Plot Streaming Data with Python and Plotly Express

Tracking the ISS in real time

8 min read · Mar 26, 2023

Lee Vaughan Follow

Listen Share ••• More

The International Space Station (image from NASA Image Gallery)

Streaming data refers to real-time data which is continuously flowing from a source to a target. It includes audio, video, text, or numerical data that is generated by sources such as social media platforms, sensors, and servers. The data transmission is in a steady stream with no fixed beginning or end. Streaming data is important in fields such as healthcare, finance, and transportation, and it forms a key component of the *Internet of Things (IoT)*.

The ability to handle streaming data is an important skill for a data scientist. In this *Quick Success Data Science* project, we'll use streaming data to track the International Space Station (ISS) as it orbits the Earth. For coding we'll use Python and Plotly Express in a Jupyter Notebook.

International Space Station Telemetry

Telemetry is the *in-situ* collection and automatic transmission of remote sensor data to receiving equipment for monitoring. While there are numerous sources for ISS telemetry, we'll use the *WTIA REST API* (WTIA stands for *Where the ISS at?*).

This API was written by Bill Shupp to include more features than are provided by typical ISS tracking/notification sites. At the end of the article, I'll list some additional sources for ISS telemetry and tracking, in case you want to try them out or compare our results to theirs.

The Plotly Express Library

Plotly Express is a built-in part of the Plotly graphing library. As a simpler, higher-level version of Plotly, it's the recommended starting point for creating most common figures.

Plotly Express contains more than 30 functions for creating entire figures at once, and the API for these functions was carefully designed to be as consistent and easy to learn as possible. This makes it easy to switch between figure types during a data exploration session.

While Plotly Express is easy to use and creates beautiful, interactive plots, they're not as customizable as plots generated in lower-level libraries like Plotly or matplotlib. As always, you have to give up some control for ease of use.

Installing Plotly gives you access to Plotly Express. You can install it using `pip`:

```
pip install plotly==5.13.1
```

or conda:

```
conda install -c plotly plotly=5.13.1
```

Note that the version number will change in time, so be sure to check the Plotly docs, which also include support for classic Jupyter Notebook and JupyterLab, if needed. Additionally, you can find the *PyPi* page for Plotly Express here.

The Process

Data streams aren't truly continuous. Websites are updated at *some* frequency, such as once per second. Likewise, we can't get the data faster than it's created, nor should we try to get it as fast as possible. If you ping a website too frequently, it may think it's under attack and block your access. So, when possible, we should request data at the polite rate of every 5 or 10 seconds.

In this project, we'll handle streaming data *incrementally*. Here's the workflow:

1. Get the data from a website.
2. Put it in a pandas DataFrame.
3. Plot it with Plotly Express.
4. Clear the plot.
5. Repeat.

Importing Libraries and Assigning Constants

In addition to Plotly Express, we'll need the `time` module, to control how often we access the streaming data; the requests library, for retrieving the data from a URL; the pandas library, for preparing the data for plotting; and `IPython.display`, for clearing the plot prior to posting an updated ISS location.

We'll use a constant named `ISS_URL` to store the WTIA REST API URL. And since streaming data never ends, we'll assign another constant, `ORBIT_TIME_SECS`, to help us determine when to end the program. This constant represents the approximate time — in seconds — for one complete ISS orbit of the Earth (about 92 minutes).

```
import time
import requests
import pandas as pd
import plotly.express as px

from IPython.display import clear_output

ISS_URL = 'https://api.wheretheiss.at/v1/satellites/25544'
ORBIT_TIME_SECS = 5_520  # Time required for ~1 complete orbit of ISS.
```

Getting the Streaming Data

The following `get_iss_telemetry` function retrieves the streaming data from the URL, loads it as a pandas DataFrame, drops unnecessary columns, and returns the DataFrame. Each time this function is called, it records the ISS telemetry for a *single instant in time*. Later, we'll call this function in a loop to track the ISS *over time*. Of course, for all this to work, you'll need an active internet connection.

```
def get_iss_telemetry(url):
    """Return DataFrame of current ISS telemetry from URL."""
    response = requests.get(url)
    if response.status_code != 200:
        raise Exception(f"Failed to fetch ISS position from {url}. \
        Status code: {response.status_code}")
    data = response.json()
    telemetry = pd.DataFrame(data, index=[0])
    telemetry = telemetry.drop(['id', 'footprint', 'daynum',
                                'solar_lat', 'solar_lon'], axis=1)
    return telemetry
```

The first step is to get the telemetry from the website. The requests library abstracts the complexities of HTTP (*HyperText Transfer Protocol*) requests in Python, making them simpler and more human friendly. The `get()` method retrieves the URL and assigns the output to a `response` variable, which references the `Response` object the web page returned for the request. This object's *text attribute* holds the web page as a readable text string.

The next step is to check the `response` object's HTTP status code. A code of 200 means that the client has requested documents from the server and the server has

complied. While 200 is the most common successful response, you may also see 201, 202, 205, or 206.

The website response is in JSON (*JavaScript Object Notation*) format, so we use the `.json()` method to load the text string as a Python dictionary, named `data`. Here's an example of the output:

```
{'name': 'iss', 'id': 25544, 'latitude': 38.880080494467, 'longitude': -67.40
```

Next, we turn the `data` dictionary into a pandas DataFrame named `telemetry`. Note that the `index=[0]` argument heads off an "avoid scalar values without index" error. As we don't need information like the "id" or "solar_lat," we'll drop those columns before returning the DataFrame.

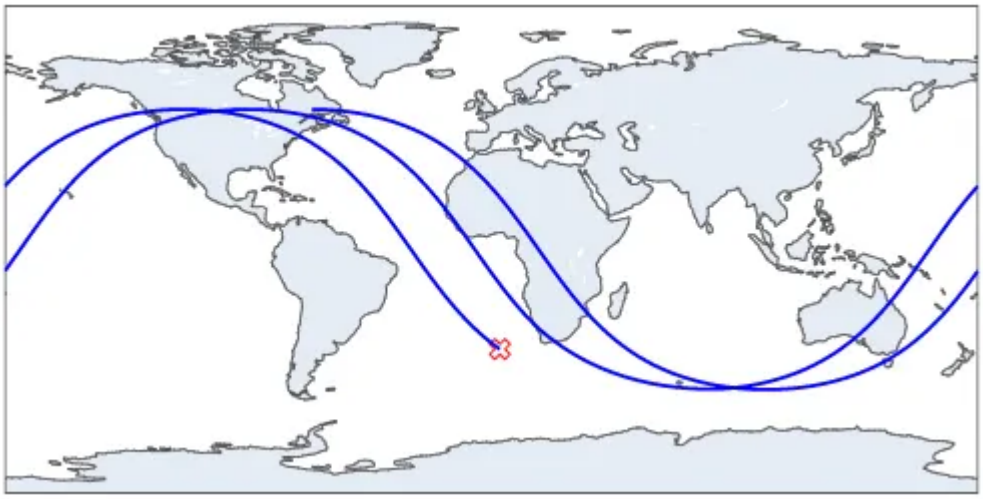
	name	latitude	longitude	altitude	velocity	visibility	timestamp	units
0	iss	7.030937	59.909125	415.386953	27595.254272	eclipsed	1679692473	kilometers

The final “telemetry” DataFrame (image by author)

Plotting the ISS

The last step is to define a function to plot the telemetry for a given number of orbits. This function will call the previous function using a `for` loop. The resulting plot will include a *marker* for the ISS and a *line* to record its *ground track* over time.

As we will be plotting a sphere onto a 2D map, this circular path will appear as an S-shaped *sinusoid*. Here’s a fun video explaining why.



Circular orbits appear sinusoidal when flattened (image by author).

```
def track_iss(url, num_orbits=2, interval=10):
    """
    Plot current ISS location from URL and record and plot its track.

    Arguments:
       url = ISS telemetry URL -> string
       num_orbits = Number of ISS orbits to plot -> integer
       interval = Wait period (seconds) between calls to URL -> integer
    """

    num_pulls = int(ORBIT_TIME_SECS * num_orbits / interval)
    latitudes = []
    longitudes = []

    for _ in range(num_pulls):
        df = get_iss_telemetry(url).round(2)
        latitudes.append(df['latitude'].iloc[0])
        longitudes.append(df['longitude'].iloc[0])
        clear_output(wait=True)

        fig = px.scatter_geo(df,
                             lat='latitude',
                             lon='longitude',
                             color='visibility',
                             color_discrete_map = {'daylight': 'red',
                                                   'visible': 'orange',
                                                   'eclipsed': 'black'},
                             hover_data=['timestamp', 'altitude',
                                         'velocity', 'units'])
        fig.add_trace(px.line_geo(lat=latitudes, lon=longitudes).data[0])
        fig.update_traces(marker=dict(symbol='x-open', size=10),
                          line_color='blue')
        fig.update_layout(width=700, height=500,
                          title='International Space Station Tracking')
        fig.show()
```

```
        time.sleep(interval)
```

The `track_iss` function takes as arguments a URL, the number of orbits to plot, and a time interval, in seconds, for pausing execution. This pause is to prevent the website from being overwhelmed with requests. The last two arguments use keywords which means they will be treated as defaults.

The `num_pulls` variable refers to how many times we call our `get_iss_telemetry()` function and "pull" data from the website. This number is based on the orbital time, the number of orbits, and the pause interval.

To draw the path of the ISS across the map, we'll need to store the lat-lon pair from each pull in a list. So, we initialize an empty list for each attribute prior to starting the loop.

To plot the telemetry, we loop through the `num_pulls` variable and first call the `get_iss_telemetry()` function, rounding the result to two decimal places. We then append the latitude and longitude results to our lists and clear the screen, so that we start each loop with a fresh plot.

Now to generate the figure. First, we call the `px.scatter_geo()` method to plot points on a map of the Earth. The inputs are intuitive. The color of our ISS marker is determined by the DataFrame's "visibility" column. The output consists of "daylight," "eclipsed," and "visible." The latter refers to when the ISS is still reflecting sunlight and thus visible in the night sky. To convert these outcomes to colors, we pass the `color_discrete_map` argument a dictionary.

To support "hover window" popups, we pass the `hover_data` argument a list of column names for the data we want to see. To see the *complete* DataFrame, we'd pass `df.columns`, rather than a list.

In Plotly, a *figure* is comprised of one or more *traces*, where each trace is a plot element like a scatter plot, line plot, or bar plot. The `px.line_geo()` method returns a figure object containing a *single* trace object that represents a line drawn with the given latitudes and longitudes. We add this object using its *data attribute* (`.data[0]`). The index is "0" because there's only one trace.

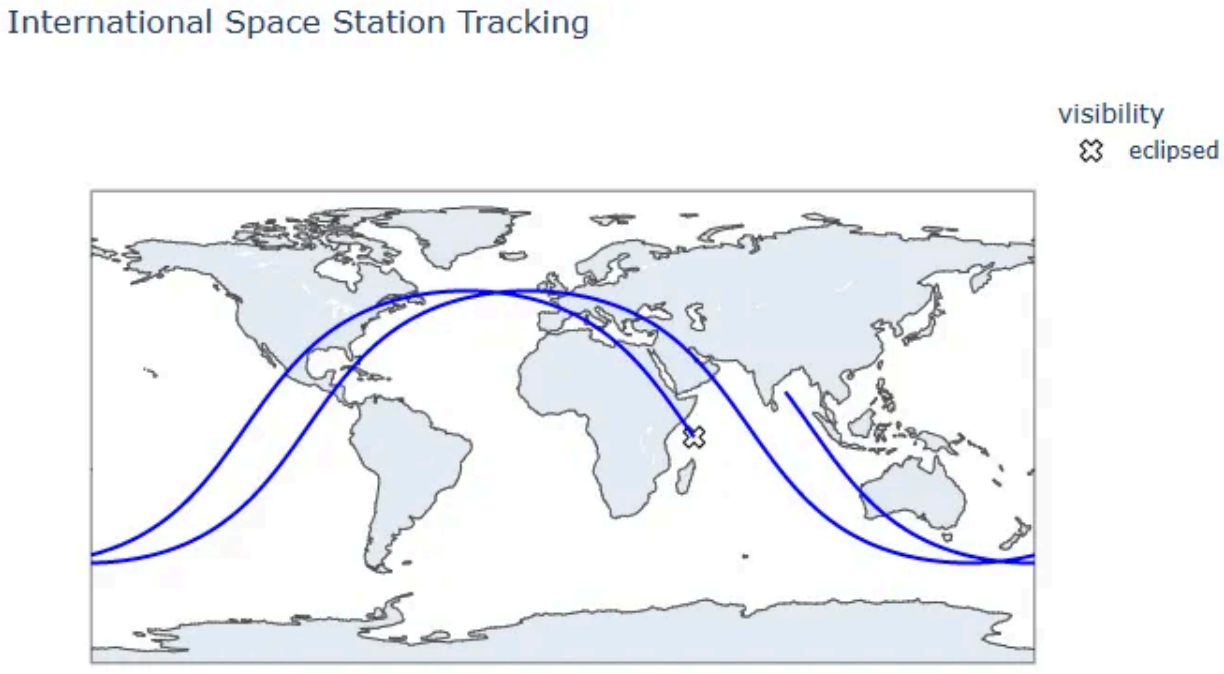
After adding a trace, we need to update the traces in the figure with the `update_traces()` method. Our ISS marker parameters are specified using a dictionary. We use an open "X" for the marker symbol. For the line plot, we specify a color of blue.

To finish the figure, we call the `update_layout()` method and pass it a width, height, and title. Then we call the `show()` method to display the results.

The loop ends by calling the `time` module's `sleep()` method and passing it the `interval` variable. In this case, it will pause the loop for 10 seconds. Even though the WTIA REST API is *rate limited* to one second, there's no need to be greedy!

All that's left to do is call our function:

```
track_iss(ISS_URL)
```

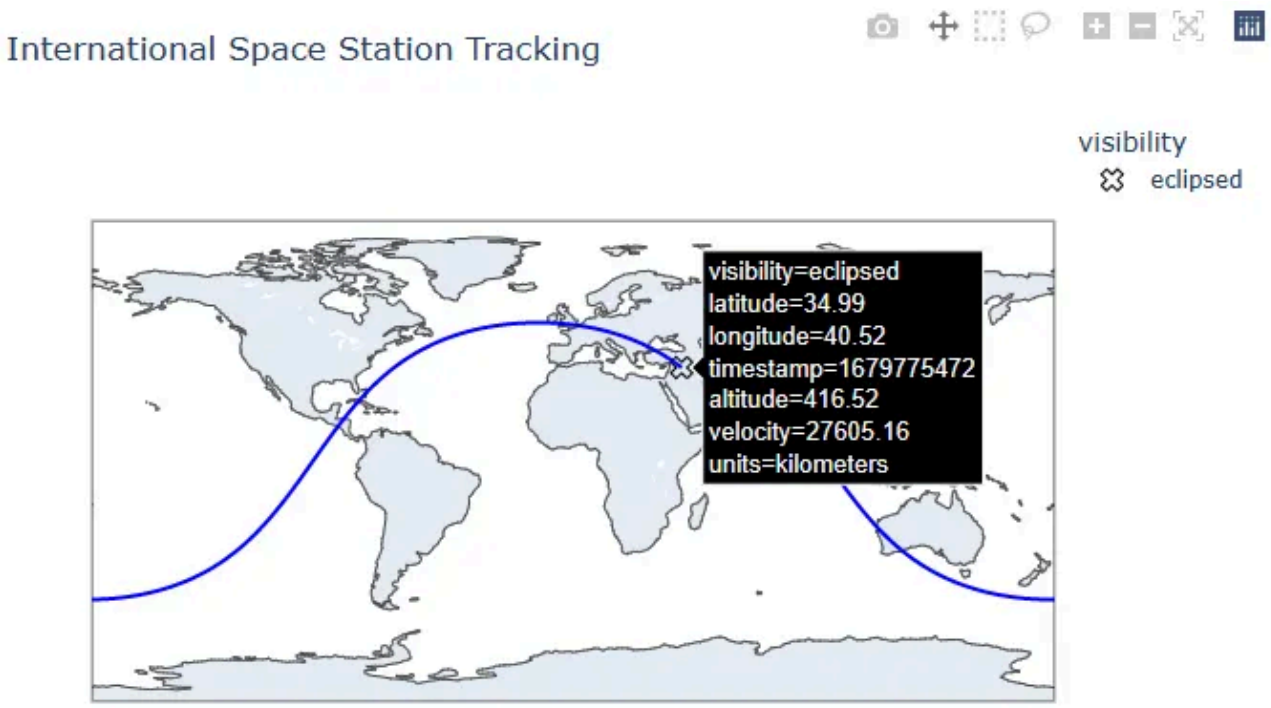


The ISS tracker showing ~2 orbits of the ISS (image by author)

The Outcome

Despite a trivial amount of code, this plot possesses a *lot* of functionality.

If you hover the cursor over the marker, a popup window will display telemetry such as the station's altitude, velocity (in km/hour), visibility, and so on.



The Plotly Express figure with the ISS marker's hover window active (image by author)

You can zoom in and out with a mouse scroller or the toolbar. You can take screenshots, pan, and reset the view. The ISS marker will change colors depending on how the ISS is illuminated.

The ISS marker turns red when its position corresponds to daytime (image by author)

Finally, we handled the streaming ISS data by treating it *incrementally*. And even though it's being streamed at a frequency of one hertz, we pulled the data every ten secconds to go easy on the source API.

More Trackers

You can also access the ISS's telemetry through the *Open-Notify-API*.

And as always with Python, there's more than one way to accomplish a task. Here are some alternative approaches for tracking the station:

- Use Plotly Express with an orthographic (globe) projection.
- Use Plotly Express and calculate the station's velocity.
- Use an ISS icon for tracking and include crew information.
- Track the station with Raspberry Pi.

Thanks!

Thanks for reading and be sure to follow me for more *Quick Success Data Science* projects in the future.