**Pandas Cheat Sheet**

Note: df - A pandas DataFrame object, s - A pandas Series object

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

%matplotlib inline #enables the rendering of Matplotlib plots directly below code cells

|  |
| --- |
| **Importing data** |
| **pd.read\_csv(filename)** # From a CSV file  **pd.DataFrame(dict)** # From a dict, keys for column names, and values for data as lists or seriess  **pd.options.display.max\_columns =150** # to avoid truncated output |

**Viewing/inspecting data**

**df.head(n)**

**df.tail(n)**

**df.shape**

**df.info()**

**df.describe()** # Summary statistics for numerical columns

**df.isna().sum()** # Sum of null values per col

**df.notna()**

**df.col.unique()** # Check unique values in 'col'

**df['col'].unique()**

**df['col'].nunique()** # Return number of unique elements in the object

**s.value\_counts(dropna=False)** # View unique values and counts

**df.value\_counts(dropna=False, normalize=True,ascending=True) df.columns** # check column names

**df.index** # return the index

**df.duplicated()** #Return duplicated rows **df['col'].quantile(0.25)** # Return the value at the 0.25 quantile for a column

**s.str.contains(‘sub\_str’**)# Return boolean Series or Index based on whether a given pattern or regex is contained within a string of a Series or Index.

|  |
| --- |
| **Selection and reference** |
| *# Slicing with [], work on rows* |
| **s[:5]** |
| **df[:5]** # Return first 5 rows |
| **df[::2]**# Return dataframe with a step of 2 on row selection |
| **df[::-1]** # Return a reversed dataframe |
| *# Select columns and rows* |
| **df["col"]** # Return the column as `Series` |
| **df[["col"]]** # Return the col as `DataFrame` |
| **df[["col1","col2"]]**  **df[df['col1']>0]** # Return the rows where the values of 'col1'>0  **df[df['col1']>0]['col2']** # Return 'col2' on selected rows based on values of 'col1' as a Series  **df[df['col1']>0][['col2']]** # Return 'col2' on selected rows based on values of 'col1' as a DataFrame |
| *# Select by label via .loc[]* |
| **df.loc[0]** #Return a Series for row 0 |
| **df.loc[[0]]** #Return a DataFrame for row 0 |
| **df.loc[[0,3]]** #Return a DataFrame -- the rows with integer index 0 and 3 |
| **df.loc[:5]** # Return a DataFrame with rows index from 0 to 5 (note include index 5) |
| **df.loc["row1":"row2"]** # Return a DataFrame with rows index from "row1" to "row2" |
| **df.loc[:5,"col1":"col2"]** # Return a DataFrame with rows index from 0 to 5, columns index between "col1" and "col2" |
| **df.loc[["a", "c"],"col2":]** # Return a DataFrame with rows index "a" and "c", columns index from "col2" to the end |
| **df.loc['a', 'A']** # Return a cell value at row "a", col "A" |
| **df.loc[:,"col1"]>0** # Return a boolean type Series along rows |
| **df.loc[df.loc[:,"col1"]>0,:]** # Return rows selected by a boolen array |
| **df.loc[df.col1>0,:]** # Return rows selected by a boolen array, a simplified way |
| **df.loc[lambda df: df.col1>0,:]** # Selection by Callable lambda function |
| #lambda with .loc be more useful in complex operations or when chaining multiple methods together, as it allows you to pass functions dynamically.  **s.idxmax()** # Return the (first) index of the maximum value  **df['col'].idxmax()** # Return the (first) row index of the maximum value for 'col'  **df.idxmax()** # Returns a Series with the index of the maximum value for each column  **df.loc[df['col'].idxmax(),:]** # Isolate the row of df with the max in 'col' |
| **df.loc[lambda df: (df.col1>0) & (df.col2=="2009-01-02"),:].assign(col3=lambda df: df.col1 - 2, col4=lambda df: df.col3\*2)** |
| #Select by position via .iloc[], integer based indexing |
| **df.iloc[0]** #Return a Series -- the first row |
| **df.iloc[[0]]** #Return a DataFrame |
| **df.iloc[[0,3]]** # Return 1st and 4th rows |
| **df.iloc[:4, :4]** # Return the top\_left\_corner (first four rows, and first four columns) |
| **df.iloc[-4:, -4:]** #Return the bottom right corner |
| **df.iloc[[1, 3, 5], [1, 3]]** |
| **df.iloc[0,0]** # Retrun the cell value at first row, first column |

**Data cleaning**

*# Change column or index names*

**df.columns= ['new1', 'new2', 'new3']**

# Rename columns

**df.rename(columns={"old1":"new1","old2":"new2"})** # Selective renaming using a mapping

**df.rename(index={0: "x", 1: "y", 2: "z"}) #Rename index using a mapping**

**df.rename(index=lambda x: x+1)**

# Mass renaming index

**df.set\_index("col\_A")** # Set the DataFrame index using existing column "col\_A"

*# Dealing with null values*

**df.isna()** #Return a boolean same-sized object

**df["col1"].isna()** # Return a boolean Series

**df.notna()** # opposite of .isna()

**df.dropna()** # Drop all rows that contain null values, default inplace=False

**df.dropna(inplace=True)** # Change inplace

**df.dropna().reset\_index()** # Drops rows with null values and reset index

**df.dropna().reset\_index(drop=True)** #  Avoid the old index being added as a column

**df.dropna(axis=1)** # Drop columns that contain null values

**df.dropna(how='all')** # Drop rows when all elements are missing ; Default= "any" (at least one element is missing)

**df.dropna(thresh=n)** # Drop rows that have less than thresh n non-null values

**df.fillna(x)** # Replace all null values with x

**df.fillna(value={"A": 0, "B": 1})** # Replace NaN in column ‘A’, ‘B’, with 0, 1 respectively.

**s.fillna(s.mean())** # Replace NaN with mean

**df.fillna(df.mean(axis=0))** # Fill NaN values with the mean of each column

**df["col1"].fillna(df["col1"].mode()[0])**

# Fill NaN values with the mode of the column**;** Note that .mode() here returns a Series object so use [0] to select the (first) element/value

**df.isin(values) #** DataFrame of booleans showing whether each element in the DataFrame is contained in values.

**s.replace(0,1, inplace=True)** # Replace 0s with 1s

*# Dealing with dulplicates*

**df.duplicated(subset=['col1', 'col2'])**

**df.drop\_duplicates(subset=['col1',** **'col2'], keep='last')**

*# Drop rows or columns*

**df.drop([0,1])** # Drop rows by index

**df.drop(['A','B'], axis=1)** #Drop columns

**df[df['col1'] <= 2]** # Drop rows where the values of “col1" >2

*# Change data type*

**s.astype(float)** # Convert to float

*# Replace values*

**s.replace(1,"one")** # Replace 1 with "one"

**df.replace([1,3],["one","three"])**

**df.replace({1:"one", 3: "three"})**

|  |
| --- |
| **Filter, sort, groupby and transform** |
| *# Filtering*  **df[df['col']>0.5] # Select rows that meet the condition**  **df[(df['col']>0.5) & (df['col']<1)]**  **df.loc[df['col']>0.5,:]**  **df.where(df>0.5)** # Keeps the original values where the condition is True and replaces with np.nan where the condition is false  **df.where(df>0.5, other=0)** # replace with 0 where the condition is false  *# Sorting*  **df.sort\_index()** # Sort by index along rows in ascending order  **df.sort\_index(ascending=False, inplace=True)** # Sort by index along rows in descending order, inplace change  **df.sort\_values(by="col",ascending=False)** # Sort values by col in descending order;  **df.sort\_values(by=["col1','col2'],ascending=[False, True] )**  **df.reset\_index()** # Reset index  **df['col\_rank']=df['col'].rank()** # Creates a new column where each entry corresponds to the rank (1 through n) of that row’s value in column 'col'  **df['col\_rank']=df['col'].rank(method='min',ascending=False)** # the records that have the same values are ranked using the lowest rank; by default the average rank is used; other methods are **‘max’, ‘first’, ‘dense’**.  *# Grouping*  **df.groupby('col')** # Returns a groupby object for values from col; this did a mapping to df  **df.groupby(['col1','col2'])**  **df.groupby('col').size()** # Return the number of rows in each group, grouped by ‘col’  **df.groupby('col').count()** # Retrun the number of NON\_NULL values for each column  **df.groupby('col')['col2'].count()** # Retrun the number of NON\_NULL values for 'col2' as a Series  **df.groupby('col')[['col2']].count()** # Retrun the number of NON\_NULL values for 'col2' as a DataFrame  **df.groupby('col').mean()** # Produces a DataFrame with the group names as its new index and the mean values for each numeric column by group; other methods include **median(), mode(), sum(), size(), count(), min(), max(), std(), var(), describe() , nunique()**  **df.groupby('col1')[['col2', 'col3']].mean()** #Return the mean of the values in col2 and col3, grouped by the values in col1  **df.groupby('col1')[['col2', 'col3']].agg(['sum', 'mean', 'std'])** # Apply functions at once, using pandas' optimized groupby sum(), mean(), std()methods  **df.groupby('col1').agg({"col2": "mean", "col3": "std"})** # Apply "mean" to "col2", "std" to "col3"  **df.groupby('col1')['col2'].transform(lambda x: (x - x.mean()) / x.std())** # Rreturn a new DataFrame with the same row numbers and indexing as the original one but with transformed individual values  **df.groupby('col1')['col2'].transform(lambda x: x.fillna(x.mode()[0]))**  **df.groupby('col1').head(n)** # Returns the first n rows (5, by default) of each group correspondingly  *# Other transformation*  **df[['B', 'A']] = df[['A', 'B']]** # Swap column contents  **df.assign(name = ["Emil", "Tobias", "Linus"])** # Assign a new column "name" to a df, returning a new object with the new columns added to the original ones  **df.assign(temp\_f=lambda x: x.temp\_c \* 9 / 5 + 32)** # Assign a new column "temp\_f" from values of column "temp\_c" via lambda function  *# Convert categorical variable into dummy/indicator variables*  **df['status'].map({'active': 1, 'inactive': 0})** # Apply simple element-wise transformations or mappings to a column  **pd.get\_dummies(df)** # All the columns in df with object, string, or category dtype will be converted; Each /column( or variable) is converted in as many 0/1 variables as there are different catergories.  **pd.get\_dummies(df, drop\_first=True)**  # Whether to get k-1 dummies out of k categorical levels by removing the first level.  *# .apply(), .map(), .applymap()*  s.map() works on a series, element-wise;  .apply() works on both series and dataframe, operates on entire rows or columns at a time for Dataframe and element-wide for series; df.applymap() works on dataframe, element-wise.  **df.apply(my\_function)** # Apply my\_function to each column  **df.apply(my\_function, axis=1)** # Apply my\_function to each row  **df['FirstName'] = df['EmployeeName'].apply(lambda x : x.split()[0])** # Apply lambda function on 'EmployeeName' col to create a new column  **df['Value3']=df['Value1'].apply(lambda x: x\*\*2)**  **mask=df.apply(lambda x: True if x ['Gender'] == 'F' and x['Kids'] > 0 else False, axis=1)** # return a boolean Series  **df['BMI'] = df.apply(lambda x: calc\_bmi(x['Weight'], x['Height']), axis=1) # Apply the custom function calc\_bmi to the data frame, across each row**  *# PIVOT\_TABLE*  **table=pd.pivot\_table(df,index=['col','col2'],values=['col3','col4'],columns='col5', aggfunc='sum')** # Create a pivot table that groups by 'col1' and 'col2'; by default, aggfunc='mean' |

**Join/Combine**

**pd.concat([df1,df2])** # Concatenate along rows, retain original index

**pd.concat([df1,df2],join="inner",ignore\_index=True)** # Concatenate along rows and return only overlapping columns, reset index

**pd.concat([df1,df2],axis=1) #** Concatenate along columns (only work if the tables have the same height)

# Note df.append() had been deprecated

**pd.merge(left, right, on="key")** # Inner join on "key" in both dataframes

**pd.merge(left, right, how="left", on=["key1", "key2"])** # Left join on ["key1", "key2"] in left

**pd.merge(left, right, left\_on="key", right\_index=True, how="left", sort=False)** # Join DataFrame left’s column "key" with DataFrame right’s index

**df1.merge(df2, left\_on='lkey', right\_on='rkey')** # how="inner" by default

**df1.join(df2, how="inner")** # .join() joins on indexes by default, how="left" by default

|  |
| --- |
| **Statistics and Some Common Operations** |
| **df.sum()** # Defual axis=0, sum of columns  **df.min()**  **df.max()**  **df.median()**  **df.mode()** # Retruns the mode of each column, a Series or a DataFrame  **df.std()**  **df.count()** # Return the number of non-null values in each column  **df.nunique()** # number of unique values  **df.corr()** # Returns the correlation coefficients between all numerical columns  **s1.corr(s2)** #correlation between s1 and s2  **df['col1'].corr(df['col2'])**  **df.copy()**  **df.T** # Transpose rows and cols  **df.size** # nrows\* ncols; different to the .size() for groupby object  **df.values** # Get a numpy array for df  **df.sample(n=20, replace=True, random\_state=1)** # take a random sample from df; replace=True for bootstrap sample |

**Datetime in Pandas**

# Create Pandas Timestamp object

**date = pd.Timestamp('2013-01-01')**

**date2 = pd.Timestamp('2013-01-01 21:15:06')**

**date3 = pd.Timestamp('Sep 04, 1982 1:35.18')**

# Create a Period object

**month = pd.Period('2013-01', freq='M')**

# Create a sequence of dates

**dates=pd.date\_range('2022-2-7', periods=7)** # Return a fixed frequency DatetimeIndex. Each date in the DatetimeIndex instance is an instance of the Timestamp.

**pd.date\_range(start='1/1/2018', periods=5, freq='3ME')** # 3 month end frequency, by default, freq='D'

# Separate element of a Timestamp object from built-in attributes

**year=date.year**

**month=date.month**

**day=date.day**

**hour=date.hour**

**month\_name=date.month\_name()**

**week\_day=date.weekday()** # Return the day of the week as a number, counting from 0 (for monday)

**day\_name=date.day\_name()**

# Convert to a pandas datetime object

**df['datetime'] = pd.to\_datetime(df['datetime'],yearfirst=True)** # Convert column 'datetime' (string object) to a datetime object

**df['datetime'] = pd.to\_datetime(df['datetime'], format="%y-%m-%d")** # Convert column 'datetime' (string object) to a datetime object by providing an exact format

**df['hour']=df['datetime'].dt.hour** # get the hour from the datetime column

**mask = (df['datetime'] >= pd.Timestamp('2019-03-06')) & (df.datetime < pd.Timestamp('2019-03-07'))** # Create a Boolean mask to select the DataFrame rows between two specific dates

**df[mask]**

**df.set\_index('datetime', inplace=True)** # Set the datetime column as the index of the DataFrame for Timestamp slicing

**df.loc['03-04-2019':'04-04-2019']** # Return rows within a date range.

|  |
| --- |
| **Visualizations** |

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**## Boxplot**

# to visualize distribution of `col1` and detect any outliers

**plt.figure(figsize=(5,1))**

**plt.title('Boxplot to detect outliers', fontsize=12)**

**plt.xticks(fontsize=12)**

**plt.yticks(fontsize=12)**

**sns.boxplot(x=df['col1'])**

**plt.show()**

**## Histogram**

# Histogram for 'col1'

**plt.figure(figsize=(5,3))**

**sns.histplot(df['col1'], bins=range(start,stop,step))** # start inclusive, stop non-inclusive

**plt.title('col1 histogram');**

# Histogram for 'col1' and set the label format for x-axies

**ax = sns.histplot(df['col1'], bins=range(0,(7\*10\*\*5+1),10\*\*5))**

**labels = [0] + [str(i) + 'k' for i in range(100, 701, 100)]**

**ax.set\_xticks(range(0,7\*10\*\*5+1,10\*\*5), labels=labels)**

**plt.title('col1 histogram');**

# Histogram for 'col1' with hue mapping to 'col2'

**plt.figure(figsize=(5,3))**

**sns.histplot(data=df, x='col1', hue='col2', multiple='dodge', shrink=0.5)**

**plt.title('col1 histogram')**

**## Scatterplot**

**plt.figure(figsize=(6, 4))**

**sns.scatterplot(data=df, x='col1', y='col3', hue='col2', s=10, alpha=0.4)** # s for marker size, alpha for transparency degree

**plt.axvline(x=X, color='#ff6361', label='X', ls='--') # add a vertical dashline**

**plt.title('col1 by col3', fontsize='14');**

**## Pie graph**

**fig = plt.figure(figsize=(3,3))**

**plt.pie(df.groupby('col1')['col2'].sum(), labels=['label1', 'label2'])**

**plt.title('col2 by col1');**

**## bar chart**

**df2 = df.groupby(['col1']).median(**

**numeric\_only=True).reset\_index()**

**plt.figure(figsize=(5,3))**

**sns.barplot(data=df2,**

**x='col1',**

**y='col2',**

**order=['str1', str2', …,'strn'],**

**palette={'str1':'green', str2':'orange',…, 'strn':'red'},**

**alpha=0.5)**

**plt.title('Median view count by col1');**

**## Pairplot**

**sns.pairplot(**

**df,**

**vars=["col1", "col2", "col3", "col4", "col5"],**

**hue="col5",**

**);**

**## Heatmap**

**plt.figure(figsize=(16, 9))**

**sns.heatmap(df[['col1','col2','col3','col4']].corr(), cmap="crest")**

**plt.title('Heatmap of the dataset')**

**plt.show()**

**## subplots**

# single Axes

**fig, ax = plt.subplots()**

# multiple Axes

**fig, ax = plt.subplots(1, 2)**

**sns.histplot(df['col1'], bins=100,ax=ax[0])**

**ax[0].set\_title('title1', fontsize='14')**

**sns.histplot(df['col2'], bins=100,ax=ax[1])**

**ax[1].set\_title('title2', fontsize='14')**

# using tuple unpacking for multiple Axes

**fig, (ax1, ax2) = plt.subplots(1, 2)**

**fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)**