**Pandas Dataframes**

**What they are** – Pandas dataframes are Python data structures that work with data in a tabular format. Unlike lists and numpy arrays, Pandas dataframes have header names above each column. This not only helps with indexing, but also improves preserves the relationships between cells in a row or column. The first column is always an index column, labeling the rows in a similar manner as the columns. While columns must be named, row indices default to the integer sequence 0,1,2,…n, n being the number of elements in each column. However, the index can be set to whatever is desired by the user, regardless of datatype. This dual row-column index format allows for users to easily work with entire rows and columns of the dataframe. Below is an example of the dataframe structure, the streamflow dataframe consistently used in our homework; the first 5 values are obtained using the .head() method.

Table

Description automatically generated

Here, the datetime column is set as the index, with each row index therefore in datetime format (more on this later). All the columns are named above the data they represent, with the column names themselves being in string format. Note that the index column name is one cell lower than the data column names. Within the dataframe, each element has both a row and a column index. For example, the first flow value (207.0) would have a row index of 1989-01-01 and a column index of flow.

There are additional important differences between dataframes and lists/arrays. While lists can have any combination of datatypes in any order, and arrays must have all elements of the same datatype, dataframes can have datatypes vary between columns, but all elements in a column must have the same datatype.

**How to make one** – To make a dataframe from scratch, use the pd.dataframe() function, first defining the column names with columns = [“column names”], and then inputting the data as data = [“data”]. Both the columns and the data must be arguments inside the function, and individual column names and data points must be comma-separated. However, this method is tedious even with small datasets, and becomes completely impractical for larger datasets. Instead, the data should be read in directly from its source, be it a local directory or an online repository. First, the location of the data must be defined; a directory path and filename for a local source, or a url for an online source. Next, use pd.read\_table() to read the data into a dataframe, providing the data source, separation, column names, and index names as arguments. The 2 figures below show an example of this operation, again using homework flow data, one for local files and the other for online urls.

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Note that in this example, no index column is defined (it is instead defined further in the code). Additionally, Pandas is instructed to skip the top 30 rows, as these first rows do not contain data. For some urls, data can also be read in as a specific format. If the data was supposed to be read in as a csv, then, at the end of the url, &format=csv would be added before closing the quoted string.

**Slicing Pandas Data** – There are multiple means of indexing Pandas dataframes. The base numpy method of stating the row and column number still works, but the introduction of named rows and columns means simpler and more efficient methods are available. These methods are label-based, unlike numpy’s location-based method. For location-based indexing and slicing, the methods are similar to those used in numpy arrays, but the iloc attribute must be called first. To get the flow column from the first figure, a location-based slice would read as flow\_data.iloc[:,2] or flow\_data.iloc[:,2:3]. Label indexing uses the .loc sttribute, and works best for slicing index-based rows. For the same example dataframe, typing flow\_data.loc[“1989-01-01”] would return all the values in the first row of data. The quotes are necessary as the data, despite being read in as a datetime, still has the base format of a string. Although columns can be indexed and sliced by labels, it is often much simpler to, when grabbing an entire column, to simply use dataframe[[‘column’]]. Grabbing the same flow data the iloc attribute did could be done using flow\_data[[“flow”]]. The double brackets specify a dataframe output; using only single brackets would return a Pandas series, which is much like a dataframe but does not support multiple non-index columns.

**More on dataframe indices** – The index column, despite being part of the dataframe, is distinct from each of the data columns. The index column can be extracted as its own object from within the dataframe or series object and can be thought of as an array or an ordered set. Functionally, the index column is very similar to a numpy array, as it uses the same indexing/slicing methods, and has many of the same attributes, such as .size, .shape, .ndim, and .dtype. However, unlike arrays, indices cannot be modified via normal means, to prevent the negative effects of inadvertent index modification. Instead, any changes to the index must be done using setindex(), using either one of the data columns or an array of equal size to any given dataframe column as arguements. If a data column is set as the index, it will no longer be included with the data when the operation is complete. In the first figure, the index was originally its own column upon reading in the data, but was changed to be the index, replacing the generic [0,1,2,…,n] series the index defaults to. Any slicing/indexing by index must be done with either .loc or .iloc, as the simplified means for working with data columns do not work on rows defined by index values.

**Dataframe methods** – Pandas dataframes have a variety of methods associated with them. Similarly to numpy arrays, dataframes have .mean(), .median(), .max(), and .min(), all very useful for finding the key statistics of a dataframe. These methods can be instructed to operate by column by putting the column name in brackets prior to defining the method. Additionally, the groupby() method is very powerful, as it rearranges data according to the values within the defined column. This can be combined with any of the basic statistical methods mentioned beforehand to create a Pandas series of the desired statistic over each possible value in another column, provided this other column is composed of integers. In the homework, such a method was used to generate both plots and timeseries of the climatological mean and median flow according to the day of the month of any given month with the lines of code below.

Oct\_data = flow\_data[flow\_data[‘month’] == 10]

Oct\_mean = Oct\_data.groupby(‘day’)[‘flow’].mean

Oct\_median = Oct\_data.groupby(‘day’)[‘flow’].median

Other dataframe methods also proved useful, particularly when broadening to additional, remotely-accessed satasets. The method .cumsum(), which finds the cumulative sum of elements over a dataframe or series axis, was used to compute climatological precipitation accumulation at 3 different locations for the month of October. Later, the method .bool(), which returns either 0 or 1 for each element of a series or dataframe depending on whether a condition is true or false, was used to define whether or not a day had precipitation, a step in finding the climatological likelihood of precipitation occurring for any given day in the month of October. Countless other dataframe methods exist, some unique to dataframe objects while others are shared with simpler data structures such as numpy arrays.

**Dataframe attributes** – Compared to the number of available bethods, dataframe elements are rather limited in scope, but as the .loc and .iloc attributes show, can be very useful in working with dataframes. Dataframes also have the same basic attributes as numpy arrays and index columns (.size, .shape, .ndim, .dtypes), with almost identical outputs. Both columns and indeces of a dataframe are represented by attributes, .columns for columns and .index for indices. Lastly, dataframes can be converted to a numpy representation using .values.