```
In [ ]: # Initialize Otter
        import otter
        grader = otter.Notebook("lab03.ipynb")
```

1 Lab 3: Data Cleaning and EDA

In this lab you will be working on visualizing a dataset from the City of Berkeley containing data on calls to the Berkeley Police Department. Information about the dataset can be found at this link.

Due Date: Saturday, July 2, 11:59 PM PT.

1.0.1 Content Warning

This lab includes an analysis of crime in Berkeley. If you feel uncomfortable with this topic, **please contact your GSI or the instructors.**

1.0.2 Collaboration Policy

Data science is a collaborative activity. While you may talk with others about this assignment, we ask that you **write your solutions individually**. If you discuss the assignment with others, please **include their names** in the cell below.

Collaborators: *list names here*

1.1 Setup

In this lab, we'll perform Exploratory Data Analysis and learn some preliminary tips for working with matplotlib (a Python plotting library). Note that we configure a custom default figure size. Virtually every default aspect of matplotlib can be customized.

Collaborators: *list names here*

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

```
import zipfile
import matplotlib
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (12, 9)
```

2 Part 1: Acquire the Data

1. Obtain data To retrieve the dataset, we will use the `ds100_utils.fetch_and_cache` utility.

In [2]:
```
# just run this cell
from ds100_utils import download_lab3_data

dest_path = download_lab3_data()
print(f'Located at {dest_path}')
```

```
Using cached version that was downloaded (UTC): Sat Jun 11 02:17:51 2022
Located at data/lab03_data_sp22.zip
```

2. Unzip file We will now directly unzip the ZIP archive and start working with the uncompressed files.

In [4]:
```
# just run this cell
my_zip = zipfile.ZipFile(dest_path, 'r')
my_zip.extractall('data')
```

Note: There is no single right answer regarding whether to work with compressed files in their compressed state or to uncompress them on disk permanently. For example, if you need to work with multiple tools on the same files, or write many notebooks to analyze them—and they are not too large—it may be more convenient to uncompress them once. But you may also have situations where you find it preferable to work with the compressed data directly.

Python gives you tools for both approaches, and you should know how to perform both tasks in order to choose the one that best suits the problem at hand.

3. View files

Now, we'll use the `os` package to list all files in the `data` directory. `os.walk()` recursively traverses the directory, and `os.path.join()` creates the full pathname of each file.

If you're interested in learning more, check out the Python3 documentation pages for `os.walk` (link) and `os.path` (link).

We use Python 3 format strings to nicely format the printed variables `dpath` and `fpath`.

```
In [6]: # just run this cell
        import os

        for root, directories, filenames in os.walk('data'):
            # first, print out all directories
            for directory in directories:
                dpath = os.path.join(root, directory)
                print(f"d {dpath}")

            # next, print out all files
            for filename in filenames:
                fpath = os.path.join(root,filename)
                print(f"  {fpath}")
```

```
d data/secret
  data/Berkeley_PD_-_Calls_for_Service.csv
  data/ben_kurtovic.py
  data/dummy.txt
  data/hello_world.py
  data/lab03_data_sp22.zip
  data/secret/do_not_readme.md
```

In this Lab, we'll be working with the `Berkeley_PD_-_Calls_for_Service.csv` file. Feel free to check out the other files, though.

3 Part 2: Clean and Explore the Data

Let's now load the CSV file we have into a `pandas.DataFrame` object and start exploring the data.

```
In [7]: # just run this cell
        calls = pd.read_csv("data/Berkeley_PD_-_Calls_for_Service.csv")
        calls.head()
```

```
Out[7]:      CASENO                    OFFENSE                  EVENTDT EVENTTM  \
        0  21014296   THEFT MISD. (UNDER $950)  04/01/2021 12:00:00 AM   10:58
        1  21014391   THEFT MISD. (UNDER $950)  04/01/2021 12:00:00 AM   10:38
        2  21090494   THEFT MISD. (UNDER $950)  04/19/2021 12:00:00 AM   12:15
        3  21090204  THEFT FELONY (OVER $950)   02/13/2021 12:00:00 AM   17:00
        4  21090179              BURGLARY AUTO  02/08/2021 12:00:00 AM    6:20
```

```
            CVLEGEND  CVDOW                InInDbDate  \
0            LARCENY      4  06/15/2021 12:00:00 AM
1            LARCENY      4  06/15/2021 12:00:00 AM
2            LARCENY      1  06/15/2021 12:00:00 AM
3            LARCENY      6  06/15/2021 12:00:00 AM
4  BURGLARY - VEHICLE      1  06/15/2021 12:00:00 AM

                                      Block_Location              BLKADDR  \
0           Berkeley, CA\n(37.869058, -122.270455)                  NaN
1           Berkeley, CA\n(37.869058, -122.270455)                  NaN
2  2100 BLOCK HASTE ST\nBerkeley, CA\n(37.864908,…    2100 BLOCK HASTE ST
3  2600 BLOCK WARRING ST\nBerkeley, CA\n(37.86393…  2600 BLOCK WARRING ST
4  2700 BLOCK GARBER ST\nBerkeley, CA\n(37.86066,…   2700 BLOCK GARBER ST

       City State
0  Berkeley    CA
1  Berkeley    CA
2  Berkeley    CA
3  Berkeley    CA
4  Berkeley    CA
```

We see that the fields include a case number, the offense type, the date and time of the offense, the "CVLEGEND" which appears to be related to the offense type, a "CVDOW" which has no apparent meaning, a date added to the database, and the location spread across four fields. We can read more about each field from the City of the Berkeley's open dataset webpage.

Let's also check some basic information about this DataFrame using the `DataFrame.info` (documentation) and `DataFrame.describe` methods (documentation).

In [8]:
```
# df.info() displays
# name and type of each column,
# number of non-null entries, and
# size of dataframe
calls.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2632 entries, 0 to 2631
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   CASENO          2632 non-null   int64
 1   OFFENSE         2632 non-null   object
 2   EVENTDT         2632 non-null   object
 3   EVENTTM         2632 non-null   object
 4   CVLEGEND        2632 non-null   object
 5   CVDOW           2632 non-null   int64
 6   InDbDate        2632 non-null   object
 7   Block_Location  2632 non-null   object
 8   BLKADDR         2612 non-null   object
```

```
 9   City            2632 non-null   object
 10  State           2632 non-null   object
dtypes: int64(2), object(9)
memory usage: 226.3+ KB
```

Note that the `BLKADDR` column only has 2612 non-null entries, while the other columns all have 2632 entries. This is because the `.info()` method only counts non-null entries.

```
In [9]: calls.describe()
```

```
Out[9]:              CASENO        CVDOW
        count  2.632000e+03  2632.000000
        mean   2.095978e+07     3.071049
        std    2.452665e+05     1.984136
        min    2.005721e+07     0.000000
        25%    2.100568e+07     1.000000
        50%    2.101431e+07     3.000000
        75%    2.102256e+07     5.000000
        max    2.109066e+07     6.000000
```

Notice that the functions above reveal type information for the columns, as well as some basic statistics about the numerical columns found in the DataFrame. However, we still need more information about what each column represents. Let's explore the data further in Question 1.

Before we go over the fields to see their meanings, the cell below will verify that all the events happened in Berkeley by grouping on the `City` and `State` columns. You should see that all of our data falls into one group.

```
In [10]: calls.groupby(["City","State"]).count()
```

```
Out[10]:                 CASENO  OFFENSE  EVENTDT  EVENTTM  CVLEGEND  CVDOW  InDbDate  \
         City     State
         Berkeley CA       2632     2632     2632     2632      2632   2632      2632

                         Block_Location  BLKADDR
         City     State
         Berkeley CA               2632     2612
```

When we called `head()` on the Dataframe `calls`, it seemed like `OFFENSE` and `CVLEGEND` both contained information about the type of event reported. What is the difference in meaning between the two columns? One way to probe this is to look at the `value_counts` for each Series.

```
In [11]: calls['OFFENSE'].value_counts().head(10)
```

```
Out[11]: THEFT MISD. (UNDER $950)    559
         VEHICLE STOLEN              277
         BURGLARY AUTO               218
         THEFT FELONY (OVER $950)    215
         DISTURBANCE                 204
         BURGLARY RESIDENTIAL        178
         VANDALISM                   166
         THEFT FROM AUTO             163
         ASSAULT/BATTERY MISD.       116
         ROBBERY                      90
         Name: OFFENSE, dtype: int64
```

```
In [12]: calls['CVLEGEND'].value_counts().head(10)
```

```
Out[12]: LARCENY                    782
         MOTOR VEHICLE THEFT        277
         BURGLARY - VEHICLE         218
         DISORDERLY CONDUCT         204
         BURGLARY - RESIDENTIAL     178
         VANDALISM                  166
         LARCENY - FROM VEHICLE     163
         ASSAULT                    150
         FRAUD                       93
         ROBBERY                     90
         Name: CVLEGEND, dtype: int64
```

It seems like `OFFENSE` is more specific than `CVLEGEND`, e.g. "LARCENY" vs. "THEFT FELONY (OVER $950)". If you're unfamiliar with the term, "larceny" is a legal term for theft of personal property.

To get a sense of how many subcategories there are for each `OFFENSE`, we will set `calls_by_cvlegend_and_offense` equal to a multi-indexed series where the data is first indexed on the `CVLEGEND` and then on the `OFFENSE`, and the data is equal to the number of offenses in the database that match the respective `CVLEGEND` and `OFFENSE`. As you can see, `calls_by_cvlegend_and_offense["LARCENY", "THEFT FROM PERSON"]` returns 8 which means there are 8 instances of larceny with offense of type "THEFT FROM PERSON" in the database.

```
In [13]: calls_by_cvlegend_and_offense = calls.groupby(["CVLEGEND", "OFFENSE"]).size()
         calls_by_cvlegend_and_offense["LARCENY", "THEFT FROM PERSON"]
```

```
Out[13]: 8
```

3.1 Question 1

In the cell below, set `answer1` equal to a list of strings corresponding to the possible values for `OFFENSE` when `CVLEGEND` is "LARCENY". You can type the answer manually, or you can create an expression that automatically extracts the names.

```
In [14]: answer1 = list(calls_by_cvlegend_and_offense['LARCENY'].index) # SOLUTION
```

```
In [ ]: grader.check("q1")
```

4 Part 3: Visualize the Data

4.0.1 Matplotlib demo

You've seen some `matplotlib` in this class already, but now we will explain how to work with the object-oriented plotting API mentioned in this matplotlib.pyplot tutorial useful. In matplotlib, plotting occurs on a set of Axes which are associated with a Figure. An analogy is that on a blank canvas (Figure), you choose a location to plot (`Axes`) and then fill it in (plot).

There are two approaches to labeling and manipulating figure contents, which we'll discuss below. Approach 1 is closest to the plotting paradigm of MATLAB, the namesake of matplotlib; Approach 2 is also common because many matplotlib-based packages (such as Seaborn) explicitly return the current set of axes after plotting data. Both are essentially equivalent, and at the end of this class you'll be comfortable with both.

Approach 1: matplotlib (or Seaborn) will auto-plot onto the current set of Axes or (if none exists) create a new figure/set of default axes. You can plot data using methods from `plt`, which is shorthand for the `matplotlib.pyplot` package. Then subsequent `plt` calls all edit the same set of default-created axes.

Approach 2:
After creating the initial plot, you can also use `plt.gca()` to explicitly get the current set of axes, and then edit those specific axes using axes methods. Note the method naming is slightly different!

As an example of the built-in plotting functionality of pandas, the following example uses `plot` method of the `Series` class to generate a `barh` plot type to visually display the value counts for `CVLEGEND`.

There are also many other plots that we will explore throughout the lab.

Side note: Pandas also offers basic functionality for plotting. For example, the `DataFrame` and `Series` classes both have a `plot` method, which uses matplotlib under the hood. For now we'll focus on matplotlib itself so you get used to the syntax, but just know that convenient Pandas plotting methods exist for your own future data science exploration.

Below, we show both approaches by generating a horizontal bar plot to visually display the value counts for `CVLEGEND`. See the `barh`documentation for more details.

```python
In [20]: # DEMO CELL: assign demo to 1 or 2.
         demo = ...

         calls_cvlegend = calls['CVLEGEND'].value_counts()

         if demo == 1:
             plt.barh(calls_cvlegend.index, calls_cvlegend) # creates figure and axes
             print(f"Demo {demo}: Using plt methods to update plot")
             plt.ylabel("Crime Category")               # uses most recently plotted axes
             plt.xlabel("Number of Calls")
             plt.title("Number of Calls by Crime Type")
         elif demo == 2:
             print(f"Demo {demo}: Using axes methods to update plot")
             plt.barh(calls_cvlegend.index, calls_cvlegend) # creates figure and axes
             ax = plt.gca()
             ax.set_ylabel("Crime Category")
             ax.set_xlabel("Number of Calls")
             ax.set_title("Axes methods: Number of Calls by Crime Type")
         else:
             print("Error: Please assign the demo variable to 1 or 2.")

         plt.show()
```

```
Error: Please assign the demo variable to 1 or 2.
```

4.0.2 An Additional Note on Plotting in Jupyter Notebooks

You may have noticed that many of our plotting code cells end with a semicolon `;` or `plt.show()`. The former prevents any extra output from the last line of the cell; the latter explicitly returns (and outputs) the figure. Try adding this to your own code in the following questions!

4.1 Question 2

Now it is your turn to make a plot using `matplotlib`. Let's start by transforming the data so that it is easier to work with.

The `CVDOW` field isn't named helpfully and it is hard to see the meaning from the data alone. According to the website linked at the top of this notebook, `CVDOW` is actually indicating the day that events happened. 0->Sunday, 1->Monday … 6->Saturday.

4.2 Question 2a

Add a new column `Day` into the `calls` dataframe that has the string weekday (eg. 'Sunday') for the corresponding value in CVDOW. For example, if the first 3 values of `CVDOW` are `[3, 6, 0]`, then the first 3 values of the `Day` column should be `["Wednesday", "Saturday", "Sunday"]`.

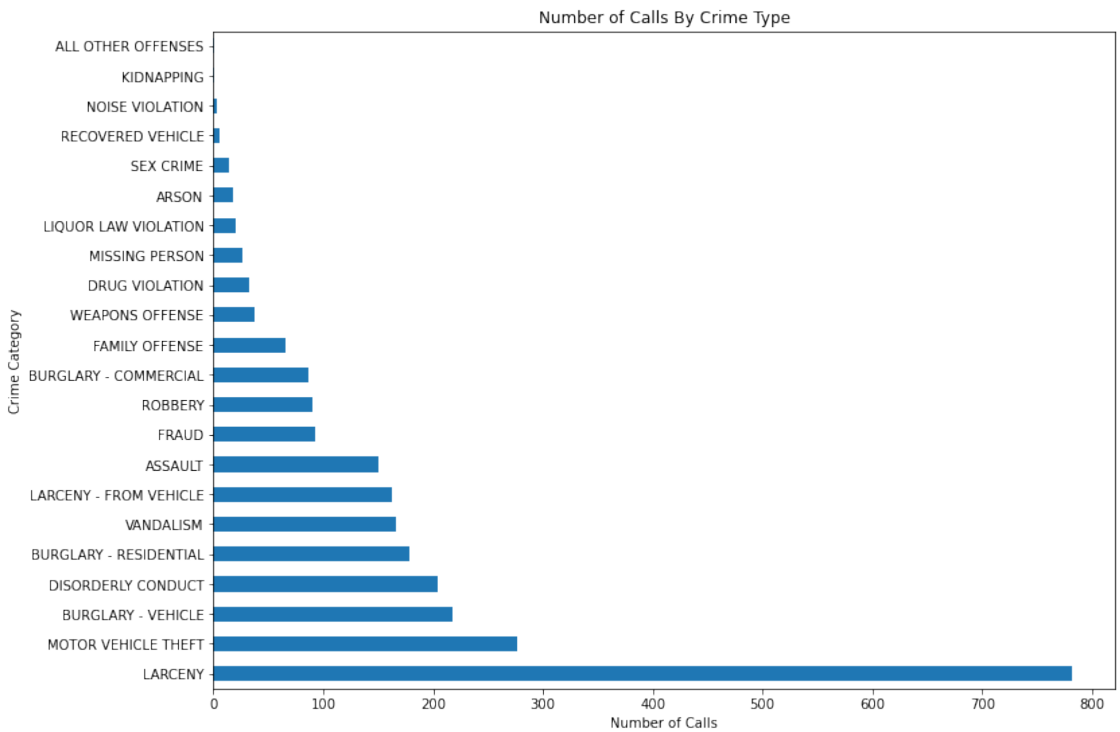
Hint: *Try using the Series.map function on* `calls["CVDOW"]`*. Can you assign this to the new column* `calls["Day"]`*?*

```
In [21]: days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
         day_indices = range(7)
         indices_to_days_dict = dict(zip(day_indices, days)) # Should look like {0:"Sunday", 1:"Monday"

         calls["Day"] = calls["CVDOW"].map(indices_to_days_dict) # SOLUTION
         # BEGIN SOLUTION NO PROMPT
         # challenge solution, commented out
         # we drop the column "Day" if it already exists, otherwise
         # duplicate "Day" columns are created
         # calls = pd.merge(calls.drop(columns="Day", errors="ignore"),
         #                  pd.DataFrame(days, columns=["Day"]),
         #                  left_on='CVDOW', right_index=True).sort_index()
         # END SOLUTION
```

```
In [ ]: grader.check("q2a")
```

```
In [24]: # just run this example cell
         ax = calls['CVLEGEND'].value_counts().plot(kind='barh')
         ax.set_ylabel("Crime Category")
         ax.set_xlabel("Number of Calls")
         ax.set_title("Number of Calls By Crime Type");
```



Challenge (OPTIONAL): You could also accomplish this part as a table left join with `pd.merge` (documentation), instead of using `Series.map`. You would need to merge `calls` with a new dataframe that just contains the days of the week. If you have time, try it out in the below cell!

```
In [25]: # scratch space for optional challenge
         dow_df = pd.DataFrame(days, columns=["Day"])

         ...
```

```
Out[25]: Ellipsis
```

4.3 Question 2b

Now let's look at the `EVENTTM` column which indicates the time for events. Since it contains hour and minute information, let's extract the hour info and create a new column named `Hour` in the `calls` dataframe. You should save the hour as an `int`.

Hint: *Your code should only require one line.* **Hint 2:** The vectorized `Series.str[ind]` performs integer indexing on an array entry.

```
In [26]: calls["Hour"] = calls['EVENTTM'].str.split(':').str[0].astype(int) # SOLUTION
         calls["Hour"]
```

```
Out[26]: 0       10
         1       10
         2       12
         3       17
         4        6
                 ..
         2627    12
         2628    15
         2629     0
         2630    18
         2631     2
         Name: Hour, Length: 2632, dtype: int64
```

```
In [ ]: grader.check("q2b")
```

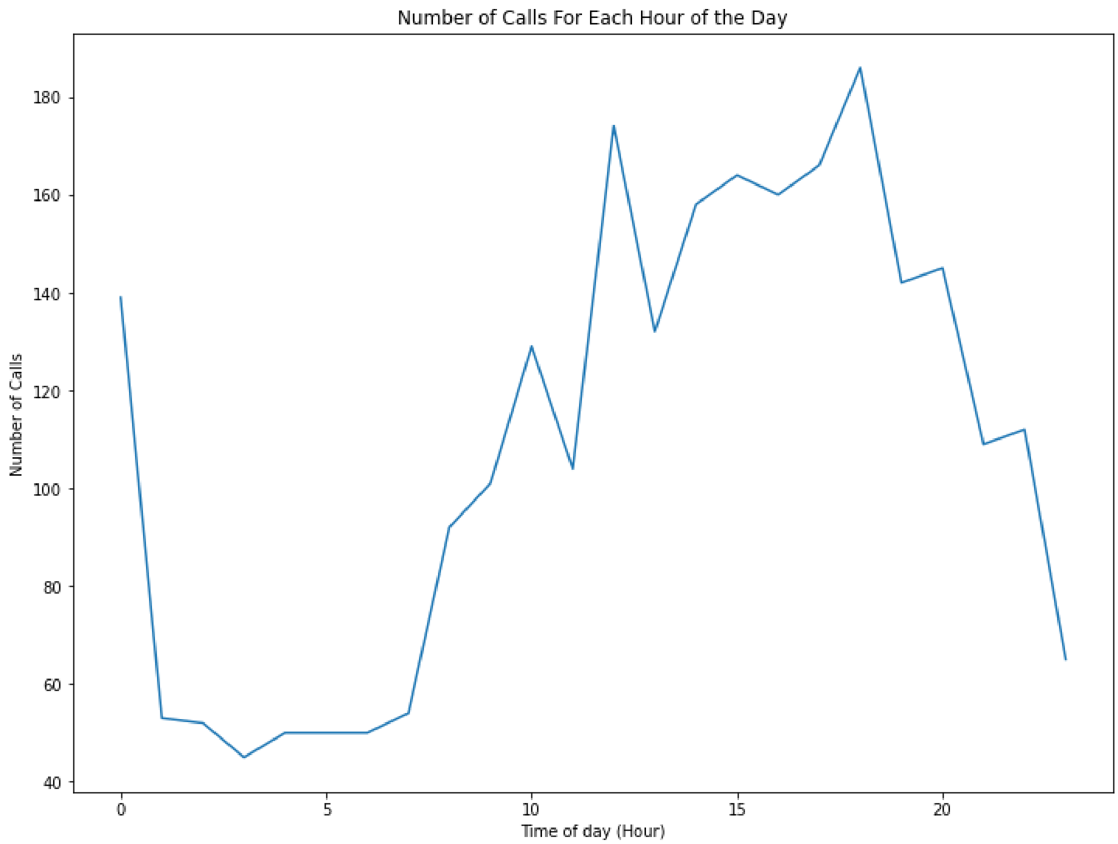
4.4 Question 2c

Using `matplotlib`, construct a line plot with the count of the number of calls (entries in the table) for each hour of the day **ordered by the time** (eg. `12:00 AM`, `1:00 AM`, ...). Please use the provided variable `hours` in your answer. Be sure that your axes are labeled and that your plot is titled.

Hint: Check out the `plt.plot` method in the matplotlib tutorial, as well as our demo above.

```
In [29]: hours = list(range(24))
         # BEGIN SOLUTION
         calls_hour = calls["Hour"].value_counts().sort_index()
         plt.plot(calls_hour.index, calls_hour)
         ax = plt.gca()
         ax.set_xlabel("Time of day (Hour)")
         ax.set_ylabel("Number of Calls")
         ax.set_title("Number of Calls For Each Hour of the Day");
         # END SOLUTION
```

```
# Leave this for grading purposes
ax_3d = plt.gca()
```



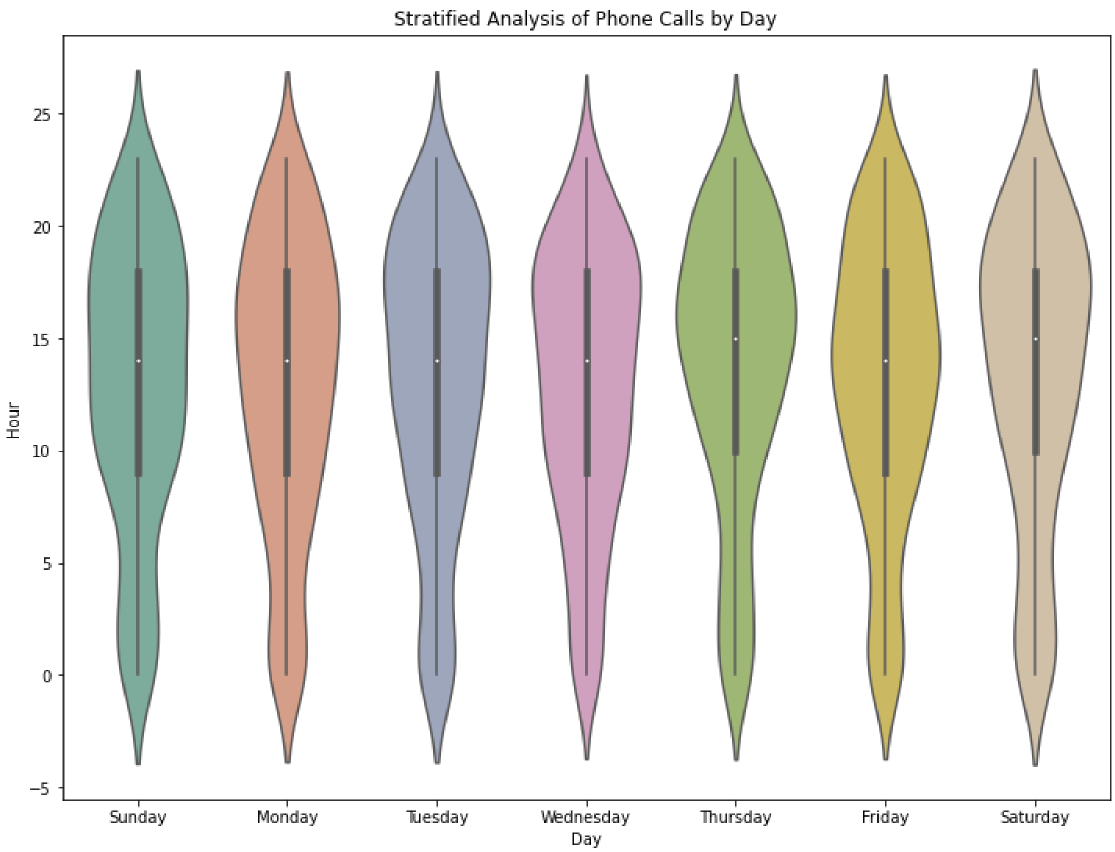

```
In [ ]: grader.check("q2c")
```

To better understand the time of day a report occurs we could **stratify the analysis by the day of the week.** To do this we will use **violin plots** (a variation of a **box plot**), which you will learn in more detail next week.

For now, just know that a violin plot shows an estimated distribution of quantitative data (e.g., distribution of calls by hour) over a categorical variable (day of the week). More calls occur in hours corresponding to the fatter part of each violin; the median hour of all calls in a particular day is marked by the white dot in the corresponding violin.

```
In [32]: # for now, just run this cell.
         # we will learn the seaborn visualization library next week.
```

```python
import seaborn as sns
ax = sns.violinplot(data=calls.sort_values("CVDOW"),
                    x="Day", y="Hour",
                    saturation=0.5, palette="Set2")
ax.set_title("Stratified Analysis of Phone Calls by Day");
```



4.5 Question 2d

Based on your line plot and our violin plot above, what observations can you make about the patterns of calls? Here are some dimensions to consider: * Are there more calls in the day or at night? * What are the most and least popular times? * Do call patterns vary by day of the week?

Type your answer here, replacing this text.

SOLUTION: In the simple line plot we see the standard pattern of limited activity early in the morning around here 6:00AM. The violin plot has no very clear patterns. However it does appear that weekends have more calls later into the night.

4.6 Question 3

In this last part of the lab, let's extract the GPS coordinates (latitude, longitude) from the `Block_Location` of each record.

In [33]:
```
# an example block location entry
calls.loc[4, 'Block_Location']
```

Out[33]:
```
'2700 BLOCK GARBER ST\nBerkeley, CA\n(37.86066, -122.253407)'
```

4.7 Question 3a: Regular Expressions

Use regular expressions to create a dataframe `calls_lat_lon` that has two columns titled `Lat` and `Lon`, containing the respective latitude and longitude of each record in `calls`. You should use the `Block_Location` column to extract the latitude and longitude coordinates.

Hint: Check out the `Series.str.extract` documentation.

In [34]:
```
calls_lat_lon = ...

# BEGIN SOLUTION NO PROMPT
# unnamed groups
calls_lat_lon = (
    calls['Block_Location']
    .str.extract("\((\d+\.\d+)\, (-\d+\.\d+)\)")
)
calls_lat_lon.columns = ['Lat', 'Lon']

# fancy version
calls_lat_lon = (
    calls['Block_Location']
    .str.extract(".*\((?P<Lat>\d*\.\d*)\, (?P<Lon>-?\d*\.\d*)\)", expand=True)
)
# END SOLUTION

calls_lat_lon.head(10)
```

Out[34]:
```
         Lat          Lon
0  37.869058  -122.270455
1  37.869058  -122.270455
2  37.864908  -122.267289
3  37.863934  -122.250262
4   37.86066  -122.253407
5  37.881957  -122.269551
6  37.867426  -122.269138
7  37.858116  -122.268002
8  37.868355  -122.274953
9  37.851491   -122.28563
```

In [ ]:
```
grader.check("q3a")
```

4.8 Question 3b: Join Tables

Let's include the GPS data into our `calls` data. In the below cell, use `calls_lat_lon` to add two new columns called `Lat` and `Lon` to the `calls` dataframe.

Hint: `pd.merge` (documentation) could be useful here. Note that the order of records in `calls` and `calls_lat_lon` are the same.

In [37]:
```
# BEGIN SOLUTION
# approach 1:
calls["Lat"] = calls_lat_lon["Lat"]
calls["Lon"] = calls_lat_lon["Lon"]

# approach 2:
# Remove Lat and Lon if they already existed before
calls.drop(["Lat", "Lon"], axis=1, inplace=True, errors="ignore")
# Join in the the latitude and longitude data
calls = calls.merge(calls_lat_lon, left_index=True, right_index=True)

# END SOLUTION
calls.sample(5)      # random rows
```

Out[37]:
```
        CASENO                     OFFENSE                 EVENTDT EVENTTM  \
1615  21023792  THEFT FELONY (OVER $950)  05/29/2021 12:00:00 AM   10:30
1722  21090048              BURGLARY AUTO  01/08/2021 12:00:00 AM   18:00
1540  21019622                  VANDALISM  05/04/2021 12:00:00 AM   13:00
479   21012020  THEFT MISD. (UNDER $950)  03/18/2021 12:00:00 AM   19:45
2220  21090621                  VANDALISM  05/19/2021 12:00:00 AM    8:00
```

```
                CVLEGEND  CVDOW                InDbDate  \
1615             LARCENY      6  06/15/2021 12:00:00 AM
1722  BURGLARY - VEHICLE      5  06/15/2021 12:00:00 AM
1540           VANDALISM      2  06/15/2021 12:00:00 AM
479              LARCENY      4  06/15/2021 12:00:00 AM
2220           VANDALISM      3  06/15/2021 12:00:00 AM

                                         Block_Location  \
1615  2426 MCGEE AVE\nBerkeley, CA\n(37.863593, -122…
1722  2200 BLOCK MARIN AVE\nBerkeley, CA\n(37.891755…
1540  1800 BLOCK 4TH ST\nBerkeley, CA\n(37.869888, -…
479   1900 BLOCK SHATTUCK AVE\nBerkeley, CA\n(37.873…
2220  2700 BLOCK SAN PABLO AVE\nBerkeley, CA\n(37.85…

                     BLKADDR      City State        Day  Hour        Lat  \
1615          2426 MCGEE AVE  Berkeley    CA   Saturday    10  37.863593
1722    2200 BLOCK MARIN AVE  Berkeley    CA     Friday    18  37.891755
1540       1800 BLOCK 4TH ST  Berkeley    CA    Tuesday    13  37.869888
479    1900 BLOCK SHATTUCK AVE  Berkeley    CA   Thursday    19  37.873687
2220  2700 BLOCK SAN PABLO AVE  Berkeley    CA  Wednesday     8  37.857714

             Lon
1615  -122.276751
1722  -122.269881
1540  -122.300618
479   -122.268616
2220  -122.288536
```

In [ ]:
```python
grader.check("q3b")
```

4.9 Question 3c: Check for Missing Values

It seems like every record has valid GPS coordinates:

In [41]:
```python
# just run this cell
# fraction of valid lat/lon entries
(~calls[["Lat", "Lon"]].isna()).mean()
```

Out[41]:
```
Lat    1.0
Lon    1.0
dtype: float64
```

However, a closer examination of the data reveals something else. Here's the first few records of our data again:

```
In [42]: calls.head(5)
```

```
Out[42]:      CASENO                    OFFENSE                 EVENTDT EVENTTM  \
         0  21014296  THEFT MISD. (UNDER $950)  04/01/2021 12:00:00 AM   10:58
         1  21014391  THEFT MISD. (UNDER $950)  04/01/2021 12:00:00 AM   10:38
         2  21090494  THEFT MISD. (UNDER $950)  04/19/2021 12:00:00 AM   12:15
         3  21090204  THEFT FELONY (OVER $950)  02/13/2021 12:00:00 AM   17:00
         4  21090179             BURGLARY AUTO  02/08/2021 12:00:00 AM    6:20

                    CVLEGEND  CVDOW               InDbDate  \
         0           LARCENY      4  06/15/2021 12:00:00 AM
         1           LARCENY      4  06/15/2021 12:00:00 AM
         2           LARCENY      1  06/15/2021 12:00:00 AM
         3           LARCENY      6  06/15/2021 12:00:00 AM
         4  BURGLARY - VEHICLE      1  06/15/2021 12:00:00 AM

                                              Block_Location                BLKADDR  \
         0             Berkeley, CA\n(37.869058, -122.270455)                    NaN
         1             Berkeley, CA\n(37.869058, -122.270455)                    NaN
         2  2100 BLOCK HASTE ST\nBerkeley, CA\n(37.864908,…    2100 BLOCK HASTE ST
         3  2600 BLOCK WARRING ST\nBerkeley, CA\n(37.86393…  2600 BLOCK WARRING ST
         4  2700 BLOCK GARBER ST\nBerkeley, CA\n(37.86066,…   2700 BLOCK GARBER ST

                City State       Day  Hour        Lat         Lon
         0  Berkeley    CA  Thursday    10  37.869058 -122.270455
         1  Berkeley    CA  Thursday    10  37.869058 -122.270455
         2  Berkeley    CA    Monday    12  37.864908 -122.267289
         3  Berkeley    CA  Saturday    17  37.863934 -122.250262
         4  Berkeley    CA    Monday     6   37.86066 -122.253407
```

There is another field that tells us whether we have a valid `Block_Location` entry per record—i.e., with GPS coordinates (latitude, longitude) that match the listed block location. What is it?

In the below cell, use the field you found to create a new dataframe, `missing_lat_lon`, that contains only the rows of `calls` that have invalid latitude and longitude data. Your new dataframe should have all the same columns of `calls`.

```
In [43]: missing_lat_lon = calls[calls["BLKADDR"].isna()] # SOLUTION
         missing_lat_lon.head()
```

```
Out[43]:        CASENO                    OFFENSE                 EVENTDT EVENTTM  \
         0    21014296  THEFT MISD. (UNDER $950)  04/01/2021 12:00:00 AM   10:58
         1    21014391  THEFT MISD. (UNDER $950)  04/01/2021 12:00:00 AM   10:38
```

```
215  21019124      BURGLARY RESIDENTIAL  04/30/2021 12:00:00 AM   10:00
260  21000289            VEHICLE STOLEN  01/01/2021 12:00:00 AM   12:00
633  21013362             BURGLARY AUTO  03/27/2021 12:00:00 AM    4:20

                   CVLEGEND  CVDOW                InDbDate  \
0                   LARCENY      4  06/15/2021 12:00:00 AM
1                   LARCENY      4  06/15/2021 12:00:00 AM
215  BURGLARY - RESIDENTIAL      5  06/15/2021 12:00:00 AM
260     MOTOR VEHICLE THEFT      5  06/15/2021 12:00:00 AM
633      BURGLARY - VEHICLE      6  06/15/2021 12:00:00 AM

                               Block_Location BLKADDR      City State       Day  \
0    Berkeley, CA\n(37.869058, -122.270455)     NaN  Berkeley    CA  Thursday
1    Berkeley, CA\n(37.869058, -122.270455)     NaN  Berkeley    CA  Thursday
215  Berkeley, CA\n(37.869058, -122.270455)     NaN  Berkeley    CA    Friday
260  Berkeley, CA\n(37.869058, -122.270455)     NaN  Berkeley    CA    Friday
633  Berkeley, CA\n(37.869058, -122.270455)     NaN  Berkeley    CA  Saturday

     Hour        Lat         Lon
0      10  37.869058  -122.270455
1      10  37.869058  -122.270455
215    10  37.869058  -122.270455
260    12  37.869058  -122.270455
633     4  37.869058  -122.270455
```

In [ ]:
```python
grader.check("q3c")
```

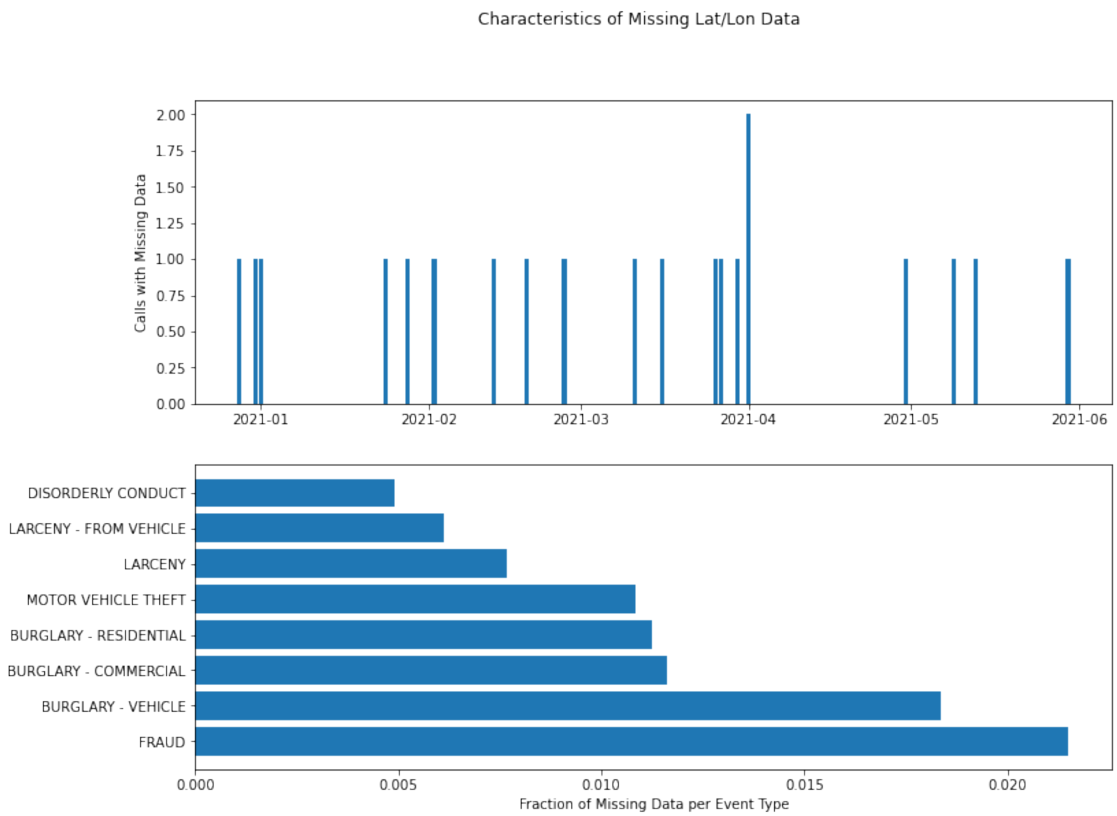
4.10 Question 3d: Check Missing Values

Now let us explore if there is a pattern to which types of records have missing latitude and longitude entries.

We've implemented the plotting code for you below, but read through it and verify you understand what we're doing (we've thrown in a bonus `plt.subplots()` call, documentation here).

In [46]:
```python
# just run this cell
missing_by_time = (pd.to_datetime(missing_lat_lon['EVENTDT'])
                   .value_counts()
                   .sort_index()
                  )
missing_by_crime = (missing_lat_lon['CVLEGEND']
                    .value_counts()
                    / calls['CVLEGEND'].value_counts()
                   ).dropna().sort_values(ascending=False)
```

```
fig, ax = plt.subplots(2)
ax[0].bar(missing_by_time.index, missing_by_time)
ax[0].set_ylabel("Calls with Missing Data")
ax[1].barh(missing_by_crime.index, missing_by_crime)
ax[1].set_xlabel("Fraction of Missing Data per Event Type")
fig.suptitle("Characteristics of Missing Lat/Lon Data")
plt.show()
```



Based on the plots above, are there any patterns among entries that are missing latitude/longitude data? The dataset information linked at the top of this notebook may also give more context.

Type your answer here, replacing this text.

SOLUTION: While some dates have more unlabeled data than others, it seems that a small percentage of Burglary and Fraud calls don't have GPS coordinates.

4.11 Question 3d: Explore

The below cell plots a map of phonecalls by GPS coordinates (latitude, longitude); we drop missing location data.

```
In [47]: # just run this cell
         import folium
         import folium.plugins

         SF_COORDINATES = (37.87, -122.28)
         sf_map = folium.Map(location=SF_COORDINATES, zoom_start=13)
         locs = calls.drop(missing_lat_lon.index)[['Lat', 'Lon']].astype('float').values
         heatmap = folium.plugins.HeatMap(locs.tolist(), radius=10)
         sf_map.add_child(heatmap)
```

```
Out[47]: <folium.folium.Map at 0x7f4309d406a0>
```

Based on the above map, what could be some **drawbacks** of using the location fields in this dataset to draw conclusions about crime in Berkeley? Here are some sub-questions to consider: * Is campus really the safest place to be? * Why are all the calls located on the street and often at intersections?

Type your answer here, replacing this text.

SOLUTION: This dataset is Berkeley Police crime data, not UC Police Department crime data. UC Berkeley has campus police, and that data is not included. Furthermore, calls are at intersections because the data only collects block-level granularity of locations (`BLOCKADDR` and `Block_Location`). While the location data may be useful for this type of broad human visualization, the data has missing values about a key portion of Berkeley (i.e., campus), and anyone using this dataset must acknowledge that location data granularity is block-level, and not address level.

Important: To make sure the test cases run correctly, click `Kernel>Restart & Run All` and make sure all of the test cases are still passing. Doing so will submit your code for you.

If your test cases are no longer passing after restarting, it's likely because you're missing a variable, or the modifications that you'd previously made to your DataFrame are no longer taking place (perhaps because you deleted a cell).

You may submit this assignment as many times as you'd like before the deadline.

You must restart and run all cells before submitting. Otherwise, you may pass test cases locally, but not on our servers. We will not entertain regrade requests of the form, "my code passed all of my local test cases, but failed the autograder".

4.12 Congratulations!

Congrats! You are finished with this lab.

To double-check your work, the cell below will rerun all of the autograder tests.

```
In [ ]: grader.check_all()
```

4.13 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

```
In [ ]: # Save your notebook first, then run this cell to export your submission.
        grader.export(pdf=False)
```