OVERVIEW

This documentation provides an in-depth explanation of a suite of Python scripts designed for geospatial data processing and analysis. The key components of this suite include:

1. **Web Crawler Script** (Data_Downloader Class):
    - Automates the downloading of data files from a specified URL directory, saving them to a local directory.
2. **Quarterly.Py Script**:
    - Processes NetCDF data files to calculate average soil moisture. The script generates visualizations of the processed data, presenting the results in a PDF format.
3. **Time Series Matrix.py**:
    - Creates a detailed time series matrix of daily soil moisture data for individual wards in Kenya.

1. Web crawler script

Data_Downloader Class

The Data_downloader class provides methods to download files from a specified URL directory and save them to a local directory. It includes functionality to handle both files and directories, and it supports recursive downloading. Uses REGEX to filter relative links

Attributes:

directory_url (str): The URL of the directory from which files will be downloaded.

download_path (str): The local directory path where the files will be saved.

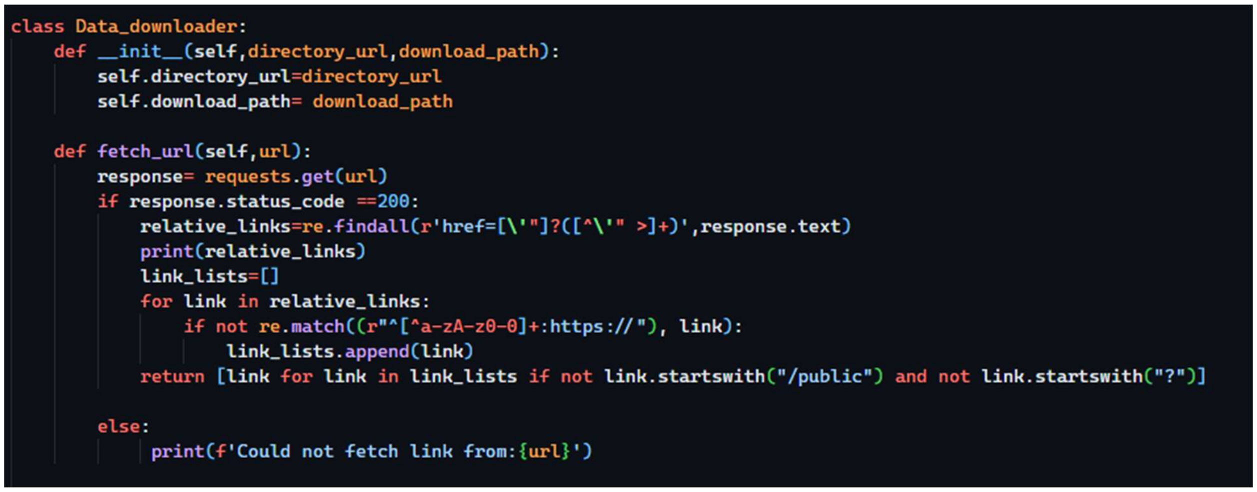
Method: fetch_Url

Fetches the HTML content from the given URL and extracts relative links.

Parameters:
url (str): The URL to fetch the HTML content from

Returns:

List: A list of relative links found in the HTML content



```python
class Data_downloader:
    def __init__(self,directory_url,download_path):
        self.directory_url=directory_url
        self.download_path= download_path

    def fetch_url(self,url):
        response= requests.get(url)
        if response.status_code ==200:
            relative_links=re.findall(r'href=[\'"]?([^\'" >]+)',response.text)
            print(relative_links)
            link_lists=[]
            for link in relative_links:
                if not re.match((r"^[^a-zA-z0-0]+:https://"), link):
                    link_lists.append(link)
            return [link for link in link_lists if not link.startswith("/public") and not link.startswith("?")]

        else:
             print(f'Could not fetch link from:{url}')
```

Method: file_downloader

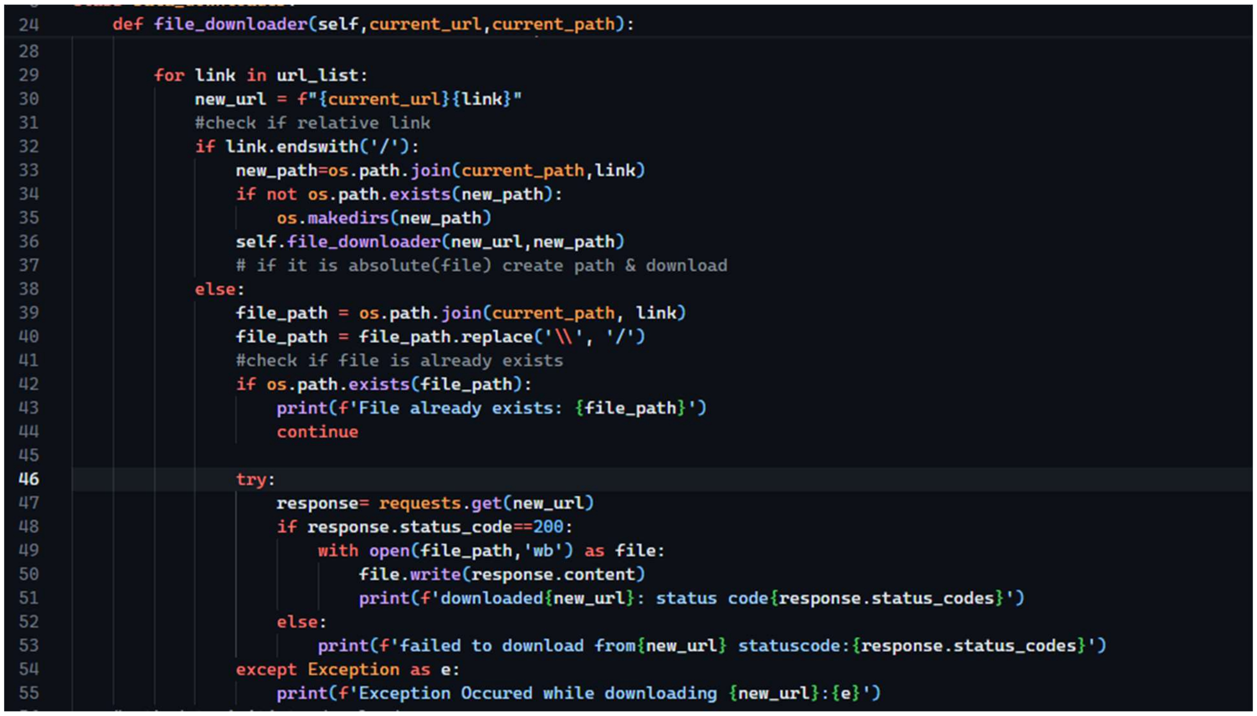
It invokes the fetch url method to get the list of the relative links and recursively iterates through the relative link while it creates directories in the local storage to store the data in a similar format to the TAMSAT data repository

Parameters:

current_url (str): The current URL to fetch data from.
current_path (str): The local directory path where files will be saved

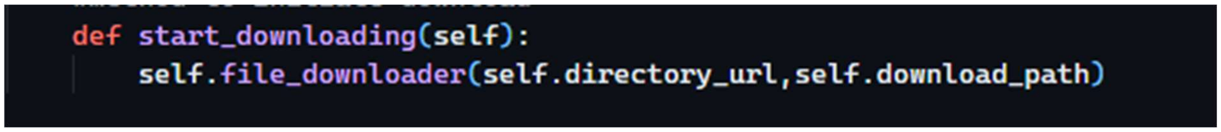
Raises:

Exception: If an error occurs during the downloading process



```python
    def file_downloader(self,current_url,current_path):

        for link in url_list:
            new_url = f"{current_url}{link}"
            #check if relative link
            if link.endswith('/'):
                new_path=os.path.join(current_path,link)
                if not os.path.exists(new_path):
                    os.makedirs(new_path)
                self.file_downloader(new_url,new_path)
                # if it is absolute(file) create path & download
            else:
                file_path = os.path.join(current_path, link)
                file_path = file_path.replace('\\', '/')
                #check if file is already exists
                if os.path.exists(file_path):
                    print(f'File already exists: {file_path}')
                    continue

                try:
                    response= requests.get(new_url)
                    if response.status_code==200:
                        with open(file_path,'wb') as file:
                            file.write(response.content)
                            print(f'downloaded{new_url}: status code{response.status_codes}')
                    else:
                        print(f'failed to download from{new_url} statuscode:{response.status_codes}')
                except Exception as e:
                    print(f'Exception Occured while downloading {new_url}:{e}')
```

Method: start_downloading
Initiates the downloading process by calling the **file_downloader** method with the starting URL and path

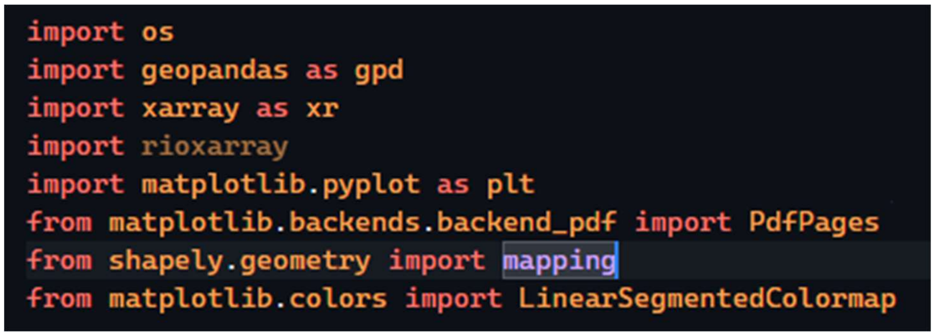


```python
    def start_downloading(self):
        self.file_downloader(self.directory_url,self.download_path)
```

2. Quaterly.Py Script

Importing necessary Packages:

The script processes NetCDF data files, calculates average soil moisture, and visualizes the results in a PDF



```python
import os
import geopandas as gpd
import xarray as xr
import rioxarray
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
from shapely.geometry import mapping
from matplotlib.colors import LinearSegmentedColormap
```

Geopands – python library for handling shape files

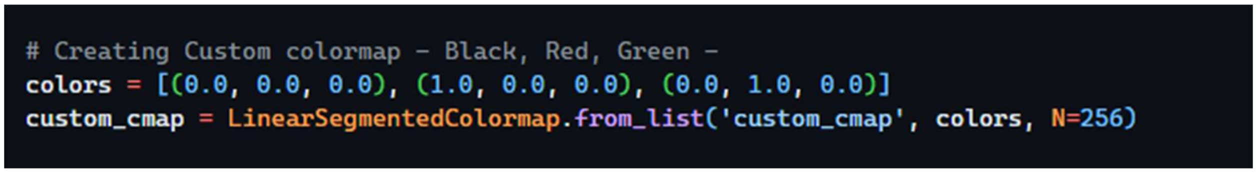
Xarray – python library for handling NetCDF data

Rioxarray – works with shapely in performing clipping and mapping operation

Os - allows manipulation of local file systems and directories

Matplotlib – Library for creating plots

Custom Color Map:



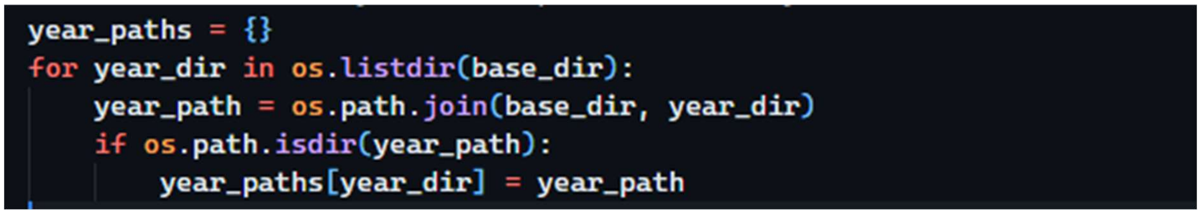
```python
# Creating Custom colormap - Black, Red, Green -
colors = [(0.0, 0.0, 0.0), (1.0, 0.0, 0.0), (0.0, 1.0, 0.0)]
custom_cmap = LinearSegmentedColormap.from_list('custom_cmap', colors, N=256)
```

This block creates a custom cmap for matplotlib tailored to specific need.

Colors variable stores RGB colors of (red, black and green) in a list

`LinearSegmentedColormap.from_list(name, colors, N)`: Creates and linearly interpolates the colormap from the list of colors, allowing smooth color transitions.

Year paths:

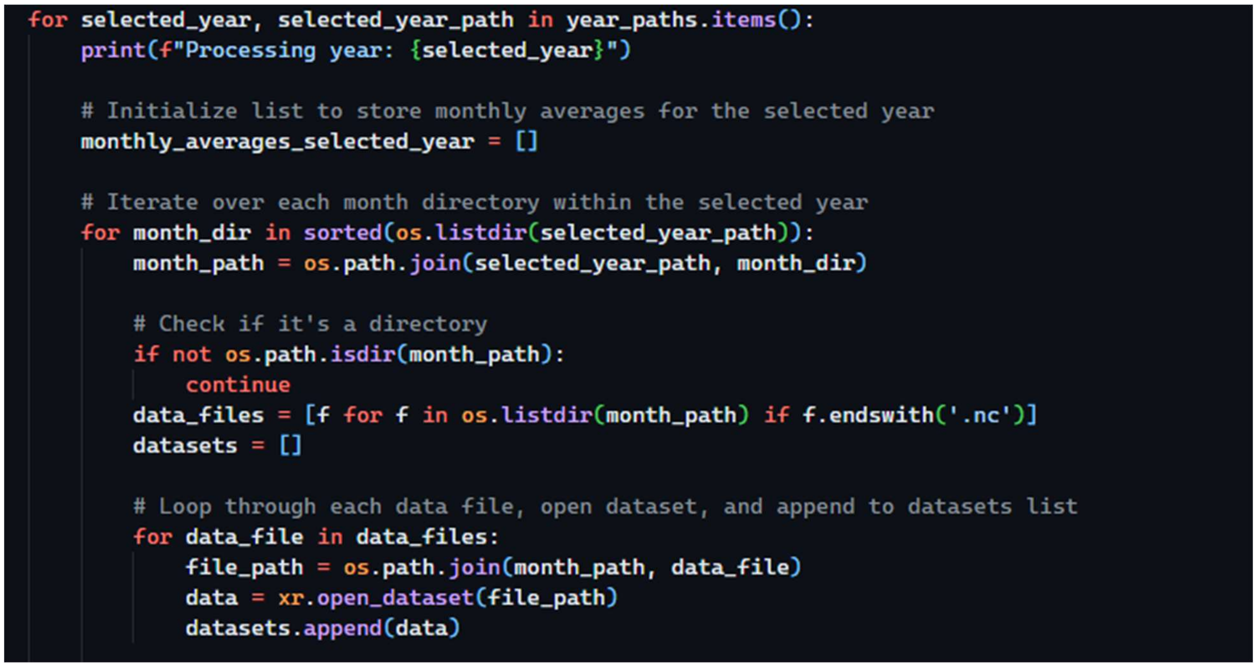


```python
year_paths = {}
for year_dir in os.listdir(base_dir):
    year_path = os.path.join(base_dir, year_dir)
    if os.path.isdir(year_path):
        year_paths[year_dir] = year_path
```

This block creates a dictionar of year paths .

`os.listdir(base_dir)`: Lists all files and directories in the specified path.
`os.path.join(base_dir, year_dir)`: Joins two path components into a single path.

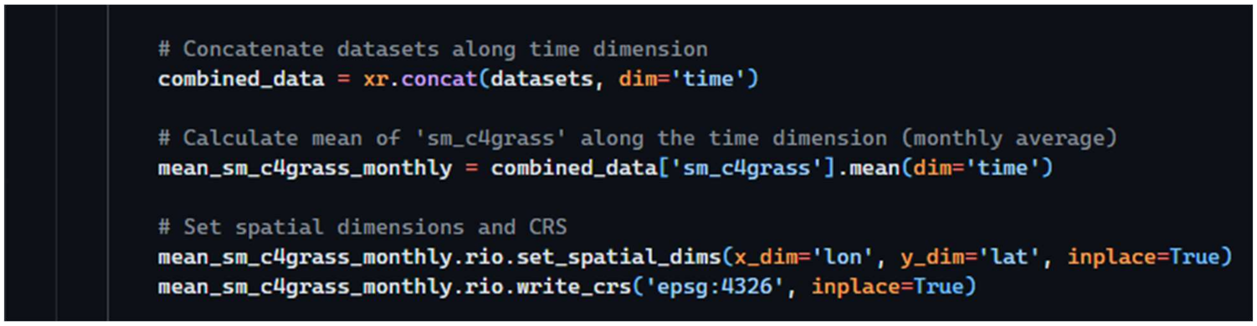
Data processing:



```python
for selected_year, selected_year_path in year_paths.items():
    print(f"Processing year: {selected_year}")

    # Initialize list to store monthly averages for the selected year
    monthly_averages_selected_year = []

    # Iterate over each month directory within the selected year
    for month_dir in sorted(os.listdir(selected_year_path)):
        month_path = os.path.join(selected_year_path, month_dir)

        # Check if it's a directory
        if not os.path.isdir(month_path):
            continue
        data_files = [f for f in os.listdir(month_path) if f.endswith('.nc')]
        datasets = []

        # Loop through each data file, open dataset, and append to datasets list
        for data_file in data_files:
            file_path = os.path.join(month_path, data_file)
            data = xr.open_dataset(file_path)
            datasets.append(data)
```

Iterates through file directories, retrieves NetCDF files, and stores them in a data list.

Calculation of Averages:



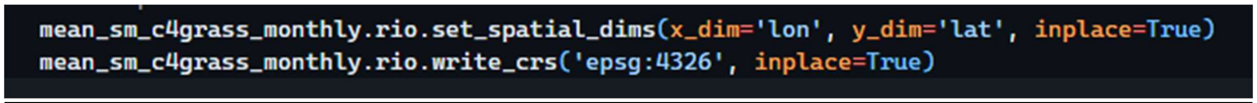
```python
        # Concatenate datasets along time dimension
        combined_data = xr.concat(datasets, dim='time')

        # Calculate mean of 'sm_c4grass' along the time dimension (monthly average)
        mean_sm_c4grass_monthly = combined_data['sm_c4grass'].mean(dim='time')

        # Set spatial dimensions and CRS
        mean_sm_c4grass_monthly.rio.set_spatial_dims(x_dim='lon', y_dim='lat', inplace=True)
        mean_sm_c4grass_monthly.rio.write_crs('epsg:4326', inplace=True)
```

xr.concat(datasets, dim='time') - concatenates the datasets along the time dimension.

Purpose: Individual NetCDF files contain partial time period data. To analyze the entire dataset, these files are combined into a continuous time series by concatenating them along the time dimension since the latitudes and longitudes are repetitive across all files

combined_data['sm_c4grass'].mean(dim='time') – calculate the monthly soil moisture average along time dimension

Setting spatial dimensions and CRS for plotting

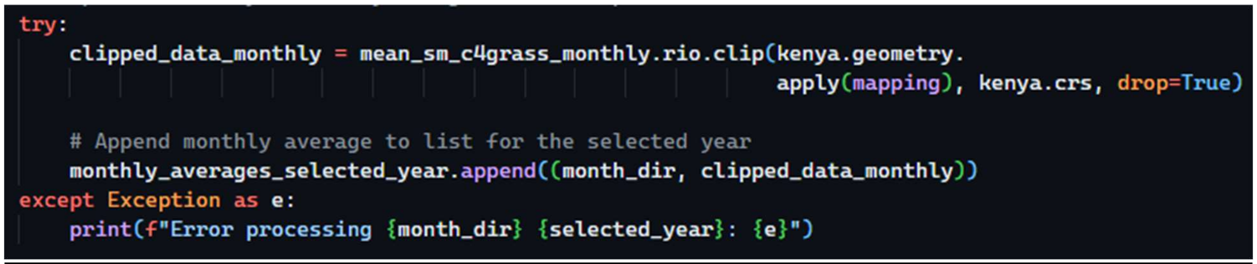


```
mean_sm_c4grass_monthly.rio.set_spatial_dims(x_dim='lon', y_dim='lat', inplace=True)
mean_sm_c4grass_monthly.rio.write_crs('epsg:4326', inplace=True)
```

Rio.set_spatial_dims(x_dim='lon', y_dim='lat',inplace=true) this line assigns dimensions to the rioxarray as specified in the datasets to correctly handle and visualise spatial data

Rio.write_crs('epsg:4326, inplace=true) this line sets the coordinate reference system for the xarray data. The CRS defines how spatial coordinates in the data are interpreted and mapped to real-world locations. The shape file uses (epsg:4326) , inplace=true updates the dataset.

Clipping the Data:



```
try:
    clipped_data_monthly = mean_sm_c4grass_monthly.rio.clip(kenya.geometry.
                                                            apply(mapping), kenya.crs, drop=True)

    # Append monthly average to list for the selected year
    monthly_averages_selected_year.append((month_dir, clipped_data_monthly))
except Exception as e:
    print(f"Error processing {month_dir} {selected_year}: {e}")
```

Attempts to clip the raster dataset for the monthly soil moisture to Kenya's geographical boundary defined in the shapefile. It includes error handling to try and catch errors encountered in the clipping process

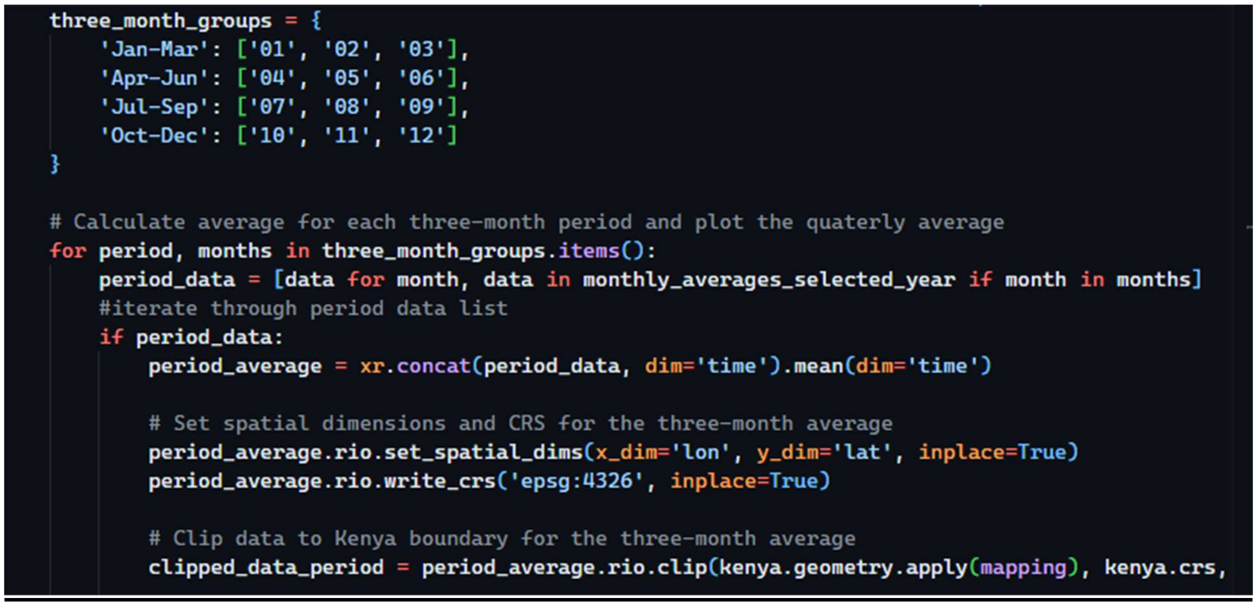
kenya.geometry.apply(mapping): Converts the geometries of the shapefile to a format compatible with rioxarray clipping.

Kenya.crs : specifies the coordinate reference system

Drop=true : drops the regions that fall outside the geographical boundary of kenya

The data is then appended to **monthly_averages_selected_year** and will be used in calculating the quarterly averages

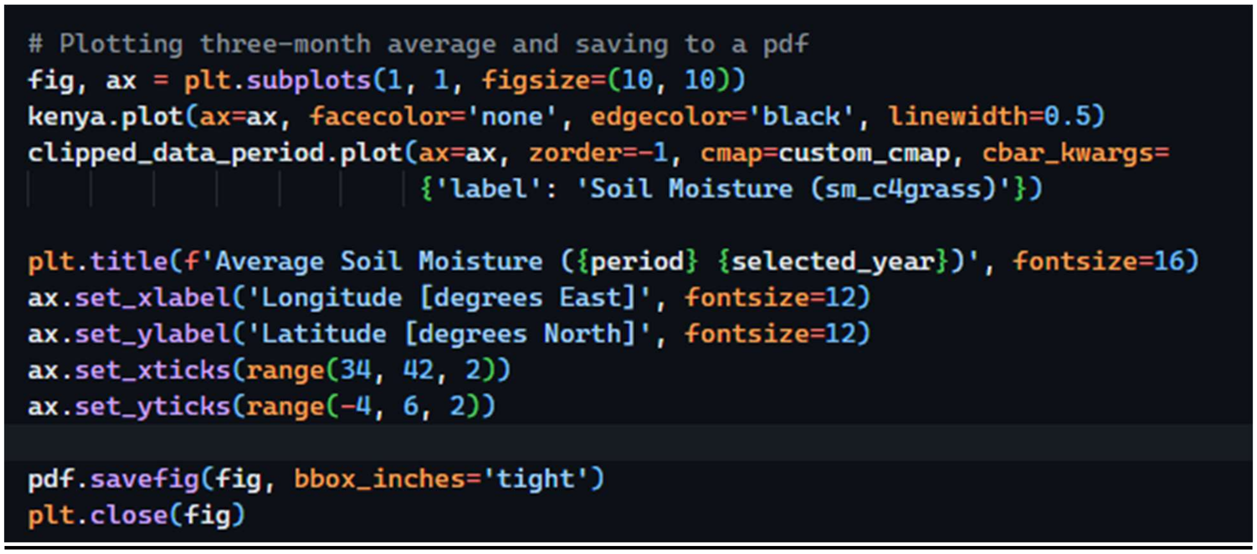
Quarterly Processing:



```
three_month_groups = {
    'Jan-Mar': ['01', '02', '03'],
    'Apr-Jun': ['04', '05', '06'],
    'Jul-Sep': ['07', '08', '09'],
    'Oct-Dec': ['10', '11', '12']
}

# Calculate average for each three-month period and plot the quaterly average
for period, months in three_month_groups.items():
    period_data = [data for month, data in monthly_averages_selected_year if month in months]
    #iterate through period data list
    if period_data:
        period_average = xr.concat(period_data, dim='time').mean(dim='time')

        # Set spatial dimensions and CRS for the three-month average
        period_average.rio.set_spatial_dims(x_dim='lon', y_dim='lat', inplace=True)
        period_average.rio.write_crs('epsg:4326', inplace=True)

        # Clip data to Kenya boundary for the three-month average
        clipped_data_period = period_average.rio.clip(kenya.geometry.apply(mapping), kenya.crs,
```

Groups data into 3 month groups/ 4 quarters, it then iterates through the **monthly_averages_selected_year** to get monthly data and then calculate a 3 month average sets spatial dimensions, CRS and stores data to the **clipped_data_period**

Quarterly plots



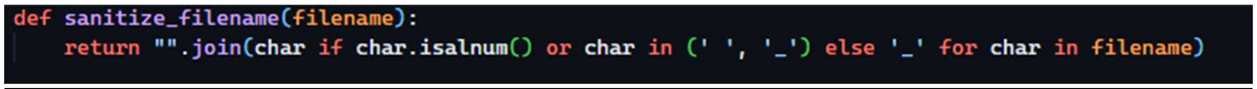
```
# Plotting three-month average and saving to a pdf
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
kenya.plot(ax=ax, facecolor='none', edgecolor='black', linewidth=0.5)
clipped_data_period.plot(ax=ax, zorder=-1, cmap=custom_cmap, cbar_kwargs=
                        {'label': 'Soil Moisture (sm_c4grass)'})

plt.title(f'Average Soil Moisture ({period} {selected_year})', fontsize=16)
ax.set_xlabel('Longitude [degrees East]', fontsize=12)
ax.set_ylabel('Latitude [degrees North]', fontsize=12)
ax.set_xticks(range(34, 42, 2))
ax.set_yticks(range(-4, 6, 2))

pdf.savefig(fig, bbox_inches='tight')
plt.close(fig)
```

This block plots the clipped data, sets title, colors, labels and saves the plot to a pdf

3. Time Series Matrix.py

Functions:

sanitize_filename :



```python
def sanitize_filename(filename):
    return "".join(char if char.isalnum() or char in (' ', '_') else '_' for char in filename)
```

Replace any characters in the filename that are not alphanumeric with underscores.

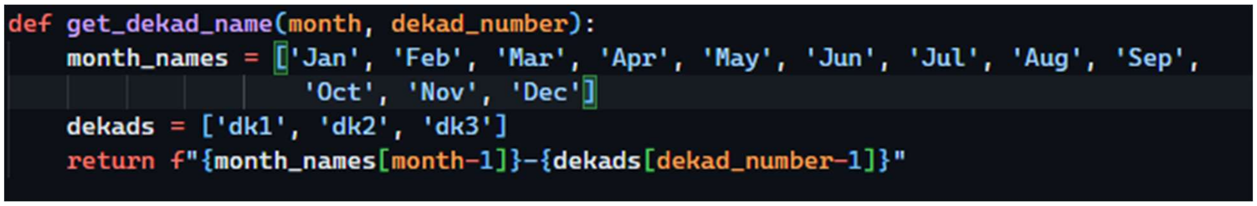
Parameters:

- filename (str): The original filename.

Returns:

- str: A sanitized version of the filename, with non-alphanumeric characters replaced by Underscores

get_dekad_name:



```python
def get_dekad_name(month, dekad_number):
    month_names = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep',
                   'Oct', 'Nov', 'Dec']
    dekads = ['dk1', 'dk2', 'dk3']
    return f"{month_names[month-1]}-{dekads[dekad_number-1]}"
```

Generate a string representing the dekad name based on month and dekad number.

Parameters:

- month (int): Month as a number (1-12).

- dekad_number (int): Dekad number (1-3)

Returns:

- str: A formatted string for the dekad name, e.g., 'Jan-dk1'

Main Function:

create_daily_soil_moisture:

This is the main Function that process soil moisture data and store it in matrix format for each ward. Clips data to the Kenya boundary and saves the results to Excel files.

Parameters:

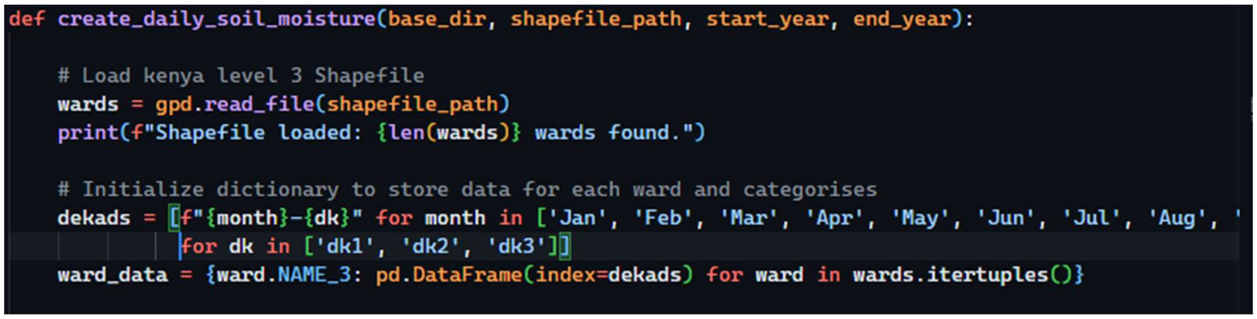
- base_dir (str): Base directory containing the yearly directories of NetCDF files.

- shapefile_path (str): Path to the shapefile containing ward boundaries.

- start_year (int): The start year for processing data.

- end_year (int): The end year for processing data.

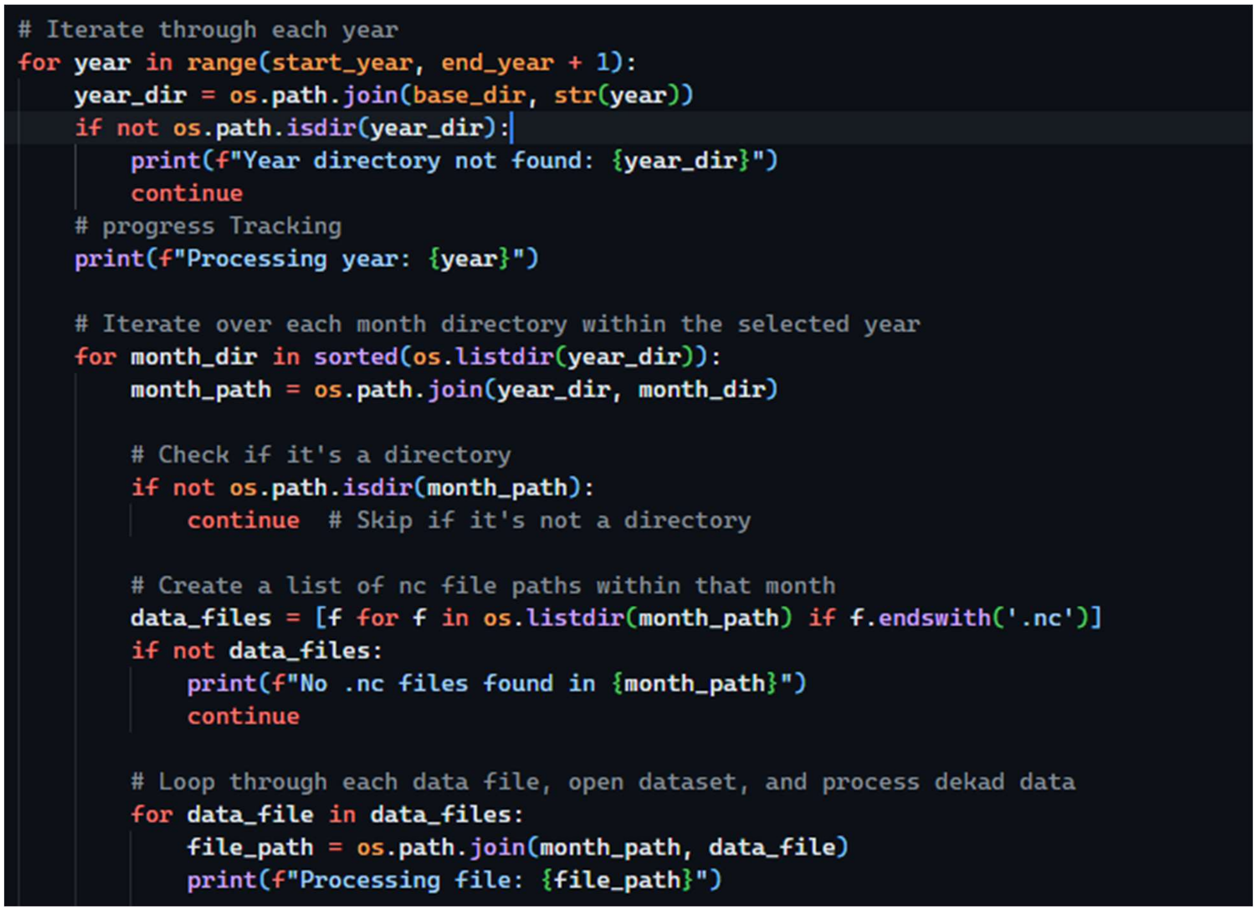
Break down of the function:



```
def create_daily_soil_moisture(base_dir, shapefile_path, start_year, end_year):

    # Load kenya level 3 Shapefile
    wards = gpd.read_file(shapefile_path)
    print(f"Shapefile loaded: {len(wards)} wards found.")

    # Initialize dictionary to store data for each ward and categorises
    dekads = [f"{month}-{dk}" for month in ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', '
              for dk in ['dk1', 'dk2', 'dk3']]
    ward_data = {ward.NAME_3: pd.DataFrame(index=dekads) for ward in wards.itertuples()}
```

This block reads the shape file using geopands and prints the number of wards found in the console for debugging and verification purpose late

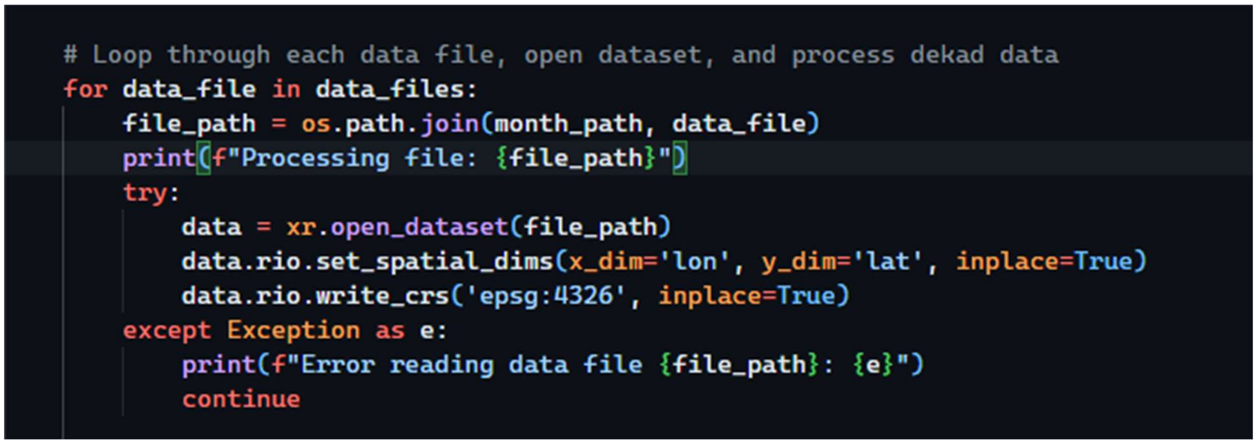
It then initializes a dictionary to hold soil moisture data for each ward, organizing the data by dekads within each month. This setup ensures that data for each ward is stored in a structured manner, allowing for efficient data writing to excel in matrix format.



```
# Iterate through each year
for year in range(start_year, end_year + 1):
    year_dir = os.path.join(base_dir, str(year))
    if not os.path.isdir(year_dir):
        print(f"Year directory not found: {year_dir}")
        continue
    # progress Tracking
    print(f"Processing year: {year}")

    # Iterate over each month directory within the selected year
    for month_dir in sorted(os.listdir(year_dir)):
        month_path = os.path.join(year_dir, month_dir)

        # Check if it's a directory
        if not os.path.isdir(month_path):
            continue  # Skip if it's not a directory

        # Create a list of nc file paths within that month
        data_files = [f for f in os.listdir(month_path) if f.endswith('.nc')]
        if not data_files:
            print(f"No .nc files found in {month_path}")
            continue

        # Loop through each data file, open dataset, and process dekad data
        for data_file in data_files:
            file_path = os.path.join(month_path, data_file)
            print(f"Processing file: {file_path}")
```

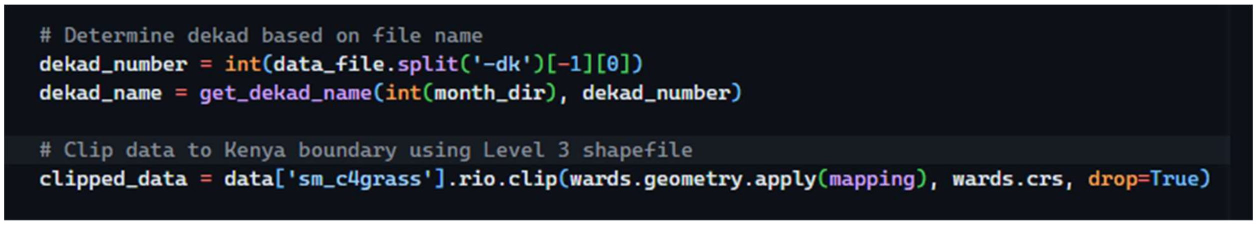
The above block is responsible for navigating through the directory structure, locating the relevant NetCDF files, and storing them in a **data_file list**.

It recursively loops through the entire Data directory fetching nc files. It ensures that only valid directories and files are considered and provides feedback about the progress of the processing tasks



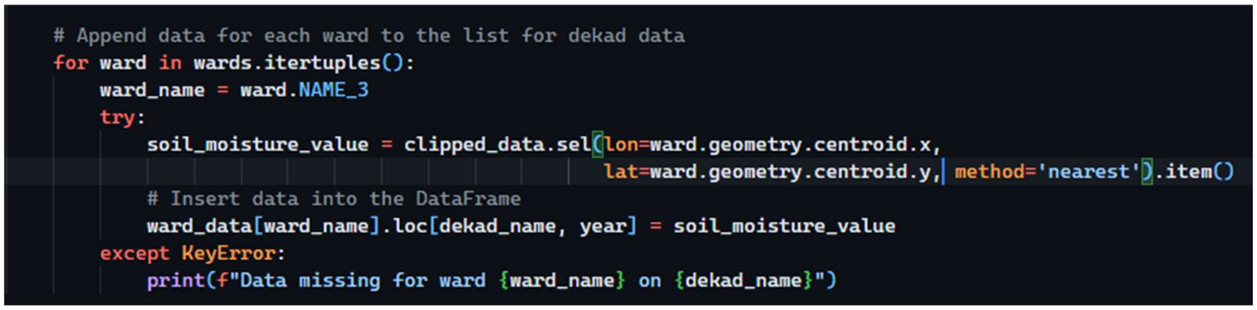
```
# Loop through each data file, open dataset, and process dekad data
for data_file in data_files:
    file_path = os.path.join(month_path, data_file)
    print(f"Processing file: {file_path}")
    try:
        data = xr.open_dataset(file_path)
        data.rio.set_spatial_dims(x_dim='lon', y_dim='lat', inplace=True)
        data.rio.write_crs('epsg:4326', inplace=True)
    except Exception as e:
        print(f"Error reading data file {file_path}: {e}")
        continue
```

The above code block iterates through the **data_file** list and then opens the dataset, sets spatial dimensions and CRS and then updates the dataset



```python
# Determine dekad based on file name
dekad_number = int(data_file.split('-dk')[-1][0])
dekad_name = get_dekad_name(int(month_dir), dekad_number)

# Clip data to Kenya boundary using Level 3 shapefile
clipped_data = data['sm_c4grass'].rio.clip(wards.geometry.apply(mapping), wards.crs, drop=True)
```

The block above determines the dekad number based on naming since they follow a similar naming convention across all years.. it then clips the data to kenya boundaries.. (clipping operation is explained in page 5)



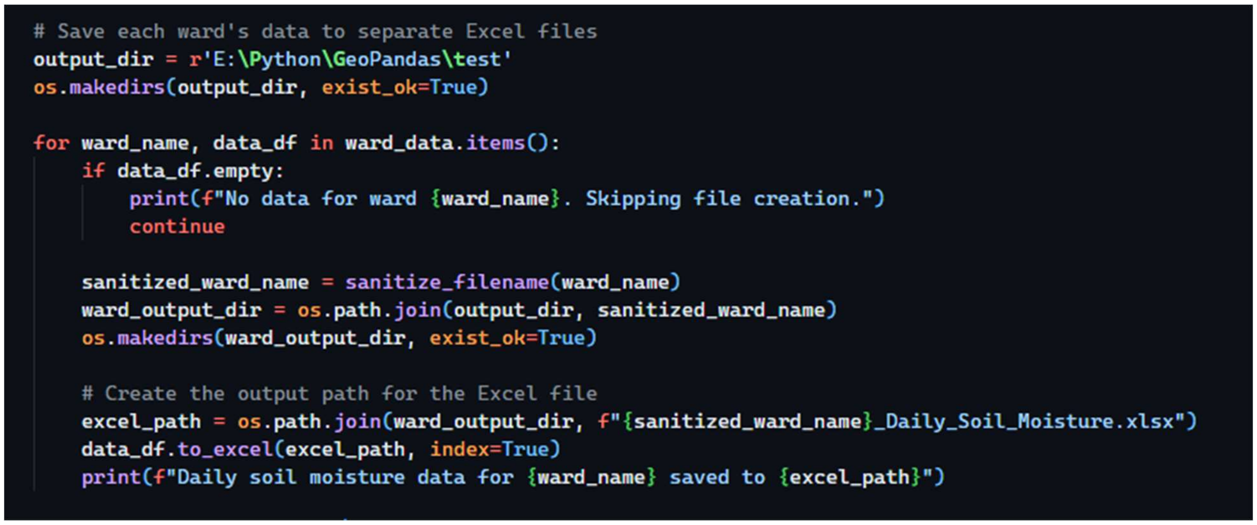
```python
# Append data for each ward to the list for dekad data
for ward in wards.itertuples():
    ward_name = ward.NAME_3
    try:
        soil_moisture_value = clipped_data.sel(lon=ward.geometry.centroid.x,
                                               lat=ward.geometry.centroid.y, method='nearest').item()
        # Insert data into the DataFrame
        ward_data[ward_name].loc[dekad_name, year] = soil_moisture_value
    except KeyError:
        print(f"Data missing for ward {ward_name} on {dekad_name}")
```

The code above is responsible for extracting soil moisture data at the centroid of each ward and storing it in a DataFrame. The try-except structure ensures that missing data or errors during the extraction process are handled gracefully, with appropriate messages printed for any issues encountered.

clipped_data.sel(lon=ward.geometry.centroid.x, lat=ward.geometry.centroid.y, method='nearest'): Selects soil moisture values from the clipped_data at the coordinates of the ward's center(centroid) (**ward.geometry.centroid.x for longitude and ward.geometry.centroid.y for latitude**). The method='nearest' ensures that the nearest available data point to the centroid is used if the exact coordinate is not present in the dataset. **.item()** extracts a single value of the selected data array

ward_data[ward_name].loc[dekad_name, year] = soil_moisture_value: Inserts the extracted soil moisture value into the ward_data DataFrame for the corresponding ward, dekad, and year

It raises an exception if any error is encountered in inserting the data to the dataframe



```python
# Save each ward's data to separate Excel files
output_dir = r'E:\Python\GeoPandas\test'
os.makedirs(output_dir, exist_ok=True)

for ward_name, data_df in ward_data.items():
    if data_df.empty:
        print(f"No data for ward {ward_name}. Skipping file creation.")
        continue

    sanitized_ward_name = sanitize_filename(ward_name)
    ward_output_dir = os.path.join(output_dir, sanitized_ward_name)
    os.makedirs(ward_output_dir, exist_ok=True)

    # Create the output path for the Excel file
    excel_path = os.path.join(ward_output_dir, f"{sanitized_ward_name}_Daily_Soil_Moisture.xlsx")
    data_df.to_excel(excel_path, index=True)
    print(f"Daily soil moisture data for {ward_name} saved to {excel_path}")
```

This block of code saves the soil moisture data for each ward into separate Excel files. It ensures that the output directory and subdirectories are created as needed, checks for and skips empty DataFrames, sanitizes ward names for use in file names, and saves each DataFrame to an Excel file in its respective directory. The process is documented with print statements that provide feedback on the progress and any issues encountered, such as missing data for a ward.

Directory Creation:

Output_dir- Specifies the directory where the excel data will be stored

os.makedirs(output_dir, exist_ok=True): Creates the output directory if it doesn't exist. The **exist_ok=True** parameter ensures that no error is raised if the directory already exists.

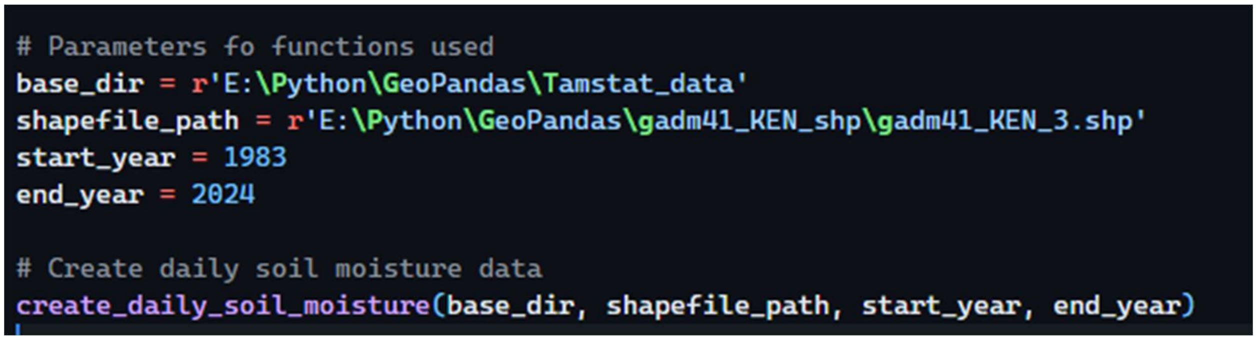
Ward data Iteration:

The for loop iterates over the ward items for every ward, it checks if the Dataframe is empty, and then prints an output and skips file creation for the empty WardsIt then call the **sanitize_filename** to replace any characters not suitable for windows file naming system

Writing the Dataframe:

excel_path = os.path.join(ward_output_dir, f"{sanitized_ward_name}_Daily_Soil_Moisture.xlsx"): Constructs the full file path for the Excel file, including the directory and file name.

data_df.to_excel(excel_path, index=True): Writes the DataFrame data_df to an Excel file at the specified path. The index=True parameter ensures that the DataFrame's index (dekads) is included in the Excel file.



```
# Parameters fo functions used
base_dir = r'E:\Python\GeoPandas\Tamstat_data'
shapefile_path = r'E:\Python\GeoPandas\gadm41_KEN_shp\gadm41_KEN_3.shp'
start_year = 1983
end_year = 2024

# Create daily soil moisture data
create_daily_soil_moisture(base_dir, shapefile_path, start_year, end_year)
```

Sets the stage for processing the soil moisture data by defining the directory paths and the time range(Parameters for the functions). The **create_daily_soil_moisture** function is then called with these parameters to perform the data processing and output generation tasks.