7 Pandas I: Introduction

Lab Objective: *Though NumPy and SciPy are powerful tools for numerical computing, they lack some of the high-level functionality necessary for many data science applications. Python's* pandas *library, built on NumPy, is designed specifically for data management and analysis. In this lab, we introduce pandas data structures, syntax, and explore its capabilities for quickly analyzing and presenting data.*

Series

A pandas `Series` is generalization of a one-dimensional NumPy array. Like a NumPy array, every `Series` has a data type (`dtype`), and the entries of the `Series` are all of that type. Unlike a NumPy array, every `Series` has an *index* that labels each entry, and a `Series` object can also be given a name to label the entire data set.

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

# Initialize a Series of random entries with an index of letters.
>>> pd.Series(np.random.random(4), index=['a', 'b', 'c', 'd'])
a    0.474170
b    0.106878
c    0.420631
d    0.279713
dtype: float64

# The default index is integers from 0 to the length of the data.
>>> pd.Series(np.random.random(4), name="uniform draws")
0    0.767501
1    0.614208
2    0.470877
3    0.335885
Name: uniform draws, dtype: float64
```

The index in a `Series` is a pandas object of type `Index` and is stored as the `index` attribute of the `Series`. The plain entries in the `Series` are stored as a NumPy array and can be accessed as such via the `values` attribute.

```
>>> s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'], name="some ints")

>>> s1.values                           # Get the entries as a NumPy array.
array([1, 2, 3, 4])

>>> print(s1.name, s1.dtype, sep=", ")  # Get the name and dtype.
some ints, int64

>>> s1.index                            # Get the pd.Index object.
Index(['a', 'b', 'c', 'd'], dtype='object')
```

The elements of a `Series` can be accessed by either the regular position-based integer index, or by the corresponding label in the index. New entries can be added dynamically as long as a valid index label is provided, similar to adding a new key-value pair to a dictionary. A `Series` can also be initialize from a dictionary: the keys become the index labels, and the values become the entries.

```
>>> s2 = pd.Series([10, 20, 30], index=["apple", "banana", "carrot"])
>>> s2
apple     10
banana    20
carrot    30
dtype: int64

# s2[0] and s2["apple"] refer to the same entry.
>>> print(s2[0], s2["apple"], s2["carrot"])
10 10 30

>>> s2[0] += 5              # Change the value of the first entry.
>>> s2["dewberry"] = 0      # Add a new value with label 'dewberry'.
>>> s2
apple       15
banana      20
carrot      30
dewberry     0
dtype: int64

# Initialize a Series from a dictionary.
>>> pd.Series({"eggplant":3, "fig":5, "grape":7}, name="more foods")
eggplant    3
fig         5
grape       7
Name: more foods, dtype: int64
```

Slicing and fancy indexing also work the same way in `Series` as in NumPy arrays. In addition, multiple entries of a `Series` can be selected by indexing a list of labels in the index.

```
>>> s3 = pd.Series({"lions":2, "tigers":1, "bears":3}, name="oh my")
>>> s3
bears     3
lions     2
tigers    1
Name: oh my, dtype: int64

# Get a subset of the data by regular slicing.
>>> s3[1:]
lions     2
tigers    1
Name: oh my, dtype: int64

# Get a subset of the data with fancy indexing.
>>> s3[np.array([len(i) == 5 for i in s3.index])]
bears    3
lions    2
Name: oh my, dtype: int64

# Get a subset of the data by providing several index labels.
>>> s3[ ["tigers", "bears"] ]
tigers    1                    # Note that the entries are reordered,
bears     3                    # and the name stays the same.
Name: oh my, dtype: int64
```

Problem 1. Create a pandas `Series` where the index labels are the even integers $0, 2, \ldots, 50$, and the entries are $n^2 - 1$, where n is the entry's label. Set all of the entries equal to zero whose labels are divisible by 3.

Operations with Series

A `Series` object has all of the advantages of a NumPy array, including entry-wise arithmetic, plus a few additional features (see Table 7.1). Operations between a `Series` S_1 with index I_1 and a `Series` S_2 with index I_2 results in a new `Series` with index $I_1 \cup I_2$. In other words, the index dictates how two `Series` can interact with each other.

```
>>> s4 = pd.Series([1, 2, 4], index=['a', 'c', 'd'])
>>> s5 = pd.Series([10, 20, 40], index=['a', 'b', 'd'])
>>> 2*s4 + s5
a    12.0
b     NaN                        # s4 doesn't have an entry for b, and
c     NaN                        # s5 doesn't have an entry for c, so
d    48.0                        # the combination is Nan (np.nan / None).
dtype: float64
```

Method	Returns
`abs()`	Object with absolute values taken (of numerical data)
`argmax()`	The index label of the maximum value
`argmin()`	The index label of the minimum value
`count()`	The number of non-null entries
`cumprod()`	The cumulative product over an axis
`cumsum()`	The cumulative sum over an axis
`max()`	The maximum of the entries
`mean()`	The average of the entries
`median()`	The median of the entries
`min()`	The minimum of the entries
`mode()`	The most common element(s)
`prod()`	The product of the elements
`sum()`	The sum of the elements
`var()`	The variance of the elements

Table 7.1: Numerical methods of the `Series` and `DataFrame` pandas classes.

Many `Series` are more useful than NumPy arrays primarily because of their index. For example, a `Series` can be indexed by time with a pandas `DatetimeIndex`, an index with date and/or time values. The usual way to create this kind of index is with `pd.date_range()`.

```
# Make an index of the first three days in July 2000.
>>> pd.date_range("7/1/2000", "7/3/2000", freq='D')
DatetimeIndex(['2000-07-01', '2000-07-02', '2000-07-03'],
              dtype='datetime64[ns]', freq='D')
```

Problem 2. Suppose you make an investment of d dollars in a particularly volatile stock. Every day the value of your stock goes up by \$1 with probability p, or down by \$1 with probability $1-p$ (this is an example of a *random walk*).

Write a function that accepts a probability parameter p and an initial amount of money d, defaulting to 100. Use `pd.date_range()` to create an index of the days from 1 January 2000 to 31 December 2000. Simulate the daily change of the stock by making one draw from a Bernoulli distribution with parameter p (a binomial distribution with one draw) for each day. Store the draws in a pandas `Series` with the date index and set the first draw to the initial amount d. Sum the entries cumulatively to get the stock value by day. Set any negative values to 0, then plot the series using the `plot()` method of the `Series` object.

Call your function with a few different values of p and d to observe the different possible kinds of behavior.

Note

The `Series` in Problem 2 is an example of a *time series*, since it is indexed by time. Time series show up often in data science; we will explore them in more depth in another lab.

Method	Description
`append()`	Concatenate two or more `Series`.
`drop()`	Remove the entries with the specified label or labels
`drop_duplicates()`	Remove duplicate values
`dropna()`	Drop null entries
`fillna()`	Replace null entries with a specified value or strategy
`reindex()`	Replace the index
`sample()`	Draw a random entry
`shift()`	Shift the index
`unique()`	Return unique values

Table 7.2: Methods for managing or modifying data in a pandas `Series` or `DataFrame`.

Data Frames

A `DataFrame` is a collection of `Series` that share the same index, and is therefore a two-dimensional generalization of a NumPy array. The row labels are collectively called the *index*, and the column labels are collectively called the *columns*. An individual column in a `DataFrame` object is one `Series`.

There are many ways to initialize a `DataFrame`. In the following code, we build a `DataFrame` out of a dictionary of `Series`.

```
>>> x = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), ['a', 'b', 'd', 'e', 'f'])
>>> df1 = pd.DataFrame({"series 1": x, "series 2": y})
>>> df1
   series 1  series 2
a -0.365542  1.227960
b  0.080133  0.683523
c  0.737970       NaN
d  0.097878 -1.102835
e       NaN  1.345004
f       NaN  0.217523
```

Note that the index of this `DataFrame` is the union of the index of `Series` `x` and that of `Series` `y`. The columns are given by the keys of the dictionary `d`. Since `x` doesn't have a label `e`, the value in row `e`, column `1` is `NaN`. This same reasoning explains the other missing values as well. Note that if we take the first column of the `DataFrame` and drop the missing values, we recover the `Series` `x`:

```
>>> df1["series1"].dropna()
a   -0.365542
b    0.080133
c    0.737970
d    0.097878
Name: series 1, dtype: float64
```

Achtung!

A pandas `DataFrame` cannot be sliced in exactly the same way as a NumPy array. Notice how we just used `df1["series 1"]` to access a *column* of the the `DataFrame df1`. We will discuss this in more detail later on.

We can also initialize a `DataFrame` using a NumPy array, creating custom row and column labels.

```
>>> data = np.random.random((3, 4))
>>> pd.DataFrame(data, index=['A', 'B', 'C'], columns=np.arange(1, 5))

          1         2         3         4
A  0.065646  0.968593  0.593394  0.750110
B  0.803829  0.662237  0.200592  0.137713
C  0.288801  0.956662  0.817915  0.951016
3 rows     4 columns
```

As with Series, if we don't specify the index or columns, the default is `np.arange(n)`, where `n` is either the number of rows or columns.

Viewing and Accessing Data

In this section we will explore some elementary accessing and querying techniques that enable us to maneuver through and gain insight into our data. Try using the `describe()` and `head()` methods for quick data summaries.

Basic Data Access

We can slice the rows of a `DataFrame` much as with a NumPy array.

```
>>> df = pd.DataFrame(np.random.randn(4, 2), index=['a', 'b', 'c', 'd'],
                      columns = ['I', 'II'])
>>> df[:2]

          I        II
a  0.758867  1.231330
b  0.402484 -0.955039

[2 rows x 2 columns]
```

More generally, we can select subsets of the data using the `iloc` and `loc` indexers. The `loc` index selects rows and columns based on their *labels*, while the `iloc` method selects them based on their integer *position*. Accessing `Series` and `DataFrame` objects using these indexing operations is more efficient than using bracket indexing, because the bracket indexing has to check many cases before it can determine how to slice the data structure. Using `loc`/`iloc` explicitly, bypasses the extra checks.

```
>>> # select rows a and c, column II
>>> df.loc[['a','c'], 'II']

a    1.231330
c    0.556121
Name: II, dtype: float64

>>> # select last two rows, first column
>>> df.iloc[-2:, 0]

c   -0.475952
d   -0.518989
Name: I, dtype: float64
```

Finally, a column of a `DataFrame` may be accessed using simple square brackets and the name of the column, or alternatively by treating the label as an object:

```
>>> # get second column of df
>>> df['II']        # or, equivalently, df.II

a    1.231330
b   -0.955039
c    0.556121
d    0.173165
Name: II, dtype: float64
```

All of these techniques for getting subsets of the data may also be used to set subsets of the data:

```
>>> # set second columns to zeros
>>> df['II'] = 0
>>> df['II']
a    0
b    0
c    0
d    0
Name: II, dtype: int64

>>> # add additional column of ones
>>> df['III'] = 1
>>> df
            I  II  III
a   -0.460457   0    1
b    0.973422   0    1
c   -0.475952   0    1
d   -0.518989   0    1
```

SQL Operations in pandas

The `DataFrame`, being a tabular data structure, bears an obvious resemblance to a typical relational database table. SQL is the standard for working with relational databases, and in this section we will explore how pandas accomplishes some of the same tasks as SQL. The SQL-like functionality of pandas is one of its biggest advantages, since it can eliminate the need to switch between programming languages for different tasks. Within pandas we can handle both the querying *and* data analysis.

For the following examples, we will use the following data:

```
>>> #build toy data for SQL operations
>>> name = ['Mylan', 'Regan', 'Justin', 'Jess', 'Jason', 'Remi', 'Matt', '↩
    Alexander', 'JeanMarie']
>>> sex = ['M', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'F']
>>> age = [20, 21, 18, 22, 19, 20, 20, 19, 20]
>>> rank = ['Sp', 'Se', 'Fr', 'Se', 'Sp', 'J', 'J', 'J', 'Se']
>>> ID = range(9)
>>> aid = ['y', 'n', 'n', 'y', 'n', 'n', 'n', 'y', 'n']
>>> GPA = [3.8, 3.5, 3.0, 3.9, 2.8, 2.9, 3.8, 3.4, 3.7]
>>> mathID = [0, 1, 5, 6, 3]
>>> mathGd = [4.0, 3.0, 3.5, 3.0, 4.0]
>>> major = ['y', 'n', 'y', 'n', 'n']
>>> studentInfo = pd.DataFrame({'ID': ID, 'Name': name, 'Sex': sex, 'Age': age,↩
     'Class': rank})
>>> otherInfo = pd.DataFrame({'ID': ID, 'GPA': GPA, 'Financial_Aid': aid})
>>> mathInfo = pd.DataFrame({'ID': mathID, 'Grade': mathGd, 'Math_Major': major↩
    })
```

Before querying our data, it is important to know some of its basic properties, such as number of columns, number of rows, and the datatypes of the columns. This can be done by simply calling the `info()` method on the desired `DataFrame`:

```
>>> mathInfo.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns (total 3 columns):
Grade         5 non-null float64
ID            5 non-null int64
Math_Major    5 non-null object
dtypes: float64(1), int64(1), object(1)
```

We can also get some basic information about the structure of the `DataFrame` using the `head()` or `tail()` methods.

```
>>> mathInfo.head()
   Grade  ID Math_Major  ID  Age  GPA
0    4.0   0          y   0   20  3.8
1    3.0   1          n   2   18  3.0
2    3.5   5          y   4   19  2.8
3    3.0   6          n   6   20  3.8
```

```
4    4.0   3          n   7   19  3.4
```

The method `isin()` is a useful way to find certain values in a `DataFrame`. It compares the input (a list, dictionary, or `Series`) to the `DataFrame` and returns a boolean `Series` showing whether or not the values match. You can then use this boolean array to select appropriate locations. Now let's look at the pandas equivalent of some SQL SELECT statements.

```
>>> # SELECT ID, Age FROM studentInfo
>>> studentInfo[['ID', 'Age']]

>>> # SELECT ID, GPA FROM otherInfo WHERE Financial_Aid = 'y'
>>> otherInfo[otherInfo['Financial_Aid']=='y'][['ID', 'GPA']]

>>> # SELECT Name FROM studentInfo WHERE Class = 'J' OR Class = 'Sp'
>>>  studentInfo[studentInfo['Class'].isin(['J','Sp'])]['Name']
```

Problem 3. The example above shows how to implement a simple WHERE condition, and it is easy to have a more complex expression. Simply enclose each condition by parentheses, and use the standard boolean operators `&` (AND), `|` (OR), and `~` (NOT) to connect the conditions appropriately. Use pandas to execute the following query:

```
SELECT ID, Name from studentInfo WHERE Age > 19 AND Sex = 'M'
```

Next, let's look at JOIN statements. In pandas, this is done with the `merge` function, which takes as arguments the two `DataFrame` objects to join, as well as keyword arguments specifying the column on which to join, along with the type (left, right, inner, outer).

```
>>> # SELECT * FROM studentInfo INNER JOIN mathInfo ON studentInfo.ID = ↩
    mathInfo.ID
>>> pd.merge(studentInfo, mathInfo, on='ID') # INNER JOIN is the default
   Age Class  ID    Name Sex  Grade Math_Major
0   20    Sp   0   Mylan   M    4.0          y
1   21    Se   1   Regan   F    3.0          n
2   22    Se   3    Jess   F    4.0          n
3   20     J   5    Remi   F    3.5          y
4   20     J   6    Matt   M    3.0          n
[5 rows x 7 columns]

>>> # SELECT GPA, Grade FROM otherInfo FULL OUTER JOIN mathInfo ON otherInfo.ID↩
     = mathInfo.ID
>>> pd.merge(otherInfo, mathInfo, on='ID', how='outer')[['GPA', 'Grade']]
   GPA  Grade
0  3.8    4.0
1  3.5    3.0
2  3.0    NaN
3  3.9    4.0
```

```
4  2.8    NaN
5  2.9    3.5
6  3.8    3.0
7  3.4    NaN
8  3.7    NaN
[9 rows x 2 columns]
```

Problem 4. Using a join operation, create a `DataFrame` containing the ID, age, and GPA of all male individuals. You ought to be able to accomplish this in one line of code.

Be aware that other types of SQL-like operations are also possible, such as UNION. When you find yourself unsure of how to carry out a more involved SQL-like operation, the online pandas documentation will be of great service.

Analyzing Data

Although pandas does not provide built-in support for heavy-duty statistical analysis of data, there are nevertheless many features and functions that facilitate basic data manipulation and computation, even when the data is in a somewhat messy state. We will now explore some of these features.

Basic Data Manipulation

Because the primary pandas data structures are subclasses of the `ndarray`, they are valid input to most NumPy functions, and can often be treated simply as NumPy arrays. For example, basic vectorized operations work just fine:

```
>>> x = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), index=['a', 'b', 'd', 'e', 'f'])
>>> x**2
a    1.710289
b    0.157482
c    0.540136
d    0.202580
dtype: float64
>>> z = x + y
>>> z
a    0.123877
b    0.278435
c         NaN
d   -1.318713
e         NaN
f         NaN
dtype: float64
>>> np.log(z)
a   -2.088469
b   -1.278570
```

```
c         NaN
d         NaN
e         NaN
f         NaN
dtype: float64
```

Notice that pandas automatically aligns the indexes when adding two `Series` (or `DataFrames`), so that the the index of the output is simply the union of the indexes of the two inputs. The default missing value `NaN` is given for labels that are not shared by both inputs.

It may also be useful to transpose `DataFrames`, re-order the columns or rows, or sort according to a given column. Here we demonstrate these capabilities:

```
>>> df = pd.DataFrame(np.random.randn(4,2), index=['a', 'b', 'c', 'd'], columns↩
    =['I', 'II'])
>>> df
          I        II
a -0.154878 -1.097156
b -0.948226  0.585780
c  0.433197 -0.493048
d -0.168612  0.999194

[4 rows x 2 columns]

>>> df.transpose()
           a         b         c         d
I  -0.154878 -0.948226  0.433197 -0.168612
II -1.097156  0.585780 -0.493048  0.999194

[2 rows x 4 columns]

>>> # switch order of columns, keep only rows 'a' and 'c'
>>> df.reindex(index=['a', 'c'], columns=['II', 'I'])
         II         I
a -1.097156 -0.154878
c -0.493048  0.433197

[2 rows x 2 columns]

>>> # sort descending according to column 'II'
>>> df.sort_values('II', ascending=False)
          I        II
d -0.168612  0.999194
b -0.948226  0.585780
c  0.433197 -0.493048
a -0.154878 -1.097156

[4 rows x 2 columns]
```

Dealing with Missing Data

Missing data is a ubiquitous problem in data science. Fortunately, pandas is particularly well-suited to handling missing and anomalous data. As we have already seen, the pandas default for a missing value is `NaN`. In basic arithmetic operations, if one of the operands is `NaN`, then the output is also `NaN`. The following example illustrates this concept:

```
>>> x = pd.Series(np.arange(5))
>>> y = pd.Series(np.random.randn(5))
>>> x.iloc[3] = np.nan
>>> x + y
0    0.731521
1    0.623651
2    2.396344
3         NaN
4    3.351182
dtype: float64
```

If we are not interested in the missing values, we can simply drop them from the data altogether:

```
>>> (x + y).dropna()
0    0.731521
1    0.623651
2    2.396344
4    3.351182
dtype: float64
```

This is not always the desired behavior, however. It may well be the case that missing data actually corresponds to some default value, such as zero. In this case, we can replace all instances of `NaN` with a specified value:

```
>>> # fill missing data with 0, add
>>> x.fillna(0) + y
0    0.731521
1    0.623651
2    2.396344
3    1.829400
4    3.351182
dtype: float64
```

Other functions, such as `sum()` and `mean()` ignore NaN values in the computation. When dealing with missing data, make sure you are aware of the behavior of the pandas functions you are using.

Data I/O

Being able to import and export data is a fundamental skill in data science. Unfortunately, with the multitude of data formats and conventions, importing data can often be a painful task. The pandas

library seeks to reduce some of the difficulty by providing file readers for various types of formats, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

Method	Description
`describe()`	Return a `Series` describing the data structure
`head()`	Return the first n rows, defaulting to 5
`tail()`	Return the last n rows, defaulting to 5
`to_csv()`	Write the index and entries to a CSV file
`to_json()`	Convert the object to a JSON string
`to_pickle()`	Serialize the object and store it in an external file
`to_sql()`	Write the object data to an open SQL database

Table 7.3: Methods for viewing or exporting data in a pandas `Series` or `DataFrame`.

The CSV (comma separated values) format is a simple way of storing tabular data in plain text. Because CSV files are one of the most popular file formats for exchanging data, we will explore the `read_csv()` function in more detail. To learn to read other types of file formats, see the online pandas documentation. To read a CSV data file into a `DataFrame`, call the `read_csv()` function with the path to the CSV file, along with the appropriate keyword arguments. Below we list some of the most important keyword arguments:

- `delimiter`: This argument specifies the character that separates data fields, often a comma or a whitespace character.
- `header`: The row number (starting at 0) in the CSV file that contains the column names.
- `index_col`: If you want to use one of the columns in the CSV file as the index for the `DataFrame`, set this argument to the desired column number.
- `skiprows`: If an integer n, skip the first n rows of the file, and then start reading in the data. If a list of integers, skip the specified rows.
- `names`: If the CSV file does not contain the column names, or you wish to use other column names, specify them in a list assigned to this argument.

There are several other keyword arguments, but this should be enough to get you started.

When you need to save your data, pandas allows you to write to several different file formats. A typical example is the `to_csv()` function method attached to `Series` and `DataFrame` objects, which writes the data to a CSV file. Keyword arguments allow you to specify the separator character, omit writing the columns names or index, and specify many other options. The code below demonstrates its typical usage:

```
>>> df.to_csv("my_df.csv")
```

Problem 5. The file `crime_data.csv` contains data on types of crimes committed in the United States from 1960 to 2016.

- Load the data into a pandas `DataFrame`, using the column names in the file and the column titled "Year" as the index. Make sure to skip lines that don't contain data.

- Insert a new column into the data frame that contains the crime rate by year (the ratio of "Total" column to the "Population" column).
- Plot the crime rate as a function of the year.
- List the 5 years with the highest crime rate in descending order.
- Calculate the average number of total crimes as well as burglary crimes between 1960 and 2012.
- Find the years for which the total number of crimes was below average, but the number of burglaries was above average.
- Plot the number of murders as a function of the population.
- Select the Population, Violent, and Robbery columns for all years in the 1980s, and save this smaller data frame to a CSV file `crime_subset.csv`.

Problem 6. In 1912 the RMS *Titanic* sank after colliding with an iceberg. The file `titanic.csv` contains data on the incident. Each row represents a different passenger, and the columns describe various features of the passengers (age, sex, whether or not they survived, etc.)

Start by cleaning the data.

- Read the data into a `DataFrame`. Use the first row of the file as the column labels, but do not use any of the columns as the index.
- Drop the columns `"Sibsp"`, `"Parch"`, `"Cabin"`, `"Boat"`, `"Body"`, and `"home.dest"`.
- Drop any entries without data in the `"Survived"` column, then change the remaining entries to `True` or `False` (they start as 1 or 0).
- Replace null entries in the `"Age"` column with the average age.
- Save the new `DataFrame` as `titanic_clean.csv`.

Next, answer the following questions.

- How many people survived? What percentage of passengers survived?
- What was the average price of a ticket? How much did the most expensive ticket cost?
- How old was the oldest survivor? How young was the youngest survivor? What about non-survivors?

8 Pandas 2: Plotting

Lab Objective: *Clear, insightful visualizations are a crucial part of data analysis. To facilitate quick data visualization, pandas includes several tools that wrap around matplotlib. These tools make it easy to compare different parts of a data set and explore the data as a whole.*

Overview of Plotting Tools

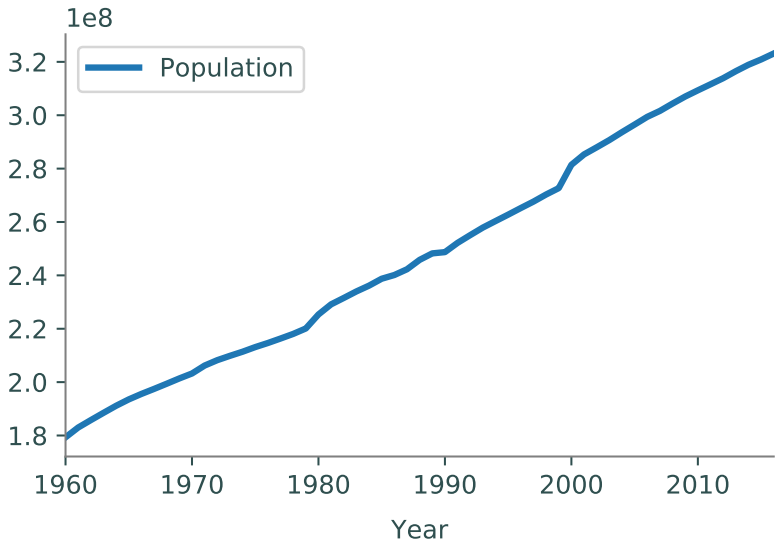
The main tool for visualization in pandas is the `plot()` method of the `Series` and `DataFrame`. The method has a keyword argument `kind` that specifies the type of plot to draw. The valid options for `kind` are detailed below.

Plot Type	`plot()` ID	Uses and Advantages
Line plot	`"line"`	Show trends ordered in data; easy to compare multiple data sets
Scatter plot	`"scatter"`	Compare exactly two data sets, independent of ordering
Bar plot	`"bar"`, `"barh"`	Compare categorical or sequential data
Histogram	`"hist"`	Show frequencies of one set of values, independent of ordering
Box plot	`"box"`	Display min, median, max, and quartiles; compare data distributions
Hexbin plot	`"hexbin"`	2D histogram; reveal density of cluttered scatter plots

Table 8.1: Uses for the `plot()` method of the pandas `Series` and `DataFrame`. The plot ID is the value of the keyword argument `kind`. That is, `df.plot(kind="scatter")` creates a scatter plot. The default `kind` is `"line"`.

The `plot()` method calls `plt.plot()`, `plt.hist()`, `plt.scatter()`, or another matplotlib plotting function, but it also assigns axis labels, tick marks, legends, and a few other things based on the index and the data. Most calls to `plot()` specify the `kind` of plot and which `Series` to use as the `x` and `y` axes. By default, the `index` of the `Series` or `DataFrame` is used for the `x` axis.

```
>>> import pandas as pd
>>> from matplotlib import pyplot as plt

>>> crime = pd.read_csv("crime_data.csv", index_col="Year")
>>> crime.plot(y="Population")  # Plot population against the index (years).
```



In this case, the call to the `plot()` method is essentially equivalent to the following code.

```
>>> plt.plot(crime.index, crime["Population"], label="Population")
>>> plt.xlabel(crime.index.name)
>>> plt.xlim(min(crime.index), max(crime.index))
>>> plt.legend(loc="best")
```

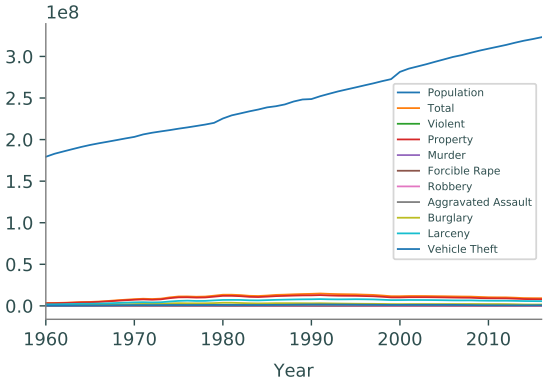
The `plot()` method also takes in many keyword arguments for matplotlib plotting and annotation functions. For example, setting `legend=False` disables the legend, providing a value for `title` sets the figure title, `grid=True` turns a grid on, and so on. For more customizations, see `https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html`.

Visualizing an Entire Data Set

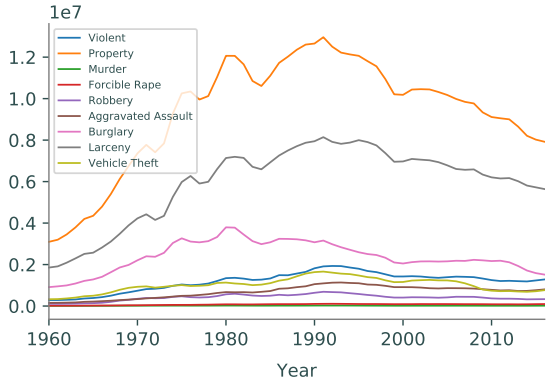
A good way to start analyzing an unfamiliar data set is to visualize as much of the data as possible to determine which parts are most important or interesting. For example, since the columns in a `DataFrame` share the same index, the columns can all be graphed together using the index as the x-axis. In fact, the `plot()` method attempts by default to plot **every** `Series` (column) in the `DataFrame`. This is especially useful with sequential data, like the crime data set.

The crime data set has 11 columns, so the resulting figure, Figure 8.1a, is fairly cluttered. However, it does shows that the `"Population"` column is on a completely different scale than the others. Dropping a few columns gives a better overview of the data, shown in Figure 8.1b.

```
# Plot all columns together against the index.
>>> crime.plot(linewidth=1)

# Plot all columns together except for 'Population' and 'Total'.
>>> crime.drop(["Population", "Total"], axis=1).plot(linewidth=1)
```



(a) All columns of the crime data set on the same figure, using the index as the x-axis.



(b) All columns of the crime data set except `"Population"` and `"Total"`.

Figure 8.1

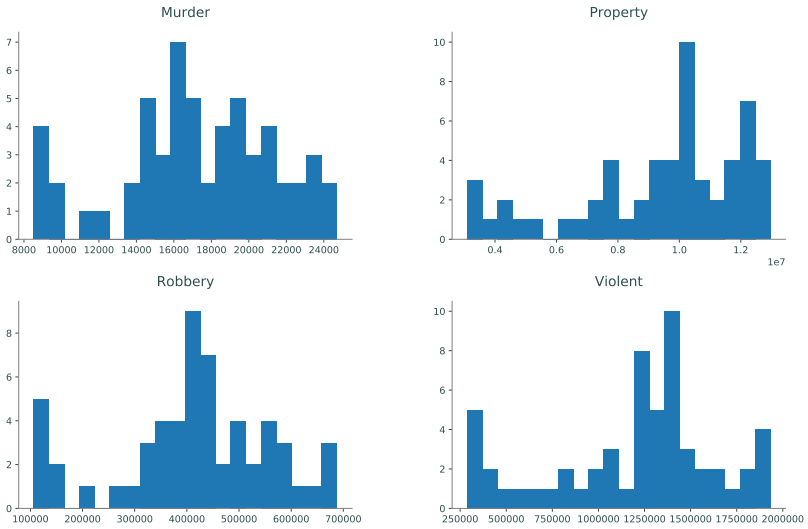
> **Achtung!**
>
> The `"Population"` column differs from the other columns because it is has **different units of measure**: population is measured by "number of people," but all other columns are measured in "number of crimes." Be careful not to plot parts of a data set together if those parts don't have the same units or are otherwise incomparable.

To quickly plot several columns in separate subplots, use `subplots=True` and specify a shape tuple as the `layout` for the plots. Subplots automatically share the same x-axis; set `sharey=True` to force them to share the same y-axis as well.

```
>>> crime.plot(y=["Property", "Larceny", "Burglary", "Violent"],
...            subplots=True, layout=(2,2), sharey=True,
...            style=['-', '--', '-.', ':'])
```

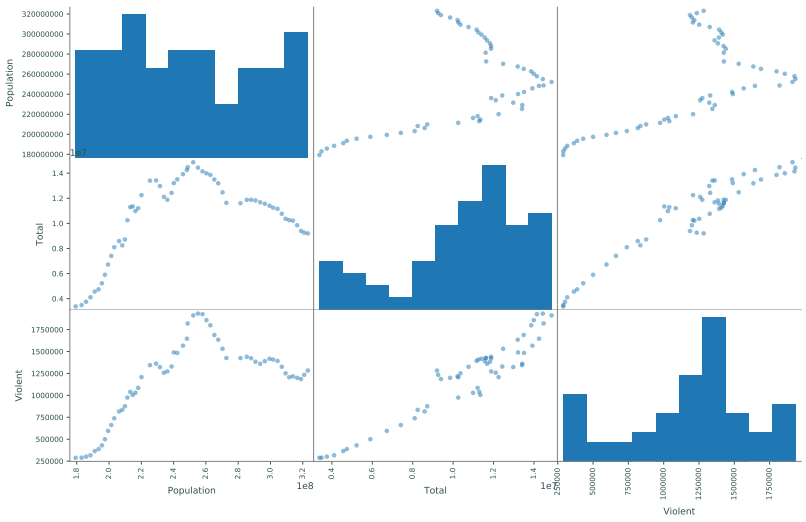
The `plot()` method can generate subplots of any kind of plot. However, since subplots share an x-axis by default, histograms turn out poorly whenever there are columns with very different data ranges. For histograms, use the `hist()` method of the `DataFrame` instead of the `plot()` method. Specify the number of bins with the `bins` parameter.

```
>>> crime[["Violent","Murder","Robbery","Property"]].hist(grid=False, bins=20)
```



Finally, the function `pd.plotting.scatter_matrix()` produces a table of plots where each column is plotted against each other column in separate scatter plots. The plots on the diagonal, instead of plotting a column against itself, displays a histogram of that column. This provides a way to very quickly do an initial analysis of the correlation between different columns.

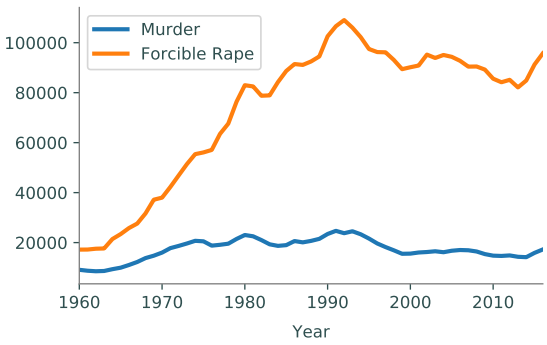
```
>>> pd.plotting.scatter_matrix(crime[["Population", "Total", "Violent"]])
```

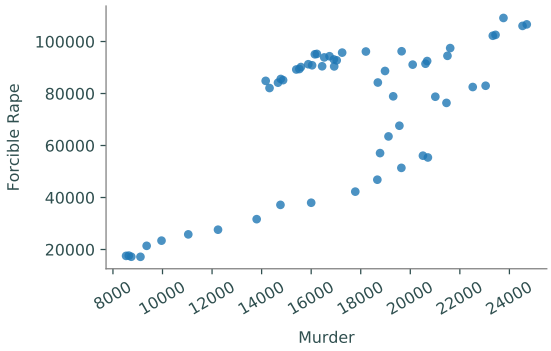


Patterns and Correlations

After visualizing the entire data set initially, the next step is usually to closely compare related parts of the data. For example, Figure 8.1b suggests that the `"Murder"` and `"Forcible Rape"` columns are roughly on the same scale. Since this data is sequential (indexed by time), start by plotting these two columns against the index. Next, create a scatter plot of one of the columns versus the other to investigate correlations that are independent of the index. Unlike other types of plots, using `kind="scatter"` requires both an `x` and a `y` column as arguments.

```
# Plot 'Murder' and 'Forcible Rape' as lines against the index.
>>> crime.plot(y=["Murder", "Forcible Rape"])

# Make a scatter plot of 'Murder' against 'Forcible Rape', ignoring the index.
>>> crime.plot(kind="scatter", x="Murder", y="Forcible Rape", alpha=.8, rot=30)
```





What does these graphs show about the data? First of all, rape is more common than murder. Second, rates of rape appear to be steadily increased from the mid 1960's to the mid 1990's before leveling out, while murder rates stay relatively constant. The disparity between rape and murder is confirmed in the scatter plot: the clump of data points at about 15,000 murders and 90,000 rapes shows that there have been many years where rape was relatively high while murder was somewhat low.

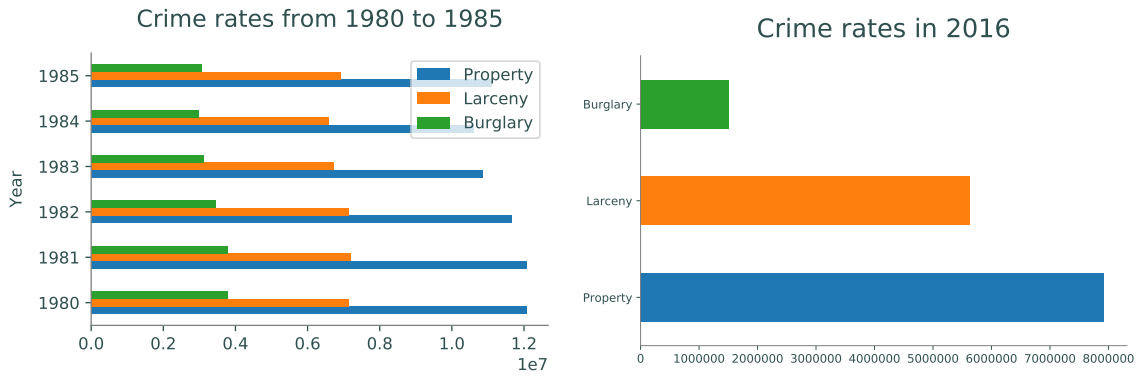
Achtung!

While analyzing data, especially while searching for patterns and correlations, **always** ask yourself if the data makes sense and is trustworthy. What lurking variables could have influenced the data measurements as they were being gathered?

The crime data set is somewhat suspect in this regard. The murder rate is likely accurate, since murder is conspicuous and highly reported, but what about the rape rate? Are the number of rapes increasing, or is the percentage of rapes being reported increasing? (It's probably both!) Be careful about drawing conclusions for sensitive or questionable data.

Figure 8.1b also reveals some general patterns relative to time. For instance, there seems to be a small bump in each type of crime in the early 1980's. Slicing the entries from 1980 to 1985 provides a closer look. Since there are only a few entries, we can treat the data as categorical and make a bar chart.

```
# Plot 'Property' and 'Larceny' rates from 1980 to 1985.
>>> crime.loc[1980:1985, ["Property", "Larceny", "Burglary"]].plot(kind="barh",
...                       title="Crime rates from 1980 to 1985")

# Plot the most recent year's crime rates for comparison.
>>> crime.iloc[-1][["Property", "Larceny", "Burglary"]].plot(kind="barh",
...                       title="Crime rates in 2016", color=["C0", "C1", "C2"])
>>> plt.tight_layout()
```

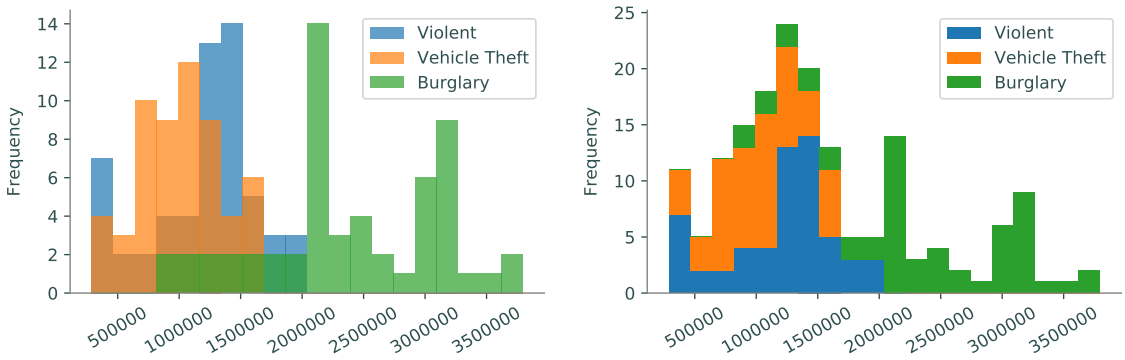


> **Note**
>
> As a general rule, horizontal bar charts (`kind="hbar"`) are better than the default vertical bar charts (`kind="bar"`) because most humans can detect horizontal differences more easily than vertical differences. If the labels are too long to fit on a normal figure, use `plt.tight_layout()` to adjust the plot boundaries to fit the labels in.

Distributional Visualizations

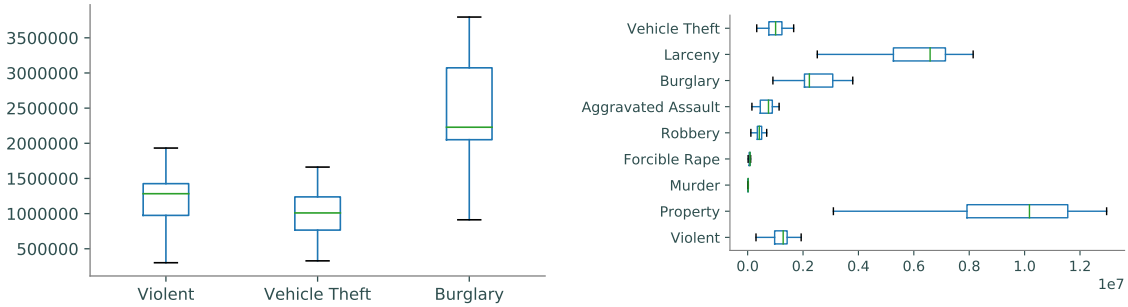
Histograms are good for examining the distribution of a **single** column in a data set. While pandas is capable of plotting several histograms on the same plot, the results are usually hard to read.

```
# Plot three histograms together.
>>> crime.plot(kind="hist", y=["Violent", "Vehicle Theft", "Burglary"],
...            bins=20, alpha=.7, rot=30)

# Plot three histograms, stacking one on top of the other.
>>> crime.plot(kind="hist", y=["Violent", "Vehicle Theft", "Burglary"],
...            bins=20, stacked=True, rot=30)
```



Instead of using histograms to compare distributions of data, use box plots. A *box plot* (sometimes called a "cat-and-whisker" plot) shows the five number summary: the minimum, first quartile, median, third quartile, and maximum of the data. While not quite the same as a histogram, box plots are much better suited to quickly compare relatable distributions.

```
# Compare the distributions of three columns.
>>> crime.plot(kind="box", y=["Violent", "Vehicle Theft", "Burglary"])

# Compare the distributions of all columns but 'Population' and 'Total'.
>>> crime.drop(["Population", "Total"], axis=1).plot(kind="box", vert=False)
```

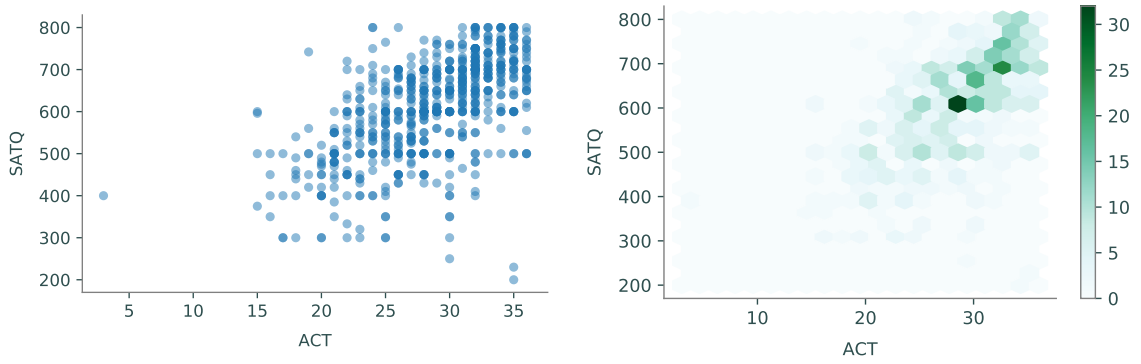


Hexbin Plots

A scatter plot is essentially a plot of samples from the joint distribution of two columns. However, scatter plots can be uninformative for large data sets when the points in a scatter plot are closely clustered. *Hexbin plots* solve this problem by plotting point density in hexagonal bins—essentially creating a 2-dimensional histogram.

The file `sat_act.csv` contains 700 self reported scores on the SAT Verbal, SAT Quantitative and ACT, collected as part of the Synthetic Aperture Personality Assessment (SAPA) web based personality assessment project. The obvious question with this data set is "how correlated are ACT and SAT scores?" The scatter plot of ACT scores versus SAT Quantitative scores, Figure 8.6a, is highly cluttered, even though the points have some transparency. A hexbin plot of the same data, Figure 8.6b, reveals the **frequency** of points in binned regions.

```
>>> satact = pd.read_csv("sat_act.csv", index_col="ID")
>>> list(satact.columns)
['gender', 'education', 'age', 'ACT', 'SATV', 'SATQ']

# Plot the ACT scores against the SAT Quant scores in a regular scatter plot.
>>> satact.plot(kind="scatter", x="ACT", y="SATQ", alpha=.8)

# Plot the densities of the ACT vs. SATQ scores with a hexbin plot.
>>> satact.plot(kind="Hexbin", x="ACT", y="SATQ", gridsize=20)
```



(a) ACT vs. SAT Quant scores.

(b) Frequency of ACT vs. SAT Quant scores.

Figure 8.6

Just as choosing a good number of `bins` is important for a good histogram, choosing a good `gridsize` is crucial for an informative hexbin plot. A large `gridsize` creates many small bins and a small `gridsize` creates fewer, larger bins.

See http://pandas.pydata.org/pandas-docs/stable/visualization.html for more types of plots available in Pandas and further examples.

Principles of Good Data Visualization

Visualization is much more than a set of pretty pictures scattered throughout a paper for the sole purpose of providing contrast to the text. When properly implemented, data visualization is a powerful tool for analysis and communication. When writing a paper or report, the author must make many decisions about how to use graphics effectively to convey useful information to the reader. Here we will go over a simple process for making deliberate, effective, and efficient design decisions.

Attention to Detail

Consider the plot in Figure 8.7. What does it depict? We can tell from a simple glance that it is a scatter plot of positively correlated data of some kind, with `temp`–likely temperature–on the x axis and `cons` on the y axis. However, the picture is not really communicating anything about the dataset. We have not specified the units for the x or the y axis, we have no idea what `cons` is, there is no title, and we don't even know where the data came from in the first place.

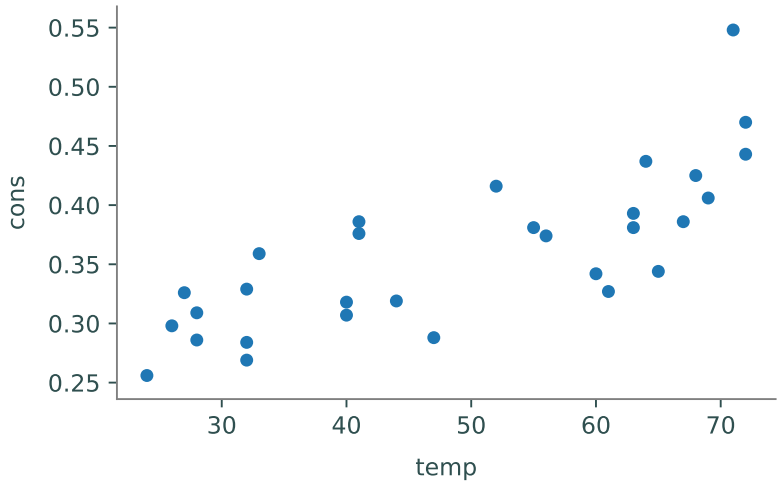


Figure 8.7: Non-specific data.

Labels and Citations

In a homework or lab setting, we sometimes (mistakenly) think that it is acceptable to leave off appropriate labels, legends, titles, and sourcing. In a published report or presentation, this kind of carelessness is confusing at best and, when the source is not included, even plagiaristic. Clearly, we need to explain our data in a useful manner that includes all of the vital information.

Consider again Figure 8.7. This figure comes from the `Icecream` dataset within the `pydataset` package, which we store here in a dataframe and then plot:

```
>>> from pydataset import data
>>> icecream = data("Icecream")
>>> icecream.plot(kind="scatter", x="temp", y="cons")
```

We have at this point reproduced the rather substandard plot in Figure 8.7. Using `data('Icecream', show_doc=True)` we find the following information:

1. The dataset details ice cream consumption via four-weekly observations from March 1951 to July 1953 in the United States.

2. `cons` corresponds to "consumption of ice cream per head" and is measured in pints.

3. `temp` corresponds to temperature, degrees Fahrenheit.

4. The listed source is: "Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University."

We add these important details using the following code. As we have seen in previous examples, pandas automatically generates legends when appropriate. However, although pandas also automatically labels the x and y axes, our data frame column titles may be insufficient. Appropriate titles for the x and y axes must also list appropriate units. For example, the y axis should specify that the consumption is in units of *pints per head*, in place of the ambiguous label `cons`.

```
>>> icecream = data("Icecream")
# Set title via the title keyword argument
>>> icecream.plot(kind="scatter", x="temp", y="cons", title="Ice Cream ↩
    Consumption in the U.S., 1951-1953",)
# Override pandas automatic labelling using xlabel and ylabel
>>> plt.xlabel("Temp (Farenheit)")
>>> plt.ylabel("Consumption per head (pints)")
```

To arbitrarily add the necessary text to the figure, use either `plt.annotate()` or `plt.text()`. Alternatively, add text immediately below wherever the figure is displayed.

```
>>> plt.text(20, .1, r"Source: Hildreth, C. and J. Lu (1960) \emph{Demand"
...     "relations with autocorrelated disturbances}\nTechnical Bulletin No"
...     "2765, Michigan State University.", fontsize=7)
```

Both of these methods are imperfect, however, and can normally be just as easily replaced by a caption attached to the figure in your presentation or document setting. We again reiterate how important it is that you source any data you use. Failing to do so is plagiarism.

Finally, we have a clear and demonstrative graphic in Figure 8.8.

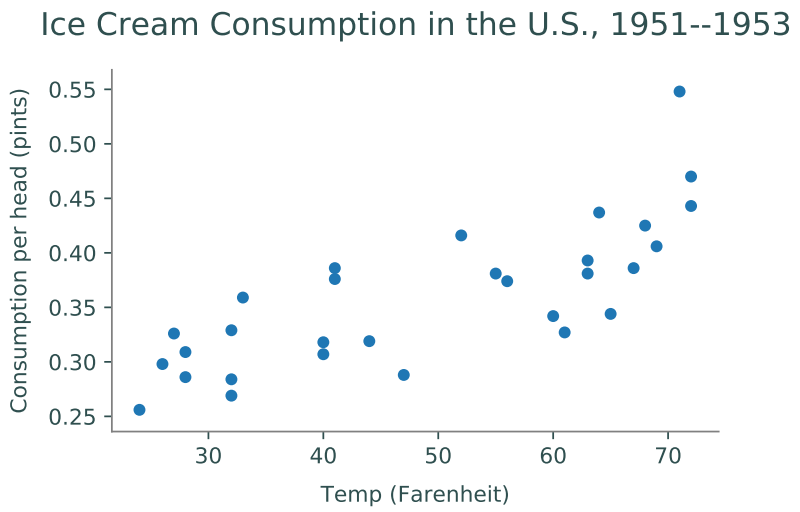


Figure 8.8: Source: Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.

Problem 1. The `pydataset` module[a] contains numerous data sets, each stored as a pandas `DataFrame`.

```
>>> from pydataset import data

# Call data() to see the entire list of data sets.
```

```
# To load a particular data set, enter its ID as an argument to data().
>>> titanic = data("Titanic")
# To see the information about a data set, call data() with show_doc=True.
>>> data("Titanic", show_doc=True)
Titanic

PyDataset Documentation (adopted from R Documentation. The displayed ↩
    examples
are in R)

## Survival of passengers on the Titanic
```

Visualize and describe at least 5 of the following data sets with 2 or 3 figures each. Comment on the implications and significance of each visualization and give a comprehensive summary of the data set.

- `"Arbuthnot"`: Ratios of male to female births in London from 1629-1710
- `"trees"`: Girth, height and volume for black cherry trees
- `"road"`: Road accident deaths in the United States
- `"birthdeathrates"`: Birth and death rates by country
- `"bfeed"`: Child breast feeding records
- `"heart"`: Survival of patients on the waiting list for the Stanford heart transplant program
- `"lung"`: Survival in patients with advanced lung cancer from the North Central Cancer Treatment group
- `"birthwt"`: Risk factors associated with low infant birth weight
- A data set of your choice

Include each of the following in each visualization.

- A clear title, with relevant information for the period or region the data was collected in.
- Axis labels that specify units.
- A legend (if appropriate).
- The source. You may include the source information in your plot or print it after the plot.

[a] Run `pip install pydataset` if needed.

9 Pandas III: Grouping

Lab Objective: *Many data sets contain categorical values that naturally sort the data into groups. Analyzing and comparing such groups is an important part of data analysis. In this lab we explore pandas tools for grouping data and presenting tabular data more compactly, primarily through grouby and pivot tables.*

Groupby

The file `mammal_sleep.csv`[1] contains data on the sleep cycles of different mammals, classified by order, genus, species, and diet (carnivore, herbivore, omnivore, or insectivore). The `"sleep_total"` column gives the total number of hours that each animal sleeps (on average) every 24 hours. To get an idea of how many animals sleep for how long, we start off with a histogram of the `"sleep_total"` column.

```
>>> import pandas as pd
>>> from matplotlib import pyplot as plt

# Read in the data and print a few random entries.
>>> msleep = pd.read_csv("mammal_sleep.csv")
>>> msleep.sample(5)
      name     genus   vore         order  sleep_total  sleep_rem  sleep_cycle
51  Jaguar  Panthera  carni     Carnivora         10.4        NaN          NaN
77  Tenrec    Tenrec   omni  Afrosoricida         15.6        2.3          NaN
10    Goat     Capri  herbi  Artiodactyla          5.3        0.6          NaN
80   Genet   Genetta  carni     Carnivora          6.3        1.3          NaN
33   Human      Homo   omni      Primates          8.0        1.9          1.5

# Plot the distribution of the sleep_total variable.
>>> msleep.plot(kind="hist", y="sleep_total", title="Mammalian Sleep Data")
>>> plt.xlabel("Hours")
```

[1] Proceedings of the National Academy of Sciences, 104 (3):1051–1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia. Available in `pydataset` (with a few more columns) under the key `"msleep"`.

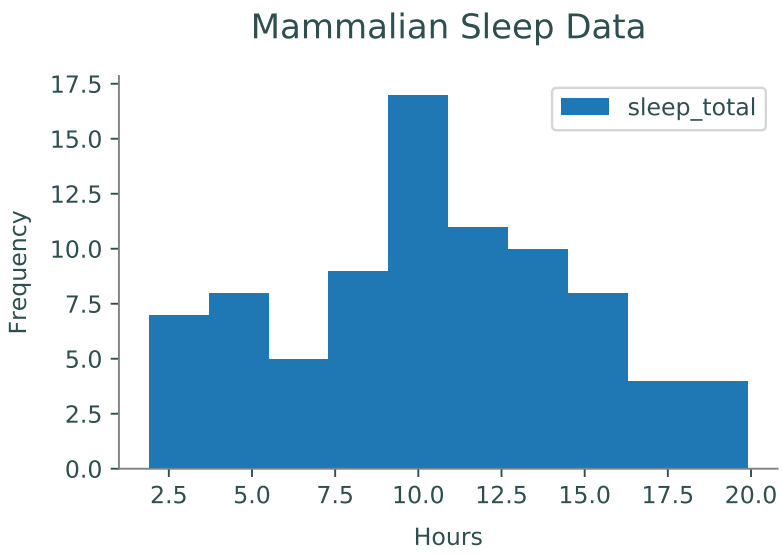


Figure 9.1: `"sleep_total"` frequencies from the mammalian sleep data set.

While this visualization is a good start, it doesn't provide any information about how different kinds of animals have different sleeping habits. How long do carnivores sleep compared to herbivores? Do mammals of the same genus have similar sleep patterns?

A powerful tool for answering these kinds of questions is the `groupby()` method of the pandas `DataFrame` class, which partitions the original `DataFrame` into groups based on the values in one or more columns. The `groupby()` method does **not** return a new `DataFrame`; it returns a pandas `GroupBy` object, an interface for analyzing the original `DataFrame` by groups.

For example, the columns `"genus"`, `"vore"`, and `"order"` in the mammal sleep data all have a discrete number of categorical values that could be used to group the data. Since the `"vore"` column has only a few unique values, we start by grouping the animals by diet.

```
# List all of the unique values in the 'vore' column.
>>> set(msleep["vore"])
{nan, 'herbi', 'omni', 'carni', 'insecti'}

# Group the data by the 'vore' column.
>>> vores = msleep.groupby("vore")
>>> list(vores.groups)
['carni', 'herbi', 'insecti', 'omni']       # NaN values for vore were dropped.

# Get a single group and sample a few rows. Note vore='carni' in each entry.
>>> vores.get_group("carni").sample(5)
       name     genus   vore      order  sleep_total  sleep_rem  sleep_cycle
80    Genet   Genetta  carni  Carnivora          6.3        1.3          NaN
50    Tiger  Panthera  carni  Carnivora         15.8        NaN          NaN
8       Dog     Canis  carni  Carnivora         10.1        2.9        0.333
0   Cheetah  Acinonyx  carni  Carnivora         12.1        NaN          NaN
82  Red fox    Vulpes  carni  Carnivora          9.8        2.4        0.350
```

For starters, `groupby()` is useful for filtering a `DataFrame` by column values: the command `df.groupby(col).get_group(value)` returns the rows of `df` where the entry of the `col` column is `value`. The real advantage of `groupby()`, however, is how easy it makes it to compare groups of data. Standard `DataFrame` methods like `describe()`, `mean()`, `std()`, `min()`, and `max()` all work on `GroupBy` objects to produce a new data frame that describes the statistics of each group.

```
# Get averages of the numerical columns for each group.
>>> vores.mean()
         sleep_total  sleep_rem  sleep_cycle
vore
carni         10.379      2.290        0.373
herbi          9.509      1.367        0.418
insecti       14.940      3.525        0.161
omni          10.925      1.956        0.592

# Get more detailed statistics for 'sleep_total' by group.
>>> vores["sleep_total"].describe()
         count    mean    std  min   25%   50%     75%   max
vore
carni     19.0  10.379  4.669  2.7  6.25  10.4  13.000  19.4
herbi     32.0   9.509  4.879  1.9  4.30  10.3  14.225  16.6
insecti    5.0  14.940  5.921  8.4  8.60  18.1  19.700  19.9
omni      20.0  10.925  2.949  8.0  9.10   9.9  10.925  18.0
```

Multiple columns can be used simultaneously for grouping. In this case, the `get_group()` method of the `GroupBy` object requires a tuple specifying the values for each of the grouping columns.

```
>>> msleep_small = msleep.drop(["sleep_rem", "sleep_cycle"], axis=1)
>>> vores_orders = msleep_small.groupby(["vore", "order"])
>>> vores_orders.get_group(("carni", "Cetacea"))
                    name          genus   vore    order  sleep_total
30           Pilot whale  Globicephalus  carni  Cetacea          2.7
59       Common porpoise       Phocoena  carni  Cetacea          5.6
79  Bottle-nosed dolphin       Tursiops  carni  Cetacea          5.2
```

Visualizing Groups

There are a few ways that `groupby()` or similar techniques can simplify the process of visualizing groups of data. First of all, `groupby()` makes it easy to visualize one group at a time. The following visualization improve on Figure 9.1 by grouping mammals by their diets.

```
# Plot histograms of 'sleep_total' for two separate groups.
>>> vores.get_group("carni").plot(kind="hist", y="sleep_total", legend="False",
                                  title="Carnivore Sleep Data")
>>> plt.xlabel("Hours")
>>> vores.get_group("herbi").plot(kind="hist", y="sleep_total", legend="False",
                                  title="Herbivore Sleep Data")
>>> plt.xlabel("Hours")
```

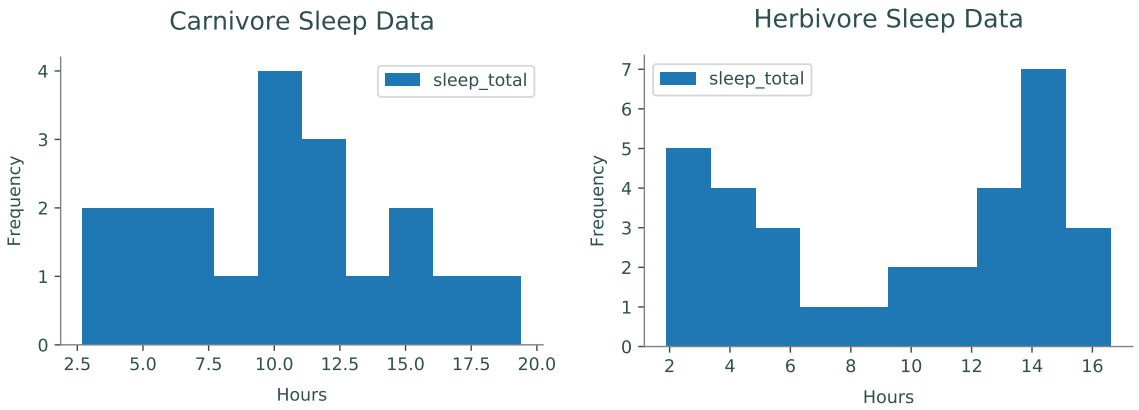
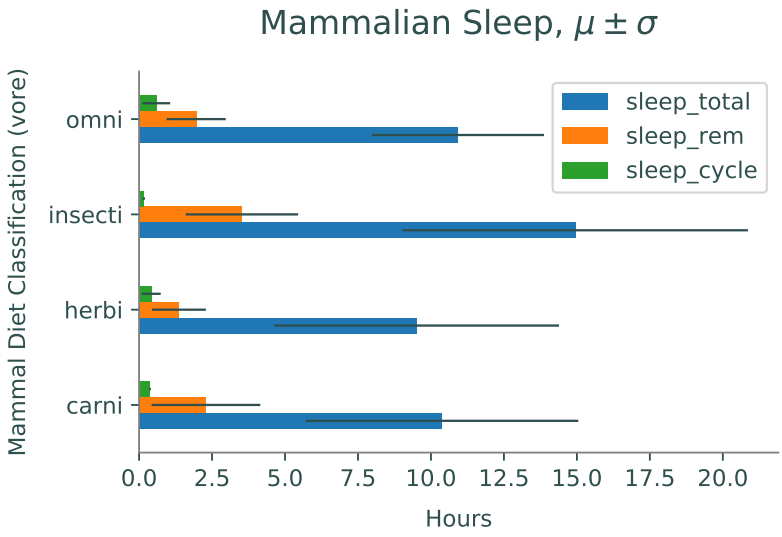


Figure 9.2: `"sleep_total"` histograms for two groups in the mammalian sleep data set.

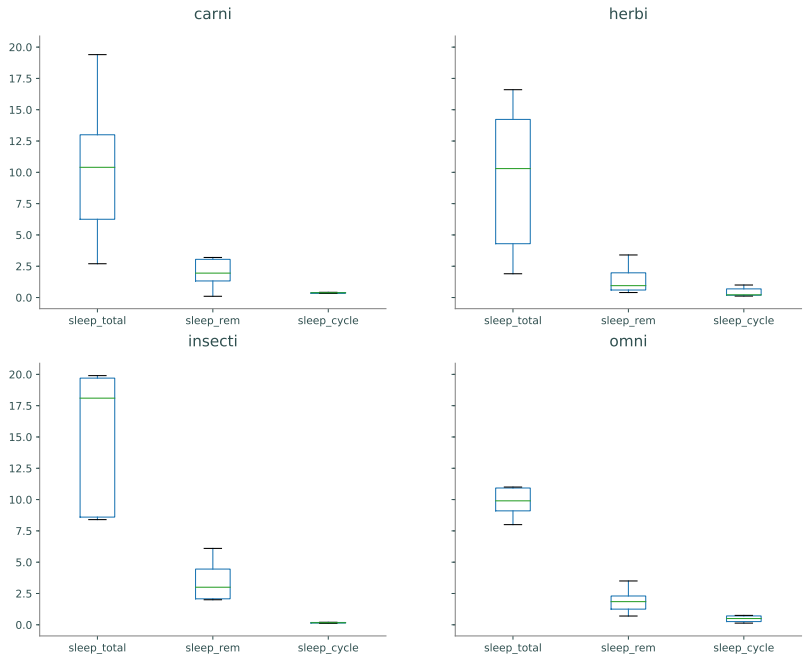
The statistical summaries from the `GroupBy` object's `mean()`, `std()`, or `describe()` methods also lend themselves well to certain visualizations for comparing groups.

```
>>> vores[["sleep_total", "sleep_rem", "sleep_cycle"]].mean().plot(kind="barh",
                xerr=vores.std(), title=r"Mammallian Sleep, $\mu\pm\sigma$")
>>> plt.xlabel("Hours")
>>> plt.ylabel("Mammal Diet Classification (vore)")
```



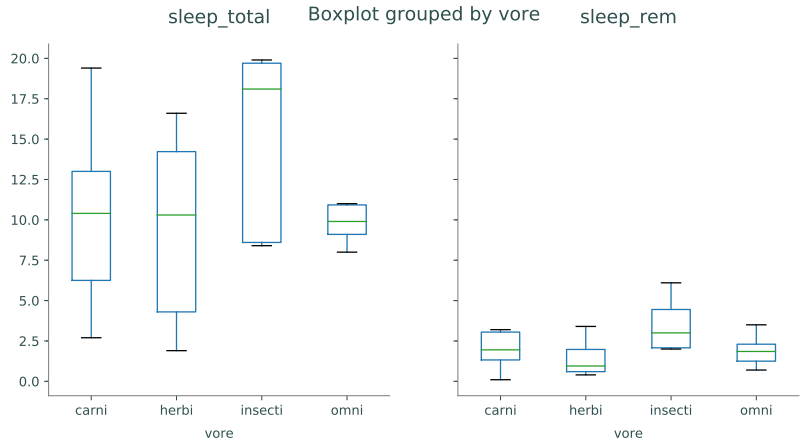
Box plots are well suited for comparing similar distributions. The `boxplot()` method of the `GroupBy` class creates one subplot **per group**, plotting each of the columns as a box plot.

```
# Use GroupBy.boxplot() to generate one box plot per group.
>>> vores.boxplot(grid=False)
```



Alternatively, the `boxplot()` method of the `DataFrame` class creates one subplot **per column**, plotting each of the columns as a box plot. Specify the `by` keyword to group the data appropriately.

```
# Use DataFrame.boxplot() to generate one box plot per column.
>>> msleep.boxplot(["sleep_total", "sleep_rem"], by="vore", grid=False)
```



Like `groupby()`, the `by` argument can be a single column label or a list of column labels. Similar methods exist for creating histograms (`GroupBy.hist()` and `DataFrame.hist()` with `by` keyword), but generally box plots are better for comparing multiple distributions.

Problem 1. Examine the following data sets from `pydataset` and answer the corresponding questions. Use visualizations to support your conclusions.

- `"iris"`, measurements of various species of iris flowers.
  1. Which species is easiest to distinguish from the others? How?
  2. Given iris data without a species label, what strategies could you use to identify the flower's species?
- `"poisons"`, experimental results of three different poisons and four different treatments.
  1. In general, which poison is most deadly? Which treatment is most effective?
  2. If you were poisoned, how would you choose the treatment if you did not know which poison it was? What if you did know which poison it was?
     (Hint: group the data by poison, then group each subset by treatment.)
- `"diamonds"`, prices and characteristics of almost 54,000 round-cut diamonds.
  1. How does the color and cut of a diamond affect its price?
  2. Of the diamonds with color `"H"`, those with a `"Fair"` cut sell, on average, for a **higher** price than those with an `"Ideal"` (superior) cut. What other factors could explain this unintuitive statistic?

Pivot Tables

One of the downfalls of `groupby()` is that a typical `GroupBy` object has too much information to display coherently. A *pivot table* intelligently summarizes the results of a `groupby()` operation by aggregating the data in a specified way. The standard tool for making a pivot table is the `pivot_table()` method of the `DataFrame` class. As an example, consider the `"HairEyeColor"` data set from `pydataset`.

```
>>> from pydataset import data
>>> hec = data("HairEyeColor")              # Load and preview the data.
>>> hec.sample(5)
     Hair    Eye     Sex  Freq
3     Red  Brown    Male    10
1   Black  Brown    Male    32
14  Brown  Green    Male    15
31    Red  Green  Female     7
21  Black   Blue  Female     9

>>> for col in ["Hair", "Eye", "Sex"]:      # Get unique values per column.
...     print("{}: {}".format(col, ", ".join(set(str(x) for x in hec[col]))))
...
Hair: Brown, Black, Blond, Red
Eye: Brown, Blue, Hazel, Green
Sex: Male, Female
```

There are several ways to group this data with `groupby()`. However, since there is only one entry per unique hair-eye-sex combination, the data can be completely presented in a pivot table.

```
>>> hec.pivot_table(values="Freq", index=["Hair", "Eye"], columns="Sex")
Sex          Female  Male
Hair  Eye
Black Blue        9    11
      Brown      36    32
      Green       2     3
      Hazel       5    10
Blond Blue       64    30
      Brown       4     3
      Green       8     8
      Hazel       5     5
Brown Blue       34    50
      Brown      66    53
      Green      14    15
      Hazel      29    25
Red   Blue        7    10
      Brown      16    10
      Green       7     7
      Hazel       7     7
```

Listing the data in this way makes it easy to locate data and compare the female and male groups. For example, it is easy to see that brown hair is more common than red hair and that about twice as many females have blond hair and blue eyes than males.

Unlike `"HairEyeColor"`, many data sets have more than one entry in the data for each grouping (for example, if there were two or more rows in the original data for females with blond hair and blue eyes). To construct a pivot table, data of similar groups must be *aggregated* together in some way. By default entries are aggregated by averaging the non-null values. Other options include taking the min, max, standard deviation, or just counting the number of occurrences.

As an example, consider again the Titanic data set found in `titanic.csv`[2]. For this analysis, take only the `"Survived"`, `"Pclass"`, `"Sex"`, `"Age"`, `"Fare"`, and `"Embarked"` columns, replace null age values with the average age, then drop any rows that are missing data. To begin, we examine the average survival rate grouped by sex and passenger class.

```
>>> titanic = pd.read_csv("titanic")
>>> titanic = titanic[["Survived", "Pclass", "Sex", "Age", "Fare", "Embarked"]]
>>> titanic["Age"].fillna(titanic["Age"].mean(), inplace=True)
>>> titanic.dropna(inplace=True)

>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass")
Pclass     1.0    2.0    3.0
Sex
female   0.965  0.887  0.491
male     0.341  0.146  0.152
```

[2]There is a `"Titanic"` data set in `pydataset`, but it does not contain as much information as the data in `titanic.csv`.

Note

The `pivot_table()` method is just a convenient way of performing a potentially complicated `groupby()` operation with aggregation and some reshaping. For example, the following code is equivalent to the previous example.

```
>>> titanic.groupby(["Sex", "Pclass"])["Survived"].mean().unstack()
Pclass    1.0    2.0    3.0
Sex
female  0.965  0.887  0.491
male    0.341  0.146  0.152
```

The `stack()`, `unstack()`, and `pivot()` methods provide more advanced shaping options.

Among other things, this pivot table clearly shows how much more likely females were to survive than males. To see how many entries fall into each category, or how many survived in each category, aggregate by counting or summing instead of taking the mean.

```
# See how many entries are in each category.
>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass",
...                     aggfunc="count")
Pclass  1.0  2.0  3.0
Sex
female  144  106  216
male    179  171  493

# See how many people from each category survived.
>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass",
...                     aggfunc="sum")
Pclass    1.0   2.0    3.0
Sex
female  137.0  94.0  106.0
male     61.0  25.0   75.0
```

Discretizing Continuous Data

So far we have examined survival rates based on sex and passenger class. Another factor that could have played into survival is age. Were male children as likely to die as females in general? We can investigate this question by *multi-indexing*, or pivoting on more than just two variables, by adding in another index.

In the original dataset, the `"Age"` column has a floating point value for the age of each passenger. If we just added `"Age"` as another pivot, then the table would create a new row for **each** age present. Instead, we partition the `"Age"` column into intervals with `pd.cut()`, thus creating a categorical that can be used for grouping.

```
# pd.cut() maps continuous entries to discrete intervals.
>>> pd.cut([6, 1, 2, 3, 4, 5, 6, 7], [0, 4, 8])
[(0, 4], (0, 4], (0, 4], (0, 4], (4, 8], (4, 8], (4, 8], (0, 4]]
Categories (2, interval[int64]): [(0, 4] < (4, 8]]

# Partition the passengers into 3 categories based on age.
>>> age = pd.cut(titanic['Age'], [0, 12, 18, 80])

>>> titanic.pivot_table(values="Survived", index=["Sex", age],
                        columns="Pclass", aggfunc="mean")
Pclass              1.0    2.0    3.0
Sex    Age
female (0, 12]   0.000  1.000  0.467
       (12, 18]  1.000  0.875  0.607
       (18, 80]  0.969  0.871  0.475
male   (0, 12]   1.000  1.000  0.343
       (12, 18]  0.500  0.000  0.081
       (18, 80]  0.322  0.093  0.143
```

From this table, it appears that male children (ages 0 to 12) in the 1st and 2nd class were very likely to survive, whereas those in 3rd class were much less likely to. This clarifies the claim that males were less likely to survive than females. However, there are a few oddities in this table: zero percent of the female children in 1st class survived, and zero percent of teenage males in second class survived. To further investigate, count the number of entries in each group.

```
>>> titanic.pivot_table(values="Survived", index=["Sex", age],
                        columns="Pclass", aggfunc="count")
Pclass           1.0  2.0  3.0
Sex    Age
female (0, 12]     1   13   30
       (12, 18]   12    8   28
       (18, 80]  129   85  158
male   (0, 12]     4   11   35
       (12, 18]    4   10   37
       (18, 80]  171  150  420
```

This table shows that there was only 1 female child in first class and only 10 male teenagers in second class, which sheds light on the previous table.

Achtung!

The previous pivot table brings up an important point about partitioning datasets. The Titanic dataset includes data for about 1300 passengers, which is a somewhat reasonable sample size, but half of the groupings include less than 30 entries, which is **not** a healthy sample size for statistical analysis. Always carefully question the numbers from pivot tables before making any conclusions.

Pandas also supports multi-indexing on the columns. As an example, consider the price of a passenger tickets. This is another continuous feature that can be discretized with `pd.cut()`. Instead, we use `pd.qcut()` to split the prices into 2 equal quantiles. Some of the resulting groups are empty; to improve readability, specify `fill_value` as the empty string or a dash.

```
# pd.qcut() partitions entries into equally populated intervals.
>>> pd.qcut([1, 2, 5, 6, 8, 3], 2)
[(0.999, 4.0], (0.999, 4.0], (4.0, 8.0], (4.0, 8.0], (4.0, 8.0], (0.999, 4.0]]
Categories (2, interval[float64]): [(0.999, 4.0] < (4.0, 8.0]]

# Cut the ticket price into two intervals (cheap vs expensive).
>>> fare = pd.qcut(titanic["Fare"], 2)
>>> titanic.pivot_table(values="Survived",
                        index=["Sex", age], columns=[fare, "Pclass"],
                        aggfunc="count", fill_value='-')
Fare            (-0.001, 14.454]          (14.454, 512.329]
Pclass                       1.0 2.0  3.0               1.0 2.0 3.0
Sex    Age
female (0, 12]                 -   -    7                 1  13  23
       (12, 18]                -   4   23                12   4   5
       (18, 80]                -  31  101               129  54  57
male   (0, 12]                 -   -    8                 4  11  27
       (12, 18]                -   5   26                 4   5  11
       (18, 80]                8  94  350               163  56  70
```

Not surprisingly, most of the cheap tickets went to passengers in 3rd class.

Problem 2. Suppose that someone claims that the city from which a passenger embarked had a strong influence on the passenger's survival rate. Investigate this claim.

1. Check the survival rates of the passengers based on where they embarked from (given in the `"Embarked"` column).
2. Create a pivot table to examine survival rates based on both place of embarkment and gender.
3. What do these tables suggest to you about the significance of where people embarked in influencing their survival rate? Examine the context of the problem, and explain what you think this really means.
4. Investigate the claim further with at least two more pivot tables, exploring other criteria (e.g., class, age, etc.). Carefully explain your conclusions.

Problem 3. Examine the following data sets from `pydataset` and answer the corresponding questions. Use visualizations and/or pivot tables as appropriate to support your conclusions.

- `"npk"`, an experiment on the effects of nitrogen (N), phosphate (P), and potassium (K) on the growth of peas.
  1. Which element is most effective in general for simulating growth? Which is the least effective?
  2. What combination of N, P, and K is optimal? What combination is the worst?
- `"swiss"`, standardized fertility measures and socio-economic indicators for French-speaking provinces of Switzerland at about 1888.
  1. What is the relationship in the data between fertility rates and infant mortality?
  2. How are provinces that are predominantly Catholic different from non-Catholic provinces, if at all?
  3. What factors in the data are the most important for predicting fertility?
- Examine a data set of your choice. Formulate simple questions about the data and hypothesize the answers to those questions. Demonstrate the correctness of incorrectness of each hypothesis. Explain your conclusions.

10 Pandas IV: Time Series

Lab Objective: *Many real world data sets—stock market measurements, ocean tide levels, website traffic, seismograph data, audio signals, fluid simulations, quarterly dividends, and so on—are time series, meaning they come with time-based labels. There is no universal format for such labels, and indexing by time is often difficult with raw data. Fortunately, pandas has tools for cleaning and analyzing time series. In this lab we use pandas to clean and manipulate time-stamped data and introduce some basic tools for time series analysis.*

Working with Dates and Times

The `datetime` module in the standard library provides a few tools for representing and operating on dates and times. The `datetime.datetime` object represents a *time stamp*: a specific time of day on a certain day. Its constructor accepts a four-digit year, a month (starting at 1 for January), a day, and, optionally, an hour, minute, second, and microsecond. Each of these arguments must be an integer, with the hour ranging from 0 to 23 (military time).

```
>>> from datetime import datetime

# Represent November 18th, 1991, at 2:01 PM.
>>> bday = datetime(1991, 11, 18, 14, 1)
>>> print(bday)
1991-11-18 14:01:00

# Find the number of days between 11/18/1991 and 11/9/2017.
>>> dt = datetime(2017, 11, 9) - bday
>>> dt.days
9487
```

The `datetime.datetime` object has a parser method, `strptime()`, that converts a string into a new `datetime.datetime` object. The parser is flexible because the user must specify the format that the dates are in. For example, if the dates are in the format `"Month/Day//Year::Hour"`, specify `format="%m/%d//%Y::%H"` to parse the string appropriately. See Table 10.1 for formatting options.

Pattern	Description
`%Y`	4-digit year
`%y`	2-digit year
`%m`	1- or 2-digit month
`%d`	1- or 2-digit day
`%H`	Hour (24-hour)
`%I`	Hour (12-hour)
`%M`	2-digit minute
`%S`	2-digit second

Table 10.1: Formats recognized by `datetime.strptime()`

```
>>> print(datetime.strptime("1991-11-18 / 14:01", "%Y-%m-%d / %H:%M"),
...       datetime.strptime("1/22/1996", "%m/%d/%Y"),
...       datetime.strptime("19-8, 1998", "%d-%m, %Y"), sep='\n')
1991-11-18 14:01:00                 # The date formats are now standardized.
1996-01-22 00:00:00                 # If no hour/minute/seconds data is given,
1998-08-19 00:00:00                 #  the default is midnight.
```

Converting Dates to an Index

The `TimeStamp` class is the pandas equivalent to a `datetime.datetime` object. An pandas index for composed of `TimeStamp` objects is a `DatetimeIndex`, and a `Series` or `DataFrame` with such an index is called a *time series*. The function `pd.to_datetime()` converts a collection of dates in a parsable format to a `DatetimeIndex`. The format of the dates is inferred if possible, but it can be specified explicitly with the same syntax as `datetime.strptime()`.

```
>>> import pandas as pd

# Convert some dates (as strings) into a DatetimeIndex.
>>> dates = ["2010-1-1", "2010-2-1", "2012-1-1", "2012-1-2"]
>>> pd.to_datetime(dates)
DatetimeIndex(['2010-01-01', '2010-02-01', '2012-01-01', '2012-01-02'],
              dtype='datetime64[ns]', freq=None)

# Create a time series, specifying the format for the dates.
>>> dates = ["1/1, 2010", "1/2, 2010", "1/1, 2012", "1/2, 2012"]
>>> date_index = pd.to_datetime(dates, format="%m/%d, %Y")
>>> pd.Series([x**2 for x in range(4)], index=date_index)
2010-01-01    0
2010-01-02    1
2012-01-01    4
2012-01-02    9
dtype: int64
```

Problem 1. The file `DJIA.csv` contains daily closing values of the Dow Jones Industrial Average from 2006–2016. Read the data into a `Series` or `DataFrame` with a `DatetimeIndex` as the index. Drop rows with missing values, cast the `"VALUES"` column to floats, then plot the data. (Hint: Use `lw=.5` to make the line thin enough for the data.)

Generating Time-based Indices

Some time series datasets come without explicit labels, but still have instructions for deriving timestamps. For example, a list of bank account balances might have records from the beginning of every month, or heart rate readings could be recorded by an app every 10 minutes. Use `pd.date_range()` to generate a `DatetimeIndex` where the timestamps are equally spaced. The function is analogous to `np.arange()` and has the following parameters.

Parameter	Description
`start`	Starting date
`end`	End date
`periods`	Number of dates to include
`freq`	Amount of time between consecutive dates
`normalize`	Whether or not to trim the time to midnight

Table 10.2: Parameters for `pd.date_range()`.

Exactly two of the parameters `start`, `end`, and `periods` must be specified to generate a range of dates. The `freq` parameter accepts a variety of string representations, referred to as *offset aliases*. See Table 10.3 for a sampling of some of the options. For a complete list of the options, see `http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases`.

Parameter	Description
`"D"`	calendar daily (default)
`"B"`	business daily
`"H"`	hourly
`"T"`	minutely
`"S"`	secondly
`"MS"`	first day of the month
`"BMS"`	first weekday of the month
`"W-MON"`	every Monday
`"WOM-3FRI"`	every 3rd Friday of the month

Table 10.3: Options for the `freq` parameter to `pd.date_range()`.

```
# 5 consecutive days staring with September 28, 2016.
>>> pd.date_range(start='9/28/2016 16:00', periods=5)
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-29 16:00:00',
               '2016-09-30 16:00:00', '2016-10-01 16:00:00',
               '2016-10-02 16:00:00'],
```

```
              dtype='datetime64[ns]', freq='D')

# The first weekday of every other month in 2016.
>>> pd.date_range(start='1/1/2016', end='1/1/2017', freq="2BMS" )
DatetimeIndex(['2016-01-01', '2016-03-01', '2016-05-02', '2016-07-01',
               '2016-09-01', '2016-11-01'],
              dtype='datetime64[ns]', freq='2BMS')


# 10 minute intervals between 4:00 PM and 4:30 PM on September 9, 2016.
>>> pd.date_range(start='9/28/2016 16:00',
            end='9/28/2016 16:30', freq="10T")
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-28 16:10:00',
               '2016-09-28 16:20:00', '2016-09-28 16:30:00'],
              dtype='datetime64[ns]', freq='10T')
```

The `freq` parameter also supports more flexible string representations.

```
>>> pd.date_range(start='9/28/2016 16:30', periods=5, freq="2h30min")
DatetimeIndex(['2016-09-28 16:30:00', '2016-09-28 19:00:00',
               '2016-09-28 21:30:00', '2016-09-29 00:00:00',
               '2016-09-29 02:30:00'],
              dtype='datetime64[ns]', freq='150T')
```

Problem 2. The file `paychecks.csv` contains values of an hourly employee's last 93 paychecks. He started working March 13, 2008. This company hands out paychecks on the first and third Fridays of the month.

Read in the data, using `pd.date_range()` to generate the `DatetimeIndex`. Plot the data. (Hint: use the `union()` method of `DatetimeIndex` class.)

Periods

The pandas `Timestamp` object represents a precise moment in time on a given day. Some data, however, is recorded over a time interval, and it wouldn't make sense to place an exact timestamp on any of the measurements. For example, a record of the number of steps walked in a day, box office earnings per week, quarterly earnings, and so on. This kind of data is better represented with the pandas `Period` object and the corresponding `PeriodIndex`.

The constructor of the `Period` accepts a `value` and a `freq`. The `value` parameter indicates the label for a given `Period`. This label is tied to the **end** of the defined `Period`. The `freq` indicates the length of the `Period` and also (in some cases) indicates the offset of the `Period`. The `freq` parameter accepts the majority, but not all, of frequencies listed in Table 10.3.

```
# The default value for 'freq' is "M" for months.
>>> p1 = pd.Period("2016-10")
>>> p1.start_time                    # The start and end times of the period
Timestamp('2016-10-01 00:00:00')     # are recorded as Timestamps.
```

```
>>> p1.end_time
Timestamp('2016-10-31 23:59:59.999999999')

# Represent the annual period ending in December that includes 10/03/2016.
>>> p2 = pd.Period("2016-10-03", freq="A-DEC")
>>> p2.start_time
Timestamp('2007-01-01 00:00:00')
>>> p2.end_time
Timestamp('2007-12-31 23:59:59.999999999')

# Get the weekly period ending on a Saturday that includes 10/03/2016.
>>> print(pd.Period("2016-10-03", freq="W-SAT"))
2016-10-02/2016-10-08
```

Like the `pd.date_range()` method, the `pd.period_range()` method is useful for generating a `PeriodIndex` for unindexed data. The syntax is essentially identical to that of `pd.date_range()`. When using `pd.period_range()`, remember that the `freq` parameter marks the end of the period.

```
# Represent quarters from 2008 to 2010, with Q4 ending in December.
>>> pd.period_range(start="2008", end="2010-12", freq="Q-DEC")
PeriodIndex(['2008Q1', '2008Q2', '2008Q3', '2008Q4', '2009Q1', '2009Q2',
             '2009Q3', '2009Q4', '2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='period[Q-DEC]', freq='Q-DEC')
```

After creating a `PeriodIndex`, the `freq` parameter can be changed via the `asfreq()` method.

```
# Get every three months form March 2010 to the start of 2011.
>>> p = pd.period_range("2010-03", "2011", freq="3M")
>>> p
PeriodIndex(['2010-03', '2010-06', '2010-09', '2010-12'],
            dtype='period[3M]', freq='3M')

# Change frequency to be quarterly.
>>> p.asfreq("Q-DEC")
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'],
            dtype='period[Q-DEC]', freq='Q-DEC')
```

Say you have created a `PeriodIndex`, but the bounds are not exactly where you expected they would be. You can actually shift `PeriodIndex` obejcts by adding or subtracting an integer, n. The resulting `PeriodIndex` will be shifted by $n \times$ `freq`.

```
# Shift index by 1
>>> p -= 1
>>> p
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='int64', freq='Q-DEC')
```

If for any reason you need to switch from periods to timestamps, pandas provides a very simple method to do so.

```
# Convert to timestamp (last day of each quarter)
>>> p = p.to_timestamp(how='end')
>>> p
DatetimeIndex(['2010-03-31', '2010-06-30', '2010-09-30', '2010-12-31'],
              dtype='datetime64[ns]', freq='Q-DEC')
```

Similarly, you can switch from timestamps to periods.

```
>>> p.to_period("Q-DEC")
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='int64', freq='Q-DEC')
```

Problem 3. The file `finances.csv` contains a list of simulated quarterly earnings and expense totals from a fictional company. Load the data into a `Series` or `DataFrame` with a `PeriodIndex` with a quarterly frequency. Assume the fiscal year starts at the beginning of September and that the data begins in September 1978. Plot the data.

Operations on Time Series

There are certain operations only available to Series and DataFrames that have a `DatetimeIndex`. A sampling of this functionality is described throughout the remainder of this lab.

Slicing

Slicing is much more flexible in pandas for time series. We can slice by year, by month, or even use traditional slicing syntax to select a range of dates.

```
# Select all rows in a given year
>>> df["2010"]
                   0         1
2010-01-01  0.566694  1.093125
2010-02-01 -0.219856  0.852917
2010-03-01  1.511347 -1.324036

# Select all rows in a given month of a given year
>>> df["2012-01"]
                   0         1
2012-01-01  0.212141  0.859555
2012-01-02  1.483123 -0.520873
2012-01-03  1.436843  0.596143

# Select a range of dates using traditional slicing syntax
>>> df["2010-1-2":"2011-12-31"]
                   0         1
2010-02-01 -0.219856  0.852917
```

```
2010-03-01  1.511347 -1.324036
2011-01-01  0.300766  0.934895
```

Resampling

Imagine you have a dataset that does not have datapoints at a fixed frequency. For example, a dataset of website traffic would take on this form. Because the datapoints occur at irregular intervals, it may be more difficult to procure any meaningful insight. In situations like these, *resampling* your data is worth considering.

The two main forms of resampling are *downsampling* (aggregating data into fewer intervals) and *upsampling* (adding more intervals).

To downsample, use the `resample()` method of the `Series` or `DataFrame`. This method is similar to `groupby()` in that it groups different entries o together, and requires some kind of aggregation to produce a new data set. The first parameter to `resample()` is an offset string from Table 10.3: `"D"` for daily, `"H"` for hourly, and so on.

```
>>> import numpy as np

# Get random data for every day from 2000 to 2010.
>>> dates = pd.date_range(start="2000-1-1", end='2009-12-31', freq='D')
>>> df = pd.Series(np.random.random(len(days)), index=dates)
>>> df
2000-01-01    0.559
2000-01-02    0.874
2000-01-03    0.774
                ...
2009-12-29    0.837
2009-12-30    0.472
2009-12-31    0.211
Freq: D, Length: 3653, dtype: float64

# Group the data by year.
>>> years = df.resample("A")        # 'A' for 'annual'.
>>> years.agg(len)                  # Number of entries per year.
2000-12-31    366.0
2001-12-31    365.0
2002-12-31    365.0
                ...
2007-12-31    365.0
2008-12-31    366.0
2009-12-31    365.0
Freq: A-DEC, dtype: float64

>>> years.mean()                    # Average entry by year.
2000-12-31    0.491
2001-12-31    0.514
2002-12-31    0.484
                ...
```

```
2007-12-31    0.508
2008-12-31    0.521
2009-12-31    0.523
Freq: A-DEC, dtype: float64

# Group the data by month.
>>> months = df.resample("M")
>>> len(months.mean())              # 12 months x 10 years = 120 months.
120
```

Problem 4. The file `website_traffic.csv` contains records for different visits to a fictitious website. Read in the data, calculate the duration of each visit (in seconds), and convert the index to a `DatetimeIndex`. Use downsampling to calculate the average visit duration by minute, and the average visit duration by hour. Plot both results on the same graph.

Elementary Time Series Analysis

Shifting

`DataFrame` and `Series` objects have a `shift()` method that allows you to move data up or down relative to the index. When dealing with time series data, we can also shift the `DatetimeIndex` relative to a time offset.

```
>>> df = pd.DataFrame(dict(VALUE=np.random.rand(5)),
                      index=pd.date_range("2016-10-7", periods=5, freq='D'))
>>> df
               VALUE
2016-10-07  0.127895
2016-10-08  0.811226
2016-10-09  0.656711
2016-10-10  0.351431
2016-10-11  0.608767

>>> df.shift(1)
               VALUE
2016-10-07       NaN
2016-10-08  0.127895
2016-10-09  0.811226
2016-10-10  0.656711
2016-10-11  0.351431

>>> df.shift(-2)
               VALUE
2016-10-07  0.656711
2016-10-08  0.351431
2016-10-09  0.608767
```

```
2016-10-10       NaN
2016-10-11       NaN

>>> df.shift(14, freq="D")
               VALUE
2016-10-21  0.127895
2016-10-22  0.811226
2016-10-23  0.656711
2016-10-24  0.351431
2016-10-25  0.608767
```

Shifting data makes it easy to gather statistics about changes from one timestamp or period to the next.

```
# Find the changes from one period/timestamp to the next
>>> df - df.shift(1)            # Equivalent to df.diff().
               VALUE
2016-10-07       NaN
2016-10-08  0.683331
2016-10-09 -0.154516
2016-10-10 -0.305279
2016-10-11  0.257336
```

Problem 5. Compute the following information about the DJIA dataset from Problem 1.

- The single day with the largest gain.
- The single day with the largest loss.
- The month with the largest gain.
- The month with the largest loss.

For the monthly statistics, define the gain (or loss) to be the difference between the DJIA on the last and first days of the month.

Rolling Functions and Exponentially-Weighted Moving Functions

Many time series are inherently noisy. To analyze general trends in data, we use *rolling functions* and *exponentially-weighted moving (EWM)* functions.

Rolling functions, or *moving window functions*, perform some kind of calculation on just a window of data. There are a few rolling functions that come standard with pandas.

Rolling Functions (Moving Window Functions)

One of the most commonly used rolling functions is the *rolling average*, which takes the average value over a window of data.

```
# Generate a time series using random walk from a uniform distribution.
N = 10000
bias = 0.01
s = np.zeros(N)
s[1:] = np.random.uniform(low=-1, high=1, size=N-1) + bias
s = pd.Series(s.cumsum(),
              index=pd.date_range("2015-10-20", freq='H', periods=N))

# Plot the original data together with a rolling average.
ax1 = plt.subplot(121)
s.plot(color="gray", lw=.3, ax=ax1)
s.rolling(window=200).mean().plot(color='r', lw=1, ax=ax1)
ax1.legend(["Actual", "Rolling"], loc="lower right")
ax1.set_title("Rolling Average")
```

The function call `s.rolling(window=200)` creates a `pd.core.rolling.Window` object that can be aggregated with a function like `mean()`, `std()`, `var()`, `min()`, `max()`, and so on.

Exponentially-Weighted Moving (EWM) Functions

Whereas a moving window function gives equal weight to the whole window, an *exponentially-weighted moving* function gives more weight to the most recent data points.

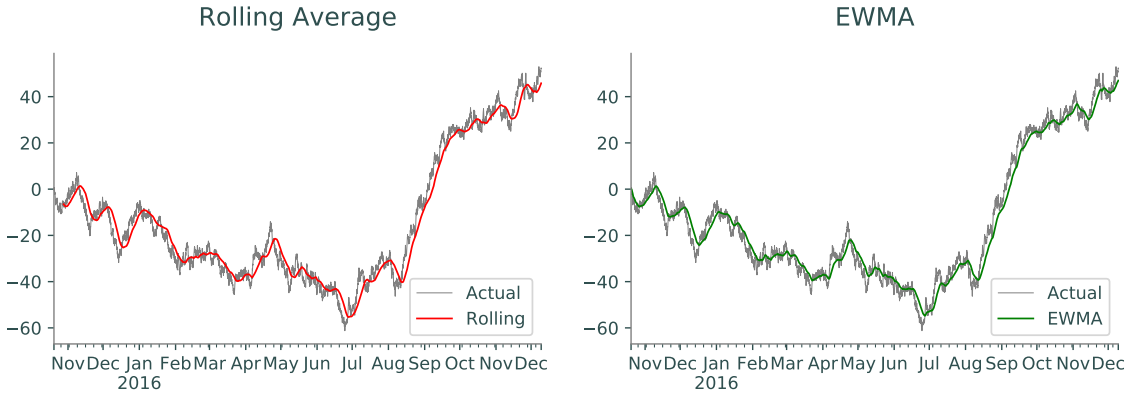
In the case of a *exponentially-weighted moving average* (EWMA), each data point is calculated as follows.

$$z_i = \alpha \bar{x}_i + (1 - \alpha) z_{i-1},$$

where z_i is the value of the EWMA at time i, $\bar{x}_i$ is the average for the i-th window, and α is the decay factor that controls the importance of previous data points. Notice that $\alpha = 1$ reduces to the rolling average.

More commonly, the decay is expressed as a function of the window size. In fact, the `span` for an EWMA is nearly analogous to `window` size for a rolling average.

Notice the syntax for EWM functions is very similar to that of rolling functions.



(a) (b)

Figure 10.1: Rolling average and EWMA.

```
ax2 = plt.subplot(121)
s.plot(color="gray", lw=.3, ax=ax2)
s.ewm(span=200).mean().plot(color='g', lw=1, ax=ax2)
ax2.legend(["Actual", "EWMA"], loc="lower right")
ax2.set_title("EWMA")
```

Problem 6. Plot the following from the DJIA dataset with a window or span of 30, 120, and 365.

- The original data points.
- Rolling average.
- Exponential average.
- Minimum rolling values.
- Maximum rolling values.

Describe how varying the length of the window changes the approximation to the data.