FTDS // PANDAS BASIC

Hacktiv8 DS Curriculum Team

Phase 0
Day 5 AM
2021

Phase 0

2021

Contents

Phase 0

2021

Contents

//03

WEEK 1

Pandas: Basics

Introduction to Pandas

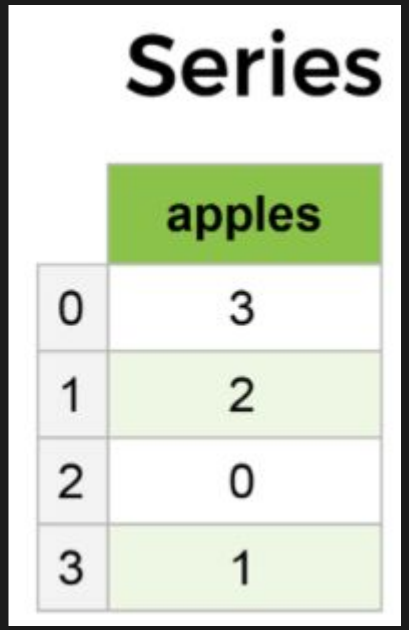
- Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.
- Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, prepare, manipulate, model, and analyze.
- Pandas generally provide two data structures for manipulating data, They are:
  - **Series**
  - **DataFrame**

WEEK 1

Pandas: Basics

Series

- Pandas Series is a **one-dimensional** labelled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called indexes.


Series

	apples
0	3
1	2
2	0
3	1

```
import pandas as pd
import numpy as np


# Creating empty series
ser = pd.Series()

print(ser)

# simple array
data = np.array(['3', '2', '0', '1'])

ser = pd.Series(data)
print(ser)
```

WEEK 1

Pandas: Basics

Understanding Series Objects

Python's most basic data structure is the list, which is also a good starting point for getting to know pandas.Series objects. Create a new Series object based on a list:

revenues = pd.Series([5555, 7000, 1980])

You've used the list [5555, 7000, 1980] to create a Series object called revenues. A Series object wraps two components:

- **A sequence of values**
- **A sequence of identifiers, which is the index**

You can access these components with .values and .index, respectively:

revenues.values

revenues.index

Understanding Series Objects

A Pandas Series also has an integer index that's implicitly defined. This implicit index indicates the element's position in the Series.

However, a Series can also have an arbitrary type of index. You can think of this explicit index as labels for a specific row:

```
[ ]  city_revenues = pd.Series(
         [4200, 8000, 6500],
         index=["Amsterdam", "Toronto", "Tokyo"]
     )
     city_revenues
```

```
Amsterdam    4200
Toronto      8000
Tokyo        6500
dtype: int64
```

for n in x:
    print(n)

WEEK 1

Pandas: Basics

Understanding Series Objects

Here's how to construct a Series with a label index from a Python dictionary:

```
[ ] city_employee_count = pd.Series({"Amsterdam": 5, "Tokyo": 8})
    city_employee_count

    Amsterdam    5
    Tokyo        8
    dtype: int64
```

The dictionary keys become the index, and the dictionary values are the Series values.

Just like dictionaries, Series also support .keys() and the in keyword:

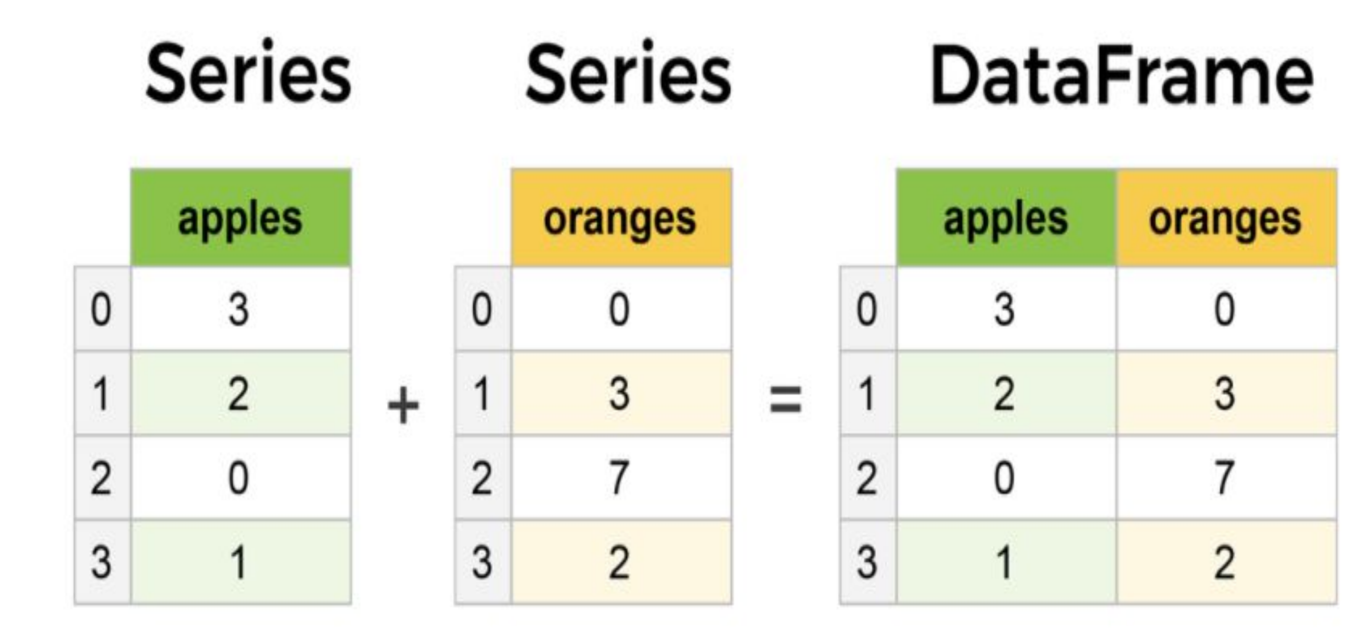
```
[ ] city_employee_count.keys()

    Index(['Amsterdam', 'Tokyo'], dtype='object')

[ ] "Tokyo" in city_employee_count

    True

[ ] "New York" in city_employee_count

    False
```

WEEK 1

Pandas: Basics

DataFrame

❖ DataFrame is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Pandas DataFrame consists of three principal components, the data, rows, and columns.




```
import pandas as pd

# Calling DataFrame constructor
df = pd.DataFrame()
print(df)

# Dictionary
data = {
  "Apples": [420, 380, 390],
  "Oranges": [50, 40, 45]
}

# Calling DataFrame
df = pd.DataFrame(data)
print(df)
```

WEEK 1

Pandas: Basics

Understanding DataFrame Objects

As you've seen with the nba dataset, which features 23 columns, the Pandas Python library has more to offer with its DataFrame. This data structure is a sequence of Series objects that share the same index.

If you've followed along with the Series examples, then you should already have two Series objects with cities as keys:

- city_revenues
- city_employee_count

Understanding DataFrame Objects

You can combine city_revenues and city_employee_count objects into a DataFrame by providing a dictionary in the constructor. The dictionary keys will become the column names, and the values should contain the Series objects:

```
[ ] city_data = pd.DataFrame({
        "revenue": city_revenues,
        "employee_count": city_employee_count
    })
```

```
[ ] city_data
```

	revenue	employee_count
Amsterdam	4200	5.0
Tokyo	6500	8.0
Toronto	8000	NaN

Note how Pandas replaced the missing employee_count value for Toronto with NaN.

WEEK 1

Pandas: Basics

Understanding DataFrame Objects

The new DataFrame index is the union of the two Series indices:

```
city_data.index

Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object')
```

Just like a Series, a DataFrame also stores its values in a NumPy array:

```
[ ]  city_data.values

     array([[4.2e+03, 5.0e+00],
            [6.5e+03, 8.0e+00],
            [8.0e+03,     nan]])
```

You can also refer to the 2 dimensions of a DataFrame as axes:

```
[ ]  city_data.axes

     [Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object'),
      Index(['revenue', 'employee_count'], dtype='object')]

[ ]  city_data.axes[0]

     Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object')
```

```
city_data.axes[1]

Index(['revenue', 'employee_count'], dtype='object')
```

WEEK 1

Pandas: Basics

Accessing Series Elements

You can conveniently access the values in a Series with both the label and positional indices:

```
[ ] city_revenues["Toronto"]

8000

[ ] city_revenues[1]

8000
```

You can also use negative indices and slices, just like you would for a list:

```
[ ] city_revenues[-1]

6500

[ ] city_revenues[1:]

Toronto     8000
Tokyo       6500
dtype: int64

[ ] city_revenues["Toronto":]

Toronto     8000
Tokyo       6500
dtype: int64
```

WEEK 1

Pandas: Basics

Using .loc and .iloc

The indexing operator ([]) is convenient, but there's a caveat. What if the labels are also numbers? Say you have to work with a Series object like this:

```
[ ]  colors = pd.Series(
         ["red", "purple", "blue", "green", "yellow"],
         index=[1, 2, 3, 5, 8]
     )

[ ]  colors

     1        red
     2     purple
     3       blue
     5      green
     8     yellow
     dtype: object
```

The Pandas Python library provides two data access methods:

- **.loc refers to the label index.**
- **.iloc refers to the positional index.**

WEEK 1

Pandas: Basics

Using .loc and .iloc

colors.loc[1] returned "red", the element with the label 1. colors.iloc[1] returned "purple", the element with the index 1.

.loc and .iloc also support the features you would expect from indexing operators, like slicing. While .iloc excludes the closing element, .loc includes it. Take a look at this code block:

```
[ ]  # Return the elements with the implicit index: 1, 2

     colors.iloc[1:3]

     2    purple
     3      blue
     dtype: object
```

On the other hand, .loc includes the closing element:

```
[ ]  # Return the elements with the explicit index between 3 and 8

     colors.loc[3:8]

     3      blue
     5     green
     8    yellow
     dtype: object
```

WEEK 1

Pandas: Basics

Accessing DataFrame Elements

If you think of a DataFrame as a dictionary whose values are Series, then it makes sense that you can access its columns with the indexing operator:

```
[ ]  city_data["revenue"]

     Amsterdam    4200
     Tokyo        6500
     Toronto      8000
     Name: revenue, dtype: int64
```

If the column name is a string, then you can use attribute-style accessing with dot notation as well:

```
[ ]  city_data.revenue

     Amsterdam    4200
     Tokyo        6500
     Toronto      8000
     Name: revenue, dtype: int64
```

Accessing DataFrame Elements

Similar to Series, a DataFrame also provides .loc and .iloc data access methods.

Remember, .loc uses the label and .iloc the positional index:

```
[ ] city_data.loc["Amsterdam"]

revenue           4200.0
employee_count       5.0
Name: Amsterdam, dtype: float64
```

```
[ ] city_data.loc["Tokyo": "Toronto"]
```

	revenue	employee_count
Tokyo	6500	8.0
Toronto	8000	NaN

```
[ ] city_data.iloc[1]

revenue           6500.0
employee_count       8.0
Name: Tokyo, dtype: float64
```

WEEK 1

Pandas: Basics

Combining Multiple Datasets

you'll take this one step further and use .concat() to combine city_data with another DataFrame.

Say you've managed to gather some data on two more cities:

```
further_city_data = pd.DataFrame(
    {"revenue": [7000, 3400], "employee_count": [2, 2]},
    index=["New York", "Barcelona"]
)
```

```
further_city_data
```

	revenue	employee_count
New York	7000	2
Barcelona	3400	2

```
city_data
```

	revenue	employee_count
Amsterdam	4200	5.0
Tokyo	6500	8.0

```
all_city_data = pd.concat([city_data, further_city_data], sort=False)
```

```
all_city_data
```

	revenue	employee_count
Amsterdam	4200	5.0
Tokyo	6500	8.0
Toronto	8000	NaN
New York	7000	2.0
Barcelona	3400	2.0

WEEK 1

Pandas: Basics

Using the Pandas Python Library

In this session, you'll analyze NBA results provided by FiveThirtyEight in a 17MB CSV file. Here, you follow the convention of importing Pandas in Python with the pd alias. Then, you use .read_csv() to read in your dataset and store it as a DataFrame object in the variable df:

```
import numpy as np
import pandas as
df=pd.read_csv('https://raw.githubusercontent.com/ardhiraka/PFDS_sources/master/nbaallelo.csv')
```

You can use the Python built-in function len() to determine the number of rows. You also use the .shape attribute of the DataFrame to see its dimensionality.

```
len(df)
df.shape
```

WEEK 1

Pandas: Basics

Using the Pandas Python Library

You can have a look at the first five rows with .head():

```
df.head()
```

Unless your screen is quite large, your output probably won't display all 23 columns. Somewhere in the middle, you'll see a column of ellipses (...) indicating the missing data. You can configure Pandas to display all 23 columns like this:

```
pd.set_option("display.max.columns", None)
```

While it's practical to see all the columns, you probably won't need six decimal places! Change it to two:

```
pd.set_option("display.precision", 2)
```

WEEK 1

Pandas: Basics

Python Modules: Overview

you can display the last five rows with .tail() instead:

df.tail()

You can discover some further possibilities of .head() and .tail() with a small exercise. For example you can display the last three lines:

df.tail(3)

#the first three rows

df.head(3)

WEEK 1

Pandas: Basics

Getting to Know Your Data

You can display all columns and their data types with .info():

df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 126314 entries, 0 to 126313
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   gameorder      126314 non-null  int64
 1   game_id        126314 non-null  object
 2   lg_id          126314 non-null  object
 3   _iscopy        126314 non-null  int64
 4   year_id        126314 non-null  int64
 5   date_game      126314 non-null  object
 6   seasongame     126314 non-null  int64
 7   is_playoffs    126314 non-null  int64
 8   team_id        126314 non-null  object
 9   fran_id        126314 non-null  object
 10  pts            126314 non-null  int64
 11  elo_i          126314 non-null  float64
 12  elo_n          126314 non-null  float64
 13  win_equiv      126314 non-null  float64
 14  opp_id         126314 non-null  object
 15  opp_fran       126314 non-null  object
 16  opp_pts        126314 non-null  int64
```

You'll see a list of all the columns in your dataset and the type of data each column contains. Here, you can see the data types int64, float64, and object.

WEEK 1

Pandas: Basics

Showing Basics Statistics

Now that you've seen what data types are in your dataset, it's time to get an overview of the values each column contains. You can do this with .describe():

df.describe()

	gameorder	_iscopy	year_id	seasongame	is_playoffs	pts	elo_i	elo_n	win_equiv	opp_pts
count	126314.00	126314.0	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00
mean	31579.00	0.5	1988.20	43.53	0.06	102.73	1495.24	1495.24	41.71	102.73
std	18231.93	0.5	17.58	25.38	0.24	14.81	112.14	112.46	10.63	14.81
min	1.00	0.0	1947.00	1.00	0.00	0.00	1091.64	1085.77	10.15	0.00
25%	15790.00	0.0	1975.00	22.00	0.00	93.00	1417.24	1416.99	34.10	93.00
50%	31579.00	0.5	1990.00	43.00	0.00	103.00	1500.95	1500.95	42.11	103.00
75%	47368.00	1.0	2003.00	65.00	0.00	112.00	1576.06	1576.29	49.64	112.00
max	63157.00	1.0	2015.00	108.00	1.00	186.00	1853.10	1853.10	71.11	186.00

WEEK 1

Pandas: Basics

Exploring Your Dataset

Exploratory data analysis can help you answer questions about your dataset. For example, you can examine how often specific values occur in a column:

df["team_id"].value_counts()

```
BOS    5997
NYK    5769
LAL    5078
DET    4985
PHI    4533
       ...
PIT      60
TRH      60
INJ      60
DTF      60
SDS      11
Name: team_id, Length: 104, dtype: int64
```

Find out who the other "Lakers" team is:

df.loc[df["fran_id"] == "Lakers", "team_id"].value_counts()

```
LAL    5078
MNL     946
Name: team_id, dtype: int64
```

WEEK 1

Pandas: Basics

Exploring Your Dataset

Indeed, the Minneapolis Lakers ("MNL") played 946 games. You can even find out when they played those games:

```
df.loc[df["team_id"] == "MNL", "date_game"].min()

'1/1/1949'

[ ] df.loc[df["team_id"] == "MNL", "date_game"].max()

'4/9/1959'

[ ] df.loc[df["team_id"] == "MNL", "date_game"].agg(("min", "max"))

min    1/1/1949
max    4/9/1959
Name: date_game, dtype: object
```

It looks like the Minneapolis Lakers played between the years of 1949 and 1959. That explains why you might not recognize this team!

WEEK 1

Pandas: Basics

Exploring Your Dataset

Find out how many points the Boston Celtics have scored during all matches contained in this dataset.

```
[ ]  df.loc[df["team_id"] == "BOS", "pts"].sum()

     626484
```

The Boston Celtics scored a total of 626,484 points.

WEEK 1

Pandas: Basics

Querying Your Dataset

you can create a new DataFrame that contains only games played after 2010:

current_decade = df[df["year_id"] > 2010]

```
current_decade
```

	gameorder	game_id	lg_id	_iscopy	year_id	date_game	seasongame	is_playoffs	team_id	fran_id	pts	elo_i	elo_n	win_equiv	opp_id
113656	56829	201010260BOS	NBA	1	2011	10/26/2010	1	0	MIA	Heat	80	1547.36	1543.16	45.14	BOS
113657	56829	201010260BOS	NBA	0	2011	10/26/2010	1	0	BOS	Celtics	88	1625.10	1629.30	53.75	MIA
113658	56830	201010260LAL	NBA	1	2011	10/26/2010	1	0	HOU	Rockets	110	1504.20	1502.60	40.90	LAL
113659	56830	201010260LAL	NBA	0	2011	10/26/2010	1	0	LAL	Lakers	112	1647.60	1649.20	55.61	HOU
113660	56831	201010260POR	NBA	1	2011	10/26/2010	1	0	PHO	Suns	92	1643.02	1630.62	53.88	POR
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
126309	63155	201506110CLE	NBA	0	2015	6/11/2015	100	1	CLE	Cavaliers	82	1723.41	1704.39	60.31	GSW
126310	63156	201506140GSW	NBA	0	2015	6/14/2015	102	1	GSW	Warriors	104	1809.98	1813.63	68.01	CLE

WEEK 1

Pandas: Basics

Querying Your Dataset

You can also select the rows where a specific field is not null:

```
games_with_notes = df[df["notes"].notnull()]
games_with_notes.shape
```

```
(5424, 23)
```

This can be helpful if you want to avoid any missing values in a column. You can also use .notna() to achieve the same goal.

You can even access values of the object data type as str and perform string methods on them:

```
ers = df[df["fran_id"].str.endswith("ers")]
```

```
ers.shape
```

```
(27797, 23)
```

You use .str.endswith() to filter your dataset and find all games where the home team's name ends with "ers".

WEEK 1

Pandas: Basics

Grouping and Aggregating Your Data

You may also want to learn other features of your dataset, like the sum, mean, or average value of a group of elements. Luckily, the Pandas Python library offers grouping and aggregation functions to help you accomplish this task. For example:

```
[ ]  df.groupby("fran_id", sort=False)["pts"].sum()

     fran_id
     Huskies            3995
     Knicks           582497
     Stags             20398
     Falcons            3797
     Capitols          22387
     Celtics          626484
     Steamrollers      12372
     Ironmen            3674
     Bombers           17793
     Rebels             4474
     Warriors         591224
     Baltimore         37219
     Jets               4482
     Pistons          572758
```

By default, Pandas sorts the group keys during the call to .groupby(). If you don't want to sort, then pass sort=False. This parameter can lead to performance gains.

WEEK 1

Pandas: Basics

Grouping and Aggregating Your Data

You can also group by multiple columns:

```
[ ]  df[
         (df["fran_id"] == "Spurs") &
         (df["year_id"] > 2010)
     ].groupby(["year_id", "game_result"])["game_id"].count()

     year_id  game_result
     2011     L              25
              W              63
     2012     L              20
              W              60
     2013     L              30
              W              73
     2014     L              27
              W              78
     2015     L              31
              W              58
     Name: game_id, dtype: int64
```

WEEK 1

Pandas: Basics

Manipulating Columns

You can define new columns based on the existing ones:

```
nba["difference"] = df.pts - df.opp_pts
nba
```

team_id	fran_id	pts	win_equiv	opp_id	opp_fran	opp_pts	game_location	game_result	forecast	notes	difference
TRH	Huskies	66	40.29	NYK	Knicks	68	H	L	0.64	NaN	-2
NYK	Knicks	68	41.71	TRH	Huskies	66	A	W	0.36	NaN	2
CHS	Stags	63	42.01	NYK	Knicks	47	H	W	0.63	NaN	16

WEEK 1

Pandas: Basics

Manipulating Columns

You can also rename the columns of your dataset. It seems that "game_result" and "game_location" are too verbose, so go ahead and rename them now:

```
renamed_nba = nba.rename(
    columns={"game_result": "result", "game_location": "location"}
)
renamed_nba.head()
```

You can delete the four columns related to Elo:

```
elo_columns = ["elo_i", "elo_n", "opp_elo_i", "opp_elo_n"]
nba.drop(elo_columns, inplace=True, axis=1)
nba.shape
```

WEEK 1

Pandas: Basics

Missing Values

Sometimes, the easiest way to deal with records containing missing values is to ignore them. You can remove all the rows with missing values using .dropna():

```
[ ]  rows_without_missing_data = nba.dropna()

[ ]  rows_without_missing_data.shape

     (5424, 20)
```

Of course, this kind of data cleanup doesn't make sense for your nba dataset, because it's not a problem for a game to lack notes. You can also drop problematic columns if they're not relevant for your analysis. To do this, use .dropna() again and provide the axis=1 parameter:

data_without_missing_columns = nba.dropna(axis=1)

data_without_missing_columns.shape

WEEK 1

Pandas: Basics

Missing Values

If there's a meaningful default value for your use case, then you can also replace the missing values with that:

```
[ ]  data_with_default_notes = nba.copy()
```

```
[ ]  data_with_default_notes["notes"].fillna(
         value="no notes at all",
         inplace=True
     )
```

```
[ ]  data_with_default_notes["notes"].describe()

     count                126314
     unique                  232
     top        no notes at all
     freq                 120890
     Name: notes, dtype: object
```

WEEK 1

Pandas: Basics

Invalid Values

Invalid values can be even more dangerous than missing values. Often, you can perform your data analysis as expected, but the results you get are peculiar. Invalid values are often more challenging to detect, but you can implement some sanity checks with queries and aggregations.

One thing you can do is validate the ranges of your data. For this, .describe() is quite handy.
Recall that it returns the following output:
nba.describe()

The year_id varies between 1947 and 2015. That sounds plausible.
What about pts? How can the minimum be 0? Let's have a look at those games:
nba[nba["pts"] == 0]

It seems the game was forfeited. Depending on your analysis, you may want to remove it from the dataset.

WEEK 1

Pandas: Basics

Visualizing Your Pandas DataFrame

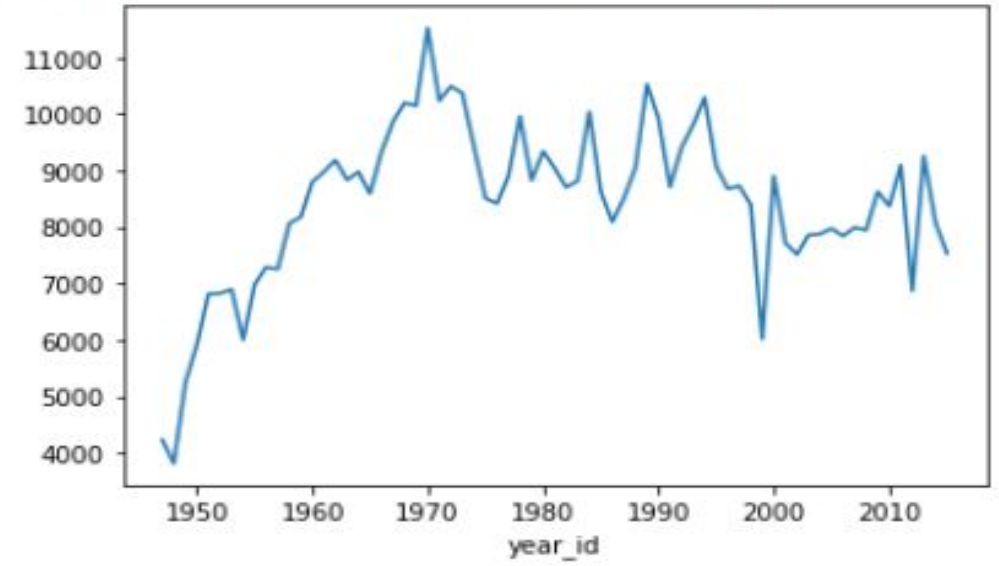
Data visualization is one of the things that works much better in a Jupyter notebook than in a terminal:

%matplotlib inline

Both Series and DataFrame objects have a .plot() method, which is a wrapper around matplotlib.pyplot.plot(). Visualize how many points the Knicks scored throughout the seasons:

```
nba[nba["fran_id"] == "Knicks"].groupby("year_id")["pts"].sum().plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x129e72790>
```



WEEK 1

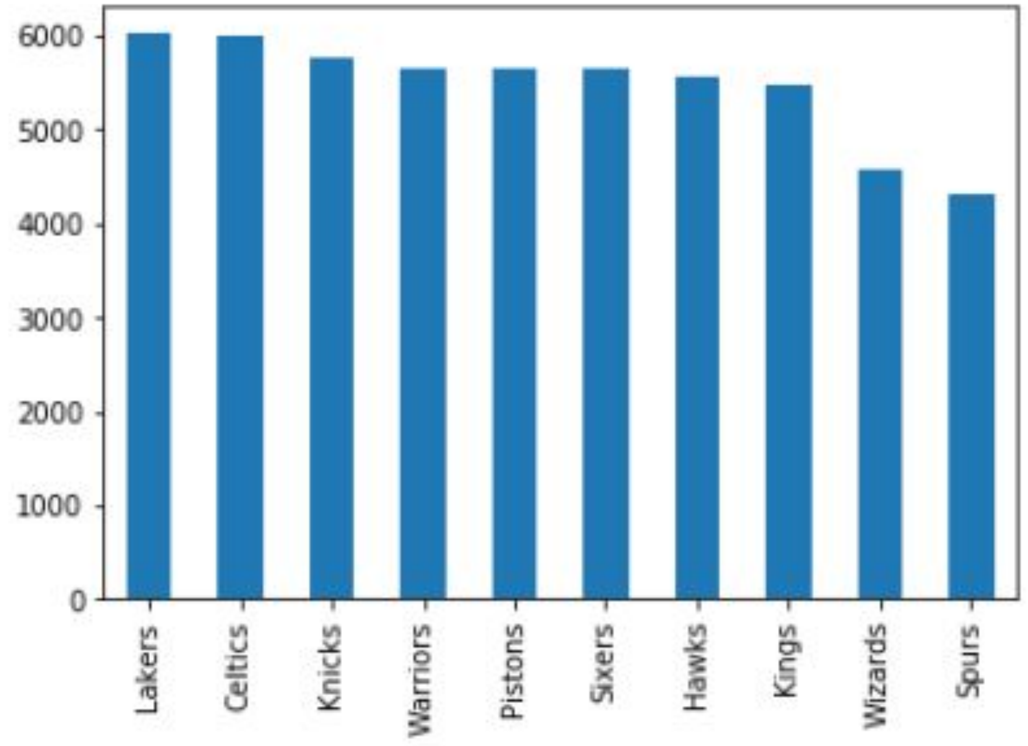
Pandas: Basics

Visualizing Your Pandas DataFrame

You can also create other types of plots, like a bar plot:

```
nba["fran_id"].value_counts().head(10).plot(kind="bar")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x12d4b3710>
```



WEEK 1

Pandas: Data Cleaning

Data Cleaning with Pandas

According to IBM Data Analytics you can expect to spend up to 80% of your time cleaning data. Before we dive into code, it's important to understand the sources of missing data. Here's some typical reasons why data is missing:

- **User forgot to fill in a field.**
- **Data was lost while transferring manually from a legacy database.**
- **There was a programming error.**
- **Users chose not to fill out a field tied to their beliefs about how the results would be used or interpreted.**

As you can see, some of these sources are just simple random mistakes. Other times, there can be a deeper reason why data is missing.

WEEK 1

Pandas: Data Cleaning

Data Cleaning with Pandas

The data we're going to work with is a very small. Here's a quick look at the data:

```
import numpy as np
import pandas as pd
```

```
df = pd.read_csv('property_data.csv')
```

```
df.head(10)
```

	PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	100001000.0	104.0	PUTNAM	Y	3	1	1000
1	100002000.0	197.0	LEXINGTON	N	3	1.5	--
2	100003000.0	NaN	LEXINGTON	N	NaN	1	850
3	100004000.0	201.0	BERKELEY	12	1	NaN	700
4	NaN	203.0	BERKELEY	Y	3	2	1600
5	100006000.0	207.0	BERKELEY	Y	NaN	1	800
6	100007000.0	NaN	WASHINGTON	NaN	2	HURLEY	950
7	100008000.0	213.0	TREMONT	Y	--	1	NaN
8	100009000.0	215.0	TREMONT	Y	na	2	1800

WEEK 1

Pandas: Data Cleaning

Standard Missing Values

Taking a look at the column, we can see that Pandas filled in the blank space with “NaN”. Using the isnull() method, we can confirm that both the missing value and “NaN” were recognized as missing values. Both boolean responses are True.

```
df['ST_NUM']

0    104.0
1    197.0
2      NaN
3    201.0
4    203.0
5    207.0
6      NaN
7    213.0
8    215.0
Name: ST_NUM, dtype: float64
```

```
[ ]  df['ST_NUM'].isnull()

0    False
1    False
2     True
3    False
4    False
5    False
6     True
7    False
8    False
Name: ST_NUM, dtype: bool
```

WEEK 1

Pandas: Data Cleaning

Non-Standard Missing Values

Sometimes it might be the case where there's missing values that have different formats. From the previous section, we know that Pandas will recognize "NA" as a missing value, but what about the others? Let's take a look.

```
[ ] missing_values = ["n/a", "na", "--"]

[ ] df = pd.read_csv("property_data.csv", na_values = missing_values)

df['NUM_BEDROOMS']

0    3.0
1    3.0
2    NaN
3    1.0
4    3.0
5    NaN
6    2.0
7    NaN
8    NaN
Name: NUM_BEDROOMS, dtype: float64
```

```
df['NUM_BEDROOMS'].isnull()

0    False
1    False
2     True
3    False
4    False
5     True
6    False
7     True
8     True
Name: NUM_BEDROOMS, dtype: bool
```

WEEK 1

Pandas: Data Cleaning

Unexpected Missing Values

From our previous examples, we know that Pandas will detect the empty cell in row seven as a missing value. Let's confirm with some code.

```
df['OWN_OCCUPIED']
```

```
0      Y
1      N
2      N
3     12
4      Y
5      Y
6    NaN
7      Y
8      Y
Name: OWN_OCCUPIED, dtype: object
```

In the fourth row, there's the number 12. The response for Owner Occupied should clearly be a string (Y or N), so this numeric type should be a missing value.

WEEK 1

Pandas: Data Cleaning

Unexpected Missing Values

This example is a little more complicated so we'll need to think through a strategy for detecting these types of missing values. There's a number of different approaches, but here's the way that I'm going to work through this one.

- Loop through the OWN_OCCUPIED column
- Try and turn the entry into an integer
- If the entry can be changed into an integer, enter a missing value
- If the number can't be an integer, we know it's a string, so keep going

WEEK 1

Pandas: Data Cleaning

Unexpected Missing Values

In the code we're looping through each entry in the "Owner Occupied" column. To try and change the entry to an integer, we're using int(row). If the value can be changed to an integer, we change the entry to a missing value using Numpy's np.nan.On the other hand, if it can't be changed to an integer, we pass and keep going. You'll notice that we used try and except ValueError.

```
cnt=0
for row in df['OWN_OCCUPIED']:
    try:
        int(row)
        df.loc[cnt, 'OWN_OCCUPIED']=np.nan
    except ValueError:
        pass
    cnt+=1
```

```
df.head(9)
```

	PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	100001000.0	104.0	PUTNAM	Y	3.0	1	1000.0
1	100002000.0	197.0	LEXINGTON	N	3.0	1.5	NaN
2	100003000.0	NaN	LEXINGTON	N	NaN	1	850.0
3	100004000.0	201.0	BERKELEY	NaN	1.0	NaN	700.0
4	NaN	203.0	BERKELEY	Y	3.0	2	1600.0
5	100006000.0	207.0	BERKELEY	Y	NaN	1	800.0
6	100007000.0	NaN	WASHINGTON	NaN	2.0	HURLEY	950.0
7	100008000.0	213.0	TREMONT	Y	NaN	1	NaN
8	100009000.0	215.0	TREMONT	Y	NaN	2	1800.0

WEEK 1

Pandas: Data Cleaning

Summarizing Missing Values

After we've cleaned the missing values, we will probably want to summarize them. For instance, we might want to look at the total number of missing values for each feature.

```
df.isnull().sum()

PID             1
ST_NUM          2
ST_NAME         0
OWN_OCCUPIED    2
NUM_BEDROOMS    4
NUM_BATH        1
SQ_FT           2
dtype: int64
```

Other times we might want to do a quick check to see if we have any missing values at all.

```
df.isnull().values.any()

True
```

We might also want to get a total count of missing values.

```
df.isnull().sum().sum()

12
```

WEEK 1

Pandas: Data Cleaning

Replacing

Often times you'll have to figure out how you want to handle missing values.

Sometimes you'll simply want to delete those rows, other times you'll replace them.

```
# Maybe you just want to fill in missing values with a single value.
df['ST_NUM'].fillna(125, inplace=True)

# you might want to do a location based imputation. Here's how you would do that.
df.loc[2,'ST_NUM'] = 125

#A very common way to replace missing values is using a median.
median = df['NUM_BEDROOMS'].median()
df['NUM_BEDROOMS'].fillna(median, inplace=True)
```

Hacktiv8 DS
Curriculum
Team

External References

Colab Link	Visit Here
Colab Link (NBA Analysis)	Visit Here