Build an Interactive Data Analytics Dashboard with Python

by
Teddy Petrou

Contents

Chapter 1

Getting Started

Welcome to **Build an Interactive Data Analytics Dashboard with Python**. You will be building the Coronavirus Forecasting Dashboard available for viewing at coronavirus-project.dunderdata.com. At the end of this book, you'll have all the skills to launch your own dashboard using Dash on a remote web server for the world to see.

1.1 Get the Digital Material

There is a large amount of digital material that accompanies this text and is necessary to complete the dashboard. All code, Jupyter Notebooks, plus 12 hours of video lectures is available to purchase at dunderdata.com/build-an-interactive-data-analytics-dashboard-with-python for $5 when using the code **KDP_DASHBOARD**. This is 90% off the current price. This book assumes you have access to this material.

1.2 Components

This book is divided into the following five major components:

- Data preparation (Chapters 2-4)
- Model building (Chapters 5-7)
- Code organization (Chapters 8-9)
- Application development (Chapters 10-12)
- Deployment (Chapter 13)

1.3 Requirements

We'll begin by ensuring that we can run the dashboard application from our local machine. The only necessary software before getting started is Python 3.7 or above.

1.4 Install necessary packages

There are several third-party Python libraries that need to be installed in order to run the dashboard. After downloading all of the course contents, open up the command prompt/terminal and change directories so that you are in the **project** directory of this course. You should see several python files, some folders, and a requirements.txt file containing a list of all the necessary packages.

```
→  project ls
assets           models.py        update.py
dashboard.py     prepare.py       update.sh
data             requirements.txt wsgi.py
```

Creating a virtual environment

It is recommended that you create a virtual environment at this point dedicated to this project. Within this virtual environment, you will install all the necessary packages needed to launch the dashboard. Take the following steps to create the virtual environment:

1. Open your terminal/command prompt and activate your current Python environment. This may not be necessary if your environment is automatically active or if the `python` command maps to the version of Python you wish to run in the virtual environment.
2. `cd` into the `project` directory
3. Run the command **`python -m venv dashboard_venv`** to create the virtual environment. A new folder called `dashboard_venv` will be created.
4. Activate this virtual environment using one of the following commands. Afterwards, you'll see "(dashboard_venv)" prepended to your prompt.
   - macOS and Linux users - run the command **`source dashboard_venv/bin/activate`**
   - Windows users - run the command **`dashboard_venv\Scripts\activate.bat`**
5. Run **`pip install -U pip`** to upgrade pip to the latest version
6. Run **`pip install wheel`** to install the wheel package, which helps install the other packages
7. Run **`pip install -r requirements.txt`** to install all the necessary packages into this environment. This will take some time to complete.
8. Run **`pip install jupyter matplotlib`** to install Jupyter Notebook and matplotlib into this environment. These libraries are NOT part of the requirements as they are only necessary for the course and not for deployment in production
9. Run **`pip install jupyter-dash==0.4`** to install Jupyter-Dash, a library for creating dashboards with Dash within a Jupyter Notebook. It too is only required for the course and not deployment.

1.5 Do not upgrade the packages

As time moves on, there will be newer versions of the packages installed containing more features. Do not upgrade to these packages within this virtual environment or your dashboard may not work. It is tempting to use the latest and greatest versions of the libraries, but there is no guarantee that the dashboard code will work with them. At the time of this writing, Dash 2.0 was just released, but is incompatible with the code for the dashboard. If you wish to upgrade libraries, then make a copy of all the code into a separate location in your file system and create a new virtual environment with the updated libraries.

1.6 Launching the dashboard

After creating the environment, and while still in the project directory, remain on the command line and run the following:

- `python dashboard.py`

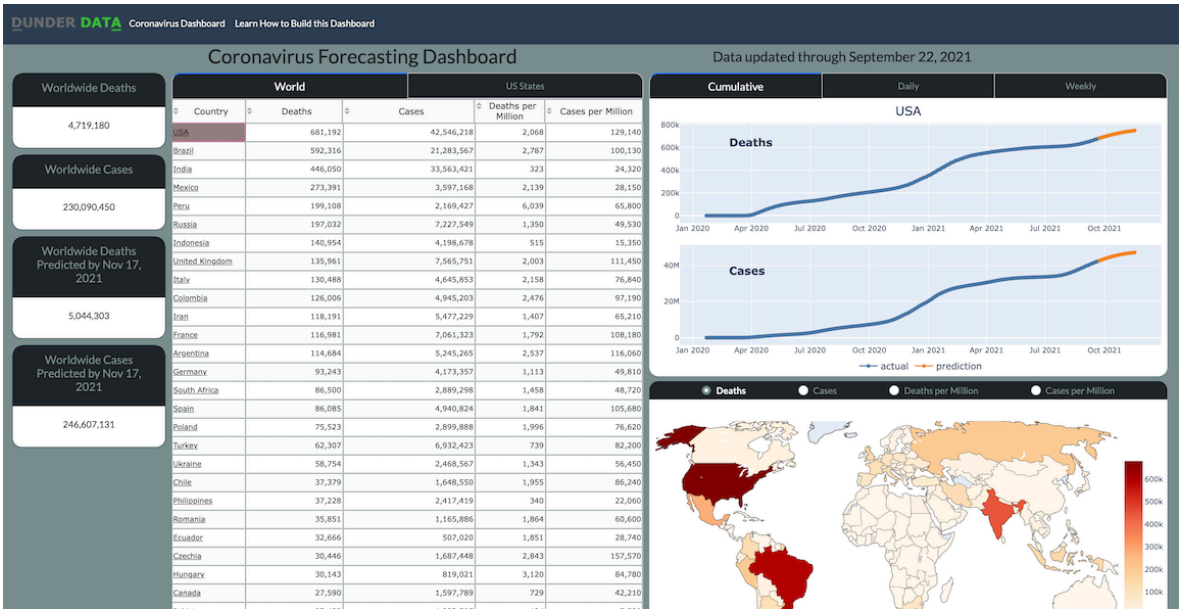
Right after running this command, you'll get a message informing you that the Dash app is running

at http://127.0.0.1:8050. This will be followed by a message warning you not to use this method in production. Deployment on a production server will be covered at the end of this course. We'll use this simple method to run our app until then.

```
(dashboard_venv) ➜  project python dashboard.py
Dash is running on http://127.0.0.1:8050/

 * Serving Flask app "dashboard" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
```

Viewing the dashboard

Navigate to http://127.0.0.1:8050 in your browser (most terminals allow you to open web pages via ctrl + click) to see the dashboard.



Dashboard features

The main purpose of this dashboard is to show historical and future predicted deaths and cases of the ongoing coronavirus pandemic for all countries and US states. The dashboard is set to show its "current" data at the date the last time this course was updated. In the future, you'll learn how to update the dashboard with new data. The dashboard is interactive and meant for the user to explore different views by clicking on different text, tabs, and radio buttons. There are three major components of the dashboard:

- Table of data on the left
- Line and bar graphs of both deaths and cases in the upper right
- Maps of the world and US states in the lower right

There are two separate views, **World** and **US States**, offered by the dashboard as tabs above the data table on the left. The world view is selected by default and shows data for all countries in the world. Switching to US States shows all 50 US States plus its territories.

The data table on the left is sorted by deaths by default. You should notice that each area is underlined, indicating to the user that it is clickable, with the top area selected in red. The three graphs in the upper right correspond to the selected area in the table. All three graphs show the same data, actual and predicted deaths/cases as cumulative (default), daily, and weekly totals. The map on the bottom right colors each area with respect to the absolute or per capita deaths/cases. Explore the dashboard now by clicking on all of the different views to verify it is working properly.

Stopping the dashboard

To stop the dashboard, go back to the command line and press `ctrl + C`, which will exit the program and bring up a new prompt.

Deactivate and reactivate the environment in the future

If you created a virtual environment, you can deactivate it now by running the **`deactivate`** command. In the future, when you want to run this dashboard again, you'll need to reactivate the environment by running **`source dashboard_venv/bin/activate`** or **`dashboard_venv\Scripts\activate.bat`** from the project directory. Then you can run **`python dashboard.py`** to launch the dashboard.

1.7 Jupyter Notebooks

Jupyter Notebooks contain the written text, code examples, and exercises for this course. They are found in the **notebooks** directory. The notebooks provide a great way to learn while going through the material as you can execute and modify the code as well as answer the exercises.

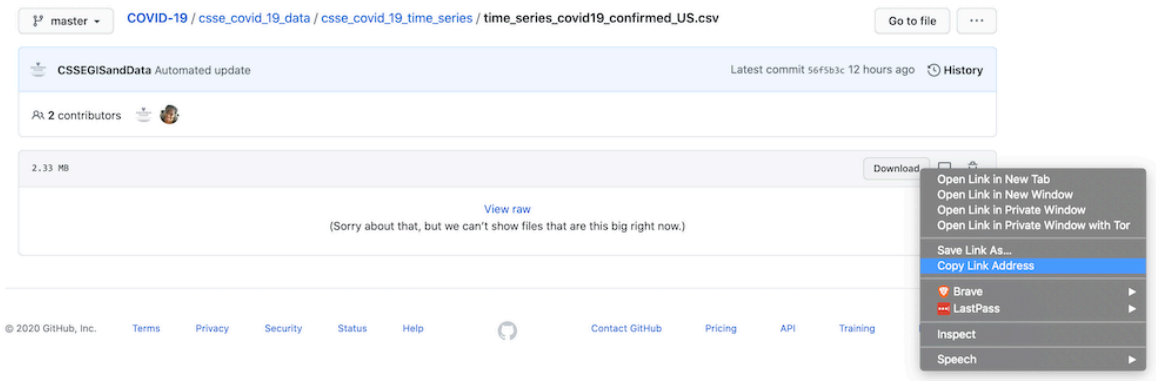
Chapter 2

Getting the Data

Our data comes directly from the John Hopkins COVID-19 GitHub repository, which tracks all deaths and cases from each country in the world as well as many regions within some countries. All of the data needed for this project is within the time series directory, which contains four CSV files that summarize the deaths and cases for the world and the USA. The repository uses the word "confirmed" to refer to cases.

2.1 Reading in the data into a pandas DataFrame

The pandas `read_csv` function can read in remote CSV files by passing it the URL. The exact URL on GitHub is a bit tricky. You must use the "raw" data file, which can be retrieved by clicking on the file name (taking you to the next page), then right-clicking the "view raw" or "download" button and copying the link. The image below shows the screen you'll see for the first CSV.



2.2 Naming conventions

Before we write any code, let's cover some naming conventions that we will use throughout the project.

`group`

We will use the name `group` to refer to the two separate "groups" of data.

- `"world"` - represents all data from each country

- `"usa"` - represents all data from each US state

`kind`

We will use the name `kind` to refer to the two different kinds of COVID-19 data.

- `"deaths"`
- `"cases"`

`area`

Occasionally, we will refer to either a specific country or state with the name `area`.

2.3 Downloading the data

Now that we have the URL, we can download the data with pandas. Below, we write a function to download all four files as DataFrames. This function accepts the `kind` and `group` (as one of the possible strings above) and maps that value to a different string used in the GitHub repository URL. This value is then placed in the `DOWNLOAD_URL` string in its respective location. Finally, this URL is used to download the CSV as a DataFrame.

[1]:
```
import pandas as pd

DOWNLOAD_URL = (
    "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/"
    "master/csse_covid_19_data/csse_covid_19_time_series/"
    "time_series_covid19_{kind}_{group}.csv"
)


def download_data(group, kind):
    """
    Reads in a single dataset from the John Hopkins GitHub repo
    as a DataFrame

    Parameters
    ----------
    group : "world" or "usa"

    kind : "deaths" or "cases"

    Returns
    -------
    DataFrame
    """
    group_change_dict = {"world": "global", "usa": "US"}
    kind_change_dict = {"deaths": "deaths", "cases": "confirmed"}
    group = group_change_dict[group]
    kind = kind_change_dict[kind]
    return pd.read_csv(DOWNLOAD_URL.format(kind=kind, group=group))
```

Verifying the `download_data` function

Let's read in the world deaths file as a DataFrame and output the head to verify that it works.

[2]:
```
df_world_deaths = download_data('world', 'deaths')
df_world_deaths.head()
```

	Province/State	Country/Region	Lat	Long	1/22/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	NaN	Afghanistan	33.93911	67.709953	0	...	7295	7297	7297	7361	7363
1	NaN	Albania	41.15330	20.168300	0	...	3004	3014	3022	3029	3035
2	NaN	Algeria	28.03390	1.659600	0	...	5997	6005	6009	6015	6017
3	NaN	Andorra	42.50630	1.521800	0	...	130	130	130	130	130
4	NaN	Angola	-11.20270	17.873900	0	...	1729	1729	1729	1729	1730

5 rows × 673 columns

Let's write another function which uses `download_data` to read in all four DataFrames. Below, we write a function that reads in all four CSVs as DataFrames returning them in a dictionary. The the group and kind are concatenated and separated by an underscore as the key (i.e. `"world_deaths"`). The `GROUPS` and `KINDS` variables are tuples to store the two possible values of each.

[3]:
```
GROUPS = "world", "usa"
KINDS = "deaths", "cases"

def read_all_data():
    """
    Read in all four CSVs as DataFrames

    Returns
    -------
    Dictionary of DataFrames
    """
    data = {}
    for group in GROUPS:
        for kind in KINDS:
            df = download_data(group, kind)
            data[f"{group}_{kind}"] = df
    return data
```

Let's use this function to read in all of the data and output the head of two of them.

[4]:
```
data = read_all_data()
data['world_cases'].head(3)
```

	Province/State	Country/Region	Lat	Long	1/22/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	NaN	Afghanistan	33.93911	67.709953	0	...	156649	156739	156739	156812	156864
1	NaN	Albania	41.15330	20.168300	0	...	193856	194472	195021	195523	195988
2	NaN	Algeria	28.03390	1.659600	0	...	208245	208380	208532	208695	208839

3 rows × 673 columns

[5]:
```
data['usa_cases'].head(3)
```

	UID	iso2	iso3	code3	FIPS	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	84001001	US	USA	840	1001.0	...	10419	10423	10439	10457	10468
1	84001003	US	USA	840	1003.0	...	37891	37914	37940	37959	37981
2	84001005	US	USA	840	1005.0	...	3653	3655	3659	3660	3683

3 rows × 680 columns

2.4 Saving the data locally

Since the raw data must be downloaded from the internet, let's save a copy of our current data to a local folder so that we have access to it immediately at any time. The function below accepts a dictionary of DataFrames and a directory name, and writes them to that directory as CSVs using the key as the filename. Any extra keyword arguments are collected by `kwargs` and passed to the DataFrame `to_csv` method.

[6]:
```
def write_data(data, directory, **kwargs):
    """
    Writes each raw data DataFrame to a file as a CSV

    Parameters
    ----------
    data : dictionary of DataFrames

    directory : string name of directory to save files i.e. "data/raw"

    kwargs : extra keyword arguments for the `to_csv` DataFrame method

    Returns
    -------
    None
    """
    for name, df in data.items():
        df.to_csv(f"{directory}/{name}.csv", **kwargs)
```

Let's write those DataFrames as CSVs (without their index) to the "data/raw" directory.

[7]:
```
write_data(data, "data/raw", index=False)
```

Let's write a function similar to `download_data`, but have it read in the local data that we just saved.

[8]:
```
def read_local_data(group, kind, directory):
    """
    Read in one CSV as a DataFrame from the given directory

    Parameters
    ----------
    group : "world" or "usa"

    kind : "deaths" or "cases"
```

```
    directory : string name of directory to save files i.e. "data/raw"

    Returns
    -------
    DataFrame
    """
    return pd.read_csv(f"{directory}/{group}_{kind}.csv")
```

Here, we use our function to read in the local data that we just saved.

[9]:
```
read_local_data('world', 'deaths', 'data/raw').head(3)
```

	Province/State	Country/Region	Lat	Long	1/22/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	NaN	Afghanistan	33.93911	67.709953	0	...	7295	7297	7297	7361	7363
1	NaN	Albania	41.15330	20.168300	0	...	3004	3014	3022	3029	3035
2	NaN	Algeria	28.03390	1.659600	0	...	5997	6005	6009	6015	6017

3 rows × 673 columns

Here, we write a function that uses `read_all_data` to read in each of the local CSVs that we just saved. It returns a dictionary of all four DataFrames. The function name is `run` since we will be slowly adding all of our data cleaning and transformation steps to it in the next chapter.

[10]:
```
def run():
    """
    Run all cleaning and transformation steps

    Returns
    -------
    Dictionary of DataFrames
    """
    data = {}
    for group in GROUPS:
        for kind in KINDS:
            df = read_local_data(group, kind, "data/raw")
            data[f"{group}_{kind}"] = df
    return data
```

Here, we verify that `run` works properly.

[11]:
```
data = run()
data['usa_deaths'].tail(3)
```

	UID	iso2	iso3	code3	FIPS	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
3339	84090056	US	USA	840	90056.0	...	0	0	0	0	0
3340	84056043	US	USA	840	56043.0	...	35	35	35	35	35
3341	84056045	US	USA	840	56045.0	...	13	13	13	13	13

3 rows × 681 columns

This concludes the section on downloading the data.

Chapter 3

Data Cleaning and Transformation

In the last chapter, we created several functions to download and save the raw data. In this chapter, we take steps to find and clean bad data, and transform it to a structure that is suitable for modeling. We begin by reading in the raw local data with the `run` function.

Important note on importing from solutions.py

Throughout this chapter and the rest of the book, you'll see statements that import specific functions from the solutions module. In the version of this book contained within Jupyter Notebooks, all of these important functions are kept in the `solutions.py` module. Thus, when you see the import statement, assume a function defined in a previous chapter is being imported into our namespace to be used in this chapter. Now, let's read in all of our local data.

[1]:
```
import pandas as pd
from solutions import run, read_local_data
GROUPS = "world", "usa"
KINDS = "deaths", "cases"
data = run()
data['world_cases'].head(3)
```

	Province/State	Country/Region	Lat	Long	1/22/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	NaN	Afghanistan	33.93911	67.709953	0	...	156649	156739	156739	156812	156864
1	NaN	Albania	41.15330	20.168300	0	...	193856	194472	195021	195523	195988
2	NaN	Algeria	28.03390	1.659600	0	...	208245	208380	208532	208695	208839

3 rows × 673 columns

[2]:
```
data['usa_cases'].head(3)
```

	UID	iso2	iso3	code3	FIPS	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	84001001	US	USA	840	1001.0	...	10419	10423	10439	10457	10468
1	84001003	US	USA	840	1003.0	...	37891	37914	37940	37959	37981
2	84001005	US	USA	840	1005.0	...	3653	3655	3659	3660	3683

3 rows × 680 columns

3.1 Selecting the correct columns

Take a look at the world and USA DataFrames above and you'll notice a difference in the names and number of columns. Below, we write a function that accepts a single DataFrame and selects the `"Country/Region"` column for the world DataFrames, `"Province_State"` column for the USA DataFrames, and all the date columns for both. Date columns are identified because they always contain two forward slashes in them. It returns a DataFrame with just those columns.

[3]:
```
def select_columns(df):
    """
    Selects the "Country/Region" column for world DataFrames and
    "Province_State" for USA along with all of the date columns

    Parameters
    ----------
    df : DataFrame

    Returns
    -------
    df : DataFrame
    """
    cols = df.columns

    # we don't need to know the group since the world and usa have
    # different column names for their areas
    areas = ["Country/Region", "Province_State"]
    is_area = cols.isin(areas)

    # date columns are the only ones with two slashes
    has_two_slashes = cols.str.count("/") == 2
    filt = is_area | has_two_slashes
    return df.loc[:, filt]
```

Let's use this function to select the needed columns from both the world and USA DataFrames.

[4]:
```
select_columns(data['world_cases']).head(3)
```

	Country/Region	1/22/20	1/23/20	1/24/20	1/25/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	Afghanistan	0	0	0	0	...	156649	156739	156739	156812	156864
1	Albania	0	0	0	0	...	193856	194472	195021	195523	195988
2	Algeria	0	0	0	0	...	208245	208380	208532	208695	208839

3 rows × 670 columns

[5]:
```
select_columns(data['usa_cases']).head(3)
```

	Province_State	1/22/20	1/23/20	1/24/20	1/25/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	Alabama	0	0	0	0	...	10419	10423	10439	10457	10468
1	Alabama	0	0	0	0	...	37891	37914	37940	37959	37981
2	Alabama	0	0	0	0	...	3653	3655	3659	3660	3683

3 rows × 670 columns

Updating the run function

After each step in this chapter, we'll update our `run` function to pass each DataFrame through the newly created function. Each `run` function will be uniquely labeled with an ending integer. This `run` function includes the step above.

[6]:
```
def run2():
    """
    Run all cleaning and transformation steps

    Returns
    -------
    Dictionary of DataFrames
    """
    data = {}
    for group in GROUPS:
        for kind in KINDS:
            df = read_local_data(group, kind, "data/raw")
            df = select_columns(df)
            data[f"{group}_{kind}"] = df
    return data
```

Let's test it and run all of our steps thus far.

[7]:
```
data = run2()
data['usa_cases'].head(3)
```

	Province_State	1/22/20	1/23/20	1/24/20	1/25/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
0	Alabama	0	0	0	0	...	10419	10423	10439	10457	10468
1	Alabama	0	0	0	0	...	37891	37914	37940	37959	37981
2	Alabama	0	0	0	0	...	3653	3655	3659	3660	3683

3 rows × 670 columns

3.2 Updating area names

In both groups of data, there are a few area names that can be updated so that they use a more common name. There are three cruise ships, which we will replace with the string "Cruise Ship". The function below uses the DataFrame `replace` method to replace the names in the first column with the provided dictionary below. Also, the US data in the world data is filtered out because we already have a separate table of data for it. We will add the US back to the world data once modeling is complete.

[8]:
```
REPLACE_AREA = {
    "Korea, South": "South Korea",
    "Taiwan*": "Taiwan",
    "Burma": "Myanmar",
    "Holy See": "Vatican City",
    "Diamond Princess": "Cruise Ship",
    "Grand Princess": "Cruise Ship",
    "MS Zaandam": "Cruise Ship"
```

```
}

def update_areas(df):
    """
    Replace a few of the area names using the REPLACE_AREA dictionary.

    Parameters
    ----------
    df : DataFrame

    Returns
    -------
    df : DataFrame
    """
    area_col = df.columns[0]
    df[area_col] = df[area_col].replace(REPLACE_AREA)
    df = df[df[area_col] != "US"]
    return df
```

Let's update the `run` function to include the above step.

[9]:
```
def run3():
    """
    Run all cleaning and transformation steps

    Returns
    -------
    Dictionary of DataFrames
    """
    data = {}
    for group in GROUPS:
        for kind in KINDS:
            df = read_local_data(group, kind, "data/raw")
            df = select_columns(df)
            df = update_areas(df)
            data[f"{group}_{kind}"] = df
    return data
```

We verify our function works by searching for the cruise ships.

[10]:
```
data = run3()
data['usa_cases'].query("Province_State == 'Cruise Ship'")
```

	Province_State	1/22/20	1/23/20	1/24/20	1/25/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
338	Cruise Ship	0	0	0	0	...	49	49	49	49	49
572	Cruise Ship	0	0	0	0	...	103	103	103	103	103

2 rows × 670 columns

3.3 Aggregate repeating areas

In each DataFrame, many areas repeat multiple times as the raw data tracks deaths/cases by the province/state/county level. We desire a single row for each unique area. We write a function that accepts a single DataFrame, groups by the area column (first column in each DataFrame), and sums up the counts for each date column.

[11]:
```
def group_area(df):
    """
    Gets a single total for each area

    Parameters
    ----------
    df : DataFrame

    Returns
    -------
    df : DataFrame
    """
    grouping_col = df.columns[0]
    return df.groupby(grouping_col).sum()
```

Again, we update the `run` function to include this step.

[12]:
```
def run4():
    """
    Run all cleaning and transformation steps

    Returns
    -------
    Dictionary of DataFrames
    """
    data = {}
    for group in GROUPS:
        for kind in KINDS:
            df = read_local_data(group, kind, "data/raw")
            df = select_columns(df)
            df = update_areas(df)
            df = group_area(df)
            data[f"{group}_{kind}"] = df
    return data
```

Let's run all of our steps. Notice that the area column is now in the index, which is where pandas places it after grouping.

[13]:
```
data = run4()
data['usa_cases'].head(3)
```

Province_State	1/22/20	1/23/20	1/24/20	1/25/20	1/26/20	...	11/16/21	11/17/21	11/18/21	11/19/21	11/20/21
Alabama	0	0	0	0	0	...	840495	840991	841483	841956	842636
Alaska	0	0	0	0	0	...	146598	147040	147560	148083	148083
American Samoa	0	0	0	0	0	...	4	4	4	4	4

3 rows × 669 columns

3.4 Transposing the data to time series

We have time series data (a sequence of data over time), but it's not in the customary format where date is along the vertical axis. We write a function that accepts a single DataFrame and transposes it so that the current date columns become the index, making sure to convert the dates to a datetime data type (they are currently strings).

[14]:
```
def transpose_to_ts(df):
    """
    Transposes the DataFrame and converts the index to datetime

    Parameters
    ----------
    df : DataFrame

    Returns
    -------
    df : DataFrame
    """
    df = df.T
    df.index = pd.to_datetime(df.index)
    return df
```

We update the `run` function and output the current status of one of our DataFrames.

[15]:
```
def run5():
    """
    Run all cleaning and transformation steps

    Returns
    -------
    Dictionary of DataFrames
    """
    data = {}
    for group in GROUPS:
        for kind in KINDS:
            df = read_local_data(group, kind, "data/raw")
            df = select_columns(df)
            df = update_areas(df)
            df = group_area(df)
            df = transpose_to_ts(df)
```

```
            data[f"{group}_{kind}"] = df
    return data
```

[16]:
```
data = run5()
data['usa_cases'].tail()
```

Province_State	Alabama	Alaska	American Samoa	Arizona	Arkansas	...	Virginia	Washington	West Virginia	Wisconsin	Wyoming
2021-11-16	840495	146598	4	1220433	520725	...	947271	756310	284165	932930	108413
2021-11-17	840991	147040	4	1223892	521553	...	949803	758483	285135	937212	108658
2021-11-18	841483	147560	4	1228076	522460	...	951698	760482	286249	941550	109083
2021-11-19	841956	148083	4	1233146	523192	...	953460	762118	287612	945715	109318
2021-11-20	842636	148083	4	1238249	523866	...	953460	762118	288502	945715	109318

5 rows × 57 columns

3.5 Finding and handling bad data

In this section, we will search for bad data and then come up with a solution for handling it. Our DataFrames contain the cumulative count of deaths and cases at each date. These values should never decrease. In order to verify that the values never decrease, we can test whether each day's value is at least as large as all the values preceding it. To do this, we call the `cummax` method which returns the cumulative maximum of each column up to each date. We then compare each value with this cumulative maximum. We'll work with just the world deaths DataFrame for now.

[17]:
```
world_deaths = data['world_deaths']
bad_data = world_deaths < world_deaths.cummax()
bad_data.tail(3)
```

Country/Region	Afghanistan	Albania	Algeria	Andorra	Angola	...	Vietnam	West Bank and Gaza	Yemen	Zambia	Zimbabwe
2021-11-18	False	False	False	False	False	...	False	False	False	False	False
2021-11-19	False	False	False	False	False	...	False	False	False	False	False
2021-11-20	False	False	False	False	False	...	False	False	False	False	False

3 rows × 194 columns

If any of these values are `True`, then we've found bad data. Let's sum each column and sort the results to see which columns have the most bad data.

[18]:
```
bad_data.sum().sort_values(ascending=False).head(10)
```

[18]:
```
Country/Region
Kyrgyzstan                221
Monaco                    128
Congo (Brazzaville)       127
China                     123
Sao Tome and Principe     103
Spain                      86
Estonia                    79
Finland                    38
Czechia                    34
Belgium                    34
dtype: int64
```

Let's locate the bad data for Spain, and see if we can find out what's happening.

[19]:
```
spain_bad = bad_data['Spain']
spain_bad[spain_bad].head()
```

[19]:
```
2020-05-25    True
2020-05-26    True
2020-05-27    True
2020-05-28    True
2020-05-29    True
Name: Spain, dtype: bool
```

Let's inspect a small subset of the data around the first date of bad data.

[20]:
```
world_deaths.loc['2020-05-21':'2020-05-26', 'Spain']
```

[20]:
```
2020-05-21    27940
2020-05-22    28628
2020-05-23    28678
2020-05-24    28752
2020-05-25    26834
2020-05-26    27117
Name: Spain, dtype: int64
```

A drop of nearly 2,000 deaths appears on May 25th. Let's make a plot of Spain's total deaths beginning from the beginning of May to get a better picture of what is happening.

[21]:
```
import matplotlib.pyplot as plt
plt.style.use('dashboard.mplstyle')
world_deaths.loc['2020-05-01':'2020-06-30', 'Spain'].plot();
```

It appears that almost no new deaths are reported after the sudden decrease on May 25th until a huge increase in the latter half of June, followed again by a period of very few deaths. Various other data aggregators have reported similar issues with Spain's data.

We'll provide a simple solution so that all dates have a value greater than or equal to the prior day's values. In order to make this replacement, we'll change all the values for dates below the current maximum to missing values with the `mask` method. First, we create a boolean mask, a Series of booleans with the same length as the original Series that meet some criteria.

[22]:
```
spain = world_deaths['Spain']
mask = spain < spain.cummax()
mask.tail()
```

[22]:
```
2021-11-16    False
2021-11-17    False
2021-11-18    False
2021-11-19    False
2021-11-20    False
Name: Spain, dtype: bool
```

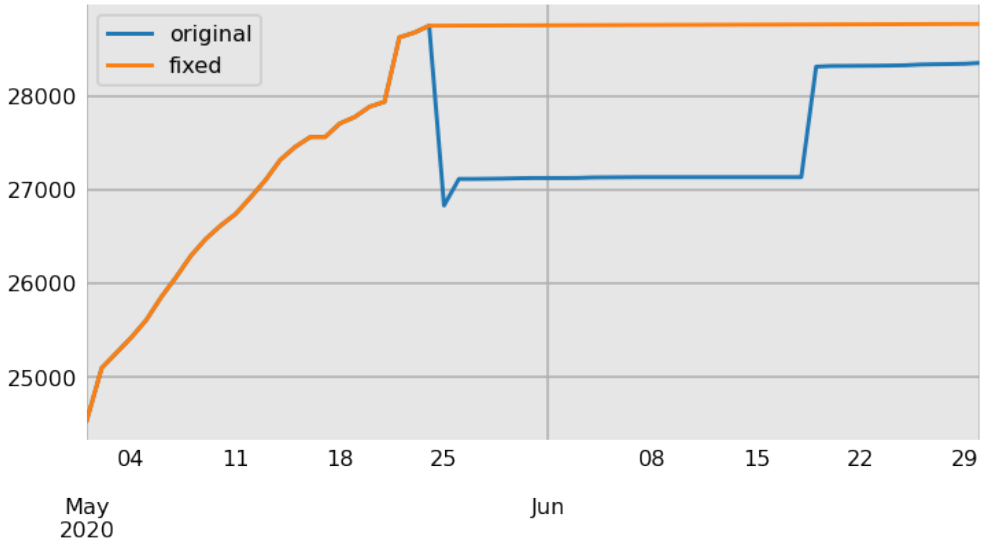
We pass this Series to the `mask` method to "mask" them - cover them up and replace them with missing values. We show the first 10 dates where the data is now missing.

[23]:
```
spain_masked = spain.mask(mask)
spain_masked[spain_masked.isna()].head(10)
```

[23]:
```
2020-05-25   NaN
2020-05-26   NaN
2020-05-27   NaN
2020-05-28   NaN
2020-05-29   NaN
2020-05-30   NaN
2020-05-31   NaN
2020-06-01   NaN
2020-06-02   NaN
2020-06-03   NaN
Name: Spain, dtype: float64
```

We can then linearly interpolate the missing values with the `interpolate` method and plot the updated data.

[24]:
```
orig = world_deaths.loc['2020-05-01':'2020-06-30', 'Spain']
fixed = spain_masked.interpolate().loc['2020-05-01':'2020-06-30']
orig.plot(label='original', legend=True)
fixed.plot(label='fixed', legend=True);
```



This "fixes" the data such that each value is always at least as large as the preceding value. In this particular example, this simple fix doesn't seem to connect the points in a way pleasing to the eye. A better estimation might linearly interpolate from the middle of May to the middle of July.

Instead of developing a more complex method to fix bad data, we'll use this simple method and complete a process called **smoothing** later on, which will really help out the model handle these uneven jumps in the data.

Fixing all bad data

Let's fix all of the bad data in our DataFrame with the same logic from above, rounding the totals to whole numbers.

[25]:
```
mask = world_deaths < world_deaths.cummax()
world_deaths_fixed = world_deaths.mask(mask).interpolate().round(0).astype('int64')
world_deaths_fixed.tail(3)
```

Country/Region	Afghanistan	Albania	Algeria	Andorra	Angola	...	Vietnam	West Bank and Gaza	Yemen	Zambia	Zimbabwe
2021-11-18	7297	3022	6009	130	1729	...	23476	4767	1934	3666	4699
2021-11-19	7361	3029	6015	130	1729	...	23578	4770	1935	3667	4699
2021-11-20	7363	3035	6017	130	1730	...	23685	4770	1938	3667	4699

3 rows × 194 columns

Let's verify that all values are at least as large as the previous day's value.

[26]:
```
mask = world_deaths_fixed < world_deaths_fixed.cummax()
mask.sum().sum()
```

[26]: 0

We write a function that accepts a single DataFrame and fixes all of the bad data within it. We also update the `run` function.

[27]:
```
def fix_bad_data(df):
    """
    Replaces all days for each country where the value of
    deaths/cases is lower than the current maximum

    Parameters
    ----------
    df : DataFrame

    Returns
    -------
    DataFrame
    """
    mask = df < df.cummax()
    df = df.mask(mask).interpolate().round(0).astype("int64")
    return df


def run6():
    """
    Run all cleaning and transformation steps

    Returns
    -------
    Dictionary of DataFrames
    """
    data = {}
    for group in GROUPS:
        for kind in KINDS:
            df = read_local_data(group, kind, "data/raw")
            df = select_columns(df)
            df = update_areas(df)
            df = group_area(df)
            df = transpose_to_ts(df)
            df = fix_bad_data(df)
            data[f"{group}_{kind}"] = df
    return data
```

Let's verify that this has worked on Spain.

[28]:
```
data = run6()
data['world_deaths'].loc['2020-05-25':'2020-06-02', 'Spain']
```

[28]:
```
2020-05-25    28753
2020-05-26    28753
2020-05-27    28754
2020-05-28    28754
2020-05-29    28755
2020-05-30    28755
```

```
2020-05-31    28756
2020-06-01    28756
2020-06-02    28757
Name: Spain, dtype: int64
```

3.6 Data preparation complete

These steps complete the data preparation process. Let's use one of our previous functions to write this prepared data to the `data/prepared` folder.

[29]:
```
from solutions import write_data
write_data(data, 'data/prepared', index=True, index_label='date')
```

3.7 Encapsulate all steps into a single class

All of the steps in the last two chapters may be encapsulated into a single class. Here, we define a class that has a method for each of the steps from the last two chapters. We add a `run` method that runs all of the steps and returns the dictionary of DataFrames. When initializing a new instance, the `download_new` boolean parameter allows the user to decide whether to download new data from the online repository or read in the local data.

[30]:
```
DOWNLOAD_URL = (
    "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/"
    "master/csse_covid_19_data/csse_covid_19_time_series/"
    "time_series_covid19_{kind}_{group}.csv"
)
REPLACE_AREA = {
    "Korea, South": "South Korea",
    "Taiwan*": "Taiwan",
    "Burma": "Myanmar",
    "Holy See": "Vatican City",
    "Diamond Princess": "Cruise Ship",
    "Grand Princess": "Cruise Ship",
    "MS Zaandam": "Cruise Ship",
}
GROUPS = "world", "usa"
KINDS = "deaths", "cases"


class PrepareData:
    """
    Downloads the data from the John Hopkins repository and applies several
    successive transformations to prepare it for modeling. The `run`
    method calls all the steps
    """

    def __init__(self, download_new=True):
        """
```

```
        Parameters
        ----------
        download_new : bool, determines whether new data will be downloaded
            or whether local saved data will be used
        """
        self.download_new = download_new

    def download_data(self, group, kind):
        """
        Reads in a single dataset from the John Hopkins GitHub repo
        as a DataFrame

        Parameters
        ----------
        group : "world" or "usa"

        kind : "deaths" or "cases"

        Returns
        -------
        DataFrame
        """
        group_change_dict = {"world": "global", "usa": "US"}
        kind_change_dict = {"deaths": "deaths", "cases": "confirmed"}
        group = group_change_dict[group]
        kind = kind_change_dict[kind]
        df = pd.read_csv(DOWNLOAD_URL.format(kind=kind, group=group))
        return df

    def write_data(self, data, directory, **kwargs):
        """
        Writes each raw data DataFrame to a file as a CSV

        Parameters
        ----------
        data : dictionary of DataFrames

        directory : string name of directory to save files i.e. "data/raw"

        Returns
        -------
        None
        """
        for name, df in data.items():
            df.to_csv(f"{directory}/{name}.csv", **kwargs)

    def read_local_data(self, group, kind):
        """
        Read in one CSV as a DataFrame from the data/raw directory
```

```
        Parameters
        ----------
        group : "world" or "usa"

        kind : "deaths" or "cases"

        Returns
        -------
        DataFrame
        """
        name = f"{group}_{kind}"
        return pd.read_csv(f"data/raw/{name}.csv")

    def select_columns(self, df):
        """
        Selects the Country/Region column for world DataFrames and
        Province_State for USA

        Parameters
        ----------
        df : DataFrame

        Returns
        -------
        df : DataFrame
        """
        cols = df.columns

        # we don't need to know the group since the world and usa have
        # different column names for their areas
        areas = ["Country/Region", "Province_State"]
        is_area = cols.isin(areas)

        # date columns are the only ones with two slashes
        has_two_slashes = cols.str.count("/") == 2
        filt = is_area | has_two_slashes
        return df.loc[:, filt]

    def update_areas(self, df):
        """
        Replace a few of the area names using the REPLACE_AREA dictionary.

        Parameters
        ----------
        df : DataFrame

        Returns
        -------
```

```
        df : DataFrame
        """
        area_col = df.columns[0]
        df[area_col] = df[area_col].replace(REPLACE_AREA)
        df = df[df[area_col] != "US"]
        return df

    def group_area(self, df):
        """
        Gets a single total for each area

        Parameters
        ----------
        df : DataFrame

        Returns
        -------
        df : DataFrame
        """
        grouping_col = df.columns[0]
        return df.groupby(grouping_col).sum()

    def transpose_to_ts(self, df):
        """
        Transposes the DataFrame and converts the index to datetime

        Parameters
        ----------
        df : DataFrame

        Returns
        -------
        df : DataFrame
        """
        df = df.T
        df.index = pd.to_datetime(df.index)
        return df

    def fix_bad_data(self, df):
        """
        Replaces all days for each country where the value of
        deaths/cases is lower than the current maximum

        Parameters
        ----------
        df : DataFrame

        Returns
        -------
```

```
        DataFrame
        """
        mask = df < df.cummax()
        df = df.mask(mask).interpolate().round(0).astype("int64")
        return df

    def run(self):
        """
        Run all cleaning and transformation steps

        Returns
        -------
        Dictionary of DataFrames
        """
        data = {}
        for group in GROUPS:
            for kind in KINDS:
                if self.download_new:
                    df = self.download_data(group, kind)
                else:
                    df = self.read_local_data(group, kind)
                df = self.select_columns(df)
                df = self.update_areas(df)
                df = self.group_area(df)
                df = self.transpose_to_ts(df)
                df = self.fix_bad_data(df)
                data[f"{group}_{kind}"] = df
        return data
```

Let's verify that our new class works by instantiating it and calling the `run` method. By default, new data will be downloaded from the online repository.

[31]:
```
prepare_data = PrepareData()
data = prepare_data.run()
data['world_deaths'].tail()
```

Country/Region	Afghanistan	Albania	Algeria	Andorra	Angola	...	Vietnam	West Bank and Gaza	Yemen	Zambia	Zimbabwe
2021-11-16	7295	3004	5997	130	1729	...	23270	4761	1926	3666	4698
2021-11-17	7297	3014	6005	130	1729	...	23337	4764	1929	3666	4699
2021-11-18	7297	3022	6009	130	1729	...	23476	4767	1934	3666	4699
2021-11-19	7361	3029	6015	130	1729	...	23578	4770	1935	3667	4699
2021-11-20	7363	3035	6017	130	1730	...	23685	4770	1938	3667	4699

5 rows × 194 columns

Chapter 4

Data Smoothing

In this chapter, we'll cover various methods for smoothing the data, which is an important step to take before attempting to build a predictive model.

4.1 Smoothing the data

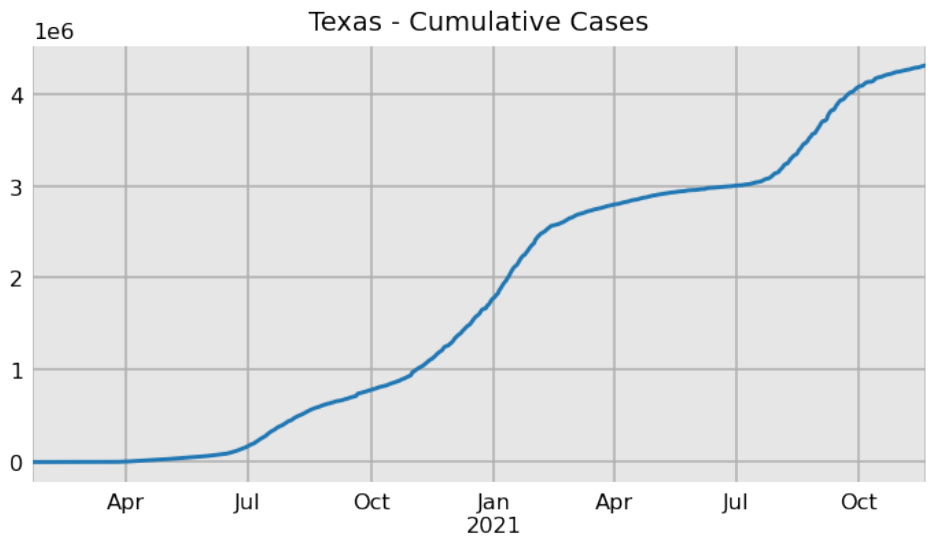
The daily reported data has a tremendous amount of variation due to many factors (weekdays vs weekends, holidays, and other reporting delays). Our goal when modeling is to understand the **general trend** of the data and not the variations due to reporting. Let's instantiate our `PrepareData` class and use it to read in the dictionary of all four DataFrames.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('dashboard.mplstyle')

from prepare import PrepareData
data = PrepareData(download_new=False).run()
```

Smoothing Texas's data

To show how smoothing works, we'll examine the cases in Texas. Let's get an overall view of cumulative cases with a line plot.

[2]:
```
usa_cases = data['usa_cases']
texasc = usa_cases['Texas']
texasc.plot(kind='line', title="Texas - Cumulative Cases");
```



Except for a few small bumps, the graph looks fairly smooth. Let's investigate further and use the `diff` method to find the daily cases. By default, `diff` returns the difference between the current and previous values.

[3]:
```
texasc_daily = texasc.diff()
texasc_daily.head()
```

[3]:
```
2020-01-22    NaN
2020-01-23    0.0
2020-01-24    0.0
2020-01-25    0.0
2020-01-26    0.0
Name: Texas, dtype: float64
```

The first value will always be missing when finding the 1-day difference as there is no data from the previous day. Also, there were no cases for the first several weeks of recorded data in Texas. Let's find the last date where no cases were recorded.

[4]:
```
last_zero_date = texasc[texasc == 0].index[-1]
last_zero_date
```

[4]:
```
Timestamp('2020-03-04 00:00:00')
```

We filter the cumulative data from this date onwards and reassign the data to the same variable name.

[5]:
```
texasc = texasc.loc[last_zero_date:]
texasc.head()
```

[5]:
```
2020-03-04     0
2020-03-05     3
2020-03-06     4
2020-03-07     8
2020-03-08    11
Name: Texas, dtype: int64
```

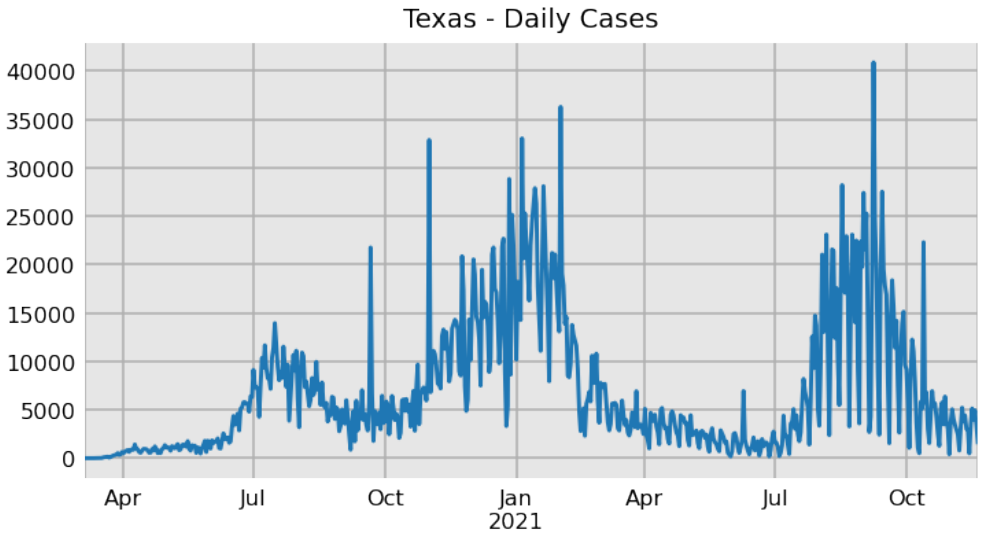
Now, when we use the `diff` method, we can simply drop the first missing value to get only the days from when the first case was recorded.

[6]:
```
texasc_daily = texasc.diff().dropna().astype('int')
texasc_daily.head()
```

[6]:
```
2020-03-05    3
2020-03-06    1
2020-03-07    4
2020-03-08    3
2020-03-09    2
Name: Texas, dtype: int64
```

Plotting the daily cases shows quite a large amount of variation.

[7]:
```
texasc_daily.plot(kind='line', title="Texas - Daily Cases");
```



4.2 Smoothing methods

A wide variety of data smoothing methods exist, three of which we'll cover in detail.

- Moving average
- Weighted smoothing
- Locally weighted scatterplot smoothing (LOWESS)

4.3 Moving average

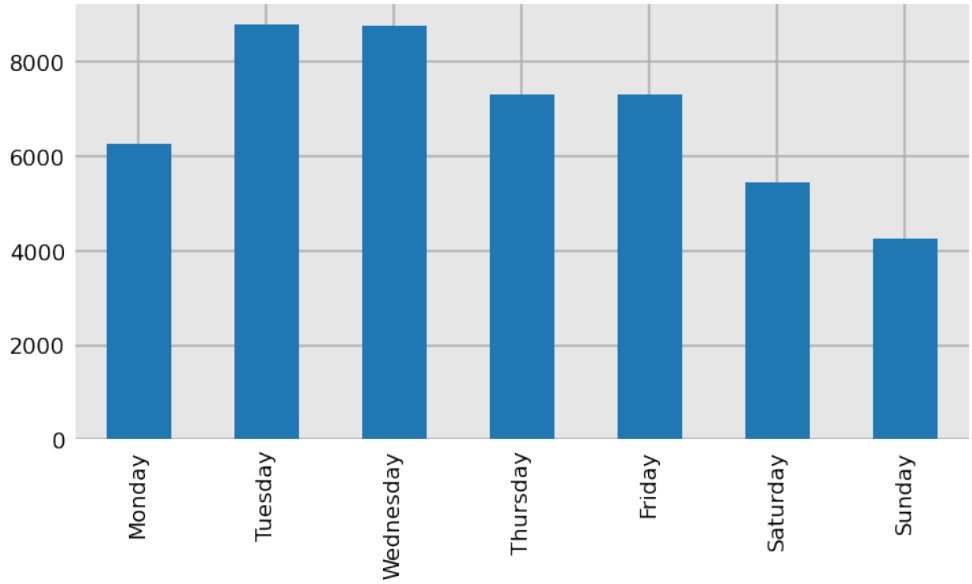
A simple moving average takes a window of points around each point and calculates the average of those points as the new value for that date. The starting and ending window is chosen by the user.

Weekly seasonality

The above graph of daily cases appears like what you would see monitoring a heart beat. There are spikes and dips at regular intervals. The cycles might appear on a weekly basis. **Seasonality** is a term

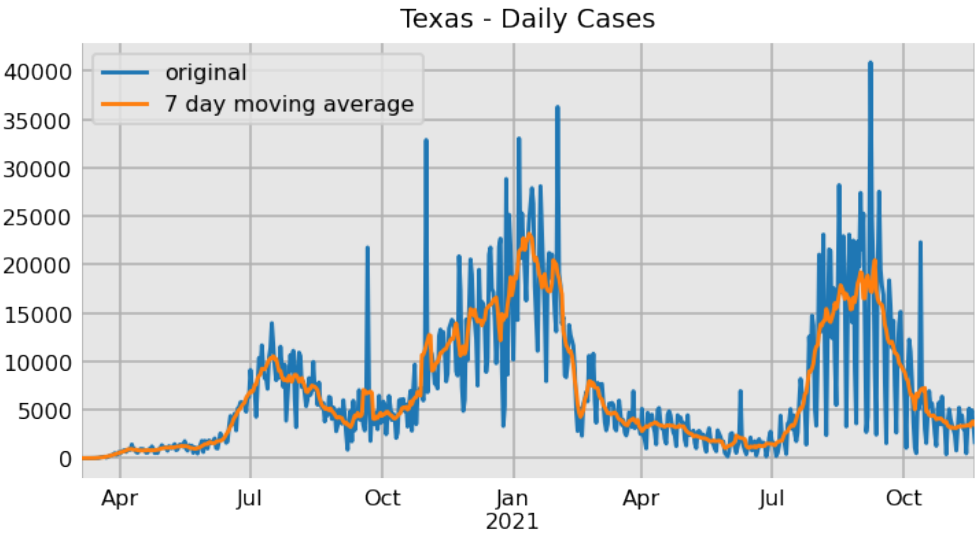
that describes a regular repeating pattern in time series data. Let's get the average cases by day name to see if we can show that weekly seasonality occurs. All days should be roughly equal if there is no seasonality.

[8]:
```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday', 'Sunday']
texasc_daily.groupby(lambda x: x.day_name()).mean().loc[days].plot(kind='bar');
```



As expected, a significant difference in the average between days exists. To account for this weekly seasonality, we will calculate a 7-day moving average. In, pandas, the moving average is calculated with the `rolling` method. Setting `center` to `True` chooses the three days preceding/following the current point and averages them together. Here, we plot the smoothed and original data.

[9]:
```
ma = texasc_daily.rolling(7, min_periods=1, center=True).mean()
texasc_daily.plot(kind='line', title="Texas - Daily Cases", label='original')
ma.plot(kind='line', label='7 day moving average').legend();
```

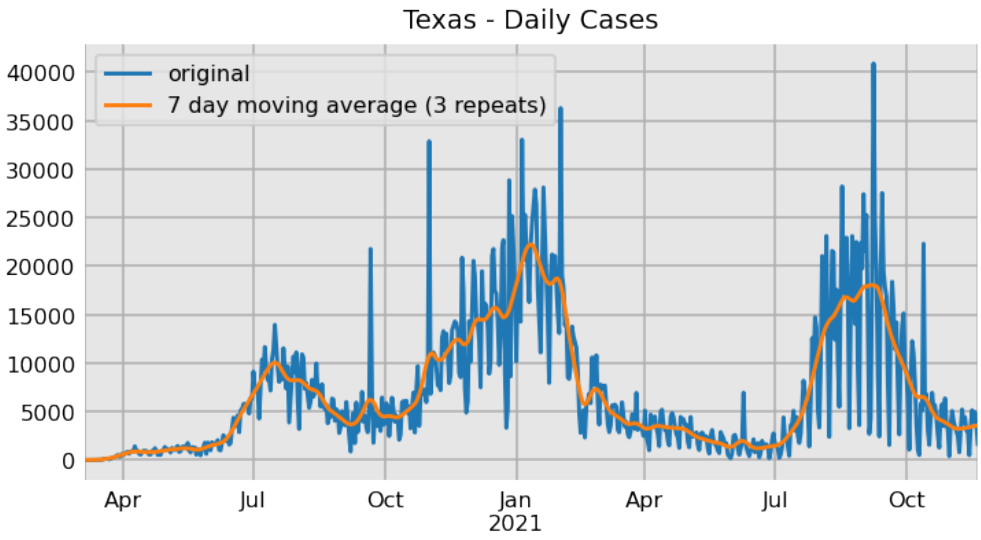


There is substantially less variation, but still not a smooth line like we could draw by hand.

Repeated moving average

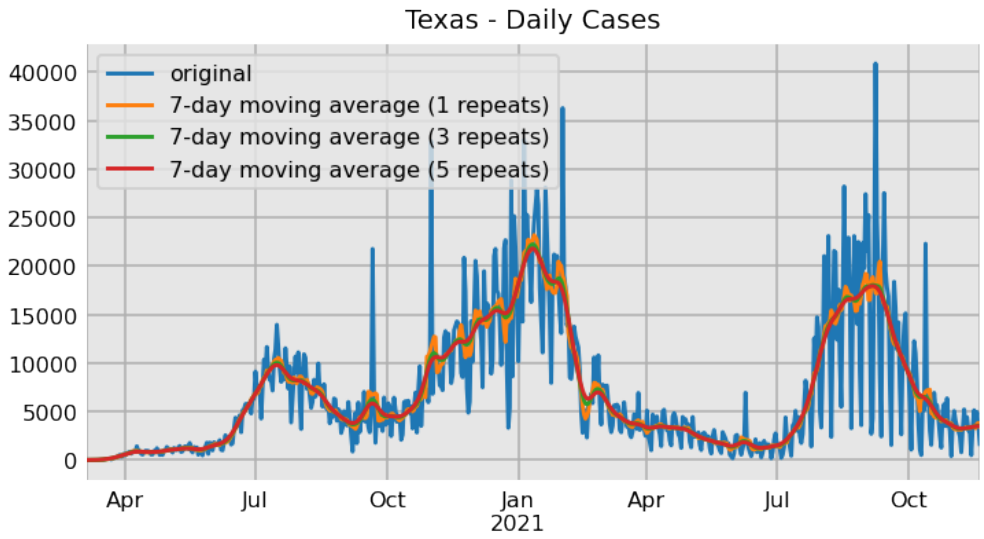
We can take the moving average repeatedly to smooth the data further. Here, we complete three iterations of the 7-day moving average to produce an even smoother plot.

[10]:
```
ma = texasc_daily
for _ in range(3):
    ma = ma.rolling(7, min_periods=1, center=True).mean()

texasc_daily.plot(title="Texas - Daily Cases", label='original')
ma.plot(label='7 day moving average (3 repeats)').legend();
```



Here, we write a loop to plot differing number of repeats of the 7-day moving average.

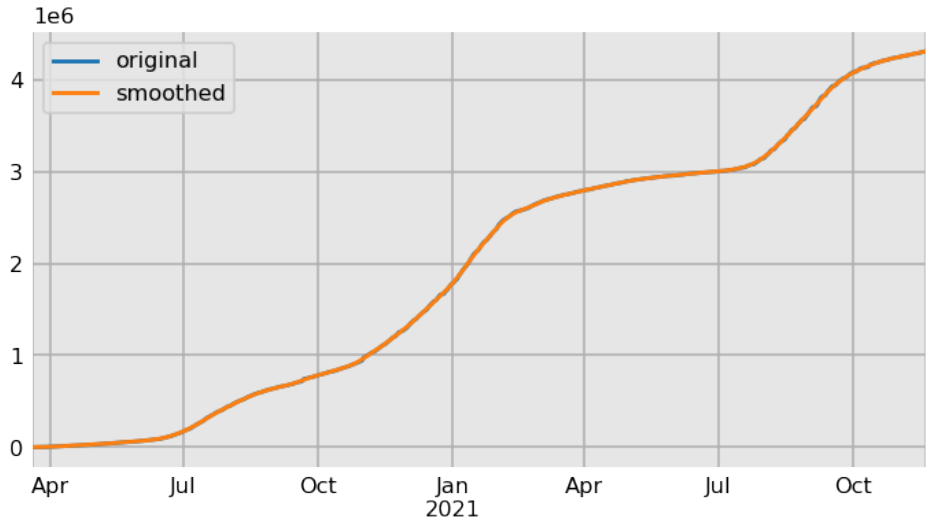
[11]:
```
texasc_daily.plot(title="Texas - Daily Cases", label='original')
repeats_to_plot = [0, 2, 4]
ma = texasc_daily
for i in range(5):
    ma = ma.rolling(7, min_periods=1, center=True).mean()
    if i in repeats_to_plot:
        ma.plot(label=f'7-day moving average ({i + 1} repeats)').legend();
```



Get the smoothed cumulative total

We can take the cumulative sum of the smoothed daily cases to get a smoothed cumulative total. Here, we repeat the smoothing process three times, take its cumulative sum, and plot it against the original.

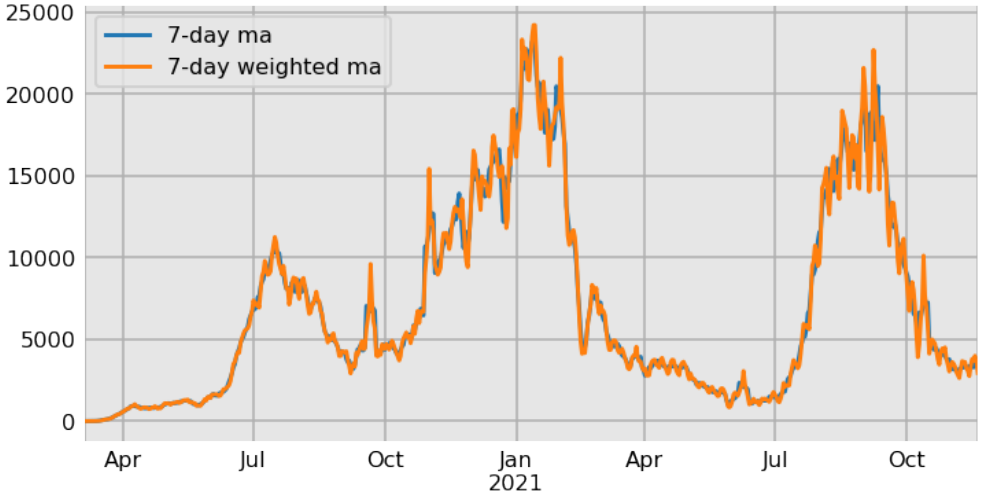
[12]:
```
ma = texasc_daily
for i in range(3):
    ma = ma.rolling(7, min_periods=1, center=True).mean()

texasc_smoothed = ma.cumsum()
texasc.loc['2020-03-20':].plot(kind='line', label='original');
texasc_smoothed.loc['2020-03-20':].plot(label='smoothed').legend();
```



4.4 Weighted smoothing

A slightly different technique involves taking a weighted average of the surrounding points with the points closer to the actual point given more weight. Here, we use a 7-day window and give 30% of the weight to the current observation, 15% each to the two nearest points, 12% each to the next two points, and 8% to the outer two points. This weighted moving average is plotted with the original moving average, so you can see the difference. In pandas, we had to create a custom function with `apply` to calculate the weighted sum.

[13]:
```
weight = np.array([.08, .12, .15, .30, .15, .12, .08])
def weighted_sum(x):
    w = weight[:len(x)]
    return (w * x).sum() / w.sum()

ma = texasc_daily.rolling(7, min_periods=1, center=True).mean()
maw = texasc_daily.rolling(7, min_periods=1, center=True).apply(weighted_sum)
ma.plot(label='7-day ma')
maw.plot(label='7-day weighted ma').legend();
```



Exponentially weighted smoothing

Instead of setting all the weights ourselves, we can use a function that exponentially decreases the weights with respect to their distance from the current point.

$$w_i = \alpha(1-\alpha)^i$$

The above function calculates the weight at each point where i represents the ith previous observation beginning at 0 from the current observation. The parameter α determines the rate at which to discount previous observations. Higher values of α place more importance on the closest observations.

In pandas, we use the `ewm` method, passing it in the value of alpha. Let's output the first 10 observations and then calculate the exponentially weighted mean with pandas.

[14]:
```
texasc_daily.head(10)
```

[14]:
```
2020-03-05     3
2020-03-06     1
2020-03-07     4
2020-03-08     3
2020-03-09     2
2020-03-10     3
2020-03-11     5
2020-03-12     6
2020-03-13    17
2020-03-14    16
Name: Texas, dtype: int64
```

[15]:
```
texasc_daily.ewm(alpha=.7).mean().head(10).round(0)
```

[15]:
```
2020-03-05     3.0
2020-03-06     1.0
2020-03-07     3.0
2020-03-08     3.0
2020-03-09     2.0
2020-03-10     3.0
2020-03-11     4.0
2020-03-12     6.0
2020-03-13    14.0
2020-03-14    15.0
Name: Texas, dtype: float64
```

Verify by calculating manually

Let's calculate the exponentially weighted average of the last observation by hand to help understand exponential smoothing better. First, we calculate the weights using the formula above. The weight array is reversed to align with the Series which has the last observation at the end.

[16]:
```
a = .7
w = a * (1 - a) ** np.arange(10)
w = w[::-1]
w
```

[16]:
```
array([1.37781e-05, 4.59270e-05, 1.53090e-04, 5.10300e-04, 1.70100e-03,
       5.67000e-03, 1.89000e-02, 6.30000e-02, 2.10000e-01, 7.00000e-01])
```

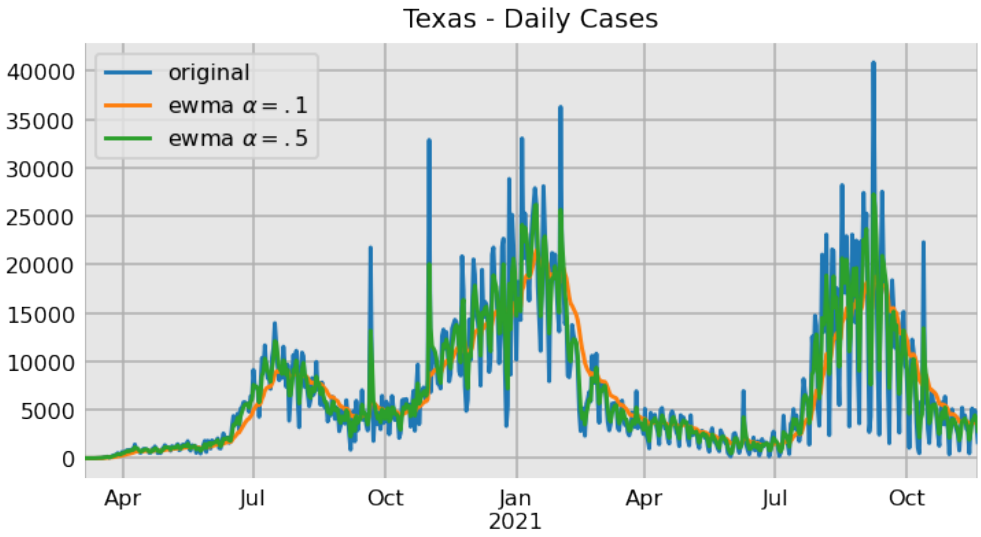
We can now take the weighted average to replicate the last result from pandas `ewm`.

[17]:
```
(texasc_daily.head(10) * w).sum() / w.sum()
```

[17]:
```
15.26523266097234
```

[18]:
```
texasc_daily.plot(title="Texas - Daily Cases", label='original')
texasc_daily.ewm(alpha=.1).mean().plot(kind='line', label=r'ewma $\alpha=.1$')
```

```
texasc_daily.ewm(alpha=.5).mean().plot(kind='line', label=r'ewma $\alpha=.5$').
↪legend();
```



One issue with the `ewm` method is that it does not allow you to center the calculation and use points on both sides of the current point. In order to do this, we'll reverse the Series, use `ewm` from the right side, and then average the results together.

[19]:
```
left = texasc_daily.ewm(alpha=.1).mean()
right = texasc_daily[::-1].ewm(alpha=.1).mean()
```

Let's output the estimates for each to show the difference.

[20]:
```
left.tail()
```

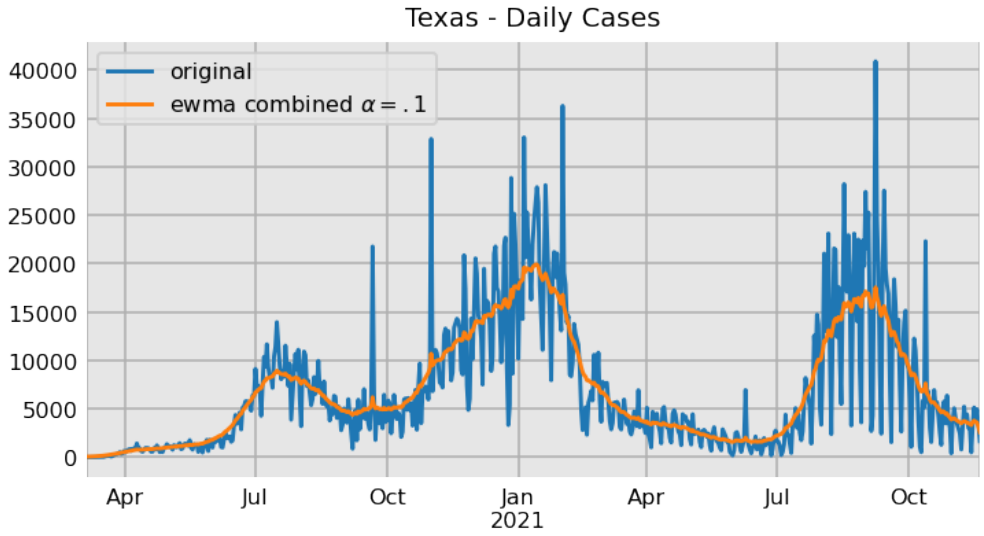
[20]:
```
2021-11-16    3575.275647
2021-11-17    3614.048082
2021-11-18    3746.043274
2021-11-19    3712.538946
2021-11-20    3505.985052
Name: Texas, dtype: float64
```

[21]:
```
right.head()
```

[21]:
```
2021-11-20    1647.000000
2021-11-19    2575.421053
2021-11-18    3445.745387
2021-11-17    3596.153824
2021-11-16    3971.198676
Name: Texas, dtype: float64
```

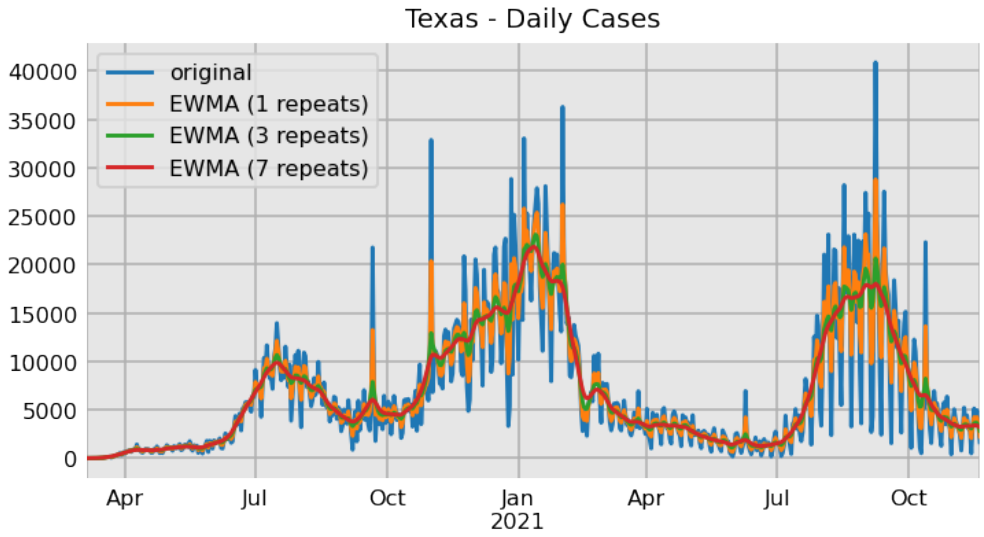
Thanks to pandas automatic alignment of the index, we can add the two Series together knowing that the dates will align properly.

[22]:
```
ewm_both = (left + right) / 2
texasc_daily.plot(title="Texas - Daily Cases", label='original')
ewm_both.plot(kind='line', label=r'ewma combined $\alpha=.1$').legend();
```



We can repeat the procedure multiple times like we did above to generate much smoother data. This repetition allows us to use a higher alpha.

[23]:
```
texasc_daily.plot(title="Texas - Daily Cases", label='original')
n = [0, 2, 6]
ewma = texasc_daily
for i in range(30):
    left = ewma.ewm(alpha=.5).mean()
    right = ewma[::-1].ewm(alpha=.5).mean()
    ewma = (left + right) / 2
    if i in n:
        ewma.plot(label=f'EWMA ({i + 1} repeats)').legend();
```



4.5 LOWESS

Locally weighted scatterplot smoothing, or LOWESS, is a procedure that also places more weight on the nearest observations. It fits a low-degree polynomial regression line through these weighted points. LOWESS is a computationally expensive procedure and not available directly in pandas. Below, we import the `lowess` function from the statsmodels package.

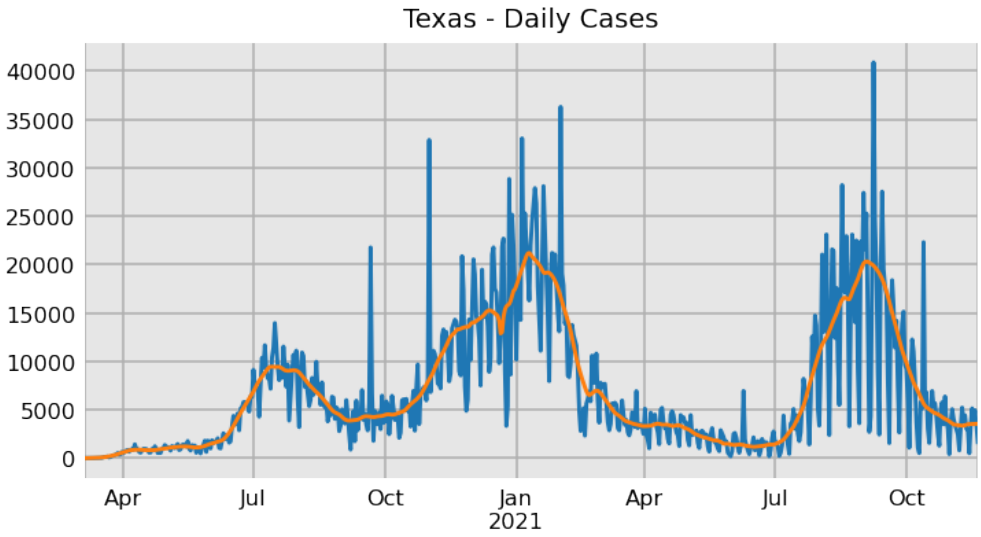
The main parameter is `frac` which is a number between 0 and 1 and determines the window size to consider as a fraction of the data. To be consistent as time goes on (and the length of our series increases), we'll choose a constant number of data points and use it to calculate `frac`. Here, we smooth based on 15 points. The last 10 smoothed points are shown below as a numpy array.

[24]:
```
from statsmodels.nonparametric.smoothers_lowess import lowess
y = texasc_daily
x = y.index
frac = 20 / len(x)
y_lowess = lowess(y, x, frac=frac, is_sorted=True, return_sorted=False)
y_lowess[-10:].round()
```

[24]:
```
array([3400., 3458., 3486., 3500., 3508., 3514., 3521., 3529., 3536.,
       3541.])
```

Let's convert this array to a pandas Series by using the original datetime index and then plot it.

[25]:
```
s_lowess = pd.Series(data=y_lowess, index=x)
texasc_daily.plot(title="Texas - Daily Cases", label='original')
s_lowess.plot();
```

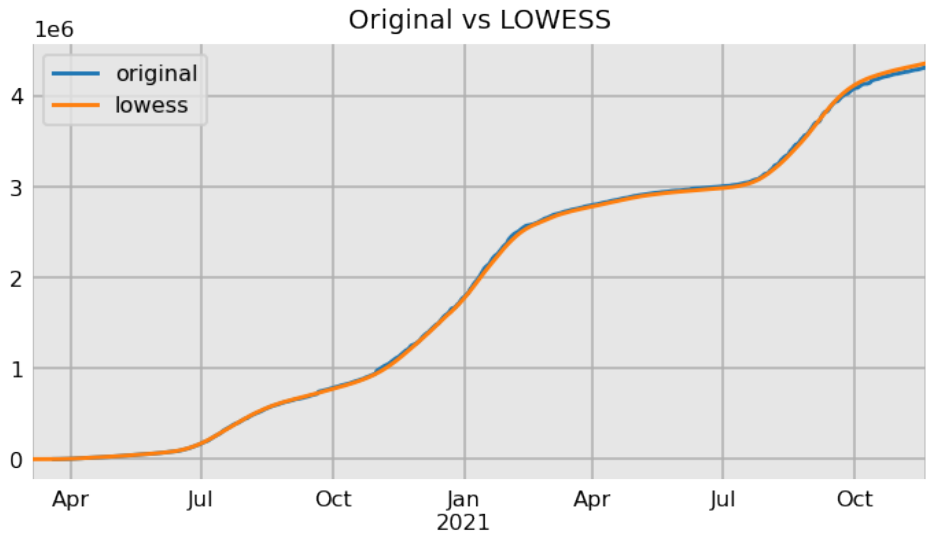


Smoothed cumulative total

Let's compute the smoothed cumulative total from the smoothed daily values.

[26]:
```
texasc.loc['2020-03-20':].plot(kind='line', label='original')
s_lowess_cumulative = s_lowess.cumsum().round(0).astype('int')
```

```
ax = s_lowess_cumulative.plot(label='lowess', title='Original vs LOWESS')
ax.legend();
```



Aligning the cumulative total to the actual

Smoothing the daily values has the effect of the cumulative total not being in direct alignment with the original data. Take a look at the last values from the actual and smoothed series.

[27]:
```
# actual
last_actual = texasc.values[-1]
last_actual
```

[27]: 4306389

[28]:
```
# smoothed
last_smoothed = s_lowess_cumulative.values[-1]
last_smoothed
```

[28]: 4353483

To align the two series, we'll multiply the smoothed values by the ratio of their last values. The new last smoothed cumulative value is output to verify it is equal to the last actual value.

[29]:
```
s_lowess_cumulative = s_lowess_cumulative * last_actual / last_smoothed
s_lowess_cumulative = s_lowess_cumulative.round(0).astype('int')
s_lowess_cumulative.values[-1]
```

[29]: 4306389

Choosing LOWESS for our model

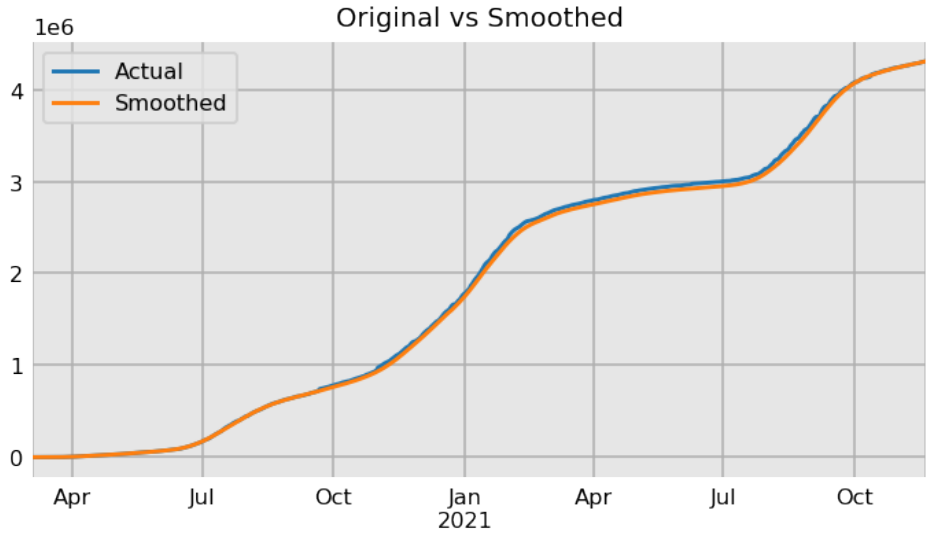
While the repeated average smoothing technique is simple, we'll use LOWESS as the smoother for our data. It is a more robust way to obtain the general direction of the data. Below, we write a function to

smooth the cumulative total given one Series of data using LOWESS. We allow the user to set `n`, the number of points used to produce the smoothed curve, converting it to a fraction of the total points to use as the `frac` parameter in the `lowess` function. We return the smoothed cumulative total as Series with the same index as the original. At the end, we multiply the smoothed values so that the last smoothed value equals the last actual recorded value.

[30]:
```
def smooth(s, n):
    """
    Smooth data using lowess function from statsmodels

    Parameters
    ----------
    s : Series of cumulative cases

    n : int, number of points to be used by lowess function

    Returns
    -------
    Series of smoothed values with same index as the original
    """
    if s.values[0] == 0:
        # Filter the data if the first value is 0
        last_zero_date = s[s == 0].index[-1]
        s = s.loc[last_zero_date:]
        s_daily = s.diff().dropna()
    else:
        # If first value not 0, use it to fill in the
        # first missing value
        s_daily = s.diff().fillna(s.iloc[0])

    # Don't smooth data with less than 15 values
    if len(s_daily) < 15:
        return s

    y = s_daily.values
    frac = n / len(y)
    x = np.arange(len(y))
    y_pred = lowess(y, x, frac=frac, is_sorted=True, return_sorted=False)
    s_pred = pd.Series(y_pred, index=s_daily.index).clip(0)
    s_pred_cumulative = s_pred.cumsum()

    # Align the smoothed value to the actual recorded value
    last_actual = s.values[-1]
    last_smoothed = s_pred_cumulative.values[-1]
    s_pred_cumulative *= last_actual / last_smoothed
    return s_pred_cumulative
```

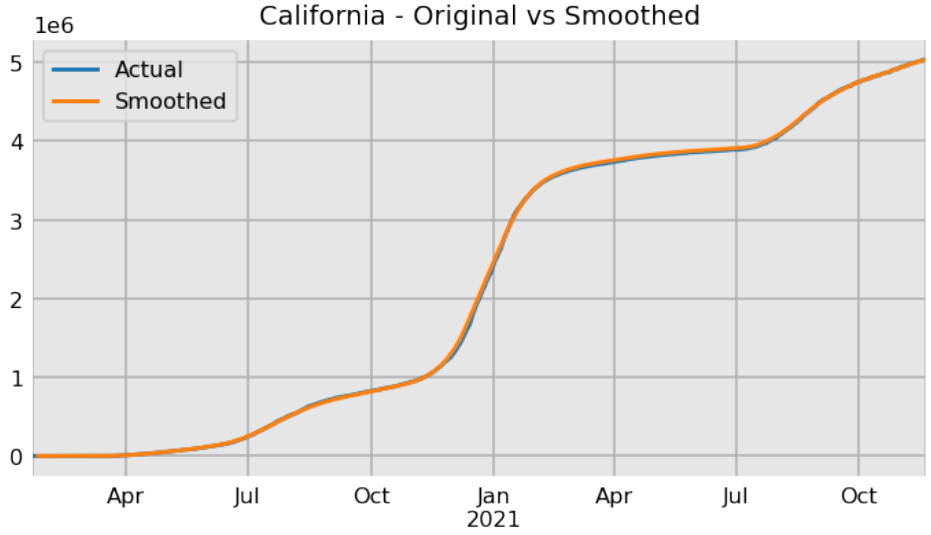
We verify that our function works by making the same plot as above.

[31]:
```
smoothed = smooth(texasc, 20)
texasc.plot(label='Actual', legend=True)
smoothed.plot(title='Original vs Smoothed',
              label='Smoothed', legend=True);
```



Here, we plot actual vs smoothed cases in California.

[32]:
```
s = usa_cases['California']
smoothed = smooth(s, 20)
s.plot(label='Actual', legend=True)
smoothed.plot(title='California - Original vs Smoothed',
              label='Smoothed', legend=True);
```



Chapter 5

Exponential Growth and Decline Models

In this chapter, we'll fit our data to a few models that are useful for predicting both exponential growth and decline.

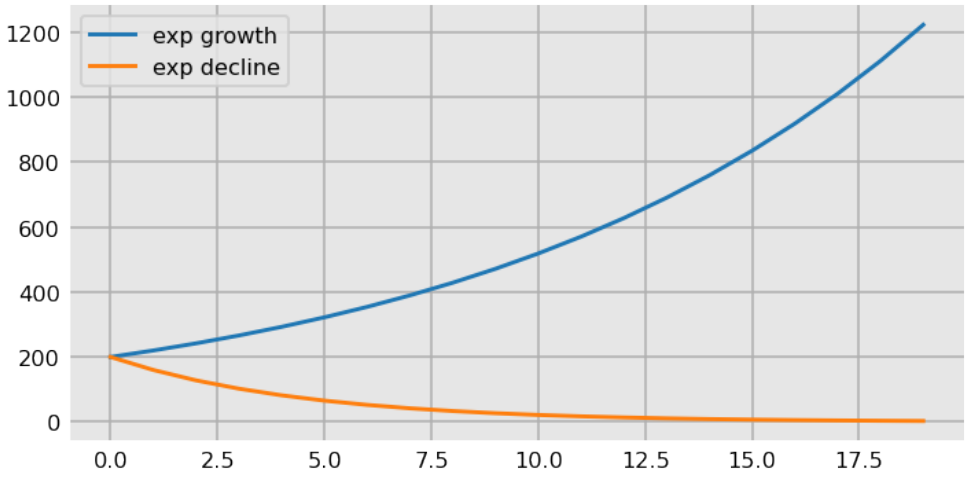
5.1 Exponential growth and decline

Before working with our data, we'll cover the basics of exponential growth and decline. In the most general form, we can write an exponential function as the following:

$$f(x) = ab^x$$

Where `a` is the initial starting value, `b` is the growth rate (must be greater than 0), and `x` is time. The starting value will decrease towards 0 when `b` is less than 1 and explode to infinity when greater than 1. At every time period, the previous value increases/decreases by multiplying it by `b`. Exponential growth and decline examples are shown below, each starting with the same initial value.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('dashboard.mplstyle')

a = 200
x = np.arange(20)

# growth at 10% each day
y_inc = a * 1.1 ** x
# decline at 20% each day
y_dec = a * 0.8 ** x

fig, ax = plt.subplots()
ax.plot(y_inc, label='exp growth')
ax.plot(y_dec, label='exp decline')
ax.legend();
```



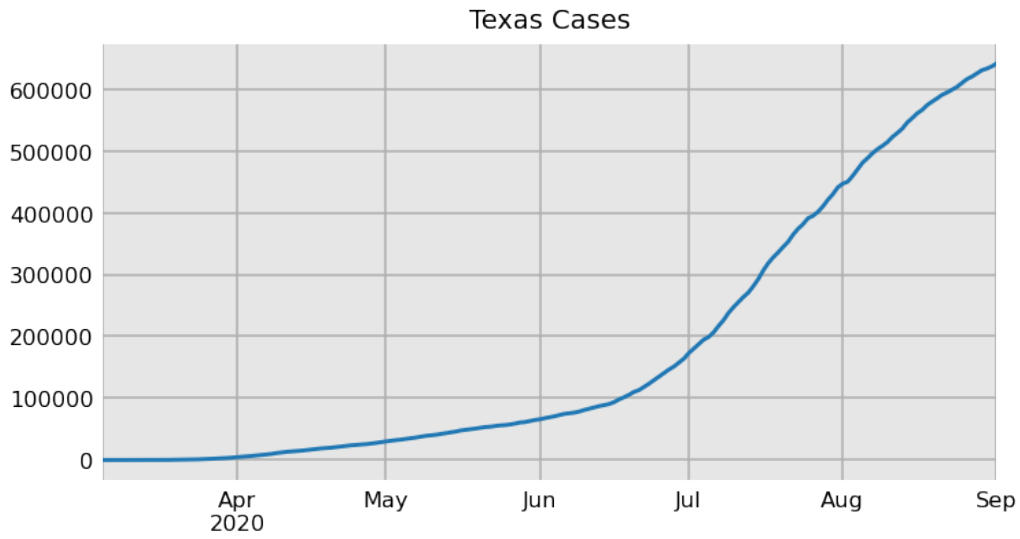
5.2 Modeling total cases with scipy's `least_squares`

Let's use this simple approach to model the cumulative total cases for different areas. Let's read in our data and select cases in Texas as a pandas Series. We'll set the last overall date as September 1, 2020.

[2]:
```
from prepare import PrepareData
data = PrepareData(download_new=False).run()
usa_cases = data['usa_cases']
usa_cases = usa_cases.loc[:'2020-09-01']
texasc = usa_cases['Texas']
texasc = texasc[texasc > 0]
texasc.head()
```

[2]:
```
2020-03-05     3
2020-03-06     4
2020-03-07     8
2020-03-08    11
2020-03-09    13
Name: Texas, dtype: int64
```

[3]:
```
texasc.plot(title='Texas Cases');
```



To use this simple exponential model, we define a function that calculates its value when given the x-values and the two parameters, `a` and `b`.

[4]:
```
def simple_exp(x, a, b):
    """
    Simple exponential model

    Parameters
    ----------
    x : array of x-values, usually just np.arange(len(y))

    a : initial starting value

    b : growth rate

    Returns
    -------
    Evaluated function values as an array
    """
    return a * b ** x
```

Here's an example with a starting value of 100 with a growth rate of 10% (1.1) for 10 days.

[5]:
```
x = np.arange(0, 10)
simple_exp(x, 100, 1.1)
```

[5]:
```
array([100.       , 110.       , 121.       , 133.1      , 146.41     ,
       161.051    , 177.1561   , 194.87171  , 214.358881 , 235.7947691])
```

Attempting to use the simple exponential function to model cases

We can attempt to use this simple exponential function to model the cases in Texas. Attempting to align a known function to specific values is know as **curve fitting**. We want the curve we are using (the

simple exponential function in this case) to lay right on top of the known cases in Texas. Specifically, we want to find values of the parameters `a` and `b` such that the simple exponential function lies as close to the curve of cumulative cases in Texas.

To find the optimal values of the parameters `a` and `b`, we'll use scipy's `least_squares` function, found in the `optimize` module. It is a complex function to use correctly. In order to use it, we must define a function that calculates the error between the function value and the y-values. The `optimize_func` function below accepts the parameters as a sequence as the first argument along with the data `x` and `y` as separate arguments. It will also accept `model`, which is the function that computes the actual value of the model.

The function calculates the value of the model at the given `x` value. It subtracts `y`, the actual recorded data value, from it to produce the error which is returned.

[6]:
```
def optimize_func(params, x, y, model):
    """
    Function to be passed as first argument to least_squares

    Parameters
    ----------
    params : sequence of parameter values for model

    x : x-values from data

    y : y-values from data

    model : function to be evaluated at x with params

    Returns
    -------
    Error between function and actual data
    """
    y_pred = model(x, *params)
    error = y - y_pred
    return error
```

The `least_squares` function requires an initial guess of the parameters, `x0`. It attempts to find the values of the parameters that minimize the total squared error (the "least squares"). We must use the `args` keyword parameter to pass in `x`, `y`, and `model` as a tuple. These arguments are forwarded to the `optimize_func` function. The general form of `least_squares` will look like this:

```
least_squares(optimize_func, initial_guess, args=(x, y, model))
```

Since we are modeling exponential growth, `a` must be positive and `b` must be greater than 1. We'll initially guess 1 for each of the parameters. It's also important to provide boundaries for the parameter values by setting `bounds` to a two-item tuple of the lower and upper bounds, which are each themselves tuples the same length as the number of parameters. We allow `a` to be between 1 and infinity and `b` to be between 1 and 10. The `least_squares` function returns the results as an object containing lots of information about the optimization. Here, we assign it to the `res` variable name.

Note that we use the variable name `p0` and not `x0`. In the next chapter, we'll use another model containing the parameter name `x0`. This is an unfortunate name collision, so we choose to always use

p0 for the initial parameter values ("parameter guess zero") passed to least_squares.

[7]:
```
from scipy.optimize import least_squares
y = texasc.values
x = np.arange(len(y))
lower_bounds = 1, 1
upper_bounds = np.inf, 10
bounds = lower_bounds, upper_bounds
p0 = 1, 1
res = least_squares(optimize_func, p0, args=(x, y, simple_exp), bounds=bounds)
```

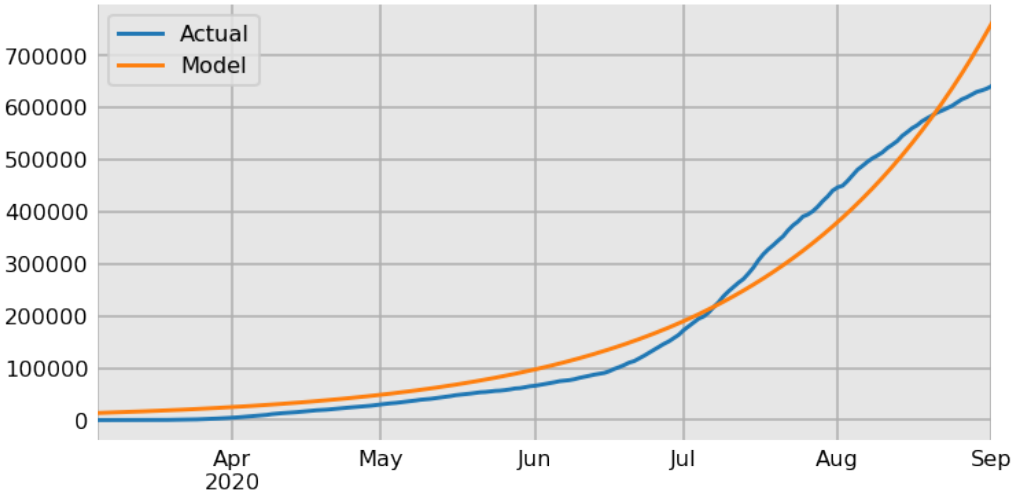
The optimal values of the parameters are found in the x attribute.

[8]:
```
res.x
```

[8]:
```
array([1.35406305e+04, 1.02263632e+00])
```

The model found the growth rate to be around 2.3% per day beginning at an initial value of 13,500. Let's pass simple_exp the x-values and estimated parameters to generate the predicted values and plot them against the actual data.

[9]:
```
y_pred = simple_exp(x, *res.x)
s_pred = pd.Series(data=y_pred, index=texasc.index)
texasc.plot(label="Actual")
s_pred.plot(label="Model").legend();
```

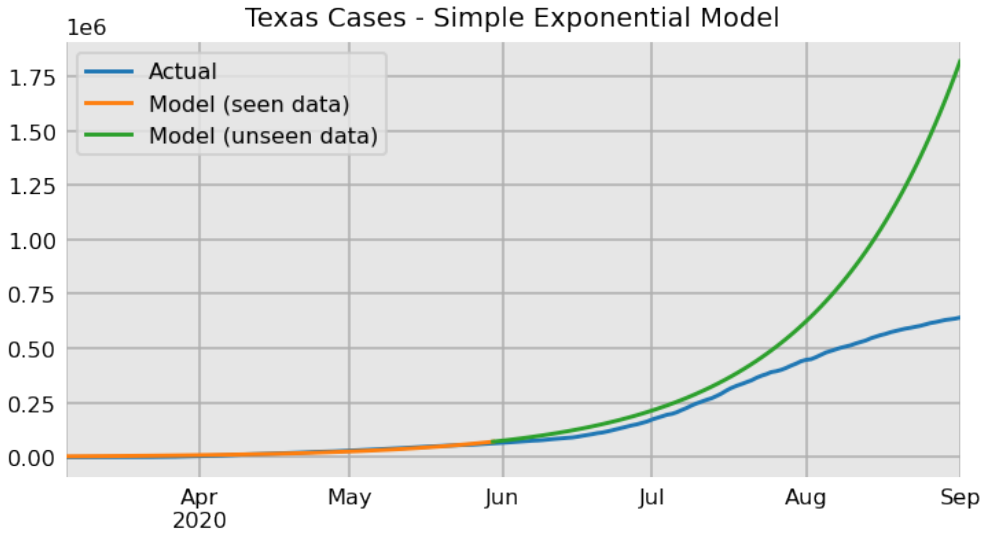


Predicting unseen data

While the model matches the data somewhat well, the utility of our model depends on data it has yet to see. Let's build our model on data only up to May 30th and then make predictions for all of the dates. As you can see below, the true evaluation for this model is substantially worse.

[10]:
```
y = texasc['2020-05-30']
x = np.arange(len(y))
res = least_squares(optimize_func, p0, args=(x, y, simple_exp), bounds=bounds)
```

```
x_pred = np.arange(len(texasc))
y_pred = simple_exp(x_pred, *res.x)
s_pred = pd.Series(data=y_pred, index=texasc.index)
texasc.plot(label="Actual")
s_pred[:'2020-05-30'].plot(label="Model (seen data)",
                           title="Texas Cases - Simple Exponential Model")
s_pred['2020-05-30':].plot(label="Model (unseen data)").legend();
```



Optimizing `least_squares`

There are many options in `least_squares` to alter the way the parameters are found. By default, `least_squares` finds a set of parameters that minimize the sum of squared errors (SSE) between the actual values and the model. It does so using a complex iterative algorithm that slowly changes the parameters each iteration calculating a new SSE. If the change in the SSE is below a pre-defined threshold, the algorithm stops and returns the parameters. The main two thresholds are given by the following parameters:

- `ftol` - change in SSE (default 1e-8)
- `xtol` - change in parameter values (default 1e-8)

Some other useful parameters:

- `max_nfev` - Maximum number of function evaluations before algorithm stops (default: number of observations * 100)
- `verbose` - Set to 1 for printed results and 2 for detailed results (default: 0).

Let's run `least_squares` setting `verbose` to 1 to view the results of the above process.

[11]:
```
res = least_squares(optimize_func, p0, args=(x, y, simple_exp),
                    bounds=bounds, verbose=1)
res.x
```

```
`ftol` termination condition is satisfied.
Function evaluations 40, initial cost 3.8442e+10, final cost 7.2124e+08, first-order
optimality 2.65e+04.
```

[11]:
```
array([3.65457081e+03, 1.03510002e+00])
```

It took around 40 function evaluations (on my machine) to meet the `ftol` condition for stopping. Let's reduce the tolerance to a smaller numbers and re-run the fit.

[12]:
```
res = least_squares(optimize_func, p0, args=(x, y, simple_exp),
                    bounds=bounds, verbose=1, ftol=1e-14)
res.x
```

```
`xtol` termination condition is satisfied.
Function evaluations 46, initial cost 3.8442e+10, final cost 7.2124e+08, first-order
optimality 1.72e+04.
```

[12]:
```
array([3.65458230e+03, 1.03509998e+00])
```

A few more iterations were run to meet the new threshold (`xtol` this time). The values for the parameters are essentially unchanged.

Function to train the model

We define a function below to train the model given a Series of cumulative case data and the last date to use for training. The function is also passed the model, bounds, and initial guess. Any additional keyword arguments will be forwarded to the `least_squares` function. The value of the parameters that minimize the squared error are returned.

[13]:
```
def train_model(s, last_date, model, bounds, p0, **kwargs):
    """
    Train a model using scipy's least_squares function
    to find the fitted parameter values

    Parameters
    ----------
    s : pandas Series with data used for modeling

    last_date : string of last date to use for model

    model : function returning model values

    bounds : two-item tuple of lower and upper bounds of parameters

    p0 : tuple of initial guess for parameters

    kwargs : extra keyword arguments forwarded to the least_squares function
                (ftol, xtol, max_nfev, verbose)

    Returns
    -------
    numpy array: fitted model parameters
    """
    y = s.loc[:last_date]
```

```
    n_train = len(y)
    x = np.arange(n_train)
    res = least_squares(optimize_func, p0, args=(x, y, model), bounds=bounds,␣
↪**kwargs)
    return res.x
```

Let's use this function to build the model trained only up to May 15 and output the fitted parameters.

[14]:
```
last_date = '2020-05-15'
params = train_model(texasc, last_date=last_date,
                     model=simple_exp, p0=p0, bounds=bounds)
params
```

[14]:
```
array([2.22179099e+03, 1.04511482e+00])
```

Here, we write a function that returns the **daily** predicted cases given the model, fitted parameters, number of observations trained, and number of predictions desired. Remember that the model provides cumulative values, not daily.

[15]:
```
def get_daily_pred(model, params, n_train, n_pred):
    """
    Makes n_pred daily predictions given a trained model

    Parameters
    ----------
    model : model that has already been trained

    params : parameters of trained model

    n_train : number of observations model trained on

    n_pred : number of predictions to make

    Returns
    -------
    numpy array: predicted daily values
    """
    x_pred = np.arange(n_train - 1, n_train + n_pred)
    y_pred = model(x_pred, *params)
    y_pred_daily = np.diff(y_pred)
    return y_pred_daily
```

Let's use this function to predict the next 50 daily new cases in Texas.

[16]:
```
n_train = len(texasc.loc[:last_date])
y_pred_daily = get_daily_pred(simple_exp, params, n_train, n_pred=50).round()
y_pred_daily
```

[16]:
```
array([ 2300.,  2403.,  2512.,  2625.,  2744.,  2867.,  2997.,  3132.,
        3273.,  3421.,  3575.,  3737.,  3905.,  4081.,  4265.,  4458.,
        4659.,  4869.,  5089.,  5318.,  5558.,  5809.,  6071.,  6345.,
        6631.,  6931.,  7243.,  7570.,  7912.,  8268.,  8641.,  9031.,
        9439.,  9865., 10310., 10775., 11261., 11769., 12300., 12855.,
       13435., 14041., 14674., 15336., 16028., 16751., 17507., 18297.,
       19122., 19985.])
```

We can use these daily values to calculate the cumulative total. First we find the last known total.

[17]:
```
last_actual_value = texasc['2020-5-15']
last_actual_value
```

[17]:
```
45891
```

Adding this value to the cumulative sum gives us the predicted cumulative total.

[18]:
```
y_pred_cumulative = y_pred_daily.cumsum() + last_actual_value
y_pred_cumulative
```

[18]:
```
array([ 48191.,  50594.,  53106.,  55731.,  58475.,  61342.,  64339.,
        67471.,  70744.,  74165.,  77740.,  81477.,  85382.,  89463.,
        93728.,  98186., 102845., 107714., 112803., 118121., 123679.,
       129488., 135559., 141904., 148535., 155466., 162709., 170279.,
       178191., 186459., 195100., 204131., 213570., 223435., 233745.,
       244520., 255781., 267550., 279850., 292705., 306140., 320181.,
       334855., 350191., 366219., 382970., 400477., 418774., 437896.,
       457881.])
```

Here, we write a function that returns the cumulative predicted values starting with the first day after the last date used in the model. A Series is returned with an index that has the correct dates.

[19]:
```
def get_cumulative_pred(last_actual_value, y_pred_daily, last_date):
    """
    Returns the cumulative predicted values beginning with the
    first date after the last known date

    Parameters
    ----------
    last_actual_value : int, last recorded value

    y_pred_daily : array of predicted values

    last_date : string of last date used in model

    Returns
    -------
    Series with correct dates in the index
    """
    first_pred_date = pd.Timestamp(last_date) + pd.Timedelta("1D")
```

```
    n_pred = len(y_pred_daily)
    index = pd.date_range(start=first_pred_date, periods=n_pred)
    return pd.Series(y_pred_daily.cumsum(), index=index) + last_actual_value
```

Using the predicted daily values for the cumulative total resolves the issue we had with using the smoothed data which cumulative values did not exactly align with the actual last cumulative value.

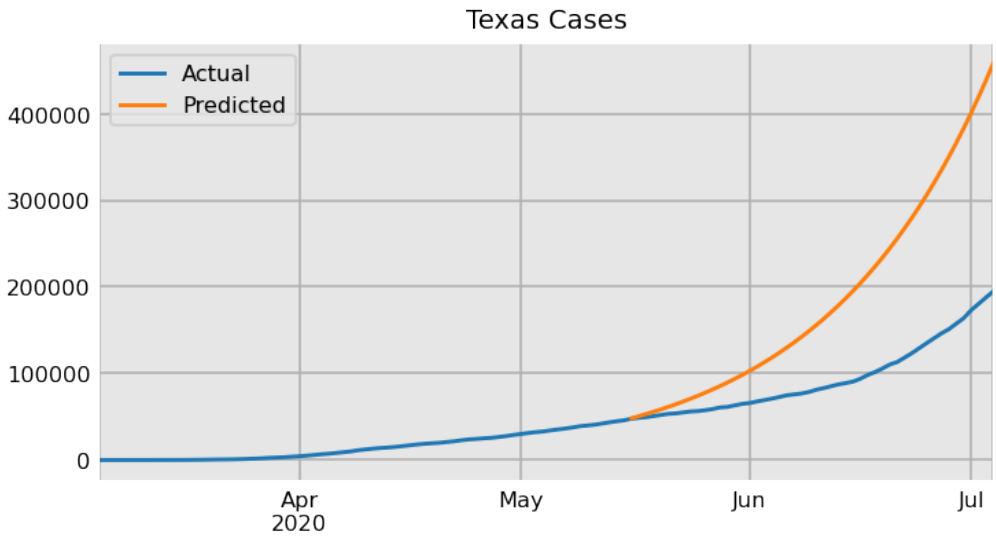
[20]:
```
s_pred_cumulative = get_cumulative_pred(last_actual_value, y_pred_daily,␣
↪"2020-05-15")
s_pred_cumulative.tail()
```

[20]:
```
2020-06-30    382970.0
2020-07-01    400477.0
2020-07-02    418774.0
2020-07-03    437896.0
2020-07-04    457881.0
Freq: D, dtype: float64
```

A separate function is defined to plot the original and predicted Series. The original data is plotted only up to the last predicted date.

[21]:
```
def plot_prediction(s, s_pred, title=""):
    """
    Plots both the original and predicted values

    Parameters
    ----------
    s : Series of original data

    s_pred : Series of predictions

    title : title of plot

    Returns
    -------
    None
    """
    last_pred_date = s_pred.index[-1]
    ax = s[:last_pred_date].plot(label="Actual")
    s_pred.plot(label="Predicted", title=title).legend()
```

[22]:
```
plot_prediction(texasc, s_pred_cumulative, title="Texas Cases")
```



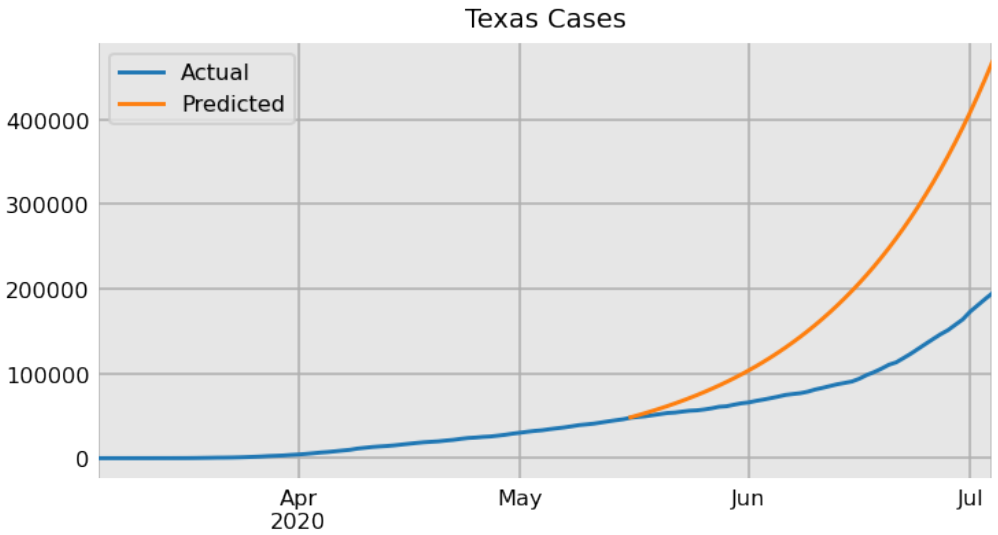
Let's put all of our work together into a single function that smooths, predicts, and plots the results. The `smooth`, `train_model`, `get_daily_pred`, `get_cumulative_pred` and `plot_prediction` functions are used to build a model and plot it. The function returns the array of fitted parameter values and the array of daily predicted values as a two-item tuple. The `start_date` is used to select the beginning of the modeling period (which will be useful when we model new waves).

[23]:
```
from solutions import smooth

def predict_all(s, start_date, last_date, n_smooth, n_pred, model,
                bounds, p0, title="", **kwargs):
    """
    Smooth, train, predict, and plot a Series of data

    Parameters
    ----------
    s : pandas Series with data used for modeling

    start_date : string of first date to use for model

    last_date : string of last date to use for model

    n_smooth : number of points of data to be used by lowess function

    n_pred : number of predictions to make

    model : function returning model values

    bounds : two-item tuple of lower and upper bounds of parameters

    p0 : tuple of initial guess for parameters

    title : title of plot
```

```
    kwargs : extra keyword arguments forwarded to the least_squares function
                (bounds, ftol, xtol, max_nfev, verbose)

    Returns
    -------
    Array of fitted parameters
    """
    # Smooth up to the last date
    s_smooth = smooth(s[:last_date], n=n_smooth)

    # Filter for the start of the modeling period
    s_smooth = s_smooth[start_date:]

    params = train_model(s_smooth, last_date=last_date,
                         model=model, bounds=bounds, p0=p0, **kwargs)
    n_train = len(s_smooth)
    y_daily_pred = get_daily_pred(model, params, n_train, n_pred)
    last_actual_value = s.loc[last_date]
    s_cumulative_pred = get_cumulative_pred(last_actual_value, y_daily_pred,␣
↪last_date)
    plot_prediction(s[start_date:], s_cumulative_pred, title=title)
    return params, y_daily_pred
```

Let's test our function with cases in Texas, passing in the original Series of cumulative cases. This function will be used often in the future to complete all of the steps of the model building process.

[24]:
```
bounds = (1, 1), (np.inf, 10)
p0 = 1, 1
params, y_pred = predict_all(texasc, start_date=None, last_date="2020-05-15",
                             n_smooth=15, n_pred=50, model=simple_exp,
                             bounds=bounds, p0=p0, title="Texas Cases")
```



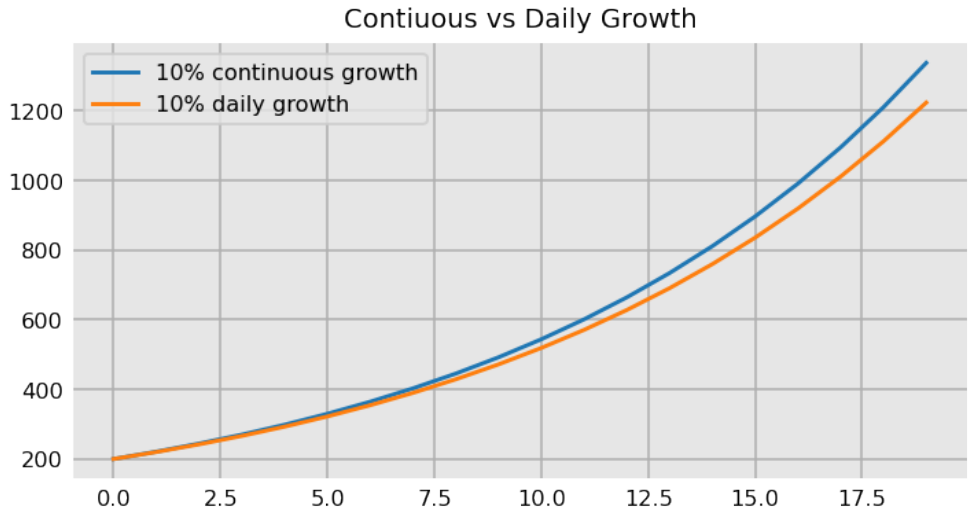
Continuous growth with e

Our data is reported once per day, but we can think of the growth/decline as continuously happening. For continual growth processes, we use the mathematical constant e (2.718...), which represents the limit to the growth rate if measured in infinitesimally small intervals, as opposed to once per day. Our model changes to:

$$f(x) = ae^{bx}$$

Where `a` and `b` still represent the initial value and growth rate, but `b` can now be negative or positive. A negative value of `b` results in exponential decline, while a positive value signifies exponential growth. In this formulation, `b` will approximately equal one less than its value in the first model (assuming `b` is fairly small). For instance, in our very first plot in this chapter, we used a value of 1.1 for `b` which represented a constant 10% increase each day. Using 10% continuous growth with the above formula (setting `b = 0.1`) yields similar results. Both formulations of the exponential model will work similarly.

[25]:
```
x = np.arange(20)
a = 200
b_old = 1.1
b_new = b_old - 1
y_old = a * b_old ** x
y_new = a * np.exp(b_new * x)

fig, ax = plt.subplots()
ax.set_title('Contiuous vs Daily Growth')
ax.plot(y_new, label='10% continuous growth')
ax.plot(y_old, label='10% daily growth')
ax.legend();
```



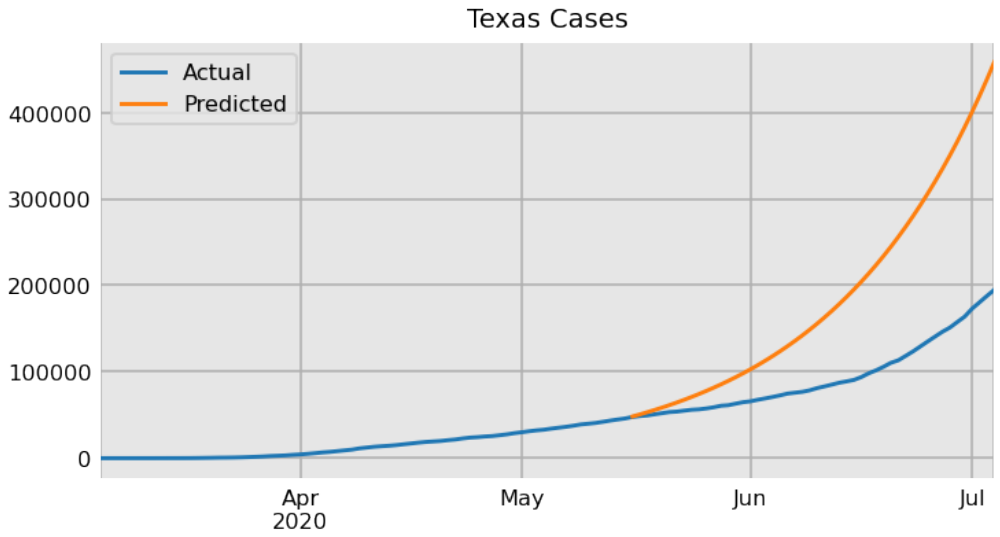
Let's build a predictive model with this new formulation in the same manner as before.

[26]:
```
def simple_exp_cont(x, a, b):
    """
    Simple exponential model using continuous growth
```

```
    Parameters
    ----------
    x : array of x-values, usually just np.arange(len(y))

    a : initial starting value

    b : growth rate
    """
    return a * np.exp(b * x)
```

We'll change the bounds to adapt to this new formulation and then run `predict_all` to build and plot the model. This function makes it simple to train with any model of your choice and plot the results.

[27]:
```
bounds = [1, 0.001], [10_000, 0.1]
p0 = 1, .01
predict_all(texasc,  start_date=None, last_date="2020-05-15",
            n_smooth=0.15, n_pred=50, model=simple_exp_cont,
            bounds=bounds, p0=p0, title="Texas Cases");
```



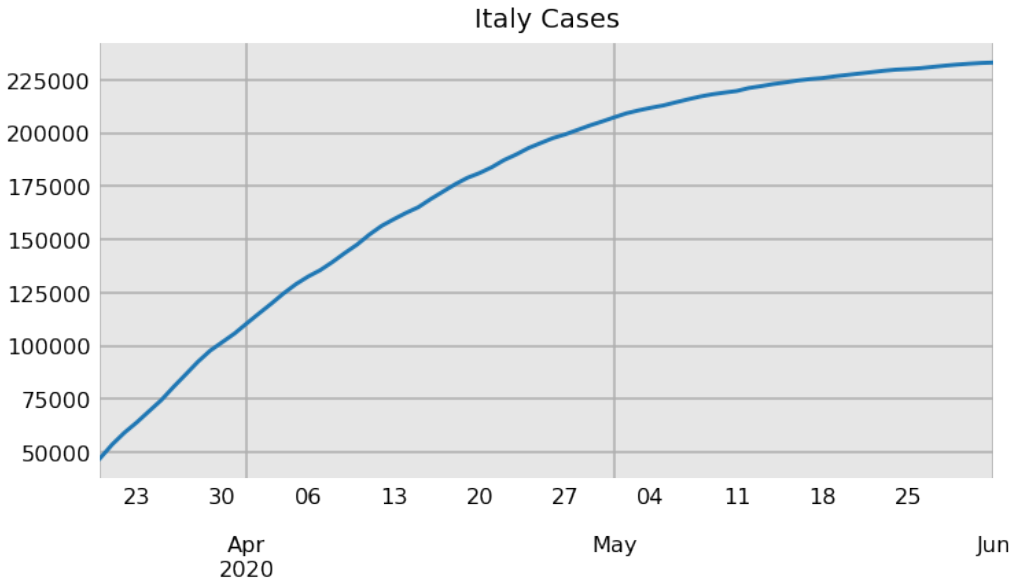
The main issue with nearly all simple exponential models is that the actual growth rate changes over time. With our coronavirus data, the total deaths/cases is capped by the world's population, so cannot explode to infinity.

5.3 Modeling exponential decline

It is common for the spread of pandemics throughout a population to begin with exponential growth, but then decrease until there is little/steady growth. Take a look at the plot below showing cases from Italy from March 20 through June 1. Exponential growth took place beginning in March, before a slowdown began in early to mid April.

[28]:
```
italyc = data['world_cases']['Italy']
italyc = italyc['2020-03-20':'2020-06-01']
```

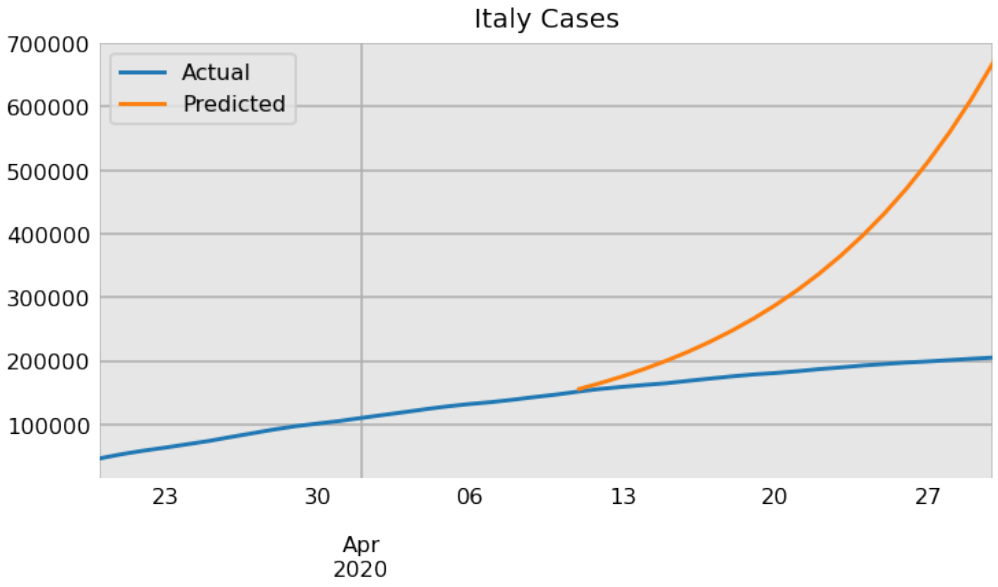
```
italyc.plot(title="Italy Cases");
```



Attempt to fit an exponential model

Let's use the `predict_all` function to fit our simple exponential model to Italy's data, giving it data through the beginning of April.

[29]:
```
predict_all(italyc, start_date=None, last_date="2020-04-10",
            n_smooth=15, n_pred=20, model=simple_exp_cont,
            bounds=bounds, p0=p0, title="Italy Cases");
```

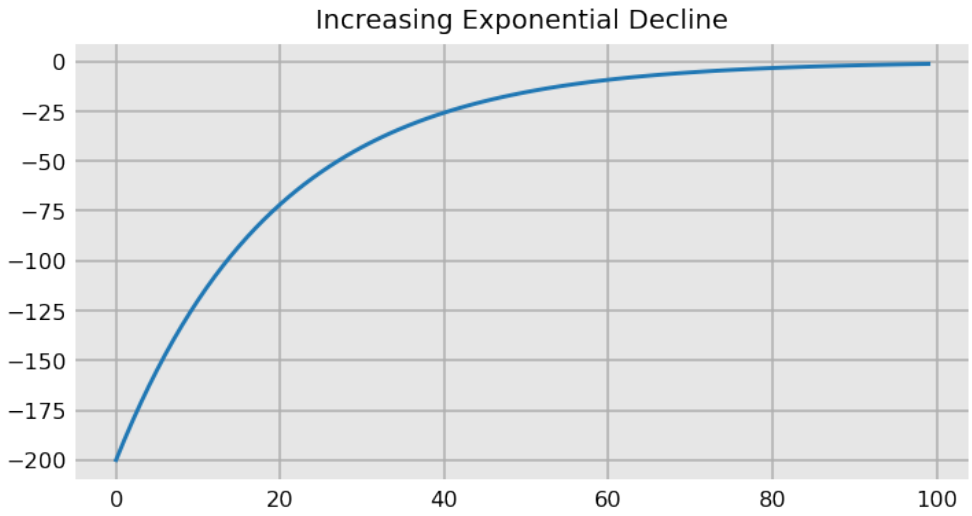


Increasing exponential decline

Clearly, we have the wrong choice of model. Our simple approach is only good for modeling exponential growth in the beginning but not during the slowdown. Our model can be transformed so that it shows **decreasing exponential decline**, (when the values head towards 0), by selecting `b` that is less than 1, or less than 0 in the second (continuous) formulation.

The issue here is that our data is not decreasing to 0, but increasing towards an asymptote. This is referred to as **increasing exponential decline**. The values increase, but do so at slower and slower rates as time goes on. If we multiply the decreasing exponential decline model by -1, we get the desired shape. Let's create an example by choosing a negative value for `a`.

[30]:
```
a = -200
b = .95
x = np.arange(100)
y = a * b ** x
fig, ax = plt.subplots()
ax.plot(y)
ax.set_title('Increasing Exponential Decline');
```



Since our data doesn't start off negative, we can add a new parameter, `c`, to shift the values upwards. Our new model becomes:

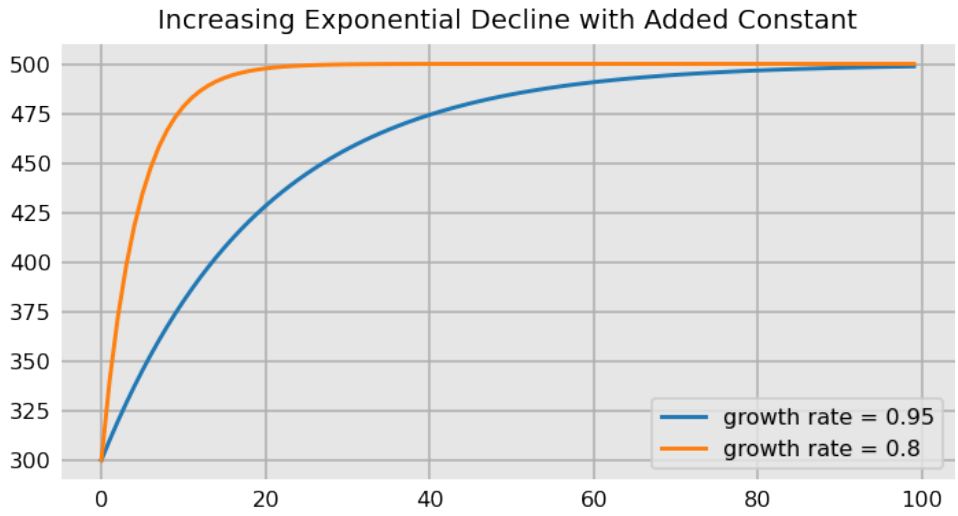
$$f(x) = ab^x + c$$

or

$$f(x) = ae^{bx} + c$$

This will move the asymptote from 0 to `c` with the initial value now at `a + c`. For increasing exponential decline, `a` will always be negative and `b` less than 1 (or less than 0 in the continuous formulation). Let's shift the above graph up 500 units and add a second line with a smaller growth rate. It's important to note that a smaller growth rate results in a faster increase when modeling increasing exponential decline.

[31]:
```
a = -200
b = 0.95
b2 = 0.8
c = 500
x = np.arange(100)
y = a * b ** x + c
y2 = a * b2 ** x + c

fig, ax = plt.subplots()
ax.plot(y, label=f'growth rate = {b}')
ax.plot(y2, label=f'growth rate = {b2}')
ax.set_title('Increasing Exponential Decline with Added Constant')
ax.legend();
```



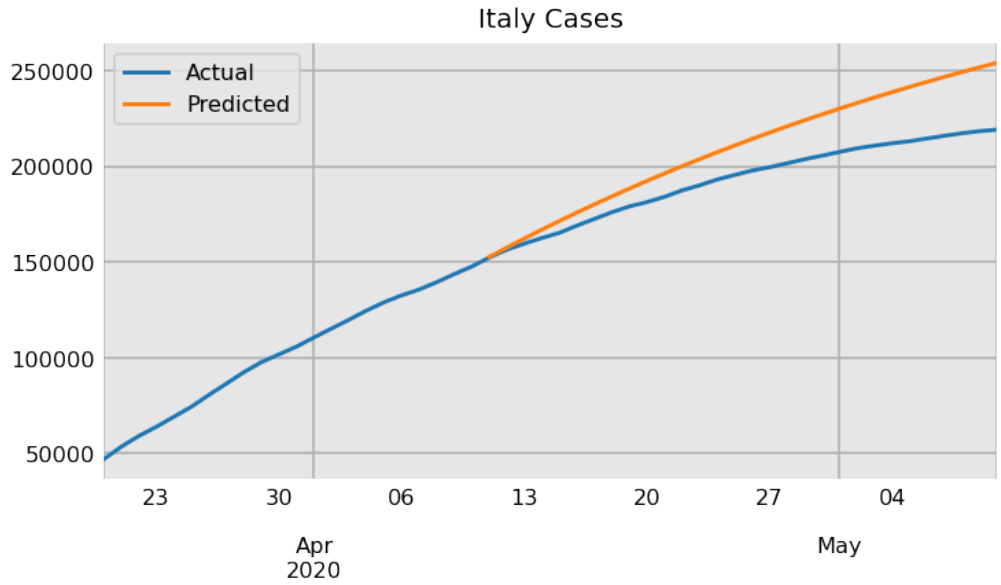
Let's define a new function for this model.

[32]:
```
def exp_decline(x, a, b, c):
    """
    Simple exponential decline model

    Parameters
    ----------
    x : array of x-values, usually just np.arange(len(y))

    a : initial value

    b : growth rate

    c : vertical shift
    """
    return a * b ** x + c
```

We must change our bounds in order for this model to work. Here, `a` must be negative, `b` between 0 and 1, and `c` positive. We set the bounds for both `a` and `c` to be a very wide interval as we do not

know the final total of cases in Italy. Let's use the `predict_all` function to build a model using data up through April 10.

[33]:
```
bounds = [-1000000, .1, 0], [-1, 1, np.inf]
p0 = -1000, .5, 100
predict_all(italyc, start_date=None, last_date="2020-04-10",
            n_smooth=15, n_pred=30,
            model=exp_decline, bounds=bounds,
            p0=p0, title="Italy Cases");
```

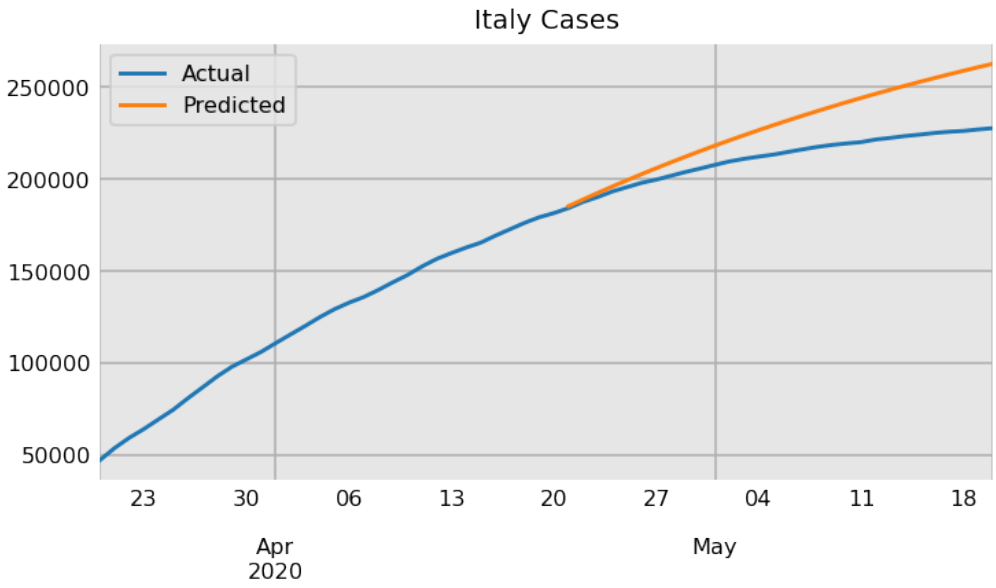


The shape of the model looks good and the predictions are decent. Below, we build the same model using continuous growth.

[34]:
```
def exp_decline_cont(x, a, b, c):
    """
    Simple exponential decline with continuous growth

    Parameters
    ----------
    x : array of x-values, usually just np.arange(len(y))

    a : initial value

    b : growth rate

    c : vertical shift
    """
    return a * np.exp(b * x) + c
```

The bounds for the growth rate must be set to be negative.

[35]:
```
bounds = [-1000000, -1, 0], [-1, 0, np.inf]
p0 = -1, -1, 1
predict_all(italyc['2020-03-10':], start_date=None, last_date="2020-04-20",
            n_smooth=15, n_pred=30, model=exp_decline_cont, bounds=bounds,
            p0=p0, title="Italy Cases");
```

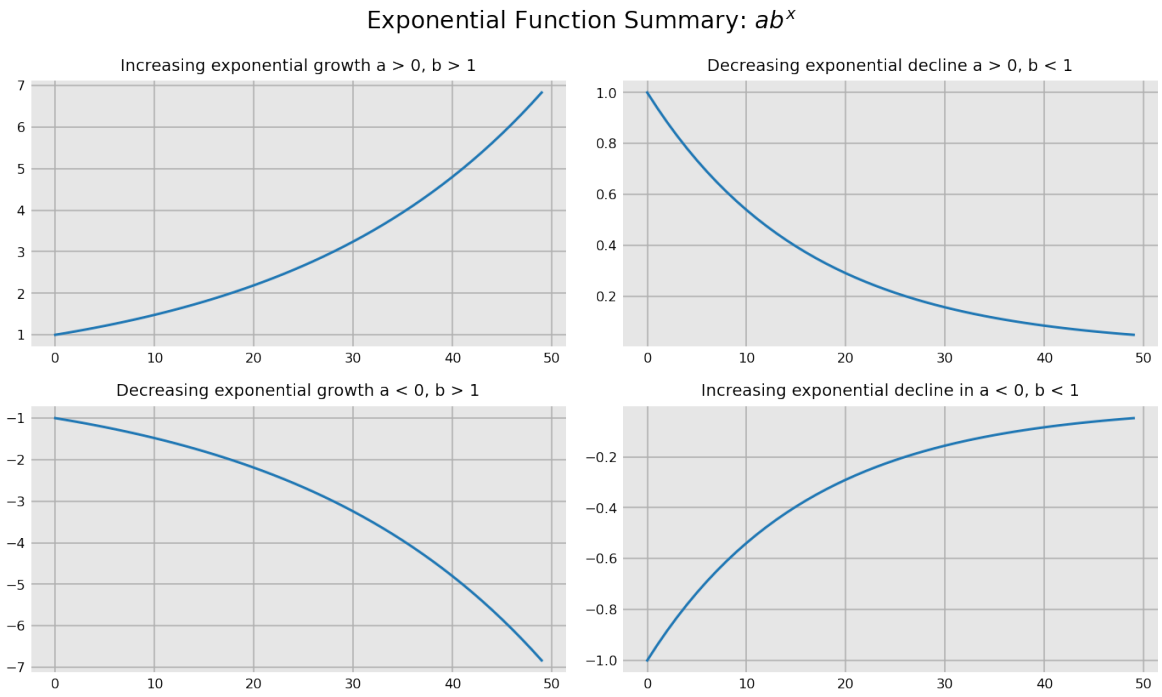


This concludes our exploration of exponential growth and decline models. These models do a good job at fitting the beginning or the end of the curve, but not both simultaneously. In the next chapter, we'll learn about logistic functions, which are able to fit exponential growth and decline in the same model.

5.4 Exponential Function Summary

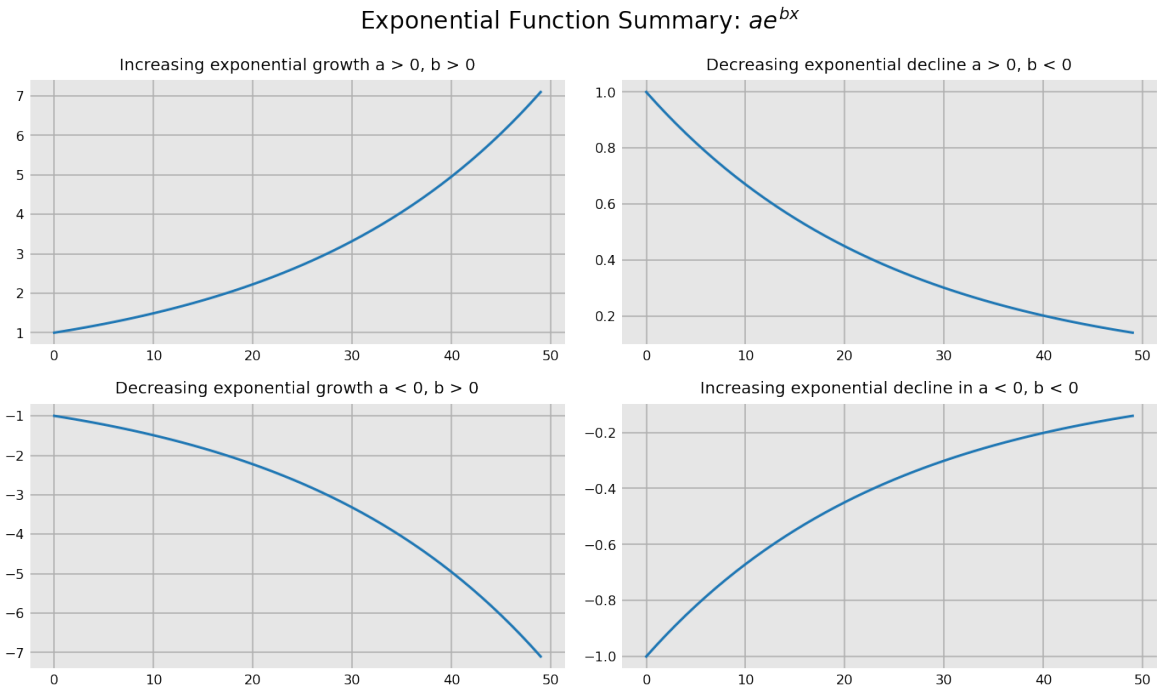
The four kinds of exponential functions are summarized below:

[36]:
```
def exp_explained(x, a, b, ax, title):
    y = a * b ** x
    ax.plot(y)
    ax.set_title(title)

x = np.arange(50)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10, 6),␣
 ↪tight_layout=True)
fig.suptitle('Exponential Function Summary: $ab^x$', y=.98, fontsize='x-large')
y = exp_explained(x, 1, 1.04, ax1, "Increasing exponential growth a > 0, b > 1")
y = exp_explained(x, 1, .94, ax2, "Decreasing exponential decline a > 0, b < 1")
y = exp_explained(x, -1, 1.04, ax3, "Decreasing exponential growth a < 0, b > 1")
y = exp_explained(x, -1, .94, ax4, "Increasing exponential decline in a < 0, b < 1")
```



The same models using continuous growth are plotted below:

[37]:
```
def exp_e_explained(x, a, b, ax, title):
    y = a * np.exp(b * x)
    ax.plot(y)
    ax.set_title(title)

x = np.arange(50)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10, 6),␣
 ↪tight_layout=True)
fig.suptitle('Exponential Function Summary: $ae^{bx}$', y=.98, fontsize='x-large')
y = exp_e_explained(x, 1, .04, ax1, "Increasing exponential growth a > 0, b > 0")
y = exp_e_explained(x, 1, -.04, ax2, "Decreasing exponential decline a > 0, b < 0")
y = exp_e_explained(x, -1, .04, ax3, "Decreasing exponential growth a < 0, b > 0")
y = exp_e_explained(x, -1, -.04, ax4, "Increasing exponential decline in a < 0, b <␣
 ↪0")
```


Exponential Function Summary: ae^{bx}
Increasing exponential growth a > 0, b > 0
Decreasing exponential decline a > 0, b < 0
Decreasing exponential growth a < 0, b > 0
Increasing exponential decline in a < 0, b < 0

Chapter 6

Logistic Growth Models

In the previous chapter we built exponential growth and decline models that were able to model the beginning and end of the pandemic curves. In this chapter, we'll cover the logistic growth model, which is capable of modeling both exponential growth and decline within the same model.

6.1 Asymptotes

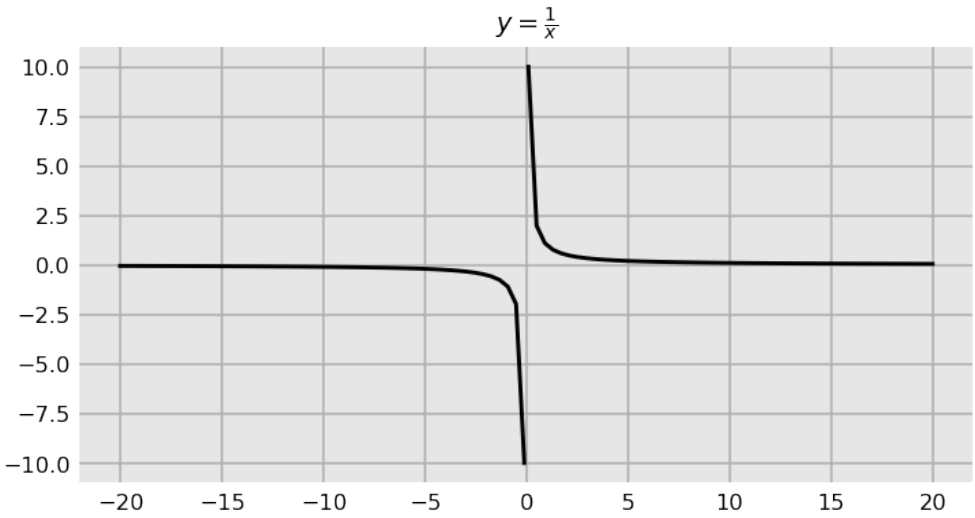
This section provides brief coverage of asymptotes and the types of functions that produce them. An asymptote is the limit of a function's value as its input approaches some number. The exponential models of the previous chapter all had a single asymptote at one end. We need a model that has two asymptotes, one at either end. Let's take a look at a few functions with asymptotes on either end and determine if one exists that fits the shape of our coronavirus case curve.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('dashboard.mplstyle')
```

Simple asymptotic functions

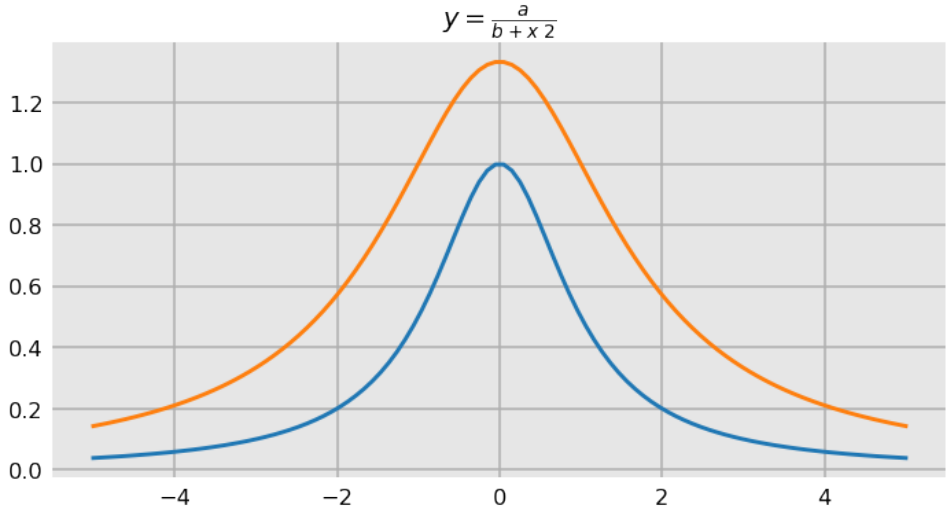
The simple function $\frac{1}{x}$ has asymptotes on both ends, but is undefined at zero and doesn't match the shape of our coronavirus case curves, so we'll need a different approach.

[2]:
```
x = np.linspace(-20, -0.1, 50)
x1 = np.linspace(0.1, 20, 50)
y = 1 / x
y1 = 1 / x1
fig, ax = plt.subplots()
ax.plot(x, y, color='black')
ax.plot(x1, y1, color='black')
ax.set_title(r'$y = \frac{1}{x}$');
```



The function $\frac{a}{b+x^2}$ is always defined and always positive. The parameters, a and b control the height and growth rate (steepness of decline from 0).

[3]:
```
x = np.linspace(-5, 5, 100)
y1 = 1 / (1 + x ** 2)
y2 = 4 / (3 + x ** 2)
fig, ax = plt.subplots()
ax.plot(x, y1)
ax.plot(x, y2)
ax.set_title(r'$y = \frac{a}{b + x ^ 2}$');
```



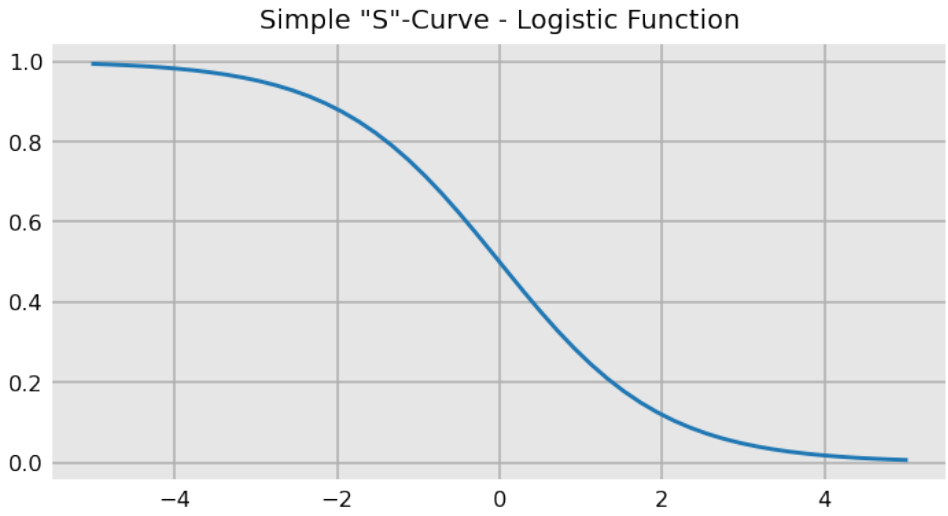
6.2 S-Curves

While the above curves have asymptotes on either end, they are not suitable for our data. We are looking for a class of functions known as S-Curves or Sigmoid functions that are shaped like an "S". The most basic form of this function is called the **logistic function** defined by the following equation:

$$f(x) = \frac{1}{1 + e^x}$$

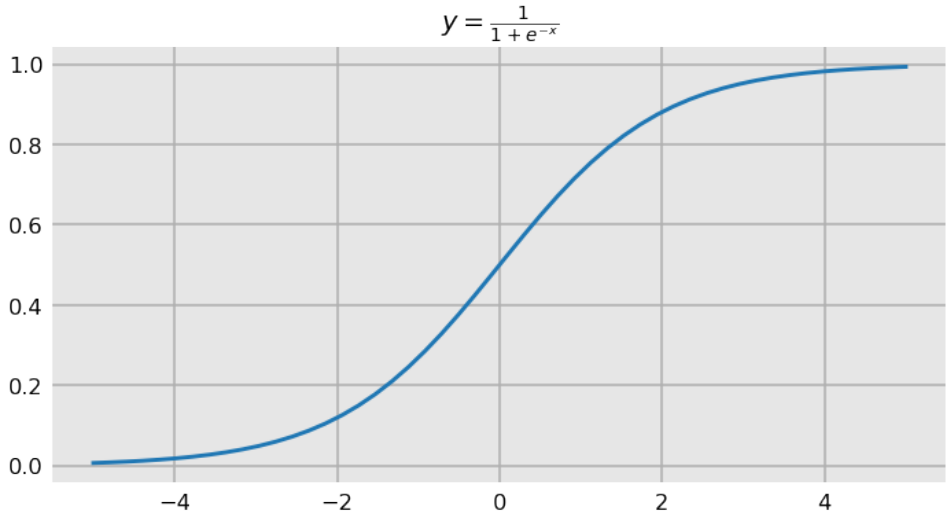
Any base can be used for the exponential, but e is the most common. Let's make a plot of it below.

[4]:
```
x = np.linspace(-5, 5, 50)
y = 1 / (1 + np.exp(x))
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple "S"-Curve - Logistic Function');
```



This is almost the exact shape we need for our model. We can rotate the curve around the y-axis so that it ends at the higher asymptote by negating `x`.

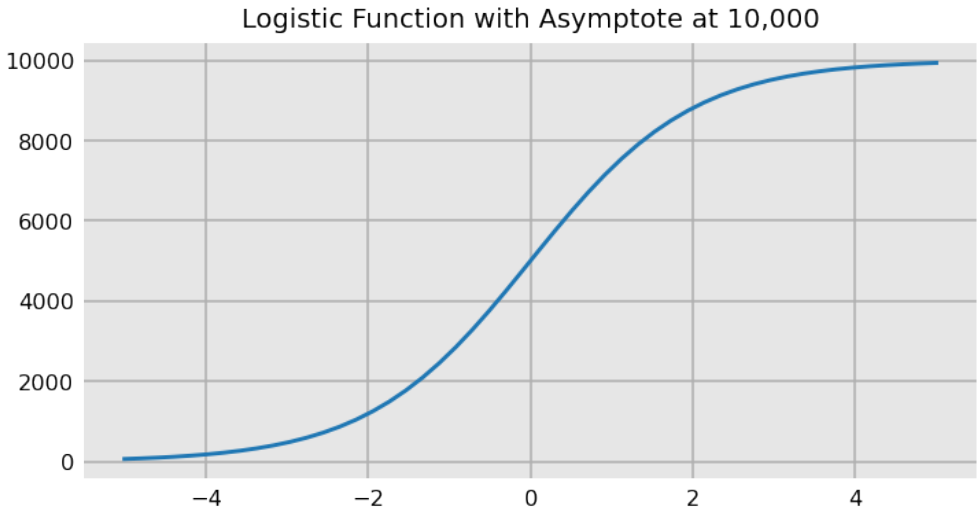
[5]:
```
x = np.linspace(-5, 5, 50)
y = 1 / (1 + np.exp(-x))
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title(r'$y = \frac{1}{1 + e ^{-x}}$');
```



Controlling the upper asymptote

The numerator controls the scale and determines the upper asymptote. Instead of using the constant value one, we'll model it using variable name `L`. Here, we use 10,000 as the upper asymptote. The shape of the curve is the exact same, just the y-values have changed.

[6]:
```
L = 10_000
x = np.linspace(-5, 5, 50)
y = L / (1 + np.exp(-x))
plt.plot(x, y)
plt.title(f'Logistic Function with Asymptote at {L:,}');
```

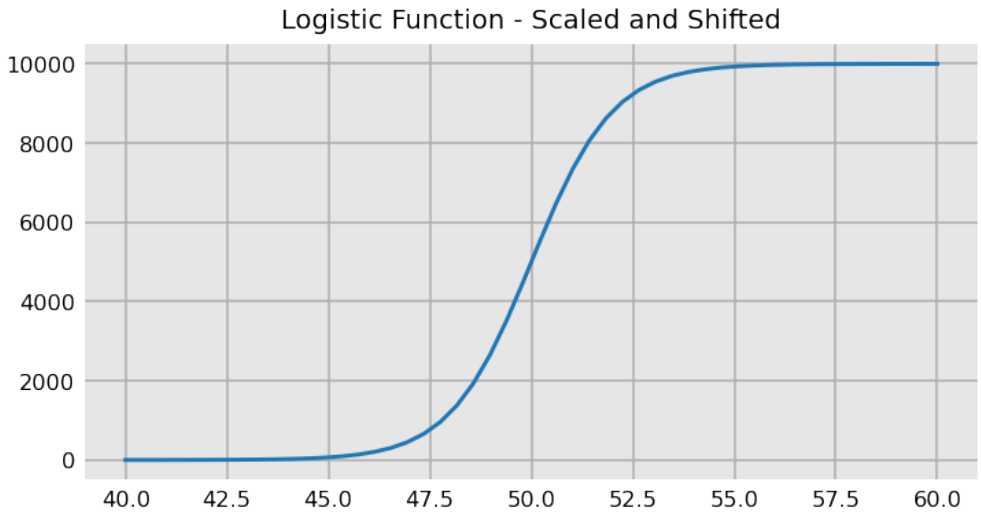


Shifting to the right

Notice that when x equals 0, the curve is at its midpoint (5,000 above). With our data, `x` represents the number of days since the start of the coronavirus pandemic. Since we can't have negative x-values, we can fix this by introducing a new term, x_0, that shifts the graph horizontally. The value for x_0 will be the location of the midpoint, that is, $\frac{L}{2}$.

$$f(x) = \frac{L}{1 + e^{-(x - x_0)}}$$

[7]:
```
L = 10_000
x0 = 50
x = np.linspace(40, 60)
y = L / (1 + np.exp(-(x - x0)))
plt.plot(x, y)
plt.title('Logistic Function - Scaled and Shifted');
```

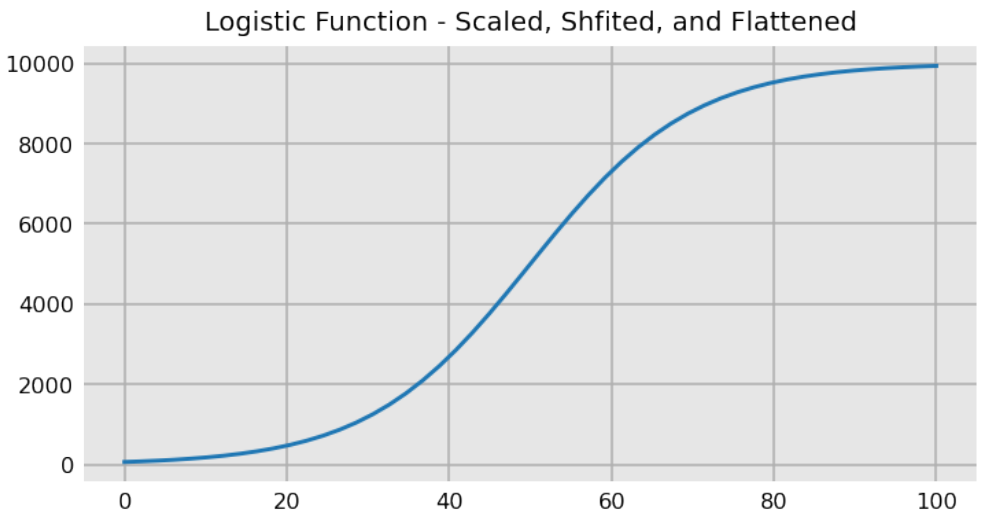


Control steepness with growth rate

The growth rate (steepness of the curve) can be modified by adding another parameter, k, to the equation. The above curves had a growth rate of $k = 1$. Here, we change it to a much smaller number, 0.1, which significantly flattens the curve. Notice the x-axis ranges from 0 to 100 now instead of from 40 to 60.

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}}$$

[8]:
```
L = 10_000
x0 = 50
k = 0.1
x = np.linspace(0, 100, 50)
y = L / (1 + np.exp(-k * (x - x0)))
plt.plot(x, y)
plt.title('Logistic Function - Scaled, Shfited, and Flattened');
```



Here, we write a function that computes the logistic function given values of x and its three parameters.

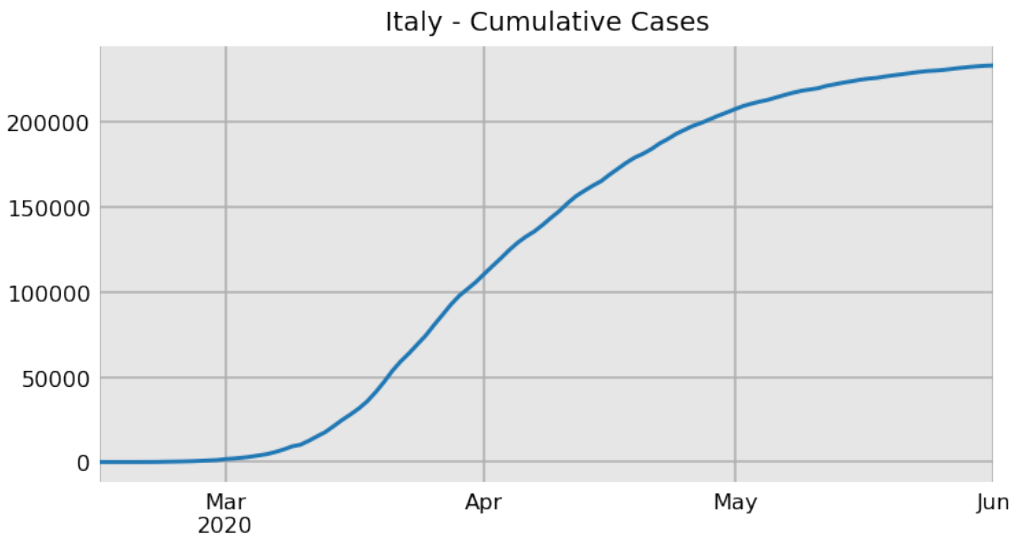
[9]:
```
def logistic_func(x, L, x0, k):
    """
    Computes the value of the logistic function

    Parameters
    ----------
    x : array of x values

    L : Upper asymptote

    x0 : horizontal shift

    k : growth rate
    """
    return L / (1 + np.exp(-k * (x - x0)))
```

[10]:
```
logistic_func(np.arange(20), 5000, 12, 0.1).round(-1)
```

[10]:
```
array([1160., 1250., 1340., 1450., 1550., 1660., 1770., 1890., 2010.,
       2130., 2250., 2380., 2500., 2620., 2750., 2870., 2990., 3110.,
       3230., 3340.])
```

Let's take a look at the cases in Italy from mid-February to the first of June and see if we can find good parameters of our newly created logistic function so that it fits the curve well.

[11]:
```
from prepare import PrepareData
data = PrepareData(download_new=False).run()
italyc = data['world_cases']['Italy']
italyc = italyc.loc["2020-02-15":"2020-06-01"]
italyc.plot(title="Italy - Cumulative Cases");
```

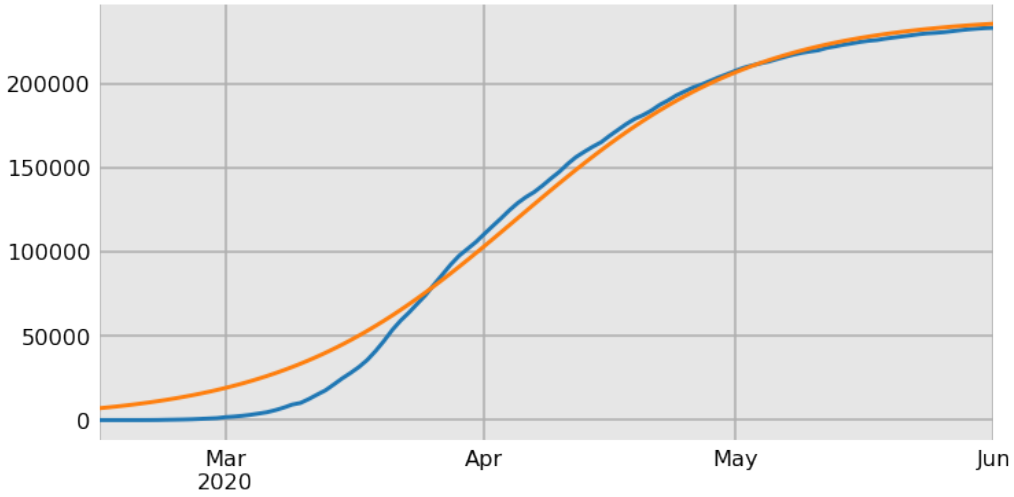


To better understand the logistic function, we define a function below that accepts a Series of data and the three parameters for the logistic function. It then plots the given data against the values calculated by the logistic function. We can use it to experiment with different parameter values until you a curve that closely resembles the one from Italy is produced.

[12]:
```
def logistic_guess_plot(s, L, x0, k):
    """
    Plots the given series of data along with its
    estimated values using the logistic function.

    Parameters
    ----------
    s : Series of actual data

    L, x0, k : same as above

    Returns
    -------
    None
    """
    x = np.arange(len(s))
    y = logistic_func(x, L, x0, k)
    s_guess = pd.Series(y, index=s.index)
    s.plot()
    s_guess.plot()
```

By looking at the cases in Italy, we guess that the upper asymptote is 240,000 and the midpoint is reached at about 50 days. For the growth rate, I experimented with a few values, stopping at 0.07 which made a very good fit.

[13]:
```
logistic_guess_plot(italyc, 240_000, 50, 0.07)
```



6.3 Estimating logistic function parameters

Now that we have the function that follows the shape of our data, we begin the process of finding the best parameters for the curve. We begin by smoothing our data and only letting our model see data through April 1.

[14]:
```
from solutions import smooth
last_date = '2020-04-01'
y = italyc.loc[:last_date]
y_smooth = smooth(y, n=15)
y_smooth.head()
```

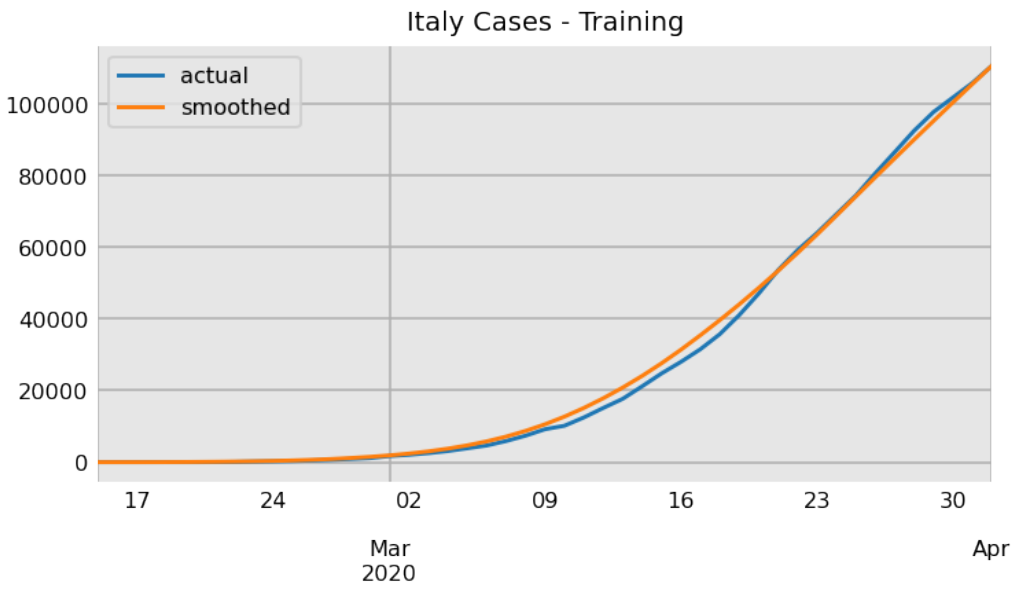
[14]:
```
2020-02-15     0.000000
2020-02-16     0.000000
2020-02-17     0.000000
2020-02-18     8.144961
2020-02-19    27.319120
dtype: float64
```

[15]:
```
y_smooth.tail()
```

[15]:
```
2020-03-28     89992.495694
2020-03-29     95198.975637
2020-03-30    100365.643598
2020-03-31    105491.416174
2020-04-01    110574.000000
dtype: float64
```

The smoothed data is plotted alongside the actual data below.

[16]:
```
y.plot(label='actual', title="Italy Cases - Training");
y_smooth.plot(label='smoothed').legend();
```



Parameter bounds

The parameter bounds are important to get correct as the trajectory of the curve can vary widely with just small deviations. The parameter `L` represents the upper asymptote, or the maximum number of cases. Given the current state of the data on April 1, it's difficult to place a reasonable bound for it. We can choose the last known value as the minimum and a very large number as the maximum.

The parameter `x0` represents the midpoint (also known as the inflection point) and is the number of days since the start where half of the cases have occurred.

The parameter `k` represents the growth rate, or the steepness of the curve. A larger `k` creates a steeper curve, and a smaller `k` creates a flatter curve. It can be helpful to create multiple curves for different values of `k` while keeping `L` and `x0` constant to better understand the logistic function.

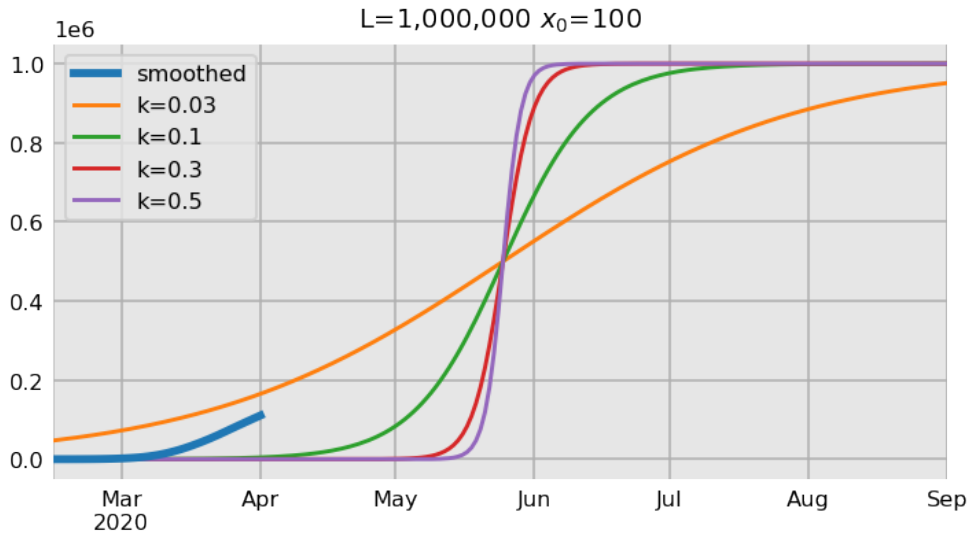
Here, we write a function that accepts the data to be modeled as a Series, a list of potential values of `k`, a single value for `L`, and a single value for `x0`. It plots the logistic curves starting on the same date as the passed data and continuing for twice the amount of `x0`.

[17]:
```
def plot_ks(s, ks, L, x0):
    """
    Plots the data to be modeled along with the logistic curves
    for each of the given ks on the same plot. This function
    helps find good bounds for k in curve_fit.

    Parameters
    ----------
    s : data to be modeled

    ks : list of floats for different values of the growth rate

    L : Upper asymptote

    x0 : horizontal shift
```

```
    Returns
    -------
    None
    """
    start = s.index[0]
    index = pd.date_range(start, periods=2 * x0)
    x = np.arange(len(index))
    s.plot(label="smoothed", lw=3, title=f"L={L:,} $x_0$={x0}", zorder=3)
    for k in ks:
        y = logistic_func(x, L, x0, k)
        y = pd.Series(y, index=index)
        y.plot(label=f"k={k}").legend()
```

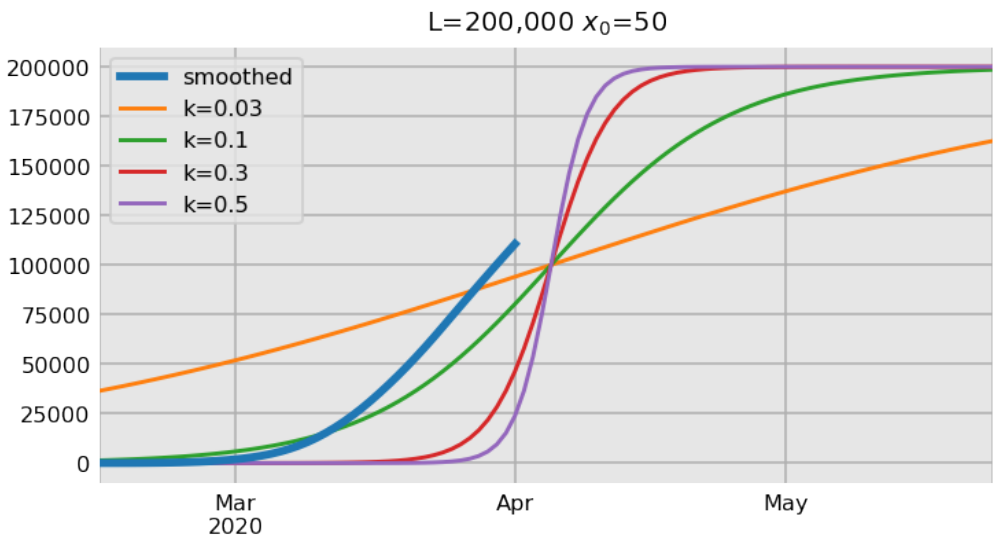
We'll make a few different calls to `plot_ks` pairing up very high and very low values of `L` and `x0` to see if we can triangulate the area where `k` might be. Below, we have a scenario where `L` is 1,000,000 and x_0 is 100 days. Comparing the known, smoothed data to the steepness of the other curves would make `k` less than 0.1 and perhaps greater than 0.01.

[18]:
```
from solutions import plot_ks
ks = [0.03, 0.1, 0.3, 0.5]
plot_ks(y_smooth, ks, 1_000_000, 100)
```



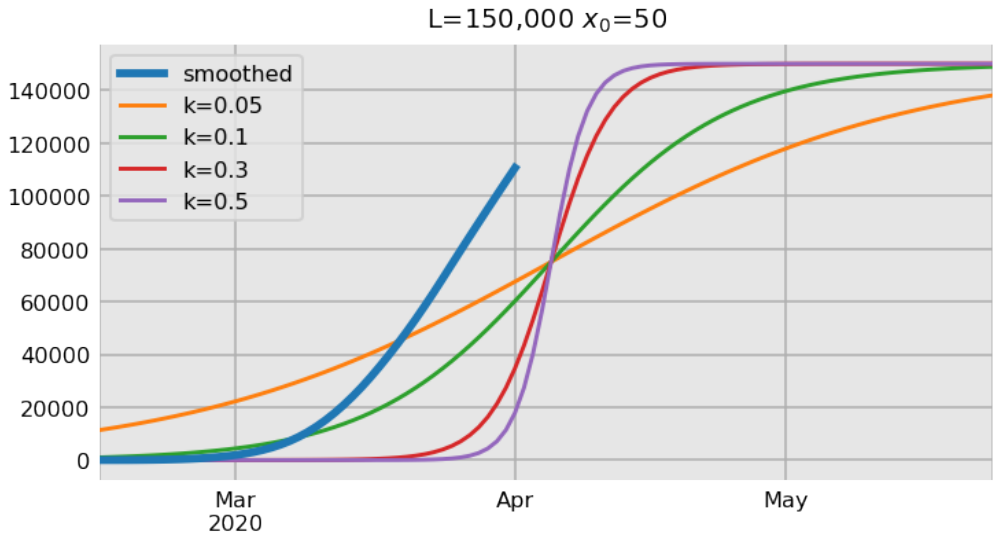
When using a shorter amount of time to the midpoint and an `L` of 200,000, the shape of the smoothed curve is closer to 0.1.

[19]:
```
ks = [0.03, 0.1, 0.3, 0.5]
plot_ks(y_smooth, ks, 200_000, 50)
```



If we assume just a small future increase in total cases, the smoothed slope is closer to 0.3.

[20]:
```
ks = [.05, 0.1, 0.3, 0.5]
plot_ks(y_smooth, ks, 150_000, 50)
```



From this analysis, it appears that `k` will almost certainly be less than 0.5. By definition, it must be positive (or else the curve would flip). Let's use a lower bound of 0.01. We set the upper bounds of `L` and `x0` to 1,000,000 and 150 and use the lower bound as the initial guess.

[21]:
```
lower = y_smooth[-1], 20, 0.01
upper = 1_000_000, 150, 0.5
bounds = lower, upper
p0 = lower
```

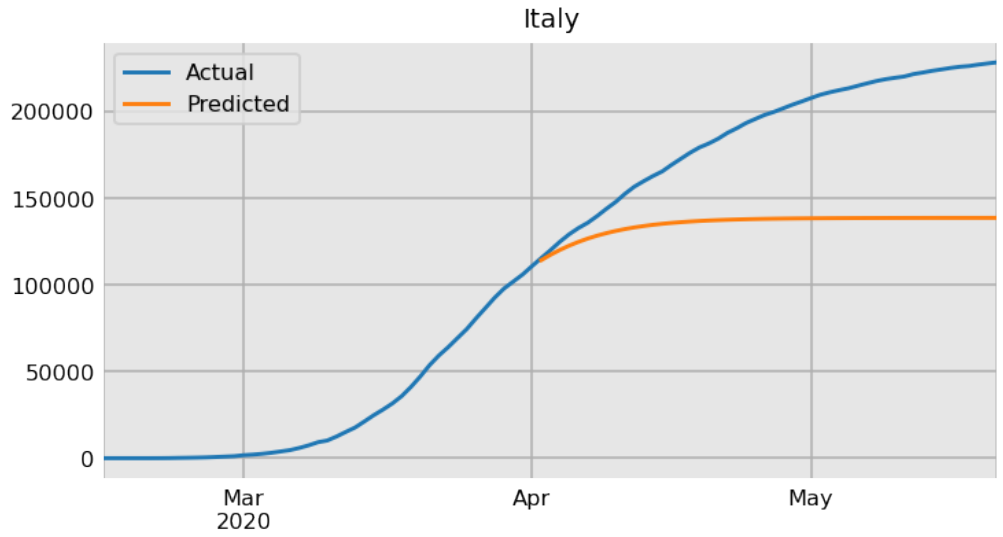
We use `least_squares` just as before to get the fitted parameter values.

[22]:
```
from solutions import logistic_func, optimize_func
from scipy.optimize import least_squares
y = y_smooth
x = np.arange(len(y))
res = least_squares(optimize_func, p0, args=(x, y, logistic_func), bounds=bounds)
L, x0, k = res.x
print(f'L = {L:,.0f}\nx0 = {x0:.0f}\nk = {k:.3f}')
```

```
L = 136,194
x0 = 38
k = 0.164
```

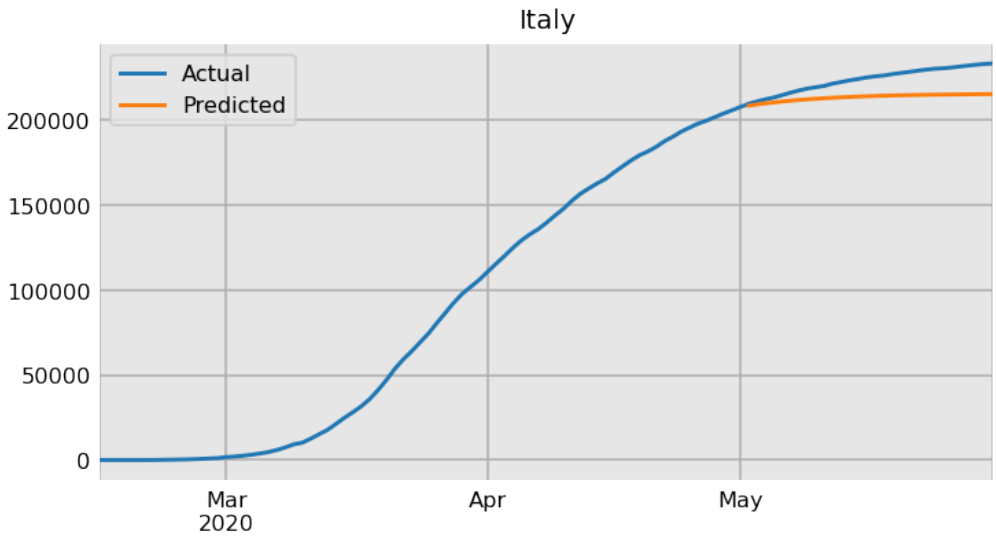
Let's use our handy function `predict_all` to plot the next 50 days with this model.

[23]:
```
from solutions import predict_all
predict_all(italyc, start_date=None, last_date="2020-04-01", n_pred=50, n_smooth=15,
            model=logistic_func, bounds=bounds, p0=p0, title="Italy");
```



Although we have the right shape for the curve, this prediction quickly fell off the target. Let's test out our model with more training data by allowing it to see data up to May 1. Unfortunately, the result is still poor.

[24]:
```
predict_all(italyc, start_date=None, last_date="2020-05-01", n_pred=30, n_smooth=15,
            model=logistic_func, bounds=bounds, p0=p0, title="Italy");
```



6.4 Generalized Logistic Function

Our first results using the logistic model do not look good as the curve is flattening much too fast. The basic form of the logistic function does not allow for asymmetry. The current form of the logistic function is perfectly symmetric - the first half of the curve is an exact mirror of the second half.

The overwhelming majority of areas have cases/deaths that slowly trail off towards a maximum. The current form of the logistic function cannot fit this kind of data. There exists a more broad generalized logistic function that is able to be tuned such that a long tail is possible. One form is written below.

$$f(x) = \frac{L}{(1 + e^{-k(x-x_0)})^{\frac{1}{v}}}$$

`L` and `k` represent the same values as before, the maximum of the curve and the growth rate. The new parameter `v` changes the symmetry of the curve and shifts the entire curve horizontally. The following summarizes `v`, which must be positive:

- `v < 1`
  - growth is faster before the midpoint
  - curve shifted to the right
- `v > 1`
  - growth is faster after the midpoint
  - curve shifted to the left

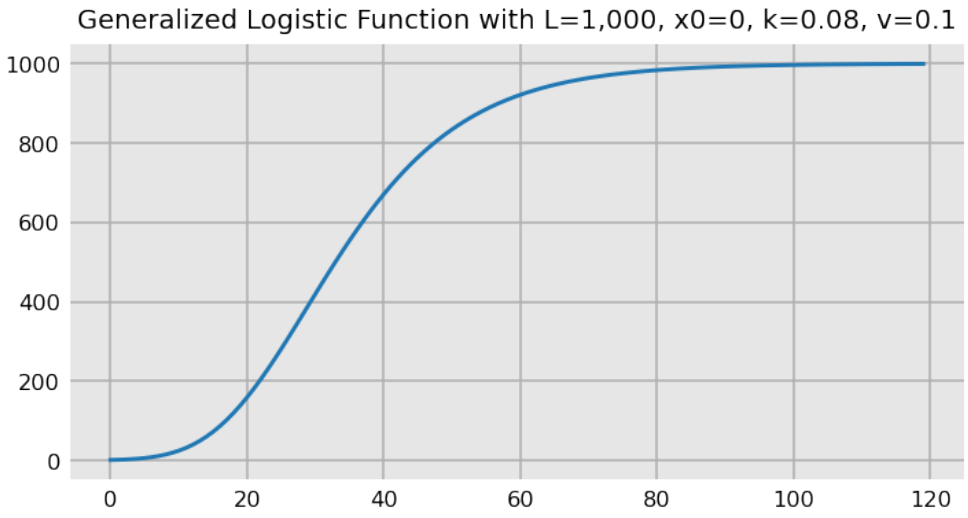
Let's create a function that computes the value for the generalized logistic function.

[25]:
```
def general_logistic(x, L, x0, k, v):
    return L / ((1 + np.exp(-k * (x - x0))) ** (1 / v))
```

Here, we choose a set of parameters that show our new S-curve. Take note that although `x0` is 0. The midpoint (`y=500`) occurs at about `x=35`.

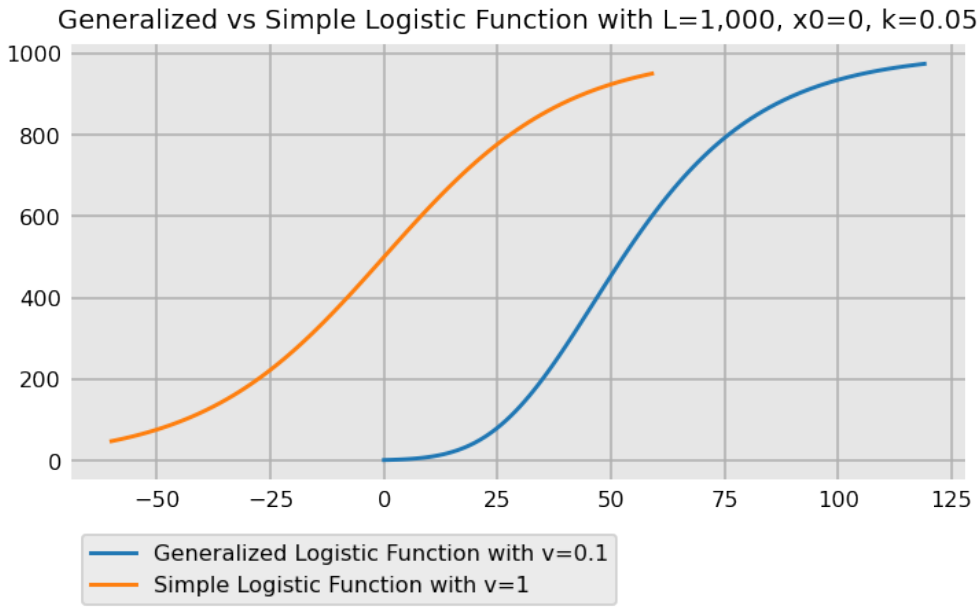
[26]:
```
L, x0, k, v = 1_000, 0, 0.08, 0.1
x = np.arange(120)
```

```
y = general_logistic(x, L, x0, k, v)
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title(f"Generalized Logistic Function with L={L:,}, x0={x0}, k={k}, v={v}");
```



When v equals 1, the equation simplifies to our previous logistic function which has its midpoint at x0. The same generalized logistic function with v set to 0.1 is plotted below along with a simple logistic function with v equal to 1. Notice how v shifts the generalized logistic function much further to the right even though they have the same values for L, x0, and k.

[27]:
```
# same as above
L, x0, k, v = 1_000, 0, 0.05, 0.1
x = np.arange(120)
y = general_logistic(x, L, x0, k, v)

# v set to 1 - simple logistic function
L, x0, k, v = 1_000, 0, 0.05, 1
x1 = np.arange(-60, 60)
y1 = general_logistic(x1, L, x0, k, v)

fig, ax = plt.subplots()
ax.plot(x, y, label='Generalized Logistic Function with v=0.1')
ax.plot(x1, y1, label='Simple Logistic Function with v=1')
ax.legend(bbox_to_anchor=(0, -.1), loc='upper left')
ax.set_title(f"Generalized vs Simple Logistic Function with L={L:,}, x0={x0},␣
 ↪k={k}");
```



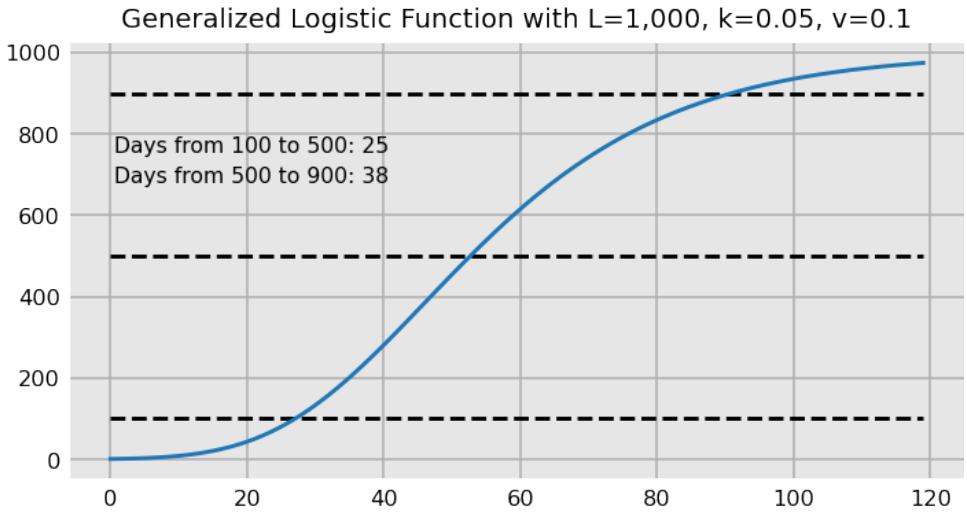
Plotting Asymmetry

To get a better understanding of the asymmetry created by the `v` parameter, the following function is defined which plots the generalized logistic function along with horizontal lines at the 10th, 50th, and 90th percent of `L`. It then calculates and displays the number of days it takes to go from the 10th to 50th percent and from the 50th to 90th percent. When `v` is less than 1, it will take less days to go from the 10th to 50th percent than it does from the 50th to 90th percent. The opposite is true when `v` is greater than 1.

[28]:
```
def plot_asymmetry(x, L, x0, k, v):
    y = general_logistic(x, L, x0, k, v)
    fig, ax = plt.subplots()
    ax.plot(x, y)
    low, mid, high = int(0.1 * L), int(0.5 * L), int(0.9 * L)
    ax.hlines([low, mid, high], x[0], x[-1], color='black', ls='--')

    days_to_10 = np.argmax(y > low)
    days_to_50 = np.argmax(y > mid)
    days_to_90 = np.argmax(y > high)
    days_10_to_50 = days_to_50 - days_to_10
    days_50_to_90 = days_to_90 - days_to_50
    ax.set_title(f"Generalized Logistic Function with L={L:,}, k={k}, v={v}")
    ax.text(0.05, 0.75, f'Days from {low} to {mid}: {days_10_to_50}',
            transform=ax.transAxes, size=8)
    ax.text(0.05, 0.68, f'Days from {mid} to {high}: {days_50_to_90}',
            transform=ax.transAxes, size=8)
```

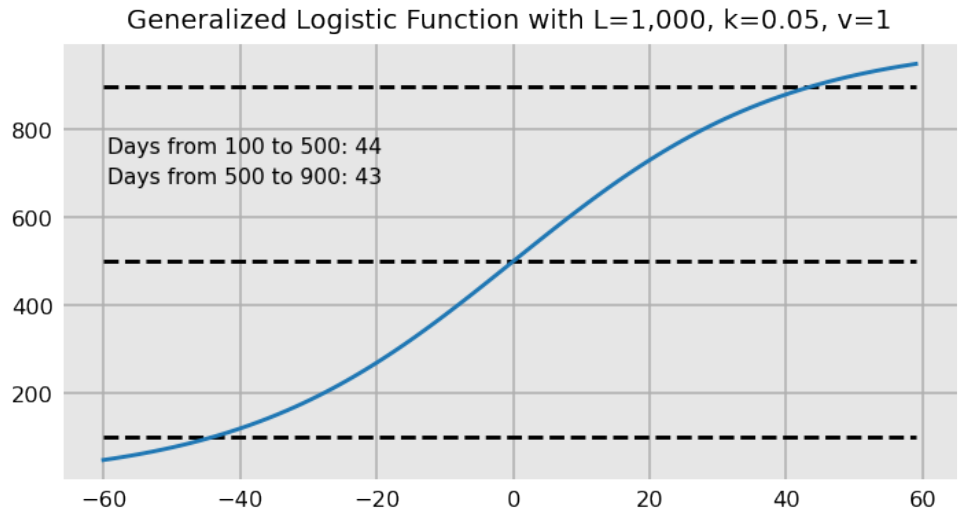
We will hold `L`, `x0`, and `k` constant and change `v` to show the asymmetry. Here, when `v` is 0.1 it takes 13 less days to go from 100 to 500, than it does from 500 to 900. This value of `v` would be used whenever a curve has a longer tail like the coronavirus case curve for most areas.

[29]:
```
x = np.arange(120)
plot_asymmetry(x, L=1000, x0=0, k=0.05, v=0.1)
```



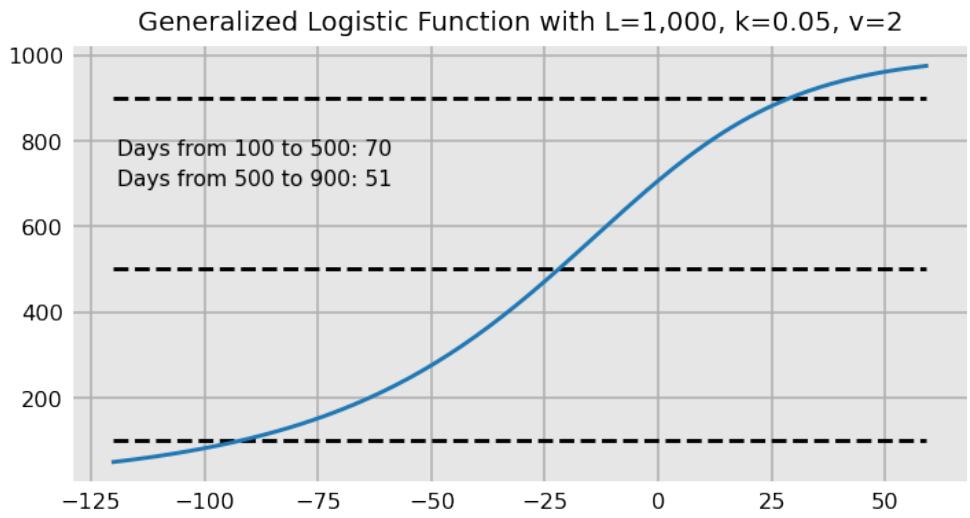
Here, we set `v` to 1 simplifying the logistic function so that it will be perfectly symmetric again. The one day difference observed below is due to rounding.

[30]:
```
x = np.arange(-60, 60)
plot_asymmetry(x, L=1000, x0=0, k=0.05, v=1)
```



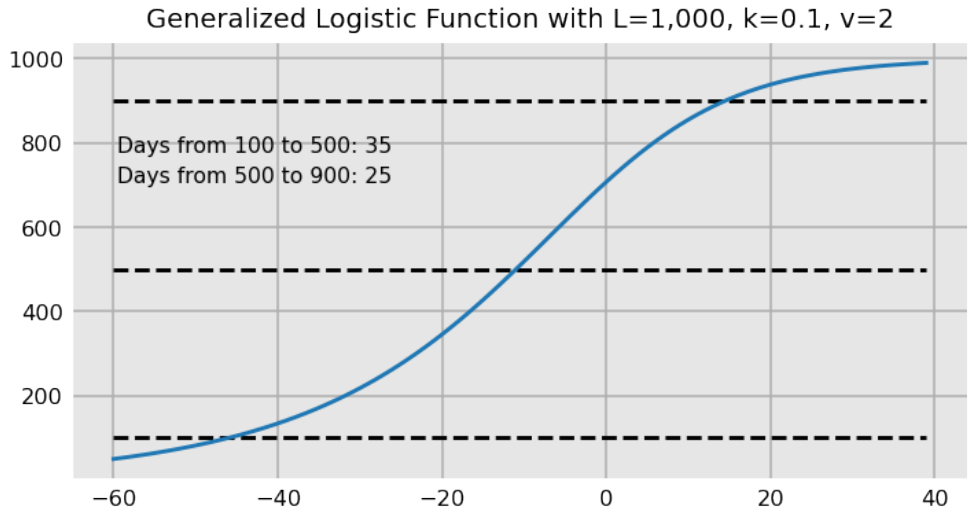
Here, we set the value of `v` to be 2, which has the effect of the curve reaching its right asymptote faster.

[31]:
```
x = np.arange(-120, 60)
plot_asymmetry(x, L=1000, x0=0, k=0.05, v=2)
```



The total time from 10th to 90th percent was nearly double (121 vs 63) when `v` was 2 as opposed to 0.1 in the first graph. The growth rate, `k`, is affected by the value of `v`. To decrease the total amount of time from the 10th to 90th percent, we increase `k` as `v` increases. Here, we set `k` to 0.1.

[32]:
```
x = np.arange(-60, 40)
plot_asymmetry(x, L=1000, x0=0, k=0.1, v=2)
```

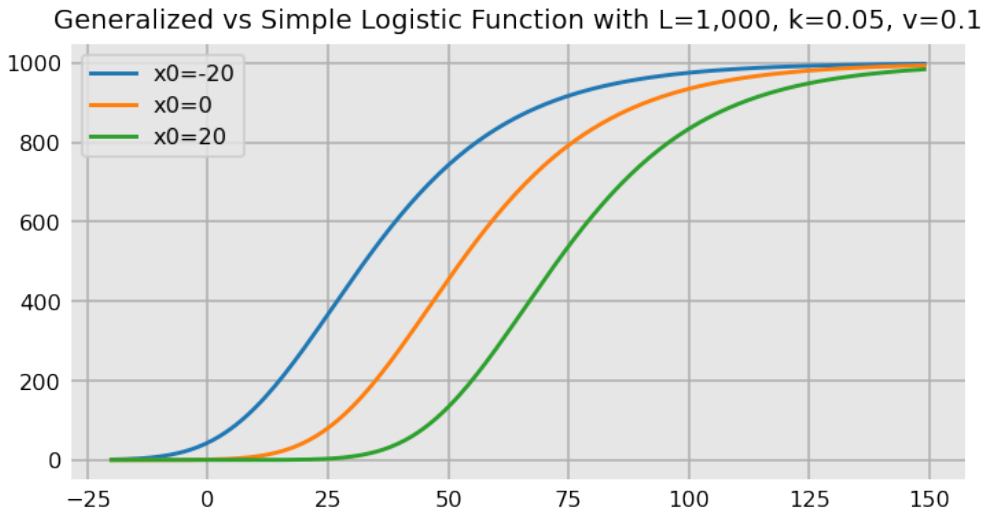


What does x0 represent now?

The parameter `x0` still shifts the data horizontally, but no longer represents the midpoint of the graph. Here, we plot different values of `x0` holding the other parameters constant. The curve is the exact same shape, just at a different location on the x-axis.

[33]:
```
L, k, v = 1_000, 0.05, 0.1
x = np.arange(-20, 150)

fig, ax = plt.subplots()
for x0 in [-20, 0, 20]:
```

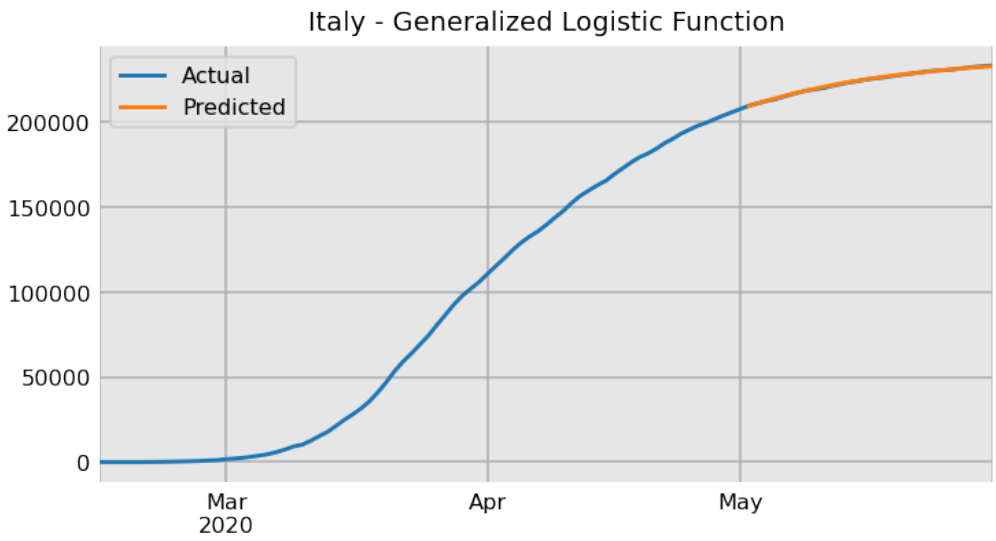
```
    y = general_logistic(x, L, x0, k, v)
    ax.plot(x, y, label=f'x0={x0}')
ax.legend()
ax.set_title(f"Generalized vs Simple Logistic Function with L={L:,}, k={k}, v={v}");
```



6.5 Predictions with the generalized logistic function

With our `predict_all` function, it's easy to plug in a new model and see the results. We just need to change the limits and set the bounds of the parameters. We use the same limits as before for `L` and `k`. We allow `x0` to be positive or negative as `v` also shifts the data horizontally. For `v`, some experimentation was needed to find the feasible limits.

[34]:
```
last_date = "2020-04-01"
L_min, L_max = italyc[last_date], 1_000_000
x0_min, x0_max = -50, 50
k_min, k_max = 0.01, 0.5
v_min, v_max = 0.01, 2
lower = L_min, x0_min, k_min, v_min
upper = L_max, x0_max, k_max, v_max
bounds = lower, upper
p0 = L_max, 0, 0.1, 0.1

params, y_pred = predict_all(italyc, start_date=None, last_date="2020-05-01",
                             n_smooth=15, n_pred=30, model=general_logistic,
                             bounds=bounds, p0=p0,
                             title="Italy - Generalized Logistic Function");
```



The new model provides an excellent fit and predicts future cases almost perfectly. Let's view the optimal parameters.

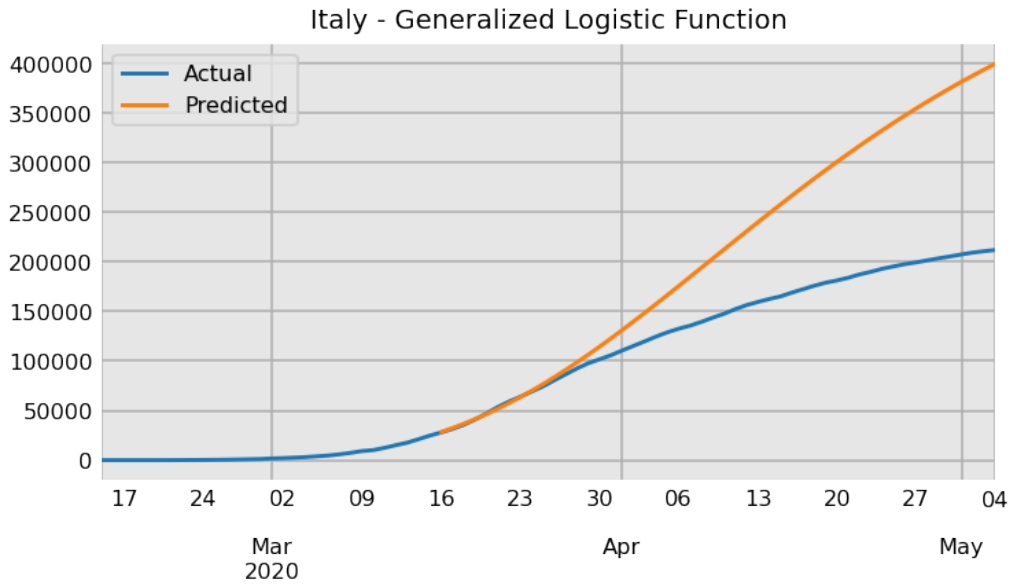
[35]:
```
params
```

[35]:
```
array([ 2.36841824e+05, -3.61763200e+01,  5.86413407e-02,  1.00000000e-02])
```

Have we built the perfect model?

Although our model provides a very good fit, we won't be able to reproduce this performance on all other countries or other time periods. For instance, our model performs worse if we only allow it to see data only up through March 15.

[36]:
```
predict_all(italyc, start_date=None, last_date="2020-03-15", n_smooth=15,
            n_pred=50, model=general_logistic, bounds=bounds, p0=p0,
            title="Italy - Generalized Logistic Function");
```

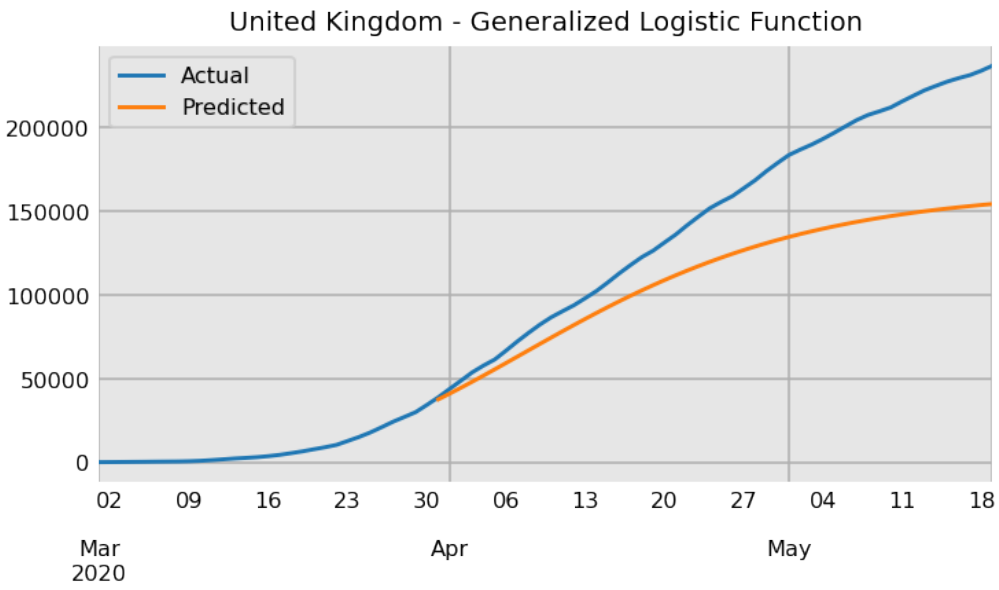


6.6 Modeling other countries

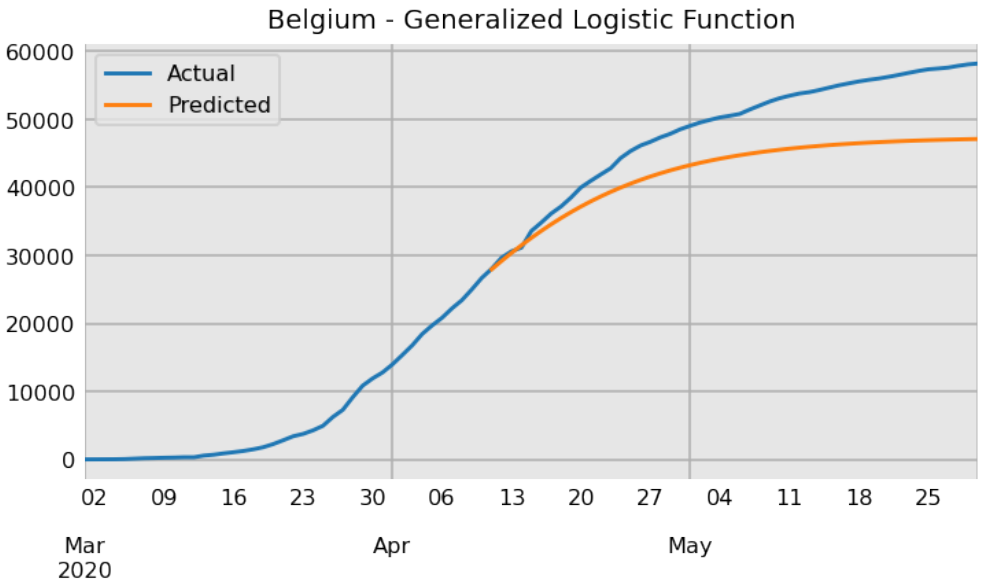
Let's model the cases from a few more countries, defining a function to automate the process for us. It chooses the initial guess and upper bound for `L` to be 5 and 1,000 times the last value.

[37]:
```
def model_country(data, name, start_date, last_date):
    s = data['world_cases'][name]
    L_min, L_max = s.iloc[0], s.iloc[-1] * 1000
    x0_min, x0_max = -50, 50
    k_min, k_max = 0.01, 0.5
    v_min, v_max = 0.01, 2
    lower = L_min, x0_min, k_min, v_min
    upper = L_max, x0_max, k_max, v_max
    bounds = lower, upper
    p0 = L_min * 5, 0, 0.1, 0.1
    predict_all(s, start_date=start_date, last_date=last_date, n_smooth=15,
                n_pred=50, model=general_logistic, bounds=bounds, p0=p0,
                title=f"{name} - Generalized Logistic Function");
```

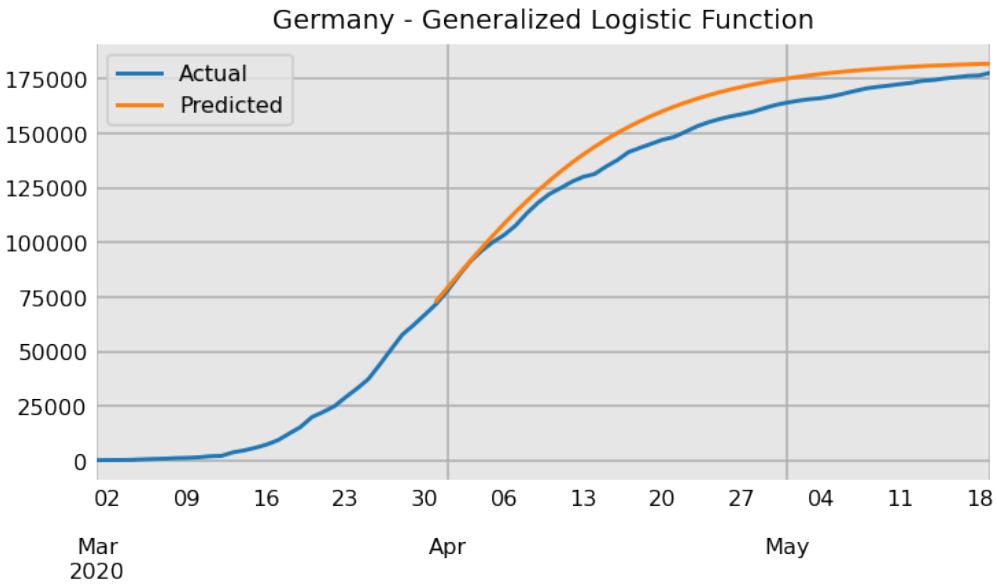
[38]:
```
model_country(data, "United Kingdom", "2020-03-01", "2020-03-30")
```



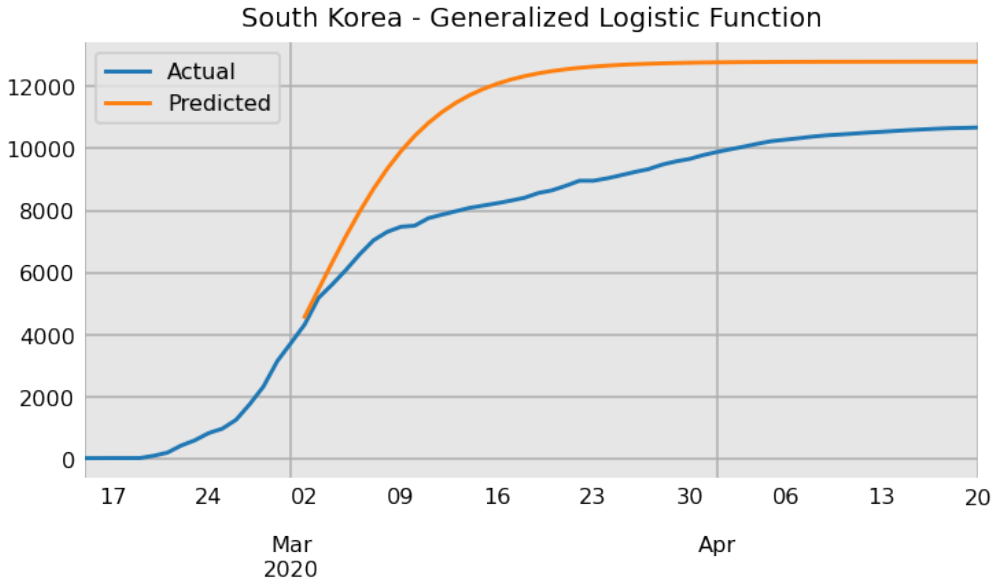
[39]:
```
model_country(data, "Belgium", "2020-03-01", "2020-04-10")
```



[40]:
```
model_country(data, "Germany","2020-03-01", "2020-03-30")
```



[41]:
```
model_country(data, "South Korea","2020-02-15", "2020-03-1")
```



While a single generalized logistic function works fairly well for modeling a single wave with exponential growth and decline, it will need to be modified in order to handle new waves of the virus. We'll see an approach for modeling this in the next chapter.

Chapter 7

Modeling New Waves

Unfortunately, viruses won't adhere to the exponential decline portion of our model and cease to spread. Viruses typically have multiple periods of exponential growth and decline. Our current generalized logistic function can only capture a single "wave" of growth and decline.

7.1 Limit the data

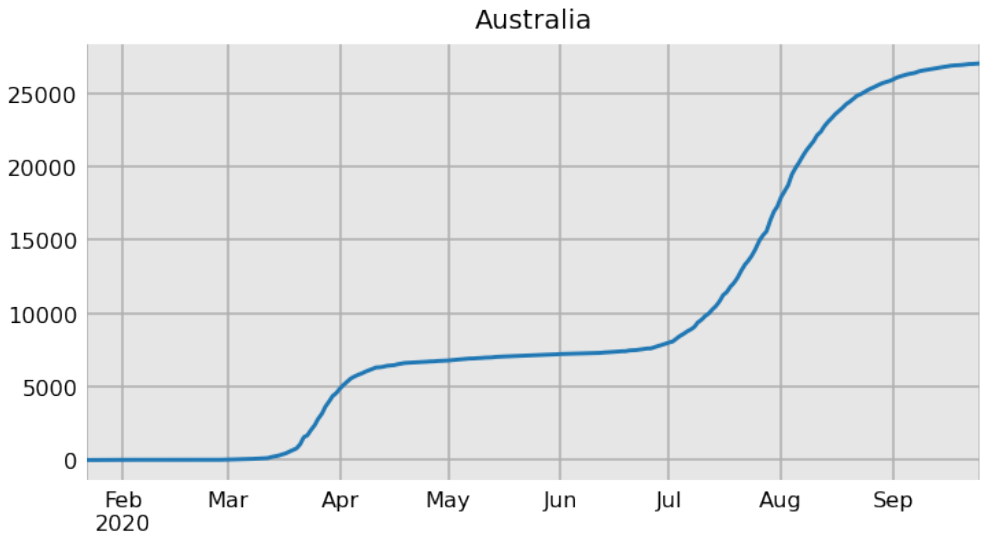
One solution to this problem is to limit the number of previous days that our model trains on. For instance, we could build a model based on just the last 50 days worth of data. The model simply "forgets" about all data before this period, making it able to continually capture new waves in the future regardless of how many occur.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from prepare import PrepareData
from solutions import predict_all
plt.style.use('dashboard.mplstyle')

def general_logistic(x, L, x0, k, v):
    return L / ((1 + np.exp(-k * (x - x0))) ** (1 / v))
```

Concrete example

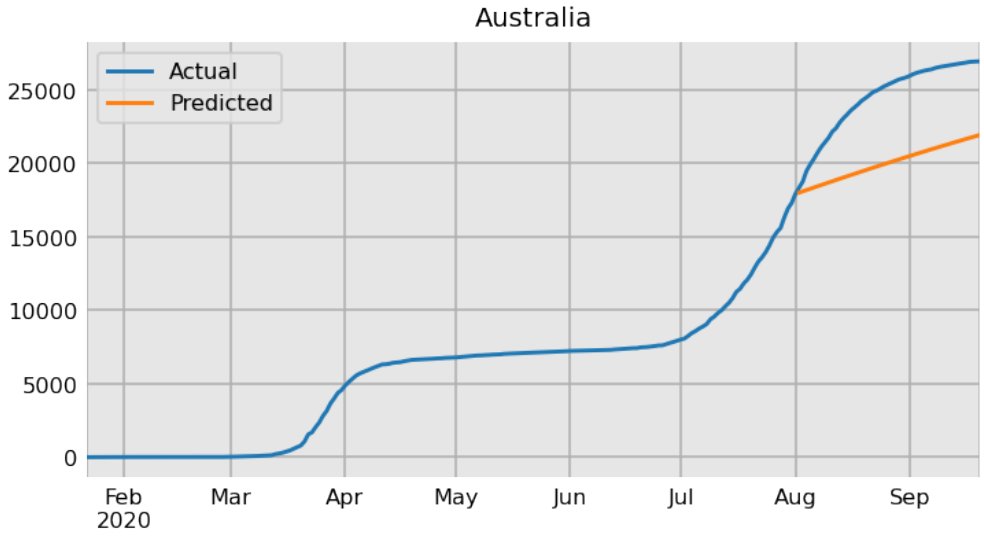
Let's walk through a concrete example to see how this works. Take a look at the cases from Australia through September 25, 2020. You'll notice two distinct waves, one beginning in the second half of March and another beginning in July.

[2]:
```
data = PrepareData(download_new=False).run()
area = "Australia"
cases = data['world_cases'][area][:'2020-09-25']
cases.plot(title=area);
```



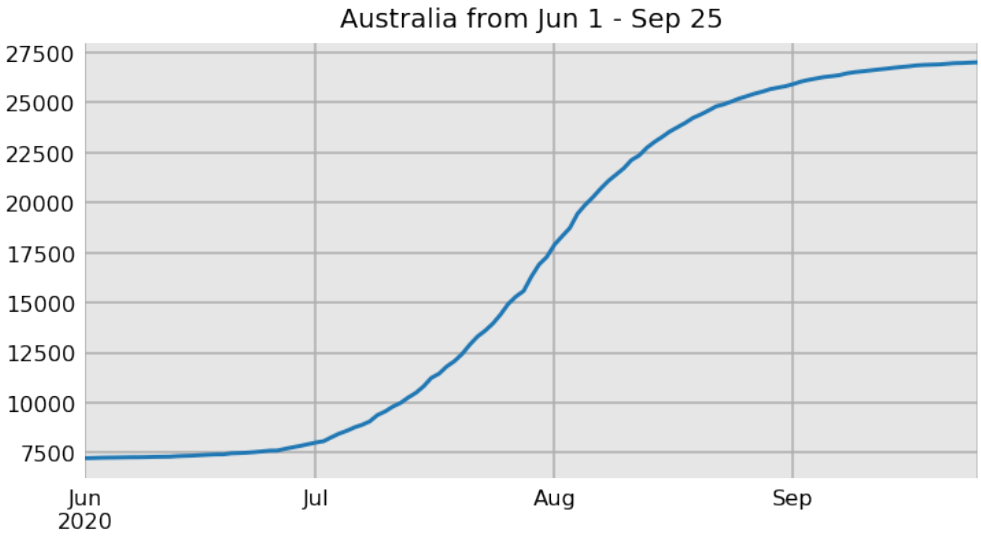
Attempting to model all of this data using the generalized logistic model yields poor results. Here, we use August 1st as the last day of training data.

[3]:
```
last_date = "2020-08-01"
L_min, L_max = cases[last_date], 100_000
x0_min, x0_max = -50, 50
k_min, k_max = 0.01, 0.5
v_min, v_max = 0.01, 2
lower = L_min, x0_min, k_min, v_min
upper = L_max, x0_max, k_max, v_max
bounds = lower, upper
p0 = L_min * 2, 0, 0.1, 0.1
predict_all(cases, start_date=None, last_date=last_date, n_smooth=15, n_pred=50,
            model=general_logistic, bounds=bounds, p0=p0, title="Australia");
```



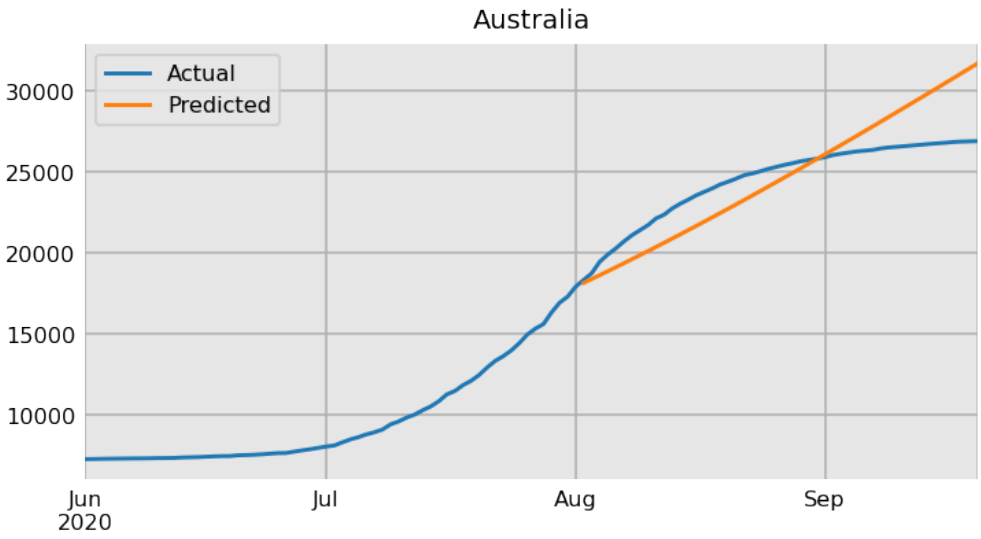
If we select data beginning after the first wave, then the generalized logistic model should work.

[4]:
```
start_date = '2020-06-01'
cases[start_date:].plot(title='Australia from Jun 1 - Sep 25');
```



Note, that this is the first time we've used something other than `None` for the `start_date`.

[5]:
```
predict_all(cases, start_date=start_date, last_date=last_date, n_smooth=15,␣
↪n_pred=50,
            model=general_logistic, p0=p0, bounds=bounds, title="Australia");
```



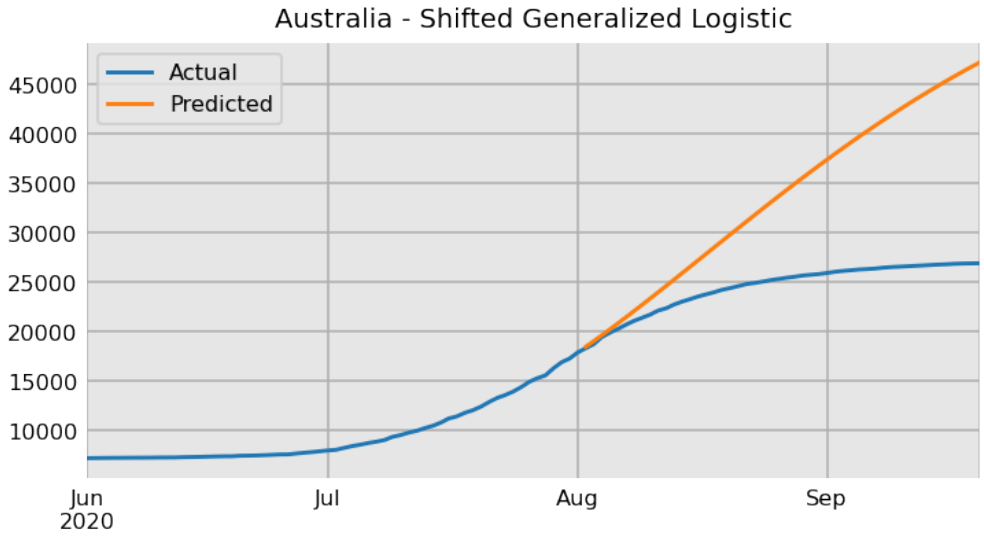
What went wrong?

Unfortunately, our model produced an obviously wrong curve. The issue here is that the left tail of the curve is very flat, but does not begin at 0 like the initial data. Our model has the ability to shift the curve horizontally using `v`, but not vertically. Let's add a new parameter, `s`, to the model that shifts the data vertically.

$$f(x) = \frac{L - s}{(1 + e^{-k(x - x_0)})^{\frac{1}{v}}} + s$$

Here, we define a function to calculate the values with this new model. We'll allow `s` to range from 0 up to the value of the last day.

[6]:
```
def general_logistic_shift(x, L, x0, k, v, s):
    return (L - s) / ((1 + np.exp(-k * (x - x0))) ** (1 / v)) + s

start_date = "2020-6-1"
last_date = "2020-08-01"
k_min, k_max = 0.01, 0.5
v_min, v_max = 0.01, 2
L_min, L_max = cases[last_date], 100_000

s_min, s_max = 0, cases[-1]
x0_min, x0_max = -50, 50
lower = L_min, x0_min, k_min, v_min, s_min
upper = L_max, x0_max, k_max, v_max, s_max
bounds = lower, upper
p0 = L_min * 5, 0, 0.1, 0.1, s_min
params, y_pred = predict_all(cases, start_date=start_date, last_date=last_date,
                             n_smooth=15, n_pred=50, model=general_logistic_shift,
                             p0=p0, bounds=bounds,
            title="Australia - Shifted Generalized Logistic");
```



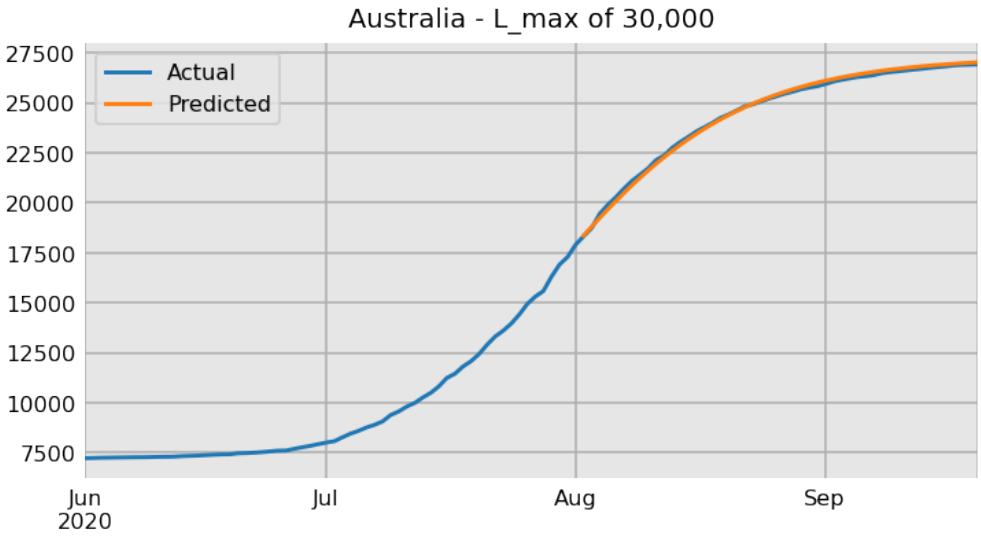
While the model did not make an accurate prediction, it was able to capture the shape of the curve. Below we output the fitted parameter values.

7.2 Defining functions to create the limits and bounds

The upper limit of the curve (`L_max`) and its initial guess can have great influence on the trajectory of the fitted curve. For instance, if we change the maximum value of L from 100,000 to 30,000, we get a

drastically different curve.

[7]:
```
L_max = 30_000
lower = L_min, x0_min, k_min, v_min, s_min
upper = L_max, x0_max, k_max, v_max, s_max
bounds = lower, upper
p0 = L_max, 0, 0.1, 0.1, 0
predict_all(cases, start_date=start_date, last_date=last_date, n_smooth=15,
            n_pred=50, model=general_logistic_shift, p0=p0, bounds=bounds,
            title="Australia - L_max of 30,000");
```



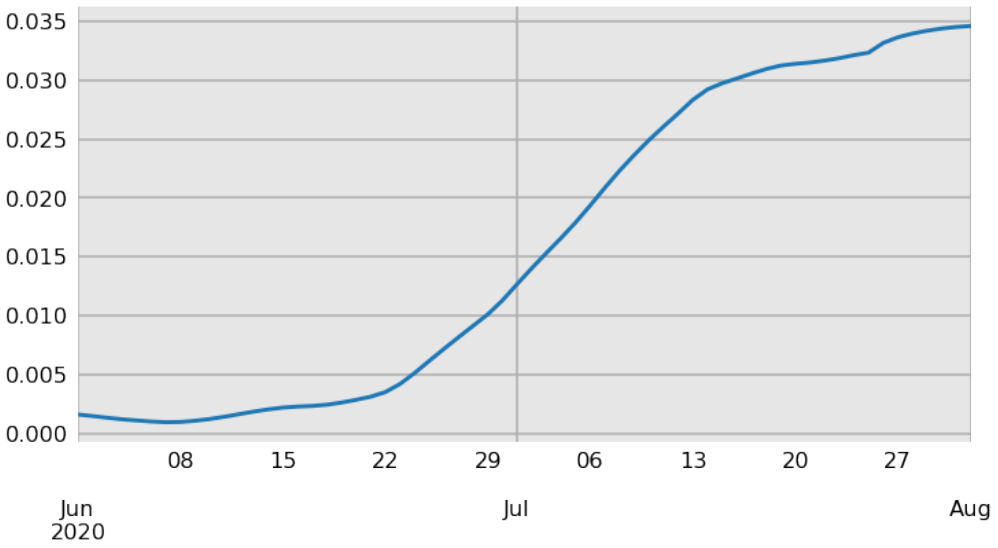
Of course, there is no way to know ahead of time which value for `L_max` will show superior results, but we still need to have a systematic way of choosing a value for it and its initial guess for every area. A simple idea involves finding the approximate daily percentage increase of the cumulative total on the last known date. We can use it as a rate of growth for the next 30 to 100 days. Let's see this on the smoothed data by calculating the daily percent change on the cumulative total.

[8]:
```
from solutions import smooth
s_smooth = smooth(cases[:last_date], n=15)
s_smooth_pct = s_smooth.pct_change()
s_smooth_pct.tail()
```

[8]:
```
2020-07-28    0.033968
2020-07-29    0.034210
2020-07-30    0.034395
2020-07-31    0.034530
2020-08-01    0.034618
dtype: float64
```

Plotting this data shows a better picture of the percentage growth.

[9]:
```
s_smooth_pct[start_date:].plot();
```



Australia shows about 3.5% growth in the last day. Using this value of growth for the next 50 days would yield about 100,000 cases as the upper bound. Making an estimation like this is just an alternative and simpler way of using exponential growth.

[10]:
```
s_smooth[-1] * (1 + s_smooth_pct[-1]) ** 50
```

[10]:
```
98115.40942084325
```

The function below accepts the smoothed series of data, finds the last percentage change and uses it to estimate the L bounds guess, and the midpoint between the two, which we'll use as the initial guess. In the event that an area has 0 cases, the percent change will return a missing value, so we set the initial guess to 0.

[11]:
```
def get_L_limits(s, n1, n2):
    """
    Finds the min and max bounds for L and its initial guess

    Parameters
    ----------
    s : smoothed Series

    n1, n2 : min and max days of exponential growth

    Returns
    -------
    three-item tuple - min/max L bounds and initial
    """
    last_val = s[-1]
    last_pct = s.pct_change()[-1] + 1
    L_min = last_val * last_pct ** n1
    L_max = last_val * last_pct ** n2 + 1
    L0 = (L_max - L_min) / 2 + L_min
    if np.isnan(L_min):
```

```
        L_min, L_max, L0 = 0, 1, 0
    return L_min, L_max, L0
```

Let's use this function to get the bounds and initial guess.

[12]:
```
get_L_limits(s_smooth, 5, 50)
```

[12]:
```
(21214.473332194153, 98116.40942084325, 59665.4413765187)
```

We can write one more function to return the bounds for all of the parameters along with the initial guess.

[13]:
```
def get_bounds_p0(s, n1=5, n2=50):
    """
    Finds the bounds and initial guesses for each
    parameter of the shifted logistic function

    Parameters
    ----------
    s : smoothed Series

    n1, n2 : min and max days of exponential growth

    Returns
    -------
    two-item tuple - bounds and p0
    """
    L_min, L_max, L0 = get_L_limits(s, n1, n2)
    x0_min, x0_max = -50, 50
    k_min, k_max = 0.01, 0.1
    v_min, v_max = 0.01, 2
    s_min, s_max = 0, s.iloc[-1] + 0.01
    s0 = s_max / 2
    lower = L_min, x0_min, k_min, v_min, s_min
    upper = L_max, x0_max, k_max, v_max, s_max
    bounds = lower, upper
    p0 = L0, 0, 0.1, 0.1, s0
    return bounds, p0
```

Let's get the bounds and initial parameter guess and output their values.

[14]:
```
bounds, p0 = get_bounds_p0(s_smooth)
bounds
```

[14]:
```
((21214.473332194153, -50, 0.01, 0.01, 0),
 (98116.40942084325, 50, 0.1, 2, 17895.01))
```

[15]:
```
p0
```

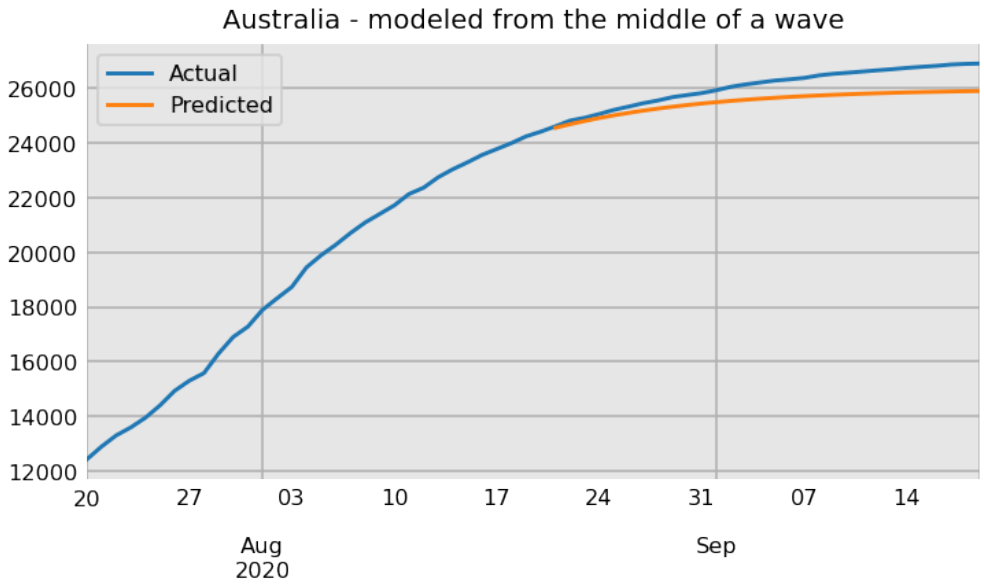
[15]:
```
(59665.4413765187, 0, 0.1, 0.1, 8947.505)
```

7.3 Finding new waves

Not every country will have perfectly separated waves like Australia. Fortunately, it's not necessary to pick the exact start date of a new wave. Our model is now capable of shifting vertically and horizontally, so will be able to capture the beginning, middle, and tail of the curves regardless of where the starting point is.

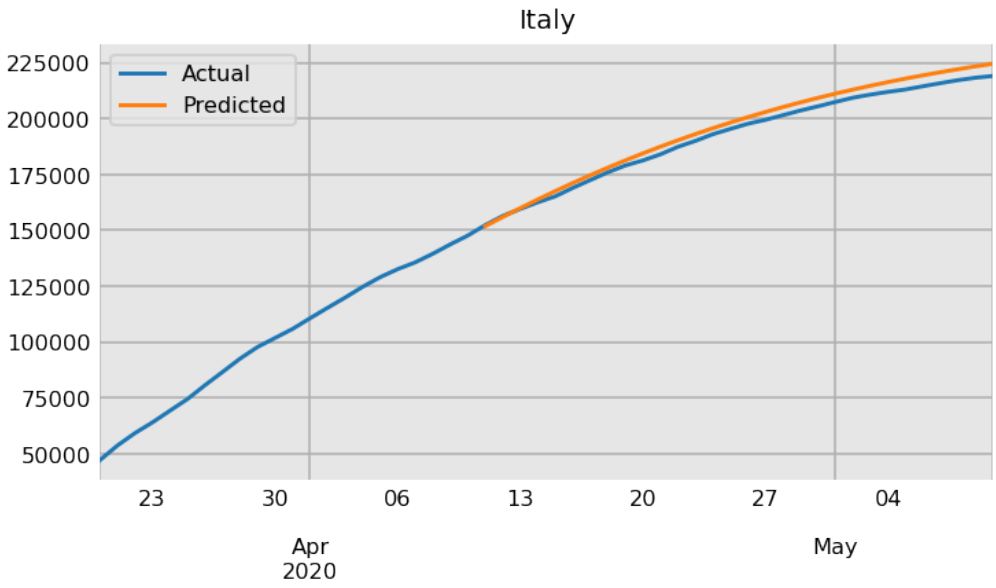
Here, we use a start date of July 20, well past the beginning of the initial exponential increase in Australia. We allow the model to see data up to August 20, when it had entered into exponential decline. This time the model makes an accurate prediction. Modeling typically becomes easier as the curve approaches the tail.

[16]:
```
start_date = "2020-07-20"
last_date = "2020-08-20"
bounds, p0 = get_bounds_p0(cases[:last_date])
params = predict_all(cases, start_date=start_date, last_date=last_date, n_smooth=15,
                     n_pred=30, model=general_logistic_shift, bounds=bounds, p0=p0,
                     title="Australia - modeled from the middle of a wave")
```



Here, we select cases in Italy from March 20th to April 10th as our training data and build a model to make predictions over the next 30 days. Again, our model is capable of fitting data beginning at any point during its wave.

[17]:
```
area = 'Italy'
start_date = '2020-03-20'
last_date='2020-04-10'
cases = data['world_cases'][area]
bounds, p0 = get_bounds_p0(cases[:last_date])
params = predict_all(cases, start_date=start_date, last_date=last_date, n_smooth=15,
                     n_pred=30, model=general_logistic_shift, p0=p0, bounds=bounds,
                     title=area)
```



7.4 Summarizing the final model

The final model that we'll use for predicting cases is a generalized logistic function with upper asymptote `L`, growth rate `k`, horizontal shift `x0`, asymmetric control `v`, and vertical shift `s`. The data is smoothed beforehand using LOWESS and only recent data is used during training in order to capture new waves.

$$f(x) = \frac{L - s}{(1 + e^{-k(x - x_0)})^{\frac{1}{v}}} + s$$

Chapter 8

Encapsulation into Classes

In this chapter, we'll encapsulate all of our work from the smoothing and model building chapters into Python classes. Encapsulation is just a fancy word for bringing together (capturing in a capsule, if you will) data and functions that act together to complete a task.

8.1 The `CasesModel` class

We now write a class, `CasesModel`, that is responsible for performing nearly all tasks involving cases after downloading the data. For each area, it smooths, trains, and predicts the number of cases. It encapsulates all of the steps into a single class. Several of our previous functions (`smooth`, `get_bounds_p0`, `get_L_limits`, `predict`) have been rewritten as methods.

Once instantiated, the `run` method must be called to complete the smoothing, training, and predicting. When the `run` method is called, a number of dictionaries are created to hold DataFrames with the following data:

- smoothed data
- bounds
- initial parameter guess
- fitted parameters
- daily predicted cases
- cumulative predicted cases
- combined daily/cumulative actual and predicted cases
- combined daily/cumulative smoothed actual and predicted cases

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import least_squares
from statsmodels.nonparametric.smoothers_lowess import lowess
plt.style.use('dashboard.mplstyle')

GROUPS = 'world', 'usa'
KINDS = 'cases', 'deaths'
MIN_OBS = 15  # Minimum observations needed to make prediction
```

```
def general_logistic_shift(x, L, x0, k, v, s):
    return (L - s) / ((1 + np.exp(-k * (x - x0))) ** (1 / v)) + s

def optimize_func(params, x, y, model):
    y_pred = model(x, *params)
    error = y - y_pred
    return error

class CasesModel:
    def __init__(self, model, data, last_date, n_train, n_smooth,
                 n_pred, L_n_min, L_n_max, **kwargs):
        """
        Smooths, trains, and predicts cases for all areas

        Parameters
        ----------
        model : function such as general_logistic_shift

        data : dictionary of data from all areas - result of PrepareData().run()

        last_date : str, last date to be used for training

        n_train : int, number of preceding days to use for training

        n_smooth : integer, number of points used in LOWESS

        n_pred : int, days of predictions to make

        L_n_min, L_n_max : int, min/max number of days used to estimate L_min/L_max

        **kwargs : extra keyword arguments passed to scipy's least_squares function
        """
        # Set basic attributes
        self.model = model
        self.data = data
        self.last_date = self.get_last_date(last_date)
        self.n_train = n_train
        self.n_smooth = n_smooth
        self.n_pred = n_pred
        self.L_n_min = L_n_min
        self.L_n_max = L_n_max
        self.kwargs = kwargs

        # Set attributes for prediction
        self.first_pred_date = pd.Timestamp(self.last_date) + pd.Timedelta("1D")
        self.pred_index = pd.date_range(start=self.first_pred_date, periods=n_pred)

    def get_last_date(self, last_date):
        # Use the most current date as the last actual date if not provided
```

```
        if last_date is None:
            return self.data['world_cases'].index[-1]
        else:
            return pd.Timestamp(last_date)

    def init_dictionaries(self):
        # Create dictionaries to store results for each area
        # Executed first in `run` method
        self.smoothed = {'world_cases': {}, 'usa_cases': {}}
        self.bounds = {'world_cases': {}, 'usa_cases': {}}
        self.p0 = {'world_cases': {}, 'usa_cases': {}}
        self.params = {'world_cases': {}, 'usa_cases': {}}
        self.pred_daily = {'world_cases': {}, 'usa_cases': {}}
        self.pred_cumulative = {'world_cases': {}, 'usa_cases': {}}

        # Dictionary to hold DataFrame of actual and predicted values
        self.combined_daily = {}
        self.combined_cumulative = {}

        # Same as above, but stores smoothed and predicted values
        self.combined_daily_s = {}
        self.combined_cumulative_s = {}

    def smooth(self, s):
        s = s[:self.last_date]
        if s.values[0] == 0:
            # Filter the data if the first value is 0
            last_zero_date = s[s == 0].index[-1]
            s = s.loc[last_zero_date:]
            s_daily = s.diff().dropna()
        else:
            # If first value not 0, use it to fill in the
            # first missing value
            s_daily = s.diff().fillna(s.iloc[0])

        # Don't smooth data with less than MIN_OBS values
        if len(s_daily) < MIN_OBS:
            return s_daily.cumsum()

        y = s_daily.values
        frac = self.n_smooth / len(y)
        x = np.arange(len(y))
        y_pred = lowess(y, x, frac=frac, is_sorted=True, return_sorted=False)
        s_pred = pd.Series(y_pred, index=s_daily.index).clip(0)
        s_pred_cumulative = s_pred.cumsum()

        if s_pred_cumulative[-1]  == 0:
            # Don't use smoothed values if they are all 0
            return s_daily.cumsum()
```

```
        last_actual = s.values[-1]
        last_smoothed = s_pred_cumulative.values[-1]
        s_pred_cumulative *= last_actual / last_smoothed
        return s_pred_cumulative

    def get_train(self, smoothed):
        # Filter the data for the most recent to capture new waves
        return smoothed.iloc[-self.n_train:]

    def get_L_limits(self, s):
        last_val = s[-1]
        last_pct = s.pct_change()[-1] + 1
        L_min = last_val * last_pct ** self.L_n_min
        L_max = last_val * last_pct ** self.L_n_max + 1
        L0 = (L_max - L_min) / 2 + L_min
        return L_min, L_max, L0

    def get_bounds_p0(self, s):
        L_min, L_max, L0 = self.get_L_limits(s)
        x0_min, x0_max = -50, 50
        k_min, k_max = 0.01, 0.5
        v_min, v_max = 0.01, 2
        s_min, s_max = 0, s[-1] + 0.01
        s0 = s_max / 2
        lower = L_min, x0_min, k_min, v_min, s_min
        upper = L_max, x0_max, k_max, v_max, s_max
        bounds = lower, upper
        p0 = L0, 0, 0.1, 0.1, s0
        return bounds, p0

    def train_model(self, s, bounds, p0):
        y = s.values
        n_train = len(y)
        x = np.arange(n_train)
        res = least_squares(optimize_func, p0, args=(x, y, self.model),
                            bounds=bounds, **self.kwargs)
        return res.x

    def get_pred_daily(self, n_train, params):
        x_pred = np.arange(n_train - 1, n_train + self.n_pred)
        y_pred = self.model(x_pred, *params)
        y_pred_daily = np.diff(y_pred)
        return pd.Series(y_pred_daily, index=self.pred_index)

    def get_pred_cumulative(self, s, pred_daily):
        last_actual_value = s.loc[self.last_date]
        return pred_daily.cumsum() + last_actual_value
```

```
    def convert_to_df(self, gk):
        # convert dictionary of areas mapped to Series to DataFrames
        self.smoothed[gk] = pd.DataFrame(self.smoothed[gk]).fillna(0).astype('int')
        self.bounds[gk] = pd.concat(self.bounds[gk].values(),
                                    keys=self.bounds[gk].keys()).T
        self.bounds[gk].loc['L'] = self.bounds[gk].loc['L'].round()
        self.p0[gk] = pd.DataFrame(self.p0[gk],
                                   index=['L', 'x0', 'k', 'v', 's'])
        self.p0[gk].loc['L'] = self.p0[gk].loc['L'].round()
        self.params[gk] = pd.DataFrame(self.params[gk],
                                       index=['L', 'x0', 'k', 'v', 's'])
        self.pred_daily[gk] = pd.DataFrame(self.pred_daily[gk])
        self.pred_cumulative[gk] = pd.DataFrame(self.pred_cumulative[gk])

    def combine_actual_with_pred(self):
        for gk, df_pred in self.pred_cumulative.items():
            df_actual = self.data[gk][:self.last_date]
            df_comb = pd.concat((df_actual, df_pred))
            self.combined_cumulative[gk] = df_comb
            self.combined_daily[gk] = (df_comb.diff()
                                           .fillna(df_comb.iloc[0])
                                           .astype('int'))

            df_comb_smooth = pd.concat((self.smoothed[gk], df_pred))
            self.combined_cumulative_s[gk] = df_comb_smooth
            self.combined_daily_s[gk] = (df_comb_smooth.diff()
                                                       .fillna(df_comb.iloc[0])
                                                       .astype('int'))

    def run(self):
        self.init_dictionaries()
        for group in GROUPS:
            gk = f'{group}_cases'
            df_cases = self.data[gk]
            for area, s in df_cases.items():
                smoothed = self.smooth(s)
                train = self.get_train(smoothed)
                n_train = len(train)
                if n_train < MIN_OBS:
                    bounds = np.full((2, 5), np.nan)
                    p0 = np.full(5, np.nan)
                    params = np.full(5, np.nan)
                    pred_daily = pd.Series(np.zeros(self.n_pred),
                                           index=self.pred_index)
                else:
                    bounds, p0 = self.get_bounds_p0(train)
                    params = self.train_model(train, bounds=bounds,  p0=p0)
                    pred_daily = self.get_pred_daily(n_train, params).round(0)
                pred_cumulative = self.get_pred_cumulative(s, pred_daily)
```

```
                # save results to dictionaries mapping each area to its result
                self.smoothed[gk][area] = smoothed
                self.bounds[gk][area] = pd.DataFrame(bounds,
                                                     index=['lower', 'upper'],
                                                     columns=['L', 'x0', 'k',
                                                              'v', 's'])
                self.p0[gk][area] = p0
                self.params[gk][area] = params
                self.pred_daily[gk][area] = pred_daily.astype('int')
                self.pred_cumulative[gk][area] = pred_cumulative.astype('int')
            self.convert_to_df(gk)
        self.combine_actual_with_pred()

    def plot_prediction(self, group, area, **kwargs):
        group_kind = f'{group}_cases'
        actual = self.data[group_kind][area]
        pred = self.pred_cumulative[group_kind][area]
        first_date = self.last_date - pd.Timedelta(self.n_train, 'D')
        last_pred_date = self.last_date + pd.Timedelta(self.n_pred, 'D')
        actual.loc[first_date:last_pred_date].plot(label='Actual', **kwargs)
        pred.plot(label='Predicted').legend()
```

An instance of the `CasesModel` class is created below. It uses the 60 days leading up to November 5, 2020 as training data for the model and makes predictions for the next 30 days. If `last_date` is not provided, then the last date from the given data is used. The integers `L_n_min` and `L_n_max` are used to find the bounds of `L`.

[2]:
```
from prepare import PrepareData
data = PrepareData(download_new=False).run()
cm = CasesModel(model=general_logistic_shift, data=data, last_date='2020-11-05',
                n_train=60, n_smooth=15, n_pred=30, L_n_min=5, L_n_max=50)
```

The `run` method must be called in order to smooth, train, and predict. Executing the following cell took about 25 seconds on my machine, as it completed the process for all areas.

[3]:
```
cm.run()
```

```
/var/folders/0x/whw882fj4qv0czzqzrngxvdh0000gn/T/ipykernel_28553/3512878361.py:16:
RuntimeWarning: overflow encountered in power
  return (L - s) / ((1 + np.exp(-k * (x - x0))) ** (1 / v)) + s
```

Results

Let's take a look at all of the results which are stored as DataFrames within dictionaries with keys `world_cases` and `usa_cases`. The original unprocessed data is in the `data` attribute. We select the last five rows of the first 10 areas.

[4]:
```
cm.data['world_cases'].iloc[-5:, :10]
```

Country/Region	Afghanistan	Albania	Algeria	Andorra	Angola	Antigua and Barbuda	Argentina	Armenia	Australia	Austria
2021-11-16	156649	193856	208245	15929	64940	4122	5308781	329913	192845	981904
2021-11-17	156739	194472	208380	15972	64968	4129	5310334	330895	194112	996320
2021-11-18	156739	195021	208532	16035	64985	4129	5312089	331914	195617	1011465
2021-11-19	156812	195523	208695	16086	64997	4131	5313607	332713	196977	1027274
2021-11-20	156864	195988	208839	16086	65011	4135	5314702	333583	198442	1042571

The smoothed data is what is used for training and is therefore only calculated through the date we wish to make a prediction from.

[5]:
```
cm.smoothed['usa_cases'].iloc[-5:, :10]
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-01	194112	16618	0	245466	112110	945711	109563	72182	152	25000
2020-11-02	195684	17004	0	247143	113359	950790	112252	73278	152	25186
2020-11-03	197278	17393	0	248919	114651	955996	115056	74457	152	25373
2020-11-04	198893	17785	0	250794	115984	961330	117974	75717	152	25562
2020-11-05	200528	18182	0	252768	117360	966795	121006	77060	152	25753

The bounds for each of the parameters when fitting are below.

[6]:
```
cm.bounds['world_cases'].iloc[:, :10]
```

	Afghanistan		Albania		Algeria		Andorra		Angola	
	lower	**upper**	**lower**	**upper**	**lower**	**upper**	**lower**	**upper**	**lower**	**upper**
L	42314.00	47092.00	24805.00	54642.00	62059.00	81974.00	5493.00	10071.00	13643.00	40119.00
x0	-50.00	50.00	-50.00	50.00	-50.00	50.00	-50.00	50.00	-50.00	50.00
k	0.01	0.50	0.01	0.50	0.01	0.50	0.01	0.50	0.01	0.50
v	0.01	2.00	0.01	2.00	0.01	2.00	0.01	2.00	0.01	2.00
s	0.00	41814.01	0.00	22721.01	0.00	60169.01	0.00	5135.01	0.00	12102.01

The initial guess for each parameter:

[7]:
```
cm.p0['world_cases'].iloc[:, :10]
```

	Afghanistan	Albania	Algeria	Andorra	Angola	Antigua and Barbuda	Argentina	Armenia	Australia	Austria
L	44703.000	39723.000	72017.000	7782.000	26881.000	130.000	1584838.000	216086.000	27902.000	771185.000
x0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
k	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100
v	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.100
s	20907.005	11360.505	30084.505	2567.505	6051.005	65.005	608514.005	49781.505	13822.005	66257.505

The first five predicted values for daily cases:

[8]:
```
cm.pred_daily['usa_cases'].iloc[:5, :10]
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-06	1276	354	0	1196	1061	3776	2341	975	0	159
2020-11-07	1277	357	0	1201	1062	3763	2392	999	0	159
2020-11-08	1278	360	0	1205	1063	3749	2442	1023	0	159
2020-11-09	1278	362	0	1208	1064	3735	2491	1045	0	158
2020-11-10	1278	364	0	1211	1064	3719	2539	1066	0	158

The first five predicted values for cumulative cases:

[9]:
```
cm.pred_cumulative['usa_cases'].iloc[:5, :10]
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-06	201804	18536	0	253964	118421	970571	123347	78035	152	25912
2020-11-07	203081	18893	0	255165	119483	974334	125739	79034	152	26071
2020-11-08	204359	19253	0	256370	120546	978083	128181	80057	152	26230
2020-11-09	205637	19615	0	257578	121610	981818	130672	81102	152	26388
2020-11-10	206915	19979	0	258789	122674	985537	133211	82168	152	26546

The `combined_daily` attribute contains the actual and predicted values combined in a single DataFrame. Below, we have the last three days of actual data and the first three predicted values.

[10]:
```
cm.combined_daily['usa_cases'].loc['2020-11-03':'2020-11-08', :'Delaware']
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-03	1812	392	0	1679	878	5718	2562	985	0	115
2020-11-04	1423	413	0	815	1293	5193	2928	530	0	108
2020-11-05	1601	314	0	2135	1548	7077	3369	1687	0	219
2020-11-06	1276	354	0	1196	1061	3776	2341	975	0	159
2020-11-07	1277	357	0	1201	1062	3763	2392	999	0	159
2020-11-08	1278	360	0	1205	1063	3749	2442	1023	0	159

Similarly, the `combined_cumulative` dictionary holds the actual cumulative along with the predicted values.

[11]:
```
cm.combined_cumulative['usa_cases'].loc['2020-11-03':'2020-11-08', :'Delaware']
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-03	197504	17455	0	249818	114519	954525	114709	74843	152	25426
2020-11-04	198927	17868	0	250633	115812	959718	117637	75373	152	25534
2020-11-05	200528	18182	0	252768	117360	966795	121006	77060	152	25753
2020-11-06	201804	18536	0	253964	118421	970571	123347	78035	152	25912
2020-11-07	203081	18893	0	255165	119483	974334	125739	79034	152	26071
2020-11-08	204359	19253	0	256370	120546	978083	128181	80057	152	26230

The `combined_daily_s` and `combined_cumulative_s` have the smoothed actual values with the predicted values.

[12]:
```
cm.combined_daily_s['usa_cases'].loc['2020-11-03':'2020-11-08', :'Delaware']
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-03	1594	389	0	1776	1292	5206	2804	1179	0	187
2020-11-04	1615	392	0	1875	1333	5334	2918	1260	0	189
2020-11-05	1635	397	0	1974	1376	5465	3032	1343	0	191
2020-11-06	1276	354	0	1196	1061	3776	2341	975	0	159
2020-11-07	1277	357	0	1201	1062	3763	2392	999	0	159
2020-11-08	1278	360	0	1205	1063	3749	2442	1023	0	159

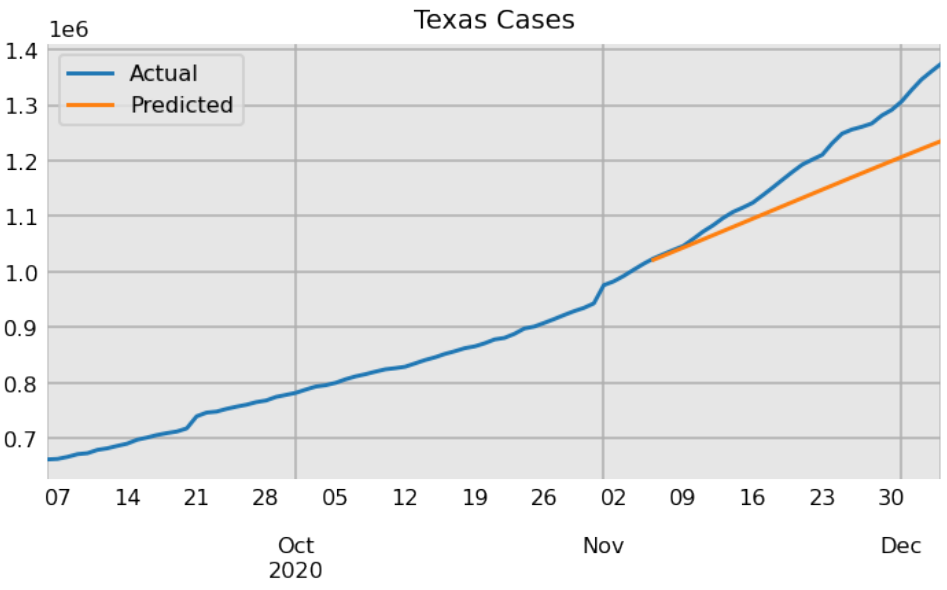
[13]:
```
cm.combined_cumulative_s['usa_cases'].loc['2020-11-03':'2020-11-08', :'Delaware']
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-03	197278	17393	0	248919	114651	955996	115056	74457	152	25373
2020-11-04	198893	17785	0	250794	115984	961330	117974	75717	152	25562
2020-11-05	200528	18182	0	252768	117360	966795	121006	77060	152	25753
2020-11-06	201804	18536	0	253964	118421	970571	123347	78035	152	25912
2020-11-07	203081	18893	0	255165	119483	974334	125739	79034	152	26071
2020-11-08	204359	19253	0	256370	120546	978083	128181	80057	152	26230

Plotting results

A `plot_prediction` method was also defined to visualize the actual vs predicted values for a given area.

[14]:
```
cm.plot_prediction('usa', 'Texas', title="Texas Cases")
```



8.2 Create Prediction for Deaths using Case Fatality Ratio

We've only discussed making predictions for cases and not deaths. We could use a generalized logistic function to model deaths and get reasonable results, though a simpler approach exists using the Case Fatality Ratio or CFR. The CFR is the fraction of cases resulting in death. Knowing this ratio can help us estimate the number of deaths that result from the number of recorded cases. While the CFR can

change substantially over the course of a pandemic as treatments change and more people have access to tests, it should be reasonably stable over a short amount of time.

From clinical data, deaths usually occur two to three weeks after the initial coronavirus infection. Using this knowledge, we can estimate the CFR based on historical cases and deaths. To calculate the CFR, we do the following:

- Find total cases between 15 and 45 days prior
- Find total deaths between 0 and 30 days prior
- Divide the total cases by the total deaths

The function below takes the unprocessed data and the last date of known values and then calculates the CFR for each area. A CFR of 0.005 is used for countries that have no cases in the last 30 days.

[15]:
```
def calculate_cfr(data, last_date):
    last_day_deaths = last_date
    first_day_deaths = last_date - pd.Timedelta('30D')
    last_day_cases = last_day_deaths - pd.Timedelta('15D')
    first_day_cases = last_day_cases - pd.Timedelta('30D')

    cfr = {}
    for group in GROUPS:
        deaths, cases = data[f'{group}_deaths'], data[f'{group}_cases']
        deaths_total = deaths.loc[last_day_deaths] - deaths.loc[first_day_deaths]
        cases_total = cases.loc[last_day_cases] - cases.loc[first_day_cases]
        cfr[group] = (deaths_total / cases_total).fillna(0.005)
    return cfr
```

Let's use the function to get the CFR for all areas and output some of the calculated values.

[16]:
```
last_date = pd.Timestamp('2020-11-05')
cfr = calculate_cfr(data, last_date)
cfr['world'].head(10).round(3)
```

[16]:
```
Country/Region
Afghanistan            0.059
Albania                0.026
Algeria                0.048
Andorra                0.010
Angola                 0.021
Antigua and Barbuda    0.000
Argentina              0.028
Armenia                0.023
Australia              0.019
Austria                0.015
dtype: float64
```

[17]:
```
cfr['usa'].head(10).round(3)
```

[17]:
```
Province_State
Alabama           0.014
```

```
Alaska            0.006
American Samoa    0.005
Arizona           0.019
Arkansas          0.023
California        0.017
Colorado          0.012
Connecticut       0.015
Cruise Ship       0.005
Delaware          0.023
dtype: float64
```

8.3 Create class to model deaths

We create another class, `DeathsModel`, to model the deaths of each area. It allows the user to set the `lag`, number of days between cases and deaths, and the `period`, number of days to tabulate the total cases/deaths for the CFR calculation. The `predict` method multiplies the CFR by the number of cases that happened `lag` days ago. For example, if we want to predict the number of deaths on November 6, we look back at the number of cases on October 22 (assuming the lag is 15) and multiply this number by the CFR of that area. To help get smoother results, we use a 7-day rolling average instead of the actual value as the final predicted value.

[18]:
```
class DeathsModel:
    def __init__(self, data, last_date, cm, lag, period):
        """
        Build simple model based on CFR to predict deaths for all areas

        Parameters
        ----------
        data : dictionary of data from all areas - result of PrepareData().run()

        last_date : str, last date to be used for training

        cm : CasesModel instance after calling `run` method

        lag : int, number of days between cases and deaths, used to calculate CFR

        period : int, window size of number of days to calculate CFR
        """
        self.data = data
        self.last_date = self.get_last_date(last_date)
        self.cm = cm
        self.lag = lag
        self.period = period
        self.pred_daily = {}
        self.pred_cumulative = {}

        # Dictionary to hold DataFrame of actual and predicted values
        self.combined_daily = {}
```

```
        self.combined_cumulative = {}

    def get_last_date(self, last_date):
        if last_date is None:
            return self.data['world_cases'].index[-1]
        else:
            return pd.Timestamp(last_date)

    def calculate_cfr(self):
        first_day_deaths = self.last_date - pd.Timedelta(f'{self.period}D')
        last_day_cases = self.last_date - pd.Timedelta(f'{self.lag}D')
        first_day_cases = last_day_cases - pd.Timedelta(f'{self.period}D')

        cfr = {}
        for group in GROUPS:
            deaths = self.data[f'{group}_deaths']
            cases = self.data[f'{group}_cases']
            deaths_total = deaths.loc[self.last_date] - deaths.loc[first_day_deaths]
            cases_total = cases.loc[last_day_cases] - cases.loc[first_day_cases]
            cfr[group] = (deaths_total / cases_total).fillna(0.01)
        return cfr

    def run(self):
        self.cfr = self.calculate_cfr()
        for group in GROUPS:
            group_cases = f'{group}_cases'
            group_deaths = f'{group}_deaths'
            cfr_start_date = self.last_date - pd.Timedelta(f'{self.lag}D')

            daily_cases_smoothed = self.cm.combined_daily_s[group_cases]
            pred_daily = daily_cases_smoothed[cfr_start_date:] * self.cfr[group]
            pred_daily = pred_daily.iloc[:self.cm.n_pred]
            pred_daily.index = self.cm.pred_daily[group_cases].index

            # Use repeated rolling average to smooth out the predicted deaths
            for i in range(5):
                pred_daily = pred_daily.rolling(14, min_periods=1, center=True).
↪mean()

            pred_daily = pred_daily.round(0).astype("int")
            self.pred_daily[group_deaths] = pred_daily
            last_deaths = self.data[group_deaths].loc[self.last_date]
            self.pred_cumulative[group_deaths] = pred_daily.cumsum() + last_deaths
        self.combine_actual_with_pred()

    def combine_actual_with_pred(self):
        for gk, df_pred in self.pred_cumulative.items():
            df_actual = self.data[gk][:self.last_date]
```

```
            df_comb = pd.concat((df_actual, df_pred))
            self.combined_cumulative[gk] = df_comb
            self.combined_daily[gk] = (df_comb.diff()
                                               .fillna(df_comb.iloc[0])
                                               .astype('int'))

    def plot_prediction(self, group, area, **kwargs):
        group_kind = f'{group}_deaths'
        actual = self.data[group_kind][area]
        pred = self.pred_cumulative[group_kind][area]
        first_date = self.last_date - pd.Timedelta(60, 'D')
        last_pred_date = self.last_date + pd.Timedelta(30, 'D')
        actual.loc[first_date:last_pred_date].plot(label='Actual', **kwargs)
        pred.plot(label='Predicted').legend()
```

Let's instantiate this class and then call the `run` method, which should execute immediately as the model for deaths is far simpler than it is for cases.

[19]:
```
dm = DeathsModel(data=data, last_date='2020-11-05', cm=cm, lag=15, period=30)
dm.run()
```

Let's output the daily and cumulative predictions.

[20]:
```
dm.pred_daily['usa_deaths'].iloc[:5, :10]
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-06	20	2	0	23	25	74	23	12	0	4
2020-11-07	20	2	0	23	25	74	23	12	0	4
2020-11-08	20	2	0	23	25	74	23	12	0	4
2020-11-09	20	2	0	24	25	74	24	12	0	4
2020-11-10	20	2	0	24	25	74	24	12	0	4

[21]:
```
dm.pred_cumulative['usa_deaths'].iloc[:5, :10]
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-06	3046	88	0	6110	2062	17962	2376	4668	3	751
2020-11-07	3066	90	0	6133	2087	18036	2399	4680	3	755
2020-11-08	3086	92	0	6156	2112	18110	2422	4692	3	759
2020-11-09	3106	94	0	6180	2137	18184	2446	4704	3	763
2020-11-10	3126	96	0	6204	2162	18258	2470	4716	3	767

Just as with the cases, `combined_daily` and `combined_cumulative` store the combined actual and predicted values. Again, we look at the three days preceding and following the predicted date.

[22]:
```
dm.combined_daily['usa_deaths'].loc['2020-11-03':'2020-11-08', :'Delaware']
```

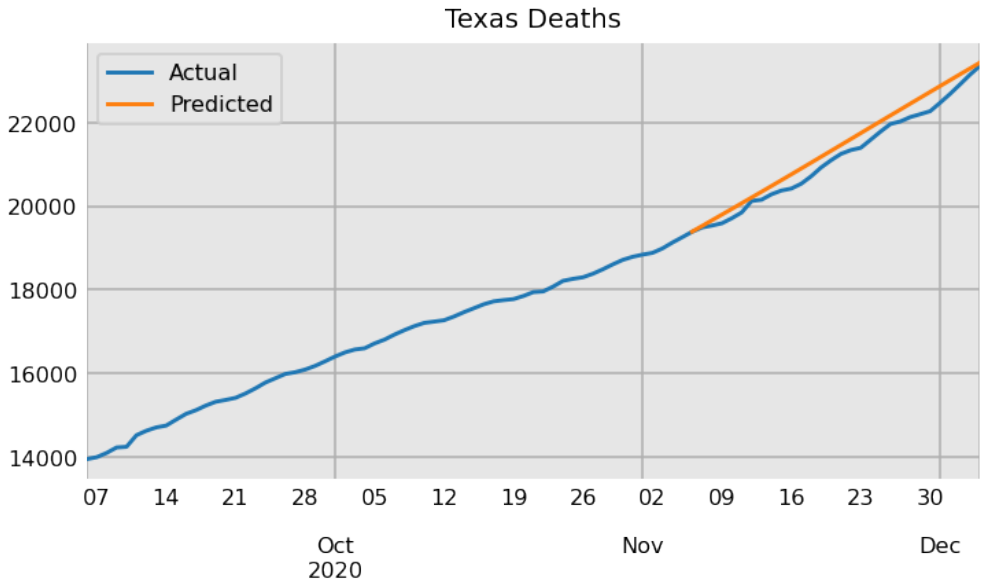
	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-03	14	0	0	38	18	71	19	8	0	4
2020-11-04	19	0	0	39	23	65	22	10	0	1
2020-11-05	20	0	0	28	11	65	20	11	0	2
2020-11-06	20	2	0	23	25	74	23	12	0	4
2020-11-07	20	2	0	23	25	74	23	12	0	4
2020-11-08	20	2	0	23	25	74	23	12	0	4

[23]:
```
dm.combined_cumulative['usa_deaths'].loc['2020-11-03':'2020-11-08', :'Delaware']
```

	Alabama	Alaska	American Samoa	Arizona	Arkansas	California	Colorado	Connecticut	Cruise Ship	Delaware
2020-11-03	2987	86	0	6020	2003	17758	2311	4635	3	744
2020-11-04	3006	86	0	6059	2026	17823	2333	4645	3	745
2020-11-05	3026	86	0	6087	2037	17888	2353	4656	3	747
2020-11-06	3046	88	0	6110	2062	17962	2376	4668	3	751
2020-11-07	3066	90	0	6133	2087	18036	2399	4680	3	755
2020-11-08	3086	92	0	6156	2112	18110	2422	4692	3	759

Use the `plot_prediction` method to plot the actual and predicted values of deaths for a particular area.

[24]:
```
dm.plot_prediction('usa', 'Texas', title='Texas Deaths')
```



8.4 Creating final tables for the dashboard

Now that we have all the information for our dashboard, we'll need to extract it from these classes and store them as files. For simplicity, we will store them as CSV files.

Adding USA to world tables

Originally, we filtered out the USA from the world tables. We did this knowing that the predictions for the entire country were likely to be different than the sum of the predictions of each individual

state. Now that we have each state's predictions, we will total them up and append them to the world tables. Here, the daily and cumulative cases and deaths are assigned to new variables. Each of the world DataFrames has the USA added as a new column.

[25]:
```
# Get Daily Cases and Deaths from dictionaries
world_cases_d = cm.combined_daily['world_cases']
usa_cases_d = cm.combined_daily['usa_cases']
world_deaths_d = dm.combined_daily['world_deaths']
usa_deaths_d = dm.combined_daily['usa_deaths']

# Add USA to world
world_cases_d = world_cases_d.assign(USA=usa_cases_d.sum(axis=1))
world_deaths_d = world_deaths_d.assign(USA=usa_deaths_d.sum(axis=1))

# Get Cumulative Cases and Deaths
world_cases_c = cm.combined_cumulative['world_cases']
usa_cases_c = cm.combined_cumulative['usa_cases']
world_deaths_c = dm.combined_cumulative['world_deaths']
usa_deaths_c = dm.combined_cumulative['usa_deaths']

# Add USA to world
world_cases_c = world_cases_c.assign(USA=usa_cases_c.sum(axis=1))
world_deaths_c = world_deaths_c.assign(USA=usa_deaths_c.sum(axis=1))
```

We verify that the USA has been added to the world cases DataFrame.

[26]:
```
world_cases_d.iloc[-5:, -10:]
```

	Uzbekistan	Vanuatu	Vatican City	Venezuela	Vietnam	West Bank and Gaza	Yemen	Zambia	Zimbabwe	USA
2020-12-01	91	0	0	120	0	289	0	31	10	80949
2020-12-02	88	0	0	115	0	285	0	30	10	80351
2020-12-03	85	0	0	111	0	281	0	30	10	79732
2020-12-04	83	0	0	107	0	278	0	29	9	79085
2020-12-05	80	0	0	103	0	274	0	29	9	78411

Creating a single table to store all historical and future values

A single table will be used to hold the daily and cumulative cases and deaths for each area for each date. We'll reshape the DataFrames using the `stack` method so that all values are in a single column with the index containing the date and the area name.

[27]:
```
world_cases_d.stack().tail()
```

[27]:
```
2020-12-05  West Bank and Gaza      274
            Yemen                     0
            Zambia                   29
            Zimbabwe                  9
            USA                   78411
dtype: int64
```

We can place all four Series as columns in a single DataFrame using the `concat` function using the `keys` parameter to label each new column.

[28]:
```
df_world = pd.concat((world_deaths_d.stack(), world_cases_d.stack(),
                      world_deaths_c.stack(), world_cases_c.stack()), axis=1,
                     keys=['Daily Deaths', 'Daily Cases', 'Deaths', 'Cases'])
df_world.tail()
```

		Daily Deaths	Daily Cases	Deaths	Cases
2020-12-05	West Bank and Gaza	5	274	681	66604
	Yemen	0	0	601	2063
	Zambia	0	29	349	17848
	Zimbabwe	0	9	271	8783
	USA	1476	78411	280215	12226484

The same thing is done for the USA data.

[29]:
```
df_usa = pd.concat((usa_deaths_d.stack(), usa_cases_d.stack(),
                    usa_deaths_c.stack(), usa_cases_c.stack()), axis=1,
                   keys=['Daily Deaths', 'Daily Cases', 'Deaths', 'Cases'])
df_usa.tail()
```

		Daily Deaths	Daily Cases	Deaths	Cases
2020-12-05	Virginia	16	894	4189	217527
	Washington	13	762	2768	142733
	West Virginia	6	319	662	36850
	Wisconsin	66	5996	4335	444479
	Wyoming	5	443	226	28675

All of the above code is placed in a function that accepts instances of the `CasesModel` and `DeathsModel` as arguments.

[30]:
```
def combine_all_data(cm, dm):
    # Get Daily Cases and Deaths
    world_cases_d = cm.combined_daily['world_cases']
    usa_cases_d = cm.combined_daily['usa_cases']
    world_deaths_d = dm.combined_daily['world_deaths']
    usa_deaths_d = dm.combined_daily['usa_deaths']

    # Add USA to world
    world_cases_d = world_cases_d.assign(USA=usa_cases_d.sum(axis=1))
    world_deaths_d = world_deaths_d.assign(USA=usa_deaths_d.sum(axis=1))

    # Get Cumulative Cases and Deaths
    world_cases_c = cm.combined_cumulative['world_cases']
    usa_cases_c = cm.combined_cumulative['usa_cases']
    world_deaths_c = dm.combined_cumulative['world_deaths']
    usa_deaths_c = dm.combined_cumulative['usa_deaths']
```

```
    # Add USA to world
    world_cases_c = world_cases_c.assign(USA=usa_cases_c.sum(axis=1))
    world_deaths_c = world_deaths_c.assign(USA=usa_deaths_c.sum(axis=1))

    df_world = pd.concat((world_deaths_d.stack(), world_cases_d.stack(),
                          world_deaths_c.stack(), world_cases_c.stack()), axis=1,
                         keys=['Daily Deaths', 'Daily Cases', 'Deaths', 'Cases'])

    df_usa = pd.concat((usa_deaths_d.stack(), usa_cases_d.stack(),
                        usa_deaths_c.stack(), usa_cases_c.stack()), axis=1,
                       keys=['Daily Deaths', 'Daily Cases', 'Deaths', 'Cases'])
    df_all = pd.concat((df_world, df_usa), keys=['world', 'usa'],
                       names=['group', 'date', 'area'])
    df_all.to_csv('data/all_data.csv')
    return df_all
```

Notice at the very end of this function, the world and USA DataFrames are concatenated one on top of each other for the final DataFrame. A new index level, 'group', was added to the DataFrame. The data is written to the file `all_data.csv`.

[31]:
```
df_all = combine_all_data(cm, dm)
df_all.tail()
```

			Daily Deaths	Daily Cases	Deaths	Cases
group	**date**	**area**				
usa	**2020-12-05**	**Virginia**	16	894	4189	217527
		Washington	13	762	2768	142733
		West Virginia	6	319	662	36850
		Wisconsin	66	5996	4335	444479
		Wyoming	5	443	226	28675

8.5 Create summary table

The main table in our dashboard is a summary of the data at the current date. We'll create it now and begin by selecting the current day's rows.

[32]:
```
df_summary = df_all.query('date == @last_date')
df_summary.head()
```

			Daily Deaths	Daily Cases	Deaths	Cases
group	**date**	**area**				
world	**2020-11-05**	**Afghanistan**	4	86	1548	41814
		Albania	7	421	543	22721
		Algeria	12	642	2011	60169
		Andorra	0	90	75	5135
		Angola	3	289	299	12102

We read in a file called population.csv that has the population and country code (used in the map) of

each area.

[33]:
```
pop = pd.read_csv("data/population.csv")
pop.head()
```

	group	area	code	population
0	world	Afghanistan	AFG	38.928341
1	world	Albania	ALB	2.877800
2	world	Algeria	DZA	43.851043
3	world	Andorra	AND	0.077265
4	world	Angola	AGO	32.866268

Let's merge these two tables together and add columns for deaths and cases per million.

[34]:
```
df_summary = df_summary.merge(pop, how='left', on=['group','area'])
df_summary["Deaths per Million"] = (df_summary["Deaths"] /
                                    df_summary["population"]).round(0)
df_summary["Cases per Million"] = (df_summary["Cases"] /
                                   df_summary["population"]).round(-1)
df_summary.head()
```

	group	area	Daily Deaths	Daily Cases	Deaths	Cases	code	population	Deaths per Million	Cases per Million
0	world	Afghanistan	4	86	1548	41814	AFG	38.928341	40.0	1070.0
1	world	Albania	7	421	543	22721	ALB	2.877800	189.0	7900.0
2	world	Algeria	12	642	2011	60169	DZA	43.851043	46.0	1370.0
3	world	Andorra	0	90	75	5135	AND	0.077265	971.0	66460.0
4	world	Angola	3	289	299	12102	AGO	32.866268	9.0	370.0

Let's place all of this code within its own function which also writes the data to a file.

[35]:
```
def create_summary_table(df_all, last_date):
    df = df_all.query('date == @last_date')
    pop = pd.read_csv("data/population.csv")
    df = df.merge(pop, how='left', on=['group','area'])
    df["Deaths per Million"] = (df["Deaths"] / df["population"]).round(0)
    df["Cases per Million"] = (df["Cases"] / df["population"]).round(-1)
    df['date'] = last_date
    df.to_csv('data/summary.csv', index=False)
    return df
```

Let's use the function to create the table and output the first few rows.

[36]:
```
create_summary_table(df_all, last_date).head()
```

	group	area	Daily Deaths	Daily Cases	Deaths	...	code	population	Deaths per Million	Cases per Million	date
0	world	Afghanistan	4	86	1548	...	AFG	38.928341	40.0	1070.0	2020-11-05
1	world	Albania	7	421	543	...	ALB	2.877800	189.0	7900.0	2020-11-05
2	world	Algeria	12	642	2011	...	DZA	43.851043	46.0	1370.0	2020-11-05
3	world	Andorra	0	90	75	...	AND	0.077265	971.0	66460.0	2020-11-05
4	world	Angola	3	289	299	...	AGO	32.866268	9.0	370.0	2020-11-05

5 rows × 11 columns

8.6 Code within the modules

The `CasesModel` and `DeathsModel` class are placed in the `models.py` file. The `PrepareData` class and `combine_all_data` and `create_summary_table` functions are placed in the `prepare.py` file. In the next chapter, we'll run all of our code for the entire project to prepare the data, make predictions, and save the final tables.

Chapter 9

Running all of the Code

In this chapter, we'll run all of the code by executing the `update.py` file, which has its contents displayed below.

9.1 Examining `update.py`

The `update.py` file begins by importing all of the necessary functions and classes from `prepare.py` and `models.py`. Several parameters are defined, which will be used to instantiate `CasesModel` and `DeathsModel`. You may wish to change these values to get better results.

The `if __name__ == "__main__"` condition is a common way to "protect" code from being arbitrarily executed when a module is imported. For instance, if we have the code `import update` in another module, the code block within the `if` statement will NOT be run.

The `if` condition isn't actually necessary here since we will not be importing `update` from other modules, but is still good to have as it signals that this file is meant for executing from the command line. The variable name `__name__` is a string that all Python modules have as attributes when they are executed or imported. If the file is executed from the command line, this string is set to `'__main__'`. Therefore, the code block within the `if` statement executes whenever this file is run from the command line.

Command line arguments

There are two ways to execute `update.py`, which are shown below.

```
python update.py
python update.py 20200720
```

You can run it with or without a date which may be given in the form YYYYMMDD. Within the `if` statement's code block, the standard library's `sys` module is used to retrieve the command line arguments. All the values following the word `python` that are separated by a space are considered 'arguments' including the name of the file. For instance, when you execute `python update.py 20200720` there are two arguments, `update.py` and `20200720`. Both are stored as strings in the list `sys.argv`.

The first thing we check in the `if` code block is the length of the argument list. If the length is 1, then just the name of the executable file was provided without a date. In this case, the `last_date` variable is set to `None`, which the models will interpret as the last available date when the John Hopkins repository was updated. This is the most common scenario as we would like to keep our dashboard updated with the most recent data.

If a date is provided as the second argument then it is assigned to `last_date`. This is useful when you want to test a model on historical data. By default, the `all_data.csv` and `summary.csv` files are overwritten. If you attempt to execute `update.py` with any extra arguments an error will be raised with a message returned on how to execute it properly.

The rest of the code block runs all of our previous work in just a few lines of code. It instantiates and calls the `run` method from `PrepareData`, `CasesModel`, and `DeathsModel`, before calling the `combine_all` and `create_summary_table` functions to create the final data files.

```
# update.py file
import sys
from prepare import PrepareData, combine_all_data, create_summary_table
from models import CasesModel, DeathsModel, general_logistic_shift

# Parameters for CasesModel - Feel free to change these
N_TRAIN = 60   # Number of observations used in training
N_SMOOTH = 15  # Number of observations used in smoothing
N_PRED = 56    # Number of new observations to predict
L_N_MIN = 5    # Number of days of exponential growth for L min boundary
L_N_MAX = 50   # Number of days of exponential growth for L max boundary

# Parameters for DeathsModel - Feel free to change these
LAG = 15       # Number of days to lag cases for calculation of CFR
PERIOD = 30    # Number of days to total for CFR

if __name__ == "__main__":
    if len(sys.argv) == 1:
        last_date = None
    elif len(sys.argv) == 2:
        last_date = sys.argv[1]
    else:
        raise TypeError(
            """
            When calling `python update.py` from the command line,
            pass 0 or 1 arguments.
                0 arguments: make prediction for latest data (downloads latest data)
                1 argument: provide the last date that the model will see
                            i.e. 20200720
            """
        )
    data = PrepareData().run()
    cm = CasesModel(
        model=general_logistic_shift,
        data=data,
        last_date=last_date,
        n_train=N_TRAIN,
        n_smooth=N_SMOOTH,
        n_pred=N_PRED,
        L_n_min=L_N_MIN,
        L_n_max=L_N_MAX,
```

```
)
cm.run()

dm = DeathsModel(data=data, last_date=last_date, cm=cm, lag=15, period=30)
dm.run()

df_all = combine_all_data(cm, dm)
create_summary_table(df_all, cm.last_date)
```

Run all of the code now

Go to your command line and run `python update.py` twice, once within each of the `project` and `notebooks` directory to update the data in each one and to verify that it works.

Chapter 10

Visualizations with Plotly

In this chapter, we'll get introduced to the plotly library, which has the ability to create interactive data visualizations for the web. All previous chapters used matplotlib, which is a better tool for static visualizations.

10.1 Plotly vs Dash

Both the plotly and dash libraries are products of the company Plotly. Both libraries are free and open source with an enterprise version available for extra features and services. The plotly python library is closely related to the dash python library, but each have different purposes. The plotly library creates the visualizations, producing them as independent HTML and JavaScript files that can be embedded in any page, including Jupyter Notebooks.

The dash library creates the dashboards with tools such as data tables, tabs, dropdowns, radio buttons, and many more. It also runs the application, allowing an interactive experience for the users. All graphs in a dash application are created from the plotly library. We will build our dashboard with dash, but must learn enough plotly first to make our visualizations.

10.2 Introduction to Plotly

The plotly python library is enormous and covering all details is out of scope for this book. This chapter presents the most relevant components of the library for our specific application. I suggest keeping the documentation open, so that you can have a reference to the official tutorials on all parts of the library. Before we get started, let's read in the `all_data.csv` file which has all of the historical and predicted data for all areas. The exact data that you have depends on the last time you ran `python update.py` in the notebooks directory.

[1]:
```
import pandas as pd
df_all = pd.read_csv('data/all_data.csv', parse_dates=['date'])
df_all.tail()
```

	group	date	area	Daily Deaths	Daily Cases	Deaths	Cases
80383	usa	2020-12-05	Virginia	16	894	4189	217527
80384	usa	2020-12-05	Washington	13	762	2768	142733
80385	usa	2020-12-05	West Virginia	6	319	662	36850
80386	usa	2020-12-05	Wisconsin	66	5996	4335	444479
80387	usa	2020-12-05	Wyoming	5	443	226	28675

We'll select the state of Texas for our plotting examples and place the date in the index.

[2]:
```
df_texas = df_all.query('group == "usa" and area == "Texas"')
df_texas = df_texas.set_index('date')
df_texas.tail()
```

date	group	area	Daily Deaths	Daily Cases	Deaths	Cases
2020-12-01	usa	Texas	140	7282	22872	1206684
2020-12-02	usa	Texas	140	7250	23012	1213934
2020-12-03	usa	Texas	140	7216	23152	1221150
2020-12-04	usa	Texas	139	7181	23291	1228331
2020-12-05	usa	Texas	139	7143	23430	1235474

We'll also read in the summary table which has a column containing the last date of known data.

[3]:
```
df_summary = pd.read_csv('data/summary.csv', parse_dates=['date'])
df_summary.head(3)
```

	group	area	Daily Deaths	Daily Cases	Deaths	...	code	population	Deaths per Million	Cases per Million	date
0	world	Afghanistan	4	86	1548	...	AFG	38.928341	40.0	1070.0	2020-11-05
1	world	Albania	7	421	543	...	ALB	2.877800	189.0	7900.0	2020-11-05
2	world	Algeria	12	642	2011	...	DZA	43.851043	46.0	1370.0	2020-11-05

3 rows × 11 columns

We assign this last known date to its own variable and calculate the first predicted date as the very next day. These values will be useful when graphing the actual and predicted values separately.

[4]:
```
last_date = df_summary['date'].iloc[0]
first_pred_date = last_date + pd.Timedelta('1D')
last_date, first_pred_date
```

[4]:
```
(Timestamp('2020-11-05 00:00:00'), Timestamp('2020-11-06 00:00:00'))
```

General steps to create a plotly graph

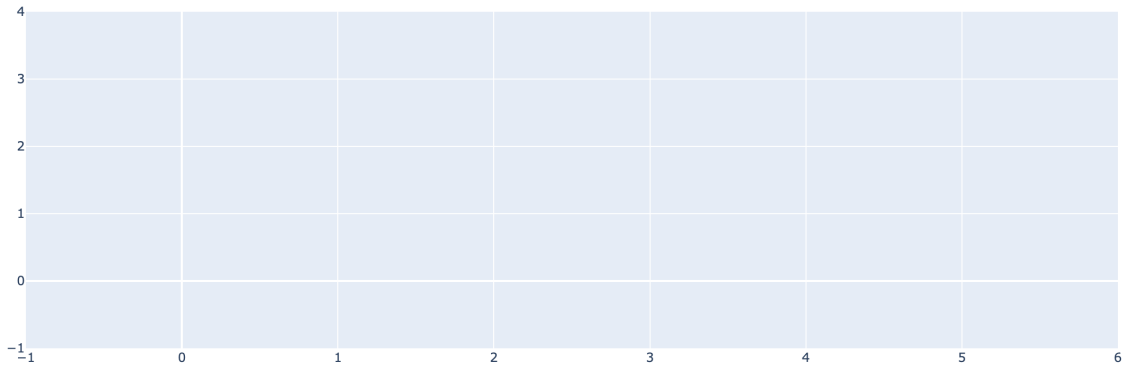
There are multiple ways to create graphs in plotly, but since this is not a comprehensive tutorial, we will show just a single straightforward path and use it for all of our graphs. The following three steps will be used to create our graphs:

1. Create Figure - with `go.Figure` or `make_subplots`
2. Add trace - with `fig.add_*`
3. Update layout - with `fig.update_layout` or `fig.update_*`

10.3 Plotly Figure Object

All of our plots begin with the creation of a plotly figure which is done by importing the `graph_objects` module. Here, it is imported an aliased as `go`. We then create an empty figure, assign it to a variable, and then output it to the screen.

[5]:
```
import plotly.graph_objects as go
fig = go.Figure()
fig
```



Adding traces

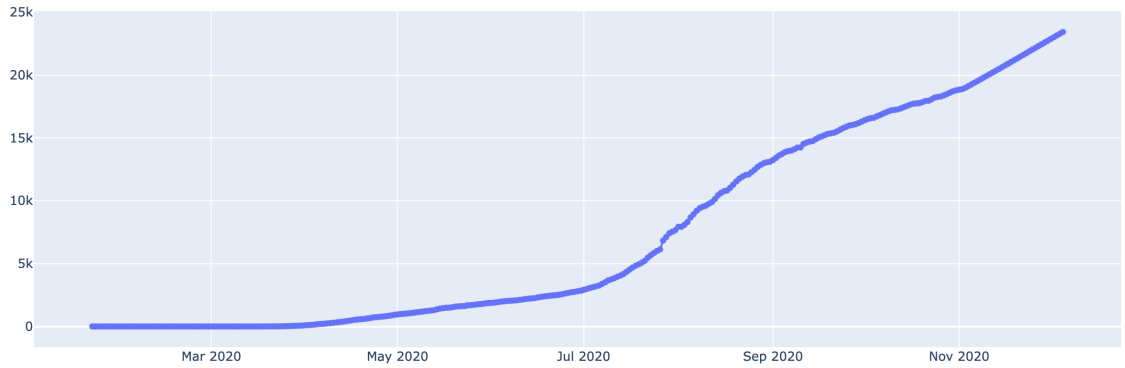
All "traces" can be added to the figure with one of the `add_*` methods, where the `*` references one of the trace names. In plotly, a **trace** is one of several dozen kinds of visualizations able to be added to a figure (scatter, bar, pie, histogram, etc…). In as few words as possible, a trace is a "type of plot". Visit this reference page to see a list of all possible traces in the left margin. Click on one of the traces to view a description of each parameter.

Here, we create a scatter (and line) plot using the `add_scatter` method. We set `x` to be the index (containing the date) and `y` to be the column for deaths from our DataFrame. The `mode` parameter has three common settings:

- `"lines"` - connect the points without showing the markers
- `"markers"` - show just the markers
- `"lines+markers"` - connect the points and show the markers

There is no `add_line` method in plotly. Instead, use `add_scatter` with `mode` set to `"lines"` to create a line plot.

[6]:
```
x = df_texas.index
y = df_texas['Deaths']
fig = go.Figure()
fig.add_scatter(x=x, y=y, mode="lines+markers")
```



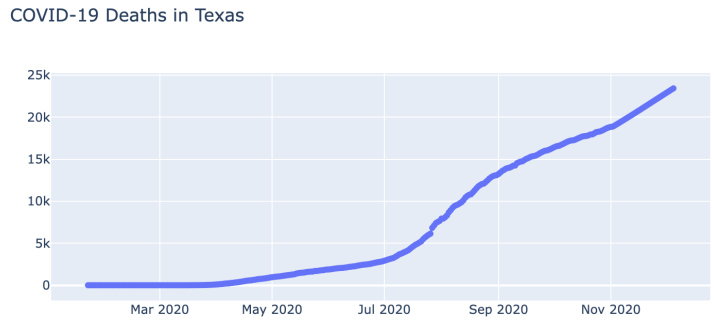
Updating the layout

In plotly, the **layout** consists of the following graph properties plus several more:

- height
- width
- title
- xaxis/yaxis
- legend
- margin
- annotations

Here, we plot the same trace as above, but change the height and width (given in pixels) of the figure and provide a title.

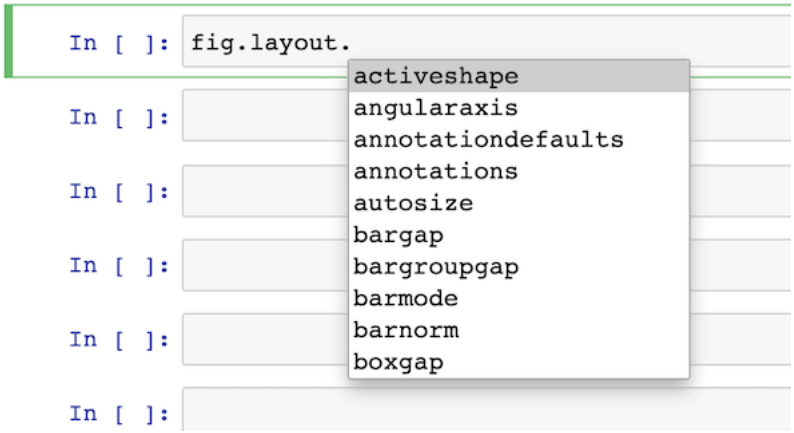
[7]:
```
fig = go.Figure()
fig.add_scatter(x=x, y=y, mode="lines+markers")
fig.update_layout(height=400,
                  width=800,
                  title="COVID-19 Deaths in Texas")
```



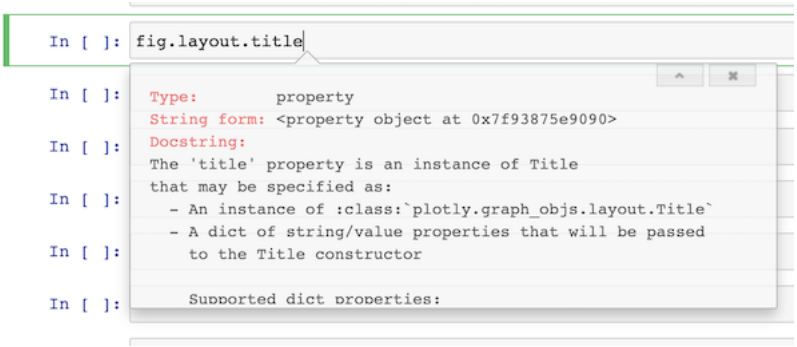
Finding all of the layout properties

The `update_layout` method does not show any of its properties in its docstrings. To view all of the layout properties, visit this layout reference page. You'll notice that many of the properties are **nested**, meaning that these properties have properties themselves that can be set using a dictionary.

Another way to find the layout properties (while in a Jupyter Notebook) is to access the layout object directory using `fig.layout`. Place a single `.` after it and then **press tab**. A list of all properties will appear in a dropdown menu as seen in the image below.

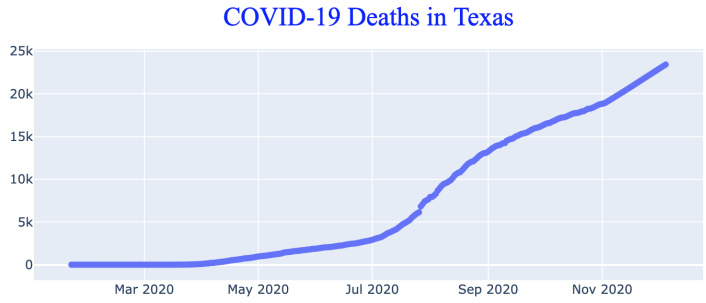


From here, choose one of the properties and press **shift + tab + tab** to reveal the docstrings. Below, the docstrings for the `title` property are shown.



Let's create a more specific title by setting several of its properties with a dictionary. Notice that `font` is a further nested property with three more properties (color, family, and size). Find more information with `fig.layout.title.font` (pressing **shift + tab + tab**). The coordinates for `x` and `y` use the range 0 to 1 (relative position left to right and bottom to top).

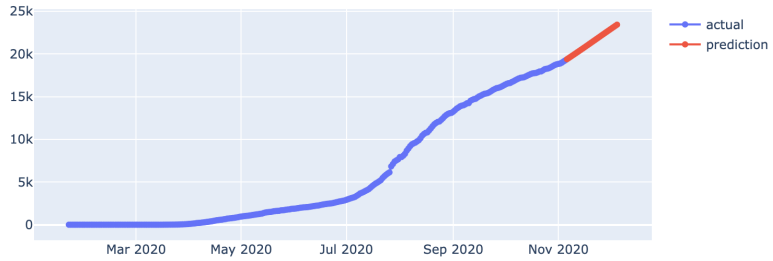
[8]:
```
fig.update_layout(title={
    "text": "COVID-19 Deaths in Texas",
    "x": 0.5,
    "y": 0.85,
    "font": {
        "color": "blue",
        "family": "dejavu sans",
        "size": 25
    }
})
```



10.4 Creating a figure with multiple traces

Any number of traces may be added to the same figure. Here, we split the DataFrame into actual and predicted values and make two separate calls to the `add_scatter` method. The `name` parameter is used as a label in the legend. Notice, that the color of the second line will automatically be different than the first. The default color sequence for successive traces is titled "Plotly" and is found here.

[9]:
```
df_texas_actual = df_texas[:last_date]
df_texas_pred = df_texas[first_pred_date:]
fig = go.Figure()
fig.add_scatter(x=df_texas_actual.index,
                y=df_texas_actual['Deaths'],
                mode="lines+markers",
                name='actual')
fig.add_scatter(x=df_texas_pred.index,
                y=df_texas_pred['Deaths'],
                mode="lines+markers",
                name='prediction')
fig.update_layout(height=400, width=800)
```

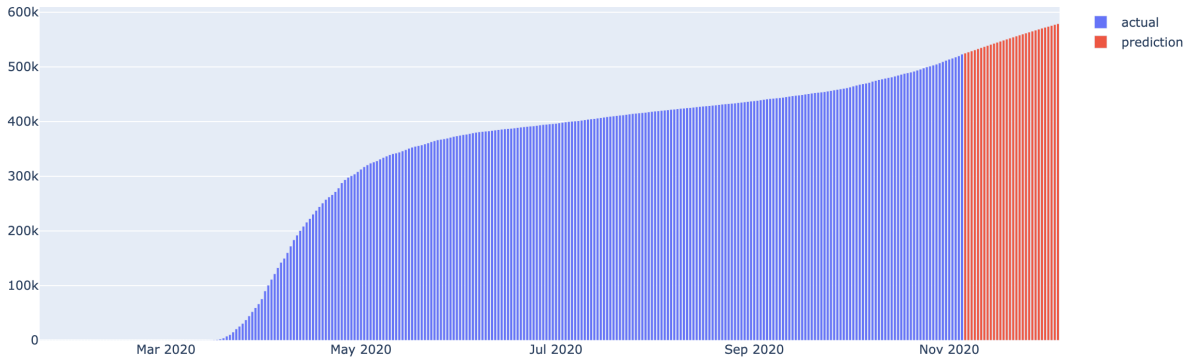


Here, we write a function that accepts a group, area, and kind and returns a bar plot of the actual and predicted kind for that area.

[10]:
```
def area_bar_plot(df, group, area, kind, last_date, first_pred_date):
    """
    Creates a bar plot of actual and predicted values for given kind
    from one area

    Parameters
    ----------
    df - All data DataFrame
```

```
    group - "world" or "usa"

    area - A country or US state

    kind - "Daily Deaths", "Daily Cases", "Deaths", "Cases"

    last_date - last known date of data

    first_pred_date - first predicted date
    """
    df = df.query("group == @group and area == @area").set_index("date")
    df_actual = df[:last_date]
    df_pred = df[first_pred_date:]
    fig = go.Figure()
    fig.add_bar(x=df_actual.index, y=df_actual[kind], name="actual")
    fig.add_bar(x=df_pred.index, y=df_pred[kind], name="prediction")
    return fig
```

[11]:
```
area_bar_plot(df_all, 'usa', 'New York', 'Cases', last_date, first_pred_date)
```

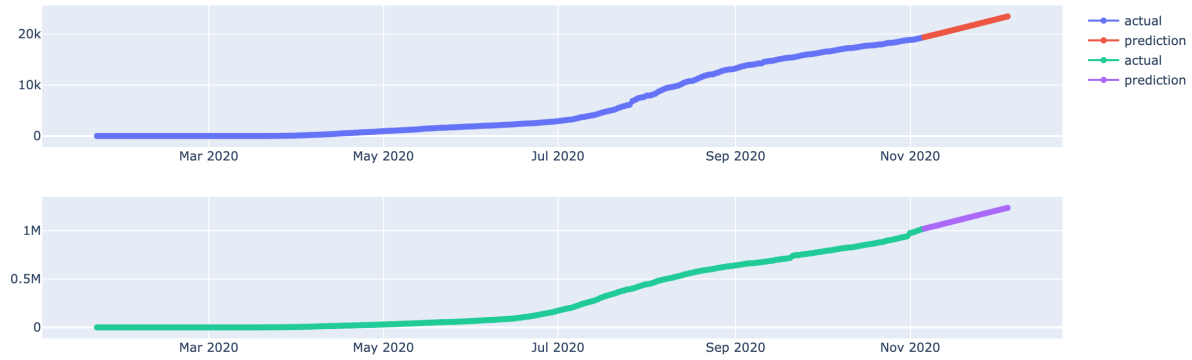


10.5 Creating subplots

Multiple plots within a single figure can be created with the `make_subplots` function from the `subplots` module. It creates a rectangular grid of subplots using the provided `rows` and `cols` parameters. To add a trace to a specific subplot, set the `row` and `col` parameters in the `add_*` methods to integers. Here, we plot both actual and predicted traces for both daily deaths and cases.

[12]:
```
from plotly.subplots import make_subplots
fig = make_subplots(rows=2, cols=1)

# top subplot
fig.add_scatter(x=df_texas_actual.index,
                y=df_texas_actual['Deaths'],
                mode="lines+markers",
                name='actual',
                row=1,
                col=1)
fig.add_scatter(x=df_texas_pred.index,
```

```
                y=df_texas_pred['Deaths'],
                mode="lines+markers",
                name='prediction',
                row=1,
                col=1)

# bottom subplot
fig.add_scatter(x=df_texas_actual.index,
                y=df_texas_actual['Cases'],
                mode="lines+markers",
                name='actual',
                row=2,
                col=1)
fig.add_scatter(x=df_texas_pred.index,
                y=df_texas_pred['Cases'],
                mode="lines+markers",
                name='prediction',
                row=2,
                col=1)
```



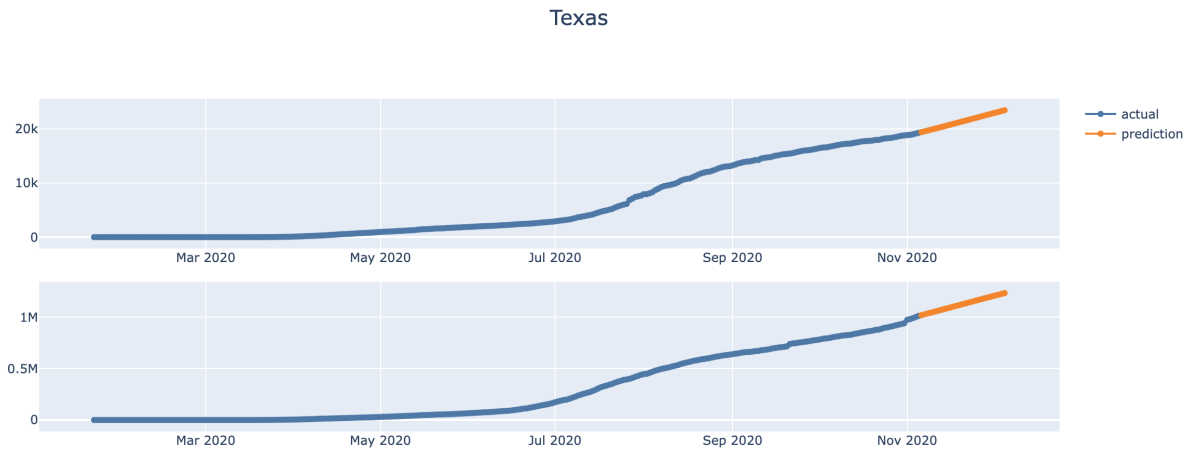
Cleaning up the subplots

While we have our traces plotted correctly, there are a few changes we can make to improve this graph. The colors for the actual/prediction should be the same in each graph and repeated names in the legend should be removed. Below, we write a nested for-loop to iterate over the kinds ("Deaths" and "Cases") and again over the actual and predicted DataFrames, which are stored in a dictionary. We choose the first two colors from the T10 qualitative color sequence (this is the data visualization software Tableau's default colors).

To prevent the legend from repeating the same names, we use the `update_traces` method, which allows us to specify which subplot to hide the legend. The `update_layout` method uses the same parameter `showlegend`, but applies its changes to ALL subplots. There are several other `update_*` methods that allow you to specify the subplot. Use the `update_layout` method when you want to change a property for the entire figure.

[13]:
```
from plotly.colors import qualitative
COLORS = qualitative.T10[:2]
KINDS = 'Deaths', 'Cases'
dfs = {'actual': df_texas_actual, 'prediction': df_texas_pred}
```

```
fig = make_subplots(rows=2, cols=1, vertical_spacing=.1)
for row, kind in enumerate(KINDS, start=1):
    for i, (name, df) in enumerate(dfs.items()):
        fig.add_scatter(x=df.index,
                        y=df[kind],
                        mode="lines+markers",
                        name=name,
                        line={"color": COLORS[i]},
                        row=row,
                        col=1)

fig.update_traces(showlegend=False, row=2, col=1)
fig.update_layout(title={"text": "Texas", "x": 0.5, "y": 0.97, "font": {"size":␣
 ↪20}})
fig
```



10.6 Adding annotations

The `make_subplots` function allows you to set the titles with the `subplot_titles` parameter, but does not give you control over any of its properties (color, size, font, etc…). You can only provide it text. To create titles with any non-default properties, you'll need to make an annotation using either the `add_annotation` method or the `update_layout` method. We choose the latter below to add two annotations (they act as titles for our subplots).

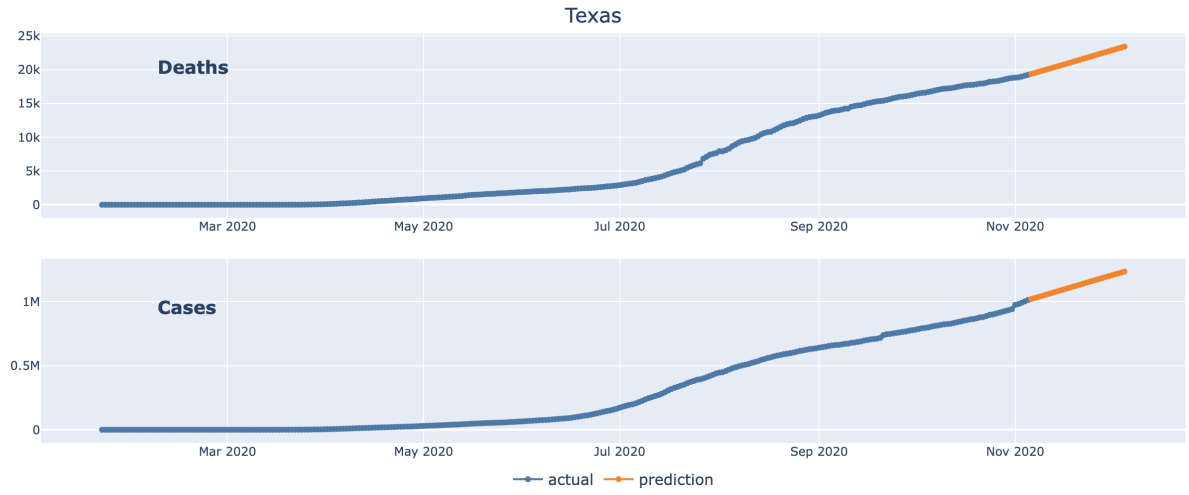
You must set the `annotations` parameter within `update_layout` to be a list of dictionaries, with each dictionary representing a single annotation. If all annotations share some properties, you can provide all of the shared properties to the `update_annotations` method instead of repeating them in the `update_layout` method.

The `xref`/`yref` parameters refer to the coordinate system used for `x` and `y`. When set to "paper", the values correspond to the proportion of the figure and must be in the range 0 to 1. To make the text bold, wrap the text in `<b></b>` HTML tags.

The margin is the space between the four edges of the plot and the figure. They default to 80 pixels for the left and right margins and 100 for the top and bottom. We decrease this space so that the plots fill out more of the figure. We also move the legend below the bottom subplot. This graph should now

look almost exactly like the one in the dashboard.

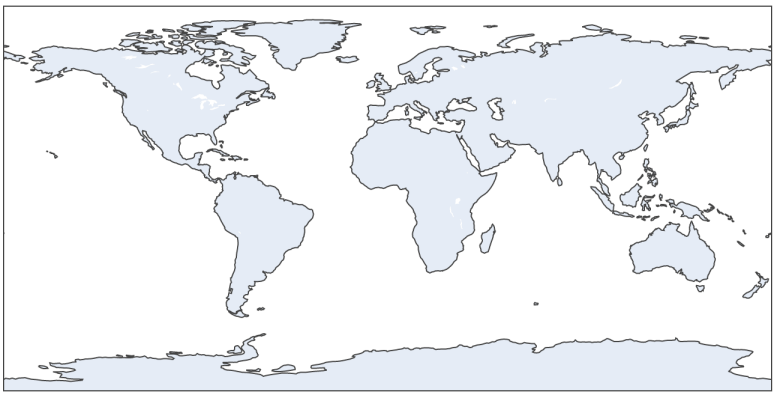
[14]:
```
fig.update_layout(
            annotations=[
                {"y": 0.95, "text": "<b>Deaths</b>"},
                {"y": 0.3, "text": "<b>Cases</b>"},
            ],
            margin={"t": 40, "l": 50, "r": 10, "b": 0},
            legend={
                "x": 0.5,
                "y": -0.05,
                "xanchor": "center",
                "orientation": "h",
                "font": {"size": 15}},
        )
annot_props = {
        "x": 0.1,
        "xref": "paper",
        "yref": "paper",
        "xanchor": "left",
        "showarrow": False,
        "font": {"size": 18},
    }
fig.update_annotations(annot_props)
fig
```



10.7 Choropleth maps

The choropleth trace creates a variety of polygons (countries and US states for our project) colored by the value of a given numeric variable. Let's create the default (base) map by creating a figure and then calling `add_choropleth` with no arguments.

[15]:
```
fig = go.Figure()
fig.add_choropleth()
```



Coloring countries by deaths

Let's read in the summary table and select the world group to get a single row of data per country. We also filter for countries with at least 1 million in population.

[16]:
```
df_world = df_summary.query("group == 'world' and population > 1")
df_world.head(3)
```

	group	area	Daily Deaths	Daily Cases	Deaths	...	code	population	Deaths per Million	Cases per Million	date
0	world	Afghanistan	4	86	1548	...	AFG	38.928341	40.0	1070.0	2020-11-05
1	world	Albania	7	421	543	...	ALB	2.877800	189.0	7900.0	2020-11-05
2	world	Algeria	12	642	2011	...	DZA	43.851043	46.0	1370.0	2020-11-05

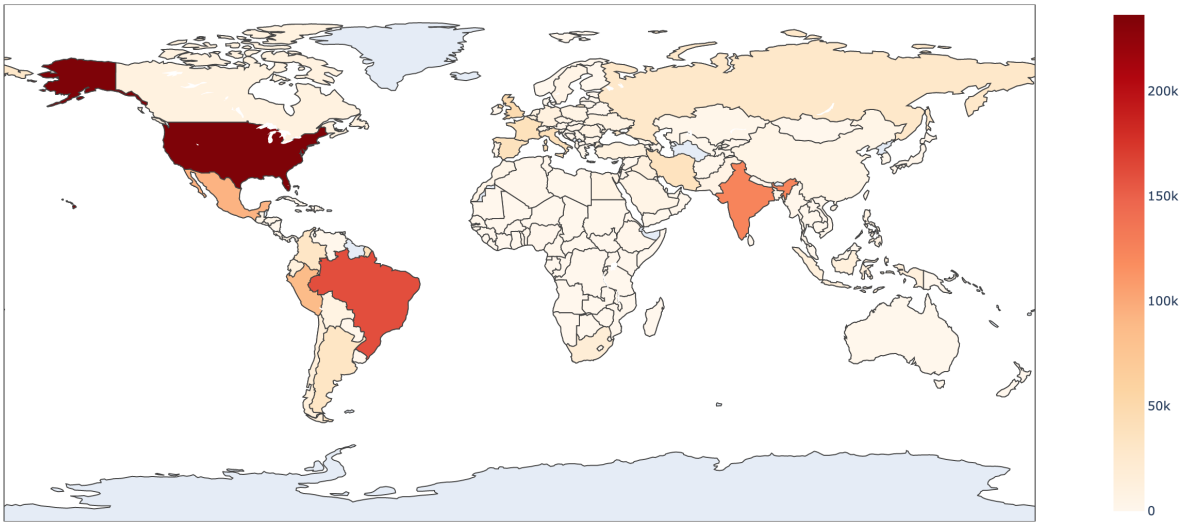
3 rows × 11 columns

Each country has a standardized ISO-3 code that plotly understands. Let's assign these codes and the deaths column as their own variables.

[17]:
```
locations = df_world['code']
z = df_world['Deaths']
```

Let's recreate the choropleth map from the dashboard with this information, setting the parameter `z` to the total number of deaths. We select a continuous color scale called "orrd". You can find all continuous color scales here.

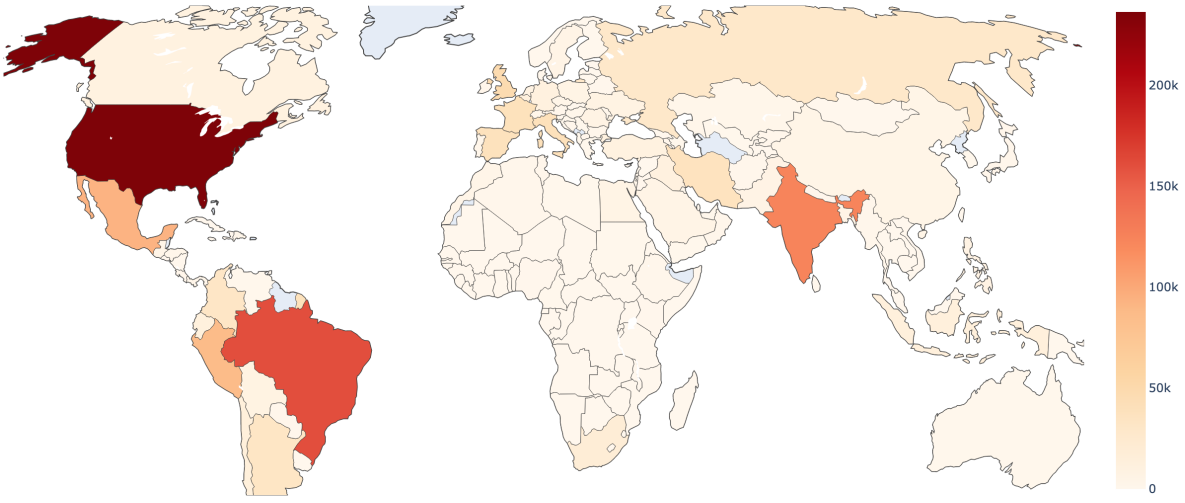
[18]:
```
fig = go.Figure()
fig.add_choropleth(locations=locations, z=z, zmin=0, colorscale="orrd")
fig.update_layout(margin={"t": 0, "l": 10, "r": 10, "b": 0})
```



Selecting a better range and projection

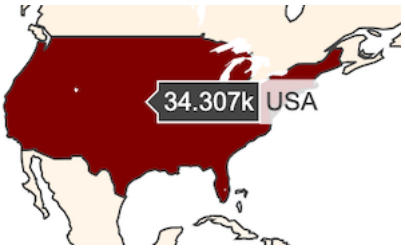
It's unnecessary to show the very northern and southern areas of the world as well as the swath of emptiness in the Pacific Ocean. There are also a large number of projections to choose from. Projection "robinson" is chosen below, but feel free to experiment with others. We can select the latitude and longitude range, and the projection by setting the `geo` parameter in `update_layout`.

[19]:
```
fig = go.Figure()
fig.add_choropleth(locations=locations, z=z, zmin=0, colorscale="orrd",
                   marker_line_width=0.5)
fig.update_layout(
    geo={
        "showframe": False,
        "lataxis": {"range": [-37, 68]},
        "lonaxis": {"range": [-130, 150]},
        "projection": {"type": "robinson"}
    },
    margin={"t": 0, "l": 10, "r": 10, "b": 0})
```



Customizing the hover text

Hovering over each country shows only the value of `z` and the country code like in the image below.



We can customize this text to be anything we desire by supplying a sequence of the exact string to display for each country. The `hover_text` function below is applied to each row in the `df_world` DataFrame to create a long string of all of the data nicely formatted with line breaks (`<br>`) between each statistic. The DataFrame `apply` method is used to iterate over each row and apply this function to the values. The string for each of the first few rows is outputted below.

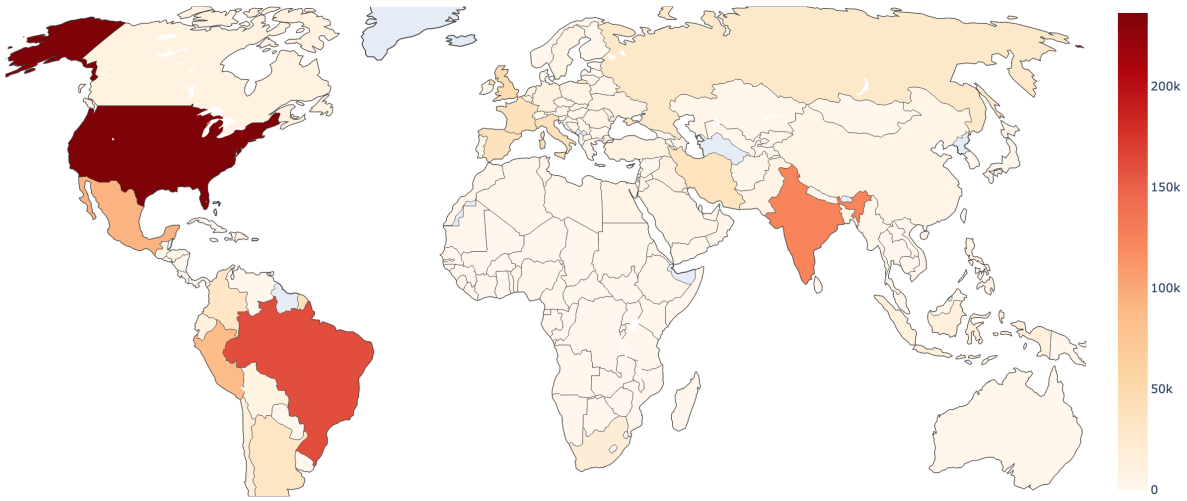
[20]:
```
def hover_text(x):
    name = x["area"]
    deaths = x["Deaths"]
    cases = x["Cases"]
    deathsm = x["Deaths per Million"]
    casesm = x["Cases per Million"]
    pop = x["population"]
    return (
        f"<b>{name}</b><br>"
        f"Deaths - {deaths:,.0f}<br>"
        f"Cases - {cases:,.0f}<br>"
        f"Deaths per Million - {deathsm:,.0f}<br>"
        f"Cases per Million - {casesm:,.0f}<br>"
        f"Population - {pop:,.0f}M"
    )

text = df_world.apply(hover_text, axis=1)
text.head()
```

[20]:
```
0    <b>Afghanistan</b><br>Deaths - 1,548<br>Cases …
1    <b>Albania</b><br>Deaths - 543<br>Cases - 22,7…
2    <b>Algeria</b><br>Deaths - 2,011<br>Cases - 60…
4    <b>Angola</b><br>Deaths - 299<br>Cases - 12,10…
6    <b>Argentina</b><br>Deaths - 32,766<br>Cases -…
dtype: object
```

Set the hover text with the `text` parameter, and force plotly to just use this provided text by setting `hoverinfo` to "text".

[21]:
```
fig = go.Figure()
fig.add_choropleth(locations=locations, z=z, zmin=0, colorscale="orrd",
                   marker_line_width=0.5, text=text, hoverinfo="text")
```

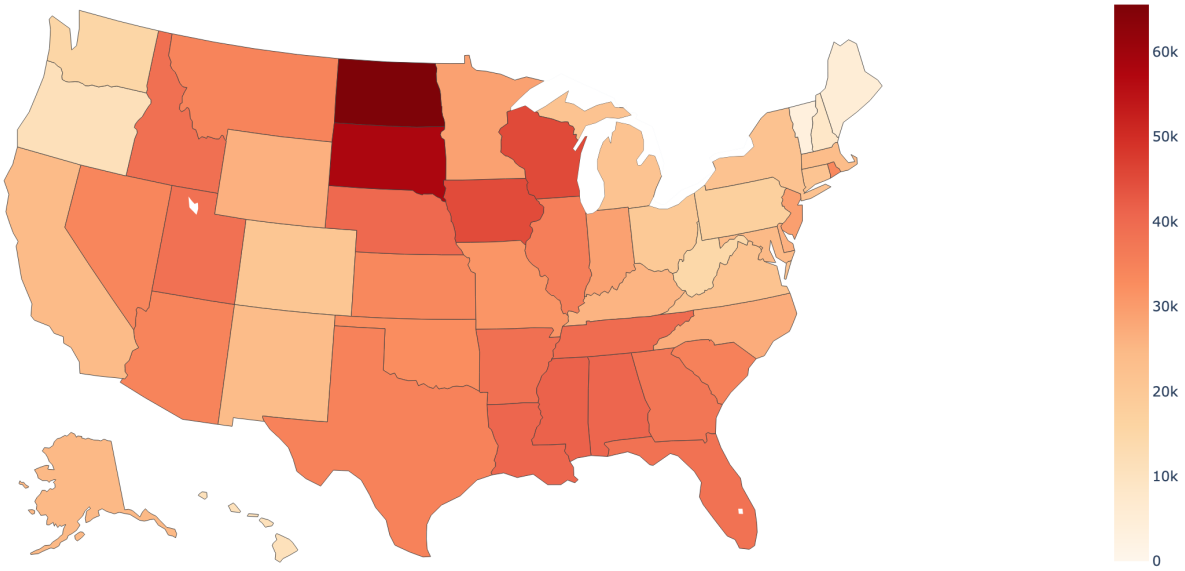
```
fig.update_layout(
    geo={
        "showframe": False,
        "lataxis": {"range": [-37, 68]},
        "lonaxis": {"range": [-130, 150]},
        "projection": {"type": "robinson"}
    },
    margin={"t": 0, "l": 10, "r": 10, "b": 0})
```



USA Choropleth

There are two differences when making a similar map for the USA. Set the `locationmode` parameter to "USA-states" so that plotly recognizes the two-character state code and choose the projection to be "albers usa" which moves Alaska and Hawaii near the other 48 states. Here, we color by "Cases per Million".

[22]:
```
df_states = df_summary.query("group == 'usa'")
locations = df_states['code']
z = df_states['Cases per Million']
text = df_states.apply(hover_text, axis=1)

fig = go.Figure()
fig.add_choropleth(locations=locations, locationmode='USA-states', z=z, zmin=0,
                   colorscale="orrd", marker_line_width=0.5, text=text,
                   hoverinfo="text")
fig.update_layout(
    geo={
        "showframe": False,
        "projection": {"type": "albers usa"}
    },
    margin={"t": 0, "l": 10, "r": 10, "b": 0})
```



10.8 Plotly Summary

Plotly is a great tool for creating interactive data visualizations for the web. The three main steps for creating a visualization are:

1. Create Figure - with `go.Figure` or `make_subplots`
2. Add trace - with `fig.add_*`
3. Update layout - with `fig.update_layout` or `fig.update_*`

Traces

- A trace is plotly terminology for a "kind of plot" (scatter, bar, pie, box, choropleth, etc…)
- Find the trace you want on the left side of this page
  - Or type `fig.add_` and press tab
- Read documentation for a specific trace once selected e.g. `fig.add_scatter` -> shift + tab + tab
- Add as many traces as you want to one figure

Layout

- The layout is where properties such as height, width, title, xaxis/yaxis, legend, annotations, etc… are set
- Use `fig.update_layout` to set properties for entire figure
- Documentation does NOT show parameters with `fig.update_layout`
  - Discover them with `fig.layout.` + tab
  - Read documentation on specific property `fig.layout.title` -> shift + tab + tab

Subplots

- Create grid of subplots with `make_subplots` using `rows` and `cols`
- All trace methods, `fig.add_*`, have `row` and `col` to specify subplot
- Use `fig.update_layout` to change properties on entire figure
- Other `fig.update_*` methods exist that have `row` and `col` parameters to change specific subplot

Choropleth

- Colored polygons (countries and states for our project)
- Some properties are in `fig.add_choropleth`, others are in `fig.update_layout` using `geo` parameter
- Set `locations` to be code (ISO-3 for countries and two-character abbreviation for states)
- Set `locationmode` to be "USA-States" for USA
- Set projection and range (`latrange`/`lonrange`) for world
- Set projection to be "albers usa" for USA

10.9 More to Plotly

The purpose of this chapter was to provide you with a simple and straightforward approach to using plotly for our project. There is much more to the library and multiple ways to interface with it. One newer and popular way for creating plotly graphs is with plotly express, which is similar to the seaborn libray, in that it automatically groups and aggregates values for you. If you are interested in learning more about plotly, I would recommend waiting until after the completion of this book, as there is already a tremendous number of items covered and getting side tracked on the details of plotly will not help. The methods taught in this chapter (create figure, add trace, update layout) should give you the power to create nearly any plot and style it as you desire.

Chapter 11

Intro to HTML and CSS

In this chapter, a brief introduction to HTML and CSS, the building blocks of all web pages, will be presented. As this topic can fill several volumes of text, only the basics necessary to understand the dashboard will be shown. Along the way, many references will be made to the MDN web docs from Mozilla, an excellent source for HTML/CSS material.

11.1 What is a web page?

A simple web page is composed of HTML, a language to describe the components of the page, and CSS, a language to arrange and add style to the components. Your web browser is what is responsible for interpreting the HTML and CSS and rendering it as the web page you see.

11.2 Intro to HTML

HTML is composed of **elements**, which are the components of a web page. Each element has a specific **tag** that is used to define it. A tag is a name surrounded by **angle brackets**. Visit this page to view all 100+ HTML elements.

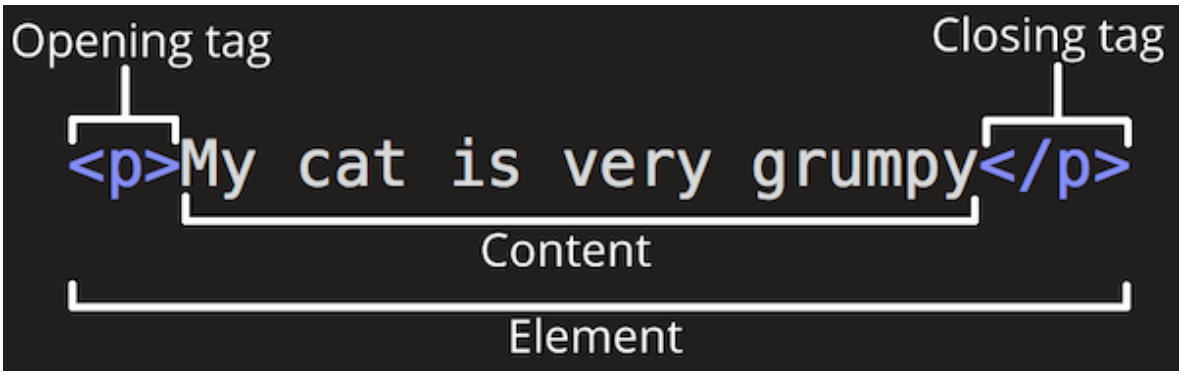
Common HTML elements

The most common elements with their tags are shown below and are linked to their MDN page with a detailed description.

- `<p>` - paragraph of text
- `<h1>`, `<h2>`, `<h6>` - largest, second largest, and smallest headers
- `<a>` - anchor, a clickable hyperlink
- `<img>` - image
- `<ul>` - unordered list
- `<ol>` - ordered list
- `<li>` - list element
- `<table>` - table with rows and columns
- `<div>` - logical division of content. Does not affect structure without CSS

Creating HTML elements

Most HTML elements are created with opening and closing tags surrounding the content. This image courtesy of MDN shows the basic structure of an HTML element.



Element attributes

HTML elements can have attributes that change the appearance or functionality of the element. Different elements will have different attributes. They are similar to function parameters in Python. Again, courtesy of MDN, we see an attribute within the paragraph element.

Attributes will always appear **within the opening tag**, are followed by an equal sign and contain their value within quotes.

Global attributes

There are two kinds of element attributes, **global** attributes and **element-specific** attributes. Global attributes are available to every single HTML element. The most common two are `class` and `id`, which help identify a group of similar elements or a single element, and are useful when applying CSS. Element-specific attributes are those available to a single element, such as `href` for the anchor element `<a>`.

Empty elements

A few elements have only opening tags. They have no content and no closing tag. However, they can have attributes. The image tag, `<img>`, is probably the most common one, with an example below. A list of all the empty elements can be found here.

```
<img src="images/cover.png">
```

11.3 Writing HTML in the notebook

You can write HTML directly in a code cell of a Jupyter Notebook by using the `%%html` magic command as the first line in that cell, passing it the option `--isolated` so that it does not interact with any of

the other HTML or CSS on the page. The cell contents will be interpreted as HTML and not python. Here, we create two different headers and two paragraphs with the rendered HTML below.

[1]:
```
%%html --isolated

<h1>Our first HTML</h1>
<p>This is an HTML paragraph element </p>
<h3>HTML elements have opening and closing tags surrounding content</h4>
<p>The elements have attributes in the opening tag</p>
```

Our first HTML

This is an HTML paragraph element

HTML elements have opening and closing tags surrounding content

The elements have attributes in the opening tag

11.4 Examples of common elements

In this section we'll create examples of the most common HTML elements. Make sure to click on the links to MDN's documentation to read more about each element and to see its list of element-specific attributes. Two of the simplest elements are the headers, `<h1>` through `<h6>`, and the paragraph, `<p>`, which we saw above. We'll continue with other elements below.

The "anchor" element - creating hyperlinks - `<a>`

The anchor element, created with the `<a>` tag, makes hyperlinks by setting the `href` attribute to the URL. The content will be underlined and colored blue in most browsers. When hovering over it, the cursor will transform into a hand with an index finger. We also set the `target` attribute to `"_blank"`, which will open the link in a new tab. By default, the browser navigates away from the current page, remaining in the current tab.

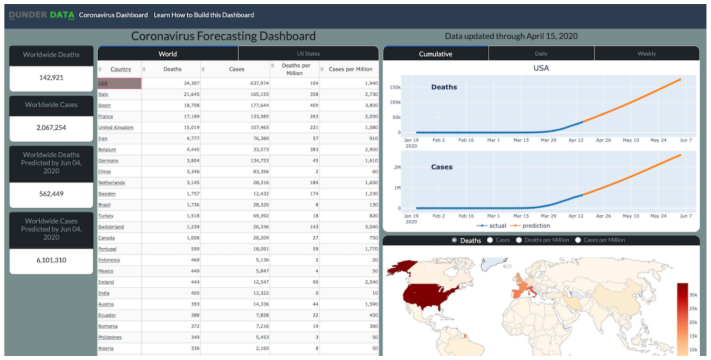
[2]:
```
%%html --isolated

<a href="https://coronavirus.dunderdata.com" target="_blank">Coronavirus Dashboard</
↪a>
```

Coronavirus Dashboard

The image element - `<img>`

The image element embeds images into the document using the empty `<img>` tag. It's main attribute is `src` which is required and set to the URL of the image. You can link to a local image, as we do below. We also use the `height` attribute to set its height in pixels, which automatically scales the width along with it. Take note that there is no content or closing tag with `<img>` as it is an empty tag.

[3]:
```
%%html --isolated

<img src="images/dash.png" height="300">
```



Unordered and ordered lists and list items - , <ol>, and

Both unordered (<ul>) and ordered (<ol>) lists require **nesting** elements - placing elements within another element's content section. Nested elements are usually indented to help indicate visually that it is nested. The outer element is referred to as the **parent** and the nested element as the **child**.

Both unordered and ordered lists use the list item element, <li>, as their nested child element. Concretely, <li> must appear within <ul> or <ol>. Here, we create an unordered list nested with two further unordered lists and an ordered list.

[4]:
```
%%html --isolated

<h3>There are over 100 HTML elements</h3>
<ul>
    <li>anchor element</li>
    <ul>
        <li>href - URL of hyperlink</li>
        <li>target</li>
        <ul>
            <li>"_self" - opens link in current tab</li>
            <li>"_blank" - opens link in new tab</li>
        </ul>
    </ul>
</ul>

<h3>Data Steps</h3>
<ol>
    <li>Collect data</li>
    <li>Clean data</li>
    <li>Smooth data</li>
    <li>Model data</li>
</ol>
```

There are over 100 HTML elements

- anchor element
  - href - URL of hyperlink
  - target
    - "_self" - opens link in current tab
    - "_blank" - opens link in new tab

Data Steps

1. Collect data
2. Clean data
3. Smooth data
4. Model data

Table elements - `<table>`

The HTML table, `<table>`, is one of the more complex elements requiring nesting of the table header `<thead>` and table body `<tbody>`, and within those, the rows `<tr>`, and within that, the row/column headers `<th>` and the table data `<td>`. There are four levels to the table hierarchy shown below.

- `<table>`
  - `<thead>`
    - `<tr>`
      - `<th>` or `<td>`
  - `<tbody>`
    - `<tr>`
      - `<th>` or `<td>`

The `thead` and `tbody` tags are not strictly necessary to create a table and it will render the same way without them, but it is considered good practice, as it logically separates the two table components. The table head,`<th>`, and table data, `<td>`, elements are similar and children of the table row, `<tr>`. Although the first element of each table row uses `<th>`, this isn't necessary. They can all be either `<th>` or `<td>`.

There are only global attributes available to `<table>`, `<thead>`, `<tbody>`, and `<tr>`. Both `<th>` and `<td>` have element-specific attributes, with `rowspan` and `colspan` being common. Set them equal to an integer for the element to span multiple rows and/or columns.

[5]:
```
%%html --isolated
<table>
    <thead>
        <tr>
            <th>Area</th>
            <th>Deaths</th>
            <th>Cases</th>
        </tr>
    </thead>

    <tbody>
        <tr>
```

```
            <th>Texas</th>
            <td>10</td>
            <td>20</td>
        </tr>

        <tr>
            <th>Florida</th>
            <td>8</td>
            <td>17</td>
        </tr>

        <tr>
            <th>Alaska</th>
            <td>2</td>
            <td>6</td>
        </tr>

        <tr>
            <th colspan="3">Countries</th>
        </tr>

        <tr>
            <th>Brazil</th>
            <td>21</td>
            <td>116</td>
        </tr>
    </tbody>
</table>
```

Area	Deaths	Cases
Texas	10	20
Florida	8	17
Alaska	2	6
Countries		
Brazil	21	116

Logical division of content - `<div>` - the element that does (almost) nothing

The content division element, `<div>`, is used to logically group together sections of your page. It has no effect on the content until you style that section with CSS. A `<div>` has no element-specific attributes, but is usually labeled with the `class` or `id` global attributes so it can be referenced in CSS. Here, we've used `div` to logically divide the page into two sections, using the `id` attribute to label them as either "html_info" or "data_info". This is the exact same HTML as presented in the unordered and ordered lists section above and will render the exact same. In the future, we can reference one of the sections using its `id` for styling with CSS.

[6]:
```
%%html --isolated

<div id="html_info">
    <h3>There are over 100 HTML elements</h3>
    <ul>
        <li>anchor element</li>
        <ul>
            <li>href - URL of hyperlink</li>
            <li>target</li>
            <ul>
                <li>"_self" - opens link in current tab</li>
                <li>"_blank" - opens link in new tab</li>
            </ul>
        </ul>
    </ul>
</div>

<div id="data_info">
    <h3>Data Steps</h3>
    <ol>
        <li>Collect data</li>
        <li>Clean data</li>
        <li>Smooth data</li>
        <li>Model data</li>
    </ol>
</div>
```

There are over 100 HTML elements

- anchor element
  - href - URL of hyperlink
  - target
    - "_self" - opens link in current tab
    - "_blank" - opens link in new tab

Data Steps

1. Collect data
2. Clean data
3. Smooth data
4. Model data

11.5 HTML Syntax

HTML is not a programming language, but a **markup** language. It cannot be used to solve technical problems, run logical conditions, iterate with loops, do calculations, etc… It simply describes the components of the page. A web browser is needed to interpret the tags and render them on your screen.

One interesting fact about HTML is that your browser will always render it, regardless of its format, no

matter how many mistakes there are in it. You can forget closing tags, have typos, not nest elements properly, and yet all browsers will render something. While this can be good, in that a website won't be brought down because of a forgotten closing tag, it can make debugging much harder. Fortunately, there are many tools, both in the browser and in modern editors that can make finding these mistakes much easier. We will not be writing HTML directly, so we won't have to worry about this.

Whitespace in your HTML will mostly be ignored. You can write all your HTML on a single line and the browser will not complain. However, if you have consecutive white spaces inside the content of certain elements such as headers and paragraphs, they will be treated as a single space.

[7]:
```
%%html --isolated

<p>Words      separated     by    many spaces     only  render as a single space</p>
```

Words separated by many spaces only render as a single space

11.6 Block vs Inline elements

All elements are classified as either **block** or **inline**.

- Block elements:
  - Are placed on a new line of their own
  - Have their width as large as the screen (or their parent, if nested)
- Inline elements:
  - Remain in the current line they are in
  - Have width only as wide as necessary to fit in the line

This division between block and inline elements will become clearer once we add some styling. But, before we do that, take a look at the following HTML to see if you can determine which elements are block and which are inline.

[8]:
```
%%html --isolated

<h3>Quiz - Can you determine whether an element is block or inline?</h3>
<p>Some paragraph of text</p>
<p>Another paragraph</p>   <p>YET Another paragraph</p>
<p>A paragraph with a <a href="https://google.com">link to google</a> inside of it</
 ↪p>
<p>How about images?</p>
<img src="images/dash.png" height="100">
<img src="images/dash.png" height="100">
```

Quiz - Can you determine whether an element is block or inline?

Some paragraph of text

Another paragraph

YET Another paragraph

A paragraph with a link to google inside of it

How about images?

If the element is placed on a separate line, then it's a block-level element. If it remains on the same line, then it's inline. All headers and the paragraph elements are block-level. Even if there is space for the contents of a paragraph to fit on the same line as another element, it will always be placed on a separate line.

Anchor elements are inline and remain in the same line as the elements they are placed within. Images are also inline as each of the two above are placed in the same line.

11.7 Styling with CSS

Cascading Stylesheets, or CSS, is the language used to style the elements and control their position on the page. It is completely different than HTML with its own syntax. It's best to write CSS in a separate file and link it to the HTML page via the empty `<link>` element. For example, you would place the following in your HTML and write your CSS in the `style.css` file.

```
<link href="style.css" rel="stylesheet">
```

CSS can also be placed within the style HTML element, `<style>`, which we will do in this notebook. This is done so that we have it together in the same cell with the HTML. For our project, the CSS will be written in a separate file.

Basic CSS Syntax

The goal of CSS is to change the properties of particular HTML elements. We change the properties by creating **rules**. A rule consists of a **selector** and the **property-value pairs**. A selector selects one or more elements to apply the styling. There are many ways to select elements with the most basic being by tag, class name, or id. The generic syntax for a single CSS rule is as follows:

```
selector {
 property1: value1;
 property2: value2;
 property3: value3;
}
```

A set of opening and closing curly braces follow the selector. Within the curly braces are the property-value pairs with a colon separating the property and value. A semicolon ends each property-value pair.

11.8 CSS properties

There are a huge number of CSS properties available. Visit this page from Tutorial Republic to see them broken into categories (background, border, font, etc…). To help narrow the list, I've created the following table with some of the most common properties and common values. The list of values below is incomplete. Click the link to take you to the MDN reference page for a complete description.

Property	Values	Notes
`color`	One of ~140 color names	Sets text color. Could have been named `text-color`
`text-align`	left, right, center, justify	Horizontal alignment
`vertical-align`	baseline, top, middle, bottom	Vertical alignment
`font-family`	Helvetica, Arial, sans-serif	List of font families separated by a comma
`font-size`	20px	Size of font. Many units of measurements + absolute/relative sizes
`font-weight`	bold	Boldness of font. Can use a number 1-1000
`background-color`	One of ~140 color names	Sets background color of element
`border`	1px dashed red	Sets three properties at once - border width, style, and color
`margin`	20px	Sets the margin size of all four sides of the element
`height` / `width`	500px	Sets the element's height/width
`max-height` / `max-width`	300px	Sets the max height/width
`min-height` / `min-width`	100px	Sets the min height/width
`display`	inline, block, flex, grid	Controls the layout of the container

11.9 Applying CSS using tags as selectors

The simplest selector is just the name of a tag (`h1`, `p`, `div`, `a`, etc…) without the angle brackets. We'll begin by creating one header and one paragraph element without any CSS styling.

[9]:
```
%%html --isolated

<h2>Beginning CSS</h2>
<p>Here is some text in a paragraph element</p>
```

Beginning CSS

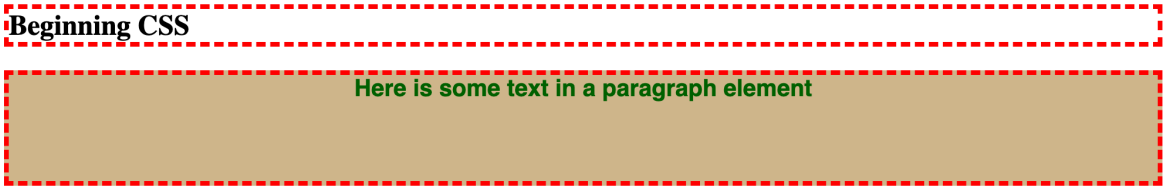
Here is some text in a paragraph element

Adding the style

The same two elements will now be styled. Below, we create one rule for each element using the tag name as the selector. We set many properties of the paragraph, while only adding a border to the header. The `px` is one of the several types of units that CSS understands and represents pixels, where 1px is 1/96th of an inch.

[10]:
```
%%html --isolated

<style>
```

```
h2 {
    border: 4px red dashed;
}

p {
    color: darkgreen;
    font-family: Helvetica;
    font-size: 20px;
    font-weight: 800;
    height: 90px;
    text-align: center;
    border: 4px dashed red;
    background-color: tan;
}
</style>

<h2>Beginning CSS</h2>
<p>Here is some text in a paragraph element</p>
```



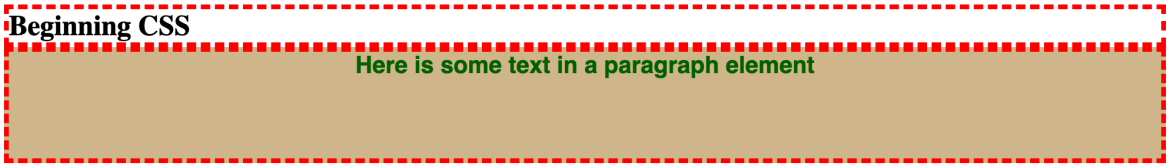
Element borders

I think it's helpful to show the borders of elements to visually see the extent of each one. You should clearly see that the width of each element stretches the entire width of the screen (or in this case, the parent, which is the notebook cell). They stretch the entire width of their parent because they are block elements.

Also notice that there is some vertical space between the two elements. They aren't stacked directly on top of one another. Almost all block elements have some default value for their top and bottom **margin**. We can eliminate this margin by setting it to 0px.

[11]:
```
%%html --isolated

<style>
h2 {
    border: 4px red dashed;
    margin: 0px;
}

p {
    color: darkgreen;
    font-family: Helvetica;
    font-size: 20px;
    font-weight: 800;
```

```
    height: 90px;
    text-align: center;
    border: 4px dashed red;
    background-color: tan;
    margin: 0px;
}
</style>

<h2>Beginning CSS</h2>
<p>Here is some text in a paragraph element</p>
```



11.10 Shorthand property names

Both `margin` and `border` (as well as several others) are **shorthand** property names, meaning that they allow you to set multiple CSS properties at once. For instance, `margin` is shorthand for the following properties:

- `margin-top`
- `margin-right`
- `margin-bottom`
- `margin-left`

Setting `margin` to a single value, sets all four properties to that value. You can also give it four separate values, e.g. `margin: 10px 30px 0px 5px;`

Similarly, `border` is shorthand for:

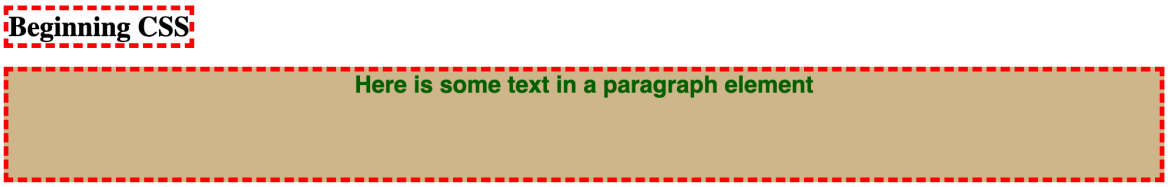
- `border-width` - 4px
- `border-style` - dashed, solid, dotted
- `border-color` - same as `color`

11.11 Changing block to inline

By default, these elements are both block-level. We can change this by setting the `display` property to `inline`. Below, the paragraph is kept as a block element so remains on its own line.

[12]:
```
%%html --isolated

<style>
h2 {
    border: 4px red dashed;
    display: inline;
}
```

```
p {
    color: darkgreen;
    font-family: Helvetica;
    font-size: 20px;
    font-weight: 800;
    height: 90px;
    text-align: center;
    border: 4px dashed red;
    background-color: tan;
}
</style>

<h2>Beginning CSS</h2>
<p>Here is some text in a paragraph element</p>
```



Making both elements inline moves the paragraph up to the same line as the header. The height property is ignored for inline elements.

[13]:
```
%%html --isolated

<style>
h2 {
    border: 4px dashed red;
    display: inline;
}

p {
    color: darkgreen;
    font-family: Helvetica;
    font-size: 20px;
    font-weight: 800;
    height: 90px;
    text-align: center;
    border: 4px dashed red;
    background-color: tan;
    display: inline;
}

</style>

<h2>Beginning CSS</h2>
<p>Here is some text in a paragraph element</p>
```

Beginning CSS Here is some text in a paragraph element

Here, we add an anchor link nested within the paragraph tag. Anchor links have a default display of inline, which we change to block. This has the affect of placing it on its own line. Notice that the text properties for the anchor element inherit from its parent.

[14]:
```
%%html --isolated

<style>
h2 {
    border: 4px red dashed;
}

p {
    color: darkgreen;
    font-family: Helvetica;
    font-size: 20px;
    font-weight: 800;
    height: 90px;
    text-align: center;
    border: 4px dashed red;
    background-color: tan;
    display: inline;
}

a {
    display: block;
}

</style>

<h2>Beginning CSS</h2>
<p>Text <a href="http://google.com" target="_blank">with google link</a>
    in a paragraph element</p>
```

Beginning CSS

Text with google link in a paragraph element

11.12 Selecting elements by class and id

Selecting elements by their tag name is often too broad for most web pages as it targets too many elements. More often, you'll select elements by their class or id. Both class and id are global HTML attributes. They both provide a way to label elements so that they can be referenced later. The class name is used to label one or more elements, while id is used to label a single element and therefore should be unique.

Selectors - Use . for class and # for id

In the stylesheet, the selector must be preceded by a period for classes and a hash sign for ids. Our dashboard uses classes and ids for nearly all selectors.

Here, we create three paragraph elements and use the `p` selector to share several of the same properties amongst them. The top two paragraphs are labeled with the class name `large`. The unique id `center_small` is given to the bottom one. Note the coloring of the syntax below makes it appear that `#center_small` is commented out. Unfortunately, the notebook treats it as python code, for which the `#` is used for comments.

[15]:
```
%%html --isolated

<style>
p {
    border: 2px dashed red;
    color: darkgreen;
    font-family: Helvetica;
    background-color: tan;
    width: 500px;
}

.large {
    font-size: 20px;
}

#center_small {
    text-align: center;
    font-size: x-small;
}
</style>

<p class="large">Paragraph 1</p>
<p class="large">Paragraph 2</p>
<p id="center_small">Here is some text in a paragraph element</p>
```

Paragraph 1

Paragraph 2

Here is some text in a paragraph element

11.13 Page layout with Flexbox and Grid

As you can see, the `display` property is important for changing the position of elements on the page. The position of all the elements of a page will be referred to as the **layout**. In modern CSS, **flexbox** and **grid** are two good choices to lay out all of the elements in a page. They are easier to work with and more powerful than their predecessors.

Boxes - Containers and Items

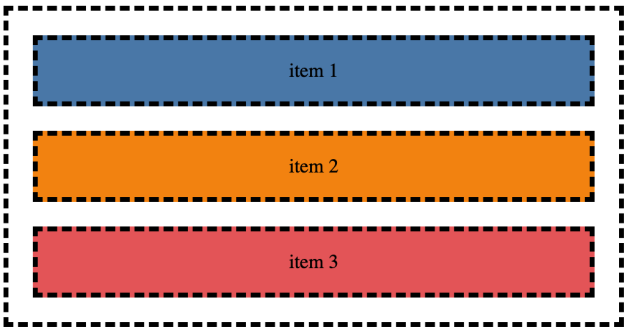
It's often best to think of each element of your web page as a rectangular box. Both flexbox and grid will help you arrange these rectangular boxes on the screen. There are usually two kinds of boxes, the **container** and the **items**. The container, usually a single `<div>`, contains several **items** (any other element including divs).

Below, we create one container div (the parent) and three item divs (the children). Since divs are block-level elements, they are each displayed on a separate line. Each item div contains a single paragraph element with text labeling the item. The container div is the outer div and not labeled.

Each item is given a class and id so that styles can be applied to each of them simultaneously and individually. We use the id selector to color the background of each individually, and the class name to provide the same margin (20px) to all of them. All divs have a black dashed border so that you can see their outer rectangular boundary.

[16]:
```
%%html --isolated

<style>
    div {
        border: 4px black dashed;
    }

    p {
        text-align: center;
    }

    .container {
        width: 500px;
    }

    .item {
        margin: 20px;
    }

    #item1 {
        background-color: #4C78A8;
    }

    #item2 {
        background-color: #F58518;
    }

    #item3 {
        background-color: #E45756;
    }
</style>

<div class="container">
    <div class="item" id="item1">
```

```
        <p>item 1</p>
    </div>
    <div class="item" id="item2">
        <p>item 2</p>
    </div>
    <div class="item" id="item3">
        <p>item 3</p>
    </div>
</div>
```



11.14 CSS Flexbox layout

Flexbox is a **one-dimensional** layout system for placing items either horizontally or vertically. Only a portion of the flexbox features will be covered here. Visit the MDN documentation for a full tutorial. Flexbox's default layout is **horizontal**. To get started, set `display: flex` for the **container**. We do that below, and remove the margin so that you can see exactly how flexbox places the items.

[17]:
```
%%html --isolated

<style>
    div {
        border: 4px black dashed;
    }

    p {
        text-align: center;
    }

    .container {
        width: 500px;
        display: flex;
    }

    #item1 {
        background-color: #4C78A8;
    }

    #item2 {
```

```
        background-color: #F58518;
    }

    #item3 {
        background-color: #E45756;
    }
</style>

<div class="container">
    <div class="item" id="item1">
        <p>item 1</p>
    </div>
    <div class="item" id="item2">
        <p>item 2</p>
    </div>
    <div class="item" id="item3">
        <p>item 3</p>
    </div>
</div>
```

With one small change, we get our items in a single row and they no longer stretch the entire width of their parent. You'll often use the `justify-content` property of the container to space the items to your liking. Here, we set it to the value `space-evenly` to space the items evenly in the container.

[18]:
```
%%html --isolated

<style>
    div {
        border: 4px black dashed;
    }

    p {
        text-align: center;
    }

    .container {
        width: 500px;
        display: flex;
        justify-content: space-evenly;
    }

    #item1 {
        background-color: #4C78A8;
    }

    #item2 {
```

```
        background-color: #F58518;
    }

    #item3 {
        background-color: #E45756;
    }
</style>

<div class="container">
    <div class="item" id="item1">
        <p>item 1</p>
    </div>
    <div class="item" id="item2">
        <p>item 2</p>
    </div>
    <div class="item" id="item3">
        <p>item 3</p>
    </div>
</div>
```

By default, items will only take up as much space as their content. Use `flex-grow` as an **item** property (not a container property) to have a specific item fill the empty space. This value is 0 by default. The value of 1 represents 100% of the remaining space. Below, we have item1 and item2 fill up 30% and 50% of the remaining space, with item3 remaining at its original size.

[19]:
```
%%html --isolated

<style>
    div {
        border: 4px black dashed;
    }

    p {
        text-align: center;
    }

    .container {
        width: 500px;
        display: flex;
        justify-content: space-evenly;
    }

    #item1 {
        background-color: #4C78A8;
        flex-grow: .3;
```

```
    }

    #item2 {
        background-color: #F58518;
        flex-grow: .5;
    }

    #item3 {
        background-color: #E45756;
    }
</style>

<div class="container">
    <div class="item" id="item1">
        <p>item 1</p>
    </div>
    <div class="item" id="item2">
        <p>item 2</p>
    </div>
    <div class="item" id="item3">
        <p>item 3</p>
    </div>
</div>
```

Flexbox summary

- Container
  - Use `display: flex`
  - Default layout is horizontal. Use `flex-direction: column` to switch
  - Use `justify-content` with possible values `flex-start`, `flex-end`, `space-around`, `space-evenly`, `space-between`
  - Wrap items onto new lines with `flex-wrap: wrap`
- Item
  - Expand individual items with `flex-grow` setting it to a number, where 1 represents 100% of the available free space.

11.15 CSS Grid layout

The grid layout is similar to flexbox, but is used for **two-dimensional** layouts with rows and columns. There are a variety of ways to create the rows and columns, but only the method using `grid-template-areas` will be shown. Use the MDN documentation for a full tutorial of all the features. Using the grid layout is a bit more work than flexbox, but often is a good choice when you have many elements across and down your page.

Grid syntax using `grid-template-areas`

The outer **container** must have its display property set to `grid`. The property `grid-template-areas` defines the grid using names for each cell of the grid. In the example below, a grid with 3 rows and 4 columns is created. Names can be any string of non-space characters and separated from one another by a space. Each row of the grid must be enclosed in quotes.

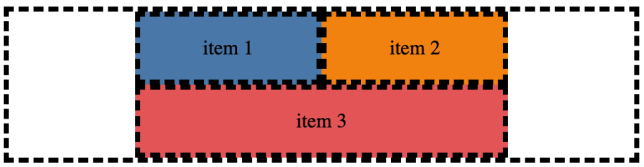
Below, we use single characters as names for each of the 12 cells. Each **item** is then assigned to one cell with the `grid-area` property. The element with id equal to item1 is assigned to grid area `a`, with item2 assigned to `g`.

```
.container {
        display: grid;
        grid-template-areas:
            "a b c d"
            "e f g h"
            "i j k l";
}

#item1 {
        grid-area: a;
}

#item2 {
        grid-area: g;
}
```

Continuing with our previous example with three items, we create a grid with two rows and four columns and use more descriptive names for each of the areas. Multiple grid areas can have the same name, allowing one item to take the space of multiple cells. The div with id equal to item3 is assigned the grid area `bottom` which labels the two middle cells on the bottom row. The grid areas with labels `left` and `right` have no items assigned to them.

[20]:
```
%%html --isolated

<style>
    div {
        border: 4px black dashed;
    }

    p {
        text-align: center;
    }

    .container {
        width: 500px;
        display: grid;
        grid-template-areas:
            "left topleft topright right"
            "left bottom  bottom   right";
```

```
    }

    #item1 {
        background-color: #4C78A8;
        grid-area: topleft;
    }

    #item2 {
        background-color: #F58518;
        grid-area: topright;
    }

    #item3 {
        background-color: #E45756;
        grid-area: bottom;
    }
</style>

<div class="container">
    <div class="item" id="item1">
        <p>item 1</p>
    </div>
    <div class="item" id="item2">
        <p>item 2</p>
    </div>
    <div class="item" id="item3">
        <p>item 3</p>
    </div>
</div>
```

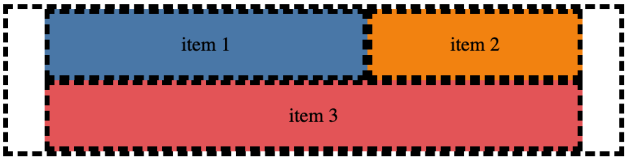


Sizing grid cells

By default, all cells have the same width with elements expanding to that width. We use `grid-template-columns` to specify the width of each column and set it to be four separate values. The outer columns are set to 30 pixels each. The unit `fr` is similar to `flex-grow` and represents the proportion of free space remaining. In this case, it represents the 440 pixels (500 - 2 * 30) remaining. The length `1fr` equals this amount. Below, we use `2fr` and `3fr` for the top two items, which splits the free space proportionally between them.

[21]:
```
%%html --isolated

<style>
    div {
```

```
        border: 4px black dashed;
    }

    p {
        text-align: center;
    }

    .container {
        width: 500px;
        display: grid;
        grid-template-areas:
            "left topleft topright right"
            "left bottom  bottom   right";
        grid-template-columns: 30px 3fr 2fr 30px;
    }

    #item1 {
        background-color: #4C78A8;
        grid-area: topleft;
    }

    #item2 {
        background-color: #F58518;
        grid-area: topright;
    }

    #item3 {
        background-color: #E45756;
        grid-area: bottom;
    }
</style>

<div class="container">
    <div class="item" id="item1">
        <p>item 1</p>
    </div>
    <div class="item" id="item2">
        <p>item 2</p>
    </div>
    <div class="item" id="item3">
        <p>item 3</p>
    </div>
</div>
```



CSS Grid Summary

- Container
  - Use `display: grid`
  - Create grid and name all cells with `grid-template-areas`
  - Set column width with `grid-template-columns`
    - Can use `fr` unit for free space
  - Set row height with `grid-template-rows`
- Item
  - Use `grid-area` to assign an element to a grid cell

11.16 Much more to HTML and CSS

This chapter provides the essentials for building simple web pages with HTML and CSS. Many topics were omitted to keep the focus on the components that make an appearance in our dashboard.

Chapter 12

Building the Dashboard with Dash

We're finally ready to start building our dashboard with Dash, a free and open source Python library created and maintained by the company Plotly. Dash provides many tools to display data visually as well as allow users to interact with it.

Dash is built directly on top of Flask, one of the most popular web frameworks for Python, and react.js, an open source JavaScript library originally created by Facebook. Dash directly integrates with the plotly library. It does not make any plots on its own, but is equipped to present any plotly figure within it.

12.1 Parts of a Dash application

There are two parts of a Dash application:

- Layout
- Interactivity (Callbacks)

Layout

The layout describes the physical components of the application. There are two broad types of layout components - **HTML elements** and **dash components**. The HTML elements are the exact same elements we covered in the previous chapter. We will not be writing HTML directly, but accessing the elements as Python classes from the **dash_html_components** module.

The dash components combine together multiple HTML elements, CSS, and JavaScript into a single component. Examples of these include dropdown menus, checklists, tabs, placeholders for plotly figures, and sliders. They are accessed as single Python classes from the **dash_core_components** module. Additionally, there is an entire separate library, **dash_table**, that provides functionality for interactive data tables.

Interactivity (Callbacks)

The interactivity is the other part of the application, and is the code that makes changes to the layout components when a particular event occurs. Almost any event that occurs within a dash application can be mapped to some function that changes one of the layout components. These functions that are triggered are named **callbacks.** For example, when a country is clicked in our data table, three graphs are updated to show the clicked country's information.

12.2 Beginning a dash application

Typically, dash applications are written in a separate file, such as `dashboard.py` in our case. For purposes of instruction, the dashboard will be written in this Jupyter Notebook. When writing real dashboard applications, I strongly recommend beginning them in a normal text file and using an editor such as Sublime Text or Visual Studio Code.

Installing JupyterDash

Even though this tutorial takes place in a notebook, the code will be virtually identical to code that would appear in a separate text file. I recommend opening up `dashboard.py` in a separate editor now so that you can reference it during the chapter. The only difference is that we'll need to install a new library for it to work properly. Plotly released the JupyterDash library to help those develop Dash within a notebook. You should have installed this library during chapter 1.

Minimal dashboard

Your dashboard will need a minimum of three lines of code to run - one to create the application with `JupyterDash`, another to set the `layout` attribute, and lastly, one to execute the `run_server` method with `mode` set to `"inline"`, so it displays in the notebook. Here, we assign the layout to be a single HTML element, an h2 header and limit the height of the entire dashboard.

[1]:
```
from jupyter_dash import JupyterDash
import dash_html_components as html

app = JupyterDash(__name__)
app.layout = html.H2('Coronavirus Forecasting Dashboard')
app.run_server(mode='inline', height=100)
```

Coronavirus Forecasting Dashboard

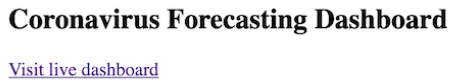
12.3 HTML elements in dash

Nearly all normal HTML elements are available as Python classes after importing `dash_html_components` and aliasing it as `html`. Element names have their first letter capitalized. The first argument for every HTML class in dash is the element's content, formally given the parameter name **children**. The content is often a string for paragraph, header, and anchor elements. Use a list when the content consists of multiple other HTML elements (which is often the case for divs).

Below, we create an anchor element and then place it and the header inside of a div. Dash does not allow you to assign the `layout` attribute to a list of elements. You must set it to a single HTML element, so we are forced to use a div to contain all of the elements.

[2]:
```
title = html.H2('Coronavirus Forecasting Dashboard')
link = html.A('Visit live dashboard', href='https://coronavirus.dunderdata.com')
layout = html.Div([title, link])

app = JupyterDash(__name__)
```

```
app.layout = layout
app.run_server(mode='inline', height=100)
```



Adding CSS

In our actual project, most CSS will be kept in the external stylesheet, style.css. For this tutorial, it will be easier to see the CSS together with elements. Unfortunately, the `dash_html_components` library does not have a style element. However, CSS may be added with the `style` parameter, setting it to a dictionary with each property mapped to its value, with the caveat that attribute names are written in **camelCase**. Also, you cannot set the `class` or `id` within the `style` dictionary, but must use the `id` and `className` parameters instead. The following list summarizes the differences writing HTML in dash:

- Nearly all normal HTML elements are available in `dash_html_components`
- Element names are capitalized
- Attributes are the same and available as parameters
- Set the `style` parameter to a dictionary to apply CSS
- Properties in the style dictionary are **camelCase**
- Set id and class with parameters `id` and `className`

Here, we use the same layout from above, but add style to the header.

[3]:
```
title = html.H2('Coronavirus Forecasting Dashboard',
                style={
                    'backgroundColor': 'tan',
                    'fontFamily': 'verdana',
                    'textAlign': 'center'
                }
               )
link = html.A('Visit live dashboard', href='https://coronavirus.dunderdata.com')
layout = html.Div([title, link])

app = JupyterDash(__name__)
app.layout = layout
app.run_server(mode='inline', height=100)
```

12.4 Creating a data table

A separate package, `dash_table`, contains the class `DataTable` to create interactive tables of data. Visit the official documentation for full coverage. There are many parameters that you can set during construction of the table:

- `data` - List of dictionaries where each list represents one row.
  - `df.to_dict('records')` converts a pandas DataFrame into the proper data structure
- `columns` - List of dictionaries where each item represents information (name, id, type, format, etc…) on one column.
  - `[{'name': name, 'id': name} for name in df.columns]`
- `sort_action` - Allow sorting of columns by setting to string `'native'`
- `active_cell` - Initial active cell - will be highlighted - `{"row": 0, "column": 0}`
- `style_table` - CSS for `table` HTML element (not for individual rows, columns, or cells)
- `style_cell` - CSS for ALL cells
- `style_header` - CSS for only the header (columns)
- `style_data` - CSS for only the data
- `style_data_conditional` - CSS based on if/else condition
  - `{"if": {"column_id": first_col}, "width": "120px", "textAlign": "left"}`
- `fixed_rows` - Keep the first `n` rows fixed when scrolling down. Mainly used to keep the column names on top.
  - `{ 'headers': True, 'data': n }`

Let's read in the summary table as a pandas DataFrame placing the group (either "world" or usa") in the index. It has one row per area for the "current" date.

[4]:
```
import pandas as pd
SUMMARY = pd.read_csv("data/summary.csv", index_col="group", parse_dates=["date"])
SUMMARY.head(3)
```

group	area	Daily Deaths	Daily Cases	Deaths	Cases	code	population	Deaths per Million	Cases per Million	date
world	Afghanistan	4	86	1548	41814	AFG	38.928341	40.0	1070.0	2020-11-05
world	Albania	7	421	543	22721	ALB	2.877800	189.0	7900.0	2020-11-05
world	Algeria	12	642	2011	60169	DZA	43.851043	46.0	1370.0	2020-11-05

This is the data we'd like to place within a dash data table. Before doing so, we'll select a subset of columns for just the world group and change the area column to "Country".

[5]:
```
group = "world"
used_columns = ["area", "Deaths", "Cases", "Deaths per Million", "Cases per␣
 ↪Million"]
df = SUMMARY.loc[group, used_columns]
df = df.rename(columns={"area": "Country"})
df.head(3)
```

group	Country	Deaths	Cases	Deaths per Million	Cases per Million
world	Afghanistan	1548	41814	40.0	1070.0
world	Albania	543	22721	189.0	7900.0
world	Algeria	2011	60169	46.0	1370.0

Now that we have our data, we need to provide column info to dash by creating a list of dictionaries. At a minimum, the `name` (visible label for column) and `id` (internal identification) must be present. We set both the name and id to the column name. The `type` ("any", "numeric", "text", or "datetime") is given along with setting `format` to a dictionary that uses the key `specifier` to set the formatting

(based on D3 format). The first column is the area, which is a text column. We iterate to append all of the other columns, which are numeric.

[6]:
```
columns = [{"name": "Country",
            "id": "Country", "type": "text"}]
for name in df.columns[1:]:
    col_info = {
        "name": name,
        "id": name,
        "type": "numeric",
        "format": {'specifier': ','}
    }
    columns.append(col_info)
```

We sort the DataFrame by deaths and then convert it to a list of dictionaries, which is necessary for the dash table.

[7]:
```
data = df.sort_values("Deaths", ascending=False).to_dict("records")
data[:3]
```

[7]:
```
[{'Country': 'USA',
  'Deaths': 236457,
  'Cases': 9721044,
  'Deaths per Million': 718.0,
  'Cases per Million': 29510.0},
 {'Country': 'Brazil',
  'Deaths': 161106,
  'Cases': 5590025,
  'Deaths per Million': 758.0,
  'Cases per Million': 26300.0},
 {'Country': 'India',
  'Deaths': 124985,
  'Cases': 8411724,
  'Deaths per Million': 91.0,
  'Cases per Million': 6100.0}]
```

We are now set to create our data table. We use `active_cell` to highlight and make the first cell active. We also use conditional styling to underline values in the first column and turn the cursor into a pointer to inform the user that it is clickable.

Note on table height

Unfortunately, behavior for table height is not consistent and requires us to set it using both `minHeight` and `height` CSS properties when using `fixed_rows`/`fixed_columns`. We set it using the units `vh`, which stands for **viewport height**, where 1vh represents 1% of the viewable height of the screen. Below, we set it to 80vh so that the table takes up 80% of the viewable screen height.

[8]:
```
from dash_table import DataTable
world_table = DataTable(
    id=f"{group}-table",
```

```
    columns=columns,
    data=data,
    fixed_rows={"headers": True},
    active_cell={"row": 0, "column": 0},
    sort_action="native",
    derived_virtual_data=data,
    style_table={
        "minHeight": "80vh",
        "height": "80vh",
        "overflowY": "scroll"
        },
    style_cell={
        "whitespace": "normal",
        "height":"auto",
        "fontFamily": "verdana"
    },
    style_header={
        "textAlign": "center",
        "fontSize": 14
    },
    style_data={
        "fontSize": 12
    },
    style_data_conditional=[
        {
            "if": {"column_id": "Country"},
            "width": "120px",
            "textAlign": "left",
            "textDecoration": "underline",
            "cursor": "pointer"
        },
        {
            "if": {"row_index": "odd"},
            "backgroundColor": "#fafbfb"
        }
    ],
)
layout = html.Div([title, world_table])

app = JupyterDash(__name__)
app.layout = layout
app.run_server(mode='inline', height=500)
```

Coronavirus Forecasting Dashboard

Country	Deaths	Cases	Deaths per Million	Cases per Million
USA	236,514	9,709,053	718	29,470
Brazil	161,106	5,590,025	758	26,300
India	124,985	8,411,724	91	6,100
Mexico	93,772	949,197	734	7,430
United Kingdom	48,210	1,126,471	710	16,590
Italy	40,192	824,879	665	13,640
France	39,088	1,650,965	599	25,290
Spain	38,486	1,306,316	823	27,940
Iran	36,985	654,936	440	7,800
Peru	34,730	914,722	1,053	27,740
Argentina	32,766	1,217,028	725	26,930
Colombia	32,209	1,117,977	633	21,970

The function below encapsulates all of our work from above and is nearly identical to the one found in `dashboard.py`.

[9]:
```
def create_table(group):
    used_columns = [
        "area",
        "Deaths",
        "Cases",
        "Deaths per Million",
        "Cases per Million",
    ]
    df = SUMMARY.loc[group, used_columns]
    first_col = "Country" if group == "world" else "State"
    df = df.rename(columns={"area": first_col})

    columns = [{"name": first_col, "id": first_col}]
    for name in df.columns[1:]:
        col_info = {
            "name": name,
            "id": name,
            "type": "numeric",
            "format": {'specifier': ','},
        }
        columns.append(col_info)

    data = df.sort_values("Deaths", ascending=False).to_dict("records")
    return DataTable(
        id=f"{group}-table",
        columns=columns,
        data=data,
        active_cell={"row": 0, "column": 0},
        fixed_rows={"headers": True},
```

```
        sort_action="native",
        derived_virtual_data=data,
        style_table={
            "minHeight": "80vh",
            "height": "80vh",
            "overflowY": "scroll",
            "borderRadius": "0px 0px 10px 10px",
        },
        style_cell={
            "whiteSpace": "normal",
            "height": "auto",
            "fontFamily": "verdana",
        },
        style_header={
            "textAlign": "center",
            "fontSize": 14,
        },
        style_data={
            "fontSize": 12,
        },
        style_data_conditional=[
            {
                "if": {"column_id": first_col},
                "width": "120px",
                "textAlign": "left",
                "textDecoration": "underline",
                "cursor": "pointer",
            },
            {
                "if": {"row_index": "odd"},
                "backgroundColor": "#fafbfb"
            }
        ],
    )
```

Let's use it to create the USA data table and recreate the world table as well. These tables will be accessible from their respective tab.

[10]:
```
world_table = create_table("world")
usa_table = create_table("usa")
```

12.5 Dash core components

The dash_core_components library, aliased as `dcc`, contains many interactive widgets such as checklists, dropdown menus, buttons, tabs, and others. You can think of these widgets as a collection of HTML, CSS, and JavaScript wrapped into a single Python class that is ready to use.

Creating tabs

Let's create two tabs, one for each of the world and USA. To do so we'll need both the `Tabs` and `Tab` classes from `dash_core_components`. The `Tabs` component is the container for each individual `Tab`, which is the container for the content (data table in this example). Here, we create two individual tabs, using the data table as the content. Dash allows you to provide a normal class name, and another one for when it is selected. This enables us to apply different styling based on which tab is selected.

[11]:
```
import dash_core_components as dcc

def create_tab(content, label, value):
    return dcc.Tab(
        content,
        label=label,
        value=value,
        id=f"{value}-tab",
        className="single-tab",
        selected_className="single-tab--selected",
    )

world_tab = create_tab(world_table, "World", "world")
usa_tab = create_tab(usa_table, "US States", "usa")
```

We pass the individual tabs as a list to the `Tabs` component and update our layout to show the tabs with the tables.

[12]:
```
table_tabs = dcc.Tabs(
    [world_tab, usa_tab],
    className="tabs-container",
    id="table-tabs",
    value="world"
)
layout = html.Div([title, table_tabs])

app = JupyterDash(__name__)
app.layout = layout
app.run_server(mode='inline', height=500)
```

Coronavirus Forecasting Dashboard

World | US States

Country	Deaths	Cases	Deaths per Million	Cases per Million
USA	236,514	9,709,053	718	29,470
Brazil	161,106	5,590,025	758	26,300
India	124,985	8,411,724	91	6,100
Mexico	93,772	949,197	734	7,430
United Kingdom	48,210	1,126,471	710	16,590
Italy	40,192	824,879	665	13,640
France	39,088	1,650,965	599	25,290
Spain	38,486	1,306,316	823	27,940
Iran	36,985	654,936	440	7,800
Peru	34,730	914,722	1,053	27,740
Argentina	32,766	1,217,028	725	26,930

12.6 Adding plotly figures with `dcc.Graph`

Plotly figures must be placed within the `dcc.Graph` component in order to be added to the dashboard. Before we make the graphs, let's read in all of the data, putting the group, area, and date in the index.

[13]:
```
ALL_DATA = pd.read_csv("data/all_data.csv",
                       index_col=["group", "area", "date"],
                       parse_dates=["date"]).sort_index()
ALL_DATA.head()
```

			Daily Deaths	Daily Cases	Deaths	Cases
group	**area**	**date**				
usa	**Alabama**	**2020-01-22**	0	0	0	0
		2020-01-23	0	0	0	0
		2020-01-24	0	0	0	0
		2020-01-25	0	0	0	0
		2020-01-26	0	0	0	0

We made all of the figures for our dashboard in the chapter covering plotly visualizations. The cell below defines the following functions:

- `create_figures` - creates three empty plotly figures with two plots each
- `make_cumulative_graphs` - cumulative line graphs of total deaths/cases
- `make_daily_graphs` - daily bar chart of deaths/cases
- `make_weekly_graphs` - aggregated weekly totals of deaths/cases
- `create_graphs` - runs the above four functions and returns three completed figures.

[14]:
```
from plotly.subplots import make_subplots
from plotly.colors import qualitative
COLORS = qualitative.T10[:2]
LAST_DATE = SUMMARY['date'].iloc[-1]
```

```
FIRST_PRED_DATE = LAST_DATE + pd.Timedelta('1D')

def create_figures(title, n=3):
    figs = []
    annot_props = {"x": 0.1, "xref": "paper", "yref": "paper", "xanchor": "left",
                   "showarrow": False, "font": {"size": 18},}
    for _ in range(n):
        fig = make_subplots(rows=2, cols=1, vertical_spacing=0.1)
        fig.update_layout(
            title={"text": title, "x": 0.5, "y": 0.97, "font": {"size": 20}},
            annotations=[
                {"y": 0.95, "text": "<b>Deaths</b>"},
                {"y": 0.3, "text": "<b>Cases</b>"},
            ],
            margin={"t": 40, "l": 50, "r": 10, "b": 0},
            legend={"x": 0.5, "y": -0.05, "xanchor": "center", "orientation": "h",
                    "font": {"size": 15}})
        fig.update_traces(showlegend=False, row=2, col=1)
        fig.update_traces(hovertemplate="%{x} - %{y:,}")
        fig.update_annotations(annot_props)
        figs.append(fig)
    return figs

def make_cumulative_graphs(fig, df_dict, kinds):
    for row, kind in enumerate(kinds, start=1):
        for i, (name, df) in enumerate(df_dict.items()):
            fig.add_scatter(x=df.index, y=df[kind], mode="lines+markers",
                            showlegend=row==1, line={"color": COLORS[i]},
                            name=name, row=row, col=1)

def make_daily_graphs(fig, df_dict, kinds):
    for row, kind in enumerate(kinds, start=1):
        for i, (name, df) in enumerate(df_dict.items()):
            fig.add_bar(x=df.index, y=df[kind], marker={"color": COLORS[i]},
                        showlegend=row==1, name=name, row=row, col=1)

def make_weekly_graphs(fig, df_dict, kinds):
    offset = "W-" + LAST_DATE.strftime("%a").upper()
    df_dict = {name: df.resample(offset, kind="timestamp", closed="right")[kinds].
 ↪sum()
                for name, df in df_dict.items()}

    for row, kind in enumerate(kinds, start=1):
        for i, (name, df) in enumerate(df_dict.items()):
            fig.add_scatter(x=df.index, y=df[kind], mode="lines+markers",
                            showlegend=row==1, line={"color": COLORS[i]},
                            name=name, row=row, col=1)
```

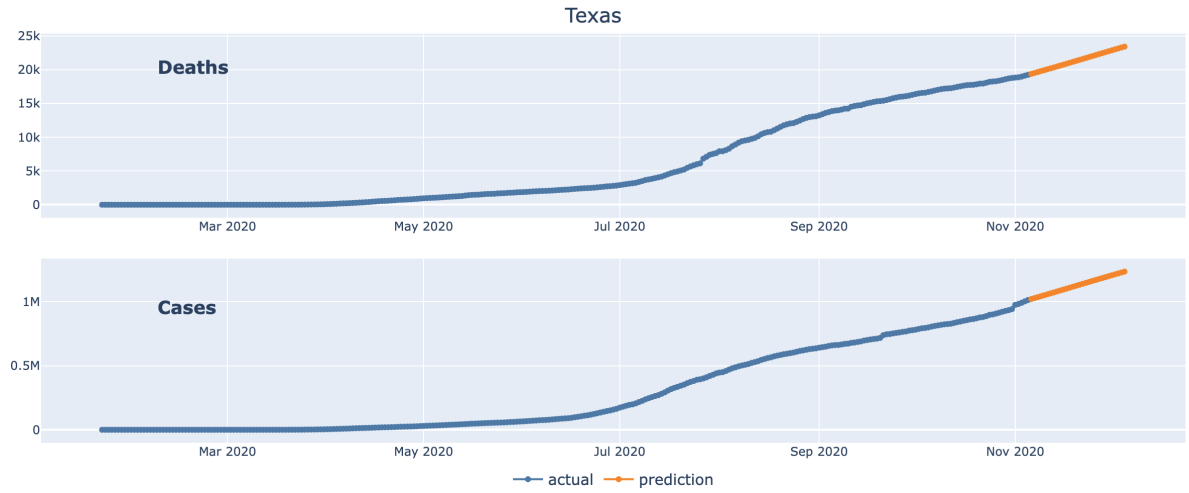
```
def create_graphs(group, area):
    df = ALL_DATA.loc[(group, area)]
    df_dict = {"actual": df.loc[:LAST_DATE], "prediction": df.loc[FIRST_PRED_DATE:]}
    kinds = ["Deaths", "Cases"]
    new_kinds = ["Daily Deaths", "Daily Cases"]
    figs = create_figures(area)
    make_cumulative_graphs(figs[0], df_dict, kinds)
    make_daily_graphs(figs[1], df_dict, new_kinds)
    make_weekly_graphs(figs[2], df_dict, new_kinds)
    return figs
```

Let's choose one area to make the three graphs.

[15]:
```
figs = create_graphs('usa', 'Texas')
```

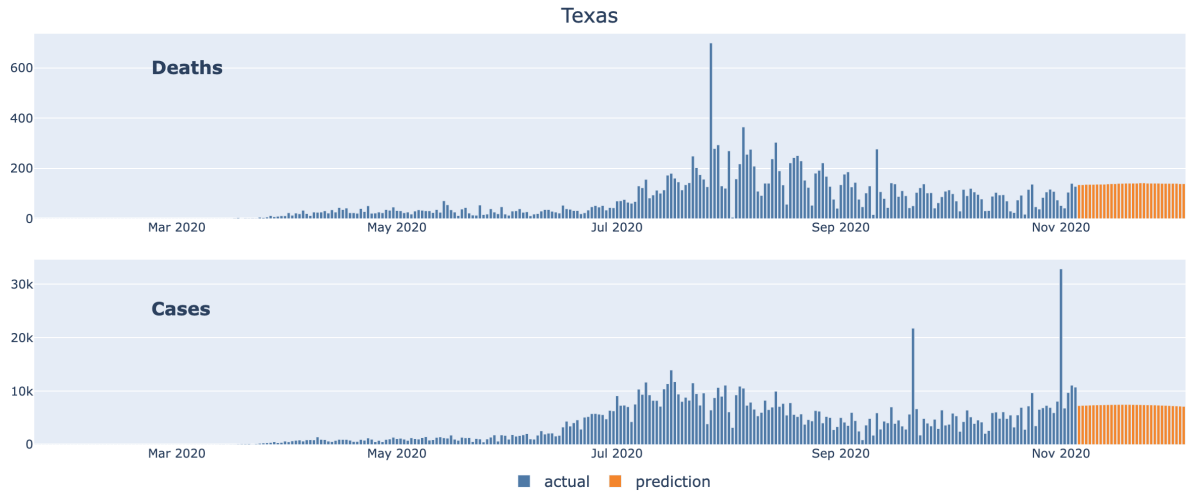
Here are the cumulative line graphs for cases in deaths as a plotly figure. This is independent of dash at this point.

[16]:
```
figs[0]
```



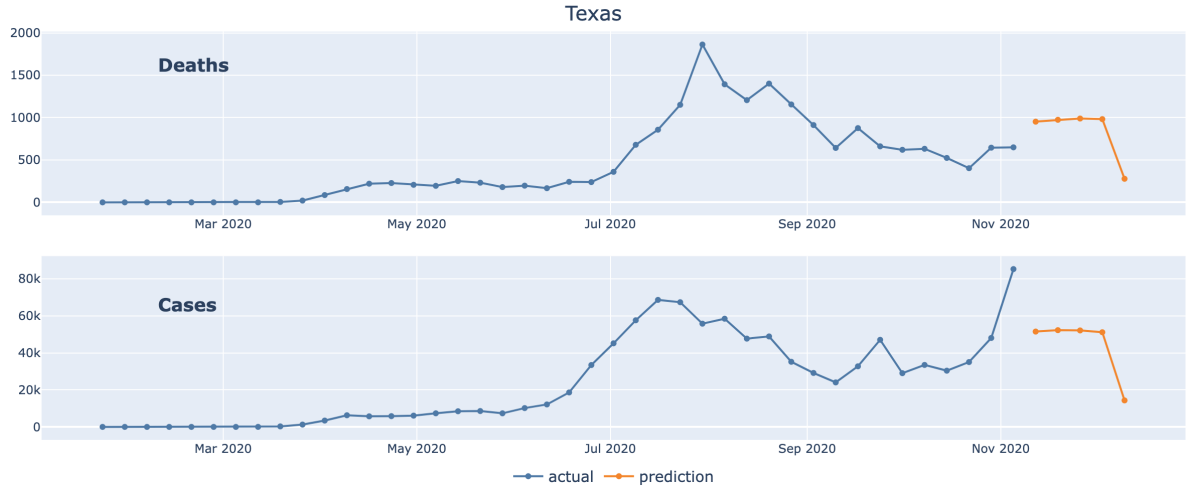
Daily bar graph:

[17]:
```
figs[1]
```



Weekly aggregate line graph:

[18]:
```
figs[2]
```



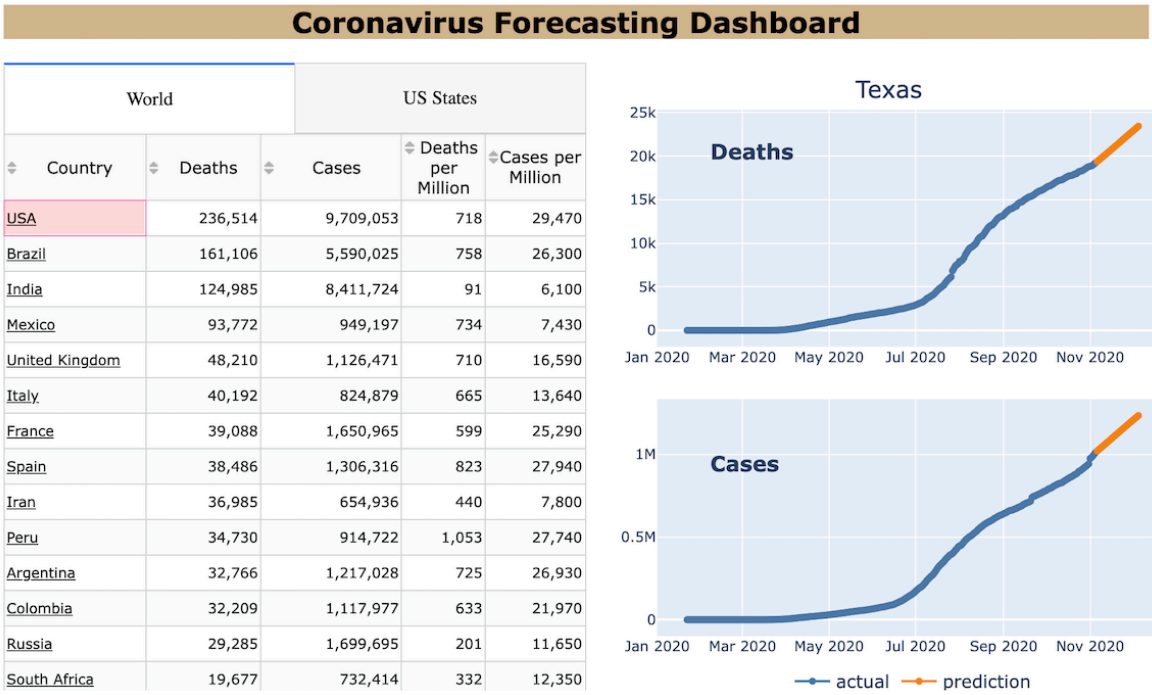
We set the `figure` parameter of `dcc.Graph` to be one of the figures and then place that object in our layout. This new object would appear under the table by default, since it's a block-level element. To display it to the side of the table, we place the tabs and graph within a div and set its `display` property to `grid`. A 2 x 2 grid is created using the following `gridTemplateAreas`.

```
"tables graphs"
"tables maps"
```

Each column is set to be 50% of the viewport width, `vw`. The graph is given the `gridArea` of "graphs", while the style of `table_tables` is updated so that its `gridArea` is "tables". Lastly, `columnGap` is used to separate the left and right columns by 10 pixels.

[19]:
```
cumulative_graph = dcc.Graph(figure=figs[0], style={'gridArea': "graphs"})
table_tabs.style = {'gridArea': "tables"}
container = html.Div([table_tabs, cumulative_graph],
                     style={'display': 'grid',
                            'gridTemplateAreas': '"tables graphs" "tables maps"',
                            'gridTemplateColumns': "50vw 50vw",
```

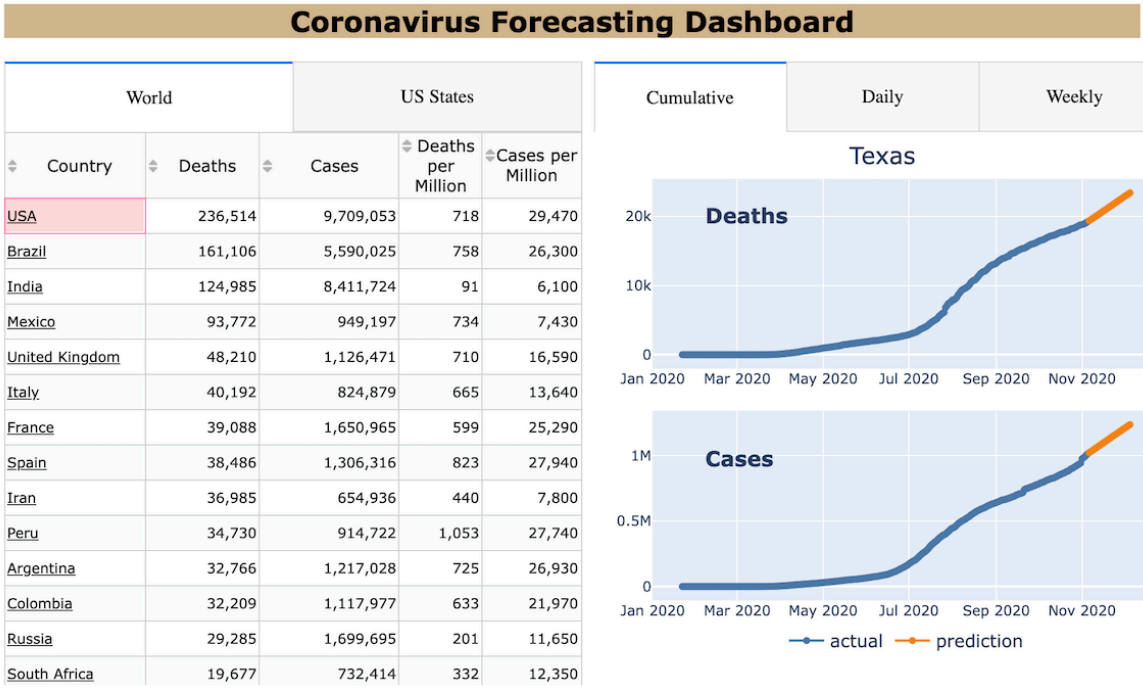
```
                              'columnGap': '10px'})
layout = html.Div([title, container])

app = JupyterDash(__name__)
app.layout = layout
app.run_server(mode='inline', height=600)
```



In the actual dashboard, we have tabs to cycle through each graph. We'll duplicate the procedure from above, adding each graph to its own tab.

[20]:
```
cumulative_graph = dcc.Graph(figure=figs[0], id="cumulative-graph")
daily_graph = dcc.Graph(figure=figs[1], id="daily-graph")
weekly_graph = dcc.Graph(figure=figs[2], id="weekly-graph")

cumulative_tab = create_tab(cumulative_graph, "Cumulative", "cumulative")
daily_tab = create_tab(daily_graph, "Daily", "daily")
weekly_tab = create_tab(weekly_graph, "Weekly", "weekly")

graph_tabs = dcc.Tabs(
    [cumulative_tab, daily_tab, weekly_tab],
    className="tabs-container",
    id="graph-tabs",
    value="cumulative",
    style={'gridArea': 'graphs', 'margin': '0px'}
)

container = html.Div([table_tabs, graph_tabs],
                     style={'display': 'grid',
                            'gridTemplateAreas': '"tables graphs" "tables maps"',
```

```
                              'gridTemplateColumns': "50vw 50vw",
                              'columnGap': '10px'})
layout = html.Div([title, container])

app = JupyterDash(__name__)
app.layout = layout
app.run_server(mode='inline', height=600)
```



12.7 Adding the maps

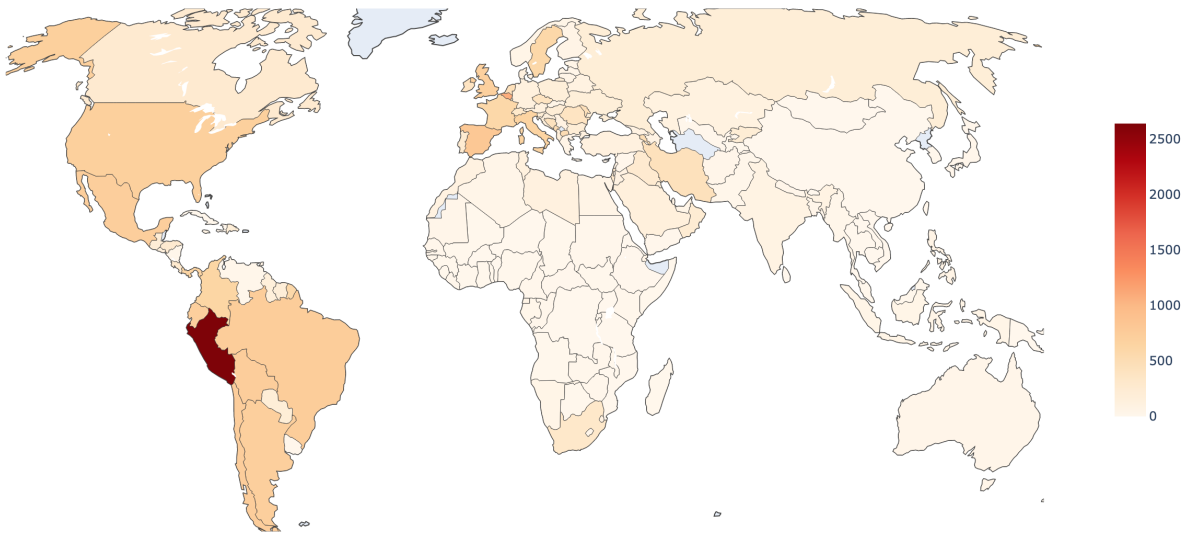
The last main component of the dashboard are the world and USA maps. Here, we copy the code from a previous chapter that creates the maps with custom hover text.

[21]:
```
import plotly.graph_objects as go

def hover_text(x):
    name = x["area"]
    deaths = x["Deaths"]
    cases = x["Cases"]
    deathsm = x["Deaths per Million"]
    casesm = x["Cases per Million"]
    return (
        f"<b>{name}</b><br>"
        f"Deaths - {deaths:,.0f}<br>"
        f"Cases - {cases:,.0f}<br>"
        f"Deaths per Million - {deathsm:,.0f}<br>"
        f"Cases per Million - {casesm:,.0f}<br>"
    )
```

```
def create_map(group, radio_value):
    df = SUMMARY.loc[group].query("population > 0.5")
    lm = None if group == "world" else "USA-states"
    proj = "robinson" if group == "world" else "albers usa"

    fig = go.Figure()
    fig.add_choropleth(
        locations=df["code"],
        z=df[radio_value],
        zmin=0,
        locationmode=lm,
        colorscale="orrd",
        marker_line_width=0.5,
        text=df.apply(hover_text, axis=1),
        hoverinfo="text",
        colorbar=dict(len=0.6, x=1, y=0.5),
    )
    fig.update_layout(
        geo={
            "lataxis": {"range": [-50, 68]},
            "lonaxis": {"range": [-130, 150]},
            "projection": {"type": proj},
            "showframe": False,
        },
        margin={"t": 0, "l": 10, "r": 10, "b": 0},
    )
    return fig
```

Now we create a single map and color it by deaths per million.

[22]:
```
fig_map = create_map('world', 'Deaths per Million')
fig_map
```



12.8 Adding radio buttons above the map

Our dashboard allows the user to choose the coloring of the map with four radio buttons. The dash core components library does have radio buttons, but does not supply an easy way to change the style whenever the radio button is checked.

Dash Bootstrap Components

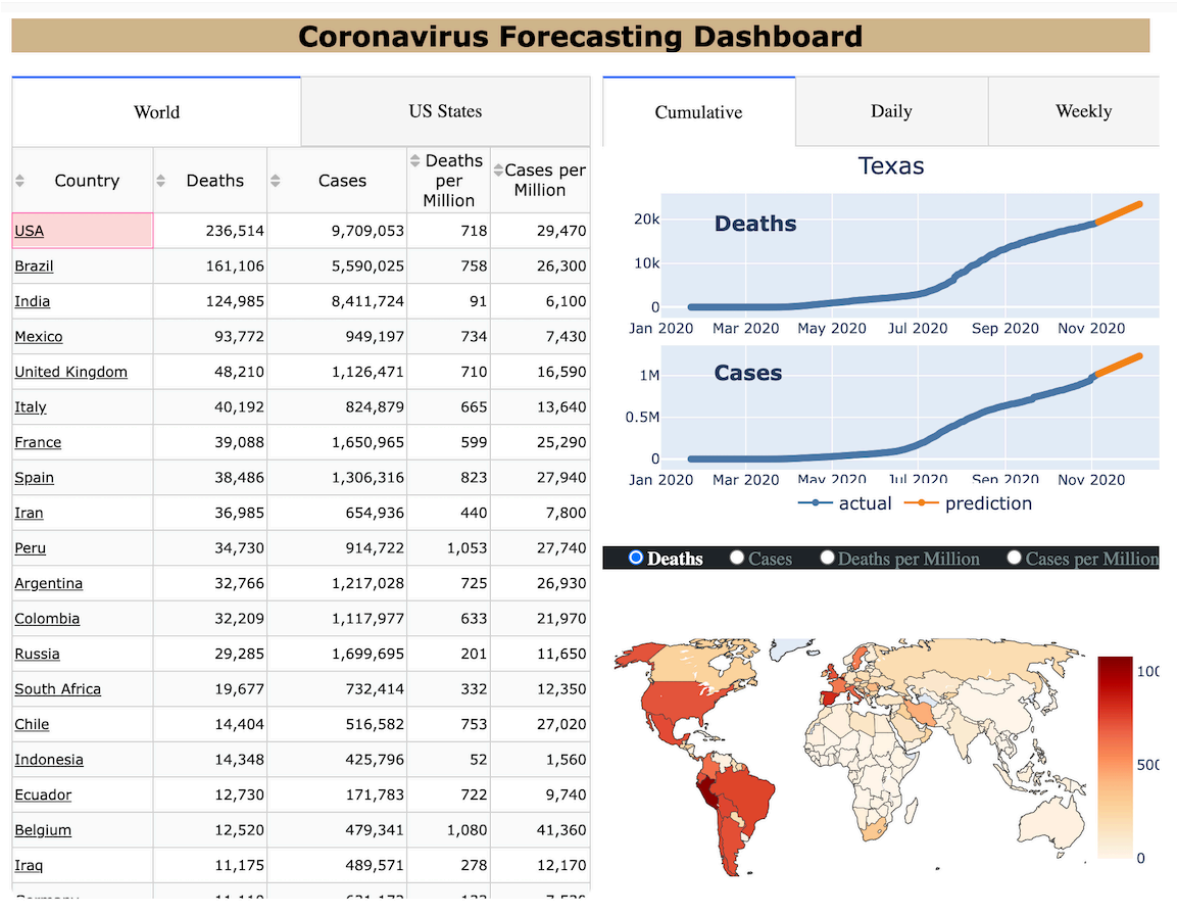
The third-party library Dash Bootstrap Components has many components for dash that are built using Bootstrap, a very popular CSS and JavaScript library containing nicely styled HTML components. Here, we create the radio buttons, which require a list of dictionaries setting the visible `label` and internal `value` to strings for all options. The `labelCheckedStyle` allows us to apply a separate style to the checked label. We create a dashboard with just the radio buttons to view them before adding them above the map.

[23]:
```
import dash_bootstrap_components as dbc

radio_items = dbc.RadioItems(
    options=[
        {"label": "Deaths", "value": "Deaths"},
        {"label": "Cases", "value": "Cases"},
        {"label": "Deaths per Million", "value": "Deaths per Million"},
        {"label": "Cases per Million", "value": "Cases per Million"},
    ],
    value="Deaths",
    id="map-radio-items",
    style={'display': 'flex',
           'justifyContent': 'space-evenly',
           'backgroundColor': '#212529',
           'color': '#798d8f'},
    labelCheckedStyle={'fontWeight': 800, 'color': 'white'}
)


app = JupyterDash(__name__)
app.layout = radio_items
app.run_server(mode='inline', height=50)
```

The radio buttons and map are wrapped in a div and added to the bottom right corner. The buttons will have no effect until we add the interactivity. Note that we had to set the style of each graph component in order to set the height. This is a bug in dash as the graph does not expand or contract to fit the size of its container.

[24]:
```
cumulative_graph.style = {'height': '40vh'}
daily_graph.style = {'height': '40vh'}
weekly_graph.style = {'height': '40vh'}

map_graph = dcc.Graph(figure=fig_map, id="map-graph", style={'height': '40vh'})
```

```
map_div = html.Div([radio_items, map_graph], style={'gridArea': 'maps'})
container = html.Div([table_tabs, graph_tabs, map_div],
                     style={'display': 'grid',
                            'gridTemplateAreas': '"tables graphs" "tables maps"',
                            'gridTemplateColumns': "50vw 50vw",
                            'gridTemplateRows': "50vh 40vh",
                            'columnGap': '10px',
                            })
layout = html.Div([title, container])

app = JupyterDash(__name__)
app.layout = layout
app.run_server(mode='inline', height=800)
```



Country	Deaths	Cases	Deaths per Million	Cases per Million
USA	236,514	9,709,053	718	29,470
Brazil	161,106	5,590,025	758	26,300
India	124,985	8,411,724	91	6,100
Mexico	93,772	949,197	734	7,430
United Kingdom	48,210	1,126,471	710	16,590
Italy	40,192	824,879	665	13,640
France	39,088	1,650,965	599	25,290
Spain	38,486	1,306,316	823	27,940
Iran	36,985	654,936	440	7,800
Peru	34,730	914,722	1,053	27,740
Argentina	32,766	1,217,028	725	26,930
Colombia	32,209	1,117,977	633	21,970
Russia	29,285	1,699,695	201	11,650
South Africa	19,677	732,414	332	12,350
Chile	14,404	516,582	753	27,020
Indonesia	14,348	425,796	52	1,560
Ecuador	12,730	171,783	722	9,740
Belgium	12,520	479,341	1,080	41,360
Iraq	11,175	489,571	278	12,170

12.9 Interactivity using callbacks

The **layout** of our dashboard is now complete. We are ready to move on to the second half of our application, the **interactivity**. Code for the interactivity MUST appear after `app.layout` has been set or you'll get an error. The interactivity is user-defined functions named callbacks that get triggered by some event on the dashboard, usually a click.

Writing a callback

A summary of the callback function is provided below:

- `app.callback` method decorates the function and takes three main arguments:
  - Outputs - A list of layout components to be changed
  - Inputs - A list of layout components that trigger the function
  - States - A list of non-triggered layout components to pass to the function
- Function defined with one parameter for every input
- Function must return the number of outputs

`Output`, `Input`, and `State`

You will use the objects `Output`, `Input`, and `State` to describe the arguments to your callback function and the return values. All three objects are found in the `dash.dependencies` module and all require two arguments, the **id** of the component, and the targeted **property**. Here are some examples:

- `Output("cumulative-graph", "figure")` - informs dash that the component with id of "cumulative-graph" should have its "figure" property replaced with a returned value.
- `Input('world-table', 'active_cell')` - informs dash to trigger the function whenever the `active_cell` property of the component with id "world-table" is changed.
- `State('world-table', 'derived_virtual_data')` - tells dash to pass the value of the property `derived_virtual_data` of the component with id "world-table" to the function. `State` does not trigger the function. It's just extra data to pass to the function whenever it does get triggered.

Take a look at the interactivity below. We have a single function that is decorated by `app.callback`. The function name can be anything you desire. This function changes three components (the "figure" property of each graph in the upper right corner). From the inputs section, we see that it gets triggered when the active cell changes in the world table. Whenever this function gets triggered, it also gets passed the value of `derived_virtual_data` from the world table.

`data` vs `derived_virtual_data`

You might be wondering what the difference is between the `data` and `derived_virtual_data` properties from a dash data table. They are both lists containing a dictionary for every row of data. But, the `data` property is **static** and never changes. If the table gets sorted by the user, the list from `data` remains the same. The `derived_virtual_data` is **dynamic** and always matches the data that the user sees. This is why we use it and not the value for the `data` property.

All callbacks called when dashboard starts

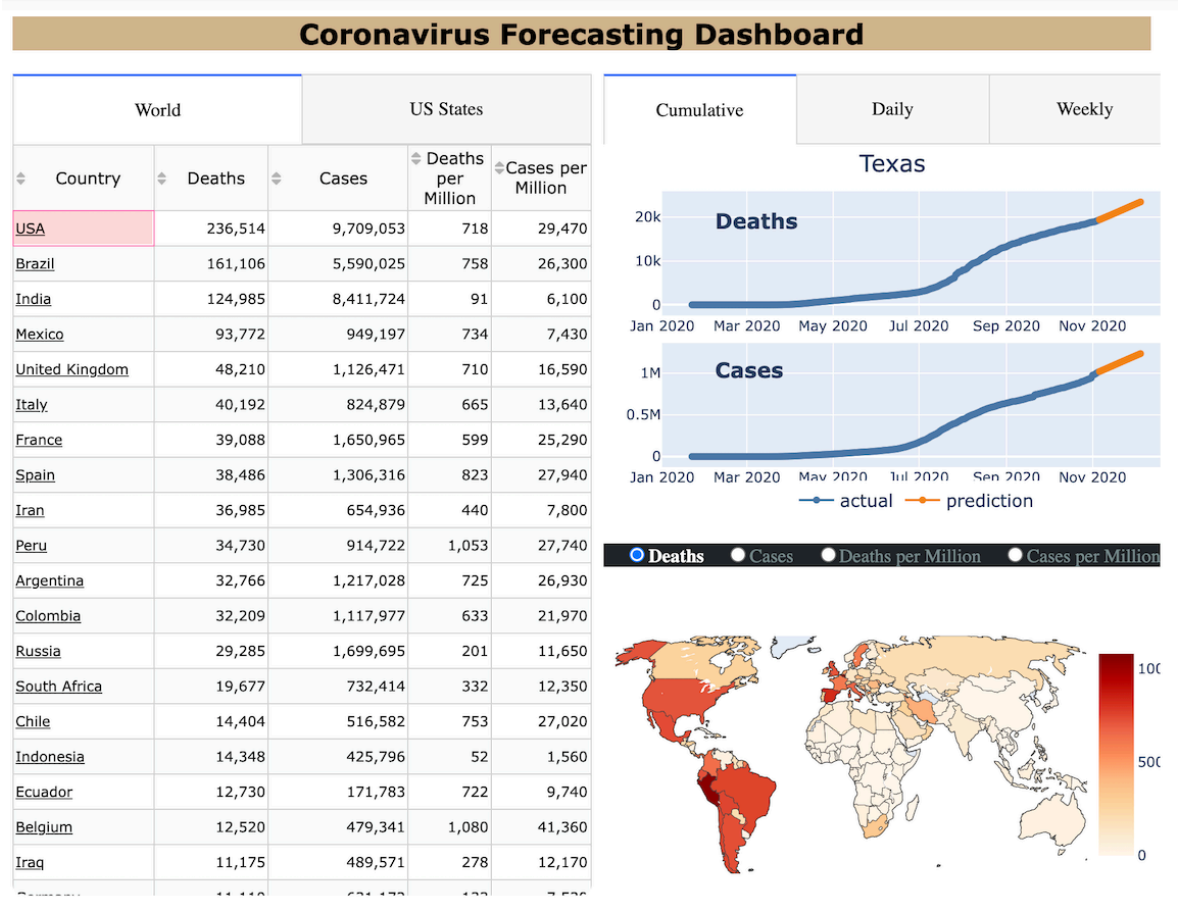
By default, all callbacks are called at the start of the application. This may seem like undesired behavior, but it can actually save you from repeating code, as you don't have to set the components that will be changed during the callback.

Our function below will be called by dash whenever the active cell changes, and will be passed two arguments. The parameter names for the function are completely arbitrary, and can be anything you choose. The order of the arguments passed comes directly from the order of the inputs and states in the decorator.

Our function receives the default values for the `active_cell` and `derived_virtual_data`, in that order. The text value of `active_cell` is (unfortunately) not provided by dash, just a dictionary of row and column integers. We use this info to find the actual value from the list of dictionaries

from `derived_virtual_data`. Once we get the country name, we call the `create_graphs` function, which returns three figures, replacing the "figure" property of each of the `dcc.Graph` objects. To help understand what is passed to the function, a print statement is added to show some variable values.

[25]:
```
from dash.dependencies import Output, Input, State

app = JupyterDash(__name__)
app.layout = layout

@app.callback(
    [
        Output("cumulative-graph", "figure"),
        Output("daily-graph", "figure"),
        Output("weekly-graph", "figure"),
    ],
    [
        Input('world-table', 'active_cell')
    ],
    [
        State('world-table', 'derived_virtual_data')
    ]
)
def change_area_graphs(world_cell, world_data):
    """
    Change the all three graphs in the upper right hand corner of the app

    Parameters
    ----------
    world_cell : dict with keys `row` and `cell` mapped to integers of cell location

    world_data : list of dicts of one country per row.
                    Has keys Country, Deaths, Cases,
                    Deaths per Million, Cases per Million

    Returns
    -------
    List of three plotly figures, one for each of the `Output`
    """
    row_number = world_cell["row"]
    row_data = world_data[row_number]
    country = row_data['Country']
    print("active_cell", world_cell,
          "\nrow_number", row_number,
          "\nrow_data", row_data,
          "\ncountry", country)
    return create_graphs('world', country)

app.run_server(mode='inline', height=800)
```



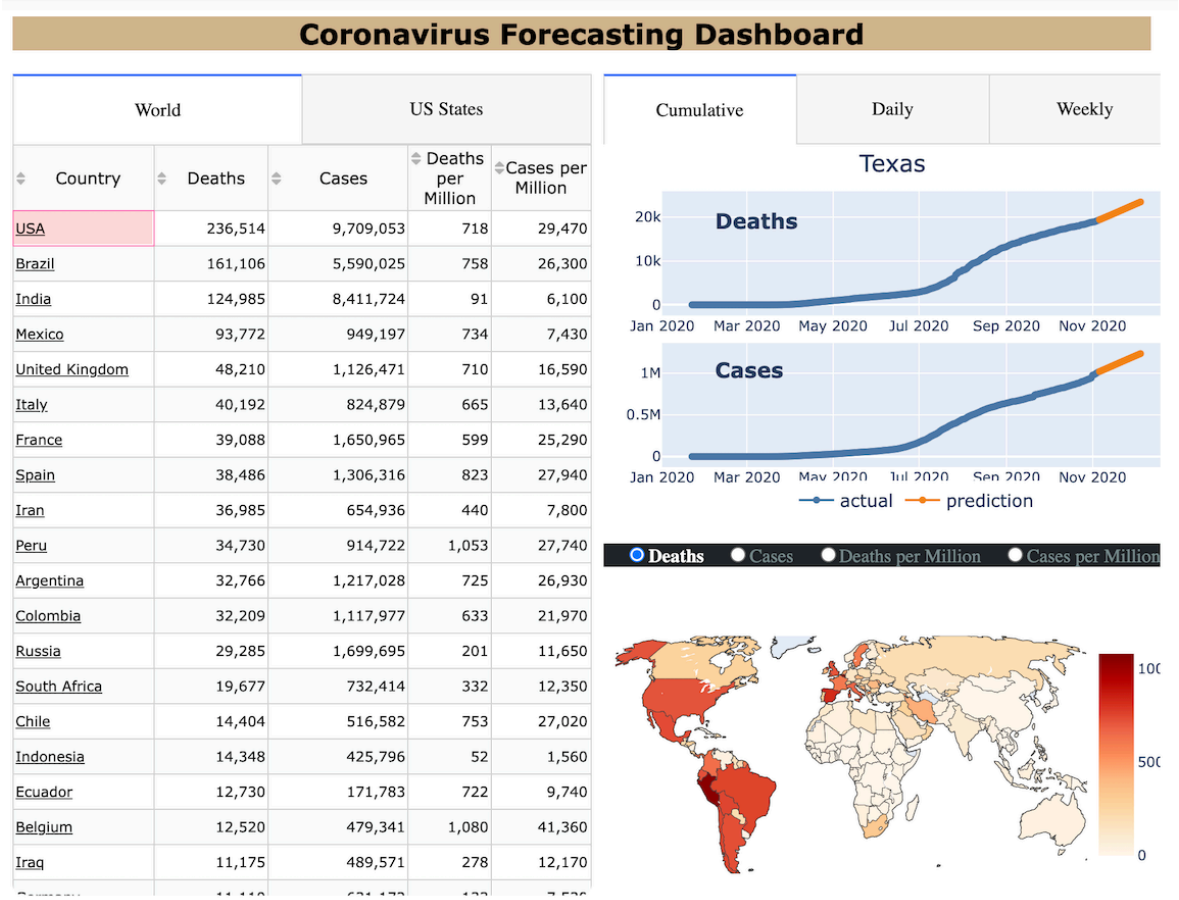
Single Callback for updating both World and USA data

A component may only appear as an output in exactly **one** callback. This means we cannot write separate callbacks for the world and USA tables, as they update the same three graphs in the upper right corner. To work around this limitation, we redefine our callback to have a total of three input triggers - both data tables and the tab above them. If the active cell from either of the tables change or the tab switches, the function will be triggered. It gets passed those three inputs plus the underlying list of dictionaries (the `derived_virtual_data`) from both tables.

The body of the function is updated to perform some logic to determine whether the world or USA tab is active. Since any cell can be the active cell, it only updates the graph when the first column (index equal to 0) is active. It's also possible that there is no active cell, making its value `None`. This happens whenever a sort is performed. The statement `if cell and cell["column"] == 0` ensures that there is an active cell and it's in the first column. If this check fails, we must raise the `PreventUpdate` exception (that you must import from the `dash.exceptions` module) as dash expects three figures to be returned. This is a useful exception to know about, as there will be times a user triggers a callback, but the desired behavior is to not change anything.

[26]:
```
from dash.exceptions import PreventUpdate

app = JupyterDash(__name__)
app.layout = layout
```

```
@app.callback(
    [
        Output("cumulative-graph", "figure"),
        Output("daily-graph", "figure"),
        Output("weekly-graph", "figure"),
    ],
    [
        Input("world-table", "active_cell"),
        Input("usa-table", "active_cell"),
        Input("table-tabs", "value"),
    ],
    [
        State("world-table", "derived_virtual_data"),
        State("usa-table", "derived_virtual_data"),
    ],
)
def change_area_graphs(world_cell, usa_cell, group, world_data, usa_data):
    area, cell, data = "Country", world_cell, world_data
    if group == "usa":
        area, cell, data = "State", usa_cell, usa_data
    if cell and cell["column"] == 0:
        country_state = data[cell["row"]][area]
        return create_graphs(group, country_state)
    else:
        raise PreventUpdate

app.run_server(mode='inline', height=800)
```

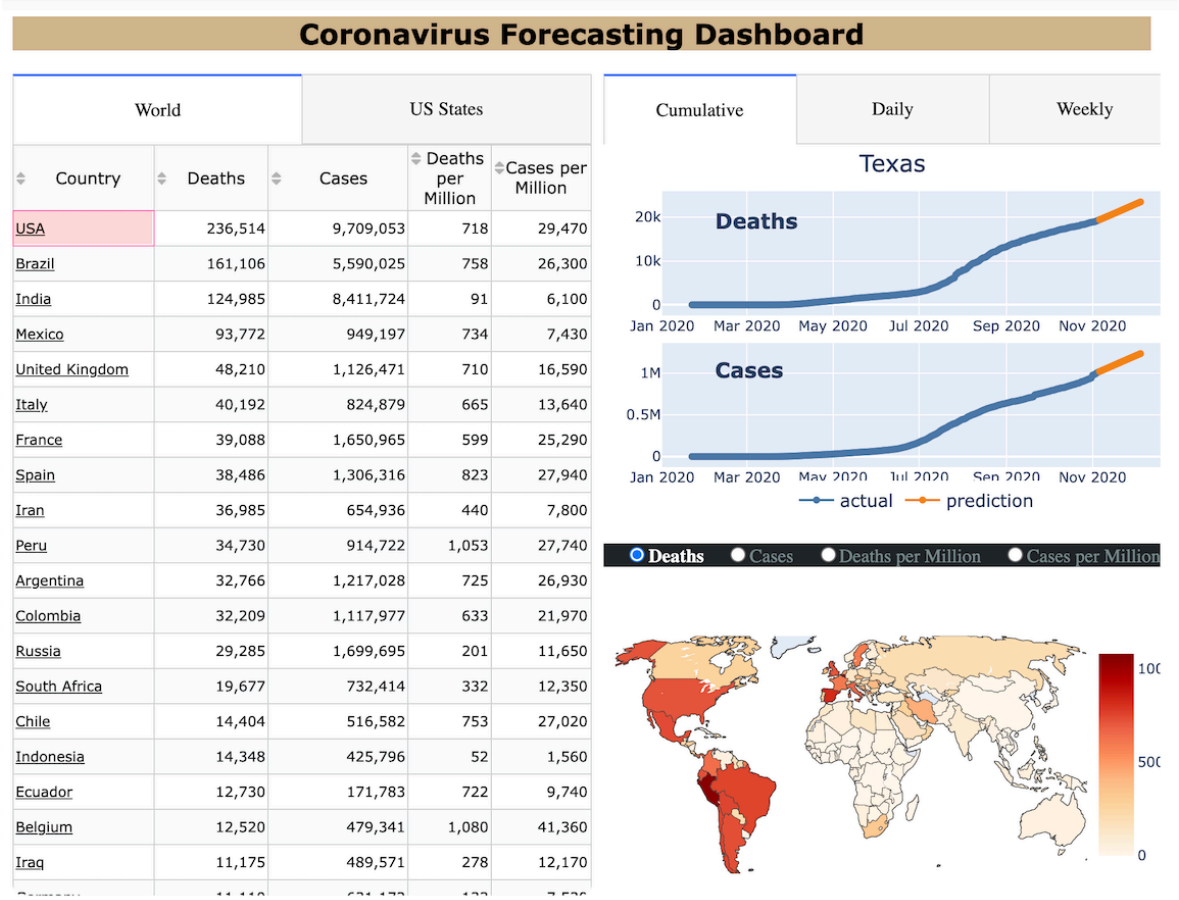


12.10 Callback to change the map

We need to add one more callback to change the map and/or its coloring. This function gets triggered when either the tabs above the table change or one of the radio items is clicked. We return a single figure using our `create_map` function defined above.

[27]:
```
app = JupyterDash(__name__)
app.layout = layout

@app.callback(
    [
        Output("cumulative-graph", "figure"),
        Output("daily-graph", "figure"),
        Output("weekly-graph", "figure"),
    ],
    [
        Input("world-table", "active_cell"),
        Input("usa-table", "active_cell"),
        Input("table-tabs", "value"),
    ],
    [
        State("world-table", "derived_virtual_data"),
```

```
        State("usa-table", "derived_virtual_data"),
    ],
)
def change_area_graphs(world_cell, usa_cell, group, world_data, usa_data):
    area, cell, data = "Country", world_cell, world_data
    if group == "usa":
        area, cell, data = "State", usa_cell, usa_data
    if cell and cell["column"] == 0:
        country_state = data[cell["row"]][area]
        return create_graphs(group, country_state)
    else:
        raise PreventUpdate

@app.callback(
    Output("map-graph", "figure"),
    [
        Input("table-tabs", "value"),
        Input("map-radio-items", "value")
    ],
)
def change_map(group, radio_value):
    return create_map(group, radio_value)

app.run_server(mode='inline', height=800)
```



Differences between this tutorial and `dashboard.py`

Open the `dashboard.py` file and review the code. It is largely the same as it was presented in this tutorial, however, there are a few differences:

- The dash application is instantiated with `Dash` from the library `dash` and not `JupyterDash`
- All of the CSS (except for the data tables) has moved to assets/style.css
- The CSS for Bootstrap (and any other external CSS) must be linked using the `external_stylesheets` parameter when instantiating `Dash`. It is set to a list of URLs containing CSS.
- A new column in our CSS grid is added on the left. Four bootstrap "cards" from `dash_bootstrap_components` are placed in this column.
- A navigation bar is added to the top of the page using `dash_bootstrap_components`
- CSS is added to target screens with a width less than 1000px using a media query.
  - Take a look at the bottom of assets/style.css. You'll see the following selector
    * `@media only screen and (max-width: 1000px)`

12.11 Dash Summary

- Dash vs Plotly
  - Dash builds the dashboard application with HTML components and provides interactivity with callbacks
  - Dash does not do visualization

  - Plotly creates all visualizations
  - Plotly visualizations must be placed in a `dcc.Graph` component
- Two parts of a dash application
  - **Layout**
    - Dash HTML Components - nearly all regular HTML components
    - Dash Core Components - interactive widgets
    - Dash Data Tables - two dimensional tables
    - Dash Bootstrap Components - third-party library with more widgets styled with Bootstrap
    - Must set `app.layout` to a dash component to finalize layout
  - **Interactivity**
    - Functions triggered by user events
    - Functions receive component values as inputs
    - Functions return new component values as outputs
    - Interactivity must come after `app.layout` is set
- Dash HTML Components
  - Separate library - `dash_html_components` aliased as `html`
  - Uses same tag names as regular HTML, but capitalized
  - First parameter is content
  - Provide id and class as `id` and `className` parameters
  - Can provide CSS with `style` parameter, but better to do so in `assets/style.css`
- Dash Core Components
  - Separate library - `dash_core_components` aliased as `dcc`
  - Many widgets that bundle together HTML/CSS and JavaScript
  - Dropdown menus, radio buttons, tabs, and more
  - Use `dcc.Graph` to add plotly figure
- Dash Data Tables
  - Separate Library - `dash_table`
  - Provide data as a list of dictionaries. Use `df.to_dict('records')`
  - Set style for whole table, each cell, just the headers, or just the data with `style_*` parameters
- Dash Bootstrap Components
  - Third-party library - `dash_bootstrap_components` - aliased as dbc
  - Must set `external_stylesheets` in `Dash` constructor to a list of URLs of the specific flavor of Bootstrap you want.
  - Many extra components not provided by `dash_core_components`
- Callback functions
  - Must be defined after `app.layout` is set
  - Function and parameter names are arbitrary
  - Decorate functions with `app.callback`
  - Pass `app.callback` a list of the outputs, inputs, and states
  - Use `Input`, `Output`, and `State` from `dash.dependencies` to target the component and its property
    - Example: `Input("component-id", "component-property")`
  - Function must return a value for each of the outputs
  - Raise the `dash.exceptions.PreventUpdate` exception if you don't want the function to update any component
  - Every callback is called when the app first launches
  - Each component may appear as an output in exactly one callback
    - May need to combine many triggers into one function to work around this limitation

- Each component may appear as an input in ANY number of callbacks

Chapter 13

Deployment

In this chapter, we'll learn how to deploy our application on a remote server so that it is accessible on the web by anyone from anywhere. Two deployment options will be shown. First, will be an easy (and free) option with PythonAnywhere.com. In the second option we'll set up an Ubuntu server on our own. This option is more complex, targeted for those that want full control, and costs a few dollars per month.

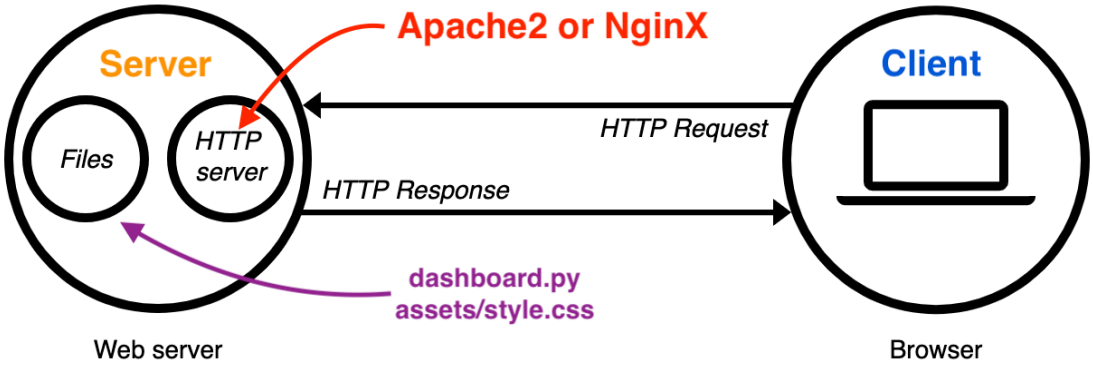
13.1 What is a web server?

As in the chapter on web development, the MDN documentation on web servers is a great introduction, and an important read if you have no experience with them.

We've run our application from our local machine by executing the command `python dashboard.py`, but this only allows us to access the application from our localhost. Nobody else can view our dashboard besides us. In order to allow others to access our application, we'll need web server software such as Apache2 or NginX (pronounced "Engine-X") to **serve** our website to **clients** that **request** it using the **HTTP** protocol. Some definitions of common terms follows:

- Serve (verb) - the transferring of files (HTML, CSS, JavaScript, images, PDFs, etc…) from server to client
- Server - the machine running the web server software and hosting the files to be served to the client
- Client - user (a human or another machine) that makes requests, typically with a web browser
- Request - a specific message that the client sends to the server to take an action. The message is precisely defined by the HTTP protocol. The most common requests are `GET` and `POST`.
- HTTP - HyperText Transfer Protocol - A specific protocol that defines exactly how to make each type of request
- Response - a message from the server in response to a client request - Comes with a status code (i.e. 200, 404, 500)

The image below (from MDN with added annotations) depicts this process of a client (you on your computer using a web browser) making requests for content from a server (a machine probably in a warehouse hosting files for https://coronavirus.dunderdata.com). For example, when a user clicks on one of the countries in the data table, a `GET` request is made from the browser to the server. The `dashboard.py` file on the server runs the code within the callback and makes a response with the new HTML and CSS. Your browser will then interpret this response and render the updates on the page.



Where are these servers and how do I get one?

Any computer connected to the internet can work as a web server. Even your personal machine has the ability to become a web server. Most people don't use their own machines as web servers as they would need to keep it running at all times and have enough resources to serve content to all the clients. Some individuals will buy machines, set them up in their homes, and dedicate them to hosting their web sites.

Most often, the better choice is to rent a computer from a company to use as a web server. The company will be responsible for managing all of the hardware and keeping the computer up and running. There are a huge number of companies that rent web servers, with the biggest names being Amazon (with AWS), Microsoft (with Azure), and Google (with Google Cloud). While these companies offer good servers, the overwhelming number of services makes it difficult (in my opinion) to navigate their platforms. They also cost more than a company like Vultr, which we will use.

How do I control this web server?

To use the computer acting as the web server, you'll have to log into it. Most companies provide you access to their servers from a web browser. You just log into your online account and find the **console**, which opens up a command line interface running on the server's machine. A few companies, such as Python Anywhere, have a web interface that allows you to control many of the server functions, such as uploading files, using point and click.

You can also log in from your local computer using a **Secure Shell** or **ssh**, which establishes a secure connection between your machine and the server. Using ssh will drop you into the command line on the server.

13.2 Python Anywhere

Python Anywhere is a great service for those looking to launch their own web application with no prior experience. All of the server configuration (which can be a huge pain) is done for us. We'll just need to upload our files, install dash, update the data, and the dashboard will be up and running on a public web address. The following outline summarizes the steps we'll take to create our app.

- Make Python Anywhere account
- Create new web app
- Select Flask with Python 3.8

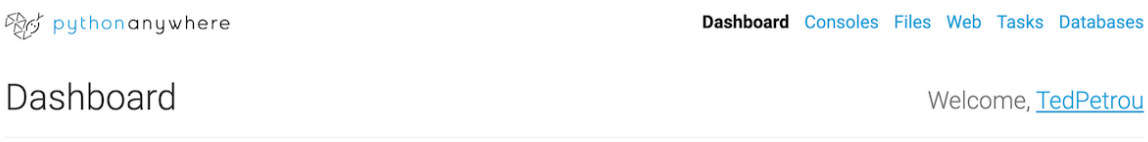
- Add all files
- Modify wsgi.py file
- Install libraries
- Run `update.py`
- Verify dashboard
- Create daily task

Make Python Anywhere Account

Navigate to PythonAnywhere, create a free account, and verify your email. The free tier provides you with the ability to create a single web application hosted from username.pythonanywhere.com. Your server is **shared** and not **dedicated** just to your app. You will share resources with other users and have a small amount of disk space (512 MB) partitioned for you. You will also be limited to just 100 seconds of CPU time per day. After this time is up, your tasks will still run, but won't be prioritized, and take much longer to complete. The CPU seconds don't apply to the actual running of the web app, but to commands executed on the command line (such as `pip3.8 install ...`).

Create new web app

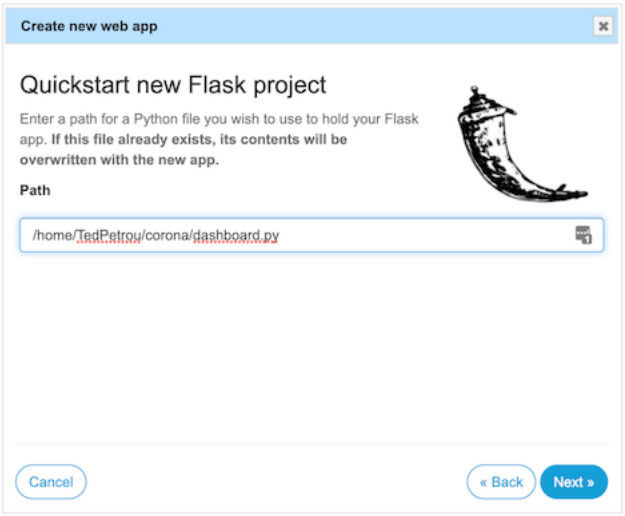
After you create an account, you'll be in the Python Anywhere dashboard. On the top right of the page is a menu. Click on **Web** and then **Add a new web app**.



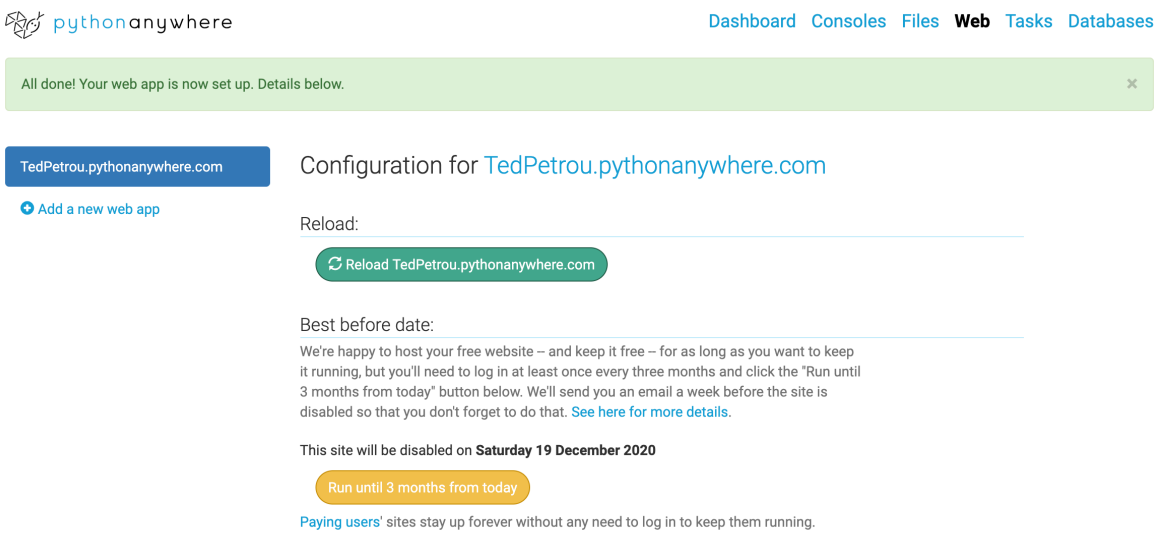
You'll be prompted with a couple of questions. The first provides you the domain name of your app which will be http://username.pythonanwhere.com. You can find mine at http://tedpetrou.pythonanywhere.com.

Select Flask with Python 3.8

You'll then be prompted to select a web framework. Dash is not a choice, so choose **Flask**, which is what it is built upon. Select **Python 3.8** at the next prompt. You'll then be asked to choose a file location for the app. Change it to `/home/username/corona/dashboard.py` substituting in your username.



Navigate to your URL to view the default web app that's been created for you.



Upload all files

We need to upload our local project files to the server and will use the web interface to do so. Click on the **Files** menu. You should get a display showing your home directory, with directories on the left side and files on the right. The location of the current directory is displayed on top and will be `/home/username`. Click on the **corona** directory and then the upload button to add the following four files:

- dashboard.py
- models.py
- prepare.py
- update.py

The dashboard.py file will overwrite the simple default that was already created. We won't be using our `wsgi.py` file since Python Anywhere has created one for us. We also won't be using a virtual

environment, as our disk space is limited, and many of the libraries we need are already installed, so we don't need to upload `requirements.txt`.

Create an **assets** directory and upload the `dark_logo.png` and `style.css` files to it. Navigate back up one level to the corona directory and create a **data** directory and upload the CSV files. Do NOT create any other directories. Your directory structure should look like this when complete:

- **corona**
  - dashboard.py
  - models.py
  - prepare.py
  - update.py
  - **assets**
    - dark_logo.png
    - style.css
  - **data**
    - all_data.csv
    - population.csv
    - summary.csv

Modify wsgi file

Go back to the **Web** tab and look down to find the **Code** section. The **Source code** directory should be correct and be set to `/home/username/corona`. However, we must change the **Working directory** to `/home/username/corona` as well. Make the change now.

Click on the **WSGI configuration file** to open up an editor with its contents. Add the following line to the bottom of the file, then click the **save** button.

```
application = application.server
```

```
1  # This file contains the WSGI configuration required to serv
2  # web application at http://<your-username>.pythonanywhere.c
3  # It works by setting the variable 'application' to a WSGI h
4  # description.
5  #
6  # The below has been auto-generated for your Flask project
7
8  import sys
9
10 # add your project directory to the sys.path
11 project_home = '/home/TedPetrou/corona'
12 if project_home not in sys.path:
13     sys.path = [project_home] + sys.path
14
15 # import flask app but need to call it "application" for WSG
16 from dashboard import app as application  # noqa
17 application = application.server
```

Dash holds the underlying flask app in its `server` attribute. Python Anywhere's web server is automatically configured to use the `application` variable name, so we reassign it here.

Install libraries

Click on the **Consoles** tab and then click **Bash** under **Start a new console**. Your browser should show a black screen with a blinking cursor. You are now working directly with your server (running an Ubuntu Linux operating system) on the command line. We would normally create a virtual environment, but our resources are limited and most of the library dependencies for our project are already installed. We just need to install dash_bootstrap_components and statsmodels. Run the following command, making sure to use **pip3.8**. We don't have privileges to install system-wide libraries, which is why the `--user` option must be specified.

```
pip3.8 install --user dash_bootstrap_components==0.13.1 statsmodels==0.12.2
```

Verify dashboard is up and running

Click the menu in the top right corner and return to the **Web** tab. Click the big button towards the top of the page that **reloads** the app. Once it has finished reloading, visit your website to verify that the dashboard is running correctly. You'll need to reload your app every time you modify one of the Python files in order for it to use the new version.

Create daily task to run `update.py`

We can create a task to update the dashboard's data on a daily basis. Go back to your Python Anywhere account and click on **Tasks**. Create a new task with the following command (substituting both username instances):

```
cd /home/username/corona && python3.8 update.py && touch /var/www/username_pythonanywhere_com_wsgi
```

This is a bash command that will run once a day at the time you selected. It changes directories to the project directory and then runs `update.py` to get the latest data. It finally executes the `touch` command on the wsgi file, which changes the time it was last accessed. This has the effect of reloading the application automatically. Without this command, you'd have to manually reload your app by clicking the button in the **Web** tab. Make sure to substitute your actual Python Anywhere username in both places above.

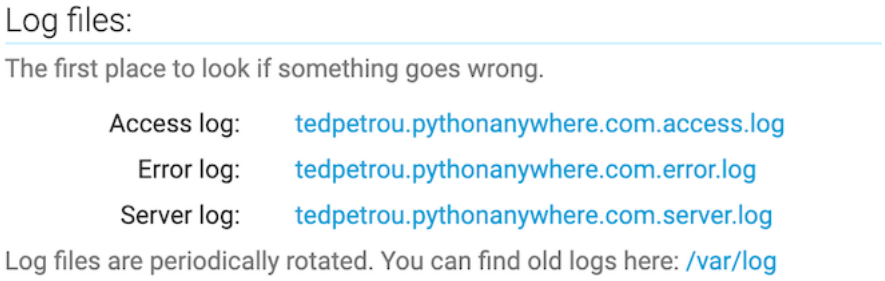
Log files

It's important to be able to find and fix errors in your application when they arise. Running the dashboard locally with `python dashboard.py` shows errors and warnings directly in the terminal. While this is fine for debugging errors locally, this isn't a solution when errors take place on a server with many users simultaneously accessing the dashboard.

Logging is the capturing of errors and other messages during the running of an application in a log file. Python Anywhere creates three log files available under the **Web** tab.

- **Access log** - Every time a user visits your website, a new line is written to this file. The date and time, operating system, browser and response time are recorded. In fact, every request (such as clicking an area or changing a tab) that the user makes on your site is recorded.

- **Error log** - The error log is probably the most important log and records any Python errors that arise when the app is running. They appear just as they do when running Python normally on your machine. All warnings will also be written here.
- **Server log** - When you click the button to reload the website, information on the server will be written here.



Your daily task from above also has a log file that you can inspect by clicking the first button under the **Actions** part directly to the right of the command.

Python Anywhere Deployment Complete

This completes the section on deploying your dashboard on Python Anywhere. In order to have it run indefinitely, you will need to log in from time to time to extend the lifetime of the app and task you created. If not, it will shut down, but should be easily brought back up by pressing the reload button in the **Web** tab.

13.3 Deploying on Ubuntu with Vultr

In this section, we'll cover a different and more complex deployment option. We'll rent a server running the Ubuntu operating system, a Linux distribution, from Vultr, a company with good reviews and much lower prices than the bigger names. An outline of the steps follows:

- Create Vultr account
- Launch Ubuntu server
- Upgrade account to use IPv4
- SSH into server
- Updating and upgrading with `apt`
- Install ZSH
- Install Python
- Install NginX
- Transfer project files to server
- Create virtual environment
- Configure Gunicorn with systemd
- Configure NginX to communicate with Gunicorn
- Access log files with `journalctl`
- Automatic daily updates with a cron job

Create Vultr Account

Vultr servers are not free, but are low cost. We will be using their lowest tier server, which costs $3.50 per month. Navigate to vultr.com to create an account.

Launch Ubuntu server

From your home page in Vultr, go to the **products** page, and click the **plus sign** on the right side of the page. The new page will show **Deploy new instance** at the top. Use the following list for the available options:

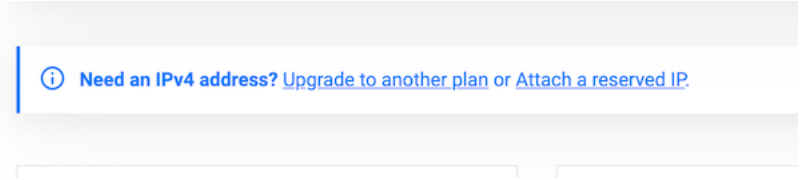
- Choose server - Cloud Compute
- Server location - Choose a location where $2.50 per month servers are available - scroll down the page to see the cost of the server. Some locations have a minimum of 5 dollars per month.
- Server type - Ubuntu 20.10 x64
- Server size - 10GB SSD, $2.50 per month, 1 CPU, 512 MB memory
- Server Hostname & Label - Enter a name of your choice for the server, i.e. “Coronavirus”

We will upgrade our server to use an IPv4 address later on, which will cost an additional one dollar per month. Click **deploy now**. At some point you’ll have entered your credit card information. You’ll have to wait 1-2 minutes until the instance has started before proceeding to the next step.

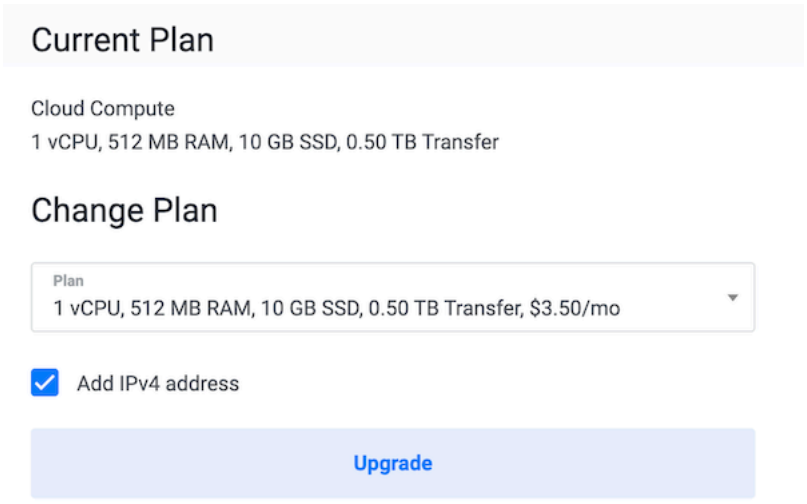
Upgrade account to use IPv4

To make it easier for everyone to connect to their instance and access their site online, we’ll upgrade our instance so that it uses an IPv4 address instead of the IPv6. There are a limited number of IPv4 addresses, which is why it costs additional money.

From the **products** page, click on your instance. Towards the top of the page, there will be a box asking if you need an IPv4 instance.



Click **Upgrade to another plan**. On the next page, change the plan to the one with the same specs as your current, but costs $3.50 per month. Check the box to **add IPv4 address**. The server will take a small amount of time to upgrade.

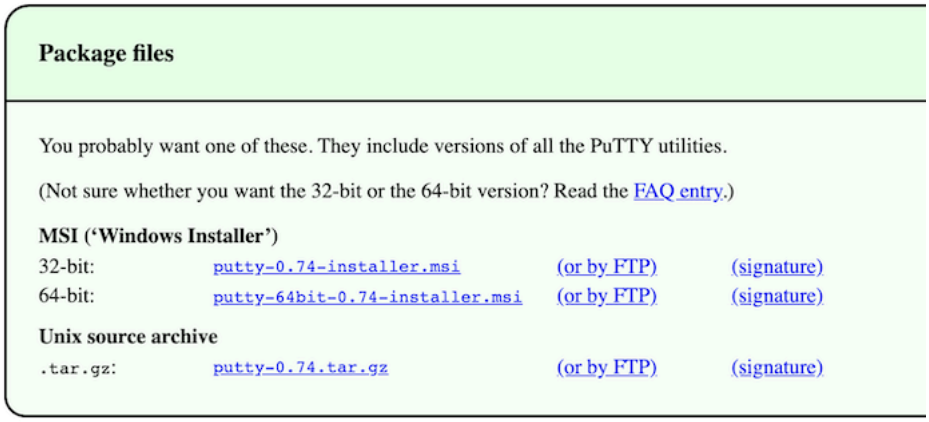


SSH into server

We will now log in to our remote machine using **SSH**. Windows users will need to take the following additional step in order to use SSH.

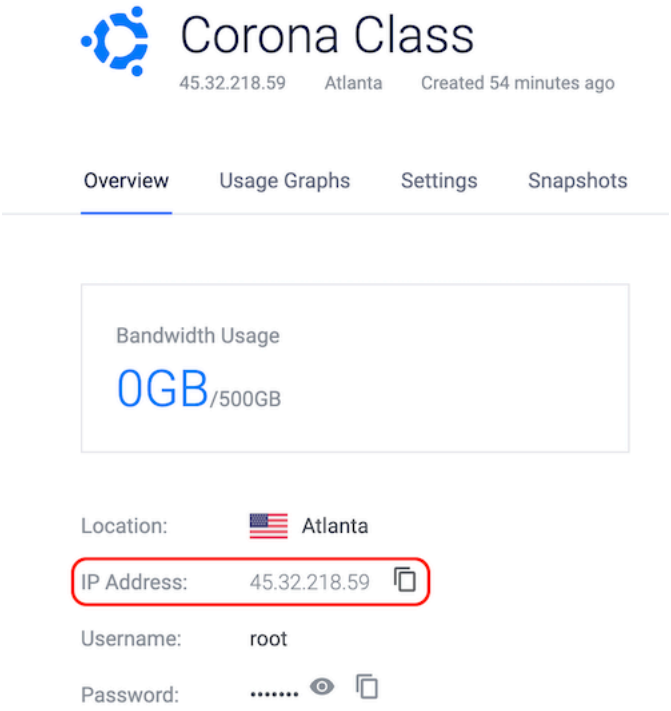
Windows users

You will need to install a program called **PuTTY**. Visit the installation page and download the correct version under the **Package Files** section.



All users

In the **Overview** section of your instance home page on Vultr, copy the **IP Address**.



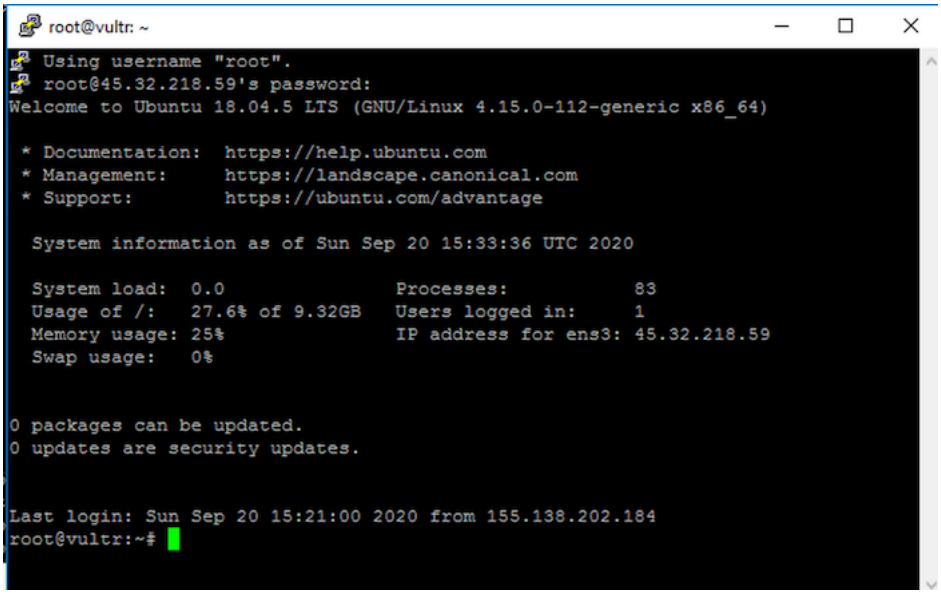
Windows users

Open up the **Command Prompt** program and run the following command, replacing **your_ip_address** with your actual IP Address:

```
putty -ssh root@your_ip_address
```

A new window will open up asking for your password. Copy the password from Vultr and paste it in the provided space. You CANNOT paste using ctrl + v. You MUST **right-click** once to paste the password. No new characters will appear on the screen. Press enter to connect to the server.

You'll get a window telling you that authenticity cannot be established. Click **yes** to continue connecting. You won't be prompted with this again. You should now be connected to the remote server and see the following prompt.



macOS and Linux users

Open up your terminal and run the following command, replacing **your_ip_address** with your actual IP Address:

```
ssh root@your_ip_address
```

You'll be prompted to enter your password. Copy and paste it from the Vultr page and press enter. You'll be asked to continue connecting because authenticity cannot be established. Respond with **yes**. You'll only need to do this once. You are now connected to the remote server.

All users

Now that we are connected to the remote server, we'll be using the same commands for all operating systems except when transferring files from our local machine to the server.

Updating and upgrading with `apt`

Ubuntu has the `apt` command which stands for **advanced packaging tool** that is used to install and manage software packages. Begin by running the following two commands

```
apt update
apt upgrade
```

The first updates the list of packages that have upgrades available. It does not upgrade any of the software packages. Once we have the updated list of packages that can get upgraded, we upgrade all of them with the second command.

Install ZSH (optional)

This step is optional. The default shell (program that runs our commands) is called bash. While this is fairly standard, I recommend using ZSH (the Z shell) which has many improvements over bash, namely that it is easier to find previous commands. The second command installs Oh My Zsh, which allows more customization and a nice coloring of commands.

```
apt install zsh
sh -c "$(curl -fsSL https://raw.github.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"
```

Enter yes to switch your shell to ZSH. After installation, your prompt will simplify to just an arrow and tilde.

Install Python

Python 3.8 comes pre-installed on this machine, but the package manager, pip, and the virtual environment creation module, venv, are not bundled together like they normally are. Install each of them with the following.

```
apt install -y python3-pip python3.8-venv
```

Install NginX

Install the NginX web server with the following command:

```
apt install nginx
```

The web server automatically starts up after installation. Verify that it is running by opening a new tab in your browser and navigating to the IP Address (just paste the address directly into the address bar). You should see the following page:

Transfer project files to server

We need to transfer our local files from the **project** folder over to the server. First, let's make a directory on the server to hold the files. The most common location to hold website files is in `/var/www/html`. Run the following command to make the "corona" directory within it.

```
mkdir /var/www/html/corona
```

Now, **switch back to your local machine** to transfer the files.

Windows users

Open up another **Command Prompt** on your local machine. Change directories so that you are in the **project** directory (with dashboard.py, prepare.py, etc…).

When you installed PuTTY, you also installed `pscp`, a program to copy files from one machine to another. Run the following command to recursively copy all the files and directories to the remote server. Replace **your__ip__address** with your actual IP Address. Enter your password when prompted to complete the transfer.

```
pscp -P 22 -r *.py data assets update.sh requirements.txt root@your_ip_address:/var/www/html/corona
```

macOS and Linux users

Open up another terminal window on your local machine. Change directories so that you are in the **project** directory (with dashboard.py, prepare.py, etc…). Run the following command to recursively copy all the files and directories to the remote server. Replace **your_ip_address** with your actual IP Address. Enter your password when prompted to complete the transfer.

```
scp -r *.py data assets update.sh requirements.txt root@your_ip_address:/var/www/html/corona
```

All users

Back on the server's command line, run the following to list the files in our project directory to verify the transfer happened successfully.

```
ls /var/www/html/corona
```

Create virtual environment

We have Python 3.8 installed, but need to create a virtual environment for the project. First, make sure you are in the project directory.

```
cd /var/www/html/corona
```

Create the virtual environment using the following code which runs the `venv` module as a script and names the virtual environment `dashbard_venv`.

```
python3 -m venv dashboard_venv
```

You should have a new directory titled `dashboard_venv`. Run `ls` to see it. Now, activate the virtual environment with:

```
source dashboard_venv/bin/activate
```

Your prompt should now have `(dashboard_venv)` prepended to it.

Update pip

An older version of pip is installed that we can update to the latest version with the following command.

```
pip install -U pip
```

Install the wheel package

The wheel Python package must be installed in order to correctly install the other packages:

```
pip install wheel
```

Install the requirements

Finally, install the remaining requirements with:

```
pip install -r requirements.txt
```

Configuring Gunicorn with systemd

When we run `python dashboard.py` on our local machine, only one user can interact with our program. There's even a warning in the terminal that states the following:

> Warning: This is a development server. Do not use app.run_server in production, use a production WSGI server like gunicorn instead.

Gunicorn is a Python package listed in the requirements.txt file that is able to process multiple requests from many different users. We need our server to continually run Gunicorn in the background. In order to do this, we'll use **systemd**, the main software on most Linux machines to start and manage background processes.

In order to use systemd to operate Gunicorn as one of its services, we need to create a configuration file called a **unit file**. Run the following command to create a new file for the service and drop you into a text editor (`nano` is a simple text editor available on all Linux distributions).

```
nano /etc/systemd/system/corona.service
```

Copy and paste the following into the editor and press **ctrl + X** to exit, enter **Y** to save.

```
[Unit]
Description=Gunicorn instance to serve Coronavirus Dashboard
After=network.target

[Service]
User=root
Group=www-data
WorkingDirectory=/var/www/html/corona
Environment="PATH=/var/www/html/corona/dashboard_venv/bin"
ExecStart=/var/www/html/corona/dashboard_venv/bin/gunicorn --workers 3 --bind unix:corona.sock -m 007 wsgi:app

[Install]
WantedBy=multi-user.target
```

Start the service and enable it on boot with the following commands:

```
systemctl start corona
systemctl enable corona
```

After starting the corona service, a corona.sock file will be generated in your project folder. You can verify this by running `ls /var/www/html/corona`. This socket is what Gunicorn uses to communicate with NginX.

Configure NginX to communicate with Gunicorn

Currently, NginX is configured to show its default site. In this step we'll add a separate configuration file so that it directs traffic to the Gunicorn server which runs our application. Run the following command to create a new NginX configuration file.

```
nano /etc/nginx/sites-available/corona
```

Copy and paste the following in the editor and exit (**ctrl + X** then **Y**).

```
server {
    listen 80;

    location / {
```

```
        include proxy_params;
        proxy_pass http://unix:/var/www/html/corona/corona.sock;
    }
}
```

NginX only looks for sites in the `sites-enabled` directory, so we create a symbolic link to it with the following:

```
ln -s /etc/nginx/sites-available/corona /etc/nginx/sites-enabled
```

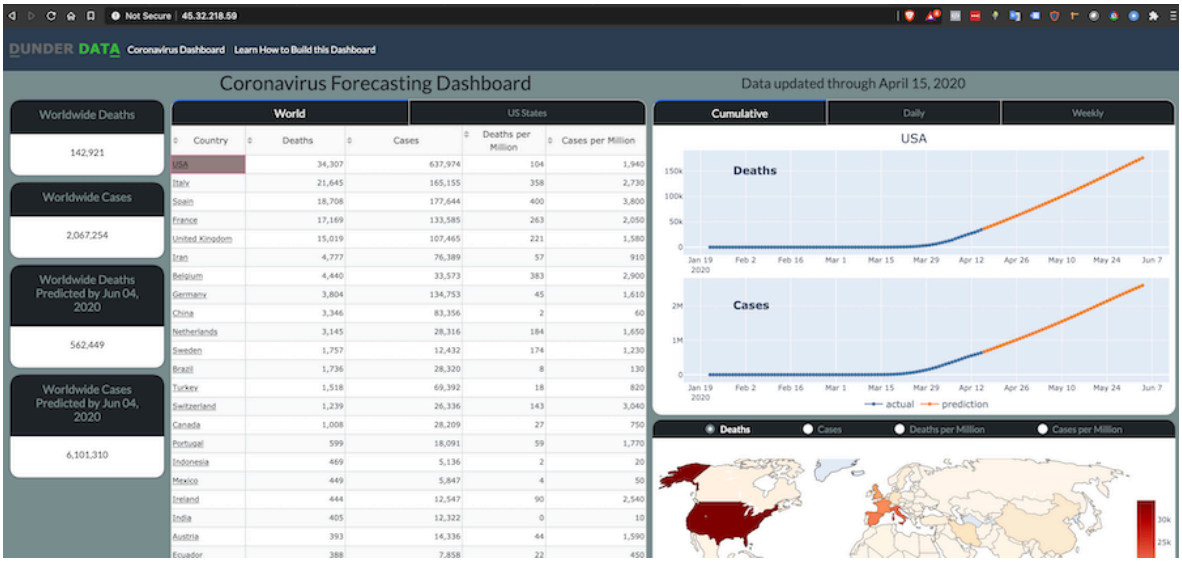
You can think of the sites-available as a staging area for the live links in sites-enabled. Finally, we delete the default file in sites-enabled so that only our site is enabled. The original copy of this file is still in `/etc/nginx/sites-available`.

```
rm /etc/nginx/sites-enabled/default
```

Restart NginX, which is also run as a systemd service, to make the changes:

```
systemctl restart nginx
```

This completes the deployment. Visit your IP Address once again to verify that you can see the dashboard.



Accessing log files with `journalctl`

All of the log files for Ubuntu are stored in the `/var/log` directory, which you can access directly. The `journalctl` command provides an alternative way to access the logs without knowing the exact path to them. The `journalctl` "accesses the systemd journal" and called without arguments shows all of the logs from the beginning.

It's rare that you'll want to access the entire journal. More likely, you'll access logs from specific systemd services. The following command retrieves the logs from our dashboard service with the `-u` option, passing it the name of our unit file. You can use either **corona** or **corona.service**. By default, the beginning of the log file is displayed. Use option `-n` to show the newest lines. Here we retrieve the last 100 lines of the log file.

```
journalctl -u corona -n 100
```

When the journal displays, use arrow keys to scroll through it or press **d** or **u** to page down or up half of the screen at a time. Press **q** to quit and return to the command line. See this tutorial from Digital Ocean to learn more about the options available from `journalctl`.

Automatic daily updates with a cron job

The `update.py` file needs to be run daily to update our dashboard with the latest data. Cron jobs provide a way to schedule tasks to run at regular intervals. There are two main parts to each cron job, the time interval, and the command to execute. Take a look at the image below (courtesy of Wikipedia) labeling the parts of a cron job.

```
# ┌──────────── minute (0 - 59)
# │ ┌────────── hour (0 - 23)
# │ │ ┌──────── day of the month (1 - 31)
# │ │ │ ┌────── month (1 - 12)
# │ │ │ │ ┌──── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * <command to execute>
```

There are five units of time that can be provided and a few different syntaxes to represent multiple of a particular unit. An asterisk is used to represent every possible integer value for a time unit. Check out the crontab guru for an interactive way to learn about the syntax. Run the following command to open up the crontab editor. Press 1 to use the nano editor.

```
crontab -e
```

Copy and paste the cron job below. When you exit and save you'll see a message stating "installing new crontab".

```
0 6 * * * . /var/www/html/corona/update.sh
```

This cron job runs daily at 6 a.m. UTC and executes the commands in the `update.sh` shell script which is copied here.

```
cd /var/www/html/corona
dashboard_venv/bin/python update.py >> /var/log/corona_cron.log 2>&1
systemctl restart corona
```

It changes to the home directory of the app, then runs the `update.py` script using the python executable from our environment. It redirects both standard output and error to a file and then restarts the systemd service that is running our app. We haven't created the log file, so let's do that now.

```
touch /var/log/corona_cron.log
```

You can check this log by running `tail /var/log/corona_cron.log -n 100` to view the last 100 lines of output or errors if there are any.

13.4 Deployment complete

Deployment of your dashboard is now fully complete. It should run indefinitely, updating once per day. Use the journal and cron job log to find errors.

13.5 Summary of all commands

1. Create Vultr account
2. Launch Ubuntu server
    1. Choose server - Cloud Compute
    2. Server location - Choose any location
    3. Server type - Ubuntu 20.10 x64
    4. Server size - 10GB SSD, $2.50 per month, 1 CPU, 512 MB memory
    5. Server Hostname & Label - Enter a name of your choice for the server, i.e. “Corona”
3. Upgrade to IPv4
    1. Click upgrade to IPv4 and upgrade to $3.50 per month plan
4. SSH into server
    1. Windows - `putty -ssh root@your_ip_address`
    2. macOS/Linux - `ssh root@your_ip_address`
5. Updating and upgrading with `apt`
    1. `apt update`
    2. `apt upgrade`
6. Install ZSH (optional)
    1. `apt install zsh`
    2. `sh -c "$(curl -fsSL https://raw.github.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"`
7. Install Python
    1. `apt install python3-pip python3.8-venv`
8. Install NginX
    1. `apt install nginx`
    2. Navigate to your IP Address in your browser to see ngninx default page
9. Transfer project files to server
    1. `cd /var/www/html`
    2. `mkdir corona`
    3. Switch to local computer
    4. cd to project directory
        1. Windows

```
pscp -P 22 -r *.py data assets update.sh requirements.txt root@your_ip_address:/var/www/html/corona
```

        2. macOS/Linux

```
scp -r *.py data assets update.sh requirements.txt root@your_ip_address:/var/www/html/corona
```

    5. Switch back to remote server
    6. Verify files transfered with `ls /var/www/html/corona`
10. Create virtual environment
    1. `cd /var/www/html/corona`
    2. `python3 -m venv dashboard_venv`
    3. `source dashboard_venv/bin/activate`
    4. `pip install -U pip`
    5. `pip install wheel`
    6. `pip install -r requirements.txt`
11. Configuring Gunicorn with systemd
    1. `nano /etc/systemd/system/corona.service`
    2. Place the following in the file

```
[Unit]
Description=Gunicorn instance to serve Coronavirus Dashboard
After=network.target

[Service]
User=root
Group=www-data
WorkingDirectory=/var/www/html/corona
Environment="PATH=/var/www/html/corona/dashboard_venv/bin"
ExecStart=/var/www/html/corona/dashboard_venv/bin/gunicorn --workers 3 --bind unix:corona.sock -m 007 wsgi:app

[Install]
WantedBy=multi-user.target
```

3. Ctrl + X to exit
4. `systemctl start corona`
5. `systemctl enable corona`
6. Verify corona.sock file is in /var/www/html/corona

12. Configure NginX to communicate with Gunicorn
    1. `nano /etc/nginx/sites-available/corona`
    2. Place the following in the file

```
server {
    listen 80;

    location / {
        include proxy_params;
        proxy_pass http://unix:/var/www/html/corona/corona.sock;
    }
}
```

    3. `ln -s /etc/nginx/sites-available/corona /etc/nginx/sites-enabled`
    4. `rm /etc/nginx/sites-enabled/default`
    5. `nginx -s reload`
13. Navigate to IP Address in browser to verify dashboard is working
14. Use the journal to access logs for the corona unit
    1. `journalctl -u corona -n 100` - displays last 100 log lines
15. Setup a cronjob to run `update.py` daily
    1. `crontab -e`
    2. `0 6 * * * . /var/www/html/corona/update.sh`
    3. Create log file specifically for this cron job `touch /var/log/corona_cron.log`