

# DataFrames, Indexing, Filtering, Selecting

UBCO Master of Data Science – DATA 542

Fatemeh Fard



# Lecture 1 - review

---

Data wrangling importance

Python dynamic data type vs static data types in languages like C

Overhead of python data types

More efficient data types in Python packages: NumPy and Pandas

NumPy ndarrays

NumPy array creation:

- From python lists and arrays
- From scratch with functions: zeros, ones, full, arrange, linspace, random, eye, empty
- Define data types and array dimensions and size in the functions
- Computations on arrays

# Lecture 2 learning outcomes

---

At the end of this lecture you should be able to:

- Understand the DataFrames and Series objects
- Perform functions on DataFrames and Series to index and select data
- Perform functions on DataFrames and Series to sub-set data (slicing)

# Why Pandas?

---

NumPy data structure (ndarray) is good for computations with:

- Numerical values
- Clean data
- No-missing values

NumPy limitations:

- Categorical data
- Work with labels (attach labels to data)
- Work with missing values
- Heterogeneous data
- Operations that do not map on each element (e.g. grouping, pivot)

# Pandas

---

Built on NumPy

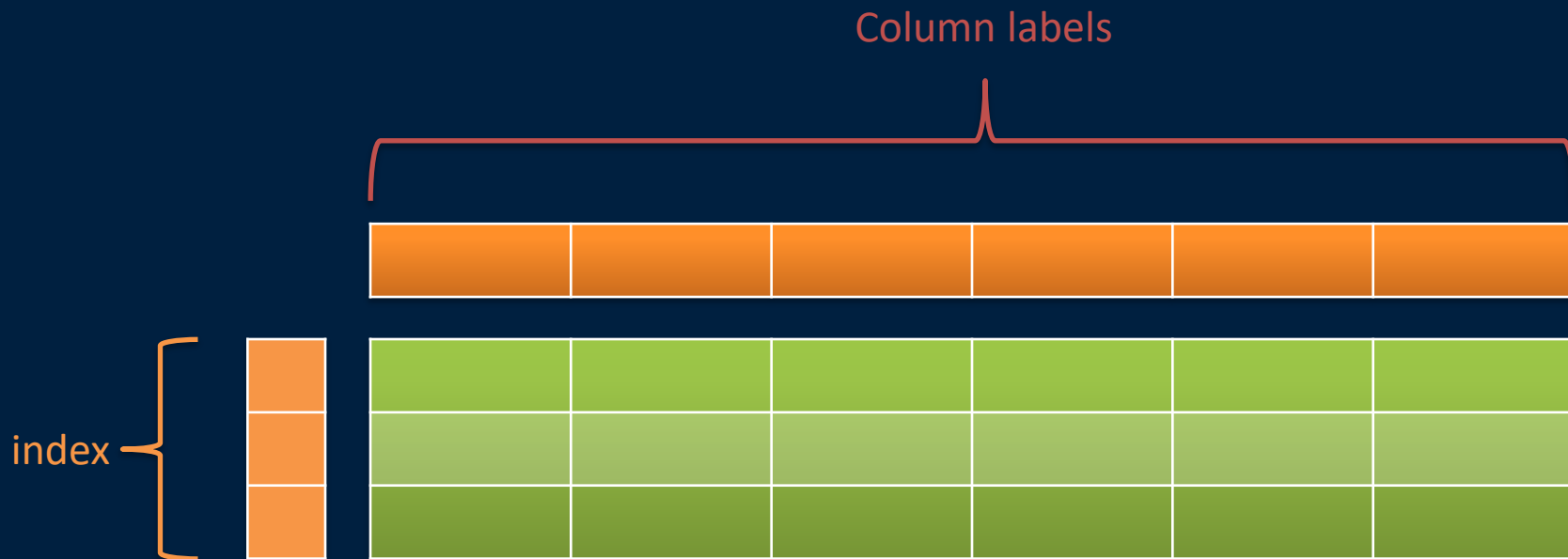
Data structures:

- Series
- DataFrames

Rectangular grids with labels for rows and columns instead of integer indices

Powerful operations on data

# DataFrames



# Pandas vs NumPy

---

## Pandas vs NumPy

# Pandas installation

---

Refer to Pandas website for installation and documents.

- <http://pandas.pydata.org>

Install NumPy first

Run

```
conda install pandas
```

Or

```
python3 -m pip install --upgrade pandas
```



# Import Pandas

---

```
import pandas as pd
```

Access built-in documentation

```
pd.<TAB>
```

```
pd.?
```

# Anatomy of a DataFrame and a Series

The diagram illustrates the structure of a DataFrame with the following labels and annotations:

- columns axis=1**: A green box highlighting the column headers.
- column name**: An arrow pointing to the `director_name` header.
- more columns to display**: An arrow pointing to the ellipsis (`...`) between `duration` and `actor_2_facebook_likes`.
- index label**: An arrow pointing to the index values (0, 1, 2, 3, 4).
- index axis=0**: A green box highlighting the index values.
- missing values**: An arrow pointing to the `NaN` value in the `director_name` column for index 4.
- data (values)**: An arrow pointing to the numerical value `12.0` in the `actor_2_facebook_likes` column for index 4.

	color	director_name	num_critic_for_reviews	duration	...	actor_2_facebook_likes	imdb_score	aspect_ratio	movie_facebook_likes
0	Color	James Cameron	723.0	178.0	...	936.0	7.9	1.78	33000
1	Color	Gore Verbinski	302.0	169.0	...	5000.0	7.1	2.35	0
2	Color	Sam Mendes	602.0	148.0	...	393.0	6.8	2.35	85000
3	Color	Christopher Nolan	813.0	164.0	...	23000.0	8.5	2.35	164000
4	NaN	Doug Walker	NaN	NaN	...	12.0	7.1	NaN	0

Source: <https://medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c>

# Pandas Series

One-dimensional array of indexed data. More general and flexible than NumPy array.

It can be created from a list or array.

```
data = pd.Series([25., 50., 75., 100])
```

```
data
```

Output:

```
0    25.0
1    50.0
2    75.0
3   100.0
4  dtype: float64
```

Attributes:

- values
- index

# Pandas Series cont.

Numpy Array defines integer index *implicitly*

Pandas Series defines index *explicitly* that is associated with the values

```
data = pd.Series( [25., 50., 75., 100] 
```

values

```
index=[ 'a', 'b', 'c', 'd' ]
)
```

index

```
data[ 'b' ]
```

Output: 50.0

# Pandas Series as dictionaries

Specialized form of dictionary, more efficient than python dictionary

```
population_dict = {'California': 38332521,
                  'Texas': 26448193,
                  'New York': 19651127,
                  'Florida': 19552860}
```

**Try it**

```
population = pd.Series( population_dict )
```

```
population['California']
```

```
Output: 38332521
```

↑  
Pass a dictionary

# Anatomy of a DataFrame and a Series

The diagram illustrates the structure of a DataFrame with the following labels and annotations:

- columns axis=1**: A green box highlighting the column headers.
- column name**: An arrow pointing to the `director_name` header.
- more columns to display**: An arrow pointing to the ellipsis (`...`) between `duration` and `actor_2_facebook_likes`.
- index label**: An arrow pointing to the index values (0, 1, 2, 3, 4) on the left.
- index axis=0**: A green box highlighting the index values.
- missing values**: An arrow pointing to the `NaN` values in the `director_name` and `num_critic_for_reviews` columns for index 4.
- data (values)**: An arrow pointing to the data values in the `actor_2_facebook_likes` column for index 4.

	color	director_name	num_critic_for_reviews	duration	...	actor_2_facebook_likes	imdb_score	aspect_ratio	movie_facebook_likes
0	Color	James Cameron	723.0	178.0	...	936.0	7.9	1.78	33000
1	Color	Gore Verbinski	302.0	169.0	...	5000.0	7.1	2.35	0
2	Color	Sam Mendes	602.0	148.0	...	393.0	6.8	2.35	85000
3	Color	Christopher Nolan	813.0	164.0	...	23000.0	8.5	2.35	164000
4	NaN	Doug Walker	NaN	NaN	...	12.0	7.1	NaN	0

Source: <https://medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c>

# Pandas DataFrames (1 of 3)

Data structures as aligned Pandas Series: they share same indices

```
population = pd.Series({'California': 38332521,
                        'Florida': 19552860,
                        'Illinois': 12882135,
                        'New York': 19651127,
                        'Texas': 26448193})
```

California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

# Pandas DataFrames (2 of 3)

```
area = pd.Series({'California': 423967,
                  'Texas': 695662,
                  'New York': 141297,
                  'Florida': 170312,
                  'Illinois': 149995})
```

California	423967
Texas	695662
New York	141297
Florida	170312
Illinois	149995



# Pandas DataFrames (3 of 3)

```
states = pd.DataFrame({'population_':  
population, 'area_': area})
```

	population_	area_
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

# Try IT

---

Create a dataframe called States

1- Using Series:

```
states = pd.DataFrame({'population_':  
population, 'area_': area})
```

2- Reading from a CSV (next slide)

# DataFrames from csv files

```
df = pd.read_csv("US-population-areas.csv",
index_col = 0)
```

Pass the first column as the labels (row index)



Index Labels  
you want to  
work with

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

# Indexing and Slicing

---

# Why indexing/sub setting

Data values  
from one  
row

	department	age	height	food	color
<b>Jane</b>	biology	32	160	steak	blue
<b>Sara</b>	chemistry	40	158	lamb	red
<b>Nicole</b>	biology	35	170	apple	orange
<b>Kaden</b>	computer	50	180	cheese	yellow
<b>Jeff</b>	statistics	35	175	steak	blue
<b>Reza</b>	chemistry	45	165	lamb	red
<b>John</b>	statistics	45	162	apple	orange
<b>Ramon</b>	computer	40	175	cheese	yellow
<b>Bryce</b>	engineering	28	180	steak	blue

# Why indexing/sub setting: Slicing rows

Data values  
for everyone  
who is in  
biology

	department	age	height	food	color
<b>Jane</b>	biology	32	160	steak	blue
<b>Sara</b>	chemistry	40	158	lamb	red
<b>Nicole</b>	biology	35	170	apple	orange
<b>Kaden</b>	computer	50	180	cheese	yellow
<b>Jeff</b>	statistics	35	175	steak	blue
<b>Reza</b>	chemistry	45	165	lamb	red
<b>John</b>	statistics	45	162	apple	orange
<b>Ramon</b>	computer	40	175	cheese	yellow
<b>Bryce</b>	engineering	28	180	steak	blue

# Why indexing/sub setting: Indexing columns

Data values for one or multiple columns. For example department and/or color for all samples

	department	age	height	food	color
<b>Jane</b>	biology	32	160	steak	blue
<b>Sara</b>	chemistry	40	158	lamb	red
<b>Nicole</b>	biology	35	170	apple	orange
<b>Kaden</b>	computer	50	180	cheese	yellow
<b>Jeff</b>	statistics	35	175	steak	blue
<b>Reza</b>	chemistry	45	165	lamb	red
<b>John</b>	statistics	45	162	apple	orange
<b>Ramon</b>	computer	40	175	cheese	yellow
<b>Bryce</b>	engineering	28	180	steak	blue

# Why indexing/sub setting: selected rows and columns

Data values for selected rows and columns. For example: everyone is chemistry and his favorite color

	department	age	height	food	color
Jane	biology	32	160	steak	blue
Sara	chemistry	40	158	lamb	red
Nicole	biology	35	170	apple	orange
Kaden	computer	50	180	cheese	yellow
Jeff	statistics	35	175	steak	blue
Reza	chemistry	45	165	lamb	red
John	statistics	45	162	apple	orange
Ramon	computer	40	175	cheese	yellow
Bryce	engineering	28	180	steak	blue



# Why indexing/sub setting: Filtering columns based on a condition



Data values  
for everyone  
who is  
shorter than  
160

	department	age	height	food	color
Jane	biology	32	160	steak	blue
Sara	chemistry	40	158	lamb	red
Nicole	biology	35	170	apple	orange
Kaden	computer	50	180	cheese	yellow
Jeff	statistics	35	175	steak	blue
Reza	chemistry	45	165	lamb	red
John	statistics	45	162	apple	orange
Ramon	computer	40	175	cheese	yellow
Bryce	engineering	28	180	steak	blue

# Slicing Series

## Slicing with array-style operations

```
population_dict = {'California': 38332521, 'Texas': 26448193,
                   'New York': 19651127, 'Florida': 19552860}
```

```
population = pd.Series( population_dict )
```

```
population['California': 'New York']
```

Output:

```
California 38332521
```

```
Texas 26448193
```

```
New York 19651127
```

```
dtype: int64
```

Inclusive



# Pandas DataFrames

DataFrames as specialized dictionaries

Dictionary: Maps Key to Value

DataFrame: Maps Column name to Series



```
States[ 'area' ]
```

```
Output:  California  423967
         Texas      695662
         New York   141297
         Florida    170312
         Illinois    149995
         Name: area, dtype: int64
```

# Try IT

---

1- Slice series

2- Slice Dataframe

Use the previous 2 slides

# Indexing/sub-setting operations syntax

---

[ ]

loc [ starting row label : **inclusive** ending row label ,  
starting column label : **inclusive** ending column label ]

iloc [ starting row number : **exclusive** ending row number ,  
starting column number : **exclusive** ending column number ]

df[ ]

df.loc[ ]

df.iloc[ ]

# Indexers: loc

Use indexing attributes

**loc**: allows indexing and slicing that always references the **explicit** index: **LOCATION (Labels)**

```
data.loc[1]
```

Output: 'a'

```
data.loc[1:3]
```

Output:

1 a

3 b

```
data = pd.Series(['a', 'b', 'c'],
index=[1, 3, 5])
```

1	a
3	b
5	c

# Indexers: iloc

**iloc**: allows indexing and slicing that always references the **implicit** index: **integer indexing**

```
data.iloc[1]
```

Output: 'b'

```
data.iloc[1:3]
```

Output:

3 b

5 c


```
data = pd.Series(['a', 'b', 'c'],  
index=[1, 3, 5])
```

1	a
3	b
5	c

# Indexing/sub-setting operations

[ ]


loc [ starting row : **inclusive** ending row , starting column : **inclusive**  
ending column ]



If did not mention, means start from the first one until the ending row/column



iloc [ starting row : **exclusive** ending row , starting column : **exclusive**  
ending column ]



If did not mention, means start from the starting row/column until the end  
THE SAME APPLIES FOR ENDING IN THE LOC OPERATION



# Data selection in DataFrames

Dictionary-style:

```
data[ 'area' ]
```



Column name

Attribute-style:

```
data.area
```



Column name

	population	area
<b>California</b>	38332521	423967
<b>Texas</b>	26448193	695662
<b>New York</b>	19651127	141297
<b>Florida</b>	19552860	170312
<b>Illinois</b>	12882135	149995

# Data selection in DataFrames cont.

---

Attribute-style does not work for:

- Column name is not string
- Column name conflicts with DataFrame functions (such as `pop()`)

`data.area` **is** `data['area']` output: True

`data.pop` **is** `data['pop']` output: False

Do **NOT** use attribute style for assignment in DataFrames

`pop(item)`: returns an item from DataFrame

# Data selection in DataFrames cont.

```
data['density'] =  
data['population']/data['area']
```

	population	area	density
<b>California</b>	38332521	423967	90.413926
<b>Texas</b>	26448193	695662	38.018740
<b>New York</b>	19651127	141297	139.076746
<b>Florida</b>	19552860	170312	114.806121
<b>Illinois</b>	12882135	149995	85.883763

# More Examples



# Indexers: loc

Use indexing attributes

**loc**: allows indexing and slicing that always references the **explicit** index: **LOCATION (Labels)**

```
data.loc[1]
```

Output: 'a'

```
data.loc[1:3]
```

Output:

1 a

3 b

```
data = pd.Series(['a', 'b', 'c'],
index=[1, 3, 5])
```

1	a
3	b
5	c

# Indexers: iloc

**iloc**: allows indexing and slicing that always references the **implicit** index: **integer indexing**

```
data.iloc[1]
```

Output: 'b'

```
data.iloc[1:3]
```

Output:

3 b

5 c

```
data = pd.Series(['a', 'b', 'c'],  
index=[1, 3, 5])
```

1	a
3	b
5	c

# Data selection in DataFrames

Dictionary-style:

```
data[ 'area' ]
```



Column name

Attribute-style:

```
data.area
```



Column name

	population	area
<b>California</b>	38332521	423967
<b>Texas</b>	26448193	695662
<b>New York</b>	19651127	141297
<b>Florida</b>	19552860	170312
<b>Illinois</b>	12882135	149995

# Data selection in DataFrames cont.

---

Attribute-style does not work for:

- Column name is not string
- Column name conflicts with DataFrame functions (such as `pop()`)

`data.area` **is** `data['area']` output: True

`data.pop` **is** `data['pop']` output: False

Do **NOT** use attribute style for assignment in DataFrames

`pop(item)`: returns an item from DataFrame



# Data selection in DataFrames cont.

```
data['density'] =  
data['population']/data['area']
```

	population	area	density
<b>California</b>	38332521	423967	90.413926
<b>Texas</b>	26448193	695662	38.018740
<b>New York</b>	19651127	141297	139.076746
<b>Florida</b>	19552860	170312	114.806121
<b>Illinois</b>	12882135	149995	85.883763

# Data selection in DataFrames cont.

Array-style:

```
data.values
```

```
array([[3.83325210e+07, 4.23967000e+05, 9.04139261e+01],  
       [2.64481930e+07, 6.95662000e+05, 3.80187404e+01],  
       [1.96511270e+07, 1.41297000e+05, 1.39076746e+02],  
       [1.95528600e+07, 1.70312000e+05, 1.14806121e+02],  
       [1.28821350e+07, 1.49995000e+05, 8.58837628e+01]])
```

```
data.values[0]
```

```
array([3.83325210e+07, 4.23967000e+05, 9.04139261e+01])
```

# Data selection in DataFrames cont.

Array-style:

`data.T`      `#Transpose`

	California	Texas	New York	Florida	Illinois
<b>population</b>	3.833252e+07	2.644819e+07	1.965113e+07	1.955286e+07	1.288214e+07
<b>area</b>	4.239670e+05	6.956620e+05	1.412970e+05	1.703120e+05	1.499950e+05
<b>density</b>	9.041393e+01	3.801874e+01	1.390767e+02	1.148061e+02	8.588376e+01

# Data selection in DataFrames cont.

iloc and loc:

```
data.iloc[:4, :2]
```

	population	area
<b>California</b>	38332521	423967
<b>Texas</b>	26448193	695662
<b>New York</b>	19651127	141297
<b>Florida</b>	19552860	170312

```
Data.loc['California':'New York', 'area':]
```

	area	density
<b>California</b>	423967	90.413926
<b>Texas</b>	695662	38.018740
<b>New York</b>	141297	139.076746

# Data selection in DataFrames cont.

Combining NumPy-style data access patterns with indexers in DataFrames:

```
# combine masking and fancy indexing
data.loc[data.density > 100, ['population', 'area']]
```

**Discuss the  
output**

	population	area	density
<b>California</b>	38332521	423967	90.413926
<b>Texas</b>	26448193	695662	38.018740
<b>New York</b>	19651127	141297	139.076746
<b>Florida</b>	19552860	170312	114.806121
<b>Illinois</b>	12882135	149995	85.883763

# Data selection in DataFrames cont.

Combining NumPy-style data access patterns with indexers in DataFrames:

```
data.iloc[0,2]=90
```

	population	area	density
<b>California</b>	38332521	423967	90.000000
<b>Texas</b>	26448193	695662	38.018740
<b>New York</b>	19651127	141297	139.076746
<b>Florida</b>	19552860	170312	114.806121
<b>Illinois</b>	12882135	149995	85.883763

# DataFrame Slicing: sub- setting rows

`data[1:3]`

	population	area	density
<b>Texas</b>	26448193	695662	38.018740
<b>New York</b>	19651127	141297	139.076746

`data['Texas': 'Florida']`

	population	area	density
<b>Texas</b>	26448193	695662	38.018740
<b>New York</b>	19651127	141297	139.076746
<b>Florida</b>	19552860	170312	114.806121

# DataFrame Slicing: sub- setting rows through masking

Masking is a row-wise operation:

```
data[(80 < data.density) & (data.density < 115)]
```

	population	area	density
<b>California</b>	38332521	423967	90.000000
<b>Florida</b>	19552860	170312	114.806121
<b>Illinois</b>	12882135	149995	85.883763



# In Practice More on Series and Dataframes



# Pandas Series cont.

---

Attributes:

- `values`
- `index`

Values are NumPy arrays.

```
data.values
```

```
Output: array([ 25.,  50.,  75., 100.])
```

# Pandas Series cont.

---

index is an array-like object of type `pd.Index`.

See index:

```
data.index
```

```
Output: RangeIndex(start=0, stop=4, step=1)
```

Or access by [] :

```
Data[1:3]
```

```
Output:      1  50.0  
        2  75.0  
        dtype: float64
```

# Pandas Series cont.

Numpy Array defines integer index *implicitly*

Pandas Series defines index *explicitly* that is associated with the values

```
data = pd.Series( [25., 50., 75., 100] 
```

values

```
index=[ 'a', 'b', 'c', 'd' ]
)
```

index

```
data[ 'b' ]
```

Output: 50.0

# Pandas Series cont.

```
data = pd.Series([25., 50., 75., 100],
                  index=['a', 'b', 'c', 'd'])
```

```
data = pd.Series([25., 50., 75., 100],
                  index=[7, 2, 3, 6])
```

```
data = pd.Series([25., 50., 75., 100],
                  index=['a', 5, 'b', 7])
```

Not desirable for  
operation : when  
slicing

# Pandas Series as dictionaries

Specialized form of dictionary, more efficient than python dictionary

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860}
```

**Try it**

```
population = pd.Series( population_dict )
population['California']
```

Output: 38332521

↑  
Pass a dictionary

# Pandas Series as dictionaries cont.

## Slicing with array-style operations

```
population_dict = {'California': 38332521, 'Texas': 26448193,
                   'New York': 19651127, 'Florida': 19552860}
```

```
population = pd.Series( population_dict )
```

```
population['California': 'New York']
```

Output:

```
California 38332521
```

```
Texas 26448193
```

```
New York 19651127
```

```
dtype: int64
```

Inclusive

# Constructing Series objects

```
pd.Series(data, index=index)
```

Data can be:

optional

## 1- List or NumPy array

```
pd.Series([2, 4, 6])
```

## 2- Scalar

```
pd.Series(5, index=[100, 200, 300])
```

## 3- dictionary

```
pd.Series({2:'a', 1:'b', 3:'c'})
```



# Constructing Series objects cont.

## 4- explicitly set index to have a different result

```
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```



Output:

```
3 c
2 a
dtype: object
```

Desired indices  
(Explicitly identified keys)

# Question

**Question 1:** What is the result of executing this code:

```
my_list = [2, 3, '2', 5, 'me']
index = ['i', 'j', 'k', 2, 3]
data = pd.Series(my_list, index)
```

**A)** Value error

**B)** Type error

**C)**

i	2
j	3
k	2
2	5
3	me

**D)**

0	2
1	3
2	2
3	5
4	me

# Anatomy of a DataFrame and a Series

The diagram illustrates the structure of a DataFrame with the following labels and annotations:

- columns axis=1**: A green box highlighting the column headers.
- column name**: An arrow pointing to the `director_name` header.
- more columns to display**: An arrow pointing to the ellipsis (`...`) between `duration` and `actor_2_facebook_likes`.
- index label**: An arrow pointing to the index values (0, 1, 2, 3, 4).
- index axis=0**: A green box highlighting the index values.
- missing values**: Arrows pointing to the `NaN` values in the `color` and `num_critic_for_reviews` columns for index 4.
- data (values)**: An arrow pointing to the data values in the `actor_2_facebook_likes` column.

	color	director_name	num_critic_for_reviews	duration	...	actor_2_facebook_likes	imdb_score	aspect_ratio	movie_facebook_likes
0	Color	James Cameron	723.0	178.0	...	936.0	7.9	1.78	33000
1	Color	Gore Verbinski	302.0	169.0	...	5000.0	7.1	2.35	0
2	Color	Sam Mendes	602.0	148.0	...	393.0	6.8	2.35	85000
3	Color	Christopher Nolan	813.0	164.0	...	23000.0	8.5	2.35	164000
4	NaN	Doug Walker	NaN	NaN	...	12.0	7.1	NaN	0

Source: <https://medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c>

# Pandas DataFrames

Data structures as aligned Pandas Series: they share same indices

```
population = pd.Series({'California': 38332521,
                        'Florida': 19552860,
                        'Illinois': 12882135,
                        'New York': 19651127,
                        'Texas': 26448193})
```

California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

# Pandas DataFrames cont.

```
area = pd.Series({'California': 423967,
                  'Texas': 695662,
                  'New York': 141297,
                  'Florida': 170312,
                  'Illinois': 149995})
```

California	423967
Texas	695662
New York	141297
Florida	170312
Illinois	149995

# Pandas DataFrames cont.

```
states = pd.DataFrame({'population_':  
population, 'area_': area})
```

	population_	area_
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

# Pandas DataFrames cont.

Attributes: index , columns

`states.index`

Dot notation

Output:

```
Index(['California', 'Texas', 'New York',
       'Florida', 'Illinois'], dtype='object')
```

`states.columns`

Dot notation

Output:

```
Index(['population_', 'area_'], dtype='object')
```

# Creating DataFrame objects

---

## 1- From an single Series

```
pd.DataFrame(population, columns=['population'])
```



Optional, if no column name is chosen  
you will see integer index starting at 0

## 2- From a *list of dictionaries*

```
data = [{'a': i, 'b': 2 * i} for i in range(3)]  
pd.DataFrame(data)
```



# Question

**Question 1:** What is the result of the following code?

```
data = [{ 'a': i, 'b': 2 * i} for i in range(3)]
[{'a': 0, 'b': 0}, {'a': 1, 'b': 2}, {'a': 2, 'b': 4}]
```

**Question 2:** What is the result of the following code?

```
pd.DataFrame(data)
```

	a	b
0	0	0
1	1	2
2	2	4

# Creating DataFrame objects cont.

## 3- From a list of dictionaries, with missing keys

Pandas will fill the missing values with **NaN**

```
pd.DataFrame([{'a': 1, 'b': 2},
               {'b': 3, 'c': 4}])
```

*(Note: In the original image, the first dictionary is circled in green, and a red bracket labeled 'row' connects it to the second dictionary.)*

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

Null values are filled with NaN

# Creating DataFrame objects cont.

---

## 4- From a dictionary of Series objects

```
pd.DataFrame({'population': population,  
              'area': area})
```

# Creating DataFrame objects cont.

## 5- From a two-dimensional NumPy array

```
pd.DataFrame(np.random.rand(3, 2),
columns=['foo', 'bar'], index=['a', 'b', 'c'])
```



Optional, if column name and index is not defined,  
integer index will be used for each

	foo	bar
<b>a</b>	0.997026	0.985137
<b>b</b>	0.175123	0.834443
<b>c</b>	0.606030	0.725598

# Creating DataFrame objects cont.

## 6-Reading from a file

```
df = pd.read_csv("US-population-areas.csv")
```

↑ File Path

**Creates  
integer  
indexes**

	Unnamed: 0	population	area
0	California	38332521	423967
1	Texas	26448193	695662
2	New York	19651127	141297
3	Florida	19552860	170312
4	Illinois	12882135	149995

# DataFrames from csv files

```
df = pd.read_csv("US-population-areas.csv",
index_col = 0)
```

Pass the first column as the labels (row index)



Index Labels  
you want to  
work with

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

# Pandas DataFrames cont.

---

Attributes:

- `values`
- `index`
- `columns`

```
index = df.index
```

```
columns = df.columns
```

```
values = df.values
```

```
type(index)
```

```
type(columns)
```

```
type(values)
```

# Question

**Question 1:** What is the resulting dataframe?

```
pd.DataFrame([{'a': 1, 'b': 2},
               {'b': 3, 'c': 4}])
```

**A**

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

**B**

	a	b	c
0	1.0	2.0	NaN
1	NaN	NaN	4.0

**C**

	a	b
0	1	2
1	4	3



# Pandas DataFrames cont.

DataFrames as specialized dictionaries

Dictionary: Maps Key to Value

DataFrame: Maps Column name to Series



```
States[ 'area' ]
```

```
Output:   California  423967
        Texas      695662
        New York  141297
        Florida   170312
        Illinois   149995
        Name: area, dtype: int64
```

# Pandas index object

Like **immutable** arrays and ordered set in Python

Indexing, slicing, and NumPy array attributes

```
ind = pd.Index([2, 3, 5, 7, 11])
```

```
ind[1]
```

Output: 3

```
ind[::2]
```

Output: Int64Index([2, 5, 11], dtype='int64')

```
print(ind.size, ind.shape, ind.ndim, ind.dtype)
```

Output: 5 (5,) 1 int64

# Pandas index object cont.

## Unions, intersections, differences and other Set operations

```
indA = pd.Index([1, 3, 5, 7, 9])
```

```
indB = pd.Index([2, 3, 5, 7, 11])
```

```
indA & indB # intersection
```

```
indA.intersection(indB) ← Object methods
```

```
Output: Int64Index([3, 5, 7], dtype='int64')
```

```
indA | indB # union
```

```
indA ^ indB # symmetric difference
```

```
Output: Int64Index([1, 2, 9, 11],  
dtype='int64')
```

# Question

**Question 1:** Index object in Pandas is immutable. The following statement will give an error. True or False?

```
ind = pd.Index([2, 3, 5, 7, 11])
ind[2]=8
```

- 1) True
- 2) False

## Try it:

---

Create a Series of three of your family members.

Create a Series of their favorite fruit.

Create a Series of their age.

Create a DataFrame for your family members from the above Series.

Select the first two members and their ages.



# **In Practice More on Slicing and Indexing Series and Dataframes**



# Series

## Dictionary-like operations

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=['a', 'b', 'c', 'd'])
```

```
1- data['b']
```

```
2- 'a' in data
```

```
3- data.keys()
```

```
4- list(data.items())
```

```
5- data['e'] = 1.25
```

```
data
```

**Discuss the output**

# Series cont.

NumPy arrays operations: *slicing*

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=['a', 'b', 'c', 'd'])
```

**Discuss the output**

```
data['a':'c'] # slicing by explicit index
```

**Start**      **Inclusive End**



```
data[0:2] # slicing by implicit integer index
```

**Exclusive End**





# Series cont.

## Discuss the output

NumPy arrays operations: *masking*, and *fancy indexing*

```
data = pd.Series([0.25, 0.5, 0.75, 1.0, 1.25],
                 index=['a', 'b', 'c', 'd', 'e'])
```

Desired conditions

Combination operations

```
data[(data > 0.3) & (data < 0.8)] # masking
```

```
b    0.50
c    0.75
dtype: float64
```

# Series cont.

## Discuss the output

NumPy arrays operations: *masking*, and *fancy indexing*

```
data = pd.Series([0.25, 0.5, 0.75, 1.0, 1.25],
                 index=['a', 'b', 'c', 'd', 'e'])
```

```
data[['a', 'e']] # fancy indexing
```

**List of desired indicies**

```
a    0.25
e    1.25
dtype: float64
```

# Source of **CONFUSION** with integer indexing

## Try it yourself

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
```

```
# explicit index when indexing - Output: 'a'
```

```
data[1]
```

```
# implicit index when slicing -
```

```
data[1:3]
```

```
Output:
```

```
3 b
```

```
5 c dtype: object
```

# Indexers: loc

Use indexing attributes

**loc**: allows indexing and slicing that always references the **explicit** index: **LOCATION (Labels)**

```
data.loc[1]
```

Output: 'a'

```
data.loc[1:3]
```

Output:

1 a

3 b

```
data = pd.Series(['a', 'b', 'c'],
index=[1, 3, 5])
```

1	a
3	b
5	c

# Indexers: iloc

**iloc**: allows indexing and slicing that always references the **implicit** index: **integer indexing**

```
data.iloc[1]
```

Output: 'b'

```
data.iloc[1:3]
```

Output:

3 b

5 c

```
data = pd.Series(['a', 'b', 'c'],
index=[1, 3, 5])
```

1	a
3	b
5	c

# Data selection in DataFrames

Dictionary-style:

```
data[ 'area' ]
```



Column name

Attribute-style:

```
data.area
```



Column name

	population	area
<b>California</b>	38332521	423967
<b>Texas</b>	26448193	695662
<b>New York</b>	19651127	141297
<b>Florida</b>	19552860	170312
<b>Illinois</b>	12882135	149995

# Data selection in DataFrames cont.

---

Attribute-style does not work for:

- Column name is not string
- Column name conflicts with DataFrame functions (such as `pop()`)

`data.area` **is** `data['area']` output: True

`data.pop` **is** `data['pop']` output: False

Do **NOT** use attribute style for assignment in DataFrames

`pop(item)`: returns an item from DataFrame

## Try it:

---

Create a Series of three of your family members.

Create a Series of their favorite fruit.

Create a Series of their age.

Create a DataFrame for your family members from the above Series.

Select the first two members and their ages.





THE UNIVERSITY OF BRITISH COLUMBIA

