





## [Creating Interactive Visualizations with Plotly](#) [\(/en/lessons/interactive-visualization-with-plotly\)](#)

Grace Di Méo  (<https://orcid.org/0000-0002-3227-8053>).

This lesson demonstrates how to create interactive data visualizations in Python with Plotly's open-source graphing libraries using materials from the Historical Violence Database.

 Peer-reviewed (<https://github.com/programminghistorian/ph-submissions/issues/518>).

 CC-BY 4.0 (<https://creativecommons.org/licenses/by/4.0/deed.en>).

 [Support PH \(/en/individual\)](#).

### edited by

- Scott Kleinman  (<https://orcid.org/0000-0001-7477-1308>).

### reviewed by

- Mario Bañuelos
- Rob Lewis

### published


| 2023-12-13

### modified

| 2024-05-15

### difficulty

| Medium

 <https://doi.org/10.46430/phen0115>

## Contents🔗

- Introduction
  - Lesson Goals
  - Prerequisites
  - What is Plotly?
  - Plotly's Python Graphing Library: Plotly Express vs. Plotly Graph Objects vs. Plotly Dash
  - Why Plotly?
- Sample Dataset
- Building Graphs with Plotly Express
  - Setting Up Plotly Express
  - Importing and Cleaning Data
  - Bar Charts
  - Line Graphs
  - Scatterplots
  - Facet Plots
  - Adding Animations: Animation Frames
  - Adding Animations: Dropdown Bars
- Building Graphs with Plotly Graph Objects
  - Setting Up Plotly Graph Objects
  - They're All Objects! Data Structure of Plotly Graph Objects
  - Using Plotly Graph Objects vs. Plotly Express
  - Why Use Graph Objects?
  - Tables
  - Subplots
    - Step 1: Import the subplots module and get data
    - Step 2: Create an empty subplot with a 3x1 grid using the `make_subplots()` function
    - Step 3: Add the first graph (the bar chart) using the `.add_trace()` method
    - Step 4: Add the second graph (the line graph)
    - Step 5: Add the final graph (the boxplot)
    - Step 6: Format the figure
    - Step 7: Add annotations to the line graph
    - Step 8: Add annotations below the figure
- Viewing and Exporting Figures
  - Viewing Figures
  - Exporting Figures
    - Exporting as HTML
    - Exporting static images
- Summary
- Endnotes

## Introduction🔗

### Lesson Goals🔗

This lesson demonstrates how to create interactive data visualizations in Python using Plotly's open-source graphing libraries (<https://perma.cc/94J3-8LAS>). In particular, you will learn:

- The distinction between Plotly Express, Plotly's Graph Objects, and Plotly Dash
- How to create and export graphs using `plotly.express` and `plotly.graph_objects`
- How to add custom features to graphs

## Prerequisites🔗

In order to follow this lesson, it is assumed that you have:

- Installed [Python 3](https://www.python.org/downloads/) and the [pip package installer](https://pypi.org/project/pip/)
- An intermediate level understanding of the Python programming language
- Some familiarity with [pandas](https://perma.cc/UT9Y-KR76) and [NumPy](https://perma.cc/JQW9-AM8Y), which should also be installed
- Knowledge of basic data visualization techniques (especially bar charts, histograms and scatterplots)
- Some familiarity with data preprocessing (we will be using pandas)

This lesson was developed using Jupyter Notebook. For those who are unfamiliar with this software, the *Programming Historian* offers an excellent lesson on how to create, edit and export Jupyter notebooks [here](/en/lessons/jupyter-notebooks). You may also follow this lesson using your own preferred code editor (VSCode, PyCharm, etc.).

## What is Plotly?🔗

[Plotly](https://plotly.com/) is a company which provides a number of open-source libraries allowing users to build interactive graphs. Unlike static images, Plotly graphs can respond to user actions with popup labels, pan-and-zoom abilities, faceted data displays, and more. Plotly libraries are available in Python — the focus of this tutorial — as well as various programming languages, including R and Julia.<sup>1</sup> A wide variety of graphs is available through Plotly libraries, ranging from the statistical or scientific to the financial or geographic. These graphs can be displayed using various methods, including Jupyter notebooks, HTML files, and web applications produced with Plotly's Dash framework. Static (non-interactive) graphs can also be exported either as raster or vector images.

**Throughout this lesson, Plotly graphs will be displayed as static images. To access their interactive features, either click on the image or the link in the caption below the image.**

## Plotly's Python Graphing Library: Plotly Express vs. Plotly Graph Objects vs. Plotly Dash🔗

To understand how to use Plotly, it is vital to understand the differences between Plotly Express, Plotly Graph Objects, and Plotly Dash.

Essentially, these are three distinct — but often overlapping — Plotly modules with their own use cases:

- Plotly Express ( `plotly.express` , usually imported as `px` ) is an accessible, high-level interface for creating data visualizations, offering around 30

different graph types. The module provides functions which create figures in just one line of code (although more lines are required for certain customizations), making graphs quick and easy to create. Since this is a 'high-level' interface, users do not need to interact with the underlying data structure of graphs when using `plotly.px`. Plotly recommends that new users start with Express before working directly with Plotly Graph Objects.

- Plotly Graph Objects (`plotly.graph_objects`, usually imported as `go`) are the actual figures created and rendered by Plotly 'under the hood': in essence, when a user creates a figure in `plotly.px`, Plotly will generate a 'Graph Object' to store the graph's data. These data include the information visualized in the graph as well as various attributes such as graph colors, sizes, and shapes. It is therefore possible to create visualizations with the lower-level `plotly.go` module; in fact, it is possible to recreate anything made with `plotly.px` using `plotly.go`. It is generally advised to use `plotly.px` where possible, since using `plotly.go` often involves generating many lines of code. However, as we will see later, there are some specific use cases for `plotly.go`.
- The Plotly Dash module (imported as `dash`) is a framework for building interactive web applications (typically dashboards) which can be embedded into websites and other platforms. Users often integrate figures created using `plotly.px` and/or `plotly.go` into their Dash apps, making the Plotly Python stack a full suite for creating, manipulating, and publishing interactive data visualizations. Plotly Dash is built on top of `React.js` and `Plotly.js` to enable integration with the web, meaning that users do not need to have any knowledge of Javascript, CSS or HTML (only Python).<sup>2</sup>

Plotly provides comprehensive documentation for working with [Express and Graph Objects \(https://perma.cc/94J3-8LAS\)](https://perma.cc/94J3-8LAS) and for using [Dash \(https://perma.cc/E7S3-6W3H\)](https://perma.cc/E7S3-6W3H).

## Why Plotly?<sup>2</sup>

There are currently a plethora of graphing libraries available to Python users, including [Matplotlib \(https://perma.cc/N3Y9-CPWN\)](https://perma.cc/N3Y9-CPWN), [Seaborn \(https://perma.cc/3AQL-5H6C\)](https://perma.cc/3AQL-5H6C), [Bokeh \(https://perma.cc/L2MP-43BL\)](https://perma.cc/L2MP-43BL) and [Pygal \(https://perma.cc/UZM9-2LBE\)](https://perma.cc/UZM9-2LBE). With so many options to choose from, users will need to select one library over another. Factors such as use case, stylistic taste, and ease of use will be important here, with each library having its own merits. Some of the notable advantages of working with Plotly include:

- Plotly is one of the only packages to be directed at interactive graphs: options such as Matplotlib and Pygal provide only limited interactivity (although Bokeh is also designed for interactivity and is a viable alternative)<sup>3</sup>
- Plotly is the only Python graphing suite which facilitates both the creation of graphs and the integration of these graphs within web apps
- Plotly has easy (seamless) integration with pandas (for example, DataFrames can be added directly into graph objects)
- Interactive 3D graphs are available (typically not available in other libraries)
- Plotly is simple to use (adding features like animation and dropdown bars is relatively easy)

## Sample Dataset

The dataset for this lesson is a subset of Roger Lane's '[Homicides in Philadelphia, 1839-1932](https://perma.cc/93XS-LFG4)' dataset (https://perma.cc/93XS-LFG4)<sup>4</sup>, covering only the years 1902-1932. If you wish to work along with this lesson, you can [download this particular dataset directly \(/assets/interactive-visualization-with-plotly/sample-dataset-philadelphia-homicides-1902-1932.csv\)](/assets/interactive-visualization-with-plotly/sample-dataset-philadelphia-homicides-1902-1932.csv). (The corresponding 'Philadelphia homicide codebook 1902-1932' document was also consulted to verify variables in the dataset but is not required for completing this lesson.) As its title suggests, the dataset records homicides which occurred in Philadelphia in the early twentieth century. It is informed by arrest reports filed by the Philadelphia police and covers the years 1902, 1908, 1914, 1920, 1926, and 1932. In its downloaded format, the dataset contains 26 columns and 717 rows, although we will be scaling this down.

## Building Graphs with Plotly Express

### Setting Up Plotly Express

1. Before starting, you will need to install three modules to your environment:<sup>5</sup>
  - Plotly (using the terminal command `pip install plotly`)
  - pandas (using the terminal command `pip install pandas`)<sup>6</sup>
  - Kaleido (using the terminal command `pip install kaleido`)<sup>7</sup>
2. With these packages installed, create a new Jupyter notebook (or a new Python file in your chosen code editor). Ideally, your notebook should be stored in the same folder as the downloaded sample dataset.
3. `import` these modules at the start of your new notebook:

```
import numpy as np
import pandas as pd
import plotly.express as px
```

### Importing and Cleaning Data

Next, we will import and clean the Philadelphia homicide dataset using pandas. This will involve:

- Importing only the required columns from our dataset
- Replacing any missing numeric values as a NumPy 'non-number' (the `NaN` data type)
- Relabeling and removing certain data points for clarity and accuracy

```

# Import data as DataFrame (only the columns specified under 'fields'
list will be kept)
fields = [
    "Year",
    "Charge",
    "Gender of accused",
    "Age of accused",
    "Victim age",
    "Weapon",
    "Gang",
]

# Store csv data as DataFrame, keeping specified columns with
'usecols' parameter
phl_crime = pd.read_csv("philadelphia homicides 1902-1932 5-
2004.csv", usecols=fields)

# Replace missing numeric values (classified in original dataset as
'99') to NumPy NaN
phl_crime.replace(99, np.NaN, inplace=True)

# Drop rows with where crime type is unknown (those classified as
either '4' or '9' under the 'Charge' column)
phl_crime = phl_crime.drop(
    phl_crime[(phl_crime.Charge == 4) | (phl_crime.Charge ==
9)].index
)

# Re-label crime types (under "Charge" column) as nouns (originally
coded numerically)
phl_crime["Charge"].replace(
    {1: "Murder", 2: "Manslaughter", 3: "Abortion"}, inplace=True
)

# Re-label gender (under "Gender of accused" column) as nouns
(originally coded numerically)
phl_crime["Gender of accused"].replace(
    {1: "Male", 2: "Female", 3: np.NaN, 9: np.NaN}, inplace=True
)

# Replace erroneous data and typos in "Year" column
phl_crime["Year"].replace(
    {1514: 1914, 1520: 1920, 1526: 1926, 1532: 1932, 1915: 1914},
inplace=True
)

# Re-label gang affiliation (under "Gang" column) as nouns (originally
coded numerically)
phl_crime["Gang"].replace({1: "No gang", 2: "Teen gang", 3: "Adult
gang"}, inplace=True)

# Re-label weapons (under "Weapon" column) as nouns (originally coded
numerically)
phl_crime["Weapon"].replace(
    {

```

```

1: "Gun",
2: "Knife, sharp instrument",
3: "Blunt object",
4: "Fist, other body part",
5: "Vehicle",
6: "Other",
7: "Poison",
9: np.NaN,
},
inplace=True,
)

```

## Bar Charts

Now that we've created a pandas `DataFrame` for our historical homicide data, we can start building some basic graphs using Plotly Express. Let's begin by creating a bar chart to represent the count of prosecutions per type of crime. Since our existing dataset does not contain counts of offences (each row represents one individual prosecution), we will first need to create a new `DataFrame` which groups cases by offence type and provides a count:

```

# Creates new DataFrame containing count of prosecutions per offence
type
phl_by_charge = phl_crime.groupby(["Charge"], as_index=False).size()

```

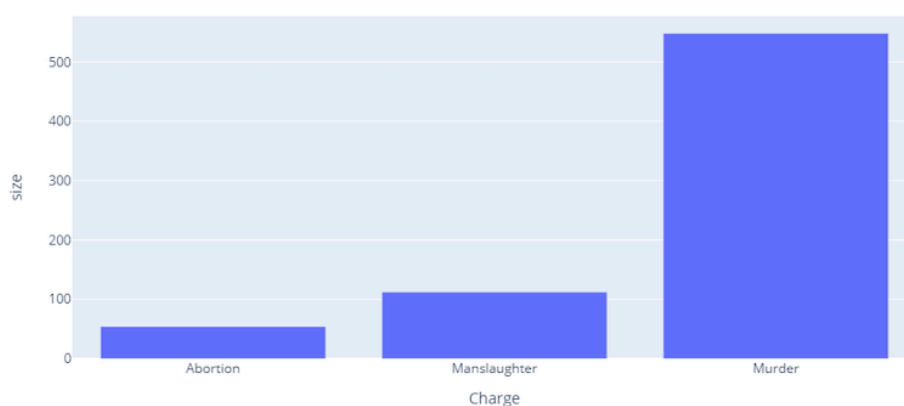
The next step is to create the bar chart using this new `phl_by_charge` dataset. Notice that the graph has been saved to the variable `fig`, which is a common convention when working with Plotly:

```

# Create bar chart using the .bar() method
fig = px.bar(phl_by_charge, x="Charge", y="size")

# Display figure using the .show() method
fig.show()

```



(</assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-01.html>)

Figure 1. Simple bar graph with basic interactivity created using Plotly Express. If readers hover over the bars, they will notice floating labels appear. [Click to explore an interactive variant of this plot](#) (</assets/interactive-visualization-with->

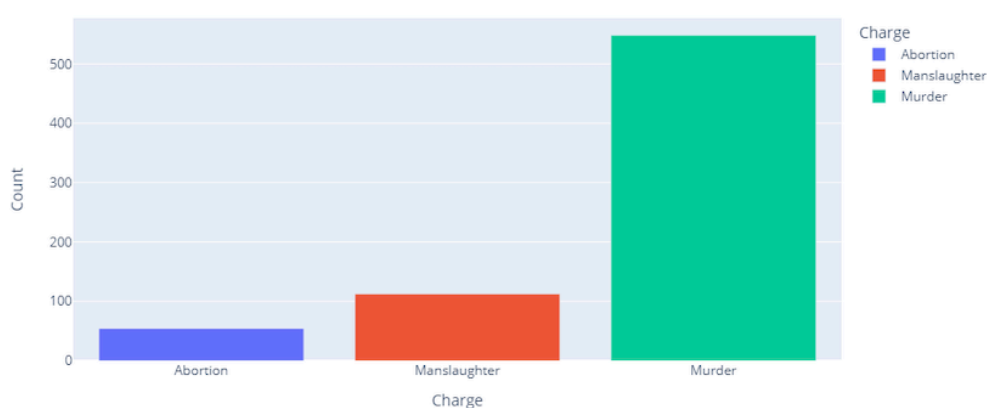
[plotly/interactive-visualization-with-plotly-01.html](#)).

So we have our first `px` graph! Notice that this graph already has some interactivity: hovering over each bar will specify its crime type and prosecution count. Another notable feature is that users can easily save this graph (as a static image) by navigating to the top-right corner and clicking on the camera icon to download the plot as a `.png` file. In this same corner, users have the option to zoom, pan, autoscale or reset their view of the plot. These features are available throughout all the following Plotly visualizations.

However, this isn't the most visually appealing graph: it could use a title, some colors, and a clearer y-axis label. We could have done this when we initially created the bar chart by passing additional arguments into the `.bar()` method. We can use the `labels` argument to change the y-axis labels from 'size' to 'Count' and the `color` argument to color the bars according to a given variable (in this example, we will use the crime type, 'Charge'). To add a title, uncomment the `title` argument in the code below and add a title of your choice.

```
# Create bar chart using the .bar() method (in a new code cell)
fig = px.bar(
    phl_by_charge,
    x="Charge",
    y="size",
    # title="Add a title here",
    labels={"size": "Count"},
    color="Charge", # Note that the 'color' parameter takes the name
of our column ('Charge') as a string
)

fig.show()
```



[\(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-02.html\)](#)

Figure 2. Simple bar graph with basic interactivity created using Plotly Express. This plot is a variant of that produced in Figure 1, now featuring color attributes as well as an interactive legend which allows readers to isolate or remove data points. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-02.html\)](#).



As demonstrated above, Plotly will automatically add a legend to the graph if you are dividing attributes by color (this can be avoided if desired). The legend is also interactive: clicking once on an element will remove its corresponding bar from the graph; double-clicking on an element will isolate all others.

## Line Graphs

Let's move on to creating a line graph. As a general rule, Plotly Express graphs are created using the syntax `px.somegraph()`, where `somegraph` represents the graph being created. While we used the syntax `px.bar()` to create a bar chart, we will use `px.line()` to create a line graph. The exact term needed for each graph type can be found via the [Plotly documentation \(https://perma.cc/U4N7-2VM5\)](https://perma.cc/U4N7-2VM5).

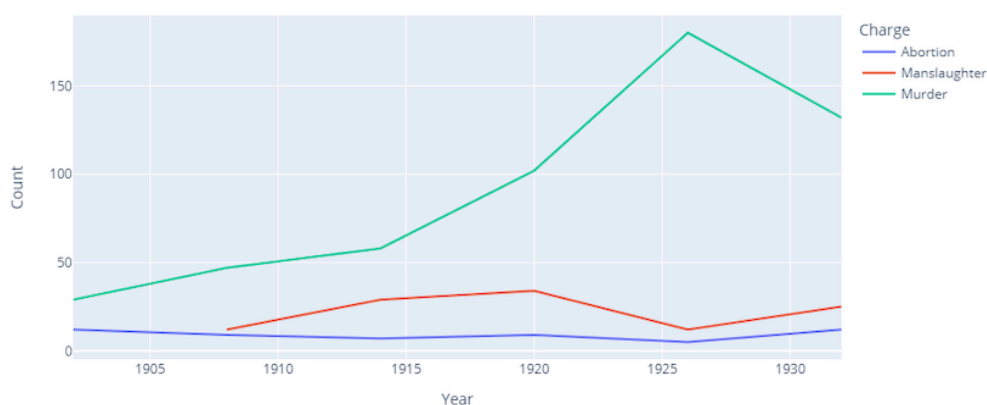
Our line graph will illustrate changes in prosecution rates per crime type over the research period. As before, we will need to create a new `DataFrame` which groups cases both by year and offence type:

```
# Creates new DataFrame containing counts of prosecutions per offence
type and year
phl_by_year = phl_crime.groupby(["Charge",
                                "Year"],as_index=False).size()
```

Next, we will create a line graph using the `.line()` method and will use the `labels` and `color` keywords to add some formatting. Again, uncomment the `title` argument in the code below (and in subsequent code samples) if you wish to add a title:

```
# Use px.line() to build line graph and add some formatting
fig = px.line(
    phl_by_year,
    x="Year",
    y="size",
    # title="Add a title here",
    labels={"size": "Count"},
    color="Charge",
)

fig.show()
```



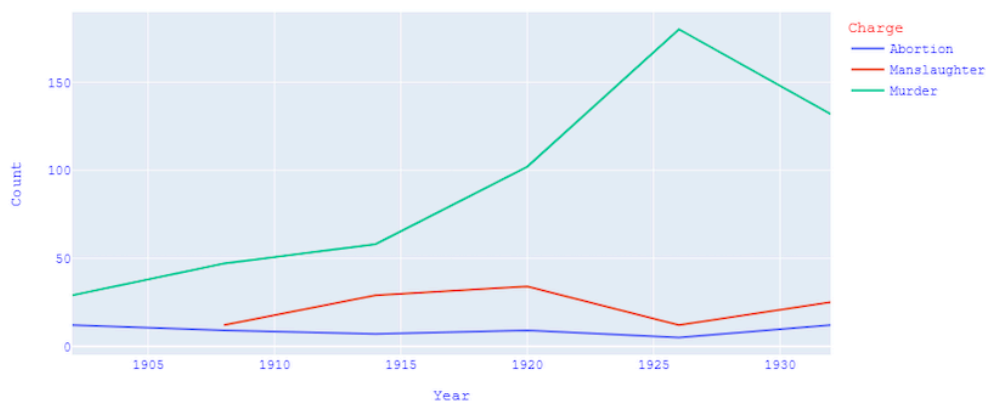
[03.html](#))

Figure 3. Simple line graph with basic interactivity created using Plotly Express. Hovering over the lines at plot points invokes a floating label. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-03.html\)](#).

We have now learned to create new graphs with some added formatting — but what if we wanted to add our formatting after creating the graph instead? We can use the `.update_layout()` method on our `fig` object to edit the graph at a later stage. This method can be applied to any Plotly Express graph and accepts a very wide range of adjustable parameters. As an example, let's use this method to update our font family, font colors, and the text of our title:

```
fig.update_layout(  
    font_family="Courier New", # Update font  
    font_color="blue", # Make font blue  
    legend_title_font_color="red", # Make legend title red  
    title="A formatted title!",  
)  
  
fig.show()
```

Fig. 4. A formatted title!



[\(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-04.html\)](#)

Figure 4. Simple line graph with basic interactivity created using Plotly Express. Hovering over the lines at plot points invokes a floating label. This plot is a variant of that produced in Figure 3, now featuring updated font, font colors, and an embedded figure title. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-04.html\)](#).

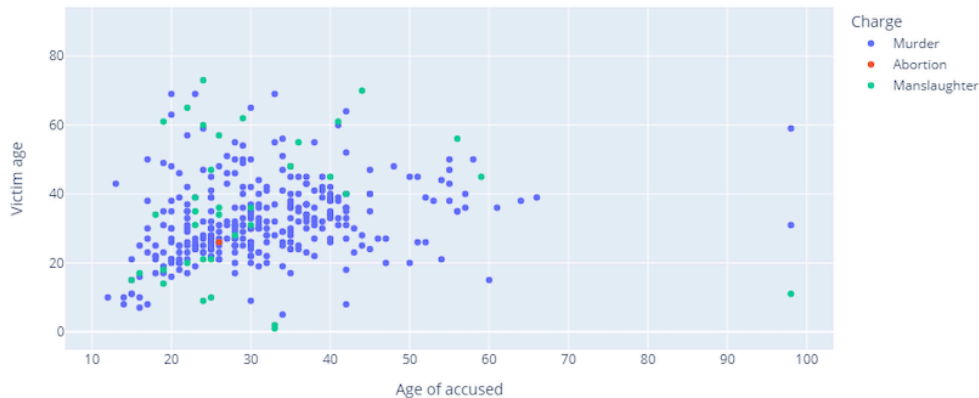
## Scatterplots

Scatterplots, commonly used for visualizing relationships between continuous variables, can be created with Plotly Express using the `.scatter()` method. For our sample dataset, it might be appropriate to use a scatterplot to depict the relationship between victim and assailant ages.

```

fig = px.scatter(
    phl_crime,
    x="Age of accused",
    y="Victim age",
    color="Charge", # Add
    # title="Add a title here",
)
fig.show()

```



(</assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-05.html>)

Figure 5. Simple scatterplot with basic interactivity created using Plotly Express. Readers can hover over individual plot points to invoke floating labels. Additionally, an interactive legend allows isolation, comparison or removal of data categories. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-05.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-05.html).

As you can see, scatterplots also contain some inherent interactivity: hovering over a unique data point will display the specific charge and the ages of both the accused and the victim. Clicking and double-clicking on the legend allows you to isolate certain elements.

## Facet Plots

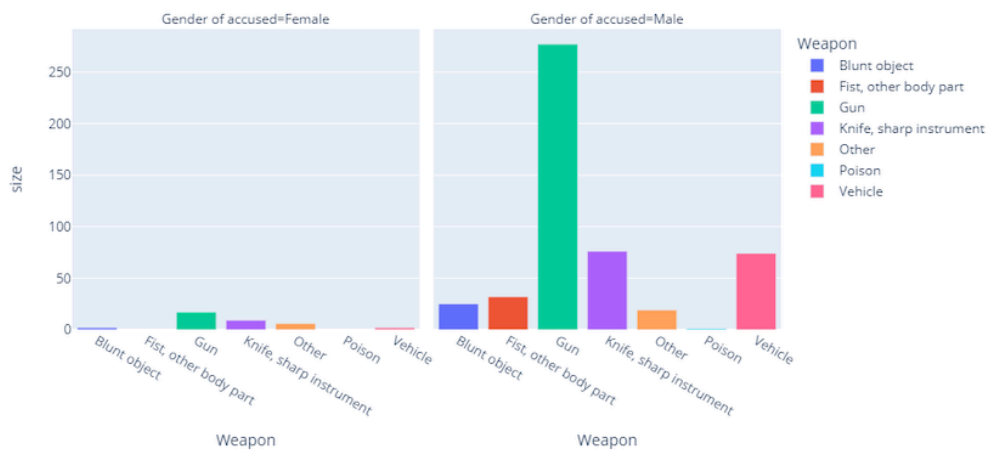
Facet plots are made up of the multiple subplots which a) share the same set of axes; and b) show a subset of the data (for the same set of variables). These can be made very easily in Plotly Express. First, using the same procedure as outlined in the above examples, you'll need to specify the type of graph you want to use in your subplots. Second, the `facet_col` parameter allows you to specify which variable to use for splitting the data into subplots. In the example below, a 2x1 grid of bar charts shows prosecution counts for each weapon type used by suspects in homicide cases. One bar chart provides counts for male suspects and the other for female suspects:

```

# Create DataFrame which groups by gender and weapon and adds a count
('size') column
ph1_by_weapon = ph1_crime.groupby(
    ["Weapon", "Gender of accused"], as_index=False
).size()

# Use px.bar() to indicate that we want to create bar charts
fig = px.bar(
    ph1_by_weapon,
    x="Weapon",
    y="size",
    facet_col="Gender of accused", # Use facet_col parameter to
specify which field to split graph by
    color="Weapon",
    # title="Add a title here",
)
fig.show()

```



[\(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-06.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-06.html)

Figure 6. Two bar graph subplots with basic interactivity created using Plotly Express, separating the data between the two genders. Readers can hover over the bars to invoke floating labels. An interactive legend allows isolation, comparison or removal of data categories. [Click to explore an interactive variant of this plot](#)

[\(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-06.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-06.html).

Note that this method circumvents the need to specify your grid dimensions, as Plotly Express will automatically divide the grid into the number of categories available (in this case, a 2x1 grid — one chart for males and one for females). However, the method only works for creating a figure which contains just one type of graph. We will discuss how to create figures which contain specified dimensions and multiple types of graph in [this section](#) on using Graph Objects.

### Adding Animations: Animation Frames

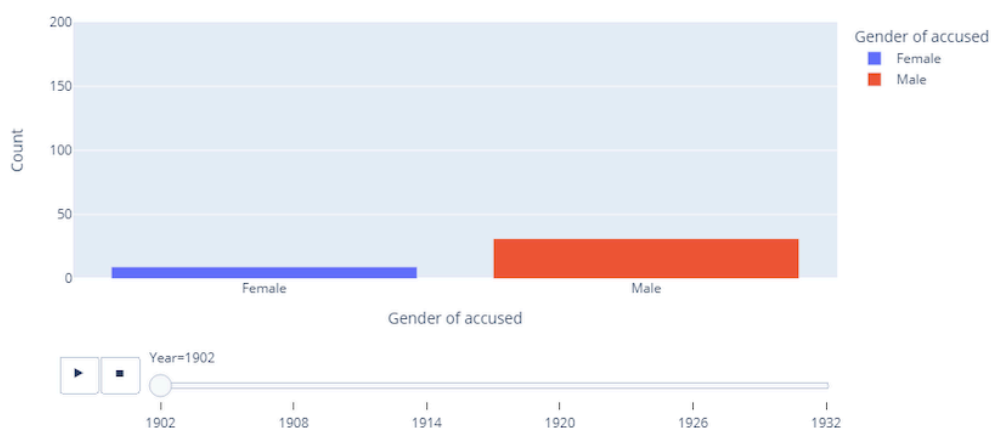
As we have seen, Plotly Express figures already feature some inbuilt interactivity. Yet, there are a number of other features which we could add to increase interactivity, including animation frames.

An animation frame depicts the way data change in relation to a certain measure. In historical research, the measure which is most likely to be useful is time, although most other numerical variables with some inherent rankability (e.g. ordinal or interval data) should work. A Plotly Express figure with an animation frame will contain an interactive toolbar which allows users not only to play/stop the animation but also to manually scroll to the data dispersion at selected points.

To create a figure with an animation frame, start by using the usual method outlined in the above examples to specify which type of graph is desired. Then, within that method, use the `animation_frame` parameter to specify which variable should be used for visualizing change. The example below builds a bar chart depicting changes in male and female homicide prosecutions over the sample period:

```
# Create DataFrame which provides counts of prosecutions by gender
and year
phl_by_gender_year = phl_crime.groupby(
    ["Gender of accused", "Year"], as_index=False
).size()

# Use px.bar() to create a bar chart
fig = px.bar(
    phl_by_gender_year,
    x="Gender of accused",
    y="size",
    labels={"size": "Count"},
    range_y=[
        0,
        200,
    ], # The range_y parameter allows customization of the y-axis
    range (optional)
    color="Gender of accused",
    # title="Add a title here",
    animation_frame="Year", # Use animation_frame to specify which
    variable to measure for change
)
fig.show()
```



(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-07.html).

Figure 7. Animated bar graph featuring an interactive slider created using Plotly Express. As before, readers have the option to hover over the bars to display floating labels. Readers can either invoke an animation of the graph using Play and Stop buttons, or use the slider function to manually shift their view of the data. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-07.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-07.html).

## Adding Animations: Dropdown Bars

Dropdown bars are slightly more complicated than animation frames. They can allow users to switch between a wide variety of display options, including changing colors, lines, axes and even variables. When creating figures with dropdown bars, the first step is to create the initial graph without a dropdown bar (this will be the first graph which your users will see). In this example, we'll be working with a scatter plot which visualises the ages of perpetrators and victims, so we'll create this as follows:

```
fig = px.scatter(
    ph1_crime,
    x="Age of accused",
    y="Victim age",
    color="Charge",
    # title="Add a title here",
    labels={"Age of accused": "Assailant age"},
)
```

Note that the graph has been created but is not visible since we have not used the `fig.show()` command yet. This figure will be displayed once we have added a dropdown bar in the following steps.

After creating our initial graph, we can use the `update_layout` method again to add a dropdown bar. This is a more complex step, since Plotly Express objects' data are nested at many levels under the hood, so we'll need to go a few layers deeper than normal to access the dropdown feature.

Once we have called the `update_layout` method:

- We first need to access the `updatemenus` parameter: this stores a list of dictionaries, each storing the metadata for various design features.
- The only design feature we are currently interested in is the dropdown box, which is stored under the `buttons` dictionary.
- The `buttons` key stores as its value *another* list of dictionaries, each representing the options available in your dropdown bar.
- We'll need to create two `buttons` — one for the stacked bar chart and one for the pie chart — so our `buttons` list will store two dictionaries.
- Each of these two dictionaries will need three key-value pairs:
  - The first, under the `args` key, will specify which type of graph to display.
  - The second, under the `label` key, will specify the text to display in the dropdown bar.
  - The third, under the `method` key, will specify how to modify the chart (update, restyle, animate, etc.).

In the example below, we will look at how to use a dropdown bar to toggle between different categories of a variable. Since our initial scatterplot displays the ages of the accused and their victims, we'll add a dropdown bar which allows users to see data points for either a) all cases, b) murder charges only, c) manslaughter charges only, or d) abortion charges only.

To create the dropdown, we need to take the following steps:

- Under the `label` key, the value will specify the text to display in the dropdown bar.
- Under the `method` key, the value will be 'update' since we are altering the layout *and* the data.
- Under the `args` key, the value (which is another list of dictionaries) will specify which data will be `visible` (more on this issue below), the title for this graph view (optional), and the titles for the x- and y-axes of this graph view (optional).

You need to enter a list for the `visible` key: each item in the list indicates whether the data at that specific index should be displayed. In our example, we have partitioned our dataset into three groups: the data corresponding to murder charges, the data for the manslaughter charges, and the data for the abortion charges. As such, our list for the `visible` key should have three items. Our first `button`, which represents the first graph displayed to the user, should therefore specify `[True, True, True]` since we want all charges to be shown in that first view. However, the remaining three `buttons` will only specify `True` for one item, because we want to show the data for only one type of crime.

Now let's put this into practice:

```

# Use .update_layout() method to add dropdown bar
fig.update_layout(
    updatemenus=[
        dict(
            buttons=list(
                [ # Create the 'buttons' list to store a dictionary
for each dropdown option
                    dict(
                        label="All charges", # Add Label for first
'view'
                        method="update",
                        args=[
                            {
                                "visible": [True, True, True]
                            }, # This 'view' show all three types of
crime
                            {
                                "title": "Victim and assailant ages,
Philadelphia homicides (1902-1932)",
                                "xaxis": {"title": "Age of accused"},
                                "yaxis": {"title": "Victim age"},
                            },
                        ],
                    ),
                dict(
                    label="Murder", # Add Label for second
'view'
                    method="update",
                    args=[
                        {
                            "visible": [True, False, False]
                        }, # Will only show data for first item
(murder)
                        {
                            "title": "Dynamic title: victim and
assailant ages in murder charges", # Can change titles here to make
the graph more dynamic
                            "xaxis": {
                                "title": "Dynamic label: age of
accused"
                            }, # As above
                            "yaxis": {"title": "Dynamic label:
victim age"},
                        },
                    ],
                ), # As above
                dict(
                    label="Manslaughter", # Add Label for third
'view'
                    method="update",
                    args=[
                        {
                            "visible": [False, False, True]
                        }, # Will only show data for second item
(manslaughter)
                    ],
                ),
            ],
        ),
    ],
)

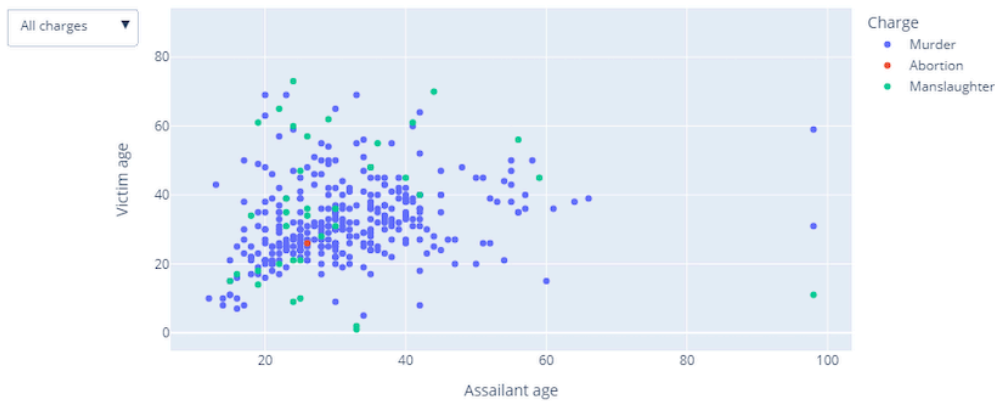
```



```

        {
            "title": "Another dynamic title:
victim and assailant in manslaughter charges", # New title
            "xaxis": {
                "title": "Another dynamic label:
age of accused"
            }, # New x- and y-axis titles
            "yaxis": {"title": "Another dynamic
label: victim age"}},
        },
    ],
),
dict(
    label="Abortion", # Add label for fourth
'view'
    method="update",
    args=[
        {
            "visible": [False, True, False]
        }, # Will only show data for third item
(abortion)
        {
            "title": "More dynamism: ages of
accused and victims in abortion charges", # New title
            "xaxis": {
                "title": "More dynamism: age of
accused"
            }, # New x- and y-axes titles
            "yaxis": {"title": "More dynamism:
victim age"}},
        },
    ],
),
]
)
]
)
fig.show()

```



(</assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-08.html>)

Figure 8. Scatterplot featuring an interactive dropdown filter created using Plotly Express. This iteration of the plot also features a dropdown menu which facilitates filtering by category of Charge or to display All Charges. As before, an interactive legend allows readers to isolate, compare or remove data categories, and hover-over invokes floating labels for individual data points. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-08.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-08.html).

Creating the dropdown bar in the above example provides users with the ability to isolate (and examine) a given element from the wider visualization. We visited this Plotly feature earlier when noting that double-clicking on an element in the graph's legend will remove it from the visualization. However, the dropdown menu offers an additional advantage: it provides us with the ability to create dynamic headings, which means our titles and labels change depending on which option we have selected from the dropdown box.

The above examples demonstrate that it is very easy to create graphs using Plotly Express and relatively simple to add interactivity such as animation frames and dropdown bars. We will now look at creating graphs with Plotly Graph Objects. Specifically, we will focus on what 'Graph Objects' are, how they work, and when (and why) you might want to create graphs using `plotly.go` instead of `plotly.px`.

## Building Graphs with Plotly Graph Objects

### Setting Up Plotly Graph Objects

To start working with Plotly's Graph Objects, you'll need to import the `graph_objects` module:

```
import plotly.graph_objects as go
from plotly.graph_objs.scatter.marker import Line # Not required, but
avoids raising 'DeprecationWarning'.
```

Note that in a conventional `.py` script the libraries should all be imported at the start of the script. The code to import these libraries is provided at this stage for clarity.

## They're All Objects! Data Structure of Plotly Graph Objects

As mentioned in the introduction to this lesson, all Plotly Express figures are actually Graph Objects 'under the hood'. This means that, when you create a figure using `plotly.px`, you are creating an instance of a Graph Object.

This becomes evident if we call the `type` function on the `fig` variable:

```
# Output the figure's 'type'  
print(type(fig))  
  
# <class 'plotly.graph_objs._figure.Figure'>
```

It is important to note therefore that all figures created in Plotly are effectively Graph Objects.

Graph Objects are represented by tree-like (hierarchical) data structures with three top levels: `data`, `layout`, and `frames`:

- The `data` level contains information such as the type of chart, the categories available, the data points falling under each category, whether to show the category in the legend, the types of markers being used for data points, and the text/data to display when hovering over data points.
- The `layout` attribute contains information such as the figure dimensions, the fonts and colors used, any annotations, the coordinates of subplots, the metadata associated with any `buttons` (as discussed in a previous example), and whether any images should be used in the background.
- The `frames` attribute stores information relating to animations used in the figure, such as the data to be displayed at each stop point on a sliding bar. This attribute will not be created unless you add an animation to the figure.

It is easy to view the underlying data structure of a figure by printing it as a Python dictionary with the `fig.to_dict()` function. We can format the structure for easier reading by viewing it in JSON format with `fig.to_json(pretty=True)`. In the example below, we display only the first 500 characters to provide a sample of the output when we use this method (again using the `fig` variable we created above).

```
# print(fig.to_dict())  
print(fig.to_json(pretty=True)[0:500] + "\n...")
```

```

{
  "data": [
    {
      "hovertemplate": "\u003c\u003eGender=\u003c\u002fb\u003e %
{x}\u003cbr\u003e\u003c\u003eCount=\u003c\u002fb\u003e %
{y}\u003cextra\u003e\u003c\u002fextra\u003e",
      "name": "Suspect gender",
      "x": [
        "Female",
        "Male"
      ],
      "y": [
        72,
        617
      ],
      "type": "bar",
      "xaxis": "x",
      "yaxis": "y"
    },
    {
      "hovertemplate":
"\u003c\u003eGender=\u003c\u002fb\u003eFemale\u003cbr\u003e\u003c\u003e
    }
  ]
}

```

Examining the output of a figure should help you to understand the underlying data structure and properties of a graph object. If you print the full output (using the `fig.to_dict()` method), you will notice that:

- Our `data` attribute stores data for each of the three categories (murder, manslaughter, and abortion) under separate dictionaries
- The `data` attribute qualifies which type of graph is being used (in this case 'scatter')
- The `layout` attribute contains the figure title
- The `layout` attribute contains the data associated with the `buttons` (i.e. the dropdown bar)
- There is no `traces` attribute since there is no animation frame associated with this figure

## Using Plotly Graph Objects vs. Plotly Express

Another key point to be aware of is that creating graphs with `plotly.go` typically requires much more code than making the same graph with `plotly.px`.

Consider the following example: building a simple horizontal bar chart to show male vs. female homicide prosecutions. First, let's create a `DataFrame` which tallies prosecution counts by gender:

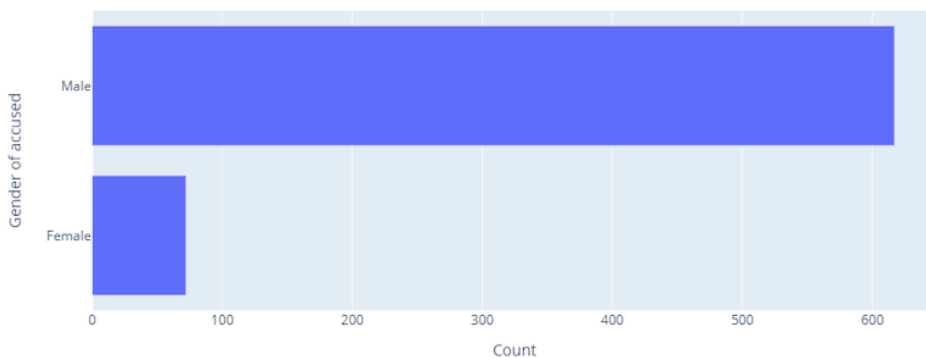
```

phl_by_gender=phl_crime.groupby(["Gender of accused"],
as_index=False).size()

```

Now let's build the horizontal bar chart with these data with `plotly.go` :

```
fig = go.Figure(  
    go.Bar(x=phl_by_gender["size"], # Use go.Bar() to specify chart  
           type as bar chart  
           y=phl_by_gender["Gender of accused"],  
           orientation='h',  
           hovertemplate="Gender=%{y}<br>Count=%{x}<extra></extra>"), # Need  
           to format hover text (this is automatic with plotly.px)  
           # layout={"title": "Add a title here"})  
  
fig.update_layout( # Need to use .update_layout to add x- and y-axis  
                   Labels (this is automatic with plotly.px)  
                   xaxis=dict(title="Count"),  
                   yaxis=dict(title="Gender of accused"))  
  
fig.show()
```



[\(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-09.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-09.html)

Figure 9. Horizontal bar chart with basic interactivity created using Plotly Graph Objects. Readers can hover over the bars to invoke floating labels. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-09.html\)](#).

Note that when using Plotly Graph Objects, you can supply a title using the `layout` argument, which takes a dictionary containing the `title` keyword and its value.

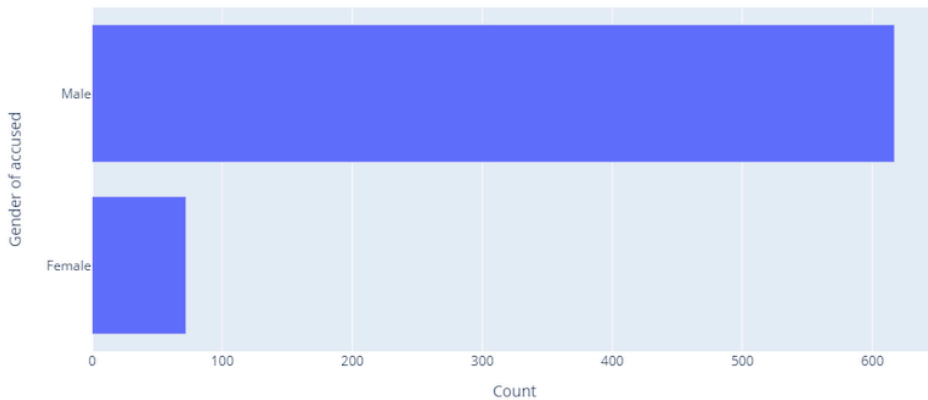
Now, let's create the same figure using `plotly.px` :

```

fig = px.bar(
    ph1_by_gender,
    x="size",
    y="Gender of accused",
    orientation="h",
    # title="Add a title here",
    labels={"size": "Count"},
)

fig.show()

```



(</assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-10.html>)

Figure 10. Horizontal bar chart with basic interactivity created using Plotly Express. Readers can hover over the bars to invoke floating labels. [Click to explore an interactive variant of this plot](#) (</assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-10.html>).

It becomes clear from the above examples that `plotly.go` requires more code than `plotly.px` because many features need to be manually created in `plotly.go`. Thus, it is usually better to use `plotly.px` where possible.

## Why Use Graph Objects?🔗

This leads us to a key question: if it's so much easier to create graphs using `plotly.px`, why should we bother using `plotly.go` at all? The simple answer is that there are several useful features and capabilities which are only available by using `plotly.go`. We will look at two such capabilities in this section of the tutorial: tables and subplots.

## Tables🔗

One of the most useful features provided through the `plotly.go` module is the option to create neat, interactive tables. This requires four steps:

1. Create a new figure using the `.Figure()` method.
2. Under the `data` attribute, call the `.Table()` method to specify that the figure should be a table.
3. Within the `.Table()` method, create a `header` dictionary to store a list of column headings.

4. Also within the `.Table()` method, create a `cells` dictionary to store the data (values).

It is also possible to customise it with labels, colors and alignment options.

In the example below, we'll create a table to store the entire Philadelphia homicides dataset:

```
fig = go.Figure(  
    data=[  
        go.Table( # Create table  
            header=dict(  
                values=list(  
                    phl_crime.columns  
                ), # Get list of all columns in 'phl_crime'  
                DataFrame to use for header  
                fill_color="paleturquoise", # Change heading color  
                align="left",  
            ), # Change header text alignment  
            cells=dict(  
                values=phl_crime.transpose().values.tolist(), # Get  
                values from all columns in dataframe for values  
                fill_color="lavender",  
                align="left",  
            ),  
        )  
    ]  
)  
  
fig.show()
```

Year	Charge	Gender of accused	Age of accused	Victim age	Weapon	Gang
1902	Murder	Male	null	null	Fist, other body part	Teen gang
1902	Murder	Female	null	null	null	No gang
1902	Murder	Male	null	null	Knife, sharp instrument	No gang
1902	Abortion	Female	47	null	null	No gang
1902	Murder	Male	null	null	Knife, sharp instrument	No gang
1902	Abortion	null	null	null	null	No gang
1902	Abortion	Female	null	38	null	No gang

[\(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-11.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-11.html)

Figure 11. Table containing the Philadelphia Homicides dataset created with Plotly Graph Objects. Readers can scroll through the entire dataset as they would in a spreadsheet. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-11.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-11.html).

As with `plotly.px`, figures created with `plotly.go` have some inherent interactivity. Tables, for example, provide users the ability to scroll through rows (either using a trackpad or the scrollbar on the right) and are therefore excellent for saving space. It is also easy to move columns around by clicking on the column headers and dragging them left or right.

## Subplots

Another useful feature of the `plotly.go` module is its capacity for building subplots. Although `plotly.px` can build facet plots, these are comparatively limited, since they must all share the same graph type, axes and variables. Subplots, on the other hand, allow you to create a grid containing different types of graphs with their own axes and variables, which ends up looking like a sort of 'dashboard'.

Since the code is particularly lengthy for creating subplots, this example will be provided on a step-by-step basis. We will create a 3x1 grid containing three different charts: the first will be a standard bar chart to quantify prosecution counts for male vs. female suspects; the second will be a line graph showing changes in male vs. female prosecutions over time; and the third will be a boxplot showing the minimum, inter-quartile range and maximum ages of male vs. female suspects.

### Step 1: Import the subplots module and get data

```
# Import make_subplots
from plotly.subplots import make_subplots

# Gather data for subplot
phl_women, phl_men = (
    phl_crime.loc[phl_crime["Gender of accused"] == "Female"],
    phl_crime.loc[phl_crime["Gender of accused"] == "Male"],
)
phl_women_year, phl_men_year = (
    phl_women.groupby(["Year"], as_index=False).size(),
    phl_men.groupby(["Year"], as_index=False).size(),
)
```

### Step 2: Create an empty subplot with a 3x1 grid using the `make_subplots()` function

```
fig = make_subplots(rows=1, cols=3) # Use the rows and cols
parameters to create smaller/bigger grid
```

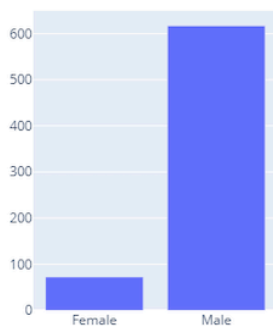
### Step 3: Add the first graph (the bar chart) using the `.add_trace()` method



```

fig.add_trace(
    # Use go.Bar() to specify chart type as bar chart
    go.Bar(
        x=ph1_by_gender[
            "Gender of accused"
        ],
        y=ph1_by_gender["size"],
        name="Suspect gender",
        hovertemplate="<b>Gender=</b> %{x}<br><b>Count=</b> %{y}"
<extra></extra>",
    ),
    # Use the row and col parameters to change the position of the
    subplot within the grid
    row=1,
    col=1,
)

```



</assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-12.html>)

Figure 12. A three-column plot with basic interactivity created using Plotly Graph Objects, with a bar chart on the left showing the number of accused by gender, and two empty columns on the right. Readers can hover over the bars to invoke floating labels. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-12.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-12.html).

Note: If you are creating a subplot using Jupyter Notebook, rerunning the code may duplicate the trace you added and thereby add more items to the legend. If you need to rerun the code, it is best to restart the kernel first.

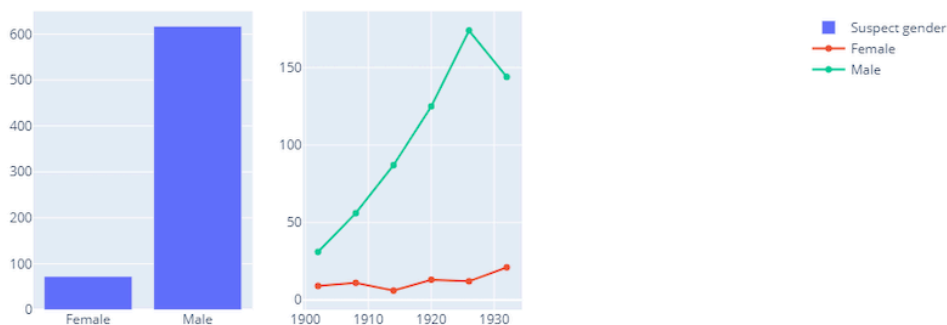
**Step 4: Add the second graph (the line graph)**

```

fig.add_trace(
    # Use go.Line() here to specify graph type as line graph
    go.Line(
        x=phl_women_year[
            "Year"
        ],
        y=phl_women_year["size"],
        name="Female", # Specify that this line represents female
        prosecutions
        hovertemplate="<b>Gender=</b>Female<br><b>Year=</b> %<{x}<br>
<b>Count=</b> %<{y}</b><extra></extra>",
    ),
    # the col parameter is now 2 (rather than 1) since we want to
    position this graph next to the bar chart.
    row=1,
    col=2,
)

# Since we want separate lines for male and female charges, we need
to add two 'Line' traces to the plot.
fig.add_trace(
    go.Line(
        x=phl_men_year["Year"],
        y=phl_men_year["size"],
        name="Male", # Specify that this line represents male
        prosecutions
        hovertemplate="<b>Gender=</b>Male<br><b>Year=</b> %<{x}<br>
<b>Count=</b> %<{y}</b><extra></extra>",
    ),
    row=1,
    col=2,
)

```



(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-13.html)

Figure 13. A three-column plot created using Plotly Graph Objects, with a bar chart showing the number of accused by gender and a line graph showing the number of male and female accused by year. Hover-over invokes floating labels for each bar

and individual data points. The third column is still empty. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-13.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-13.html).

Note that if you did not import `plotly.graph_objs.scatter.Line`, you may get the following warning (which you can safely ignore):

```
/Library/Frameworks/Python.framework/Versions/3.10/lib/python
packages/plotly/graph_objs/_deprecations.py:378:
DeprecationWarning:

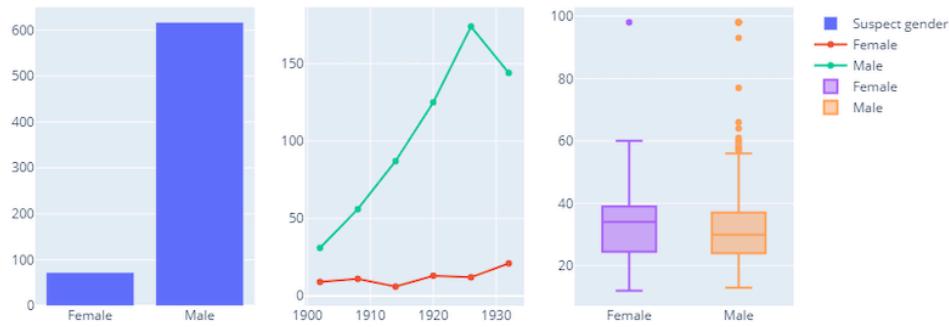
plotly.graph_objs.Line is deprecated.
Please replace it with one of the following more specific
types
- plotly.graph_objs.scatter.Line
- plotly.graph_objs.layout.shape.Line
- etc.
```

### Step 5: Add the final graph (the boxplot)🔗

We have not looked at boxplots yet, but they are created in a similar way to other graphs and have similar interactive behaviour (e.g. scrolling over a box will show the minimum, maximum, median, and interquartile range of the data).

```
fig.add_trace(
    # Use go.Box() to specify graph type as boxplot
    go.Box(
        y=phl_women["Age of accused"], name="Female"
    ),
    row=1,
    col=3, # col=3 now because it is the third graph in the grid
    figure
)

# As before, we need to add another trace since we want a separate
box for males.
fig.add_trace(go.Box(y=phl_men["Age of accused"], name="Male"),
    row=1, col=3)
```



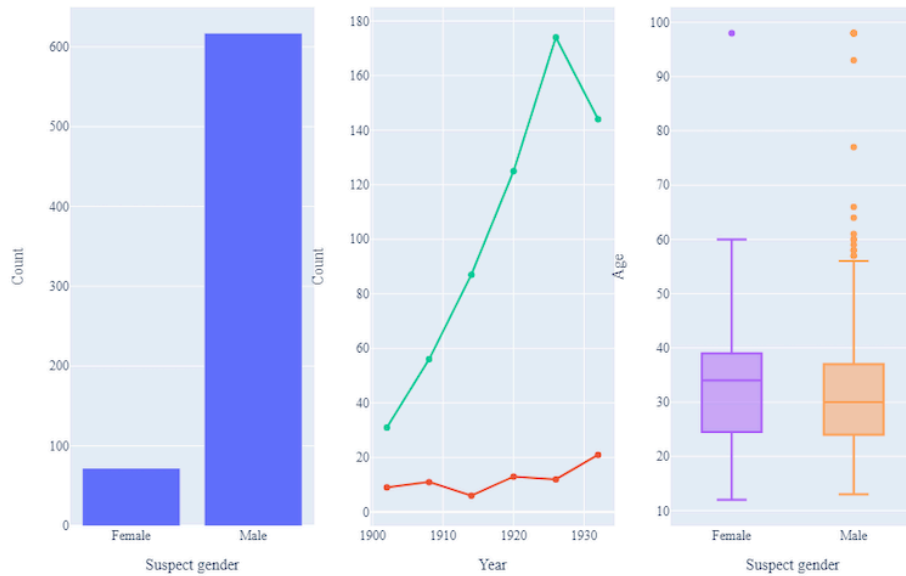
(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-14.html)

Figure 14. A three-column plot created using Plotly Graph Objects, with a bar chart showing the number of accused by gender, a line graph showing the number of male and female accused by year, and a box plot showing the distribution and outlier values of male and female accused by age. Hover-over invokes floating labels for each bar or individual data point and shows the minimum, maximum, median, and interquartile range on the box plot. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-14.html\)](/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-14.html).

### Step 6: Format the figure

There are still some tweaks needed, like adding a main title for the figure and subtitles for each subplot. You might also want to change fonts, text positioning, and the figure size — you can use the `.update_layout()` method to change these properties:

```
fig.update_layout(
    font_family="Times New Roman", # Change font for the figure
    hoverlabel_font_family="Times New Roman", # Change font for
    hover Labels
    hoverlabel_font_size=16, # Change font size for hover Labels
    # title_text="Add a title here", # Main title
    # title_x=0.5, # Position main title at center of graph (note:
    the title_x parameter only takes integers or floats)
    xaxis1_title_text="Suspect gender", # Add Label for x-axis in
    1st subplot
    yaxis1_title_text="Count", # Add Label for y-axis in 1st subplot
    xaxis2_title_text="Year",
    yaxis2_title_text="Count",
    xaxis3_title_text="Suspect gender",
    yaxis3_title_text="Age",
    showlegend=False, # Remove Legend
    height=650, # Set height for graph - not needed, but can be
    useful for digital publishing
)
```



(</assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-15.html>).

Figure 15. A three-column plot created using Plotly Graph Objects, with a bar chart showing the number of accused by gender, a line graph showing the number of male and female accused by year, and a box plot showing the distribution and outlier values of male and female accused by age. This plot is a variant of that produced in Figure 14, with added subtitles for each subplot. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-15.html\)](#).

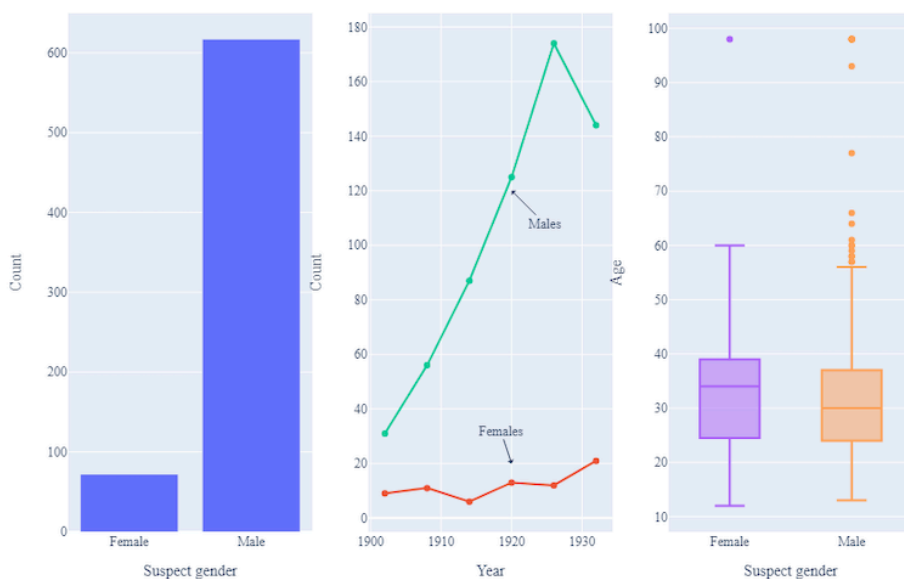
### Step 7: Add annotations to the line graph

Since the legend has been removed, it's now impossible to distinguish between the lines which represent males and those which represent females in the line graph. We can use the `.update_layout()` method to add annotated arrows pointing to each line:

```

fig.update_layout(
    # Pass in a list of dicts where each dict represents one
    # annotation
    annotations=[
        # Our first annotation will be for the 'males' line
        dict(
            x=1920,
            y=120, # X- and y- co-ordinates for the annotation point
            xref="x2", # Specify xref and yref as x2 and y2 because
            # we want the second graph in the grid (the line graph)
            yref="y2",
            text="Males", # Text for annotation is 'Males'
            showarrow=True, # Use False to add a line without an
            # arrowhead
            arrowhead=1, # Change size of arrowhead
            ax=30, # Use the ax and ay parameters to change length
            # of line
            ay=30,
        ),
        # Our second annotation will be for the 'females' line
        dict(
            x=1920,
            y=20,
            xref="x2",
            yref="y2",
            text="Females",
            showarrow=True,
            arrowhead=1,
        ),
    ],
)

```



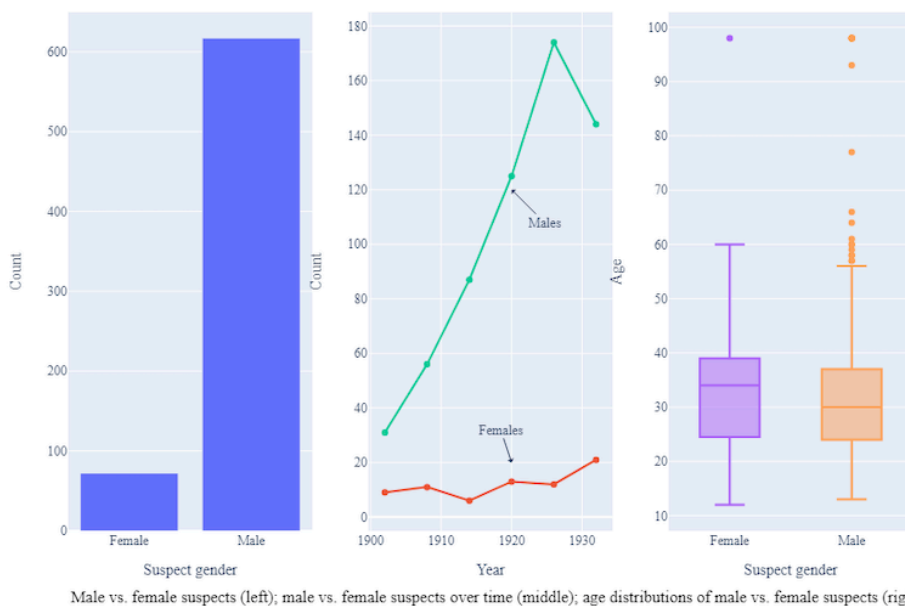
(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-16.html)

Figure 16. A three-column plot created using Plotly Graph Objects, with a bar chart showing the number of accused by gender, a line graph showing the number of male and female accused by year, and a box plot showing the distribution and outlier values of male and female accused by age. This plot is a variant of that produced in Figure 15, with added annotations in the line graph. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-16.html\).](#)

### Step 8: Add annotations below the figure

We might want to add annotations below the figure to specify the focus of each subplot (particularly useful for academic publishing), which we can do using the `.add_annotation()` method:

```
fig.add_annotation(
    dict(
        font=dict(color="black", size=15), # Change font color and
        size
        x=0, # Use x and y to specify annotation position
        y=-0.15,
        showarrow=False,
        text="Male vs. female suspects (left); male vs. female
        suspects over time (middle); age distributions of male vs. female
        suspects (right).",
        textangle=0, # Option to rotate text (sometimes useful to
        save space)
        xanchor="left",
        xref="paper", # Set xref and yref to 'paper' so that x and y
        coordinates are absolute refs.
        yref="paper",
    )
)
```



Male vs. female suspects (left); male vs. female suspects over time (middle); age distributions of male vs. female suspects (right).

[\(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-](#)

[17.html](#))

Figure 17. A three-column plot created using Plotly Graph Objects, with a bar chart showing the number of accused by gender, a line graph showing the number of male and female accused by year, and a box plot showing the distribution and outlier values of male and female accused by age. This plot is a variant of that produced in Figure 16, with an additional annotation added below all three subplots. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-17.html\)](#).

## Viewing and Exporting Figures

In the previous sections of the lesson, we saw how to create and modify interactive graphs using both the `plotly.px` and `plotly.go` modules. We will next consider how to view and export these graphs for publications or other research outputs.

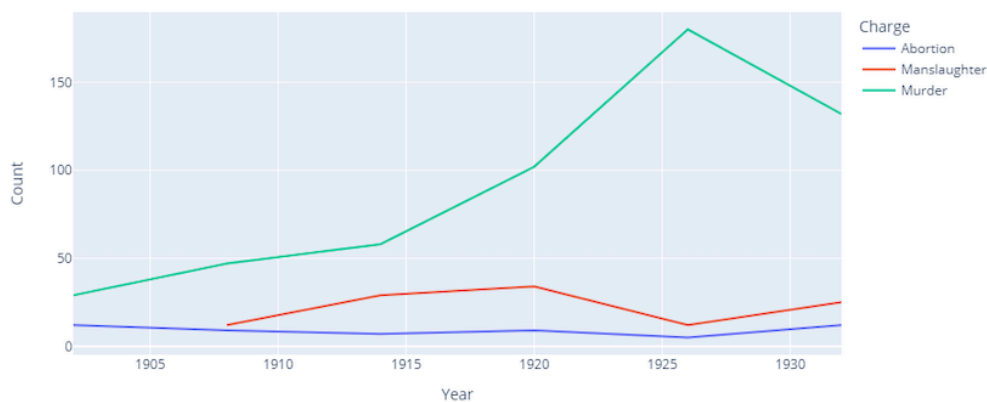
The methods discussed here will use a basic line graph, identical to that created earlier in the tutorial (see Figure 3). Let's start by recreating that figure:

```
fig = px.line(  
    ph1_by_year,  
    x="Year",  
    y="size",  
    # title="Add a title here",  
    labels={"size": "Count"},  
    color="Charge",  
)
```

## Viewing Figures

As we have seen throughout this lesson, the `.show()` method can be used to output a figure. By default, this method uses the inbuilt Plotly renderer and therefore provides interactivity:

```
fig.show()
```



[\(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-03.html\)](#)

Figure 18. Reproduction of Figure 3, illustrating the `fig.show()` method. [Click to explore an interactive variant of this plot \(/assets/interactive-visualization-with-plotly/interactive-visualization-with-plotly-03.html\)](#).



## Exporting Figures🔗

Plotly figures can be exported either as interactive or static (non-interactive) graphs. Interactive graphs may be useful for research websites and (some) digital publications, whereas static graphs are appropriate for print publications.

### Exporting as HTML🔗

Exporting figures as HTML retains their interactivity when viewed in a web browser. Any figure can be saved as an HTML file by using the `.write_html()` method:

```
# Save HTML file of the graph (which we have been storing under the variable name 'fig' throughout this lesson)  
fig.write_html("your_file_name.html")
```

By default, any exported figure will be saved in the same folder as that in which your script is stored. If you want to store the figure elsewhere (a different folder), you can specify the exact directory when you specify the file name (for example, `fig.write_html("your_path/your_file_name.html")` ).

### Exporting static images🔗

Plotly provides numerous options for exporting both raster images ( `.png` or `.jpg` ) and vector images ( `.pdf` or `.svg` ). To do this, use the `.write_image()` function and specify the image type within the file name:

```
# Export to raster graphic, either png or jpg:  
fig.write_image("your_file_name.png")  
fig.write_image("your_file_name.jpeg")  
  
# Export to vector graphic, either pdf or svg:  
fig.write_image("your_file_name.svg")  
fig.write_image("your_file_name.pdf")
```

## Summary🔗

Plotly offers the ability to create publication-quality, interactive figures using Python and other programming languages. This lesson provides an overview of what Plotly is, why it's useful, and how it can be used with Python. It also demonstrates the different modules in the Plotly framework (Plotly Express and Plotly Graph Objects) and the methods required to create, edit, and export data visualizations. The key syntaxes covered in this lesson are:


- Installing Plotly using `pip install plotly`
- Importing Plotly Express and Plotly Graph Objects using `import plotly.express as px` and `import plotly.graph_objects as go`
- With Plotly Express:
  - Creating graphs using `px.bar()` , `px.line()` , and `px.scatter()`
  - Adding features to graphs using parameters such as `title` , `labels` , `color` , and `animation_frame`
  - Using `.update_layout()` to edit figures after creation and/or add dropdown bars
- With Plotly Graph Objects:

- Recognizing the `data`, `layout`, and `traces` attributes as key underlying structure of a graph object
- Creating new (empty) graph objects with `go.Figure()`
- Creating graphs using `go.Bar()`, `go.Table()`, `go.Line()`, and `go.Box()`
- Editing features using the `layout` attribute
- Creating subplots (importing the subplots module using `from plotly.subplots import make_subplots`, implementing subplots with `make_subplots()` function, and adding traces to subplots using `.add_trace()` method)
- Using `.update_layout()` to edit figures after creation
- Exporting figures created with Express or Graph Objects using `.write_html()` and `.write_image()`

## Endnotes

1. Under the hood, these libraries are built on top of the Plotly JavaScript library. [↵](#)
2. Plotly Dash is outside the scope of this lesson, which instead focuses on `plotly.px` and `plotly.go`. [↵](#)
3. For further information on Bokeh, see Charlie Harper's lesson on [Visualizing Data with Bokeh and Pandas \(/en/lessons/visualizing-with-bokeh\)](#) here on *Programming Historian*. [↵](#)
4. The dataset and its related documents are available freely via the [Historical Violence Database \(https://perma.cc/WCW9-YRX9\)](#) project organized by Ohio State University and licensed under a [Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License \(https://perma.cc/3BYZ-UDYW\)](#). [↵](#)
5. If you already work with Jupyter notebooks, there is a good chance that other dependencies are already installed. However, if you are working in a clean Python environment or in a code editor like VS Code, it may also be necessary to run `pip install ipykernel` and `pip install nbformat`. [↵](#)
6. We will also be using the NumPy module, but this is automatically installed with the installation of pandas. [↵](#)
7. Kaleido is a Python library for generating static images (e.g. JPG and SVG files) and will therefore be needed when exporting non-interactive graphs. [↵](#)

## ABOUT THE AUTHOR

Grace Di Méo is a Lecturer in Criminology in the Faculty of Humanities and Social Sciences at Oxford Brookes University.  <https://orcid.org/0000-0002-3227-8053>.

## SUGGESTED CITATION

Grace Di Méo, "Creating Interactive Visualizations with Plotly," *Programming Historian* 12 (2023), <https://doi.org/10.46430/phen0115>.

The *Programming Historian* (ISSN: 2397-2068) is released under a [CC-BY](https://creativecommons.org/licenses/by/4.0/deed.en) (<https://creativecommons.org/licenses/by/4.0/deed.en>), license.

This project is administered by ProgHist Ltd, Charity Number [1195875](https://register-of-charities.charitycommission.gov.uk/charity-search/-/charity-details/5181272/charity-overview) (<https://register-of-charities.charitycommission.gov.uk/charity-search/-/charity-details/5181272/charity-overview>) and Company Number [12192946](https://find-and-update.company-information.service.gov.uk/company/12192946) (<https://find-and-update.company-information.service.gov.uk/company/12192946>).

**[ISSN 2397-2068 \(English\) \(/\)](#)**


[ISSN 2517-5769 \(Spanish\) \(/es\)](#)


[ISSN 2631-9462 \(French\) \(/fr\)](#)


[ISSN 2753-9296 \(Portuguese\) \(/pt\)](#)


 [Hosted on GitHub \(https://github.com/programminghistorian/jekyll\)](https://github.com/programminghistorian/jekyll)

 [Site last updated 17 October 2024 \(https://github.com/programminghistorian/jekyll/commits/gh-pages\)](https://github.com/programminghistorian/jekyll/commits/gh-pages)

 [RSS feed subscriptions \(https://programminghistorian.org/feed.xml\)](https://programminghistorian.org/feed.xml)

 [See page history \(https://github.com/programminghistorian/jekyll/commits/gh-pages/en/lessons/interactive-visualization-with-plotly.md\)](https://github.com/programminghistorian/jekyll/commits/gh-pages/en/lessons/interactive-visualization-with-plotly.md)

 [Make a suggestion \(/en/feedback\)](/en/feedback)      [Lesson retirement policy \(/en/lesson-retirement-policy\)](/en/lesson-retirement-policy)

 [Translation concordance \(/translation-concordance\)](/translation-concordance)