Enhancing Seismic Hazard Assessment: Expanding the Fragile Geologic Feature Database Across California and Nevada

Sagar Kapri[1]; Xiaofeng Meng[2]

[1]University of California Berkeley, [2]Statewide California Earthquake Center; University of Southern California

Motivations

- For instance, California and Nevada are earthquake-prone areas that risk their critical infrastructure such as roads, dams, buildings, and nuclear plants.
- To be precise and make the structures resilient, it is very important to access seismic hazards accurately.
- The Fragile Geological Features(FGFs) serve as natural indicators of maximum ground motions in previous earthquakes.

Background

- The current FGF database developed by SCEC is limited to Southern California and does not include the area such as Central and Northern California and Nevada, leaving a significant gap in seismic hazard assesment for these regions.
- To address the gap, we collaborated with the University of Nevada, Reno and added Metadata and over 3,000 FGF photograph from underrepresented regions to the database.

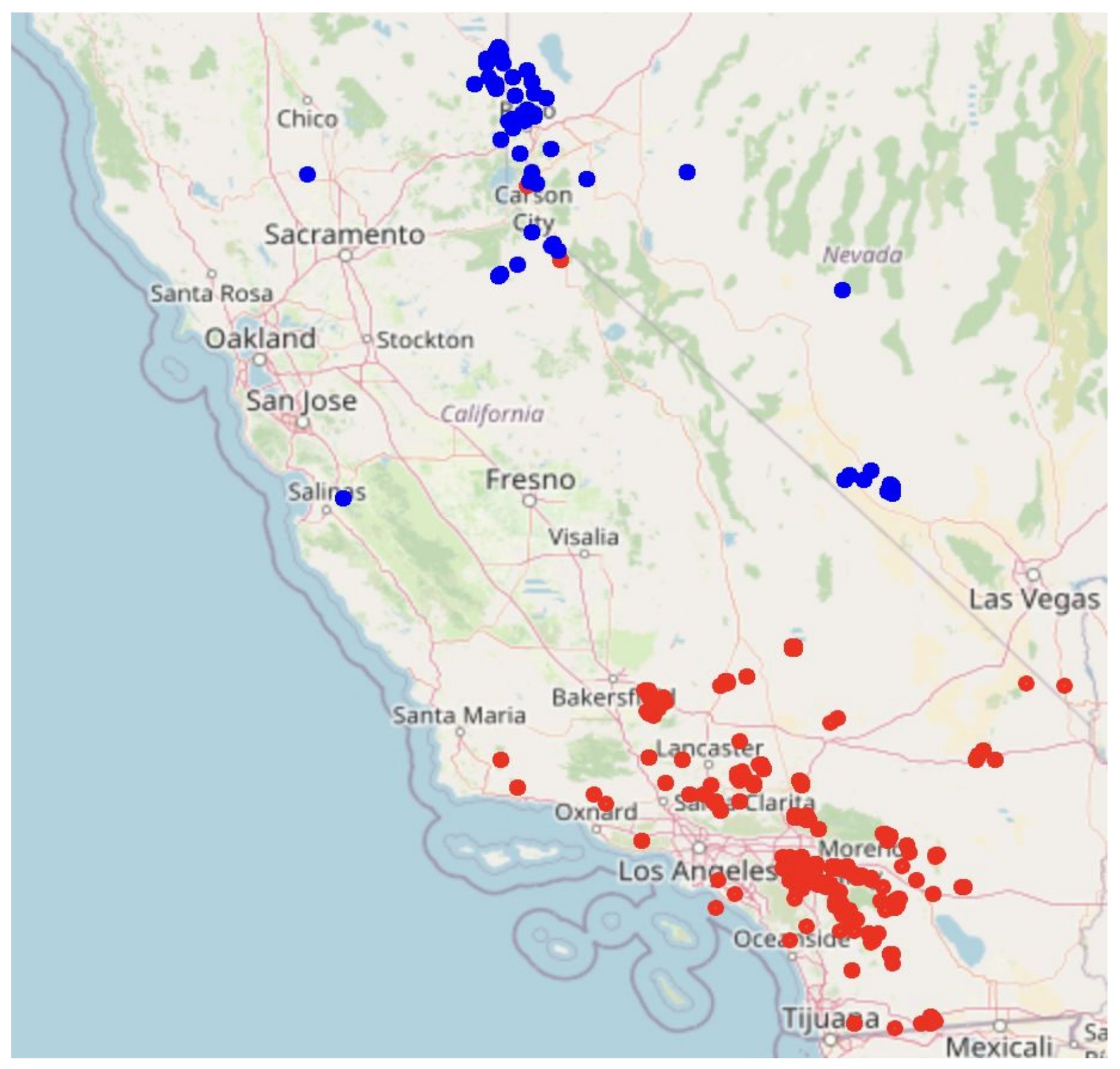


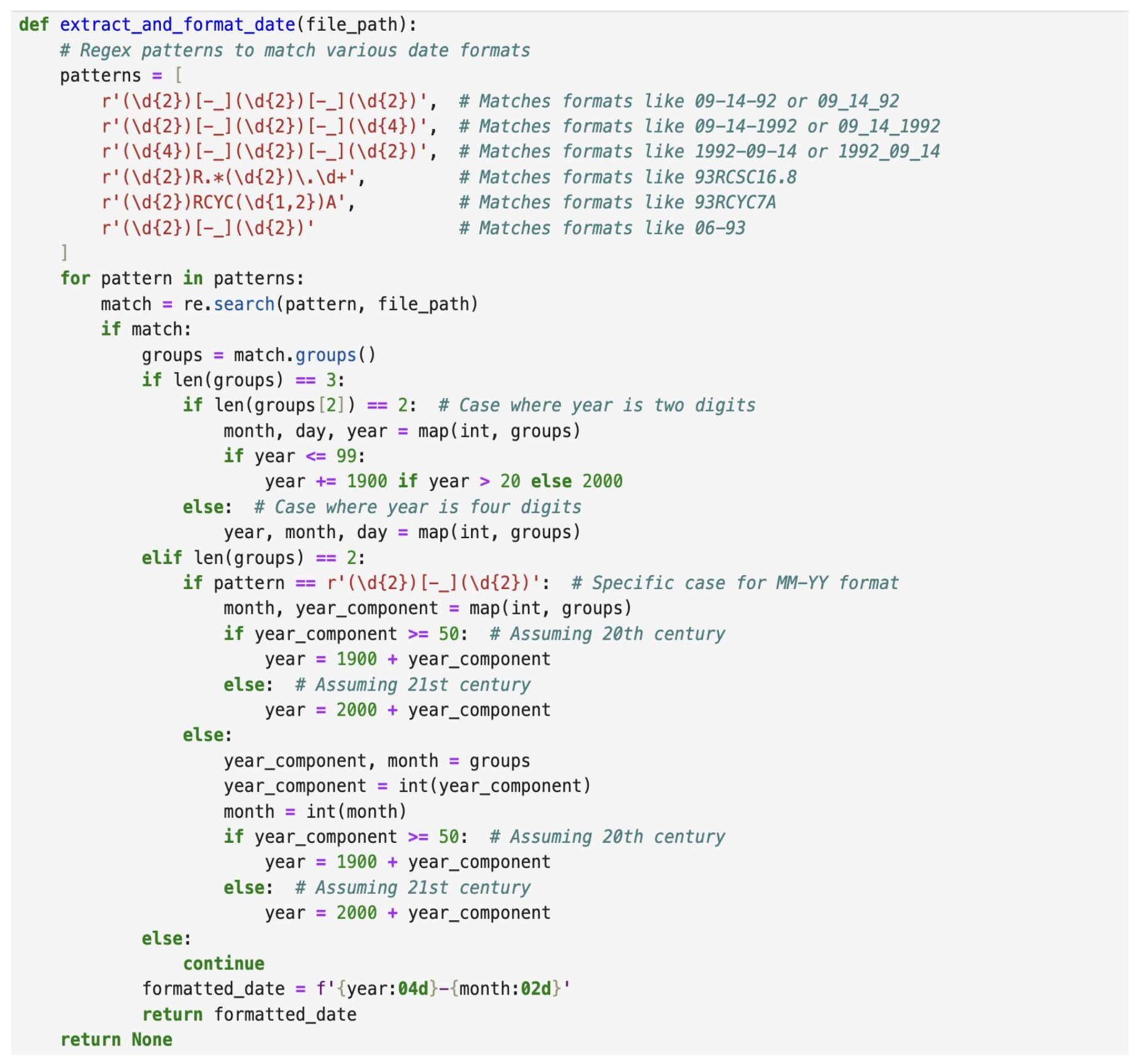
Figure 1: Scatter plot showing the distribution of FGF in California. Red points indicate FGFs from Southern California, while blue points represent newly added data from Central and Northern California, and Nevada.

Figure 2: Precariously Balanced Rocks in California photographed by Jim Bune.

Methodology

In developing the Region 1 database for the FGF project, various code section were used in organizing, processing and developing the database.

- Data Collection
  The first step involved loading both excel and text files into pandas Dataframe to manage the name of geolocation and file path data.
- Name Matching
  The matching of names between the Excel file i.e location and file path
- Date Extraction
  The python functions is built to extract the dates from the file path.



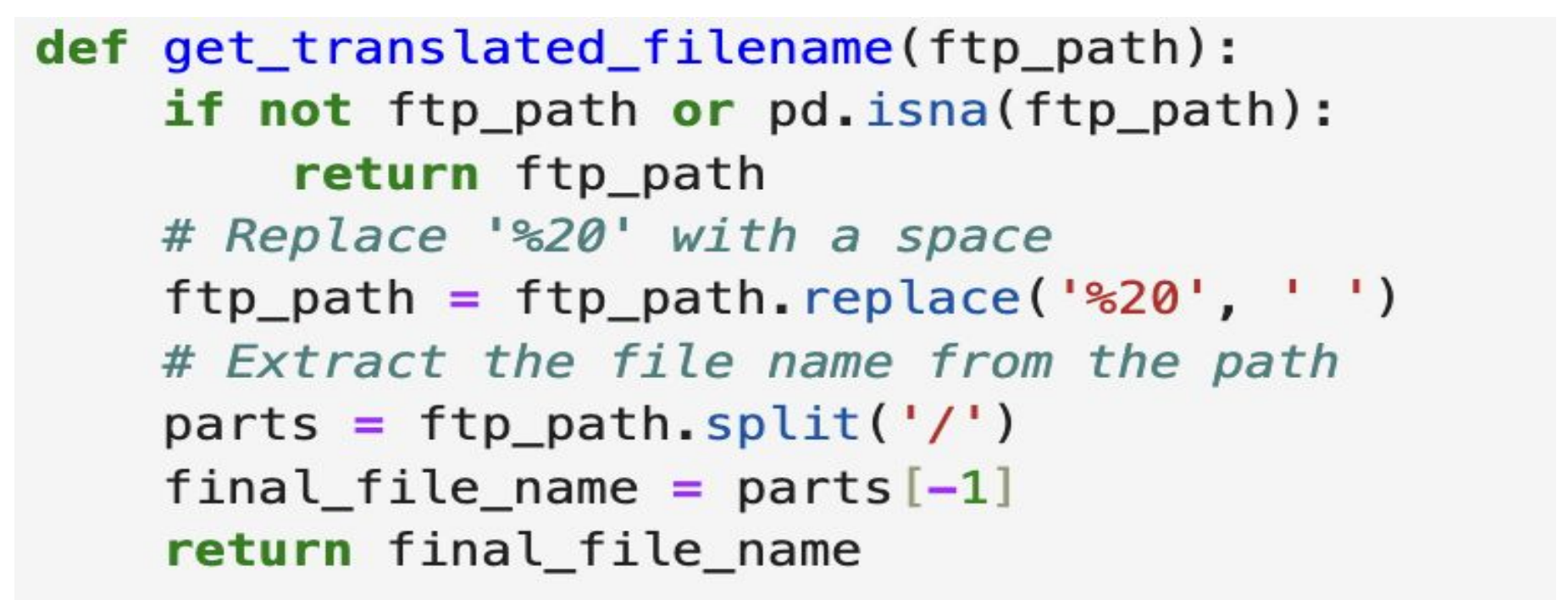
```python
def extract_and_format_date(file_path):
    # Regex patterns to match various date formats
    patterns = [
        r'(\d{2})[-_](\d{2})[-_](\d{2})',  # Matches formats like 09-14-92 or 09_14_92
        r'(\d{2})[-_](\d{2})[-_](\d{4})',  # Matches formats like 09-14-1992 or 09_14_1992
        r'(\d{4})[-_](\d{2})[-_](\d{2})',  # Matches formats like 1992-09-14 or 1992_09_14
        r'(\d{2})R.*(\d{2})\.\d+',         # Matches formats like 93RCSC16.8
        r'(\d{2})RCYC(\d{1,2})A',          # Matches formats like 93RCYC7A
        r'(\d{2})[-_](\d{2})'              # Matches formats like 06-93
    ]
    for pattern in patterns:
        match = re.search(pattern, file_path)
        if match:
            groups = match.groups()
            if len(groups) == 3:
                if len(groups[2]) == 2:  # Case where year is two digits
                    month, day, year = map(int, groups)
                    if year <= 99:
                        year += 1900 if year > 20 else 2000
                else:  # Case where year is four digits
                    year, month, day = map(int, groups)
            elif len(groups) == 2:
                if pattern == r'(\d{2})[-_](\d{2})':  # Specific case for MM-YY format
                    month, year_component = map(int, groups)
                    if year_component >= 50:  # Assuming 20th century
                        year = 1900 + year_component
                    else:  # Assuming 21st century
                        year = 2000 + year_component
                else:
                    year_component, month = groups
                    year_component = int(year_component)
                    month = int(month)
                    if year_component >= 50:  # Assuming 20th century
                        year = 1900 + year_component
                    else:  # Assuming 21st century
                        year = 2000 + year_component
            else:
                continue
            formatted_date = f'{year:04d}-{month:02d}'
            return formatted_date
    return None
```

- Final database for region 1

For each area, a temporary DataFrame was crated that contained key information such as area name, file paths, latitude, longitude and extracted dates. The process was repeated for all areas in datasets, and concatenated into a single final database.

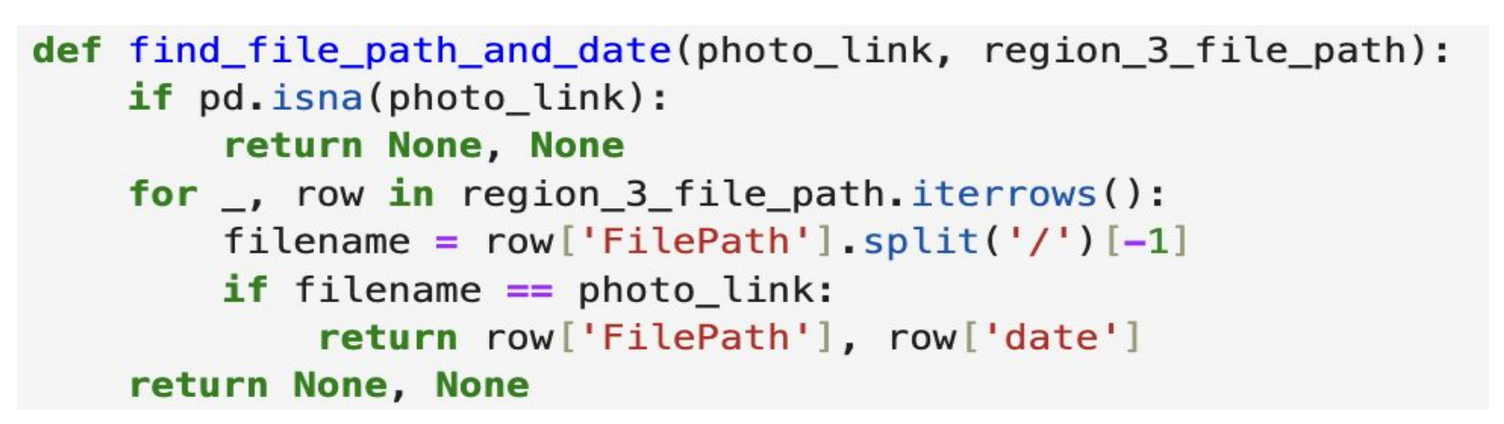
The region 3 and region 6 database was developed using same approach.

- File Name Translation
  The functions cleans up file path by replacing encoding spaces('%20'') with actual spaces and extract the file name from the given path.



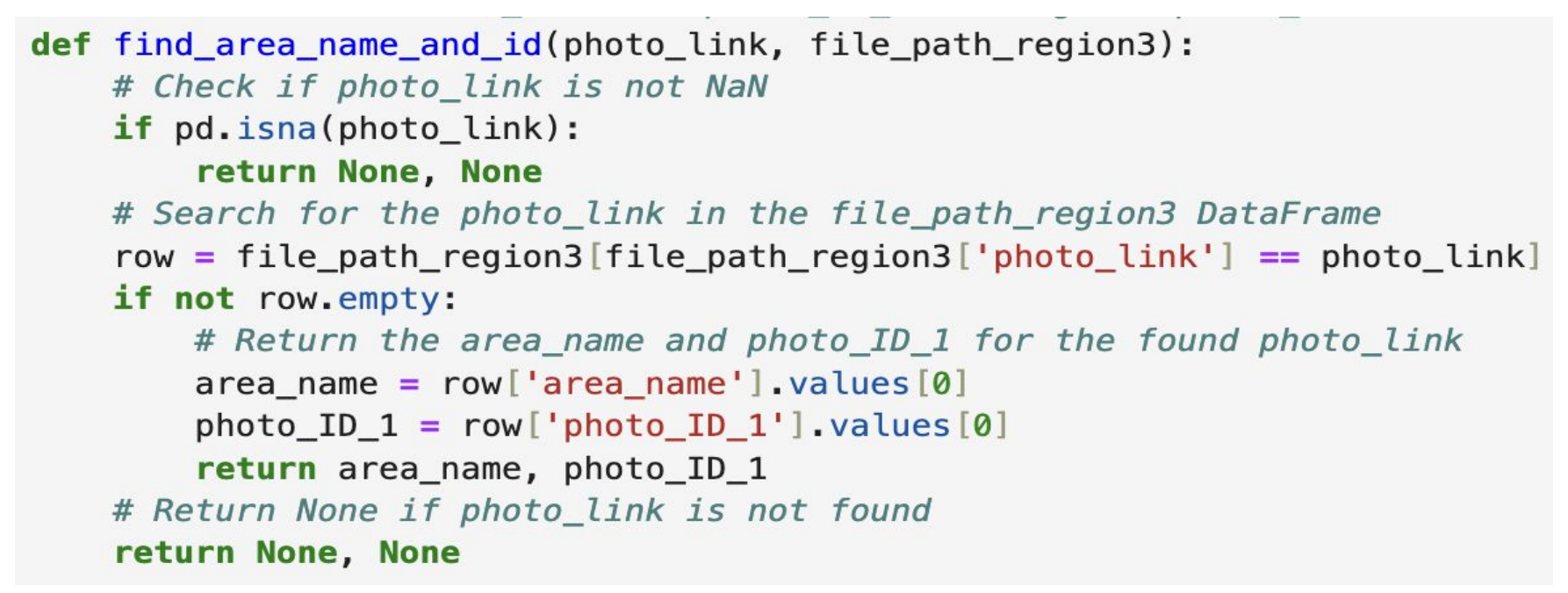
```python
def get_translated_filename(ftp_path):
    if not ftp_path or pd.isna(ftp_path):
        return ftp_path
    # Replace '%20' with a space
    ftp_path = ftp_path.replace('%20', ' ')
    # Extract the file name from the path
    parts = ftp_path.split('/')
    final_file_name = parts[-1]
    return final_file_name
```

- Date Extraction
  Similar approach like before was used for region 6 to extract a date from file path.
- File Path and Date Finder
  Matches the 'photo_link' to the corresponding file path and retrieves the associated date.



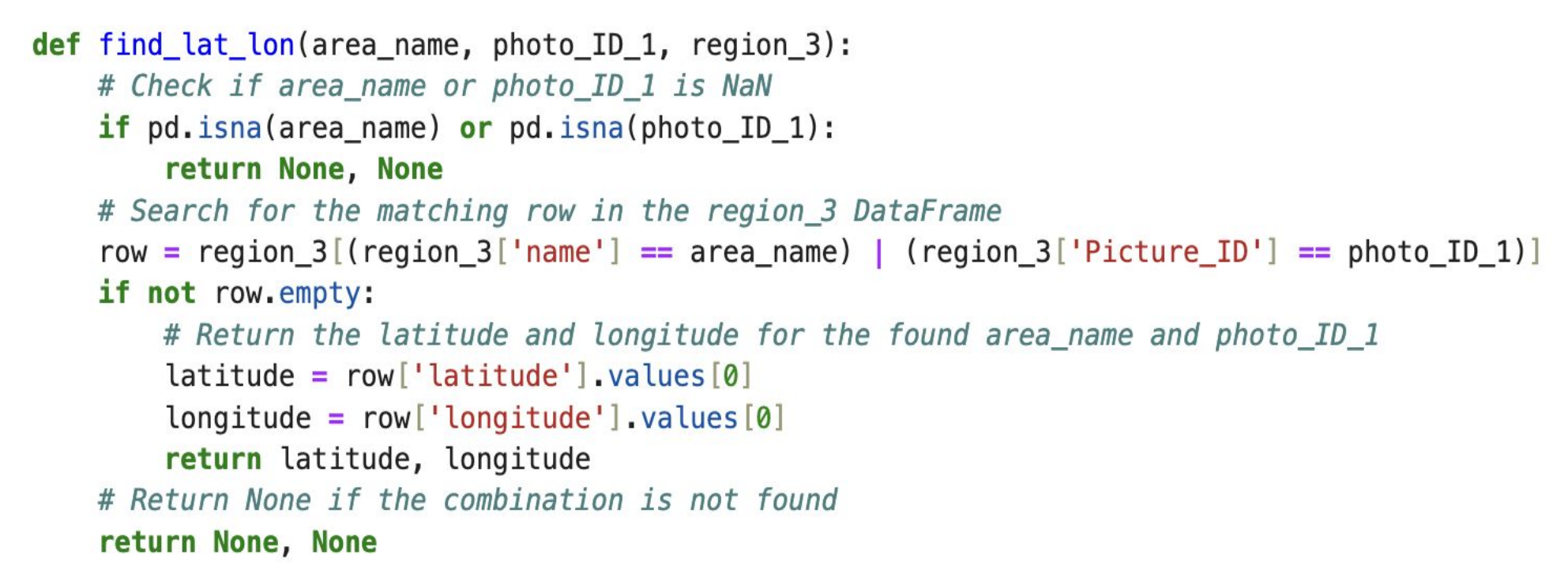
```python
def find_file_path_and_date(photo_link, region_3_file_path):
    if pd.isna(photo_link):
        return None, None
    for _, row in region_3_file_path.iterrows():
        filename = row['FilePath'].split('/')[-1]
        if filename == photo_link:
            return row['FilePath'], row['date']
    return None, None
```

- Area Name and ID Finder
  Find the 'area_name' and 'photo_ID' associated with the given 'photo_link'



```python
def find_area_name_and_id(photo_link, file_path_region3):
    # Check if photo_link is not NaN
    if pd.isna(photo_link):
        return None, None
    # Search for the photo_link in the file_path_region3 DataFrame
    row = file_path_region3[file_path_region3['photo_link'] == photo_link]
    if not row.empty:
        # Return the area_name and photo_ID_1 for the found photo_link
        area_name = row['area_name'].values[0]
        photo_ID_1 = row['photo_ID_1'].values[0]
        return area_name, photo_ID_1
    # Return None if photo_link is not found
    return None, None
```

- Latitude and Longitude Finder
  Retrieves the latitude and longitude for a given area name and photo ID.



```python
def find_lat_lon(area_name, photo_ID_1, region_3):
    # Check if area_name or photo_ID_1 is NaN
    if pd.isna(area_name) or pd.isna(photo_ID_1):
        return None, None
    # Search for the matching row in the region_3 DataFrame
    row = region_3[(region_3['name'] == area_name) | (region_3['Picture_ID'] == photo_ID_1)]
    if not row.empty:
        # Return the latitude and longitude for the found area_name and photo_ID_1
        latitude = row['latitude'].values[0]
        longitude = row['longitude'].values[0]
        return latitude, longitude
    # Return None if the combination is not found
    return None, None
```

Manual Identification of PBR
To ensure indentifiaction, each Precariously Balanced Rockin region 3 was manually inspected and assigned a unique Rock ID. If two or more photograph depicted the same PBR from different angles ot different conditions, they were labeled the same rock ID.

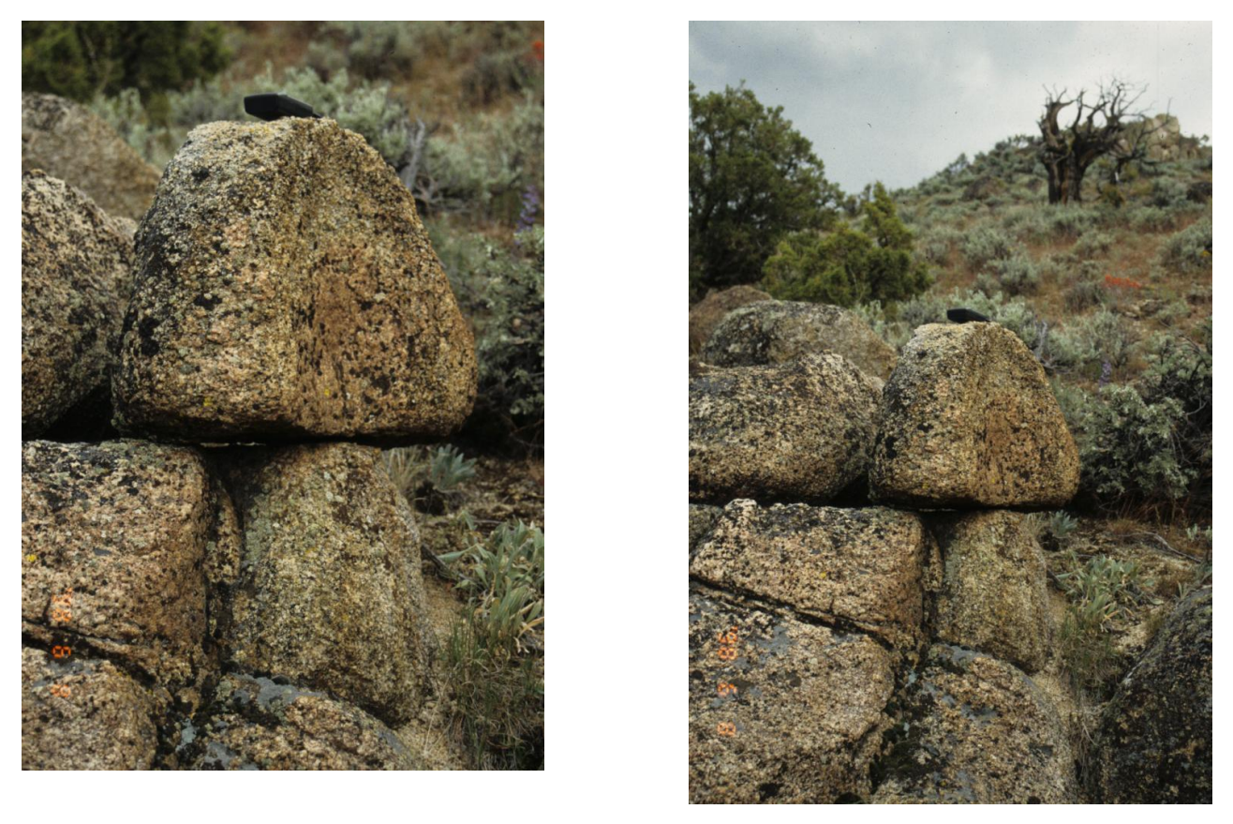


Figure 3: A PBR rock that is identified in two picture in a PBR database photographed by Jim Bune.

Results

Final Database Schema
Initially, it has anticipated that the database would contain around 3000 rows but due to the data quality, we're able to collect 2189 rows of valuable data.

Region ID	Area Name	Filename	Lon	Lat	Date
1	Rattlesnake	./Scanned:	-116.472	36.8133	1993-03

- Region 1 : Used an initial approach focused on manual data handling, integration and research.
- Region 3 and 6: Utilized a improved method incorporating different types of functions, resulting in more efficient data and compilation process.
- Identification and developed PBR database:
  The identification of PBR images and development of comprehensive database were conducted to enhance the accuracy and utility of seismic hazard assessment.