International Space Station Tracker App

Bethany Grimm

March 14, 2025

1 Introduction

This project uses data from the International Space Station to track its position and velocity at given times. NASA publishes ISS data multiple times a week, which displays the projected positions and velocities of the ISS in four-minute increments in a multiple-week timeframe. However, this is a huge amount of data to parse through manually. This software package is designed to parse through this satellite data and return useful data and computations from the dataset. In addition, modularity and portability were important considerations when designing this software package. Hence, the software package is containerized with Docker to make it simple to run on different kinds of machines.

2 Software Used

This project was programmed in Python. Packages such as Astropy, Requests, and Redis were also installed to facilitate computations. Among the most important installations are:

- Docker - Docker allows code to be containerized. Since different machines have different specifications and installations, they may not necessarily be able to run the same program in the same way. Docker creates a virtual container from a base image that attaches to the user's machine, and includes all the files and installations needed to run the program as intended.
- Flask - Flask is an application programming interface (or API), and in this case, allows the user to communicate with the program. This application allows the user to interactively enter routes such as `curl localhost:5000/epochs` to retrieve desired data.
- Redis - Redis is a persistent database, and in this program stores the a copy of the NASA ISS data to the user's local machine. This way, the database will not vanish when the container is exited.

3 Files and Setup

This project contains the following files:

- `Dockerfile` - builds the containers needed to run this software package
- `docker-compose.yml` - facilitates running the containers
- `requirements.txt` - the list of dependencies required to run the containers
- `iss_tracker.py` - houses all the functions used to analyze data. This script also initializes the Flask server and Redis object used for user communication with the program and accessing the database, respectively
- `test_iss_tracker.py` - contains unit tests for `iss_tracker.py`

The program automatically extracts ISS trajectory data from NASA's website and backs up this data to the user's local machine.
Since the code is containerized, no installations are required - just a machine that is capable of running Docker containers. The user must navigate to the working directory containing the files `Dockerfile`, `docker-compose.yml`, `requirements.txt`, `iss_tracker.py`, and `test_iss_tracker.py`. At this point, the directory `data` must be created manually by the user; otherwise, Redis will lack the permissions to save data on the user's local machine. Then, the user must build the container with the command `docker build -t [username]/iss_tracker:1.0 ./`, in which `[username]` is replaced with the user's own username. This builds the Flask app.
Since there are two containers - the Redis database and the Flask app - that need to be coordinated, `docker-compose.yml` will facilitate building images of both containers. The user must first edit `docker-compose.yml` such that instances of `[username]` are replaced with their own username. The user may run the containers with the command `docker compose up -d`. The containers run in the background, leaving the user's terminal free to run commands. During this process, the local folder `data` is mounted to the Redis container, allowing the container to back up the data to the user's local machine.
After all the desired scripts have been run, the user may use the command `docker compose down` to stop and remove the running containers.

4 Running Scripts and Unit Tests

Before running the scripts and unit tests, it is critical that the Flask and Redis containers have been initialized and are running in the background. The supported curl commands and their functionality are described as follows:

- `curl localhost:5000/epochs`
  - This route returns the entire dataset in JSON format. Using this route is not recommended, as there are several thousand state vectors'

worth of ISS data, and this route displays them all directly on the screen.

- `curl "localhost:5000/epochs?limit=[limit]&offset=[offset]"`
  - `[limit]` and `[offest]` must both be integers. This route returns `offset` amount of epochs starting from index `limit`. To get a better sense of the range of data and the number of indices, the route `curl localhost:5000/range` is also included. This is recommended for viewing the state vectors for several epochs without returning the entire dataset.
- `curl localhost:5000/[epoch]`
  - `[epoch]` must be a string, the epoch queried. This route returns all state vectors for the epoch requested (or the closest one). A rather specific string format is needed: %Y-%jT%H:%M:%S.%fZ, that is [year]-[days in year]T[hour]:[minute]:[second].[microsecond]Z (example "2025-32T12:00:00.000Z" for precisely 12:00 on February 1, 2025). The output of `curl localhost:5000/epochs` also gives examples of this epoch format for easier reference.
- `curl localhost:5000/[epoch]/speed`
  - `[epoch]` must be a string, the epoch queried. This route returns the instantaneous speed for the epoch requested. This must also be in the format %Y-%jT%H:%M:%S.%fZ
- `curl localhost:5000/[epoch]/position`
  - `[epoch]` must be a string, the epoch queried. This route returns the location information for the epoch requested: latitude, longitude, altitude, and approximate geographic location (with "none" meaning the ISS is above the ocean). This must also be in the format %Y-%jT%H:%M:%S.%fZ
- `curl localhost:5000/now`
  - This route returns state vectors, instantaneous speed, and position data for the epoch closest to the current time.
- `curl localhost:5000/range`
  - This route returns information on the number of epochs in the dataset and the times the epochs range from.

To execute `test_iss_tracker.py`, the user may run the command `pytest`. Pytest is automatically installed in the container build, so all the unit tests for `iss_tracker.py` should readily run. The output of `test_iss_tracker.py` should list ten successes, one for each unit test. Additionally, there may be a

warning returned from `test_return_state_vectors`: the app is designed to use the very first epoch (January 1, 1970) if the Redis database cannot be accessed, and Astropy throws a warning if the epoch is from too long ago. However, this is not an issue while using `pytest`, and demonstrates that the functions do return dictionaries of data, as desired.

5 Ethical and Professional Responsibilities

Careful considerations must be taken when working with data that is not one's own. The software engineer must be sure to cite others' intellectual property and not misrepresent the data. Additionally, the software engineer has a responsibility not to use the data for malicious purposes. This line blurs when engineering large-scale products that directly influence people's lives, in which case it is important to consider if anyone could be harmed by the product or data used.
The software engineer is responsible for respecting a user's privacy. In this program, the Redis container can write directly to the user's machine. An ethical program must not extract more data from the user's machine than is strictly necessary, and must be careful not to write anything that could crash or otherwise damage the user's system.
The usage of artificial intelligence is rising and hotly debated - at minimum, the software engineer is responsible to cite AI usage and carefully check for errors.

6 Citations

National Aeronautics and Space Administration (NASA), "Spot the Station - ISS Trajectory Data". https://spotthestation.nasa.gov/trajectory_data.cfm. Accessed March 14, 2025