Pandas Performance

In this notebook we will be exploring the performance differences between different approaches of iterating through a Pandas column.

First we will start by loading our data. The data is from Lyft's Go Bike program and inclues every trip from 2017:
https://www.lyft.com/bikes/bay-wheels/system-data

In [1]:
```
import pandas as pd
import time

df = pd.read_csv('2017-fordgobike-tripdata.csv',
                 dtype={"start_station_latitude":float, "start_station_longitude":float,
                        "end_station_latitude":float, "end_station_longitude":float})
```

In [2]:
```
benchmarking_results = []
```

Next we define a function to calculate distance based on two GPS locations

In [3]:
```
import numpy as np

# Define a basic Haversine distance formula
def haversine(lat1, lon1, lat2, lon2):
    MILES = 3959
    lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    total_miles = MILES * c
    return total_miles
```

First, let's try iterating through the dataframe using `iterrows()`

In [4]:
```
wall_time_1 = time.time()
CPU_time_1 = time.process_time()
haversine_series = []
for index, row in df.iterrows():
    haversine_series.append(haversine(row['start_station_latitude'], row['start_station_longitude'],
                                      row['end_station_latitude'], row['end_station_longitude']))
df['distance'] = haversine_series
end_wall_time_1 = time.time()
end_CPU_time_1 = time.process_time()
execution_wall_time_1 = end_wall_time_1 - wall_time_1
execution_CPU_time_1 = end_CPU_time_1 - CPU_time_1
benchmarking_results.append(['Using the iterrows() method', execution_wall_time_1, execution_CPU_time_1])
```

The next approach is to loop through the dataframe using `iloc`

In [5]:
```
wall_time_2 = time.time()
CPU_time_2 = time.process_time()
def haversine_looping(df):
    distance_list = []
    for i in range(0, len(df)):
        d = haversine(df['start_station_latitude'].iloc[i], df['start_station_longitude'].iloc[i],
                      df['end_station_latitude'].iloc[i], df['end_station_longitude'].iloc[i])
        distance_list.append(d)
    return distance_list
df['distance'] = haversine_looping(df)
end_wall_time_2 = time.time()
end_CPU_time_2 = time.process_time()
execution_wall_time_2 = end_wall_time_2 - wall_time_2
execution_CPU_time_2 = end_CPU_time_2 - CPU_time_2
benchmarking_results.append(['Using the iloc() method', execution_wall_time_2, execution_CPU_time_2])
```

Add benchmarking to the previous cells, and take a moment to reflect on these result. Is iterrows() or iloc() faster?

Next, lets use some functional programming! Try using `apply`

In [6]:
```
df['distance'] = df.apply(lambda row: haversine(row['start_station_latitude'], \
                                                row['start_station_longitude'], \
                                                row['end_station_latitude'], \
                                                row['end_station_longitude']), axis=1)
```

Lets vectorize!

In [7]:
```
wall_time_3 = time.time()
CPU_time_3 = time.process_time()
df['distance'] = haversine(df['start_station_latitude'], df['start_station_longitude'], \
                           df['end_station_latitude'], df['end_station_longitude'])
end_wall_time_3 = time.time()
end_CPU_time_3 = time.process_time()
execution_wall_time_3 = end_wall_time_3 - wall_time_3
execution_CPU_time_3 = end_CPU_time_3 - CPU_time_3
benchmarking_results.append(['Vectorizing without values', execution_wall_time_3, execution_CPU_time_3])
```

Lets try numpy vectorize

In [8]:
```
wall_time_4 = time.time()
CPU_time_4 = time.process_time()
df['distance'] = haversine(df['start_station_latitude'], df['start_station_longitude'], \
                           df['end_station_latitude'].values, df['end_station_longitude'].values)
end_wall_time_4 = time.time()
end_CPU_time_4 = time.process_time()
execution_wall_time_4 = end_wall_time_4 - wall_time_4
execution_CPU_time_4 = end_CPU_time_4 - CPU_time_4
benchmarking_results.append(['Vectorizing with values method', execution_wall_time_3, execution_CPU_time_3])
```

Is there anything you can do to the cell above to further improve the performance? Look carefully!

Create a table summarizing the performance results.

In [9]:
```
benchmark_df = pd.DataFrame(benchmarking_results, columns=["Method", "CPU Time (s)", "Wall Time (s)"])
benchmark_df
```

Out[9]:

	Method	CPU Time (s)	Wall Time (s)
0	Using the iterrows() method	27.007194	27.012193
1	Using the iloc() method	17.641447	17.646564
2	Vectorizing without values	0.023097	0.026989
3	Vectorizing with values method	0.023097	0.026989