# DASC5301 Data Science, Fall 2021, Chengkai Li, Unversity of Texas at Arlington

# Programming Assignment 2

# Due: Friday, Octoer 29, 2021, 11:59pm

## Requirements

1. When you work on this assignment, you should make a copy of this notebook in Google Colab. This can be done using the option `File > Save a copy in Drive` in Google Colab.
2. You should fill in your answer for each task inside the code block right under the task.
3. You should only insert your code into the designated code blocks, as mentioned above. Other than that, you shouldn't change anything else in the notebook.
4. The correct output for each task is also included right below the corresponding code block.
5. Each task can be solved using one line of code. (An exception is Tasks 12, for which it is much easier to use three lines.) Nevertheless, for any task, if you have to use up to three lines, that's fine. But you are not allowed to use more than three lines of code. Note that we have two designated code blocks in both Task 3 and Task 4.
6. You may not use any other imports to solve the tasks. In other words, you shouldn't use `import` in any desinated code blocks for the tasks.
7. You should not use any loops, if statements, or list/dictionary comprehensions. You can solve all the tasks by only using features and functions from pandas.
8. Even if you can only partially solve a task, you should include your code in the code block, which allows us to consider partial credit.
9. However, your code should not raise errors. Any code raising errors will not get partial credit.
10. We tried to minimize the interdependence of the tasks. In most cases, even if you don't solve a task, it won't affect you working on the tasks afterwards, although the output of a task may not be fully correct if you didn't correctly solve the preceding tasks. If you get stuck on a task, you can move on to work on the rest and try to come back to that task later.
11. To submit your assignment, download your Colab into a .ipynb file. This can be done using the option `Download > Download .ipynb` in Google Colab.
12. Submit the downloaded .ipynb file into the Programming Assignment 2 entry in Canvas.

## Dataset

In this assignment, we will do data munging and analysis on a dataset about board games. The dataset is from Kaggle, at https://www.kaggle.com/andrewmvd/board-games. We have already downloaded the CSV file and provided the file in the assignment's entry in canvas. You will need to upload the CSV file to your Google Colab working directory. After that, let's load the CSV file into a pandas DataFrame.

```
import pandas as pd

games = pd.read_csv('bgg_dataset.csv', delimiter=';', decimal=",")
```

We set `display.max_rows` to `None` for now so that we can see more values in code output.

```
pd.set_option('display.max_rows', None)
```

Let's gain some basic understanding of the dataset by using `info()`, `head()`, and `describe()`.

```
games.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20343 entries, 0 to 20342
Data columns (total 14 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   ID                  20327 non-null  float64
 1   Name                20343 non-null  object 
 2   Year Published      20342 non-null  float64
 3   Min Players         20343 non-null  int64  
 4   Max Players         20343 non-null  int64  
 5   Play Time           20343 non-null  int64  
 6   Min Age             20343 non-null  int64  
 7   Users Rated         20343 non-null  int64  
 8   Rating Average      20343 non-null  float64
 9   BGG Rank            20343 non-null  int64  
 10  Complexity Average  20343 non-null  float64
 11  Owned Users         20320 non-null  float64
 12  Mechanics           18745 non-null  object 
 13  Domains             10184 non-null  object 
dtypes: float64(5), int64(6), object(3)
memory usage: 2.2+ MB
```

```
games.head()
```

| | ID | Name | Year Published | Min Players | Max Players | Play Time | Min Age | Users Rated | Rating Average | BGG C Rank |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 174430.0 | Gloomhaven | 2017.0 | 1 | 4 | 120 | 14 | 42055 | 8.79 | 1 |
| **1** | 161936.0 | Pandemic Legacy: Season 1 | 2015.0 | 2 | 4 | 60 | 13 | 41643 | 8.61 | 2 |

```
games.describe()
```

| | ID | Year Published | Min Players | Max Players | Play Time | Min Ag |
|---|---|---|---|---|---|---|
| **count** | 20327.000000 | 20342.000000 | 20343.000000 | 20343.000000 | 20343.000000 | 20343.00000 |
| **mean** | 108216.245142 | 1984.249877 | 2.019712 | 5.672221 | 91.294548 | 9.60148 |
| **std** | 98682.097298 | 214.003181 | 0.690366 | 15.231376 | 545.447203 | 3.64545 |
| **min** | 1.000000 | -3500.000000 | 0.000000 | 0.000000 | 0.000000 | 0.00000 |
| **25%** | 11029.000000 | 2001.000000 | 2.000000 | 4.000000 | 30.000000 | 8.00000 |
| **50%** | 88931.000000 | 2011.000000 | 2.000000 | 4.000000 | 45.000000 | 10.00000 |
| **75%** | 192939.500000 | 2016.000000 | 2.000000 | 6.000000 | 90.000000 | 12.00000 |
| **max** | 331787.000000 | 2022.000000 | 10.000000 | 999.000000 | 60000.000000 | 25.00000 |

## Need for data cleaning and preprocessing

The results of these several functions indicate a few needs for cleaning and preprocessing the data:

1) The columns `ID`, `Year Published` and `Owned Users` should be integers, but they are floating point numbers.

2) There are null values in various columns.

3) It seems there could be wrong values. For instance, the minimal value in `Max Players` is 0. What kind of game is that if it allows at most zero player?

4) The values in columns `Mechanics` and `Domains` are comma-separated lists. We need to parse these values and get the individual items from the lists.

Let's find out which columns have null values. This could be derived from the `Non-Null Count` in the output of `games.info()`. But there are much simpler ways.

## Task 1: For each column, find the number of rows with null value in that column. (5 points)

If your code is correct, its output should tell you that five columns have null values --- column `ID` has missing value in 16 rows, `Year Published` has 1, `Owned Users` has 23, `Mechanics` has 1598, and `Domains` has 10159! Other columns have no null values.

```
# Code for Task 1
games.isnull().sum()
```

```
ID                     16
Name                    0
Year Published          1
Min Players             0
Max Players             0
Play Time               0
Min Age                 0
Users Rated             0
Rating Average          0
BGG Rank                0
Complexity Average      0
Owned Users            23
Mechanics            1598
Domains             10159
dtype: int64
```

Since column `ID` has null values, it couldn't be used for uniquely identifying games. Hence, let's take it out.

## Task 2: Remove the `ID` column from the `DataFrame games`. (5 points)

```
# Code for Task 2
games.drop("ID",axis=1,inplace=True)
```

## Task 3: Replace null values in column `Year Published` by 5000. Replace null values in column `Owned Users` by -1. Note that this task has two desiganated code blocks. (5 points)

The column `Year Published` has negative values, as `games.descrie()` shows. It actually has 0 in its values too. Hence, we are using a year in the future (5000) to indicate the dataset doesn't provide

the value for a game. The column `Owned Users` has 0s too. We thus use -1 to indicate missing values.

```
# Code for Task 3 : code block for replaceing null values in column Year Published by 5000.
games["Year Published"].fillna("5000",inplace = True)
```

```
# Code for Task 3: code block for replacing null values in column Owned Users by -1.
games["Owned Users"].fillna("-1",inplace = True)
```

## Task 4: Convert the data type of column `Year Pubblished` to integer. Convert the data type of column `Owned Users` to integer too. Note that this task has two desiganated code blocks. (5 points)

```
# Code for Task 4: code block for converting the data type of column ``Year Pubblished`` to i
games['Year Published']=games['Year Published'].astype('int16')
```

```
# Code for Task 4: code block for converting the data type of column ``Owned Users`` to integ
games['Owned Users']=games['Owned Users'].astype('int32')
```

After you finish Tasks 1 to 4, run `games.info()`, `games.head()`, and `games.describe()` again, to verify you have achieved the goals. In fact, you could do this from time to time, in various places, to make sure you haven't messed up the data.

```
games.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20343 entries, 0 to 20342
Data columns (total 13 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   Name                20343 non-null  object 
 1   Year Published      20343 non-null  int16  
 2   Min Players         20343 non-null  int64  
 3   Max Players         20343 non-null  int64  
 4   Play Time           20343 non-null  int64  
 5   Min Age             20343 non-null  int64  
 6   Users Rated         20343 non-null  int64  
 7   Rating Average      20343 non-null  float64
 8   BGG Rank            20343 non-null  int64  
 9   Complexity Average  20343 non-null  float64
 10  Owned Users         20343 non-null  int32  
 11  Mechanics           18745 non-null  object 
```

```
 12  Domains             10184 non-null  object
dtypes: float64(2), int16(1), int32(1), int64(6), object(3)
memory usage: 1.8+ MB
```

```
games.head()
```

| | Name | Year Published | Min Players | Max Players | Play Time | Min Age | Users Rated | Rating Average | BGG Rank | Complexity Average |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | Gloomhaven | 2017 | 1 | 4 | 120 | 14 | 42055 | 8.79 | 1 | 3.86 |
| **1** | Pandemic Legacy: Season 1 | 2015 | 2 | 4 | 60 | 13 | 41643 | 8.61 | 2 | 2.84 |
| | Brass: | | | | | | | | | |

```
games.describe()
```

| | Year Published | Min Players | Max Players | Play Time | Min Age | Users Rate |
|---|---|---|---|---|---|---|
| **count** | 20343.000000 | 20343.000000 | 20343.000000 | 20343.000000 | 20343.000000 | 20343.00000 |
| **mean** | 1984.398122 | 2.019712 | 5.672221 | 91.294548 | 9.601485 | 840.97139 |
| **std** | 215.039951 | 0.690366 | 15.231376 | 545.447203 | 3.645458 | 3511.56222 |
| **min** | -3500.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 30.00000 |
| **25%** | 2001.000000 | 2.000000 | 4.000000 | 30.000000 | 8.000000 | 55.00000 |
| **50%** | 2011.000000 | 2.000000 | 4.000000 | 45.000000 | 10.000000 | 120.00000 |
| **75%** | 2016.000000 | 2.000000 | 6.000000 | 90.000000 | 12.000000 | 385.00000 |
| **max** | 5000.000000 | 10.000000 | 999.000000 | 60000.000000 | 25.000000 | 102214.00000 |

.Earlier we noticed the existence of value 0 in certain columns which shouldn't have such values. Let's find out how prevalent the prolem is.

## Task 5: For each column, show how many rows have 0 as the value. (5 points)

```
# Code for Task 5
```

```
(games == 0).sum(axis=0)
```

```
Name                    0
Year Published        185
Min Players            46
Max Players           161
Play Time             556
Min Age              1251
Users Rated             0
Rating Average          0
BGG Rank                0
Complexity Average    426
Owned Users             1
Mechanics               0
Domains                 0
dtype: int64
```

If you get the correct code, you will see from the output that 46 rows have value 0 in column `Min Players`, 161 in column `Max Players`, 185 in `Year Published`, and so on. We need to keep this mind when we analyze the data so that we don't draw inaccurate conclusions.

Particularly, let's examine `Year Published`. We discovered earlier that it also has negative values. We can take a closer look now.

## Task 6: Get the number of games published in each year. Sort the years by frequency, in descending order. (10 points)

If you get the code correct, you will find that Year 2017 has 1274 games published, which is the most among all years.

```
# Code for Task 6
games['Year Published'].value_counts()
```

```
2017    1274
2016    1257
2018    1254
2019    1134
2015    1131
2014     987
2013     850
2012     815
2011     735
2010     692
2020     684
2009     631
2008     580
2005     544
2007     522
2006     516
```

```
2004    490
2003    408
2002    335
2001    298
2000    295
1999    268
1998    246
1997    221
1996    216
1995    212
1992    201
0       185
1993    183
1994    180
1991    175
1986    149
1981    149
1990    146
2021    144
1987    136
1989    132
1983    130
1985    129
1988    128
1979    126
1980    125
1982    124
1975    113
1977    102
1978     97
1984     96
1973     73
1974     69
1972     66
1976     64
1971     36
1970     33
1965     26
1969     25
1968     24
1967     20
1964     19
```

The oldest game was from 3500 BC. Let's find out which game it is.

## Task 7: Find the name of the oldest game, based on column `Year Published`. (5 points)

If your code is correct, it will return Senet as the name of the oldest game. Some Googling wnd Wikipediaing will verify this game is indeed ancient, although we couldn't verify the accuracy of the exact value -3500. That's fine.

```
# Code for Task 7
games.loc[games['Year Published'].idxmin()]['Name']
```

```
'Senet'
```

In the output of Task 6, you see that in general the number of games published in a year has been steadily increasing. However, Year 0 appears to be an outlier, as it has 185 games according to the dataset. This doesn't seem right. To further verify, let's do the following Task 8.

## Task 8: Find how many games in total have been publisehd before 1900. Do not include Year 0 in the count. (5 points)

If your code is correct, it shall return 111. We can thus conclude the statistics for Year 0 cannot be correct. Most likely this is because 0 was used to indicate unknown/missing publishing year when the dataset was created. How confusing that is. This reminds us it is important to make good choices in dealing with missing values.

```
# Code for Task 8
games.loc[(games['Year Published'] < 1900) & (games['Year Published'] != 0)]['Year Published'
```

```
111
```

For the same reason, we believe value 0 in all other columns are not reliable either. Our next task will replace 5000 in `Year Published` and -1 in `Owned Users` by 0. Remember they were actually null values (Task 3). Later we will ignore them together with all 0 values in our analysis.

## Task 9: Replace 5000 in `Year Published` and -1 in `Owned Users` by 0. (5 points)

```
# Code for Task 9
games.loc[games['Year Published'] == 0,'Year Published'] = 5000
games.loc[games['Owned Users'] == 0,'Owned Users'] = -1
```

Also, let's take a look at the frequency of each value in column `Min Players`. If you write a piece of code to find out, the code will be similiar to the one in Task 6. Hence, we don't provide it here. Instead, we directly provide the values, as follows.

```
pd.Series({0: 46, 1: 3270, 2: 14076, 3: 2365, 4: 474, 5: 57, 6: 21, 7: 14, 8: 17, 9: 1, 10: 2
```

```
0        46
1      3270
2     14076
3      2365
4       474
5        57
6        21
7        14
8        17
9         1
10        2
Name: Min Players, dtype: int64
```

Our next task attempts to examine the relationship between `Min Players` and popularity of games measured by ownership.

## Task 10: For each value of `Min Players`, find the average `Owned Users` for games with the corresponding `Min Players` value. Exclude the games with values 0, 9, 10 in `Min Players` and value 0 in `Owned Users`. (10 points)

As we discussed in Task 8, we don't trust the value 0 in any of the columns. We can thus ignore the games with 0 in `Min Players`. Furthermore, there are only 1 and 2 games for `Min Players` 9 and 10, respectively. The statistics of these games won't be meaningful. When we focus on the rest of the games in the output of Task 10, we will observe a general pattern of decresing ownership by minimum required players. That is probably not surprising, since it is easier to find people to play games with less required players.

```
# Code for Task 10
games[(games['Owned Users']>0)&(games['Min Players']>0)&(games['Min Players']<9)].groupby('Mi
```

```
Min Players
1    1762.693909
2    1348.215520
3    1309.574756
4    1218.301688
5    2827.526316
6    1154.761905
7     332.571429
8    1661.705882
Name: Owned Users, dtype: float64
```

The pattern from Task 10 appears to have some exceptions, in games with `Min Players` being 5 and 8. Let's take a further look. Before we continue, let's change `display.max_rows` to 50.

```
pd.set_option('display.max_rows', 50)
```

## Task 11: Find the top-5 owned games for each group of games based on `Min Players`. Ideally, you should also exclude the games with values 0, 9, 10 in `Min Players`, like what we did in Task 10. It is fine if you don't do it here. (10 points)

From the output, you see that there are some quite popular games with 5 and 8 `Min Players`.

```
# Code for Task 11
games[(games['Min Players']>0)&(games['Min Players']<9)].sort_values(['Owned Users'],ascendin
```

| | Name | Year Published | Min Players | Max Players | Play Time | Min Age | Users Rated | Rating Average | BGG Rank | Comp A |
|---|---|---|---|---|---|---|---|---|---|---|
| **98** | Pandemic | 2008 | 2 | 4 | 45 | 8 | 102214 | 7.61 | 99 | |
| **394** | Catan | 1995 | 3 | 4 | 120 | 10 | 101510 | 7.15 | 395 | |
| **177** | Carcassonne | 2000 | 2 | 5 | 45 | 7 | 101853 | 7.42 | 178 | |
| **60** | 7 Wonders | 2010 | 2 | 7 | 30 | 10 | 84371 | 7.75 | 61 | |
| **92** | Codenames | 2015 | 2 | 8 | 15 | 14 | 67688 | 7.62 | 93 | |

## Task 12: Produce a pivot table using `Rating Average` as the rows (i.e., index) and `Complexity Average` as the columns. The cells of the pivot table should show the number of games having the corresponding rating average and complexity average. Since both `Rating Average` and `Complexity Average` are floating point numbers, we should use bins on both. Let's create ten bins [1,2], (2,3], (3,4], (4,5], (5,6], (6,7], (7,8], (8,9], (9,10] on `Rating Average` and four bins [1,2], (2,3], (3,4], (4,5] on `Complexity Average`. (10 points)

The output of `games.describe()` shows that column `Complexity Average` has value 0. Based on our earlier analysis, we shouldn't put much faith in this value. Besides 0, the smallest value in that column is 1. Therefore we decide to have four bins for `Complexity Average`.

```
# Code for Task 12
RA = pd.cut(games['Rating Average'], [0.999,2,3,4,5,6,7,8,9,10])
```

```
CA = pd.cut(games['Complexity Average'], [0.999,2,3,4,5])
games.pivot_table('Name', index=RA, columns=CA, aggfunc='count')
```

| Complexity Average | (0.999, 2.0] | (2.0, 3.0] | (3.0, 4.0] | (4.0, 5.0] |
|---|---|---|---|---|
| Rating Average | | | | |
| **(0.999, 2.0]** | 6.0 | 2.0 | NaN | NaN |
| **(2.0, 3.0]** | 19.0 | 4.0 | 1.0 | NaN |
| **(3.0, 4.0]** | 154.0 | 16.0 | 3.0 | 1.0 |
| **(4.0, 5.0]** | 1023.0 | 101.0 | 18.0 | 2.0 |
| **(5.0, 6.0]** | 3972.0 | 860.0 | 127.0 | 8.0 |
| **(6.0, 7.0]** | 4856.0 | 2906.0 | 650.0 | 57.0 |
| **(7.0, 8.0]** | 1209.0 | 1978.0 | 1029.0 | 163.0 |
| **(8.0, 9.0]** | 102.0 | 276.0 | 256.0 | 88.0 |
| **(9.0, 10.0]** | 9.0 | 9.0 | 9.0 | 3.0 |

Resistance

The pivot table suggests a positive correlation between these two columns. As the complexity of games increases, the rating also tends to increase. Perhaps this is intuitive. For complex games to have a market, it needs to be of higher quality.

In fact, we can directly calculate the correlation using `corr`, as follows. The value of 0.5 in Pearson correlation coefficient suggests a fairly large positive correlation.

```
g = games[games['Complexity Average']>0]

g['Rating Average'].corr(g['Complexity Average'], method='pearson')
```

```
0.510870853513205
```

In the next task, we are going to produce a similar pivot table, focusing on `Owned Users`. Ideally we want to exclude the games with value 0 on this column. To simplify things, we are not requiring you to do it. The pattern we will be seeing is not changed, since only 23 games would have been removed.

## Task 13: Produce another pivot table using `Rating Average` and `Complexity Average`, with the same binning. However, this time the

## cells of the pivot table should show average `Owned Users` of games matching the corresponding rating average and complexity average. (5 points)

```
# Code for Task 13
games.pivot_table('Owned Users', index=RA, columns=CA, aggfunc='mean')
```

| Complexity Average / Rating Average | (0.999, 2.0] | (2.0, 3.0] | (3.0, 4.0] | (4.0, 5.0] |
|---|---|---|---|---|
| **(0.999, 2.0]** | 21.000000 | 90.000000 | NaN | NaN |
| **(2.0, 3.0]** | 526.736842 | 119.750000 | 133.000000 | NaN |
| **(3.0, 4.0]** | 364.181818 | 234.125000 | 207.666667 | 176.000000 |
| **(4.0, 5.0]** | 436.128055 | 232.287129 | 240.444444 | 71.000000 |
| **(5.0, 6.0]** | 539.151309 | 485.956977 | 388.976378 | 321.000000 |
| **(6.0, 7.0]** | 1150.021417 | 1091.726084 | 885.710769 | 811.631579 |
| **(7.0, 8.0]** | 3296.241522 | 3273.023256 | 2928.022352 | 1832.558282 |
| **(8.0, 9.0]** | 734.127451 | 2583.463768 | 4430.777344 | 3669.193182 |
| **(9.0, 10.0]** | 62.222222 | 66.555556 | 75.555556 | 196.000000 |

If your code is correct, this pivot table also shows some interesting patterns. At every rating tier till (7, 8], simpler games enjoy larger ownerships. However, for the really good games with ratings greater than 8, players are not afraid of their complexity. In fact, the more complex games in this rating tier get owned by more players.

Now we will process the `Mechanics` and `Domains` columns. They store values as strings. Each string is a comma-separated list of items. The following code will turn `Mechanics` into a DataFrame itself, with each column corresponding to a unique item from the comma-separated lists. Similarly, we are creating a new DataFrame for the `Domains` column.

```
mechanics = games['Mechanics'].str.get_dummies(sep=", ")
domains = games['Domains'].str.get_dummies(sep=", ")

mechanics = pd.concat([games['Name'], mechanics], 1)
domains = pd.concat([games['Name'], domains], 1)
```

Let's take a look at the columns in the new DataFrame `mechanics`. The values are 1 and 0, i.e., essentially Boolean, indicating whether a game uses the corresponding mechanics or not. This is also called *one-hot encoding*.

```
mechanics.head()
```

| | Name | Acting | Action Drafting | Action Points | Action Queue | Action Retrieval | Action Timer | Action/Event | Advan T |
|---|---|---|---|---|---|---|---|---|---|
| **0** | Gloomhaven | 0 | 0 | 0 | 1 | 1 | 0 | 0 | |
| **1** | Pandemic Legacy: Season 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| **2** | Brass: Birmingham | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **3** | Terraforming Mars | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **4** | Twilight Imperium: Fourth Edition | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |

5 rows × 183 columns

Similarly the new DataFrame `domains` uses one-hot encoding to record the games' domain types.

```
domains.head()
```

| | Name | Abstract Games | Children's Games | Customizable Games | Family Games | Party Games | Strategy Games | Thematic Games | W |
|---|---|---|---|---|---|---|---|---|---|
| **0** | Gloomhaven | 0 | 0 | 0 | 0 | 0 | 1 | 1 | |
| **1** | Pandemic Legacy: Season 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | |
| **2** | Brass: Birmingham | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |

Let's find out which are the most common game domains and which are the least common ones. If your code for the following task is correct, the output should show there are 3316 `Wargames`, the

most common type. The least common type is `Customizable Games`, with 297 games.

## Task 14: For each domain, list its frequency, i.e., the number of games belonging to that domain. (5 points)

```
# Code for Task 14
domains[domains!=0].groupby(['Name']).count().sum()
```

```
Abstract Games        1070
Children's Games       849
Customizable Games     297
Family Games          2173
Party Games            605
Strategy Games        2205
Thematic Games        1174
Wargames              3316
dtype: int64
```

## Task 15: Find out the average `Complexity Average` of games that belong to `Wargames`. You should exclude games with 0 on `Complexity Average`. (10 points)

```
# Code for Task 15
games[(games['Complexity Average']>0)&(domains['Wargames']>0)]['Complexity Average'].mean()
```

```
2.8743964996982436
```

```
# Code for Task 15 (Children's Games)
games[(games['Complexity Average']>0)&(domains["Children's Games"]>0)]['Complexity Average'].
```

```
1.1750418160095577
```

If you code is correct, 2.874 is the value. If you perform the same task on `Children's Games`, you will get 1.175. These two are the two types of games with the largest and smallest average `Complexity Average`. If you want to calculate this for every type of games, the following code does it.

```
results = \
    (games[games['Complexity Average']>0].set_index(games.columns.drop('Domains',1).tolist())
    .Domains.str.split(', ', expand=True)
    .stack()
```

```
    .reset_index()
    .rename(columns={0:'domain'})
    .loc[:,['domain','Owned Users', 'Rating Average', 'Complexity Average']]
    .groupby('domain').agg({'Owned Users':['mean'], 'Rating Average':['mean'], 'Complexity Av
    )
results
```

| | Owned Users | Rating Average | Complexity Average |
|---|---|---|---|
| | mean | mean | mean |
| domain | | | |
| **Abstract Games** | 1362.797170 | 6.228330 | 1.968028 |
| **Children's Games** | 733.255675 | 5.516237 | 1.175042 |
| **Customizable Games** | 2247.594595 | 6.365473 | 2.408581 |
| **Family Games** | 4071.729867 | 6.405568 | 1.658739 |
| **Party Games** | 4089.066116 | 6.285752 | 1.346545 |
| **Strategy Games** | 4438.449433 | 6.957982 | 2.710018 |
| **Thematic Games** | 4045.533220 | 6.789233 | 2.441235 |
| **Wargames** | 705.910984 | 6.847182 | 2.874396 |