Chapter 14: Data Wrangling: Munging, Processing, and Cleaning

Once you have collected data, whether by scraping the Web, using an API, or grabbing it from a more organized source, the next step is to get the data into a format that is useable for analysis. This **wrangling** step can require a considerable amount of time and a large number of operations. Some of these operations include munging the data; removing rows, adding rows, changing the shape, discretizing, merging, separating, renaming, etc. Other operations might include cleaning the data; assuring that values are within expected ranges, removing outliers or incorrect values, normalizing, removing duplicates, etc.

There are many methods and tools that can be used for data wrangling including visually accessing portions of the data. To illustrate several such methods and tools, this chapter will first review basic file input and output (for txt and csv), will introduce Python pandas, will show a few contrived examples to further illustrate pandas methods, and will offer a Case Study (AirNow) that will progress through gathering, munging, and cleaning data from start to finish. Following chapters will continue into more advanced data processing, including binning (discretization), normalization, reductions, and preliminary visualization.

A Quick Review of File Input and Output

Text Files (.txt)

A text file, ending is .txt, can contain anything – as text.

Example 1: Creating, writing to, and closing a text file

```
TextFile="MyTextFile.txt"
File1=open(TextFile, "w")
DictText='{"ID":"D1234", "Firstname":"John", "Car":["BMW",
"Honda","Kia"]}\n'
File1.write(DictText)
DictText='{"ID":"D5555", "Firstname":"Benny", "Car":["Ford",
"Mazda","Kia"]}\n'
File1.write(DictText)
File1.close()
```

Example 2: Open a text file for reading

```
File1=open(TextFile, "r")
data=File1.read()
print("Entire File Contents ", data)
File1.seek(0)
data=File1.readline(11)
print("Read first line and first 11 chars ", data)
```

```
File1.seek(0)
data=File1.readlines()
print("Reads until the End of File (EOF) ", data)
```

Note that seek() will relocate the file pointer to a character location in the file. Therefore, seek(0) returns the file pointer to the start of the file.

Example 3: Open a text file for appending:

```
TextFile="MyTextFile.txt"
File1=open(TextFile, "a")
DictText='{"ID":"D7878", "Firstname":"Paul", "Car":["Chevy"]}\n'
File1.write(DictText)
DictText='{"ID":"D9199", "Firstname":"Alan",
"Car":["Mazda","Kia"]}\n'
File1.write(DictText)
File1.close()
File1=open(TextFile, "r")
data=File1.read()
print(data)
print(len(data))
print(type(data))
File1.close()
```

The length (len) of the data is a count of all characters in the text file. The type of data in this case is string (str). Because text files contain strings, string manipulation methods and operations can be applied to the contents of text files.

CSV (Comma Separated Values) Files

To use and manage csv files in Python, be sure to **import csv**. A csv file contains data that is separated by a **delimiter**, such as with commas or tabs.

Example 1: Creating, writing to, and closing a csv file

```
csvFile="MyCSVFile.csv"

File2=open(csvFile, "w", newline='')
CSVList=(["FirstName", "John", "Lastname", "Smith", "Car", "BMW",
"Zipcode", "20001"])
Fwriter=csv.writer(File2, delimiter=",")
Fwriter.writerow((CSVList[0],CSVList[2],CSVList[4],CSVList[6]))
Fwriter.writerow((CSVList[1],CSVList[3],CSVList[5],CSVList[7]))
```

```
File2.close()
```

In this example, the list called CSVList contains the data. The `writerow` method is used to write data to each row of the scv file using a comma to delimit each item written.

Example 2: Reading a csv file

```
File2=open(csvFile, newline='')
Freader=csv.reader(File2, delimiter=",")
for row in Freader:
    print(row)
File2.close()
```

Each row from a csv file is returned as a **list** of strings.

The output:

```
['FirstName', 'Lastname', 'Car', 'Zipcode']
['John', 'Smith', 'BMW', '20001']
```

Example 3: Reading and writing a csv file using DictReader and DictWriter

The csv module also offers dictionary-based methods called, DictWriter and DictReader. These methods will read and write csv files using dictionary objects.

```
import csv
columnNames=['Firstname', 'Car']
Filename="csvFile2.csv"
CSV1=open(Filename,"w", newline='')

writer=csv.DictWriter(CSV1, fieldnames=columnNames)
writer.writeheader()
writer.writerow({"Firstname":"Sally", "Car":"Honda"})
writer.writerow({"Firstname":"Pam", "Car":"Buick"})
CSV1.close()

with open(Filename) as CSV2:
    reader=csv.DictReader(CSV2)
    for line in reader:
           #The dict value for Car will be printed
        print(line["Car"])

CSV2.close()
```

The Output:

```
Honda
Buick
```

For more details on reading and writing csv files, https://docs.python.org/3/library/csv.html.

Using Python pandas and dataframes

Python pandas (http://pandas.pydata.org/) is an open source library that offers excellent data structures, such as the pandas **dataframe**, as well as a number of analysis tools. The pandas library is installed with Anaconda and can be used by including the following import statement:

```
import pandas as pd
```

The pandas web site (http://pandas.pydata.org/) notes the following quote:

> "Python has long been great for data munging and preparation, but less so for data analysis and modeling. *pandas* helps fill this gap, enabling you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language like R. Combined with the excellent IPython toolkit and other libraries, the environment for doing data analysis in Python excels in performance, productivity, and the ability to collaborate. *pandas* does not implement significant modeling functionality outside of linear and panel regression; for this, look to statsmodels and scikit-learn. More work is still needed to make Python a first class statistical modeling environment, but we are well on our way toward that goal."

The pandas library is often used in conjunction with the NumPy library. NumPy offers an array data structure. To include NumPy, use the following import statement:

```
import numpy as np
```

An Introduction to pandas data structures

The pandas Series

The **series** is a one-dimensional data structure that can be considered a "labeled" array. It can contain any data type, such as strings, lists, dictionaries, numerical values, and so on. In pandas, the word **axis** is used to denote the current "direction" or "dimension" of the data of interest. In a one-dimensional series, there is only one axis. However, in a two-dimensional structure (dataframe), axis=0 designates rows and axis=1 designates columns.

To create a pandas series:

```
MySeries=pd.Series(Mydata, index=indexvalue)
```

Example 1: Creating a Series

```
import numpy as np
import pandas as pd
#Create an array from 0 to 4
Mydata=np.arange(5)
#Note the index (row) value names
indexvalue=["C1", "C2", "C3", "C4", "C5"]
MySeries=pd.Series(Mydata, index=indexvalue)
print(MySeries)
```

The output:

```
C1    0
C2    1
C3    2
C4    3
C5    4
```

Example 2: Series and Dictionaries

```
MyDict={"Name":"Bob", "Age":29, "Degree":"MS"}
print(pd.Series(MyDict))
```

The Output:

```
Age        29
Degree     MS
Name      Bob
```

Example 3: Series, NumPy Arrays, and Functions

The pandas Series can behave very much like an array.

```
MyDict2={"Grade1":90.1, "Grade2":88.5, "Grade3":93.6}
MySeries=pd.Series(MyDict2)
print(MySeries)
print("Grade 2 is: ", MySeries[1])
print("The mean of the grades: ",MySeries.mean())
print("Grades plus 5 points added is:\n", MySeries+5)
print("Grade 1 is: ", MySeries.get("Grade1"))
```

The Output:

```
Grade1    90.1
Grade2    88.5
Grade3    93.6

Grade 2 is:  88.5

The mean of the grades:  90.73
Grades plus 5 points added is:
Grade1    95.1
Grade2    93.5
Grade3    98.6

Grade 1 is:  90.1
```

The pandas Dataframe

A **dataframe** is a two-dimensional structure (much like a spreadsheet or table), that is made up or rows and columns. Each column in a pandas dataframe may be a different type. The **index** represents the row labels and the **columns** represents the column labels. There are several methods for creating dataframes. The following examples illustrates a few options. For all examples, assume the following:

```
import pandas as pd
import numpy as np
```

Example 1: Using Series to create a dataframe

```
Gradebook={"Student1": pd.Series([89.3, 78.7, 92.2],
index=['Grade1', 'Grade2', 'Grade3']),
           "Student2": pd.Series([77.3, 83.4, 91.8],
index=['Grade1', 'Grade2', 'Grade3']),
           "Student3": pd.Series([97.1, 88.6, 98.5],
index=['Grade1', 'Grade2', 'Grade3'])
           }
GradeBookDF=pd.DataFrame(Gradebook)
print(GradeBookDF)
```

The Output:

```
           Student1  Student2  Student3
Grade1      89.3      77.3      97.1
Grade2      78.7      83.4      88.6
Grade3      92.2      91.8      98.5
```

Example 2: Creating an empty dataframe and adding a value

```
#Create an empty dataframe
Gradebook2 = pd.DataFrame(Gradebook, index=['G1', 'G2', 'G3'],
columns=['Bob Smith', 'Sandy Stern'])
print(Gradebook2)
#Fill in values
Gradebook2.ix["G1","Bob Smith"]=98.1
print(Gradebook2)
```

Note the use of the .ix (for index). This allows you to specify the row (in this case "G1") and the column (in this case "Bob Smith").

The Output:

```
      Bob Smith Sandy Stern
G1        NaN         NaN
G2        NaN         NaN
G3        NaN         NaN


      Bob Smith Sandy Stern
G1         98.1         NaN
G2         NaN          NaN
G3         NaN          NaN
```

Example 3: Adding a new column to a dataframe

```
#Create an empty dataframe
Gradebook2 = pd.DataFrame(Gradebook, index=['G1', 'G2', 'G3'],
columns=['Bob Smith', 'Sandy Stern'])
print(Gradebook2)
#Create a new column
Gradebook2["NewColumn"]="NaN"
print(Gradebook2)
```

The Output

```
   Bob Smith Sandy Stern NewColumn
G1        NaN         NaN       NaN
G2        NaN         NaN       NaN
G3        NaN         NaN       NaN
```

Example 4: Adding values to dataframes

In this example, the .ix is used to update the i^{th} index (row) and the column called "BobSmith". Notice that I have renamed the columns in the Gradebook2 dataframe so that they do not contain spaces. Rather than Bob Smith, the column name is BobSmith. This allows me to reference the column using the dot operator: `Gradebook2.BobSmith.`

```
import random
for i in range(len(Gradebook2.BobSmith)):
    Gradebook2.ix[i,"BobSmith"]=random.randint(50,100)
print(Gradebook2)
```

The Output:

```
   BobSmith SandyStern NewColumn
G1       91        NaN       NaN
G2       56        NaN       NaN
G3       63        NaN       NaN
```

Example 5: Converting a list of dictionaries into a dataframe and adding a column

```
MyDict=[{"Name":"Bob", "Age":29, "Degree":"MS"}, {"Name":"Rob",
"Age":34, "Degree":"PhD"}]
DictDF=pd.DataFrame.from_dict(MyDict)
DictDF.insert(2, 'NewColumn', [20007, 23604])
print(DictDF)
```

The `insert` method will add a column in location "2" with name, "NewColumn", and the noted contents.

The Output:

```
   Age Degree  NewColumn Name
0   29     MS      20007  Bob
1   34    PhD      23604  Rob
```

Example 6: Removing (dropping) rows and columns from a dataframe

```
MyDict=[{"Name":"Bob", "Age":29, "Degree":"MS"}, {"Name":"Rob",
"Age":34, "Degree":"PhD"}]
DictDF=pd.DataFrame.from_dict(MyDict)
```

```
DictDF.insert(2, 'NewColumn', [20007, 23604])
DictDF=DictDF.drop("Degree", axis=1)
print(DictDF)
```

In this case, the dataframe has a column called “Degree”. This code uses the “drop” method to remove this column by name. Notice that “axis=1” is included to specify that the axis of interest is “1”, which represents the column. Note that axis = 0 represents rows. The next code bit will remove row “0” (the first row) from the dataframe.

```
MyDict=[{"Name":"Bob", "Age":29, "Degree":"MS"}, {"Name":"Rob",
"Age":34, "Degree":"PhD"}]
DictDF=pd.DataFrame.from_dict(MyDict)
DictDF.insert(2, 'NewColumn', [20007, 23604])
DictDF=DictDF.drop(0)
print(DictDF)
```

Using pandas with csv files

The pandas dataframe is very versatile and there are many methods and operations related to the dataframe. One common use of pandas dataframes is to read data from a csv file into a dataframe.

In this next example, we will create our own small csv file and then will read it in as a dataframe.

Example 1: Reading a csv file into a dataframe

```
csvFile="MyCSVFile3.csv"
File2=open(csvFile, "w", newline='')
Header=(["FirstName", "Lastname", "Grade1", "Grade2", "Grade3"])
Data1=(["John", "Smith", 90.3, 87.5, 77.2])
Data2=(["Bob", "Benson", 88.8, 77.7, 66.6])
Fwriter=csv.writer(File2)
Fwriter.writerow(Header)
Fwriter.writerow(Data1)
Fwriter.writerow(Data2)
File2.close()

csvDataFrame=pd.read_csv(csvFile)
print(csvDataFrame)
```

The Output:

```
 FirstName Lastname  Grade1  Grade2  Grade3
0      John    Smith    90.3    87.5    77.2
1       Bob   Benson    88.8    77.7    66.6
```

This same concept can certainly be used for larger or preformed csv files.

Example 2: Reading and writing from dataframe to csv

```
import csv
import pandas as pd
import random

Gradebook2 = pd.DataFrame(Gradebook, index=['G1', 'G2', 'G3'],
columns=['BobSmith', 'SandyStern'])
for i in range(len(Gradebook2.BobSmith)):
    Gradebook2.ix[i, "BobSmith"]=random.randint(60,100)
    Gradebook2.ix[i, "SandyStern"]=random.randint(60,100)
print("Gradebok2 is:\n", Gradebook2)

#Write Dataframe Gradebook2 into csv
Gradebook2.to_csv("csvGradefile.csv", sep=",", header=True)

#Read dataframe from csv file
Gradebook3=pd.read_csv("csvGradefile.csv", index_col=0)
print("Gradebook3 is:\n", Gradebook3))
```

The Output:

```
Gradebok2 is:
    BobSmith SandyStern
G1       90         74
G2       69         73
G3       90         97

Gradebook3 is:
     BobSmith  SandyStern
G1        90          74
G2        69          73
G3        90          97
```

A note on pandas

The Python pandas library is very extensive and contains many further methods and options beyond the common ones presented above. Manipulating data is dependent on the nature of the data and the goals of the analysis. The following Case Study offers an example program that makes use of a number of the methods covered above, as well as the ideas and concepts related to munging and cleaning data.

Case Study: Getting, Munging, and Cleaning AirNow Historical Data for Ozone and PM2.5

For this Case Study illustration, I am interested in historical and current measures for Ozone and PM2.5.

The AirNow API URL for this query has the following format:

http://www.airnowapi.org/aq/observation/zipCode/historical/?format=text/csv&zipCode=20002&date=2014-09-03T00-0000&distance=25&API_KEY=D9AA91E7-070D-4221-867C-EFF5E0D8C2C7

Recall that Chapter 13 offers more detail on the AirNow API, getting a necessary KEY, and using the URLs to gather data via a post request.

Example 1: Use the AirNow API to gather data.

```
# AirNow API Python Code
# Author Ami Gates
# AirNowCh14Example.py

import urllib
from urllib.request import urlopen
import pandas as pd
import re
import numpy


def main():
    FileName="AirNowExample.csv"

#Header=['"DateObserved","HourObserved","LocalTimeZone","ReportingArea","StateC
ode","Latitude","Longitude","ParameterName","AQI","CategoryNumber","CategoryNam
e"']

    ziplist=["20007", "90210", "32605", "10001", "97202","33432"]
    datelist=["2004-01-01", "2006-01-01", "2008-01-01","2010-01-01","2012-01-
01","2014-01-01","2016-01-01"]

    GetAirNowData(FileName, ziplist, datelist)


def GetAirNowData(FileName, ziplist,datelist):

##http://www.airnowapi.org/aq/observation/zipCode/historical/?format=text/csv&z
ipCode=20002&date=2014-09-03T00-0000&distance=25&API_KEY=D9AA91E7-070D-4221-
867C-EFF5E0D8C2C7
    ZipDict={}
    #Create a new file - if exists - will delete
    File=open(FileName, "w")
    File.close()
    #----
    #Open for append
    File=open(FileName, "a")
```

```
    baseURL="http://www.airnowapi.org/aq/observation/zipCode/historical/?"
    miles=5

    for zipcode in ziplist:
        for date in datelist:
            zipURL=baseURL + urllib.parse.urlencode({
            'format': "text/csv",
            #'format':'application/json',
            'zipCode':zipcode,
            'date':date+'T00-0000',
            #yyyy-MM-ddThh-mmss"
            'distance':miles,
            'API_KEY':'D9AA91E7-070D-4221-867C-XXXXXXXXXXXXXXXXX'
            })
            #print(zipURL)

            response=urlopen(zipURL).read().decode('utf-8')
            responseCopy=response
            #Build Dict for zip code and city
            ZipDict[str(responseCopy)]=zipcode
            response=response+"\n"
            #response=urlopen(zipURL).read()
            File.write(response)



    File.close()
    #Create a file for the zipcode/city dict for later
    File=open("DictFile.txt","w")
    File.write(str(ZipDict))
    File.close()

main()
```

The Program above creates a new csv file called AirNowExample.csv. It is important to note that we can use a csv file here because the AirNow API allows for the option of getting results as text/csv. This is not always the case. The AirNow API also allows for the results to be returned as JSON or XML. As a word of warning, if you choose JSON, but run the API many times (as I have done here), the result will produce JSON code that may require extra steps to load and work with. Because this example will use Python **pandas** and dataframes, the election of csv is efficient and effective. Also note that you can load results into .txt files and then format them in any way that you wish.

Notice that the URL format had to be created perfectly (or the request will not work):

```
baseURL="http://www.airnowapi.org/aq/observation/zipCode/historical/?"
    miles=5
    for zip in ziplist:
        for date in datelist:
            zipURL=baseURL + urllib.parse.urlencode({
            'format': "text/csv",
            #'format':'application/json',
            'zipCode':zip,
            'date':date+'T00-0000',
            #yyyy-MM-ddThh-mmss"
            'distance':miles,
```

```
            'API_KEY':'D9AA91E7-070D-4221-867C-XXXXXXXXXXXXXXXXX'
        })
```

This is true whether using an API or performing a direct scrape with a GET or POST. The `urllib.parse.urlencode` method will properly encode the parameters of the post. That being said, if an error occurs at this point, print out the completed URL and see if it has the correct format.

Notice that this request is made within a double `for loop` which will request each *zip code* and each *date* combination per the `ziplist` and `datelist` variables (both lists) in `main()`. To purposefully create extra unclean data (to better illustrate this example), I also specifically used a zip code (32605) that will not return any data and so will generate missing values. While this data will be cleaner to start with than say straight HTML data directly scraped from the web, it will still require wrangling, munging, and cleaning. We will be taking text (due to the way we are collecting the data) and we will have to remove rows, remove empty values, remove a column, add a column, and re-organize the data. We will be using regular expressions, csv files, and Python **pandas** dataframes in this example. The program above will generate the following csv data file: "AirNowExample.csv" (Figure 1).

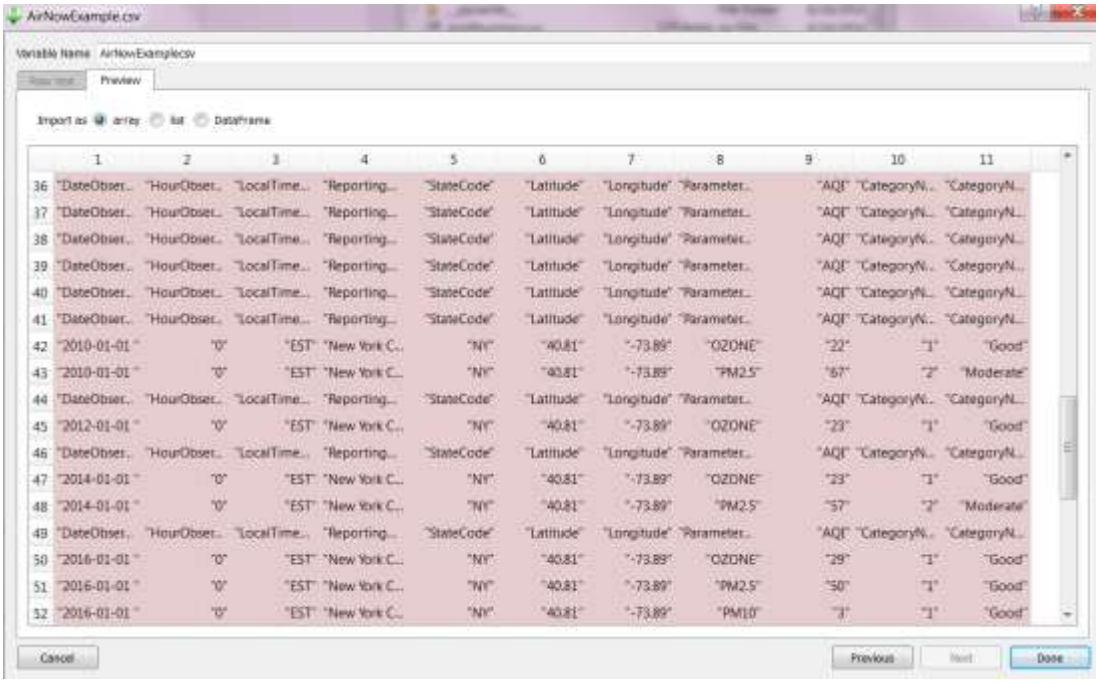


Figure1: A preview of the AirNowExample.csv dataset gathered using the AirNow API

Here, you can see that the column labels are repeated. This will have to be changed so that the column labels only appear one time at the top. Notice also that the zip code is not an element of the data, but I want it to be. I will have to add it along with a new column. In some rows, there is no data, these will have to be removed. These are either examples of zipcodes that returned no data or examples of dates that returned no data. As part of this example, we will also separate the dataset into two datasets, one for ozone and one for PM2.5.

Therefore, even after using an API and expecting clean and organized data, there are still a number of cleaning and munging steps that will have to take place. It will also be necessary to remove any repeated elements (if they exist), and to assure that all values fit within an expected range (no outliers or odd values).

Before the cleaning and munging can begin, however, the next step is to read this data into a format from which updates to it can be made. There are many options for this. Here, we will use **pandas** (Python Data Analysis Library)(http://pandas.pydata.org/). The pandas library is standard in Anaconda and can be used by importing pandas:

```
import pandas as pd
```

Creating the pandas DataFrame

During this next step, the goal is to get the data organized into columns and rows that make sense. From there, we can be to begin further clean and reorganize the data. By viewing a portion of the data, and because the code calls the API many times and places all the results into one file, I can see that many rows in the file are duplicates of the column names. The following code will remove these.

Removing Rows with pandas Drop

```
#pd is pandas and df is the dataframe
#Header=['"DateObserved","HourObserved","LocalTimeZone","Reportin
gArea","StateCode","Latitude","Longitude","ParameterName","AQI","
CategoryNumber","CategoryName"']
#The full set of code will follow – this is a subsection
df=pd.read_csv("AirNowExample.csv")
    Newdf=df
    #Remove the rows that repeat the headers
    for i in range(len(df.DateObserved)):
        if(df.ix[i,"DateObserved"]) == "DateObserved":
            Newdf=Newdf.drop(i)
```

Here, a dataframe called, df, is created using pandas (as pd). The results from the data scrape (shown above using the AirNow API, etc.) are read into the new dataframe from the csv file. Next, a new dataframe is created as a copy of the current dataframe. This allows us to test changes without affecting the original dataframe.

In this example, the row labels of the dataframe are integers, starting with “0”. The columns are labeled with the names listed in the Headers variable above. For example, the first column name is “DateObserved”.

The goal here is to remove (drop) every row in the dataframe that contains the string “DateObserved”, because this implies that the row is a duplicate. Non-duplicate rows contain an actual date in this column (see Figure 1 above).

The line of code,

```
if(df.ix[i,"DateObserved"]) == "DateObserved":
```

will determine if the given row with column name “DateObserved” contains the value “DateObserved”. Note that *.ix* stands for “index” and represents rows. As such, *df.ix [i, “DateObserved”]* represents the ith row with column name “DateObserved”.

If a row contains the string, “DateObserved”, the following code will remove it.

```
Newdf=Newdf.drop(i)
```

Figure 2 illustrates a partial image of the new data. Now, the duplicate rows that contain column names have been removed. Because of the file width, each row is continued below.

	DateObserved	HourObserved	LocalTimeZone	ReportingArea	StateCode
3	2010-01-01	0	EST	Metropolitan Washington	DC
4	2010-01-01	0	EST	Metropolitan Washington	DC
6	2012-01-01	0	EST	Metropolitan Washington	DC
7	2012-01-01	0	EST	Metropolitan Washington	DC
9	2014-01-01	0	EST	Metropolitan Washington	DC
10	2014-01-01	0	EST	Metropolitan Washington	DC
12	2016-01-01	0	EST	Metropolitan Washington	DC
13	2016-01-01	0	EST	Metropolitan Washington	DC
14	2016-01-01	0	EST	Metropolitan Washington	DC
19	2010-01-01	0	PST	NW Coastal LA	CA
20	2010-01-01	0	PST	NW Coastal LA	CA
22	2012-01-01	0	PST	NW Coastal LA	CA
23	2012-01-01	0	PST	NW Coastal LA	CA
25	2014-01-01	0	PST	NW Coastal LA	CA
27	2016-01-01	0	PST	NW Coastal LA	CA
28	2016-01-01	0	PST	NW Coastal LA	CA
40	2010-01-01	0	EST	New York City Region	NY
41	2010-01-01	0	EST	New York City Region	NY
43	2012-01-01	0	EST	New York City Region	NY
45	2014-01-01	0	EST	New York City Region	NY
46	2014-01-01	0	EST	New York City Region	NY
48	2016-01-01	0	EST	New York City Region	NY
49	2016-01-01	0	EST	New York City Region	NY
50	2016-01-01	0	EST	New York City Region	NY
55	2010-01-01	0	PST	Portland	OR
56	2010-01-01	0	PST	Portland	OR
58	2012-01-01	0	PST	Portland	OR
59	2012-01-01	0	PST	Portland	OR
61	2014-01-01	0	PST	Portland	OR
62	2014-01-01	0	PST	Portland	OR
64	2016-01-01	0	PST	Portland	OR
65	2016-01-01	0	PST	Portland	OR

	Latitude	Longitude	ParameterName	AQI	CategoryNumber	\
3	38.919	-77.013	OZONE	29	1	
4	38.919	-77.013	PM2.5	54	2	
6	38.919	-77.013	OZONE	32	1	

Figure 2: The updated AirNow dataset with duplicated column names removed.

Removing Columns

The next step will be to remove columns that are not necessary and do not contain information. In this case, the only column to remove is “HourObserved” because the value here is always “0”. The line of code to do this is:

```
Newdf=Newdf.drop('HourObserved', axis=1)
```

Note that, axis=1, specifies that **column axis** of the dataframe. The row axis is the default.

Dividing DataFrames and Creating Files

At this point, the data is looking nicer. As a next step, we will take the data and will create two dataframes, one for "ozone" and one for "PM2.5". We will re-index both. The code for that is the following:

```
#Create two dataframes. Ozone_df and PM25_df
#Assume the Newdf2 is the latest dataframe of the data
    Ozonedf=Newdf2
    PM25df=Newdf2

    for i in range(len(Newdf2.DateObserved)):
        if(Newdf2.ix[i,"ParameterName"]) != "PM2.5":
            PM25df=PM25df.drop(i)
        else:
            Ozonedf=Ozonedf.drop(i)

    #Re-index to start from 0
    Ozonedf=Ozonedf.reset_index(drop=True)
    PM25df=PM25df.reset_index(drop=True)

    #View the current results and dataframe
    File=open("ViewAirFilePM25.txt" , "w")
    File.write(str(PM25df))
    File.close()

    File=open("ViewAirFileOzone.txt" , "w")
    File.write(str(Ozonedf))
    File.close()
```

In this case, first create two new pandas dataframes. Recall that `Newdf2` is our current dataframe that contains our "cleaner" AirNow data.

```
Ozonedf=Newdf2
PM25df=Newdf2
```

Next, use the for loop to investigate each row of the current dataframe. If the "ParameterName" is not equal to "PM2.5", drop the row from the new PM25df. Otherwise drop it from the Ozonedf.

```
for i in range(len(Newdf2.DateObserved)):
      if(Newdf2.ix[i,"ParameterName"]) != "PM2.5":
            PM25df=PM25df.drop(i)
      else:
            Ozonedf=Ozonedf.drop(i)
```

This will result in two new dataframes. The first is `PM25df` and will contain only PM2.5 results. The other is `Ozonedf` and will contain only Ozone results. Next, re-index both and then place the results into two separate text files for viewing.

```
#Re-index to start from 0
    Ozonedf=Ozonedf.reset_index(drop=True)
    PM25df=PM25df.reset_index(drop=True)

    #View the current results and dataframe
    File=open("ViewAirFilePM25.txt" , "w")
    File.write(str(PM25df))
    File.close()

    File=open("ViewAirFileOzone.txt" , "w")
    File.write(str(Ozonedf))
    File.close()
```

Figure 3 below illustrate the current Ozone data file.

	DateObserved	LocalTimeZone	ReportingArea	StateCode	Latitude	\
0	2010-01-01	EST	Metropolitan Washington	DC	38.919	
1	2012-01-01	EST	Metropolitan Washington	DC	38.919	
2	2014-01-01	EST	Metropolitan Washington	DC	38.919	
3	2016-01-01	EST	Metropolitan Washington	DC	38.919	
4	2016-01-01	EST	Metropolitan Washington	DC	38.919	
5	2010-01-01	PST	NW Coastal LA	CA	34.0505	
6	2012-01-01	PST	NW Coastal LA	CA	34.0505	
7	2014-01-01	PST	NW Coastal LA	CA	34.0505	
8	2016-01-01	PST	NW Coastal LA	CA	34.0505	
9	2010-01-01	EST	New York City Region	NY	40.81	
10	2012-01-01	EST	New York City Region	NY	40.81	
11	2014-01-01	EST	New York City Region	NY	40.81	
12	2016-01-01	EST	New York City Region	NY	40.81	
13	2016-01-01	EST	New York City Region	NY	40.81	
14	2010-01-01	PST	Portland	OR	45.538	
15	2012-01-01	PST	Portland	OR	45.538	
16	2014-01-01	PST	Portland	OR	45.538	
17	2016-01-01	PST	Portland	OR	45.538	

	Longitude	ParameterName	AQI	CategoryNumber	CategoryName
0	-77.013	OZONE	29	1	Good
1	-77.013	OZONE	32	1	Good
2	-77.013	OZONE	29	1	Good
3	-77.013	OZONE	32	1	Good
4	-77.013	PM10	7	1	Good
5	-118.4566	OZONE	36	1	Good
6	-118.4566	OZONE	33	1	Good
7	-118.4566	OZONE	26	1	Good
8	-118.4566	OZONE	37	1	Good
9	-73.89	OZONE	22	1	Good
10	-73.89	OZONE	23	1	Good
11	-73.89	OZONE	23	1	Good
12	-73.89	OZONE	29	1	Good
13	-73.89	PM10	3	1	Good
14	-122.656	OZONE	31	1	Good
15	-122.656	OZONE	23	1	Good
16	-122.656	OZONE	3	1	Good

Figure3: The current Ozone Dataset from the original AirNow API scrape.

Fixing Outlier Data

At this point, we have good news and interesting news. The good news is that we now have two files, one with ozone data that is well organized and one with PM2.5 data that is well organized. The interesting news is that we have two aberrant data values for PM10.

In a large dataset, aberrant values are easy to miss and often must be tested for or avoided. There are many options for this, depending on how much information you have or can view about the data. Here,

the goal is to assure that only Ozone results are placed in the Ozone file and similarly that only PM2.5 results are placed in the PM25 file.

This will require a few more updates to the code.

To do this, update the `for loop` in the code above to the following:

```
for i in range(len(Newdf2.DateObserved)):
        if(Newdf2.ix[i,"ParameterName"]) != "PM2.5":
            PM25df=PM25df.drop(i)

        if(Newdf2.ix[i,"ParameterName"]) != "OZONE":
            Ozonedf=Ozonedf.drop(i)
```

This will assure that anything that is not PM2.5 will be dropped from the PM2.5 file and anything not OZONE will be dropped from the Ozone file.

At each point in the cleaning and munging process, it is a good idea to see what you have and to make sure it is what you want. At this point, I determined that my two new files, Ozone and PM25 were not formatted exactly as I wanted them. First, I want them to have a new column for zipcode that is correctly associated with the city. Next, I want them to be in dataframe format. I determined that using .txt files was not as helpful for my goals as using .csv files. I also determined that extra code is required to build a dictionary of zipcode/city associations so that these can then be added to the Ozone and PM25 dataframes (and files) later.

The Final Product

The following programming illustrates all the steps required to collect, reorganize, and clean the data. It begins with using the AirNow API to collect the data and then progresses through several steps to end up with two datasets (one for Ozone and one for PM2.5) that are ready to use. Follow this Program is a line by line discussion of the code.

```
# AirNow API Python Code
# Author Ami Gates
# AirNowCh14Example.py

import urllib
from urllib.request import urlopen
import pandas as pd
import re
import numpy


def main():
    FileName="AirNowExample.csv"
```

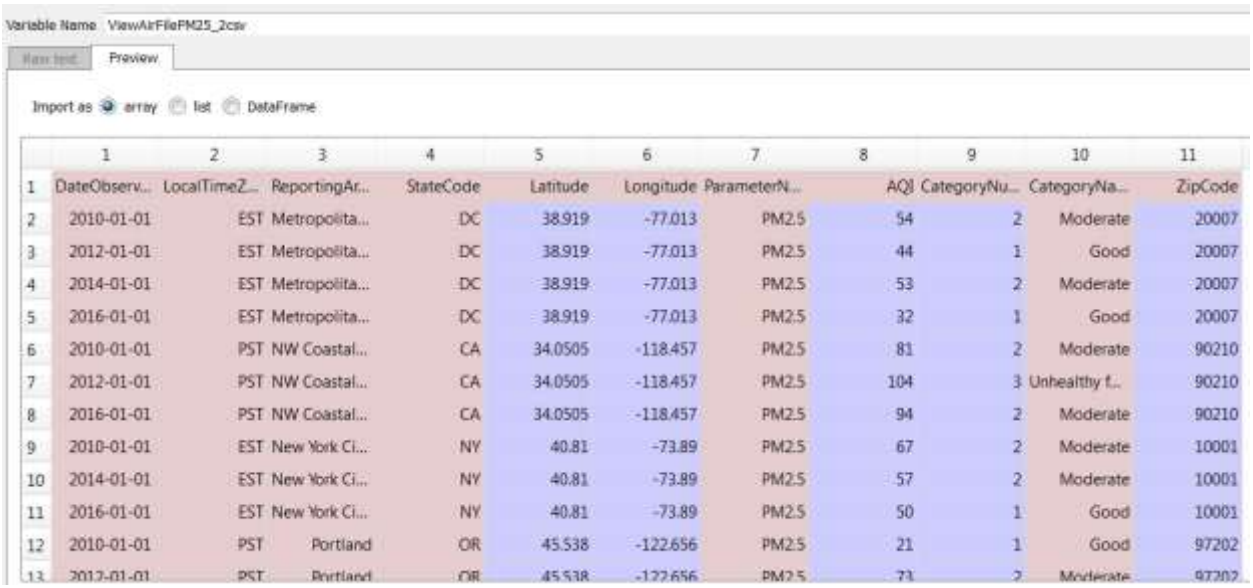
```
#This Header is FYI - it is not used in the code directly
#Header=['"DateObserved","HourObserved","LocalTimeZone","ReportingArea","StateC
ode","Latitude","Longitude","ParameterName","AQI","CategoryNumber","CategoryNam
e"']

    ziplist=["20007", "90210", "32605", "10001", "97202","33432"]
    datelist=["2004-01-01", "2006-01-01", "2008-01-01","2010-01-01","2012-01-
01","2014-01-01","2016-01-01"]

    GetAirNowData(FileName, ziplist, datelist)
    CleanData(FileName, ziplist, datelist)


def GetAirNowData(FileName, ziplist,datelist):
##Note: This is an API format example. You must USE YOUR API KEY
##http://www.airnowapi.org/aq/observation/zipCode/historical/?format=text/csv&z
ipCode=20002&date=2014-09-03T00-0000&distance=25&API_KEY=D9AA91E7-070D-4221-
867C-XXXXXXXXXX
    ZipDict={}
    #Create a new file - if exists - will delete
    File=open(FileName, "w")
    File.close()
    #----
    #Open for append
    File=open(FileName, "a")
    baseURL="http://www.airnowapi.org/aq/observation/zipCode/historical/?"
    miles=5

    for zipcode in ziplist:
        for date in datelist:
            zipURL=baseURL + urllib.parse.urlencode({
            'format': "text/csv",
            #'format':'application/json',
            'zipCode':zipcode,
            'date':date+'T00-0000',
            #yyyy-MM-ddThh-mmss"
            'distance':miles,
#Use YOUR API KEY
            'API_KEY':'D9AA91E7-070D-4221-867C-XXXXXXXXXXXX'
            })
            #print(zipURL)

            response=urlopen(zipURL).read().decode('utf-8')
            responseCopy=response
            #Build Dict for zip code and city
            ZipDict[str(responseCopy)]=zipcode
            response=response+"\n"
            #response=urlopen(zipURL).read()
            File.write(response)
    File.close()

    File=open("DictFile.txt","w")
    File.write(str(ZipDict))
    File.close()

def CleanData(FileName, ziplist, datelist):

    df=pd.read_csv(FileName)
```

```
    Newdf=df
    #Remove the rows that repeat the headers
    for i in range(len(df.DateObserved)):
        if(df.ix[i,"DateObserved"]) == "DateObserved":
            Newdf=Newdf.drop(i)
            #print(df.ix[i,"DateObserved"])

    #Remove columns that are not needed
    #Remove HourObserved
    Newdf=Newdf.drop('HourObserved', axis=1)

    #Re-index Newdf to start from 0
    Newdf2=Newdf
    Newdf2=Newdf2.reset_index(drop=True)
    #print(Newdf2)

    #Create two new dataframes. Ozonedf and PM25df
    Ozonedf=Newdf2
    PM25df=Newdf2
    #Keep only Ozone data in Ozonedf and PM25 data in PM25df
    #Drop rows that are not wanted...
    for i in range(len(Newdf2.DateObserved)):
        if(Newdf2.ix[i,"ParameterName"]) != "PM2.5":
            PM25df=PM25df.drop(i)

        if(Newdf2.ix[i,"ParameterName"]) != "OZONE":
            Ozonedf=Ozonedf.drop(i)

    #Re-index to start from 0
    Ozonedf=Ozonedf.reset_index(drop=True)
    PM25df=PM25df.reset_index(drop=True)

    #Create two csv files for Ozone and for PM25
    Ozonedf.to_csv("ViewAirFileOzone.csv", index=False)
    PM25df.to_csv("ViewAirFilePM25.csv", index=False)

    #Test the file contents
    PM25df2=pd.read_csv("ViewAirFilePM25.csv")
    Ozonedf2=pd.read_csv("ViewAirFileOzone.csv")

    #Print the values and columns for the data frames
    print(PM25df2.values)
    print(Ozonedf2.columns)

    #Check a couple of columns. Both methods work.
    print(PM25df2[["ReportingArea","AQI"]])
    print(PM25df2.AQI)

    #Printing rows using ix for index. This prints rows 2 through 5
    print(Ozonedf2.ix[2:5])

    #Add a new column to both. Here, I am adding ZipCode as a new column
    # and assigning it values from 0 to dataframe length
    Ozonedf2["ZipCode"]=numpy.arange(len(Ozonedf2))
    PM25df2["ZipCode"]=numpy.arange(len(PM25df2))

   ##Create new values in the new column
    ##First - I will create a dict that contains the values I need...
```

```
    #Create dict for city and zip using data in file
    #Recall: DictFile.txt was created in GetAirNowData() above
    File=open("DictFile.txt","r")
    ZipData=File.read()
    File.close()
    # The following uses re to find the city and zip
    CityZipDict={}
    MyList=ZipData.split(",")
    counter=0
    for item in MyList:
        counter=counter+1
        if re.match(r".*\:\s\'\d{5}.*",item):
            zipc=re.findall(r".*\:\s\'(.+?)\'",item)
            city=MyList[counter-8]
            #strip the extra quotes from the city values
            city=city[1:-1]
            #zipc is a list of one string so use [0]
            CityZipDict[city]=zipc[0]


    #print (CityZipDict)
    #Use the CityZipDict to place the corrent zipcodes for each city
    for i in range(len(Ozonedf2.DateObserved)):
        #Row i column ReportingArea
        key=(Ozonedf2.ix[i,"ReportingArea"])
        #Set row i column ZipCode to the zip in the CityZipDict
        Ozonedf2.ix[i,"ZipCode"]=(CityZipDict[key])

    for i in range(len(PM25df2.DateObserved)):
        #Row i column ReportingArea
        key=(PM25df2.ix[i,"ReportingArea"])
        #Set row i column ZipCode to the zip in the CityZipDict
        PM25df2.ix[i,"ZipCode"]=(CityZipDict[key])

    #Confirm that Ozonedf2 and PM25df2 have new column called ZipCode
        #with correct values
    print(Ozonedf2)
    print(PM25df2)

  #write the updated dataframes back to files
    Ozonedf2.to_csv("ViewAirFileOzone_2.csv", index=False)
    PM25df2.to_csv("ViewAirFilePM25_2.csv", index=False)


# Call to Main
main()
```

Discussion of the Program for AIrNow Data

The program generates two key datafiles, `ViewAirFileOzone_2.csv` and `ViewAirFilePM25_2.csv`. Figure 4 shows a portion of the `ViewAirFilePM25_2.csv` file as a Python preview.



Variable Name: ViewAirFilePM25_2csv

Preview

Import as: array, list, DataFrame

	1	2	3	4	5	6	7	8	9	10	11
1	DateObserv...	LocalTimeZ...	ReportingAr...	StateCode	Latitude	Longitude	ParameterN...	AQI	CategoryNu...	CategoryNa...	ZipCode
2	2010-01-01	EST	Metropolita...	DC	38.919	-77.013	PM2.5	54	2	Moderate	20007
3	2012-01-01	EST	Metropolita...	DC	38.919	-77.013	PM2.5	44	1	Good	20007
4	2014-01-01	EST	Metropolita...	DC	38.919	-77.013	PM2.5	53	2	Moderate	20007
5	2016-01-01	EST	Metropolita...	DC	38.919	-77.013	PM2.5	32	1	Good	20007
6	2010-01-01	PST	NW Coastal...	CA	34.0505	-118.457	PM2.5	81	2	Moderate	90210
7	2012-01-01	PST	NW Coastal...	CA	34.0505	-118.457	PM2.5	104	3	Unhealthy f...	90210
8	2016-01-01	PST	NW Coastal...	CA	34.0505	-118.457	PM2.5	94	2	Moderate	90210
9	2010-01-01	EST	New York Ci...	NY	40.81	-73.89	PM2.5	67	2	Moderate	10001
10	2014-01-01	EST	New York Ci...	NY	40.81	-73.89	PM2.5	57	2	Moderate	10001
11	2016-01-01	EST	New York Ci...	NY	40.81	-73.89	PM2.5	50	1	Good	10001
12	2010-01-01	PST	Portland	OR	45.538	-122.656	PM2.5	21	1	Good	97202
13	2012-01-01	PST	Portland	OR	45.538	-122.656	PM2.5	73	2	Moderate	97202

Figure 4: Portion of `ViewAirFilePM25_2.csv`

The next few pages will discuss the program line by line.

Lines 1 - 8

```
1. # AirNow API Python Code
2. # Author Ami Gates
3. # AirNowCh14Example.py

4. import urllib
5. from urllib.request import urlopen
6. import pandas as pd
7. import re
8. import numpy
```

Because this program will use numpy arange, I have imported numpy. I have also imported “re” so that regular expressions can be used. The import pandas is needed to use the pandas dataframes and variaous dataframe manipulations. The urllib and urllib request and urlopen are used to collect the data.

Lines 9 – 13

```
9. def main():
10.     FileName="AirNowExample.csv"

11.#Header=['"DateObserved","HourObserved","LocalTimeZone","ReportingArea","S
tateCode","Latitude","Longitude","ParameterName","AQI","CategoryNumber","Cate
goryName"']

12.     ziplist=["20007", "90210", "32605", "10001", "97202","33432"]
13.     datelist=["2004-01-01", "2006-01-01", "2008-01-01","2010-01-01","2012-
01-01","2014-01-01","2016-01-01"]
```

Here the main() function is defined. A file, "AirNowExample.csv" is created. This file will initially house all calls and results from the AirNow API requests that will be made. Line 11 is not used in the program and is for information purposes. It notes the column values expected. This information can be gained from the AirNow API site or by viewing the results. Line 12 is a list of the zipcodes that will be posted with the request to AirNow. Line 13 is a list of the dates that will be posted with the request to AirNow. A single URL will be created for each zipcode and date combination using a double for loop. We will see this shortly.

Lines 14 and 15

```
14.GetAirNowData(FileName, ziplist, datelist)
15.CleanData(FileName, ziplist, datelist)
```

Two functions will be called from main(). The first function, GetAirNowData, will collect the data from AirNow and will also create a dictionary that associates zipcode with city name for later use. The reason why this dictionary must be created is that the results returned from the API requests do not include the zipcode. However, I would like to add on a column that does include the correct zip code for each city/location. Therefore, as the data is collected, a dictionary is built that associates zipcode and location. The second function, CleanData, will mung and clean the data, will use the dictionary of location/zipcode to create a new column, and will generate two files (one for Ozone and one for PM2.5 data).

Lines 16 - 24

```
16.def GetAirNowData(FileName, ziplist,datelist):

17.##http://www.airnowapi.org/aq/observation/zipCode/historical/?format=text/
csv&zipCode=20002&date=2014-09-03T00-0000&distance=25&API_KEY=D9AA91E7-070D-
4221-867C-XXXXXXXXXXXXXXXXX
18.    ZipDict={}
19.    #Create a new file - if exists - will delete
20.    File=open(FileName, "w")
21.    File.close()
22.    #----
23.    #Open for append
24.    File=open(FileName, "a")
```

Line 16 starts the definition for function, . Line 17 is a reminder of the general format for the POST AirNow API URL. Note that you must use YOUR KEY in place of the fake API_KEY listed here. Line 18 initializes a dictionary called ZipDict. This will be the dictionary that associates zipcodes with city location names. Lines 20-24 create a new file where the AirNow data will be placed. This will be the raw data that we will then clean. Note that when opening a file to write , “w”, a new and empty file will be created. If the file exists, it will be overwritten. Opening for append, “a”, allows new information to be added to the file without losing current content.

Lines 25 - 38

```
25.    baseURL="http://www.airnowapi.org/aq/observation/zipCode/historical/?"
26.    miles=5

27.    for zipcode in ziplist:
28.        for date in datelist:
29.            zipURL=baseURL + urllib.parse.urlencode({
30.            'format': "text/csv",
31.            #'format':'application/json',
32.            'zipCode':zipcode,
33.            'date':date+'T00-0000',
34.            #yyyy-MM-ddThh-mmss"
35.            'distance':miles,
36.            'API_KEY':'D9AA91E7-070D-4221-867C-XXXXX YOUR API KEY HERE'
37.            })
38.            #print(zipURL)
```

This portion of the code sets up the URL needed to post the request to the AirNow site. Line 25 is the "base" URL. Then, lines 27-37 define and format all required parameters for the URL. The `urllib.parse.urlencode` method will encode the parameters properly. Notice that line 29 concatenates (via the +) the base URL to the result of the encoding of the parameters. Lines 27 and 28 are the for loops that loop through all zipcodes and dates in the lists defined in main(). A request will be posted for all combinations of zipcode and date. The format in this case will be text/csv, but JSON and XML are also options. Line 38 is commented out, but is used to assure that the URL created is in the correct and expected format.

Lines 39 - 45

```
39.            response=urlopen(zipURL).read().decode('utf-8')
40.            responseCopy=response
41.            #Build Dict for zip code and city
42.            ZipDict[str(responseCopy)]=zipcode
43.            response=response+"\n"
44.            #response=urlopen(zipURL).read()
45.            File.write(response)
```

Line 39 uses the `urlopen` method to send the post request URL to the AirNow site. The `response` will contain the resulting data. Notice that lines 39-45 are all contained **inside** of the double for loop and so will be repeated for each zipcode and date combination. Line 45 writes each request response to the file (which is why the file was opened for append). Line 42 collects the entire response (a copy of it) and associates it with the current zipcode. If this is unclear, print a copy of the response to see what it is. Later in the program, we will have to parse out, using regular expressions, just the city/location from that entire response. Line 43 includes a newline after each request to make the file easier to parse later.

Lines 46- 49

```
46.    File.close()
```

```
47.     File=open("DictFile.txt","w")
48.     File.write(str(ZipDict))
49.     File.close()
```

Line 46 closes the `AirNowExample.csv` file. Lines 47-49 create a file to store the dictionary results. This file will be used by the next function.

Lines 50 - 58

```
50.def CleanData(FileName, ziplist, datelist):

51.#Header=['"DateObserved","HourObserved","LocalTimeZone","ReportingArea","S
tateCode","Latitude","Longitude","ParameterName","AQI","CategoryNumber","Cate
goryName"']
52.     df=pd.read_csv(FileName)
53.     Newdf=df
54.     #Remove the rows that repeat the headers
55.     for i in range(len(df.DateObserved)):
56.         if(df.ix[i,"DateObserved"]) == "DateObserved":
57.             Newdf=Newdf.drop(i)
58.             #print(df.ix[i,"DateObserved"])
```

Line 50 starts the definition of the function, `CleanData(FileName, ziplist, datelist)`. This function requires the FileName, which in this case is `AirNowExample.csv`. It also requires the zip list and date list defined in main(). Line 52 uses pandas (as pd) to create a data frame (called df here). The method `pd.read_csv` is used to read the csv file and place the results into a dataframe pandas object. Line 54 makes a copy of the dataframe so that changes can be made while still maintaining (for reference) the original dataframe. Lines 55-58 loop through the length (number of rows) of the dataframe. The code, `df.ix[i,"DateObserved"]` looks at each row of the dataframe (the index ix) and the column in that row called "DateObserved". If the value in row i and column DateObserved is the string, "DateObserved", then line 57 will remove (drop) this row from the dataframe. The reason for this removal is that only the top row should contain the names of the columns. If any other row contains the column names, it should be eliminated as it is not part of the data. Note that throughout the code there are print statements and other notes that are commented-out with "#". These are there for testing purposes.

Lines 59 – 61

```
59.     #Remove columns that are not needed
60.     #Remove HourObserved
61.     Newdf=Newdf.drop('HourObserved', axis=1)
```

Line 61 uses the drop method to remove a column from the dataframe. Within the dataframe is a column called, "HourObserved". There is no data for this column and all values are "0". This column is not needed and so is dropped.

Lines 62 - 65

```
62.     #Re-index Newdf to start from 0
63.     Newdf2=Newdf
64.     Newdf2=Newdf2.reset_index(drop=True)
65.     #print(Newdf2)
```

Lines 63 and 64 will re-index the new dataframe. In lines 56 and 57 above, certain rows were dropped as they contained no data and were only duplicates of the column names. When a row is dropped, the index (the row name) is dropped with it. In this dataframe, there are no explicit row names and so the indexing is 0, 1, 2, … However, once a row is dropped that index is removed. This creates an index with missing rows, such as 0, 2, 5, 7…

In this case, we can re-index the rows to start again from 0 and to proceed in order (0, 1, 2, …). Line 64 uses the `reset_index` method to do this.

Lines 66-78

```
66.     #Create two new dataframes. Ozonedf and PM25df
67.     Ozonedf=Newdf2
68.     PM25df=Newdf2
69.     #Keep only Ozone data in Ozonedf and PM25 data in PM25df
70.     #Drop rows that are not wanted...
71.     for i in range(len(Newdf2.DateObserved)):
72.         if(Newdf2.ix[i,"ParameterName"]) != "PM2.5":
73.             PM25df=PM25df.drop(i)

74.         if(Newdf2.ix[i,"ParameterName"]) != "OZONE":
75.             Ozonedf=Ozonedf.drop(i)

76.     #Re-index to start from 0
77.     Ozonedf=Ozonedf.reset_index(drop=True)
78.     PM25df=PM25df.reset_index(drop=True)
```

At this point in the program, the dataset has been updated to only contain data (rather than column header duplicates) and one of the columns has been removed. Lines 66-78 will break the dataframe into two separate dataframes, one for Ozone and one for PM2.5. This is certainly not required and is just a preference as well as instructional. To accomplish this task, lines 67 and 68 create two copies of the current dataframe. Lines 71 – 75 loop through the all rows of current dataframe and drop all entries from the PM2.5 dataframe that do not contain PM2.5 data and similarly all entries from the Ozone dataframe that do not contain Ozone data. The result is two new dataframes, `Ozonedf` and `PM25df`.

Lines 79 – 87

```
79.     #Create two csv files for Ozone and for PM25
80.     Ozonedf.to_csv("ViewAirFileOzone.csv", index=False)
81.     PM25df.to_csv("ViewAirFilePM25.csv", index=False)

82.     #Test the file contents
83.     PM25df2=pd.read_csv("ViewAirFilePM25.csv")
```

```
84.    Ozonedf2=pd.read_csv("ViewAirFileOzone.csv")

85.    #Print the values and columns for the data frames
86.    print(PM25df2.values)
87.    print(Ozonedf2.columns)
```

Lines 80 and 81 create new scv files in which to store the new dataframes. Lines 83 and 84 test the file contents by reading the files. Lines 86 and 87 also test the data in the files by printing values and columns.

Lines 88-90

```
88.    #Check a couple of columns. Both methods work.
89.    print(PM25df2[["ReportingArea","AQI"]])
90.    print(PM25df2.AQI)
```

Lines 89 and 90 show examples of how to view specific data in the dataframe. Line 89 will print all the data contained in the columns named `ReportingArea` and `AQI`.

Lines 91 and 92

```
91.    #Printing rows using ix for index. This prints rows 2 through 5
92.    print(Ozonedf2.ix[2:5])
```

Line 92 is a an example of how to view the data in the dataframe by row. The “ix” stands for index or row. This will print rows 2, 3, 4, and 5.

Lines 93- 96

```
93.    #Add a new column to both. Here, I am adding ZipCode as a new column
94.    # and assigning it values from 0 to dataframe length
95.    Ozonedf2["ZipCode"]=numpy.arange(len(Ozonedf2))
96.    PM25df2["ZipCode"]=numpy.arange(len(PM25df2))
```

Lines 95 and 96 are the first steps in adding a new zipcode column to the dataframe Ozone (called Ozonedf2) and to the dataframe PM2.5 (called PM25df2). Recall that the results returned by the AirNow API did not include the zipcode for the city location. Therefore, the code first created a dictionary that associated the entire results with the zip code. Below in following lines and with the use of regular expressions, just the city will be extracted and will be linked to the correct zipcode. Line 95 creates a new column in the Ozone dataframe and fills the column with numbers from 0 to the length of the dataframe. Line 96 does the same thing for the PM25 dataframe. If you were to view both dataframes at this point in the code, you would see that both now have a new column for “ZipCode”. The next step is to get the proper values to place into the new columns.

Lines 97 - 104

```
97.     ##Create new values in the new column
98.     ##First - I will create a dict that contains the values I need...

99.     #Create dict for city and zip using data in file
100     #Recall: DictFile.txt was created in GetAirNowData() above
101     File=open("DictFile.txt","r")
102     ZipData=File.read()
103     File.close()
104     # The following uses re to find the city and zip
```

Recall that line 42, `ZipDict[str(responseCopy)]=zipcode,` in the `GetAirNowFunction` (above) created a dictionary that associated the entire results from the response of the URL request with the *zipcode* for the request. However, the entire response (`responseCopy`) now needs to be searched for the city location. The goal to find the city location for each zip code and create a new dictionary that associates just the city with the zipcode. Once we have this, we can place the zipcodes into the dataframes. Line 101 opens the dictionary file created from lines 47 and 48. Line 102 reads that dictionary into a variable called, `ZipData` .

Lines 105-107

```
105     CityZipDict={}
106     MyList=ZipData.split(",")
107     counter=0
```

Line 105 creates a new dictionary that is empty. Line 106 creates a list of all strings (split by ",") from `ZipData`. Line 107 starts a counter to keep track of progression through the list. The best way to really see and understand what is going on and why this is necessary is to type in this code and to view the contents of `ZipData.`

Lines 108 - 116

```
108     for item in MyList:
109         counter=counter+1
110         if re.match(r".*\:\s\'\d{5}.*",item):
111             zipc=re.findall(r".*\:\s\'(.+?)\'",item)
112             city=MyList[counter-8]
113             #strip the extra quotes from the city values
114             city=city[1:-1]
115             #zipc is a list of one string so use [0]
116             CityZipDict[city]=zipc[0]
```

Line 108 loops through all strings (items) in the list called `MyList.` In `MyList` is the city location and the zipcode. The goal is to locate both. Lines 110 and 111 use regular expressions `(re)` to locate the zip code. The `r"` means, "this is the regular expression".

```
r".*\:\s\'\d{5}.*"
```

```
r".*\:\s\'(.+?)\"
```

The first regular expression above says: “Find anything any number of times until you find a “ : ”, then one space, then one ', then 5 digits, and then anything. This will locate all zip codes. The next regular expression says: “Find anything until you find a “ : “ then one space, then one ', then return the text that is in that location. This regular expression will return the zipcode. To see why and how, run the code and investigate.

Line 112 assigns the city location name to the current location (thus the use of the counter) minus 8. This can be seen by viewing a few lines of the original dictionary of the data.

Line 114 strips the front and end quotes from the string that is the city location name.

Line 116 assignments the city location name to the zipcode using the dictionary, `CityZipDict`. Note that `zipc[0]` is needed because `zipc` is a list, but we want to assign the string within the list. Again, to really understand that part of the code, it is best to review the original dictionary contents.

Lines 117 - 128

```
117    #print (CityZipDict)
118    #Use the CityZipDict to place the corrent zipcodes for each city
119    for i in range(len(Ozonedf2.DateObserved)):
120        #Row i column ReportingArea
121        key=(Ozonedf2.ix[i,"ReportingArea"])
122        #Set row i column ZipCode to the zip in the CityZipDict
123        Ozonedf2.ix[i,"ZipCode"]=(CityZipDict[key])

124    for i in range(len(PM25df2.DateObserved)):
125        #Row i column ReportingArea
126        key=(PM25df2.ix[i,"ReportingArea"])
127        #Set row i column ZipCode to the zip in the CityZipDict
128        PM25df2.ix[i,"ZipCode"]=(CityZipDict[key])
```

Lines 119 – 123 loop through the Ozone dataframe by row. The value in each row under “ReportingArea” (which is the city location) is saved as variable, `key`. Next, the column called “ZipCode” (for each row) is filled in with the correct zipcode that matches the city location `key`.

Lines 124-128 do the same thing, but for the PM2.5 dataframe. The idea here is that we have now added a new column to the dataframe. For each city location, we add the correct zip code. We can do this because we first created a dictionary that associated all city locations to zipcodes.

Lines 129 - 136

```
129    #Confirm that Ozonedf2 and PM25df2 have new column called ZipCode
130    #with correct values
```

```
131     print(Ozonedf2)
132     print(PM25df2)

133     #write the updated dataframes back to files
134     Ozonedf2.to_csv("ViewAirFileOzone_2.csv", index=False)
135     PM25df2.to_csv("ViewAirFilePM25_2.csv", index=False)



#Call to main
136 main()
```

Lines 131 and 132 confirm the contents of the two new dataframes, Ozonedf2 and PM25df2. Lines 134 and 135 write these dataframes to two csv files for further processing. Line 136 calls main and starts the program.

Comments and Notes

The above program and all related portions are only one small example of grabbing data from the web, munging and cleaning the data, and created usable data files from the data that can then be submitted to analysis and investigation.