Reference

This is a DataCamp course

Course Description

A vital component of data science involves acquiring raw data and getting it into a form ready for analysis. In fact, it is commonly said that data scientists spend 80% of their time cleaning and manipulating data, and only 20% of their time actually analyzing it. This course will equip you with all the skills you need to clean your data in Python, from learning how to diagnose your data for problems to dealing with missing values and outliers. At the end of the course, you'll apply all of the techniques you've learned to a case study in which you'll clean a real-world Gapminder dataset!

Exploring your data

So you've just got a brand new dataset and are itching to start exploring it. But where do you begin, and how can you be sure your dataset is clean? This chapter will introduce you to the world of data cleaning in Python! You'll learn how to explore your data with an eye for diagnosing issues such as outliers, missing values, and duplicate rows.

Write a list of what to do in clearning data

Loading and viewing your data

In this chapter, you're going to look at a subset of the Department of Buildings Job Application Filings dataset from the NYC Open Data portal. This dataset consists of job applications filed on January 22, 2017.

Your first task is to load this dataset into a DataFrame and then inspect it using the .head() and .tail() methods. However, you'll find out very quickly that the printed results don't allow you to see everything you need, since there are too many columns. Therefore, you need to look at the data in another way.

The .shape and .columns attributes let you see the shape of the DataFrame and obtain a list of its columns. From here, you can see which columns are relevant to the questions you'd like to ask of the data. To this end, a new DataFrame, df_subset, consisting only of these relevant columns, has been pre-loaded. This is the DataFrame you'll work with in the rest of the chapter.

Get acquainted with the dataset now by exploring it with pandas! This initial exploratory analysis is a crucial first step of data cleaning.

In [2]:
```
# Import pandas
import pandas as pd

# Read the file into a DataFrame: df
df = pd.read_csv('dob_job_application_filings_subset.csv')

# Print the head of df
print(df.head())

# Print the tail of df
print(df.tail())

# Print the shape of df
print(df.shape) #not shape()

# Print the columns of df
print(df.columns) #not columns()

# Print the head and tail of df_subset
# print(df_subset.head())
# print(df_subset.tail())
```

```
2  635 RIVERSIDE DRIVE NY LLC                619
3  48 W 25 ST LLC C/O BERNSTEIN              150
4  HYUNG-HYANG REALTY CORP                   614

          Owner'sHouse Street Name            City  State    Zip  \
0  EAST 56TH STREET                  NEW YORK           NY  10222
1  KNOX PLACE                        STATEN ISLAND      NY  10314
2  WEST 54TH STREET                  NEW YORK           NY  10016
3  WEST 30TH STREET                  NEW YORK           NY  10001
4  8 AVENUE                          NEW YORK           NY  10001

  Owner'sPhone #                                    Job Description  \
0     2125545837  GENERAL MECHANICAL & PLUMBING MODIFICATIONS AS...
1     3477398892  BUILDERS PAVEMENT PLAN 143 LF.                ...
2     2127652555  GENERAL CONSTRUCTION TO INCLUDE NEW PARTITIONS...
3     2125941414  STRUCTURAL CHANGES ON THE 5TH FLOOR (MOONDOG E...
4     2019881222  FILING HEREWITH FACADE REPAIR PLANS. WORK SCOP...

               DOBRunDate
0  04/26/2013 12:00:00 AM
```

Great work! In addition to the suspicious number of 0 values, which may represent missing data, notice that the columns that contain monetary values - 'Initial Cost' and 'Total Est. Fee' - have a dollar sign in the beginning. These columns may be coded as strings instead of numeric values. You will check this in the next exercise.

Further diagnosis

In the previous exercise, you identified some potentially unclean or missing data. Now, you'll continue to diagnose your data with the very useful .info() method.

The .info() method provides important information about a DataFrame, such as the number of rows, number of columns, number of non-missing values in each column, and the data type stored in each column. This is the kind of information that will allow you to confirm whether the 'Initial Cost' and 'Total Est. Fee' columns are numeric or strings. From the results, you'll also be able to see whether or not all columns have complete data in them.

The full DataFrame df and the subset DataFrame df_subset have been pre-loaded. Your task is to use the .info() method on these and analyze the results.

In [3]:
```
# Print the info of df
print(df.info())

# # Print the info of df_subset
# print(df_subset.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12846 entries, 0 to 12845
Data columns (total 82 columns):
Job #                           12846 non-null int64
Doc #                           12846 non-null int64
Borough                         12846 non-null object
House #                         12846 non-null object
Street Name                     12846 non-null object
Block                           12846 non-null int64
Lot                             12846 non-null int64
Bin #                           12846 non-null int64
Job Type                        12846 non-null object
Job Status                      12846 non-null object
Job Status Descrp               12846 non-null object
Latest Action Date              12846 non-null object
Building Type                   12846 non-null object
Community - Board               12846 non-null object
Cluster                         0 non-null float64
Landmarked                      2067 non-null object
Adult Estab                     1 non-null object
Loft Board                      65 non-null object
City Owned                      1419 non-null object
Little e                        365 non-null object
PC Filed                        0 non-null float64
eFiling Filed                   12846 non-null object
Plumbing                        12846 non-null object
Mechanical                      12846 non-null object
Boiler                          12846 non-null object
Fuel Burning                    12846 non-null object
Fuel Storage                    12846 non-null object
Standpipe                       12846 non-null object
Sprinkler                       12846 non-null object
Fire Alarm                      12846 non-null object
Equipment                       12846 non-null object
Fire Suppression                12846 non-null object
Curb Cut                        12846 non-null object
```

```
Other                           12846 non-null object
Other Description               12846 non-null object
Applicant's First Name          12846 non-null object
Applicant's Last Name           12846 non-null object
Applicant Professional Title    12846 non-null object
Applicant License #             12846 non-null object
Professional Cert               6908 non-null object
Pre- Filing Date                12846 non-null object
Paid                            11961 non-null object
Fully Paid                      11963 non-null object
Assigned                        3817 non-null object
Approved                        4062 non-null object
Fully Permitted                 1495 non-null object
Initial Cost                    12846 non-null object
Total Est. Fee                  12846 non-null object
Fee Status                      12846 non-null object
Existing Zoning Sqft            12846 non-null int64
Proposed Zoning Sqft            12846 non-null int64
Horizontal Enlrgmt              231 non-null object
Vertical Enlrgmt                142 non-null object
Enlargement SQ Footage          12846 non-null int64
Street Frontage                 12846 non-null int64
ExistingNo. of Stories          12846 non-null int64
Proposed No. of Stories         12846 non-null int64
Existing Height                 12846 non-null int64
Proposed Height                 12846 non-null int64
Existing Dwelling Units         12846 non-null object
Proposed Dwelling Units         12846 non-null object
Existing Occupancy              12846 non-null object
Proposed Occupancy              12846 non-null object
Site Fill                       8641 non-null object
Zoning Dist1                    11263 non-null object
Zoning Dist2                    1652 non-null object
Zoning Dist3                    88 non-null object
Special District 1              3062 non-null object
Special District 2              848 non-null object
Owner Type                      0 non-null float64
Non-Profit                      971 non-null object
Owner's First Name              12846 non-null object
Owner's Last Name               12846 non-null object
Owner's Business Name           12846 non-null object
Owner's House Number            12846 non-null object
Owner'sHouse Street Name        12846 non-null object
```

```
City                            12846 non-null object
State                           12846 non-null object
Zip                             12846 non-null int64
Owner'sPhone #                  12846 non-null int64
Job Description                 12699 non-null object
DOBRunDate                      12846 non-null object
dtypes: float64(3), int64(15), object(64)
memory usage: 8.0+ MB
None
```

Excellent! Notice that the columns 'Initial Cost' and 'Total Est. Fee' are of type object. The currency sign in the beginning of each value in these columns needs to be removed, and the columns need to be converted to numeric. In the full DataFrame, note that there are a lot of missing values. You saw in the previous exercise that there are also a lot of 0 values. Given the amount of data that is missing in the full dataset, it's highly likely that these 0 values represent missing data.

Calculating summary statistics

You'll now use the .describe() method to calculate summary statistics of your data.
describe() only handle numerical values

In this exercise, the columns 'Initial Cost' and 'Total Est. Fee' have been cleaned up for you. That is, the dollar sign has been removed and they have been converted into two new numeric columns: initial_cost and total_est_fee. You'll learn how to do this yourself in later chapters. It's also worth noting that some columns such as Job # are encoded as numeric columns, but it does not make sense to compute summary statistics for such columns.

This cleaned DataFrame has been pre-loaded as df. Your job is to use the .describe() method on it in the IPython Shell and select the statement below that is False.

In [2]:
```
#f.describe()
```

Frequency counts for categorical data

As you've seen, **.describe() can only be used on numeric columns**. So how can you diagnose data issues when you have categorical data? One way is by using the **.value_counts()** method, which returns the frequency counts for each unique value in a column!

This method also has an optional parameter called dropna which is True by default. What this means is if you have missing data in a column, it will not give a frequency count of them. **You want to set the dropna column to False so if there are missing values in a column, it will give you the frequency counts**.

In this exercise, you're going to look at the 'Borough', 'State', and 'Site Fill' columns to make sure all the values in there are valid. When looking at the output, do a sanity check: Are all values in the 'State' column from NY, for example? Since the dataset consists of applications filed in NY, you would expect this to be the case.

In [4]:
```
# Print the value counts for 'Borough'
print(df['Borough'].value_counts(dropna=False))

# Print the value_counts for 'State'
print(df['State'].value_counts(dropna=False))

# Print the value counts for 'Site Fill'
print(df['Site Fill'].value_counts(dropna=False))
```

```
MANHATTAN        6310
BROOKLYN         2866
QUEENS           2121
BRONX             974
STATEN ISLAND     575
Name: Borough, dtype: int64
NY    12391
NJ      241
PA       38
CA       20
OH       19
FL       17
IL       17
CT       16
TX       13
TN       10
MD        7
DC        7
MA        6
GA        6
KS        6
VA        5
CO        4
MN        3
AZ        3
SC        3
WI        3
NC        2
UT        2
RI        2
IN        1
```

```
VT        1
MI        1
WA        1
NM        1
Name: State, dtype: int64
NOT APPLICABLE                          7806
NaN                                     4205
ON-SITE                                  519
OFF-SITE                                 186
USE UNDER 300 CU.YD                      130
Name: Site Fill, dtype: int64
```

Fantastic work! Notice how not all values in the 'State' column are NY. This is an interesting find, as this data is supposed to consist of applications filed in NYC. Curiously, all the 'Borough' values are correct. A good start as to why this may be the case would be to find and look at the codebook for this dataset. Also, for the 'Site Fill' column, you may or may not need to recode the NOT APPLICABLE values to NaN in your final analysis.

Visualizing single variables with histograms

Up until now, you've been looking at descriptive statistics of your data. One of the best ways to confirm what the numbers are telling you is to plot and visualize the data.

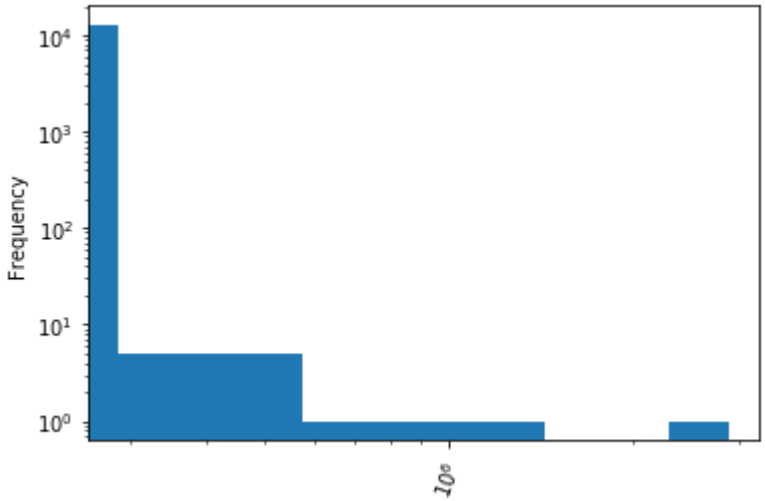
You'll start by visualizing single variables using a histogram for numeric values. The column you will work on in this exercise is 'Existing Zoning Sqft'.

The .plot() method allows you to create a plot of each column of a DataFrame. The kind parameter allows you to specify the type of plot to use - kind='hist', for example, plots a histogram.

In the IPython Shell, begin by computing summary statistics for the 'Existing Zoning Sqft' column using the .describe() method. You'll notice that there are extremely large differences between the min and max values, and the plot will need to be adjusted accordingly. In such cases, it's good to look at the plot on a log scale. The keyword arguments logx=True or logy=True can be passed in to .plot() depending on which axis you want to rescale.

Finally, note that Python will render a plot such that the axis will hold all the information. That is, if you end up with large amounts of whitespace in your plot, it indicates counts or values too small to render.

In [15]:
```
# Import matplotlib.pyplot
import matplotlib.pyplot as plt

# Plot the histogram
df['Existing Zoning Sqft'].plot(kind='hist', rot=70, logx=True, logy=True)

# Display the histogram
plt.show()
```



Excellent work! While visualizing your data is a great way to understand it, keep in mind that no one technique is better than another. As you saw here, you still needed to look at the summary statistics to help understand your data better. You expected a large amount of counts on the left side of the plot because the 25th, 50th, and 75th percentiles have a value of 0. The plot shows us that there are barely any counts near the max value, signifying an outlier.

Visualizing multiple variables with boxplots

Histograms are great ways of visualizing single variables. To visualize multiple variables, boxplots are useful, especially when one of the variables is categorical.

In this exercise, your job is to use a boxplot to compare the 'initial_cost' across the different values of the 'Borough' column. The pandas .boxplot() method is a quick way to do this, in which you have to specify the column and by parameters. Here, you want to visualize how 'initial_cost' varies by 'Borough'.

pandas and matplotlib.pyplot have been imported for you as pd and plt, respectively, and the DataFrame has been pre-loaded as df.

In [ ]:
```
# Import necessary modules
import pandas as pd
import matplotlib.pyplot as plt

# Create the boxplot
df.boxplot(column='initial_cost', by='Borough', rot=90)

# Display the plot
plt.show()

##Check ealier  comments how initial_cost is calculated. And I should do that later to make the code here running
```

Great work! You can see the 2 extreme outliers are in the borough of Manhattan. An initial guess could be that since land in Manhattan is extremely expensive, these outliers may be valid data points. Again, further investigation is needed to determine whether or not you can drop or keep those points in your data.

Visualizing multiple variables with scatter plots

Boxplots are great when you have a numeric column that you want to compare across different categories. When you want to visualize two numeric columns, scatter plots are ideal.

In this exercise, your job is to make a scatter plot with 'initial_cost' on the x-axis and the 'total_est_fee' on the y-axis. You can do this by using the DataFrame .plot() method with kind='scatter'. You'll notice right away that there are 2 major outliers shown in the plots.

Since these outliers dominate the plot, an additional DataFrame, df_subset, has been provided, in which some of the extreme values have been removed. After making a scatter plot using this, you'll find some interesting patterns here that would not have been seen by looking at summary statistics or 1 variable plots.

When you're done, you can cycle between the two plots by clicking the 'Previous Plot' and 'Next Plot' buttons below the plot.

In [ ]:
```
# Import necessary modules
import pandas as pd
import matplotlib.pyplot as plt

# Create and display the first scatter plot
df.plot(kind='scatter', x='initial_cost', y='total_est_fee', rot=70)
plt.show()

# Create and display the second scatter plot
df_subset.plot(kind='scatter', x='initial_cost', y='total_est_fee', rot=70)
plt.show()
```

Excellent work! In general, from the second plot it seems like there is a strong correlation between 'initial_cost' and 'total_est_fee'. In addition, take note of the large number of points that have an 'initial_cost' of 0. It is difficult to infer any trends from the first plot because it is dominated by the outliers.

Tidying data for analysis

Here, you'll learn about the principles of tidy data and more importantly, why you should care about them and how they make subsequent data analysis more efficient. You'll gain first hand experience with reshaping and tidying your data using techniques such as pivoting and melting.

Recognizing tidy data

For data to be tidy, it must have:

Each variable as a separate column. Each row as a separate observation. As a data scientist, you'll encounter data that is represented in a variety of different ways, so it is important to be able to recognize tidy (or untidy) data when you see it.

In this exercise, two example datasets have been pre-loaded into the DataFrames df1 and df2. Only one of them is tidy. Your job is to explore these further in the IPython Shell and identify the one that is not tidy, and why it is not tidy.

In the rest of this course, you will frequently be asked to explore the structure of DataFrames in the IPython Shell prior to performing different operations on them. Doing this will not only strengthen your comprehension of the data cleaning concepts covered in this course, but will also help you realize and take advantage of the relationship **between working in the Shell and in the script.**

In [ ]:

```
In [2]: df1.head()
Out[2]:
   Ozone  Solar.R  Wind  Temp  Month  Day
0   41.0    190.0   7.4    67      5    1
1   36.0    118.0   8.0    72      5    2
2   12.0    149.0  12.6    74      5    3
3   18.0    313.0  11.5    62      5    4
4    NaN      NaN  14.3    56      5    5

In [3]: df2.head()
Out[3]:
   Month  Day variable  value
0      5    1    Ozone   41.0
1      5    2    Ozone   36.0
2      5    3    Ozone   12.0
3      5    4    Ozone   18.0
4      5    5    Ozone    NaN
29     5   30    Ozone  115.0
..     ...  ...      ...    ...
582    9    1     Temp   91.0
583    9    2     Temp   92.0
```

Exactly! Notice that the variable column of df2 contains the values Solar.R, Ozone, Temp, and Wind. For it to be tidy, these should all be in separate columns, as in df1.

Reshaping your data using melt

Melting data is the process of turning columns of your data into rows of data. Consider the DataFrames from the previous exercise. In the tidy DataFrame, the variables Ozone, Solar.R, Wind, and Temp each had their own column. If, however, you wanted these variables to be in rows instead, you could melt the DataFrame. In doing so, however, you would make the data untidy! This is important to keep in mind: Depending on how your data is represented, you will have to reshape it differently.

In this exercise, you will practice melting a DataFrame using pd.melt(). There are two parameters you should be aware of: id_vars and value_vars. The id_vars represent the columns of the data you do not want to melt (i.e., keep it in its current shape), while the value_vars represent the columns you do wish to melt into rows. By default, if no value_vars are provided, all columns not set in the id_vars will be melted. This could save a bit of typing, depending on the number of columns that need to be melted.

The (tidy) DataFrame airquality has been pre-loaded. Your job is to melt its Ozone, Solar.R, Wind, and Temp columns into rows. Later in this chapter, you'll learn how to bring this melted DataFrame back into a tidy form.

In [9]:
```
import pandas as pd

airquality = pd.read_csv("airquality.csv")
# Print the head of airquality
print(airquality.head())

# Melt airquality: airquality_melt
airquality_melt = pd.melt(airquality, id_vars=['Month', 'Day'])

# Print the head of airquality_melt
print(airquality_melt.head())

#PRINT all the rows and see how the columns are treated in melt.
```

```
   Ozone  Solar.R  Wind  Temp  Month  Day
0   41.0    190.0   7.4    67      5    1
1   36.0    118.0   8.0    72      5    2
2   12.0    149.0  12.6    74      5    3
3   18.0    313.0  11.5    62      5    4
4    NaN      NaN  14.3    56      5    5
   Month  Day variable  value
0      5    1    Ozone   41.0
1      5    2    Ozone   36.0
2      5    3    Ozone   12.0
3      5    4    Ozone   18.0
4      5    5    Ozone    NaN
```

Well done! This exercise demonstrates that melting a DataFrame is not always appropriate if you want to make it tidy. You may have to perform other transformations depending on how your data is represented.

Customizing melted data

When melting DataFrames, it would be better to have column names more meaningful than variable and value.

The default names may work in certain situations, but it's best to always have data that is self explanatory.

You can rename the variable column by specifying an argument to the var_name parameter, and the value column by specifying an argument to the value_name parameter. You will now practice doing exactly this. The DataFrame airquality has been pre-loaded for you.

In [10]:
```
# Print the head of airquality
print(airquality.head())

# Melt airquality: airquality_melt
airquality_melt = pd.melt(airquality, id_vars=['Month', 'Day'], var_name='measurement', value_name='reading')

# Print the head of airquality_melt
print(airquality_melt.head())
```

```
   Ozone  Solar.R  Wind  Temp  Month  Day
0   41.0    190.0   7.4    67      5    1
1   36.0    118.0   8.0    72      5    2
2   12.0    149.0  12.6    74      5    3
3   18.0    313.0  11.5    62      5    4
4    NaN      NaN  14.3    56      5    5
   Month  Day measurement  reading
0      5    1       Ozone     41.0
1      5    2       Ozone     36.0
2      5    3       Ozone     12.0
3      5    4       Ozone     18.0
4      5    5       Ozone      NaN
```

Great work! The DataFrame is more informative now. In the next video, you'll learn about pivoting, which is the opposite of melting. You'll then be able to convert this DataFrame back into its original, tidy, form!

Pivot data

Pivoting data is the opposite of melting it. Remember the tidy form that the airquality DataFrame was in before you melted it? You'll now begin pivoting it back into that form using the .pivot_table() method!

While melting takes a set of columns and turns it into a single column, pivoting will create a new column for each unique value in a specified column.

.pivot_table() has an index parameter which you can use to specify the columns that you don't want pivoted: It is similar to the id_vars parameter of pd.melt(). Two other parameters that you have to specify are columns (the name of the column you want to pivot), and values (the values to be used when the column is pivoted). The melted DataFrame airquality_melt has been pre-loaded for you.

In [11]:
```
# Print the head of airquality_melt
print(airquality_melt.head())

# Pivot airquality_melt: airquality_pivot
airquality_pivot = airquality_melt.pivot_table(index=['Month', 'Day'], columns='measurement', values='reading')

# Print the head of airquality_pivot
print(airquality_pivot.head())
```

```
   Month  Day measurement  reading
0      5    1       Ozone     41.0
1      5    2       Ozone     36.0
2      5    3       Ozone     12.0
3      5    4       Ozone     18.0
4      5    5       Ozone      NaN
measurement  Ozone  Solar.R  Temp  Wind
Month Day
5     1       41.0    190.0  67.0   7.4
      2       36.0    118.0  72.0   8.0
      3       12.0    149.0  74.0  12.6
      4       18.0    313.0  62.0  11.5
      5        NaN      NaN  56.0  14.3
```

Excellent work! Notice that the pivoted DataFrame does not actually look like the original DataFrame. In the next exercise, you'll turn this pivoted DataFrame back into its original form.

Resetting the index of a DataFrame

After pivoting airquality_melt in the previous exercise, you didn't quite get back the original DataFrame.

What you got back instead was a pandas DataFrame with a hierarchical index (also known as a MultiIndex).

Hierarchical indexes are covered in depth in Manipulating DataFrames with pandas. In essence, they allow you to group columns or rows by another variable - in this case, by 'Month' as well as 'Day'.

There's a very simple method you can use to get back the original DataFrame from the pivoted DataFrame: .reset_index(). Dan didn't show you how to use this method in the video, but you're now going to practice using it in this exercise to get back the original DataFrame from airquality_pivot, which has been pre-loaded.

In [12]:
```
# Print the index of airquality_pivot
print(airquality_pivot.index)

# Reset the index of airquality_pivot: airquality_pivot
airquality_pivot_reset = airquality_pivot.reset_index()

# Print the new index of airquality_pivot_reset
print(airquality_pivot_reset.index)

# Print the head of airquality_pivot_reset
print(airquality_pivot_reset.head())
```

```
MultiIndex(levels=[[5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31]],
           labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2
0, 21, 22, 23, 24, 25, 26, 27, 28, 29, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2
0, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
9, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]],
           names=['Month', 'Day'])
RangeIndex(start=0, stop=153, step=1)
measurement  Month  Day  Ozone  Solar.R  Temp  Wind
0                5    1   41.0    190.0  67.0   7.4
1                5    2   36.0    118.0  72.0   8.0
2                5    3   12.0    149.0  74.0  12.6
3                5    4   18.0    313.0  62.0  11.5
4                5    5    NaN      NaN  56.0  14.3
```

Pivoting duplicate values

So far, you've used the .pivot_table() method when there are multiple index values you want to hold constant during a pivot. In the video, Dan showed you how you can also use pivot tables to deal with duplicate values by providing an aggregation function through the aggfunc parameter. Here, you're going to combine both these uses of pivot tables.

Let's say your data collection method accidentally duplicated your dataset. Such a dataset, in which each row is duplicated, has been pre-loaded as airquality_dup. In addition, the airquality_melt DataFrame from the previous exercise has been pre-loaded. Explore their shapes in the IPython Shell by accessing their .shape attributes to confirm the duplicate rows present in airquality_dup.

You'll see that by using .pivot_table() and the aggfunc parameter, you can not only reshape your data, but also remove duplicates. Finally, you can then flatten the columns of the pivoted DataFrame using .reset_index().

NumPy and pandas have been imported as np and pd respectively.

Example of duplicate values:

	date	element	value
0	2010-01-30	tmax	27.8
1	2010-01-30	tmin	14.5
2	2010-02-02	tmax	27.3
3	2010-02-02	tmin	14.4

	date	element	value
0	2010-01-30	tmax	27.8
1	2010-01-30	tmin	14.5
2	2010-02-02	tmax	27.3
3	2010-02-02	tmin	14.4
4	2010-02-02	tmin	16.4

In [ ]:
```
# Pivot airquality_dup: airquality_pivot
airquality_pivot = airquality_dup.pivot_table(index=['Month', 'Day'], columns='measurement', values='reading', ag
#Note only pivot_table method can handle duplicate values, while pivot method cannot.

# Reset the index of airquality_pivot
airquality_pivot = airquality_pivot.reset_index()

# Print the head of airquality_pivot
print(airquality_pivot.head())

# Print the head of airquality
print(airquality.head())
```

Fantastic! The default aggregation function used by .pivot_table() is np.mean(). So you could have pivoted the duplicate values in this DataFrame even without explicitly specifying the aggfunc parameter.

Splitting a column with .str

The dataset you saw in the video, consisting of case counts of tuberculosis by country, year, gender, and age group, has been pre-loaded into a DataFrame as tb.

In this exercise, you're going to tidy the 'm014' column, which represents males aged 0-14 years of age. In order to parse this value, you need to extract the first letter into a new column for gender, and the rest into a column for age_group. Here, since you can parse values by position, you can take advantage of pandas' vectorized string slicing by using the str attribute of columns of type object.

Begin by printing the columns of tb in the IPython Shell using its .columns attribute, and take note of the problematic column.

In [25]:
```
import pandas as pd

tb = pd.read_csv('tb.csv')
print (tb.head())
# Melt tb: tb_melt
tb_melt = pd.melt(tb, id_vars=['country', 'year'])
print (tb_melt.head())
# Create the 'gender' column
tb_melt['gender'] = tb_melt.variable.str[0]

# Create the 'age_group' column
tb_melt['age_group'] = tb_melt.variable.str[1:] #we only want 14, so start from 1.

# Print the head of tb_melt
print(tb_melt.head())
```

```
  country  year  m014  m1524  m2534  m3544  m4554  m5564   m65  mu  f014  \
0      AD  2000   0.0    0.0    1.0    0.0    0.0    0.0   0.0 NaN   NaN
1      AE  2000   2.0    4.0    4.0    6.0    5.0   12.0  10.0 NaN   3.0
2      AF  2000  52.0  228.0  183.0  149.0  129.0   94.0  80.0 NaN  93.0
3      AG  2000   0.0    0.0    0.0    0.0    0.0    0.0   1.0 NaN   1.0
4      AL  2000   2.0   19.0   21.0   14.0   24.0   19.0  16.0 NaN   3.0

   f1524  f2534  f3544  f4554  f5564   f65  fu
0    NaN    NaN    NaN    NaN    NaN   NaN NaN
1   16.0    1.0    3.0    0.0    0.0   4.0 NaN
2  414.0  565.0  339.0  205.0   99.0  36.0 NaN
3    1.0    1.0    0.0    0.0    0.0   0.0 NaN
4   11.0   10.0    8.0    8.0    5.0  11.0 NaN
  country  year variable  value
0      AD  2000     m014    0.0
1      AE  2000     m014    2.0
2      AF  2000     m014   52.0
3      AG  2000     m014    0.0
4      AL  2000     m014    2.0
  country  year variable  value gender age_group
0      AD  2000     m014    0.0      m       014
1      AE  2000     m014    2.0      m       014
2      AF  2000     m014   52.0      m       014
3      AG  2000     m014    0.0      m       014
4      AL  2000     m014    2.0      m       014
```

Splitting a column with .split() and .get()

Another common way multiple variables are stored in columns is with a delimiter. You'll learn how to deal with such cases in this exercise, using a dataset consisting of Ebola cases and death counts by state and country. It has been pre-loaded into a DataFrame as ebola.

Print the columns of ebola in the IPython Shell using ebola.columns. Notice that the data has column names such as Cases_Guinea and Deaths*Guinea. Here, the underscore* serves as a delimiter between the first part (cases or deaths), and the second part (country).

This time, you cannot directly slice the variable by position as in the previous exercise. You now need to use Python's built-in string method called .split(). By default, this method will split a string into parts separated by a space. However, in this case you want it to split by an underscore. You can do this on Cases_Guinea, for example, using Cases*Guinea.split("*), which returns the list ['Cases', 'Guinea'].

The next challenge is to extract the first element of this list and assign it to a type variable, and the second element of the list to a country variable. You can accomplish this by accessing the str attribute of the column and using the .get() method to retrieve the 0 or 1 index, depending on the part you want.

In [28]:
```
import pandas as pd
ebola = pd.read_csv('ebola.csv')
print(ebola.head())
# Melt ebola: ebola_melt
ebola_melt = pd.melt(ebola, id_vars=['Date', 'Day'], var_name='type_country', value_name='counts')
print(ebola_melt.head())
# Create the 'str_split' column
ebola_melt['str_split'] = ebola_melt.type_country.str.split('_')

# Create the 'type' column
ebola_melt['type'] = ebola_melt.str_split.str.get(0)

# Create the 'country' column
ebola_melt['country'] = ebola_melt.str_split.str.get(1)

# Print the head of ebola_melt
print(ebola_melt.head())
```

```
         Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone  \
0    1/5/2015  289        2776.0            NaN            10030.0
1    1/4/2015  288        2775.0            NaN             9780.0
2    1/3/2015  287        2769.0         8166.0             9722.0
3    1/2/2015  286           NaN         8157.0                NaN
4  12/31/2014  284        2730.0         8115.0             9633.0

   Cases_Nigeria  Cases_Senegal  Cases_UnitedStates  Cases_Spain  Cases_Mali  \
0            NaN            NaN                 NaN          NaN         NaN
1            NaN            NaN                 NaN          NaN         NaN
2            NaN            NaN                 NaN          NaN         NaN
3            NaN            NaN                 NaN          NaN         NaN
4            NaN            NaN                 NaN          NaN         NaN

   Deaths_Guinea  Deaths_Liberia  Deaths_SierraLeone  Deaths_Nigeria  \
0         1786.0             NaN              2977.0             NaN
1         1781.0             NaN              2943.0             NaN
2         1767.0          3496.0              2915.0             NaN
3            NaN          3496.0                 NaN             NaN
4         1739.0          3471.0              2827.0             NaN

   Deaths_Senegal  Deaths_UnitedStates  Deaths_Spain  Deaths_Mali
0             NaN                  NaN           NaN          NaN
1             NaN                  NaN           NaN          NaN
```

```
2            NaN                 NaN           NaN          NaN
3            NaN                 NaN           NaN          NaN
4            NaN                 NaN           NaN          NaN
         Date  Day  type_country  counts
0    1/5/2015  289  Cases_Guinea  2776.0
1    1/4/2015  288  Cases_Guinea  2775.0
2    1/3/2015  287  Cases_Guinea  2769.0
3    1/2/2015  286  Cases_Guinea     NaN
4  12/31/2014  284  Cases_Guinea  2730.0
         Date  Day  type_country  counts        str_split   type country
0    1/5/2015  289  Cases_Guinea  2776.0  [Cases, Guinea]  Cases  Guinea
1    1/4/2015  288  Cases_Guinea  2775.0  [Cases, Guinea]  Cases  Guinea
2    1/3/2015  287  Cases_Guinea  2769.0  [Cases, Guinea]  Cases  Guinea
3    1/2/2015  286  Cases_Guinea     NaN  [Cases, Guinea]  Cases  Guinea
4  12/31/2014  284  Cases_Guinea  2730.0  [Cases, Guinea]  Cases  Guinea
```

Combining data for analysis

The ability to transform and combine your data is a crucial skill in data science, because your data may not always come in one monolithic file or table for you to load. A large dataset may be broken into separate datasets to facilitate easier storage and sharing. Or if you are dealing with time series data, for example, you may have a new dataset for each day. No matter the reason, it is important to be able to combine datasets so you can either clean a single dataset, or clean each dataset separately and then combine them later so you can run your analysis on a single dataset. In this chapter, you'll learn all about combining data.

Combining rows of data

The dataset you'll be working with here relates to NYC Uber data. The original dataset has all the originating Uber pickup locations by time and latitude and longitude. For didactic purposes, you'll be working with a very small portion of the actual data.

Three DataFrames have been pre-loaded: uber1, which contains data for April 2014, uber2, which contains data for May 2014, and uber3, which contains data for June 2014. Your job in this exercise is to concatenate these DataFrames together such that the resulting DataFrame has the data for all three months.

Begin by exploring the structure of these three DataFrames in the IPython Shell using methods such as .head().

In [30]:
```
import pandas as pd
uber1 = pd.read_csv('uber1.csv', index_col = 0)
uber2 = pd.read_csv('uber2.csv', index_col = 0)
uber3 = pd.read_csv('uber3.csv', index_col = 0)
# Concatenate uber1, uber2, and uber3: row_concat
row_concat = pd.concat([uber1, uber2, uber3])

# Print the shape of row_concat
print(row_concat.shape)

# Print the head of row_concat
print(row_concat.head())
```

```
(297, 4)
       Date/Time      Lat      Lon    Base
0  4/1/2014 0:11  40.7690 -73.9549  B02512
1  4/1/2014 0:17  40.7267 -74.0345  B02512
2  4/1/2014 0:21  40.7316 -73.9873  B02512
3  4/1/2014 0:28  40.7588 -73.9776  B02512
4  4/1/2014 0:33  40.7594 -73.9722  B02512
```

Combining columns of data

Think of column-wise concatenation of data as stitching data together from the sides instead of the top and bottom. To perform this action, you use the same pd.concat() function, but this time with the keyword argument axis=1. The default, axis=0, is for a row-wise concatenation.

You'll return to the Ebola dataset you worked with briefly in the last chapter. It has been pre-loaded into a DataFrame called ebola_melt. In this DataFrame, the status and country of a patient is contained in a single column. This column has been parsed into a new DataFrame, status_country, where there are separate columns for status and country.

Explore the ebola_melt and status_country DataFrames in the IPython Shell. Your job is to concatenate them column-wise in order to obtain a final, clean DataFrame.

In [51]:
```
import pandas as pd
ebola = pd.read_csv('ebola.csv')
# Melt ebola: ebola_melt
ebola_melt = pd.melt(ebola, id_vars=['Date', 'Day'], var_name='type_country', value_name='counts')
ebola_melt['str_split'] = ebola_melt.type_country.str.split('_')
# Create the 'type' column
ebola_melt['type'] = ebola_melt.str_split.str.get(0)
# Create the 'country' column
ebola_melt['country'] = ebola_melt.str_split.str.get(1)
status_country = ebola_melt[['type','country']]
ebola_melt = pd.melt(ebola, id_vars=['Date', 'Day'], var_name='type_country', value_name='counts')
#redo ebola_melt to have the same data as Data Camp.
# print(status_country.head())
# print(ebola_melt.head())

# Concatenate ebola_melt and status_country column-wise: ebola_tidy
ebola_tidy = pd.concat([ebola_melt, status_country], axis=1) #axis = 1  is the key for column concatenation.

# Print the shape of ebola_tidy
print(ebola_tidy.shape)

# Print the head of ebola_tidy
print(ebola_tidy.head())
```

```
(1952, 6)
         Date  Day  type_country  counts   type country
0    1/5/2015  289  Cases_Guinea  2776.0  Cases  Guinea
1    1/4/2015  288  Cases_Guinea  2775.0  Cases  Guinea
2    1/3/2015  287  Cases_Guinea  2769.0  Cases  Guinea
3    1/2/2015  286  Cases_Guinea     NaN  Cases  Guinea
4  12/31/2014  284  Cases_Guinea  2730.0  Cases  Guinea
```

Finding files that match a pattern

You're now going to practice using the glob module to find all csv files in the workspace. In the next exercise, you'll programmatically load them into DataFrames.

As Dan showed you in the video, the glob module has a function called glob that takes a pattern and returns a list of the files in the working directory that match that pattern.

For example, if you know the pattern is part *single digit number .csv, you can write the pattern as 'part*?.csv' (which would match part_1.csv, part_2.csv, part_3.csv, etc.)

Similarly, you can find all .csv files with '*.csv', or all parts with 'part_*'. The ? wildcard represents any 1 character, and the * wildcard represents any number of characters.

In [54]:
```
# Import necessary modules
import glob
import pandas as pd

# Write the pattern: pattern
pattern = 'uber*.csv'

# Save all file matches: csv_files
csv_files = glob.glob(pattern)
#print(type(csv_files))

# Print the file names
print(csv_files)

# Load the second file into a DataFrame: csv2
csv2 = pd.read_csv(csv_files[1])

# Print the head of csv2
print(csv2.head())
```

```
<class 'list'>
['uber1.csv', 'uber2.csv', 'uber3.csv']
   Unnamed: 0      Date/Time      Lat      Lon    Base
0           0  5/1/2014 0:02  40.7521 -73.9914  B02512
1           1  5/1/2014 0:06  40.6965 -73.9715  B02512
2           2  5/1/2014 0:15  40.7464 -73.9838  B02512
3           3  5/1/2014 0:17  40.7463 -74.0011  B02512
4           4  5/1/2014 0:17  40.7594 -73.9734  B02512
```

Iterating and concatenating all matches

Now that you have a list of filenames to load, you can load all the files into a list of DataFrames that can then be concatenated.

You'll start with an empty list called frames. Your job is to use a for loop to iterate through each of the filenames, read each filename into a DataFrame, and then append it to the frames list.

You can then concatenate this list of DataFrames using pd.concat(). Go for it!

In [57]:
```
# Create an empty list: frames
frames = []

#  Iterate over csv_files
for csv in csv_files: #csv_files is a list generated before.

    #  Read csv into a DataFrame: df
    df = pd.read_csv(csv)

    # Append df to frames
    frames.append(df)

# Concatenate frames into a single DataFrame: uber
uber = pd.concat(frames)

# Print the shape of uber
print(uber.shape)

# Print the head of uber
print(uber.head())
```

```
(297, 5)
   Unnamed: 0      Date/Time      Lat      Lon    Base
0           0  4/1/2014 0:11  40.7690 -73.9549  B02512
1           1  4/1/2014 0:17  40.7267 -74.0345  B02512
2           2  4/1/2014 0:21  40.7316 -73.9873  B02512
3           3  4/1/2014 0:28  40.7588 -73.9776  B02512
4           4  4/1/2014 0:33  40.7594 -73.9722  B02512
```

1-to-1 data merge

Merging data allows you to combine disparate datasets into a single dataset to do more complex analysis.

Here, you'll be using survey data that contains readings that William Dyer, Frank Pabodie, and Valentina Roerich took in the late 1920 and 1930 while they were on an expedition towards Antarctica. The dataset was taken from a sqlite database from the Software Carpentry SQL lesson.

Two DataFrames have been pre-loaded: site and visited. Explore them in the IPython Shell and take note of their structure and column names. Your task is to perform a 1-to-1 merge of these two DataFrames using the 'name' column of site and the 'site' column of visited.

Check the following for how pandas achieve the same purpose of common sql commands. If I don't have a database server in hand, I can check the link to use pandas to realize the data manipulating with pandas.
https://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html (https://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html)

In [ ]:
```
# Merge the DataFrames: o2o
o2o = pd.merge(left=site, right=visited, left_on='name', right_on='site')

# Print o2o
print(o2o)

#See output below
```

site name lat long 0 DR-1 -49.85 -128.57 1 DR-3 -47.15 -126.72 2 MSK-4 -48.87 -123.40

visited ident site dated 0 619 DR-1 1927-02-08 1 734 DR-3 1939-01-07 2 837 MSK-4 1932-01-14

o2o name lat long ident site dated 0 DR-1 -49.85 -128.57 619 DR-1 1927-02-08 1 DR-3 -47.15 -126.72 734 DR-3 1939-01-07 2 MSK-4 -48.87 -123.40 837 MSK-4 1932-01-14

Many-to-1 data merge

In a many-to-one (or one-to-many) merge, one of the values will be duplicated and recycled in the output. That is, one of the keys in the merge is not unique.

Here, the two DataFrames site and visited have been pre-loaded once again. Note that this time, visited has multiple entries for the site column. Confirm this by exploring it in the IPython Shell.

The .merge() method call is the same as the 1-to-1 merge from the previous exercise, but the data and output will be different.

In [ ]:
```
# Merge the DataFrames: m2o
m2o = pd.merge(left=site, right=visited, left_on='name', right_on='site')

# Print m2o
print(m2o)
```

site name lat long 0 DR-1 -49.85 -128.57 1 DR-3 -47.15 -126.72 2 MSK-4 -48.87 -123.40

visited ident site dated 0 619 DR-1 1927-02-08 1 622 DR-1 1927-02-10 2 734 DR-3 1939-01-07 3 735 DR-3 1930-01-12 4 751 DR-3 1930-02-26 5 752 DR-3 NaN 6 837 MSK-4 1932-01-14 7 844 DR-1 1932-03-22

m2o name lat long ident site dated 0 DR-1 -49.85 -128.57 619 DR-1 1927-02-08 1 DR-1 -49.85 -128.57 622 DR-1 1927-02-10 2 DR-1 -49.85 -128.57 844 DR-1 1932-03-22 3 DR-3 -47.15 -126.72 734 DR-3 1939-01-07 4 DR-3 -47.15 -126.72 735 DR-3 1930-01-12 5 DR-3 -47.15 -126.72 751 DR-3 1930-02-26 6 DR-3 -47.15 -126.72 752 DR-3 NaN 7 MSK-4 -48.87 -123.40 837 MSK-4 1932-01-14

Many-to-many data merge

The final merging scenario occurs when both DataFrames do not have unique keys for a merge. What happens here is that for each duplicated key, every pairwise combination will be created.

Two example DataFrames that share common key values have been pre-loaded: df1 and df2. Another DataFrame df3, which is the result of df1 merged with df2, has been pre-loaded. All three DataFrames have been printed - look at the output and notice how pairwise combinations have been created. This example is to help you develop your intuition for many-to-many merges.

Here, you'll work with the site and visited DataFrames from before, and a new survey DataFrame. Your task is to merge site and visited as you did in the earlier exercises. You will then merge this merged DataFrame with survey.

Begin by exploring the site, visited, and survey DataFrames in the IPython Shell.

In [ ]:
```
# Merge site and visited: m2m
m2m = pd.merge(left=site, right=visited, left_on='name', right_on='site')

# Merge m2m and survey: m2m
m2m = pd.merge(left=m2m, right=survey, left_on='ident', right_on='taken')

# Print the first 20 lines of m2m
print(m2m.head(20))
```

Cleaning data for analysis

Here, you'll dive into some of the grittier aspects of data cleaning. You'll learn about string manipulation and pattern matching to deal with unstructured data, and then explore techniques to deal with missing or duplicate data. You'll also learn the valuable skill of programmatically checking your data for consistency, which will give you confidence that your code is running correctly and that the results of your analysis are reliable!

Converting data types

In this exercise, you'll see how ensuring all categorical variables in a DataFrame are of type category reduces memory usage.

The tips dataset has been loaded into a DataFrame called tips. This data contains information about how much a customer tipped, whether the customer was male or female, a smoker or not, etc.

Look at the output of tips.info() in the IPython Shell. You'll note that two columns that should be categorical - sex and smoker - are instead of type object, which is pandas' way of storing arbitrary strings. Your job is to convert these two columns to type category and note the reduced memory usage.

In [58]:
```
import pandas as pd
tips = pd.read_csv('tips.csv')
# Convert the sex column to type 'category'
tips.sex = tips.sex.astype('category')

# Convert the smoker column to type 'category'
tips.smoker = tips.smoker.astype('category')

# Print the info of tips
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
total_bill    244 non-null float64
tip           244 non-null float64
sex           244 non-null category
smoker        244 non-null category
day           244 non-null object
time          244 non-null object
size          244 non-null int64
dtypes: category(2), float64(2), int64(1), object(2)
memory usage: 10.3+ KB
None
```

Working with numeric data

If you expect the data type of a column to be numeric (int or float), **but instead it is of type object**, this typically means that there is a non numeric value in the column, which also signifies bad data.

You can use the pd.to_numeric() function to convert a column into a numeric data type. If the function raises an error, you can be sure that there is a bad value within the column. **You can either use the techniques you learned in Chapter 1 to do some exploratory data analysis and find the bad value, or you can choose to ignore or coerce the value into a missing value, NaN.**

A modified version of the tips dataset has been pre-loaded into a DataFrame called tips. For instructional purposes, it has been pre-processed to introduce some 'bad' data for you to clean. Use the .info() method to explore this. You'll note that the total_bill and tip columns, which should be numeric, are instead of type object. Your job is to fix this.

The following DataFrame has no bad data, so the results are different from DataCamp.

In [59]:
```
# Convert 'total_bill' to a numeric dtype
tips['total_bill'] = pd.to_numeric(tips['total_bill'], errors='coerce')

# Convert 'tip' to a numeric dtype
tips['tip'] = pd.to_numeric(tips['tip'], errors='coerce')

# Print the info of tips
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
total_bill    244 non-null float64
tip           244 non-null float64
sex           244 non-null category
smoker        244 non-null category
day           244 non-null object
time          244 non-null object
size          244 non-null int64
dtypes: category(2), float64(2), int64(1), object(2)
memory usage: 10.3+ KB
None
```

String parsing with regular expressions

In the video, Dan introduced you to the basics of regular expressions, which are powerful ways of defining patterns to match strings. This exercise will get you started with writing them.

When working with data, it is sometimes necessary to write a regular expression to look for properly entered values. Phone numbers in a dataset is a common field that needs to be checked for validity. Your job in this exercise is to define a regular expression to match US phone numbers that fit the pattern of xxx-xxx-xxxx.

The regular expression module in python is re. When performing pattern matching on data, since the pattern will be used for a match across multiple rows, it's better to compile the pattern first using re.compile(), and then use the compiled pattern to match values.

Import re. Compile a pattern that matches a phone number of the format xxx-xxx-xxxx. Use \d{x} to match x digits. Here you'll need to use it three times: twice to match 3 digits, and once to match 4 digits. Place the regular expression inside re.compile(). Using the .match() method on prog, check whether the pattern matches the string '123-456-7890'. Using the same approach, now check whether the pattern matches the string '1123-456-7890'.

In [60]:
```
# Import the regular expression module
import re

# Compile the pattern: prog
prog = re.compile('\d{3}-\d{3}-\d{4}')

# See if the pattern matches
result = prog.match('123-456-7890')
print(bool(result))

# See if the pattern matches
result2 = prog.match('1123-456-7890')
print(bool(result2))
```

```
True
False
```

Extracting numerical values from strings

Extracting numbers from strings is a common task, particularly when working with unstructured data or log files.

Say you have the following string: 'the recipe calls for 6 strawberries and 2 bananas'.

It would be useful to extract the 6 and the 2 from this string to be saved for later use when comparing strawberry to banana ratios.

When using a regular expression to extract multiple numbers (or multiple pattern matches, to be exact), you can use the re.findall() function. Dan did not discuss this in the video, but it is straightforward to use: You pass in a pattern and a string to re.findall(), and it will return a list of the matches.

In [61]:
```
# Import the regular expression module
import re

# Find the numeric values: matches
matches = re.findall('\d+', 'the recipe calls for 10 strawberries and 1 banana')

# Print the matches
print(matches)
```

```
['10', '1']
```

Pattern matching

In this exercise, you'll continue practicing your regular expression skills. For each provided string, your job is to write the appropriate pattern to match it.

INSTRUCTIONS Write patterns to match: A telephone number of the format xxx-xxx-xxxx. You already did this in a previous exercise. A string of the format: A dollar sign, an arbitrary number of digits, a decimal point, 2 digits. Use $ to match the dollar sign, \d *to match an arbitrary number of digits, . to match the decimal point, and \d{x} to match x number of digits. A capital letter, followed by an arbitrary number of alphanumeric characters. Use [A-Z] to match any capital letter followed by \w* to match an arbitrary number of alphanumeric characters.

In [63]:
```
# Write the first pattern
pattern1 = bool(re.match(pattern='\d{3}-\d{3}-\d{4}', string='123-456-7890'))
print(pattern1)

# Write the second pattern
pattern2 = bool(re.match(pattern='\$\d*\.\d{2}', string='$123.45'))
# need escape $ because it has special meaning: end of string
# need escape . because
print(pattern2)

# Write the third pattern
pattern3 = bool(re.match(pattern='[A-Z]\w*', string='Australia'))
print(pattern3)
```

```
True
True
False
```

Custom functions to clean data

You'll now practice writing functions to clean data.

The tips dataset has been pre-loaded into a DataFrame called tips. It has a 'sex' column that contains the values 'Male' or 'Female'. Your job is to write a function that will recode 'Male' to 1, 'Female' to 0, and return np.nan for all entries of 'sex' that are neither 'Male' nor 'Female'.

Recoding variables like this is a common data cleaning task. Functions provide a mechanism for you to abstract away complex bits of code as well as reuse code. This makes your code more readable and less error prone.

As Dan showed you in the videos, you can use the .apply() method to apply a function across entire rows or columns of DataFrames. However, note that each column of a DataFrame is a pandas Series. Functions can also be applied across Series. Here, you will apply your function over the 'sex' column.

apply seems similar to R

In [66]:
```
import pandas as pd
tips = pd.read_csv('tips.csv')
# Define recode_sex()
def recode_sex(sex_value):

    # Return 1 if sex_value is 'Male'
    if sex_value == 'Male':
        return 1

    # Return 0 if sex_value is 'Female'
    elif sex_value == 'Female':
        return 0

    # Return np.nan
    else:
        return np.nan

# Apply the function to the sex column
print(tips.head())
tips['sex_recode'] = tips.sex.apply(recode_sex)

# Print the first five rows of tips
print(tips.head())
```

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
   total_bill   tip     sex smoker  day    time  size  sex_recode
0       16.99  1.01  Female     No  Sun  Dinner     2           0
1       10.34  1.66    Male     No  Sun  Dinner     3           1
2       21.01  3.50    Male     No  Sun  Dinner     3           1
3       23.68  3.31    Male     No  Sun  Dinner     2           1
4       24.59  3.61  Female     No  Sun  Dinner     4           0
```

Lambda functions

You'll now be introduced to a powerful Python feature that will help you clean your data more effectively: lambda functions. Instead of using the def syntax that you used in the previous exercise, lambda functions let you make simple, one-line functions.

For example, here's a function that squares a variable used in an .apply() method:

def my_square(x): return x ** 2

df.apply(my_square) The equivalent code using a lambda function is:

df.apply(lambda x: x ** 2) The lambda function takes one parameter - the variable x. The function itself just squares x and returns the result, which is whatever the one line of code evaluates to. In this way, lambda functions can make your code concise and Pythonic.

The tips dataset has been pre-loaded into a DataFrame called tips. Your job is to clean its 'total_dollar' column by removing the dollar sign. You'll do this using two different methods: With the .replace() method, and with regular expressions. The regular expression module re has been pre-imported.

Use the .replace() method inside a lambda function to remove the dollar sign from the 'total_dollar' column of tips. You need to specify two arguments to the .replace() method: The string to be replaced ('$'), and the string to replace it by (''). Apply the lambda function over the 'total_dollar' column of tips. Use a regular expression to remove the dollar sign from the 'total_dollar' column of tips. The pattern has been provided for you: It is the first argument of the re.findall() function. Complete the rest of the lambda function and apply it over the 'total_dollar' column of tips. Notice that because

re.findall() returns a list, **you have to slice it in order to access the actual value.** Hit 'Submit Answer' to verify that you have removed the dollar sign from the column.

In [ ]:
```
# Write the lambda function using replace
tips['total_dollar_replace'] = tips.total_dollar.apply(lambda x: x.replace('$', ''))

# Write the lambda function using regular expressions
tips['total_dollar_re'] = tips.total_dollar.apply(lambda x: re.findall('\d+\.\d+', x)[0])

# Print the head of tips
print(tips.head())
```

Dropping duplicate data

Duplicate data causes a variety of problems. From the point of view of performance, they use up unnecessary amounts of memory and cause unneeded calculations to be performed when processing data. In addition, they can also bias any analysis results.

A dataset consisting of the performance of songs on the Billboard charts has been pre-loaded into a DataFrame called billboard. Check out its columns in the IPython Shell. Your job in this exercise is to subset this DataFrame and then drop all duplicate rows.

This should be a must in cleaning data.

In [ ]:
```
# Create the new DataFrame: tracks
tracks = billboard[['year', 'artist', 'track', 'time']]

# Print info of tracks
print(tracks.info())

# Drop the duplicates: tracks_no_duplicates
tracks_no_duplicates = tracks.drop_duplicates()

# Print info of tracks_no_duplicates
print(tracks_no_duplicates.info())
```

Filling missing data

Here, you'll return to the airquality dataset from Chapter 2. It has been pre-loaded into the DataFrame airquality, and it has missing values for you to practice filling in. Explore airquality in the IPython Shell to checkout which columns have missing values.

It's rare to have a (real-world) dataset without any missing values, and it's important to deal with them because certain calculations cannot handle missing values while some calculations will, by default, skip over any missing values.

Also, understanding how much missing data you have, and thinking about where it comes from is crucial to making unbiased interpretations of data.

In [ ]:
```
# Calculate the mean of the Ozone column: oz_mean
oz_mean = airquality.Ozone.mean()

# Replace all the missing values in the Ozone column with the mean
airquality['Ozone'] = airquality.Ozone.fillna(oz_mean)

# Print the info of airquality
print(airquality.info())
```

Testing your data with asserts

Here, you'll practice writing assert statements using the Ebola dataset from previous chapters to programmatically check for missing values and to confirm that all values are positive. The dataset has been pre-loaded into a DataFrame called ebola.

In the video, you saw Dan use the .all() method together with the .notnull() DataFrame method to check for missing values in a column. The .all() method returns True if all values are True. When used on a DataFrame, it returns a Series of Booleans - one for each column in the DataFrame. So if you are using it on a DataFrame, like in this exercise, you need to chain another .all() method so that you return only one True or False value. When using these within an assert statement, nothing will be returned if the assert statement is true: This is how you can confirm that the data you are checking are valid.

Note: You can use pd.notnull(df) as an alternative to df.notnull().

In [ ]:
```
# Assert that there are no missing values
assert pd.notnull(ebola).all().all()

# Assert that all values are >= 0
assert (ebola >= 0).all().all()
```

Case study

In this final chapter, you'll apply all of the data cleaning techniques you've learned in this course towards tidying a real-world, messy dataset obtained from the Gapminder Foundation. Once you're done, not only will you have a clean and tidy dataset, you'll also be ready to start working on your own data science projects using the power of Python!

Exploratory analysis

Whenever you obtain a new dataset, your first task should always be to do some exploratory analysis to get a better understanding of the data and diagnose it for any potential issues.

The Gapminder data for the 19th century has been loaded into a DataFrame called g1800s. In the IPython Shell, use pandas methods such as .head(), .info(), and .describe(), and DataFrame attributes like .columns and .shape to explore it.

Use the information that you acquire from your exploratory analysis to choose the true statement from the options provided below.

In [89]:
```
import pandas as pd
g1800s = pd.read_csv("g1800s.csv")
g1800s.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 260 entries, 0 to 259
Columns: 101 entries, Life expectancy to 1899
dtypes: float64(100), object(1)
memory usage: 205.2+ KB
```

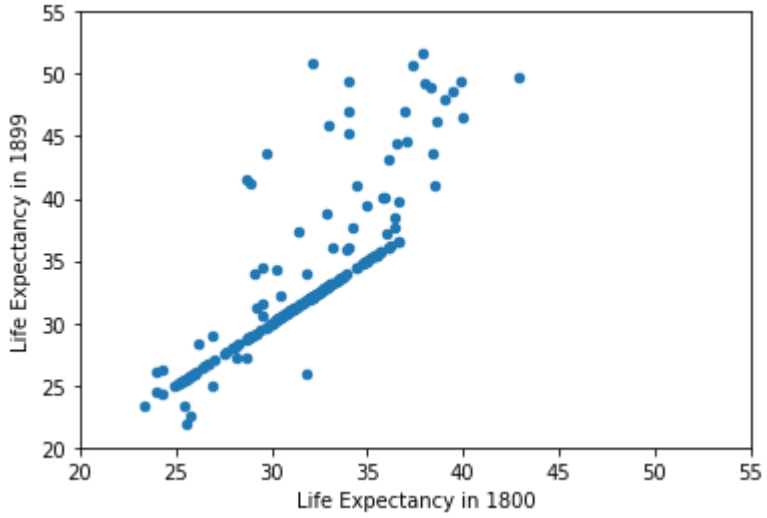
Visualizing your data

Since 1800, life expectancy around the globe has been steadily going up. You would expect the Gapminder data to confirm this.

The DataFrame g1800s has been pre-loaded. Your job in this exercise is to create a scatter plot with life expectancy in '1800' on the x-axis and life expectancy in '1899' on the y-axis.

Here, the goal is to visually check the data for insights as well as errors. When looking at the plot, pay attention to whether the scatter plot takes the form of a diagonal line, and which points fall below or above the diagonal line. This will inform how life expectancy in 1899 changed (or did not change) compared to 1800 for different countries. If points fall on a diagonal line, it means that life expectancy remained the same!

In [90]:
```
# Import matplotlib.pyplot
import matplotlib.pyplot as plt
import pandas as pd
g1800s = pd.read_csv("g1800s.csv")
# Create the scatter plot
g1800s.plot(kind='scatter', x='1800', y='1899')

# Specify axis labels
plt.xlabel('Life Expectancy in 1800')
plt.ylabel('Life Expectancy in 1899')

# Specify axis limits
plt.xlim(20, 55)
plt.ylim(20, 55)

# Display the plot
plt.show()
```



Excellent work! As you can see, there are a surprising number of countries that fall on the diagonal line. In fact, examining the DataFrame reveals that the life expectancy for 140 of the 260 countries did not change at all in the 19th century! This is possibly a result of not having access to the data for all the years back then. In this way, visualizing your data can help you uncover insights as well as diagnose it for errors.

Thinking about the question at hand

Since you are given life expectancy level data by country and year, you could ask questions about how much the average life expectancy changes over each year.

Before continuing, however, it's important to make sure that the following assumptions about the data are true:

'Life expectancy' is the first column (index 0) of the DataFrame. The other columns contain either null or numeric values. The numeric values are all greater than or equal to 0. There is only one instance of each country. You can write a function that you can apply over the entire DataFrame to verify some of these assumptions. Note that spending the time to write such a script will help you when working with other datasets as well.

Define a function called check_null_or_valid() that takes in one argument: row_data. Inside the function, convert no_na to a numeric data type using pd.to_numeric(). Write an assert statement to make sure the first column (index 0) of the g1800s DataFrame is 'Life expectancy'. Write an assert statement to test that all the values are valid for the g1800s DataFrame. Use the check_null_or_valid() function placed inside the .apply() method for this. Note that because you're applying it over the entire DataFrame, and not just one column, you'll have to chain the .all() method twice, and remember that you don't have to use () for functions placed inside .apply(). Write an assert statement to make sure that each country occurs only once in the data. Use the .value_counts() method on the 'Life expectancy' column for this. Specifically, index 0 of .value_counts() will contain the most frequently occuring value. If this is equal to 1 for the 'Life expectancy' column, then you can be certain that no country appears more than once in the data.

In [93]:
```
def check_null_or_valid(row_data):
    """Function that takes a row of data,
    drops all missing values,
    and checks if all remaining values are greater than or equal to 0
    """
    no_na = row_data.dropna()[1:-1]
    numeric = pd.to_numeric(no_na)
    ge0 = numeric >= 0
    return ge0

# Check whether the first column is 'Life expectancy'
assert g1800s.columns[0] == 'Life expectancy'

# Check whether the values in the row are valid
assert g1800s.iloc[:, 1:].apply(check_null_or_valid, axis=1).all().all()

# Check that there is only one instance of each country
assert g1800s['Life expectancy'].value_counts()[0] == 1
```

Assembling your data

Here, three DataFrames have been pre-loaded: g1800s, g1900s, and g2000s. These contain the Gapminder life expectancy data for, respectively, the 19th century, the 20th century, and the 21st century.

Your task in this exercise is to concatenate them into a single DataFrame called gapminder. This is a row-wise concatenation, similar to how you concatenated the monthly Uber datasets in Chapter 3.

In [96]:
```
import pandas as pd
g2000s = pd.read_csv("g2000s.csv")
g2000s.shape

g1900s = pd.read_csv("g1900s.csv")
g1900s.shape
# Concatenate the DataFrames row-wise
gapminder = pd.concat([g1800s, g1900s, g2000s])

# Print the shape of gapminder
print(gapminder.shape)

# Print the head of gapminder
print(gapminder.head())
```

```
(780, 218)
    1800   1801   1802   1803   1804   1805   1806   1807   1808   1809  \
0    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
1  28.21  28.20  28.19  28.18  28.17  28.16  28.15  28.14  28.13  28.12
2    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
3  35.40  35.40  35.40  35.40  35.40  35.40  35.40  35.40  35.40  35.40
4  28.82  28.82  28.82  28.82  28.82  28.82  28.82  28.82  28.82  28.82

           ...            2008  2009  2010  2011  2012  2013  2014  2015  \
0          ...             NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
1          ...             NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
2          ...             NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
3          ...             NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
4          ...             NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN

   2016        Life expectancy
0   NaN               Abkhazia
1   NaN            Afghanistan
2   NaN  Akrotiri and Dhekelia
3   NaN                Albania
4   NaN                Algeria

[5 rows x 218 columns]
```

In [97]:
```
# Melt gapminder: gapminder_melt
gapminder_melt = pd.melt(gapminder, id_vars='Life expectancy')

# Rename the columns
gapminder_melt.columns = ['country', 'year', 'life_expectancy']

# Print the head of gapminder_melt
print(gapminder_melt.head())
```

```
                country  year  life_expectancy
0               Abkhazia  1800              NaN
1            Afghanistan  1800            28.21
2  Akrotiri and Dhekelia  1800              NaN
3                Albania  1800            35.40
4                Algeria  1800            28.82
```

Checking the data types

Now that your data is in the proper shape, you need to ensure that the columns are of the proper data type. That is, you need to ensure that country is of type object, year is of type int64, and life_expectancy is of type float64.

The tidy DataFrame has been pre-loaded as gapminder. Explore it in the IPython Shell using the .info() method. Notice that the column 'year' is of type object. This is incorrect, so you'll need to use the pd.to_numeric() function to convert it to a numeric data type.

NumPy and pandas have been pre-imported as np and pd.

In [101]:
```
import numpy as np
# Convert the year column to numeric
gapminder_melt.year = pd.to_numeric(gapminder_melt.year)

# Test if country is of type object
assert gapminder_melt.country.dtypes == np.object

# Test if year is of type int64
assert gapminder_melt.year.dtypes == np.int64

# Test if life_expectancy is of type float64
assert gapminder_melt.life_expectancy.dtypes == np.float64
```

Looking at country spellings

Having tidied your DataFrame and checked the data types, your next task in the data cleaning process is to look at the 'country' column to see if there are any special or invalid characters you may need to deal with.

It is reasonable to assume that country names will contain:

The set of lower and upper case letters. Whitespace between words. Periods for any abbreviations. To confirm that this is the case, you can leverage the power of regular expressions again. For common operations like this, Python has a built-in string method - str.contains() - which takes a regular expression pattern, and applies it to the Series, returning True if there is a match, and False otherwise.

Since here you want to find the values that do not match, you have to invert the boolean, which can be done using ~. This Boolean series can then be used to get the Series of countries that have invalid names.

In [102]:
```
# Create the series of countries: countries
countries = gapminder_melt['country']

# Drop all the duplicates from countries
countries = countries.drop_duplicates()

# Write the regular expression: pattern
pattern = '^[A-Za-z\.\s]*$'

# Create the Boolean vector: mask
mask = countries.str.contains(pattern)

# Invert the mask: mask_inverse
mask_inverse = ~mask

# Subset countries using mask_inverse: invalid_countries
invalid_countries = countries.loc[mask_inverse]

# Print invalid_countries
print(invalid_countries)
```

```
49            Congo, Dem. Rep.
50                 Congo, Rep.
53               Cote d'Ivoire
73       Falkland Is (Malvinas)
93                Guinea-Bissau
98              Hong Kong, China
118    United Korea (former)\n
131                Macao, China
132              Macedonia, FYR
145       Micronesia, Fed. Sts.
161             Ngorno-Karabakh
187              St. Barthélemy
193      St.-Pierre-et-Miquelon
225                 Timor-Leste
251       Virgin Islands (U.S.)
252        North Yemen (former)
253        South Yemen (former)
258                      Åland
Name: country, dtype: object
```

Excellent work! As you can see, not all these country names are actually invalid so maybe the assumptions need to be tweaked a little. However, there certainly are a few cases worth further investigation, such as St. Barth?lemy. Whenever you are dealing with columns of raw data consisting of strings, it is important to check them for consistency like this.

More data cleaning and processing

It's now time to deal with the missing data. There are several strategies for this: You can drop them, fill them in using the mean of the column or row that the missing value is in (also known as imputation), or, if you are dealing with time series data, use a forward fill or backward fill, in which you replace missing values in a column with the most recent known value in the column. See pandas Foundations for more on forward fill and backward fill.

In general, it is not the best idea to drop missing values, because in doing so you may end up throwing away useful information. In this data, the missing values refer to years where no estimate for life expectancy is available for a given country. You could fill in, or guess what these life expectancies could be by looking at the average life expectancies for other countries in that year, for example. Whichever strategy you go with, it is important to carefully consider all options and understand how they will affect your data.

In this exercise, you'll practice dropping missing values. Your job is to drop all the rows that have NaN in the life_expectancy column. Before doing so, it would be valuable to use assert statements to confirm that year and country do not have any missing values.

Begin by printing the shape of gapminder in the IPython Shell prior to dropping the missing values. Complete the exercise to find out what its shape will be after dropping the missing values!

Assert that country and year do not contain any missing values. The first assert statement has been written for you. Note the chaining of the .all() method to pd.notnull() to confirm that all values in the column are not null. Drop the rows in the data where any observation in life_expectancy is missing. As you confirmed that country and year don't have missing values, you can use the .dropna() method on the entire gapminder DataFrame, because any missing values would have to be in the life_expectancy column. The .dropna() method has the default keyword arguments axis=0 and how='any', which specify that rows with any missing values should be dropped. Print the shape of gapminder.

In [103]:
```
# Assert that country does not contain any missing values
assert pd.notnull(gapminder_melt.country).all()

# Assert that year does not contain any missing values
assert pd.notnull(gapminder_melt.year).all()

# Drop the missing values
gapminder = gapminder_melt.dropna(how='any')

# Print the shape of gapminder
print(gapminder.shape)
```

```
(20100, 3)
```

The above result is different from that of DataCamp. (43857, 3)

After dropping the missing values from 'life_expectancy', the number of rows in the DataFrame has gone down from 169260 to 43857. In general, you should avoid dropping too much of your data, but if there is no reasonable way to fill in or impute missing values, then dropping the missing data may be the best solution.

Wrapping up

Now that you have a clean and tidy dataset, you can do a bit of visualization and aggregation. In this exercise, you'll begin by creating a histogram of the life_expectancy column. You should not get any values under 0 and you should see something reasonable on the higher end of the life_expectancy age range.

Your next task is to investigate how average life expectancy changed over the years. To do this, you need to subset the data by each year, get the life_expectancy column from each subset, and take an average of the values. You can achieve this using the .groupby() method. This .groupby() method is covered in greater depth in Manipulating DataFrames with pandas.

Finally, you can save your tidy and summarized DataFrame to a file using the .to_csv() method.

Matplotlib and pandas have been pre-imported as plt and pd. Go for it!

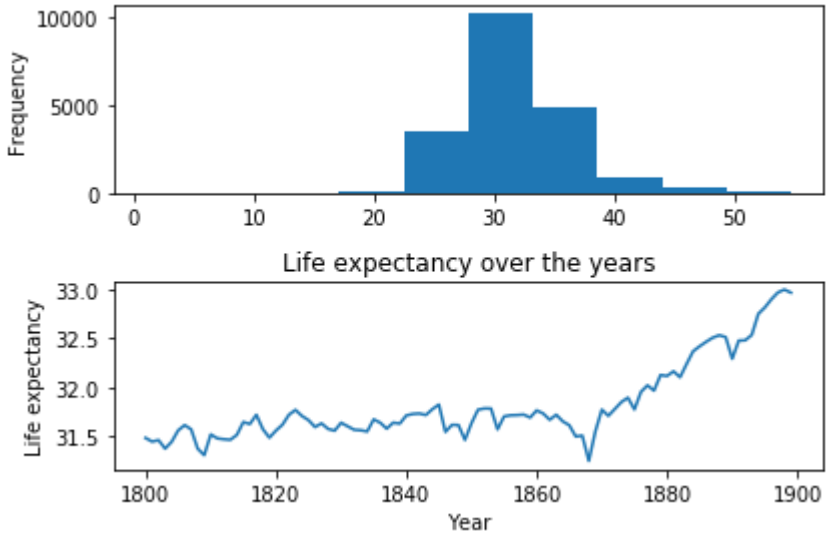
Create a histogram of the life_expectancy column using the .plot() method of gapminder. Specify kind='hist'. Group gapminder by 'year' and aggregate 'life_expectancy' by the mean. To do this: Use the .groupby() method on gapminder with 'year' as the argument. Then select 'life_expectancy' and chain the .mean() method to it. Print the head and tail of gapminder_agg. This has been done for you. Create a line

plot of average life expectancy per year by using the .plot() method (without any arguments) on gapminder_agg. Save gapminder and gapminder_agg to csv files called 'gapminder.csv' and 'gapminder_agg.csv', respectively, using the .to_csv() method.

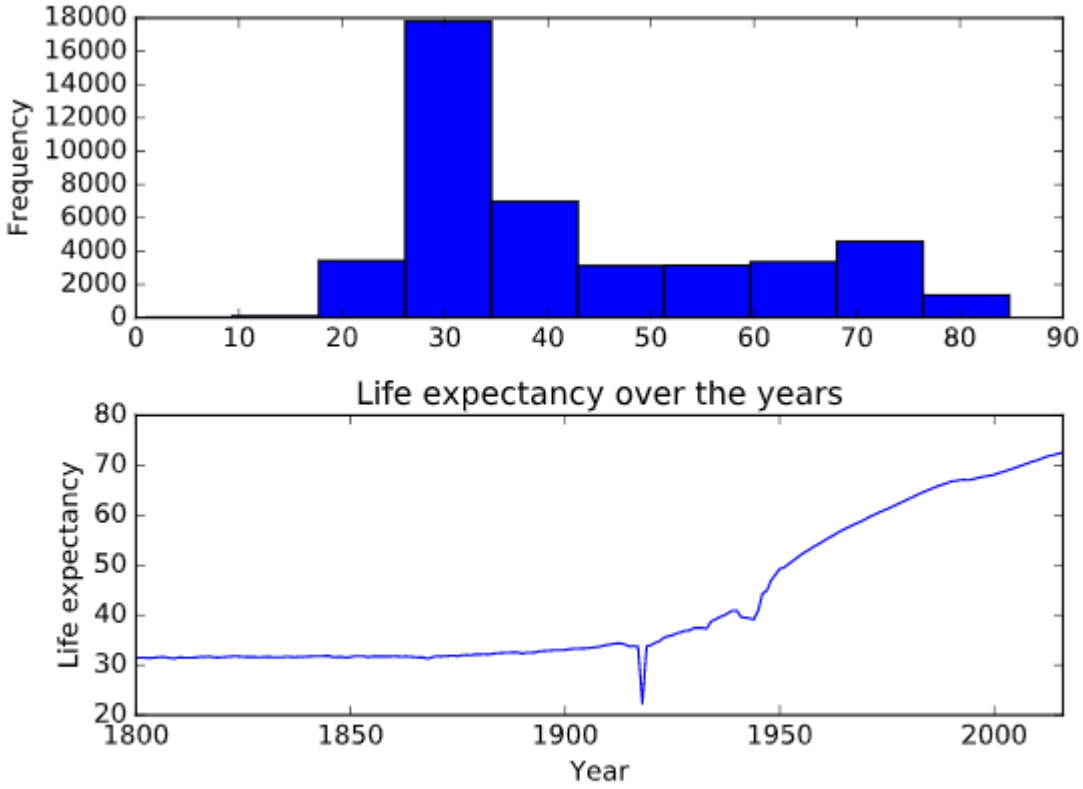
In [104]:
```
# Add first subplot
plt.subplot(2, 1, 1)

# Create a histogram of life_expectancy
gapminder_melt.life_expectancy.plot(kind='hist')

# Group gapminder: gapminder_agg
gapminder_agg = gapminder_melt.groupby('year')['life_expectancy'].mean()

# Print the head of gapminder_agg
print(gapminder_agg.head())

# Print the tail of gapminder_agg
print(gapminder_agg.tail())

# Add second subplot
plt.subplot(2, 1, 2)

# Create a line plot of life expectancy per year
gapminder_agg.plot()

# Add title and specify axis labels
plt.title('Life expectancy over the years')
plt.ylabel('Life expectancy')
plt.xlabel('Year')

# Display the plots
plt.tight_layout()
plt.show()

# Save both DataFrames to csv files
gapminder.to_csv('gapminder.csv')
gapminder_agg.to_csv('gapminder_agg.csv')
```

```
year
1800    31.486020
1801    31.448905
1802    31.463483
1803    31.377413
1804    31.446318
Name: life_expectancy, dtype: float64
```

```
year
2012   NaN
2013   NaN
2014   NaN
2015   NaN
2016   NaN
Name: life_expectancy, dtype: float64
```



The above picture is different from that of DataCamp as below.



1/1

Amazing work! You've stepped through each stage of the data cleaning process and your data is now ready for serious analysis! Looking at the line plot, it seems like life expectancy has, as expected, increased over the years. There is a surprising dip around 1920 that may be worth further investigation!