CS167: Machine Learning

Useful Pandas Functions
Practice Exercises

Wednesday, February 4th, 2026

Drake
UNIVERSITY

Recap: Helpful DataFrame Methods

- The `.head()` method can be called on any `DataFrame`, and by default will display the first 5 lines/rows of the data, as well as the names of the columns.

```
df_rest.head(7)
```

	alt	bar	fri	hun	pat	price	rain	res	type	est	target
0	Yes	No	No	Yes	Some	$$$	No	Yes	French	0-10	Yes
1	Yes	No	No	Yes	Full	$	No	No	Thai	30-60	No
2	No	Yes	No	No	Some	$	No	No	Burger	0-10	Yes
3	Yes	No	Yes	Yes	Full	$	No	No	Thai	10-30	Yes
4	Yes	No	Yes	No	Full	$$$	No	Yes	French	>60	No
5	No	Yes	No	Yes	Some	$$	Yes	Yes	Italian	0-10	Yes
6	No	Yes	No	No	NaN	$	Yes	No	Burger	0-10	No

- The `.shape()` method can be called on any `DataFrame`, and it will show the dimensions ie, *number of rows* and *number of columns*

There are 12 rows
And 11 columns

```
[ ] df_rest.shape

(12, 11)
```

Recap: Helpful DataFrame Methods

- Want to see a list of all of the column names in your dataset? Try using `df.columns`

```
[4] df_rest.columns

Index(['alt', 'bar', 'fri', 'hun', 'pat', 'price', 'rain', 'res', 'type',
       'est', 'target'],
      dtype='object')
```

- If there are no spaces in the name of a column, you can also reference it using dot notation like so: `df.type`

```
df_rest.type
```

	type
0	French
1	Thai
2	Burger
3	Thai
4	French
5	Italian
6	Burger
7	Thai
8	Burger
9	Italian
10	Thai
11	Burger

Recap: Creating `DataFrame`

- The syntax for creating a `DataFrame` from scratch looks like this: pandas.DataFrame(data, index, columns)

Empty DataFrame

```
df = pd.DataFrame() # creates an empty DataFrame
print(df)
```

```
Empty DataFrame
Columns: []
Index: []
```

DataFrame from 1D List

```
[15] data = [10, 20, 30, 40, 50, 60]
     df_1 = pd.DataFrame(data, columns=['numbers'])
     print('size of the dataframe df_2', df_1.shape)
     df_1
```

```
size of the dataframe df_2 (6, 1)
```

	numbers
0	10
1	20
2	30
3	40
4	50
5	60

Recap: Creating `DataFrame`

- The syntax for creating a `DataFrame` from scratch looks like this: pandas.DataFrame(data, index, columns)

DataFrame from 2D List

```
# Example#8: initialize list of lists (each inner list corresponds to one row in the DataFrame)
data_2d_list = [['reza', 1], ['chris', 2], ['eric', 3]]

# Create the pandas DataFrame
df_3 = pd.DataFrame(data_2d_list, columns=['name', 'score'])

# print dataframe.
df_3
```

	name	score
0	reza	1
1	chris	2
2	eric	3

Recap: Creating `DataFrame`

- The syntax for creating a `DataFrame` from scratch looks like this: pandas.DataFrame(data, index, columns)

DataFrame from Dictionary

```
# Example#6: Initializing a DataFrame with a dictionary of items allows you to specify the column names along with their corresponding values.
data_source = {'first name': ['a', 'b', 'c'], 'last name':['A', 'B', 'C']}
df_2 = pd.DataFrame(data_source)
print(df_2)

# Example#7: Initializing a DataFrame with a dictionary of items allows you to specify the column names along with their corresponding values.
data_source = {"first name":["alimoor", "chris", "eric"], "last name":["reza", "porter", "manley"], "scores":[2, 3, 4]}
df_2 = pd.DataFrame(data_source)
df_2.head()
```

```
  first name last name
0          a         A
1          b         B
2          c         C
```

	first name	last name	scores
0	alimoor	reza	2
1	chris	porter	3
2	eric	manley	4

Recap: Selecting Rows in DataFrames using `loc` and `iloc`:

- Simply put:
  - `loc` gets DataFrame rows and columns by **labels/names**
  - `iloc` gets DataFrame rows and columns by **index/position**

Recap: Selecting Rows in DataFrames: `loc`

- `loc` gets DataFrame rows and columns by **labels/names**
- Let's take a subset of titanic and try to use `loc`:

```
[51] subset = df_titanic.loc[800:805] # since it's a label, it will take rows labeled 800, 801, 802, 803, 804, and 805.
     print(subset)
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
800	0	2	male	34.00	0	0	13.0000	S	Second	man	True	NaN	Southampton	no	True
801	1	2	female	31.00	1	1	26.2500	S	Second	woman	False	NaN	Southampton	yes	False
802	1	1	male	11.00	1	2	120.0000	S	First	child	False	B	Southampton	yes	False
803	1	3	male	0.42	0	1	8.5167	C	Third	child	False	NaN	Cherbourg	yes	False
804	1	3	male	27.00	0	0	6.9750	S	Third	man	True	NaN	Southampton	yes	True
805	0	3	male	31.00	0	0	7.7750	S	Third	man	True	NaN	Southampton	no	True

labels/names (not indices)

ALERT: `print(subset)` shows all 6 rows

Recap: Selecting Rows in DataFrames: iloc

- `iloc` gets DataFrame rows and columns by **index/position**

```
[51] subset = df_titanic.loc[800:805] # since it's a label, it will take rows labeled 800, 801, 802, 803, 804, and 805.
     print(subset)
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
800	0	2	male	34.00	0	0	13.0000	S	Second	man	True	NaN	Southampton	no	True
801	1	2	female	31.00	1	1	26.2500	S	Second	woman	False	NaN	Southampton	yes	False
802	1	1	male	11.00	1	2	120.0000	S	First	child	False	B	Southampton	yes	False
803	1	3	male	0.42	0	1	8.5167	C	Third	child	False	NaN	Cherbourg	yes	False
804	1	3	male	27.00	0	0	6.9750	S	Third	man	True	NaN	Southampton	yes	True
805	0	3	male	31.00	0	0	7.7750	S	Third	man	True	NaN	Southampton	no	True

```
subset.iloc[0]  #works
```

```
survived                0
pclass                  2
sex                  male
age                  34.0
sibsp                   0
parch                   0
fare                 13.0
embarked                S
class              Second
who                   man
adult_male           True
deck                  NaN
embark_town    Southampton
alive                  no
alone                True
Name: 800, dtype: object
```

```
subset.iloc[1]  #works
```

```
survived                1
pclass                  2
sex                female
age                  31.0
sibsp                   1
parch                   1
fare                26.25
embarked                S
class              Second
who                 woman
adult_male          False
deck                  NaN
embark_town    Southampton
alive                  yes
alone                False
Name: 801, dtype: object
```

```
subset.iloc[5]  #works
```

```
survived                0
pclass                  3
sex                  male
age                  31.0
sibsp                   0
parch                   0
fare                7.775
embarked                S
class              Third
who                   man
adult_male           True
deck                  NaN
embark_town    Southampton
alive                   no
alone                True
Name: 805, dtype: object
```

Today's Agenda

- Rows in a DataFrame

- Subsetting (Columns, Rows, or both) in a DataFrame
  - Select subset of Columns in a DataFrame
  - Select subset of Rows in a DataFrame
  - Select **subsets** of the DataFrame (both rows and columns)

Subsetting Rows in a DataFrame

- Why might you want a subset of the rows?

- Maybe you want only rows that satisfy a certain condition--in the restaurant dataset, maybe:
  - Italian Restaurants
  - only examples when it didn't rain
  - etc.

Subsetting Rows in a DataFrame

- To understand the syntax for subsetting rows in a DataFrame, we need to understand how conditionals work in Python/ Pandas:
  - to check whether each row in a `DataFrame` meets a criteria, use the following syntax
  - it will return a `Series` with `True/False`, where rows that are `True` meet the criteria, and `False` do not

What is a Series?
Next slide

```
df_rest['type'] == 'French'
```

Pandas Datatypes: `Series`

- Pandas Documentation defines `Series` as:
  - `Series` are 1D arrays with axis labels
    - Each **row** in a DataFrame is a `Series`
    - Each **column** in a DataFrame is also a `Series`.

```
print(type(restaurant_data.iloc[0])) #the first row in the dataframe
print(type(restaurant_data['type'])) #the column 'type' from the dataframe
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
```

Subsetting Rows in a DataFrame

```
data is a  <class 'pandas.core.frame.DataFrame'>
```

	alt	bar	fri	hun	pat	price	rain	res	type	est	target
0	Yes	No	No	Yes	Some	$$$	No	Yes	French	0-10	Yes
1	Yes	No	No	Yes	Full	$	No	No	Thai	30-60	No
2	No	Yes	No	No	Some	$	No	No	Burger	0-10	Yes
3	Yes	No	Yes	Yes	Full	$	No	No	Thai	10-30	Yes
4	Yes	No	Yes	No	Full	$$$	No	Yes	French	>60	No

```
condition = df_rest['type'] == 'French'
condition
```

```
0      True
1     False
2     False
3     False
4      True
5     False
6     False
7     False
8     False
9     False
10    False
11    False
```

Subsetting Rows in a DataFrame

- Taking this one step further, we can use this boolean Series to filter our rows:
  - `condition = df[ 'column name' ] == 'something'`
  - `subset_rows = df[ condition ]`

```
condition = df_rest['type'] == 'French'
french_rest = df_rest[condition]
french_rest
```

	alt	bar	fri	hun	pat	price	rain	res	type	est	target
0	Yes	No	No	Yes	Some	$$$	No	Yes	French	0-10	Yes
4	Yes	No	Yes	No	Full	$$$	No	Yes	French	>60	No

Subsetting Rows in a DataFrame

- It can be done in one step as follows:
  - `subset_rows = df[ df[ 'column name' ] == 'something' ]`

```
french_rest = df_rest[ df_rest['type'] == 'French' ]
french_rest
```

	alt	bar	fri	hun	pat	price	rain	res	type	est	target
0	Yes	No	No	Yes	Some	$$$	No	Yes	French	0-10	Yes
4	Yes	No	Yes	No	Full	$$$	No	Yes	French	>60	No

Exercise#1:
Practice Time, Your Turn

💬 Group Exercise:

See if you can create a subset called rainy_day, of rows where it rained from the dataset 'restaurant.csv'

```
[ ]  # your code
     # ...
```

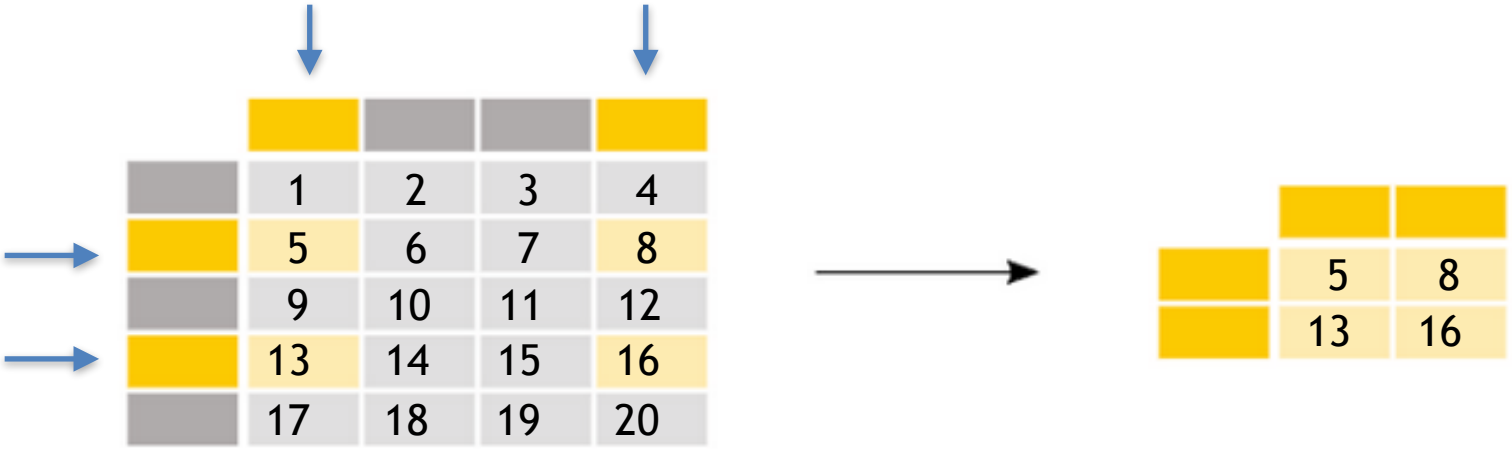
Today's Agenda

- Rows in a DataFrame

- Subsetting (Columns, Rows, or both) in a DataFrame
  - Select subset of Columns in a DataFrame
  - Select subset of Rows in a DataFrame
  - Select **subsets** of the DataFrame (both rows and columns)

Subsetting Both Columns and Rows

- Let's imagine we want a subset that contains the ages of people who did not survive the `Titanic`. Technically, you have the knowledge now to be able to do this, if you just break it up into two steps
  - **Step 1 (Row Selection):** make a subset, `victims`, of rows where `survived == 0`
  - **Step 2 (Column Selection):** use `victims` to create a second subset that only contains the `'Age'` column.



Subsetting Columns and Rows

- Let's imagine we want a subset that contains the ages of people who did not survive the `Titanic`. Technically, you have the knowledge now to be able to do this, if you just break it up into two steps

  - **Step 1 (Row Selection):** make a subset, `victims`, of rows where `survived == 0`
  - **Step 2 (Column Selection):** use `victims` to create a second subset that only contains the `'Age'` column.

```
import pandas as pd
df_titanic = pd.read_csv('/content/drive/MyDrive/cs167_sp26/datasets/titanic.csv') #make sure the path on the line below correspond
df_titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

Approach#1: Subsetting Columns and Rows

- Let's imagine we want a subset that contains the ages of people who did not survive the `Titanic`. Technically, you have the knowledge now to be able to do this, if you just break it up into two steps

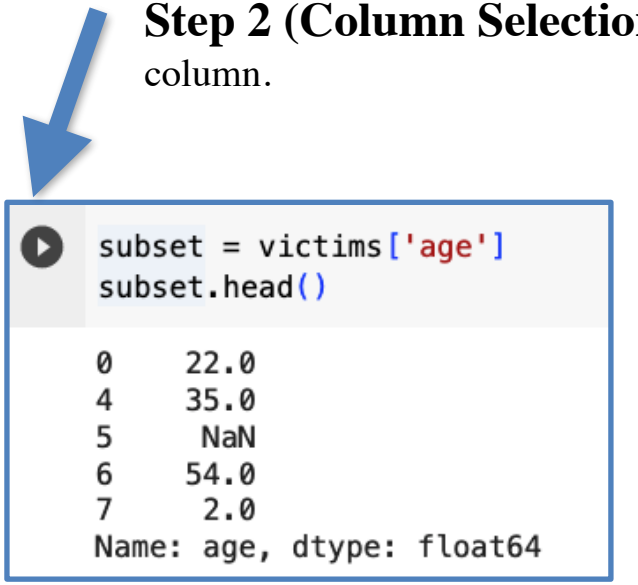
- **Step 1 (Row Selection):** make a subset, `victims`, of rows where `survived == 0`
- **Step 2 (Column Selection):** use `victims` to create a second subset that only contains the `'Age'` column.

```
victims = df_titanic[ df_titanic['survived'] == 0 ]
victims.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True
5	0	3	male	NaN	0	0	8.4583	Q	Third	man	True	NaN	Queenstown	no	True
6	0	1	male	54.0	0	0	51.8625	S	First	man	True	E	Southampton	no	True
7	0	3	male	2.0	3	1	21.0750	S	Third	child	False	NaN	Southampton	no	False

Approach#1: Subsetting Columns and Rows

- Let's imagine we want a subset that contains the ages of people who did not survive the `Titanic`. Technically, you have the knowledge now to be able to do this, if you just break it up into two steps
  - **Step 1 (Row Selection):** make a subset, `victims`, of rows where `survived == 0`
  - **Step 2 (Column Selection):** use `victims` to create a second subset that only contains the `'Age'` column.



```
subset = victims['age']
subset.head()
```

```
0    22.0
4    35.0
5     NaN
6    54.0
7     2.0
Name: age, dtype: float64
```

Approach#2: Subsetting Columns and Rows with *loc*

- Let's imagine we want a subset that contains the *ages* of people who did not survive the `Titanic`. We can actually do this on one step if we use `loc`:

In a single step

```
subset = df_titanic.loc[ df_titanic['survived']==0, 'age' ]
subset.head()
```

	age
0	22.0
4	35.0
5	NaN
6	54.0
7	2.0

Approach#2: Subsetting Columns and Rows with *loc*

- Let's imagine we want a subset that contains the *`fare`* and *`ages`* from `Titanic`. We can actually do this on one step if we use `loc`:

In a single step

```
victims = df_titanic.loc[ df_titanic['survived']== 0, ['fare', 'age'] ]
victims.head()
```

	fare	age
0	7.2500	22.0
4	8.0500	35.0
5	8.4583	NaN
6	51.8625	54.0
7	21.0750	2.0

Approach#3: Subsetting Columns and Rows with *[][]*

- Let's imagine we want a subset that contains the *ages* of people who did not survive the `Titanic`. We can actually do this on one step if we use *[][]*:

```
victims = df_titanic[df_titanic['survived']==0]['age']
victims.head()
```

...

	age
0	22.0
4	35.0
5	NaN
6	54.0
7	2.0

Approach#3: Subsetting Columns and Rows with *[][]*

- Let's imagine we want a subset that contains *`fare`* and *`ages`* from `Titanic`. We can actually do this on one step if we use *[][]*:

```
# selecting multiple columns
titanic_victims = titanic[titanic.survived==0][["fare", "age"]]
titanic_victims.head()
```

	fare	age
0	7.2500	22.0
4	8.0500	35.0
5	8.4583	NaN
6	51.8625	54.0
7	21.0750	2.0

Exercise#2:
Practice Time, Your Turn

💬 Group Exercise:

Try the above approaches using a different subset of columns

> Step 1 (Row Selection): make a subset, survived, of rows where survived == 1

> Step 2 (Column Selection): use survived to create a second subset that only contains the *'pclass'* and *'embark_town'* columns.

```
[ ]  # your code
     # ...
```

Multiple Conditions

- What if we want to filter rows by multiple conditions? Make sure each condition is in parentheses and use the old school | and & for operators

```
#women and children on titanic
women_and_children = titanic[(titanic.age < 18) | (titanic.sex == 'female')]
women_and_children.shape[0]
women_and_children.survived.sum()
```

```
256
```

```
# men who survived
men_who_survived = titanic[(titanic.sex == 'male') & (titanic.survived == 1)]
men_who_survived.shape[0]
men_who_survived.age.mean()
```

```
27.276021505376345
```

Some Handy Functions

- `mean(), median(), sum(), unique()`

```
#average age of titanic passengers:
titanic.age.mean()
#titanic['age'].mean()
```

```
29.69911764705882
```

```
#median ticket fare for titanic passengers:
titanic.fare.median()
```

```
14.4542
```

```
#number of survivors
titanic.survived.sum()
total_num_people = titanic.shape[0]
did_not_survive= total_num_people - titanic.survived.sum()
did_not_survive
```

```
549
```

```
#get the unique values of the Deck column
titanic.deck.unique()
```

```
array([nan, 'C', 'E', 'G', 'D', 'A', 'B', 'F'], dtype=object)
```

Some Handy Functions

- `groupby()`

```
titanic.groupby(['survived'])['age'].mean()

survived
0    30.626179
1    28.343690
Name: age, dtype: float64
```

- `Explanation of groupby(): it can be done separately for each group`

```
[19] condition = titanic['survived'] == 0
     survivor_0 = titanic[condition]['age']
     survivor_0.mean()

     30.62617924528302

[20] condition = titanic['survived'] == 1
     survivor_1 = titanic[condition]['age']
     survivor_1.mean()

     28.343689655172415

     titanic.groupby('survived')['age'].mean()

     survived
     0    30.626179
     1    28.343690
     Name: age, dtype: float64
```

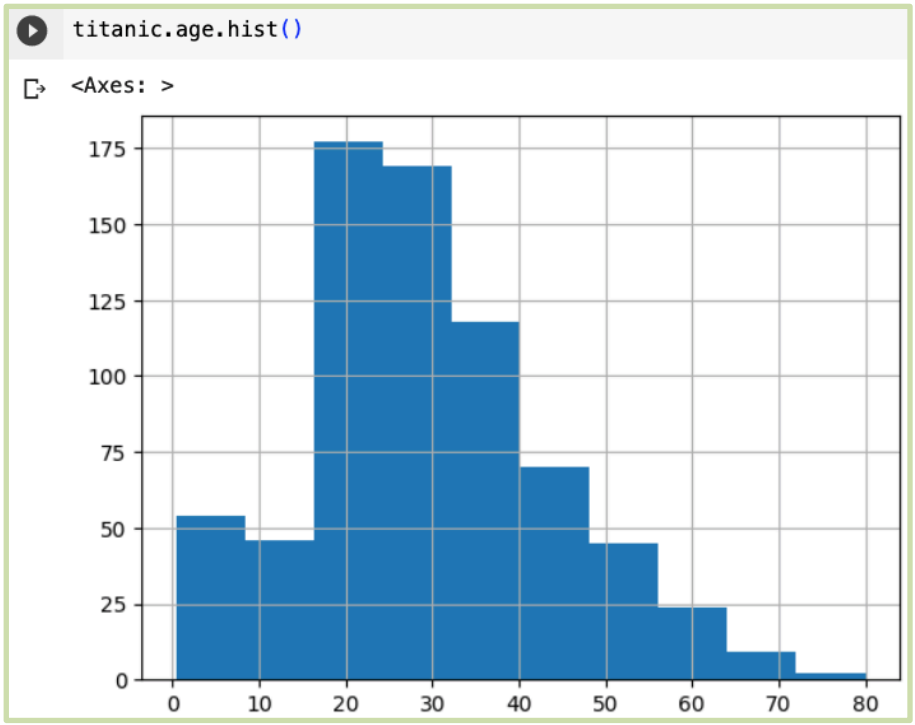
Some Handy Functions

- `describe()`, `hist()`

```
titanic.age.describe()
```

```
count    714.000000
mean      29.699118
std       14.526497
min        0.420000
25%       20.125000
50%       28.000000
75%       38.000000
max       80.000000
Name: age, dtype: float64
```

```
titanic.age.hist()
```



Today's Agenda

- Notebook # 1 Released

Before we get started, let's load in our datasets:

- Make sure you change the path to match your Google Drive.
  - Also, go ahead and download the `vehicles.csv` file from Blackboard and put it in your Google Drive

```
[2]  #run this cell if you're using Colab:
     from google.colab import drive
     drive.mount('/content/drive')
```

```
#import the data:
import pandas as pd

path2 = '/content/drive/MyDrive/cs167_fall24/datasets/irisData.csv'
iris= pd.read_csv(path2)
iris.head()
```

Notebook #1 Help

- A few helpful functions for Notebook #1:
  - `max()` will return the maximum value in a Series
  - `idxmax()` will return the **index** of the maximum value

```
[ ]  #find the deck of the passenger who was the oldest on the titanic
     titanic.age.max()

     80.0

[10] ndx = titanic.age.idxmax() ## returns the name of the 'row' and NOT integer index of the row
     print(ndx)
     titanic.loc[ndx].embark_town

     630
     'Southampton'
```

Notebook #1 Help

- Other functions worth mentioning:
  - `min()` will return the minimum value in a Series
  - `mode()` will return the mode — the most common value of the dataset

```
titanic.age.min()

0.42
```

```
[12] titanic.age.mode()

0    24.0
Name: age, dtype: float64
```

Notebook #1 Help

- Pandas Exercises Solutions (Day04 Solutions)

  - The solution will be posted on Blackboard over the weekend. I want you to try it first and then review the solution for verification.