**Working with Data**

Working with data details the earliest processes of data science problem solving. The 1st step is to identify the problem, which determines all else that needs to be done. The 2nd step is to gather data. The 3rd step is to wrangle (munge) data, which is critical. Wrangling is getting data into a form that is useful for machine learning and other data science problems. Of course, wrangled data will probably have to be cleaned. The 4th step is to visualize the data. Visualization helps you get to know the data and, hopefully, identify patterns. This chapter presents the following examples:

1. One-dimensional data example
2. Two-dimensional data example
3. Data correlation and basic statistics
4. Pandas correlation and heat map examples
5. Various visualization examples
6. Cleaning a CSV file with Pandas and JSON
7. Slicing and dicing
8. Data cubes
9. Data scaling and wrangling

*One-Dimensional Data Example*

The code example generates visualizations of two very common data distributions – uniform and normal. The uniform distribution has constant probability. That is, all events that belong to the distribution are equally probable. The normal distribution is symmetrical about the center, which means that 50% of its values are less than the mean and 50% of its values are greater than the mean. Its shape resembles a bell curve. The normal distribution is extremely important because it models many naturally occurring events.  
  
import matplotlib.pyplot as plt

import numpy as np

if \_\_name\_\_ == "\_\_main\_\_":

plt.figure('Uniform Distribution')

uniform = np.random.uniform(-3, 3, 1000)

count, bins, ignored = plt.hist(uniform, 20, facecolor='lime')

plt.xlabel('Interval: [-3, 3]')

plt.ylabel('Frequency')

plt.title('Uniform Distribution')

plt.axis([-3,3,0,100])

plt.grid(True)

plt.figure('Normal Distribution')

normal = np.random.normal(0, 1, 1000)

count, bins, ignored = plt.hist(normal, 20, facecolor='fuchsia')

plt.xlabel('Interval: [-3, 3]')

plt.ylabel('Frequency')

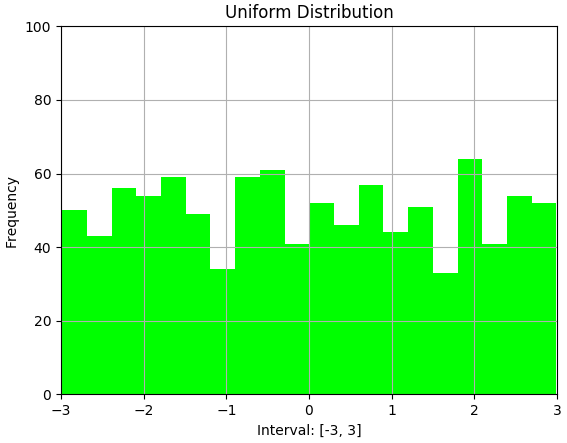
plt.title('Normal Distribution')

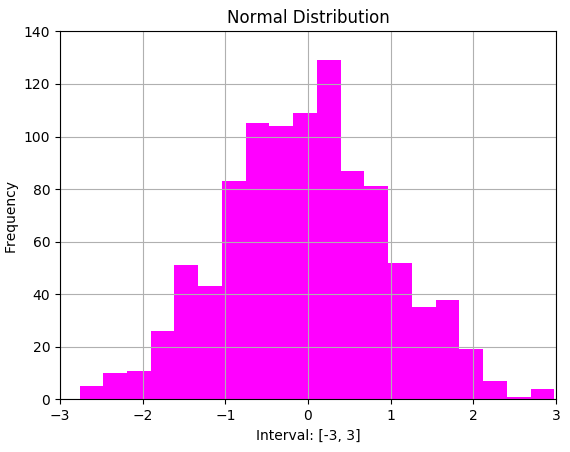
plt.axis([-3,3,0,140])

plt.grid(True)

plt.show()

Output:





The code example begins by importing matplotlib and numpy. The main block begins by creating a figure and data for a uniform distribution. Next, a histogram is created and plotted based on the data. A figure for a normal distribution is then created and plotted.

*Two-Dimensional Data Example*

Modeling 2-dimensional data offers a more realistic picture of naturally occurring events. The code example compares two normally distributed distributions of randomly generated data with the same mean and standard deviation (SD). SD measures the amount of variation (dispersion) of a set of data values. Although both data sets are normally distributed with the same mean and SD, each has a very different joint distribution (correlation). Correlation is the interdependence of two variables.

import matplotlib.pyplot as plt

import matplotlib.gridspec as gridspec

import numpy as np, random

from scipy.special import ndtri

def inverse\_normal\_cdf(r):

return ndtri(r)

def random\_normal():

return inverse\_normal\_cdf(random.random())

def scatter(loc):

plt.scatter(xs, ys1, marker='.', color='black', label='ys1')

plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')

plt.xlabel('xs')

plt.ylabel('ys')

plt.legend(loc=loc)

plt.tight\_layout()

if \_\_name\_\_ == "\_\_main\_\_":

xs = [random\_normal() for \_ in range(1000)]

ys1 = [ x + random\_normal() / 2 for x in xs]

ys2 = [-x + random\_normal() / 2 for x in xs]

gs = gridspec.GridSpec(2, 2)

fig = plt.figure()

ax1 = fig.add\_subplot(gs[0,0])

plt.title('ys1 data')

n, bins, ignored = plt.hist(ys1, 50, normed=1,

facecolor='chartreuse', alpha=0.75)

ax2 = fig.add\_subplot(gs[0,1])

plt.title('ys2 data')

n, bins, ignored = plt.hist(ys2, 50, normed=1,

facecolor='fuchsia', alpha=0.75)

ax3 = fig.add\_subplot(gs[1,:])

plt.title('Correlation')

scatter(6)

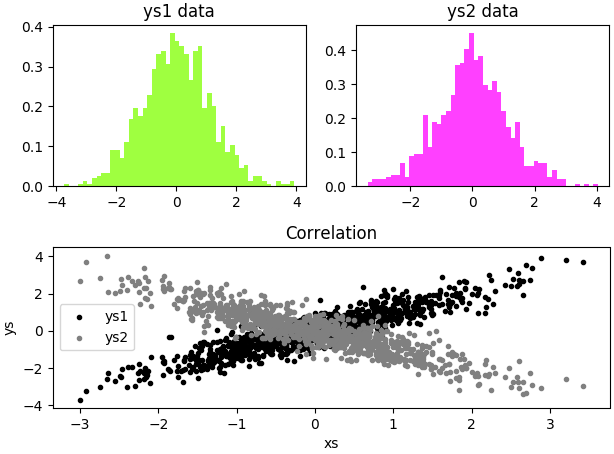
print (np.corrcoef(xs, ys1)[0, 1])

print (np.corrcoef(xs, ys2)[0, 1])

plt.show()

Output:





The code example begins by importing matplotlib, numpy, random, and scipy libraries. Method gridspec specifies the geometry of a grid where a subplot will be placed. Method ndtri returns the standard normal cumulative distribution function (CDF). CDF is the probability that a random variable X takes on a value less than or equal to x, where x represents the area under a normal distribution. The code continues with three functions. Function inverse\_normal\_cdf() returns the CDF based on a random variable. Function random\_normal() calls function inverse\_normal\_cdf() with a randomly generated value X and returns the CDF. Function scatter() creates a scatter plot. The main block begins by creating randomly generated x and y values xs, ys1, and ys2. A gridspec() is created to hold the distributions. Histograms are created for xs, ys1 and xs, ys2 data respectively. Next, a correlation plot is created for both distributions. Finally, correlations are generated for the two distributions.

The code example spawns two important lessons. First, creating a set of randomly generated numbers with ndtri() creates a normally distributed dataset. That is, function ndtri() returns the CDF of a randomly generated value. Second, two normally distributed datasets are not necessarily similar even though they look alike. In this case, the correlations are opposite. So, visualization and correlations are required to demonstrate the difference between the datasets.

*Data Correlation and Basic Statistics*

Correlation is the extent that two or more variables fluctuate (move) together. A correlation matrix is a table showing correlation coefficients between sets of variables. Correlation coefficients measure strength of association between two or more variables.

The code example creates three datasets with x and y coordinates, calculates correlations, and plots. The 1st dataset represents a positive correlation. The 2nd a negative correlation. The 3rd a weak correlation.

import random, numpy as np

import matplotlib.pyplot as plt

import matplotlib.gridspec as gridspec

if \_\_name\_\_ == "\_\_main\_\_":

np.random.seed(0)

x = np.random.randint(0, 50, 1000)

y = x + np.random.normal(0, 10, 1000)

print ('highly positive:\n', np.corrcoef(x, y))

gs = gridspec.GridSpec(2, 2)

fig = plt.figure()

ax1 = fig.add\_subplot(gs[0,0])

plt.title('positive correlation')

plt.scatter(x, y, color='springgreen')

y = 100 - x + np.random.normal(0, 10, 1000)

print ('\nhighly negative:\n', np.corrcoef(x, y))

ax2 = fig.add\_subplot(gs[0,1])

plt.title('negative correlation')

plt.scatter(x, y, color='crimson')

y = np.random.normal(0, 10, 1000)

print ('\nno/weak:\n', np.corrcoef(x, y))

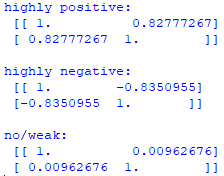
ax3 = fig.add\_subplot(gs[1,:])

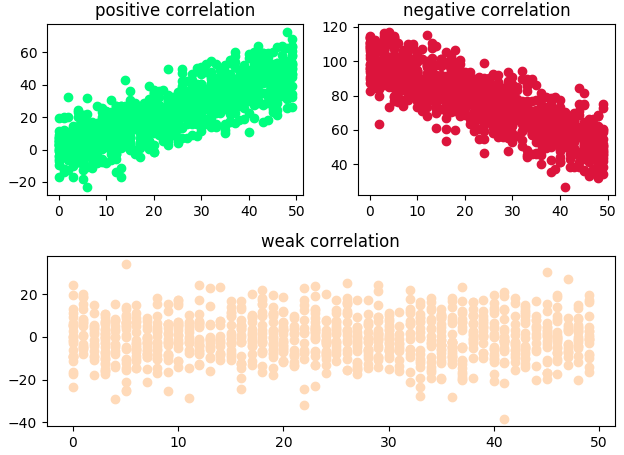
plt.title('weak correlation')

plt.scatter(x, y, color='peachpuff')

plt.tight\_layout()

plt.show()  
  
Output:





The code example begins by importing random, numpy, and matplotlib libraries. The main block begins by generating x and y coordinates with a positive correlation and displaying the correlation matrix. It continues by creating a grid to hold the subplot, the 1st subplot grid, and a scatterplot. Next, x and y coordinates are created with a negative correlation and the correlation matrix is displayed. The 2nd subplot grid is created and plotted. Finally, x and y coordinates are created with a weak correlation and the correlation matrix is displayed. The 3rd subplot grid is created and plotted, and all 3 scatterplots are displayed.

*Pandas Correlation and Heat Map Examples*

Pandas is a Python package that provides fast, flexible, and expressive data structures to make working with virtually any type of data easy, intuitive, and practical in real world data analysis. A DataFrame (df) is a 2-dimensional labeled data structure and the most commonly used object in pandas.

The 1st code example creates a correlation matrix with an associated visualization:

import random, numpy as np, pandas as pd

import matplotlib.pyplot as plt

import matplotlib.cm as cm

import matplotlib.colors as colors

if \_\_name\_\_ == "\_\_main\_\_":

np.random.seed(0)

df = pd.DataFrame({'a': np.random.randint(0, 50, 1000)})

df['b'] = df['a'] + np.random.normal(0, 10, 1000)

df['c'] = 100 - df['a'] + np.random.normal(0, 5, 1000)

df['d'] = np.random.randint(0, 50, 1000)

colormap = cm.viridis

colorlist = [colors.rgb2hex(colormap(i))

for i in np.linspace(0, 1, len(df['a']))]

df['colors'] = colorlist

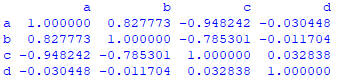
print (df.corr())

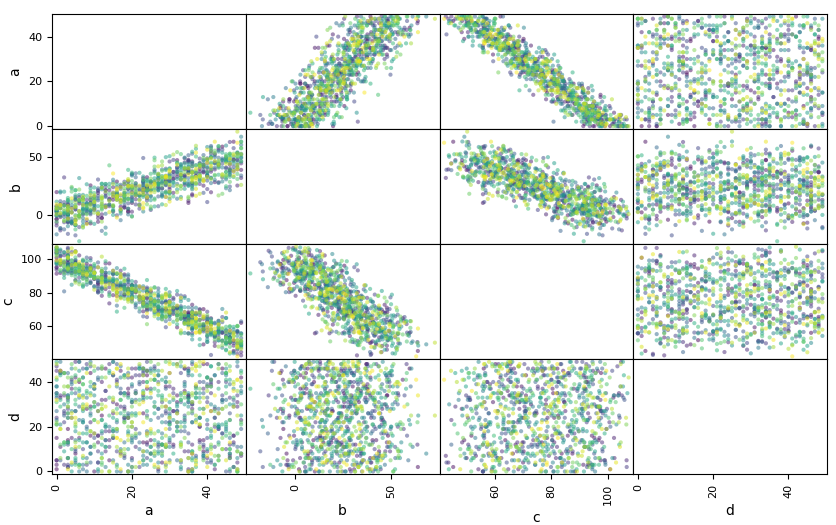
pd.plotting.scatter\_matrix(df, c=df['colors'], diagonal='d',

figsize=(10, 6))

plt.show()

Output:





The code example begins by importing random, numpy, pandas, and matplotlib libraries. The main block begins by creating a df with four columns populated by various random number possibilities. It continues by creating a color map of the correlations between each column, printing the correlation matrix, and plotting the color map.

We can see from the correlation matrix that the most highly correlated variables are a and b (0.83), a and c (-0.95), and b and c (-0.79). From the color map, we can see that a and b are positively correlated, a and c are negatively correlated, and b and c are negatively correlated. However, the actual correlation values are not apparent from the visualiztion.

A Heat map is a graphical representation of data where individual values in a matrix are represented as colors. It is a popular visualization technique in data science. With pandas, a Heat map provides a sophisticated visualization of correlations where each variable is represented by its own color.

The 2nd code example uses a Heat map to visualize variable correlations:

import random, numpy as np, pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

if \_\_name\_\_ == "\_\_main\_\_":

np.random.seed(0)

df = pd.DataFrame({'a': np.random.randint(0, 50, 1000)})

df['b'] = df['a'] + np.random.normal(0, 10, 1000)

df['c'] = 100 - df['a'] + np.random.normal(0, 5, 1000)

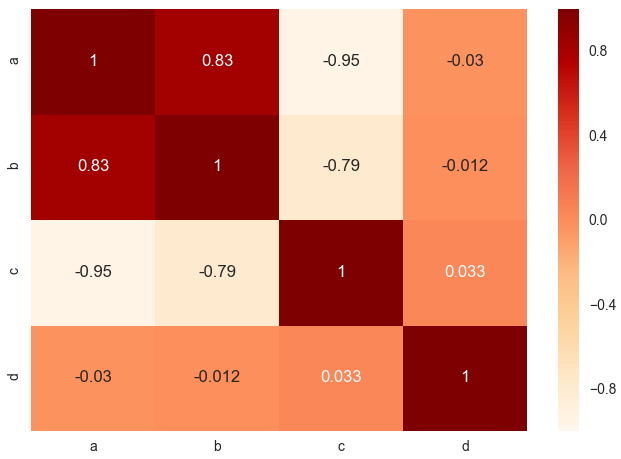
df['d'] = np.random.randint(0, 50, 1000)

plt.figure()

sns.heatmap(df.corr(), annot=True, cmap='OrRd')

plt.show()

Output:



The code begins by importing random, numpy, pandas, matplotlib, and seaborn libraries. Seaborn is a Python visualization library based on matplotlib. The main block begins by generating four columns of data (variables), and plots a Heat map. Attribute cmap uses a colormap. A list of matplotlib colormaps can be found at: <https://matplotlib.org/examples/color/colormaps_reference.html>.

*Various Visualization Examples*

The 1st code example introduces the Andrews curve, which is a way to visualize structure in high-dimensional data. Data for this example is the Iris dataset, which is one of the best known in the pattern recognition literature. The Iris dataset consists of three different types of irises’ (Setosa, Versicolour, and Virginica) petal and sepal lengths.

Andrews curves allow multivariate data plotting as a large number of curves that are created using the attributes (variable) of samples as coefficients. By coloring the curves differently for each class, it is possible to visualize data clustering. Curves belonging to samples of the same class will usually be closer together and form larger structures. Raw data for the iris dataset is located at the following URL:

<https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/data/iris.csv>

import matplotlib.pyplot as plt

import pandas as pd

from pandas.plotting import andrews\_curves

if \_\_name\_\_ == "\_\_main\_\_":

data = pd.read\_csv('data/iris.csv')

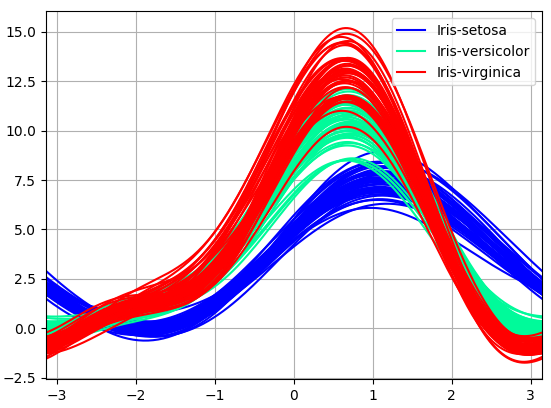
plt.figure()

andrews\_curves(data, 'Name',

color=['b','mediumspringgreen','r'])

plt.show()

Output:



The code example begins by importing matplotlib and pandas. The main block begins by reading the iris dataset into pandas df data. Next, Andrews curves are plotted for each class – Iris-setosa, Iris-versicolor, and Iris-virginica. From this visualization, it is difficult to see which attributes are more distinctly define a class.

The 2nd code example introduces parallel coordinates:

import matplotlib.pyplot as plt

import pandas as pd

from pandas.plotting import parallel\_coordinates

if \_\_name\_\_ == "\_\_main\_\_":

data = pd.read\_csv('data/iris.csv')

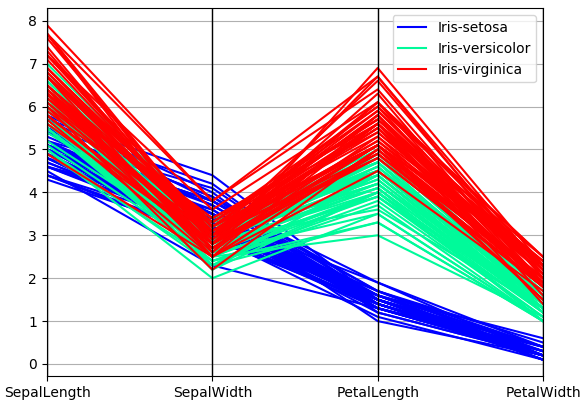
plt.figure()

parallel\_coordinates(data, 'Name',

color=['b','mediumspringgreen','r'])

plt.show()

Output:



Parallel coordinates is another technique for plotting multivariate data. It allows visualization of clusters in data and estimation of other statistics visually. Points are represented as connected line segments. Each vertical line represents one attribute. One set of connected line segments represents one data point. Points that tend to cluster appear closer together.

The code example begins by importing matplotlib and pandas. The main block begins by reading the iris dataset into pandas df data. Next, parallel coordinates are plotted for each class. From this visualization, attributes PetalLength and PetalWidth are most distinct for the three species (classes of Iris). So, PetalLength and PetalWidth are the best classifiers for species of Iris. Andrews curves just don’t clearly provide this important information.

A nice URL:

<http://wilkelab.org/classes/SDS348/2016_spring/worksheets/class9.html>

The 3rd code example introduces RadViz:

import matplotlib.pyplot as plt

import pandas as pd

from pandas.plotting import radviz

if \_\_name\_\_ == "\_\_main\_\_":

data = pd.read\_csv('data/iris.csv')

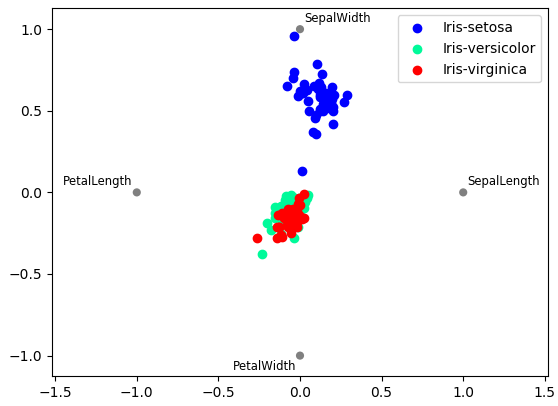
plt.figure()

radviz(data, 'Name',

color=['b','mediumspringgreen','r'])

plt.show()

Output:



RadVis is yet another technique for visualizing multivariate data. The code example begins by importing matplotlib and pandas. The main block begins by reading the iris dataset into pandas df data. Next, RadVis coordinates are plotted for each class. With this visualization, it is not easy to see any distinctions. So, the parallel coordinates technique appears to be the best of the three in terms of recognizing variation.

*Cleaning a CSV File with Pandas and JSON*

The code example loads a dirty CSV file into a Pandas df and displays to locate bad data. It then loads the same CSV file into a list of dictionary elements for cleaning. Finally, the cleansed data is saved to JSON.

import csv, pandas as pd, json

def to\_dict(d):

return [dict(row) for row in d]

def dump\_json(f, d):

with open(f, 'w') as f:

json.dump(d, f)

def read\_json(f):

with open(f) as f:

return json.load(f)

if \_\_name\_\_ == "\_\_main\_\_":

df = pd.read\_csv("data/audio.csv")

print (df, '\n')

data = csv.DictReader(open('data/audio.csv'))

d = to\_dict(data)

for row in d:

if (row['pno'][0] not in ['a', 'c', 'p', 's']):

if (row['pno'][0] == '8'):

row['pno'] = 'a' + row['pno']

elif (row['pno'][0] == '7'):

row['pno'] = 'p' + row['pno']

elif (row['pno'][0] == '5'):

row['pno'] = 's' + row['pno']

if (row['color']) == '-':

row['color'] = 'silver'

if row['model'] == '-':

row['model'] = 'S1'

if (row['mfg']) == '100':

row['mfg'] = 'Linn'

if (row['desc'] == '0') and row['pno'][0] == 'p':

row['desc'] = 'preamplifier'

elif (row['desc'] == '-') and row['pno'][0] == 's':

row['desc'] = 'speakers'

if (row['price'][0] == '$'):

row['price'] =\

row['price'].translate({ord(i): None for i in '$,.'})

json\_file = 'data/audio.json'

dump\_json(json\_file, d)

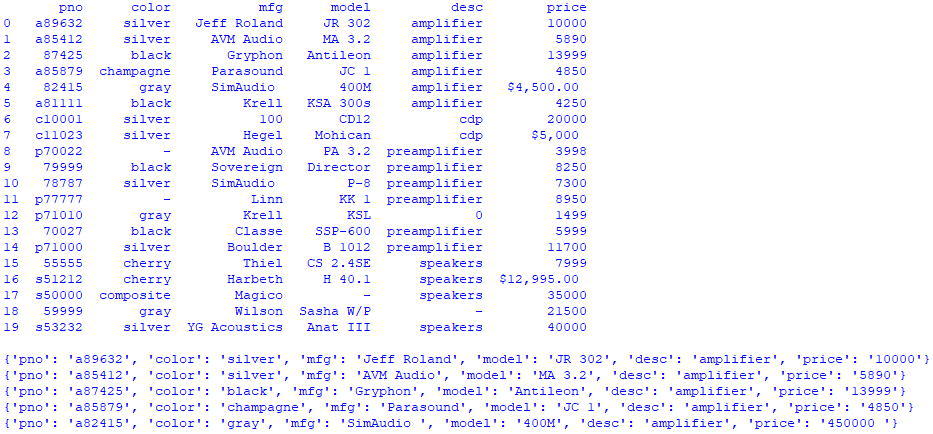
data = read\_json(json\_file)

for i, row in enumerate(data):

if i < 5:

print (row)

Output:



The code example begins by importing csv, pandas, and json libraries. Function to\_dict() converts a list of OrderedDict elements to a list of regular dictionary elements for easier processing. Function dump\_json() saves data to a JSON file. Function read\_json() reads JSON data into a Python list. The main block begins by loading a CSV file into a Pandas df and displaying it to visualize dirty data. It continues by loading the same CSV file into a list of dictionary elements for easier cleansing. Next, all dirty data is cleansed. The code continues by saving the cleansed data to JSON file audio.json. Finally, audio.json is loaded and a few records are displayed to ensure that everything worked properly.

*Slicing and Dicing*

Slicing and dicing is breaking data into smaller parts or views to better understand and present it as information in a variety of different and useful ways. A slice in multidimensional arrays is a column of data corresponding to a single value for one or more members of the dimension of interest. While a slice filters on a particular attribute, a dice is like a zoom feature that selects a subset of all dimensions, but only for specific values of the dimension.

The code example loads audio.json into a Pandas df, slices data by column and row, and displays:

import pandas as pd

if \_\_name\_\_ == "\_\_main\_\_":

df = pd.read\_json("data/audio.json")

amps = df[df.desc == 'amplifier']

print (amps, '\n')

price = df.query('price >= 40000')

print (price, '\n')

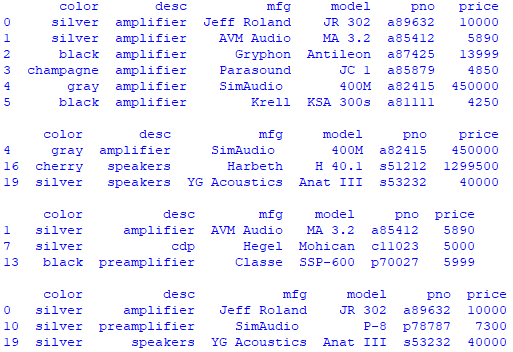
between = df.query('4999 < price < 6000')

print (between, '\n')

row = df.loc[[0, 10, 19]]

print (row)

Output:



The code example begins by importing Pandas. The main block begins by loading audio.json into a Pandas df. Next, the df is sliced by amplifier from the desc column. The code continues by slicing by the price column for equipment more expensive than $40,000. The next slice is by price column for equipment between $5,000 and $6,000. The final slice is by rows 0, 10, and 19.

*Data Cubes*

A data cube is a n-dimensional array of values. Since it is hard to conceptualize a n-dimensional cube, most are 3-dimensional in practice.

Let’s build a cube that holds three stocks – GOOGL, AMZ, and MKL. For each stock, include 5 days of data. Each day includes data for open, high, low, close, adj close, and volume values. So, the 3 dimensions are stock, day, and values. Data was garnered from actual stock quotes.

The code example creates a cube, saves it to a JSON file, reads the JSON, and displays some information:

import json

def dump\_json(f, d):

with open(f, 'w') as f:

json.dump(d, f)

def read\_json(f):

with open(f) as f:

return json.load(f)

def rnd(n):

return '{:.2f}'.format(n)

if \_\_name\_\_ == "\_\_main\_\_":

d = dict()

googl = dict()

googl['2017-09-25'] =\

{'Open':939.450012, 'High':939.750000, 'Low':924.510010,

'Close':934.280029, 'Adj Close':934.280029, 'Volume':1873400}

googl['2017-09-26'] =\

{'Open':936.690002, 'High':944.080017, 'Low':935.119995,

'Close':937.429993, 'Adj Close':937.429993, 'Volume':1672700}

googl['2017-09-27'] =\

{'Open':942.739990, 'High':965.429993, 'Low':941.950012,

'Close':959.900024, 'Adj Close':959.900024, 'Volume':2334600}

googl['2017-09-28'] =\

{'Open':956.250000, 'High':966.179993, 'Low':955.549988,

'Close':964.809998, 'Adj Close':964.809998, 'Volume':1400900}

googl['2017-09-29'] =\

{'Open':966.000000, 'High':975.809998, 'Low':966.000000,

'Close':973.719971, 'Adj Close':973.719971, 'Volume':2031100}

amzn = dict()

amzn['2017-09-25'] =\

{'Open':949.309998, 'High':949.419983, 'Low':932.890015,

'Close':939.789978, 'Adj Close':939.789978, 'Volume':5124000}

amzn['2017-09-26'] =\

{'Open':945.489990, 'High':948.630005, 'Low':931.750000,

'Close':937.429993, 'Adj Close':938.599976, 'Volume':3564800}

amzn['2017-09-27'] =\

{'Open':948.000000, 'High':955.299988, 'Low':943.299988,

'Close':950.869995, 'Adj Close':950.869995, 'Volume':3148900}

amzn['2017-09-28'] =\

{'Open':951.859985, 'High':959.700012, 'Low':950.099976,

'Close':956.400024, 'Adj Close':956.400024, 'Volume':2522600}

amzn['2017-09-29'] =\

{'Open':960.109985, 'High':964.830017, 'Low':958.380005,

'Close':961.349976, 'Adj Close':961.349976, 'Volume':2543800}

mkl = dict()

mkl['2017-09-25'] =\

{'Open':1056.199951, 'High':1060.089966, 'Low':1047.930054,

'Close':1050.250000, 'Adj Close':1050.250000, 'Volume':23300}

mkl['2017-09-26'] =\

{'Open':1052.729980, 'High':1058.520020, 'Low':1045.000000,

'Close':1045.130005, 'Adj Close':1045.130005, 'Volume':25800}

mkl['2017-09-27'] =\

{'Open':1047.560059, 'High':1069.099976, 'Low':1047.010010,

'Close':1064.040039, 'Adj Close':1064.040039, 'Volume':21100}

mkl['2017-09-28'] =\

{'Open':1064.130005, 'High':1073.000000, 'Low':1058.079956,

'Close':1070.550049, 'Adj Close':1070.550049, 'Volume':23500}

mkl['2017-09-29'] =\

{'Open':1068.439941, 'High':1073.000000, 'Low':1060.069946,

'Close':1067.979980, 'Adj Close':1067.979980 , 'Volume':20700}

d['GOOGL'], d['AMZN'], d['MKL'] = googl, amzn, mkl

json\_file = 'data/cube.json'

dump\_json(json\_file, d)

d = read\_json(json\_file)

s = ' '

print ('\'Adj Close\' slice:')

print (10\*s, 'AMZN', s, 'GOOGL', s, 'MKL')

print ('Date')

print ('2017-09-25', rnd(d['AMZN']['2017-09-25']['Adj Close']),

rnd(d['GOOGL']['2017-09-25']['Adj Close']),

rnd(d['MKL']['2017-09-25']['Adj Close']))

print ('2017-09-26', rnd(d['AMZN']['2017-09-26']['Adj Close']),

rnd(d['GOOGL']['2017-09-26']['Adj Close']),

rnd(d['MKL']['2017-09-26']['Adj Close']))

print ('2017-09-27', rnd(d['AMZN']['2017-09-27']['Adj Close']),

rnd(d['GOOGL']['2017-09-27']['Adj Close']),

rnd(d['MKL']['2017-09-27']['Adj Close']))

print ('2017-09-28', rnd(d['AMZN']['2017-09-28']['Adj Close']),

rnd(d['GOOGL']['2017-09-28']['Adj Close']),

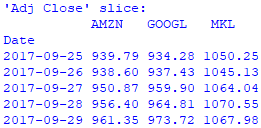
rnd(d['MKL']['2017-09-28']['Adj Close']))

print ('2017-09-29', rnd(d['AMZN']['2017-09-29']['Adj Close']),

rnd(d['GOOGL']['2017-09-29']['Adj Close']),

rnd(d['MKL']['2017-09-29']['Adj Close']))

Output:



The code example begins by importing json. Function dump\_json() and read\_json() save and read JSON data respectively. The main block creates a cube by creating a dictionary d, dictionaries for each stock, and adding data by day and attribute to each stock dictionary. The code continues by saving the cube to JSON file cube.json. Finally, the code reads cube.json and displays a slice from the cube.

*Data Scaling and Wrangling*

Data scaling is changing type, spread and/or position to compare data that are otherwise incomparable. Data scaling is very common in data science. Mean centering is the 1st technique, which transforms data by subtracting out the mean. Normalization is the 2nd technique, which transforms data to fall within the range between 0 and 1. Standardization is the 3rd technique, which transforms data to zero mean and unit variance (SD = 1), which is commonly referred to as standard normal.

The 1st code example generates and centers a normal distribution:

import numpy as np

import matplotlib.pyplot as plt

def rnd\_nrml(m, s, n):

return np.random.normal(m, s, n)

def ctr(d):

return [x-np.mean(d) for x in d]

if \_\_name\_\_ == "\_\_main\_\_":

mu, sigma, n, c1, c2, b = 10, 15, 100, 'pink',\

'springgreen', True

s = rnd\_nrml(mu, sigma, n)

plt.figure()

ax = plt.subplot(211)

ax.set\_title('normal distribution')

count, bins, ignored = plt.hist(s, 30, color=c1, normed=b)

sc = ctr(s)

ax = plt.subplot(212)

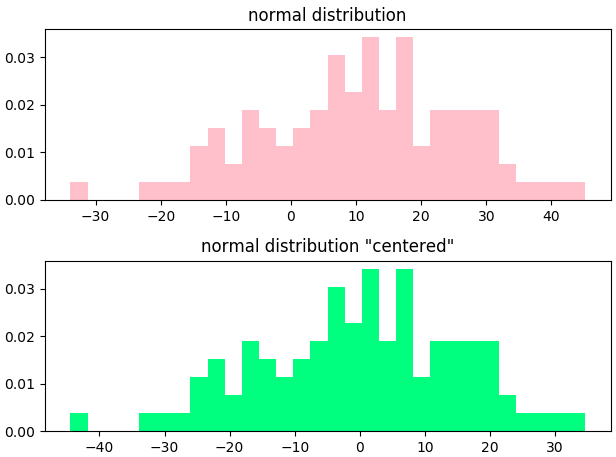
ax.set\_title('normal distribution "centered"')

count, bins, ignored = plt.hist(sc, 30, color=c2, normed=b)

plt.tight\_layout()

plt.show()

Output:



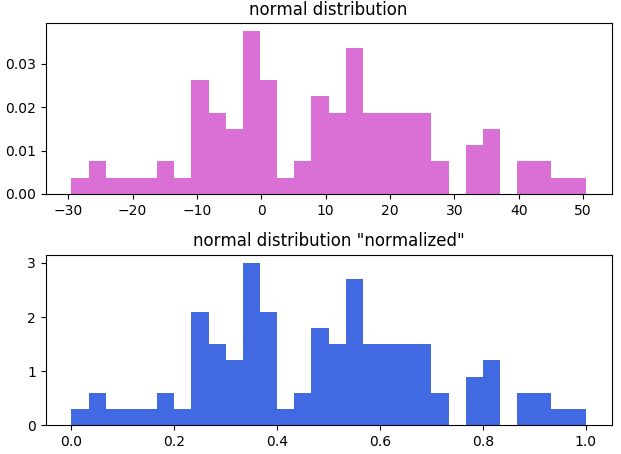
The code example begins by importing numpy and matplotlib. Function rnd\_nrml() generates a normal distribution based on mean (mu), SD (sigma), and n number of data points. Function ctr() subtracts out the mean from every data point. The main block begins by creating the normal distribution. The code continues by plotting the original and centered distributions. Notice that the distributions are exactly the same, but the 2nd distribution is centered with mean of 0.

The 2nd code example generates and normalizes a normal distribution:

import numpy as np  
import matplotlib.pyplot as plt  
  
def rnd\_nrml(m, s, n):  
 return np.random.normal(m, s, n)  
  
def nrml(d):  
 return [(x-np.amin(d))/(np.amax(d)-np.amin(d)) for x in d]  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 mu, sigma, n, c1, c2, b = 10, 15, 100, 'orchid',\  
 'royalblue', True  
 s = rnd\_nrml(mu, sigma, n)  
 plt.figure()  
 ax = plt.subplot(211)  
 ax.set\_title('normal distribution')

count, bins, ignored = plt.hist(s, 30, color=c1, normed=b)  
 sn = nrml(s)  
 ax = plt.subplot(212)  
 ax.set\_title('normal distribution "normalized"')  
 count, bins, ignored = plt.hist(sn, 30, color=c2, normed=b)  
 plt.tight\_layout()  
 plt.show()

Output:



The code example begins by importing numpy and matplotlib. Function rnd\_nrml() generates a normal distribution based on mean (mu), SD (sigma), and n number of data points. Function nrml() transforms data to fall within the range between 0 and 1. The main block begins by creating the normal distribution. The code continues by plotting the original and normalized distributions. Notice that the distributions are exactly the same, but the 2nd distribution is normalized between 0 and 1.

The 3rd code example transforms data to zero mean and unit variance (standard normal):  
  
import numpy as np, csv

import matplotlib.pyplot as plt

def rnd\_nrml(m, s, n):

return np.random.normal(m, s, n)

def std\_nrml(d, m, s):

return [(x-m)/s for x in d]

if \_\_name\_\_ == "\_\_main\_\_":

mu, sigma, n, b = 0, 1, 1000, True

c1, c2 = 'peachpuff', 'lime'

s = rnd\_nrml(mu, sigma, n)

plt.figure(1)

plt.title('standard normal distribution')

count, bins, ignored = plt.hist(s, 30, color=c1, normed=b)

plt.plot(bins, 1/(sigma \* np.sqrt(2 \* np.pi)) \*

np.exp( - (bins - mu)\*\*2 / (2 \* sigma\*\*2) ),

linewidth=2, color=c2)

start1, start2 = 5, 600

mu1, sigma1, n, b = 10, 15, 500, True

x1 = np.arange(start1, n+start1, 1)

y1 = rnd\_nrml(mu1, sigma1, n)

mu2, sigma2, n, b = 25, 5, 500, True

x2 = np.arange(start2, n+start2, 1)

y2 = rnd\_nrml(mu2, sigma2, n)

plt.figure(2)

ax = plt.subplot(211)

ax.set\_title('dataset1 (mu=10, sigma=15)')

count, bins, ignored = plt.hist(y1, 30, color='r', normed=b)

ax = plt.subplot(212)

ax.set\_title('dataset2 (mu=5, sigma=5)')

count, bins, ignored = plt.hist(y2, 30, color='g', normed=b)

plt.tight\_layout()

plt.figure(3)

ax = plt.subplot(211)

ax.set\_title('Normal Distributions')

g1, g2 = (x1, y1), (x2, y2)

data = (g1, g2)

colors = ('red', 'green')

groups = ('dataset1', 'dataset2')

for data, color, group in zip(data, colors, groups):

x, y = data

ax.scatter(x, y, alpha=0.8, c=color, edgecolors='none',

s=30, label=group)

plt.legend(loc=4)

ax = plt.subplot(212)

ax.set\_title('Standard Normal Distributions')

ds1 = (x1, std\_nrml(y1, mu1, sigma1))

y1\_sn = ds1[1]

ds2 = (x2, std\_nrml(y2, mu2, sigma2))

y2\_sn = ds2[1]

g1, g2 = (x1, y1\_sn), (x2, y2\_sn)

data = (g1, g2)

for data, color, group in zip(data, colors, groups):

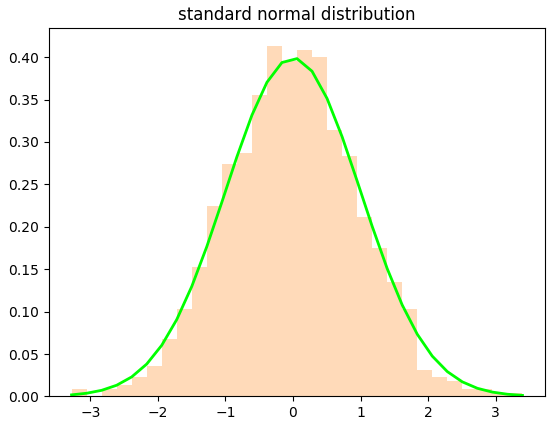
x, y = data

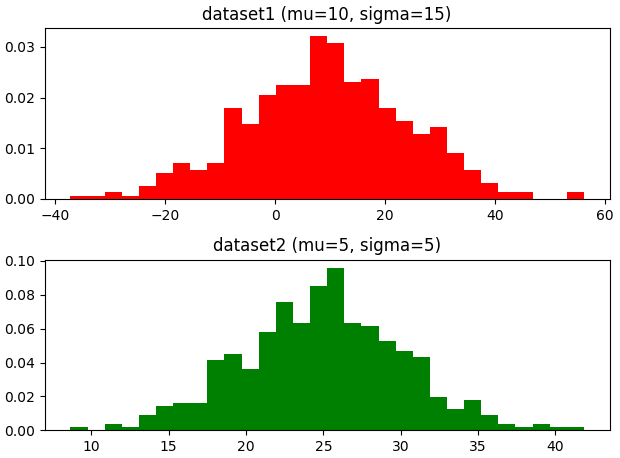
ax.scatter(x, y, alpha=0.8, c=color, edgecolors='none',

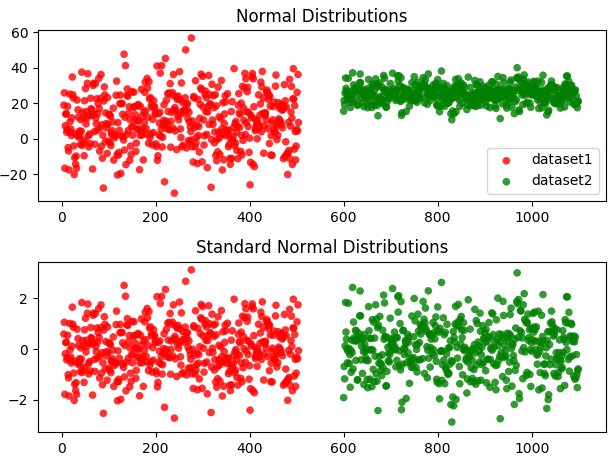
s=30, label=group)

plt.tight\_layout()

plt.show()  
  
Output:





  
  
The code example begins by importing numpy and matplotlib. Function rnd\_nrml() generates a normal distribution based on mean (mu), SD (sigma), and n number of data points. Function std\_nrml() transforms data to standard normal. The main block begins by creating a standard normal distribution as a histogram and a line. The code continues by creating and plotting two different normally distributed datasets. Next, both data sets are rescaled to standard normal and plotted. Now, the datasets can be compared to each other. Although the original plots of the datasets appear to be very different, they are actually very similar distributions.  
  
The 4th code example reads a CSV dataset, saves it to JSON, wrangles it, and prints a few records. The URL for the data is: [**https://community.tableau.com/docs/DOC-1236**](https://community.tableau.com/docs/DOC-1236). However, the data on this site changes, so please use the data from our website to work with this example.

import csv, json

def read\_dict(f):

return csv.DictReader(open(f))

def to\_dict(d):

return [dict(row) for row in d]

def dump\_json(f, d):

with open(f, 'w') as fout:

json.dump(d, fout)

def read\_json(f):

with open(f) as f:

return json.load(f)

def mk\_data(d):

for i, row in enumerate(d):

e = {}

e['\_id'] = i

e['cust'] = row['Customer Name']

e['item'] = row['Sub-Category']

e['sale'] = rnd(row['Sales'])

e['quan'] = row['Quantity']

e['disc'] = row['Discount']

e['prof'] = rnd(row['Profit'])

e['segm'] = row['Segment']

yield e

def rnd(v):

return str(round(float(v),2))

if \_\_name\_\_ == "\_\_main\_\_":

f= 'data/superstore.csv'

d = read\_dict(f)

data = to\_dict(d)

jsonf = 'data/superstore.json'

dump\_json(jsonf, data)

print ('"superstore" data added to JSON\n')

json\_data = read\_json(jsonf)

print ("{:20s} {:15s} {:10s} {:3s} {:5s} {:12s} {:10s}".

format('CUSTOMER', 'ITEM', 'SALES', 'Q', 'DISC',

'PROFIT', 'SEGMENT'))

generator = mk\_data(json\_data)

for i, row in enumerate(generator):

if i < 10:

print ("{:20s} {:15s}".format(row['cust'], row['item']),

"{:10s} {:3s}".format(row['sale'], row['quan']),

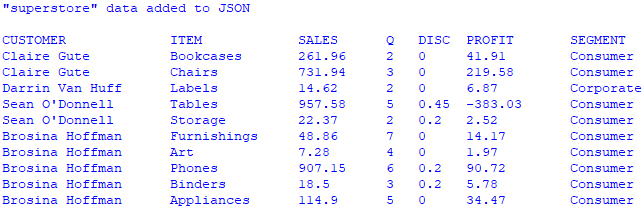
"{:5s} {:12s}".format(row['disc'], row['prof']),

"{:10s}".format(row['segm']))

else:

break

Output:



The code example begins by importing csv and json libraries. Function read\_dict() reads a CSV file as an OrderedDict. Function to\_dict() converts an OrderedDict to a regular dictionary. Function dump\_json() saves a file to JSON. Function read\_json() reads a JSON file. Function mk\_data() creates a generator object consisting of wrangled data from the JSON file. Function rnd() rounds a number to 2 decimal places. The main block begins by reading a CSV file and converting it to JSON. The code continues by reading the newly created JSON data. Next, a generator object is created from the JSON data. The generator object is critical because it speeds processing orders of magnitude faster than a list. Since the dataset is close to 10,000 records, speed is important. To verify that the data was created correctly, the generator object is iterated a few times to print some of the wrangled records.

The 5th and final code example reads the JSON file created in the previous example, wrangles it, and saves the wrangled data set to JSON:

import json

def read\_json(f):

with open(f) as f:

return json.load(f)

def mk\_data(d):

for i, row in enumerate(d):

e = {}

e['\_id'] = i

e['cust'] = row['Customer Name']

e['item'] = row['Sub-Category']

e['sale'] = rnd(row['Sales'])

e['quan'] = row['Quantity']

e['disc'] = row['Discount']

e['prof'] = rnd(row['Profit'])

e['segm'] = row['Segment']

yield e

def rnd(v):

return str(round(float(v),2))

if \_\_name\_\_ == "\_\_main\_\_":

jsonf = 'data/superstore.json'

json\_data = read\_json(jsonf)

l = len(list(mk\_data(json\_data)))

generator = mk\_data(json\_data)

jsonf= 'data/wrangled.json'

with open(jsonf, 'w') as f:

f.write('[')

for i, row in enumerate(generator):

j = json.dumps(row)

if i < l - 1:

with open(jsonf, 'a') as f:

f.write(j)

f.write(',')

else:

with open(jsonf, 'a') as f:

f.write(j)

f.write(']')

json\_data = read\_json(jsonf)

for i, row in enumerate(json\_data):

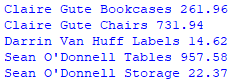
if i < 5:

print (row['cust'], row['item'], row['sale'])

else:

break

Output:



The code example imports json. Function read\_json() reads a JSON file. Function mk\_data() creates a generator object consisting of wrangled data from the JSON file. Function rnd() rounds a number to 2 decimal places. The main block begins by reading a JSON file. A generator object must be created twice. The 1st generator allows us to find the length of the JSON file. The 2nd generator consists of wrangled data from the JSON file. Next, the generator is traversed so we can create a JSON file of the wrangled data. Although the generator object is created and can be traversed very fast, it takes a bit of time to create a JSON file consisting of close to 10,000 wrangled records. On my machine, it took a bit over 33 second so be patient.