Pandas II

Aidan Hunt, University of Washington

Learning Objectives

In this lecture, students will practice working with DataFrames, including how to:

- Performing data analysis with DataFrames
- Merging multiple data frames into a single DataFrame
- Writing DataFrames to .csv files

Check-in

- Please fill out survey
- Homework 3 feedback video
- Homework 4 due Friday, updated spec

Summary from last time

- Pandas gives us data structures for handling tabular data that are built on top of NumPy
  - `DataFrame` : like a spreadsheet
  - `Series` : like a column of the spreadsheet
  - Both `DataFrame` and `Series` objects have an "explicit" index (like dictionary keys) and "implicit" numerical index (like NumPy arrays)
  - All columns are single type
  - Like dictionary keys, index can be anything immutable
- Very easy to import data .txt, .csv, etc. data directly as a DataFrame
- Can index `DataFrame` and `Series` objects using labels or using integer index
- Can use timestamps as our index and slice by year, year and month, etc.

We used this to create import data from the USGS readings on the Skykomish River as a DataFrame

In [7]:
```
import numpy as np
import pandas as pd

def importUSGSData(fileName, metricName):
    '''
    Reads a USGS .txt file given by fileName (as a string) and returns
    the result as a dataframe. The 5th column is labeled using the
    given metricName (as a string).
    '''

    # Import data, keeping only the columns that we are interested in
    colNames = ['Timestamp', metricName]
    df = pd.read_csv(fileName, skiprows=29, delimiter='\t',
                     header=None, usecols=[2, 4], names=colNames)
```

```
    # Convert timestamps to datetime and set as index
    df.set_index('Timestamp', inplace=True)
    df.index = pd.to_datetime(df.index)

    # Return object
    return df
```

In [13]:
```
# Import flow time series
skyFlow = importUSGSData('sky_flow.txt', 'Flow')
skyFlow
```

Out[13]:

	Flow
Timestamp	
2020-01-01 01:15:00	17800
2020-01-01 01:30:00	17900
2020-01-01 01:45:00	18300
2020-01-01 02:00:00	18500
2020-01-01 02:15:00	18800
...	...
2022-12-30 23:45:00	6330
2022-12-31 00:00:00	6300
2022-12-31 00:15:00	6330
2022-12-31 00:30:00	6380
2022-12-31 00:45:00	6380

105048 rows × 1 columns

This time: work in groups to accomplish several tasks with DataFrames

Task #1: Data Analysis

Objective: For each year, identify the month with the highest average flow rate.

- Return the result as a length N vector, where N is the number of years.
- Determine the months and years to consider *programmatically* based on the data (i.e., don't hard-code the years).

Hints:

- For *programmatically* determining the months and years to consider:
  - Remember that we can easily access the components of a DateTimeIndex (e.g., .year or .day)
  - Consider using the `np.unique` function.
- For finding maximum flows, consider the `np.max` and `np.argmax` functions.

Organization:

- 2 minutes - pseudocode

- 5 minutes - talk about pseudocode
- 10 minutes: work on this in groups
- 10 minutes: talk about together

Building a pseudocode outline

First, recall that when using time-stamps as our `DataFrame` index, we can slice the `DataFrame` based on parts of the date:

In [28]:
```
skyFlow.loc['2021'] # Get all data from 2021
```

Out[28]:

	Flow
Timestamp	
2021-01-01 00:00:00	5010
2021-01-01 00:15:00	5080
2021-01-01 00:30:00	5010
2021-01-01 00:45:00	5040
2021-01-01 01:00:00	5040
...	...
2021-12-31 22:45:00	2130
2021-12-31 23:00:00	2130
2021-12-31 23:15:00	2130
2021-12-31 23:30:00	2120
2021-12-31 23:45:00	2130

35034 rows × 1 columns

In [29]:
```
skyFlow.loc['2021-01'] # Get all data from January 2021
```

Out[29]:

	Flow
Timestamp	
2021-01-01 00:00:00	5010
2021-01-01 00:15:00	5080
2021-01-01 00:30:00	5010
2021-01-01 00:45:00	5040
2021-01-01 01:00:00	5040
...	...
2021-01-31 22:45:00	1660
2021-01-31 23:00:00	1660
2021-01-31 23:15:00	1670
2021-01-31 23:30:00	1660
2021-01-31 23:45:00	1660

2976 rows × 1 columns

This gives us a way to examine each month-year combination in the `DataFrame`. We just need to generate a string that matches the expected format. Something like:

In [30]:
```
# For each year
    # For each month
        # Generate 'year-month' string
        # Slice data frame
        # Determine maximum flow for this year month
        # Store this maximum in a matrix of some kind?
```

We will also need:

- some way to know the months and years to consider based on the data.
- some way to store the month-year maximums.

In [30]:
```
# Determine year range
# Determine month range

# Initialize results matrix

# For each year
    # For each month
        # Generate 'year-month' string
        # Slice data frame
        # Determine average flow for this year month
        # Store this average in a matrix of some kind?

# Analyze the reuslts matrix to determine month of each yearly maximum
```

Building on the pseudocode outline

To get the years and months from the data, recall that we can extract these from the DateTimeIndex:

In [31]:
```
# Get year and month values (yay Datetimes!)
years = skyFlow.index.year
months =skyFlow.index.month
print(years)
print(months)
```

```
Int64Index([2020, 2020, 2020, 2020, 2020, 2020, 2020, 2020, 2020, 2020,
            ...
            2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022],
           dtype='int64', name='Timestamp', length=105048)
Int64Index([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
            ...
            12, 12, 12, 12, 12, 12, 12, 12, 12, 12],
           dtype='int64', name='Timestamp', length=105048)
```

However, we only need unique values to determine the year-month combinations. Use `np.unique!`

In [32]:
```
years = np.unique(skyFlow.index.year)
months = np.unique(skyFlow.index.month)
print(years)
print(months)
```

```
[2020 2021 2022]
[ 1  2  3  4  5  6  7  8  9 10 11 12]
```

Let's preallocate a NumPy array to put our results in:

- Rows: each year
- Columns: each month

In [36]:
```
# Preallocate
monthlyAverages = np.zeros([len(years), len(months)])
print(monthlyAverages)
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Now fill in the loop with what we know so far:

In [ ]:
```
for i, y in enumerate(years): # For each year
    for j, m in enumerate(months): # For each month
        # Generate 'year-month' string
        # Slice data frame
        # Determine average flow for this year month
        # monthlyAverages[i,j] = max we determined before
```

The last step is to create the string from the year and month that we can use to slice the `DataFrame`!

In [46]:
```
for i, y in enumerate(years): # For each year
    for j, m in enumerate(months): # For each month
        # Build date and month spec
        date = str(y) + '-' + str(m)

        # Slice the dataframe
        currSlice = skyFlow.loc[date]

        # Get the average flow
        monthlyAverages[i,j] = np.mean(currSlice.Flow)
```

In [47]:
```
print(monthlyAverages)
```

```
[[ 9012.29054054  7792.37734488  2385.01516684  4546.56944444
   8433.9635996   6560.15277778  2546.85608608   881.0124328
   1449.65486111  5300.18218487  5928.54368932  4879.22326833]
 [ 5822.89650538  3379.63169643  2447.37213997  4140.46543939
   7072.75201613  8372.64930556  2049.60819892   740.42789916
   1327.95486111  4352.45289367 11239.02912621  5010.43682796]
 [ 6859.70400269  3385.88169643  6071.98519515  3094.23958333
   6217.65120968  9312.38888889  3365.78288806   878.69522849
    470.65578326   956.87264785  3719.96705964  3613.47156727]]
```

In [48]:
```
# Get the max flow each year
np.max(monthlyAverages, axis=1)
```

Out[48]:
```
array([ 9012.29054054, 11239.02912621,  9312.38888889])
```

In [49]:
```
# Get index of the max average each year
monthInd = np.argmax(monthlyAverages, axis=1)
```

Out[49]:
```
array([ 0, 10,  5], dtype=int64)
```

In [50]:
```
# Determine the month corresponding to each index
months[monthInd]
```

Out[50]:
```
array([ 1, 11,  6], dtype=int64)
```

So:

- In 2020, max average flow was in January
- In 2021, max average flow was in November
- In 2022, max average flow was in June

Task 2: Creating a combined DataFrame

The USGS timeseries can only be accessed for one quantity of interest at a time.

- We have three data files: `skykomish_flow.txt` and `skykomish_temp.txt`.
- Since the files are of similar formats, lets define a function to do this for us!
- Then, lets parse all three

In [52]:

```python
# Import temperature time series
skyTemp = importUSGSData('sky_waterTemp.txt', 'temp')
skyTemp
```

Out[52]:

	temp
Timestamp	
2022-07-12 11:45:00	12.6
2022-07-12 12:00:00	12.7
2022-07-12 12:15:00	12.8
2022-07-12 12:30:00	12.9
2022-07-12 12:45:00	13.0
...	...
2022-12-30 23:45:00	4.5
2022-12-31 00:00:00	4.5
2022-12-31 00:15:00	4.5
2022-12-31 00:30:00	4.5
2022-12-31 00:45:00	4.5

16467 rows × 1 columns

In [53]:

```python
# Import gauge height time series
skyHeight = importUSGSData('sky_height.txt', 'height')
skyHeight
```

Out[53]:

	height
Timestamp	
2020-01-01 01:15:00	11.65
2020-01-01 01:30:00	11.69
2020-01-01 01:45:00	11.78
2020-01-01 02:00:00	11.82

2020-01-01 02:15:00	11.88
...	...
2022-12-30 23:45:00	8.40
2022-12-31 00:00:00	8.39
2022-12-31 00:15:00	8.40
2022-12-31 00:30:00	8.42
2022-12-31 00:45:00	8.42

105048 rows × 1 columns

Now, we want to merge these individual data frames into a single data frame with shared time stamps.

There are multiple ways to do this, each with their own settings for how to merge the DataFrames. Key things to keep in mind are:

- "Join Keys": What is our reference for joining the DataFrames together? Usually the reference is the index (i.e., appending columns with shared index).
- "Inner" join vs "outer" join. An inner join keeps the *intersection* between the two DataFrames with respect to the join key, while the outer join keeps the *union* with respect to the join keys.

`pd.merge(df1, df2, ...)` : merge `df1` and `df2`, with multiple options for how to combine them.

`pd.concat([df1, df2, df3...], axis=...)` : like matrix concatenation. Specify which axis to concatenate the DataFrames along.

`df1.join(df2, ...)` : Join df2 to df1, with multiple options for how to combine them.

Let's use `pd.concat` since we want to join more than one DataFrame

In [54]:
```
sky = pd.concat([skyFlow, skyHeight, skyTemp], axis=1, join='outer')
```

```
---------------------------------------------------------------------------
InvalidIndexError                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18280\3229506871.py in <module>
----> 1 sky = pd.concat([skyFlow, skyHeight, skyTemp], axis=1, join='outer')

~\anaconda3\lib\site-packages\pandas\util\_decorators.py in wrapper(*args, **kwargs)
    309                     stacklevel=stacklevel,
    310                 )
--> 311             return func(*args, **kwargs)
    312 
    313         return wrapper

~\anaconda3\lib\site-packages\pandas\core\reshape\concat.py in concat(objs, axis, join, ignore_index, keys, levels, names, verify_integrity, sort, copy)
    358     )
    359 
--> 360     return op.get_result()
    361 
    362 

~\anaconda3\lib\site-packages\pandas\core\reshape\concat.py in get_result(self)
    589                     obj_labels = obj.axes[1 - ax]
    590                     if not new_labels.equals(obj_labels):
--> 591                         indexers[ax] = obj_labels.get_indexer(new_labels)
```

```
    592
    593                     mgrs_indexers.append((obj._mgr, indexers))

~\anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_indexer(self, target, method, limit, tolerance)
   3727
   3728         if not self._index_as_unique:
-> 3729             raise InvalidIndexError(self._requires_unique_msg)
   3730
   3731         if len(target) == 0:

InvalidIndexError: Reindexing only valid with uniquely valued Index objects
```

We are running into an issue because there are duplicate time stamps in the data

- This occurs because of duplicate time stamps around daylight savings time
- So, need to remove those timestamps in our parsing function.

In [56]:

```python
# Update the function to remove duplicate time stamps
def importUSGSData(fileName, metricName):
    '''
    Reads a USGS .txt file given by fileName (as a string) and returns
    the result as a dataframe. The 5th column is labeled using the
    given metricName (as a string). If the time series contains
    duplicate timestamps, the last entry with the duplicate timestamp
    is kept.
    '''

    # Import data, keeping only the columns that we are interested in
    colNames = ['Timestamp', metricName]
    df = pd.read_csv(fileName, skiprows=29, delimiter='\t',
                     header=None, usecols=[2, 4], names=colNames)

    # Convert timestamps to datetime and set as index
    df.set_index('Timestamp', inplace=True)
    df.index = pd.to_datetime(df.index)

    # Remove the "first" instance of duplicated time stamps
    dupInd = df.index.duplicated('first')
    df = df[~dupInd]

    # Return object
    return df

skyFlow = importUSGSData('sky_flow.txt', 'Flow')
skyHeight = importUSGSData('sky_height.txt', 'Height')
skyTemp =  importUSGSData('sky_waterTemp.txt', 'Temp')
```

In [57]:

```python
# Concatenate
sky = pd.concat([skyFlow, skyHeight, skyTemp], axis=1)
sky
```

Out[57]:

	Flow	Height	Temp
Timestamp			
2020-01-01 01:15:00	17800.0	11.65	NaN
2020-01-01 01:30:00	17900.0	11.69	NaN
2020-01-01 01:45:00	18300.0	11.78	NaN
2020-01-01 02:00:00	18500.0	11.82	NaN
2020-01-01 02:15:00	18800.0	11.88	NaN

...	...	...	...
2022-12-30 23:45:00	6330.0	8.40	4.5
2022-12-31 00:00:00	6300.0	8.39	4.5
2022-12-31 00:15:00	6330.0	8.40	4.5
2022-12-31 00:30:00	6380.0	8.42	4.5
2022-12-31 00:45:00	6380.0	8.42	4.5

105056 rows × 3 columns

Note that missing values were automatically populated with `NaN`:

- water temp gauge didn't come online until mid-2022.
- Pandas is good at handling missing data and has multiple options for this.

Writing the full result to a file

Now that we have the full time series, lets write it to a .txt file using the `.to_csv` method of the `DataFrame`.

In [58]:
```
sky.to_csv('sky_total.txt')
```