Pandas

A Gentle Introduction to Pandas Data Analysis (on Kaggle) -
https://www.kaggle.com/code/robikscube/pandas-introduction-youtube-tutorial?scriptVersionId=94752062

Economic Data Analysis Project with Python Pandas - Data scraping, cleaning and exploration! -
https://www.youtube.com/watch?v=R67XuYc9NQ4&ab_channel=RobMulla

1. Take note of index in dataframe

```
df.to_csv('output.csv', index=False) # Remember to use index=False when writing to csv, to remove the left hand indexes
#or
df = pd.read_csv(’output.csv’, index_col=[0) # In the first place, when you read the csv, you can remove unamed index
```

2. Don’t use columns names that contain spaces. You will lose ability to do this. df.First_Initial
3. Make use of df.query to simplify your code

```
df.loc[(df['Year'] < 1980) & (df['Time' > 10)] # this is good but not good enough
df.query('Year < 1980 and Time > 10') # this is using query power, better when your query is complex
```

4. Call out variables using @

```
min_year = 1980
min_time = 10
df = df.query(f'Year < {min_year} and Time > {min_time}') # this is a bit older
df = df.query('Year < ' + str(min_year) + 'and Time > ' + str(min_time))
# the above methods arent necessary as pandas queries can acess external variables by simply using the @ symbol
# this is better
df = df.query('Year < @min_year and Time > @min_time')
```

5. Don’t use inplace, not good practice

```
# not good # will overwrite df unknowingly
df.fillna(0, inplace = True)
df.reset_index(inplace = True)
# this is better
df = df.fillna(0)
df = df.reset_index()
```

6. Iterating over the rows in the data frame when vectorization is an option is bad

```
for i, row in df.iterrows():
  if row['Year'] > 2000:
    df.loc[i, 'is recent'] = True
...
```

```
# this is better
df['result'] = df['Year] > 1980
```

7. Using apply method is not so good too. Vectorization is better

```
df['year_square'] = df.apply(lambda row: row['Year'] ** 2, axis=1)
df['year_square'] = df['Year'] ** 2
```

8. copy() method

```
df_fast = df.query('Time < 10')
df_fast['First Name'] = df_fast['Name'].str[=5:]
# this will case set with copy warning.
# this warning occurs because the new modifications are actually being applied to a slice of our old data frame.
# use copy() method like below instead -- create a deep copy
df_fast = df.query('Time < 10').copy()
df_fast['First Name'] = df_fast['Name'].str[=5:]
```

9. Don't make intermediate dataframes during transformations. Use chaining commands like query, groupby, sort_values in 1 line
10. Although pandas will try its best to find the d types. You cannot just assume, need to check.

```
df.info() # To check the d types
# most often dates are not parse properly
df = pf.read_csv('abc.csv, parse_dates=['Date']
df.info()
#OR
df = pf.read_csv('abc.csv')
df['Date'] = pd.to_datetime(df['date'])
df.info()
```

11. Use boolean values (True, False) instead of introducing str value like (Yes, No)
12. Use the inbuilt pandas plot method when you need to plot graphs
13. Don't use str methods like this

```
df['Name_Uppercase'] = df['Name'].apply(lambda x: str(x).upper())
# instead use pandas str tmethods
df['Name_Uppercase'] = df['Name'].str.upper()
```

14. Don't repeat transformation codes for different dataframes in a single script

```
# pls write functions instead # below is an example

import pandas as pd

def process_data(df):
  df['Time Norm'] = df['Time'] / df['Time'].mean()
  df['Place'] = df['Place'].str.lower()
  return df

dfw = pd.read_csv('womens100m.csv')
```

```
dfm = pd.read_csv('mens100m.csv')

dfw = process_data(dfw)
dfm = process_data(dfm)
```

15. Don’t manually replace the columns name one by one

```
import pandas as pd

df = pd.read_csv('mens 100m.csv')
print('Old Columns:', df.columns.tolist())
df.columns = ['Race_Year', 'Time', 'Athlete', 'Place', 'Name']
print('New Columns:', df.columns.tolist())
df.head()

#instead use rename method
df = pd.read_csv('mens100m.csv')
df = df.rename (columns={'Year':'Race_Year'}) df.head()
```

16. Use groupby method instead

```
df = pd.read_csv('mens_womens_100m.csv')
mens_record = df.loc[df['Grouping'] == 'Mens']['Time'].min()
womens_record = df.loc[df['Grouping'] = == 'Womens']['Time'].min()
print (mens_record, womens_record)

#instead use groupby method
df = pd.read_csv('mens_womens_100m.csv')
df.groupby('Grouping')['Time'].min()
```

17. Using iteration again

```
import pandas as pd
import numpy as np
df = pd.read_csv('mens_womens_100m.csv')

mens_times = []
womens_times = []
# Get mens and womens average
for i, row in df.iterrows():
  group = row.Grouping
  if group = "Mens":
    mens_times.append(row. Time)
  else:
    womens_times.append(row. Time)

avg_mens = np.mean (mens_times)
avg_womens = np.mean (womens_times)
mens_count = len(mens_times)
womens_count = len(womens times)

print (avg_mens, avg_womens, mens_count, womens_count)

# instead use groupby
# then follow by using agg method which allow you to use more agg functions in a line of code
# instead method like .count , .mean , only allow you to perform 1 agg function in a line of code

df.groupby('Grouping')['Time'].agg(['mean','count'])
```

18. Using loop to find how value changes

```
import pandas as pd
df = pd.read_csv('womens100m.csv')
for i in range(1, len (df)):
  df.loc[i, 'perc_change'] = \
    (df.loc[i].Time - df.loc[i -1].Time) / (df.loc[i -1].Time)
  df.loc[i, 'diff'] = (df.loc[i].Time - df.loc[i - 1].Time)

df.head()

#instead there is actually pct_change() method and diff() method
import pandas as pd
df = pd.read_csv('womens100m.csv')
df['perc_change'] = df['Time'].pct_change()
df ['change'] = df['Time'].diff()
df.head()
```

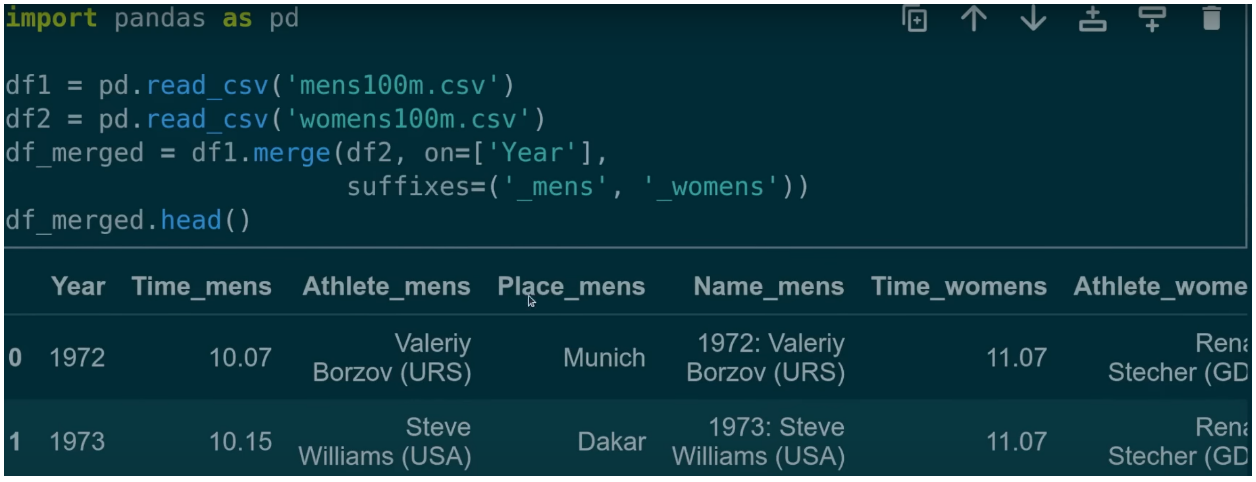
19. Saving huge file to csv. Pandas have other formats like parquet, feather, pickle

```
import pandas as pd
large_df = pd.read_csv('large_dataset.csv')
large_df.to_parquet ('output.parquet')
large_df.to_feather('output.feather')
large_df.to_pickle('output.pickle')

#retains the data type as well
```

20. Pandas dataframe has style formatting to do things like HTML style format (not very useful)

21. When merging 2 dataframes, use suffixes please, to track which column belongs to which df (Don’t think data eng should do this)

```
import pandas as pd

df1 = pd.read_csv('mens100m.csv')
df2 = pd.read_csv('womens100m.csv')
df_merged = df1.merge(df2, on=['Year'], suffixes=('_mens', '_womens'))
df_merged.head()
```



```
import pandas as pd

df1 = pd.read_csv('mens100m.csv')
df2 = pd.read_csv('womens100m.csv')
df_merged = df1.merge(df2, on=['Year'],
                      suffixes=('_mens', '_womens'))
df_merged.head()
```

	Year	Time_mens	Athlete_mens	Place_mens	Name_mens	Time_womens	Athlete_wome
0	1972	10.07	Valeriy Borzov (URS)	Munich	1972: Valeriy Borzov (URS)	11.07	Rena Stecher (GD
1	1973	10.15	Steve Williams (USA)	Dakar	1973: Steve Williams (USA)	11.07	Rena Stecher (GD

22. Use validate parameter to check if merging rows is matching

```
import pandas as pd

#you may want to check after merge the len of the dataframe is still 1:1 match with your old df
df1 = pd.read_csv('mens 100m.csv')
df2 = pd.read_csv('womens 100m.csv')
df_merged = df1.merge(df2, on=['Year'], suffixes=('_mens', 'womens')
len(df_merged) == len(df1)

#instead use validate parameter
df_merged = df1.merge(df2, on=['Year'], suffixes=('_mens', 'womens'), validate='m:1')

#this will throw a MergeError if failed validate condition
```

23. Don't chain all your methods into 1 line . Spilt your code for it to be more readable

```
df = pd.read_csv('mens_womens_100m.csv')
df_agg = (
  df
  .groupby(['Grouping', 'Year'])['Time']
  .min()
  .reset_index()
  .fillna(0)
  sort_values('Year')
)
df_agg
```

24. If your column only has a few potential values like 'Men' and 'Women'. Instead of storing columns like this as string object, better to store them as categorical data type as it take up less space and can make operations much faster on large data sets. (Don't think data eng should do this)

```
import pandas as pd
df = pd.read_csv('mens_womens_100m.csv')
# lets say grouping only have 2 variables: 'men', 'female'
# can consider change the object type to category type
df['Grouping'] = df['Grouping'].astype('category')
df.info()
```

25. Don't use 2 duplicated columns

```
import pandas as pd
df = pd.read_csv('mens_womens_100m.csv')
df_double = pd.concat([df, df], axis=1)
df_double['Year']

# Pandas has a flag to help you debug
#return DuplicateLabelError if found duplicate columns
df = pd.read_csv('mens_womens_100m.csv')
df_double = pd.concat([df, df], axis=1) \
    .set_flags(allows_duplicate_labels=False)
df_double['Year']


#If you want to prevent it, remove duplicate column beforehand
import pandas as pd
pd.read_csv('mens_womens_100m.csv')
df_double = pd.concat([df, df], axis=1)
df_double = df_double.loc[:,~df_double.columns.duplicated()].copy()
df_double['Year']
```

26. NamedAgg trick

```
# We have performed the desired calculations but there is a little problem.
# We can’t really tell which aggregate function is applied to which column.
# One solution is to rename the columns afterwards. For instance, we can rename them as “avgSalary” and “totalSpent”.
df[['Age','Salary','AmountSpent']].groupby('Age')\
.agg({'Salary':'mean', 'AmountSpent':'sum'}).round(2)

# There is a more practical solution which is our NamedAgg trick.
# It allows for customizing the names of the aggregated columns. Here is how we implement it in the agg function.
df[['Age','Salary','AmountSpent']].groupby('Age')\
.agg(
   avgSalary = pd.NamedAgg('Salary','mean'),
   totalSpent = pd.NamedAgg('AmountSpent','sum')
)
```

27. Convert sql to pandas

```
# Groupby sql
SELECT state, count(name)
FROM df
ORDER BY state;

#equivalent in pandas
n_by_state = df.groupby("state")["last_name"].count()
n_by_state.head(10)
```