25 OCTOBER 2016 / PANDAS

Pandas Tutorial: Data analysis with Python: Part 1

Python is a great language for doing data analysis, primarily because of the fantastic ecosystem of data-centric Python packages. Pandas is one of those packages, and makes importing and analyzing data much easier. Pandas builds on packages like NumPy and matplotlib to give you a single, convenient, place to do most of your data analysis and visualization work.

In this introduction, we'll use Pandas to analyze data on video game reviews from IGN, a popular video game review site. The data was scraped by Eric Grinstein, and can be found here. As we analyze the video game reviews, we'll learn key Pandas concepts like indexing.

Do games like the Witcher 3 tend to get better reviews on the PS4 than the Xbox One? This dataset can help us find out.

Just as a note, we'll be using Python 3.5 and Jupyter Notebook to do our analysis.

Importing Data with Pandas

The first step we'll take is to read the data in. The data is stored as a comma-separated values, or csv, file, where each row is separated by a new line, and each column by a comma (`,`). Here are the first few rows of the `ign.csv` file:

```
,score_phrase,title,url,platform,score,genre,editors_choice,rele
0,Amazing,LittleBigPlanet PS Vita,/games/littlebigplanet-vita/vi
1,Amazing,LittleBigPlanet PS Vita -- Marvel Super Hero Edition,/
2,Great,Splice: Tree of Life,/games/splice/ipad-141070,iPad,8.5,
3,Great,NHL 13,/games/nhl-13/xbox-360-128182,Xbox 360,8.5,Sports
```

As you can see above, each row in the data represents a single game that was reviewed by IGN. The columns contain information about that game:

- `score_phrase` — how IGN described the game in one word. This is linked to the score it received.
- `title` — the name of the game.
- `url` — the URL where you can see the full review.

- `platform` — the platform the game was reviewed on (PC, PS4, etc).
- `score` — the score for the game, from `1.0` to `10.0`.
- `genre` — the genre of the game.
- `editors_choice` — `N` if the game wasn't an editor's choice, `Y` if it was. This is tied to score.
- `release_year` — the year the game was released.
- `release_month` — the month the game was released.
- `release_day` — the day the game was released.

There's also a leading column that contains row index values. We can safely ignore this column, but we'll dive into what index values are later on. In order to be able to work with the data in Python, we'll need to read the csv file into a Pandas DataFrame. A DataFrame is a way to represent and work with tabular data. Tabular data has rows and columns, just like our csv file.

In order to read in the data, we'll need to use the pandas.read_csv function. This function will take in a csv file and return a DataFrame. The below code will:

- Import the `pandas` library. We rename it to `pd` so it's faster to type out.
- Read `ign.csv` into a DataFrame, and assign the result to `reviews`.

```
import pandas as pd

reviews = pd.read_csv("ign.csv")
```

Once we read in a DataFrame, Pandas gives us two methods that make it fast to print out the data. These functions are:

- pandas.DataFrame.head -- prints the first N rows of a DataFrame. By default `5`.
- pandas.DataFrame.tail -- prints the last N rows of a DataFrame. By default `5`.

We'll use the `head` method to see what's in `reviews`:

```
reviews.head()
```

	Unnamed: 0	score_phrase	title
0	0	Amazing	LittleBigPlanet PS Vita
1	1	Amazing	LittleBigPlanet PS Vita -- Marvel Super He
2	2	Great	Splice: Tree of Life
3	3	Great	NHL 13
4	4	Great	NHL 13

We can also access the pandas.DataFrame.shape property to see row many rows and columns are in `reviews`:

```
reviews.shape
```

```
(18625, 11)
```

As you can see, everything has been read in properly -- we have `18625` rows and `11` columns.

One of the big advantages of Pandas vs just using NumPy is that Pandas allows you to have columns with different data types. `reviews` has columns that store float values, like `score`, string values, like `score_phrase`, and integers, like `release_year`.

Now that we've read the data in properly, let's work on indexing `reviews` to get the rows and columns that we want.

Indexing DataFrames with Pandas

Earlier, we used the `head` method to print the first `5` rows of `reviews`. We could accomplish the same thing using the pandas.DataFrame.iloc method. The `iloc` method allows us to retrieve rows and columns by position. In order to do that, we'll need to specify the positions of the rows that we want, and the positions of the columns that we want as well.

The below code will replicate `reviews.head()`:

```
reviews.iloc[0:5,:]
```

	Unnamed: 0	score_phrase	title
0	0	Amazing	LittleBigPlanet PS Vita
1	1	Amazing	LittleBigPlanet PS Vita -- Marvel Super He
2	2	Great	Splice: Tree of Life
3	3	Great	NHL 13
4	4	Great	NHL 13

As you can see above, we specified that we wanted rows `0:5`. This means that we wanted the rows from position `0` up to, but not including, position `5`. The first row is considered to be in position `0`. This gives us the rows at positions `0`, `1`, `2`, `3`, and `4`.

If we leave off the first position value, like `:5`, it's assumed we mean `0`. If we leave off the last position value, like `0:`, it's assumed we mean the last row or column in the DataFrame.

We wanted all of the columns, so we specified just a colon (`:`), without any positions. This gave us the columns from `0` to the last column.

Here are some indexing examples, along with the results:

- `reviews.iloc[:5,:]` — the first `5` rows, and all of the columns for those rows.
- `reviews.iloc[:,:]` — the entire DataFrame.
- `reviews.iloc[5:,5:]` — rows from position `5` onwards, and columns from position `5` onwards.
- `reviews.iloc[:,0]` — the first column, and all of the rows for the column.
- `reviews.iloc[9,:]` — the 10th row, and all of the columns for that row.

Indexing by position is very similar to NumPy indexing. If you want to learn more, you can read our NumPy tutorial here.

Now that we know how to index by position, let's remove the first column, which doesn't have any useful information:

```
reviews = reviews.iloc[:,1:]
reviews.head()
```

	score_phrase	title	url
0	Amazing	LittleBigPlanet PS Vita	/gam
1	Amazing	LittleBigPlanet PS Vita -- Marvel Super Hero E...	/gam
2	Great	Splice: Tree of Life	/gam
3	Great	NHL 13	/gam
4	Great	NHL 13	/gam

Indexing Using Labels in Pandas

Now that we know how to retrieve rows and columns by position, it's worth looking into the other major way to work with DataFrames, which is to retrieve rows and columns by label.

A major advantage of Pandas over NumPy is that each of the columns and rows has a label. Working with column positions is possible, but it can be hard to keep track of which number corresponds to which column.

We can work with labels using the pandas.DataFrame.loc method, which allows us to index using labels instead of positions.
We can display the first five rows of `reviews` using the `loc` method like this:

```
reviews.loc[0:5,:]
```

	score_phrase	title	url
0	Amazing	LittleBigPlanet PS Vita	/gam
1	Amazing	LittleBigPlanet PS Vita -- Marvel Super Hero E...	/gam
2	Great	Splice: Tree of Life	/gam
3	Great	NHL 13	/gam
4	Great	NHL 13	/gam
5	Good	Total War Battles: Shogun	/gam

The above doesn't actually look much different from `reviews.iloc[0:5,:]`. This is because while row labels can take on any values, our row labels match the positions exactly. You can see the row labels on the very left of the table above (they're in bold). You can also see them by accessing the index property of a DataFrame. We'll display the row indexes for `reviews`:

```
reviews.index
```

```
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
```

Indexes don't always have to match up with positions, though. In the below code cell, we'll:

- Get row `10` to row `20` of `reviews`, and assign the result to `some_reviews`.
- Display the first `5` rows of `some_reviews`.

```
some_reviews = reviews.iloc[10:20,]
some_reviews.head()
```

	score_phrase	title	url
10	Good	Tekken Tag Tournament 2	/games/tekken-tag-tourna
11	Good	Tekken Tag Tournament 2	/games/tekken-tag-tourna
12	Good	Wild Blood	/games/wild-blood/iphone
13	Amazing	Mark of the Ninja	/games/mark-of-the-ninja
14	Amazing	Mark of the Ninja	/games/mark-of-the-ninja

As you can see above, in `some_reviews`, the row indexes start at `10` and end at `20`. Thus, trying `loc` along with numbers lower than `10` or higher than `20` will result in an error:

```
some_reviews.loc[9:21,:]
```

```
---------------------------------------------------------------------
KeyError                                  Traceback (most recer
<ipython-input-76-5378b774c9a7> in <module>()
----> 1 some_reviews.loc[9:21,:]

/Users/vik/python_envs/dsserver/lib/python3.4/site-packages/par
   1198     def __getitem__(self, key):
   1199         if type(key) is tuple:
-> 1200             return self._getitem_tuple(key)
   1201         else:
   1202             return self._getitem_axis(key, axis=0)

/Users/vik/python_envs/dsserver/lib/python3.4/site-packages/par
    702
    703         # no multi-index, so validate all of the indexe
--> 704         self._has_valid_tuple(tup)
    705
    706         # ugly hack for GH #836

/Users/vik/python_envs/dsserver/lib/python3.4/site-packages/par
    129             if i >= self.obj.ndim:
    130                 raise IndexingError('Too many indexers'
--> 131             if not self._has_valid_type(k, i):
    132                 raise ValueError("Location based indexi
    133                                  "types" % self._valid_

/Users/vik/python_envs/dsserver/lib/python3.4/site-packages/par
   1258                         raise KeyError(
   1259                             "start bound [%s] is not th
-> 1260                             (key.start, self.obj._get_a
   1261                         )
   1262                 if key.stop is not None:

KeyError: 'start bound [9] is not the [index]'
```

As we mentioned earlier, column labels can make life much easier when you're working with data. We can specify column labels in the `loc` method to retrieve columns by label instead of by position.

```
reviews.loc[:5,"score"]
```

```
0    9.0
1    9.0
2    8.5
3    8.5
4    8.5
5    7.0
Name: score, dtype: float64
```

We can also specify more than one column at a time by passing in a list:

```
reviews.loc[:5,["score", "release_year"]]
```

	score	release_year
0	9.0	2012
1	9.0	2012
2	8.5	2012
3	8.5	2012
4	8.5	2012
5	7.0	2012

Pandas Series Objects

We can retrieve an individual column in Pandas a few different ways. So far, we've seen two types of syntax for this:

- `reviews.iloc[:,1]` — will retrieve the second column.
- `reviews.loc[:,"score_phrase"]` — will also retrieve the second column.

There's a third, even easier, way to retrieve a whole column. We can just specify the column name in square brackets, like with a dictionary:

```
reviews["score"]
```

```
0      9.0
1      9.0
2      8.5
3      8.5
4      8.5
5      7.0
6      3.0
7      9.0
8      3.0
9      7.0
10     7.5
11     7.5
12     7.0
13     9.0
14     9.0
...
18610     6.0
18611     5.8
18612     7.8
18613     8.0
18614     9.2
18615     9.2
18616     7.5
18617     8.4
18618     9.1
18619     7.9
18620     7.6
```

```
18621     9.0
18622     5.8
18623    10.0
18624    10.0
Name: score, Length: 18625, dtype: float64
```

We can also use lists of columns with this method:

```
reviews[["score", "release_year"]]
```

	score	release_year
18596	6.5	2016
18597	4.9	2016
18598	6.8	2016
18599	7.0	2016
18600	7.4	2016
18601	7.4	2016
18602	7.4	2016
18603	7.8	2016
18604	8.6	2016
18605	6.0	2016
18606	6.4	2016
18607	7.0	2016
18608	5.4	2016
18609	8.0	2016
18610	6.0	2016
18611	5.8	2016

18612	7.8	2016
18613	8.0	2016
18614	9.2	2016
18615	9.2	2016
18616	7.5	2016
18617	8.4	2016

When we retrieve a single column, we're actually retrieving a Pandas Series object. A DataFrame stores tabular data, but a Series stores a single column or row of data.

We can verify that a single column is a Series:

```
type(reviews["score"])
```

```
pandas.core.series.Series
```

We can create a Series manually to better understand how it works. To create a Series, we pass a list or NumPy array into the Series object when we instantiate it:

```
s1 = pd.Series([1,2])
s1
```

```
0    1
1    2
dtype: int64
```

A Series can contain any type of data, including mixed types. Here, we create a Series that contains string objects:

```
s2 = pd.Series(["Boris Yeltsin", "Mikhail Gorbachev"])
s2
```

```
0        Boris Yeltsin
1    Mikhail Gorbachev
dtype: object
```

Creating A DataFrame in Pandas

We can create a DataFrame by passing multiple Series into the DataFrame class. Here, we pass in the two Series objects we just created, `s1` as the first row, and `s2` as the second row:

```
pd.DataFrame([s1,s2])
```

	0	1
0	1	2
1	Boris Yeltsin	Mikhail Gorbachev

We can also accomplish the same thing with a list of lists. Each inner list is treated as a row in the resulting DataFrame:

```
pd.DataFrame(
    [
        [1,2],
        ["Boris Yeltsin", "Mikhail Gorbachev"]
    ]
)
```

	0	1
0	1	2
1	Boris Yeltsin	Mikhail Gorbachev

We can specify the column labels when we create a DataFrame:

```
pd.DataFrame(
    [
        [1,2],
        ["Boris Yeltsin", "Mikhail Gorbachev"]
    ],
    columns=["column1", "column2"]
)
```

	column1	column2
0	1	2
1	Boris Yeltsin	Mikhail Gorbachev

As well as the row labels (the index):

```
frame = pd.DataFrame(
    [
        [1,2],
        ["Boris Yeltsin", "Mikhail Gorbachev"]
    ],
    index=["row1", "row2"],
    columns=["column1", "column2"]
)
frame
```

	column1	column2
row1	1	2
row2	Boris Yeltsin	Mikhail Gorbachev

We're then able index the DataFrame using the labels:

```
frame.loc["row1":"row2", "column1"]
```

```
row1                1
row2    Boris Yeltsin
Name: column1, dtype: object
```

We can skip specifying the `columns` keyword argument if we pass a dictionary into the `DataFrame` constructor. This will automatically setup column names:

```
frame = pd.DataFrame(
    {
        "column1": [1, "Boris Yeltsin"],
```

```
        "column2": [2, "Mikhail Gorbachev"]
    }
)
frame
```

	column1	column2
0	1	2
1	Boris Yeltsin	Mikhail Gorbachev

Pandas DataFrame Methods

As we mentioned earlier, each column in a DataFrame is a Series object:

```
type(reviews["title"])
```

```
pandas.core.series.Series
```

We can call most of the same methods on a Series object that we can on a DataFrame, including `head`:

```
reviews["title"].head()
```

```
0                              LittleBigPlanet PS Vita
1    LittleBigPlanet PS Vita -- Marvel Super Hero E...
```

```
2                                  Splice: Tree of Life
3                                               NHL 13
4                                               NHL 13
Name: title, dtype: object
```

Pandas Series and DataFrames also have other methods that make calculations simpler. For example, we can use the pandas.Series.mean method to find the mean of a Series:

```
reviews["score"].mean()
```

```
6.950459060402685
```

We can also call the similar pandas.DataFrame.mean method, which will find the mean of each numerical column in a DataFrame by default:

```
reviews.mean()
```

```
score              6.950459
release_year    2006.515329
release_month      7.138470
release_day       15.603866
dtype: float64
```

We can modify the `axis` keyword argument to `mean` in order to

compute the mean of each row or of each column. By default, `axis` is equal to `0`, and will compute the mean of each column. We can also set it to `1` to compute the mean of each row. Note that this will only compute the mean of the numerical values in each row:

```
reviews.mean(axis=1)
```

```
0      510.500
1      510.500
2      510.375
3      510.125
4      510.125
5      509.750
6      508.750
7      510.250
8      508.750
9      509.750
10     509.875
11     509.875
12     509.500
13     509.250
14     509.250
...
18610    510.250
18611    508.700
18612    509.200
18613    508.000
18614    515.050
18615    515.050
18616    508.375
18617    508.600
18618    515.025
18619    514.725
18620    514.650
18621    515.000
18622    513.950
18623    515.000
18624    515.000
Length: 18625, dtype: float64
```

There are quite a few methods on Series and DataFrames that behave like `mean`. Here are some handy ones:

- pandas.DataFrame.corr — finds the correlation between columns in a DataFrame.
- pandas.DataFrame.count — counts the number of non-null values in each DataFrame column.
- pandas.DataFrame.max — finds the highest value in each column.
- pandas.DataFrame.min — finds the lowest value in each column.
- pandas.DataFrame.median — finds the median of each column.
- pandas.DataFrame.std — finds the standard deviation of each column.

We can use the `corr` method to see if any columns correlation with `score`. For instance, this would tell us if games released more recently have been getting higher reviews (`release_year`), or if games released towards the end of the year score better (`release_month`):

```
reviews.corr()
```

	score	release_year	release_month	release_day
score	1.000000	0.062716	0.007632	0.020079
release_year	0.062716	1.000000	-0.115515	0.016867
release_month	0.007632	-0.115515	1.000000	-0.067964
release_day	0.020079	0.016867	-0.067964	1.000000

As you can see above, none of our numeric columns correlates with `score`, meaning that release timing doesn't linearly relate to review score.

DataFrame Math with Pandas

We can also perform math operations on Series or DataFrame objects. For example, we can divide every value in the `score` column by `2` to switch the scale from `0-10` to `0-5`:

```
reviews["score"] / 2
```

```
0     4.50
1     4.50
2     4.25
3     4.25
4     4.25
5     3.50
6     1.50
7     4.50
8     1.50
9     3.50
10    3.75
```

```
11    3.75
12    3.50
13    4.50
14    4.50
...
18610    3.00
18611    2.90
18612    3.90
18613    4.00
18614    4.60
18615    4.60
18616    3.75
18617    4.20
18618    4.55
18619    3.95
18620    3.80
18621    4.50
18622    2.90
18623    5.00
18624    5.00
Name: score, Length: 18625, dtype: float64
```

All the common mathematical operators that work in Python, like `+`, `-`, `*`, `/`, and `^` will work, and will apply to each element in a DataFrame or a Series.

Boolean Indexing in Pandas

As we saw above, the mean of all the values in the `score` column of `reviews` is around `7`. What if we wanted to find all the games that got an above average score? We could start by doing a comparison. The comparison compares each value in a Series to a specified value, then generate a Series full of Boolean values indicating the status of the comparison. For example, we can see which of the rows have a `score` value higher than `7`:

```
score_filter = reviews["score"] > 7
score_filter
```

```
0      True
1      True
2      True
3      True
4      True
5     False
6     False
7      True
8     False
9     False
10     True
11     True
12    False
13     True
14     True
...
18610    False
18611    False
18612     True
18613     True
18614     True
18615     True
18616     True
18617     True
18618     True
18619     True
18620     True
18621     True
18622    False
18623     True
18624     True
Name: score, Length: 18625, dtype: bool
```

Once we have a Boolean Series, we can use it to select only rows in a DataFrame where the Series contains the value `True`. So, we could only select rows in `reviews` where `score` is greater than `7`:

```
filtered_reviews = reviews[score_filter]
filtered_reviews.head()
```

	score_phrase	title	url
0	Amazing	LittleBigPlanet PS Vita	/gam
1	Amazing	LittleBigPlanet PS Vita -- Marvel Super Hero E...	/gam
2	Great	Splice: Tree of Life	/gam
3	Great	NHL 13	/gam
4	Great	NHL 13	/gam

It's possible to use multiple conditions for filtering. Let's say we want to find games released for the `Xbox One` that have a score of more than `7`. In the below code, we:

- Setup a filter with two conditions:
  - Check if `score` is greater than `7`.
  - Check if `platform` equals `Xbox One`
- Apply the filter to `reviews` to get only the rows we want.
- Use the `head` method to print the first `5` rows of `filtered_reviews`.

```
xbox_one_filter = (reviews["score"] > 7) & (reviews["platform"]
filtered_reviews = reviews[xbox_one_filter]
filtered_reviews.head()
```

	score_phrase	title	url
17137	Amazing	Gone Home	/games/gone-home/x
17197	Amazing	Rayman Legends	/games/rayman-legei
17295	Amazing	LEGO Marvel Super Heroes	/games/lego-marvel-
17313	Great	Dead Rising 3	/games/dead-rising-3
17317	Great	Killer Instinct	/games/killer-instinct

When filtering with multiple conditions, it's important to put each condition in parentheses, and separate them with a single ampersand (`&`).

Pandas Plotting

Now that we know how to filter, we can create plots to observe the review distribution for the `Xbox One` vs the review distribution for the `PlayStation 4`. This will help us figure out which console has better games. We can do this via a histogram, which will plot the frequencies for different score ranges. This will tell us which console has more highly reviewed games.

We can make a histogram for each console using the pandas.DataFrame.plot method. This method utilizes matplotlib,

the popular Python plotting library, under the hood to generate good-looking plots.

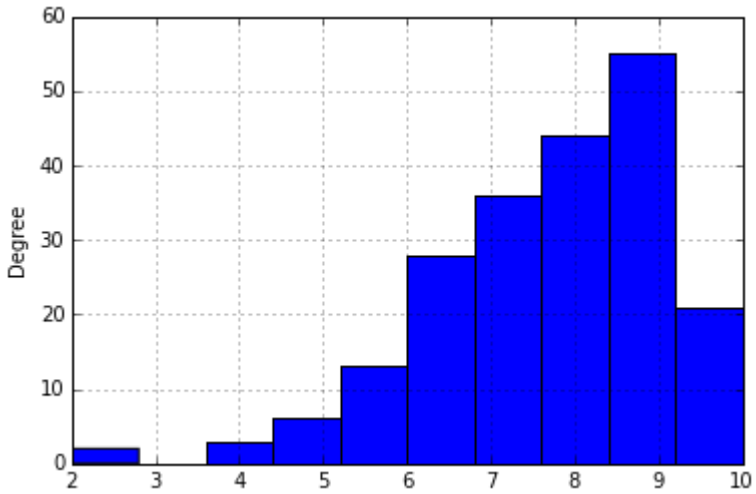
The `plot` method defaults to drawing a line graph. We'll need to pass in the keyword argument `kind="hist"` to draw a histogram instead.

In the below code, we:

- Call `%matplotlib inline` to set up plotting inside a Jupyter notebook.
- Filter `reviews` to only have data about the `Xbox One`.
- Plot the `score` column.

```
%matplotlib inline
reviews[reviews["platform"] == "Xbox One"]["score"].plot(kind="h
```

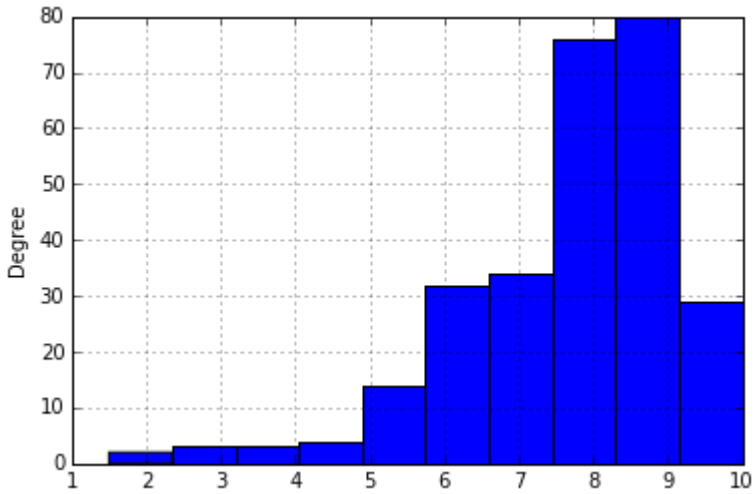
```
<matplotlib.axes._subplots.AxesSubplot at 0x10c9c5438>
```



We can also do the same for the `PS4`:

```
reviews[reviews["platform"] == "PlayStation 4"]["score"].plot(ki
```

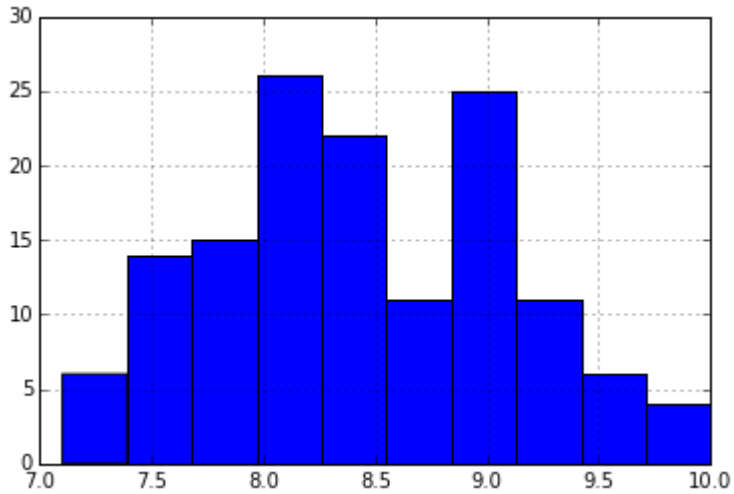
```
<matplotlib.axes._subplots.AxesSubplot at 0x10c9e0e80>
```



It appears from our histogram that the `PlayStation 4` has many more highly rated games than the `Xbox One`.

```
filtered_reviews["score"].hist()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10d520c18>
```



Free Pandas Cheat Sheet

If you're interested in learning more about pandas, check out our interactive course on NumPy and pandas. You can register and do the first missions for free.

You also might like to take your pandas skills to the next level with our free pandas cheat sheet!

Further Reading

You should now know how to read, explore, analyze, and visualize data using Pandas and Python. In the next post, we cover grouping data and doing more advanced computations. You can find it here.

If you want to read more about Pandas, check out these resources:

- Dataquest Pandas Course
- 10 minutes to Pandas
- Intro to Pandas data structures

Vik Paruchuri
Read more posts by this author.

Read More

— Dataquest —
Pandas

Jupyter Notebook for Beginners: A Tutorial

Visualizing Women's Marches: Part 1

Adding Axis Labels to Plots With pandas

See all 25 posts →

PYTHON
Nov 04, 2016
What's New in Dataquest v1.9: Console, hotkeys, and more!

Read about the latest new features from Dataquest, including console, hotkeys, and more!


JOSH DEVLIN

NUMPY
Oct 18, 2016
NumPy Tutorial: Data analysis with Python

This NumPy tutorial introduces key concepts and teaches you to analyze data efficiently. It includes comparing, filtering, reshaping, and combining NumPy arrays

includes comparing, filtering, reshaping, and combining NumPy arrays.

VIK PARUCHURI