The pandas Library: Cutting, Sorting, and Analyzing Data

Reminder: the pandas DataFrame

- A pandas DataFrame is similar to a table in Excel.
  - Each column can hold a different piece of information, like First Name, Last Name, Student ID number, and GPA.
  - In this example, each row holds data for a unique student.

```
df = pd.DataFrame()
df['a'] = [1, 1, 2, 3, 5, 8]
df['b'] = [0, 1, 2, 3, 4, 5]
df['c'] = 2 * df.a
df['d'] = df.b**2
```

Methods for Column Stats

- Mean (average): `.mean()`
  - Returns the average of a column.
- Minimum: `.min()`
  - Returns the minimum value in a column.
- Maximum: `.max()`
  - Returns the maximum value in a column.

```
df = pd.DataFrame()
df['a'] = [1, 1, 2, 3, 5, 8]
df['b'] = [0, 1, 2, 3, 4, 5]
df['c'] = 2 * df.a
df['d'] = df.b**2

test = df.b.mean()
print(test)
```

```
2.5
```

```
print(df.a.max()) # This code prints out the maximum value in the a column: 8
print(df.a.min()) # This code prints out the minimum value in the a column: 1
```

```
8
1
```

The .loc[] command

- The `.loc[]` command allows one to access a subset of rows and/or columns in a DataFrame.
  - The first argument selects rows
  - The second argument selects columns

```
# The .loc[first, second] method requires the first argument
# to select rows from a DataFrame,
# and the second argument to select columns from that DataFrame.
# The example below requires all rows where the value of a is greater than 1.
# The ':' essentially just means all, so in this case all of the columns
sliceSomeRows = df.loc[df.a > 1, :]
# this leaves us with 4 rows.
sliceSomeRows.head()
```

	a	b	c	d
2	2	2	4	4
3	3	3	6	9
4	5	4	10	16
5	8	5	16	25

The .loc[] command

- You can visualize the `.loc[]` command as filtering out unwanted rows (e.g.) in an Excel table:

DataFrame			
a	**b**	**c**	**d**
1	0	2	0
1	1	2	1
2	2	4	4
3	3	6	9
5	4	10	16
8	5	16	25

```
sliceSomeRows = df.loc[df.a > 1, :]
```

Think of this command as keeping only rows with a > 1

The .loc[] command

- You can visualize the `.loc[]` command as filtering out unwanted rows (e.g.) in an Excel table:

```
sliceSomeRows = df.loc[df.a > 1, :]
```

DataFrame			
a	**b**	**c**	**d**
1	0	2	0
1	1	2	1
2	2	4	4
3	3	6	9
5	4	10	16
8	5	16	25

	a	b	c	d
2	2	2	4	4
3	3	3	6	9
4	5	4	10	16
5	8	5	16	25

The red rows do not satisfy the condition, while the green rows do satisfy the condition

The .loc[] command returns only the green rows from our illustration

The .loc[] command

- Multiple conditions can be required simultaneously by using the & symbol.

DataFrame			
a	**b**	**c**	**d**
1	0	2	0
1	1	2	1
2	2	4	4
3	3	6	9
5	4	10	16
8	5	16	25

```
# The .loc[] method requires the first argument to select rows from a DataFrame,
# and the second argument to select columns from that DataFrame.
# Here, we are selecting all rows where a is not equal to 1 AND
# b is less than 5.  This leaves us with only three rows:
mySlice = df.loc[(df.a != 1)&(df.b < 5),['b', 'd']]
mySlice.head()
```

	b	d
2	2	4
3	3	9
4	4	16

Method Chaining

- The result of one method can be the input of another method
- Think of these as acting “from the inside-out”

This example first requires that all entries have an ‘a’ greater than 4, and then takes the mean of column ‘b’.

```
# Remember, the .loc[] method requires the first argument to select rows from a DataFrame,
# and the second argument to select columns from that DataFrame.
# Notice that we are calculating the minimum value in the 'b' column,
# But we are only considering the entries corresponding to when a > 4.
# The only entries in column a that are greater than 4 are the last two, 5 and 8.
# But since we are finding the mean of column b satisfying these cuts on column a,
# this command takes the average of the last two entries in the b column: 4 and 5.
tmp = df.loc[df.a > 4, 'b'].mean()
print(tmp)
```

```
4.5
```

Card Question

- Which of the following examples of method chaining will return the average of the Score column only for rows with the Year column set to sophomores? Assume the DataFrame is named data.
  - A) data.loc[“Score”, “Year”].mean()
  - B) data.loc[data.Score, data.Year == ‘Sophomore’].mean()
  - C) data.loc[data.Year == ’Sophomore’, ‘Score’].mean()
  - D) data.Score.mean()
  - E) None of the above.

Sorting Data

- The `.sort_values()` method enables the sorting of DataFrames in ascending or descending order.
- The `.sort_values()` method does not change the order of the data in a DataFrame, it merely returns the sorted values for display or other uses.

```
df = pd.DataFrame()
df['Name'] = ['Bob', 'Tim', 'Jan', 'Dan', "Ann"]
df['Grade'] = [90, 62, 100, 88, 75]
df['FavoriteColor'] = ['Yellow', 'Orange', 'Red', 'Blue', 'Green']

df.sort_values('Grade', ascending=False)
```

	Name	Grade	FavoriteColor
2	Jan	100	Red
0	Bob	90	Yellow
3	Dan	88	Blue
4	Ann	75	Green
1	Tim	62	Orange

Sorting Data

- The sorted values can be stored in a new DataFrame object by using the assignment operator (=).

- Note that sorting a column of string type data alphabetizes the column.

```
newdf = df.sort_values('Name', ascending=True)
newdf.head()
```

	Name	Grade	FavoriteColor
4	Ann	75	Green
0	Bob	90	Yellow
3	Dan	88	Blue
2	Jan	100	Red
1	Tim	62	Orange