Exploring Bike Share Data

How to access trip data from Citi Bike and prepare it for analysis using Python and Pandas

Many bike share systems make available their trip data for those who want to understand how their systems are used. The bike share system in New York City, Citi Bike, is one of them, but they don't provide much more than the data. I've got some experience in obtaining and preparing their data for visualization, so in this article I will show you how to get started with this rich data source.

Photo by the author

In the Before Times I commuted from suburban New Jersey to my job as a Product Manager in New York City at an office, now shuttered, above Penn Station. To get around in the City at lunch or after work I often relied on Citi Bike, New York's bike share system. I found I could get to destinations in midtown and even further afield faster than walking and cheaper than the bus or subway. When I discovered that Citi Bike made trip data publicly available I thought that it might provide an interesting use case for the data preparation product that I managed.

Using real data turned out to be much more interesting then the sample files that we had been using because there were actual anomalies that needed to be cleaned up to make the data useful for analysis, and there were interesting stories to tell from the data.

The trip data files contain one record for each ride, around two million records per month, depending on the season. It's a traditional bike share system with fixed stations where a user picks up a bike at one dock, using a key fob or a code, and returns at another. For each ride the station and time when the ride started and stopped is recorded.

Some limited information about the rider is also recorded: their gender and year of birth. Citi Bike also distinguishes between what they call Subscribers who buy an annual pass (current cost is $179 for unlimited rides up to 45 minutes) and Customers who buy a day pass ($15 for unlimited 30 minute rides) or a single ride pass ($3).

For each user type there are overage fees for longer rides. For Customers it's $4 per 15 minutes; for Subscribers it's $0.15 per minute. These fees seem to be designed to discourage longer rides, more so than to increase revenue.

The Citi Bike System Data page describes the information provided. The specific information for each ride is:

- Trip Duration (seconds)
- Start Time and Date
- Stop Time and Date
- Start Station Name
- End Station Name
- Station ID
- Station Lat/Long
- Bike ID
- User Type (Customer = 24-hour pass or single ride user; Subscriber = Annual Member)
- Gender (Zero=unknown; 1=male; 2=female)

- Year of Birth

The kinds of questions we wanted to answer included ones like these: What's the most common ride duration? What times of the day does the system get the most usage? How much does ridership vary over the course of a month? What are the most used stations? How old are the riders?

While the answers to these questions can be found in the trip data files the data needs to be augmented to provide easy answers. For example the trip duration in seconds is too granular; minutes would be more useful.

Over the years I used this data for numerous presentations to customers and at user group meetings. And the cleansed data I created was used by the product managers for a visualization tool for their own presentations.

When I happened to use Jupyter Notebook, Python and Pandas for another project I became interested in seeing what it would take to prepare the Citi Bike trip data using these tools.

Jupyter Notebook is an open-source web-based application that allows you to create and share documents that contain code, visualizations and narrative text. It's commonly used for data preparation and visualization but has many other uses as well. **Python** is the programming language used by default and **Pandas** is a software library widely used for data manipulation and analysis. I also used **Seborn** as an easy way to visualize the data.

The Jupyter Notebook with all the code and output can be found on github.

Downloading Citi Bike Trip Data

The data can be downloaded from a link on the page referenced above to downloadable files for Citibike tripdata . There's a file for each month, a zip file containing a single csv file. The files are in chronological order starting in 2013, so to find the recent ones you have to scroll down. Don't scroll down all the way to the bottom though; the files with JC in the name are for Jersey City, not New York.

For this tutorial I'm using the file for March 2020 which for reasons that will become clear is a smaller file than most. Find the file with the name starting with 202003.

On a Windows system download the file, unzip it to a new directory under your user directory.

On Linux you'll need to use the `wget` command to download the file. If you don't have it installed you can install it using the command for your Linux distribution

```
sudu apt-get install wget # Ubuntu
sudu yum install wget # RHEL and CentOS 7 and earlier
sudu dnf install wget # RHEL and CentOS 8
```

To get the file on a Linux system all you need from the web page is the URL. Right click on the file with the name starting 202003 and select Copy link address. From a Linux command prompt create a directory for these files and switch to it. Type `wget` and right click to copy the address from clipboard. Then unzip archive and delete it.

```
mkdir bikeshare
cd bikeshare
wget https://s3.amazonaws.com/tripdata/202003-citibike-
tripdata.csv.zip
unzip 202003-citibike-tripdata.csv.zip
rm 202003-citibike-tripdata.csv.zip
```

Installing Jupyter Notebook

There are multiple ways to install **Jupyter** but first you need to install Anaconda, a data science platform, and there are multiple options here too. I used **Miniconda**, "a free minimal installer for conda" which can be downloaded from miniconda. Be sure to get miniconda3 which is Python 3 based.

Then you can get Jupyter from Installing the Jupyter Software. I chose to use the classic **Jupyter Notebook**.

On Linux you can use `wget` to download a shell script, then make it executable and run it.

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-
Linux-x86_64.sh
chmod +x Miniconda3-latest-Linux-x86_64.sh
./Miniconda3-latest-Linux-x86_64.sh
```

Then add the location where minconda3 was install to your PATH, either in the current session or to your profile. For example:

```
export PATH=~/miniconda3/bin:$PATH
```

Once that is done you can install Jupyter Notebook and other libraries used here.

```
conda install -c conda-forge notebook pyarrow
conda install pandas seaborn
```

Then start the notebook and you can connect to it from a web browser.

```
jupyter notebook
```

Importing Trip Data

On the Jupyter home page click on the **bikeshare** directory created earlier to select it. You should see the Citi Bike Trip data file there.

Click on New to create a new notebook. Then in the first cell enter the commands below to import the libraries needed. Click Ctrl-Enter to execute the command. These commands have no output, the bracketed number turns to an asterisk while the command is running, and back to the number when it completes.

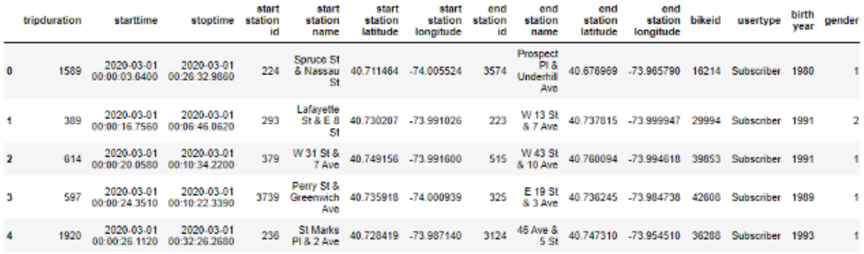
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
```

Read the trip data file into a DataFrame which is an in-memory object like a table with rows and columns. If the file is in the same directory as your notebook you can just use the file name as shown. If it's anywhere else use the full path to the file name.

```
df = pd.read_csv('202003-citibike-tripdata.csv')
```

You can view the first five rows in the data frame with the `head()` method.

```
df.head()
```



	tripduration	starttime	stoptime	start station id	start station name	start station latitude	start station longitude	end station id	end station name	end station latitude	end station longitude	bikeid	usertype	birth year	gender
0	1589	2020-03-01 00:00:03.6400	2020-03-01 00:26:32.9860	224	Spruce St & Nassau St	40.711464	-74.005524	3574	Prospect Pl & Underhill Ave	40.676969	-73.965790	16214	Subscriber	1980	1
1	389	2020-03-01 00:00:16.7560	2020-03-01 00:06:46.0620	293	Lafayette St & E 8 St	40.730207	-73.991026	223	W 13 St & 7 Ave	40.737815	-73.999947	29994	Subscriber	1991	2
2	614	2020-03-01 00:00:20.0580	2020-03-01 00:10:34.2200	379	W 31 St & 7 Ave	40.749156	-73.991600	515	W 43 St & 10 Ave	40.760094	-73.994618	39853	Subscriber	1991	1
3	597	2020-03-01 00:00:24.3510	2020-03-01 00:10:22.3390	3739	Perry St & Greenwich Ave	40.735918	-74.000939	325	E 19 St & 3 Ave	40.736245	-73.984738	42608	Subscriber	1989	1
4	1920	2020-03-01 00:00:26.1120	2020-03-01 00:32:26.2680	236	St Marks Pl & 2 Ave	40.728419	-73.987140	3124	46 Ave & 5 St	40.747310	-73.954510	36288	Subscriber	1993	1

Output from Head

You can get some more information about the columns with the `info()` method.

```
df.info()
```

The output shows the number of rows (just over a million) and the number of columns.

The values for Count are all the same, telling us there are no missing values for any columns.

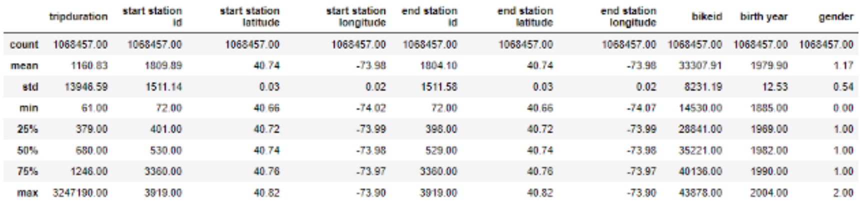
The Dtype column shows the data types of each of the columns. The types int64 and float64 indicate 64-bit integer and floating type values, object indicates strings.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1068457 entries, 0 to 1068456
Data columns (total 15 columns):
 #   Column                   Non-Null Count    Dtype  
---  ------                   --------------    -----  
 0   tripduration             1068457 non-null  int64  
 1   starttime                1068457 non-null  object 
 2   stoptime                 1068457 non-null  object 
 3   start station id         1068457 non-null  int64  
 4   start station name       1068457 non-null  object 
 5   start station latitude   1068457 non-null  float64
 6   start station longitude  1068457 non-null  float64
 7   end station id           1068457 non-null  int64  
 8   end station name         1068457 non-null  object 
 9   end station latitude     1068457 non-null  float64
 10  end station longitude    1068457 non-null  float64
 11  bikeid                   1068457 non-null  int64  
 12  usertype                 1068457 non-null  object 
 13  birth year               1068457 non-null  int64  
 14  gender                   1068457 non-null  int64  
dtypes: float64(4), int64(6), object(5)
memory usage: 122.3+ MB
None
```

For the numeric fields you can also see how the data is distributed using the `describe()` method. It shows the minimum and maximum values, the mean and median (50th percentile) and quartiles.

By default the values display in scientific notation which is hard to read, but they can be rounded to two decimal places using the the `round` method.

```
df.describe().round(2)
```



	tripduration	start station id	start station latitude	start station longitude	end station id	end station latitude	end station longitude	bikeid	birth year	gender
count	1068457.00	1068457.00	1068457.00	1068457.00	1068457.00	1068457.00	1068457.00	1068457.00	1068457.00	1068457.00
mean	1160.83	1809.89	40.74	-73.98	1804.10	40.74	-73.98	33307.91	1979.90	1.17
std	13946.59	1511.14	0.03	0.02	1511.58	0.03	0.02	8231.19	12.53	0.54
min	61.00	72.00	40.66	-74.02	72.00	40.66	-74.07	14530.00	1885.00	0.00
25%	379.00	401.00	40.72	-73.99	398.00	40.72	-73.99	28841.00	1969.00	1.00
50%	680.00	530.00	40.74	-73.98	529.00	40.74	-73.98	35221.00	1982.00	1.00
75%	1246.00	3360.00	40.76	-73.97	3360.00	40.76	-73.97	40136.00	1990.00	1.00
max	3247190.00	3919.00	40.82	-73.90	3919.00	40.82	-73.90	43878.00	2004.00	2.00

Output from Describe

Just from this summary we can see that we need to clean up the data to make it useful for analysis. For example:

tripduration. Here we can see shortest trips were 61 seconds (Citi Bike ignores trips that are 60 seconds or less). The longest was 3247190 seconds (or 902 hours) which sounds like someone didn’t dock their bike. Numbers that large will throw off the mean, so we’ll want to address that.

birth year. The earliest year is 1885. I don’t think anyone born then is still alive, let alone riding a bicycle!

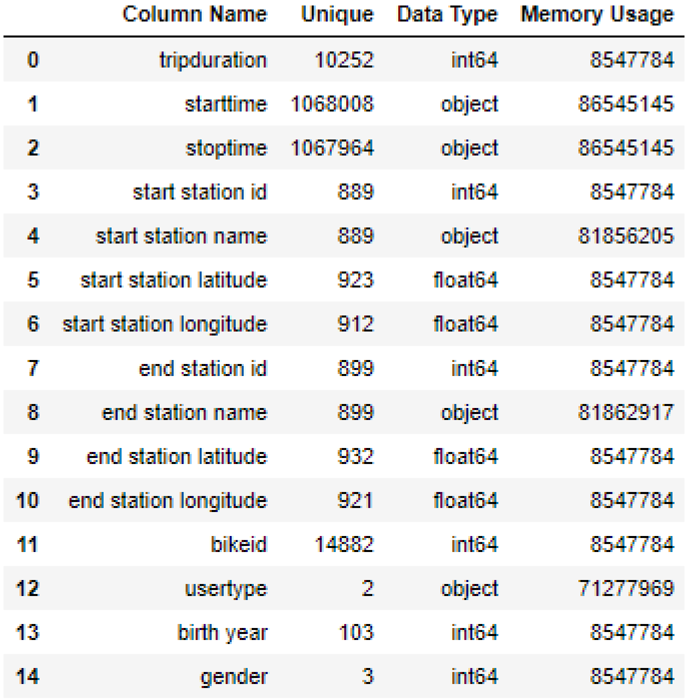
starttime/stoptime. These columns are loaded with data type object. In order for use to use the components (date or time) we need to convert them to timestamps.

The Pandas Data Frame

We saw above the column names and data types, we can get additional information about the data in the data frame with an expression like this that shows for each column its name, number of unique values, its data type and the amount of memory the column uses.

```
pd.DataFrame.from_records([(col, df[col].nunique(),
df[col].dtype, df[col].memory_usage(deep=True) )  for col in
df.columns],
  columns=['Column Name', 'Unique', 'Data Type','Memory
Usage'])
```

Here we can see that the columns stored with a data type of **object** take up around ten times the space in memory as the integer or float data types. As long as we’re just looking at one month’s data that’s not so much a problem, but for a year’s data it would be.



	Column Name	Unique	Data Type	Memory Usage
0	tripduration	10252	int64	8547784
1	starttime	1068008	object	86545145
2	stoptime	1067964	object	86545145
3	start station id	889	int64	8547784
4	start station name	889	object	81856205
5	start station latitude	923	float64	8547784
6	start station longitude	912	float64	8547784
7	end station id	899	int64	8547784
8	end station name	899	object	81862917
9	end station latitude	932	float64	8547784
10	end station longitude	921	float64	8547784
11	bikeid	14882	int64	8547784
12	usertype	2	object	71277969
13	birth year	103	int64	8547784
14	gender	3	int64	8547784

We can easily convert the start and end time columns from object to to timestamp:

```
df['starttime'] = pd.to_datetime(df['starttime'])
df['stoptime'] = pd.to_datetime(df['stoptime'])
```

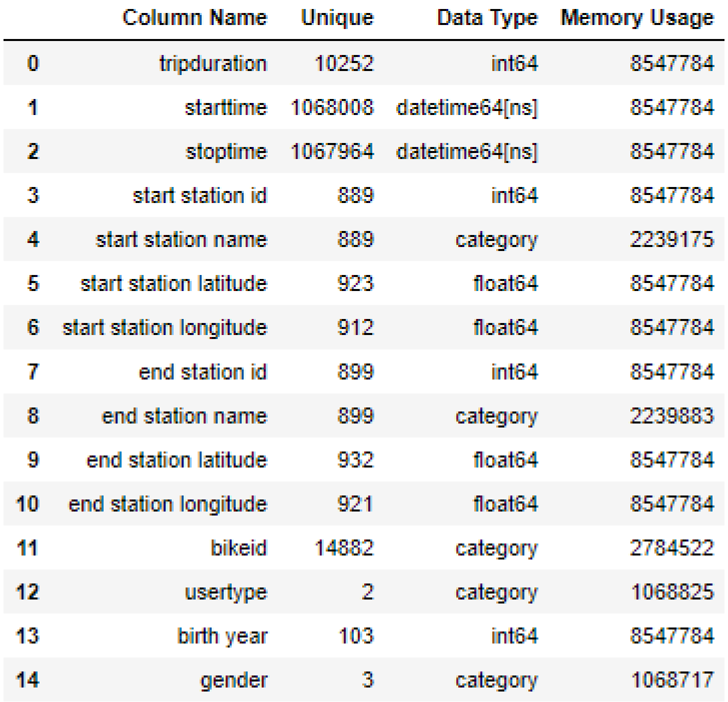
We can also reduce the amount of memory used for some of the other columns by using the **category** data type. Categories are useful when there are a limited number of unique values for a column compared to the number of rows. The actual values are stored just once, and instead of storing a long string in each row, just an integer is stored that points to the actual value.

This file contains over a million rows, but from the unique count for the start and end station names we see that there 899 stations for this month, so they are good candidates to store as categorical data.

The **usertype** column with only two unique values should get similar treatment, as should **gender** with three values. **Bikeid** could also be treated similarly.

```
cols = ['start station name', 'end station name', 'bikeid',
'usertype', 'gender']
for col in cols:
    df[col] = df[col].astype('category')
```

Now if we run the same report as before we can see that the data types have been changed and that the memory used by these columns has been greatly reduced. This will also speed up processing that uses these columns.



	Column Name	Unique	Data Type	Memory Usage
0	tripduration	10252	int64	8547784
1	starttime	1068008	datetime64[ns]	8547784
2	stoptime	1067964	datetime64[ns]	8547784
3	start station id	889	int64	8547784
4	start station name	889	category	2239175
5	start station latitude	923	float64	8547784
6	start station longitude	912	float64	8547784
7	end station id	899	int64	8547784
8	end station name	899	category	2239883
9	end station latitude	932	float64	8547784
10	end station longitude	921	float64	8547784
11	bikeid	14882	category	2784522
12	usertype	2	category	1068825
13	birth year	103	int64	8547784
14	gender	3	category	1068717

Exploring the Data

Trip Duration

We'd like to know what the most common trip duration times are. But the **tripduration** column is stored in seconds which is too precise for this purpose so we’ll create an additional column **tripminutes** that has the trip duration in minutes.

Then we can use Seaborn displot to show the count of rides for each duration trip duration.

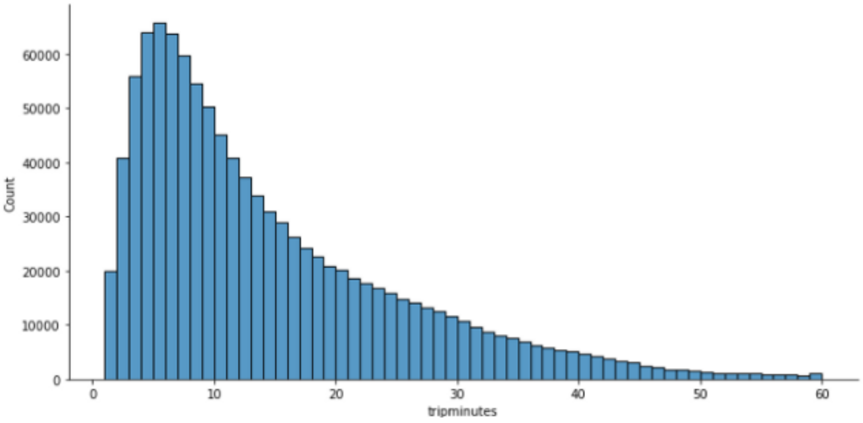
The parameters used are:

data—The name of the Data Frame.
x—The name of the column to chart.
bins—A list of values. Because of the overage fees for rides more than 30 or 45 minutes the number of longer rides is small, so we will only count rides up to an hour.
aspect—The ration of height to width. The value 10/5 gives a a wider chart than the default

The semi-colon at the end of the line prevents a spurious line of text from appearing above the chart.

```
df['tripminutes'] = df['tripduration'] // 60
sns.displot(data=df,x="tripminutes", bins=range(1,61),
aspect=10/5);
```

This chart shows that the most common trip is five minutes long (the tallest bar) followed by four and six minutes (the bars before and after that one). After that the chart shows a classic “long tail” of fewer trips for longer rides.



Rides by Hour and Day

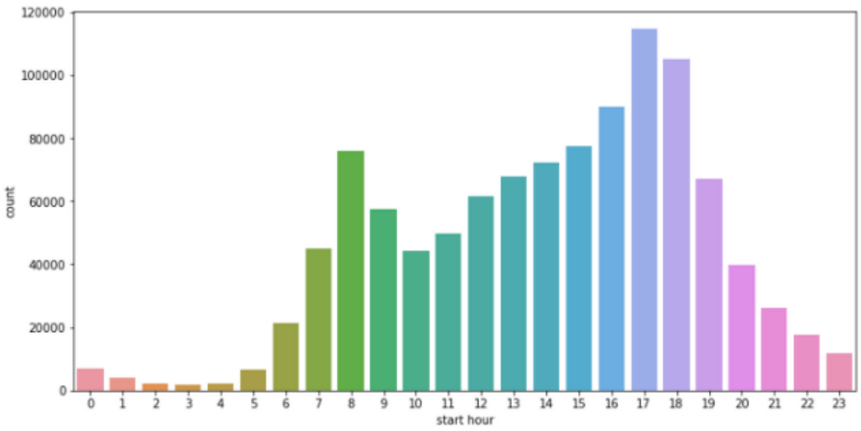
Next we'd like to see hour the rides are distributed across the day and the month. Since we converted the **starttime** column to a datetime value we can use its methods to extract the hour, day and day of week. With just 24, 31 or seven possible values these are good columns to store as categories.

```
df['start hour']=df['starttime'].dt.hour.astype('category')
df['start day']=df['starttime'].dt.day.astype('category')
df['weekday']=df['starttime'].dt.weekday.astype('category')
```

Then we can plot the rides per hour using seaborn `countplot`. Here the `figsize` setting allows for a wider chart.

```
plt.figure(figsize=(12,5))
sns.countplot(data=df, x="start hour" ) ;
```

Here we can see a morning rush hour and an evening rush hour.



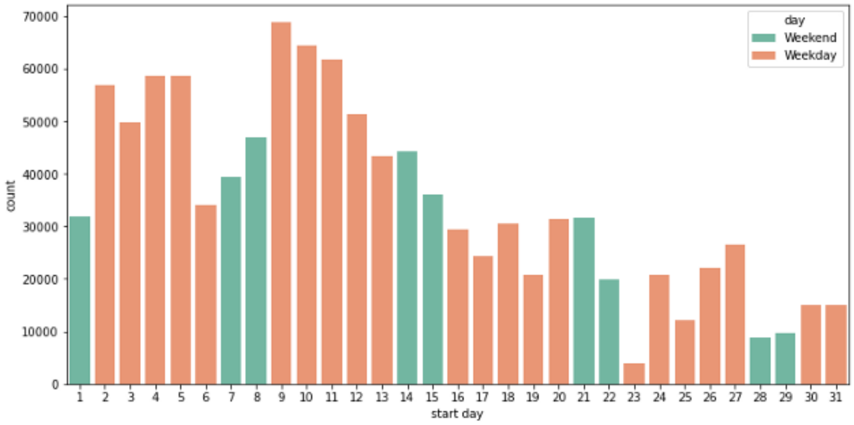
Next let's look at rides per day. It's helpful to highlight the weekends to understand the data so I'll create an additional column that indicates if a day is a weekday or weekend dat.

The additional parameters used for `countplot` are

hue—the name of the column that controls the color of the bar
dodge—show a single column for the multiple hue values

```
df['day'] = ["Weekday" if d <= 4 else "Weekend" for d in
df['weekday']]
plt.figure(figsize=(12,6))
sns.set_palette("Set2")
sns.countplot(data=df,x="start day" , hue='day' ,
dodge=False  )
```

Here we can clearly see the effect the pandemic had on Citi Bike usage in March. The number of daily rides peaked on March 9th. After it was announced that Broadway theaters would be closed on March 12th the number of daily rides decreased further, and remained so for the rest of the month.



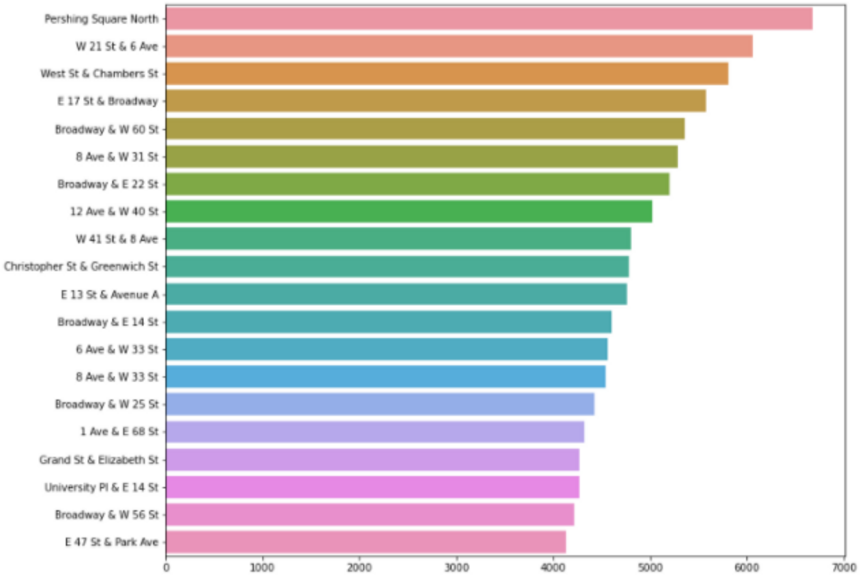
Station Use

Next we'd like to know: which stations get the most use? We can use the data frame method `value_counts()` which for a column returns a series of indexes (here, the station names) and the values (the counts). The counts are conveniently in descending order of frequency so if we select the first 20 values we will get the 20 most frequently used stations. For readability I'll make this chart taller by using a higher figsize value.

```
startstation = df['start station name'].value_counts()[:20]
plt.figure(figsize=(12,10))
```

```
sns.barplot( x=startstation.values ,
y=list(startstation.index),
 orient="h"  ) ;
```

This shows the most frequently used station is Pershing Square North. That's one of the largest stations in the system, and it's just South of Grand Central Terminal, a major railroad and subway hub, so it makes sense that should be number one.

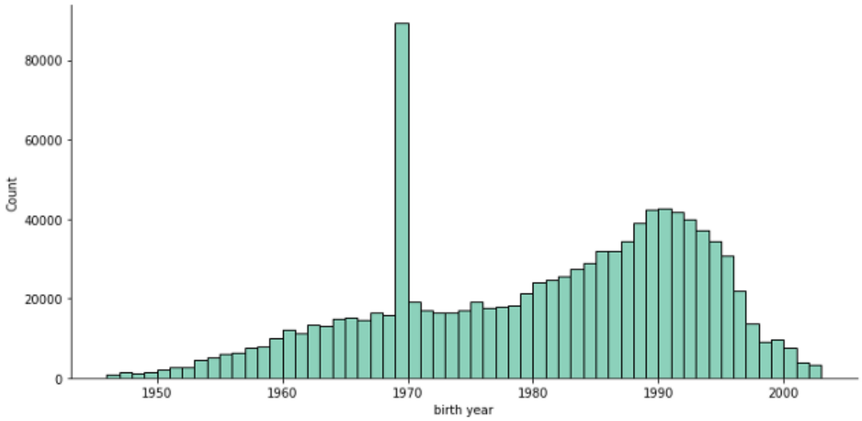


Rider's Ages

Citi Bike provides limited data about who rode the bikes: their birth year, gender and if they were an (annual) Subscriber or (daily) Customer. Still it would be interesting to know how old the riders are. As we saw the **birth year** column has year values are impossible or at best unlikely, so it makes sense to restrict our analysis. I'm going to use 1946 as a cut-off which is generally considered to be the first year of the Baby Boom generation.

```
sns.displot(data=df, x="birth year" , bins=range(1946,2004),
aspect=10/5) ;
```

The first thing we notice about this chart is that 1969 sticks out like the proverbial sore thumb:

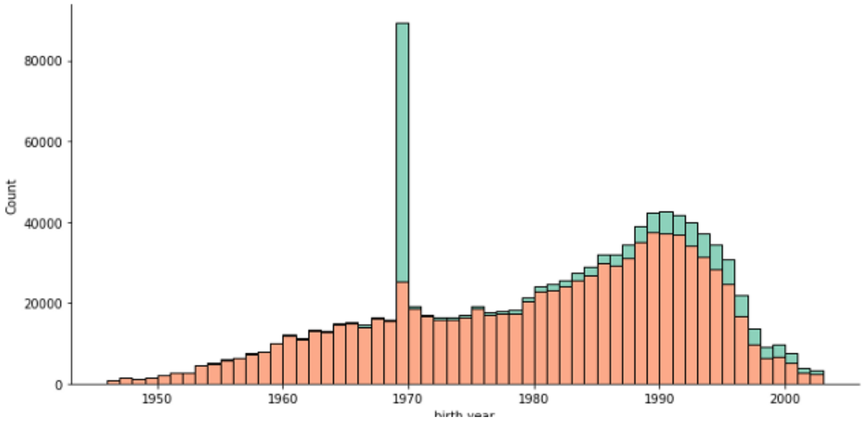


Ride Count by Birth Year

I suspect that it might have something to do with the user type. Citi Bike has profiles for their Subscribers who sign-up online, but Customers buy a day pass at a Kiosk, and may not be inclined to key in their actual year of birth.

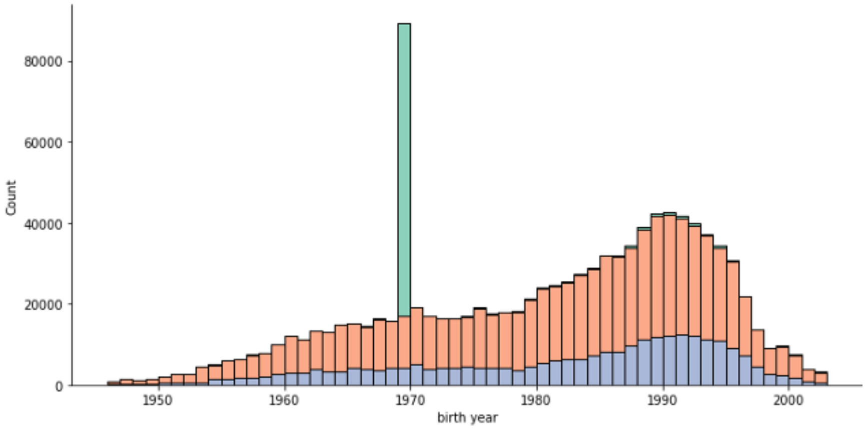
Indeed if I color the bars by User Type then I see that most of riders with a birth year of 1969 were Customers.

```
sns.displot(data=df,x="birth year", bins=range(1946,2004),
hue='usertype', multiple='stack' , hue_order=
["Subscriber","Customer"], aspect=10/5);
```



Ride Count by Birth Year showing User Tupe

But even if we remove the Customers the value for 1969 still looks too high. What about the value for **gender**? A value of zero indicates a missing value. Here the bars are colored by gender in order 0 (undeclared), 1 (male), 2 (female).



Ride Count by Birth Year showing gender

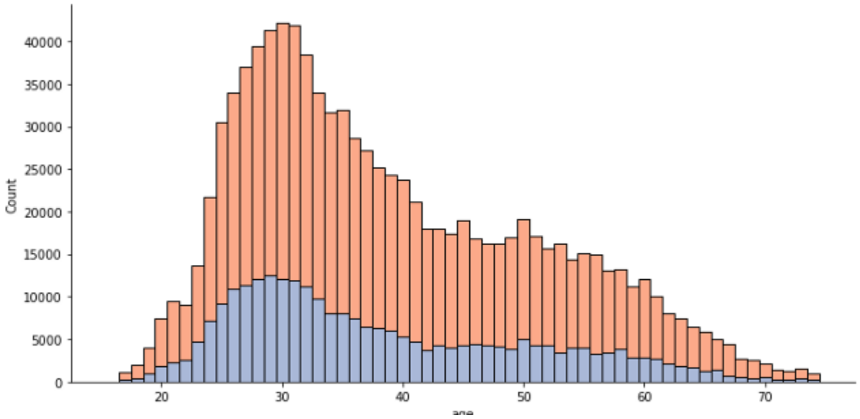
So when calculating the rider's age I want to omit rides where the rider was born before 1946 or where gender is zero. I create a mask that is True when those conditions are met and then. Then I create a new **age** column in the data flow that's set to 2020 minus the **birth year** unless is True and then it's set to None.

```
skip = (df['birth year'] < 1946) | (df['gender'] == 0)
df['age'] = (2020 - df['birth year']).mask(skip,None)
```

Finally I can do a plot by rider’s age, still showing the rider’s gender.

```
sns.displot(data=df, x='age', hue='gender',
multiple='stack',  aspect=10/5) ;
```

From here we can see the most common age (the mode) is 30, and indeed the most common ages cluster around it. We also now see that there are far more men than women using Citi Bike.



Ride Count by Age showing Gender

Trip Distances

Unlike some “dockless” bike share systems with GPS systems on the bikes, Citi Bike doesn’t have any way to record the route taken by a rider.

However with the starting and ending latitude and longitude for each trip it’s possible to at least calculate the distance “as the crow files” using the Haversine Formula which calculates the “great circle” distance between any two points on earth. When the two stations are on the same street or avenue the distance will be reasonably accurate, but otherwise not so much. Still it provides a *minimum* distance between two stations.

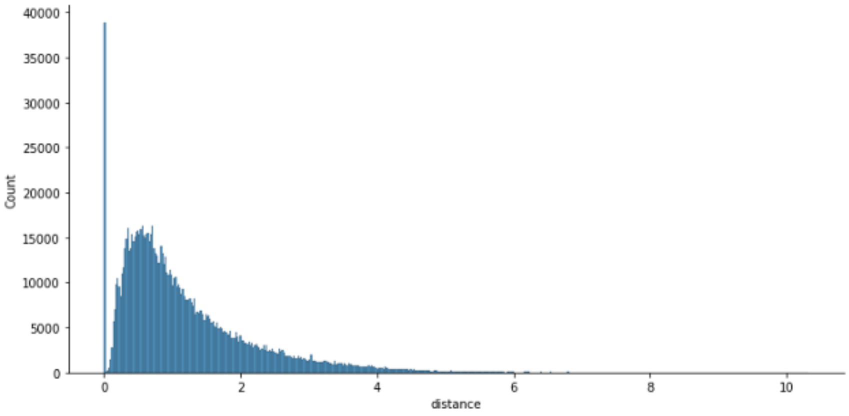
Here’s a Python program that represents the formula.

```
import numpy as np
def haversine(lat1, lon1, lat2, lon2):
    lon1, lat1, lon2, lat2 = map(np.radians ,[lon1, lat1,
lon2, lat2])


    h = np.sin((lat2-lat1)/2.0)**2 + np.cos(lat1) *
np.cos(lat2) * np.sin((lon2-lon1)/2.0)**2


    miles = 3959 * (2 * np.arcsin(np.sqrt(h)))
    return miles
```

We can use it to create a variable to add to the data frame, and use for a chart.

```
distance = haversine(df['start station latitude'],df['start
station longitude'],df['end station latitude'],df['end
station longitude'])
df['distance'] = distance
sns.displot(data=df, x="distance",  aspect=10/5  ) ;
```

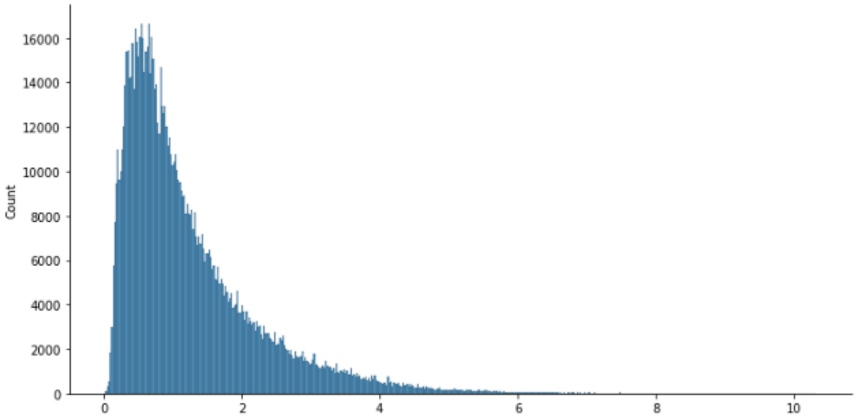
But the chart shows us that there a lot of trips with a distance of zero! It turns out that even though Citi Bike is intended for *transportation* it also gets used for *recreation* where trips start and end at the same station.



Count of trips by calculated distance in miles

We can remove those trips from the distance variable to get a more readable chart.

```
distance = list(filter(lambda distance: distance !=0 ,
distance))
sns.displot(x=distance ,  aspect=10/5  ) ;
```



Count of trips by calculated distance in miles

The combination of time and distance information offers many opportunities for further analysis, for example you could calculate the (approximate) speed for each trip and analyze it by age, gender or station.

Saving to Parquet

After we've changed the data types of some of the columns and derived new columns we'll want to save the DataFrame to a file so that we can do additional analysis. There are a number of options but I think the best choice is Parquet, a columnar file format that is well integrated with Pandas. As a column store it provides much faster retrieval times when not every column is needed.

```
df.to_parquet('202003-citibike-tripdata.parquet')
```

Parquet also takes up less space on disk than other formats, even without additional compression. We can compare the sizes of the two files and see that the parquet file takes about 15% of the space of the original csv file, even with the additional columns added.

```
202003-citibike-tripdata.csv      202,642,779
202003-citibike-tripdata.parquet   35,080,153
```

To read the Parquet file into a Data Frame for further analysis use:

```
df = pd.read_parquet('202003-citibike-tripdata.parquet')
```