5.1 - DATA WRANGLING with PANDAS

## **5.1.2 - Manipulating DataFrames with Pandas**

Chapter-1: Extracting and transforming data

|  |  |  |
| --- | --- | --- |
| Indexing Data Frames | | |
| square brackets | df[‘column’][‘row’] | df['salt']['Jan'] |
| column attribute and row label | df.column[‘row’] | df.eggs['Mar'] |
| the .loc accessor | .loc[row, column ] | df.loc['May', 'spam'] |
| the .iloc accessor | .iloc[row, column ] | df.iloc[4, 2] |
| only some columns | df[ [ ] ] | df[['salt','eggs']] |
| Slicing DataFrames | | |
| Selecting a column | df[ ] | df['eggs'] |
| Slicing and indexing a Series | df[column][row] | df['eggs'][1:4] / df['eggs'][4] |
| Using .loc[] | .loc[ ] | df.loc[:, 'eggs':'salt']  df.loc['Jan':'Apr',:]  df.loc['Mar':'May', 'salt':'spam'] |
| Reverse order | .loc['b':'a':-1, :] | p\_counties = election.loc[‘a’:’b’, :]  p\_counties\_rev = election.loc['b':'a':-1, :] |
| Using .iloc[] | .iloc[ ] | df.iloc[2:5, 1:] |
| Using lists |  | df.loc['Jan':'May', ['eggs', 'spam']]  df.iloc[[0,4,5], 0:2] |
| Filtering DataFrames | | |
| Creating a Boolean Series |  | df.salt > 60 |
| Filtering with a Boolean Series | 1st way | df[df.salt > 60] |
| 2nd way | enough\_salt\_sold = df.salt > 60  df[enough\_salt\_sold] |
| Combining filters | Both condition | df[(df.salt >= 50) & (df.eggs < 200)] |
| Either condition | df[(df.salt >= 50) | (df.eggs < 200)] |
| Select **columns** with all nonzeros | .all( ) | df2.loc[:, df2.all()]. ## ‘0’ iceren sutunlari gostermez |
| with any nonzeros | .any( ) | df2.loc[:, df2.any()]. ## ‘0’ iceren sutunlari da gosterir |
| with any NaNs | .isnull().any() | df.loc[:, df.isnull().any()] |
| without NaNs | .notnull().all() | df.loc[:, df.notnull().all()] |
| Drop **rows** with any NaNs | .dropna( ) | df.dropna(how='any') |
|  | “thresh=” keyw | titanic.dropna(thresh=1000, axis='columns') |
| Filtering a column based on another |  | df.eggs[df.salt > 55] |
| Modifying a column |  | df.eggs[df.salt > 55] += 5 |
| Transforming DataFrames | | |
| DataFrame vectorized methods | .floordiv( ) | df.floordiv(12) # Convert to dozens unit ## artik vermez |
| NumPy vectorized functions | np.floor\_divide( ) | np.floor\_divide(df, 12) # Convert to dozens unit |
| Plain Python functions | .apply( ) | def dozens(n):  return n//12  df.apply(dozens) # Convert to dozens unit |
| .apply(lambda …) | df.apply(lambda n: n//12) |
| Storing a transformation |  | df['dozens\_of\_eggs'] = df.eggs.floordiv(12) |
| index | .index | df.index |
| Working with string values | .str.upper() | df.index = df.index.str.upper() |
| .map(str.lower) | df.index = df.index.map(str.lower) |
| Defining columns using others |  | df['salty\_eggs'] = df.salt + df.dozens\_of\_eggs |

Chapter-2: Advance Indexing

|  |  |  |
| --- | --- | --- |
| Index objects and labeled data | | |
| Creating a Series | pd.Series( ) | prices = [10.70, 10.86, 10.74, 10.71, 10.79]  shares = pd.Series(prices) |
| Creating an index | “index=” keyword | days = ['Mon', 'Tue', 'Wed', 'Thur', 'Fri']  shares = pd.Series(prices, index=days) |
| Modifying index name | .name | shares.index.name = 'weekday' |
| Modifying all index entries |  | shares.index = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'] |
| Assigning the index  In [22]: | .index | unemployment.index = unemployment['Zip'] |
| Removing column | del | del unemployment['Zip'] |
| Hierarchical Indexing | | |
| Setting index | . set\_index( ) | stocks = stocks.set\_index(['Symbol', 'Date'])  print(stocks.index) ---🡪 MultiIndex( ……. |
| Sorting index | .sort\_index() | stocks = stocks.sort\_index() |
| Indexing (individual row) | .loc[ ] | stocks.loc[('CSCO', '2016-10-04')] |
| Slicing (outermost index) | .loc[ ] | stocks.loc['AAPL'] |
| outermost index |  | stocks.loc[(['AAPL', 'MSFT'], '2016-10-05'), 'Close'] |
| innermost index |  | stocks.loc[('CSCO', ['2016-10-05', '2016-10-03']), :] |
| Slicing (both indexes) | slice(None) | stocks.loc[(slice(None), slice('2016-10-03', '2016-10-04')),:]  ## to access some inner levels of the index. you need to use slice(None) |

Chapter-3: Rearranging and reshaping data

|  |  |  |
| --- | --- | --- |
| Pivoting DataFrames | | |
| Reshaping by pivoting | .pivot( ) | trials.pivot(index='treatment',  ...: columns=‘gender',  ...: values='response') |
| Pivoting multiple columns |  | trials.pivot(index='treatment', columns='gender') |
| Stacking & unstacking DataFrames | | |
| Creating a multi-level index | set\_index([ ]) | trials = trials.set\_index(['treatment', 'gender']) |
| Unstacking a multi-index | .unstack( ) | trials.unstack(level='gender'). ## index’I column’a kaydiriyor |
| trials.unstack(level=1) |
| Stacking DataFrames | .stack( ) | trials\_by\_gender.stack(level='gender') ## column’dan index’e |
| Swapping levels | .swaplevel( ) | swapped = stacked.swaplevel(0, 1). ## index sirasini degistirir |
| Sorting rows | .sort.index( ) | sorted\_trials = swapped.sort\_index() |
| Melting DataFrames (unpivoting) | | |
| Specifying parameters | .melt( ) | pandas.melt(frame, id\_vars=None, value\_vars=None, var\_name=None, value\_name='value', col\_level=None)[source]¶ |
| pd.melt(new\_trials, id\_vars=['treatment'], value\_vars=['F', 'M'], var\_name='gender', value\_name='response') |
| Pivot tables | .pivot\_table() | more\_trials.pivot\_table(index='treatment', columns='gender', values='response') |
|  |  | **\* pivot\_table** is a generalization of pivot that can handle duplicate values for one pivoted index/column pair. Specifically, you can give pivot\_table a list of aggregation functions using keyword argument aggfunc. The default aggfunc of pivot\_table is numpy.mean.  **\* pivot\_table** also supports using multiple columns for the index and column of the pivoted table. A hierarchical index will be automatically generated for you.  \* pivot() doesn't accept a list for index. / pivot\_table() accepts.  \* Internally, both of them are using reset\_index()/stack()/unstack() to do the job.  \* pivot\_table will only allow numerical types as "values=", whereas pivot will take string types as "values=". |

Chapter-4: Grouping data

|  |  |  |
| --- | --- | --- |
| Categoricals and groupby | | |
| Groupby and count | .groupby( ) | sales.groupby('weekday').count() |
| Groupby and sum | sales.groupby('weekday')['bread'].sum() |
| multiple columns | sales.groupby('weekday')[['bread','butter']].sum() |
| multi-level index | sales.groupby(['city','weekday']).mean() |
| by series | customers = pd.Series(['Dave','Alice','Bob','Alice'])  sales.groupby(customers)['bread'].sum() |
| Categorical data | .astype( ‘category’) | sales['weekday'] = sales['weekday'].astype('category')  ## Uses less memory ##. Speeds up operations like groupby() |
| Groupby and aggregation | | |
| Review: groupby | .sum() / .mean() / .count() | sales.groupby('city')[['bread','butter']].max() |
| Multiple aggregations | .agg( ) | sales.groupby('city')[['bread','butter']].agg(['max','sum']) |
| Custom aggregation | def data\_range(series):  return series.max() - series.min()  sales.groupby('weekday')[['bread', 'butter']].agg(data\_range) |
| By dictionaries | sales.groupby(customers)[['bread', 'butter']].agg({'bread':'sum', 'butter':data\_range}) |
| Groupby and transformation | | |
| The z-score |  | def zscore(series):  return (series - series.mean()) / series.std() |
| MPG z-score |  | zscore(auto['mpg']).head() |
| MPG z-score by year | .transform() | auto.groupby('yr')['mpg'].transform(zscore).head() |
| Apply transformation and aggregation |  | def zscore\_with\_year\_and\_name(group):  ...: df = pd.DataFrame(  ...: {'mpg': zscore(group['mpg']),  ...: 'year': group['yr'],  ...: 'name': group['name']})  ...: return df  auto.groupby('yr').apply(zscore\_with\_year\_and\_name).head() |
| Groupby and filtering | | |
| Mean MPG by year |  | auto.groupby('yr')['mpg'].mean() |
| groupby object |  | splitting = auto.groupby('yr')  type(splitting)  type(splitting.groups)  print(splitting.groups.keys()) |
| groupby object: iteration |  | for group\_name, group in splitting:  avg = group['mpg'].mean()  print(group\_name, avg) |
| groupby object: iteration and filtering |  | for group\_name, group in splitting:  avg = group.loc[group['name'].str.contains('chevrolet'), 'mpg'].mean()  print(group\_name, avg) |
| groupby object: comprehension |  | chevy\_means = {year:group.loc[group['name'].str.contains('chevrolet'),'mpg'].mean()  for year,group in splitting}  pd.Series(chevy\_means) |
| Boolean groupby |  | chevy = auto['name'].str.contains('chevrolet')  auto.groupby(['yr', chevy])['mpg'].mean() |

Chapter-5: Bringing it all together

|  |  |  |
| --- | --- | --- |
| Case Study: Olympic Medals | | |
| Using idxmax() | .idxmax() | weather.idxmax() # Returns month of highest temperature |
| Using idxmin() | .idxmin() | weather.T.idxmin(axis='columns') |
|  |  | by\_com\_filt = by\_company.filter(lambda g:g['Units'].sum() > 35) |
|  | .nunique() |  |
|  | .drop\_duplicates() |  |
|  | .isin() | is\_usa\_urs = medals.NOC.isin(\_\_\_\_) |
|  | pd.Categorical() | medals.Medal = pd.Categorical(values = medals.Medal, categories=['Bronze', 'Silver', 'Gold'], ordered=True) |