In this notebook, we will take a look at how to utilize Python to perform manipulation of tabular data, and then utilize a Google web API to retrieve data we need based on data we have.

Pandas is a useful Python package which allows us to handle large amounts of data in tables. It includes many useful functions for getting summary statistics of the data held in its tables, and for visualization. Numpy is a scientific package, containing useful classes and functions for manipulating quantitative and qualitative data. os is a package for dealing with file directory structures in a way that avoids typos from entering filepaths manually as strings, and which works across different operating systems.

Here, we import these 3 packages for use.

In [1]:
```
import pandas as pd
import numpy as np
import os
```

First, we open the csv file containing the latlon data of the points between which we would like to measure the distance.

In [2]:
```
file = os.path.join(os.pardir,"data","distancedummy.csv")
df=pd.read_csv(file)
```

The data is read into a pandas dataframe, using the alias 'pd' for pandas, and the 'read_csv' function of pandas. We can print out the dataframe, here named 'df', to see its contents. We can also point to a specific cell, and examine its data type. the function str() converts other value types to text strings, so that we can concatenate it with other strings.

As we can see here, the numeric columns have been automatically converted by Pandas from a text format to a numpy.float64, a numeric format.

In [3]:
```
print(df)
print()
print(str(df['homelat'][1]) + ': ' + str(type(df['homelat'][1])))
```

```
     id   homelat     homelon        mrtname    mrtlat      mrtlon
0   bob  1.271684  103.807672  Telok Blangah  1.270575  103.809731
1   tom  1.350142  103.935288       Tampines  1.353333  103.945078
2  lars  1.425065  103.834088         Yishun  1.429334  103.834966
3   ron  1.336155  103.698430        Pioneer  1.337614  103.697152

1.350142: <class 'numpy.float64'>
```

To calculate the distance between latlons, we use the haversine formula, which places the coordinates on a sphere, of radius 6367km. Here we define the function, where the distance returned is also in km.

In [4]:
```
from math import radians, cos, sin, asin, sqrt
def haversine(lon1, lat1, lon2, lat2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)
    """
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    km = 6367 * c
    return km
```

Next, apply the haversine formula to each row, to calculate the direct distance from their latlon to the nearest MRT station. Here we use a lambda function to apply the haversine formula, and specify the axis=1 to apply it to each row (the default if not specified is axis=0, which applies to each column)

The formula is directly applied to the dataframe, and assigned to the new column 'directdist'. We also round the obtained distance to 3 decimal places, to match the output we will be obtaining from Google's API later on.

In [5]:
```
df['directdist'] = df.apply(lambda person: round(haversine(person['homelon'],
person['homelat'], person['mrtlon'], person['mrtlat']),3), axis=1)
df
```

Out[5]:

	id	homelat	homelon	mrtname	mrtlat	mrtlon	directdist
0	bob	1.271684	103.807672	Telok Blangah	1.270575	103.809731	0.260
1	tom	1.350142	103.935288	Tampines	1.353333	103.945078	1.144
2	lars	1.425065	103.834088	Yishun	1.429334	103.834966	0.484
3	ron	1.336155	103.698430	Pioneer	1.337614	103.697152	0.216

Next, we want to calculate the walking distance from the home to MRT. To do this, we can use the Google Maps Distance Matrix API. However, since it has a 25,000 query per day limit, the actual calculation for the entire dataset will be performed on a separate script which limits the number of calls to the API made per day, which is to be run daily.

Nonetheless, we can examine how to make a call to such an API here using this dummy dataset.

To begin, you first have to obtain an API key from Google, at https://developers.google.com/maps/documentation/distance-matrix/ (https://developers.google.com/maps/documentation/distance-matrix/) This key acts as a password to allow you to make the calls to Google's service, and allows Google to track how many calls you are making. Consequently, using a second API key would allow you to bypass the limit, but this is against Google's Terms of Use.

In [6]:
```
gmdm_api_key = 'AIzaSyCN3CBDUR6Q0jV_cBhhKK9gES-IzqKCSEM'
```

To make the request to Google's API, we need to import the 'requests' package. Also, the Google Maps Distance Matrix API returns a JSON string response, so to manipulate JSON objects, we can import the 'json' package.

In [7]:
```
import requests
import json
```

To obtain the walking distance from google, we can write a function to call the API by supplying the relevant input parameters.

In [8]:
```
def google_distance_matrix(lat1, lon1, lat2, lon2, mode, units,
departure_time, key):
    api_url = 'https://maps.googleapis.com/maps/api/distancematrix/json?'
    params = []
    params.append('key=' + key)
    params.append('origins=' + str(lat1) + ',' + str(lon1))
    params.append('destinations=' + str(lat2) + ',' + str(lon2))
    params.append('mode=' + 'mode')
    params.append('units=' + units)
    params.append('departure_time=' + departure_time)

    full_request_url = api_url
    for param in params:
        full_request_url+=(param + '&')

    response = requests.get(full_request_url)
    return response.json()
```

For our purposes, the mode is 'walking', the units should be 'metric', and the departure_time doesn't matter since it's not contingent on public transport, so we will just set it to 'now'.

The value returned by the api includes the distance in meters, and estimated travel time in seconds. Since we only care about the distance, we will map that to a new column, much like what we did earlier with the direct distance.

In [9]:
```
df['walkingdist'] = df.apply(lambda person: google_distance_matrix(person['hom
elat'],person['homelon'],person['mrtlat'],person['mrtlon'],'walking','metric',
w',gmdm_api_key)['rows'][0]['elements'][0]['distance']['value']/1000, axis=1)
df
```

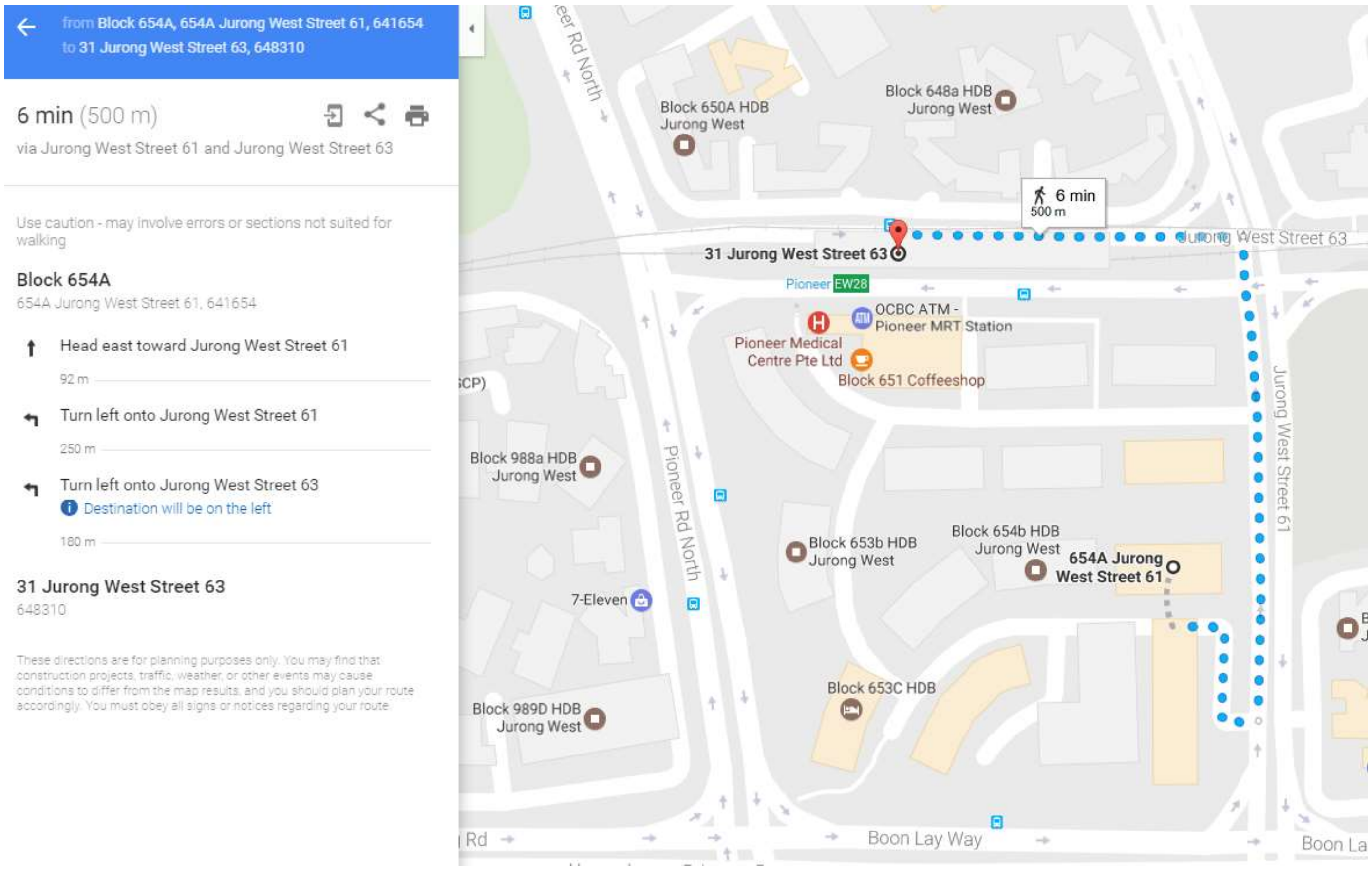
Out[9]:

	id	homelat	homelon	mrtname	mrtlat	mrtlon	directdist	walkingdist
0	bob	1.271684	103.807672	Telok Blangah	1.270575	103.809731	0.260	0.287
1	tom	1.350142	103.935288	Tampines	1.353333	103.945078	1.144	1.654
2	lars	1.425065	103.834088	Yishun	1.429334	103.834966	0.484	1.172
3	ron	1.336155	103.698430	Pioneer	1.337614	103.697152	0.216	1.469

And now we have the desired information in the data table!

Looking at the data, we can see that the walking distance can sometimes differ quite drastically from the direct distance, but it is almost always larger, as expected.

Inputting the origin and destination into Google Maps's web app in the browser, we can see why this is the case:



from Block 654A, 654A Jurong West Street 61, 641654
to 31 Jurong West Street 63, 648310

6 min (500 m)
via Jurong West Street 61 and Jurong West Street 63

Use caution - may involve errors or sections not suited for walking

Block 654A
654A Jurong West Street 61, 641654

Head east toward Jurong West Street 61
92 m

Turn left onto Jurong West Street 61
250 m

Turn left onto Jurong West Street 63
Destination will be on the left
180 m

31 Jurong West Street 63
648310

These directions are for planning purposes only. You may find that construction projects, traffic, weather, or other events may cause conditions to differ from the map results, and you should plan your route accordingly. You must obey all signs or notices regarding your route.

Pioneer Rd North
Block 650A HDB Jurong West
Block 648a HDB Jurong West
6 min
500 m
Jurong West Street 63
31 Jurong West Street 63
Pioneer EW28
OCBC ATM - Pioneer MRT Station
Pioneer Medical Centre Pte Ltd
Block 651 Coffeeshop
Block 988a HDB Jurong West
Jurong West Street 61
Block 653b HDB Jurong West
Block 654b HDB Jurong West
654A Jurong West Street 61
7-Eleven
Block 653C HDB
Block 989D HDB Jurong West
Boon Lay Way

The walking route to a destination is often circuitous, because it forces us to use major roads, while a pedestrian familiar with the area might often opt to cut through blocks or take smaller streets to reach the destination. Hence, we should consider whether or not to use Google's estimated walking distance.

Another thing to note is that even with this consideration, the walking distance from the Google Map Distance Matrix API still differs from the response from the web app, for unknown reasons. The image above depicts the result in the web app when the data from the row with id 'ron' is entered as the origin and destination. Whilst the Google API lists the walking distance as 1.469km, the web app cites it as 522m.

From here to the end of the notebook, I'll explain how we read the JSON output of the Google API, in cell 9 above.

To understand how we are handling the API response in the code above, we first need to understand what the response contains.

Each response returns a string of text, which looks something like this:

```
In [10]: sample_response_text='{"destination_addresses":["31 Jurong West Street 63, Sin
         gapore 648310"],"origin_addresses":["655A Jurong West Street 61, Block 655A, S
         ingapore 641655"],"rows":[{"elements":[{"distance":{"text":"1.5 km","value":14
         69},"duration":{"text":"5 mins","value":306},"status":"OK"}]}],"status":"OK"}'
```

This is pretty hard for a human to read, so we can use the 'indent' parameter of json.dumps, which reads a json object, to reveal the structure of the data.

Here I will first convert the json text string into a json object by loading it using json.loads, because in the function google_distance_matrix, we convert the response into a json object before returning it, by using response.json().

```
In [11]: sample_json = json.loads(sample_response_text)
         print(json.dumps(sample_json, indent=4))
```

```
{
    "origin_addresses": [
        "655A Jurong West Street 61, Block 655A, Singapore 641655"
    ],
    "status": "OK",
    "rows": [
        {
            "elements": [
                {
                    "distance": {
                        "value": 1469,
                        "text": "1.5 km"
                    },
                    "duration": {
                        "value": 306,
                        "text": "5 mins"
                    },
                    "status": "OK"
                }
            ]
        }
    ],
    "destination_addresses": [
        "31 Jurong West Street 63, Singapore 648310"
    ]
}
```

This example uses the data from the row with id 'ron'. Here we can see that after we supplied the latlon of ron's home and nearest MRT, Google converts them into addresses, then finds the walking distance between them.

Since the API actually allows us to submit more than 1 origin-destination pair at a time, the response also contains a list, 'rows'. Unfortunately, using multiple origin-destination pairs in a single request still counts as multiple requests, in terms of the 25,000 daily request limit.

The list 'rows' can contain many items, but in our case to keep things simple we have made the request in such a way that it only has one item. To see what this item is, we can select it and print it out. The first item in a list has the index '0', so we want the 0th item in the list 'rows'.

In [12]:
```
sample_request_row_item = sample_json['rows'][0]
print(json.dumps(sample_request_row_item, indent=4))
```

```
{
    "elements": [
        {
            "distance": {
                "value": 1469,
                "text": "1.5 km"
            },
            "duration": {
                "value": 306,
                "text": "5 mins"
            },
            "status": "OK"
        }
    ]
}
```

The data inside the first item in the list'rows' is also stored as a list, named 'elements'. Recall that since we can supply more than one origin and destination, Google's API actually returns the distance between every origin and every destination.

The first item in the list 'rows' holds the distance from the first origin to every destination, and the second item in 'rows' holds the distance from the second origin to every destination, and so forth.

Each item in 'rows' contains another list named 'elements', and the first item in 'elements' contains the distance from that origin to the first destination; the second item in 'elements' contains the distance from that origin to the second destination, and so forth.

In this case, we have only one origin and one destination, so we will need to grab the first item inside 'elements', too.

In [13]:
```
sample_request_row_element_item = sample_json['rows'][0]['elements'][0]
print(json.dumps(sample_request_row_element_item, indent=4))
```

```
{
    "distance": {
        "value": 1469,
        "text": "1.5 km"
    },
    "duration": {
        "value": 306,
        "text": "5 mins"
    },
    "status": "OK"
}
```

Finally, we have the data we need. From this json object, we want the numeric value of the distance between the origin and destination, so we simply point to the 'distance' object inside, and get its 'value'. To get it in kilometers, we simply divide it by 1000.

```
In [14]: sameple_request_row_element_item = sample_json['rows'][0]['elements'][0]['dist
         ance']['value']/1000
         print(json.dumps(sameple_request_row_element_item, indent=4))
```

```
1.469
```

This concludes the explanation of the json manipulation part of the Google API request.