Midterm 2, Spring 2022: Actor Network Analysis

Version 1.0.1

Change History

- 1.0 - Initial Release
- 1.0.1 - Corrected Typo in ex8 demo cell.

This problem builds on your knowledge of Pandas, base Python data structures, and using new tools. (Some exercises require you to use *very basic* features of the `networkx` package, which is well documented.) It has 9 exercises, numbered 0 to 8. There are **17** available points. However, to earn 100% the threshold is **14** points. (Therefore, once you hit **14** points, you can stop. There is no extra credit for exceeding this threshold.)

Each exercise builds logically on previous exercises, but you may solve them in any order. That is, if you can't solve an exercise, you can still move on and try the next one. Use this to your advantage, as the exercises are **not** necessarily ordered in terms of difficulty. Higher point values generally indicate more difficult exercises.

Code cells starting with the comment `### define demo inputs` load results from prior exercises applied to the entire data set and use those to build demo inputs. These must be run for subsequent demos to work properly, but they do not affect the test cells. The data loaded in these cells may be rather large (at least in terms of human readability). You are free to print or otherwise use Python to explore them, but we did not print them in the starter code.

The point values of individual exercises are as follows:

- Exercise 0: 1 point (This one is a freebie!)
- Exercise 1: 1 point
- Exercise 2: 2 point
- Exercise 3: 1 point
- Exercise 4: 2 point
- Exercise 5: 4 point
- Exercise 6: 1 point
- Exercise 7: 2 point
- Exercise 8: 3 point

Solution (mt1-sp22.html)

Exercise 0 (1 point):

Before we can do any analysis, we have to read the data from the file it is stored in. We have defined `load_data` and are using it to read from the data file.

In [47]:
```
###
### AUTOGRADER TEST - DO NOT REMOVE
###
def load_data(path):
    import pandas as pd
    return pd.read_csv(path, names=['film_id', 'film_name', 'actor', 'year'], ski
```

The cell below will test your solution for Exercise 0. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [48]:
```
### test_cell_ex0
from tester_fw.testers import Tester_ex0
tester = Tester_ex0()
for _ in range(20):
    try:
        tester.run_test(load_data)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040
Passed! Please submit.
```

Exercise 1 (1 Point):

Next we need to explore our data. Complete the function `explore_data` to return a tuple, `t`, with the following elements.

- `t[0]` - `tuple` - the shape of `df`
- `t[1]` - `pd.DataFrame` - the first five rows of `df`
- `t[2]` - `dict` - mapping year (`int`) to the number of films released that year (`int`)

The input `df` is a `pd.DataFrame` with the following columns:

- `'film_id'` - unique integer associated with a film
- `'film_name'` - the name of a film

- `'actor'` - the name of an actor who starred in the film
- `'year'` - the year which the film was released

Each row in `df` indicates an instance of an actor starring in a film, so it is possible that there will be multiple rows with the same `'film_name'` and `'film_id'`.

In [73]:
```
def explore_data(df):
    t = []
    # shape of df
    t.append(df.shape)
    # the first five rows
    t.append(df.head(5))
    # map year to the number of films released that year
    films = df[['film_id', 'year']].drop_duplicates()

    t.append(films['year'].value_counts().to_dict())

    return tuple(t)
```

The demo cell below should display the following output:

```
((15, 4),
       film_id                                film_name                 a
ctor  \
 8277      1599                           Before I Fall       Medalion Ra
himi
 6730      1150      A Million Ways to Die in the West       Seth MacFar
lane
 5770       934  The Mortal Instruments: City of Bones  Jamie Campbell B
ower
 10007     1883                 Avengers: Infinity War           Chris P
ratt
 9831      1855                           Isle of Dogs           Bob Bal
aban

       year
 8277   2017
 6730   2014
 5770   2013
 10007  2018
 9831   2018  ,
 {2011: 2, 2012: 1, 2013: 2, 2014: 1, 2016: 1, 2017: 3, 2018: 4, 2019:
 1})
```

In [74]:
```
### define demo inputs
import pickle
with open('resource/asnlib/publicdata/movie_data.pkl', 'rb') as f:
    movie_data = pickle.load(f)
demo_df_ex1 = movie_data.sample(15, random_state=6040)
```

In [75]:
```
### call demo funtion
explore_data(demo_df_ex1)
```

Out[75]:
```
((15, 4),
       film_id                                 film_name                 actor  \
 8277      1599                            Before I Fall       Medalion Rahimi
 6730      1150      A Million Ways to Die in the West       Seth MacFarlane
 5770       934  The Mortal Instruments: City of Bones  Jamie Campbell Bower
 10007     1883                  Avengers: Infinity War           Chris Pratt
 9831      1855                             Isle of Dogs           Bob Balaban

       year
 8277   2017
 6730   2014
 5770   2013
 10007  2018
 9831   2018  ,
 {2018: 4, 2017: 3, 2013: 2, 2011: 2, 2014: 1, 2019: 1, 2016: 1, 2012: 1})
```

The cell below will test your solution for Exercise 1. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [76]:
```
### test_cell_ex1

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from tester_fw.testers import Tester_ex1
tester = Tester_ex1()
for _ in range(20):
    try:
        tester.run_test(explore_data)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040
Passed! Please submit.
```

In [72]:
```
print(true_output_vars)
print(returned_output_vars)
```

```
{'t': ((200, 4),    film_id                 film_name            actor  year
0     2163  Spider-Man: Far From Home    Cobie Smulders  2019
1      310            Just Go with It  Jennifer Aniston  2011
2     2072              What Men Want      Tracy Morgan  2019
3       96                The Joneses       Amber Heard  2010
4     1600                   Table 19  Stephen Merchant  2017, {2010: 16, 2011:
28, 2012: 28, 2013: 24, 2014: 10, 2015: 12, 2016: 6, 2017: 17, 2018: 27, 2019:
18})}
{'t': ((200, 4),    film_id                 film_name            actor  year
0     2163  Spider-Man: Far From Home    Cobie Smulders  2019
1      310            Just Go with It  Jennifer Aniston  2011
2     2072              What Men Want      Tracy Morgan  2019
3       96                The Joneses       Amber Heard  2010
4     1600                   Table 19  Stephen Merchant  2017, {2011: 32, 2012:
30, 2018: 28, 2013: 25, 2019: 20, 2017: 20, 2010: 17, 2015: 12, 2014: 10, 2016:
6})}
```

Exercise 2 (2 Points):

We will continue our exploration by identifying prolific actors. Complete the function `top_10_actors` to accomplish the following:

- Determine how many films each actor has appeared in.
- Return a DataFrame containing the top 10 actors who have appeared in the most films.
  - Should have columns `'actor'` (string) and `'count'` (int) indicating the actor's name and the number of films they have appeared in.
  - Should be sorted by `'count'`
  - In the event of ties (multiple actors appearing in the same number of films), sort actor names in alphabetical order.
  - Actors should not be excluded based on their name only. More specifically if the 10th most prolific actor has appeared in X films, all actors appearing in at least X films should be included.
    - This may result in more than 10 actors in the output.
  - The index of the result should be sequential numbers, starting with 0.

The input `df` will be as described in exercise 1.

In [96]:
```
def top_10_actors(df):
    actors = df['actor'].value_counts().rename_axis('actor').reset_index(name='co

    actors = actors.sort_values(by = ['count', 'actor'], ascending = [False, True

    top_ten_min = actors['count'].iloc[9]
    final_actors = actors[actors['count'] >= top_ten_min]
    return final_actors
```

The demo cell below should display the following output:

	actor	count
0	Chloë Grace Moretz	8
1	Anna Kendrick	7
2	Jennifer Lawrence	7
3	Kevin Hart	7
4	Kristen Wiig	7
5	Melissa Leo	7
6	Melissa McCarthy	7
7	Ryan Reynolds	7
8	Bill Hader	6
9	Bryan Cranston	6
10	Christina Hendricks	6
11	Dan Stevens	6
12	Danny Glover	6
13	Idris Elba	6
14	James McAvoy	6
15	Maya Rudolph	6
16	Morgan Freeman	6
17	Nicolas Cage	6
18	Rose Byrne	6
19	Sylvester Stallone	6

Notice how all of the actors appearing in 6 or more movies are included.

In [97]:
```
### define demo inputs
import pickle
with open('resource/asnlib/publicdata/movie_data.pkl', 'rb') as f:
    movie_data = pickle.load(f)
demo_df_ex2 = movie_data.sample(3000, random_state=6040)
```

In [98]:
```
### call demo funtion
print(top_10_actors(demo_df_ex2))
```

```
[0        True
7        True
5        True
3        True
4        True
        ...
1648    False
1965    False
1130    False
1367    False
1373    False
Name: count, Length: 1974, dtype: bool]
                  actor  count
0    Chloë Grace Moretz      8
7         Anna Kendrick      7
5     Jennifer Lawrence      7
3            Kevin Hart      7
4          Kristen Wiig      7
2           Melissa Leo      7
6      Melissa McCarthy      7
1         Ryan Reynolds      7
12           Bill Hader      6
14       Bryan Cranston      6
10  Christina Hendricks      6
17          Dan Stevens      6
18         Danny Glover      6
13           Idris Elba      6
19         James McAvoy      6
15         Maya Rudolph      6
9        Morgan Freeman      6
8          Nicolas Cage      6
16           Rose Byrne      6
11   Sylvester Stallone      6
```

The cell below will test your solution for Exercise 2. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [95]:
```
### test_cell_ex2

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from tester_fw.testers import Tester_ex2
tester = Tester_ex2()
for _ in range(50):
    try:
        tester.run_test(top_10_actors)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040
Passed! Please submit.
```

Exercise 3 (1 Point):

We will continue our exploration with a look at which years an actor has appeared in movies. Complete the function `actor_years` to determine which years the given `actor` has appeared in movies based off of the data in `df`. Your output should meet the following requirements:

- Output is a `dict` mapping the actor's name to a `list` of integers (`int`) containing the years in which this actor appeared in films.
- There should not be any duplicate years. If an actor has appeared in one or more films in a year, that year should be included **once** in the list.
- The list of years should be sorted in **ascending** order.

The input `df` is a `pd.DataFrame` of the same form denoted in exercise 1.

In [115]:
```
def actor_years(df, actor):
    d = {}
    actors = df[['actor', 'year']]
    actor_year = actors.loc[actors['actor'] == actor]
    actor_year = actor_year.drop_duplicates()
    years = actor_year['year'].tolist()
    years.sort()
    d = {actor:years}
    return d
```

The demo cell below should display the following output:

```
{'James Franco': [2012, 2013]}
```

In [116]:
```
### define demo inputs
import pickle
with open('resource/asnlib/publicdata/movie_data.pkl', 'rb') as f:
    movie_data = pickle.load(f)
demo_df_ex3 = movie_data.sample(3000, random_state=6040)
```

In [117]:
```
### call demo funtion
actor_years(demo_df_ex3, 'James Franco')
```

Out[117]:
```
{'James Franco': [2012, 2013]}
```

The cell below will test your solution for Exercise 3. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [118]:
```
### test_cell_ex3

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from tester_fw.testers import Tester_ex3
tester = Tester_ex3()
for _ in range(20):
    try:
        tester.run_test(actor_years)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040
Passed! Please submit.
```

Exercise 4 (2 Points):

For our last exercise in exploration, we want to see some summary statistics on how many actors

participated in a movie. Complete the funciton `movie_size_by_year` to accomplish the following:

- Determine the size of each film in terms of the number of actors in that film. In other words, if there are X actors in film Y then the size of film Y is X.
- For each year, determine the minimum, maximum, and mean sizes of films released that year. All values in the "inner" dictionaries should be of type `int`.
- Return the results as a nested dictionary
  - { `year` : {'min': minimum size, 'max': maximum size, 'mean': mean size (rounded to the nearest integer)}}

In [135]:
```
### Define movie_size_by_year
def movie_size_by_year(df):
    d = {}
    # in each year
    years = df['year'].unique()
    years.sort()

    for i in years:
        int_i = int(i)
        d[int_i] = {}
        year = df.loc[df['year'] == i]
        counts = year['film_id'].value_counts()
        d[int_i]['min'] = min(counts)
        d[int_i]['max'] = max(counts)
        d[int_i]['mean'] = int(round(sum(counts) / len(counts)))

    return d
```

The demo cell below should display the following output:

```
{2010: {'min': 1, 'max': 8, 'mean': 2},
 2011: {'min': 1, 'max': 7, 'mean': 2},
 2012: {'min': 1, 'max': 8, 'mean': 2},
 2013: {'min': 1, 'max': 13, 'mean': 2},
 2014: {'min': 1, 'max': 4, 'mean': 1},
 2015: {'min': 1, 'max': 4, 'mean': 1},
 2016: {'min': 1, 'max': 2, 'mean': 1},
 2017: {'min': 1, 'max': 6, 'mean': 2},
 2018: {'min': 1, 'max': 6, 'mean': 2},
 2019: {'min': 1, 'max': 6, 'mean': 2}}
```

In [136]:
```
### define demo inputs
import pickle
with open('resource/asnlib/publicdata/movie_data.pkl', 'rb') as f:
    movie_data = pickle.load(f)
demo_df_ex4 = movie_data.sample(3000, random_state=6040)
```

In [137]:
```
movie_size_by_year(demo_df_ex4)
```

Out[137]:
```
{2010: {'min': 1, 'max': 8, 'mean': 2},
 2011: {'min': 1, 'max': 7, 'mean': 2},
 2012: {'min': 1, 'max': 8, 'mean': 2},
 2013: {'min': 1, 'max': 13, 'mean': 2},
 2014: {'min': 1, 'max': 4, 'mean': 1},
 2015: {'min': 1, 'max': 4, 'mean': 1},
 2016: {'min': 1, 'max': 2, 'mean': 1},
 2017: {'min': 1, 'max': 6, 'mean': 2},
 2018: {'min': 1, 'max': 6, 'mean': 2},
 2019: {'min': 1, 'max': 6, 'mean': 2}}
```

The cell below will test your solution for Exercise 4. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [138]:
```
### test_cell_ex4

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from tester_fw.testers import Tester_ex4
tester = Tester_ex4()
for _ in range(20):
    try:
        tester.run_test(movie_size_by_year)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040
Passed! Please submit.
```

Exercise 5 (4 Point):

We want to ultimately do some network analytics using this data. Our first task to that end is to

We want to ultimately do some network analysis using this data. Our first task is that end is to define our data in terms of a network. Here's the particulars of what we want in the network.

- Un-weighted, un-directed graph structure with no self-edges.
- Actors are nodes and there is an edge between two actors if they have starred in the same film.

Complete the function `make_network_dict` to process the data from `df` into this graph structure. The graph should be returned in a nested "dictionary of sets" structure.

- The keys are actor names, and the values are a set of the key actor's co-stars.
- To avoid storing duplicate data, all co-actors should be alphabetically after the key actor. If following this rule results in an key actor having an empty set of costars, that actor should not be included as a key actor. This means that actors who only appear in films without costars would not be included.
  - For example `{'Alice':{'Bob', 'Alice', 'Charlie'}, 'Bob':{'Alice', 'Bob', 'Charlie'}, 'Charlie: {'Alice', 'Bob', 'Charlie'}}` indicates that there is an edge between Alice and Bob, an edge between Bob and Charlie, and an edge between Alice and Charlie. Instead of storing all the redundant information, we would store just `{'Alice': {'Bob', 'Charlie'}, 'Bob': {'Charlie'}}`.
- **Hint:** Think about how you could use `merge` to determine all pairs of costars. Once you have that, you can worry about taking out the redundant information.

In [29]:
```
def make_network_dict(df):
    pass
```

The demo cell below should display the following output:

```
{'Kian Lawley': {'Medalion Rahimi'},
 'Maria Dizzia': {'Wendell Pierce'},
 'Chosen Jacobs': {'Sophia Lillis'},
 'David Ogden Stiers': {'Jesse Corti'},
 'Jason Clarke': {'Kate Mara'},
 'Reese Witherspoon': {'Sarah Paulson'},
 'Olivia Munn': {'Zach Woods'},
 'Faye Dunaway': {'Lucien Laviscount'},
 'Alec Baldwin': {'Rebecca Ferguson'},
 'Pierce Brosnan': {'Steve Coogan'},
 'Dakota Johnson': {'Rhys Ifans'},
 'Bokeem Woodbine': {'Flea'},
 'Nicolas Cage': {'Robert Sheehan'},
 'Bruce Dern': {'Kerry Washington'},
 'Richard Jenkins': {'Sam Shepard'},
 'Jessica Madsen': {'Vanessa Grasse'},
 'Jason White': {'Kristen Wiig'},
 'Robert Davi': {'Stephen Dorff'},
 'Maggie Gyllenhaal': {'Marianne Jean-Baptiste'},
 'Katherine Langford': {'Keiynan Lonsdale'},
 "Denis O'Hare": {'Judi Dench'},
 'Katherine Heigl': {'Michelle Pfeiffer', 'Simon Kassianides'},
 'Craig Robinson': {'Emma Watson'},
 'Colton Dunn': {'Nichole Bloom'},
 'Daniel Sunjata': {'Jennifer Carpenter'},
 'Aly Michalka': {'Cheri Oteri'},
 'John Lithgow': {'Mark Duplass'},
 'Ewan McGregor': {'Julianne Nicholson'},
 'Chris Pine': {'Kathryn Hahn'},
 'David Warner': {'Jonathan Hyde'}}
```

In [30]:
```
### define demo inputs
import pickle
with open('resource/asnlib/publicdata/movie_data.pkl', 'rb') as f:
    movie_data = pickle.load(f)
demo_df_ex5 = movie_data.sample(300, random_state=6040)
```

In [31]:
```
### call demo funtion
make_network_dict(demo_df_ex5)
```

In [32]:
```
import pandas as pd
print(pd.__version__)
```

```
1.4.0
```

The cell below will test your solution for Exercise 5. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.

- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [33]:
```
### test_cell_ex5

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from tester_fw.testers import Tester_ex5
tester = Tester_ex5()
for _ in range(20):
    try:
        tester.run_test(make_network_dict)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-33-b71dcd55679b> in <module>
      9 for _ in range(20):
     10     try:
---> 11         tester.run_test(make_network_dict)
     12         (input_vars, original_input_vars, returned_output_vars, true_
output_vars) = tester.get_test_vars()
     13     except:

~/tester_fw/tester_6040.py in run_test(self, func)
     39         if self.prevent_mod:     #  - can disable by setting `prevent
_mod`to `False` in constructor
     40             self.check_modified()    # Check to verify inputs were no
t modified
---> 41         self.check_type()        # Check to verify correct output typ
es
     42         self.check_matches()     # Check to verify correct output
     43

~/tester_fw/testers.py in check_type(self)
    317                 assert isinstance(v, set), f'network_dict values shou
ld be `set`, but {type(v)} returned.'
    318         ### end check_type_ex5
--> 319         check_type_helper(self.returned_output_vars)
    320
    321

~/tester_fw/testers.py in check_type_helper(outputs)
    312         def check_type_helper(outputs):
    313             o = outputs['network_dict']
--> 314             assert isinstance(o, dict), f'network_dict should be `dic
t`, but {type(o)} returned.'
    315             for k, v in o.items():
```

```
    316                 assert isinstance(k, str), f'network_dict keys should
be `str`, but {type(k)} returned.'

AssertionError: network_dict should be `dict`, but <class 'NoneType'> returne
d.
```

Exercise 6 (1 Points):

Now that we have our dictionary which maps actor names to a `set` of that actor's costars, we are going to use the `networkx` package to perform some graph analysis. The `networkx` framework is based on the `Graph` object - a `Graph` holds data about the graph structure, which is made of `nodes` and `edges` among other attributes. Your task for this exercise will be to add edges to a `networkx.Graph` object based on a `dict` of `sets`.

Complete the function `to_nx(dos)`. Your solution should iterate through the parameter `dos`, a `dict` which maps actors to a `set` of their costars. For each costar pair implied by the input, add an edge to the `Graph` object, `g`. We have provided some "wrapper" code to take care of constructing a `Graph` object, `g`, and returning it. All you have to do is add edges to it.

Note: Check the `networkx` documentation to find how to add edges to a graph. Part of what this exercise is evaluating is your ability to find, read, and understand information on new packages well enough to get started performing its basic tasks. The information is easy to find and straight-forward in this case.

In [34]:
```
import networkx as nx
def to_nx(dos):
    g = nx.Graph()
    ###
    ###
    return g
```

The demo cell below should display the following output:

```
{('Aaron Eckhart', 'Bill Nighy'),
 ('Aaron Eckhart', 'Cory Hardrict'),
 ('Aaron Eckhart', 'Nicole Kidman'),
 ('Aaron Eckhart', 'Ramón Rodríguez'),
 ('Akie Kotabe', 'Salma Hayek'),
 ('Akie Kotabe', 'Togo Igawa'),
 ('Akiva Schaffer', 'Cheri Oteri'),
 ('Akiva Schaffer', 'Jon Lovitz'),
 ('Akiva Schaffer', 'Nick Swardson'),
 ('Akiva Schaffer', "Shaquille O'Neal"),
 ('Alan Tudyk', 'Gal Gadot'),
 ('Alan Tudyk', 'Jennifer Lopez'),
 ('Alan Tudyk', 'John Leguizamo'),
 ('Alan Tudyk', 'Nicki Minaj'),
 ('Albert Tsai', 'Chloe Bennet'),
 ('Albert Tsai', 'Eddie Izzard'),
 ('Albert Tsai', 'Sarah Paulson'),
 ('Albert Tsai', 'Tenzing Norgay Trainor'),
 ('Chris Marquette', 'Alice Braga'),
 ('Chris Marquette', 'Ciarán Hinds'),
 ('Chris Marquette', 'Michael Sheen'),
 ('Chris Marquette', 'Rutger Hauer'),
 ('Chris Marquette', 'Stana Katic'),
 ('David Cross', 'Alison Brie'),
 ('David Cross', 'Gary Oldman'),
 ('David Cross', 'Jason Lee'),
 ('David Cross', 'Jesse Plemons'),
 ('David Cross', 'Michelle Yeoh'),
 ('Jeffrey Johnson', 'Bailee Madison'),
 ('Jeffrey Johnson', 'Ralph Waite'),
 ('Jeffrey Johnson', 'Robyn Lively'),
 ('Jeffrey Johnson', 'Tanner Maguire'),
 ('Jennifer Sipes', 'Christy Carlson Romano'),
 ('Jennifer Sipes', 'Nick Stahl'),
 ('Jennifer Sipes', 'Stephanie Honoré'),
 ('Jesse Bernstein', 'Johnny Sneed'),
 ('Megan Mullally', 'Aaron Paul'),
 ('Megan Mullally', 'Natalie Dreyfuss'),
 ('Megan Mullally', 'Octavia Spencer'),
 ('Megan Mullally', 'Richmond Arquette'),
 ('Mia Kirshner', 'Allie MacDonald'),
 ('Payman Maadi', 'Adria Arjona'),
 ('Payman Maadi', 'Ben Hardy'),
 ('Payman Maadi', 'Dave Franco'),
 ('Sophie Lowe', "James D'Arcy"),
 ('Sophie Lowe', 'Rhys Wakefield'),
 ('Zoe Saldana', 'Andrea Libman'),
 ('Zoe Saldana', 'Casey Affleck'),
 ('Zoe Saldana', 'Idris Elba'),
```

```
('Zoe Saldana', 'Method Man'),
('Zoe Saldana', 'Sylvester Stallone')}
```

In [35]:
```
### define demo inputs
import pickle
import numpy as np
rng = np.random.default_rng(6040)
with open('resource/asnlib/publicdata/network_dict.pkl', 'rb') as f:
    network_dict = pickle.load(f)
demo_dos_ex6 = {k: {v for v in rng.choice(network_dict[k], 5)} for k in rng.choic
```

In [36]:
```
### call demo funtion
set(to_nx(demo_dos_ex6).edges)
```

Out[36]:
```
set()
```

The cell below will test your solution for Exercise 6. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [37]:
```
### test_cell_ex6

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from tester_fw.testers import Tester_ex6
tester = Tester_ex6()
for _ in range(20):
    try:
        tester.run_test(to_nx)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-37-f116f5dd2881> in <module>
      9 for _ in range(20):
     10     try:
---> 11         tester.run_test(to_nx)
     12         (input_vars, original_input_vars, returned_output_vars, true_ou
tput_vars) = tester.get_test_vars()
     13     except:

~/tester_fw/tester_6040.py in run_test(self, func)
     40             self.check_modified()    # Check to verify inputs were not
 modified
     41         self.check_type()        # Check to verify correct output types
---> 42         self.check_matches()     # Check to verify correct output
     43
     44

~/tester_fw/testers.py in check_matches(self)
    382             assert set(rg.nodes) == set(tg.nodes), 'Your solution did n
ot match ours - the nodes were incorrect. You can compare the variables `true_o
utput_vars` (what we got) and `returned_output_vars` (what you got) for debuggi
ng.'
    383         ### end check_matches_ex6
--> 384         check_matches_helper(self.returned_output_vars, self.true_outpu
t_vars)
    385
    386 class Tester_ex7(ExerciseTester):

~/tester_fw/testers.py in check_matches_helper(returned_outputs, true_outputs)
    379             tg = true_outputs['movie_network']
    380             tg_edges = {(x,y) for x, y in tg.edges} | {(x,y) for y, x i
n tg.edges}
--> 381             assert rg_edges == tg_edges, 'Your solution did not match o
```

```
urs - the edges were incorrect. You can compare the variables `true_output_vars
` (what we got) and `returned_output_vars` (what you got) for debugging.'
    382             assert set(rg.nodes) == set(tg.nodes), 'Your solution did n
ot match ours - the nodes were incorrect. You can compare the variables `true_o
utput_vars` (what we got) and `returned_output_vars` (what you got) for debuggi
ng.'
    383         ### end check_matches_ex6

AssertionError: Your solution did not match ours - the edges were incorrect. Yo
u can compare the variables `true_output_vars` (what we got) and `returned_outp
ut_vars` (what you got) for debugging.
```

Exercise 7 (2 Points):

One thing that the `networkx` package makes relatively easy is calculating the *degree* of each of the nodes in our graph. Here degree would be interpreted as the number of unique costars each actor has. If you have a graph `g` then `g.degree()` will return an object that maps each node to its degree (see note).

Complete the function `high_degree_actors(g, n)` : Given the inputs described below, determine the degree of each actor in the graph, `g` . Return a `pd.DataFrame` with 2 columns (`'actor'` and `'degree'`), indicating an actor's name and degree. The output should have records for only the actors with the `n` highest degrees. In the case of ties (two or more actors having the same degree), all of the actors with the lowest included degree should be included. (for example if there's a 3-way tie for 10th place and `n` =10 then all 3 of the actors involved in the tie should be included in the output). If `n` is `None` , all of the actors should be included.

Sort your results by degree (descending order) and break ties (multiple actors w/ same degree) by sorting them in alphabetical order based on the actor's name.

The index of the result should be sequential numbers, starting with 0.

- input `g` - a `networkx` graph object having actor names as nodes and edges indicating whether the actors were costars based on our data.
- input `n` - `int` indicating how many actors to return. This argument is optional for the user and has a default value of `None` .

Note: One complication is that `g.degree()` isn't a `dict` . Keep in mind that it *can* be *cast* to a `dict` .

In [39]:
```
def high_degree_actors(g, n=None):
    pass
```

The demo cell below should display the following output:

	actor	degree
0	Elizabeth Banks	9
1	Emma Stone	9
2	Bradley Cooper	8
3	Anthony Mackie	7
4	Michael Peña	7
5	Maya Rudolph	6
6	Richard Jenkins	6
7	Stanley Tucci	6
8	Steve Carell	6

Notice how 9 actors are included even though `n = 7`.

In [40]:
```
### define demo inputs
import pickle
with open('resource/asnlib/publicdata/movie_network.pkl', 'rb') as f:
    movie_network = pickle.load(f)
demo_g_ex7 = movie_network.subgraph({a for a, _ in sorted(movie_network.degree, k
demo_n_ex7 = 7
```

In [41]:
```
### call demo funtion
print(high_degree_actors(demo_g_ex7, demo_n_ex7))
```

```
None
```

The cell below will test your solution for Exercise 7. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [42]:
```
### test_cell_ex7

###
### AUTOGRADER TEST - DO NOT REMOVE
###


from tester_fw.testers import Tester_ex7
tester = Tester_ex7()
for _ in range(20):
    try:
        tester.run_test(high_degree_actors)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-42-695ace3d3fe6> in <module>
     10 for _ in range(20):
     11     try:
---> 12         tester.run_test(high_degree_actors)
     13         (input_vars, original_input_vars, returned_output_vars, true_ou
tput_vars) = tester.get_test_vars()
     14     except:

~/tester_fw/tester_6040.py in run_test(self, func)
     39         if self.prevent_mod:     #  - can disable by setting `prevent_m
od`to `False` in constructor
     40             self.check_modified()    # Check to verify inputs were not
 modified
---> 41         self.check_type()        # Check to verify correct output types
     42         self.check_matches()     # Check to verify correct output
     43

~/tester_fw/testers.py in check_type(self)
    424             assert isinstance(o, pd.DataFrame), f'`top_actors` should b
e a DataFrame, but {type(o)} was returned.'
    425         ### end check_type_ex7
--> 426         check_type_helper(self.returned_output_vars)
    427
    428

~/tester_fw/testers.py in check_type_helper(outputs)
    422             import pandas as pd
    423             o = outputs['top_actors']
--> 424             assert isinstance(o, pd.DataFrame), f'`top_actors` should b
e a DataFrame, but {type(o)} was returned.'
```

```
425         ### end check_type_ex7
426         check_type_helper(self.returned_output_vars)
```

```
AssertionError: `top_actors` should be a DataFrame, but <class 'NoneType'> was
 returned.
```

Exercise 8 (3 Points):

Another place where `networkx` shines is in its built-in graph algorithms, like community detection. We have calculated the communities using `networkx` (check the docs for info on how to do this yourself) and have the `communities` variable set to a `list` of `sets` (you can iterate over `communities` like a list, and each set is the names of all the actors in one community).

Given

- `communities` - a `list` containing `sets` indicating membership to a particular community. The communities are a partition of the actors, so you can safely assume that an actor will only appear in one of these sets.
- `degrees` - A `pd.DataFrame` with columns `'actor'` and `'degree'` indicating the degree of each actor in the DataFrame
- `actor` - an actor's name

Complete the function `notable_actors_in_comm`. Your solution should accomplish the following:

1. Determine which community the given actor belongs to.
2. Return a `pd.DataFrame` with two columns ('actor' and 'degree') including the top 10 actors in the same community as the given actor.
    - We must handle cases where there are fewer than 10 actors in a community. In such cases, all actors in the community should be included in the result without raising an error.
3. Output should be sorted in descending order of degree with ties (two or more actors with same degree) broken by sorting alphabetically by actor name.
4. Include only actors with degree >= the 10th highest degree. This may mean that there are more than 10 actors in the result.
5. The index of the result should be sequential numbers, starting with 0.

In [43]:
```
def notable_actors_in_comm(communities, degrees, actor):
    assert actor in {a for c in communities for a in c}, 'The given actor was not
    pass
```

The demo cell below should display the following output:

	actor	degree
0	Bryan Cranston	135
1	Anthony Mackie	116
2	Johnny Depp	115
3	Idris Elba	112
4	Joel Edgerton	109
5	James Franco	107
6	Jessica Chastain	107
7	Jeremy Renner	105
8	Chris Hemsworth	104
9	Zoe Saldana	104

In [44]:
```
### define demo inputs
import pickle
path = 'resource/asnlib/publicdata/communities.pkl'
with open(path, 'rb') as f:
    communities = pickle.load(f)
path = 'resource/asnlib/publicdata/degrees.pkl'
with open(path, 'rb') as f:
    degrees = pickle.load(f)
demo_actor_ex8 = 'Christian Bale'
```

In [45]:
```
### call demo funtion
print(notable_actors_in_comm(communities, degrees, demo_actor_ex8))
```

```
None
```

The cell below will test your solution for Exercise 8. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [46]:
```
### test_cell_ex8

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from tester_fw.testers import Tester_ex8
tester = Tester_ex8()
for _ in range(20):
    try:
        tester.run_test(notable_actors_in_comm)
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars)
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
```

```
initializing tester_fw.tester_6040

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-46-6edac4e1e6d5> in <module>
      9 for _ in range(20):
     10     try:
---> 11         tester.run_test(notable_actors_in_comm)
     12         (input_vars, original_input_vars, returned_output_vars, true_ou
tput_vars) = tester.get_test_vars()
     13     except:

~/tester_fw/tester_6040.py in run_test(self, func)
     39         if self.prevent_mod:     #  - can disable by setting `prevent_m
od`to `False` in constructor
     40             self.check_modified()    # Check to verify inputs were not
 modified
---> 41         self.check_type()        # Check to verify correct output types
     42         self.check_matches()     # Check to verify correct output
     43

~/tester_fw/testers.py in check_type(self)
    475             assert isinstance(o, pd.DataFrame), f'output is required to
be a DataFrame, but {type(o)} was returned.'
    476         ### end check_type_ex8
--> 477         check_type_helper(self.returned_output_vars)
    478
    479

~/tester_fw/testers.py in check_type_helper(outputs)
    473             import pandas as pd
    474             o = outputs['output']
--> 475             assert isinstance(o, pd.DataFrame), f'output is required to
be a DataFrame, but {type(o)} was returned.'
    476         ### end check_type_ex8
```

```
    477         check_type_helper(self.returned_output_vars)

AssertionError: output is required to be a DataFrame, but <class 'NoneType'> wa
s returned.
```

Fin. This is the end of the exam. If you haven't already, submit your work.