9 Pandas 1: 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.*

Pandas Basics

Pandas is a python library used primarily to analyze data. It combines functionality of NumPy, MatPlotLib, and SQL to create an easy to understand library that allows for the manipulation of data in various ways. In this lab we focus on the use of Pandas to analyze and manipulate data in ways similar to NumPy and SQL.

Pandas Data Structures

Series

The first pandas data structure is a `Series`. A `Series` is a one-dimensional array that can hold any datatype, similar to a `ndarray`. However, a `Series` has an `index` that gives a label to each entry. An `index` generally is used to label the data.

Typically a `Series` contains information about one feature of the data. For example, the data in a `Series` might show a class's grades on a test and the `Index` would indicate each student in the class. To initialize a `Series`, the first parameter is the data and the second is the index.

```
>>> import pandas as pd
>>>
# Initialize Series of student grades
>>> math = pd.Series(np.random.randint(0,100,4), ['Mark', 'Barbara',
...     'Eleanor', 'David'])
>>> english = pd.Series(np.random.randint(0,100,5), ['Mark', 'Barbara',
...     'David', 'Greg', 'Lauren'])
```

DataFrame

The second key pandas data structure is a `DataFrame`. A `DataFrame` is a collection of multiple `Series`. It can be thought of as a 2-dimensional array, where each row is a separate datapoint and each column is a feature of the data. The rows are labeled with an `index` (as in a `Series`) and the columns are labeled in the attribute `columns`.

There are many different ways to initialize a `DataFrame`. One way to initialize a `DataFrame` is by passing in a dictionary as the data of the `DataFrame`. The keys of the dictionary will become the labels in `columns` and the values are the `Series` associated with the label.

```
# Create a DataFrame of student grades
>>> grades = pd.DataFrame({"Math": math, "English": english})
>>> grades
          Math  English
Barbara   52.0     73.0
David     10.0     39.0
Eleanor   35.0      NaN
Greg       NaN     26.0
Lauren     NaN     99.0
Mark      81.0     68.0
```

Notice that `pd.DataFrame` automatically lines up data from both `Series` that have the same index. If the data only appears in one of the `Series`, the corresponding entry for the other `Series` is `NaN`.

We can also initialize a `DataFrame` with a NumPy array. With this method, the data is passed in as a 2-dimensional NumPy array, while the column labels and the index are passed in as parameters. The first column label goes with the first column of the array, the second with the second, and so forth. The index works similarly.

```
>>> import numpy as np
# Initialize DataFrame with NumPy array. This is identical to the grades ↩
    DataFrame above.
>>> data = np.array([[52.0, 73.0], [10.0, 39.0], [35.0, np.nan],
...     [np.nan, 26.0], [np.nan, 99.0], [81.0, 68.0]])
>>> grades = pd.DataFrame(data, columns = ['Math', 'English'], index =
...     ['Barbara', 'David', 'Eleanor', 'Greg', 'Lauren', 'Mark'])

# View the columns
>>> grades.columns
Index(['Math', 'English'], dtype='object')

# View the Index
>>> grades.index
Index(['Barbara', 'David', 'Eleanor', 'Greg', 'Lauren', 'Mark'], dtype='object'↩
    )
```

A `DataFrame` can also be viewed as a NumPy array using the attribute `values`.

```
# View the DataFrame as a NumPy array
>>> grades.values
```

```
array([[ 52.,  73.],
       [ 10.,  39.],
       [ 35.,  nan],
       [ nan,  26.],
       [ nan,  99.],
       [ 81.,  68.]])
```

Data I/O

The pandas library has functions that make importing and exporting data simple. The functions allow for a variety of file formats to be imported and exported, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

Method	Description
`to_csv()`	Write the index and entries to a CSV file
`read_csv()`	Read a csv and convert into a DataFrame
`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
`read_html()`	Read a table in an html page and convert to a DataFrame

Table 9.1: Methods for 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. Some frequently-used keyword arguments include the following:

- `delimiter`: The character that separates data fields. It is often a comma or a whitespace character.
- `header`: The row number (0 indexed) in the CSV file that contains the column names.
- `index_col`: The column (0 indexed) in the CSV file that is the index for the `DataFrame`.
- `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.

Another particularly useful function is `read_html()`, which is useful when scraping data. It takes in a url or html file and an optional argument `match`, a string or regex, and returns a list of the tables that match the `match` in a DataFrame. While the resulting data will probably need to be cleaned, it is frequently much faster than scraping a website.

Data Manipulation

Accessing Data

In general, the best way to access data in a `Series` or `DataFrame` is through the indexers `loc` and `iloc`. While array slicing can be used, it is more efficient to use these indexers. Accessing `Series` and `DataFrame` objects using these indexing operations is more efficient than slicing because the bracket indexing has to check many cases before it can determine how to slice the data structure. Using `loc` or `iloc` explicitly bypasses these extra checks. The `loc` index selects rows and columns based on their labels, while `iloc` selects them based on their integer position. With these indexers, the first and second arguments refer to the rows and columns, respectively, just as array slicing.

```
# Use loc to select the Math scores of David and Greg
>>> grades.loc[['David', 'Greg'],'Math']
David    10.0
Greg      NaN
Name: Math, dtype: float64

# Use iloc to select the Math scores of David and Greg
>>> grades.iloc[[1,3], 0]
David    10.0
Greg      NaN
```

To access an entire column of a `DataFrame`, the most efficient method is to use only square brackets and the name of the column, without the indexer. This syntax can also be used to create a new column or reset the values of an entire column.

```
# Create a new History column with array of random values
>>> grades['History'] = np.random.randint(0,100,6)
>>> grades['History']
Barbara     4
David      92
Eleanor    25
Greg       79
Lauren     82
Mark       27
Name: History, dtype: int64

# Reset the column such that everyone has a 100
>>> grades['History'] = 100.0
>>> grades
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Eleanor  35.0      NaN    100.0
Greg      NaN     26.0    100.0
Lauren    NaN     99.0    100.0
Mark     81.0     68.0    100.0
```

Datasets can often be very large and thus difficult to visualize. Pandas has various methods to make this easier. The methods `head` and `tail` will show the first or last n data points, respectively, where n defaults to 5. The method `sample` will draw n random entries of the dataset, where n defaults to 1.

```
# Use head to see the first n rows
>>> grades.head(n=2)
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0

# Use sample to sample a random entry
>>> grades.sample()
        Math  English  History
Lauren   NaN     99.0    100.0
```

It may also be useful to re-order the columns or rows or sort according to a given column.

```
# Re-order columns
>>> grades.reindex(columns=['English','Math','History'])
         English  Math  History
Barbara     73.0  52.0    100.0
David       39.0  10.0    100.0
Eleanor      NaN  35.0    100.0
Greg        26.0   NaN    100.0
Lauren      99.0   NaN    100.0
Mark        68.0  81.0    100.0

# Sort descending according to Math grades
>>> grades.sort_values('Math', ascending=False)
         Math  English  History
Mark     81.0     68.0    100.0
Barbara  52.0     73.0    100.0
Eleanor  35.0      NaN    100.0
David    10.0     39.0    100.0
Greg      NaN     26.0    100.0
Lauren    NaN     99.0    100.0
```

Other methods used for manipulating `DataFrame` and `Series` panda structures can be found in Table 9.2.

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

Problem 1. The file `budget.csv` contains the budget of a college student over the course of 4 years. Write a function that performs the following operations in this order:

1. Read in `budget.csv` as a `DataFrame` with the index as column 0. Hint: Use `index_col=0` to set the first column as the index when reading in the csv.
2. Reindex the columns such that amount spent on groceries is the first column and all other columns maintain the same ordering.
3. Sort the `DataFrame` in descending order by how much money was spent on `Groceries`.
4. Reset all values in the `'Rent'` column to `800.0`.
5. Reset all values in the first 5 data points to `0.0`.

Return the values of the updated `DataFrame` as a NumPy array.

Basic Data Manipulation

Because the primary pandas data structures are based off of `ndarray`, most NumPy functions work with pandas structures. For example, basic vector operations work as would be expected:

```
# Sum history and english grades of all students
>>> grades['English'] + grades['History']
Barbara    173.0
David      139.0
Eleanor      NaN
Greg       126.0
Lauren     199.0
Mark       168.0
dtype: float64

# Double all Math grades
>>>  grades['Math']*2
Barbara    104.0
David       20.0
```

```
Eleanor     70.0
Greg         NaN
Lauren       NaN
Mark       162.0
Name: Math, dtype: float64
```

In addition to arithmetic, `Series` has a variety of other methods similar to NumPy arrays. A collection of these methods is found in Table 9.3.

Method	Returns
`abs()`	Object with absolute values taken (of numerical data)
`idxmax()`	The index label of the maximum value
`idxmin()`	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 9.3: Numerical methods of the `Series` and `DataFrame` pandas classes.

Basic Statistical Functions

The pandas library allows us to easily calculate basic summary statistics of our data, which can be useful when we want a quick description of the data. The `describe()` function outputs several such summary statistics for each column in a `DataFrame`:

```
# Use describe to better understand the data
>>> grades.describe()
            Math   English  History
count   4.000000   5.00000      6.0
mean   44.500000  61.00000    100.0
std    29.827281  28.92231      0.0
min    10.000000  26.00000    100.0
25%    28.750000  39.00000    100.0
50%    43.500000  68.00000    100.0
75%    59.250000  73.00000    100.0
max    81.000000  99.00000    100.0
```

Functions for calculating means and variances, the covariance and correlation matrices, and other basic statistics are also available.

```
# Find the average grade for each student
```

```
>>> grades.mean(axis=1)
Barbara    75.000000
David      49.666667
Eleanor    67.500000
Greg       63.000000
Lauren     99.500000
Mark       83.000000
dtype: float64

# Give correlation matrix between subjects
>>> grades.corr()
            Math  English  History
Math     1.00000  0.84996      NaN
English  0.84996  1.00000      NaN
History      NaN      NaN      NaN
```

The method `rank()` can be used to rank the values in a data set, either within each entry or with each column. This function defaults ranking in ascending order: the least will be ranked 1 and the greatest will be ranked the highest number.

```
# Rank each student's performance in their classes in descending order
# (best to worst)
# The method keyword specifies what rank to use when ties occur.
>>> grades.rank(axis=1,method='max',ascending=False)
         Math  English  History
Barbara   3.0      2.0      1.0
David     3.0      2.0      1.0
Eleanor   2.0      NaN      1.0
Greg      NaN      2.0      1.0
Lauren    NaN      2.0      1.0
Mark      2.0      3.0      1.0
```

These methods can be very effective in interpreting data. For example, the `rank()` example above shows use that Barbara does best in History, then English, and then Math.

Dealing with Missing Data

Missing data is a ubiquitous problem in data science. Fortunately, pandas is particularly well-suited to handling missing or 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`. If we are not interested in the missing values, we can simply drop them from the data altogether, or we can fill them with some other value, such as the mean. `NaN` might also mean something specific, such as some default value, which should inform what to do with `NaN` values.

```
# Grades with all NaN values dropped
>>> grades.dropna()
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
```

```
Mark      81.0     68.0    100.0

# fill missing data with 50.0
>>> grades.fillna(50.0)
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Eleanor  35.0     50.0    100.0
Greg     50.0     26.0    100.0
Lauren   50.0     99.0    100.0
Mark     81.0     68.0    100.0
```

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

ACHTUNG!

Always consider missing data carefully when analyzing a dataset. It may not always be helpful to drop the data or fill it in with a random number. Consider filling the data with the mean of surrounding data or the mean of the feature in question. Overall, the choice for how to fill missing data should make sense with the dataset.

Problem 2. Write a function which uses `budget.csv` to answer the questions "Which category affects living expenses the most? Which affects other expenses the most?" Perform the following manipulations:

1. Fill all `NaN` values with `0.0`.
2. Create two new columns, `'Living Expenses'` and `'Other'`. Set the value of `'Living Expenses'` to be the sum of the columns `'Rent'`, `'Groceries'`, `'Gas'` and `'Utilities'`. Set the value of `'Other'` to be the sum of the columns `'Dining Out'`, `'Out With Friends'` and `'Netflix'`.
3. Identify which column, other than `'Living Expenses'`, correlates most with `'Living Expenses'` and which column, other than `'Other'`, correlates most with `'Other'`. This can indicate which columns in the budget affect the overarching categories the most.

Return the names of each of those columns as a tuple. The first should be of the column corresponding to `'Living Expenses'` and the second to `'Other'`.

Complex Operations in Pandas

Often times, the data that we have is not exactly the data we want to analyze. In cases like this we use more complex data manipulation tools to access only the data that we need.

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

```
>>> 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 helpful 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)
```

Masks

Sometimes, we only want to access data from a single column. For example if we want to only access the `ID` of the students in the `studentInfo DataFrame`, then we would use the following syntax.

```
# Get the ID column from studentInfo
>>> studentInfo.ID # or studentInfo['ID']
   ID
0   0
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
```

If we want to access multiple columns at once we can use a list of column names.

```
# Get the ID and Age columns.
>>> studentInfo[['ID', 'Age']]
   ID  Age
0   0   20
1   1   21
2   2   18
3   3   22
4   4   19
5   5   20
6   6   20
7   7   19
8   8   29
```

Now we can access the specific columns that we want. However, some of these columns may still contain data points that we don't want to consider. In this case we can build a mask. Each mask that we build will return a pandas `Series` object with a `bool` value at each index indicating if the condition is satisfied.

```
# Create a mask for all student receiving financial aid.
>>> mask = otherInfo['Financial_Aid'] == 'y'
# Access other info where the mask is true and display the ID and GPA ↩
    columns.
>>> otherInfo[mask][['ID', 'GPA']]
   ID  GPA
0   0  3.8
3   3  3.9
7   7  3.4
```

We can also create compound masks with multiple statements. We do this using the same syntax you would use for a compound mask in a normal NumPy array. Useful operators are `&`, the `AND` operator; `|`, the `OR` operator; and ∼, the `NOT` operator.

```
# Get all student names where Class = 'J' OR Class = 'Sp'.
>>>  mask = (studentInfo.Class == 'J') | (studentInfo.Class == 'Sp')
>>>  studentInfo[mask].Name
0        Mylan
4        Jason
5         Remi
6         Matt
7    Alexander
Name: Name, dtype: object
# This can also be acomplished with the following command:
# studentInfo[studentInfo['Class'].isin(['J','Sp'])]['Name']
```

Problem 3. Read in the file `crime_data.csv` as a pandas object. The file contains data on types of crimes in the U.S. from 1960 to 2016. Set the index as the column `'Year'`. Answer the following questions using the pandas methods learned in this lab. The answer of each question should be saved as indicated. Return the answers to all three questions as a tuple (i.e. `(answer_1,answer_2,answer_3)`).

1. Identify the three crimes that have a mean yearly number of occurences over 1,500,000. Of these three crimes, which two are very correlated? Which of these two crimes has a greater maximum value? Save the title of this column as a variable to return as the answer.
2. Examine the data from 2000 and later. Sort this data (in ascending order) according to number of murders. Find the years where aggravated assault is greater than 850,000. Save the indices (the years) of the masked and reordered `DataFrame` as a NumPy array to return as the answer.
3. What year had the highest crime rate? In this year, which crime was committed the most? What percentage of the total crime that year was it? Save this value as a float.

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.

```
>>> 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 so 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 9.4 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 9.4: 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. A pandas index composed of `TimeStamp` objects is a `DatetimeIndex`, and a `Series` or `DataFrame` with a `DatetimeIndex` 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 DatetimeIndex.
>>> 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 4. 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 any rows without numerical values, cast the `"VALUE"` column to floats, then return the updated DataFrame.

Hint: You can change the column type the same way you'd change a numpy array type.

Generating Time-based Indices

Some time series datasets come without explicit labels but 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`	Normalizes the start and end times to midnight

Table 9.5: Parameters for `pd.date_range()`.

Exactly three of the parameters `start`, `end`, `periods`, and `freq` 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 9.6 for a sampling of some of the options. For a complete list of the options, see `https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliasesl`.

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

Table 9.6: Options for the `freq` parameter to `pd.date_range()`.

```
# Create a DatetimeIndex for 5 consecutive days starting on 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')

# Create a DatetimeIndex with 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')


# Create a DatetimeIndex for 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')

# Create a DatetimeIndex for 2 hour 30 minute intervals between 4:30 PM and ↩
   2:30 AM on September 29, 2016.
>>> 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 5. The file `paychecks.csv` contains values of an hourly employee's last 93 paychecks. Paychecks are given every other Friday, starting on March 14, 2008, and the employee started working on March 13, 2008.

Read in the data, using `pd.date_range()` to generate the `DatetimeIndex`. Set this as the new index of the `DataFrame` and return the `DataFrame`.

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 6. Compute the following information about the DJIA dataset from Problem 4 that has a DateTimeIndex.

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

Return the DateTimeIndex of the day with the largest gain and the day with the largest loss. (Hint: Call your function from Problem 4 to get the DataFrame already cleaned and with DatetimeIndex).

More information on how to use `datetime` with Pandas is in the additional material section. This includes working with `Periods` and more analysis with time series.

Additional Material

SQL Operations in pandas

`DataFrames` are tabular data structures bearing an obvious resemblance to a typical relational database table. SQL is the standard for working with relational databases; however, pandas can accomplish many of the same tasks as SQL. The SQL-like functionality of pandas is one of its biggest advantages, eliminating the need to switch between programming languages for different tasks. Within pandas, we can handle both the querying *and* data analysis.

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

```
>>> 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})
```

SQL SELECT statements can be done by column indexing. WHERE statements can be included by adding masks (just like in a NumPy array). The method `isin()` can also provide a useful WHERE statement. This method accepts a list, dictionary, or `Series` containing possible values of the `DataFrame` or `Series`. When called upon, it returns a `Series` of booleans, indicating whether an entry contained a value in the parameter pass into `isin()`.

```
# SELECT ID, Age FROM studentInfo
>>> studentInfo[['ID', 'Age']]
   ID  Age
0   0   20
1   1   21
2   2   18
3   3   22
4   4   19
5   5   20
6   6   20
7   7   19
8   8   29

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

```
   ID  GPA
0   0  3.8
3   3  3.9
7   7  3.4

# SELECT Name FROM studentInfo WHERE Class = 'J' OR Class = 'Sp'
>>>  studentInfo[studentInfo['Class'].isin(['J','Sp'])]['Name']
0        Mylan
4        Jason
5         Remi
6         Matt
7    Alexander
Name: Name, dtype: object
```

Next, let's look at JOIN statements. In pandas, this is done with the `merge` function. `merge` takes the two `DataFrame` objects to join as parameters, 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]
```

More Datetime with Pandas

Periods

A 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 `Period` class 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 in some cases can also indicate the offset of the `Period`. The default value for `freq` is "M" for months. The `freq` parameter accepts the majority, but not all, of frequencies listed in Table 9.6.

```
# Creates a period for month of Oct, 2016.
>>> 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('2016-01-01 00:00:00')
> p2.end_time
Timestamp('2016-12-31 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. After creating a `PeriodIndex`, the `freq` parameter can be changed via the `asfreq()` method.

```
# 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')

# 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')
```

The bounds of a `PeriodIndex` object can be shifted by adding or subtracting an integer. `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. The `how` parameter can be `start` or `end` and determines if the timestamp is the beginning or the end of the period. Similarly, you can switch from timestamps to periods.

```
# 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')

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

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

Some datasets do not have datapoints at a fixed frequency. For example, a dataset of website traffic has datapoints that occur at irregular intervals. In situations like these, *resampling* can help provide insight on the data.

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 together. Then aggregation produces a new data set. The first parameter to `resample()` is an offset string from Table 9.6: `"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
```

Elementary Time Series Analysis

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 a calculation on 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.

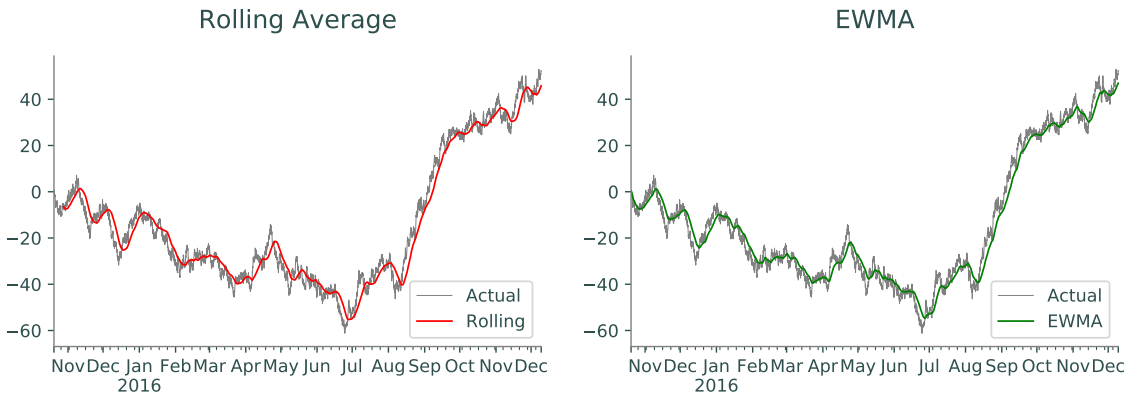
In the case of a *exponentially-weighted moving average* (EWMA), each data point is calculated as follows.

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

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 9.1: Rolling average and EWMA.

```
ax2 = plt.subplot(122)
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")
```