Burke-Gilman_commuter_traffic

May 6, 2021

1 Tracking traffic on the Burke Gilman trail

This notebook will walk you through the process of exploring traffic patterns on the Burke Gilman trail, and asking and answering specific research questions using that data. The data we're using today comes from data.seattle.gov which is an online portal where the city of Seattle hosts publicly-available datasets.

The exercises below will use the Burke Gilman trail north of NE 70th St Bicycle and Pedestrian Counter dataset, which collects north and south traffic by bicyclists and pedestrians on a high-traffic portion of the Burke Gilman trail.

2 Gathering the data

Our first step here is to gather a sample of the data to work with. The full dataset is 53,000 rows, but we can use an API query to specify what time range we want to look at, so that we don't need to download the entire dataset. For now, we'll grab one year's worth of data—2019.

If you want more practice with the Seattle Open Data API, you can use the notebook `SODA_API_demo.ipynb`.

[1]:
```
#download the requests library, a bundle of code that is useful for sending and␣
↪retrieving data over the internet
import requests
```

[2]:
```
#the URL we're retrieving the data from. Copy/paste it into your browser to␣
↪view it!
api_endpoint = "https://data.seattle.gov/resource/2z5v-ecg8.json?"

#the parameters we're passing to the API, to specify what subset of data we␣
↪want.
api_parameters = "$limit=50000&$where=date > '2019-01-01T00:00:00' AND date <␣
↪'2020-01-01T00:00:00'"
```

Now we will combine these two strings into a single long URL, which is our API request. It ends up looking like this (run the cell below).

[3]:
```
print(api_endpoint + api_parameters)
```

```
https://data.seattle.gov/resource/2z5v-ecg8.json?$limit=50000&$where=date >
'2019-01-01T00:00:00' AND date < '2020-01-01T00:00:00'
```

Copy and paste that full string into your browser to get a sneak peak at the data we'll be working with!

[4]:
```
#request the data from the API
api_request = requests.get(api_endpoint + api_parameters)

#turn this data into a list of dictionaries, so that we can work with it in␣
↪Python
raw_data = api_request.json()
```

Now we have the data, let's see what it looks like...

[5]:
```
#how many rows are in our dataset?
print(len(raw_data))
```

```
8759
```

[6]:
```
#print(raw_data)
```

That's kind of hard to read, isn't it? Fortunately, there's another useful Python library called "pretty print" (`pprint`), that we can use to make it easier to read this data structure.

[7]:
```
#import the pretty print library
from pprint import pprint
```

You use `pprint` the same way you use `print`

[8]:
```
#pprint(raw_data)
```

Now that we have our data sample and we know how the data is organized (a list of dictionaries, with one dictionary for each hour of traffic), we can start asking research questions about the data.

2.1 Question #1: how many people used the Burke Gilman during commute hours in 2019?

City planners often need to know the traffic volumes on particular roads and trails, so that they can prioritize maintenance and improvements. Anyone who's been on the Burke Gilman on a weekday at 5pm knows that it's a busy thoroughfare, but how busy is it?

For our first step, we need to decide what we mean by commute hours. Let's say between 6am and 9:59am is the "morning" commute, and 3pm-6:59pm is the "evening commute".

We'll store these values in two lists, so that as we loop through each hour in the dataset later we can check whether the traffic during that hour falls within the morning or evening commute.

[9]:
```
morning_commute_hours = ['06:00:00','07:00:00', '08:00:00', '09:00:00']
evening_commute_hours = ['15:00:00','16:00:00', '17:00:00', '18:00:00']
```

Now we need some place to store the morning/evening commute counts, as we loop through the dataset. We'll create a dictionary with keys for 'morning' and 'evening' for this purpose. The values for each of these will start at 0, and increase as we loop through the dataset and add the traffic we see.

[10]:
```
#create an empty dictionary to hold our counts
commuters = {'morning':0, 'evening':0}
```

Now we can loop through our `raw data` list and examine each dictionary in that list. If the time of day in that dictionary matches one of the times we've listed in our morning and evening commute hour lists we made above, then we take the value of `bgt_north_of_ne_70th_total` for that dictionary and add it to one of the totals in `commuters`. If it doesn't match, we move on to the next one and do the same thing.

[11]:
```
for hour_count in raw_data:

    traffic_hour = hour_count['date'][11:19] #get the timestamp

    if traffic_hour in morning_commute_hours: #if it's in the morning, add the␣
↪count to our morning total
        commuters['morning'] = commuters['morning'] +␣
↪int(hour_count['bgt_north_of_ne_70th_total'])

    elif traffic_hour in evening_commute_hours: #if it's in the evening, add␣
↪the count to our evening total
        commuters['evening'] = commuters['evening'] +␣
↪int(hour_count['bgt_north_of_ne_70th_total'])

    else:
        pass #we don't care who's on the BGT during non-commute hours, so we␣
↪ignore it and move on
```

2.1.1 Understanding the loop we just made

interlude 1: data types If you scroll up to our pprinted dataset above, you'll see that the numeric values for `bgt_north_of_ne_70th_total` (the hourly traffic counts) have '' around them. This means they are being stored as strings.

[12]:
```
#we can ask Python to tell us what type of value this is using 'type'
print(type(raw_data[0]['bgt_north_of_ne_70th_total']))
```

```
<class 'str'>
```

If we want to add these numeric values together, we will need to turn them into integers. That's what's happening when you see `int(hour_count['bgt_north_of_ne_70th_total'])` in the loop above.

[13]:
```
#converting the string to an integer
str_to_int = int(raw_data[0]['bgt_north_of_ne_70th_total'])
```

```
print(type(str_to_int))
```

```
<class 'int'>
```

interlude 2: slicing The date and time for each item in the list is stored in a single string, like `'2019-01-01T06:00:00.000'` Right now, we are only interested in the time of day, which is in the middle—between "T" (position 10) and "." (position 19), so we slice the beginning and the end of the string off before we compare the value with our commute hour lists.

[14]:
```
print(raw_data[0]['date'])
```

```
2019-01-01T01:00:00.000
```

[15]:
```
print(raw_data[0]['date'][:11]) #slice the string to just get the stuff before␣
↪the time
```

```
2019-01-01T
```

[16]:
```
print(raw_data[0]['date'][19:]) #slice the string to just get the stuff after␣
↪the time
```

```
.000
```

[17]:
```
print(raw_data[0]['date'][11:19]) #slice the string to just get the time
```

```
01:00:00
```

2.1.2 Getting the answer to Q1

Now if our loop above worked, we'll know how many people used the Burke Gilman during commute hours in 2019 (at 70th street, at least)!

[18]:
```
pprint(commuters)
```

```
{'evening': 168241, 'morning': 114366}
```

[19]:
```
#we can make these values into a human readable sentence, if we convert the␣
↪ints back into strings
print("There were "
    + str(commuters['morning'])
    + " morning commuters and "
    + str(commuters['evening'])
    + " on the Burke Gilman at 70th Street in 2019!"
     )
```

```
There were 114366 morning commuters and 168241 on the Burke Gilman at 70th
Street in 2019!
```

2.2 Question 2: what were the busiest hours on the Burke Gilman in 2019?

Let's say you want to know when the Burke Gilman is least likely to be congested, so you can plan your trip. How would you find this out?

To do this, we need to start with a list of all of the hours of the day. We could do this manually by creating a list like we did with the morning and evening commute hours lists above, but since there are 24 hours in a day that would take a long time to type out manually! Is there a quicker way?

[20]:
```
#create an empty dictionary, that we will fill with hours
traffic_by_hour = {}
```

[21]:
```
#loop through the dataset again...
for hour_data in raw_data:

    traffic_hour = hour_data['date'][11:19] #grab the hour value from the item

    #if haven't seen this hour yet during the loop...
    if traffic_hour not in traffic_by_hour.keys():

        # ...create an item for it in our dictionary that we can use later to␣
↪track our cumulative hourly counts
        traffic_by_hour[traffic_hour] = 0 #e.g {'01:00:00' : 0}

    else:
        pass
        #if we've already seen this hour and stored it in our dictionary,␣
↪ignore it and move on to the next one
```

[22]:
```
#how many hours in our list?
len(traffic_by_hour)
```

[22]:
```
24
```

[23]:
```
#what does our list look like?
pprint(traffic_by_hour)
```

```
{'00:00:00': 0,
 '01:00:00': 0,
 '02:00:00': 0,
 '03:00:00': 0,
 '04:00:00': 0,
 '05:00:00': 0,
 '06:00:00': 0,
 '07:00:00': 0,
```

```
 '08:00:00': 0,
 '09:00:00': 0,
 '10:00:00': 0,
 '11:00:00': 0,
 '12:00:00': 0,
 '13:00:00': 0,
 '14:00:00': 0,
 '15:00:00': 0,
 '16:00:00': 0,
 '17:00:00': 0,
 '18:00:00': 0,
 '19:00:00': 0,
 '20:00:00': 0,
 '21:00:00': 0,
 '22:00:00': 0,
 '23:00:00': 0}
```

2.3 Getting the answer to Q2

Awesome! Now that we have buckets ready to hold counts for each hour in the day, we can loop through the raw data again and start calculating the traffic counts for each hour.

[24]:
```
for hour_data in raw_data:

    traffic_hour = hour_data['date'][11:19]

    traffic_by_hour[traffic_hour] +=␣
↪int(hour_data['bgt_north_of_ne_70th_total'])

    #you can also un-comment the following line and run it instead of the line␣
↪directly above.
    #this is a slightly shorter way to do the same thing. Re-set␣
↪traffic_by_hour to zero and try it!
#     traffic_by_hour[traffic_hour] +=␣
↪int(hour_data['bgt_north_of_ne_70th_total'])
```

[25]:
```
pprint(traffic_by_hour)
```

```
{'00:00:00': 343,
 '01:00:00': 157,
 '02:00:00': 147,
 '03:00:00': 203,
 '04:00:00': 760,
 '05:00:00': 4304,
 '06:00:00': 15275,
 '07:00:00': 25894,
 '08:00:00': 37369,
 '09:00:00': 35828,
```

```
 '10:00:00': 36095,
 '11:00:00': 38024,
 '12:00:00': 37621,
 '13:00:00': 38328,
 '14:00:00': 39132,
 '15:00:00': 40675,
 '16:00:00': 46041,
 '17:00:00': 46997,
 '18:00:00': 34528,
 '19:00:00': 18987,
 '20:00:00': 9167,
 '21:00:00': 3789,
 '22:00:00': 1376,
 '23:00:00': 692}
```

It looks like the busiest hour is between 5 and 6pm!

interlude 3: plus-equals Writing long lines of code like `commuters['morning'] = commuters['morning'] + int(hour_count['bgt_north_of_ne_70th_total'])` can be tedious, and it makes it more likely that you will make a typo that causes your code to crash. There's an easier way!

Instead of writing `thing_one = thing_one + thing_two`, you can write `thing_one += thing_two`. It does the same thing!

You can try this out yourself by re-running cells 35-38 above, and then using the code `traffic_by_hour[traffic_hour] += int(hour_data['bgt_north_of_ne_70th_total'])`

...instead of the line `traffic_by_hour[traffic_hour] = traffic_by_hour[traffic_hour] + int(hour_data['bgt_north_of_ne_70th_total'])`

2.3.1 Going further with Question 3

- Challenge #1: How would you use Python to find the HOUR with the most traffic? This list is short, so we can probably easily identify the element with the most traffic just by looking at it. But what if it had 10k items in it? How would YOU find the hour with the most traffic?
- Challegen #2: How would you find the busiest DAY on the Burke Gilman in 2019? You should be able to answer this question with the same techniques you used to find the busiest hour!

2.4 Challenge #1 Answer

Using the max method in python to grab the timestamp of the hour with the most traffic count.

[26]:
```
# max method to iterate dictionary traffic_by_hour through it's keys
time_val = max(traffic_by_hour, key=traffic_by_hour.get)

print("The hour with the most traffic was at " + time_val + " with a traffic␣
 ↪count of " + str(traffic_by_hour[time_val]) + ".")
```

```
The hour with the most traffic was at 17:00:00 with a traffic count of 46997.
```

2.5 Challenge #2 Answer

Create a hash table to sort the dates into a dictionary, then counting the total through retrieving the total traffic count on that day. Essentially reused challenge #1's selection method.

[27]:
```
# traffic_date is the main dictionary which keeps track of daily traffic with␣
↪dates as keys
traffic_date = {}
for hour_data in raw_data:

    date = hour_data['date'][5:10] # select date
    if date not in traffic_date.keys():
        traffic_date[date] = 0
    else:
        traffic_date[date] += int(hour_data['bgt_north_of_ne_70th_total'])␣
↪#aggregate traffic count

date_val = max(traffic_date, key=traffic_date.get) # find the day with the max␣
↪traffic
print("The day with the most traffic within 2019 was on " + date_val + " with a␣
↪traffic count of " + str(traffic_date[date_val]) + ".")
```

```
The day with the most traffic within 2019 was on 05-27 with a traffic count of
3874.
```

2.6 Question 3: What is the busiest hours for bikes vs pedestrians?

One cool thing about this dataset is that it doesn't just count by hour, it counts by what kind of traffic: bikes or pedestrians. Let's run the same set of commands as above, but this time we'll break things apart into bikes and peds.

[28]:
```
#create an empty dictionary
traffic_by_hour_bp = {}

for hour_data in raw_data:

    traffic_hour = hour_data['date'][11:19]

    #if haven't seen this hour yet, create an item for it that we can use to␣
↪track hourly counts later
    if traffic_hour not in traffic_by_hour_bp.keys():

        #this time, we create a dictionary-in-a-dictionary, to hold our bike␣
↪and pedestrian counts separately
        traffic_by_hour_bp[traffic_hour] = {'bikes' : 0, 'pedestrians' : 0}

    else:
        pass
```

```
        #if we've already seen this hour and stored it in our dictionary,␣
↪ignore it and move on

pprint(traffic_by_hour_bp)
```

```
{'00:00:00': {'bikes': 0, 'pedestrians': 0},
 '01:00:00': {'bikes': 0, 'pedestrians': 0},
 '02:00:00': {'bikes': 0, 'pedestrians': 0},
 '03:00:00': {'bikes': 0, 'pedestrians': 0},
 '04:00:00': {'bikes': 0, 'pedestrians': 0},
 '05:00:00': {'bikes': 0, 'pedestrians': 0},
 '06:00:00': {'bikes': 0, 'pedestrians': 0},
 '07:00:00': {'bikes': 0, 'pedestrians': 0},
 '08:00:00': {'bikes': 0, 'pedestrians': 0},
 '09:00:00': {'bikes': 0, 'pedestrians': 0},
 '10:00:00': {'bikes': 0, 'pedestrians': 0},
 '11:00:00': {'bikes': 0, 'pedestrians': 0},
 '12:00:00': {'bikes': 0, 'pedestrians': 0},
 '13:00:00': {'bikes': 0, 'pedestrians': 0},
 '14:00:00': {'bikes': 0, 'pedestrians': 0},
 '15:00:00': {'bikes': 0, 'pedestrians': 0},
 '16:00:00': {'bikes': 0, 'pedestrians': 0},
 '17:00:00': {'bikes': 0, 'pedestrians': 0},
 '18:00:00': {'bikes': 0, 'pedestrians': 0},
 '19:00:00': {'bikes': 0, 'pedestrians': 0},
 '20:00:00': {'bikes': 0, 'pedestrians': 0},
 '21:00:00': {'bikes': 0, 'pedestrians': 0},
 '22:00:00': {'bikes': 0, 'pedestrians': 0},
 '23:00:00': {'bikes': 0, 'pedestrians': 0}}
```

2.7 Getting the answer to Q3

Now that we have the right buckets for every value we want to capture, we're ready to get counts for bikes and peds separately.

[29]:
```
for hour_data in raw_data:

    traffic_hour = hour_data['date'][11:19]

    #note that we're using the 'plus-equals' technique now, which means we have␣
↪to write less code
    traffic_by_hour_bp[traffic_hour]['bikes'] += int(hour_data['bike_north'])␣
↪#add northbound bike counts
    traffic_by_hour_bp[traffic_hour]['bikes'] += int(hour_data['bike_south'])␣
↪#add southbound bike counts
```

```
    traffic_by_hour_bp[traffic_hour]['pedestrians'] +=␣
↪int(hour_data['ped_north']) #add northbound ped counts
    traffic_by_hour_bp[traffic_hour]['pedestrians'] +=␣
↪int(hour_data['ped_south']) #add sounthbound ped counts
```

[30]:
```
pprint(traffic_by_hour_bp)
```

```
{'00:00:00': {'bikes': 245, 'pedestrians': 98},
 '01:00:00': {'bikes': 99, 'pedestrians': 58},
 '02:00:00': {'bikes': 103, 'pedestrians': 44},
 '03:00:00': {'bikes': 179, 'pedestrians': 24},
 '04:00:00': {'bikes': 640, 'pedestrians': 120},
 '05:00:00': {'bikes': 3375, 'pedestrians': 929},
 '06:00:00': {'bikes': 11413, 'pedestrians': 3862},
 '07:00:00': {'bikes': 19422, 'pedestrians': 6472},
 '08:00:00': {'bikes': 24499, 'pedestrians': 12870},
 '09:00:00': {'bikes': 21565, 'pedestrians': 14263},
 '10:00:00': {'bikes': 22624, 'pedestrians': 13471},
 '11:00:00': {'bikes': 26064, 'pedestrians': 11960},
 '12:00:00': {'bikes': 26678, 'pedestrians': 10943},
 '13:00:00': {'bikes': 28166, 'pedestrians': 10162},
 '14:00:00': {'bikes': 29157, 'pedestrians': 9975},
 '15:00:00': {'bikes': 30443, 'pedestrians': 10232},
 '16:00:00': {'bikes': 34648, 'pedestrians': 11393},
 '17:00:00': {'bikes': 37291, 'pedestrians': 9706},
 '18:00:00': {'bikes': 26761, 'pedestrians': 7767},
 '19:00:00': {'bikes': 14101, 'pedestrians': 4886},
 '20:00:00': {'bikes': 6374, 'pedestrians': 2793},
 '21:00:00': {'bikes': 2696, 'pedestrians': 1093},
 '22:00:00': {'bikes': 969, 'pedestrians': 407},
 '23:00:00': {'bikes': 480, 'pedestrians': 212}}
```

Now that there is more data in the notebook, it's a bit harder to figure out the busiest hour for bikes and pedestrians separately, just by looking. So let's use a quick loop to find this.

[31]:
```
top_bike = ['some hour', 0]
top_ped = ['some hour', 0]

for hour, traffic in traffic_by_hour_bp.items():

    if traffic['bikes'] > top_bike[1]:
        top_bike[0] = hour
        top_bike[1] = traffic['bikes']
    else:
        pass

    if traffic['pedestrians'] > top_ped[1]:
        top_ped[0] = hour
```

```
        top_ped[1] = traffic['pedestrians']
    else:
        pass

print("The hour with the most bike traffic is " + top_bike[0] + ", with " +␣
 ↪str(top_bike[1]) + " bikes!")
print("The hour with the most pedestrian traffic is " + top_ped[0] + ", with "␣
 ↪+ str(top_ped[1]) + " pedestrians!")
```

```
The hour with the most bike traffic is 17:00:00, with 37291 bikes!
The hour with the most pedestrian traffic is 09:00:00, with 14263 pedestrians!
```

3 Going further Question #1

- Were there more bikes or pedestrians using the Burke Gilman trail in 2019? Use the block below and try to answer this question.

3.1 Question #1 Answer

Using data from the traffic_by_hour_bp to get data on the bike and pedestrian traffic count then comparing with each other to see whether there were more bikes or pedestrians.

[32]:
```
total_bikes = 0
total_pedestrians = 0
# going through traffic_by_bp to aggregate the total bikes and pedestrians
for hour, traffic in traffic_by_hour_bp.items():
    total_bikes += traffic['bikes']
    total_pedestrians += traffic['pedestrians']
print("During the year of 2019 the total bikes were " + str(total_bikes) + ",␣
 ↪while the total pedestrians was " + str(total_pedestrians))
print("There were " + str(total_bikes - total_pedestrians) + " more bikes on␣
 ↪the Burke Gilham trail in 2019 than pedestrians." if total_bikes >␣
 ↪total_pedestrians else "There were " + str(total_pedestrians - total_bikes)␣
 ↪+ " more pedestrians on the Burke Gilham trail in 2019 than bikes.")
```

```
During the year of 2019 the total bikes were 367992, while the total pedestrians
was 143740
There were 224252 more bikes on the Burke Gilham trail in 2019 than pedestrians.
```

3.2 Question 4: What is the busiest hour for bikes vs. peds AND northbound vs. southbound?

If we want, we can break our data down by bikes vs. peds AND northbound vs. southbound. This is useful because it helps us understand how many people are using the Burke Gilman trail for daily commuting, not just for pleasure trips.

Splitting bike and pedestrian traffic by northboud/southbound is actually almost as easy as lumping them together. We just make our dictionary-in-a-dictionary a little more complex.

[33]:
```
#create a new empty dictionary
traffic_by_hour_bpd = {}

for hour_data in raw_data:

    traffic_hour = hour_data['date'][11:19]

    if traffic_hour not in traffic_by_hour_bpd.keys():

        traffic_by_hour_bpd[traffic_hour] = {'bikes north' : 0, 'pedestrians␣
↪north' : 0,
                                             'bikes south' : 0, 'pedestrians␣
↪south' : 0}
    else:
        pass

pprint(traffic_by_hour_bpd)
```

```
{'00:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '01:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '02:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '03:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '04:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '05:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '06:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
```

```
             'pedestrians south': 0},
'07:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'08:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'09:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'10:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'11:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'12:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'13:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'14:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'15:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'16:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'17:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
             'pedestrians south': 0},
'18:00:00': {'bikes north': 0,
             'bikes south': 0,
             'pedestrians north': 0,
```

```
              'pedestrians south': 0},
 '19:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '20:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '21:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '22:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0},
 '23:00:00': {'bikes north': 0,
              'bikes south': 0,
              'pedestrians north': 0,
              'pedestrians south': 0}}
```

And now instead of adding bike_north + bike_south etc together, we store each of those counts in their own key/value bucket.

[34]:
```
for hour_data in raw_data:

    traffic_hour = hour_data['date'][11:19]

    traffic_by_hour_bpd[traffic_hour]['bikes north'] +=␣
↪int(hour_data['bike_north']) #store count in bike_north
    traffic_by_hour_bpd[traffic_hour]['bikes south'] +=␣
↪int(hour_data['bike_south']) #store count in bike_south

    traffic_by_hour_bpd[traffic_hour]['pedestrians north'] +=␣
↪int(hour_data['ped_north'])
    traffic_by_hour_bpd[traffic_hour]['pedestrians south'] +=␣
↪int(hour_data['ped_south'])

pprint(traffic_by_hour_bpd)
```

```
{'00:00:00': {'bikes north': 57,
              'bikes south': 188,
              'pedestrians north': 50,
              'pedestrians south': 48},
 '01:00:00': {'bikes north': 30,
              'bikes south': 69,
              'pedestrians north': 31,
```

```
             'pedestrians south': 27},
'02:00:00': {'bikes north': 18,
             'bikes south': 85,
             'pedestrians north': 21,
             'pedestrians south': 23},
'03:00:00': {'bikes north': 77,
             'bikes south': 102,
             'pedestrians north': 11,
             'pedestrians south': 13},
'04:00:00': {'bikes north': 474,
             'bikes south': 166,
             'pedestrians north': 55,
             'pedestrians south': 65},
'05:00:00': {'bikes north': 2399,
             'bikes south': 976,
             'pedestrians north': 416,
             'pedestrians south': 513},
'06:00:00': {'bikes north': 7825,
             'bikes south': 3588,
             'pedestrians north': 2023,
             'pedestrians south': 1839},
'07:00:00': {'bikes north': 13207,
             'bikes south': 6215,
             'pedestrians north': 3425,
             'pedestrians south': 3047},
'08:00:00': {'bikes north': 16379,
             'bikes south': 8120,
             'pedestrians north': 6236,
             'pedestrians south': 6634},
'09:00:00': {'bikes north': 11410,
             'bikes south': 10155,
             'pedestrians north': 7637,
             'pedestrians south': 6626},
'10:00:00': {'bikes north': 10517,
             'bikes south': 12107,
             'pedestrians north': 6805,
             'pedestrians south': 6666},
'11:00:00': {'bikes north': 12097,
             'bikes south': 13967,
             'pedestrians north': 5997,
             'pedestrians south': 5963},
'12:00:00': {'bikes north': 12782,
             'bikes south': 13896,
             'pedestrians north': 5671,
             'pedestrians south': 5272},
'13:00:00': {'bikes north': 12944,
             'bikes south': 15222,
             'pedestrians north': 5355,
```

```
             'pedestrians south': 4807},
 '14:00:00': {'bikes north': 13383,
              'bikes south': 15774,
              'pedestrians north': 5067,
              'pedestrians south': 4908},
 '15:00:00': {'bikes north': 13851,
              'bikes south': 16592,
              'pedestrians north': 5143,
              'pedestrians south': 5089},
 '16:00:00': {'bikes north': 13858,
              'bikes south': 20790,
              'pedestrians north': 5680,
              'pedestrians south': 5713},
 '17:00:00': {'bikes north': 12569,
              'bikes south': 24722,
              'pedestrians north': 4765,
              'pedestrians south': 4941},
 '18:00:00': {'bikes north': 9285,
              'bikes south': 17476,
              'pedestrians north': 3841,
              'pedestrians south': 3926},
 '19:00:00': {'bikes north': 5986,
              'bikes south': 8115,
              'pedestrians north': 2562,
              'pedestrians south': 2324},
 '20:00:00': {'bikes north': 3024,
              'bikes south': 3350,
              'pedestrians north': 1525,
              'pedestrians south': 1268},
 '21:00:00': {'bikes north': 1360,
              'bikes south': 1336,
              'pedestrians north': 628,
              'pedestrians south': 465},
 '22:00:00': {'bikes north': 405,
              'bikes south': 564,
              'pedestrians north': 200,
              'pedestrians south': 207},
 '23:00:00': {'bikes north': 154,
              'bikes south': 326,
              'pedestrians north': 109,
              'pedestrians south': 103}}
```

3.2.1 Answering question 4

I will leave it up to you to find the answer to any of these questions (you already know how to do it!) - What is the busiest hour for northbound bike traffic? For southbound bike traffic? - What is the busiest hour for northbound pedestrian traffic? For southbound pedestrian traffic?

If you don't feel like writing code to answer these questions right now, that's okay. Below, I will show you how to export this data in a format that can be used to easily create timeseries graphs in a spreadsheet. It will be much easier to answer these questions with a graph than by looking at rows of numbers!

3.3 Question #1 Answer

To make the work a little easier and less redundant, I decided to create a function to handle the printing as well as finding the max value by using the max method.

[35]:
```
# creating a method to print results to what is the max hour within each␣
↪traffic type (bikes and pedestrians, north or south)
def max_key(data, query):
    print("The busiest hour for {query} were usually around ".
↪format(query=query) + str(max(data, key= lambda k: data[k][query])) + ".")


max_key(traffic_by_hour_bpd, 'bikes north')
max_key(traffic_by_hour_bpd, 'bikes south')
max_key(traffic_by_hour_bpd, 'pedestrians north')
max_key(traffic_by_hour_bpd, 'pedestrians south')
```

```
The busiest hour for bikes north were usually around 08:00:00.
The busiest hour for bikes south were usually around 17:00:00.
The busiest hour for pedestrians north were usually around 09:00:00.
The busiest hour for pedestrians south were usually around 10:00:00.
```

3.4 Exporting the bike/ped north/south dataset for visualization

Now we have some nice rich data that we can graph in a spreadsheet program like Google Sheets or Microsoft Excel.

But to do that, we will need to convert the data to CSV format and export it into a file. We ideally want a format that has a row for each hour, and columns like `hour_of_day | bikes_north | bikes_south | pedestrians_north | pedestrians_south`

The best way to do this is to convert our data from a nested dictionary to a nested list (a list-of-lists!), where each sub-list (which will be a row in our CSV file), contains the values in a consistent order, like:

`[hour_of_day, bike_north, bike_south, ped_north, ped_south]`

[36]:
```
#create a new empty 'master list'
traffic_by_hour_mode_direction = []

#for each hour in our traffic-by-hour-mode-direction dictionary...
for hour, counts in traffic_by_hour_bpd.items():

    #...create a new sub-list that will store the hour, bike, and pedestrian␣
↪data in a consistent order
```

```
    list_element = [] #new empty sub-list
    list_element.append(hour) #add the hour in, e.g. ['06:00:00']
    list_element.append(counts['bikes north']) #append bike north count, e.g.␣
↪['06:00:00', 484]
    list_element.append(counts['bikes south']) #append bike south count, e.g.␣
↪['06:00:00', 484, 82]
    list_element.append(counts['pedestrians north']) # etc...
    list_element.append(counts['pedestrians south']) # etc...

    traffic_by_hour_mode_direction.append(list_element) #add this list to the␣
↪end of our growing master list
```

[37]:
```
pprint(traffic_by_hour_mode_direction)
```

```
[['01:00:00', 30, 69, 31, 27],
 ['02:00:00', 18, 85, 21, 23],
 ['03:00:00', 77, 102, 11, 13],
 ['04:00:00', 474, 166, 55, 65],
 ['05:00:00', 2399, 976, 416, 513],
 ['06:00:00', 7825, 3588, 2023, 1839],
 ['07:00:00', 13207, 6215, 3425, 3047],
 ['08:00:00', 16379, 8120, 6236, 6634],
 ['09:00:00', 11410, 10155, 7637, 6626],
 ['10:00:00', 10517, 12107, 6805, 6666],
 ['11:00:00', 12097, 13967, 5997, 5963],
 ['12:00:00', 12782, 13896, 5671, 5272],
 ['13:00:00', 12944, 15222, 5355, 4807],
 ['14:00:00', 13383, 15774, 5067, 4908],
 ['15:00:00', 13851, 16592, 5143, 5089],
 ['16:00:00', 13858, 20790, 5680, 5713],
 ['17:00:00', 12569, 24722, 4765, 4941],
 ['18:00:00', 9285, 17476, 3841, 3926],
 ['19:00:00', 5986, 8115, 2562, 2324],
 ['20:00:00', 3024, 3350, 1525, 1268],
 ['21:00:00', 1360, 1336, 628, 465],
 ['22:00:00', 405, 564, 200, 207],
 ['23:00:00', 154, 326, 109, 103],
 ['00:00:00', 57, 188, 50, 48]]
```

This worked! BUT something odd happened. Can you see it? For some reason, our '00:00:00' hour got moved to the end of the list.

This happened because Python dictionaries do not reliably preserve the order of items! So even though the data looked like it was in the right order when we had it in the dictionary, when we created our new master list from `traffic_by_hour_bpd` it ended up a little out of order.

Fortunately, unlike dictionaries, lists-of-lists are easy to sort! Especially if you want to sort by the

first item in each sub-list, which we do!

[38]:
```
#sort the list in 'alphabetical order' by the first sub-item, which is the hour␣
↪stamp
traffic_by_hour_mode_direction.sort()
```

[39]:
```
pprint(traffic_by_hour_mode_direction)
```

```
[['00:00:00', 57, 188, 50, 48],
 ['01:00:00', 30, 69, 31, 27],
 ['02:00:00', 18, 85, 21, 23],
 ['03:00:00', 77, 102, 11, 13],
 ['04:00:00', 474, 166, 55, 65],
 ['05:00:00', 2399, 976, 416, 513],
 ['06:00:00', 7825, 3588, 2023, 1839],
 ['07:00:00', 13207, 6215, 3425, 3047],
 ['08:00:00', 16379, 8120, 6236, 6634],
 ['09:00:00', 11410, 10155, 7637, 6626],
 ['10:00:00', 10517, 12107, 6805, 6666],
 ['11:00:00', 12097, 13967, 5997, 5963],
 ['12:00:00', 12782, 13896, 5671, 5272],
 ['13:00:00', 12944, 15222, 5355, 4807],
 ['14:00:00', 13383, 15774, 5067, 4908],
 ['15:00:00', 13851, 16592, 5143, 5089],
 ['16:00:00', 13858, 20790, 5680, 5713],
 ['17:00:00', 12569, 24722, 4765, 4941],
 ['18:00:00', 9285, 17476, 3841, 3926],
 ['19:00:00', 5986, 8115, 2562, 2324],
 ['20:00:00', 3024, 3350, 1525, 1268],
 ['21:00:00', 1360, 1336, 628, 465],
 ['22:00:00', 405, 564, 200, 207],
 ['23:00:00', 154, 326, 109, 103]]
```

Now that we have our data in a list, it's easy to export to a CSV file!

[40]:
```
import csv
```

[41]:
```
with open('bg_traffic_bike_ped_2019.csv', 'w', encoding='utf-8') as f:
    writer = csv.writer(f)
    #write a header row
    writer.writerow(('hour of the day',
                     'northbound bikes',
                     'southbound bikes',
                     'northbound pedestrians',
                     'southbound pedestrians'))

    for i in traffic_by_hour_mode_direction:
        writer.writerow((i[0], i[1], i[2], i[3], i[4]))
```

3.5 Question 5 (Grand challenge!): What day of the week is busiest on the Burke Gilman?

If we really want to understand whether people use the Burke Gilman for commuting, rather than just for recreation, we probably need to know what days of the week are busiest.

We can answer this question using the same basic approach we used to answer the previous questions, but we will need a little bit of help. Since days of the week occur on different calendar days every year, we need a way to find out, for any given date in 2019, what day of the week that date fell on.

Python has a couple of tools that can help us with this: - `parse` is a function from the `dateutil` library that can turn a date that is formatted as a string into a "datetime object", which is a format that Python understands as a real date, not just a series of characters. - `weekday` is a function from the `datetime` library that can tell you, for any valid datetime object, what day of the week that date fell on.

[42]:
```
from datetime import datetime #yes, I know this seems redundant
from dateutil import parser
```

Let's try out formatting one of our strings as a datetime object.

[43]:
```
#here's a random date from our dataset, in its native string format
print(raw_data[234]['date'])
```

```
2019-01-10T19:00:00.000
```

[44]:
```
#now let's use the parse function to turn that into datetime format, which␣
↪Python can work with
parser.parse(raw_data[234]['date'])
```

[44]:
```
datetime.datetime(2019, 1, 10, 19, 0)
```

That worked! We can save this datetime object in its own variable. That will let us do things with it (like find out what day of the week January 10th, 2019 was).

[45]:
```
my_date = parser.parse(raw_data[234]['date'])
print(my_date)
print(type(my_date))
```

```
2019-01-10 19:00:00
<class 'datetime.datetime'>
```

Now that we know how to parse strings into datetime objects, we can use the `datetime.weekday()` function to find out what day of the week it was. This function will return a value between 0 (Monday) and 6 (Sunday) for any valid date you give it.

[46]:
```
my_date.weekday() #starts at 0!
```

[46]:
```
3
```

Looks like January 10th was a Thursday (you can check your calendar to confirm this).

3.5.1 Answering Question 5

Now that you know how to find out the day of the week for any date in our `raw_data` dataset, you should be able to find out which weekday or weekend day sees the most traffic on the Burke Gilman. You'll be able to use loops, "if" statements, and dictionaries to calculate the totals for each day.

Before you start, think for a minute: what do you think the answer will be, based on your own experience or your prior knowledge? Do you think the Burke Gilman is used more on weekdays or weekends? Why? Are some weekdays busier than others? Why?

3.6 Question #5 Answer

Imported calendar to find the specific day of the week in English, not numerical value. Using the raw_data, I searched for the total traffic of each day and create a hash table for each day of the week and inputting the total traffic count for each day.

[53]:
```
import calendar
daily_traffic = {}
for day in raw_data:
    date = day['date'] #find the date
    traffic_total = int(day['bgt_north_of_ne_70th_total']) #total traffic count
    date_obj = parser.parse(date)
    week_day = date_obj.weekday() #getting the week day
    if week_day not in daily_traffic:
        daily_traffic[week_day] = traffic_total
    else:
        daily_traffic[week_day] += traffic_total #aggregate!

# find the max value
max_day = max(daily_traffic, key=daily_traffic.get)
day_of_the_week = calendar.day_name[max_day]
traffic_count = str(daily_traffic[max_day])

print("The day of the week with the most traffic on Burke Gilham was on " +␣
 ↪day_of_the_week + " with a traffic count of " + traffic_count + ".")
```

```
The day of the week with the most traffic on Burke Gilham was on Sunday with a
traffic count of 95462.
```

4 Congratulations!

You have now mastered manipulating timeseries data in Python. There are plenty of other techniques, tools, and time-saving tricks that you can learn to build on these skills, but many data scientists who use Python every day do this kind of work using the same basic approach you just learned.

4.1 Challenges: going further

Here are some additional questions that you now have the tools you need to answer, based on what you've done today:

- what day of the week is busiest for bikes? Is it the same as the busiest day for pedestrians?
- what month of the year is busiest? (aka do Seattlites really like to ride in the rain?)
- has the Burke Gilman gotten busier over time? (the dataset we have goes back to 2014!)
- do fewer people commute on the Burke Gilman when it's cold out? (hint: try combining this dataset with the dataset on road temperature over time
- do more people commute into Seattle in the mornings by bike on the Burke Gilman, or on the the Mountain to Sound Trail?

4.2 Answering Additional Challenges #1

Used the raw_data to find sort the data into days of the week and getting the total pedestrians and bikes, then comparing which is the busiest day for each type of travelling.

[48]:
```
daily_traffic = {}
for day in raw_data:
    # find the date
    date = day['date']
    # find the traffic count for each type of traffic
    bike_north = int(day['bike_north'])
    bike_south = int(day['bike_south'])
    bikes = bike_north + bike_south # total within bikes
    ped_north = int(day['ped_north'])
    ped_south = int(day['ped_south'])
    pedestrians = ped_north + ped_south # total within pedestrians
    date_obj = parser.parse(date)
    week_day = date_obj.weekday()
    if week_day not in daily_traffic:
        daily_traffic[week_day] = {'bikes': bikes, 'pedestrians': pedestrians}
    else:
        daily_traffic[week_day]['bikes'] += bikes
        daily_traffic[week_day]['pedestrians'] += pedestrians

# finding the max key with the given dictionary and key selector
def max_key(data, query):
    max_date = max(data, key= lambda k: data[k][query])
    day_of_the_week = str(calendar.day_name[max_date])
    traffic_data = data[max_date][query]
    print("The busiest day for {query} on the Burke Gilham trail was usually on␣
 ↪a {date} with a {query} count of {traffic}.".format(query=query,␣
 ↪date=day_of_the_week, traffic=traffic_data))

print(daily_traffic, "\n")
max_key(daily_traffic, 'bikes')
max_key(daily_traffic, 'pedestrians')
```

```
{1: {'bikes': 48084, 'pedestrians': 18041}, 2: {'bikes': 46794, 'pedestrians':
17441}, 3: {'bikes': 45842, 'pedestrians': 17298}, 4: {'bikes': 44099,
'pedestrians': 17081}, 5: {'bikes': 66776, 'pedestrians': 26025}, 6: {'bikes':
66237, 'pedestrians': 29225}, 0: {'bikes': 50160, 'pedestrians': 18629}}

The busiest day for bikes on the Burke Gilham trail was usually on a Saturday
with a bikes count of 66776.
The busiest day for pedestrians on the Burke Gilham trail was usually on a
Sunday with a pedestrians count of 29225.
```

4.3 Answering Additional Challenges #2

The following code defines the finder(), which is a method that searches through the given dataset to find the key, counts the total traffic, then returns a traffic_data set. The max method is then used to find the highest value traffic_count within the keys. The query is any date selector: %d, %m, and %y.

[49]:
```
# accepts a dictionary and selector to find data for specific time values
def finder(data, query):
    traffic_data = {}
    for day in data:
        date = day['date']
        date_obj = parser.parse(date)
        key = int(date_obj.strftime(query)) # get the specific time value passed
        total_traffic = int(day['bgt_north_of_ne_70th_total']) # get total␣
 ↪traffic
        if key not in traffic_data:
            traffic_data[key] = total_traffic
        else:
            traffic_data[key] += total_traffic
    return traffic_data

monthly_traffic = finder(raw_data, '%m')

# get the max val from dictionary
max_month = max(monthly_traffic, key=monthly_traffic.get)
month_name = calendar.month_name[max_month]
traffic_count = str(monthly_traffic[max_month])
print(monthly_traffic, '\n')
print("The busiest month of the year is {month}, with a total traffic count of␣
 ↪{traffic}.".format(month=month_name, traffic=traffic_count))
```

```
{1: 28687, 2: 12917, 3: 44293, 4: 40353, 5: 61586, 6: 63088, 7: 63743, 8: 64550,
9: 43789, 10: 34778, 11: 29779, 12: 24169}

The busiest month of the year is August, with a total traffic count of 64550.
```

4.4 Answering Additional Challenges #3

The code below uses the finder() method defined above to retrieve a hash list of yearly traffic data. However, the dataset only includes 2019 from the API request, and the 'bgt_north_of_ne_70th_total' is not an applicable selector anymore. The code should work if all of the data was consistent in naming, and properly populating columns.

[50]:
```
# using the finder method to get data for yearly values
yearly_traffic = finder(raw_data, '%y')
max_year = max(yearly_traffic, key=yearly_traffic.get) # find the max
print(yearly_traffic, '\n')
print("The year with the highest amount of traffic was on \'{year}".
 ↪format(year=max_year))
```

```
{19: 511732}

The year with the highest amount of traffic was on '19
```

4.5 Answering Additional Challenges #4

Imported statistics to get data on the mean temperature for each day. Got data from Road Weather Information Stations and sorted them into hourly data and retrieved the temperature. Each temperature is then added to a list, and the average is calculated from that array of temperatures. Then finding the lowest temperature and traffic hours to see whether there is a correlation between the two data points.

[54]:
```
from statistics import mean

# getting data from the api
api_endpoint = "https://data.seattle.gov/resource/egc4-d24i.json?"
api_parameters = "$where=(StationName='RooseveltWay_NE80thSt')"
api_request = requests.get(api_endpoint + api_parameters)
road_raw_data = api_request.json()

# going through the api data
station_data = {}
for item in road_raw_data:
    time = item['datetime'][11:13] + ":00:00" # getting the hour only value
    temp = int(float(item['airtemperature'])) # getting the temperature
    traffic = traffic_by_hour_bp[time]['bikes'] +␣
 ↪traffic_by_hour_bp[time]['pedestrians'] # find total traffic
    date_obj = parser.parse(item['datetime'])
    key = int(date_obj.strftime('%H'))
    if key not in station_data:
        station_data[key] = {'temps': [temp], 'traffic': traffic, 'average':␣
 ↪temp}
    else:
        station_data[key]['temps'].append(temp)
```

```
        station_data[key]['average'] = mean(station_data[key]['temps'])

# finding the minimum value from a certain dictionary with a given selector
def min_key(data, query):
    min_traffic_hour = min(data, key= lambda k: data[k][query])
    min_temp_mean = min(data, key= lambda k: data[k]['average'])
    traffic = data[min_traffic_hour][query]
    print("The minimum lowest average temperature happened around {time_temp}␣
 ↪while the lowest traffic occurs at {time_traffic}.".
 ↪format(time_temp=min_temp_mean, time_traffic=min_traffic_hour))

# creating a simple visualization for the collected data
print("Time, Temperature, Traffic")
for key in station_data.keys():
    print(key, station_data[key]['average'], station_data[key]['traffic'])
print('\n')

min_key(station_data, 'traffic')
```

```
Time, Temperature, Traffic
13 46 38328
14 45.43333333333333 39132
15 45 40675
16 45 46041
17 45 46997
18 45 34528
19 45 18987
20 45 9167
21 45 3789
22 44.144444444444446 1376
23 43 692
0 43 343


The minimum lowest average temperature happened around 23 while the lowest
traffic occurs at 0.
```

4.6 Answering Additional Challenges #5

Gathered data from the MTS Trail west of I-90 Bridge Bicycle and Pedestrian Counter and used the raw_data as well. The mountain traffic data was then sorted to gather the traffic count and created a hash list from the hours. The hours were limited to below 12 or above/equal to 0 to incorporate only the mornings. The data is then compared with the hourly traffic results for the trail.

[55]:
```
# api request
api_endpoint = "https://data.seattle.gov/resource/u38e-ybnc.json?"
api_request = requests.get(api_endpoint)
```

```
trail_west_raw = api_request.json()


mountain_data = {}
total_mountain_traffic = 0
for day in trail_west_raw: # going through the collected data
    date = day['date']
    traffic = int(day['bike_east']) + int(day['bike_west']) # traffic count
    date_obj = parser.parse(date)
    hour = int(date_obj.strftime('%H')) # get the hour value
    if hour not in mountain_data and (hour < 12 and hour >= 0):
        total_mountain_traffic += traffic
        mountain_data[hour] = traffic
    elif hour in mountain_data:
        mountain_data[hour] += traffic
        total_mountain_traffic += traffic

# compare with traffic_by_hour_bp dictionary to get bike traffic data
trail_traffic = 0
for hour in traffic_by_hour_bp:
    traffic = traffic_by_hour_bp[hour]['bikes']
    hour_val = int(hour[:2])
    if hour_val >= 0 and hour_val < 12:
        trail_traffic += traffic
print("The total bicycles on the trail during the morning is {trail}, while the␣
 ↪total bicycles on the MTS Trail is {mountain}".format(trail=trail_traffic,␣
 ↪mountain=total_mountain_traffic), '\n')
print("I believe there is a massive difference in the collected dataset, but␣
 ↪from the result, I'm concluding that the Burke Gilman Trail is used at a␣
 ↪much higher rate.")
```

```
The total bicycles on the trail during the morning is 130228, while the total
bicycles on the MTS Trail is 713

I believe there is a massive difference in the collected dataset, but from the
result, I'm concluding that the Burke Gilman Trail is used at a much higher
rate.
```

4.7 Other data.seattle.gov datasets that you can do timeseries with...

- Fremont bridge bicycle counter: https://data.seattle.gov/Transportation/Fremont-Bridge-Bicycle-Counter/65db-xm6k
- Spokane Street bridge bicycle counter: https://data.seattle.gov/Transportation/Spokane-St-Bridge-Bicycle-Counter/upms-nr8w
- Mountain to Sound trail bicycle + pedestrian counter: https://data.seattle.gov/Transportation/MTS-Trail-west-of-I-90-Bridge-Bicycle-and-Pedestri/u38e-ybnc

4.8 More complex datasets

- https://data.seattle.gov/Public-Safety/Terry-Stops/28ny-9ts8
- https://data.seattle.gov/Permitting/Building-Permits/76t5-zqzr