5.1 - DATA WRANGLING with PANDAS

## **5.1.1 - Pandas Foundation**

**Chapter-1: Data Ingestion and Inspection**

|  |  |  |
| --- | --- | --- |
| Review of Pandas Data Frames | | |
| Indexes and columns | **.dtypes** | df.dtypes |
| **type()** | type(df)  type(df.columns)  type(df.index) |
| **.shape** | df.shape |
| **.columns** | df.columns |
| **.index** | df.index |
| Slicing | **.iloc[row, column]** | df.iloc[5:, :] |
|  | **.head()** | df.head() |
| **.tail()** | df.tail() |
| **.info()** | df.info() |
| Broadcasting | **np.nan** | df.iloc[::3, -1] = np.nan  # Assigning scalar value to column slice broadcasts value to each row. |
| Building  DataFrames  from scratch | **pd.read\_csv(‘filename/path’)** | df = pd.read\_csv('datasets/users.csv', index\_col=0) |
| from dictionary | 1# df = pd.DataFrame(dictionay name)  2# zipped = list(zip(list\_labels, list\_cols))  data = dict(zipped)  df = pd.DataFrame(data) |
| Broadcasting |  | users['fees'] = 0 # Broadcasts to entire column |
|  | with a dict | data = {'height': heights, 'sex': 'M'}  results = pd.DataFrame(data) |
| Index and columns |  | results.columns = ['height (in)', 'sex']  results.index = ['A', 'B', 'C', 'D', 'E', 'F', 'G'] |
| Importing & exporting data | | |
| from CSV files | pd.read\_csv(filepath) | sunspots = pd.read\_csv(filepath) |
|  | pd.read\_json(“\*.json”) |  |
| Prob: no header | “**header**” keyword | sunspots = pd.read\_csv(filepath, header=None)  # integer atar |
| “**names**” keyword | col\_names = ['year', 'month', 'day', 'dec\_date', 'sunspots']  sunspots = pd.read\_csv(filepath, header=None, names=col\_names) |
| Prob: missing val. | “**na\_values**” keyword | sunspots = pd.read\_csv(filepath, header=None, names=col\_names, na\_values={'sunspots':[' -1']})  # “-1” gordugu yerde “NaN” atayacak |
| Prob: dates | “**parse\_dates**” keyword | sunspots = pd.read\_csv(filepath, …., parse\_dates=[[0, 1, 2]]) |
| Using dates as index |  | sunspots.index = sunspots['year\_month\_day']  sunspots.index.name = 'date' |
| Trimming redundant columns |  | cols = ['sunspots', 'definite']  sunspots = sunspots[cols] |
| Writing files | **.to\_csv()** | out\_csv = 'sunspots.csv'  sunspots.to\_csv(out\_csv) |
|  | **.to\_excel(** | out\_xlsx = 'sunspots.xlsx'  sunspots.to\_excel(out\_xlsx) |
| Ploting with pandas | | |
| Plotting arrays | **plt.plot()** | close\_arr = aapl['close'].values # array olusturma  plt.plot(close\_arr) |
| Ploting Series |  | close\_series = aapl['close']  plt.plot(close\_series) |
|  |  | close\_series.plot() # plots Series directly |
| Plo!ing DataFrames |  | aapl.plot() # plots all Series at once |
|  |  | plt.plot(aapl) # plots all columns at once |
| Fixing scales |  | plt.yscale('log') # logarithmic scale on vertical axis |
| Customizing plots | **plt.axis( )** | aapl['open'].plot(color='b', style='.-', legend=True)  plt.axis(('2001', '2002', 0, 100)) |
| Saving plots | **plt.savefig( )** | aapl.loc['2001':'2004',['open', 'close', 'high', 'low']].plot()  In [27]: plt.savefig('aapl.png')  In [28]: plt.savefig('aapl.jpg')  In [29]: plt.savefig('aapl.pdf') |

**Chapter-2: Exploratory Data Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Visual exploratory data analysis | | | |
| Line plot | **df.plot(x, y)** | iris.plot(x='sepal\_length', y='sepal\_width') |
| Scatter plot | **“kind=” keyw** | iris.plot(x, y, kind='scatter') |
| Box plot |  | iris.plot(y='sepal\_length’, kind='box') |
| Histogram |  | iris.plot(y='sepal\_length', kind='hist') |
| Customizing histogram | bins=  range=( )  normed=True/False  cumulative=T/F | iris.plot(y='sepal\_length', kind='hist', bins=30, range=(4,8), cumulative=True, normed=True) |
| DataFrame plot idioms | ● iris.plot(kind=‘hist’)  ● iris.plt.hist( )  ● iris.hist( ) |  |
| Statistical exploratory data analysis | | | |
| Summarizing | **.describe( )** | iris.describe() # summary statistics |
| counts | **.count( )** | iris['sepal\_length'].count() # Applied to Series  iris['sepal\_width'].count() # Applied to Series  iris[['petal\_length', 'petal\_width']].count() # Applied to DataFrame |
| Averages | **.mean( )** | iris['sepal\_length'].mean() # Applied to Series  iris.mean() # Applied to entire DataFrame |
| Std | **.std( )** | iris.std() |
| variance | **.var( )** | Iris.var() # herbir feature’in varyansini verir |
| Median | **.median( )** | iris.median() |
|  | Medians & 0.5 quantiles | q = 0.5  iris.quantile(q) |
|  | Inter-quartile range (IQR) | q = [0.25, 0.75]  iris.quantile(q) |
| Ranges | **.min( )**  **.max( )** | iris.min()  iris.max() |
| Separating populations | | | |
| Describe column |  | iris['species'].describe()  count 150 count: # non-null entries  unique 3 unique: # distinct values  top setosa top: most frequent category  freq 50 freq: # occurrences of top |
| Unique & factors | **.unique( )** | iris['species'].unique()  array(['setosa', 'versicolor', 'virginica'], dtype=object) |
| Filtering by column | **.loc[ ]** | indices = iris['species'] == 'setosa'  setosa = iris.loc[indices,:] # extract new DataFrame |
| Checking Column | **.unique( )** | setosa['species'].unique()  array(['setosa'], dtype=object) |
| Checking indexes | **.head( )** |  |
| Visual EDA: all data | **.plot( )** | iris.plot(kind= 'hist', bins=50, range=(0,8), alpha=0.3)  plt.title('Entire iris data set')  plt.xlabel('[cm]')  plt.show() |
| Visual EDA: individual factors |  | setosa.plot(kind='hist', bins=50, range=(0,8), alpha=0.3) |
| Statistical EDA: describe() | **.describe( )** | describe\_all = iris.describe()  describe\_setosa = setosa.describe() |
| Computing errors |  | error\_setosa = 100 \* np.abs(describe\_setosa - describe\_all)  error\_setosa = error\_setosa/describe\_setosa |

**Chapter-3: Time Series in Pandas**

|  |  |  |
| --- | --- | --- |
| Indexing Time Series | | |
| ISO 8601 format |  | yyyy-mm-dd hh:mm:ss |
| Parse dates | “**parse\_dates=True**”  “**index\_col=**” | sales = pd.read\_csv('sales-feb-2015.csv', parse\_dates=True, index\_col= 'Date') |
| Selecting datetime  .loc[ ‘row’, ‘column’] | Single datetime | sales.loc['2015-02-19 11:00:00', 'Company'] |
| Whole day | sales.loc['2015-2-5'] |
| Whole month | sales.loc[‘2015-2’] |
| Whole year | sales.loc[‘2015’] |
| Alternative formats | sales.loc[‘February 5, 2015’]  sales.loc[‘2015-Feb-5’] |
| Slicing |  | sales.loc['2015-2-16':'2015-2-20'] |
| Convert strings to datetime | **pd.to\_datetime( )** | evening\_2\_11 = pd.to\_datetime(['2015-2-11 20:00', '2015-2-11 21:00', '2015-2-11 22:00', '2015-2-11 23:00']) |
| Reindexing DataFrame | **.reindex( )** | sales.reindex(evening\_2\_11) |
| Filling missing values | **method=’ffill’** | sales.reindex(evening\_2\_11, method='ffill') |
| **method='bfill'** | sales.reindex(evening\_2\_11, method='bfill') |
| Resampling time series data | | |
| Aggregating means | **.resample( ).mean( )** | daily\_mean = sales.resample('D').mean() |
| Verifying | **.loc[ ]** | sales.loc['2015-2-2', 'Units'].mean() |
| Method chaining | .resample( ).sum( ) | sales.resample('D').sum() |
|  | .resample().sum().max() | sales.resample('D').sum().max() |
| Resampling frequencies |  | Input, Description  ‘min’, ‘ T’ minute  ‘H’ hour  ‘D’ day  ‘B’ business day  ‘W’ week  ‘M’ month  ‘Q’ quarter  ‘A’ year |
| Multiplying frequencies | .resample('2W') | sales.loc[:,'Units'].resample('2W').sum() |
| Upsampling and filling |  | two\_days = sales.loc['2015-2-4': '2015-2-5', 'Units']  two\_days.resample('4H').ffill() |
| Manipulating time series data | | |
| String methods | **.str.upper( )** | sales['Company'].str.upper() |
| Substring matching | **.str.contains(‘….’)** | sales['Product'].str.contains('ware') |
| Boolean reduction |  | sales['Product'].str.contains('ware').sum() |
| Datetime methods | **.dt.year**  **.dt.month**  **.dt.day**  **.dt.hour** | def get\_date\_int(df, column):  year = df[column]**.dt.year**  month = df[column]**.dt.month**  day = df[column]**.dt.day**  return year, month, day  # bu fonksiyon sayesinde, bir “datetime object”ten istedigimiz bilgileri “integer” olarak cekebiliyoruz. |
|  | **dt.datetime** | def **get\_month(x)**: return **dt.datetime**(x.year, x.month, 1)  # bu fonksiyon sayesinde, “YYYY-MM-DD HH:MM:SS” formatinda olan bir “datetime object”in sadece “date” bilgisini cekmis oluyorum. “time” bilgisine ihtiyacim olmadigi durumlarda kullanmak icin. Asagidakine benzer bir assignment ile de yeni bir “column” olusturarak, bu sekilde kullanmaya devam edebilirim.  online['InvoiceMonth'] = online['InvoiceDate']**.apply(get\_month)**  bunlara ilave guzel bir gruplandirma ve aggregation ornegi:  grouping = online**.groupby**('CustomerID')['InvoiceMonth']  online['CohortMonth'] = grouping**.transform('min')** |
|  |  |  |
|  |  |  |
|  |  |  |
| Set timezone | **.dt.tz\_localize(‘…..')** | central = sales['Date'].dt.tz\_localize('US/Central') |
| Convert timezone | **.dt.tz\_convert('…..')** | central.dt.tz\_convert('US/Eastern') |
| Method chaining |  | sales['Date'].dt.tz\_localize('US/Central').dt.tz\_convert('US/Eastern') |
| Upsample population | **.resample().first()** | population.resample('A').first() |
| Interpolate missing data | **.interpolate( )** | population.resample('A').first().interpolate('linear') |
| Time series visualization | | |
| One week | **.loc[ ]** | sp500.loc['2012-4-1':'2012-4-7', 'Close'].plot(title='S&P 500') |
| Plot styles | **style=** | sp500.loc['2012-4', 'Close'].plot(style='k.-', title='S&P500') |
| More plot styles |  | Color // Marker // Line  b: blue // o: circle // : do"ed  g: green //\*: star // –: dashed  r: red // s: square  c: cyan // +: plus |
| Area plot | **kind=area** | sp500['Close'].plot(kind='area', title='S&P 500') |
| Multiple columns | **.loc[ ]** | sp500.loc['2012', ['Close','Volume']].plot(title='S&P 500') |
| Subplots | **subplots=True** | In [21]: sp500.loc['2012', ['Close','Volume']].plot(subplots=True) |

Codes from exercises

|  |  |  |
| --- | --- | --- |
| Set the index | **.set\_index( )** | df.set\_index('Date', inplace=True) |
|  | **.split(',')** |  |
| Creating a datetime series | **pd.date\_range(start, periods, freq))** | i = pd.date\_range('2018-04-09', periods=4, freq='2D') |
|  | **pd.to\_numeric()** | converts a Series of values to floating-point values |
|  | **errors='coerce'** | you can force strings like 'M' to be interpreted as NaN. |
|  | **.corr()** | The Pearson correlation coefficient |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

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

Chapter-1: Extracting and transforming data

|  |  |  |
| --- | --- | --- |
| Indexing Data Frames | | |
|  |  |  |
|  |  |  |
|  |  |  |

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] ### (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()] |
|  |  | df["B"].isnull().sum()  # “B” sutunundaki toplam “NaN” sayisini verir |
| without NaNs | .notnull().all() | df.loc[:, df.notnull().all()]  df[df["B"].notnull()]  # “B” sutunu degeri “NaN” olmayan observationlari gosterir |
| Drop **rows** with any NaNs | .dropna( ) | df.dropna(how='any')  # .dropna(), herhangi bir feature value “NaN” ise tum satiri siler |
|  | .drop() | # .drop([1,2,3]), # ilgili index’teki satirlari siler  # .drop(“A”, axis=1) # “A” sutununu tablodan cikarir |
|  | “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

|  |  |
| --- | --- |
|  |  |
|  |  |
|  | # .pivot() komutu, duplikasyon gordugunde hata verir, iste bu noktada pivot\_table kullanilir, bunu da basa cikmasi icin bir fonksiyon vererek saglariz (duplikasyonlari topla, ortalamasini al gibi)  # .stack() ve .unstack() komutlari, multiindex’li durumlarda en icteki katmani alir ve digger tarafta en icteki katmana tasir, tabiki ayrica bir parameter eklenmez ise  # .pivot()/.pivot\_table()’da “values=” degeri girilmezse, kalan tum sutunlarin degerlerini kullanir |

|  |  |  |
| --- | --- | --- |
| 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( ) | pd.melt(frame, id\_vars=None, value\_vars=None, var\_name=None, value\_name='value', col\_level=None)[source]¶  #id\_vars: eritmek istemedigimiz, solda kalacak olan sutunlar, anahtar sutunlar da diyebiliriz.  #value\_var |
| 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) |

5.1 - DATA WRANGLING with PANDAS

## **5.1.3 - Merging DataFrames with pandas**

Chapter-1: Preparing data

|  |  |  |
| --- | --- | --- |
| Reading multiple data files | | |
| Import tools | * pd.read\_csv() * pd.read\_excel() * pd.read\_html() * pd.read\_json() |  |
| Separate file | pd.read\_csv() | dataframe0 = pd.read\_csv('sales-jan-2015.csv') |
| Multiple files | for loop | filenames = ['sales-jan-2015.csv', 'sales-feb-2015.csv']  dataframes = []  for f in filenames:  dataframes.append(pd.read\_csv(f)) |
| comprehension | dataframes = [pd.read\_csv(f) for f in filenames] |
| glob | from glob import glob  filenames = glob('sales\*.csv')  dataframes = [pd.read\_csv(f) for f in filenames] |
| Reindexing DataFrames | | |
| importing | index\_col= | w\_mean = pd.read\_csv('quarterly\_mean\_temp.csv', index\_col='Month') |
| Type of index | .index | print(w\_mean.index)  Index(['Apr', 'Jan', 'Jul', 'Oct'], dtype='object', name='Month') |
| Reindexing | .reindex() | ordered = ['Jan', 'Apr', 'Jul', 'Oct']  w\_mean2 = w\_mean.reindex(ordered) |
| from a DataFrame Index | w\_mean.reindex(w\_max.index) |
| Sorting index | .sort\_index() | w\_mean2.sort\_index() |
|  | ascending=False | reverse alphabetical order by specifying the additional keyword argument |
| Arithmetic with Series & DataFrames | | |
| Scalar multiplication | \* | weather.loc['2013-07-01':'2013-07-07', 'PrecipitationIn'] \* 2.54 |
| Dividing | .divide() | week1\_range.divide(week1\_mean, axis='rows') |
| Percentage changes | .pct\_change() | week1\_mean.pct\_change() \* 100 |
| Adding | + | bronze + silver |
| .add() | bronze.add(silver) |
| Using a fill\_value | fill\_value=0 | bronze.add(silver, fill\_value=0) |
| Chaining .add() |  | bronze.add(silver, fill\_value=0).add(gold, fill\_value=0) |

Chapter-2: Concatenating data

|  |  |  |
| --- | --- | --- |
| Appending & concatenating Series | | |
|  | .append() | ● Invocation: s1.append(s2)  ● Stacks rows of s2 below s1  ● Method for Series & DataFrames |
|  | .concat() | ● Invocation: pd.concat([s1, s2, s3])  ● Can stack row-wise or column-wise  ● pandas module function  pd.concat(objs, axis=0, join='outer', join\_axes=None, ignore\_index=False, keys=None, levels=None, names=None, verify\_integrity=False, copy=True) |
| Using .append() |  | east = northeast.append(south). ## satirlari alta yapistirir, iki index’lidir |
| Using .reset\_index() | .reset\_index() | new\_east = northeast.append(south).reset\_index(drop=True) |
| Using concat() |  | east = pd.concat([northeast, south]) ## iki index’lidir |
| Using ignore\_index | ignore\_index= | new\_east = pd.concat([northeast, south], ignore\_index=True) |
| Appending & concatenating DataFrames | | |
| Appending Dataframes | .append() | population.append(unemployment)  ### satirlari oldugu gibi altina yapistirir, tekrar eden indexler olabilir. |
| Concenating rows | .concat (axis=0) | pd.concat([population, unemployment], axis=0)  ## satir birlestirmesi yapar, tekrar eden indexler olabilir |
| Concatenating columns | .concat (axis=1) | pd.concat([population, unemployment], axis=1)  ## sutun birlestirmesi yapar, ayni indexleri tek satirda gosterir |
| Concatenation, keys, & MultiIndexes | | |
| Using multi-index | keys=[] | rain1314 = pd.concat([rain2013, rain2014], keys=[2013, 2014], axis=0) |
| rain1314 = pd.concat([rain2013, rain2014], keys=[2013, 2014], axis='columns') |
| Accessing a multi-index | .loc[] | print(rain1314.loc[2014]) |
| pd.concat() with dict |  | rain\_dict = {2013: rain2013, 2014: rain2014}  rain1314 = pd.concat(rain\_dict, axis='columns') |
| Outer & inner joins | | |
| Using with arrays | np.arange().reshape() | A = np.arange(8).reshape(2,4) + 0.1  print(A)  [[ 0.1 1.1 2.1 3.1]  [ 4.1 5.1 6.1 7.1]] |
| Stacking arrays horizontally | np.hstack()  np.concatenate(axis=1) | np.hstack([B, A])  np.concatenate([B, A], axis=1) |
| Stacking arrays horizontally | np.vstack()  np.concatenate(axis=0) | np.vstack([A, C])  np.concatenate([A, C], axis=0) |
| Manipulating data as arrays | np.concatenate() | population\_array = np.array(population). ## index bilgisi gider  unemployment\_array = np.array(unemployment)  print(np.concatenate([population\_array, unemployment\_array], axis=1)) |
| Joins |  | ● Joining tables: Combining rows of multiple tables  ● Outer join: Union of index sets (all labels, no repetition)  Missing fields filled with NaN  ● Inner join: Intersection of index sets (only common labels) |
| Concatenation & inner | join='inner' | pd.concat([population, unemployment], axis=1, join='inner')  ## sadece iki set arasinda kesisen satirlari alir |
| Concatenation & outer join | join='outer' | pd.concat([population, unemployment], axis=1, join='outer')  ## birlestirilen datasetlerin tum satirlarini alir |
| Inner join on other axis | join='inner', axis=0 | pd.concat([population, unemployment], join='inner', axis=0) |

Chapter-3: Merging data

|  |  |  |
| --- | --- | --- |
| Merging dataframes | | |
| Merging | .merge() | pd.merge(population, cities). ## ornekte ayni indexe ait bilgiler kullanilmis  ## pd.merge(left, right, how='inner', on=None, left\_on=None, right\_on=None, left\_index=False, right\_index=False, sort=True, suffixes=('\_x', '\_y'), copy=True, indicator=False, validate=None) |
| Merging all columns |  | pd.merge(bronze, gold) |
| Merging on | on= | pd.merge(bronze, gold, on='NOC'). # index mi acaba? |
| Merging on multiple columns |  | pd.merge(bronze, gold, on=['NOC', 'Country']) |
| Using suffixes | suffixes= | pd.merge(bronze, gold, on=['NOC', 'Country'], suffixes=['\_bronze', '\_gold'])  ## sutunlarin varolan basliklarina eklenti yapar |
| Specifying columns to merge | left\_on= , right\_on= | pd.merge(counties, cities, left\_on='CITY NAME', right\_on='City') |
| Joining DataFrames | | |
| Merge with “how=” | how= | pd.merge(bronze, gold, on=['NOC', 'Country'], how='inner')  how=’inner’ -🡪 kesisenleri alir  how=’left’ -🡪 soldaki DF’in tum satirlari ve sagdakinin kesisenleri  how=’right’ -🡪 sagdaki DF’in tum satirlari ve soldakinin kesisenleri  how=’outer’ -🡪 her iki DF’in tum satirlari |
| Using .join | .join() | population.join(unemployment, how= 'right') # “how” hususu ayni |
| Ordered merges | | |
| Using merge\_ordered() | merge\_ordered() | pd.merge\_ordered(hardware, software)  # sorted\_values’un tarihleri arka arkaya dizer, “ordered” tarih sirasi yapar |

Chapter-4: Case Study - Summer Olympics

|  |  |  |
| --- | --- | --- |
|  | .copy() | pd.concat takes an Iterable as its argument. Hence, it cannot take DataFrames directly as its argument. Also Dimensions of the DataFrame should match along axis while concatenating.  pd.merge can take DataFrames as its argument, and is used to combine two DataFrames with same columns or index, which can't be done with pd.concat since it will show the repeated column in the DataFrame. |
|  | .ffill() | weather3 = weather1.reindex(year).ffill() |
|  | .last() | aggregation method to select the last element when resampling. |
|  | "%s\_top5.csv" % | file\_name = "%s\_top5.csv" % medal  ### The expression "%s\_top5.csv" % medal evaluates as a string with the value of medal replacing %s in the format string. |
|  | pd.IndexSlicep | A slicer pd.IndexSlice is required when slicing on the inner level of a MultiIndex. |
|  |  | idx = pd.IndexSlice  # Print all the data on medals won by the United Kingdom  print(medals\_sorted.loc[idx[:,'United Kingdom'],:]) |
|  | pd.merge\_asof() |  |
|  | .expanding() |  |
|  | pd.melt() |  |

5.1 - DATA WRANGLING with PANDAS

## **5.1.4 – Cleaning Data in Python**

Chapter-1: Exploring your data

|  |  |  |
| --- | --- | --- |
| Diagnose data for cleaning | | |
|  |  | Common data problems  ● Inconsistent column names  ● Missing data  ● Outliers  ● Duplicate rows  ● Untidy  ● Need to process columns  ● Column types can signal unexpected data values |
| Visually inspect | .head()  .tail()  .columns  .shape  .info() | Missing values  Capitalization  Spaces  Data type inconvenience |
| EDA | | |
| Data type inspection | .info() |  |
| Frequency counts | .column.value\_counts() | df.country.value\_counts(dropna=False).head()  ## “dropna=False” eksik verileri de saydigindan eklemekte fayda var  ## “NaN” for numeric // “missing” for strings |
| Summary statistics | .describe() | ## Outliers: Considerably higher or lower |
| Visual exploratory data analysis | | |
|  |  | ● Great way to spot outliers and obvious errors  ● More than just looking for pa!erns  ● Plan data cleaning steps |
| Look at frequencies | Histogram | Outliers |
|  | Box plot | df.boxplot(column='population', by='continent')  ### Outliers /// Min/max /// 25th, 50th, 75th percentiles |
|  | Scatter plot | ● Relationship between 2 numeric variables  ● Flag potentially bad data |

Chapter-2: Tidying data for analysis

|  |  |  |
| --- | --- | --- |
| Tidy data |  |  |
| Principles of tidy data |  | ● Columns represent separate variables  ● Rows represent individual observations  ● Observational units form tables |
| data problem: Columns containing values, instead of variables | pd.melt() | “id\_vars=” daki veriyi tutar, diger verileri bit sutunun icine eritir |
| Pivoting Data: un-melting data | | |
| Pivot |  | ● In melting, we turned columns into rows  ● Pivoting: turn unique values into separate columns |
| Pivot table |  | Has a parameter that specifies how to deal with duplicate  values |
| Beyond melt and pivot | | |
|  |  | Another common problem: Columns contain multiple bits of information ## bir sutunun birden fazla bilgi icermesi.  ## burada bitisik bilgilerin parse edilerek ayristirilmasi gerekiyor. |
| Melting and parsing | pd.melt() | pd.melt(frame=tb, id\_vars=['country', 'year']) |
|  | .str[] | tb\_melt['sex'] = tb\_melt.variable.str[0] |
|  | .str.split() | ebola\_melt['str\_split'] = ebola\_melt['type\_country'].str.split('\_') |
|  | .str.get() | ebola\_melt['type'] = ebola\_melt['str\_split'].str.get(0) |

Chapter-3: Combining data for analysis

|  |  |  |
| --- | --- | --- |
| Globbing |  |  |
|  | glob.glob() | import glob  csv\_files = glob.glob('\*.csv')  pd.concat(csv\_files) |
| Concatinating & Merging | | |

Chapter-4: Cleaning data for analysis

|  |  |  |
| --- | --- | --- |
| Converting data types | .astype() | df['treatment b'] = df['treatment b'].astype(str)  df['sex'] = df['sex'].astype('category') |
| Cleaning bad data | pd.to\_numeric() | df['treatment a'] = pd.to\_numeric(df['treatment a'], errors='coerce') |
| String manipulation | “re” | ● ‘re’ library for regular expressions  ● A formal way of specifying a pattern, Sequence of characters  ● Pattern matching, Similar to globbing |
| Using regular expressions | re.compile() & .match | In [1]: import re  In [2]: pattern = re.compile('\$\d\*\.\d{2}')  In [3]: result = pattern.match('$17.89')  In [4]: bool(result)  True |
| Find the numeric values | re.findall() | matches = re.findall('\d+', 'the recipe calls for 10 strawberries and 1 banana') |
| Using functions to clean data | | |
| Apply | .apply() | df.apply(np.mean, axis=0) |
|  | .replace() | import re  from numpy import NaN  pattern = re.compile('^\$\d\*\.\d{2}$')  def diff\_money(row, pattern):  icost = row['Initial Cost']  tef = row['Total Est. Fee']  if bool(pattern.match(icost)) and bool(pattern.match(tef)):  icost = icost.replace("$", "")  tef = tef.replace("$", "")  icost = float(icost)  tef = float(tef)  return icost - tef  else:  return(NaN)  df\_subset['diff'] = df\_subset.apply(diff\_money, axis=1, pattern=pattern) |
| Drop duplicates | .drop\_duplicates() | df = df.drop\_duplicates() ## tamami duplication ise siler |
| Drop missing values | .dropna() | tips\_dropped = tips\_nan.dropna() |
| Fill missing values | .fillna() | tips\_nan['sex'] = tips\_nan['sex'].fillna('missing')  tips\_nan[['total\_bill', 'size']] = tips\_nan[['total\_bill', 'size']].fillna(0) |
| Testing with asserts | assert | assert google.Close.notnull().all()  google\_0 = google.fillna(value=0)  assert google\_0.Close.notnull().all()  assert pd.notnull(ebola).all().all()  assert (ebola >= 0).all().all() |
|  | .apply(lambda) | ## tips['total\_dollar\_replace'] = tips.total\_dollar.apply(lambda x: x.replace('$', ''))  ## tips['total\_dollar\_re'] = tips.total\_dollar.apply(lambda x: re.findall('\d+\.\d+', x)[0]) |

Additional Notes

**Feature Scaling**

|  |  |  |
| --- | --- | --- |
| Logarithmic Normalization | | |
|  | np.log() | df[“col2\_log”] = np.log(df[“col\_2”])  np.var(df[[“col1”, “col2\_log”]] # control etmek icin |
| Feature caling |  |  |
|  | StandardScaler | from sklearn.preprocessing import StandardScaler  scaler = StandardScaler()  df\_scaled = pd.DataFrame(scaler.fit\_transform(df),  columns=df.columns) |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |