9

ANALYZING LOCATION DATA

Everything happens somewhere. That's why the location of an object can be just as important as its nonspatial attributes for the purposes of data analysis. In fact, spatial and nonspatial data often go hand in hand.

As an example, consider a ride-sharing app. Once you've ordered a ride, you might want to track the location of the car on a map in real time while it's heading to you. You might also want know some basic nonspatial information about the car and driver assigned to your order: the make and model of the car, the driver's rating, and so on.

In the last chapter, you saw how to work with location data to generate maps. In this chapter, you'll learn more about how to use Python to collect and analyze location data, and you'll see how to integrate both spatial and nonspatial data in your analysis. Throughout, we'll consider the example of a taxi management service, and we'll try to answer the central question of which cab should be assigned to a particular job.

Obtaining Location Data

The first step in performing a spatial analysis is to obtain location data for the objects of interest. In particular, this location data should take the form of *geographical coordinates* (*geo coordinates* for short); that is, latitude and longitude values. This coordinate system enables every location on the planet to be specified as a set of numbers, meaning the locations can be analyzed programatically. In this section, we'll consider ways to obtain the geo coordinates of both stationary and moving objects. This will demonstrate how our example taxi service might determine a customer's pick-up location, as well as the real-time locations of its various cabs.

Turning a Human-Readable Address into Geo Coordinates

Most humans think in terms of street names and building numbers rather than geo coordinates. That's why it's common for taxi services, food delivery apps, and the like to let users specify pick-up and drop-off locations as street addresses. Behind the scenes, however, many of these services convert human-readable addresses into the corresponding geo coordinates. That way the app can perform calculations with the location data, such as determining the nearest available cab to the specified pick-up location.

How do you convert from street addresses to geo coordinates? One way is to use Geocoding, an API provided by Google for this purpose. For that, you'll also need to get an API key using a Google Cloud account. For information about acquiring an API key, see *https://developers.google.com/maps/documentation/geocoding/get-api-key*. Details on the API's cost structure are available at *https://cloud.google.com/maps-platform/pricing*. As of this writing Google provides a $200 monthly credit to API users, which is enough for you to experiment with the code in this book.

To interact with the Geocoding API from a Python script, you'll need to use the googlemaps library. Install it using the `pip` command, as follows:

```
$ pip install -U googlemaps
```

You'll also need to get an API key for the Geocoding API, using a Google Cloud account. For information about acquiring an API key, see *https://developers.google.com/maps/documentation/geocoding/get-api-key*. Details on the API's cost structure are available at *https://cloud.google.com/maps-platform/pricing*. As of this writing Google provides a $200 monthly credit to API users, which is enough for you to experiment with the code in this book.

The following script illustrates a sample call to the Geocoding API using googlemaps. This call obtains the latitude and longitude coordinates corresponding to the address 1600 Amphitheatre Parkway, Mountain View, CA:

```
import googlemaps

gmaps = googlemaps.Client(key='YOUR_API_KEY_HERE')
address = '1600 Amphitheatre Parkway, Mountain View, CA'
```

```
geocode_result = gmaps.geocode(address)

print(geocode_result[0]['geometry']['location'].values())
```

In this script, you establish a connection to the API and send the address you want to convert. The API returns a JSON document with a nested structure. The geo coordinates are stored under the key `location`, which is a subfield of `geometry`. In the last line you access and print the coordinates, yielding the following output:

```
dict_values([37.422388, -122.0841883])
```

Getting the Geo Coordinates of a Moving Object

You now know how to obtain the geo coordinates of a fixed location via its street address, but how can you get the real-time geo coordinates of a moving object, such as a taxi? Some taxi services might use specialized GPS devices for this purpose, but we'll focus instead on a low-cost, easy-to-implement solution. All that's required is a smartphone.

Smartphones detect their location with built-in GPS sensors and can be tuned to share that information. Here, we'll look at how to collect smartphone GPS coordinates via the popular messaging app Telegram. Using the Telegram Bot API, you'll create a *bot*, an application that runs within Telegram. Bots are commonly used for natural language processing, but this one will collect and log the geolocation data of Telegram users who choose to share their data with the bot.

Setting Up a Telegram BOT

To create a Telegram bot, you'll need to download the Telegram app and create an account. Then follow these steps, using either a smartphone or a PC:

1. In the Telegram app, search for @BotFather. BotFather is a Telegram bot that manages all the other bots in your account.
2. On the BotFather page, click **Start** to see the list of commands that you can use to set up your Telegram bots.
3. Enter **`/newbot`** in the message box. You'll be prompted for a name and a username for your bot. Then you'll be given an authorization token for the new bot. Take note of this token: you'll need it when you program the bot.

After completing these steps, you can implement the bot with Python using the python-telegram-bot library. Install the library like so:

```
$ pip install python-telegram-bot -upgrade
```

The tools you'll need to program the bot are in the library's `telegram.ext` module. It's built on top of the Telegram Bot API.

Programming the Bot

Here, you use the `telegram.ext` module of the python-telegram-bot library to program the bot to listen for and log GPS coordinates:

```
from telegram.ext import Updater, MessageHandler, Filters
from datetime import datetime
import csv

❶ def get_location(update, context):
    msg = None
    if update.edited_message:
      msg = update.edited_message
    else:
      msg = update.message
  ❷ gps = msg.location
    sender = msg.from_user.username
    tm = datetime.now().strftime("%H:%M:%S")
    with open(r'/HOME/PI/LOCATION_BOT/LOG.CSV', 'a') as f:
      writer = csv.writer(f)
    ❸ writer.writerow([sender, gps.latitude, gps.longitude, tm])
    ❹ context.bot.send_message(chat_id=msg.chat_id, text=str(gps))

def main():
  ❺ updater = Updater('TOKEN', use_context=True)
  ❻ updater.dispatcher.add_handler(MessageHandler(Filters.location,
                                   get_location))
  ❼ updater.start_polling()
  ❽ updater.idle()

if __name__ == '__main__':
    main()
```

The `main()` function contains the common invocations found in a script implementing a Telegram bot. You start by creating an `Updater` object ❺, passing it your bot's authorization token (generated by BotFather). This object orchestrates the bot execution process throughout the script. You then use the `Dispatcher` object associated with the `Updater` to add a handler function called `get_location()` for incoming messages ❻. By specifying `Filters.location`, you add a filter to the handler so it will only be called when the bot receives messages that include the sender's location data. You start the bot by invoking the `start_polling()` method of the `Updater` object ❼. Because `start_polling()` is a non-blocking method, you also have to call the `Updater` object's `idle()` method ❽ in order to block the script until a message is received.

At the beginning of the script, you define the `get_location()` handler ❶. Within the handler, you store the incoming message as `msg`, then you extract the sender's location data using the message's `location` property ❷. You also log the sender's username and generate a string containing the current time. Then, using Python's `csv` module, you store all this information as a row in a CSV file ❸ at a location of your choice. You also transmit the location data back to the sender, so they know that their location has been received ❹.

Python for Data Science (Sample Chapter) © 2/25/22 by Yuli Vasiliev

Getting Data from the Bot

Run the script on an internet-connected machine. Once it's running, users can follow a few simple steps to start sharing their real-time location data with the bot:

1. Create a Telegram account.
2. In Telegram, tap the name of the bot.
3. Tap the Paperclip icon and select **Location** from the menu.
4. Choose **Share My Location For** and set how long Telegram will share live location data with the bot. Options include 15 minutes, 1 hour, or 8 hours.

The screenshot in Figure 9-1 shows how easy it is to share your real-time location in Telegram.

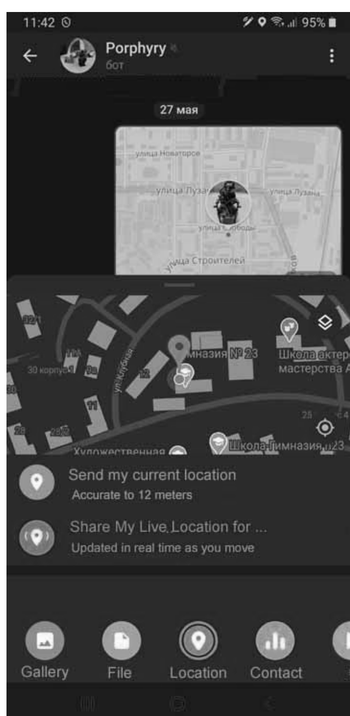

Figure 9-1: Sharing your smartphone's live location in Telegram

Once users start sharing their location data, the bot will start sending that data to a CSV file in the form of rows that might look as follows:

```
cab_26,43.602508,39.715685,14:47:44
cab_112,43.582243,39.752077,14:47:55
cab_26,43.607480,39.721521,14:49:11
cab_112,43.579258,39.758944,14:49:51
cab_112,43.574906,39.766325,14:51:53
cab_26,43.612203,39.720491,14:52:48
```

Python for Data Science (Sample Chapter) © 2/25/22 by Yuli Vasiliev

The first field in each row contains a username, the second and third fields contain the latitude and longitude of the user's location , and the fourth field contains a timestamp. For some tasks, such as finding the closest car to a certain pick-up location, you'd only need the latest row for each car. Other tasks, however, such as calculating the overall distance of a ride, would benefit from multiple rows of data for the given car, sorted by time.

Spatial Data Analysis with geopy and Shapely

Spatial data analysis boils down to answering questions about relationships: Which object is closest to a certain location? Are two objects in the same area? In this section, you'll answer these common spatial analysis questions using two Python libraries, geopy and Shapely, all within the context of our example taxi service.

Since geopy is designed for performing calculations based on geo coordinates, it's particularly suited for answering questions about distance. Meanwhile, Shapely specializes in defining and analyzing geometric planes, so it's ideal for determining whether an object falls within a certain area. As you'll see, both libraries can play a role in identifying the best cab for a given job.

Before you proceed, install the libraries as follows:

```
$ pip install geopy
$ pip install shapely
```

Finding the Closest Object

Continuing with our taxi service example, we'll look at how to use location data to identify the closest cab to a pick-up place. To start, you'll need some sample location data. If you deployed the Telegram bot discussed in the previous section, you may already have some data in the form of a CSV file. Here, you load the data into a pandas DataFrame so you can easily sort and filter it:

```
import pandas as pd
df = pd.read_csv("HOME/PI/LOCATION_BOT/LOG.CSV", names=['cab', 'lat',
                                                       'long', 'tm'])
```

If you didn't deploy a Telegram bot, you can instead create a list of tuples with some sample location data and load it into a DataFrame as follows:

```
import pandas as pd
locations = [
  ('cab_26',43.602508,39.715685,'14:47:44'),
  ('cab_112',43.582243,39.752077,'14:47:55'),
  ('cab_26',43.607480,39.721521,'14:49:11'),
  ('cab_112',43.579258,39.758944,'14:49:51'),
  ('cab_112',43.574906,39.766325,'14:51:53'),
  ('cab_26',43.612203,39.720491,'14:52:48')
```

```
]

df = pd.DataFrame(locations, columns =['cab', 'lat', 'long', 'tm'])
```

Either way, you'll get a DataFrame called df with columns for the cab ID, latitude, longitude, and timestamp.

NOTE *If you'd like to build your own set of sample location data to manipulate, a simple method is to look up latitude and longitude coordinates using Google Maps. When you right-click a location on a map, the location's latitude and longitude coordinates will be the first thing to show up in the menu.*

The DataFrame has multiple rows for each cab, but to identify the cab closest to a pick-up place you only need each cab's most recent location. You can filter out the unnecessary rows as follows:

```
latestrows = df.sort_values(['cab','tm'],ascending=False).drop_duplicates('cab')
```

Here, you sort the rows by the cab and tm fields in descending order. This operation groups the dataset by the cab column and puts the latest row for each cab first within its group. Then you apply the drop_duplicates() method to eliminate all but the first row for each cab. The resulting latestrows DataFrame looks as follows:

```
       cab        lat       long        tm
5   cab_26  43.612203  39.720491  14:52:48
3  cab_112  43.574906  39.766325  14:51:53
```

You now have a DataFrame with just the most recent location data for each cab. For the convenience of future computing, you next convert the DataFrame into a simpler Python structure, a list of lists. This way you'll be able to more easily append new fields to each row, such as a field for the distance between the cab and the pick-up place:

```
latestrows = latestrows.values.tolist()
```

The values property of latestrows returns a NumPy representation of the DataFrame, which you then convert to a list of lists using tolist().

You're now ready to calculate the distance between each cab and a pick-up place. You'll use the geopy library, which can accomplish this task with just a few lines of code. Here you use the distance() function from geopy's distance module to make the necessary calculations:

```
from geopy.distance import distance
pick_up = 43.578854, 39.754995

for i,row in enumerate(latestrows):
❶ dist = distance(pick_up, (row[1],row[2])).m
  print(row[0] + ':', round(dist))
  latestrows[i].append(round(dist))
```

Python for Data Science (Sample Chapter) © 2/25/22 by Yuli Vasiliev

For simplicity, you set the pick-up place by manually defining the latitude and longitude coordinates. In practice, however, you might use Google's Geocoding API to generate the coordinates automatically from a street address, as discussed earlier in the chapter. Next, you iterate over each row in your dataset and calculate the distance between each cab and the pick-up place by calling `distance()` ❶. This function takes two tuples containing latitude/longitude coordinates as arguments. By adding `.m`, you retrieve the distance in meters. For demonstration purposes, you'll print the result of each distance calculation; then you append it to the end of the row as a new field. The script produces the following output:

```
cab_112: 1015
cab_26: 4636
```

Clearly `cab_112` is closer, but how can you determine that programmatically? Use Python's built-in `min()` function, as follows:

```
closest = min(latestrows, key=lambda x: x[4])
print('The closest cab is: ', closest[0], ' - the distance in meters: ', closest[4])
```

You feed the data to `min()` and use a lambda function to evaluate its sorting order based on the item at index `4` of each row. This is the newly appended distance calculation. You then print the result in a human-readable format, yielding the following:

```
The closest cab is:  cab_112  - the distance in meters:  1015
```

In this example, you calculated the straight-line distance between each cab and the pick-up location. While this information can certainly be useful, real-world cars almost never drive in a perfectly straight line from one place to another. The layout of streets means that the actual distance a cab must drive to reach a pick-up location will be greater than the straight-line distance. With this in mind, next we'll look at a more reliable way to match pick-up locations to cabs.

Finding Objects in a Certain Area

Often, the right question to ask to determine the best cab for a job isn't "Which cab is the closest?" but rather "Which cab is in a certain area that includes the pick-up location?" This isn't just because the driving distance between two points is almost always greater than the straight-line distance between them. In practice, barriers such as rivers or railroad tracks often divide geographical areas into separate zones that are only connected at a limited number of points by bridges, tunnels, and the like. This can make straight-line distances highly misleading. Consider the example in Figure 9-2.

Figure 9-2: Obstacles like rivers can make distance measurements misleading.

As you can see, `cab_26` is spatially closest to the pick-up place in this scenario, but because of the river, `cab_112` will likely be able to get there faster. You can easily figure this out looking at the map, but how can you reach the same conclusion with a Python script? One way is to divide the area into a number of smaller *polygons*, or areas enclosed by a set of connected straight lines, and then check which cabs are within the same polygon as the pick-up location.

In this particular example, you should define a polygon that encompasses the pick-up location and has a boundary along the river. You can identify the polygon's boundaries manually through Google Maps: right-click several points that connect to form a closed polygon, and note each point's geo coordinates. Once you have the coordinates, you can define the polygon in Python using the Shapely library.

Here's how to create a polygon with Shapely and check whether a given point is inside that polygon:

```
❶ from shapely.geometry import Point, Polygon

  coords = [(46.082991, 38.987384), (46.075489, 38.987599), (46.079395,
             38.997684),(46.073822, 39.007297), (46.081741, 39.008842)]
❷ poly = Polygon(coords)
❸ cab_26 = Point(46.073852, 38.991890)
  cab_112 = Point(46.078228, 39.003949)
  pick_up = Point(46.080074, 38.991289)

❹ print('cab_26 within the polygon:', cab_26.within(poly))
  print('cab_112 within the polygon:', cab_112.within(poly))
  print('pick_up within the polygon:', pick_up.within(poly))
```

You first import two Shapely classes, `Point` and `Polygon` ❶, then you create a `Polygon` object using a list of latitude/longitude tuples ❷. This object represents the area north of the river, including the pick-up location. Next, you create several `Point` objects representing the locations of `cab_26`, `cab_112`, and the pick-up place, respectively ❸. Finally, you perform a series of spatial

queries to detect if a certain point is inside the polygon using Shapely's `within()` method ❹. As a result, the script should produce the following output:

```
cab_26 within the polygon: False
cab_112 within the polygon: True
pick_up within the polygon: True
```

EXERCISE: DEFINING TWO OR MORE POLYGONS

In the preceding section, you used a single polygon covering an area on the map. Now try defining two or more polygons covering adjacent urban areas divided by an obstacle such as a river. Obtain coordinates for these polygons using the Google map of your own city or town, or of any other urban area on the planet. You'll also need the coordinates of several points within those polygons to simulate the locations of some cabs and a pick-up place.

In your script, define the polygons with Shapely and group them into a dictionary, then group the points representing the cabs into another dictionary. Next, divide the cabs into groups based on which polygon they're located in. This can be accomplished using two loops: the outer one to iterate over the polygons and the inner one to iterate over the points representing the cabs, checking whether a point is within a polygon on each iteration of the inner loop. The following code fragment illustrates how this might be implemented:

```
--snip--
cabs_dict ={}
polygons = {'poly1': poly1, 'poly2': poly2}
cabs = {'cab_26': cab_26, 'cab_112': cab_112}
for poly_name, poly in polygons.items():
  cabs_dict[poly_name] = []
  for cab_name, cab in cabs.items():
    if cab.within(poly):
      cabs_dict[poly_name].append(cab_name)
--snip--
```

Next, you'll need to determine which polygon contains the pick-up place. Once you know it, you can select the corresponding list of cabs from the `cabs_dict` dictionary, using the name of the polygon as the key. Finally, use geopy to determine which cab within the chosen polygon is closest to the pick-up location.

Combining Both Approaches

So far, we've chosen the best cab for a pick-up by calculating linear distances and by finding the closest cab within a certain area. In fact, the most accurate way to find the right cab may be to use elements of both approaches. This is because it isn't necessarily safe to blindly exclude all the cabs that aren't in the same polygon as the pick-up location. A cab in an adjacent polygon may still be closest in terms of actual driving distance, even allowing for the possibility that the cab must get around a river or

other obstacle. The key is to consider the entry points between one polygon and another. Figure 9-3 shows how we might take this into account.

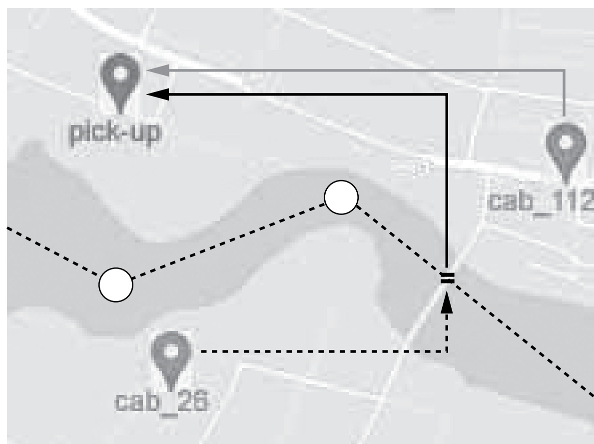

Figure 9-3: Using entry points to connect adjacent areas

The dotted line running across the middle of the figure represents the boundary dividing the area into two polygons: the one north of the river and the one south of the river. The equal sign laid on the bridge marks the entry point where cabs can move from one polygon to the other. For cabs in the polygon bordering that of the pick-up place, the distance to the pick-up place is composed of two intervals: the interval between the cab's current location and the entry point, and the interval between the entry point and the pick-up place.

To find the closest cab, you should therefore determine which polygon each cab is in and use that determination to decide how to calculate the distance from that cab to the pick-up location: either a direct straight-line distance if the cab is in the same polygon as the pick-up location, or the distance by way of the entry point if it's in an adjacent polygon. Here you make that calculation just for `cab_26`:

```
from shapely.geometry import Point, Polygon
from geopy.distance import distance

coords = [(46.082991, 38.987384), (46.075489, 38.987599), (46.079395,
           38.997684),(46.073822, 39.007297), (46.081741, 39.008842)]
❶ poly = Polygon(coords)
❷ cab_26 = Point(46.073852, 38.991890)
pick_up = Point(46.080074, 38.991289)
entry_point = Point(46.075357, 39.000298)

if cab_26.within(poly):
❸ dist = distance((pick_up.x, pick_up.y), (cab_26.x,cab_26.y)).m
else:
❹ dist = distance((cab_26.x,cab_26.y), (entry_point.x,entry_point.y)).m +
         distance((entry_point.x,entry_point.y), (pick_up.x, pick_up.y)).m

print(round(dist))
```

The script uses both Shapely and geopy. First you define a Shapely `Polygon` object including the pick-up location, as before ❶. You likewise define `Point` objects for the cab, the pick-up location, and the entry point ❷. Then you calculate the distance in meters with the help of geopy's `distance()` function. If the cab is within the polygon, you find the distance directly between the cab and the pick-up location ❸. If not, you first calculate the distance between the cab and the entry point and then the distance between the entry point and the pick-up place, summing them to get the total distance ❹. Here's the result:

```
1544
```

EXERCISE: FURTHER IMPROVING THE PICK-UP ALGORITHM

In the script we just discussed, you processed the location data related to a single cab to determine the distance between this cab and the pick-up place. Modify the script so that it can determine the distances between the pick-up place and each of several cabs. You'll need to group the points representing cabs into a list and then process this list in a loop, using the `if/else` statement from the preceding script as the loop's body. Then identify the closest cab to the pick-up place.

Combining Spatial and Nonspatial Data

So far in this chapter you've worked exclusively with spatial data, but it's important to realize that spatial analyses often need to factor in nonspatial data as well. For example, what's the use of knowing that a store is located within 10 miles of your current location if you don't know whether the item you want is currently in stock there? Or, turning back to our taxi example, what's the use of being able to determine the closest cab to a pick-up location if you don't know whether that cab is available or currently serving another order? In this section, we'll examine how to account for nonspatial data as part of a spatial analysis.

Deriving Nonspatial Attributes

Information about the current availability of cabs could be derived from a dataset containing ride orders. Once an order is assigned to a cab, this information might be placed in an `orders` data structure, where orders are listed as either open (in process) or closed (completed). According to this scheme, identifying only those orders that are open would tell you which cabs are unavailable to serve a new order. Here's how you could implement this logic in Python:

```
import pandas as pd
orders = [
  ('order_039', 'open', 'cab_14'),
```

```
  ('order_034', 'open', 'cab_79'),
  ('order_032', 'open', 'cab_104'),
  ('order_026', 'closed', 'cab_79'),
  ('order_021', 'open', 'cab_45'),
  ('order_018', 'closed', 'cab_26'),
  ('order_008', 'closed', 'cab_112')
]

df_orders = pd.DataFrame(orders, columns =['order','status','cab'])
df_orders_open = df_orders[df_orders['status']=='open']
unavailable_list = df_orders_open['cab'].values.tolist()
print(unavailable_list)
```

The `orders` list of tuples used in this example might be derived from a more complete dataset, such as a collection of all the orders opened within the last two hours, that includes additional information about each order (pick-up location, drop-off location, start time, end time, and so on). For simplicity, here the dataset has already been reduced to just the fields needed for the current task. You convert the list into a DataFrame, then filter it to include only the orders whose status is `open`. Finally, you convert the DataFrame into a list containing only the values from the `cab` column. This list of unavailable cabs looks as follows:

```
['cab_14', 'cab_79', 'cab_104', 'cab_45']
```

Armed with this list, you need to check the other cabs and determine which is the closest to the pick-up place. Append this code to the previous script:

```
from geopy.distance import distance
pickup = 46.083822, 38.967845
cab_26 = 46.073852, 38.991890
cab_112 = 46.078228, 39.003949
cab_104 = 46.071226, 39.004947
cab_14 = 46.004859, 38.095825
cab_79 = 46.088621, 39.033929
cab_45 = 46.141225, 39.124934
cabs = {'cab_26': cab_26, 'cab_112': cab_112, 'cab_14': cab_14,
        'cab_104': cab_104, 'cab_79': cab_79, 'cab_45': cab_45}
dist_list = []

for cab_name, cab_loc in cabs.items():
  if cab_name not in unavailable_list:
    dist = distance(pickup, cab_loc).m
    dist_list.append((cab_name, round(dist)))

print(dist_list)
print(min(dist_list, key=lambda x: x[1]))
```

For the purposes of the example, you manually define the geo coordinates of the pick-up place and all the cabs as tuples, and you send the coordinates of the cabs to a dictionary, where the keys are the cab names. Then

you iterate over the dictionary, and for each cab not in `unavailable_list`, you use geopy to calculate the distance between the cab and the pick-up place. Finally, you print the entire list of available cabs with their distances to the pick-up place, as well as just the closest cab, yielding the following output:

```
[('cab_26', 2165), ('cab_112', 2861)]
('cab_26', 2165)
```

In this case, `cab_26` is the closest available cab.

EXERCISE: FILTERING DATA WITH A LIST COMPREHENSION

In the preceding section, you filtered the `orders` list down to just a list of unavailable cabs by first converting `orders` to a DataFrame. Now try generating the `unavailable_list` list without pandas, using a list comprehension instead. With this approach, you can obtain the list of cabs assigned to currently open orders with a single line of code:

```
unavailable_list = [x[2] for x in orders if x[1] == 'open']
```

After this replacement, you won't need to change anything else in the rest of the script.

Joining Spatial and Nonspatial Datasets

In the previous example, you kept the spatial data (each cab's location) and the nonspatial data (which cabs were available) in separate data structures. Sometimes, however, it may be advantageous to combine spatial and non-spatial data in the same structure.

Consider that a cab may need to satisfy some other conditions apart from availability to be assigned to an order. For example, a client may need a cab with a baby seat. To find the right cab, you'll need to rely on a dataset that includes nonspatial information about the cabs as well as each cab's distance from the pick-up location. For the former, you may use a dataset that contains just two columns: the cab name and the presence of a baby seat. You create it here:

```
cabs_list = [
  ('cab_14',1),
  ('cab_79',0),
  ('cab_104',0),
  ('cab_45',1),
  ('cab_26',0),
  ('cab_112',1)
]
```

Cabs with a `1` in the second column have a baby seat. Next you convert the list to a DataFrame. You also create a second DataFrame from

dist_list, the list of available cabs and their distances to the pick-up place that you generated in the preceding section:

```
df_cabs = pd.DataFrame(cabs_list, columns =['cab', 'seat'])
df_dist = pd.DataFrame(dist_list, columns =['cab', 'dist'])
```

You now merge these DataFrames based on the cab column:

```
df = pd.merge(df_cabs, df_dist, on='cab', how='inner')
```

You use an inner join, meaning only cabs included in both df_cabs and df_dist make it into the new DataFrame. In practice, since df_dist contains only cabs that are currently available, this excludes unavailable cabs from the result set. The merged DataFrame now includes both spatial data (each cab's distance to the pick-up place) and nonspatial data (whether or not each cab has a baby seat):

```
       cab  seat  dist
0   cab_26     0  2165
1  cab_112     1  2861
```

You convert the DataFrame into a list of tuples, which you then filter, leaving only the rows where the seat field is set to 1:

```
result_list = list(df.itertuples(index=False,name=None))
result_list = [x for x in result_list if x[1] == 1]
```

You use the DataFrame's itertuples() method to convert each row into a tuple, then you wrap the tuples into a list with the list() function.

The final step is to determine the row with the lowest value in the distance field, which is identified by index 2:

```
print(min(result_list, key=lambda x: x[2]))
```

Here's the result:

```
('cab_112', 1, 2861)
```

Compare this to the result shown at the end of the previous section. As you can see, the need for a baby seat led us to choose a different cab for the job.

Summary

Using the real-world example of a taxi service, this chapter illustrated how you can perform spatial data analyses. To start with, you looked at an example of turning a human-readable address into geo coordinates using Google's Geocoding API and the googlemaps Python library. Then you learned to use a Telegram bot to collect location data from smartphones. Next, you

Python for Data Science (Sample Chapter) © 2/25/22 by Yuli Vasiliev

used the geopy and Shapely libraries to perform fundamental geospatial operations, such as measuring the distance between points and determining if points are within a certain area. With the help of these libraries, built-in Python data structures, and pandas DataFrames, you designed an application to identify the best cab for a given pick-up, based on various spatial and nonspatial criteria.