Index DATA SCIENCE

NAME: Ravi Teja. STD: SEC: ROLL:

17/7/2020

Pandas

[At any method, place cursor and press shift + Tab (to see help) / more details]

[In Jupyter working on pandas with python]

To read a tabular data file into pandas

Eg①

In[1]: import pandas as pd [(Ctrl + enter) — to run that cell
(Shift + enter) — to run & goes to new cell]

In[2]: pd.read_table('http://bit.ly/chiporders')

[read_table → can already read from an url]

And by using pd.read_table → it directly prints the data. If we don't want to print: we can save it as a data frame object

[orders = pd.read_table('http://bit.ly/chiporders')]

In[3]: orders.head()

[.head() → prints the first 5 rows]

Eg②

In: pd.read_table('http://bit.ly/movieusers')

Out:

	1|24|M|technician|85711
0	2|53|F|other|94043
1	3|23|M|writer|32067

→ this looks unsorted. "|" line that divided the columns.

[→ so, we have mention that "|" is a separator]

pd.read_table("http://...", sep='|') → [By default sep='\t']

[Now this gives a proper table]

But yet there is no header for table. 1\|24\|M\|technician is a row but it is taken as header

→ So, we have to mention there is no header

```
In: pd.read_table('http://...', sep='|', header=None)
```

Now, we want a header list for this data

```
In: user_cols = ['user_id', 'age', 'gender', 'occupation', 'zip_code']
In: pd.read_table('http://...', sep='|', header=None, names=
                                                    user_cols
```

Now get the perfect output

Note :-

→ read_table (filepath, sep = '\t', delimiter = None,
names = None, header = 'infer',

more useful ← { skiprows = None,
skipfooter = 0 etc...)

→ If file contains a header row, then have to explicitly pass header = 0 to override column names

→ index_col = False { index_col → column to use as row label, when to force pandas not to use first column as index }

* Instead of read_table we can also use read_csv

csv - comma seperated values.

csv file has sep = ',' by default

→ So, for csv files better to use read_csv

Panda Series :- [To create pseries = pd.Series([1,2,3,4,5])]

Each column or a data frame is called panda series.

→ we can also create seperate panda series.

→ we might need to do manipulate the series or to do analysis on it.

Ex:-

In: import Pandas as pd

In: ufo = pd.read_csv ('http://bit.ly/ufo-reports')

In: type(ufo) → gives type

In: ufo.head()

output

	City	Colors Reported	Shape Reported	State	Time
0	Ithaca	NaN	TRIANGLE	NY	2210
1					
2	Holyoke	NaN	OTHER	NJ	2014
3					
4					

(*) Now, basic way to select series is [ ] → bracket (or) '.' like ufo['City'] / ufo.City

In: ufo['City'] → could be safe

(or)

In: ufo.City

But, to select Color Reported by '.' we can't do that as it have space only [ ] for space having names

→ How do create new panda series in a DataFrame

In: ufo.city + ufo.state

Out: 0 Harahay
1 Holyoke CO

So, do provide ','

In: ufo.city + ', ' + ufo.state

Now it gives ','

But to add it to the dataframe

data-frame_name ['col_name'] =

In: ufo['Location'] = ufo.City + ', ' + ufo.State

→ Now, it adds new series

Note:

→ Imagine 'movies' file

Now,

In: movies.describe() → gives all statistical analysis about a series

In: movies.head()

In: movies.shape
Out: (67, 6) → rows, columns
In: movies.dtypes
Out: types of ay

→ Here describe, head need () Parenthesis but shape, dtypes doesn't. Because, describe, head are actions where as shape is property

→ How to rename columns in a dataframe

(m1) Syntax:

df_name.rename_axis('_')
to set index name

df_name.rename(columns = {'old_name': 'new_name', 'old_name': 'new_name'}, inplace = True)

→ Have to remember 'inplace = True', that's what change the name.

→ can change multiple names at a time.

ufo.rename(columns = {'Colors Reported': 'colors_Reported'}, inplace = True)

To check if

ufo.columns → to see the column names.

Index[u'City', ...]

(m2)

* But if you want to change all the column names above method could be cumbrous so type all old & new.

→ ↓ we create a list & assign to ufo.columns.

ufo_cols = ['city', 'color reported', 'shape', 'state', 'time']

* ufo.columns = ufo_cols → modified all names

method 3

we can change them while we are reading itself. must be done

```
ufo = pd.read_csv('http://...', names=ufo_cols, header=0)
```

to override the org. column names we have to use, header = 0.

Note:

* there is a direct method to remove all the spaces in columns

```
ufo.columns = ufo.columns.str.replace(' ', '_')
```

→ space with

→ To remove a column / row

we have to know the name or column index to remove.

```
ufo.drop('Colors Reported', axis=1, inplace=True)
```

only for string. this is for rows. this is for columns.

```
ufo.drop(0, axis=0, inplace=True)
```

Note: we can remove multiple rows/columns at

```
ufo.drop([0, 1], axis=0, inplace=True)
```

↳ list

How to sort a pandas DataFrame / series.

Eg:-

Syntax:- df_name.series-name.sort_values()

Int: `movies.title.sort_values()` ↳ series name

Out: Der (500) days of summer
!

(or)

Int: `movies['title'].sort_values()` → an other way with [' '] ↳ series name.

Out:

(default)

→ In the both cases sort by ascending order.

→ we can change it to descending.

Int: `movies['title'].sort_values(ascending = False)`

Out: 864 Creed
526 Zulu
!

By default ascending = True. But, we can change it.

↳ Now gives in descending order.

(*) In the both cases, it won't change the org. data, just gives a sorted output of series (not remaining data).

→ But if you want to see the completed data sorted by corresponding to a particular column, then

`movies.sort_values('title')` → column name → can see the complete data sorted (org. same)

`movies.sort_values('duration', ascending = False)`

Imp:-

→ Sorting by multiple columns.

```
movies.sort_values(['content_rating', 'duration'])
```

→ we can give multiple columns to sort.

↓ first sort by this & then sort by this ...

→

(A) How to filter rows of a pandas df by column va

Eg1 (m1) [to select all movies according to duration

```
import pandas as pd
movies = pd.read_csv('http://bit.ly/imdbratings')
longdur = [True if length >= 200 else False for length
                                                movies.du
long = pd.Series(longdur)
```

→ [creating series from a list of booleans]

```
movies[long]
```

→ Here Pandas takes the True, False. And remove all the False rows [or accepts booleans].

out

	dur
	200
	201
	207

Eg (m2) → Instead of doing all the above we name,

```
long = movies.duration >= 200
```

→ this directly creates series of True, Fa

movies[long]

only works on pandas

(*)

So, the Best method to filter rows by column value

In movies [movies.duration >= 200] → single line is enough.!!

df_name [df_name.col_name operation]

Same way,

In :- movies [movies.genre == 'Crime']

out

genre
crime
crime

→ If you want to see the genre of all the movies that are >= 200 duration.

In movies [movies.duration >= 200].genre → two operations

But sometimes this could cause trouble. Single operation

So, we prefer

movies.loc[movies.duration >= 200, 'genre']

→ .loc is very powerful (*). rows we want, column we want.

Seperates rows, columns

(we'll see.. later)

⊛ multi-filter

(using '&')

→ If we want movies ≥ 200 & Drama genre. then.

→ movies[(movies.duration >= 200) & (movies.genre == 'Drama')]

→ Parentheses all important.

→ movies.duration >= 200 ⟶ it a boolean series & False are eliminated

⊛ If you want some categories in a column,

→ like people from vizag, chirala, etc....

→ In such cases we can't give or (|) for many times like

Eg)

```
movies[(movies.genre == 'crime') | (movies.genre == 'Acti
```

↳ 'or' operation collects even every one is True.

we have simple syntax

⊛

```
movies[movies.genre.isin(['crime', 'Drama', 'Action'])]
```

simple-code ↓ easy best and 1 way. (In a column)

only for multi filter by or operation

⊛ Reading limited no. of columns

```
movies = pd.read_csv('http--', usecols=['title', 'rating'])
```

↓ instead of col-names we can use index no.

```
usecols = [0, 4]
```

```
movies.head()
```

output

	title	rating

⊛ To take a quick look at a big data

```
ufo = pd.read_csv('http--', nrows=3)
ufo
```

→ Preferable

Out: this only gives the first 3 rows (default).

⊛ Iteration (to access index)

```
for index, row in ufo.iterrows():
    print(index, row.city, row.state)
```

Out:
0, Ithaca, NY
1, Willoughe, CO

→ Selecting only numeric types from a data

```
import numpy as np
drinks.select_dtypes(include=[np.number]).dtypes
```

output: beer_servings int64, spirit int64, wine int64

↓ to check

18/7/2020

→ How to use 'inplace' parameter in pandas

So, that we can have a quick look at changes temporarily, data

changes original data

By default everything has inplace = False

→ First about inplace = True.

→ By default 'drop' → takes inplace = False that means, it won't affect the original data and just prints copied data by removing the specified.

→ inplace = False / not using inplace all same (just shows data, won't affect original)

→ while inplace = True, changes original data

```
drinks.drop('continent', axis=1).head()
```

↑ column.

inplace = False

shows without continent → Temporary

axis=0 (=) axis = 'index'

axis=1 (=) axis = 'columns'

drop - syntax

For column.

df_name.drop('col_name', axis=1, inplace=True)

For row

df_name.drop('row_name', axis=0, inplace=True)

so,

axis=1 → column axis,

axis=0 → row axis,

Eg:

If we want 'mean'.

in: drinks.mean()

out: beer_servings 106.16
wine_servings 80.99
:

in: drinks.mean(axis=0)

out: beer_servings 106.16

in: drinks.mean(axis=1)

out:
0 0.00
1 49.3
:

Both are same

axis is direction
axis=0 says go down.
row by row.
axis=1 says go left-right

→ what happens is, by default mean gives the mean of per-columns, not rows. (so, it's going down)

→ But we can tell Pandas to give mean of rows by saying axis=1, by default it has axis=0

By saying axis=1, it goes in left direction)

axis is direction movement

drinks.mean(axis=1).shape
or
drinks.mean(axis=0).shape
?

Instead of 0, 1.

we can use (axis = 'index') → axis = 0

(axis = 'column') → axis = 1

→ df_new.col_name.str.slice(-5,-4) ← can be any

(*) How to use string method [str.contains, str.upper]

(There will be many string methods)

Here are some useful methods

→ To change a complete series of DF to upper all.

→ orders.item_name.str.upper()

↳ column / row name

→ But, it won't affect original data

To override this with org. data

→ orders.item_name = order.item_name.str.upper()

↓

Now it changes in data.

→ If you want to find 'CHECKED' from all data

then, ← This returns boolean

orders[orders.item_name.str.contains('CHECKED')]

→ Here upper, contains are methods of str

to replace ()

→ order.choice_description.str.replace('(', '').str.replace(')', '')

→ consecutively two times to remove/replace two things.

(or)

→ order.choice_description.str.replace('[()]', '')

To remove multiple

'or' between them.

note:

① df_name [can take boolean values & give output]

→

How to change the datatype of Pandas series (.astype)

→ order.dtypes

o/p

item_price object

item_price
$9.0

so, to do modifications to price we have to convert it to float.

→ order.item_price.astype(float)

But, as we also need to remove '$'. we can do it in one line

→ order.item_price.str.replace('$', '').astype('float')

↳ multi-operations.

Impt useful for ML.

→ astype also useful to convert booleans,

```
orders.item_name.str.contains('chicken').astype(int)
```

→ False } → are converted to { 0
True } { 1

applies for all numeric

Syntax:

(*) using "groupby" df_name.groupby('col_name').operat

(agg - useful to give multiple

[Not compulsory to specify column. If not, it gives all num_cols]

Ex: drinks - data

Q) drinks.beer_servings.mean()

106.1606

(*) groupby - (mainly) when you want to categorize a column.

Q) drinks.head()

country	beer_servings	....	continent
			Asia
			Europe
			Asia
			Ocenia
			⋮

→ But, what if we want the mean of only Europe country or like only Asia.

```
drinks[drinks.continent == Asia].beer_serving.mean
```

→ In this way we can find

→ But, whatif we want a data in seperate for country.

Then → 'groupby' comes into role.

→ First, it 'filter' the data by each name and then does the analysis.

also we have min, max etc

so,

```
drinks.groupby('continent').beer_servings.mean()
```

out

continent	
Asia	61.47
Africa	37.04
Europe	193.77
:	

This filter the column and applied the operation for each

categorize the column,

Now, we have 'agg' method which lets you to give multiple arguments.

```
drinks.groupby('continent').beer_servings.agg(['min','max','count'])
```

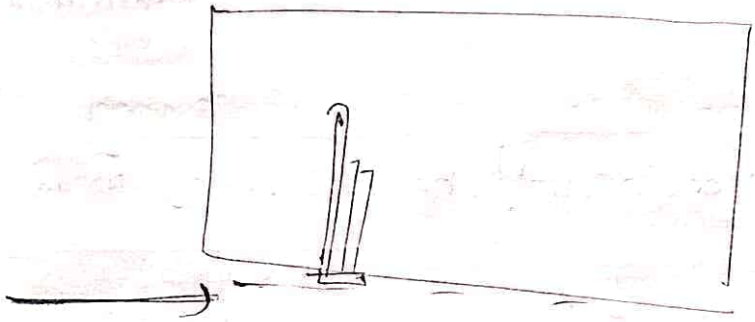
takes multiple

continent	min	max	count
Asia	44	0	:
Africa	61	0	:
Europe	53	1	

To see the above data in bar in a graph

in: `%matplotlib inline`

in: `drinks.groupby('continent').mean().plot(kind='bar')`



(*) To explore a panda series —

[useful methods
describe, value_count etc]

→ take a movies data

in: `movies.genre.describe()` → describe is so useful in case of objects

out:
```
count   979
unique   16
top      Drama
freq     327
```

→ How of given no. of unique object.
→ And the most common one & its frequency.

→ If you want to see the all unique types

in: `movies.genre.unique()` → `unique()` to know number

→ show all unique obje in genre

out: array(['crime', 'drama' ...])

→ If you want to explore about the unique objects. like of counts then ↓

(movies.genre → column name)

In] `movies.genre.value_counts()`

out]

Drama	278
comedy	156
crime	124
⋮	

→ It is highly useful for objects type

In] `movies.genre.value_counts(normalize=True)`

out]

Drama	0.2839	→ 28%
Comedy	0.1583	→ 15%
⋮		

→ give percentage data

→ If you want a crosstab b/w two columns.

In] `pd.crosstab(movies.genre, movies.content_rating)`

out]

content_rating → genre ↓	Approved	G	GP..
Action	3	1	1
Comedy	9	2	1

* describe, value_counts with int type not much useful

In] `movies.duration.describe()` | `movies.duration.value_counts()`

out]

count	979.00
mean	120.979
min	⋮
max	⋮

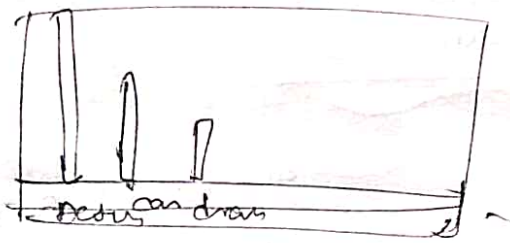
↓ with int type, give mean, min, max

42	93
⋮	

→ we can also see graph/plot for value_counts graph

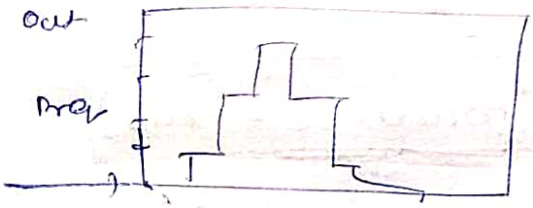
→ %matplotlib inline

→ movie1.genre.value_counts().plot(kind='bar')



→ histogram

→ movie1.duration.plot(kind='hist')



(*) To Handle missing values in pandas

(*) isnull()
notnull()
sum()

→ take 'ufo' file

→ ufo.tail()

city	colors Reported	shape
Rank	NaN	NaN
lake	NaN	Triangle
Ware	NaN	Valley
Dome	RED	light

→ If we see above table the 'NaN' characters represent null.

→ No element is present, missing values.

→ we can find the missing values by using isnull.

→ `ufo.isnull().tail()` → to see last 5

city	colors	shape
False	True	True
False	True	False
False	True	False
1	False	

→ isnull() converts all the characters into booleans and all the NaN (missing values) becomes True. Others becomes False

→ so, all the True elements here are missing elements.

→ `ufo.notnull().tail()` → notnull is reverse, null elements - False

→ To find all the missing value, we have sum().

Eg:

In: `pd.Series([True, False, True]).sum()`

Out: 2.

→ So, sum() has 'axis=0' default and counts all the 'True' values. So,

In: `ufo.isnull().sum()`

Out:
```
city      25
Colors    4359
Shape     2644
Space     0
Time      0
```

→ gives all the missing values details in a data frame

Imp

① To list out all the missing values

→ `ufo[ufo.City.isnull()]` → col_name → Gives the details of missing values in City

	City	colors	Shape	State
0	NaN	NaN	NaN	LA
22	NaN	NaN	Light	LA

(with index number)

② To remove all the data, even if there is a single missing value in row

`ufo.dropna(how='any')` → This directly prints data. But won't modify it.

`ufo.dropna(how='all')`

'or' → 'any' tells that in a row, if any element is missing it row

'and' → all tells that if it a row only if all the element are 'NaN' missing.

→ we can check the shape of pure data,

`ufo.dropna(how='any').shape` → This gives a shape of clean data without any missing.

③ Manually,

If we have to modify the missing value by something then,

`ufo['colors Reported'].fillna(value='NOTSEEN', inplace=True)`

(inplace=False) → It's temporary change

④ If we want drop a row, if either city / shape is missing.

i.e; compulsorily columns to drop. Then,

```
ufo.dropna(subset=['City', 'Shape Reported'], how='any')
```

Now, this drops a row if there is a missing value in city / shape.

→ how='all' ← this will drop a row, if there is a missing value in both columns

value_counts()

↳ By default dropna=True,

```
ufo['Shape Reported'].value_counts()
```

up

```
light
oval
)
```

→ But this won't give the count of missing value

→ By default (value_counts) has (dropna=True). so, (dropna=False) to change.

```
ufo['Shape Reported'].value_counts(dropna=False)
```

all

```
NaN 2977
Disk 1
light 1
```

Now gives missing value count

Note:- Keywords

isnull()

notnull()

isnull().sum()

dropna() / dropna(subset=['...'])

fillna() → (value = ( ))

(*) ans is 20,

at thresh=2

→ delete all rows which do not min 2 non-NAN

thresh = None

→ it requires no. of non-NAN

(how='any')

or

(how='all')

✱ About Pandas Index (labels)

- Identification
- selection
- Alignment

① Identification

→ Even if we filter the data, it keeps index / row number as it is.

Eg:

```
print(drinks[drinks.continent == 'Asia'])
```

out:

	country
6	India
20	
101	
112	

→ we can identify at what row no. it is.

② selection [loc]

→ when we want a piece of data we need index.

→ If you want to access the beer_servings of India.

then, we need its index no.

```
print(drinks.loc[6, 'beer_servings'])
```

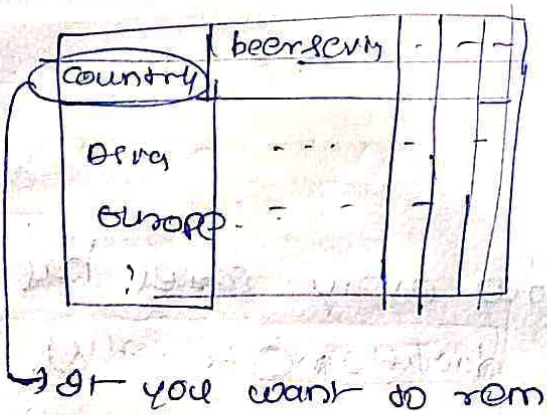
out:

→

→ But, it's kind hard to remember the index no. all the time. So, we can change the index.

```
drinks.set_index('country', inplace=True)
drinks.head(1)
```

out

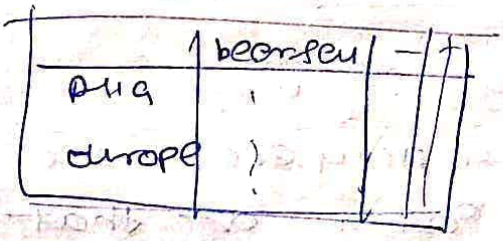


→ If you want to remove it, we can.

```
drinks.index.name = None
drinks.head(1)
```

→ It removed index name

out



Now we can access 'India beer_serv' without index number.

```
drinks.loc['India', 'beer_serv']
```

6

If you want to reset the index then, first give name

```
drinks.index.name = 'country'
drinks.reset_index(inplace=True)
```

loc

→ we can use × for any dataframe

Eg.

```
drinks.describe().loc['25%', 'beer_servings']
```

↳ we can use loc, even with describe method.

③ Alignment

→ First we have to know every series has a index (same as its dataframe - mostly).

Eg:-

Inp: `drinks.continent.value_counts()`

Out:

Africa	53
Asia	44
Europe	45
!	)

→ If we see here Africa, Asia etc. are the index of that series.

→ As we know, we can access formerly by index. Same here,

Inp: `drink.continent.value_counts()['Asia']`

Out: 44

Inp: `drinks.continent.value_counts().sort_values`

↳ index for this series.

Asia	44
Europe	45

But, we can also sort by index

Inp: `drinks.continent.value_counts.sort_index()`

→ Eg:

```
people = pd.Series([30,000,000, 85,000], index=['Albania',
                                                 'Andorra'],
                    name = 'population')
```

↳ This is creation of dataframe.

↓ This goes as index name

In: people

Out: Albania 30500000
Andorra 85000
Name: population, dtype: int64.

→ Now, if you want multiply the population of Albania, Andorra with its beer_servings in drinks. Easy.

In: drinks.beer_servings * people

Out:

Afghanistan	NaN	→ with the full index of drinks, it searched for people index & aligns there index with those & does the math.
Albania	26700000	
Algeria	NaN	
Andorra	20825000	→ they are at their same position
Asia	NaN	→ Every they are became NaN
⋮	⋮	

This is called alignment

→ Assignment, also useful, if you want to add a new column or row to the given data.

→ For column → axis=1, & index-matched / aligned

→ If you want to add people to drinks.

Eg. `pd.concat([drinks, people], axis=1).head()`

	beer				population
Afghanistan					NaN
Albania					3000000
Algeria					NaN
Andorra					77000.

→ as name is population, it takes that as a column name, and aligns the index & gives a column, [everything else is NaN.]

Eg If (axis=0)

Creates a row, with a new column as there is no such name as population

→

(*) loc, iloc & ix — · —

ix → Ix has been deprecated.

a : b → Both are inclusive (*)

loc → loc[row_label, column_label]

It is to select specified rows & columns.

inp: `ufo.head()`

out:

	City	Colors Reported	Shape Reported	State	Time
0	Ithaca				
1	—				
2	—				
3					

inp: `ufo.loc[0,:]` → this specifies take 0 row, and columns to the end.

out:

City	Ithaca
Colors Reported	NaN
Shape Rep	Triangle
State	NY
Time	6/1/1930 22:00

inp: `ufo.loc[0:3, :]` → This specifies that, we '0' to '3' rows (both are inclusive) and columns go to end.

out:

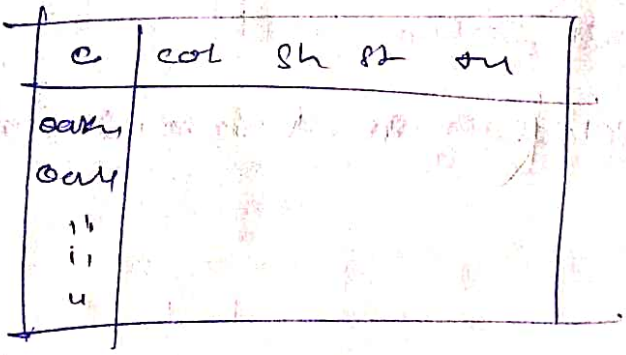
	City	C	Sh	St	Ti
0					
1					
2					
(3)					

(3) → inclusive.

inp: `ufo.loc[:3, 'city':'state']` → This says we 0 to 3 rows & city to state columns

	city	col	Sh	St
0				
1				
2				
3				

eg: auto[auto.city == 'oakland']



This can be achieved by loc also

same

eg: auto.loc[auto.city == 'oakland', :]

iloc [a:b, a:b] → exclusive of b

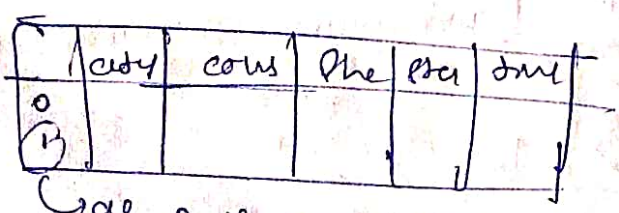
→ iloc won't accept strings.

like, eg:

auto.iloc[0:3, 'city']

→ Error → it can't be string

eg: → auto.iloc[0:2, 5]



as it is exclusive.

→ To read multiple columns

→ auto.[['city', 'state']][0:4], explicitly

— Monday — 8:00Am)

20/7/20

To make the dataframe smaller and faster

[using 'category']

→ when we import a file, it just creates a reference or object. when we check the info of it, it won't shows the full size of object. because, as we know strings consume more memory than int. so, what we can do to reduce size is convert the string to int. And this can be done by Category.

Eg:-

Inp: drinks.info() ——→ info give complete information about column & everything

out:

```
country      object
beer         int
 :
continent    object
memory-usage: 9.2+ kb
```

Here, it says it has some more data, that is it is not storing into object column deep.

If we wanna see full memory.

Inp: `drinks.info(memory_usage='deep')`

out:

```
country      object
 :
 :
memory-usage: 30.5 kb
```

→ this is the total memory used.

→ If you want to see memory consumed by seperate columns.

in: `drinks.memory_usage()`

out:

index	128
country	1544
beer_serv	1544
⋮	⋮
continent	1544

→ shows the added the object usage.

in: `drinks.memory_usage(deep=True)`

out:

index	128
country	12588
beer	1544
⋮	⋮
continent	12332

→ originally consumed data

↳ we can use `.sum()` → to see total

→ Now, to change these objects to category. First, we have to see the no. of unique types in a series. It only works when there are only a few unique types.

→ It is waste to use when there are many unique types.

in: `sorted(drinks.continent.unique())`

out: ['Africa', 'Asia', 'Europe', 'North Ame', 'Ocean', 'South Ame']

(there are only 6 unique value in continent column)

(so, we can use category

Basically, what category does is, in the sorted order for unique values, it give 0 — n (numbers).

like, Africa - 0
Asia - 1
Europe - 2
NA - 3
Oa - 4
:

→ there are just the codes for it,

in: `drinks.continent = drinks.continent.astype('category')` → col name

in: drinks.dtypes

out:
```
:
continent  category
```

in: `drinks.memory_usage(deep=True)`

out:
```
Index          128
:
continent      744
```
→ reduced from '12332' to '744'

[size has been reduced]

in: `drinks.continent.cat.codes.head()` → to see them in codes form

out:
```
0  1
1  2
2  0
3  2
4  0
:  :
```

For country

In: drinks.country = drinks.country.astype('category')

In: drinks.dtypes

Out:

Index	128
country	18094
beer	4144
⋮	
continent	2144

→ for country it has been increased too much after converting to int. That is because, it has too many unique value

In: drinks.country.unique()

Out:

~

L=193 → there are 193 values. It takes more memory in int than in type

so,

Notes

category can only be used when there are less no. of unique values.

(*) As we know category converts them as int, we can use it for another advantage. When sorting a column by a logical order. we can specify the order.

→

Eg:-

```
df = pd.DataFrame({'ID': [60, 100, 70, 80],
                   'Quality': ['good', 'excellent', good', 'vgood']})
```

```
df
```

	ID	Quality
0	60	good
1	100	excellent
2	70	good
3	80	very good

```
df.sort_values('Quality')
```

	ID	Quality
0	100	Excellent
1	60	good
2	70	good
3	80	very good

As basically, sort_values sorts by alphetical order

But we know this it's not what we want

But, if we want it to sort, on the logical order as we want then we extra categorise it as good=0, very good=1, Excellent=2...

```
from pandas.api.types import CategoricalDtype
quality_cat = CategoricalDtype(['good', 'vgood', 'excellent'], ordered=True)
df.Quality = df.Quality.astype(quality_cat)
df.sort_values('Quality')
```

	ID	Quality
0	60	good
1	70	good
2	80	very good
3	100	Excellent

→ Now, it sorted as we wanted.

→ But, this can also be taken advantage in another way.

→ If we want all the data > good.
Actually, we can't apply comparison operator on strings. But, as we changed it category in categoricalorder, now, we can apply.

```
df.loc[df.Quality > 'good']
```

→ in this way category is actually

	ID	Quality
1	100	excellent
2	20	very good

(*) Dummy Variables —•—

It is creating (0,1) for every unique value in a column. For every unique series, it creates 1 and all 0's.

Eg:-

```
import pandas as pd.
train = pd.read_csv('https://bit.ly/kaggletrain')
train.head()
```

	sex
	male
	female
	male
	male

→ now to identify easily, male & female. we created a new column.

mapping

```
train['Sex_male'] = train.Sex.map({'female':0,
                                   'male':1
                                  })
```

→ map method created a series with the way assigns values.

train.head()

	Sex	Sex_male
	male	1
	female	0
	male	1
	male	1

→ assigns as we mentioned.
→ now, here all ones are female,
→ ones → male.

→ This mapping is also called dummies,,

pd.get_dummies(train.Sex) → one hot-encoding

	female	male
0	0	1
1	1	0
2	0	1
3	0	1
4	1	0

→ get_dummies, gives all the possible series. In a column there are k unique elements it gives k columns for get_dummies. And each one has one and zero.

→ But, to identify we only need male here or female, like we only need one series.
→ If we take male all ones are male & vice versa
→ So, we always need k-1,,

inp `Pd.get_dummies(train.Sex, Prefix='Sex').iloc[:, 1:]`

out

Sex_male
1
0
1
1

inp `male_dummy = Pd.get_dummies(train.Sex, Prefix='Sex').iloc[:,1:]`

inp `Pd.concat([train, male_dummy], axis=1)`

out this gives same output as we got before

Sex	Sex_male
male	1
female	0
male	1

Same way,

inp `embarked_dummies = Pd.get_dummies(train.Embarked, Prefix='Embarked').iloc[:,1:]`

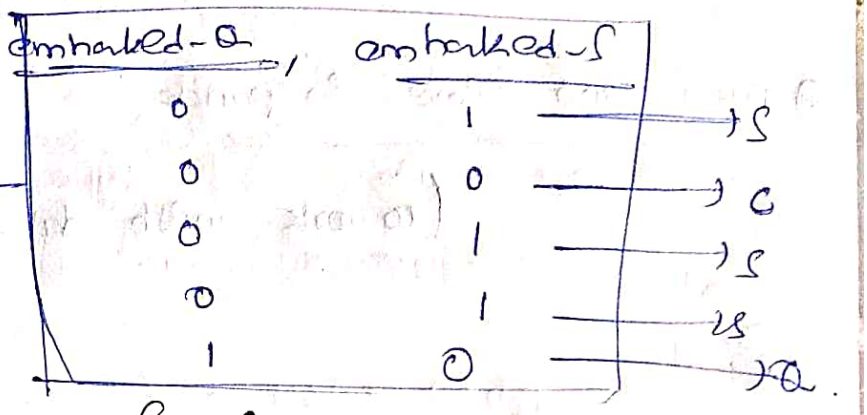
inp `Pd.concat([train, embarked_dummies], axis=1)`

out

Sex	Embarked	Sex_male	Embarked_Q	Embarked_S
male	S	1	0	1
female	C	0	0	0
male	S	1	0	1
male	S	1	0	1
female	Q	0	1	0

↓ newly added columns

→ At we see



this is why we said, we only need K-1.

→ Here, two are enough to find all.

Notes:

(1) Upto now we are passing a series (a column) to get_dummies. But, we can also pass a data frame to get_dummies.

Eg:-

Inp:
```
pd.get_dummies(train, columns = ['sex', 'Embarked'],
                                drop_first = True)
```

Out:

...	sex	embarked	...	sex_male	embarked-Q	embarked-S
train	S			1	0	1
Rema	S			0	0	0
				1	0	1
				1	0	1
				0	1	0

→ Here, we accomplished what we need on a single line without using 'iloc' or 'concat'.

→ But, a little drawback is we lost the org. sex, embark columns by this.

⇒ And we exactly get K-1 columns, by saying drop_first=True.

⊛ Date and Time in pandas

(to work with date & time)

Eg:

In: ufo.head()

Out:

	Time
0	6/1/1930 22:00
1	6/30/1930 20:00

Suppose to extract hour from time: we can do it by slice. But first extra check the type.

In: ufo.dtypes

Out:

Time	object

→ so have to convert it

In: ufo.Time.str.slice(-5,-3).astype(int).head()

Out:

0	22
1	20
2	1

→ we can do it this way. But what if we want year. again the same.

But, instead all this pandas provides to_datetime which have a ton of excellent operation to use, like weekday, year, time etc.

overriding to datetime dtype

inp: `uto['Time'] = uto.pd.to_datetime(uto.Time)`

inp: uto.head()

out:

	Time
	1980-06-11 22:00:00
	1930-06-30 20:00:00

→ Data is same, But the format has changed.

Actually what really changed dtype

inp: uto.dtypes

out:

Time	datetime

→ now, it is a 'dt' object. we can perform some dt operations.

inp: `uto.Time.dt.hour`

out:

0	22
1	20
2	1

inp: `uto.Time.dt.weekday`

out:

0	6
1	0
3	6
3	1

0-monday
⋮
6-sunday

This gives the day by calculating the date.

inp: `uto.Time.dt.dayofyear`

out:

0	162
1	181
2	46

→ gives the day of year (...)

→

And we can also perform all the math operations of datetime,

eg:-

in: `ts = pd.to_datetime('7/27/1999')` → creating a time stamp

in: `ufo.loc[ufo.Time >= ts, :]` → can apply math operation

out:

	Time
	1999-7-27 20:00
	1999-7-29 ...

↳ now this gives the data of all dates which are greater than time stamp

in: `ufo.Time.max()` → it gives the latest time.

out: `Timestamp('2000-12-31 23:59:00')`

in: `(ufo.Time.max() - ufo.Time.min()).days`

out: 25781 days

→

notes:

→ we can see its complete usage in API reference

→ ctrl+f1 opens search bar

→ we can find everything

make a date-time graph (matplotlib)

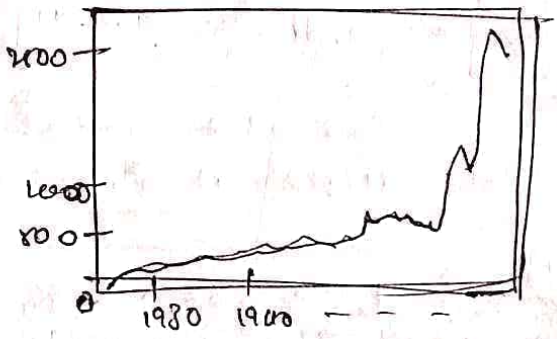
```
%matplotlib inline
```

```
ufo['Year'] = ufo.Time.dt.year
```
→ Adding a new column

```
ufo.Year.value_counts().sort_index()
```

```
1930    2
1931    2
  ⋮
```

```
ufo.Year.value_counts().sort_index().plot()
```



21/2/20

Shows both first and last → keep = False
Shows last & keeps first → keep = 'first'
Shows first & keeps last → keep = 'last'

(*) To remove and find duplicates in a column

In: import pandas as pd

In: user_cols = ['user_id', 'age', 'gender', 'occupation', 'zip_code']
users = pd.read_table('http://bit.ly/movieusers', sep='|', header=None, names=user_cols, index_col='user_id')

In: users.head()

Out:

user_id	age	...	zipcode
1			
2			
3			

(*) To check the shape

In: users.shape

Out: (943, 4)
→ rows.

Ex: But, in 943 rows there could some duplicate rows, like same data in 2 rows, or like zip_code - duplicate, repeated zip_codes.
→ If there are duplicates we have to find & remove them.

→ col-name → To check no of duplicates we use .sum() → give count of True

inp: `users.zip_code.duplicated()` → this gives a series of boolean

out:

0	False
1	False
2	False
3	True
⋮	⋮

→ In here, 'True' is something that has already there above it.
→ That means, actually duplicated by default has keep='first', which keeps the first one & show the second one as duplicate (i.e; True)

As, we know to see the Trues

inp: `users[users.zip_code.duplicated()]`

out:

1	
496	
572	
621	
684	

→ gives all the True's

`users[users.zip_code.duplicated(), :]`

we can also get same outpt by loc

inp: `users.loc[users.duplicated(), :]`

inp: `users.loc[users.duplicated(keep='last'), :]`

out:

496	21	F	Student	55419
572	51	M	Educator	20003
⋮	⋮	⋮	⋮	⋮

496 rows

out:

198	21	F	Student	55419
85	51	M	Educator	20003
⋮	⋮	⋮	⋮	⋮

496 rows

→ Here, when duplicated() → as keep='first'. It keeps (198) → first occured & shows up (496).

→ In (keep='last') → it keeps 496 & shows up (198) complete same rows

`df.name.duplicated`

This gives True, only if all the columns of dat has repeated, And by default keep='first'

→ And, now we have keep= False, which shows both of them.

```
users.loc[users.duplicated(keep=False),:]
```

out

912 rows → Both of them.

To drop-duplicated

```
users.drop_duplicates().shape
```

out (9866, 4)

* And this won't affect our data unless you use, inplace=True

we first have 943, rows, 936, so all of the 7 duplicate rows has been lost. (last ones are the last duplicated.)

```
users.duplicated(subset=['age','zip_code']).sum()
```

out 32

→ to check the duplicated by two columns.

→ It gives True, if a row has same age, same zip code

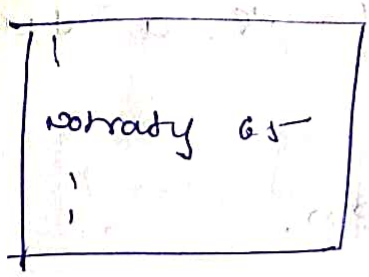
→ so, there are 32 age, zip-code same rows.

(*) Copy_warning

Case-1

In [ ] movies.content_rating.value_counts()

Out [ ]



In [ ] movies.content_rating.isna().sum()

Out [ ] 3

we have change all notrated to NaN.

→ So, for that first we have to take the 'Not rated' series and then we have to assign it to (nan) to make them null elements.

In [ ] import numpy as np

In [ ]
```
movies[movies.content_rating == 'NOT RATED'].content_rating
                                            = np.nan
```

Out [ ] ——warning——

In [ ] movies.content_rating.isna().sum()

Out [ ] (3) → Here still 3. It has not changed

→ Actually what happens with.

[movies[movies. -- ='Not rated']. content_rating = np.nan]

such line of code is. It is a two operations code.

→ this part is get_item.

→ Remaining is set_item, which performs an operation on the produced result of get_item.

→ Actually, here is the problem. Pandas can't gurante whether the get_item, returned a view or a copy of the data.

→ so, it gives a warning.

But, this is not the case with .loc

↳ loc is a single operation.

```
movies.loc[movies.content_rating = 'NOT Rated', 'content_rat
                                              = np.nan
```

```
movies.content_rating.isnull().sum()
```

out 68 ——— Now, it changed.

note ①

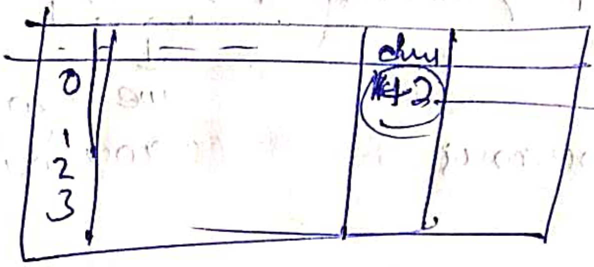
→ So, it's always better to prefer .loc

case – 2

In] top_movies = movies.loc[movies.star_rating >= 9, :]

In] top_movies

out]



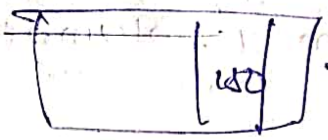
→ If we want to change this we can do by .loc

In] top_movies.loc[0, 'duration'] = 150

out] —•— warning

In] top_movies

out]



→ even though, it changes.

But the warning is, it don't know top_movies is a copy/view

note ②

→ So, it's always better to specify its copy() when copying data

In] top_movies = movies.loc[movies.star_rating >= 9, :].copy()

In] top_movies.loc[0, 'duration'] = 150

— no error —

To see the max_cols/rows → pd.get_option → this to check.

⊛ Display option

To display all the rows → None shows all the rows

Ans
```
pd.set_option('display.max_rows', None)
```
↓ we can give any number also

To reset the max_rows

Ans
```
pd.reset_option('display.max_rows')
```

To get max_columns

Ans
```
pd.set_option('display.max_columns', any number)
```

To see the full text in a column

Ans
```
pd.set_option('display.max_colwidth', None)
```
↓ can give any number

To print ',' in numbers (like 4,600.0)

Ans
```
pd.set_option('display.float_format', '{:,}'.format)
```
↓ this only works for float columns, not for ints //

dummy DataFrame

method ① → creating a df by dictionary. But, it won't follow a order. we can specify.

keys → column name
values → column values

```
import pandas as pd
pd.DataFrame({'id': [100, 101, 102],
              'color': ['red', 'blue', 'red']})
```

out

	color	id
0	red	100
1	blue	101
2	red	102

→ we can specify the columns order, if we want.
→ we can also change the index.

```
pd.DataFrame({'id': [100, 101, 102], 'color': ['red', 'blue', 'red'],
              columns = ['id', 'color'], index = ['a', 'b', 'c']})
```

out

	id	color
a	100	red
b	101	blue
c	102	red

→ now we specified the order we also gave index values.

note

1) we can create a normal dataframe by dictionary.
2) other way by list of list.
3) another way by numpy arrays

Apply, map and applymap

→ (map) is a series method created a dummy of a series.

→ (apply) is both series & dataframe method.

→ (applymap) is only for dataframe method. And applies to each & every element in dataframe.

map

applying operation on a series

a series

Eg: `train['sex_num'] = train.sex.map({'female':0, 'male':1})`

Eg: `train.loc[:4, ['sex', 'sex_num']]`

o/p:

sex	sex_num
male	1
female	0
female	1

apply (both for series, dataframe).

on series

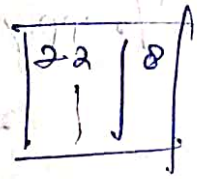
(*) But, not do the complete datafr, only for an axis

Eg: `train['name_len'] = train.Name.apply(len)`

only specifies operations, can't pass an argument

only operation

```
import numpy as np
train['fare_ceil'] = train.Fare.apply(np.ceil)
```




```
train.Name.str.split(',').head()
```

to take the first letter of Name. First split at ','

```
def get_element(my_list, position):
    return my_list[position]
train.Name.str.split(',').apply(get_element, position=0)
```

0	Braund
1	Cumings

Instead of a function, we know lambda.

```
train.Name.str.split(',').apply(lambda x: x[0])
```

0	Braund
1	Cumings

apply on dataframe

```
drinks.loc[:, 'beer_servings':'wine_servings'].apply(max, axis=0)
```

beer_servings	376
wine_servings	370

to see the max in a row

inp
```
drinks.loc[:, 'beer_serv':'wine_serv'].apply(max, axis=1)
```

out

0	0
1	132
2	25

inp
```
drinks.loc[:, 'beer_serv':'wine_serv'].apply(np.argmax, axis=1)
```

out

0	0
1	1
2	0
3	2

np.argmax → This gives the column_number of the existing max value in a row.

query

applymap → This applies to full data, irrespective of axis.

inp
```
drinks.loc[:, 'beer_servings':'wine_servings'].applymap(float)
```

out

	beer	spirit	win
0	0.0	0.0	0.0
1	0.0	132.0	0.0
2	89.0	132.0	54.0
3	25.0		54.0

→ applymap affects full dataframe.

22/2/2020

(*) multi-index

[selecting a multi-index is almost similar to a normal series]

[or works like groupby]

In: import pandas as pd.

In: stocks = pd.read_csv('http://bit.ly/smallstocks')

In: stocks

Out:

	Date	Close	Volume	Symbol
0				CSCO
1				AAPL
2				MSFT
3				AAPL
...				MSFT

In: stocks.groupby('Symbol').Close.mean()

Out:

Symbol	
AAPL	112.52
CSCO	31.68
MSFT	57.42

→ This is a general groupby output.

→ But, we can also do groupby on multiple columns.

In: stocks.groupby(['Symbol', 'Date']).Close.mean()
ser =>

Out:

Symbol	Date	
AAPL	2016-10-3	112.52
	2016-10-4	113.00
	2016-10-5	113.05
CSCO	2016-10-3	31.5
	2016-10-4	⋮
	2016-10-5	⋮
MSFT	⋮	

→ Now, this is a multi-index series with both symbol and date.

(*) this multi index series can easily be transformed into a dataframe by unstack

only

```
ser.unstack()
```

→ lowest level first index become index
→ second index become column.

output

Date	2016-10-3	2016-10-4	2016-10-5
Symbol			
APPL			
ESCO			
MSFT			

(*) But this output can also be obtained by a method 'pivot_table'.

(*) For the index & column of table the unique values of the columns of dataframe.

(*) And the values are the mean values by default. we can change that aggregate

only

same output

```
df = stocks.pivot_table(values = 'close',
                        index = 'symbol',
                        columns = 'date')
```

selecting data from multi-index series

[Almost same as normal df]

ser

Symbol	Date	
APPL	2016-10-3	112
	2016-10-4	113
	2018-10-5	113
CSCO	2016-10-3	82
	2016-10-4	2
MSFT	2016-10-5	

If we want the stock data of a company. (for all dates)

```
ser.loc['APPL']
```

→ selecting the outer index gives complete inner data & values of it.

Date	
2016-10-3	112.5
2016-10-4	113.00
2016-10-5	113.05

If we want value of that date

```
ser.loc['APPL', '2016-10-03']
```

→ this gives out the value of the data selecting outer & a inner index.

112.5

```
ser.loc[:, '2016-10-03']
```

→ selecting just a inner index, gives the values & outer index.

Symbol	
APPL	112.52
CSCO	82.00
MSFT	52.42

⊛ DataFrame with multi-index

cmd: `stocks.set_index(['symbol', 'date'], inplace=True)`

↳ setting multi-index

cmd: `stocks`

out:

Symbol	Date	close	volume
esco			
APPL			
MSFT			
APPL			

↳ To set in order.

cmd: `stocks.sort_index(inplace=True)`

cmd: `stocks`

out:

Symbol	Date	close	volume
APPL	2016-10-3		
	2016-10-4		
	2016-10-5		
CSCO	2016-10-3		
	2016-10-4		
MSFT	2016-10-5		

→ multi-index dataframes

cmd: `stocks.loc['APPL']`

out:

Date	close	volume
2016-10-3		
2016-10-4		
2016-10-5		

selecting from multi°-index dataframe

in: `stocks.loc['APPL', '2016-10-03']` → This will give output, but not a good practice, as we know `.loc[index, column]`

out: 112.52

in: `stocks.loc[('APPL', '2016-10-03'), :]` ← good practice

out: 112.52 2190 --

→ as both are index, we should give them in duply

in: `stocks.loc[('APPL', '2016-10-03'), 'close']`

out: 112.52

(*) If you want multiple outer index/inner index we have give them in a list.

in: `stocks.loc[(['APPL', 'MSFT'], '2016-10-03'), :]` → multiple outer-index & all columns

out:

Symbol	Date	close	volume
APPL	2016-10-03	112.52	2102
MSFT	2016-10-03	30.71	1011

in: `stocks.loc[(['APPL', 'MSFT'], '2016-10-03'), 'close']` → single column

out:

Symbol	Date	close
APPL	2016-10-03	
MSFT	2016-10-04	

→ multiple inner index.

in: `stocks.loc[('APPL', ['2016-10-03', '2016-10-04']), 'close']`

in: `stocks.loc[(slice(None), ['2016-10-03', '2016-10-04']), 'close']`

Cannot be used. ← `(:, [--` → Thus says all outer index for the selected multi°-index.

syn `stocks.loc[(slice(None), ['2016-10-03', '2016-10-04']), 'close']`

All outer index

For the selected inner-index

out

Symbol	Date	
AAPL	2016-10-03	112.52
	2016-10-04	
CSCO	2016-10-03	
	2016-10-04	
MSFT	2016-10-03	
	2016-10-04	

(*) Imagine two same multi-indexed dataframes, → close, volume.

(*) to combine them, we use merge.

syn
```
both = pd.merge(close, volume, left_index=True,
                right_index=True)
```

note:

→ to reset multi-index

```
df_name.reset_index()
```

merge and join()

↳ By column keys ——→ By row name

- df1.append(df2)
- pd.concat([df1, df2]) → Best way.

- df1.join(df2)
- pd.merge([df1, df2]) → Best way,

Eg:-

→ movies.head()

	movie_id	title
0	1	shawshank
1	2	inception
2	3	Darkknight
3	4	Donnie Darko
4	5	Interstellar

→ ratings.head()

	user_id	movie_id	rating	timestamp
0	196	240	3	—
1	186	302	3	—
2	140	026	1	—
4	23	386	2	—
8	45	432	1	—

shape() → 1682

these both are same movie_id. But this diff rating table without title.

shape() → 100000

For 1682 movies 100000 ppl has given rating.

→ so, we can merge both of them, to get title.

movie_ratings = pd.merge(movies, ratings)

out:

	movie_id	title	user_id	rating	timestamp
0	1	Shawshank	196	3	—
1	1	Sh	120	3	—
2	1	Sh	134	2	—
3	1	Sh	146	5	—
4	1	S	152	4	—

(*) Here, left df checks the index or right df & that merges on this. So, first movie_id, title & later userid, rating, ts

→ movie_ratings.shape

(100000, 5)

every movie is at 1682, it checks every row in ratings matched & gives its correspond value

---

(*) what if columns you want to join on don't have same name

→ movies.columns = ['m_id', 'title']

→ movies.columns

o/p Index(['m_id', 'title'])

→ ratings.columns

o/p Index(['user_id', 'movie_id', ...])

(now they diff. name)

→ pd.merge(movies, ratings, left_on='m_id', right_on='movie_id')

↳ left_on, right_on > these say which column on left have to merge with right.

(*) what if you want to join on one-index

→ movies = movies.set_index('m_id')

→ movies.head

	title
m_id	
1	Alien
2	—
3	—
4	—
5	—
6	...

→ pd.merge(movies, ratings, left_index=True, right_on='movie_id')

↳ now as the way we want to match on left df index, we gives 'left_index=True'

↳ when these both merged,

→ merge happens like -

on first dataframe, it takes the first element of the column we selected & checks it with every single element in the second df column (that we selected). And if it finds a match, it prints it.

so, output

	title	user_id	movie_id	rating	travel
24	~~Toy Story~~	308	1		
454	Shaw				
457	Shaw	287	1		
		148	1		

→ If the both have common index, if given that common index

→ But, if column we want is an index, then the resultant index changes randomly.

→ In the above case, it takes "1" & checks with everything & prints.

→ But, as the left one is index, it won't in resultant table.
Put on

---

④ Join on two indexes [considers rows]

→ ratings = ratings.set_index('movie_id')

→ ratings.head

o/p

movie_id	user_id	rating	title
122			
185			
232			
⋮			

→ pd.merge(movies, ratings, left_index=True, right_index=True)

---

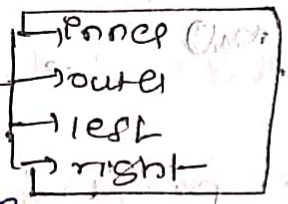
out

	title	user_id	rating	timestamp
1	Raw			
0	Shaw			
11	Shaw			
1	Shaw			
:				

↳ Gal left index searched for all the values or right & prints. It comes this way (first comes first)

→ like, first element in left searched & gets placed & after 2nd etc..

→ Types of Joins



A

	color	num
0	green	1
1	yellow	2
2	red	3

B

	color	size
0	green	S
1	yellow	M
2	pink	L

→ when we simply say merge, it merges based on common column-names

→ `pd.merge(A, B, how='inner')`

out

	color	num	size
0	green	1	S
1	yellow	2	M

↓

only common on both gets printed

→ `pd.merge(A, B, how='outer')`

out

	color	num	size
0	green	1.0	S
1	yellow	2.0	M
2	red	3.0	NaN
3	Pink	NaN	L

↳ Both or those gets all in A, all in B,

→ pd.merge (A, B, how='left')

ou

	color	num	size
0	green	1	S
1	yellow	2	M
2	red	3	NaN

common, all in left

→ pd.merge(A, B, how='right')

ou

	color	num	size
0	green	1	S
1	yellow	2	M
2	Pink	NaN	L

common, all in right

changes in pandas

① to change to type category (whole ready).

```
drinks = pd.read_csv('https://bit.ly/drinks by country',
                     dtype = {'continent': 'category'})
```

col name.

② to convert multiple datatypes at once

```
drinks = drinks.astype({'beer_servings': 'float',
                        'spirit_servings': 'float'})
```

③ apply multiple aggregation

old way

```
drinks.groupby('continent').beer_servings.agg
                                (['mean', 'max', 'min'])
```

new way → on series

```
drinks.beer_servings.agg(['mean', 'min', 'max'])
```

→ col name

→ more flexible than describe()

also

```
drinks.agg(['mean', 'min', 'max'])
```

→ on dataframe

④ isnull() - deprecated.

→ [new]

⑤ isnull() ⟷ isna()

notnull() ⟷ notna()

⑥ → on drop the need do we axis,

ufo.drop([0,1], axis=0) ✗ → no need

ufo.drop(index=[0,1]) — (same for columns)

⑦ → for rename.

ufo.rename(columns={'city':'CITY'}) → old way

ufo.rename({'city':'CITY'}, axis=1) → new way

⑧ from pandas.api.types import CategoricalDtype

```
quality_cat = CategoricalDtype(['good', 'vs', 'ideal'], ordered=True)
df['quality'] = df.quality.astype(quality_cat)
```

→ [to give a categorical order]