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?