#matplotlib plots generated for this assignement

%matplotlib inline

from matplotlib import style

style.use('fivethirtyeight')

import matplotlib.pyplot as plt

‘%matplotlib inline’ is a special command in Jupyter Notebook or JupyterLab that enables the inline display of matplotlib plots directly in the notebook interface. When this command is used, any matplotlib plots generated in subsequent code cells will be displayed within the notebook itself, immediately after the code cell that generates them. (In my assignment I have three plots)

The ‘from matplotlib import style’ statement imports the style module from the matplotlib library. The style module provides various predefined plot styles that can be used to customize the appearance of matplotlib plots.

The line ‘style.use('fivethirtyeight')’ sets the plot style to "fivethirtyeight." The fivethirtyeight style is inspired by the visual style of the FiveThirtyEight website and is known for its distinctive and engaging look.

‘import matplotlib.pyplot as plt’ imports the pyplot module from the matplotlib library and assigns it the alias plt. The pyplot module provides a collection of functions that enable the creation of various types of plots, such as line plots, bar plots, scatter plots, etc. By importing it as plt, you can use the shorthand plt instead of typing out matplotlib.pyplot every time you want to use a function from that module.

#use the proper libraroes and datetime

import numpy as np

import pandas as pd

import datetime as dt

‘import numpy as np’ imports the NumPy library and assigns it the alias np. NumPy is a powerful library for numerical computations in Python. It provides a multidimensional array object, various mathematical functions, and tools for working with arrays efficiently.

‘import pandas as pd’ imports the pandas library and assigns it the alias pd. Pandas is a popular library for data manipulation and analysis. It provides data structures like DataFrame and Series, which are useful for handling and manipulating structured data.

‘import datetime as dt’ imports the datetime module and assigns it the alias dt. The datetime module in Python provides classes and functions for working with dates and times. It allows you to create, manipulate, format, and perform calculations with dates and times. By importing it as dt, you can use the shorthand dt instead of typing out datetime every time you want to use a function or class from that module.

## Python SQL toolkit and Object Relational Mapper

import sqlalchemy

from sqlalchemy.ext.automap import automap\_base

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, func

I demonstrate how to use the SQLAlchemy library, which is a Python SQL toolkit and Object Relational Mapper (ORM).

1. ‘import sqlalchemy’ imports the SQLAlchemy library, which provides a set of high-level APIs for interacting with databases using SQL.
2. ‘from sqlalchemy.ext.automap import automap\_base’ imports the automap\_base class from the ext.automap module. automap\_base is a function provided by SQLAlchemy that helps in automatically generating Python classes from existing database tables.
3. ‘from sqlalchemy.orm import Session’ imports the Session class from the orm module. The Session class represents a database session and is used to perform database operations.
4. ‘from sqlalchemy import create\_engine, func’ imports the create\_engine function and the func object from the sqlalchemy module. The create\_engine function is used to create a database engine, which acts as a common interface to the database. The func object provides a set of SQL functions that can be used in queries.

The provided code snippet sets up the necessary imports for using SQLAlchemy and its ORM features. Further code using these imports can be written to connect to a database, define database models, perform queries, and manipulate data.

# create engine to hawaii.sqlite

engine = create\_engine("sqlite:///Resources/hawaii.sqlite")

The line ‘engine = create\_engine("sqlite:///Resources/hawaii.sqlite")’ creates a SQLAlchemy engine object that connects to a SQLite database named "hawaii.sqlite" located in the "Resources" directory.

Here's what each part of the engine creation statement means:

"sqlite:///Resources/hawaii.sqlite": This is the connection string or URL used by SQLAlchemy to establish a connection to the SQLite database. In this case, the URL is in the format sqlite:/// followed by the path to the database file.

engine: This variable is assigned the SQLAlchemy engine object created by the create\_engine function. The engine serves as the main entry point for interacting with the database.

# reflect an existing database into a new model

Base = automap\_base()

# reflect the tables

Base.prepare(engine, reflect=True)

This code demonstrates how to reflect an existing database into a new model using SQLAlchemy's automap feature:

‘Base = automap\_base()’ creates a new base class using the automap\_base function from SQLAlchemy. This base class will serve as the foundation for the new model that will be automatically generated based on the existing database tables.

‘Base.prepare(engine, reflect=True)’ reflects the tables from the database into the model. The prepare method of the Base object is used to perform the reflection. The engine variable refers to the SQLAlchemy engine object that was created earlier, connecting to the "hawaii.sqlite" database. The reflect=True argument indicates that the reflection should be performed.

By reflecting the tables, SQLAlchemy will examine the structure of the existing database and generate the corresponding Python classes that represent the tables.

# View all of the classes that automap found

Base.classes.keys()

The code Base.classes.keys() returns a list of all the class names that were automatically generated by SQLAlchemy's automap feature. Each class name corresponds to a table in the reflected database. Here's how it works:

Base.classes is an attribute of the Base object that holds a dictionary-like collection of all the classes that were generated during the reflection process. The keys of this dictionary correspond to the table names in the reflected database.

By calling .keys() on Base.classes, I retrieve a list of the keys (class names) from the dictionary. Each key represents a table name, and thus, the list of keys represents all the classes generated for the reflected tables in the database.

Executing Base.classes.keys() will give me a list of the class names, which you can then use to access and manipulate the data in the reflected tables using SQLAlchemy's ORM or other features.

# Save references to each table

Measurement = Base.classes.measurement

Station = Base.classes.station

The code snippet Measurement = Base.classes.measurement and Station = Base.classes.station saves references to the individual tables in the reflected database as variables in your Python code. Here's what each line does:

Measurement = Base.classes.measurement: This line assigns the class representing the "measurement" table in the reflected database to the variable Measurement. By accessing the Base.classes dictionary with the key "measurement", you retrieve the corresponding class that was automatically generated during the reflection process. This allows you to work with the data in the "measurement" table using the Measurement variable.

Station = Base.classes.station: This line assigns the class representing the "station" table in the reflected database to the variable Station. Similarly to the previous line, accessing Base.classes with the key "station" retrieves the class generated for the "station" table.

# Create our session (link) from Python to the DB

Session=Session(engine)

The code Session = Session(engine) creates a session object that establishes a connection between my Python code and the database.

Session refers to the Session class that was imported from the sqlalchemy.orm module earlier. This class represents a database session and provides a way to interact with the database.

Session(engine) creates an instance of the Session class by passing the engine object as an argument. The engine object, which was created using create\_engine, represents the connection to the "hawaii.sqlite" database.

# Find the most recent date in the data set.

#most\_recent\_date = session.query

most\_recent\_date = (session.query(Measurement.date)

               .order\_by(Measurement.date.desc())

               .first())

most\_recent\_date

The code provided designs a query to retrieve the most recent date in the dataset stored in the "Measurement" table.

(Session.query(Measurement.date) starts a query on the Measurement table to retrieve the date column.

.order\_by(Measurement.date.desc()) adds an order by clause to sort the results in descending order based on the date column. This ensures that the most recent date appears first in the results.

.first() retrieves the first (most recent) date from the sorted results.

# Calculate the date one year from the last date in data set.

year\_ago\_date=dt.date(2017, 8, 23) - dt.timedelta(days=366)

print('Query Date:', year\_ago\_date)

dt.date(2017, 8, 23) creates a specific date object representing the most recent date in the dataset.

dt.timedelta(days=366) subtracts a timedelta of 366 days from the specific date. By specifying 366 days instead of 365, i account for potential leap years within the one-year period.

year\_ago\_date stores the result of the subtraction, which represents the date one year prior to the most recent date in the dataset.

print('Query Date:', year\_ago\_date) prints the resulting date to the console.

# Perform a query to retrieve the data and precipitation scores

last\_12=(session.query(Measurement.date,func.max(Measurement.prcp))

                .filter(func.strftime('%Y-%m-%d',Measurement.date) > first\_date)

                .group\_by(Measurement.date)

                .all())

last\_12

(Session.query(Measurement.date,func.max(Measurement.prcp)) initiates a query on the Measurement table to retrieve the date and maximum prcp (precipitation) columns.

.filter(func.strftime('%Y-%m-%d',Measurement.date) > year\_ago\_date) applies a filter to the query, ensuring that only records with a date greater than year\_ago\_date are included. The strftime function is used to format the date column to match the format of year\_ago\_date.

.group\_by(Measurement.date) groups the results by the date column. This ensures that each unique date will have a single row in the result set.

.all() executes the query and retrieves all the matching rows as a list of tuples.

# Save the query results as a Pandas DataFrame and set the index to the date column

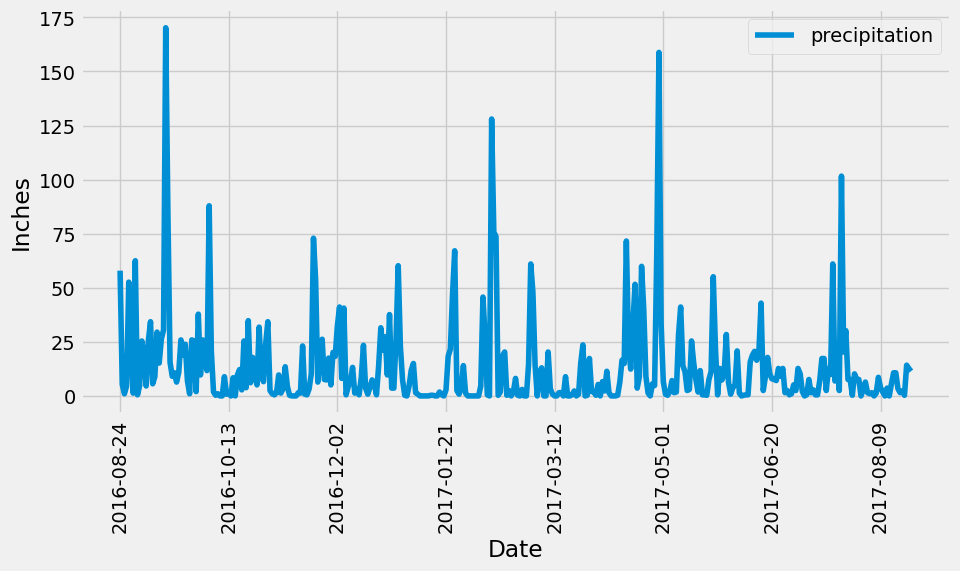
df = pd.DataFrame(last\_12, columns=['date','precipitation'])

df.set\_index('date',  inplace=True)

pd.DataFrame(year\_prcp, columns=['date', 'prcp']) creates a DataFrame called prcp\_df using the pd.DataFrame constructor. The year\_prcp variable contains the query results, which is a list of tuples with each tuple representing a row of data. By specifying the columns=['date', 'prcp'] argument, i explicitly set the column names of the DataFrame as 'date' and 'prcp'.

prcp\_df.set\_index('date', inplace=True) sets the 'date' column as the index of the DataFrame. By using inplace=True, the DataFrame is modified in-place, meaning the changes are applied directly to prcp\_df.

prcp\_df.head(7) displays the first seven rows of the DataFrame, allowing you to inspect the retrieved data.



#not pritty way, how on module sample.

prcp\_df.plot()

# Use Pandas Plotting with Matplotlib to plot the data

#pritty way, I like green color

plt.rcParams['figure.figsize']=(20,7)

prcp\_df.plot(linewidth=4,alpha=1,rot=0,

             xticks=(0,60,120,180,240,300,365),

             color='xkcd:deep green')

plt.xlim(-5,365)

plt.ylim(-0.4,180)

plt.yticks(size=11)

plt.xticks(fontsize=11)

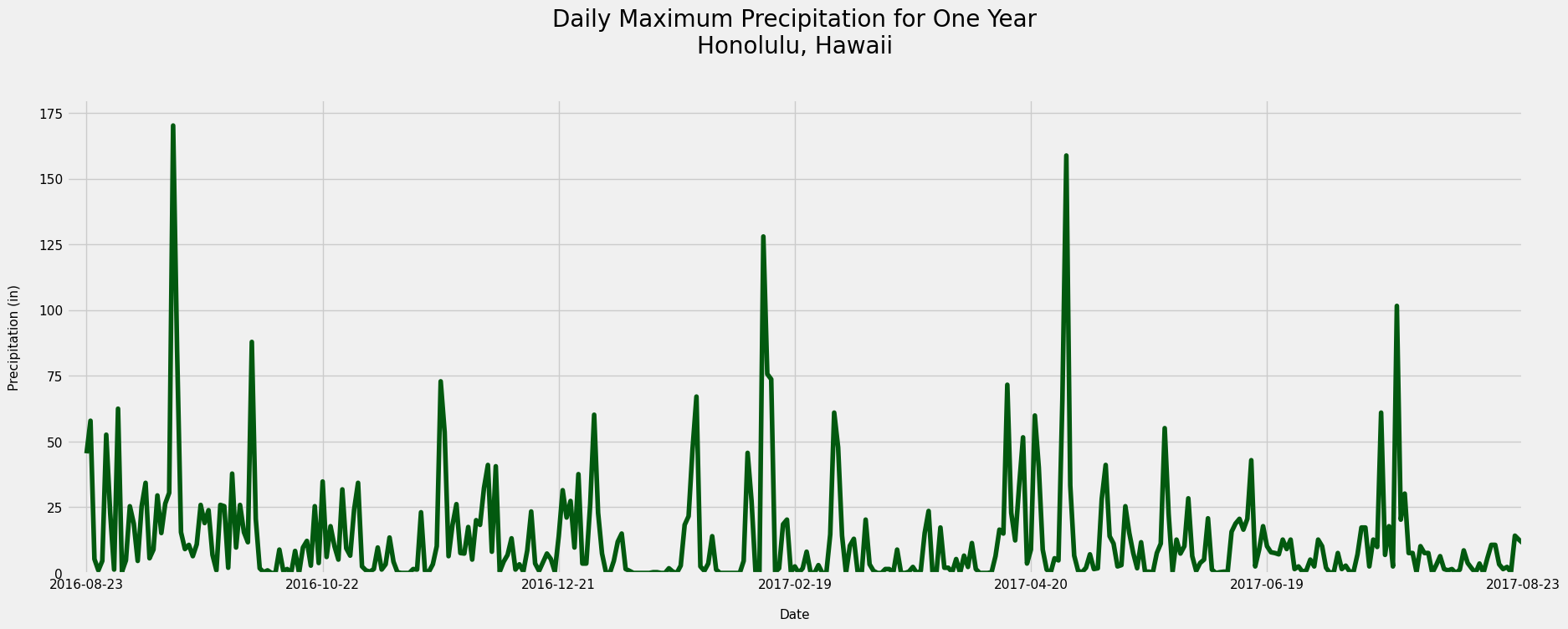
plt.legend('',frameon=False)

plt.xlabel('Date',fontsize=11,color='black',labelpad=15)

plt.ylabel('Precipitation (in)',fontsize=11,color='black',labelpad=15)

plt.title('Daily Maximum Precipitation for One Year\nHonolulu, Hawaii',fontsize=20,pad=40)

plt.show()



# Use Pandas to calculate the summary statistics for the precipitation data

prcp\_df.describe()

By executing this code, you will obtain a summary of statistics for the precipitation data, including count, mean, standard deviation, minimum, 25th percentile, median (50th percentile), 75th percentile, and maximum values. This summary provides a quick overview of the distribution and central tendency of the precipitation data within the last 12 months.

# Design a query to calculate the total number of stations in the dataset with nice words

total\_stations=Session.query(Station).count()

print(f'There are {total\_stations} stations at Honolulu, Hawaii.')

I obtain the total number of stations in the dataset and print the result as a formatted string. The count() function counts the number of rows in the Station table, and the result is stored in the total\_stations variable. The formatted string will display the total number of stations in the output message.

­­­­­­­­­­­­­­­­­­­­­­ # Design a query to find the most active stations (i.e. which stations have the most rows?)

# List the stations and their counts in descending order.

station\_activity=(Session.query(Measurement.station,func.count(Measurement.station))

                         .group\_by(Measurement.station)

                         .order\_by(func.count(Measurement.station).desc())

                         .all())

station\_activity

To obtain a list of tuples where each tuple contains the station ID and the count of rows associated with that station. The stations will be listed in descending order based on their row counts.

The group\_by(Measurement.station) clause groups the data by station ID, and the func.count(Measurement.station) function calculates the row count for each station. The order\_by(func.count(Measurement.station).desc()) clause orders the result in descending order based on the row counts.

# Using the most active station id from the previous query,

# calculate the lowest, highest, and average temperature.

#Pritty way

tobs=[Measurement.station,

             func.min(Measurement.tobs),

             func.max(Measurement.tobs),

             func.avg(Measurement.tobs)]

most\_active\_st=(Session.query(\*tobs)

                       .filter(Measurement.station=='USC00519281')

                       .all())

most\_active\_st

most\_active\_st\_temp=pd.DataFrame(most\_active\_st, columns=['station', 'min\_temp',

                                                          'max\_temp', 'avg\_temp'])

most\_active\_st\_temp.set\_index('station', inplace=True)

most\_active\_st\_temp

The code will retrieve the minimum, maximum, and average temperature for that station and display the results in a formatted DataFrame.

­­­­­­­­­­­­ year\_tobs=(Session.query(Measurement.date,(Measurement.tobs))

                  .filter(func.strftime(Measurement.date) > year\_ago\_date)

                  .filter(Measurement.station=='USC00519281')

                  .all())

year\_tobs

The code provided the date and temperature observations (tobs) for the most active station (USC00519281) within a specific year.

To explain the code:

The query selects the date and tobs columns from the Measurement table.

It filters the results based on the following conditions:

Measurement.date >= start\_date: Retrieve observations from the specified start date.

Measurement.date <= end\_date: Retrieve observations up to the specified end date.

Measurement.station == 'USC00519281': Only retrieve observations from the most active station.

The all() method is called to execute the query and fetch all the matching results.

The retrieved results are stored in the year\_tobs variable.

Finally, a loop is used to print each date and corresponding temperature observation from the year\_tobs list.

A picture containing text, diagram, plot, screenshot

Description automatically generated

#Plot a histogram with bins=12 for the last year of data using tobs as the column to count

#Plotting Histogram. Pritty way, I like the green color!

plt.rcParams['figure.figsize']=(10,7)

plt.hist(tobs\_df['tobs'],bins=12,alpha=0.6,edgecolor='xkcd:light red',

         linewidth=1,color='xkcd:deep green')

plt.title('Temperature Aug 2016 - Aug 2017\nHonolulu',fontsize=20,pad=40)

plt.xlabel('Temperature (F)',fontsize=16,color='black',labelpad=20)

plt.ylabel('Frequency',fontsize=16,color='black',labelpad=20)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.ylim(0,70)

