Data Wrangling

Bhadri Vaidhyanathan – Personal Notes

Contents

Clean up: 3

Datetime 3

List 3

Dictionary 4

Dataframes 5

Vlookup 5

Selection/Subset of Dataframe: 5

Slicing - Filter Row by Row 7

Select rows and columns of data frame / slice 7

Cosmetic changes to Dataframe: 8

Rearrange column order 8

Preprocessing: 8

Replace characters : 8

replace / rename similar values in a column 8

NANs / Nulls values 8

String nan 9

df.fillna() 9

df.dropna() 9

Drop columns 10

Drop Rows with Zero Length Strings 10

Drop rows with threshold no. of value 10

Drop rows – condition 10

Remove rows – string column contain substring 11

Row by row: 11

iterrows(),apply() 11

.apply() 11

Duplicates 12

Conversions: 12

Convert str to int 12

Convert strings with comma to numeric 13

EDA methods: 13

Value_counts: 13

NUnique() 13

Codify Categorical manually: 14

Binning numeric values to categorical variable 14

Length of text in new column: 14

Remove Rows with Zero Length Strings 14

Correlations 14

EDA Visualization tips: 15

GroupBy 15

Feature Engineering: Creating a new column 15

df.assign 15

Clean up:

- convert object (/string) to datetime
    - pd.to_datetime(df[<column>])

Datetime

- derive transaction hour from date time
    - df['date_time'].dt.hour
- derive day name
    - df['date_time'].dt.day_name()
- derive year-month alone from date time
    - df['date_time'].dt.to_period('M')
        - and groupby count of transactions
            - df_timeline01 = df.groupby(df['year_month'])[['trans_num','cc_num']].nunique().reset_index()
              df_timeline01.columns = ['year_month','num_of_transactions','customers']
- derive age
    - np.round((df[“transaction_time”] - df['dob'])/np.timedelta64(1,'Y'))

List

List comprehension:

new_list = [expression for item in iterable if condition == True]

With IF..ELSE()

List2 = [f(x) if condition else g(x) for x in sequence]

Difference between two lists
where List2 is the bigger list and elements in list1

list3 = list(set(list_big) - set(list_small))
Above, any duplicates are removed and besides order of elements also reset.
If order and duplicates are to be retained then use below.

or

ys = set(y)
list3 = [item for item in x if item not in ys]

Dictionary

Create empty dictionary:

```
dict = { k:[] for k in ['name', 'address',  'phone'] }
```

then append values later with:

```
         dict[“name”].append(‘john’)
         dict[“address”].append(‘Chicago’)
```

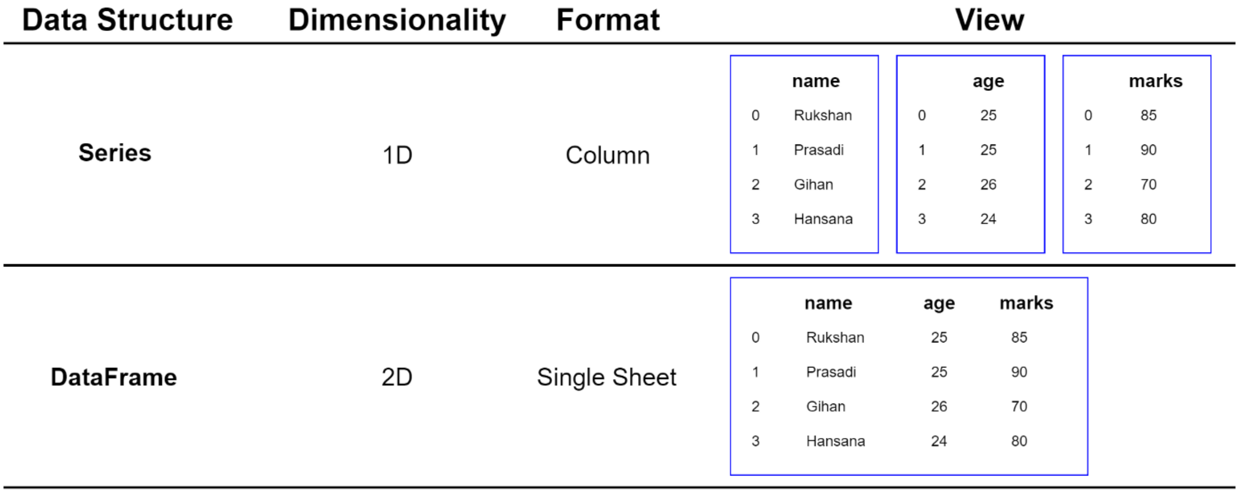
Dictionary Comprehension

```
dict = {key: process_fn(key) for key in blob.words}
# create dict processing key in a list or column using process_fn() and put the output as
value for the key
```

print only top values of a big dictionary like df.head()

```
dict(list(my_dict.items())[:3])
```

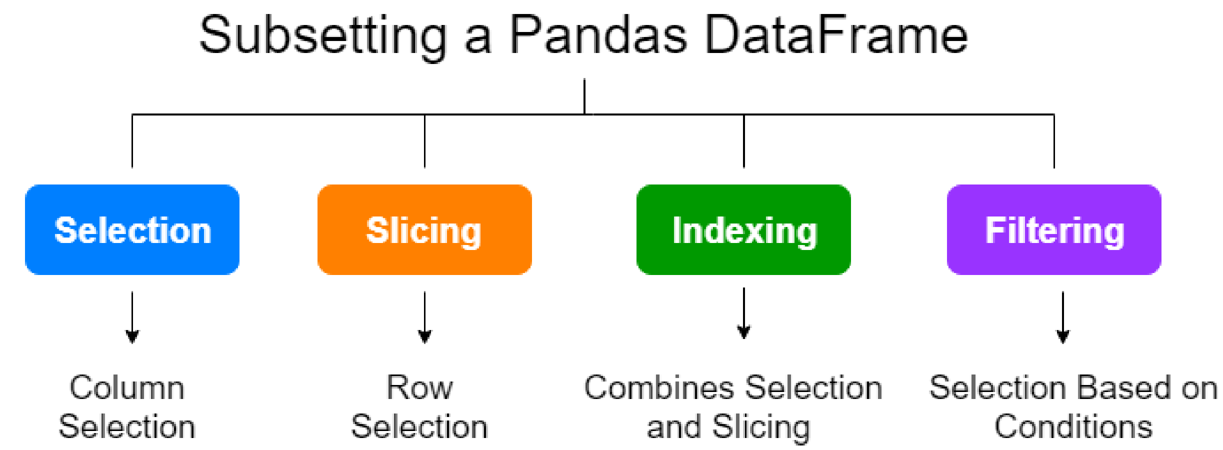
.items() returns an iterator and so converted to a list first



Data Structure	Dimensionality	Format	View
Series	1D	Column	name 0 Rukshan 1 Prasadi 2 Gihan 3 Hansana age 0 25 1 25 2 26 3 24 marks 0 85 1 90 2 70 3 80
DataFrame	2D	Single Sheet	name age marks 0 Rukshan 25 85 1 Prasadi 25 90 2 Gihan 26 70 3 Hansana 24 80

Dataframes

https://towardsdatascience.com/23-efficient-ways-of-subsetting-a-pandas-dataframe-6264b8000a77


Subsetting a Pandas DataFrame
Selection
Slicing
Indexing
Filtering
Column Selection
Row Selection
Combines Selection and Slicing
Selection Based on Conditions

Vlookup

- merge entire table and with only specific column and foreign key
  - pd.merge(df1, df2[[<foreign key>,<column needed>]], left_on=<key>,right_on=<foreign key>,how='left')

merge is equivalent to SQL joins

- Using .map:
  df_longertable['new_col'] = df_longertable.user_id.map(df_small.type)

Selection/Subset of Dataframe:

A. Single value

1) Simple approach to accessing a specific value in df:

Df['column name'][row_index_value]

Note: this method is not very reliable and besides here is it [column] [row] while in more widely used syntaxes it is [row, col]

2) loc/iloc

df.loc['row_label', 'column_label']

Access a group of rows and columns by label(s) or a boolean array.

.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a':'f' Note: here that contrary to usual python slices, both the start and the stop are included

df.iloc['row_index', 'column_index']

B. Single Column as Series

1) Specific column as Series:

```
col = df[‘<column>]
```

A series item is returned

2) Single/Multiple columns as DF

```
mul_cols = df[ [<col1>,<col2>] ]  #note double brackets []
```

A dataframe item is returned

The first looks for a specific Key in your df, a specific column, the second is a list of columns to sub-select from your df so it returns all columns matching the values in the list.

3) .loc

Syntax: df.loc['row_label', 'column_label']

single:		df.loc[:, 'alcohol'] - **Returned as Series**

Multiple:		df.loc[:, ['alcohol', 'ash', 'hue']] - **Returned as DF**

4) .iloc

Syntax: df.iloc['row_index', 'column_index']

Single: df.iloc[:, 0]   - **Returned as Series**

Multiple: df.iloc[:, [0, 2, 10]] - **Returned as DF**

Slicing - Filter Row by Row

1) Condition

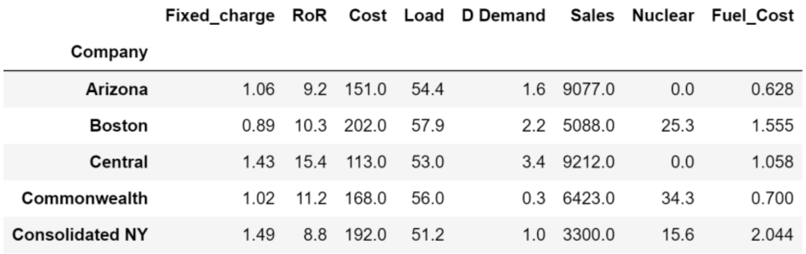
```
df = df  [  df[‘column]     >=   1000 ]
```

The condition within outer [] returns a series of Boolean values which is used like a index to include/exclude each row and output

2) isin()

```
new_df = df[   df["col"].isin([2, 3])    ]
```

Select rows and columns of data frame / slice



	Fixed_charge	RoR	Cost	Load	D Demand	Sales	Nuclear	Fuel_Cost
Company								
Arizona	1.06	9.2	151.0	54.4	1.6	9077.0	0.0	0.628
Boston	0.89	10.3	202.0	57.9	2.2	5088.0	25.3	1.555
Central	1.43	15.4	113.0	53.0	3.4	9212.0	0.0	1.058
Commonwealth	1.02	11.2	168.0	56.0	0.3	6423.0	34.3	0.700
Consolidated NY	1.49	8.8	192.0	51.2	1.0	3300.0	15.6	2.044

```
df_clusterA = utilities_df_norm.loc[slice("Arizona","Boston"), ["Sales","Fuel_Cost"]]

df_clusterB = utilities_df_norm.loc[["Central","Commonwealth","Consolidated NY"],
["Sales","Fuel_Cost"]]
```

Make New Column:

1) Make New Col based on presence in list

df['new_bool_col'] = df['user_id'].apply(lambda x: 1 if x in buyers_list else 0)

2) Make Boolean col based on sum of 3 cols being > 0

df['new_bool_col'] = df[['col1', ‘col1’, ' col3']].sum(axis=1).apply(lambda x: 1 if x > 0 else 0)

3) Compare 2 columns and 0 if columns are same and 1 if different

df['Comparison'] = df.apply(lambda row: 0 if row['Column1'] == row['Column2'] else 1, axis=1)

Cosmetic changes to Dataframe:

Rearrange column order

df.reindex([<the new ordered list of columns>], axis="columns")

note: changes are inplace

Preprocessing:

Replace characters :

In whole df:
import numpy as np
df = df.replace("?", np.nan).copy()

replace / rename similar values in a column

```
df.columnname.replace(‘value_to_be_changed’, ‘new_value’)
```

NANs / Nulls values

(1) Using **isna()** to select all rows with NaN under a *single* DataFrame column:

df[df['column name'].isna()]

(2) Using **isnull()** to select all rows with NaN under a *single* DataFrame column:

- df[df['column name'].isnull()]
- df.isnull().sum(axis = 1)

(3) Using **isna() and isna().sum()** to select all rows with NaN under an *entire* DataFrame:

```
df.isna().sum()   #list all the columns and total of NA count

#or

df[df.isna().any(axis=1)]
```

(4) Using **isnull()** to select all rows with NaN under an *entire* DataFrame:

```
df[df.isnull().any(axis=1)]

```

String nan

when preprocessing text and saving to column, sometime ‘nan’ gets saved as a string dtype and it does not get detected with isna() or dropna() but below works

```
df0.fillna('', inplace=True)
```

df.fillna()

simple, identify cols with na, create a list of these cols, use the list name as below to fill it

```
cols_to_fill_zero = [‘column1’, ‘column2’, ‘column3’]
df[cols_to_fill_zero] = df[cols_to_fill_zero].fillna(0)
```

a. fillna with the most common value of a column

```
df = df.apply(lambda x:x.fillna(x.value_counts().index[0]))
```

b. fillna with mean of the col: done column by column and not in mass

```
df[column1] = df[column1].fillna( df.column1.mean() )
```

df.dropna()

Drop all rows having at least one null value

pandas.DataFrame.dropna() : drop all rows with at least one missing value.

Drop rows having only missing values

columns’ values are all null, then how='all'

Drop rows where specific column values are null

If only specific columns, then subset argument.

For instance, let’s assume we want to drop all the rows having missing values in any of the columns colA or colC :

```
df = df.dropna(subset=['colA', 'colC'])
print(df)
```

Additionally, you can even drop all rows if they’re having missing values in both colA and colB:

```
df.dropna(subset=['colA', 'colB'], how='all', inplace=True)
```

```
print(df)
```

Drop columns

```
del train_df['PassengerId']
```

Drop Rows with Zero Length Strings

```
df = df[~df[‘colA’].eq('')]
df = df[df[‘colA’].ne('')] #ne – not equal
```

ID columns with string (na or zero length)

```
missing = (data['text'].isna()) | (data['text'].str.len() == 0)  #generator?
```

Drop rows with threshold no. of value

Finally, if you need to drop all the rows that have at least N columns with non- missing values, then you need to specify the thresh argument that specifies the number of non-missing values that should be present for each row in order not to be dropped.

For instance, if you want to drop all the columns that have more than one null values, then you need to specify thresh to be len(df.columns) — 1

```
df = df.dropna(thresh=len(df.columns)-1)
```

```
print(df)
   colA  colB colC  colD
1  False   2.0    b   2.0
2  False   NaN    c   3.0
3   True   4.0    d   4.0
```

Drop rows – condition

```
df = df.drop(some labels)
df = df.drop(df[<some boolean condition>].index)
```

Single condition:

```
df.drop(df[df.score < 50].index, inplace=True)
```

Multiple condition: & | ~

```
df = df.drop(df[(df.score < 50) & (df.score > 20)].index)
```

Remove rows – string column contain substring

1: select Rows that Contain a Specific String

```
df[df["col"].str.contains("this string")==False]  #errors out
```

```
mask =  df.apply(lambda x: x.str.contains('Chicago').any(),axis=1)  #axis=0 selects columns
```

```
sub_df = df.loc[: , mask]
```

2: seelct Rows that Contain a multiple Strings

```
df[df["team"].str.contains("A|B")==False]
```

3. **remove Rows that Contain a Partial String**

```
#identify partial string to look for
discard = ["Wes"]
#drop rows that contain the partial string "Wes" in the conference column
df[~df.conference.str.contains('|'.join(discard))]
```

Row by row:

iterrows(),apply()

https://medium.com/@filipgeppert/for-loop-in-pandas-a-k-a-pd-apply-e4a53f62d78d

NOTE: When using iterrows and UDF

How does .iterrows() work?

```
for index, row in df.iterrows():
    # Access any cell in row and set it to 0
    df.loc[index, 'column_name'] = 0

for index, row in df.iterrows():
    # Access any cell in row and set it to 0
    # Check if value in cell fulfils condition
    if df.loc[index, 'column_name'] == 1:
        df.loc[index, 'column_name'] = 0
    else:
        pass
```

.apply()

```
df[‘new column’] = df[‘old’].apply(lambda x: re.findall(“\s\S@+\.\w{2,3}”,
text)
```

.apply() is a Pandas way to perform iterations on columns/rows. It takes advantage of vectorized techniques and speeds up execution of simple and complex operations by many times.

```
df[‘month’] = df['date'].apply(lambda x: x.month)
```

.apply() with condition

```
# Create function that checks multiple conditions
def extract_month(x):
    month = x.month
    if month == 11:
        return “It’s november. So cold!”
    elif month == 6:
        return “It’s june. I love sun!”
    else:
        return “It’s ok.”# Apply function on column
df[‘month’] = df.date.apply(extract_month)
```

.apply() on multiple cells in row

```
# Calculate mean price per day
def mean_cost_per_day(row):
    mean_cost = np.mean([row.price_day, row.price_night])
    return mean_costdf[‘mean_cost_per_day’] = df.apply(mean_cost_per_day, 
axis=1)
```

Row by Row : Conditional String manipulation

```
df['new_col'] = 
df['old_col'].where(df['old_col'].str.startswith('1.4'), df['old_col'].str[:3])
```

Duplicates

sum(epl_matches.duplicated())

Conversions:

Convert str to int

Note: cells with N/A will throw error for astype()

astype (all values must be convertible to int else error)

```
df['Price'] = df['Price'].astype(int)  #singlecolumn
```

```
df[ ['Price',’date’,’name’] ] = df[['Price',’date’,’name’]].astype(int)  #multiple
and this does not lead to losing columns not mentioned in the subset. Rest of df is
retained
```

pd.to_numeric

Note: cells with comma in the string ‘1,000,004’ will be replaced by NaN

```
df['Price'] = pd.to_numeric(df['Price'],errors='coerce')  #coerce puts non-
convertible values as Nan

df[cols] = df[cols].apply(pd.to_numeric, errors='coerce')  #single

df[ ['Price',’date’,’name’] ] = df[['Price',’date’,’name’]].apply(pd.to_numeric,
errors='coerce')  #multiple
```

Convert strings with comma to numeric

```
df.apply(lambda x: x.str.replace(',', '').astype(float), axis=1)
```

EDA methods:

Describing DF - https://medium.com/codex/9-efficient-ways-for-describing-and-summarizing-a-pandas-dataframe-316234f46e6

Inspecting DF - https://medium.com/codex/10-efficient-ways-for-inspecting-a-pandas-dataframe-object-3f66563e2f2

Value_counts:

Groupby distinct items and count

```
train_df['Embarked'].value_counts()
```

```
S   644
C   168
Q    77
Name: Embarked, dtype: int64
```

NUnique()

print unique item count

```
td.nunique()
```

```

Output:
PassengerId    1309
Survived          2
Pclass            3
Name           1307
Sex               2
Age              98
SibSp             7
Parch             8
Ticket          929
Fare            281
Cabin           186
Embarked          3
dtype: int64
```

Codify Categorical manually:

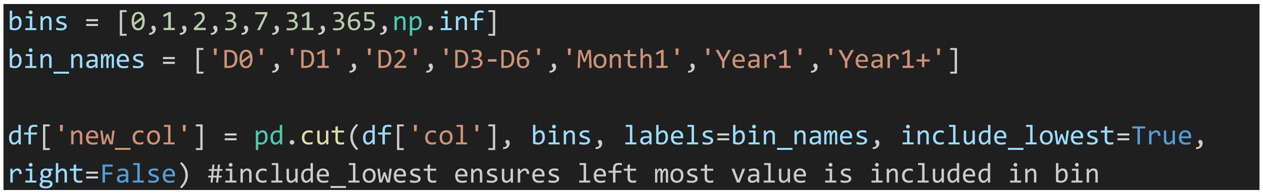
# convert label to a binary numerical variable

```
citation['violation_flag'] = citation.violation.map({'Warning':0, 'Citation':1, 'ESERO':2})
```

Binning numeric values to categorical variable

#convert continuous numeric data to categorical

```
df ['new] = pd.cut(x=df ['polarity'],bins=[-1, -0.05, 0.05, 1],
labels=['Negative', 'Neutral', 'Positive'])
```



```
bins = [0,1,2,3,7,31,365,np.inf]
bin_names = ['D0','D1','D2','D3-D6','Month1','Year1','Year1+']

df['new_col'] = pd.cut(df['col'], bins, labels=bin_names, include_lowest=True,
right=False) #include_lowest ensures left most value is included in bin
```

Col.replace()

Replace categorical values with numbers or other

```
sampleDF.housing.replace(('yes', 'no'), (1, 0), inplace=True)
```

Length of text in new column:

```
passfail_df['comm_length'] = passfail_df['violation comments'].apply(len)
```

Remove Rows with Zero Length Strings

```
df = df[~df[‘colA’].eq('')]
```

Correlations

Remove correlated features to reduce multi-collinearity

EDA Visualization tips:

SNS Missing values Heatmap:

```
import seaborn as sns
td.isnull().sum()
sns.heatmap(td.isnull(), cbar = False).set_title("Missing values heatmap")
```

GroupBy

Groupby and then convert the grouped data into a dataframe

```
df_new = df_old.groupby(“col1”)['event_name'].apply(','.join).to_frame()
```

Feature Engineering: Creating a new column

df.assign

df.assign(new_col_name= value/formula on existing_col/lamda of another column)

```
>>> df
                temp_c
Portland        17.0
Berkeley        25.0

>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)

        temp_c  temp_f
Portland   17.0   62.6
Berkeley   25.0   77.0
```

Or directly refer to multiple columns with a formula

```
df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
          temp_k=lambda x: (x['temp_f'] + 459.67) * 5 / 9)
```

```
          temp_c  temp_f  temp_k
Portland    17.0    62.6  290.15
Berkeley    25.0    77.0  298.15
```