Travel Time Calculation Across Different Data Sources

CADES Consulting Project

Meghana Cyanam

Elijah Whitham-Powell

Overview

- The Tools
- What was done
- Issues Encountered
- A better way to relate different data sources
- Some code
- Future uses

The Tools

- Python and the following libraries:
  - pandas - Data manipulation and analysis.
  - geopandas - Extends pandas to allow spatial operations on geometric types.
  - numpy - Mathematical functions on arrays and matrices.
  - matplotlib - Plotting library.
  - shapely - Geometric operations (installing geopandas should include this).
- Grit and Caffeine.

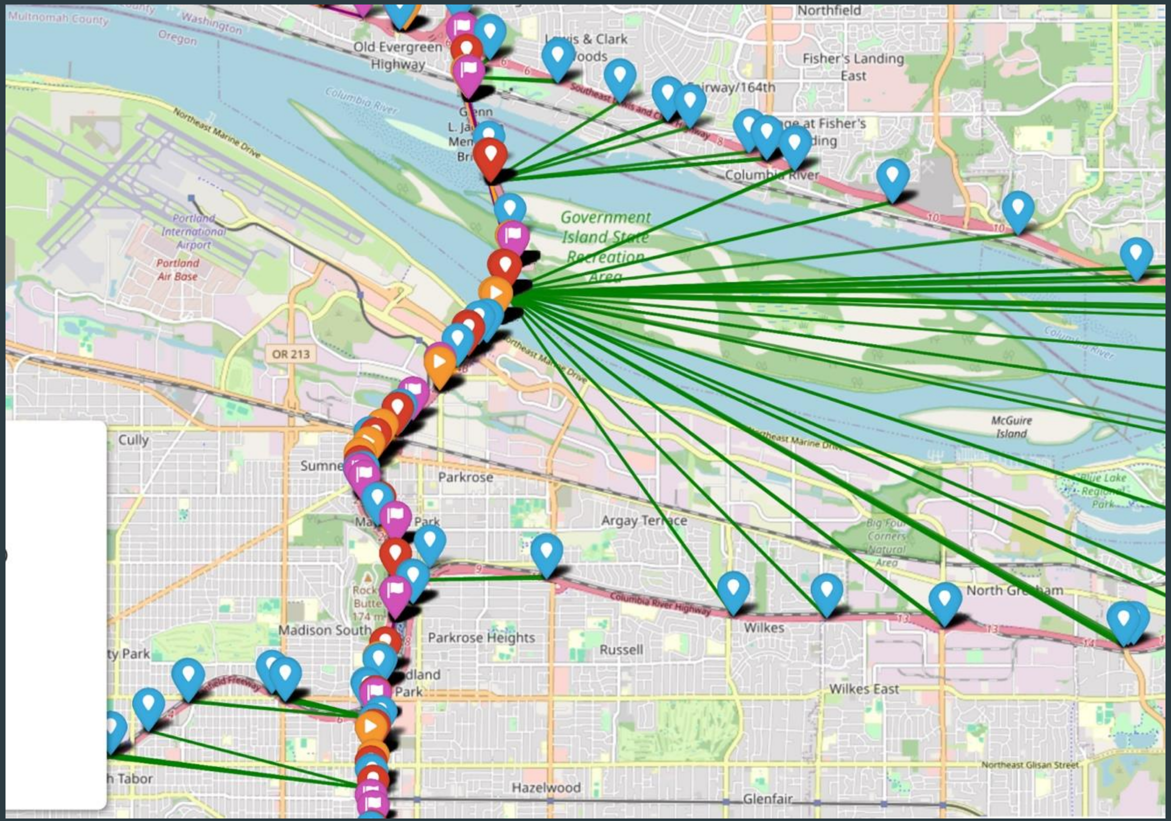
What was done

- Most of the time was spent collecting and sifting through the two data sources.
- Lots of code written.
- Little real analysis accomplished.
- A usable subset was created covering I-205.

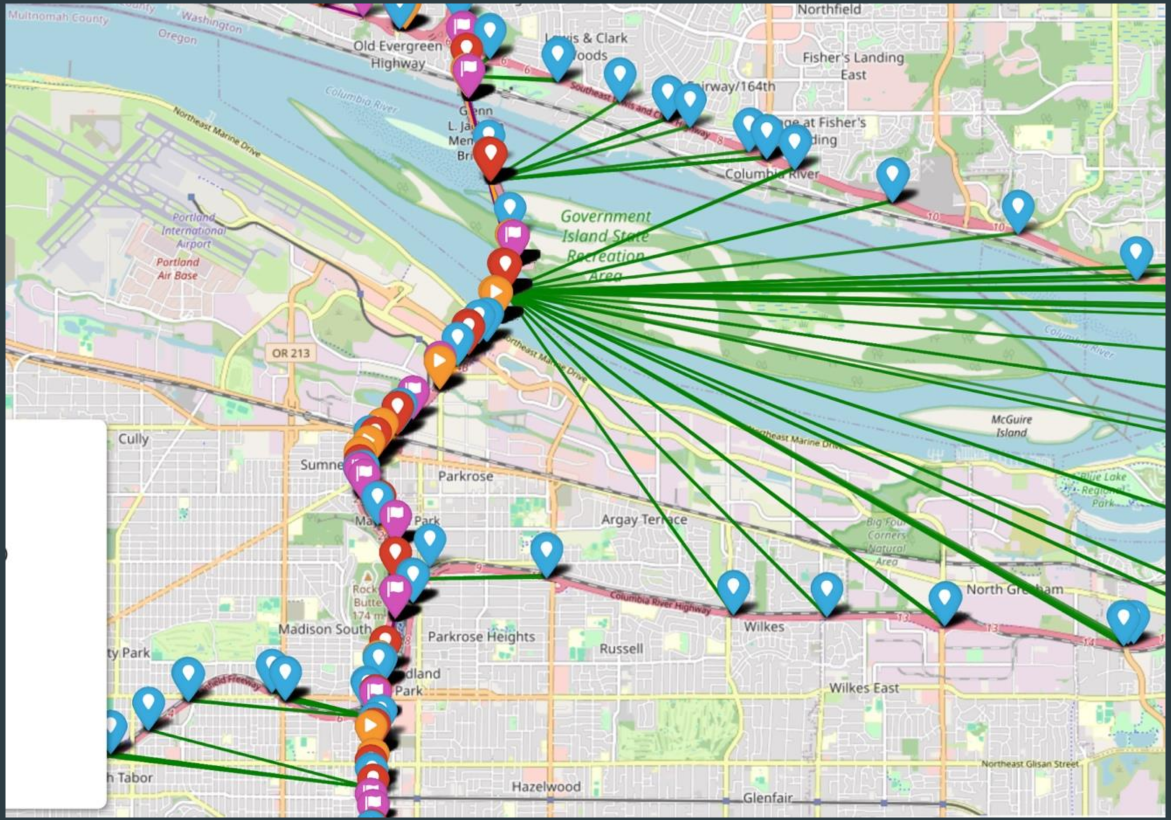
Issues Encountered

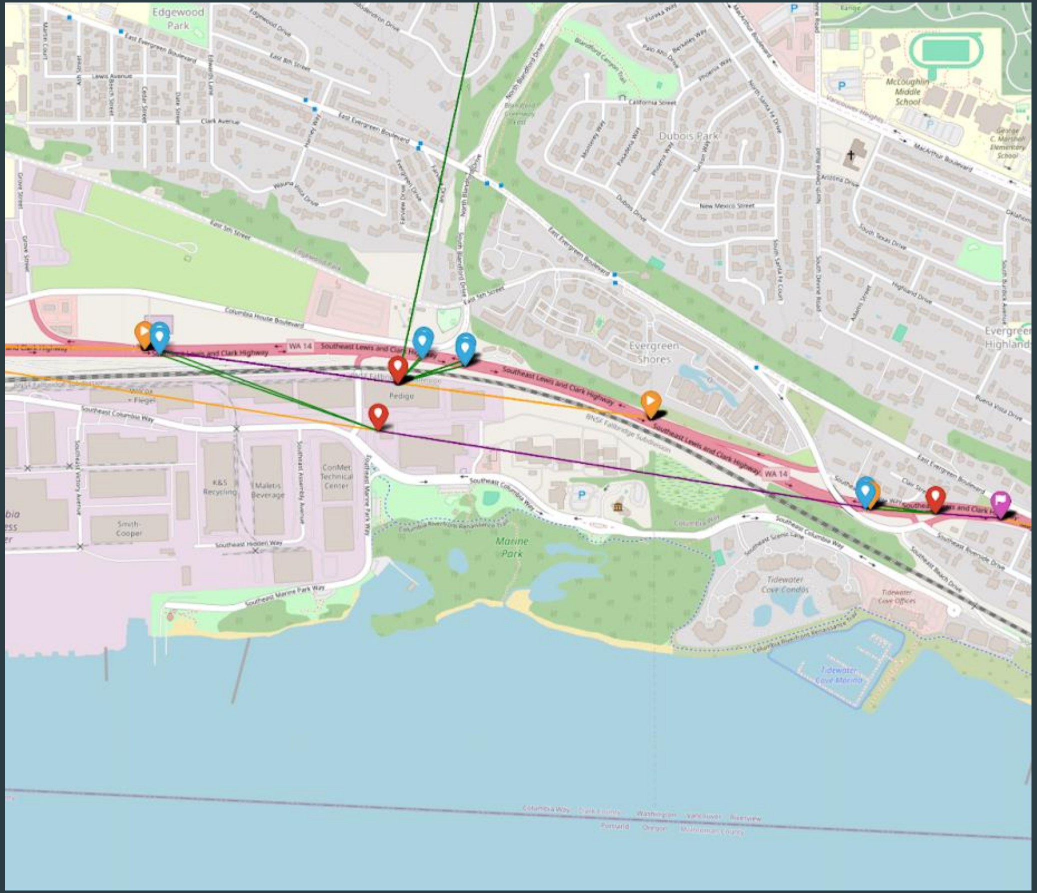
Naïve geometric closeness approach to matching INRIX readings to PORTAL readings spatially.

The first issue we ran into was determining how to properly connect INRIX (purple and orange) sensor locations to PORTAL (blue) station locations and is demonstrated here.

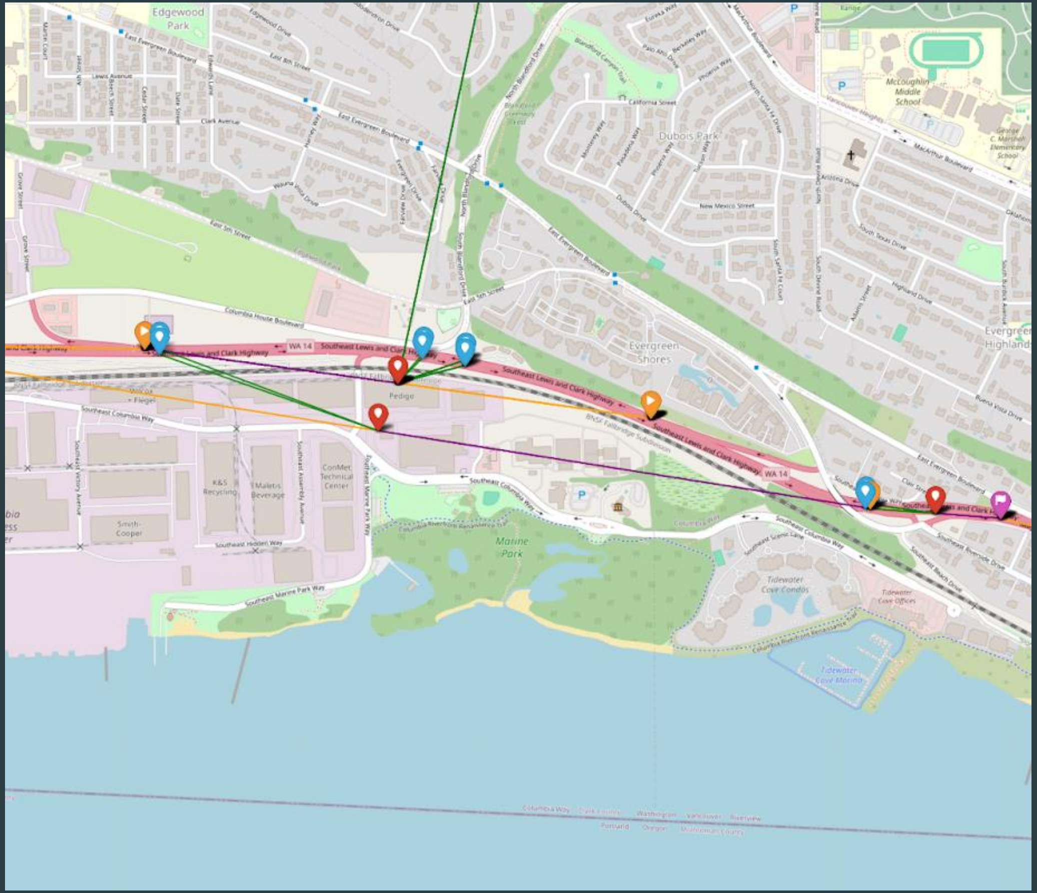


Multiple PORTAL (BLUE) station locations were matching as "closest" to the calculated INRIX midpoints (red) incorrectly without considering a distance threshold.

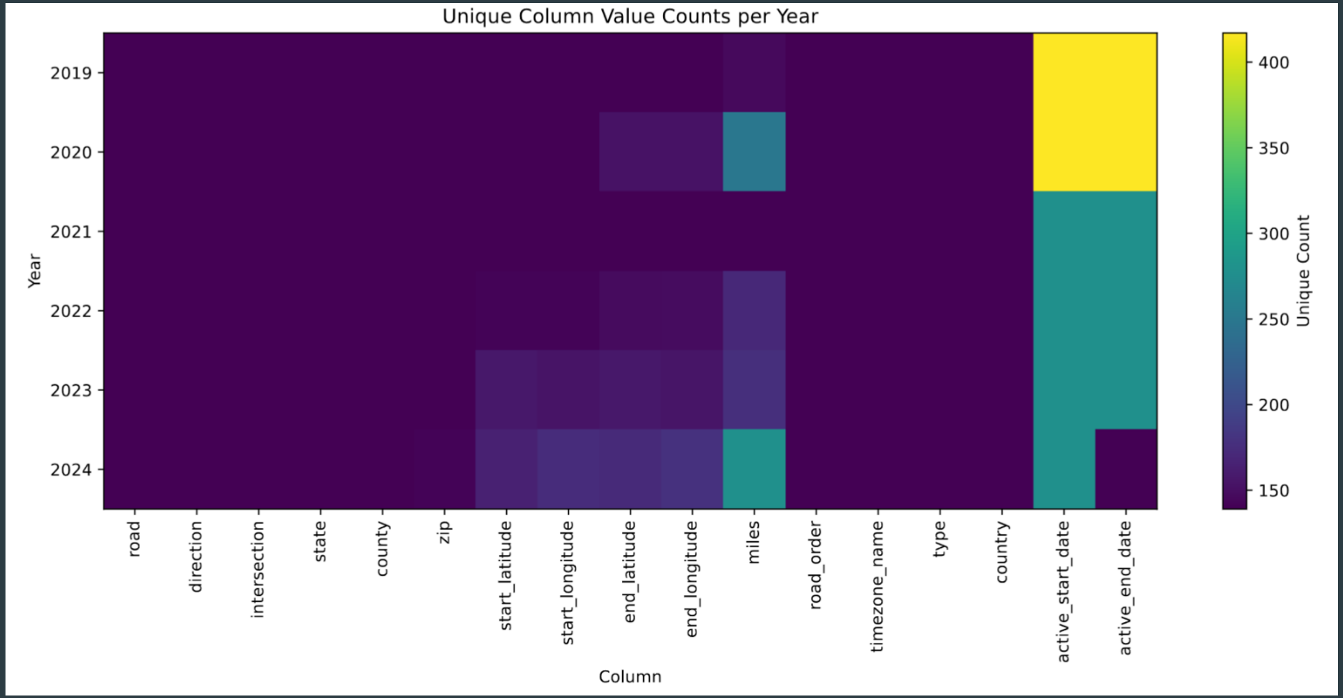




Highlighted here is the second issue that results using calculated INRIX TMC midpoints.



The PORTAL (blue) stations are closer in this example, but the midpoint (red) is off the roadway.



It was seen that the INRIX metadata not in Shapefile format had duplicate rows for each TMC that indicates they might have moved even within the same year.

A better way to relate different data sources

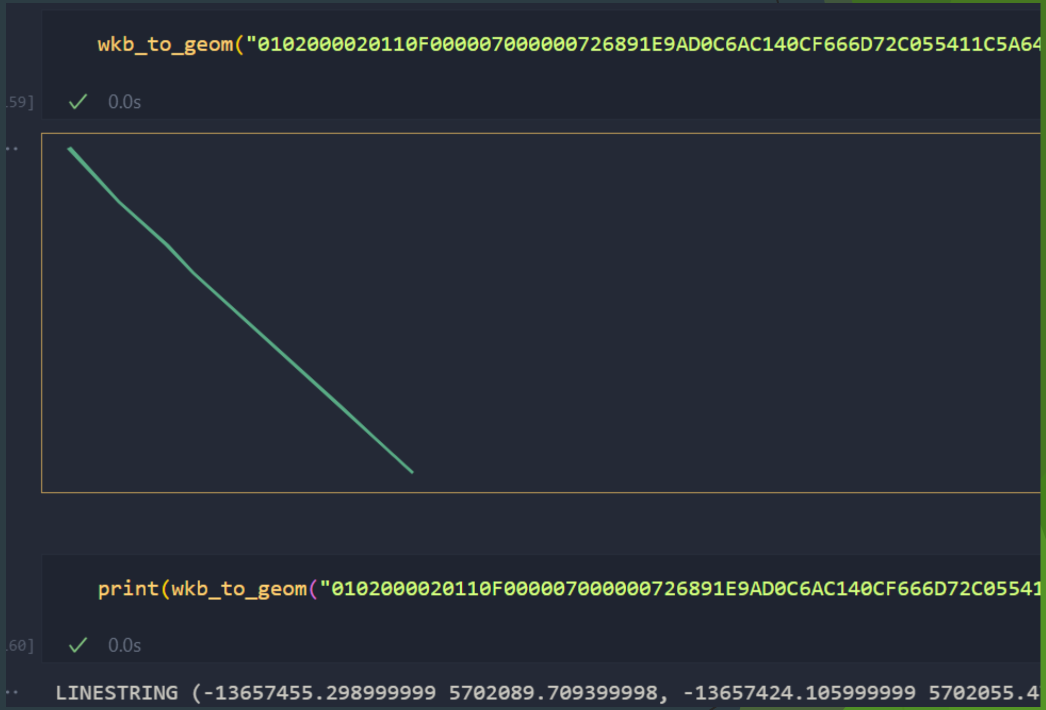
Geometry and Geography

A better way to relate different data sources.

- Noticing a column of the PORTAL stations table looked like machine code
  - *0102000020110F000007000000726891E9AD0C6AC140CF666D72C055411C5A6403AA0C6AC1482575DE69C05541B4C87638A60C6AC16888631963C0554148BF7D15A40C6AC1A8E848CA5EC055414872F9*
- Converting this HEX to a Well-Known Binary (WKB) a standard format in Geo-Spatial data analysis.

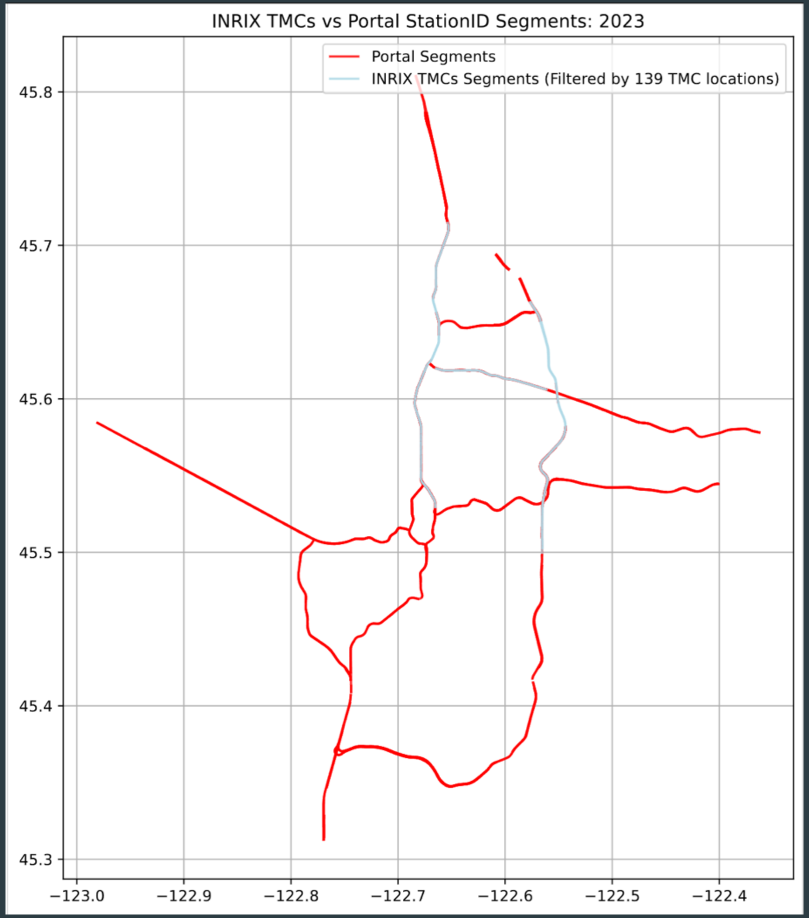
A better way to relate different data sources.

- Noticing a column of the PORTAL stations table looked like machine code
  - *0102000020110F000007000000726891E9AD0C6AC140CF666D72C055411C5A6403AA0C6AC1482575DE69C05541B4C87638A60C6AC16888631963C0554148BF7D15A40C6AC1A8E848CA5EC055414872F9*
- Converting this HEX to a Well-Known Binary (WKB) a standard format in Geo-Spatial data analysis.
- This is the shape part of a shape file.

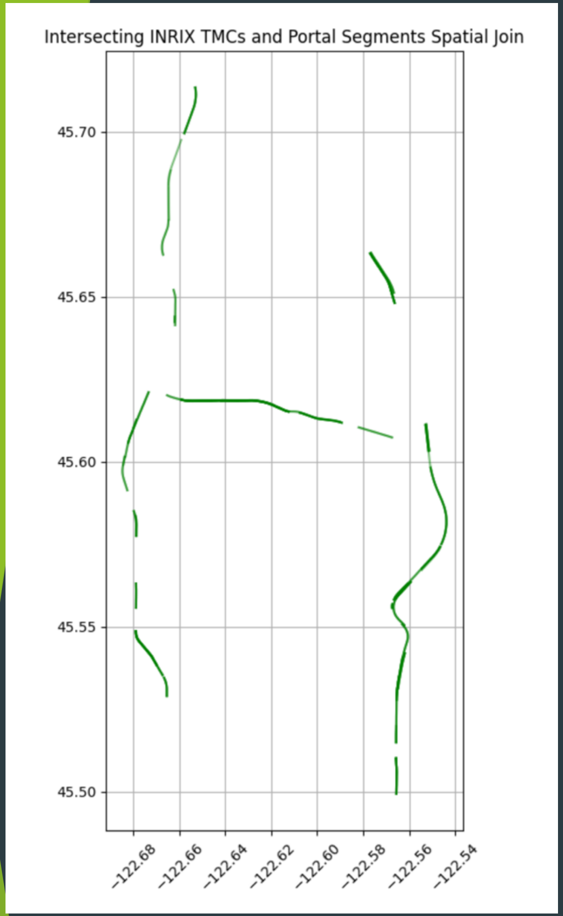


```
wkb_to_geom("0102000020110F000007000000726891E9AD0C6AC140CF666D72C055411C5A64
59]  ✓ 0.0s
```

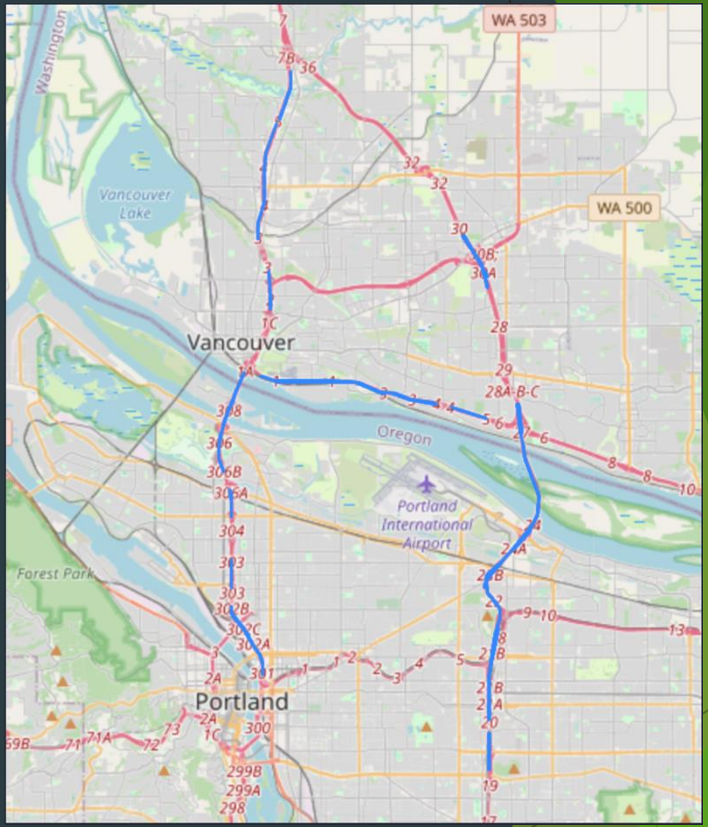
```
print(wkb_to_geom("0102000020110F000007000000726891E9AD0C6AC140CF666D72C05541
60]  ✓ 0.0s
LINESTRING (-13657455.298999999 5702089.709399998, -13657424.105999999 5702055.4
```


INRIX TMCs vs Portal StationID Segments: 2023
Portal Segments
INRIX TMCs Segments (Filtered by 139 TMC locations)
45.8
45.7
45.6
45.5
45.4
45.3
−123.0
−122.9
−122.8
−122.7
−122.6
−122.5
−122.4

The segments available using better specified line segments on the roadway.



The overlap leaves us with fewer but more spatially accurate segments.



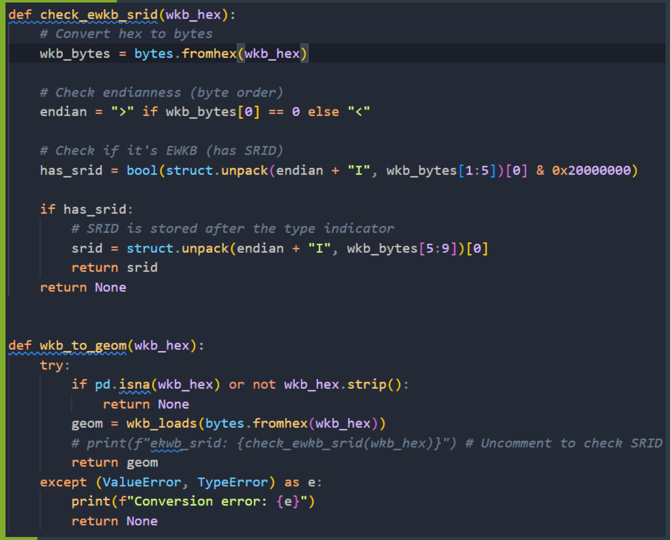
The spatial relationship between each data set was just the beginning.

- Each data set tracks timestamps differently with respect to time-zone awareness.
- INRIX offers travel time in seconds.
- PORTAL offers travel time in minutes.
- Careful data type conversion required.
- Joining of the time series datasets into the metadata location mapping.
- Verifying the spatial join was consistent with the direction or bound of the piece of roadway being matched.

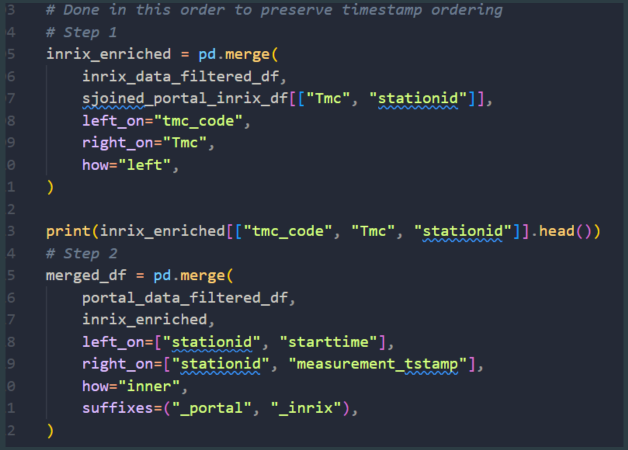
Some code

Key implementations making this possible

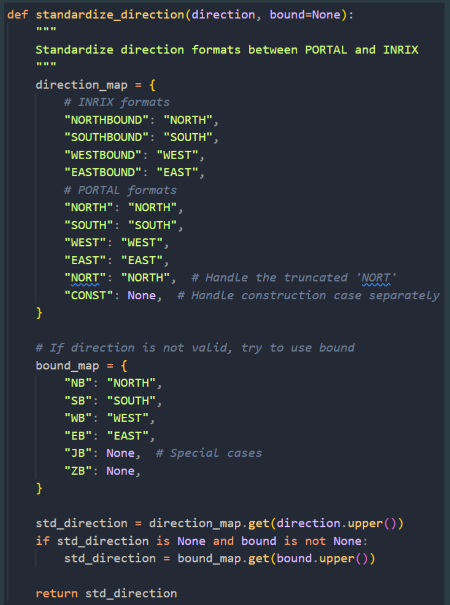
These code snippets highlight the technical challenge of wrangling the two data sources



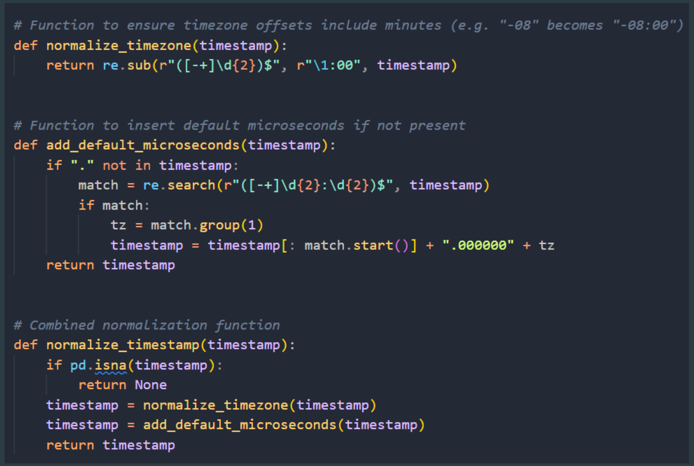
```python
def check_ewkb_srid(wkb_hex):
    # Convert hex to bytes
    wkb_bytes = bytes.fromhex(wkb_hex)

    # Check endianness (byte order)
    endian = ">" if wkb_bytes[0] == 0 else "<"

    # Check if it's EWKB (has SRID)
    has_srid = bool(struct.unpack(endian + "I", wkb_bytes[1:5])[0] & 0x20000000)

    if has_srid:
        # SRID is stored after the type indicator
        srid = struct.unpack(endian + "I", wkb_bytes[5:9])[0]
        return srid
    return None


def wkb_to_geom(wkb_hex):
    try:
        if pd.isna(wkb_hex) or not wkb_hex.strip():
            return None
        geom = wkb_loads(bytes.fromhex(wkb_hex))
        # print(f"ekwb_srid: {check_ewkb_srid(wkb_hex)}") # Uncomment to check SRID
        return geom
    except (ValueError, TypeError) as e:
        print(f"Conversion error: {e}")
        return None
```



```python
# Done in this order to preserve timestamp ordering
# Step 1
inrix_enriched = pd.merge(
    inrix_data_filtered_df,
    sjoined_portal_inrix_df[["Tmc", "stationid"]],
    left_on="tmc_code",
    right_on="Tmc",
    how="left",
)

print(inrix_enriched[["tmc_code", "Tmc", "stationid"]].head())
# Step 2
merged_df = pd.merge(
    portal_data_filtered_df,
    inrix_enriched,
    left_on=["stationid", "starttime"],
    right_on=["stationid", "measurement_tstamp"],
    how="inner",
    suffixes=("_portal", "_inrix"),
)
```

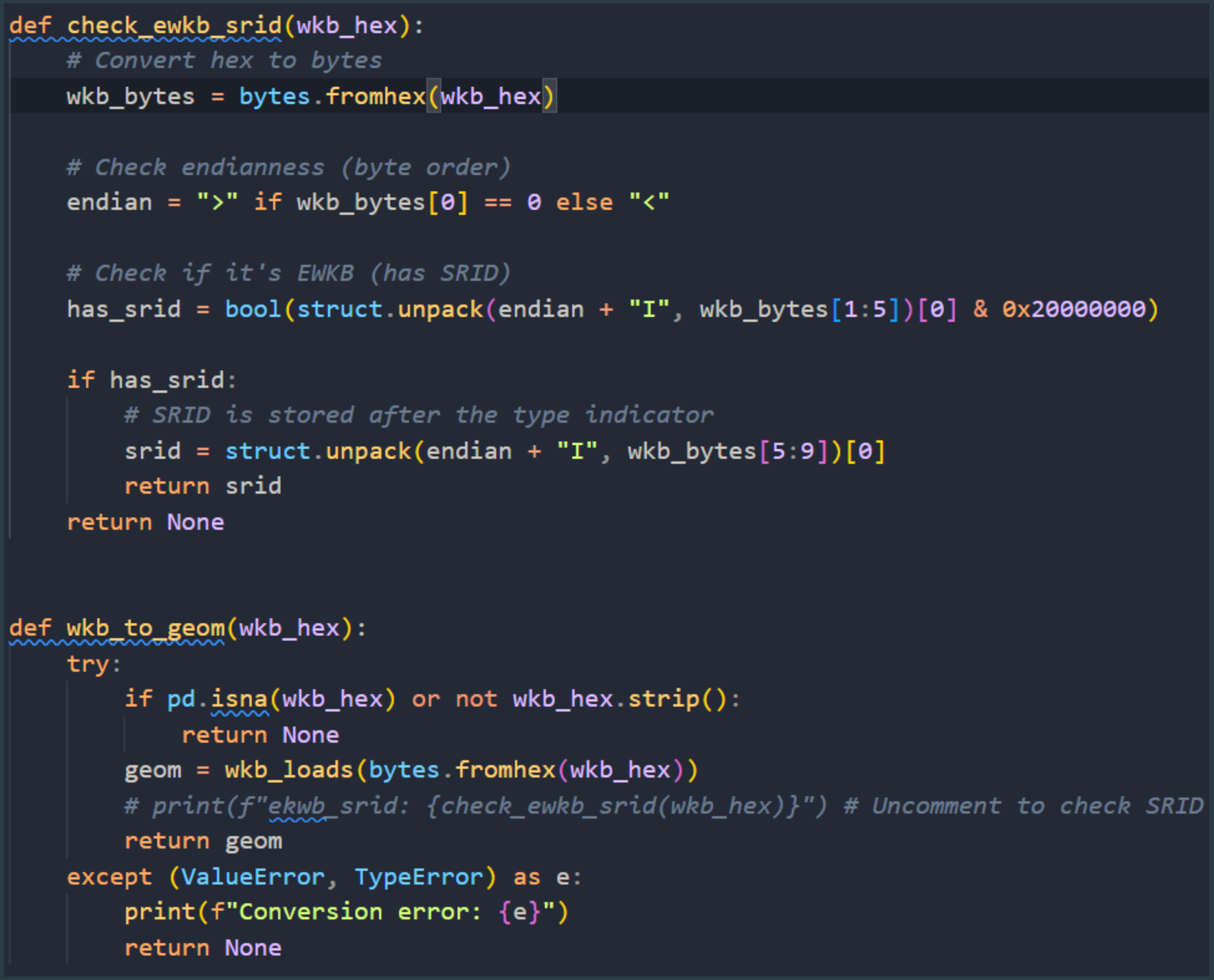


```python
def standardize_direction(direction, bound=None):
    """
    Standardize direction formats between PORTAL and INRIX
    """
    direction_map = {
        # INRIX formats
        "NORTHBOUND": "NORTH",
        "SOUTHBOUND": "SOUTH",
        "WESTBOUND": "WEST",
        "EASTBOUND": "EAST",
        # PORTAL formats
        "NORTH": "NORTH",
        "SOUTH": "SOUTH",
        "WEST": "WEST",
        "EAST": "EAST",
        "NORT": "NORTH",  # Handle the truncated 'NORT'
        "CONST": None,  # Handle construction case separately
    }

    # If direction is not valid, try to use bound
    bound_map = {
        "NB": "NORTH",
        "SB": "SOUTH",
        "WB": "WEST",
        "EB": "EAST",
        "JB": None,  # Special cases
        "ZB": None,
    }

    std_direction = direction_map.get(direction.upper())
    if std_direction is None and bound is not None:
        std_direction = bound_map.get(bound.upper())

    return std_direction
```

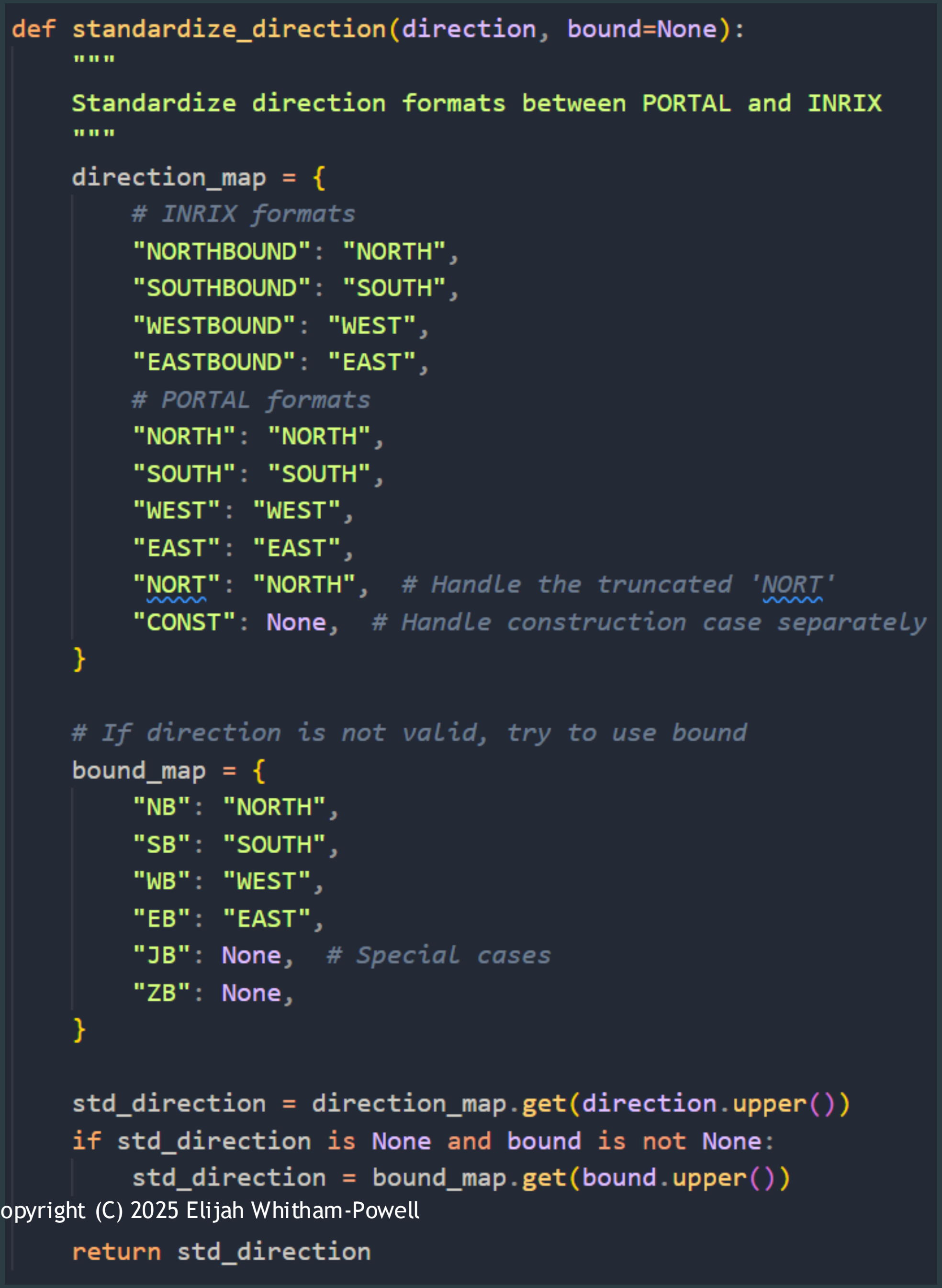


```python
# Function to ensure timezone offsets include minutes (e.g. "-08" becomes "-08:00")
def normalize_timezone(timestamp):
    return re.sub(r"([-+]\d{2})$", r"\1:00", timestamp)


# Function to insert default microseconds if not present
def add_default_microseconds(timestamp):
    if "." not in timestamp:
        match = re.search(r"([-+]\d{2}:\d{2})$", timestamp)
        if match:
            tz = match.group(1)
            timestamp = timestamp[: match.start()] + ".000000" + tz
    return timestamp


# Combined normalization function
def normalize_timestamp(timestamp):
    if pd.isna(timestamp):
        return None
    timestamp = normalize_timezone(timestamp)
    timestamp = add_default_microseconds(timestamp)
    return timestamp
```

WKB Conversion



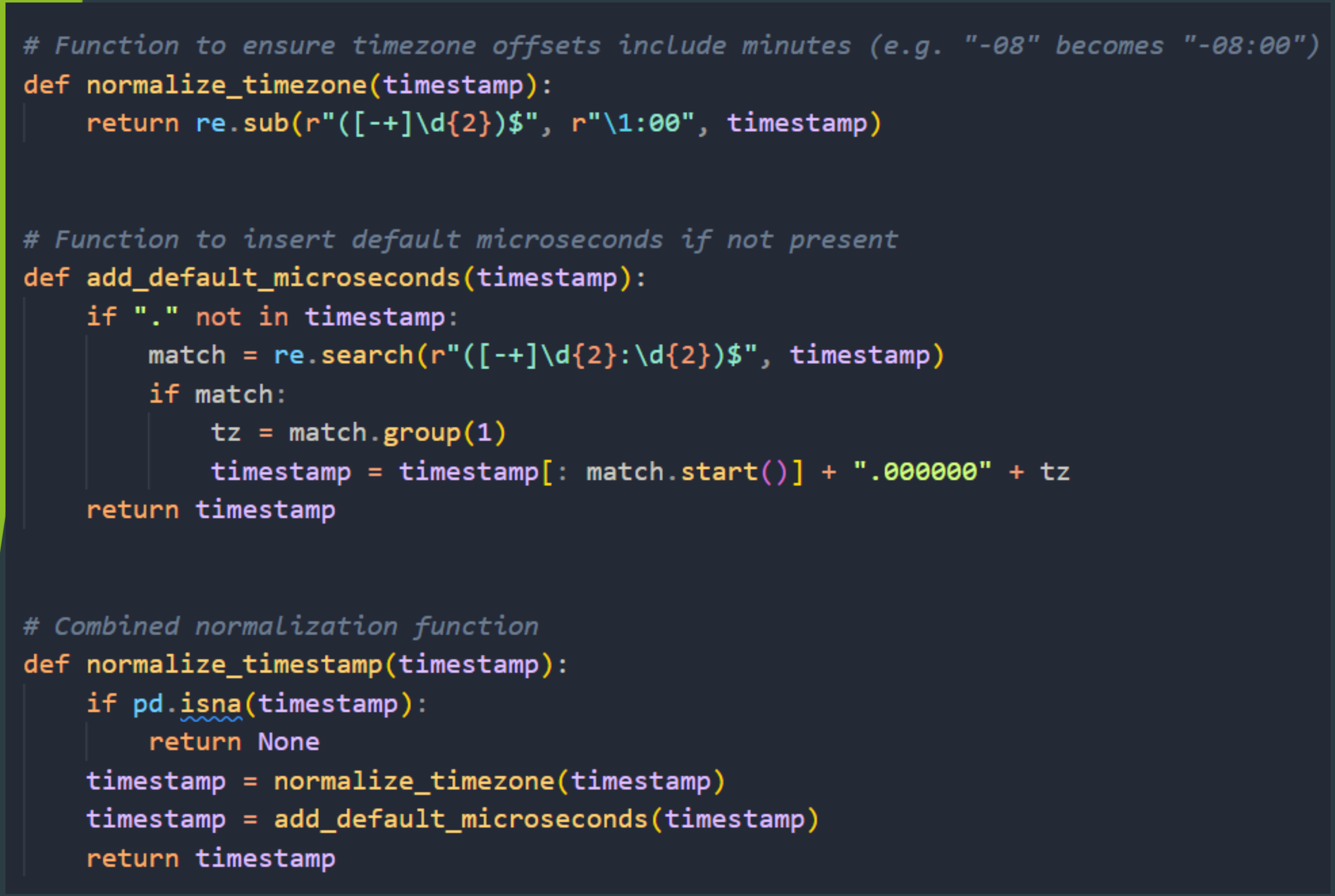
```python
def check_ewkb_srid(wkb_hex):
    # Convert hex to bytes
    wkb_bytes = bytes.fromhex(wkb_hex)

    # Check endianness (byte order)
    endian = ">" if wkb_bytes[0] == 0 else "<"

    # Check if it's EWKB (has SRID)
    has_srid = bool(struct.unpack(endian + "I", wkb_bytes[1:5])[0] & 0x20000000)

    if has_srid:
        # SRID is stored after the type indicator
        srid = struct.unpack(endian + "I", wkb_bytes[5:9])[0]
        return srid
    return None


def wkb_to_geom(wkb_hex):
    try:
        if pd.isna(wkb_hex) or not wkb_hex.strip():
            return None
        geom = wkb_loads(bytes.fromhex(wkb_hex))
        # print(f"ekwb_srid: {check_ewkb_srid(wkb_hex)}") # Uncomment to check SRID
        return geom
    except (ValueError, TypeError) as e:
        print(f"Conversion error: {e}")
        return None
```



```python
def standardize_direction(direction, bound=None):
    """
    Standardize direction formats between PORTAL and INRIX
    """
    direction_map = {
        # INRIX formats
        "NORTHBOUND": "NORTH",
        "SOUTHBOUND": "SOUTH",
        "WESTBOUND": "WEST",
        "EASTBOUND": "EAST",
        # PORTAL formats
        "NORTH": "NORTH",
        "SOUTH": "SOUTH",
        "WEST": "WEST",
        "EAST": "EAST",
        "NORT": "NORTH",  # Handle the truncated 'NORT'
        "CONST": None,  # Handle construction case separately
    }

    # If direction is not valid, try to use bound
    bound_map = {
        "NB": "NORTH",
        "SB": "SOUTH",
        "WB": "WEST",
        "EB": "EAST",
        "JB": None,  # Special cases
        "ZB": None,
    }

    std_direction = direction_map.get(direction.upper())
    if std_direction is None and bound is not None:
        std_direction = bound_map.get(bound.upper())

    return std_direction
```

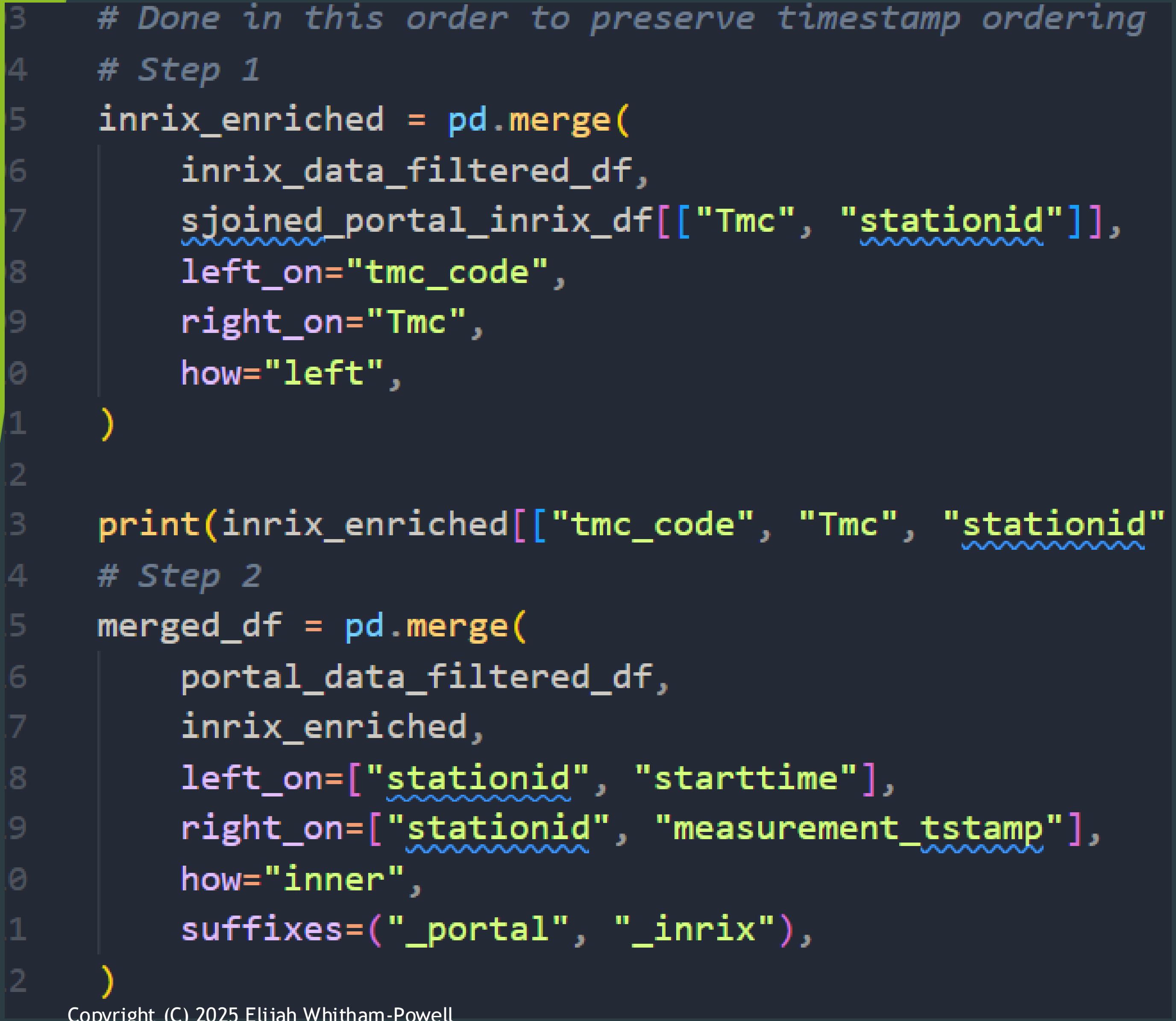
Direction Standardization

Timestamp normalization



```
# Function to ensure timezone offsets include minutes (e.g. "-08" becomes "-08:00")
def normalize_timezone(timestamp):
    return re.sub(r"([-+]\d{2})$", r"\1:00", timestamp)


# Function to insert default microseconds if not present
def add_default_microseconds(timestamp):
    if "." not in timestamp:
        match = re.search(r"([-+]\d{2}:\d{2})$", timestamp)
        if match:
            tz = match.group(1)
            timestamp = timestamp[: match.start()] + ".000000" + tz
    return timestamp


# Combined normalization function
def normalize_timestamp(timestamp):
    if pd.isna(timestamp):
        return None
    timestamp = normalize_timezone(timestamp)
    timestamp = add_default_microseconds(timestamp)
    return timestamp
```

Joining the metadata to the timeseries



```python
# Done in this order to preserve timestamp ordering
# Step 1
inrix_enriched = pd.merge(
    inrix_data_filtered_df,
    sjoined_portal_inrix_df[["Tmc", "stationid"]],
    left_on="tmc_code",
    right_on="Tmc",
    how="left",
)

print(inrix_enriched[["tmc_code", "Tmc", "stationid"
# Step 2
merged_df = pd.merge(
    portal_data_filtered_df,
    inrix_enriched,
    left_on=["stationid", "starttime"],
    right_on=["stationid", "measurement_tstamp"],
    how="inner",
    suffixes=("_portal", "_inrix"),
)
```

Now what?

Future uses of the combined data.

Future uses of this combined data

Change Point Analysis

Identifies significant shifts in time-series data (such as sudden changes in travel time).

Can be used to differentiate between routine fluctuations and real disruptions.

Could be used to assess how well either dataset detects or reacts to an indicent on the highway.

Maximum Mean Discrepancy (MMD)

Involves mathematically mapping the data in a higher dimension and calculating the distance between each data sets “average” or “central” point in that space.

Can reveal differences not detectable in the data’s native dimension.

MMD can be used in a two-sample test to see how far apart two distributions are away from each other in a higher dimensional space.

Questions?

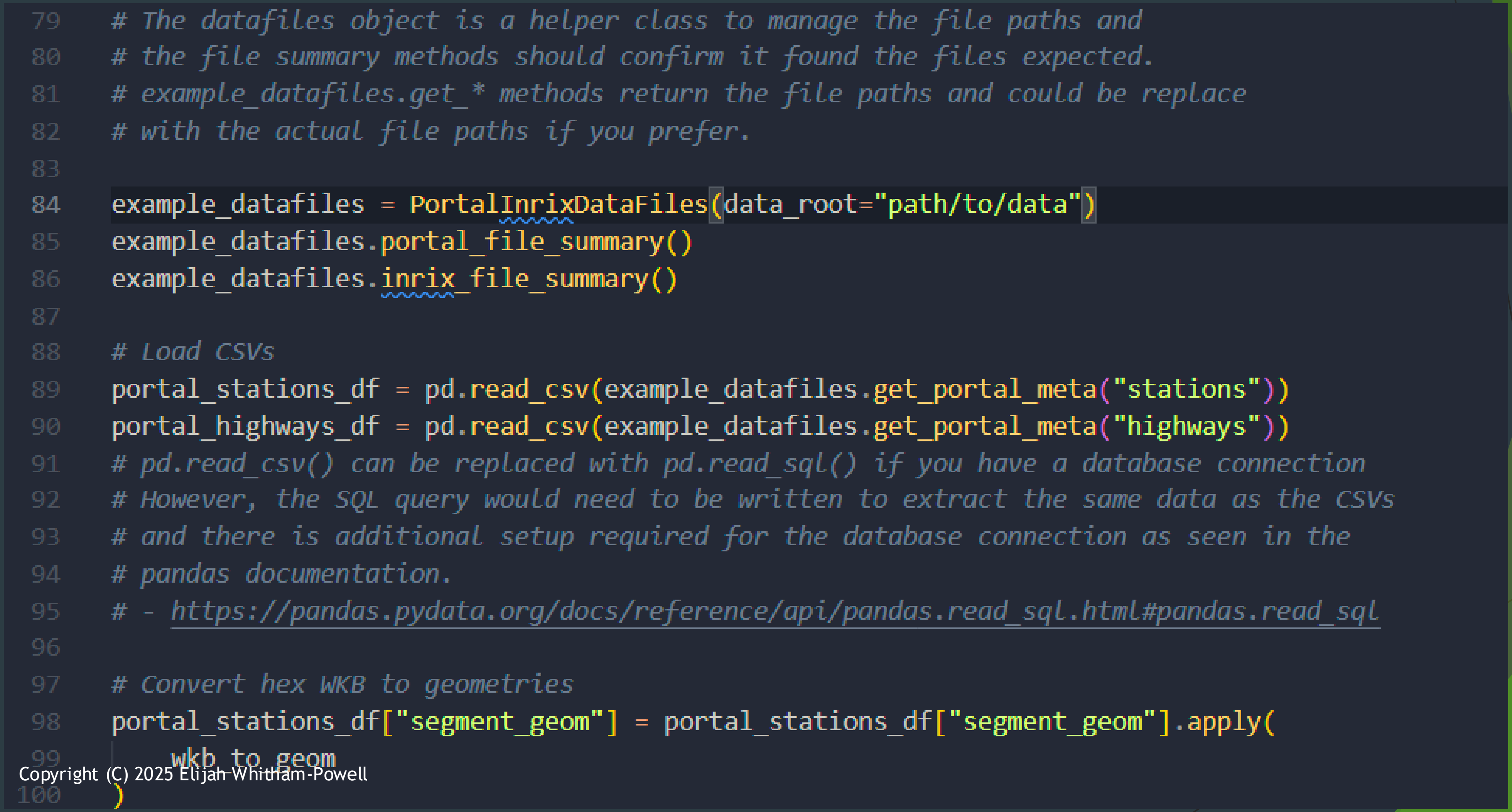
A closer look at the code involved?

GitHub Repo

github.com/whitham-powell/cades-traveltime-compare

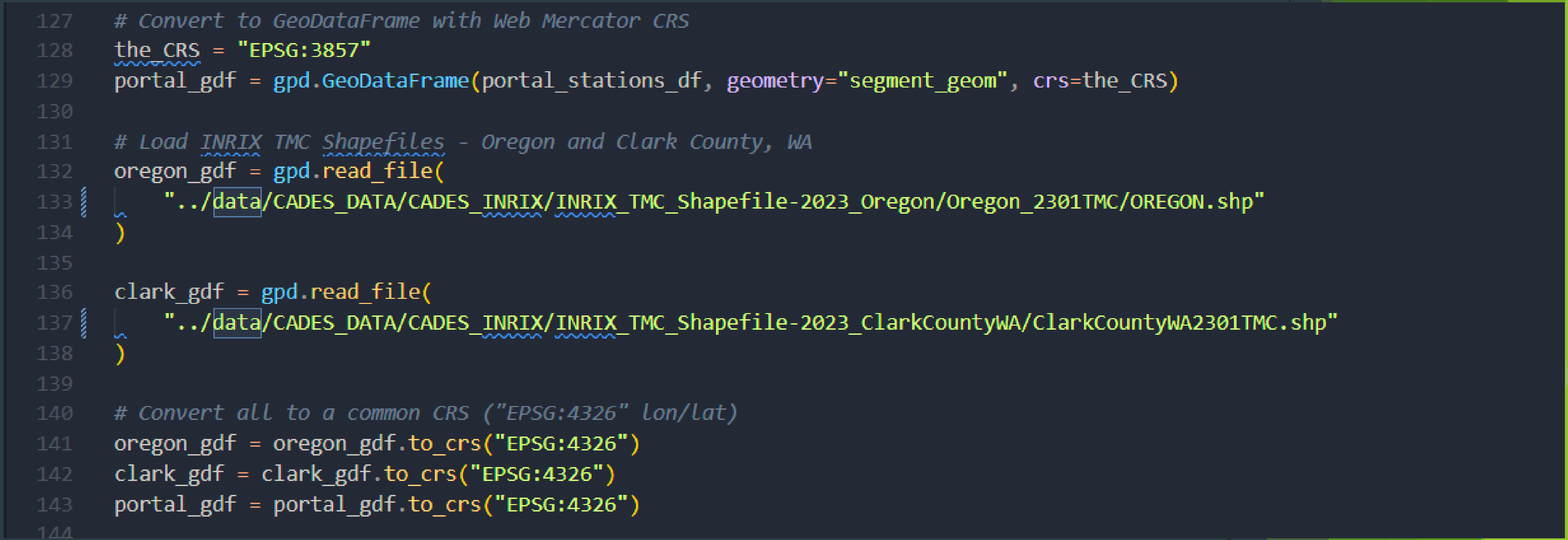
Appendix

Loading the data



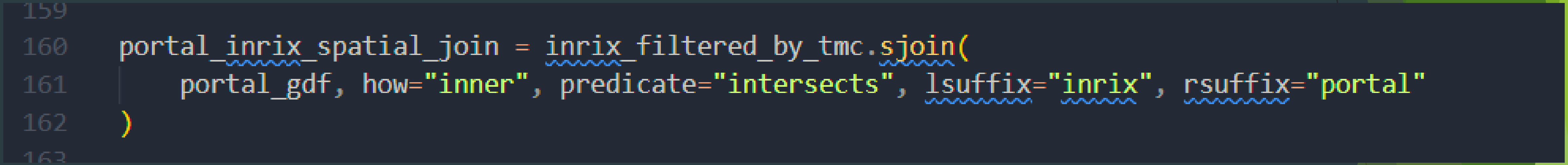
```python
# The datafiles object is a helper class to manage the file paths and
# the file summary methods should confirm it found the files expected.
# example_datafiles.get_* methods return the file paths and could be replace
# with the actual file paths if you prefer.

example_datafiles = PortalInrixDataFiles(data_root="path/to/data")
example_datafiles.portal_file_summary()
example_datafiles.inrix_file_summary()

# Load CSVs
portal_stations_df = pd.read_csv(example_datafiles.get_portal_meta("stations"))
portal_highways_df = pd.read_csv(example_datafiles.get_portal_meta("highways"))
# pd.read_csv() can be replaced with pd.read_sql() if you have a database connection
# However, the SQL query would need to be written to extract the same data as the CSVs
# and there is additional setup required for the database connection as seen in the
# pandas documentation.
# - https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html#pandas.read_sql

# Convert hex WKB to geometries
portal_stations_df["segment_geom"] = portal_stations_df["segment_geom"].apply(
    wkb_to_geom
)
```

Creating GeoPandas Dataframes



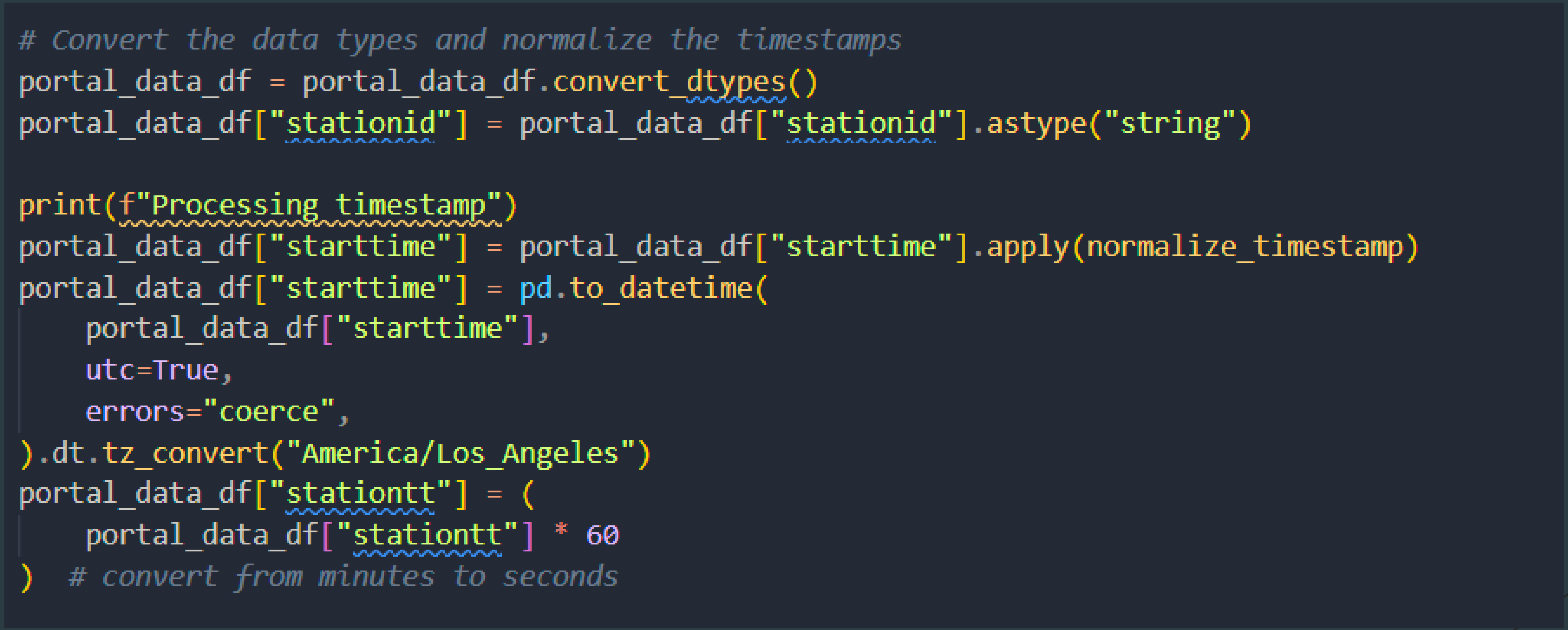
```
# Convert to GeoDataFrame with Web Mercator CRS
the_CRS = "EPSG:3857"
portal_gdf = gpd.GeoDataFrame(portal_stations_df, geometry="segment_geom", crs=the_CRS)

# Load INRIX TMC Shapefiles - Oregon and Clark County, WA
oregon_gdf = gpd.read_file(
    "../data/CADES_DATA/CADES_INRIX/INRIX_TMC_Shapefile-2023_Oregon/Oregon_2301TMC/OREGON.shp"
)

clark_gdf = gpd.read_file(
    "../data/CADES_DATA/CADES_INRIX/INRIX_TMC_Shapefile-2023_ClarkCountyWA/ClarkCountyWA2301TMC.shp"
)

# Convert all to a common CRS ("EPSG:4326" lon/lat)
oregon_gdf = oregon_gdf.to_crs("EPSG:4326")
clark_gdf = clark_gdf.to_crs("EPSG:4326")
portal_gdf = portal_gdf.to_crs("EPSG:4326")
```

The GeoPandas spatial join



```
portal_inrix_spatial_join = inrix_filtered_by_tmc.sjoin(
    portal_gdf, how="inner", predicate="intersects", lsuffix="inrix", rsuffix="portal"
)
```

Datatype conversion and normalization



```
# Convert the data types and normalize the timestamps
portal_data_df = portal_data_df.convert_dtypes()
portal_data_df["stationid"] = portal_data_df["stationid"].astype("string")

print(f"Processing timestamp")
portal_data_df["starttime"] = portal_data_df["starttime"].apply(normalize_timestamp)
portal_data_df["starttime"] = pd.to_datetime(
    portal_data_df["starttime"],
    utc=True,
    errors="coerce",
).dt.tz_convert("America/Los_Angeles")
portal_data_df["stationtt"] = (
    portal_data_df["stationtt"] * 60
)  # convert from minutes to seconds
```