylist)]

*# melt year columns and convert year to date time*

df\_melt = df.melt(id\_vars='Country Name', value\_vars = ['1990', '2015'])

df\_melt.columns = ['country','year', 'variable']

df\_melt['year'] = df\_melt['year'].astype('datetime64[ns]').dt.year

*# add column names*

df\_melt.columns = ['country', 'year', 'percentrural']

*# prepare data into x, y lists for plotting*

df\_melt.sort\_values('percentrural', ascending=**False**, inplace=**True**)

data = []

**for** country **in** countrylist:

x\_val = df\_melt[df\_melt['country'] == country].year.tolist()

y\_val = df\_melt[df\_melt['country'] == country].percentrural.tolist()

data.append((country, x\_val, y\_val))

print(country, x\_val, y\_val)

@app.route('/')

@app.route('/index')

**def** **index**():

**return** render\_template('index.html')

@app.route('/project-one')

**def** **project\_one**():

**return** render\_template('project\_one.html')

### Exercise

The next exercise will be after the section on using Plotly, Pandas, and Flask together. For now, the next part of the lesson has the refactored code shown in this screencast so that you can explore it in more detail. You'll find it in the 2\_flask+pandas\_example folder.

## Flask with Plotly and Pandas Part 1

<https://www.youtube.com/watch?v=xg7P8MnItdI&feature=emb_logo>

In this next video, you'll see an example of how to pass data from the back end to the front end of the web app. In the next four parts of this lesson, you'll get a sense for how data and Plotly code can be taken from the back end and, sent to the front end, and then used to render plots on the front end. The goal of these next few videos is to show you how the web template works, which you'll be using later in the final exercise.

### Summary Part 1

The purpose of this section is to give you an idea of how the final web app works in terms of passing information back and forth between the back end and front end. The web template you'll be using at the end of the lesson will already provide the code for sharing information between the back and front ends. Your task will be to wrangle data and set up the plotly visualizations using Python. But it's important to get a sense for how the web app works.

In the video above, the data set was sent from the back end to the front end. This was accomplished by including a variable in the render\_template() function like so:

data = data\_wrangling()

@app.route('/')

@app.route('/index')

**def** **index**():

**return** render\_template('index.html', data\_set = data)

What this code does is to first load the data using the data\_wrangling function from wrangling.py. This data gets stored in a variable called data.

In render\_template, that data is sent to the front end via a variable called data\_set. Now the data is available to the front\_end in the data\_set variable.

In the index.html file, you can access the data\_set variable using the following syntax:

{{ data\_set }}

You can do this because Flask comes with a template engine called **[Jinja](http://jinja.pocoo.org/" \t "_blank)**. Jinja also allows you to put control flow statements in your html using the following syntax:

{% for tuple in data\_set %}

<p>{{tuple}}</p>

{% end\_for %}

The logic is:

1. Wrangle data in a file (aka Python module). In this case, the file is called wrangling.py. The wrangling.py has a function that returns the clean data.
2. Execute this function in routes.py to get the data in routes.py
3. Pass the data to the front-end (index.html file) using the render\_template method.
4. Inside of index.html, you can access the data variable with the squiggly bracket syntax {{ }}

### Next

In the next part, you'll see how to create a Plotly visualization on the back end and then send the visualization code to the front end for rendering.

## Flask with Plotly and Pandas Part 2

<https://www.youtube.com/watch?v=yx-DRzMsblI&feature=emb_logo>

In this section, you'll see how to create a Plotly visualization on the back end and then send the information to the front end for rendering.

### Summary Part 2

In the second part, a Plotly visualization was set up on the back-end inside the routes.py file using Plotly's Python library. The Python plotly code is a dictionary of dictionaries. The Python dictionary is then converted to a JSON format and sent to the front-end via the render\_templates method.

Simultaneously a list of ids are created for the plots. This information is also sent to the front-end using the render\_template() method.

On the front-end, the ids and visualization code (JSON code) is then used with the Plotly javascript library to render the plots.

In summary:

1. Python is used to set up a Plotly visualization
2. An id is created associated with each visualization
3. The Python Plotly code is converted to JSON
4. The ids and JSON are sent to the front end (index.html).
5. The front end then uses the ids, JSON, and JavaScript Plotly library to render the plots.

### JavaScript or Python

You could actually do all of this with only JavaScript. You would read the data, wrangle the data, and then create the plots all using JavaScript; however, to do all of this in JavaScript, you'd need to learn more about JavaScript programming. Instead, you can use the pandas and Python skills you already have to wrangle data on the back-end.

## Flask with Plotly and Pandas Part 3

<https://www.youtube.com/watch?v=e8owK5zk-g8&feature=emb_logo>

Here, the screencast video shows how to make more complex visualizations in Plotly. This example shows a line chart containing a unique line for each country in the data set.

### Summary Part 3

In part 3, the code iterated through the data set to create a visualization with multiple lines: one for each country.

The original code for a line chart with a single line was:

graph\_one = [go.Scatter(

x = data[0][1],

y = data[0][2],

mode = 'lines',

name = country

)]

To make a visualization with multiple lines, graph\_one will be a list of line charts. This was accomplished with the following code:

graph\_one = []

**for** data\_tuple in data:

graph\_one.append(**go**.Scatter(

x = data\_tuple[1],

y = data\_tuple[2],

mode = 'lines',

name = data\_tuple[0]

))

### Next

In the last section of flask, plotly, and pandas, you'll see how to add more visualizations to the data dashboard. Then, you'll see some example code and finally you will practice using flask, plotly, and pandas together.

## Flask with Plotly and Pandas Part 4

<https://www.youtube.com/watch?v=4IF2G9Fehb4&feature=emb_logo>

In this next section, you'll see how to add more visualizations in the back end code and then render those visualizations on the front end.

### Summary Part 4

In the last part, three more visualizations were added to the wrangling Python module. The wrangling included reading in the data, cleaning the data, and preparing the Plotly code. Each visualization's code was appended to a list called figures. These visualizations were then imported into the routes.py file. This figures list was sent from the back end to the front end via the render\_template method. A list of ids were also sent from the back end to the front end.

Then on the front end (index.html), a div was created for each visualization's id. And with help from the JavaScript Plotly library, each visualization was rendered inside appropriate div.

### Beyond a CSV file

Besides storing data in a local csv file (or text, json, etc.), you could also store the data in a database such as a SQL database.

The database could be local to your website meaning that the database file is stored on the same server as your website; alternatively, the database could be stored somewhere else like on a separate database server or with a cloud service like Amazon AWS.

Using a database with your web app goes beyond the scope of this introduction to web development, here are a few resources for using databases with Flask apps:

* **[Tutorial - Using Databases with Flask](https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-iv-database" \t "_blank)**
* **[SQL Alchemy](http://docs.sqlalchemy.org/en/latest/" \t "_blank)**- a Python toolkit for working with SQL
* **[Flask SQLAlchemy](http://flask-sqlalchemy.pocoo.org/2.3/" \t "_blank)** - a Flask library for using SQLAlchemy with Flask

### Next Steps

In the next part of the lesson, you can look at the code and try running the web app from the classroom. Then in the next exercise, you'll practice adding another visualization to the web app.

## Deployment

<https://www.youtube.com/watch?v=YPfNzpnm_Rk&feature=emb_logo>

### Other Services Besides Heroku

Heroku is just one option of many for deploying a web app, and Heroku is actually owned by **[Salesforce.com](https://www.salesforce.com/" \t "_blank)**.

The big internet companies offer similar services like **[Amazon's Lightsail](https://aws.amazon.com/lightsail/" \t "_blank)**, **[Microsoft's Azure](https://azure.microsoft.com/en-us/resources/samples/python-docs-hello-world/" \t "_blank)**, **[Google Cloud](https://cloud.google.com/appengine/docs/standard/python/getting-started/python-standard-env" \t "_blank)**, and **[IBM Cloud (formerly IBM Bluemix)](https://www.ibm.com/blogs/bluemix/2015/03/simple-hello-world-python-app-using-flask/" \t "_blank)**. However, these services tend to require more configuration. Most of these also come with either a free tier or a limited free tier that expires after a certain amount of time.

### Instructions Deploying from the Classroom

Here is the code used in the screencast to get the web app running:

First, a new folder was created for the web app and all of the web app folders and files were moved into the folder:

**mkdir** web\_app

**mv** -t web\_app **data** worldbankapp wrangling\_scripts worldbank.py

The next step was to create a virtual environment and then activate the environment:

conda **update** python

python3 -**m** venv worldbankvenv

**source** worldbankenv/**bin**/**activate**

Then, pip install the Python libraries needed for the web app

**pip** install flask pandas plotly gunicorn

The next step was to install the heroku command line tools:

**curl** https://cli-assets.heroku.com/install-ubuntu.sh | sh

https://devcenter.heroku.com/articles/heroku-cli#standalone-installation

heroku —-version

And then log into heroku with the following command

**heroku** login

Heroku asks for your account email address and password, which you type into the terminal and press enter.

The next steps involved some housekeeping:

* remove app.run() from worldbank.py
* type cd web\_app into the Terminal so that you are inside the folder with your web app code.

Then create a proc file, which tells Heroku what to do when starting your web app:

**touch** Procfile

Then open the Procfile and type:

web gunicorn worldbank:app

Next, create a requirements file, which lists all of the Python library that your app depends on:

pip freeze > requirements.txt

And initialize a git repository and make a commit:

git init

git add .

git **commit** -**m** ‘**first** **commit**’

Now, create a heroku app:

heroku create **my**-app-name

where my-app-name is a unique name that nobody else on Heroku has already used.

The heroku create command should create a git repository on Heroku and a web address for accessing your web app. You can check that a remote repository was added to your git repository with the following terminal command:

**git** remote -v

Next, you need to push your git repository to the remote heroku repository with this command:

git **push** heroku master

Now, you can type your web app's address in the browser to see the results.

### Virtual Environments vs. Anaconda

Virtual environments and Anaconda serve a very similar purpose. Anaconda is a distribution of Python (and the analytics language R) specifically for data science. Anaconda comes installed with a package and environment manager called conda. You can create **[separate environments using conda](https://conda.io/docs/user-guide/tasks/manage-environments.html" \t "_blank)**. However, these environments automatically come with Python packages meant for data science.

Virtual environments, on the other hand, come with the Python language but do not pre-install other packages.

The classroom workspace has many other Python libraries pre-installed including an installation of **[Anaconda](https://www.anaconda.com/distribution/" \t "_blank)**.

When installing a web app to a server, you should only include the packages that are necessary for running your web app. Otherwise you'd be installing Python packages that you don't need.

To ensure that your app only installs necessary packages, you should create a **virtual Python environment**. A virtual Python environment is a separate Python installation on your computer that you can easily remove and won't interfere with your main Python installation.

There is more than one Python package that can set up virtual environments. In the past, you had to install these packages yourself. With Python 3.6, there is a virtual environment package that comes with the Python installation. The packaged is called **[venv](https://docs.python.org/3/library/venv.html" \l "module-venv" \t "_blank)**

However, there is a bug with anaconda's 3.6 Python installation on a Linux system. So in order to use venv in the workspace classroom, you first need to update the Python installation as shown in the instructions above.

### Creating a Virtual Environment in the Classroom

Open a terminal window in a workspace and type:

**conda** update python

When asked for confirmation, type y and hit enter. Your Python installation should update.

Next, make sure you are in the folder where you want to build your web app. In the classroom, the workspace folder is fine. But on your personal computer, you'll want to make a new folder. For example:

**mkdir** myapp

will create a new folder called myapp and cd myapp will change your current directory so that you are inside the myapp folder.

Then to create a virtual environment type:

python3 -m venv name

where name can be anything you want. You'll see a new folder appear in the workspace with your environment name.

Finally, to activate the virtual environment. Type:

source name/bin/activate

You can tell that your environment is activated because the name will show up in parenthesis on the left side of the terminal.

### Creating a Virtual Environment Locally on Your Computer

You can develop your app using the classroom workspace. If you decide to develop your app locally on your computer, you should set up a virtual environment there as well. Different versions of Python have different ways of setting up virtual environments. Assuming you are using Python 3.6 and are on a linux or macOS system, then you should be able to set up a virtual environment on your local machine just by typing:

python3 -m venv name

and then to activate:

source name/bin/activate

On Windows, the command is;

c:\>c:\Python35\python -m venv c:\path\to\myenv

and to activate:

C:\> <venv>\Scripts\activate.bat

For more information, read through this **[link](https://docs.python.org/3/tutorial/venv.html" \t "_blank)**.

### Databases for Your App

The web app in this lesson does not need a database. All of the data is stored in CSV files; however, it is possible to include a database as part of a Flask app. One common use case would be to store user login information such as username and password.

Flask is database agnostic meaning Flask can work with a number of different database types. If you are interested in learning about how to include a database as part of a Flask app, here are some resources:

* **[Flask Mega Tutorial](https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-iv-database" \t "_blank)**
* **[Heroku - Provision a Database](https://devcenter.heroku.com/articles/getting-started-with-python" \l "provision-a-database" \t "_blank)**

### Deployment

In the next part of the lesson, you'll find a workspace where you can practice deploying the world bank web app. Set up an account on **[Heroku](https://classroom.udacity.com/nanodegrees/nd009t/parts/59b40204-e44a-4d2e-9542-742120b99f8b/modules/362a2065-0966-4854-b4cd-8a58666d0493/lessons/3737ebcb-984e-4959-bf2a-95fe13de4916/concepts/www.heroku.com" \t "_blank)** and then follow the instructions shown in this part of the lesson.

You'll need to use a different name for the web app since the one used in this lesson is already taken.

## Lesson Summary

<https://www.youtube.com/watch?v=8MyuJx5yu38&feature=emb_logo>

### Portfolio Exercise

In the next section of the classroom, you'll find a portfolio exercise based on this web development lesson. Continue to the next section to get started.

# Lesson 2: Portfolio Exercise: Deploy a Data Dashboard

## Introduction

Personal portfolios are an excellent way to demonstrate your knowledge and creativity. In fact, they are little by little becoming a must-have for people working in the tech industry. In this portfolio building exercise, you will create a data dashboard using Bootstrap, Plotly, Flask and Heroku.

**Note that a portfolio exercise like this is not reviewed. So you will not submit your work on this, and you do not need to complete this assignment in order to graduate.**

Your main job will be to write Python code that reads in data, cleans the data, and then uses the data to make Plotly visualizations. This is your opportunity to show off your Python coding ability and visualization encoding skills.

In the next part of the lesson, you'll find a workspace where you can develop the web app. Note that there is also an optional advanced version of the project where you're encouraged to pull data from an API. You'll see in this lesson that there are a few sections with "[advanced version]" in the title. If you'd like to do the advanced version, then you'll want to go through this entire lesson before starting to develop your app.

### General Instructions

Develop and deploy a data dashboard. The Web Development lesson has all of the information you need. If you are new to web development, you might have to go back to the concepts and rewatch some of the videos. The "deployment" parts of the lesson should be especially helpful. The video in that part of the lesson shows how to deploy a web app to Heroku. And the associated exercise has a complete, functioning web app with visualizations.

Most of the work will involve:

1. Wrangling your chosen data set to get the data in the format you want
2. Writing Python code to read in the data set and set up Plotly plots
3. Tweaking HTML so that the website has the design and information that you want.

We are providing a template that uses the **[Bootstrap library](https://getbootstrap.com/" \t "_blank)** and **[Flask framework](http://flask.pocoo.org/" \t "_blank)**. The template is the same one used to build the app in the course except the name of the app has been changed. In the template, everything has the generic name "myapp" instead of "worldbankapp". The template is set up so that you can use pandas for loading the data and Python to create the dictionaries needed for plotly.

You'll only need to modify the following files:

* wrangle\_data.py
* index.html

Although the front-end is already set up for you, you should change the links and titles in index.html. If you want to add more visualizations or remove visualizations, you'll need to adjust the front-end code in index.html accordingly. That will involve adding or removing rows and columns in the HTML file.

For deployment, you can use a back-end service like **[Heroku](https://heroku.com/" \t "_blank)**.

### How to Build the App

You'll find a workspace in the next part of the lesson. The workspace already contains the template code with a working web app. The web app has a back-end and front-end. Recall that you can run the web app from the workspace:

To run the app from the workspace, open a terminal and type env | grep WORK. Note the WORKSPACEDOMAIN and WORKSPACEID. To start the web app, type python myapp.py.

You can open a new browser window and go to the address: http://WORKSPACESPACEID-3001.WORKSPACEDOMAIN replacing WORKSPACEID and WORKSPACEDOMAIN with your values.

However, there is no data for the visualizations. You'll need to write a Python script that reads in the data files of your choosing and sets up the plots for Plotly. The process will be exactly the same as the one presented in the web development course.

**If you need to upload any files to the workspace, you can do so by clicking on the plus (+) sign and choosing "add file" or "add folder".**

The template code is also available on GitHub as part of the **[data scientist nanodegree term 2 repo](https://github.com/udacity/DSND_Term2/tree/master/lessons/WebDevelopment" \t "_blank)**.

Test your app in the workspace to make sure that everything is working. You'll see that if you start the app without modifying any of the code, the app currently works.

You should also save your work to a **[GitHub](https://github.com/" \t "_blank)** or **[GitLab](https://about.gitlab.com/" \t "_blank)** repository so that you can use your code as part of your professional portfolio.

Once you're ready to deploy the app, don't forget to remove the app.run() line of code in the myapp.py file (In the web development lesson, myapp.py was called worldbank.py). You'll need to add a Procfile and requirements.txt file as well. Follow the instructions in the web development lesson to learn how to deploy the app from the classroom. And always comment your code :-)!

Also, at the end of this page you're reading, you'll find information about a more advanced version of the data dashboard that you can build.

### Steps

Here is a reminder of the steps you'll need to do:

* find a data set or a few data sets that you're interested in
* explore and clean the data set
* put the data into a csv file or files - you can use pandas or spreadsheet software to do this
* upload your data sets to the correct folder
* write a Python script to read in the data set and set up the Plotly visualizations
* set up a virtual environment and install the necessary libraries to run your app
* run your web app locally to make sure that everything works
* deploy the app to Heroku or some other back-end service

### Where to Build the Web App

We are providing a workspace containing a web app template. You can use this template to build and deploy your web app within the classroom.

The classroom has an Ubuntu Linux environment. Developing the app locally on macOS should be very similar. On a Windows machine, the commands are slightly different and you'll need to use the command prompt. This link contains a **[comparison of MS-DOS vs Linux commands](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Step_by_Step_Guide/ap-doslinux.html" \t "_blank)**.

To install the Heroku command line interface on a Windows machine, follow the instructions **[here](https://devcenter.heroku.com/articles/heroku-cli" \t "_blank)** on the Heroku website.

### Advanced Version of the Exercise

If you'd like an extra challenge, consider using an API to obtain your data. API stands for Application Programming Interface. An API provides a convenient way for two applications to communicate with each other. To be more concrete you can pull data directly from the World Bank API, clean the data in the back-end using pandas, and then display the results on your front-end. This would be instead of using a csv file for your data.

The benefit is that if the data ever changes, your web app will automatically have the correct data. Many companies provide APIs for accessing their data including Facebook, Twitter, Google among others. As an example, here is an API for pulling data about **[DVDs, movies, books, and games](http://developer.rovicorp.com/docs" \t "_blank)**.

After the workspace, you'll find a set of concepts that explain how to use the World Bank API. Go through that material if you'd like an extra challenge for building your web app.

## Troubleshooting Possible Errors

### Here are a few issues that students have run into:

* remember to remove the the line app.run(host='0.0.0.0', port=3001, debug=True) in the myapp.py file before deploying to Heroku; however, to run the web app from within the Udacity workspace, you do need that line of code
* In the web development lesson, the Flask application file was called worldbank.py. In the template code, the same file is called myapp.py. So the Procfile should contain the line web gunicorn myapp:app.
* The template files can be deployed to Heroku. In other words, the template files already contain a working, deployable web app. If you're having trouble deploying your app to Heroku, try deploying the template files first. That should help with debugging because then you will know if an issue relates to how you are interacting with Heroku, or if there's a bug in your code.

## Congratulations

<https://www.youtube.com/watch?time_continue=1&v=rW1YP1aSb08&feature=emb_logo>

## APIs [advanced version]

### What is an API?

Instead of downloading World Bank data via a csv file, you're going to download the data using the World Bank API.

API is an acronym that stands for application programming interface. API’s provide a standardized way for two applications to talk to each other. For this project, the applications communicating with each other are the server application where World Bank stores data and your web application.

If you wanted to pull data directly from the World Bank’s server, you’d have to know what database system the World Bank was using. You’d also need permission to log in directly to the server, which would be a security risk for the World Bank. And if the World Bank ever migrated its data to a new system, you would have to rewrite all of your code again.

The API sits between your web app and the World Bank server. And the API allows you to execute code on the World Bank server without getting direct access.

All sorts of companies have public facing APIs including Facebook, Twitter, Google and Pinterest. You can pull data from these companies to create your own applications.

In the next section, you’ll get practice using Python to pull data from the World Bank API. This will set you up for creating the web app with data from the API instead of using data from a csv file.

### APIs Besides the World Bank

All types of companies have APIs. Some of these APIs are only for internal company use while other APIs help the public consume data. A few examples of public APIs include the **[Twitter API](https://developer.twitter.com/en/docs.html" \t "_blank)**, the **[Google Maps API](https://cloud.google.com/maps-platform/" \t "_blank)**, the **[Facebook Graph API](https://developers.facebook.com/docs/graph-api" \t "_blank)**, and the **[US Government Data APIs](https://www.data.gov/developers/apis" \t "_blank)**.

In addition, oftentimes you can find open source libraries or development kits for connecting to an API. For example, here is an open source Python development kit for the **[Facebook Graph API](https://github.com/mobolic/facebook-sdk" \t "_blank)**.

Some APIs might be used for pulling data from a database. But other APIs are for adding data to a database. For example, you might make an application that automatically tweets the current weather. In that case, you would use the Twitter API to post a tweet, which in reality inserts a tweet into Twitter's database.

### Using an API

In the next few parts of the lesson, you'll see how to use the World Bank API. This API is relatively straightforward to use. Each API, however, will have a different set up and only allow you to take certain actions. In general, you send a request via a web url that specifies the information you want. You receive data back typically in **[XML](https://www.w3schools.com/xml/xml_whatis.asp" \t "_blank)** or **[JSON](https://www.w3schools.com/js/js_json_intro.asp" \t "_blank)**.

The XML standard was developed in the 1970s and 1980s and soon became a common way to transfer data over the web. JSON was developed in the mid 1990s. Over time, **[JSON has increased in popularity relative to XML](https://www.programmableweb.com/news/jsons-eight-year-convergence-xml/2013/12/26" \t "_blank)** perhaps because JSON is easier to parse.

Some APIs require authentication; essentially the company with the API gives you 'credentials' so that they can track how you are using the API and ensure you have the proper permissions.

Some APIs might let you extract data from a database. Other APIs might even let you insert data into a database depending on the use case. Most APIs include extensive documentation so that you can figure out how to use APIs.

If you ever can’t figure out how to use an API, search online for examples. You can search for something like, “Examples for using the World Bank API” or “Examples for using the Facebook API”.

Move on to the next section to see how to use the World Bank API and incorporate it into a web app.

## The World Bank API [advanced version]

<https://www.youtube.com/watch?v=nygWkgUQNfo&feature=emb_logo>

### REST Architecture

REST is a software architecture for the web. You don't need to understand how REST works in order to use an API. but you will see the term used quite frequently when working with APIs. Modern web APIs are often called RESTFul to indicate that they conform to a REST Architecture.

* **[REST API Tutorial](https://restfulapi.net/" \t "_blank)**
* **[Wikipedia Article on REST](https://en.wikipedia.org/wiki/Representational_state_transfer" \t "_blank)**

### World Bank API

Here is the website where the csv files were downloaded for the World Bank web app: **[World Bank Indicator Data](https://data.worldbank.org/indicator?tab=featured" \t "_blank)**

And here is the link to the World Bank API documenation: **[World Bank API Documentation](https://datahelpdesk.worldbank.org/knowledgebase/articles/889392-api-documentation" \t "_blank)**

One tricky aspect of working with the World Bank API is that it only gives back 50 results at a time. There is an option called per\_page that allows you to return up to 1000 results. However, some queries might have more than 1000 results. That's where the page option comes into play. You'll notice that at the very beginning of the data, there is a variable called page and another one called pages. If page=1 and pages=4, then you'd need to write 4 queries with the option page=1, page=2, page=3 and page=4.

Next, you'll practice pulling data from the API using Python code.

## World Bank Data Dashboard [advanced version]

SEND FEEDBACK

### Code Walk-Through

### Link to the Code

You can find code for this data dashboard here on **[GitHub](https://github.com/udacity/DSND_Term2/tree/master/lessons/WebDevelopment/AdvancedDataDashboardCode/world_bank_api_dashboard" \t "_blank)**.

### How the Filter Works

This version of the web app has a filter made with a form. When you check the boxes on the form and click submit, the form gets submitted to the index.html page. It's essentially a circle where the index.html loads, the form gets submitted to index.html itself, and then index.html loads again. With a web form, you could also submit the form to a different web page.

On the back-end, routes.py can access the information that was submitted with the web form; the front-end receives information about which boxes were checked.

### Code your Project

Start working on your project! Go back to the "Workspace Portfolio Exercise" with the template code. You'll find it earlier in this Portfolio Exercise lesson. Here are a few APIs that you might find interesting to work with:

* **[World Bank API](https://datahelpdesk.worldbank.org/knowledgebase/articles/889386-developer-information-overview" \t "_blank)**
* **[API for Entertainment Industry Data](http://developer.rovicorp.com/docs" \t "_blank)**
* **[City of Berlin Open Data API](https://daten.berlin.de/datensaetze/deutsche-digitale-bibliothek-ddb-api" \t "_blank)**

Many government and city agencies have APIs where you can access city data.