Python / Pandas / Numpy

Pandas & Numpy

Linear Regression

```
import statsmodels.api as sm
X_const = sm.add_constant(X)
model = sm.OLS(y_train, X_train_const)
results = model.fit()
results.summary()
```

```
# 100, 10
import numpy as np
X = np.random.normal(size=(100, 10))
# print(X[0:5,])
meanX = np.mean(X, axis=0)
# print(meanX)
stdX = np.std(X, axis=0)
# print(stdX)

weights = np.random.uniform(size=(100, 1))

weightedX = np.dot(weights.T, X)
weightedMeanX = weightedX / np.sum(weights)
# print(weightedMeanX)

Y = np.random.normal(size=(100, 1))
# Y = X * b
b = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), Y)
print(b)
```


Basic

pd.to_frame() / `pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})`

to_datetime()
activity_2nd_day = activity.loc[activity["first"] + pd.DateOffset(1) == activity["event_date"]]

df['date'].dt.date

df.value_counts()

And: `df[(df['col1'] > 5) & (df['col1'] < 10)]` (not &&!), or use |, not use ~

Add new column: df[‘new_col’] = df[‘col1’].apply(lamba x: pd.to_datetime(x[1:10]))
set(df[‘sym’])
Drop row by index: df.drop([0, 1]) / df.drop(index=('row1', 'row2'))
df.transform(lambda x: x * 2) / df.transform('sqrt')
df['col'] = df['col'].round(2)

df.reset_index(): made current index a regular column, and create new index
`DataFrame.reset_index(level=None, *, drop=False, inplace=False, col_level=0, col_fill='', allow_duplicates=<no_default>, names=None)`

NA

Fill table with id not found

```
priceIds = set(prices['product_id'].unique())
soldIds = set(units_sold['product_id'].unique())
missingIds = priceIds.difference(soldIds)
fill = pd.DataFrame({'product_id': list(missingIds), 'average_price': [0]*len(missingIds)})
return pd.concat([main, fill])
```

notna() / isna()
Drop NAN: `df.dropna()` or `df[df['col'].notna()]` or `df.loc[df['B'].notnull()]`

```
# Fill NaN values with a specific value (e.g., 0)
df_filled_0 = df.fillna(0)
# Fill NaN values with the mean of each column
df_filled_mean = df.fillna(df.mean())
# Fill NaN values using forward fill (propagate the last valid observation forward)
df_filled_ffill = df.fillna(method='ffill')
# Fill NaN values using backward fill (use the next valid observation to fill the gap)
df_filled_bfill = df.fillna(method='bfill')
```

Columns

Return selected columns: df[[‘c1’, ‘c2’]]
Rename: `df = df.rename(columns={'col1': 'new_col1', 'col2': 'new_col2'})`
Rename: `df.columns = ['new_col1', 'new_col2']`
Rename: `df = df.rename(columns=lambda x: x.replace('col', 'new_col'))`

df.drop, drop column: `df.drop(columns=['B', 'C']) / df.drop(['B', 'C'], axis=1)`

Duplicates

Find duplicates:

df.loc[df.duplicated('email') == True]

Select rows with no duplicated lat/lon

```
df = insurance[~insurance.duplicated(subset=['lat', 'lon'], keep=False)]
df_2015 = insurance[insurance.duplicated(subset=['tiv_2015'], keep=False)]
```

DataFrame.duplicated(subset=None, keep='first')
Keep: {‘first’, ‘last’, False}, default ‘first’
first : Mark duplicates as True except for the first occurrence.
last : Mark duplicates as True except for the last occurrence.
False : Mark all duplicates as True.

```
# Remove duplicate rows based on all columns
df_no_duplicates = df.drop_duplicates()

# Remove duplicates based on specific columns
df_no_duplicates_subset = df.drop_duplicates(subset=['col1'])

# Remove duplicates, keeping the last occurrence
df_no_duplicates_last = df.drop_duplicates(keep='last')

# Remove duplicates in place (modifies the original DataFrame)
df.drop_duplicates(inplace=True)
```

df.drop_duplicates(subset=None, *, keep='first', inplace=False, ignore_index=False)

Select / Filter

df.iloc[[1]]: second row
Select rows: df[df[‘code’] > 100] / `df[df['col1'].isin([1, 3, 5])]`

Certain Value in one df, but also in / not in another

```
rkdcode_0100[~rkdcode_0100['rkdcode'].isin(set(rkdcode_0341['rkdcode']))].head()
```

```
df_str[df_str['col1'].str.contains('a')]

filtered_df = df.loc[df['col2'] % 2 == 0]
# Select Rows Between 'Row_2' and 'Row_4'
```

```
selected_rows = df.loc['Row_2':'Row_4']
# Select Columns 'B' through 'D'
selected_columns = df.loc[:, 'B':'D']
result = df.loc['Row_2', 'Col_Name'] (return that one value)
```

Select some columns:

```
result = df.loc[:, ['Col_A', 'Col_D']]
# Filter columns by name
filtered_df_items = df.filter(items=['col_A', 'col_C'])
# Filter columns containing 'B'
filtered_df_like = df.filter(like='B')
# Filter columns matching the regular expression 'col.*'
filtered_df_regex = df.filter(regex='col.*')
# Filter rows with index 0
filtered_df_row = df.filter(items=[0], axis='index')
```

Merge

```
pd.merge(df1, df2, on='ID', how='left')
```

pd.merge(table1,table2, how='inner'', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=None, indicator=False, validate=None)

Sort

nlargest() / nsmallest() / sort_index()

```
# Selecting top 3 rows with highest 'Rank'
res = df.nlargest(3, ['Rank'])
df.sort_index(ascending=False)
```

sort_values()

```
# Sort by a single column
df_sorted_col1 = df.sort_values(by='col1')
# Sort in descending order
df_sorted_desc = df.sort_values(by='col1', ascending=False)

# Sort by multiple columns
df_sorted_multi = df.sort_values(by=['col1', 'col2'])
# Sorting by 'Rank' in ascending order and 'Age' in descending order
res = df.sort_values(by=['Rank', 'Age'], ascending=[True, False],
na_position='last')

# Sort the DataFrame in place
```

```
df.sort_values(by='col1', inplace=True)
```

```
df.sort_values(by, *, axis=0, ascending=True, inplace=False,
kind='quicksort', na_position='last', ignore_index=False,
key=None)
```

Rank

df['rank'] = df['score'].rank(method='dense', ascending=False).astype(int)

```
DataFrame.rank(axis=0, method='average', numeric_only=False,
na_option='keep', ascending=True, pct=False)
```

method{‘average’, ‘min’, ‘max’, ‘first’, ‘dense’}, default ‘average’
How to rank the group of records that have the same value (i.e. ties):

- average: average rank of the group
- min: lowest rank in the group
- max: highest rank in the group
- first: ranks assigned in order they appear in the array
- dense: like ‘min’, but rank always increases by 1 between groups.

GroupBy

```
df.groupby('column_name').mean()
df.groupby(['column_name1', 'column_name2']).sum()
```

Aggregation: sum(), mean(), median(), min(), max(), count(), size(), std(), var()

```
df.groupby('column_name').agg(['sum', 'mean', 'max'])
```

Custom aggregation functions:

```
def my_function(x):
   return x['column_name'].sum() * 2
df.groupby('column_name').apply(my_function)
```

Combine with apply

Purpose: Run any function on each group — very flexible.

```
def add_group_max(df_group):
    df_group['group_max_score'] = df_group['score'].max()
    return df_group
df = df.groupby('group_column').apply(add_group_max)
```

```
employee_filtered = employee.groupby('departmentId').apply(lambda x:
x[x['salary'] == x['salary'].max()]).reset_index(drop=True)
```

Combine with filter

Purpose: Keep or drop **entire groups** based on a condition.

```
# example 1: return a df with only the groups that have more than 10 rows
df.groupby('column_name').filter(lambda x: x['column_name'].count() > 10)
# example 2:
filtered_df = df.groupby('group').filter(lambda x: x['value'].mean() > 3)
```

Combine with transform

Purpose: Return a **Series** with the same shape as the original group, typically to add new columns like group means, max, etc.

```
df_grouped['score_standardized'] =
df_grouped.groupby('subject')['score'].transform(lambda x: (x - x.mean())
/ x.std())
#   subject  score  score_standardized
# 0    math      4           -1.414214
# 1    math      8            1.414214
# 2 english      6           -1.000000
# 3 english      9            1.000000
# 4  physics      5           -1.000000
# 5  physics      7            1.000000
```

df['group_max_score'] = df.groupby('group_column')['score'].transform('max')
transform('max') / transform(lambda x: x * 2)

🔁 Summary Table

Feature	`transform()`	`apply()`	`filter()`
Returns	Series/DataFrame (same shape)	Anything (flexible)	Subset of original DataFrame
Modifies	Row-wise values	Groups (flexible logic)	Keeps/drops entire groups
Typical use	Add group-level stats	Custom transformations	Filter groups by condition
Output size	Same as original	Varies	Smaller or same

Read CSV

Read specific columns: `df = pd.read_csv("people.csv", usecols=["First Name", "Email"])`

Set one column as DF index: `df = pd.read_csv("people.csv", index_col="First Name")`
Parse to datetime: `df = pd.read_csv("people.csv", parse_dates=["Date of birth"])`

pd.read_csv(filepath, sep=<no_default>, delimiter=None, header='infer', names=<no_default>, index_col=None, usecols=None, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=<no_default>, skip_blank_lines=True, parse_dates=None, infer_datetime_format=<no_default>, keep_date_col=<no_default>, date_parser=<no_default>, date_format=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, encoding_errors='strict', dialect=None, on_bad_lines='error', delim_whitespace=<no_default>, low_memory=True, memory_map=False, float_precision=None, storage_options=None, dtype_backend=<no_default>)

Window（shift / rolling）

https://leetcode.com/problems/consecutive-numbers/

df.shift(periods=1, freq=None, axis=0, fill_value=<no_default>, suffix=None)

Create a new column using shift() to show the previous rows value.

```
logs['previous'] = logs['num'].shift(1)
```

Create a new column using shift() to show the next rows value.

```
logs['next'] = logs['num'].shift(-1)
```

```
# Shift all columns down by one row
df_shifted_down = df.shift(periods=1)
# Shift all columns up by two rows
df_shifted_up = df.shift(periods=-2)
# Shift only 'col1' down by one row
df['col1_shifted'] = df['col1'].shift(periods=1)
#Shift multiple periods
df_multi_shifted = df[['col1']].shift(periods=[1,2],suffix='_shifted')
```

df.rolling(window, min_periods=None, center=False, win_type=None, on=None, axis=<no_default>, closed=None, step=None, method='single')

```
# Calculate the rolling mean with a window size of 3
df['rolling_mean'] = df['value'].rolling(window=3).mean()

# Calculate the rolling sum with a window size of 2 and a minimum of 1
period
df['rolling_sum'] = df['value'].rolling(window=2, min_periods=1).sum()

# Apply a custom function to calculate the rolling range
df['rolling_range'] = df['value'].rolling(window=3).apply(lambda x:
x.max() - x.min())
```

Time Series (asof, merge_asof)

DataFrame.asof(where, subset=None): Return the last row(s) without any NaNs before where.

```
s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
s.asof(25) # Output: 2.0
```

merge_asof

Perform a merge by key distance. similar to a left-join except that we match on nearest key rather than equal keys. **Both DataFrames must be sorted by the key**. SORT first

```
merged_df = pd.merge_asof(left_df, right_df, on='time')
```

# merges on matching `by` values, then latest `right_on` <= `left_on`
soldWithPrices = pd.merge_asof(units_sold, prices, by='product_id', left_on='purchase_date', right_on='start_date')

```
pd.merge_asof(left, right, on=None, left_on=None, right_on=None,
left_index=False, right_index=False, by=None, left_by=None,
right_by=None, suffixes=('_x', '_y'), tolerance=None,
allow_exact_matches=True, direction='backward')
```

Direction: ‘backward’ (default), ‘forward’, or ‘nearest’. “backward” search selects the last row in the right DataFrame whose ‘on’ key is less than or equal to the left’s key.

Plot

df[‘col’].plot.kde()

Plot histogram/scatter/line

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 4))
plt.hist(df['target'], bins=50, color='skyblue')
plt.scatter(df["A"], df["B"], alpha=0.4)
plt.plot(df["seconds"], df["far_price"], label="Far", linestyle="-")
plt.title('Target Variable Distribution')
plt.xlabel('Target')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)
plt.show()
```

Plot correlation:

```
import seaborn as sns

plt.figure(figsize=(16, 12))
corr_matrix = df[feature_cols].corr()
sns.heatmap(corr_matrix, cmap='coolwarm', center=0)
sns.histplot(df['target'], bins=100, kde=True, color='darkblue')
plt.title("Feature Correlation with Target")
plt.show()
```

We can also check relationships between variables.

```
# scatter plot
for c in num_cols:
    sns.scatterplot(x=df[c], y=df["y"], s=20, alpha=0.6)
    sns.regplot(x=df[c], y=df["y"], scatter=False, ci=None,
color="black")
    plt.title(f"{c} vs y");
    plt.show()

sns.pairplot(tech_rets, kind='reg')


# categorical
```

- df.nunique() or value counts with df['col'].value_counts()
    - check Imbalance

```
for col in cat_cols:
    print(df[col].value_counts(normalize=True))
    sns.countplot(x=col, data=df)
    plt.show()
```

Plot average by time buckets

```
df['bucket'] = pd.cut(df['seconds_in_bucket'], bins=[0, 200, 400, 600])
df.groupby('bucket')[target_col].mean().plot(kind='bar')
plt.title('Average Target Value by Time Bucket')
plt.ylabel('Mean Target')
plt.show()
```

Plot Boxplot for timeseries

```
df['seconds_bin'] = (df['seconds_in_bucket'] // 10) * 10
plt.figure(figsize=(12, 6))
sns.boxplot(x='seconds_bin', y='target', data=df, palette='Blues')
plt.title('Target Distribution Across Auction Timeline')
```

Numpy

- *`axis=0`: Calculates the mean along each column (vertically).*
- *`axis=1`: Calculates the mean along each row (horizontally).*

```
np.arange(10)

# Generate a 2D array (100 rows, 10 columns) of random numbers
X = np.random.normal(size=(100, 10))
# Mean along axis 0 (columns)
mean_col = np.mean(X, axis=0) # mean of each feature
# Mean along axis 1 (rows)
mean_row = np.mean(X, axis=1)
std_col = np.std(X, axis=0) # mean of each feature
weights = np.random.uniform(size=(100,1))
weightedX = np.dot(weights.T, X)
weightedMeanX = weightedX / np.sum(weights)
Y = np.random.normal(size=(100, 1))
beta =  np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)),X.T), Y)
```

Questions

Rolling/Moving Average

```
ma_day = [10, 20, 50]

for ma in ma_day:
    for company in company_list:
        column_name = f"MA for {ma} days"
        company[column_name] = company['Adj Close'].rolling(ma).mean()
```

Max DrawDown

https://leetcode.com/problems/average-selling-price/solutions/

Merge multiple tables

```
from functools import reduce

#define list of DataFrames
dfs = [ursa_0100_grid_yes_ursa_no,ursa_0100_grid_no_ursa_yes, ursa_0100_match]
#merge all DataFrames into one
ursa_summary = reduce(lambda  left,right: pd.merge(left,right,on=['eqvid'],
                                            how='outer'), dfs)
```

```
ursa_summary
```

	eqvid	Days of grid coverage, no ursa coverage	Days of no grid coverage, ursa coverage	Days of coverage match
0	1035353.0	4893	0	0
1	1036815.0	3007	0	81
2	1037991.0	1205	0	22
3	1038209.0	2862	0	85

weighted monthly average market cap per company

https://stackoverflow.com/questions/66116727/value-weighted-portfolio-returns-per-portfolio

```
# add a new column for company's total dollar value
df['total_dollar_volume'] = df.groupby(['company_id', 'year',
'month'])['dollar_volume'].transform('sum')
# calculate weights
df['weight'] = df['dollar_volume'] / df['total_dollar_volume']
# calculate weighted average by sum the weighted components
df['weighted_component'] = df['market_cap'] * df['weight']
weighted_avg = df.groupby(['company_id', 'year',
'month'])['weighted_component'].sum().reset_index()
```

Can use: `reset_index(name='weighted_avg_market_cap')`

Or

```
def weighted_mean(df, value, weight):
    vs = df[value]
    ws = df[weight]

    return (vs*ws).sum() / ws.sum()
avgPxSeries = soldWithPrices.groupby('product_id').apply(weighted_mean, 'price',
'units').rename('average_price').reset_index()
```

sort by company，月份，market cap，求出来每个月market cap最多的5个公司

```
df_company = df_company.sort_values(by=['year', 'month', 'wCap'], ascending =
[True, True, False])
df_top = df_company.groupby(['year', 'month']).head(5)
```