Pandas As described at https://pandas.pydata.org pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Resources 1. Ch 5-6 in Python for Data Analysis, 2nd Ed, Wes McKinney (UCalgary library and https://github.com/wesm/pydata-book) 2. Ch 3 in Python Data Science Handbook, Jake VanderPlas (Ucalgary library and https://github.com/jakevdp/PythonDataScienceHandbook) Let's explore some of the features. First, import Pandas, and Numpy as a good companion. In [1]: import numpy as np import pandas as pd Create pandas DataFrames There are several ways to create Pandas DataFrames, most notably from reading a csv (comma separated values file). DataFrames are 'spreadsheets' in Python. We will often use df as a variable name for a DataFrame. If data is not stored in a file, a DataFrame can be created from a dictionary of lists data = {'state': ['Ohio', 'Ohio', 'Nevada', 'Nevada'], 'year': [2000, 2001, 2002, 2001, 2002, 2003], 'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]} frame = pd.DataFrame(data) where dictionary keys become column headers. An alternative is to create from a numpy array and set column headers seperatly: In [2]: # From a numpy array df = pd.DataFrame(np.arange(20).reshape(5,4), columns=['alpha', 'beta', 'gamma', 'delta']) Out[2]: alpha beta gamma delta 0 3 0 1 2 1 5 6 9 10 11 12 13 14 15 16 17 18 19 In [3]: # checking its type type(df) pandas.core.frame.DataFrame Out[3]: Indexing Accessing data in Dataframes is done by rows and columns, either index or label based. # select a column df['alpha'] 0 Out[4]: 2 8 3 12 4 16 Name: alpha, dtype: int64 In [5]: # select two columns df[['alpha', 'gamma']] Out[5]: alpha gamma 0 2 10 14 16 18 In [6]: # select rows df.iloc[:2] Out[6]: alpha beta gamma delta 5 7 In [7]: # select rows and columns df.iloc[:2, :2] Out[7]: alpha beta In [8]: # select rows and columns, mixed df.loc[:2, ['alpha', 'beta']] Out[8]: alpha beta 0 0 1 5 DataFrame math Similar to Numpy, DataFrames support direct math In [9]: # direct math df2 = (9/5) * df + 32Out[9]: alpha beta gamma delta **0** 32.0 33.8 35.6 37.4 39.2 41.0 42.8 44.6 46.4 48.2 2 50.0 51.8 53.6 55.4 57.2 59.0 60.8 62.6 64.4 66.2 In [10]: # add two dataframes of same shape Out[10]: alpha beta gamma delta 32.0 34.8 40.4 37.6 43.2 46.0 48.8 51.6 54.4 57.2 60.0 62.8 3 65.6 68.4 71.2 74.0 76.8 79.6 82.4 85.2 In [11]: # map a function to each column f = lambda x: x.max() - x.min()df.apply(f) 16 alpha Out[11]: beta 16 16 gamma delta 16 dtype: int64 **DataFrame manipulation** Adding and deleting columns, as well as changing entries is similar to Python dictionaries. Note that most DataFrame methods do not change the DataFrame directly, but return a new DataFrame. It is always good to check how the method you are invoking behaves. In [12]: # add a column df['epsilon'] = ['low', 'medium', 'low', 'high', 'high'] df alpha beta gamma delta Out[12]: epsilon 0 2 3 0 low 4 5 6 medium 9 10 11 low 3 12 13 14 15 4 16 17 18 19 high In [13]: # What is the size? df.shape (5, 5)Out[13]: In [14]: # delete column df dropped = df.drop(columns=['gamma']) df dropped Out[14]: alpha beta delta epsilon 0 0 1 3 low medium 2 9 11 8 low 3 12 13 15 high 4 16 17 19 high In [15]: # the original dataframe is unaffected df Out[15]: alpha beta gamma delta epsilon low 6 medium 2 8 10 11 9 low 3 12 13 14 15 high 16 17 18 high Let's create a copy and assign new values to the first column: In [16]: df copy = df.copy() df copy['alpha'] = 20 print(df) print(df copy) alpha beta gamma delta epsilon 1 3 0 0 2 7 medium 1 4 5 6 2 8 9 10 11 low 3 12 13 14 15 high high 16 17 18 19 4 alpha beta gamma delta epsilon 0 20 1 2 3 5 7 medium 1 20 6 2 20 9 10 11 low 20 13 15 3 14 high 20 18 17 high DataFrames can be sorted by column: In [17]: # sorting values df.sort values(by='epsilon') Out[17]: alpha beta gamma delta epsilon 3 12 13 14 15 high 16 17 18 high 0 0 1 2 3 low 10 low 4 5 6 7 medium Load data from file Most often data will come from somewhere, often csv files, and using pd.read_csv() will allow smooth creation of DataFrames. Let's load that same heart-attack.csv that we used in Numpy before: data = pd.read fwf('auto-mpg.data', names = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration In [18]: After loading data, it is good practice to check what we have. Usually, the sequences is: 1. Check dimension 2. Peek at the first rows 3. Get info on data types and missing values 4. Summarize columns In [19]: # Check dimension (rows, columns) data.shape (398, 9) Out[19]: In [20]: # Peek at the first rows data.head() Out[20]: cylinders displacement horsepower weight acceleration model year origin car name **0** 18.0 8 307.0 130.0 3504.0 12.0 70 1 "chevrolet chevelle malibu" **1** 15.0 350.0 165.0 3693.0 11.5 70 "buick skylark 320" 2 18.0 8 318.0 150.0 3436.0 11.0 70 "plymouth satellite" 304.0 150.0 3433.0 12.0 70 "amc rebel sst" 3 16.0 "ford torino" 17.0 8 302.0 140.0 3449.0 10.5 70 1 In [21]: # Column names are data.columns Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', Out[21]: 'acceleration', 'model year', 'origin', 'car name'], dtype='object') In [22]: # Get info on data types and missing values data.info() <class 'pandas.core.frame.DataFrame'> RangeIndex: 398 entries, 0 to 397 Data columns (total 9 columns): # Column Non-Null Count Dtype _____ mpg 398 non-null float64
cylinders 398 non-null int64 0 mpg 2 displacement 398 non-null float64 3 horsepower 398 non-null object 4 weight 398 non-null float64 acceleration 398 non-null float64 model year 398 non-null int64 7 origin 398 non-null int64 8 car name 398 non-null object dtypes: float64(4), int64(3), object(2) memory usage: 28.1+ KB Summarize values What is the mean, std, min, max in each column? In [25]: data.mean() /var/folders/jd/ympy718d20q7cvpwl8zztvfr0000gn/T/ipykernel 2256/531903386.py:1: FutureWarning: Dropping of nuis ance columns in DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will ra ise TypeError. Select only valid columns before calling the reduction. data.mean() 23.514573 Out[25]: mpg 5.454774 cylinders displacement 193.425879 2970.424623 weight 15.568090 acceleration 76.010050 model year 1.572864 origin dtype: float64 In [26]: # where are the other columns? Check data types data.dtypes float64 Out[26]: mpg cylinders int64 displacement float64 horsepower object weight float64 acceleration float64 model year int64 origin int64 car name object dtype: object Notice that many columns are of type object, which is not a number. Maybe this has to do with missing values? We know from peeking at the first rows that there are '?' values in there. Let's replace these with the string NaN for not-a-number. In [27]: # replace '?' with 'NaN' data = data.replace({'?': 'NaN'}) data.head() Out[27]: mpg cylinders displacement horsepower weight acceleration model year origin car name 18.0 307.0 130.0 3504.0 12.0 1 "chevrolet chevelle malibu" 8 15.0 350.0 165.0 3693.0 11.5 70 "buick skylark 320" 2 8 318.0 150.0 3436.0 70 18.0 11.0 1 "plymouth satellite" 3 16.0 304.0 150.0 3433.0 12.0 70 "amc rebel sst" 17.0 8 302.0 140.0 3449.0 10.5 70 "ford torino" Pandas knows that 'NaN' probably means that numbers are missing. Now we can convert the data type from object to float In [28]: # convert dtypes data = data.iloc[:,0:8].astype('float') data.dtypes mpg float64
cylinders float64 mpg Out[28]: displacement float64 horsepower float64 float64 acceleration float64 model year float64 float64 dtype: object We could have loaded the data with the na_values argument to indicate that '?' means missing number: In [29]: data = pd.read_fwf('auto-mpg.data', names = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration data.dtypes float64 mpg Out[29]: cylinders int64 displacement float64 float64 horsepower float64 weight acceleration float64 model year int64 int64 origin car name object dtype: object This worked nicely. Now we can describe all columns, meaning printing basic statistics. Note that by default Pandas ignores NaN, whereas Numpy does not. data.describe() # ignores NaN In [30]: origin Out[30]: cylinders displacement horsepower weight acceleration model year mpg count 398.000000 398.000000 398.000000 392.000000 398.000000 398.000000 398.000000 398.000000 23.514573 5.454774 2970.424623 15.568090 76.010050 1.572864 mean 193.425879 104.469388 std 7.815984 1.701004 104.269838 38.491160 846.841774 2.757689 3.697627 0.802055 9.000000 3.000000 68.000000 46.000000 1613.000000 8.000000 70.000000 1.000000 min 25% 17.500000 4.000000 104.250000 75.000000 2223.750000 13.825000 73.000000 1.000000 50% 23.000000 4.000000 148.500000 93.500000 2803.500000 76.000000 1.000000 15.500000 75% 29.000000 8.000000 262.000000 126.000000 3608.000000 17.175000 79.000000 2.000000 24.800000 46.600000 8.000000 455.000000 230.000000 5140.000000 82.000000 3.000000 max We could be interested by these statistics in each of the genders. To get these, we first group values by gender, then ask for the description. We will only look at age for clarity In [31]: data.groupby(by='origin').describe().mpg Out[31]: count std min 25% 50% **75**% mean origin **1** 249.0 20.083534 6.402892 9.0 15.0 18.5 24.00 39.0 70.0 27.891429 6.723930 16.2 26.5 30.65 44.3 24.0 3 79.0 30.450633 6.090048 18.0 31.6 34.05 46.6 25.7 Find NaNs How many NaNs in each column? We can ask which entries are null, which produces a boolean array data.isnull() In [32]: Out[32]: mpg cylinders displacement horsepower weight acceleration model year origin car name O False False False False False False False False False 1 False False False False False False False False False 2 False 4 False False False False False False False False False **393** False False False False False False False False False 394 False False False False False False False False False **395** False False False False False False False False False 396 False False False False False False False False False **397** False False False False False False False False False 398 rows × 9 columns Applying sum() to this boolean array will count the number of True values in each column data.isnull().sum() In [33]: mpg Out[33]: cylinders displacement horsepower 6 weight acceleration model year origin car name dtype: int64 We get complementary information from info() data.info() In [34]: <class 'pandas.core.frame.DataFrame'> RangeIndex: 398 entries, 0 to 397 Data columns (total 9 columns): Non-Null Count Column Dtype mpg 398 non-null float64
cylinders 398 non-null int64
displacement 398 non-null float64
horsepower 392 non-null float64
weight 398 non-null float64
acceleration 398 non-null float64
model year 398 non-null int64 0 1 model year 398 non-null int64 7 origin 398 non-null int64 car name 398 non-null int64 object dtypes: float64(5), int64(3), object(1) memory usage: 28.1+ KB We can fill (replace) these missing values, for example with the minimum value in each column data.fillna(data.min()).describe() In [35]: Out[35]: cylinders displacement horsepower weight acceleration mpg model year origin count 398.000000 398.000000 398.000000 398.000000 398.000000 398.000000 398.000000 398.000000 mean 23.514573 5.454774 193.425879 103.587940 2970.424623 15.568090 76.010050 1.572864 1.701004 846.841774 std 7.815984 104.269838 38.859575 2.757689 3.697627 0.802055 3.000000 9.000000 68.000000 46.000000 1613.000000 8.000000 70.000000 1.000000 min 25% 17.500000 4.000000 104.250000 75.000000 2223.750000 13.825000 73.000000 1.000000 23.000000 4.000000 76.000000 50% 148.500000 92.000000 2803.500000 15.500000 1.000000 **75%** 29.000000 8.000000 262.000000 125.000000 3608.000000 17.175000 79.000000 2.000000 46.600000 24.800000 3.000000 8.000000 455.000000 230.000000 5140.000000 82.000000 max Count unique values (a histogram) We finish off, with our good friend the histogram data['mpg'].value counts() In [36]: 13.0 20 Out[36]: 14.0 19 18.0 17 15.0 16 26.0 14 31.9 16.9 18.2 22.3 44.0 Name: mpg, Length: 129, dtype: int64 In []: In []: