QUICK SUCCESS DATA SCIENCE

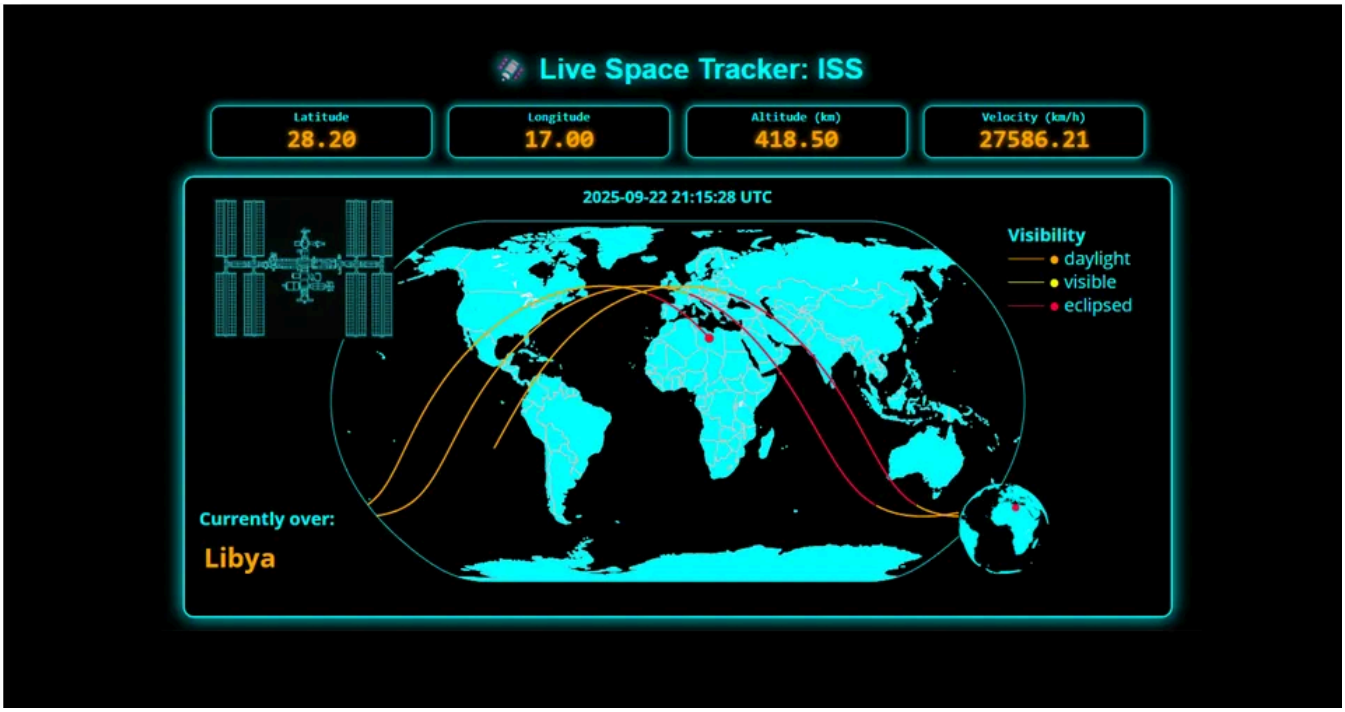
Build a Sleek Sci-Fi Dashboard with Python and Dash

From movie Inspiration to ISS Tracker

19 min read · 3 days ago

Lee Vaughan Follow

Listen Share ••• More



The ISS Tracker dashboard built with Plotly and Dash (by author)

I'm a sucker for the glowing dashboards and cool control panels in Sci-Fi films — the kind that pulse with data and look like they belong on a starship bridge. After watching *Alien Rubicon* and admiring its stunning dashboard design, I decided to build one of my own. But instead of a dense, fictional readout, I wanted something grounded in reality — something space-related and dynamic. That's when I landed on the perfect subject: tracking the International Space Station (ISS).

The final dashboard, featured in the title image, is easily adaptable to other satellites that offer comparable telemetry feeds. (*Telemetry* is the *in-situ* collection and transmission of remote sensor data to receiving equipment for monitoring.) If

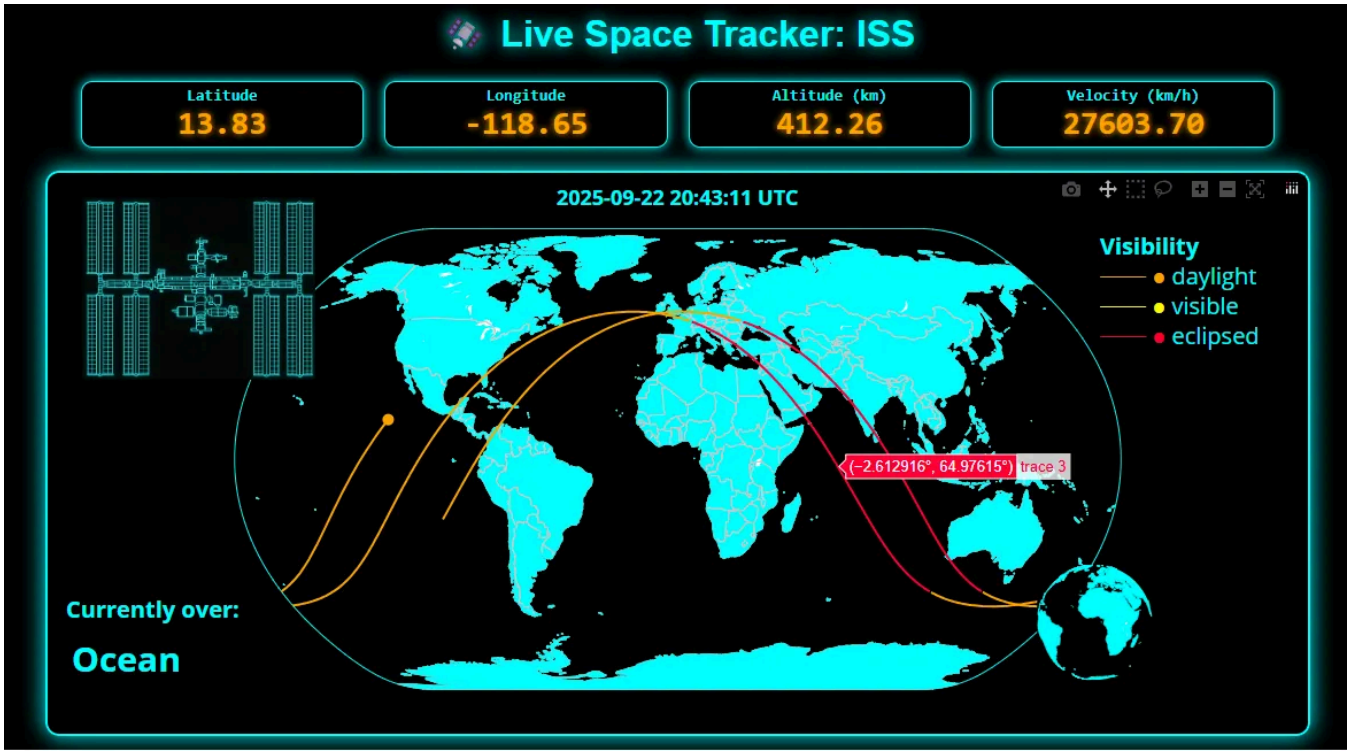
you're curious about space-themed dashboards *beyond* the ISS, like tracking exoplanets or monitoring orbital debris, you'll find a curated set of links at the end of this article.

While it's fun to have a sleek-looking, functional dashboard, the real goal of this *Quick Success Data Science* project is to introduce you to the **Dash** library and methods for handling streaming data.

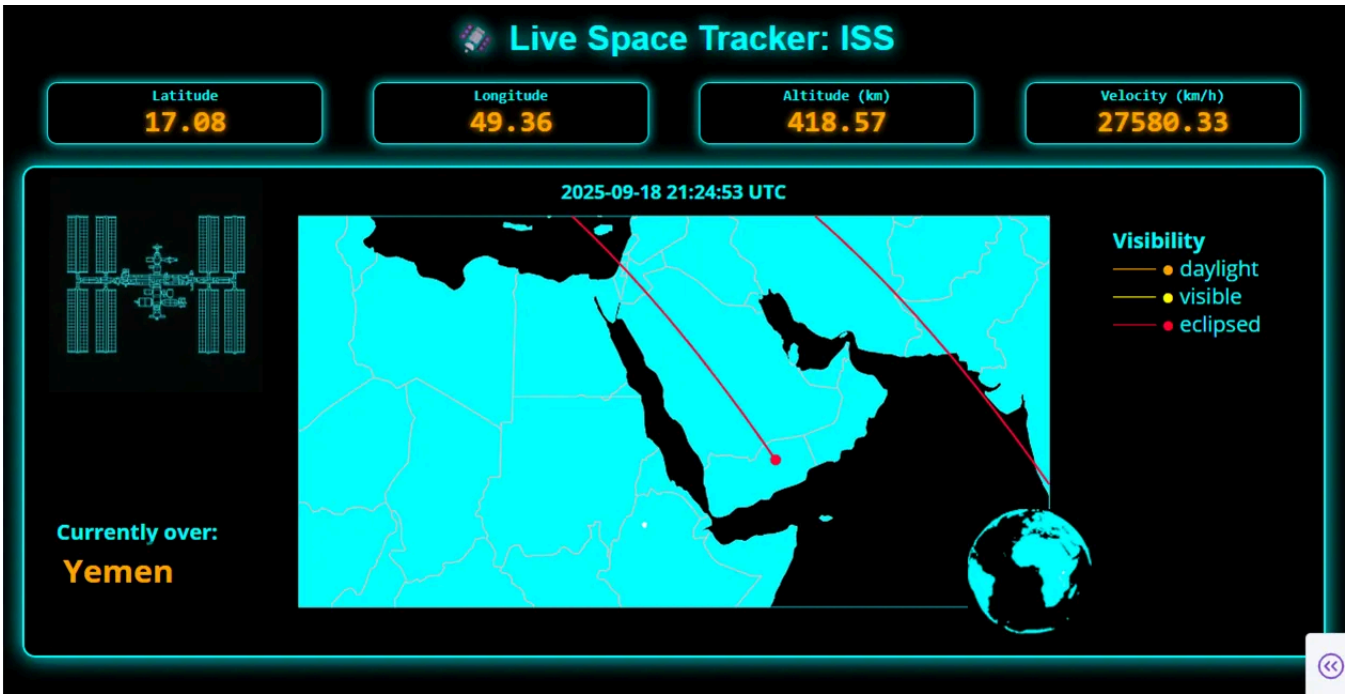
Dashboard Features

The dashboard provides telemetry readouts on the station's current latitude, longitude, altitude, and velocity. Also displayed are whether the station is experiencing day, night, or is illuminated during twilight (labeled "visible"). Other features include a schematic of the station, a readout of the country it's currently over, and a manually-rotatable globe.

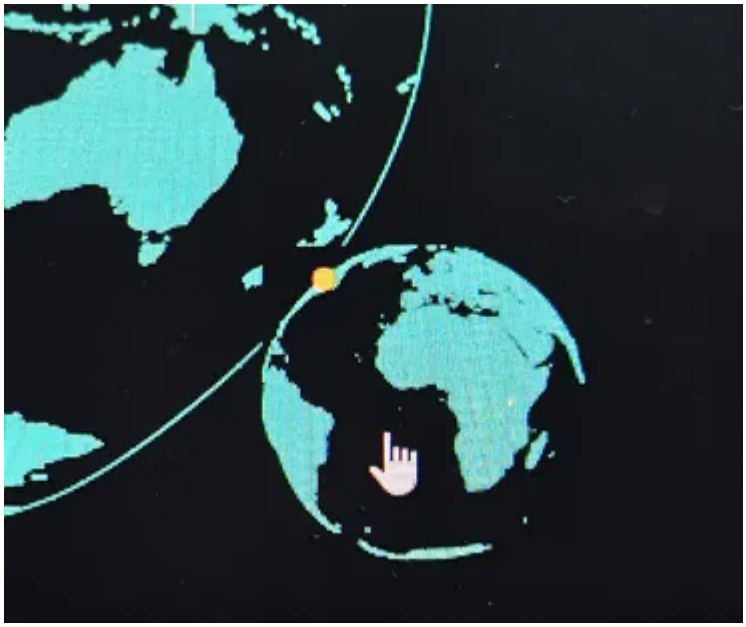
The graphical elements are built using the **Plotly** graphing library, which provides several dynamic features, such as hover data, the rotatable globe, and the ability to zoom the main map.



Hover the cursor over the orbital path to see high-precision coordinates (by author)



Use the mouse's scroll wheel to zoom the main map (by author)



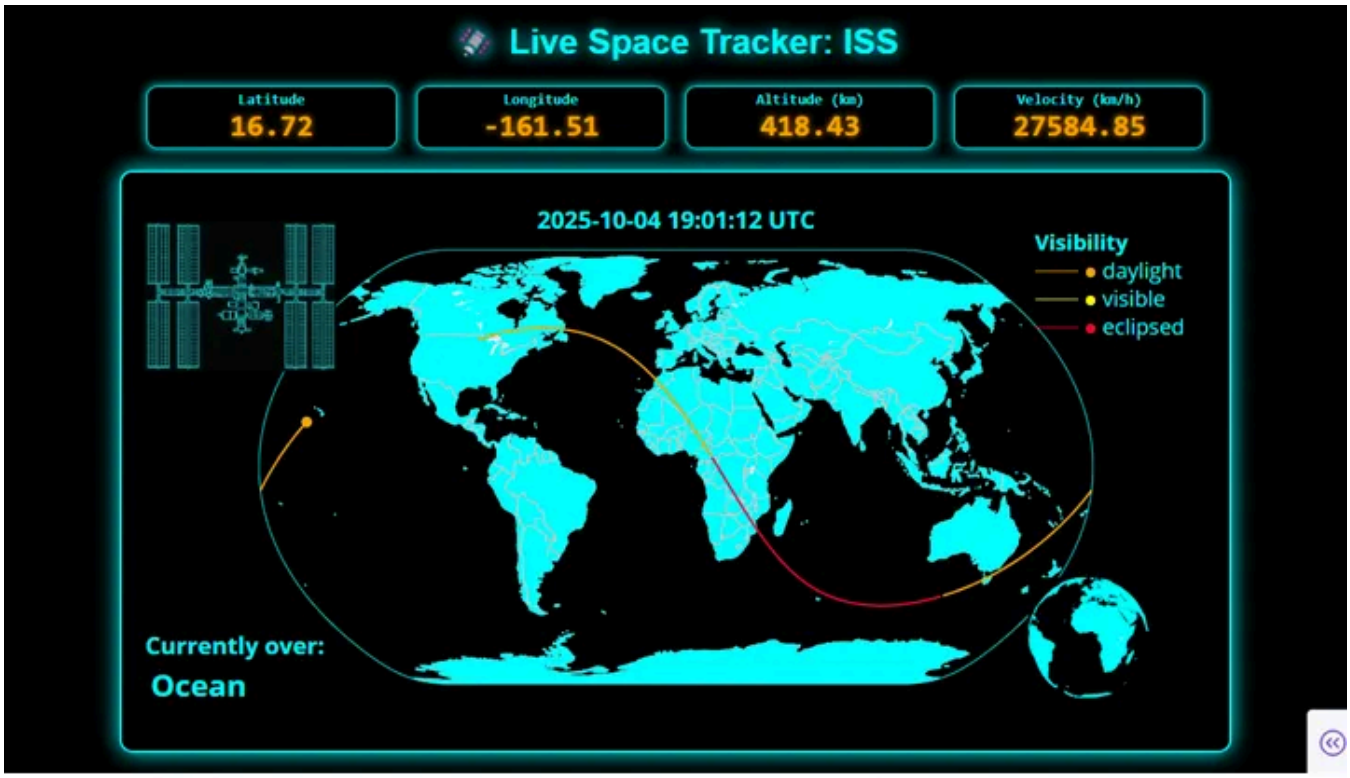
Rotate the globe by clicking MB1 and dragging in any direction (by author)

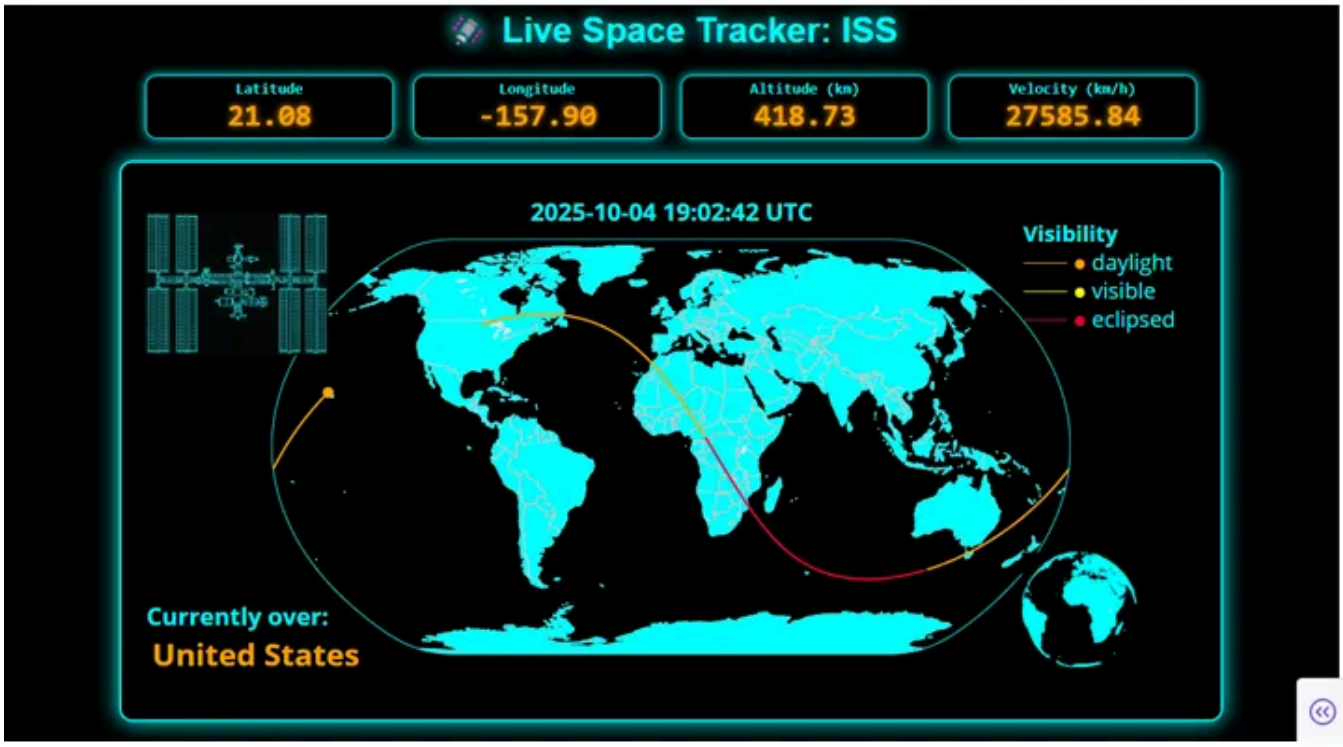
The line tracing the ISS represents its *ground track* over time. As we are plotting a sphere onto a 2D map, this circular path will appear as an S-shaped *sinusoid*. Here's a fun video explaining why.

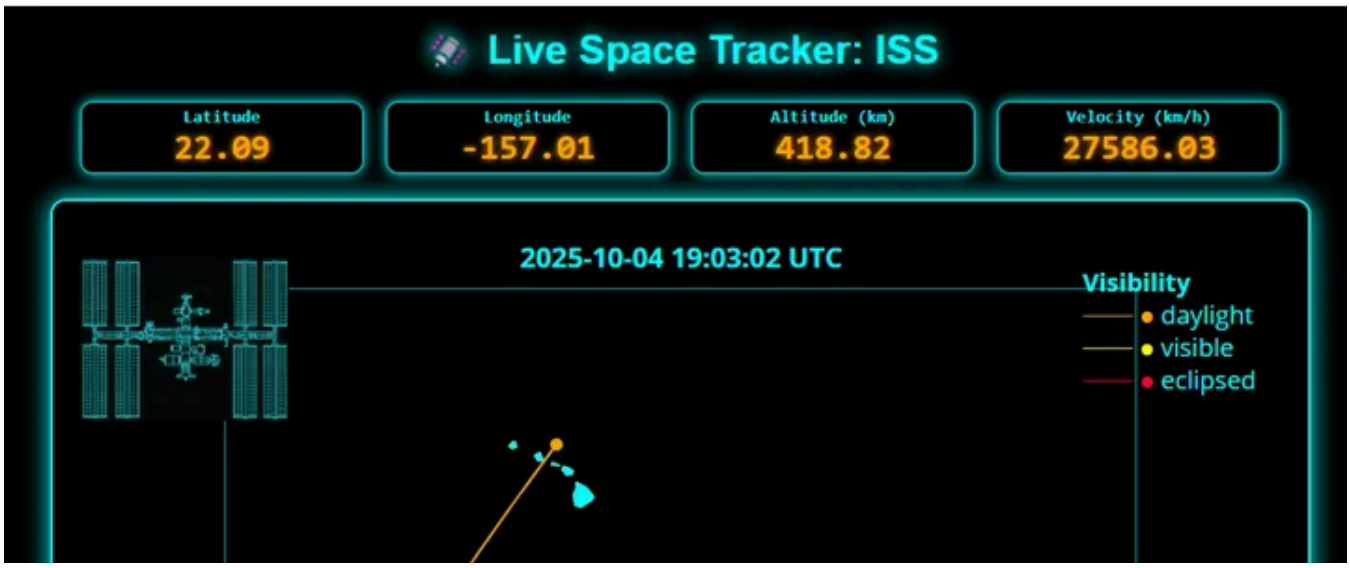
Fun with the Tracker

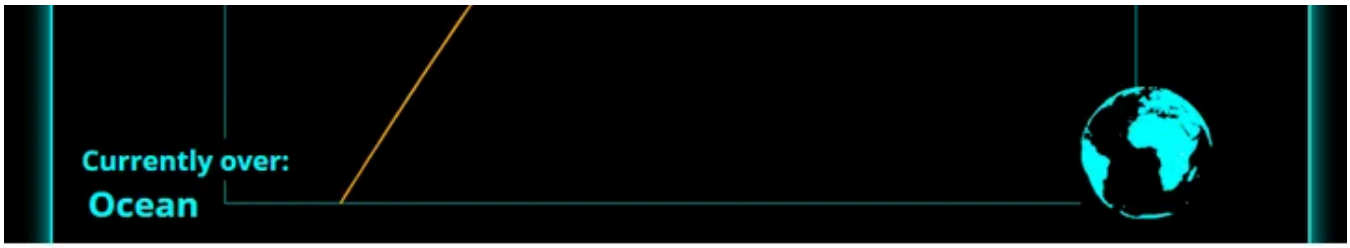
I like to keep the dashboard running when I'm working on other projects and check it periodically. It's oddly enjoyable when the ISS "finds a needle in a

haystack" and passes over a small island or an obscure geographical feature. In the example below, the track intersected both Tasmania and Hawaii in the same orbital pass.


Live Space Tracker: ISS
Latitude
16.72
Longitude
-161.51
Altitude (km)
418.43
Velocity (km/h)
27584.85
2025-10-04 19:01:12 UTC
Visibility
daylight
visible
eclipsed
Currently over:
Ocean

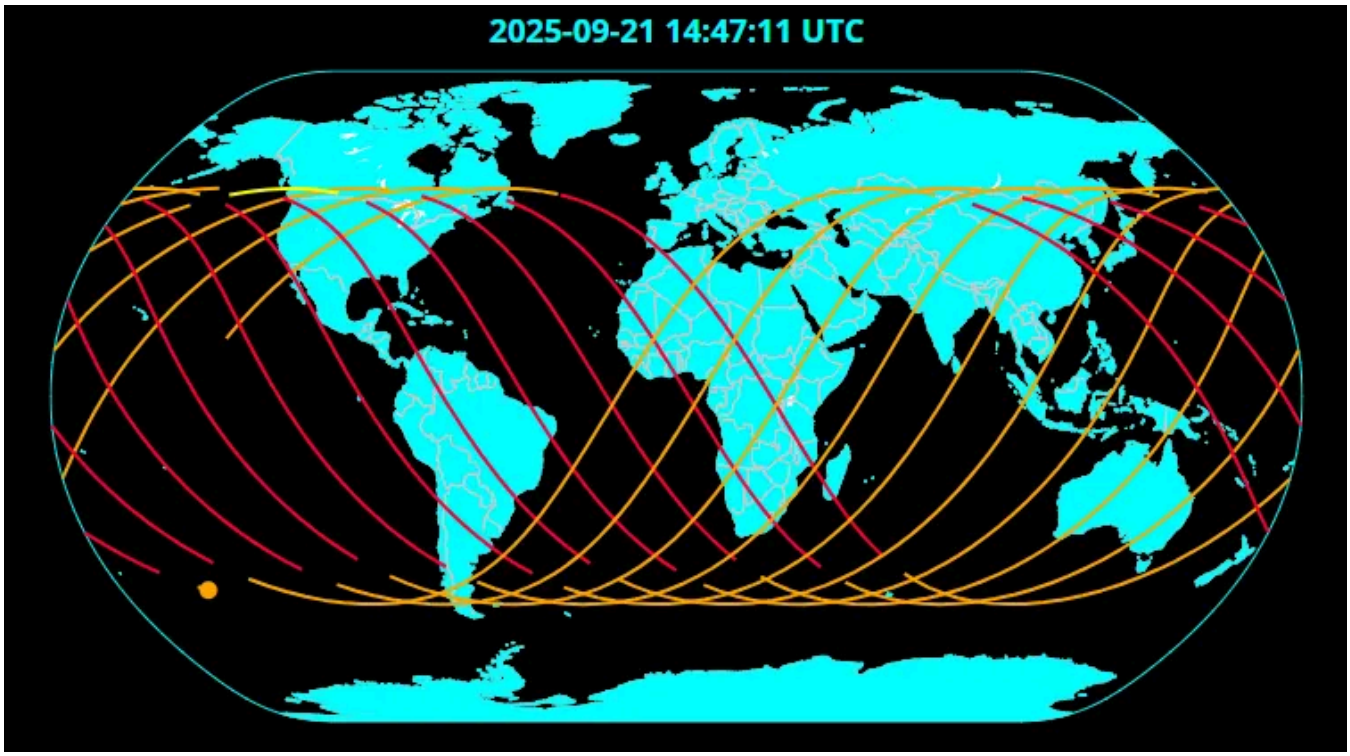

Live Space Tracker: ISS
Latitude
21.08
Longitude
-157.90
Altitude (km)
418.73
Velocity (km/h)
27585.84
2025-10-04 19:02:42 UTC
Visibility
daylight
visible
eclipsed
Currently over:
United States


Live Space Tracker: ISS
Latitude
22.09
Longitude
-157.01
Altitude (km)
418.82
Velocity (km/h)
27586.03
2025-10-04 19:03:02 UTC
Visibility
daylight
visible
eclipsed



Montage of the ISS bearing down on, and crossing over, the Hawaiian Islands (by author)

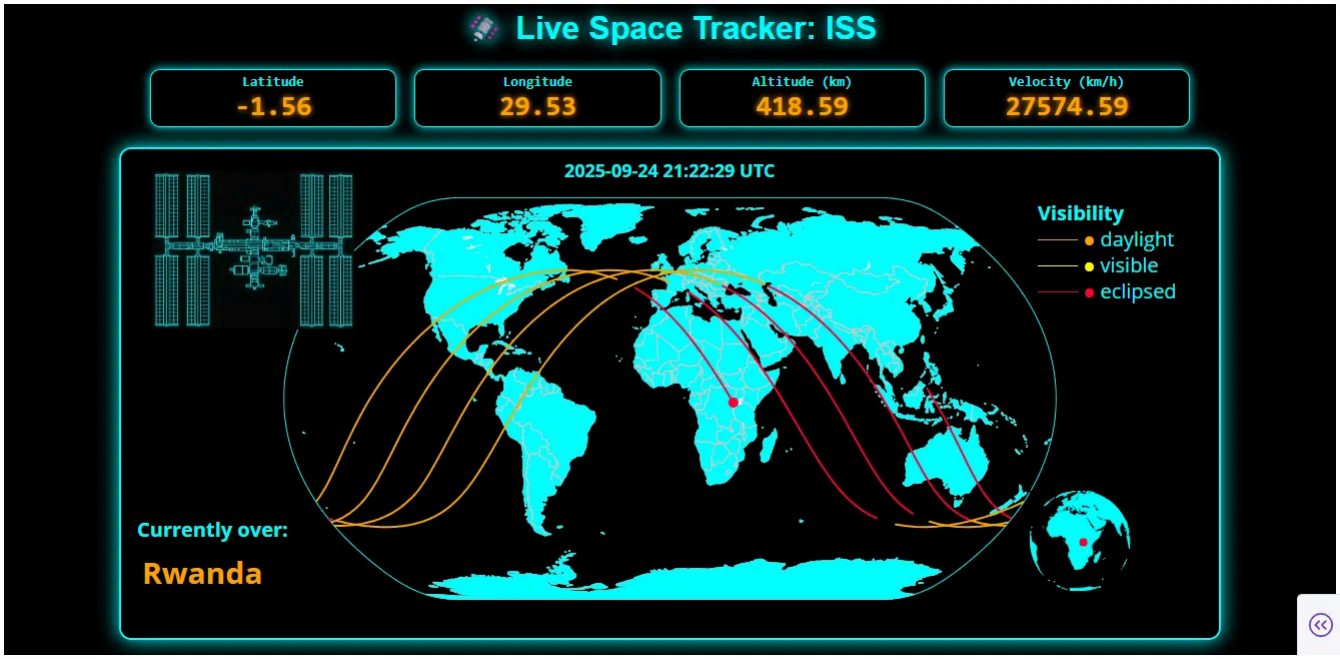
The code uses a `MAX_POINTS` constant for controlling the length of the track. Increasing it lets you record more orbits. This reveals the latitude limits of the station's orbit.



ISS ground tracks show day (orange) and night (red) distribution near the Autumnal Equinox (by author)

Generating these types of diagrams on the two equinoxes and two solstices will show variation ranges in day and night. The previous figure was recorded a day before the 2025 Autumnal Equinox. You can easily see the symmetry between day and night.

Because the tracker displays the name of the country the ISS is currently over, it can improve your knowledge of geography. Who doesn't want that?



The ISS passes over Rwanda (by the author)

Next, we'll take a quick look at **Dash** and then the **Python** script that makes it all work.

Python Dash

We'll create the dashboard with **Python Dash**, a framework for building web-based data applications. It allows you to create interactive, analytical dashboards and data visualizations entirely in **Python**.

Dash is an open-source library built on top of **Plotly.js** (for graphing), **React** (a front-end JavaScript library for building single-page applications and user interfaces), and **Flask** (a back-end **Python** web framework that handles the server-side logic of a web application, such as routing, database interactions, and user authentication). Essentially, **Flask** manages the data and the server, while **React**, running in the user's browser, manages how that data is displayed to the user.

Dash, known for its lightweight and flexible nature, provides a simple, high-level methodology for creating user interfaces for data science and machine learning applications. The framework's core components are:

- **Dash Core Components:** A set of pre-built, interactive components like sliders, dropdowns, and graphs that are used to build the dashboard's layout

and functionality.

- **Dash HTML Components:** Python classes that generate **HTML** for your layout, allowing you to structure your application using familiar **HTML** tags like `Div`, `H1`, and `P`.
- **Callbacks:** This is the reactive heart of **Dash**. Callbacks link user inputs (like a dropdown selection) to outputs (like a graph update), making the dashboard dynamic.

Dash *democratizes* web development for data scientists. Before **Dash**, building a web-based dashboard required expertise in multiple languages and technologies (such as **Python** for analysis, **JavaScript** for interactivity, and **HTML/CSS** for layout). Dash eliminates this barrier by allowing a single data scientist or analyst to build a sophisticated, production-ready web application using only **Python**.

> ***NOTE:*** *CSS refers to* Cascading Style Sheets. *This language lets you separate the HTML* content *from the* presentation *by defining styles, such as colors, spacing, fonts, and positioning, that control the layout and appearance of HTML elements on a webpage.*

With **Dash**, you can build and deploy complex web applications without leaving the **Python** ecosystem. This includes seamlessly integrating interactive plots from the **Plotly** library, using its component-based structure and reactive callbacks to quickly build and iterate on dashboards, and scale across small-scale projects and large enterprise applications.

Why Dash Fits This Project

Dash is a good choice for this project because:

- **Live updates are built in**
  The ISS tracker is all about *streaming* data. Dash's `dcc.Interval` and reactive callbacks make that trivial compared with rolling your own update loop in, say, **Flask** or plain **React**.
- **Plotly integration**
  Dash speaks **Plotly** *natively*, so you get rich interactive maps (`go.Scattergeo`, zoom, pan, hover) without extra **JavaScript**.

- **HTML/CSS freedom**
  All the glowing boxes, neon borders, and Orbitron fonts are just standard **CSS**. Dash doesn't limit you. You can drop any **CSS** rules into `style=` or a separate stylesheet. That's where we get the "sci-fi" look.

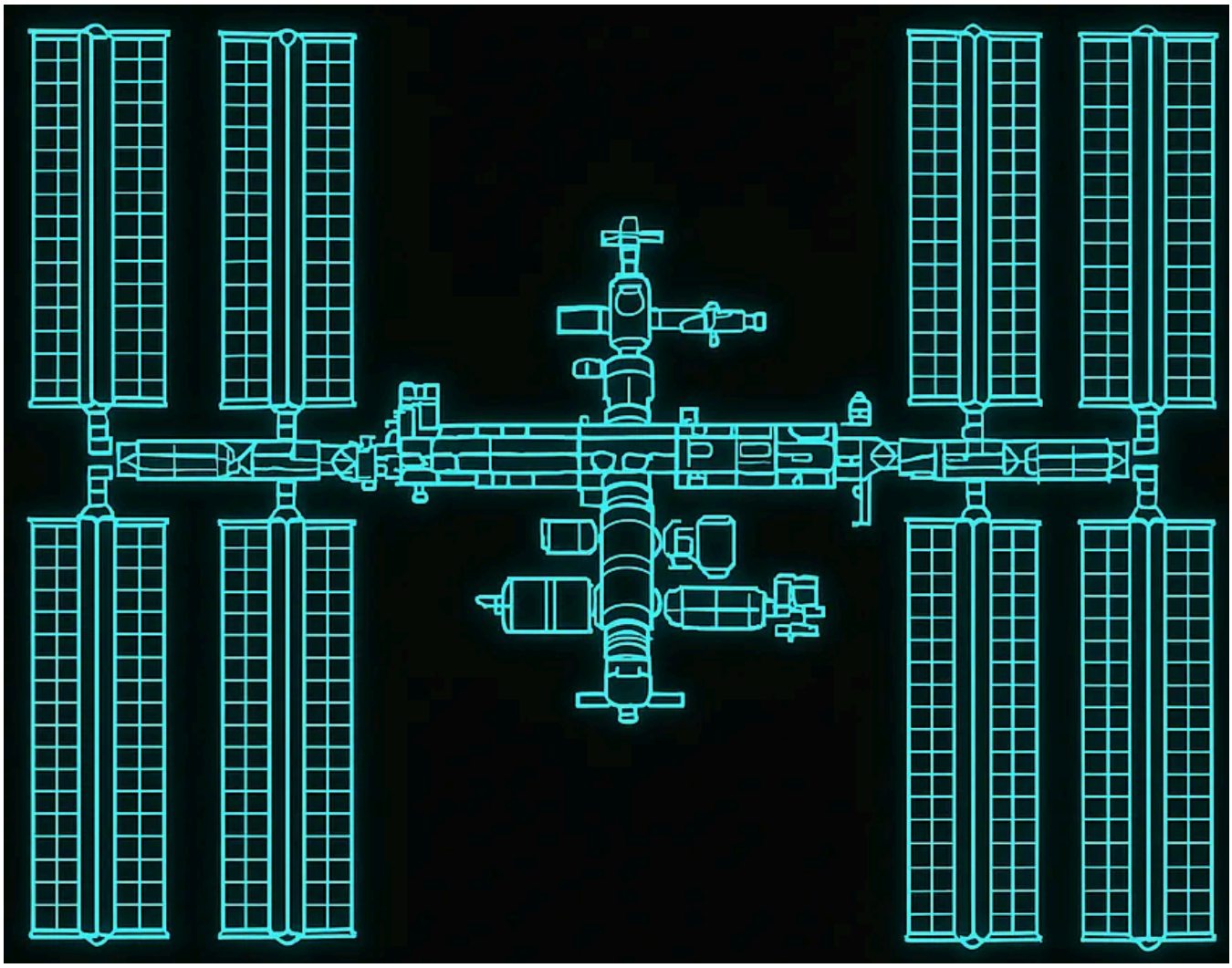
- **Single-language stack**
  Everything (data fetch, transformation, and UI) is Python. That's ideal for a data-science-leaning workflow.

The Code

You can download the dashboard code from this Gist or copy and paste it from the descriptive sections that follow. Be sure to name it *iss_dashboard.py*.

The required libraries are listed in the import section at the top of the code, or you can use the requirements file (*iss-dashboard.yml*) stored in this Gist.

You will also need a schematic of the ISS. Right-click on the image that follows and "Save As" *iss_schematic_blue.png*. Store it in the same folder as the **Python** script.



ISS schematic (iss_schematic_blue.png) by Copilot

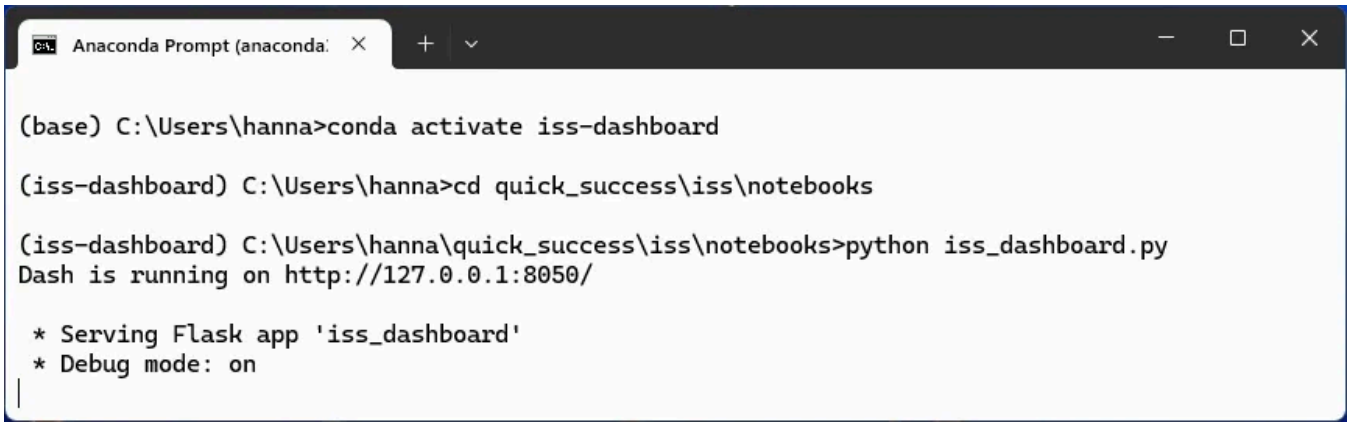
To run the code, open a prompt or terminal window and activate the appropriate virtual or conda environment. An example of a virtual environment (named `venv`) on Windows is: `venv\Scripts\activate`.

A MacOS/Linux example is: `source venv/bin/activate`.

For conda, you activate the environment directly: `conda activate venv`.

Once activated, enter the path to your script, and then run it as follows: `python iss_dashboard.py`.

Here's the full process on my computer using Anaconda with a conda environment named *iss-dashboard*; your path names will be different:



Running the dashboard script from a conda environment in an Anaconda Prompt window (by the author)

First, I activated the conda environment, then changed the directory (`cd`) to the location of the dashboard script and the ISS schematic image. I called the program using `python iss_dashboard.py`.

To launch the dashboard, in Windows, use **CTRL MB1(left click)** on the link *http://127.0.0.1:8050/*. With other operating systems, you may be able to click directly on the link without a modifier key.

To kill the process, enter **CTRL-C** in the prompt window and then close the browser tab containing the dashboard.

Below is a breakdown of the code by functional blocks:

Imports and Setup

The script starts by pulling in the core libraries:

- `datetime` for timestamps
- `base64` for embedding images directly into the dashboard
- `deque` for efficient rolling storage of ISS track points
- `requests` for API calls
- `pandas` for structuring telemetry data
- `pycountry` for country code to country name lookups
- `dash` & `plotly` for building the interactive web app

This establishes the toolkit for data handling and visualization.

```
from datetime import datetime
import base64
from collections import deque
import requests
import pandas as pd
import pycountry
from dash import Dash, dcc, html, Input, Output, State
from dash.exceptions import PreventUpdate
import plotly.graph_objects as go

# --- Environment Setup ---

# Load ISS schematic image:
# Ensure 'iss_schematic_blue.png' is in the same directory as this script.
encoded_image = ""
try:
    with open("iss_schematic_blue.png", "rb") as f:
        encoded_image = base64.b64encode(f.read()).decode()
except FileNotFoundError:
    print("Warning: iss_schematic_blue.png not found. Image will not display.

# Constants:
ISS_URL = "https://api.whereistheiss.at/v1/satellites/25544"
color_map = {"daylight": "#FFA500", "visible": "#FFFF00", "eclipsed": "#FF00
MAX_POINTS = 1080   # Keeps most recent 1080 track points (~2 orbits)
TRACK_LAT = deque(maxlen=MAX_POINTS)
TRACK_LON = deque(maxlen=MAX_POINTS)
TRACK_VIS = deque(maxlen=MAX_POINTS)
```

We start by loading the ISS schematic image. Using `base64` encoding makes the image web-safe, embeddable, and portable.

Dash and **Plotly** don't serve static files by default. If you want to include an image in the layout, you can't simply reference a local file path, such as `"iss_schematic_blue.png"` —especially if you're deploying or sharing the app. Base64 turns binary image data into a string that can be embedded directly into the **Plotly** figure using a `...` URI (Uniform Resource Identifier). That means the image is part of the layout itself—no external hosting, no file serving, and no broken links. This makes the dashboard easier to deploy, share, or run in environments like Jupyter or Colab, where file access may be limited.

We finish by assigning some constants. The first is the *web address* of the ISS data source.

While there are numerous sources for ISS telemetry, we'll use the *WTIA REST API* (WTIA stands for *Where the ISS at?*). This API, written by Bill Shupp, includes more features than are provided by other ISS tracking/notification sites. It provides s*treaming data*, which refers to real-time data that is continuously flowing from a source to a target.

Next, we assigned a Python dictionary of colors to use for the ground track. The ISS appears orange when visible during daylight and red when eclipsed by the Earth. Yellow is used for "visible," when the sky is dark but the ISS — due to its altitude — is still illuminated. This represents the time when the station can be seen with the naked eye. Unfortunately, the telemetry does not always return this code, so the tracks will mainly be either orange or red.

> ***NOTE:*** *The ground track will sometimes include gaps. These are mainly a result of the "visible" code not being returned in the telemetry or the connection with the API being temporarily dropped.*

The `MAX_POINTS = 1080` constant determines the *length of the tracks*. This ensures the dashboard shows a smooth, recent orbital path without growing indefinitely or using excess memory. Because we'll be receiving data every 10 seconds, this means we'll record 1080 x 10 = 10,800 seconds (180 minutes) worth of data. As the ISS makes a complete orbit every 90 minutes, this represents two orbits.

> ***NOTE:*** *If you change the sampling frequency, you'll need to adjust the MAX_POINTS constant to preserve the same track length.*

The final three constants record the pertinent information (latitude, longitude, and visibility) used to draw the tracks. We use `deque` (short for *double-ended queue*), which is a special type of list optimized for fast appends and pops from both ends. It's like a smart, self-cleaning list ideal for live dashboards and telemetry that need a rolling history.

Define Utility Functions

The next snippet defines utility functions used to run the dashboard. Together, these functions form the *data pipeline*: one ensures you always have fresh,

structured telemetry, while the other organizes that data into meaningful chunks for color-coded plotting.

```
# --- Utility Functions ---

def get_iss_telemetry(url=ISS_URL):
    """Fetches current ISS telemetry data from the API."""
    try:
        resp = requests.get(url, timeout=10)
        resp.raise_for_status()
        data = resp.json()
        df = pd.DataFrame([data])
        # Drop unnecessary columns:
        df.drop(["id", "footprint", "daynum", "solar_lat", "solar_lon"],
                errors="ignore", axis=1, inplace=True)
        return df
    except (requests.Timeout, requests.ConnectionError) as e:
        print(f"⚠️ Network issue: {e}")
    except requests.HTTPError as e:
        print(f"⚠️ API returned {resp.status_code}: {e}")
    return None


def split_segments(df):
    """Splits the track DataFrame into segments based on visibility type."""
    if df.empty:
        return []
    segments, start = [], 0
    for i in range(1, len(df)):
        if df["vis"].iloc[i] != df["vis"].iloc[i-1]:
            segments.append(df.iloc[start:i])
            start = i
    segments.append(df.iloc[start:])
    return segments
```

The first function (`get_iss_telemetry()`) fetches the latest ISS position and velocity data from the public API. It first makes a network request with error handling for timeouts, connection issues, or bad responses. It then converts the **JSON** payload into a tidy **Pandas** DataFrame, dropping extra fields that aren't needed for visualization. The function finishes by returning either a clean, ready-to-use DataFrame or `None` , if the request fails.

The next function (`split_segments()`) prepares the ISS orbital path for plotting. This involves iterating through the DataFrame of track points and splitting the

path into segments whenever the *visibility state* (daylight, visible, eclipsed) changes. The outcome is a list of DataFrame slices, each representing a continuous segment of the orbit with a consistent visibility type.

Dash Application Setup

The **Dash** app is the container that will hold all the layout elements and callbacks. This code defines the *look and structure* of the dashboard by setting up the title, glowing readout panels, the central map, and the update mechanism.

```
# --- Dash Application Setup ---

app = Dash(__name__)
app.title = "Live ISS Tracker"

# Styles for readouts:
readout_style = {
    "border": "1px solid #0ff",
    "backgroundColor": "black",
    "color": "#0ff",
    "padding": "4px 6px",
    "borderRadius": "10px",
    "width": "18%",
    "boxShadow": "0 0 10px #0ff",
    "textAlign": "center",
    "fontFamily": "Consolas, monospace",
    "lineHeight": "1.1",
}
value_style = {
    "fontSize": "28px",
    "fontWeight": "bold",
    "color": "#ffa500",
    "textShadow": "0 0 6px #ffa500",
    "margin": "0",
}
label_style = {"fontSize": "14px", "margin": "0 0 2px 0", "color": "#0ff"}

app.layout = html.Div(
    style={
        "fontFamily": "Orbitron, Arial, sans-serif",
        "backgroundColor": "black",
        "color": "#0ff",
        "minHeight": "100vh",
        "padding": "20px",
    },
    children=[
        html.H1(
            "🛰️ Live Space Tracker: ISS",
            style={"textAlign": "center", "color": "#0ff",
```

```
                    "textShadow": "0 0 15px #0ff"}),
            html.Div(
                style={
                    "display": "flex",
                    "justifyContent": "center",
                    "gap": "18px",
                    "flexWrap": "wrap",
                    "margin": "20px 0"
                },
                children=[
                    html.Div([html.H4("Latitude", style=label_style),
                              html.P(id="lat-box", style=value_style)], style=rea
                    html.Div([html.H4("Longitude", style=label_style),
                              html.P(id="lon-box", style=value_style)], style=rea
                    html.Div([html.H4("Altitude (km)", style=label_style),
                              html.P(id="alt-box", style=value_style)], style=rea
                    html.Div([html.H4("Velocity (km/h)", style=label_style),
                              html.P(id="vel-box", style=value_style)], style=rea
            ),
            dcc.Graph(
                id="iss-map",
                style={
                    "border": "2px solid #0ff",
                    "borderRadius": "12px",
                    "boxShadow": "0 0 20px #0ff",
                    "height": "75vh",
                    "width": "85%",
                    "margin": "0 auto",
                },
            ),
            # dcc.Store is used for maintaining the track history between updates
            dcc.Store(id="track-store", data={"lat": [], "lon": [], "vis": []}),
            # Interval set to 10 seconds (10 * 1000 ms):
            dcc.Interval(id="interval", interval=10 * 1000, n_intervals=0),
        ],
    )
```

First, we create the *Dash app instance* and set the browser tab title to *“Live ISS Tracker.”* Then we define three reusable style dictionaries:

- `readout_style` : neon-framed boxes for displaying live values.
- `value_style` : large, glowing orange numbers for telemetry readouts.
- `label_style` : smaller descriptive labels above each value.

The overall theme uses a retro-futuristic, neon-on-black aesthetic.

Next, we set up the *layout structure*. We build this with `html.Div`.

> ***NOTE:*** *The HTML* `<div>` *(division) element is a generic, block-level container used to group and structure other elements. It has no inherent semantic meaning and is primarily used as a hook for applying styles with CSS or manipulating sections of a document with JavaScript. The* `<h1>` *element is the most important heading tag, used to define the main heading or title of an entire web page or a major content section. It's a block-level element, crucial for SEO (Search Engine Optimization) and for providing clear document structure and hierarchy to both users and screen readers. There should typically be only one* `<h1>` *per page.*

The HTML divisions include:

- **Title Header** : a glowing `<h1>` element with the ISS emoji. This `<h1>` element represents the primary heading of the webpage.
- **Telemetry Readouts** : four styled boxes showing latitude, longitude, altitude, and velocity.
- **Main Map** : a large `dcc.Graph` component where the ISS orbit and position will be plotted on a Mercator projection of the Earth.
  The `dcc.Graph` component is part of the **Dash Core Components** library in **Python**, used to render interactive data visualizations powered by **Plotly**.
- **Hidden Storage** : a `dcc.Store` to keep track of orbital history between updates. This provides persistence across callbacks and user sessions in a web app context.
- **Interval Timer** : a `dcc.Interval` set to 10 seconds, triggering periodic data refreshes. This "gently" pings the ISS public website for updates to the ISS location.

The app is now "visually scaffolded" and ready to be wired to live data through callbacks.

Callbacks and Figures

The next snippet makes the calls to run the dashboard. At a high level, the `update_map()` function is the engine that powers the whole thing. Everything we set up before, imports, constants, utility functions, and **Dash** layout,

is *scaffolding* that defines *what the app looks like* and *what data it needs*. The following `update_map()` function is where those pieces come together:

- **Trigger point:** tied to the `dcc.Interval`, so it runs automatically every 10 seconds.
- **Data pipeline:** calls the telemetry function to fetch fresh ISS data, then updates the rolling history of positions.
- **Logic layer:** enriches that data (e.g., country lookup, visibility state) and organizes it for plotting.
- **Visualization:** builds a new **Plotly** figure with the orbital track, current ISS marker, inset globe, and annotations.
- **UI updates:** pushes the latest numbers into the readout boxes and refreshes the hidden store that maintains history.

In the bigger picture:

- **Layout** = the dashboard's skeleton and style.
- **Utility functions** = the data prep tools.
- `update_map()` = the heartbeat that keeps the dashboard accurate and dynamic. Without it, the app would be just a static page. With it, you get a continuously updating, Sci-Fi-styled ISS tracker that feels alive.

```
# --- Callback for Updating Map and Readouts ---

@app.callback(
    [
        Output("iss-map", "figure"),
        Output("lat-box", "children"),
        Output("lon-box", "children"),
        Output("alt-box", "children"),
        Output("vel-box", "children"),
        Output("track-store", "data"),
    ],
    Input("interval", "n_intervals"),
    State("track-store", "data")
)
```

```
    def update_map(_, track_data):
        """Fetches new data, updates track, and generates the map figure."""
        df = get_iss_telemetry()
        if df is None:
            raise PreventUpdate

        lat = float(df["latitude"].iloc[0])
        lon = float(df["longitude"].iloc[0])
        alt = float(df["altitude"].iloc[0])
        vel = float(df["velocity"].iloc[0])
        vis = df["visibility"].iloc[0]
        current_time = datetime.utcnow().strftime("%Y-%m-%d %H:%M:%S UTC")

        # Country lookup
        country_name, country_color = "Ocean", "#0ff"
        try:
            coord_url = f"https://api.wheretheiss.at/v1/coordinates/{lat},{lon}"
            resp2 = requests.get(coord_url, timeout=6)
            resp2.raise_for_status()
            cdata = resp2.json()
            code = cdata.get("country_code")
            if code and code != "??":
                match = pycountry.countries.get(alpha_2=code.upper())
                country_name = match.name if match else code
                country_color = "#ffa500"
        except requests.RequestException:
            pass # Silently fail on API or network errors

        # Append into server-side deques (these enforce MAX_POINTS automatically)
        # The initial check for track_data being None is no longer strictly neede
        # since the deques are global and the dcc.Store data is now only used
        # for return compatibility.
        TRACK_LAT.append(lat)
        TRACK_LON.append(lon)
        TRACK_VIS.append(vis)

        # Build the dataframe from the deques (already trimmed)
        track_df = pd.DataFrame({
            "lat": list(TRACK_LAT),
            "lon": list(TRACK_LON),
            "vis": list(TRACK_VIS)
        })

        # Prepare data for dcc.Store update
        track_data = {"lat": list(TRACK_LAT), "lon": list(TRACK_LON), "vis": list

        fig = go.Figure()

        # Main map track
        for seg in split_segments(track_df):
            vtype = seg["vis"].iloc[0]
            if vtype in color_map:
                fig.add_trace(go.Scattergeo(
```

```
                lat=seg["lat"],
                lon=seg["lon"],
                mode="lines",
                line=dict(width=2, color=color_map[vtype]),
                showlegend=False,
                geo="geo"
            ))

    # Current ISS marker (MAIN map) — synchronized color
    current_color = color_map.get(vis, "#FFFFFF")
    fig.add_trace(go.Scattergeo(
        lat=[lat], lon=[lon],
        mode="markers",
        marker=dict(size=10, color=current_color),
        showlegend=False,
        geo="geo"
    ))

    # Inset globe marker — uses the SAME color variable
    fig.add_trace(go.Scattergeo(
        lat=[lat],
        lon=[lon],
        mode="markers",
        marker=dict(size=8, color=current_color),
        showlegend=False,
        geo="geo2"
    ))

    legend_html = (
        "<b>Visibility</b><br>"
        f"<span style='color:{color_map['daylight']}'>━━ ●</span> daylight<br>"
        f"<span style='color:{color_map['visible']}'>━━ ●</span> visible<br>"
        f"<span style='color:{color_map['eclipsed']}'>━━ ●</span> eclipsed"
    )

    fig.update_layout(
        images=[dict(
            source=f"{encoded_image}",
            xref="paper", yref="paper",
            x=-0.01, y=0.99,
            sizex=0.30, sizey=0.30,
            xanchor="left", yanchor="top",
            layer="above"
        )],
        annotations=[
            dict(text="<b>Currently over:</b>", x=-0.01, y=0.11,
                 xref="paper", yref="paper", showarrow=False,
                 font=dict(size=22, color="#0ff"),
                 bgcolor="rgba(0,0,0,1)", borderpad=1),
            dict(text=f"<b>{country_name}</b>", x=-0.01, y=0.01,
                 xref="paper", yref="paper", showarrow=False,
                 font=dict(size=28, color=country_color),
                 bgcolor="rgba(0,0,0,0)", borderpad=6),
```

```
                dict(text=f"<b>{current_time}</b>", x=0.5, y=1.03,
                     xref="paper", yref="paper", showarrow=False,
                     font=dict(size=22, color="#0ff")),
                dict(text=legend_html, x=0.99, y=0.98,
                     xref="paper", yref="paper", showarrow=False,
                     align="left", font=dict(size=20, color="#0ff")),
            ],
            paper_bgcolor="rgba(0,0,0,0)",
            plot_bgcolor="black",
            margin=dict(l=30, r=30, t=40, b=30),
            geo=dict(
                domain=dict(x=[0.068,  0.932],  y=[0.068,  0.932]),
                projection_type="natural earth",
                showland=True, landcolor="#0ff",
                showcountries=True, countrycolor="#CCCCCC",
                showcoastlines=True, coastlinecolor="#0ff",
                bgcolor="black", showframe=True, framecolor="#0ff",
            ),
            geo2=dict(
                domain=dict(x=[0.79, 1.0], y=[0.04, 0.28]),
                showland=True, landcolor="#0ff",
                showcountries=False,
                showcoastlines=True, coastlinecolor="#0ff",
                projection_type="orthographic",
                bgcolor='black',
                showframe=False
            )
        )

        # Return values for the six Outputs
        return (fig, f"{lat:.2f}", f"{lon:.2f}", f"{alt:.2f}",
                f"{vel:.2f}", track_data)
```

Update Cycle: In the *update cycle*, a single **Dash** *callback* drives the live updates. It's triggered every 10 seconds by the `dcc.Interval` component and has six outputs: the map figure, four telemetry readouts (lat, lon, alt, vel), and the stored track history.

Data Refresh: The code refresh is built into the big `update_map()` function that starts by calling the `get_iss_telemetry()` function to fetch the latest ISS position, altitude, velocity, and visibility. It then performs a *country lookup* using a secondary API (`pycountry`), so the dashboard can display whether the ISS is over land or ocean. It appends the new point into global deques (`TRACK_LAT` , `TRACK_LON` , `TRACK_VIS`), which automatically trim to the last 1,080

samples (~2 orbits). You can alter this value to shorten or lengthen the displayed orbital tracks.

Build the Figure: The function creates a new **Plotly** `Figure` each cycle. It calls the `split_segments()` function to color-code the track by visibility (daylight, visible, eclipsed), then adds a glowing marker on both the main map and the inset globe, synchronized by visibility color. The circular marker represents the station's current location.

Rather than let **Plotly** rebuild the legend (and determine the current visibility state with each refresh), we use a more efficient *custom* **HTML** legend for visibility states. We use annotations for the current country, timestamp, and labels, and embed the ISS schematic image in the upper left corner.

Layout Styling: The main map uses a "natural earth" projection with neon coastlines and country outlines. The inset globe uses an orthographic projection to provide a 3D-like view of the Earth. This isn't necessary but it gives the dashboard more of a Sci-Fi aesthetic. It also provides the user with an *interactive* feature, as they must *manually* rotate the globe with the cursor to find the ISS when it's in the Western Hemisphere.

Outputs: The callback returns:

- The updated *map figure*.
- Formatted *latitude*, *longitude*, *altitude*, and *velocity* values for the readout boxes
- The updated *track history* for persistence

Running the Application

The script ends with the standard **Python** entry point. This final block is the switch that turns on everything. After setting up the data pipeline, layout, and callbacks, this line runs the application, allowing you to interact with it.

```
if __name__ == "__main__":
    # Standard way to run a Dash application from a command line
    app.run(debug=True)
```

This final snippet ensures the **Dash** app only launches when the script is run *directly* (not when it's imported as a module). It starts a local development server, which opens the interactive ISS dashboard in your browser.
With `debug=True`, the app automatically reloads when you make code changes and provides helpful error messages during development. This is a handy way to get instant feedback when tweaking the layout.

> ***Tip:*** *To see the full 90-minute orbital track of the ISS, you'll need to adjust your computer's sleep cycle so that it stays awake for at least 90 minutes. For different times, use the equation* `MAX_POINTS * timeout` *(from the* `get_iss_telemetry()` *function) to calculate the required number of seconds to stay awake.*

Expanding the Dashboard

Here are some additional projects to make the dashboard your own.

Calculating the Percent Ocean vs. Land

Water covers 71% of the Earth's surface. Although the ISS doesn't pass over the *entire* planet, it would be interesting to use the number of times the dashboard returns "Ocean" to see how close it gets to that value. You would want to record multiple orbits as the ISS's track shifts with each orbit.

Predicting the Position of the ISS

It's possible to both see and photograph the ISS as it passes over your location at twilight. To be ready for the event, refactor the code to predict the ISS's track a few hours ahead. Display the predicted track as a dashed line.

Alternate Projects

Here are some suggestions for other space-related dashboards:

Exoplanet Data

Several public databases contain detailed information about confirmed exoplanets, including their orbital characteristics, discovery methods, and host star properties.

- **NASA Exoplanet Archive**: This is one of the most comprehensive public databases for exoplanet data. It's a goldmine for anyone looking to analyze and visualize exoplanet catalogs. You can access data on confirmed planets, Kepler and K2 mission candidates, and stellar properties.
- **Open Exoplanet Catalogue**: A community-maintained database that aggregates data from various scientific papers and public sources. The data is available in a machine-readable XML format, making it easy to parse for dashboard applications.
- **Exoplanet-related APIs**: Many of these databases offer APIs, allowing you to fetch data programmatically for a dynamic dashboard that stays up to date with discoveries.

Orbital Debris and Satellite Tracking Data

Data on orbital debris and active satellites is also publicly available and regularly updated. This is critical for creating dashboards that track objects in Earth's orbit.

- **Space-Track.org**: The U.S. Space Force operates this official source for two-line element (TLE) data, which describes the orbits of thousands of objects, including active satellites and debris. To access the data, you must register for a free account.
- **Celestrak**: This is a great alternative to *Space-Track.org*. It provides TLE data in various formats and is a widely trusted source for satellite and debris information, often used for hobbyist and academic projects.
- **European Space Agency (ESA) Space Debris Office**: The ESA provides information and statistics on orbital debris, including regular reports and data on re-entries and collision risks.
- **Chinese Space Station Tiangong:** Track the *Tiangong*, China's version of the ISS. You can see an example here.

You can view a nice example of a satellite-tracking dashboard here. Other space-related dashboards can be found at Plotly Dash App Examples.

For a simplified ISS tracker that uses the **Plotly Express** library, visit this article:

Plot Streaming Data with Python and Plotly Express

Tracking the ISS in real time

medium.com

Alternatives to Dash

Python comes with other dashboard-building packages besides **Dash**. Here's a list of some alternatives you could use:

- **Streamlit** — simpler to learn than **Dash,** but customizing complex interactive graphics or pixel-perfect **CSS** is trickier.
- **Panel + Bokeh** — powerful but requires more manual styling for a "neon dashboard" vibe.
- **Full JS (React/Three.js)** — ultimate visual control, but much heavier for a Python-first project.

Summary

In this project, we used **Python**, **Dash**, and **Plotly** to build a sleek dashboard that looked like it belonged on a spaceship — and actually tracked one! The futuristic look isn't a **Dash** feature; it's our **CSS** choices combined with **Plotly**'s crisp rendering. **Dash** simply provided us with the easiest **Python** path to live telemetry, along with complete front-end control, which is exactly what we needed.

Now that you've got a feel for **Dash**, you can use the vast amount of public data available on exoplanets, satellites, and orbital debris to create other space-themed dashboards. As the saying goes, the sky's the limit!

Thanks!

Thanks for reading, and please follow me for more *Quick Success Data Science* projects in the future.

Python Programming | Dash | Dashboard | Plotly | Internationalspacestation

Following

Published in Data Science Collective

874K followers · Last published 4 hours ago

Advice, insights, and ideas from the Medium data science community

Follow

Written by Lee Vaughan

3.6K followers · 466 following

Author of “Python Tools for Scientists,” “Impractical Python Projects,” and “Real World Python.” Former Senior Principal Scientist for ExxonMobil.