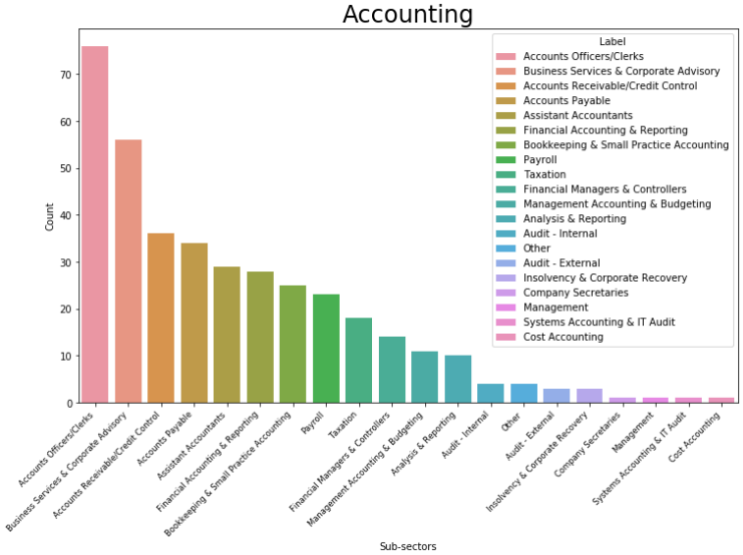
Lab 5 – Data Visualization and Visual Analysis

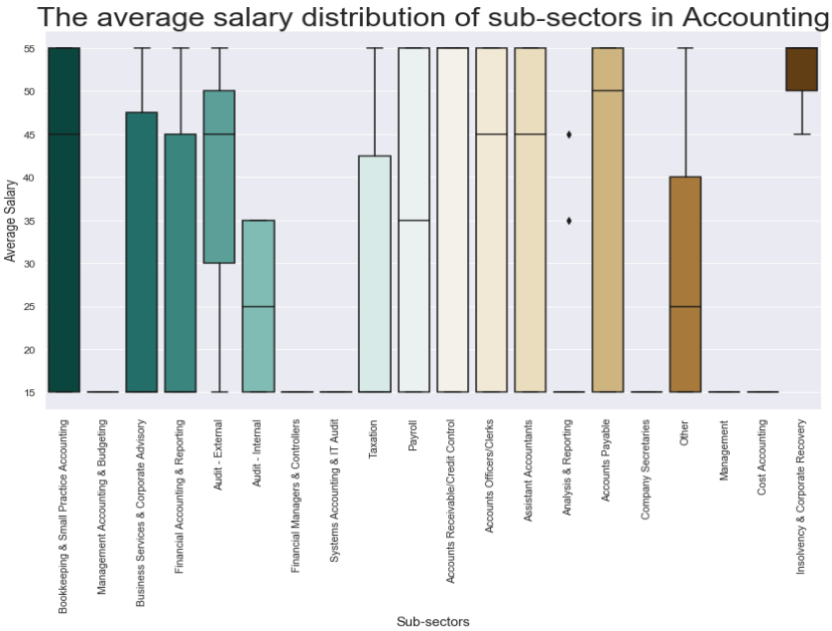
In this workshop, you will continue using Matplotlib, Seaborn and Plotly to explore the data.

Filter from the dataset all the “Accounting” job and visualize the total job postings of each sub-sectors.



Count the job postings by month and visualize in line graph using Matplotlib, Seaborn and Plotly.

Visualize salary distribution of sub-sector in Accounting using boxplot of Matplotlib, Seaborn and Plotly.



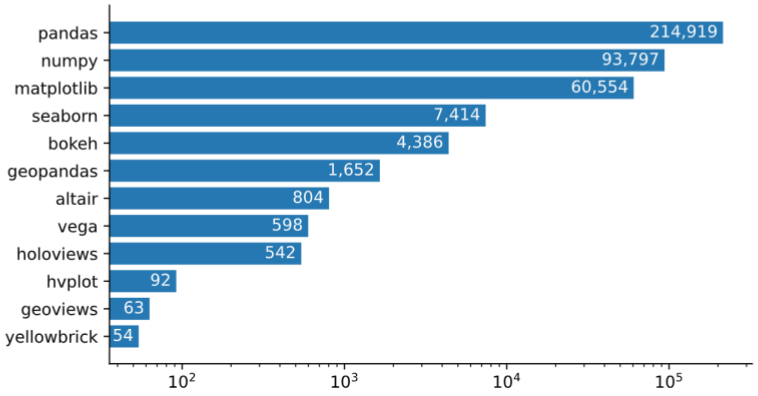
Section 2: Moving Beyond Static Visualizations

Static visualizations are limited in how much information they can show. To move beyond these limitations, we can create animated and/or interactive visualizations. Animations make it possible for our visualizations to tell a story through movement of the plot components (e.g., bars, points, lines). Interactivity makes it possible to explore the data visually by hiding and displaying information based on user interest. In this section, we will focus on creating animated visualizations using Matplotlib before moving on to create interactive visualizations in the next section.

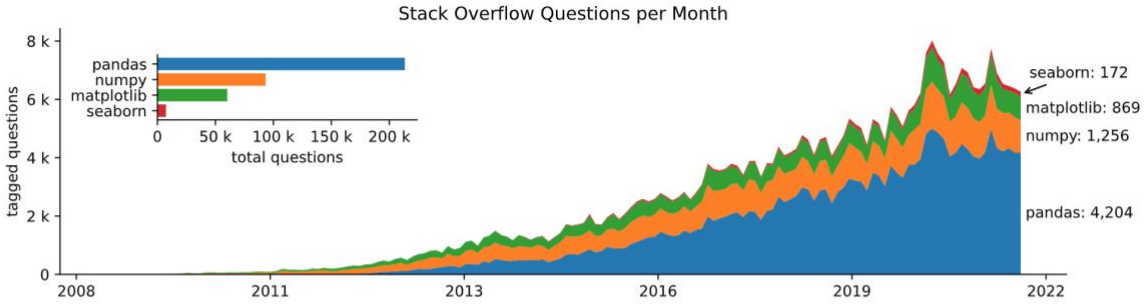
Animating cumulative values over time

In the previous section, we made a couple of visualizations to help us understand the number of Stack Overflow questions per library and how it changed over time. However, each of these came with some limitations.

We made a bar plot that captured the total number of questions per library, but it couldn't show us the growth in pandas questions over time (or how the growth rate changed over time):



We also made an area plot showing the number of questions per day over time for the top 4 libraries, but by limiting the libraries shown we lost some information:



Both of these visualizations gave us insight into the dataset. For example, we could see that pandas has by far the largest number of questions and has been growing at a faster rate than the other libraries. While this comes from studying the plots, an animation would make this much more obvious and, at the same time, capture the exponential growth in pandas questions that helped pandas overtake both Matplotlib and NumPy in cumulative questions.

Let's use Matplotlib to create an animated bar plot of cumulative questions over time to show this. We will do so in the following steps:

1. Create a dataset of cumulative questions per library over time.
2. Import the `FuncAnimation` class.
3. Write a function for generating the initial plot.
4. Write a function for generating annotations and plot text.
5. Define the plot update function.
6. Bind arguments to the update function.
7. Animate the plot.

1. Create a dataset of cumulative questions per library over time.

We will start by reading in our Stack Overflow dataset, but this time, we will calculate the total number of questions per month and then calculate the cumulative value

In [1]:
```
import pandas as pd

questions_per_library = pd.read_csv(
    '../data/stackoverflow.zip', parse_dates=True, index_col='creation_date'
).loc[:,'pandas':'bokeh'].resample('1M').sum().cumsum().reindex(
    pd.date_range('2008-08', '2021-10', freq='M')
).fillna(0)
questions_per_library.tail()
```

Out[1]:

	pandas	matplotlib	numpy	seaborn	geopandas	geoviews	altair	yellowbrick	vega	holoviews	hvplot	bokeh
2021-05-31	200734.0	57853.0	89812.0	6855.0	1456.0	57.0	716.0	46.0	532.0	513.0	84.0	4270.0
2021-06-30	205065.0	58602.0	91026.0	7021.0	1522.0	57.0	760.0	48.0	557.0	521.0	88.0	4308.0
2021-07-31	209235.0	59428.0	92254.0	7174.0	1579.0	62.0	781.0	50.0	572.0	528.0	89.0	4341.0
2021-08-31	213410.0	60250.0	93349.0	7344.0	1631.0	62.0	797.0	52.0	589.0	541.0	92.0	4372.0
2021-09-30	214919.0	60554.0	93797.0	7414.0	1652.0	63.0	804.0	54.0	598.0	542.0	92.0	4386.0

Source: Stack Exchange Network

2. Import the `FuncAnimation` class.

To create animations with Matplotlib, we will be using the `FuncAnimation` class, so let's import it now:

In [2]:
```
from matplotlib.animation import FuncAnimation
```

At a minimum, we will need to provide the following when instantiating a `FuncAnimation` object:

- The `Figure` object to draw on.
- A function to call at each frame to update the plot.

In the next few steps, we will work on the logic for these.

3. Write a function for generating the initial plot.

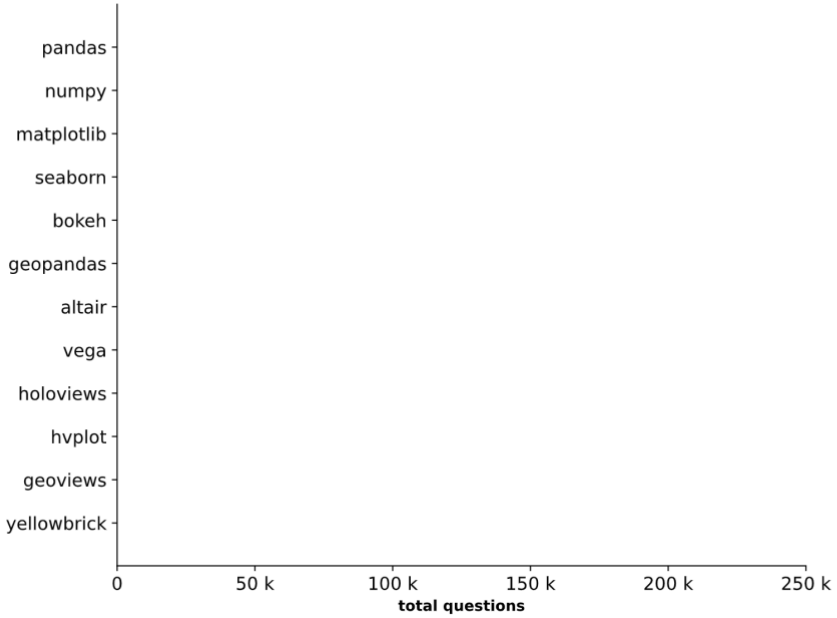
Since we are required to pass in a `Figure` object and bake all the plot update logic into a function, we will start by building up an initial plot. Here, we create a bar plot with bars of width 0, so that they don't show up for now. The y-axis is set up so that the libraries with the most questions overall are at the top:

In [3]:
```
import matplotlib.pyplot as plt
from matplotlib import ticker

def bar_plot(data):
    fig, ax = plt.subplots(figsize=(8, 6))
    sort_order = data.last('1M').squeeze().sort_values().index
    bars = [
        bar.set_label(label) for label, bar in
        zip(sort_order, ax.barh(sort_order, [0] * data.shape[1]))
    ]

    ax.set_xlabel('total questions', fontweight='bold')
    ax.set_xlim(0, 250_000)
    ax.xaxis.set_major_formatter(ticker.EngFormatter())
    ax.xaxis.set_tick_params(labelsize=12)
    ax.yaxis.set_tick_params(labelsize=12)

    for spine in ['top', 'right']:
        ax.spines[spine].set_visible(False)

    fig.tight_layout()

    return fig, ax
```

This gives us a plot that we can update:

In [4]:
```
%config InlineBackend.figure_formats = ['svg']
%matplotlib inline
bar_plot(questions_per_library)
```

Out[4]:
```
(<Figure size 576x432 with 1 Axes>, <AxesSubplot:xlabel='total questions'>)
```



4. Write a function for generating annotations and plot text.

We will also need to initialize annotations for each of the bars and some text to show the date in the animation (month and year):

In [5]:
```
def generate_plot_text(ax):
    annotations = [
        ax.annotate(
            '', xy=(0, bar.get_y() + bar.get_height()/2),
            ha='left', va='center'
        ) for bar in ax.patches
    ]

    time_text = ax.text(
        0.9, 0.1, '', transform=ax.transAxes,
        fontsize=15, ha='center', va='center'
    )
    return annotations, time_text
```

Tip: We are passing in `transform=ax.transAxes` when we place our time text in order to specify the location in terms of the `Axes` object's coordinates instead of basing it off the data in the plot so that it is easier to place.

5. Define the plot update function.

Next, we will make our plot update function. This will be called at each frame. We will extract that frame's data (the cumulative questions for that month), and then update the width of each of the bars. In addition, we will annotate the bars if their widths are greater than 0. At every frame, we will also need to update our time annotation (`time_text`):

In [6]:
```
def update(frame, *, ax, df, annotations, time_text):
    data = df.loc[frame, :]

    # update bars
    for rect, text in zip(ax.patches, annotations):
        col = rect.get_label()
        if data[col]:
            rect.set_width(data[col])
            text.set_x(data[col])
            text.set_text(f' {data[col]:,.0f}')

    # update time
    time_text.set_text(frame.strftime('%b\n%Y'))
```

Tip: The asterisk in the function signature requires all arguments after it to be passed in by name. This makes sure that we explicitly define the components for the animation when calling the function. Read more on this syntax here.

6. Bind arguments to the update function.

The last step before creating our animation is to create a function that will assemble everything we need to pass to `FuncAnimation` . Note that our `update()` function requires multiple parameters, but we would be passing in the same values every time (since we would only change the value for `frame`). To make this simpler, we create a partial function, which **binds** values to each of those arguments so that we only have to pass in `frame` when we call the partial. This is essentially a closure, where `bar_plot_init()` is the enclosing function and `update()` is the nested function, which we defined in the previous code block for readability:

In [7]:
```
from functools import partial

def bar_plot_init(questions_per_library):
    fig, ax = bar_plot(questions_per_library)
    annotations, time_text = generate_plot_text(ax)

    bar_plot_update = partial(
        update, ax=ax, df=questions_per_library,
        annotations=annotations, time_text=time_text
    )

    return fig, bar_plot_update
```

7. Animate the plot.

Finally, we are ready to create our animation. We start by calling the `bar_plot_init()` function from the previous code block to generate the `Figure` object and partial function for the update of the plot. Then, we pass in the `Figure` object and update function when initializing our `FuncAnimation` object. We also specify the `frames` argument as the index of our DataFrame (the dates) and that the animation shouldn't repeat because we will save it as an MP4 video:

In [8]:
```
fig, update_func = bar_plot_init(questions_per_library)

ani = FuncAnimation(
    fig, update_func, frames=questions_per_library.index, repeat=False
)
ani.save(
    '../media/stackoverflow_questions.mp4',
    writer='ffmpeg', fps=10, bitrate=100, dpi=300
)
plt.close()
```

Important: The `FuncAnimation` object **must** be assigned to a variable when creating it; otherwise, without any references to it, Python will garbage collect it – ending the animation. For more information on garbage collection in Python, check out this article.

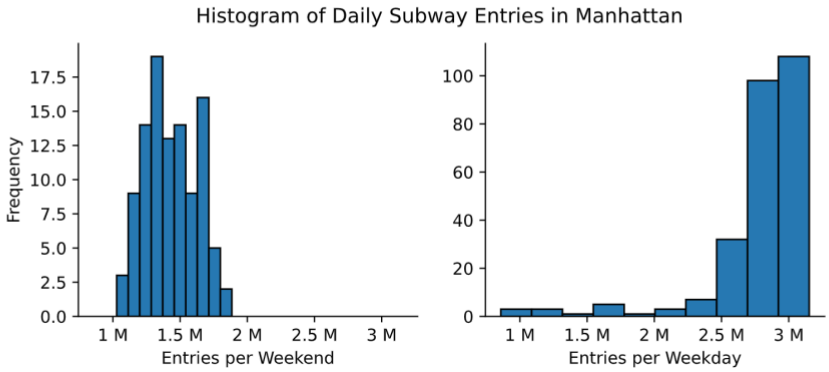
Now, let's view the animation we just saved as an MP4 file:

In [9]:
```
from IPython import display

display.Video(
    '../media/stackoverflow_questions.mp4', width=600, height=400,
    embed=True, html_attributes='controls muted autoplay'
)
```

Out[9]:

Animating distributions over time

As with the previous example, the histograms of daily Manhattan subway entries in 2018 (from the first section of the workshop) don't tell the whole story of the dataset because the distributions changed drastically in 2020 and 2021:



We will make an animated version of these histograms that enables us to see the distributions changing over time. Note that this example will have two key differences from the previous one. The first is that we will be animating subplots rather than a single plot, and the second is that we will use a technique called **blitting** to only update the portion of the subplots that has changed. This requires that we return the *artists* that need to be redrawn in the plot update function.

To make this visualization, we will work through these steps:

1. Create a dataset of daily subway entries.
2. Determine the bin ranges for the histograms.
3. Write a function for generating the initial histogram subplots.
4. Write a function for generating an annotation for the time period.
5. Define the plot update function.
6. Bind arguments for the update function.
7. Animate the plot.

1. Create a dataset of daily subway entries.

As we did previously, we will read in the subway dataset, which contains the total entries and exits per day per borough:

In [10]:
```
subway = pd.read_csv(
    '../data/NYC_subway_daily.csv', parse_dates=['Datetime'],
    index_col=['Borough', 'Datetime']
)
subway_daily = subway.unstack(0)
subway_daily.head()
```

Out[10]:

	Entries				Exits			
Borough	**Bk**	**Bx**	**M**	**Q**	**Bk**	**Bx**	**M**	**Q**
Datetime								
2017-02-04	617650.0	247539.0	1390496.0	408736.0	417449.0	148237.0	1225689.0	279699.0
2017-02-05	542667.0	199078.0	1232537.0	339716.0	405607.0	139856.0	1033610.0	268626.0
2017-02-06	1184916.0	472846.0	2774016.0	787206.0	761166.0	267991.0	2240027.0	537780.0
2017-02-07	1192638.0	470573.0	2892462.0	790557.0	763653.0	270007.0	2325024.0	544828.0
2017-02-08	1243658.0	497412.0	2998897.0	825679.0	788356.0	275695.0	2389534.0	559639.0

Source: The above dataset was resampled from this dataset provided by Kaggle user Edden.

For this visualization, we will just be working with the entries in Manhattan:

In [11]:
```
manhattan_entries = subway_daily['Entries']['M']
```

2. Determine the bin ranges for the histograms.

Before we can set up the subplots, we have to calculate the bin ranges for the histograms so that our animation is smooth. NumPy provides the `histogram()` function, which gives us both the number of data points in each bin and the bin ranges, respectively. We will also be using this function to update the histograms during the animation:

In [12]:
```
import numpy as np

count_per_bin, bin_ranges = np.histogram(manhattan_entries, bins=30)
```

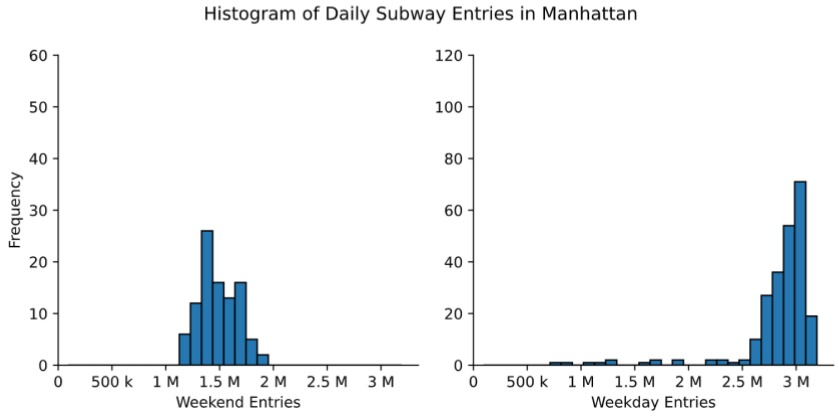
3. Write a function for generating the initial histogram subplots.

Next, we will handle the logic for building our initial histogram, packaging it in a function:

In [13]:
```
def subway_histogram(data, bins, date_range):
    _, bin_ranges = np.histogram(data, bins=bins)

    weekday_mask = data.index.weekday < 5
    configs = [
        {'label': 'Weekend', 'mask': ~weekday_mask, 'ymax': 60},
        {'label': 'Weekday', 'mask': weekday_mask, 'ymax': 120}
    ]

    fig, axes = plt.subplots(1, 2, figsize=(8, 4), sharex=True)
    for ax, config in zip(axes, configs):
        _, _, config['hist'] = ax.hist(
            data[config['mask']].loc[date_range], bin_ranges, ec='black'
        )
        ax.xaxis.set_major_formatter(ticker.EngFormatter())
        ax.set(
            xlim=(0, None), ylim=(0, config['ymax']),
            xlabel=f'{config["label"]} Entries'
        )
        for spine in ['top', 'right']:
            ax.spines[spine].set_visible(False)

    axes[0].set_ylabel('Frequency')
    fig.suptitle('Histogram of Daily Subway Entries in Manhattan')
    fig.tight_layout()

    return fig, axes, bin_ranges, configs
```

Notice that our plot this time starts out with data already – this is because we want to show the change in the distribution of daily entries in the last year:

In [14]:
```
_ = subway_histogram(manhattan_entries, bins=30, date_range='2017')
```



4. Write a function for generating an annotation for the time period.

We will once again include some text that indicates the time period as the animation runs. This is similar to what we had in the previous example:

In [15]:
```
def add_time_text(ax):
    time_text = ax.text(
        0.15, 0.9, '', transform=ax.transAxes,
        fontsize=15, ha='center', va='center'
    )
    return time_text
```

5. Define the plot update function.

Now, we will create our update function. This time, we have to update both subplots and return any artists that need to be redrawn since we are going to use blitting:

In [16]:
```
def update(frame, *, data, configs, time_text, bin_ranges):
    artists = []

    time = frame.strftime('%b\n%Y')
    if time != time_text.get_text():
        time_text.set_text(time)
        artists.append(time_text)

    for config in configs:
        time_frame_mask = \
            (data.index > frame - pd.Timedelta(days=365)) & (data.index <= frame)
        counts, _ = np.histogram(
            data[time_frame_mask & config['mask']],
            bin_ranges
        )
        for count, rect in zip(counts, config['hist'].patches):
            if count != rect.get_height():
                rect.set_height(count)
                artists.append(rect)

    return artists
```

6. Bind arguments for the update function.

As our final step before generating the animation, we bind our arguments to the update function using a partial function:

In [17]:
```
def histogram_init(data, bins, initial_date_range):
    fig, axes, bin_ranges, configs = subway_histogram(data, bins, initial_date_range)

    update_func = partial(
        update, data=data, configs=configs,
        time_text=add_time_text(axes[0]),
        bin_ranges=bin_ranges
    )

    return fig, update_func
```

7. Animate the plot.

Finally, we will animate the plot using `FuncAnimation` like before. Notice that this time we are passing in `blit=True`, so that only the artists that we returned in the `update()` function are redrawn. We are specifying to make updates for each day in the data starting on August 1, 2019:

In [18]:
```
fig, update_func = histogram_init(
    manhattan_entries, bins=30, initial_date_range=slice('2017', '2019-07')
)

ani = FuncAnimation(
    fig, update_func, frames=manhattan_entries['2019-08':'2021'].index,
    repeat=False, blit=True
)
ani.save(
    '../media/subway_entries_subplots.mp4',
    writer='ffmpeg', fps=30, bitrate=500, dpi=300
)
plt.close()
```

Tip: We are using a `slice` object to pass a date range for pandas to use with `loc[]`. More information on `slice()` can be found here.

7. Animate the plot.

Finally, we will animate the plot using `FuncAnimation` like before. Notice that this time we are passing in `blit=True`, so that only the artists that we returned in the `update()` function are redrawn. We are specifying to make updates for each day in the data starting on August 1, 2019:

In [18]:
```
fig, update_func = histogram_init(
    manhattan_entries, bins=30, initial_date_range=slice('2017', '2019-07')
)

ani = FuncAnimation(
    fig, update_func, frames=manhattan_entries['2019-08':'2021'].index,
    repeat=False, blit=True
)
ani.save(
    '../media/subway_entries_subplots.mp4',
    writer='ffmpeg', fps=30, bitrate=500, dpi=300
)
plt.close()
```

Tip: We are using a `slice` object to pass a date range for pandas to use with `loc[]`. More information on `slice()` can be found here.

Our animation makes it easy to see the change in the distributions over time:

In [19]:
```
from IPython import display

display.Video(
    '../media/subway_entries_subplots.mp4', width=600, height=400,
    embed=True, html_attributes='controls muted autoplay'
)
```

Out[19]:

Animating geospatial data with HoloViz

HoloViz provides multiple high-level tools that aim to simplify data visualization in Python. For this example, we will be looking at HoloViews and GeoViews, which extends HoloViews for use with geographic data. HoloViews abstracts away some of the plotting logic, removing boilerplate code and making it possible to easily switch backends (e.g., switch from Matplotlib to Bokeh for JavaScript-powered, interactive plotting). To wrap up our discussion on animation, we will use GeoViews to create an animation of earthquakes per month in 2020 on a map of the world.

To make this visualization, we will work through the following steps:

1. Use GeoPandas to read in our data.
2. Handle HoloViz imports and set up the Matplotlib backend.
3. Define a function for plotting earthquakes on a map using GeoViews.
4. Create a mapping of frames to plots using HoloViews.
5. Animate the plot.

1. Use GeoPandas to read in our data.

Our dataset is in GeoJSON format, so the best way to read it in will be to use GeoPandas, which is a library that makes working with geospatial data in Python easier. It builds on top of pandas, so we don't have to learn any additional syntax for this example.

Here, we import GeoPandas and then use the `read_file()` function to read the earthquakes GeoJSON data into a `GeoDataFrame` object:

In [20]:
```
import geopandas as gpd

earthquakes = gpd.read_file('../data/earthquakes.geojson').assign(
    time=lambda x: pd.to_datetime(x.time, unit='ms'),
    month=lambda x: x.time.dt.month
)[['geometry', 'mag', 'time', 'month']]

earthquakes.shape
```

Out[20]:
```
(188527, 4)
```

Our data looks like this:

In [21]:
```
earthquakes.head()
```

Out[21]:

	geometry	mag	time	month
0	POINT Z (-67.12750 19.21750 12.00000)	2.75	2020-01-01 00:01:56.590	1
1	POINT Z (-67.09010 19.07660 6.00000)	2.55	2020-01-01 00:03:38.210	1
2	POINT Z (-66.85410 17.87050 6.00000)	1.81	2020-01-01 00:05:09.440	1
3	POINT Z (-66.86360 17.89930 8.00000)	1.84	2020-01-01 00:05:36.930	1
4	POINT Z (-66.86850 17.90660 8.00000)	1.64	2020-01-01 00:09:20.060	1

Source: USGS API

2. Handle HoloViz imports and set up the Matplotlib backend.

Since our earthquakes dataset contains geometries, we will use GeoViews in addition to HoloViews to create our animation. For this example, we will be using the Matplotlib backend:

In [22]:
```
import geoviews as gv
import geoviews.feature as gf
import holoviews as hv

gv.extension('matplotlib')
```

3. Define a function for plotting earthquakes on a map using GeoViews.

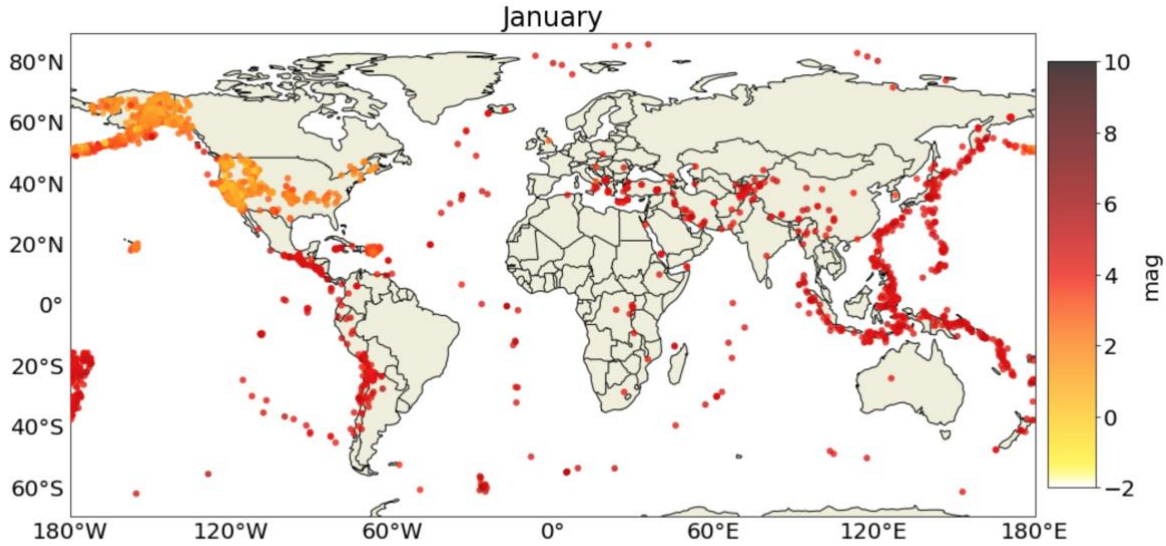
Next, we will write a function to plot each earthquake as a point on the world map. Since our dataset has geometries, we can use that information to plot them and then color each point by the earthquake magnitude. Note that, since earthquakes are measured on a logarithmic scale, some magnitudes are negative:

In [23]:
```
import calendar

def plot_earthquakes(data, month_num):
    points = gv.Points(
        data.query(f'month == {month_num}'),
        kdims=['longitude', 'latitude'], # key dimensions (for coordinates in this case)
        vdims=['mag'] # value dimensions (for modifying the plot in this case)
    ).redim.range(mag=(-2, 10), latitude=(-90, 90))

    # create an overlay by combining Cartopy features and the points with *
    overlay = gf.land * gf.coastline * gf.borders * points

    return overlay.opts(
        gv.opts.Points(color='mag', cmap='fire_r', colorbar=True, alpha=0.75),
        gv.opts.Overlay(
            global_extent=False, title=f'{calendar.month_name[month_num]}', fontscale=2
        )
    )
```

Our function returns an `Overlay` of earthquakes (represented as `Points`) on a map of the world. Under the hood GeoViews is using Cartopy to create the map:

In [24]:
```
plot_earthquakes(earthquakes, 1).opts(
    fig_inches=(6, 3), aspect=2, fig_size=250, fig_bounds=(0.07, 0.05, 0.87, 0.95)
)
```

Out[24]:



Tip: One thing that makes working with geospatial data difficult is handling projections. When working with datasets that use different projections, GeoViews can help align them – check out their tutorial here.

4. Create a mapping of frames to plots using HoloViews.

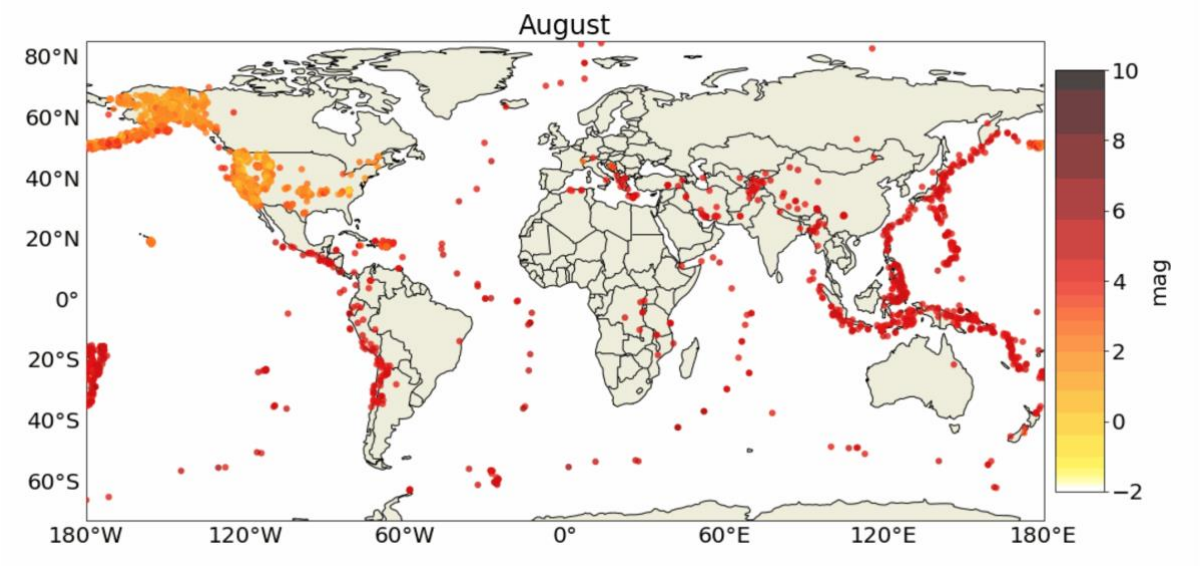
We will create a `HoloMap` of the frames to include in our animation. This maps the frame to the plot that should be rendered at that frame:

In [25]:
```
frames = {
    month_num: plot_earthquakes(earthquakes, month_num)
    for month_num in range(1, 13)
}
holomap = hv.HoloMap(frames)
```

5. Animate the plot.

Now, we will output our `HoloMap` as a GIF animation, which may take a while to run:

In [26]:
```
hv.output(
    holomap.opts(
        fig_inches=(6, 3), aspect=2, fig_size=250,
        fig_bounds=(0.07, 0.05, 0.87, 0.95)
    ), holomap='gif', fps=5
)
```



To save the animation to a file, run the following code:

```
hv.save(
    holomap.opts(
        fig_inches=(6, 3), aspect=2, fig_size=250,
        fig_bounds=(0.07, 0.05, 0.87, 0.95)
    ), 'earthquakes.gif', fps=5
)
```

Up Next: Building Interactive Visualizations for Data Exploration

Let's take a break for some exercises on animation to check your understanding:

1. Modify the animation of subway entries from this section to show both the weekday and weekend histograms on the same subplot (you only need one now). Don't forget to change the transparency of the bars to be able to visualize the overlap.
2. Modify the earthquake animation to show earthquakes per day in April 2020.

Exercises

1. Modify the animation of subway entries from this section to show both the weekday and weekend histograms on the same subplot.

In [ ]:

```
```

2. Modify the earthquake animation to show earthquakes per day in April 2020.

In [ ]:

```
```

Section 3: Building Interactive Visualizations for Data Exploration

When exploring our data, interactive visualizations can provide the most value. Without having to create multiple iterations of the same plot, we can use mouse actions (e.g., click, hover, zoom, etc.) to explore different aspects and subsets of the data. In this section, we will learn how to use a few of the libraries in the HoloViz ecosystem to create interactive visualizations for exploring our data utilizing the Bokeh backend.

1. Read in and prepare the data.

As we did in the previous section, we will use GeoPandas to read in our dataset. We are once again creating a new column for the month, but this time, we are also dropping any rows with missing information:

In [1]:

```
import geopandas as gpd
import pandas as pd

earthquakes = gpd.read_file('../data/earthquakes.geojson').assign(
    time=lambda x: pd.to_datetime(x.time, unit='ms'),
    month=lambda x: x.time.dt.month
).dropna()

earthquakes.head()
```

Out[1]:

	mag	place	time	tsunami	magType	geometry	month
0	2.75	80 km N of Isabela, Puerto Rico	2020-01-01 00:01:56.590	0	md	POINT Z (-67.12750 19.21750 12.00000)	1
1	2.55	64 km N of Isabela, Puerto Rico	2020-01-01 00:03:38.210	0	md	POINT Z (-67.09010 19.07660 6.00000)	1
2	1.81	12 km SSE of Maria Antonia, Puerto Rico	2020-01-01 00:05:09.440	0	md	POINT Z (-66.85410 17.87050 6.00000)	1
3	1.84	9 km SSE of Maria Antonia, Puerto Rico	2020-01-01 00:05:36.930	0	md	POINT Z (-66.86360 17.89930 8.00000)	1
4	1.64	8 km SSE of Maria Antonia, Puerto Rico	2020-01-01 00:09:20.060	0	md	POINT Z (-66.86850 17.90660 8.00000)	1

2. Import the required libraries and set up the Bokeh backend.

We will be working with GeoViews once again. However, this time, we are going to use the Bokeh backend. Bokeh maps use the Mercator projection, so we will also need to import the `crs` module from Cartopy in order to project back into the coordinate system used by our data (Plate Carree projection):

In [2]:
```
from cartopy import crs
import geoviews as gv
import geoviews.feature as gf

gv.extension('bokeh')
```

3. Create an overlay with tooltips and a slider.

We will start by creating our points and specifying their ranges:

In [3]:
```
points = gv.Points(
    earthquakes,
    kdims=['longitude', 'latitude'],
    vdims=['month', 'place', 'tsunami', 'mag', 'magType']
)

# set colorbar limits for magnitude and axis limits
points = points.redim.range(
    mag=(-2, 10), longitude=(-180, 180), latitude=(-90, 90)
)
```

Next, we will create an overlay with a slider for the month:

In [4]:
```
overlay = gf.land * gf.coastline * gf.borders * points.groupby('month')
```

Finally, we customize each of the components of our plot, adding the option to hover over the points to trigger a tooltip:

In [5]:
```
interactive_map = overlay.opts(
    gv.opts.Feature(projection=crs.PlateCarree()),
    gv.opts.Overlay(width=700, height=450),
    gv.opts.Points(color='mag', cmap='fire_r', colorbar=True, tools=['hover'])
)
```

4. Render the visualization.

While we could use the `hv.output()` function to render our visualization, we will use Panel for this example. Panel, which is also part of HoloViz, provides additional functionality and flexibility when it comes to the layout. As you create more complex visualizations, Panel will become a necessity:

In [6]:
```
import panel as pn

earthquake_viz = pn.panel(interactive_map, widget_location='bottom')
```

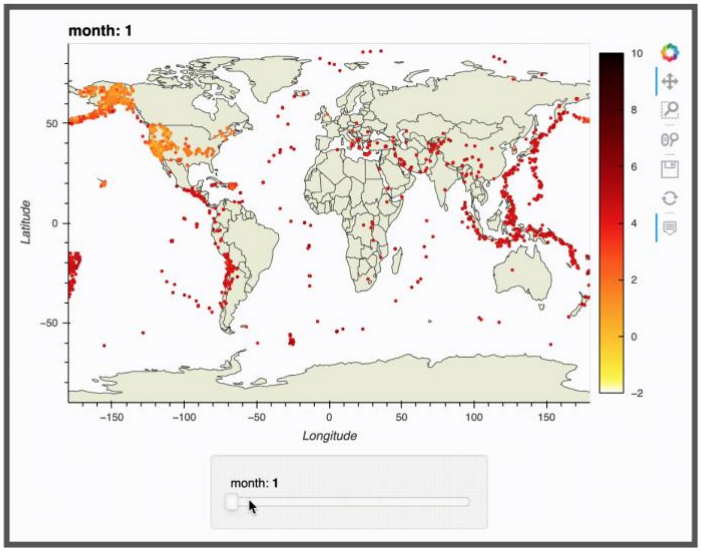
Try using the slider, tooltips, and zoom/pan functionality:

In [7]:
```
earthquake_viz.embed()
```

Out[7]:

Tip: Whenever we interact with the visualization, it has to determine what to update, which can be slow if the JavaScript visualization has to work with the Python backend. To get around this, you can use Panel to embed the visualization like we are doing above. Be careful though – this can make your notebook file size much larger. Another option is to look into the Datashader library from HoloViz.

The interactivity works best in a notebook environment – here's an example of the slider, tooltips, and zoom/pan functionality in action:



Linking plots

In the previous example, we saw that we could link together a slider and a plot. We can also link together plots, which makes using interactivity to explore our data even more powerful. For this example, we will create a link between a map of the earthquakes in January 2020 and a table of those same earthquakes that provides some additional information; we will be able to select earthquakes on the map and use that to filter our dataset. To further explore HoloViz, we will use the hvPlot library here; hvPlot makes it easy to build interactive visualizations with syntax similar to plotting in pandas.

We will work through the following steps to build this visualization:

1. Isolate the January earthquake data and prepare it for plotting.
2. Enable the use of hvPlot for interactive plotting with pandas.
3. Build a layout composed of an interactive map and a table with hvPlot.
4. Link selections across the visualizations in the layout.

1. Isolate the January earthquake data and prepare it for plotting.

Let's filter our dataset down to just January and then pull out the latitude and longitude information for our plot:

In [8]:
```
january_earthquakes = earthquakes.query('month == 1').assign(
    longitude=lambda x: x.geometry.x,
    latitude=lambda x: x.geometry.y
).drop(columns=['month', 'geometry'])
```

2. Enable the use of hvPlot for interactive plotting with pandas.

To enable interactive plotting with pandas, we have to import the following:

In [9]:
```
import hvplot.pandas
```

Important: While hvPlot is using HoloViews and GeoViews for the plotting logic, there is currently a bug with this feature in GeoViews; however, we can still put together a working example using hvPlot since the projections are handled differently.

3. Build a layout composed of an interactive map and a table with hvPlot.

Plotting with hvPlot works just like plotting with pandas – instead of calling the `plot()` method, we now call `hvplot()` to switch from static plots to interactive ones with the Bokeh backend. In doing so, hvPlot will take care of the HoloViews and GeoViews code for us. Here, we make the interactive map using tiles, which makes it possible to zoom in on the map and see more detail:

In [10]:
```
geo = january_earthquakes.hvplot(
    x='longitude', y='latitude', kind='points',
    color='mag', cmap='fire_r', clim=(-2, 10),
    tiles='CartoLight', geo=True, global_extent=True,
    xlabel='Longitude', ylabel='Latitude', title='January 2020 Earthquakes',
    frame_height=450
)
```

Next, we create the table by once again calling the `hvplot()` method:

In [11]:
```
table = january_earthquakes.sort_values(['longitude', 'latitude']).hvplot(
    kind='table', width=650, height=450, title='Raw Data'
)
```

Now, we create a layout with the map and table:

In [12]:
```
layout = geo + table
```

4. Link selections across the visualizations in the layout.

With our layout, we have everything we need to compose our visualization – we just need to link the components together. Here, we are creating an instance, so that we can use it to filter our data after interacting with the visualization:

In [13]:
```
import holoviews as hv

selection = hv.link_selections.instance()
map_and_table_tabs = selection(layout).opts(tabs=True)
```

Let's take a look at our visualization now. Try using the **Box Select** tool to select some earthquakes on the map and then take a look at the **Raw Data** tab:

In [14]:
```
map_and_table_tabs
```

Out[14]:

The result can be interacted with after displaying it, but this kind of interactivity only works in the notebook. Here's an example:

Using the selection from the visualization, we can filter our dataset as follows:

In [15]:
```
selection.filter(january_earthquakes).nlargest(3, 'mag')
```

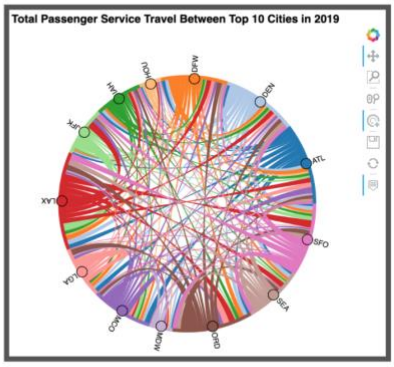
Out[15]:

	mag	place	time	tsunami	magType	longitude	latitude
16362	5.1	270 km SE of Chiniak, Alaska	2020-01-31 11:25:37.262	1	mww	-149.3295	55.7981
911	5.0	217 km SSE of Old Harbor, Alaska	2020-01-02 08:54:33.083	1	mww	-151.4274	55.5493
7831	4.3	258 km SE of Chiniak, Alaska	2020-01-13 09:00:21.044	0	mb	-149.3261	55.9471

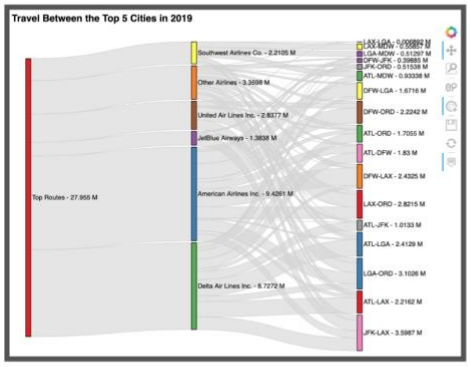
Note: Selecting something other than what is shown in the screen recording will yield different results.

Additional plot types

So far, we've seen how easy it is to make interactive visualizations with the Bokeh backend, but another benefit of using HoloViz is the ability to easily make a variety of plots that may require significant effort to create from scratch (e.g., network/graph diagrams, heatmaps, chord diagrams, and Sankey plots). In this section, we will see how to create a chord diagram and a Sankey plot in just a few lines of code using the HoloViews library directly.



Chord diagram



Sankey plot

We will be working with a new dataset for these examples: 2019 flight statistics from the United States Department of Transportation's Bureau of Transportation Statistics. The dataset contains 321,409 rows and 41 columns. Here, we read it in and perform some initial processing on it for our visualizations:

In [16]:
```
import numpy as np

flight_stats = pd.read_csv(
    '../data/T100_MARKET_ALL_CARRIER.zip',
    usecols=[
        'CLASS', 'REGION', 'UNIQUE_CARRIER_NAME', 'ORIGIN_CITY_NAME', 'ORIGIN',
        'DEST_CITY_NAME', 'DEST', 'PASSENGERS', 'FREIGHT', 'MAIL'
    ]
).rename(lambda x: x.lower(), axis=1).assign(
    region=lambda x: x.region.replace({
        'D': 'Domestic', 'I': 'International', 'A': 'Atlantic',
        'L': 'Latin America', 'P': 'Pacific', 'S': 'System'
    }),
    route=lambda x: np.where(
        x.origin < x.dest,
        x.origin + '-' + x.dest,
        x.dest + '-' + x.origin
    )
)
```

Our dataset looks like this:

In [17]:
```
flight_stats.head()
```

Out[17]:

	passengers	freight	mail	unique_carrier_name	region	origin	origin_city_name	dest	dest_city_name	class	route
0	0.0	53185.0	0.0	Emirates	International	DXB	Dubai, United Arab Emirates	IAH	Houston, TX	G	DXB-IAH
1	0.0	9002.0	0.0	Emirates	International	DXB	Dubai, United Arab Emirates	JFK	New York, NY	G	DXB-JFK
2	0.0	2220750.0	0.0	Emirates	International	DXB	Dubai, United Arab Emirates	ORD	Chicago, IL	G	DXB-ORD
3	0.0	1201490.0	0.0	Emirates	International	IAH	Houston, TX	DXB	Dubai, United Arab Emirates	G	DXB-IAH
4	0.0	248642.0	0.0	Emirates	International	JFK	New York, NY	DXB	Dubai, United Arab Emirates	G	DXB-JFK

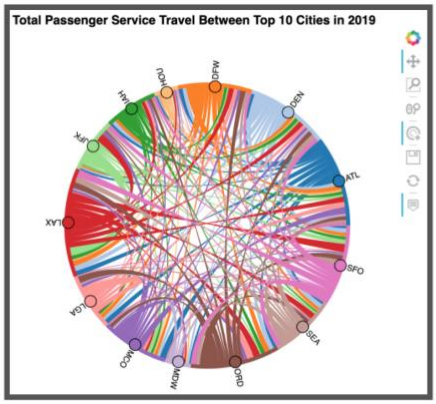
Source: T-100 Market (All Carriers) dataset provided by the United States Bureau of Transportation Statistics.

This dataset only includes travel to/from the US, so as a starting point for our analysis, we will only consider travel to/from the top 10 cities by passenger counts and – for the Sankey plot only – the top 5 airlines in the US as found in this blog post, which analyzes this dataset:

In [18]:
```
cities = [
    'Atlanta, GA', 'Chicago, IL', 'New York, NY', 'Los Angeles, CA',
    'Dallas/Fort Worth, TX', 'Denver, CO', 'Houston, TX',
    'San Francisco, CA', 'Seattle, WA', 'Orlando, FL'
]

top_airlines = [
    'American Airlines Inc.', 'Delta Air Lines Inc.', 'JetBlue Airways',
    'Southwest Airlines Co.', 'United Air Lines Inc.'
]
```

Chord diagram

A **chord diagram** is a way of showing many-to-many relationships between a set of entities called **nodes**: the nodes are arranged in a circle, and chords (which can be thought of as **edges**) are drawn between those that are connected, with the width of the chord encoding the strength of the connection. In this section, we will be making a chord diagram for total passenger service travel between the top 10 cities in 2019:



Let's work through these steps to create the chord diagram:

1. Aggregate the dataset to get total flight statistics between the top cities.
2. Create a chord diagram with HoloViews.
3. Customize the tooltips using Bokeh.

1. Aggregate the dataset to get total flight statistics between the top cities.

Our dataset contains flights that aren't considered passenger service, so we will need to filter to just passenger service between the cities in our list. After that, we are grouping by both the city and airport for each point of the trip (origin and destination) because some cities have multiple airports. Finally, we calculate the total number of passengers and pounds of mail/freight transported in 2019. Note that we are limiting the result to rows with total passengers greater than zero since our chord diagram will use this column to draw the chords:

In [19]:
```
total_flight_stats = flight_stats.query(
    f'`class` == "F" and origin_city_name != dest_city_name'
    f' and origin_city_name.isin({cities}) and dest_city_name.isin({cities})'
).groupby([
    'origin', 'origin_city_name', 'dest', 'dest_city_name'
])[['passengers', 'freight', 'mail']].sum().reset_index().query('passengers > 0')
```

Our aggregated dataset looks like this:

In [20]:
```
total_flight_stats.sample(10, random_state=1)
```

Out[20]:

	origin	origin_city_name	dest	dest_city_name	passengers	freight	mail
78	LGA	New York, NY	DEN	Denver, CO	589190.0	506023.0	293108.0
117	ORD	Chicago, IL	SEA	Seattle, WA	810594.0	1063463.0	2627325.0
31	DFW	Dallas/Fort Worth, TX	MCO	Orlando, FL	683700.0	187672.0	95570.0
5	ATL	Atlanta, GA	LAX	Los Angeles, CA	1121378.0	8707125.0	3267077.0
126	SEA	Seattle, WA	LGA	New York, NY	24.0	0.0	0.0
45	IAH	Houston, TX	ATL	Atlanta, GA	566369.0	367543.0	726670.0
14	DEN	Denver, CO	HOU	Houston, TX	305193.0	363119.0	0.0
44	HOU	Houston, TX	SFO	San Francisco, CA	1843.0	5523.0	0.0
73	LAX	Los Angeles, CA	MDW	Chicago, IL	277226.0	2022416.0	0.0
89	MCO	Orlando, FL	DEN	Denver, CO	594878.0	368516.0	138811.0

2. Create a chord diagram with HoloViews.

Next, we create an instance of `hv.Chord` by specifying that the paths are between the `origin` and `dest` columns (which are not the city names, but rather the airport codes) and that the remaining values associated with each origin-destination pair should be used as value dimensions. Note that only the first value dimension will be used to size the chords, but the rest will be accessible in the tooltip:

In [21]:
```
chord = hv.Chord(
    total_flight_stats,
    kdims=['origin', 'dest'],
    vdims=['passengers', 'origin_city_name', 'dest_city_name', 'mail', 'freight']
)
```

3. Customize the tooltips using Bokeh.

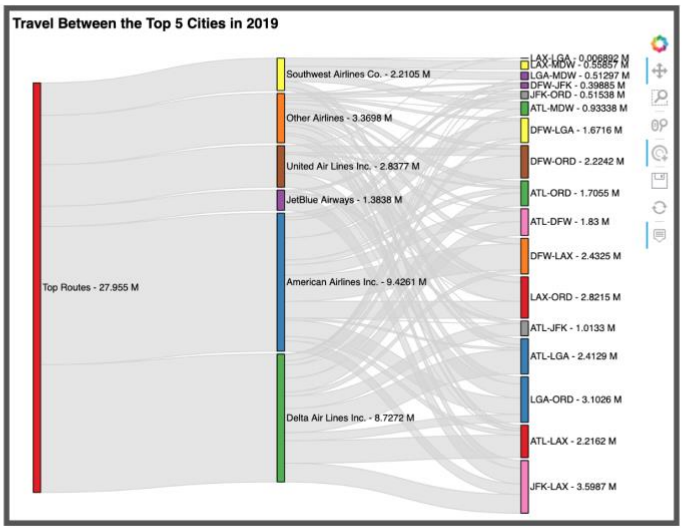
Our dataset contains large numbers, which can be hard to read in tooltips without formatting. In addition, the default tooltip is rather long since it lists all of the columns we provided as `kdims` and `vdims`. To improve usability of the tooltips, we should combine the city and airport information into a single line for each source/destination since those fields are related (e.g., `Chicago, IL (ORD)`). While this functionality is possible, we will have to use Bokeh directly to achieve it. Here, we instantiate an instance of Bokeh's `HoverTool` with our desired tooltip format:

In [22]:
```
from bokeh.models import HoverTool

tooltips = {
    'Source': '@origin_city_name (@origin)',
    'Target': '@dest_city_name (@dest)',
    'Passengers': '@passengers{0,.}',
    'Mail': '@mail{0,.} lbs.',
    'Freight': '@freight{0,.} lbs.',
}
hover = HoverTool(tooltips=tooltips)
```

Tip: Check out the Bokeh `HoverTool` documentation here for more information on the hover tool itself and here for more information on creating and formatting tooltips.

Sankey plot

For our final visualization, we will create a **Sankey plot**, which is a way to visualize flow as edges between nodes. Here, we will use it to analyze airline market share for passenger service flights between the top 5 US cities:



To build this visualization, we will work through the following steps:

1. Isolate flight statistics for top routes.
2. Convert the data into a set of edges.
3. Create the Sankey plot with HoloViews.

1. Isolate flight statistics for top routes.

We need to filter our data to just domestic passenger service between the top 5 cities. Since we want to look at market share, we need to keep information for all airlines (i.e., we can't filter to the top airlines yet):

In [25]:
```
top_cities = cities[:5]

domestic_passenger_travel = flight_stats.query(
    'region == "Domestic" and `class` == "F" and origin_city_name != dest_city_name '
    f'and origin_city_name.isin({top_cities}) and dest_city_name.isin({top_cities})'
).groupby([
    'region', 'unique_carrier_name', 'route',
    'origin_city_name', 'dest_city_name'
]).passengers.sum().reset_index()

domestic_passenger_travel.head()
```

Out[25]:

	region	unique_carrier_name	route	origin_city_name	dest_city_name	passengers
0	Domestic	Air Wisconsin Airlines Corp	ATL-ORD	Atlanta, GA	Chicago, IL	915.0
1	Domestic	Air Wisconsin Airlines Corp	ATL-ORD	Chicago, IL	Atlanta, GA	556.0
2	Domestic	Alaska Airlines Inc.	JFK-LAX	Los Angeles, CA	New York, NY	265307.0
3	Domestic	Alaska Airlines Inc.	JFK-LAX	New York, NY	Los Angeles, CA	257685.0
4	Domestic	Alaska Airlines Inc.	LAX-ORD	Chicago, IL	Los Angeles, CA	48269.0

Note: In reality, all the routes we are considering are domestic, but we are keeping the `region` column because it will serve as the basis for a root node in our Sankey plot, which allows us to easily see the total across airlines.

2. Convert the data into a set of edges.

The trickiest part of building this visualization is unraveling our dataset into a set of edges: a Sankey plot can be used to represent a directed, acyclic graph (DAG), meaning that we have to be careful there are no cycles (loops) when compiling our edge list.

We will be making two sets of edges for our Sankey plot: one set from region to airline and another from airline to route. Note that there is more data than we can display in the plot, so we have to group together any airlines that aren't in the top 5 and restrict to only routes between the top 5 cities.

Let's start by grouping all airlines outside the top 5 into a new airline called "Other Airlines" – this is necessary to keep our Sankey plot a manageable size:

In [26]:
```
domestic_passenger_travel.unique_carrier_name.replace(
    '^(?!' + '|'.join(top_airlines) + ').*$',
    'Other Airlines',
    regex=True, inplace=True
)
```

Our top 5 airlines combined have close to 88% market share of travel between the top 5 cities:

In [27]:
```
domestic_passenger_travel.groupby('unique_carrier_name').passengers.sum().div(
    domestic_passenger_travel.passengers.sum()
)
```

Out[27]:
```
unique_carrier_name
American Airlines Inc.    0.337186
Delta Air Lines Inc.      0.312187
JetBlue Airways           0.049500
Other Airlines            0.120544
Southwest Airlines Co.    0.079074
United Air Lines Inc.     0.101509
Name: passengers, dtype: float64
```

Next, we will define a function for converting a DataFrame into edges:

In [28]:
```
def get_edges(data, *, source_col, target_col):
    aggregated = data.groupby([source_col, target_col]).passengers.sum()
    return aggregated.reset_index().rename(
        columns={source_col: 'source', target_col: 'target'}
    ).query('passengers > 0')
```

Recall: The asterisk in the function signature requires both `source_col` and `target_col` to be passed in by name. This makes sure that we explicitly define the direction for the edges when calling the function. Read more on this syntax here.

Let's use our function to get our first set of edges going from region to airline. Here, we will also rename the node "Domestic" to "Top Routes" for a more descriptive name for the root node of our Sankey plot:

In [29]:
```
carrier_edges = get_edges(
    domestic_passenger_travel,
    source_col='region',
    target_col='unique_carrier_name'
).replace('Domestic', 'Top Routes')

carrier_edges
```

Out[29]:

	source	target	passengers
0	Top Routes	American Airlines Inc.	9426060.0
1	Top Routes	Delta Air Lines Inc.	8727210.0
2	Top Routes	JetBlue Airways	1383776.0
3	Top Routes	Other Airlines	3369815.0
4	Top Routes	Southwest Airlines Co.	2210533.0
5	Top Routes	United Air Lines Inc.	2837682.0

The other set of edges that we need is from airline to route for routes between the top cities:

In [30]:
```
carrier_to_route_edges = get_edges(
    domestic_passenger_travel,
    source_col='unique_carrier_name',
    target_col='route'
)

carrier_to_route_edges.sample(10, random_state=1)
```

Out[30]:

	source	target	passengers
39	Other Airlines	DFW-LGA	157366.0
41	Other Airlines	JFK-LAX	523222.0
2	American Airlines Inc.	ATL-LAX	294304.0
48	Southwest Airlines Co.	ATL-MDW	498481.0
50	Southwest Airlines Co.	LAX-MDW	558574.0
44	Other Airlines	LAX-ORD	378552.0
33	Other Airlines	ATL-LAX	146882.0
35	Other Airlines	ATL-MDW	1201.0
40	Other Airlines	DFW-ORD	241147.0
27	JetBlue Airways	DFW-JFK	140.0

Let's combine our edges into a single DataFrame now; we will also convert the total passengers number to millions for display purposes:

In [31]:
```
all_edges = pd.concat([carrier_edges, carrier_to_route_edges]).assign(
    passengers=lambda x: x.passengers / 1e6
)
```

4. Create the Sankey plot with HoloViews.

As with the chord diagram, our key dimensions are the source and target of the edges. However, this time, we will only provide the passenger total as the value dimension – note that we are able to specify that the values are in millions by using `hv.Dimension`:

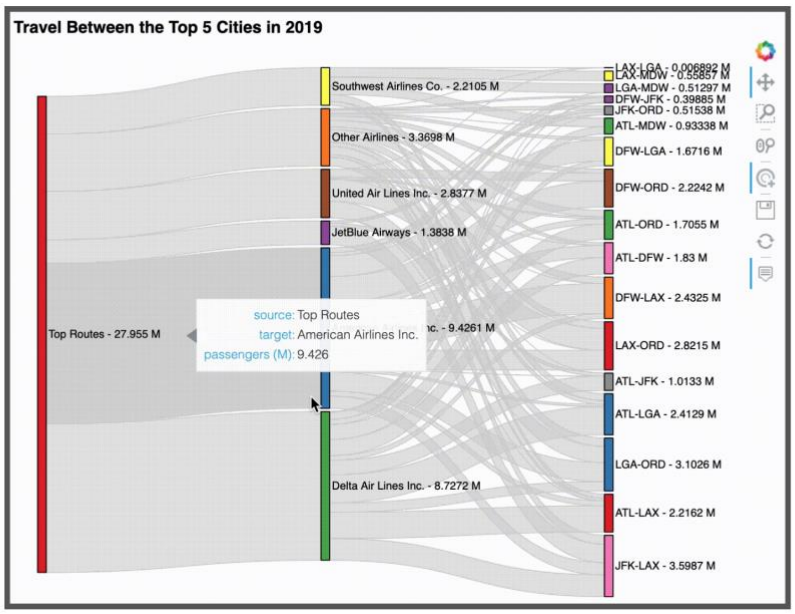
In [32]:
```
sankey = hv.Sankey(
    all_edges,
    kdims=['source', 'target'],
    vdims=hv.Dimension('passengers', unit='M')
).opts(
    labels='index', label_position='right', cmap='Set1', # node config
    edge_color='lightgray', # edge config
    width=750, height=600, # plot size config
    title='Travel Between the Top 5 Cities in 2019'
)
```

Hover over the edges to explore the data in this Sankey plot:

In [33]:
```
sankey
```

Out[33]:

The resulting visualization can be interacted with after displaying it, but it works best in the notebook. Here's an example:



Exercises

This section provided an introduction to interactive plotting with HoloViz using the Bokeh backend. We saw that the hvPlot interface is very similar to how we plot with pandas, so it is a great way to get used to HoloViz. Let's take a break for some exercises using hvPlot to check your understanding:

1. For the 10 carriers that transported the most freight, create a bar plot showing total freight transported per carrier.

In [ ]:

2. Create a line plot of total earthquakes per day with tooltips.

In [ ]:

3. Make histograms of earthquake magnitude (`mag`) for each magnitude type (`magType`) with a dropdown to select the magnitude type.

In [ ]:

REFERENCES

Molin, S. (2019). *Hands-On Data Analysis with Pandas: Efficiently perform data collection, wrangling, analysis, and visualization using Python*. Packt Publishing Ltd.