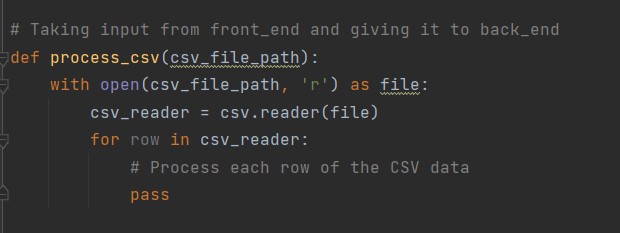
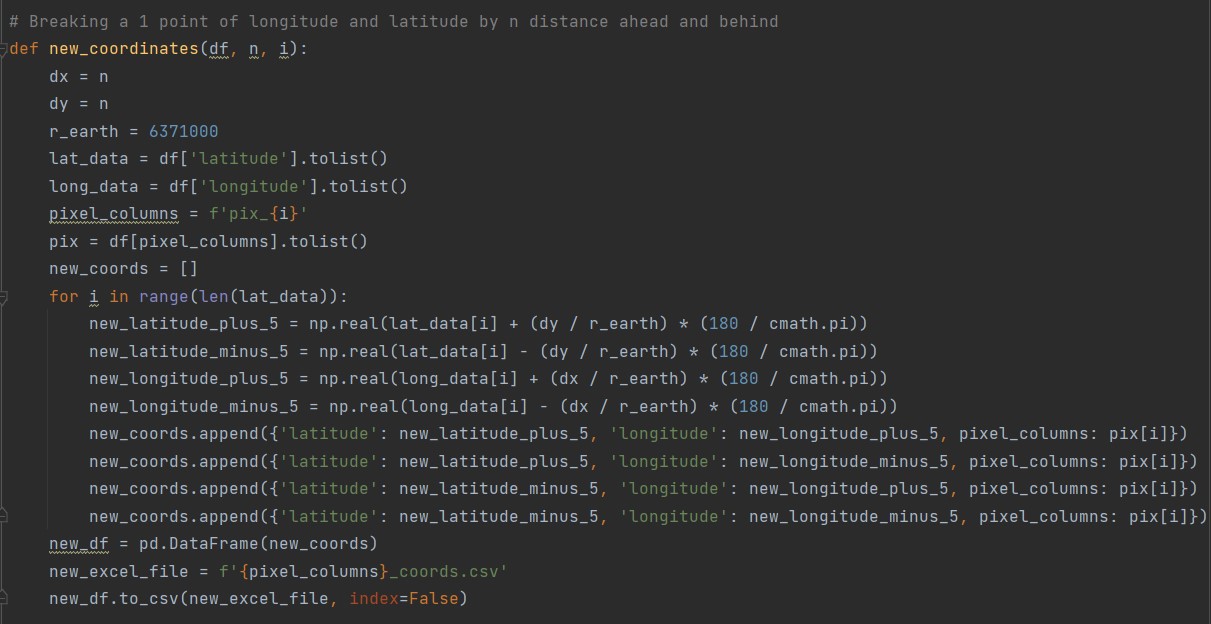
**Function wise explaination**

# Process\_csv (file\_path)



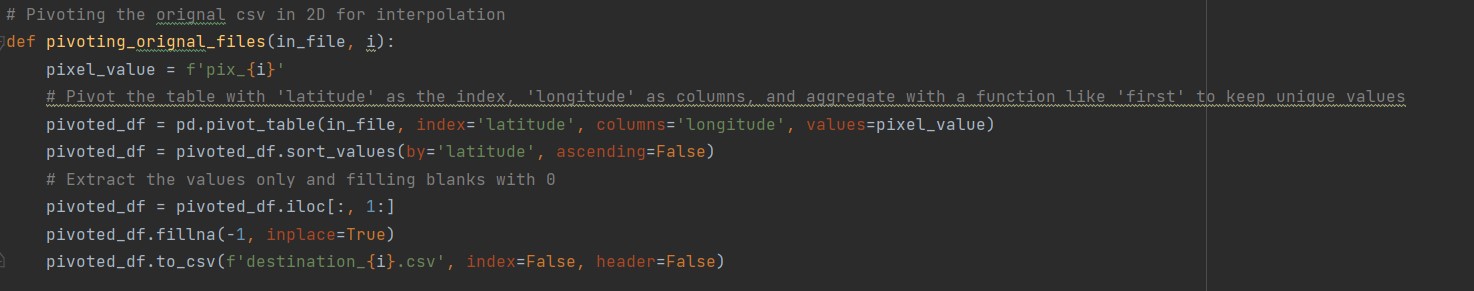
* The function **concatinate\_index\_header** takes four parameters: **input\_lat** (file path for unique latitudes CSV), **input\_long** (file path for unique longitudes CSV), **input\_file** (file path for the main input CSV), and **i** (an index or identifier).
* The code opens and reads the contents of the three CSV files (**input\_lat**, **input\_long**, and **input\_file**) using the **csv.reader** module.
* The data from each file is stored in separate lists (**data\_lat**, **data\_long**, and **data\_input**).
* The code then iterates through the columns of the main input data (**data\_input**) and replaces the header values with the unique longitudes from **data\_long**.
* It also iterates through the rows of the main input data and replaces the first column values with the unique latitudes from **data\_lat**.
* Finally, the modified data is written back to the original input file (**input\_file**) using the **csv.writer** module, effectively concatenating the unique latitude and longitude values to the existing 2D matrix.

# New\_coordinates (data frame, n, i)



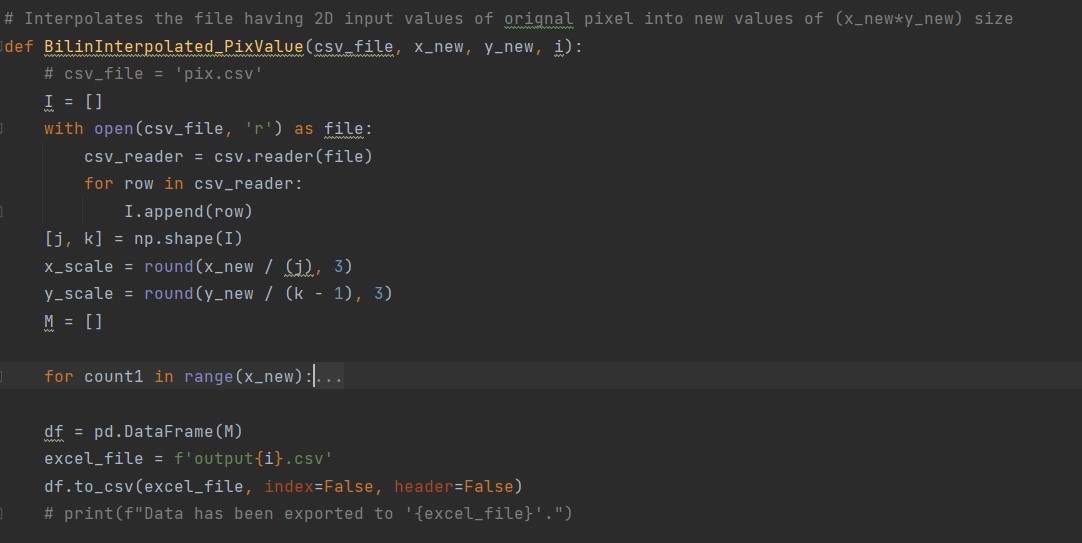
* The function **concatinate\_index\_header** takes four parameters: **input\_lat** (file path for unique latitudes CSV), **input\_long** (file path for unique longitudes CSV), **input\_file** (file path for the main input CSV), and **i** (an index or identifier).
* The code opens and reads the contents of the three CSV files (**input\_lat**, **input\_long**, and **input\_file**) using the **csv.reader** module.
* The data from each file is stored in separate lists (**data\_lat**, **data\_long**, and **data\_input**).
* The code then iterates through the columns of the main input data (**data\_input**) and replaces the header values with the unique longitudes from **data\_long**.
* It also iterates through the rows of the main input data and replaces the first column values with the unique latitudes from **data\_lat**.
* Finally, the modified data is written back to the original input file (**input\_file**) using the **csv.writer** module, effectively concatenating the unique latitude and longitude values to the existing 2D matrix.

# Pivoting\_orignal\_file (input\_file, i)



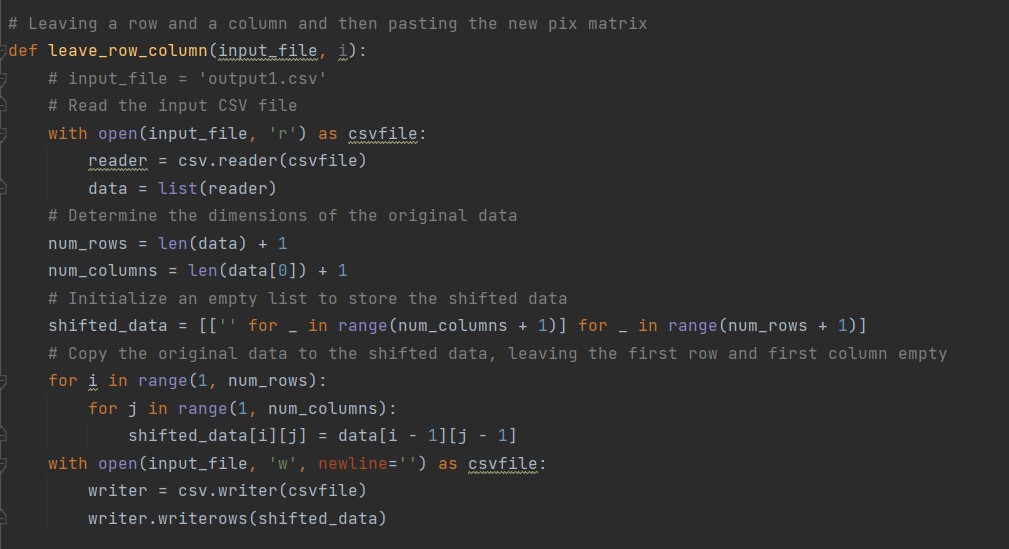
* The function **concatinate\_index\_header** takes four parameters: **input\_lat** (file path for unique latitudes CSV), **input\_long** (file path for unique longitudes CSV), **input\_file** (file path for the main input CSV), and **i** (an index or identifier).
* The code opens and reads the contents of the three CSV files (**input\_lat**, **input\_long**, and **input\_file**) using the **csv.reader** module.
* The data from each file is stored in separate lists (**data\_lat**, **data\_long**, and **data\_input**).
* The code then iterates through the columns of the main input data (**data\_input**) and replaces the header values with the unique longitudes from **data\_long**.
* It also iterates through the rows of the main input data and replaces the first column values with the unique latitudes from **data\_lat**.
* Finally, the modified data is written back to the original input file (**input\_file**) using the **csv.writer** module, effectively concatenating the unique latitude and longitude values to the existing 2D matrix.

# BilinearInterpolated\_PixValue (pivoted\_pix\_file, X , Y, i)



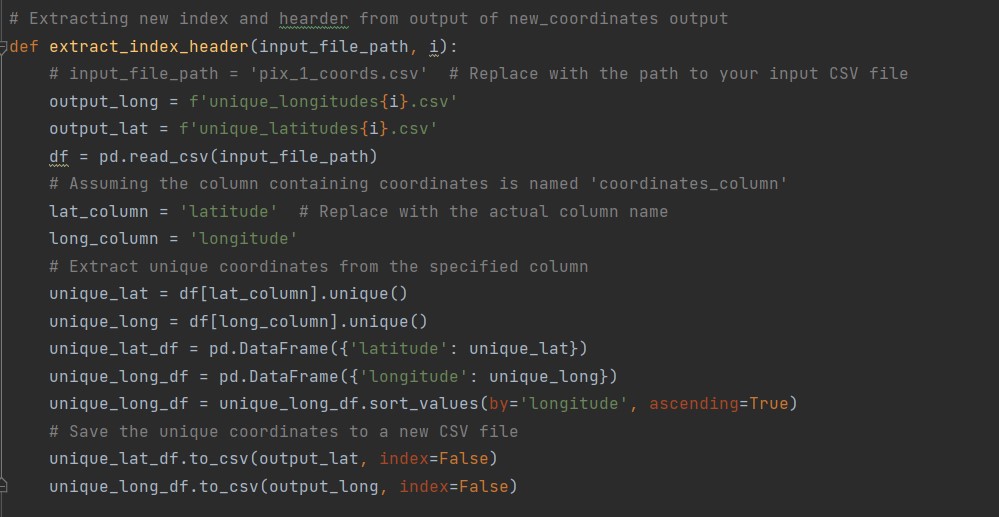
* The function **concatinate\_index\_header** takes four parameters: **input\_lat** (file path for unique latitudes CSV), **input\_long** (file path for unique longitudes CSV), **input\_file** (file path for the main input CSV), and **i** (an index or identifier).
* The code opens and reads the contents of the three CSV files (**input\_lat**, **input\_long**, and **input\_file**) using the **csv.reader** module.
* The data from each file is stored in separate lists (**data\_lat**, **data\_long**, and **data\_input**).
* The code then iterates through the columns of the main input data (**data\_input**) and replaces the header values with the unique longitudes from **data\_long**.
* It also iterates through the rows of the main input data and replaces the first column values with the unique latitudes from **data\_lat**.
* Finally, the modified data is written back to the original input file (**input\_file**) using the **csv.writer** module, effectively concatenating the unique latitude and longitude values to the existing 2D matrix.

# Leave\_row\_column (input\_file, i)



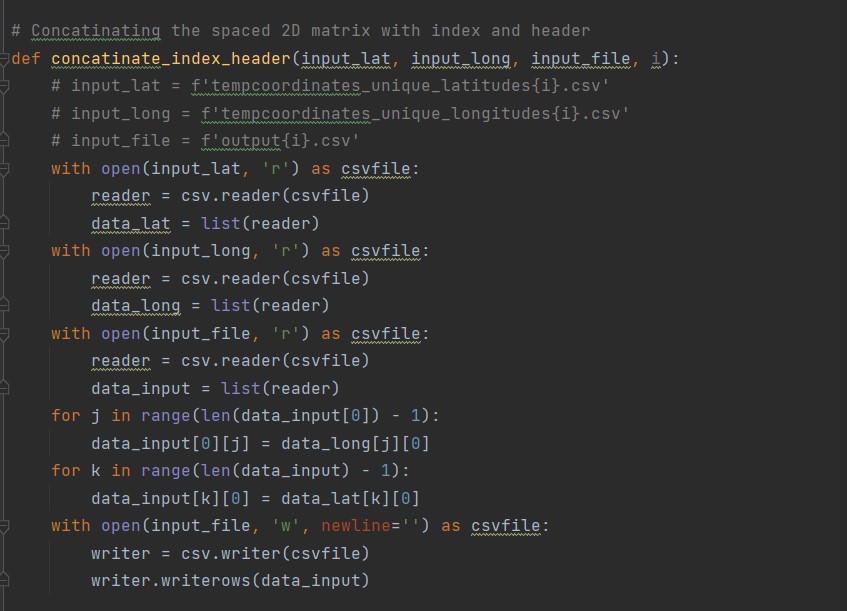
* The function **concatinate\_index\_header** takes four parameters: **input\_lat** (file path for unique latitudes CSV), **input\_long** (file path for unique longitudes CSV), **input\_file** (file path for the main input CSV), and **i** (an index or identifier).
* The code opens and reads the contents of the three CSV files (**input\_lat**, **input\_long**, and **input\_file**) using the **csv.reader** module.
* The data from each file is stored in separate lists (**data\_lat**, **data\_long**, and **data\_input**).
* The code then iterates through the columns of the main input data (**data\_input**) and replaces the header values with the unique longitudes from **data\_long**.
* It also iterates through the rows of the main input data and replaces the first column values with the unique latitudes from **data\_lat**.
* Finally, the modified data is written back to the original input file (**input\_file**) using the **csv.writer** module, effectively concatenating the unique latitude and longitude values to the existing 2D matrix.

# Extract\_index\_header (input\_file, i)



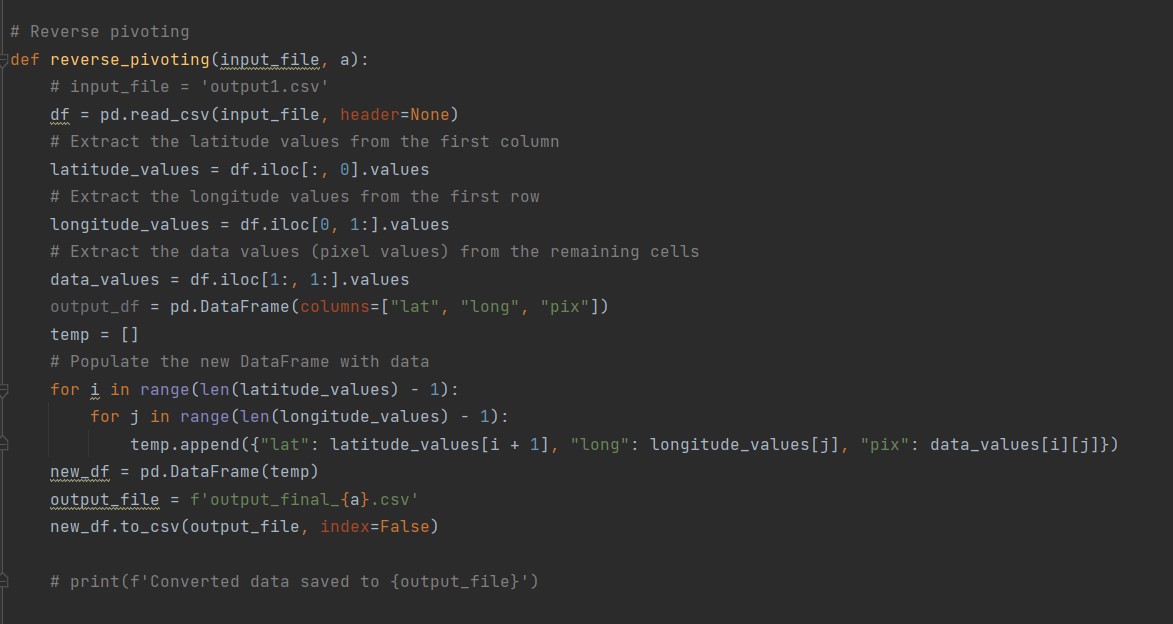
* The function **concatinate\_index\_header** takes four parameters: **input\_lat** (file path for unique latitudes CSV), **input\_long** (file path for unique longitudes CSV), **input\_file** (file path for the main input CSV), and **i** (an index or identifier).
* The code opens and reads the contents of the three CSV files (**input\_lat**, **input\_long**, and **input\_file**) using the **csv.reader** module.
* The data from each file is stored in separate lists (**data\_lat**, **data\_long**, and **data\_input**).
* The code then iterates through the columns of the main input data (**data\_input**) and replaces the header values with the unique longitudes from **data\_long**.
* It also iterates through the rows of the main input data and replaces the first column values with the unique latitudes from **data\_lat**.
* Finally, the modified data is written back to the original input file (**input\_file**) using the **csv.writer** module, effectively concatenating the unique latitude and longitude values to the existing 2D matrix.

# Concatenate\_index\_header (input\_lat, input\_long, input\_file, i)



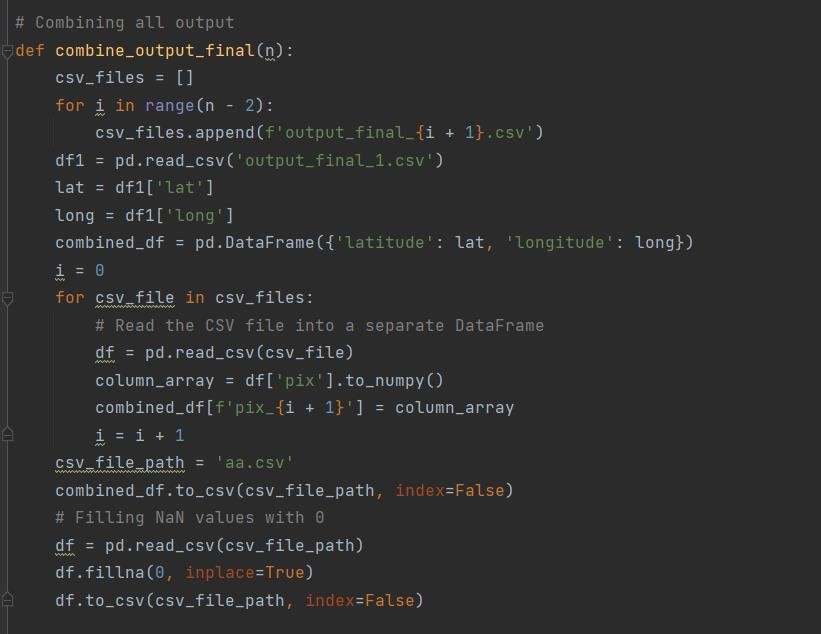
* The function **concatinate\_index\_header** takes four parameters: **input\_lat** (file path for unique latitudes CSV), **input\_long** (file path for unique longitudes CSV), **input\_file** (file path for the main input CSV), and **i** (an index or identifier).
* The code opens and reads the contents of the three CSV files (**input\_lat**, **input\_long**, and **input\_file**) using the **csv.reader** module.
* The data from each file is stored in separate lists (**data\_lat**, **data\_long**, and **data\_input**).
* The code then iterates through the columns of the main input data (**data\_input**) and replaces the header values with the unique longitudes from **data\_long**.
* It also iterates through the rows of the main input data and replaces the first column values with the unique latitudes from **data\_lat**.
* Finally, the modified data is written back to the original input file (**input\_file**) using the **csv.writer** module, effectively concatenating the unique latitude and longitude values to the existing 2D matrix.

# Reverse\_pivoting (input\_file, a)



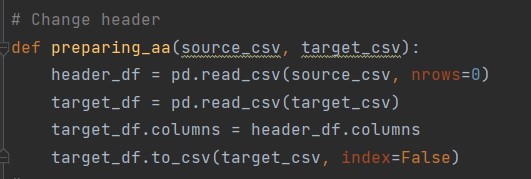
* The function **reverse\_pivoting** takes two parameters: **input\_file** (the path to the input CSV file) and **a** (some identifier or index).
* It uses the pandas library (**pd**) to read the input CSV file into a DataFrame (**df**) with no header.
* The latitude values are extracted from the first column of the DataFrame.
* The longitude values are extracted from the first row of the DataFrame.
* The pixel (data) values are extracted from the remaining cells of the DataFrame.
* A new DataFrame (**output\_df**) with columns "lat", "long", and "pix" is created.
* A temporary list (**temp**) is populated with dictionaries containing latitude, longitude, and pixel values.
* Another DataFrame (**new\_df**) is created from the temporary list.
* The new DataFrame is saved to a new CSV file (**output\_file**) with a name based on the identifier **a**.
* The index is set to False to avoid writing row indices to the CSV file.
* The function effectively reverses the process of pivoting, transforming the input data back into a "long" format.
* The commented out print statement can be used to display a message indicating where the converted data is saved.

# Combine\_output\_final (n)



* The function **combine\_output\_final** takes one parameter **n**, which represents the number of output files to be combined.
* A list **csv\_files** is created to store the file names of the output CSV files.
* A loop iterates from 0 to **n - 3** (inclusive) and appends the file names **output\_final\_{i + 1}.csv** to the **csv\_files** list.
* The first output file (**output\_final\_1.csv**) is read into a DataFrame (**df1**).
* The latitude and longitude columns are extracted from **df1** and stored in separate Series (**lat** and **long**).
* A new DataFrame (**combined\_df**) is created with 'latitude' and 'longitude' columns.
* Another loop iterates through the remaining CSV files in **csv\_files**.
* Each CSV file is read into a DataFrame (**df**), and the 'pix' column is extracted and added to **combined\_df** with a column name based on the iteration index.
* The final combined DataFrame is saved to a new CSV file (**csv\_file\_path**) named 'aa.csv'.
* The function then reads 'aa.csv' into a new DataFrame (**df**).
* NaN values in the DataFrame are filled with 0 using the **fillna** method.
* The modified DataFrame is saved back to 'aa.csv' to ensure there are no NaN values in the final output.

# Preparing\_aa (source\_csv, target\_csv)

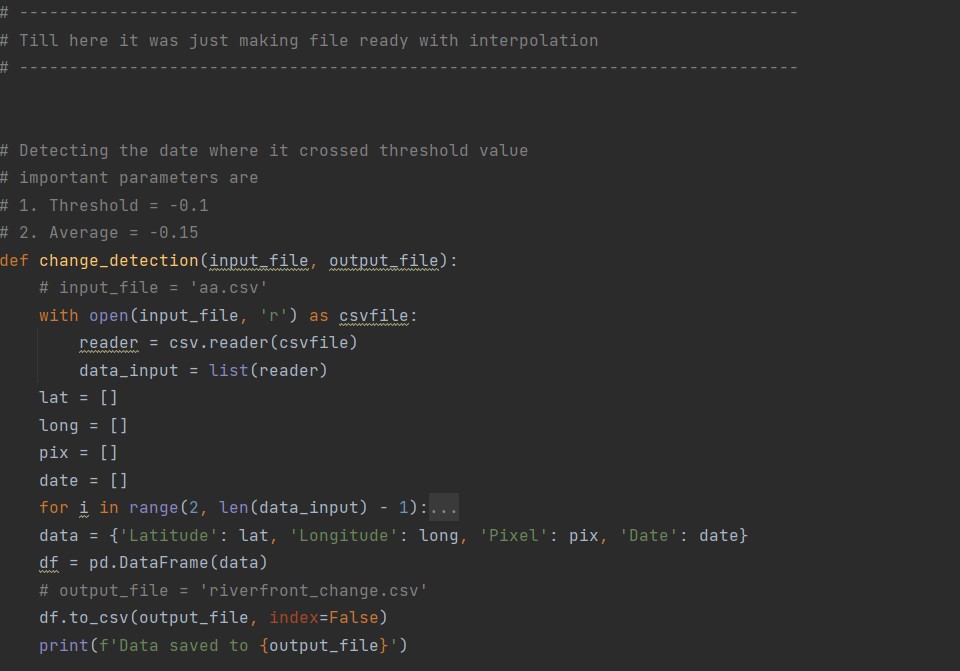


* The function **preparing\_aa** takes two parameters: **source\_csv** (the path to the source CSV file) and **target\_csv** (the path to the target CSV file).
* **pd.read\_csv(source\_csv, nrows=0)** is used to read only the header of the source CSV file into a DataFrame (**header\_df**). This is done to extract the column names.
* **pd.read\_csv(target\_csv)** reads the entire content of the target CSV file into a DataFrame (**target\_df**).
* The column names of **target\_df** are then replaced with the column names from **header\_df** using **target\_df.columns = header\_df.columns**.
* The modified DataFrame (**target\_df**) with the correct column names is then saved back to the target CSV file (**target\_csv**) using **to\_csv** with **index=False**. This overwrites the existing target CSV file.

# Change\_detection (input\_file, output\_file)

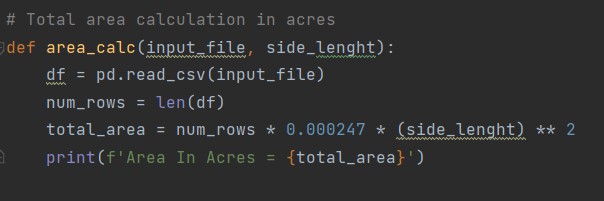
**Imp\_Parameters – Threshold = -0.1**

**Average = -0.15**



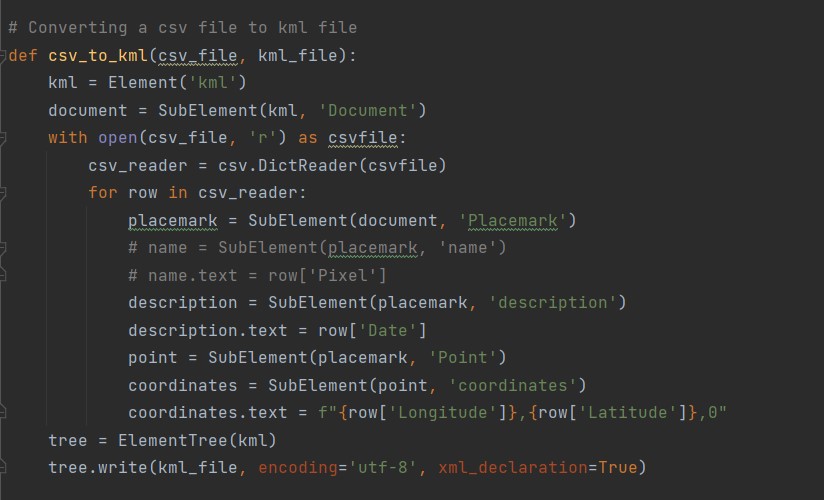
* The function **change\_detection** takes two parameters: **input\_file** (the path to the input CSV file) and **output\_file** (the path to the output CSV file).
* The code opens and reads the contents of the input CSV file (**input\_file**) using the **csv.reader** module.
* Four empty lists (**lat**, **long**, **pix**, **date**) are initialized to store latitude, longitude, pixel, and date values.
* The code then iterates over the rows and columns of the input data, starting from row index 2 and column index 2.
* For each valid condition (checking values in the input data), the code extracts latitude, longitude, pixel, and date information and appends them to the respective lists.
* The extracted data is then organized into a dictionary (**data**) and converted into a DataFrame (**df**) using pandas.
* The DataFrame is saved to the specified output CSV file (**output\_file**) using **to\_csv** with **index=False** to exclude row indices.
* A message is printed indicating where the output data is saved.

# Area\_calc (inp, area\_name)



* The function **area\_calc** takes two parameters: **input\_file** (the path to the input CSV file) and **side\_length** (the length of one side of a square).
* The code uses pandas (**pd**) to read the input CSV file into a DataFrame (**df**).
* The variable **num\_rows** is assigned the number of rows in the DataFrame, calculated using the **len** function.
* The total area is then calculated using the formula **total\_area = num\_rows \* 0.000247 \* (side\_length) \*\* 2**. The multiplication by **0.000247** is a conversion factor to convert square meters to acres.
* A message is printed to the console, indicating the calculated area in acres.

# Csv\_to\_kml (csv\_file, kml\_file)



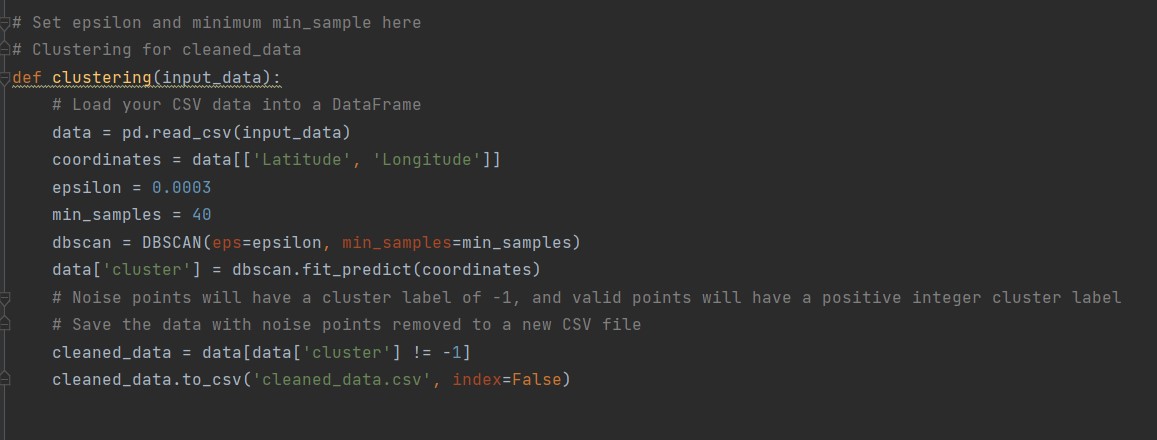
* The function **csv\_to\_kml** takes two parameters: **csv\_file** (the path to the input CSV file) and **kml\_file** (the path to the output KML file).
* The code uses the **Element** and **SubElement** classes from the **xml.etree.ElementTree** module to create an XML structure for KML.
* The root element is created as a **kml** element.
* Inside the **kml** element, a **Document** element is created.
* The code then opens the CSV file (**csv\_file**) and reads its contents using **csv.DictReader**.
* For each row in the CSV file, a **Placemark** element is created within the **Document** element.
* Inside each **Placemark** element, a **description** element is created and populated with the value from the 'Date' column of the CSV file.
* A **Point** element is created within each **Placemark** element.
* Inside each **Point** element, a **coordinates** element is created and populated with the longitude, latitude, and 0 for altitude from the corresponding columns in the CSV file.
* The resulting XML structure is written to the output KML file (**kml\_file**) using **ElementTree.write**.

# Date\_filter (inp, area\_name)



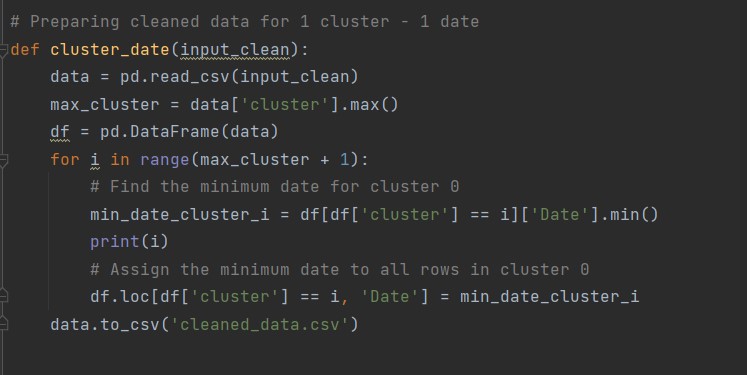
* The function **date\_filter** takes two parameters: **inp** (the path to the input CSV file) and **area\_name** (a string representing the area name).
* The code uses pandas (**pd**) to read the input CSV file into a DataFrame (**df**).
* The 'Date' column in the DataFrame is converted to datetime objects using **pd.to\_datetime**. The parameter **errors='coerce'** is used to handle invalid date strings gracefully by converting them to NaT (Not a Time) values.
* The DataFrame is then filtered to include only rows with dates after or equal to the year 2019. The filtering is done using the **dt.year** attribute of the 'Date' column.
* The path for the filtered CSV file is specified based on the **area\_name** parameter, and a prefix of '1\_' is added.
* The filtered data is saved to a new CSV file using **to\_csv**, and the **index** parameter is set to **False** to exclude row indices.
* The function is currently commented out to avoid printing a message to the console, but the message can be uncommented if needed.
* **Note**: The purpose of this function is to filter data based on the 'Date' column, specifically keeping only rows with dates after or equal to the year 2019. The filtered data is then saved to a new CSV file.

# Clustering (input\_data)



* The function **clustering** takes one parameter: **input\_data** (the path to the input CSV file).
* The code uses pandas (**pd**) to read the input CSV file into a DataFrame (**data**).
* A new DataFrame **coordinates** is created, containing only the 'Latitude' and 'Longitude' columns from the original data.
* The DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm is applied to the coordinates using the **DBSCAN** class from scikit-learn. The parameters **eps** and **min\_samples** control the clustering behavior.
* The resulting cluster labels are added as a new column ('cluster') to the original DataFrame.
* Noise points, which have a cluster label of -1, are removed from the data, and the cleaned data is saved to a new CSV file named 'cleaned\_data.csv' using **to\_csv** with **index=False**.
* **Note**: The DBSCAN algorithm is a density-based clustering algorithm that groups together points that are close to each other based on a distance metric. Noise points, which do not belong to any cluster, are labeled with -1. The code here applies DBSCAN to spatial coordinates and removes noise points, saving the cleaned data to a new CSV file. Adjustments to **epsilon** and **min\_samples** may be needed based on the specific characteristics of the data.

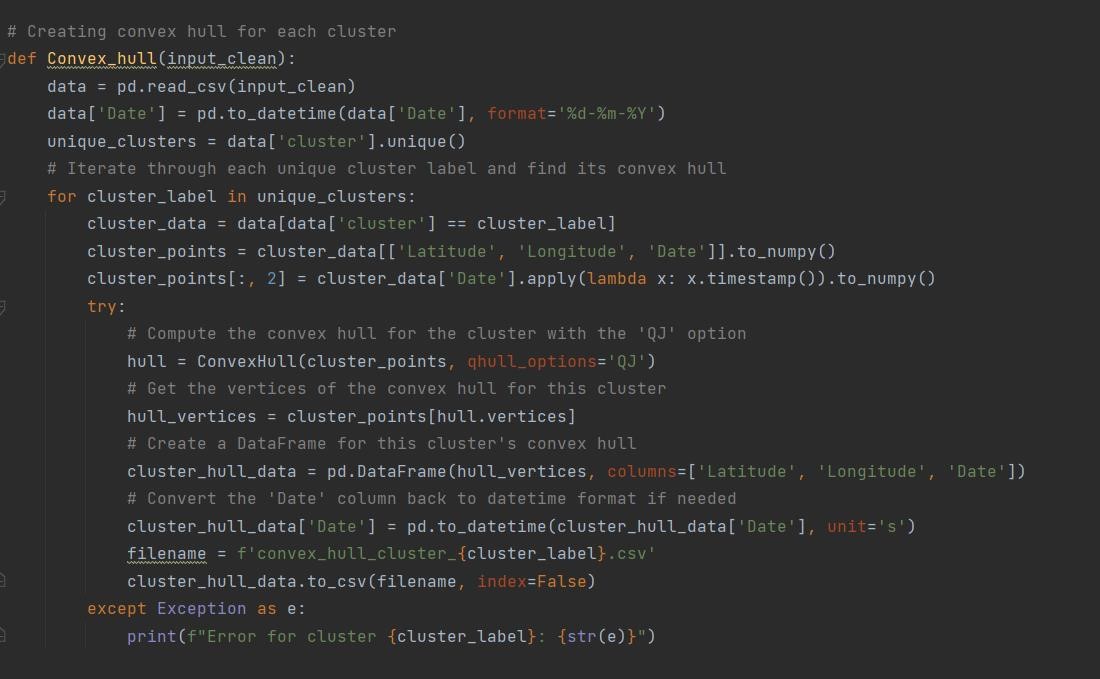
# Cluster\_date (input\_clean)



* The function **cluster\_date** takes one parameter: **input\_clean** (the path to the cleaned CSV file).
* The code uses pandas (**pd**) to read the cleaned CSV file into a DataFrame (**data**).
* The maximum cluster label is determined by finding the maximum value in the 'cluster' column.
* A new DataFrame **df** is created as a copy of the original data.
* The code then iterates over each cluster label from 0 to the maximum cluster label (**max\_cluster**).
* For each cluster label, it finds the minimum date (**min\_date\_cluster\_i**) for that cluster using the 'Date' column.
* It then updates the 'Date' column for all rows in that cluster to be the minimum date for that cluster.
* The modified data is saved back to the same CSV file ('cleaned\_data.csv') using **to\_csv**. Note that the modification is done on the DataFrame **df**, but the saving is done on the original DataFrame **data**.

**Note**: This code appears to be assigning the minimum date within each cluster to all rows in that cluster. The goal seems to be to ensure that all rows within a cluster have the same date value, which is the minimum date for that cluster. The modified data is then saved back to the same CSV file.

# Convex\_hull (input\_clean)



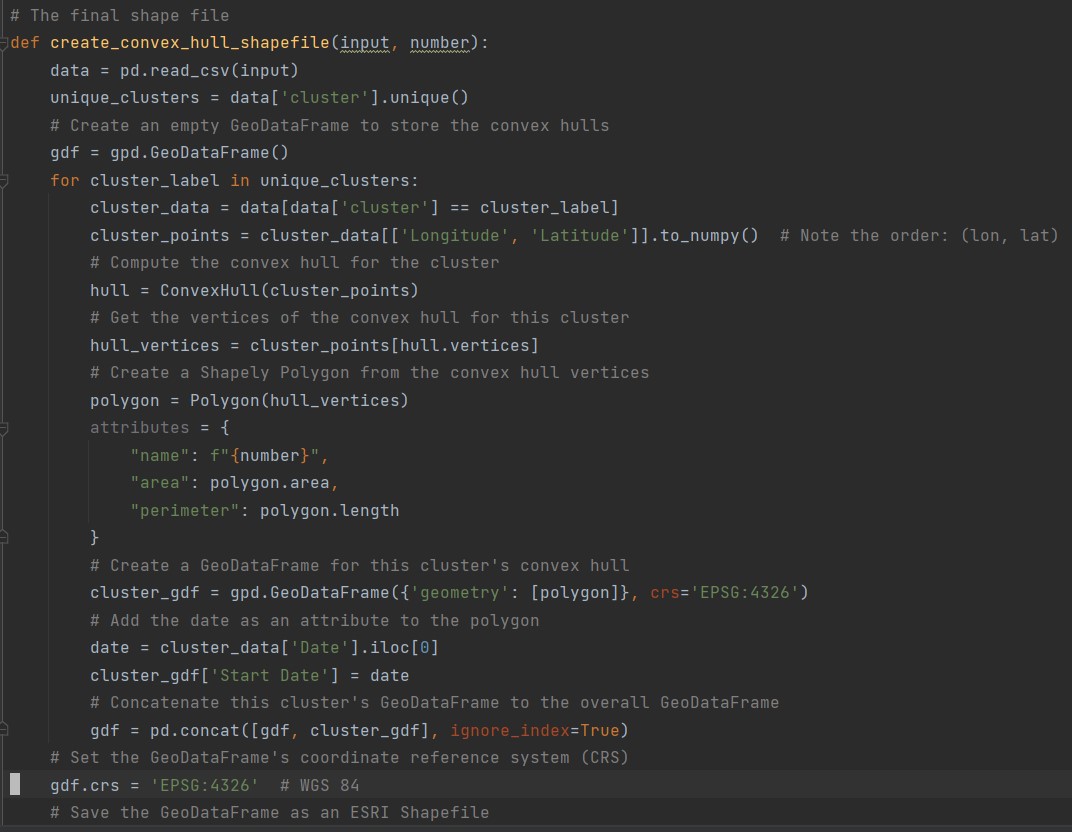
* The function **Convex\_hull** takes one parameter: **input\_clean** (the path to the cleaned CSV file).
* The code uses pandas (**pd**) to read the cleaned CSV file into a DataFrame (**data**).
* The 'Date' column in the DataFrame is converted to datetime objects using **pd.to\_datetime** with the specified format.
* Unique cluster labels are extracted from the 'cluster' column.
* The code then iterates through each unique cluster label.
* For each cluster, it extracts the relevant data ('Latitude', 'Longitude', 'Date') and converts the 'Date' column to timestamp format.
* The convex hull is computed for the cluster points using the **ConvexHull** function from the **scipy.spatial** module. The 'QJ' option is specified for better robustness.
* The vertices of the convex hull are extracted.
* A new DataFrame (**cluster\_hull\_data**) is created from the convex hull vertices.
* The 'Date' column is converted back to datetime format if needed.
* The resulting DataFrame is saved to a CSV file named

'convex\_hull\_cluster\_{cluster\_label}.csv', where **{cluster\_label}** is the specific cluster label.

* In case of any exceptions during the convex hull computation, an error message is printed to the console, specifying the cluster for which the error occurred.

Note: This code computes the convex hull for each unique cluster in the data and saves the resulting convex hull data to individual CSV files. The 'QJ' option is used to enhance robustness in convex hull computation.

# Create\_shapefile (input, number)



* The function **create\_convex\_hull\_shapefile** takes two parameters: **input** (the path to the input CSV file) and **number** (a number used in the output shapefile name).
* The code uses pandas (**pd**) to read the input CSV file into a DataFrame (**data**).
* Unique cluster labels are extracted from the 'cluster' column.
* An empty GeoDataFrame (**gdf**) is created to store the convex hulls. GeoDataFrame is a Pandas DataFrame with a geometry column for storing geometric objects.
* The code then iterates through each unique cluster label.
* For each cluster, it extracts the relevant data ('Longitude', 'Latitude').
* The convex hull is computed for the cluster points using the **ConvexHull** function from the **scipy.spatial** module.
* The vertices of the convex hull are extracted.
* A Shapely Polygon is created from the convex hull vertices.
* Attributes such as "name," "area," and "perimeter" are assigned to the polygon.
* A GeoDataFrame (**cluster\_gdf**) is created for this cluster's convex hull, with the polygon as the geometry and additional attributes.
* The GeoDataFrame is set to the coordinate reference system (CRS) 'EPSG:4326' (WGS 84).
* The GeoDataFrame is saved as an ESRI Shapefile with the filename **{number}.shp**.

Note: This code creates a shapefile containing convex hulls for each unique cluster in the input CSV data. The shapefile includes information about the convex hull's area, perimeter, start date, and other attributes. The shapefile is saved with a filename based on the provided **number**.

Main Code Explanation

1. **Initial Data Processing:**
   * The script takes a CSV file path as a command-line argument.
   * It renames columns and creates a new CSV file with standardized column names.
2. **Data Processing Loop:**
   * Iterates through pixel columns in the processed CSV file.
   * Creates new DataFrames for latitude, longitude, and each pixel column.
   * Saves each DataFrame to a separate CSV file.
   * Applies several functions (**new\_coordinates**, **pivoting\_orignal\_files**, etc.) to each pixel's DataFrame.
3. **Change Detection and Analysis:**
   * Combines the processed output files.
   * Prepares a common CSV file ('aa.csv').
   * Performs change detection and saves the result to a new CSV file.
   * Calculates and prints the area before and after change detection.
4. **Clustering and Further Processing:**
   * Applies clustering to the cleaned data.
   * Prepares the data for 1 cluster - 1 date.
   * Creates convex hulls for each cluster and saves them as shapefiles.
5. **Final Steps:**
   * Extracts a numeric identifier from the input file name.
   * Calls a function to create convex hull shapefiles based on cleaned data.

**The Work Flow**

**Step0 –** Open cmd prompt put -> **cd “path/to/your/file”** and then put -> **python app.py**

**Step1 –** Select the area of interest manually and save that as a shapefile.

**Step2** – Select the shapefile and create a grid of 0.007 in vertical and horizontal

**Step3** – Zip all the files of the grid and save it in the uploads folder for downloading the data

**Step4** – In the images folder look for the dates with cloud coverage and delete that dates from the excel file   
  
**Step5** – After getting the csv for each grid run the app.py on the system and give the csv as the input.

**Step6** – Give the path of the folder where the output files are generated in the last part of the code so that it will delete the unnecessary files   
  
**Step7** – The output shape file and the kml file will be stored in the folder

**The Logic**

* First The data is downloaded from the sentinel hub.
* The csv is then called in the front end of the code and it is linked to backend
* The csv is first called in for loop and it is prepared in format such that in each csv has latitude longitude and a single date in each csv
* Then prepare that CSV for Bilinear interpolation and perform the bilinear interpolation
* Then concatenate all the single date csv’s to a csv in which we have all new coordinates and new values
* Then we find the change detection in the give file
* Cluster the detected data
* Put the date to each cluster
* Make convex hulls and covert it to shape file and save it.