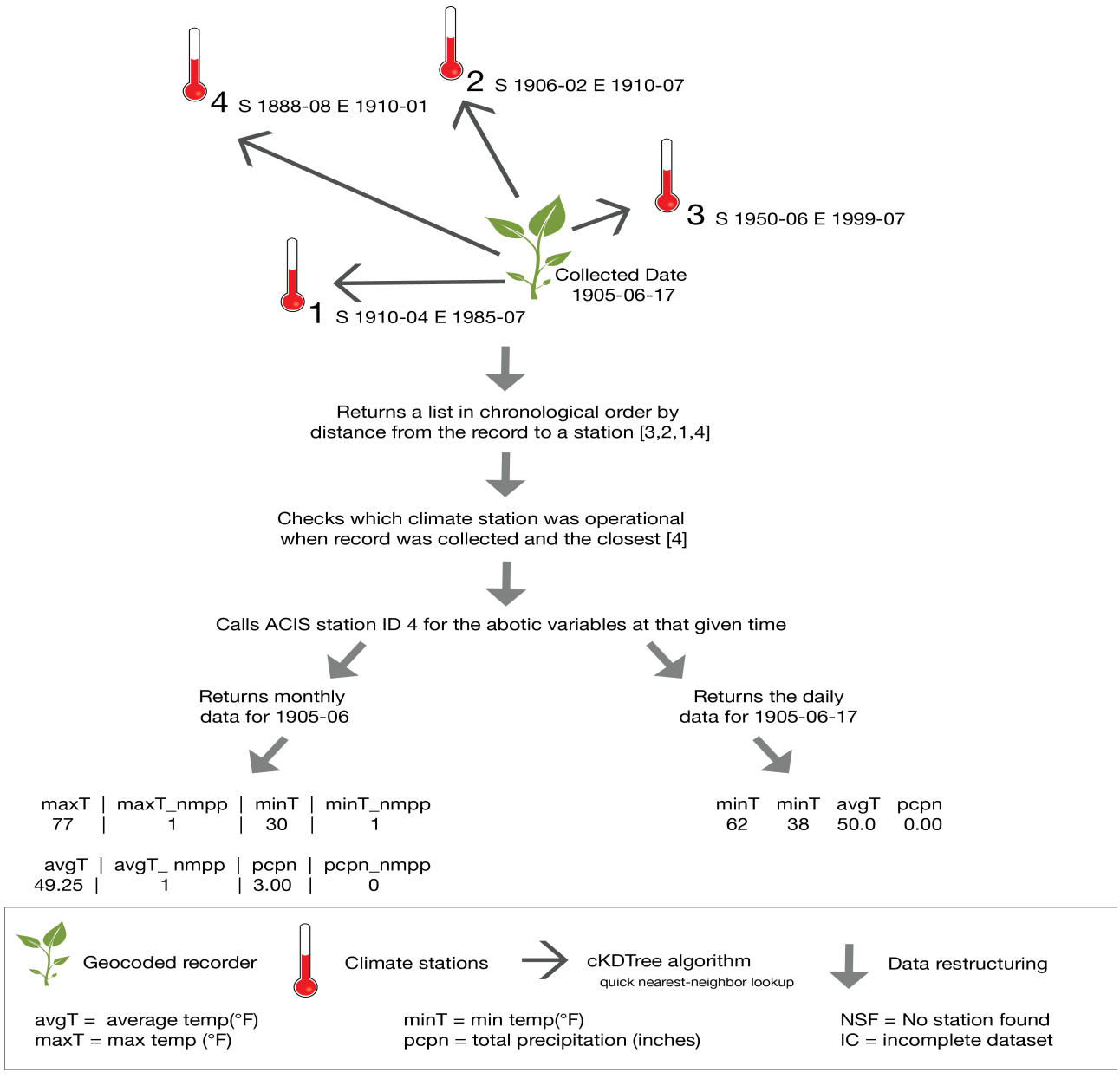
Summary:

Takes a string of specimens' locality information and geocodes the info through Google Maps' API, which returns a string of text and the lat/long coordinates. The next step is fetching all the weather stations and meta data in New England and adjacent states. Following this, all the geocoded specimen lat/longs are placed into a numpy array, and matched with a separate numpy array that is comprised of the lat/longs of the weather stations. Using a cKDTree algorithm (quick nearest-neighbor lookup), each specimen is matched with its closest weather station using euclidean distance. The returns include up to 10 of the closest weather stations, up to a 50 km radius. Iterating through each specimen, doing a dictionary lookup of whether the specimen was collected with the operation dates of the station. If true, it will make a request to ACIS data services for the climate variables. If the day of collection is known, ACIS will return that day's maximum, minimum, average, and precipitation (pcpn) variables. If only the month of collection is known, it will return the monthly max, min, avg, pncp variables as well as the number of days of missing climate observations occurred that month. See figure 1. for graphical outline



Required libraries:

- geopy
- json
- numpy
- pandas
- scipy
- urllib
- urllib2

Required column names:

- location → string of specimens' locality to be geocoded
- date → requires format YYYY-MM-DD OR columns named:
  - Year → YYYY
  - Month → MM
  - Day → DD

Note:

- Any other fields/columns will be unmodified and placed back in the edited files

Structure of files:

```
climateFetcher.py
|
|--------library folder
|                  |
|                  |--------acisRequest.py
|                  |--------dateConverter.py
|                  |--------errorCheck.py
|                  |--------googleGeocoder.py
|                  |--------__init__.py
|                  |--------cKDTree.py
|
|-------------output folder
                   |
                   |--------acis_station_ID.csv
                   |--------specimen_results.csv
```

How to run:

- python climateFetcher.py [filename]
  - ex: python climateFetcher.py exsample.csv

Detailed description of each script:

climateFetcher.py:

- **Summary:**
  - The script contains the main function that calls all the scripts within the library folder.

errorCheck.py:

- **functions:**
  - **acisDateChecker(df)** → Check the date column for the required format that is needed to send a request to ACIS Web Services. The date column needs to be in the format of YYYY-MM-DD. Uses a regular expression return true if only “0-9” and “-” characters are found. If any characters are found outside that range, the program will exit, Will see this error: “Date field contains non-digits. The require format is YYYY-MM-DD and anything outside will produce an error [location: errorcheck.py function acisDateChecker line 52”
  - **cleanUp(df)** → Drop all rows containing NULL on any row, checks if column names exist for “year,” “month,” and “day,” and checks hat all values are integers. If columns contain any values that are not integers, it will produce this error: “Date field contains non-digits. The require format is YYYY-MM-DD and anything outside will produce an error [location: errorcheck.py line 23]”
  - **checkLocation(df)** → Checks whether the column name “location” is present. If “location” is not found it will produce this error : “Absence of column name [location]” and the script will exit.
  - **errorchecking()** → Reads the first command line arguments and calls checkLocation, cleanUp, and acisDateChecker. This produces an error if a file extension is found: “File did not load correctly with **[filename.csv]**. Make use to the correct file extension and/or path [location: errorcheck.py line 42]”

googleGeocoder.py:

- **functions:**
  - **geocoder(df)** → Takes a string of location information and geocodes the info into lat/long coordinates and creates a string of locations of each specimen.

dateConverter.py

- **functions:**
  - **joinDate(df)** → If acisDateChecker() is true, this will change the column header of 'date' to 'original date'. The data from 'date' does not change, just the header name. The data is then converted to Julian Date. If any individuals units of year, month, or day do not exist, it will break the full date column into separate columns of year, month, and day. Checks again that all the values 0-9 and '-' and then labels that 'date'.
  - **yearMonthDay(string_dates)** → Takes a string of dates and breaks the data into individual units: year, month, and day.
  - **stringDateToJulianDate(string_dates)** → Takes a string of dates and converts to Julian date.
  - **changeDateformat(df)** → Takes individual units year, month, and day and concatenates into the format of YYYY-MM-DD
  - **tempColumnRename(df)** → Changes the “date” column header to “original date,” with the original data (and its incorrect formatting) unchanged. The new date, generated from changeDateformat() and in the correct format of YYYY-MM-DD, will be named 'date'.

cKDTree.py

- **functions:**
  - **nearestNeighborsSetup(df_specimens, df_stations)** → Loads the lat/long coordinates of the specimens and weather stations into numpy arrays. NearestNeighborsResults() will return the number of K (nearest stations) with the index value. Then index will be replaced by the UID to match the ASIC data serve.
  - **nearestNeighborsResults(d1, d2, r, k)** → runs a cKDTree nearest-neighbor lookup on botanical records against ACIS weather stations
  - **nearestNeighborsColumnString(k)** → Produces a string of headers of X_CWSs... columns as many as K

acisRequest.py

- **functions:**
  - **weatherStations()** → Requests all the weather stations in New England, dates of operation, and much more meta data. Converts these dates to Julian dates
  - **retrieveMlyClimateData(df_dates, start, end)** → Within each specimen, it iterates though each of the nearest neighbor weather stations in chronological order until it finds a specimen's collection date within the dates of operation of its nearest neighbor. If “IC” is returned, the date on the record was incomplete. If “NFS” is returned, there were no nearest neighbors found for that record. If “M” is returned, it means there is missing data from ASIC.
  - Returns:
    - mly_date → monthly date used for the fetching from ACIS data services
    - mly_maxt → monthly max temp (F)
    - mly_maxt_nmppm → number of missing points per month. Only for the monthly columns
    - mly_mint → monthly min temp (F)
    - mly_mint_nmppm → number of missing points per month. Only for the monthly columns
    - mly_avgt → monthly avg temp (F)
    - mly_avgt_nmppm → number of missing points per month. Only for the monthly columns
    - mly_pcpn → monthly precipitation (inches)
    - mly_pcpn_nmppm → number of missing points per month. Only for the monthly columns
    - mly_id_use → monthly UID used for the fetching from ACIS data services
  - **retrieveDlyClimateData(df_dates, start, end)** →
  - returns:
    - dly_date → daily date used for the fetching from ACIS data services
    - dly_maxt → daily max temp (F)
    - dly_mint → daily min temp (F)
    - dly_avgt → daily avg temp (F)
    - dly_pcpn → daily precipitation (inches)
    - dly_id_use → daily UID used for fetching from ACIS data services
  - **callASIC()** → Makes the request to ACIS data services requesting all the metadata of the nearest neighbors weather station.
  - **concatenateDlyAndMly()** → concatenate the daily and monthly results to the original dataset.
  - **CwsList()** → create a list of nearest neighbors weather stations to make it easier to iterate through the list.