﻿ # -\*- coding: utf-8 -\*-

"""

Spyder Editor

This is a temporary script file.

"""

# -\*- coding: utf-8 -\*-

"""

Last amended: 11th October, 2019

Data file: marathon\_data.csv.zip

Ref:

1. https://seaborn.pydata.org/introduction.html

2. https://jakevdp.github.io/PythonDataScienceHandbook/04.14-visualization-with-seaborn.html

About what are splits in raunning, see:

https://en.wikipedia.org/wiki/Negative\_split

Objectives:

1. Time and date manipulation in pandas

2. Data manipulation using pandas

2. Graphics in pandas using seaborn

i. Bivariate distribution: sns.jointplot()

ii. Histogram: sns.distplot()

iii.Density plot: sns.kdeplot()

iv. Violinplot: sns.violinplot()

v. Box plots: sns.boxplot()

vi. Grid of plots: sns.PairGrid()

vii.Bar plots: sns.countplot() ; sns.barplot()

ix. Interpreting contour plots

"""

# 1.0 Reset memory and Call libraries

%reset -f

# 1.1 Data manipulation modules

import pandas as pd # R-like data manipulation

import numpy as np # n-dimensional arrays

# 1.2 For plotting

import matplotlib.pyplot as plt # For base plotting

# 1.3 Seaborn is a library for making statistical graphics

# in Python. It is built on top of matplotlib and

# numpy and pandas data structures.

# Install latest package as:

# conda install -c conda-forge seaborn

import seaborn as sns # Easier plotting

# 1.4 Misc

import os

# 1.5 Show graphs in a separate window

%matplotlib qt5

######### Begin

# 2.0 Set working directory

#os.chdir("C:\\Users\\ashok\\OneDrive\\Documents\\python")

os.chdir("E:/lalit/Teaching/Year\_2019\_20/Term\_V/Big\_Data\_Analytics\_for\_Managers/Lecture\_Slides/Lect\_06")

os.listdir()

# 2.1 Increase number of displayed columns

pd.options.display.max\_columns = 200

# 2.2 Read data file

data = pd.read\_csv("marathon\_data.csv.zip")

# 2.3 Explore data

data.columns

data.columns.values # names()

data.dtypes # age is int64; This is a luxury. check np.iinfo('int64') and int8

np.iinfo('uint16')

data.describe() # set include = 'all' to see summary of 'object' types also

data.info()

data.shape # dim()

data.head() # head()

data.tail() # tail()

#data.iloc[0,0] = np.nan

data.count()

data['age'].plot(kind = 'hist')

# 2.4 Values in gender columns

data['gender'].value\_counts() # Distribution

data['gender'].unique() # Which values

data['gender'].nunique()

# 3. Simple time/date manipulation

# Split datetime columns to its components

# Ref: http://pandas.pydata.org/pandas-docs/version/0.23/api.html#datetimelike-properties

# 3.1 First 'final'

data['final'] = pd.to\_datetime(data['final']) # Convert to datetime

data.dtypes

data['f\_hour'] = data['final'].dt.hour #np.iinfo('int64')

# 3.2 Now create columns and extract

data['f\_hour'] = data['final'].dt.hour.astype('uint16') # Default int64

data['f\_minute'] = data['final'].dt.minute.astype('uint16') # default int64

data['f\_second'] = data['final'].dt.second.astype('uint16')

data.dtypes

data.head()

data.shape

# 3.3 Then split the column 'split'

"""

What is a 'split' time in Marathon:

Splits: A race’s total time divided

into smaller parts (usually miles),

is known as the splits. If a runner

has an even split, it means they ran

the same pace through the entire race.

If it’s a negative split, they ran the

second half faster than the first.

And that’s a good thing!

'Split hour', here, indicates time to complete

the Ist half. Given a final time, the more

is the 'split hour', the 'more' is negative-split.

"""

# 3.4 Covert 'object' type to datetime type

data['split'] = pd.to\_datetime(data['split'])

data['s\_hour'] = data['split'].dt.hour.astype('uint16')

data['s\_minute'] = data['split'].dt.minute.astype('uint16')

data['s\_second'] = data['split'].dt.second.astype('uint16')

data.head()

data.shape

# 3.5 Calculate time difference between 'split' and 'final'

# Reword data['diff'] as data[IIndhalf]

data['diff'] = data['final'] - data['split']

data.dtypes # Note the datatype of data['diff']

# It is timedelta64

# 3.6 Some timedelta operations: airthmatic operations

# 3.6.1

data['diff'][0] - data['diff'][1]

# 3.6.2

data['diff'][0].total\_seconds()

# 3.6.3

data['final'][0].total\_seconds() # Not permitted on datetime

# 3.6.4

data['diff'] \* 6 # Six times

data.head()

# 4. Also cut age into three equal parts

# ELse: pd.cut(test.days, [0,30,60], include\_lowest=True)

data['age\_cat'] = pd.cut( data.age,

bins = 3,

labels=["y","m","h"]

)

data['age\_cat'].head()

data['age\_cat'].value\_counts()

# 4.1 qcut, cuts the age keeping almost equal freq distribution

data['age\_cat\_q'] = pd.qcut(data.age,

q = 3, # Either an integer or array of quantiles

# [0, .25, .5, .75, 1.]

labels = ["l", "m", "h"]

)

data['age\_cat\_q'].value\_counts()

# 5 Convert total time taken to seconds:

# Calculate total duration in seconds both in 'split' and 'final' run

data['split\_sec'] = data['s\_hour'] \* 3600 + data['s\_minute'] \* 60 + data['s\_second']

data['split\_sec'].head()

data['final\_sec'] = data['f\_hour'] \* 3600 + data['f\_minute'] \* 60 + data['f\_second']

data['final\_sec'].head()

np.min(data['final\_sec']) # 7731

np.max(data['final\_sec']) # 36068 well within datatype unit16 limits

# 5.1 Create another column in the data, the split\_fraction, which

# measures the degree to which each runner negative-splits or

# positive-splits the race:

data['split\_frac'] = 1 - 2 \* data['split\_sec'] / data['final\_sec']

# 5.2 Create a new column listing whether a person

# is in his twenties or thirtees

data['age\_dec'] = data.age.map(lambda age: 10 \* (age // 10)) #age in lambda is var

# 5.3 Create a column listing just if split is -nev or +ve

# See: http://pandas.pydata.org/pandas-docs/stable/indexing.html#why-does-assignment-fail-when-using-chained-indexing

data['posOrneg'] = "+ve"

# 5.4.1 This assignment fails. Why? (see below for example)

data.loc[data['split\_frac'] < 0, : ]['posOrneg'] = '-ve'

##\*\*\*\*\*\*\*\*\*\*Example\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Avoid chained assignments with fancy indexing

# Ref: https://github.com/pandas-dev/pandas/pull/5390#issuecomment-27654172

# Fancy indexing makes a copy not a view

# Chained assignment may make a copy of copy instead of

# changing the values in the 'first' copy

df = pd.DataFrame({"A": [1, 2, 3, 4, 5], "B": [3, 4, 3, 6, 7]})

df2 = pd.DataFrame({"A": [1, 2, 3, 4, 5], "B": [3., 4., 3., 6., 7.]})

df

df2

df.loc[0]["A"] = 0

df2.loc[0]["A"] = 0

df # This has changed

df2 # This has not changed

df.loc[1, "A"] = 0

df2.loc[1, "A"] = 0

df # This changes

df2 # This also changes

##\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 5.4.2 This succeeds

data.loc[data['split\_frac'] < 0, 'posOrneg'] = '-ve'

data['posOrneg'].nunique()

### 6. Some queries. Can be skipped

# How many are above 80

(data.age > 80).sum()

np.sum(data.age > 80)

# 6.1 Get data for males only

data[data.gender == 'M']

data.loc[data.gender == 'M', :]

# 6.2 Get data for those above age 60

data[data['age'] > 60]

data.loc[data['age'] > 60, : ]

# 6.3 Want to see only two columns for above, say age and gender:

data.loc[data['age'] > 60, ['age', 'gender'] ] # R-code: data[data['age'] > 60, c('age', 'gender') ]

data.loc[data['age'] > 60, data.columns.values[:2] ]

####### Plot now ############

"""

What is wrong with matplotlib?

a. Matplotlib's API is relatively low level. Doing sophisticated statistical

visualization is possible, but often requires a lot of code.

b. Matplotlib predated Pandas by more than a decade, and thus is not

designed for use with Pandas DataFrames. In order to visualize data from

a Pandas DataFrame, you must extract each Series and often concatenate

them together into the right format. It would be nicer to have a

plotting library that can intelligently use the DataFrame labels in a plot.

Seaborn

a. Good Defaults: Seaborn provides an API on top of Matplotlib that offers

sane choices for plot style and color defaults,

b. Simple Stat-graphs: Defines simple high-level functions for common

statistical plot types,

c. Pandas Dataframe: And integrates with the functionality provided by

Pandas DataFrames.

d. Seaborn under the hood uses matplotlib. So many matplotlib commands

can still be used.

"""

# 7. Plotting categorical variables: Bar plots

# 7.1 Bar plots: sns.countplot()

# Ref: http://seaborn.pydata.org/tutorial/categorical.html#categorical-tutorial

# 7.2 Simple bar plot

sns.countplot("age\_dec", data = data)

# 7.2.1 Get more control over this graph using matplotlib functions

fig = plt.figure(figsize = (5,5))

ax = fig.add\_subplot(111)

sns.countplot("age\_dec", data = data, ax = ax)

ax.set\_title("My first graph")

ax.set\_xlabel("Age in decades")

plt.show()

# 7.3 What about the distribution of age\_dec, gender wise

# In seaborn we do not have stacked bar-plots

# Note that legend appears automatically

fig = plt.figure(figsize = (5,5))

ax = fig.add\_subplot(111)

sns.countplot("age\_dec", # Variable whose distribution is of interest

hue= "gender", # Distribution will be gender-wise

data = data)

# 7.4

# First define descending\_order

# value\_counts() are generally sorted

descending\_order = list(data['age\_dec'].value\_counts().index)

descending\_order

sns.countplot("age\_dec", # Variable whose distribution is of interest

hue= "gender", # Subset: Distribution will be gender-wise

data = data,

order = descending\_order

)

# 7.5 Plot with three categories

# catplot() introduced in version 0.9. Check

# sns version, as: sns.\_\_version\_\_

# Upgrade seaborn as:

# conda install -c conda-forge seaborn

fig = plt.figure(figsize = (5,5))

ax = fig.add\_subplot(111)

sns.catplot(x="age\_dec", # Variable whose distribution (count) is of interest

hue="posOrneg", # Show distribution, pos or -ve split-wise

col="gender", # Create two-charts/facets, gender-wise

data=data,

kind="count"

)

# 8.. barplot: sns.barplot()

# This plot always takes two variables.

# Continuous: The second variable is continuous

# Categorical: The height of barplot is mean of continuous

fig = plt.figure(figsize = (5,5))

ax = fig.add\_subplot(111)

sns.barplot(x = "age\_dec", # Data is groupedby this variable

y= "split\_sec", # Aggregated by this variable

# Continuous variable. Bar-ht,

# by default, is 'mean' of this

hue= "gender", # Distribution is gender-wise

ci = 95, # Confidence interval. 95 is the default

estimator = np.mean, # This is the default. Try: np.median, np.std etc

data=data

)

"""

Error bars:

Error bars are graphical representations of the variability

of data and used on graphs to indicate the error or uncertainty

in a reported measurement. They give a general idea of how

precise a measurement is, or conversely, how far from the

reported value the true (error free) value might be. Error

bars often represent one standard deviation of uncertainty,

one standard error, or a particular confidence interval

(e.g., a 95% interval). These quantities are not the same and

so the measure selected should be stated explicitly in the graph

or supporting text.

"""

"""

9. Jointplots or scatter plots

Re: https://seaborn.pydata.org/generated/seaborn.jointplot.html

Draws a plot of two continuous variables.

There are both bivariate and univariate graphs.

"""

# 9.1 Simple scatter joint plot

fig = plt.figure(figsize = (5,5))

ax = fig.add\_subplot(111)

sns.jointplot("age",

"final\_sec",

data,

kind='scatter'

)

# 9.2 Why hexagonal binning?

# Same plot as above but with hex bins

# Read: https://www.meccanismocomplesso.org/en/hexagonal-binning/

"""

Hexagonal Binning is another way to manage the

problem of having to many points that start to

overlap. Hexagonal binning plots density, rather

than points. Points are binned into gridded hexagons

and distribution (the number of points per hexagon)

is displayed using either the color or the area of

the hexagons.

"""

sns.jointplot("age",

"final\_sec",

data,

kind='hex' # kind : { “scatter” | “reg” | “resid” | “kde” | “hex” }

)

# 9.3 Drawing multiple plots on the same axes

# 9.3.1 First draw a jointplot is a plot between split\_sec and final\_sec

g = sns.jointplot("split\_sec", "final\_sec", data, kind='hex')

# 9.3.2 Next use g axes object:

# On joint-axis, plot another graph

# The dotted line shows where someone's time would lie if they ran

# the marathon at a perfectly steady pace. The fact that the

# distribution lies above this indicates (as you might expect)

# that most people slow down over the course of the marathon.

# ie final > 2 \* split

g.ax\_joint.plot( # Plot y versus x as lines and/or markers.

np.linspace(4000, 16000), # x-axis

np.linspace(8000, 32000) # y-axis

) # For even split every point on

# the line is (x, 2\*x) or (x,y)

"""

10. Histograms

Ref: https://seaborn.pydata.org/tutorial/distributions.html

When dealing with a set of data, often the first thing

one wants to do is get a sense for how the variables

are distributed. We will give a brief introduction to

some of the tools in seaborn for examining univariate

and bivariate distributions.

"""

# 10.1. Histogram: sns.distplot()

# Out of nearly 40,000 participants, there were

# only 250 people who negative-split their marathon.

g = sns.distplot(data['split\_frac'],

kde=False # kde: Kernel density estimate plot

#bins = 50

)

type(g) # matplotlib axes

# g: It is the Axes object with the plot for further tweaking.

# Most seaborn plotting functions return axex object

g.axvline(0, # axvline and axhline are matplotlib functions

# Refer : https://matplotlib.org/api/\_as\_gen/matplotlib.pyplot.axvline.html

color="red",

linestyle="--"

)

# 10.2 Other options of distribution plots

sns.distplot(data['split\_frac'],

kde=False,

rug = True, # Show vertical lines at bottom, density of points wise

bins = 50

)

# 10.2.1 Quick graph from pandas plot

# Some graphs can be very revealing

dm = data.loc[data['gender'] == "M", ['age']].reset\_index(drop = True)

dw = data.loc[data['gender'] == "W", ['age']].reset\_index(drop=True)

df = pd.concat([dm,dw], axis = 1)

df.columns = ['mage', "wage"]

df.head(2)

df.plot(kind = 'hist', subplots = True)

"""

10.3

Kernel Density Funcions:

Two types:

i) Single cont variable or single cont variable

grouped by a category

ii) Contour plots: Between two cont variables

How kde plots are drawn:

kde plots are highly computaionally intensive. The following is

worth reading. Briefly, at every point draw a kernel function.

Kernel function most used is Gaussian. Then sum up ovelapping

graphs into a smooth density function.

See here: https://seaborn.pydata.org/tutorial/distributions.html#kernel-density-estimation

And this example in Wikipedia:

https://en.wikipedia.org/wiki/Kernel\_density\_estimation#Example

Example of kernel functions?

https://en.wikipedia.org/wiki/Kernel\_(statistics)#Kernel\_functions\_in\_common\_use

"""

# 10.3.1 Single variable

# Method I

sns.distplot(data['split\_frac'],

kde=True,

rug = True,

bins = 50

)

# Method II

sns.kdeplot(

data['split\_frac'],

shade = True

)

# 10.4 Contour plots

# Two cont variables

data['gender'].value\_counts()

# 10.4.1 Sunset data by gender

mdata = data[data['gender'] == 'M']

wdata = data[data['gender'] == 'W']

# 10.4.2 Plot two contour plots together

# Run both the following together

sns.kdeplot(

mdata['age'],

mdata['final\_sec'],

cmap = 'Blues'

)

sns.kdeplot(

wdata['age'],

wdata['final\_sec'],

cmap = 'Reds'

)

# 11. Grid of plots: sns.PairGrid()

# Draw grid of scatter plots and histograms

# TAKES TIME TO DRAW

# 11.1 Take first a random sample of 1000 points

# and plot grid of contour plots

# As noted earlier, the way kde are drawn

# requires a lot of computaion

nosamples = 1000

rs = np.random.choice(data.shape[0], nosamples) # May replace 5000 with 1000

df = data.iloc[rs, :]

len(df)

# 11.2 Which continuous variables to be plotted?

varsforgrid = ['age', 'split\_sec', 'final\_sec', 'split\_frac']

g = sns.PairGrid(df,

vars=varsforgrid, # Variables in the grid

hue='gender' # Variable as per which to map plot aspects to different colors.

)

# 11.3 What to plot in the diagonal? Histogram

g = g.map\_diag(plt.hist)

# 11.4 What to plot off-diagonal? Kernel Density plots

g.map\_offdiag(sns.kdeplot)

# 11.5

g.add\_legend();

"""

Interpretaion of Contour plots:

===============================

In this contour plot, pl note:

Contour plots will vary from student-to-student being sample

(we will concentrate on x = age, y = final\_sec)

i) The contours give the counts of pairs of (x,y)

ii) Each grid has two sets of contours--Women and Men

iii) Inner circle of each contour corresponds to max(count). Why?

Match location of inner circle with the maxima of histogram

Innercircles are therefore 'peaks'

iv) The width of histogram (say, for age), matches the width of contours

both for men and women. 'Vertical width' will match corresponding

histogram of 'final\_sec' after it is rotated by 90 degrees.

v) In all plots above 'age', all contours have about the same width

So also the 'vertical width' of contours across the grid.

vi) The innermost circle of 'men' is displayed to right-bottom from the

inner-most circle of that of women. Meaning thereby, most women fall

in this group of higher age and lower final\_sec when compared to men.

That is, peaks are diagonally-across

vii) Lastly contour-plots of Men, so to say, contain or surround,

Women contours. It implies that 'women' form a rather narrow or closer

group than 'men'. Closer-group implies less variance.

viii) You may note woman contours in other charts also.

ix) Suppose it were a classification problem and we were to predict 'Gender'

Then, in some contours clear separation of peaks (or valleys) of

Men and Women show that these data-columns can be used for distinguishing.

"""

# 12. Kernel Density plot: sns.kdeplot()

# The difference between men and women here is interesting. Let's look at

# the histogram of split fractions for these two groups:

# The interesting thing here is that there are many more men

# than women who are running close to an even split!

g=sns.kdeplot( # 'g' is the axes object

data.split\_frac[data.gender=='M'],

label='men',

shade=True

)

#12.1 How to draw the following kde plot on the same axes?

sns.kdeplot(

data.split\_frac[data.gender=='W'],

label='women',

shade=True,

ax = g # <== See this. Use the same axes object here

)

#12.2

plt.xlabel('split\_frac');

# 14. Box plots: sns.boxplot()

# Between the age-groups 20-40 there are large number

# of outliers. This is natural as in this age-group

# a lot of competition would exist

"""

https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51

A boxplot is a standardized way of displaying the distribution

of data based on a five number summary:

(“minimum”, first quartile (Q1), median, third quartile (Q3), and “maximum”)

maximum whisker: Q3 + 1.5\*IQR

minimum whisker: Q1 -1.5\*IQR

Suspected outliers: between 1.5IQR and 3IQR

Confirmed outliers: > 3IQR

"""

sns.boxplot(

"age\_dec",

"split\_frac",

data= data

)

# 15. Violinplot: sns.violinplot()

# A nice way to compare distributions, say gender wise, is to use a violin plot

sns.violinplot("gender", "split\_frac", data=data ) # x-axis has categorical variable

#15.1

sns.violinplot( "split\_frac", "gender", data=data ) # y-axis has categorical variable

# 16. look a little deeper, and compare these violin plots as a function of age.

# Looking at this, we can see where the distributions of men and women differ:

# the split distributions of men in their 20s to 50s show a pronounced over-density

# toward lower splits when compared to women of the same age (or of any age, for that matter).

sns.violinplot("age\_dec", "split\_frac",

hue="gender",

data=data,

split=True, # If hue variable has two levels, draw half of a violin for each level.

inner="quartile" # Options: “box”, “quartile”, “point”, “stick”, None

)

########### Rest is upto you ##########################

##########2Amber####

#

# Objective: Color quantization--Simple experiment

# 1.0 Call libraries

%reset -f

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.pyplot import imshow

from sklearn.cluster import KMeans

# 2.0 Create an image

data = np.random.randint(low = 0, high= 255, size = (30,10,3))

data

# 2.1 Show image

imshow(data)

# 3.0 Prepare for clustering

dt = data.reshape(300,3)

dt.shape

dt

# 3.1 Cluster now

km = KMeans(n\_clusters=4)

out = km.fit(dt)

# 3.2

cc = out.cluster\_centers\_

cc.shape

cc

# 3.3 Labels of clusters

out.labels\_

# 4.0 Create new image now with reduced colors

newdata = cc[out.labels\_] # shape 300 X 3

trans\_data = np.ceil(newdata.reshape(30,10,3))

trans\_data = trans\_data.astype(int)

imshow(trans\_data)

plt.figure(2)

plt.title("New Image")

plt.imshow(trans\_data)

################################################

######3 Amber######

## K Means Clustering ##

#%reset -f

#Using k-means to cluster data

#Getting Ready - import libraries and data

import numpy as np

import pandas as pd

#data

from sklearn.datasets import make\_blobs

blobs, classes = make\_blobs(500, centers=3)

# for plotting

import matplotlib.pyplot as plt

#%matplotlib qt5

#A simple example that clusters blobs of fake data

f, ax = plt.subplots(figsize=(7.5, 7.5))

rgb = np.array(['r', 'g', 'b'])

ax.scatter(blobs[:, 0], blobs[:, 1], color=rgb[classes])

ax.set\_title("Blobs")

#we'll pretend we know that there are three centers

from sklearn.cluster import KMeans

kmean = KMeans(n\_clusters=5)

kmean.fit(blobs)

'''

KMeans(algorithm='auto', copy\_x=True, init='k-means++', max\_iter=300,

n\_clusters=3, n\_init=10, n\_jobs=1, precompute\_distances='auto',

random\_state=None, tol=0.0001, verbose=0)

'''

print("My Clusters are =", kmean.cluster\_centers\_)

#plottigg

f1, ax1 = plt.subplots(figsize=(7.5, 7.5))

ax1.scatter(blobs[:, 0], blobs[:, 1], color=rgb[classes])

ax1.scatter(kmean.cluster\_centers\_[:, 0],kmean.cluster\_centers\_[:,1],

marker='\*', s=250,color='black', label='Centers')

ax1.set\_title("Blobs")

ax1.legend(loc='best')

f.show()

#kmean.labels\_[:5]

print("Clusters Labels are are =", kmean.labels\_[:5])

#array([2, 0, 1, 1, 0])

##########

####4 amber###

# -\*- coding: utf-8 -\*-

"""

##

## Objectives:

## i) Learn to process images: read/reshape/save

## ii) Reduce no of colours in a colour palette

## using k-means

## Steps:

# 1. Read any image (skimage.imread/plt.imread)

# Shape: 419 X 640 X 3

# 2. Reshape it to: 268160 X 3 (np.reshape)

# That is, each pixel falls in a row with its

# three colour intensities listed

# 3. Scale above numpy table/color values by dividing by 255

# 4. We have three columns of 270000 rows each

# Cluster them into 64 clusters (KMeans)

# 5. Find cluster labels of each row (clust\_labels)

# 6. Replace each RGB row by its respective

# cluster-center (model.cluster\_centers\_[clust\_labels])

# 7. Reshape image back and plot it

Ref:

http://scikit-learn.org/stable/auto\_examples/cluster/plot\_color\_quantization.html#sphx-glr-auto-examples-cluster-plot-color-quantization-py

"""

%reset -f # rm(list = ls) ; gc()

## 1. Call libraries

import numpy as np

# 1.1. For displaying graphics

import matplotlib.pyplot as plt

# 1.2. For image reading/manipulation

# Images can be manipulated using opencv, pillow and skimage

# Install skimage as:

## conda install -c anaconda scikit-image

from skimage.io import imread # Read image

from skimage.io import imshow # Display image

from skimage.io import imsave # Save image

# 1.3 For clustering

from sklearn.cluster import KMeans

# 1.4 OS related

import os

import time # Measuring process time

# 2. Set working folder and read file

os.chdir(""E:\\lalit\\Teaching\\Year\_2019\_20\\Term\_V\\Big\_Data\_Analytics\_for\_Managers\\Lecture\_Slides\\Lect\_08")

# 2.1 Read the image file

china= plt.imread("china.jpg")

china # Image is a multi-dimensional array

# Three colour channels (RGB) of 419 X 640 each

china.shape # 419 X 640 X 3

### 2.2 Some Experiment on image array

# Observe some colour values in each frame

china[0,0,0] , china[0,0,1] , china[0,0,2]

# 2.3 What are max and min colour intensites

np.min(china)

np.max(china)

##############################################

# Experiment begins on reshaping image

##############################################

# 2.4 Reshaping image and reshaping back. Is it restored?

# Extract colour intensity values form

test = china[120:124, 116:121, 0:3]

test

# 2.4.1 Its shape?

test.shape # (4,5,3)

test # Or 4-rows of 5-pixels each

# Inner array is RGB coord of each pixel

china[120,116,0] # 29

china[120,116,1] # 33

china[121,116,0] # 10

china[121,116,1] # 14

china[120,117,0] # 11

china[120,117,1] # 15

# 2.4.2 Now reshape it in a 2-d array

test1 = test.reshape(20,3) # 20 rows X 3 cols

# 2.4.3 Compare the following two: one reshaped

# and the other not

test1

test

# 2.4.4 And reshape back. Does it compare with original?

test1.reshape(4,5,3)

### Experiment Ends

##############################################

# 3. Reshape china image

newchina = china.reshape(china.shape[0] \* china.shape[1],

china.shape[2]

)

newchina.shape

# 3.1 Normalize all image colors

newchina = newchina/255

newchina.shape

# 3.2 Observe normalized R-G-B colors of top-10 points

newchina[:10, : ]

# 4. Perform clustering of R-G-B

# Set kmeans parameters. Get 64 colours

# 4.1 Instantiate the class

model = KMeans(n\_clusters = 64 )

# 4.2 Perform kmeans clustering (10 minutes)

start = time.time()

clust\_labels = model.fit\_predict(X = newchina)

end = time.time()

print((end - start)//60)

# 5. Look at cluster labels

clust\_labels

# 5.1 How many labels

len(clust\_labels)

# 6. And get 64 cluster centers

cent=model.cluster\_centers\_ # Use model.<tab> to get 'model' attributes

cent

cent.shape

# 6.1 For each cluster label, get RGB values

ff = cent[clust\_labels] # model.cluster\_centers\_[clust\_labels]

ff.shape

########################################

## 6.2 Another better way of copying:

########################################

# 6.3

b = np.zeros((newchina.shape[0],3))

# 6.4

for i in range(newchina.shape[0]):

b[i] = cent[clust\_labels[i]]

# 6.5

ff = b

######################################

# 7. Get image back by reshaping

modiImage = ff.reshape(419,640,3)

# 8. Show 64-color image

plt.figure(1)

plt.title('Quantized image (64 colors)')

plt.imshow(modiImage)

# 9. Show original image

plt.figure(2)

plt.title("Original image")

plt.imshow(china)

# 10. Save image and check size. It is reduced.

plt.imsave("modiImage.jpeg", modiImage)

########################################

# http://scikit-learn.org/stable/modules/clustering.html

# http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

def doCluster(X, nclust=64):

model = KMeans(nclust)

model.fit(X)

clust\_labels = model.predict(X)

cent = model.cluster\_centers\_

return (clust\_labels,cent)

clust\_labels,cent = doCluster(newchina,64)

############################################

"""

How pixels are arranged in an image:

====================================

For clarity we have taken four rows, five columns and two frames--All different.

The array output is as follows:

china[120:124, 116:121, 0:2]

Out[61]:

array([[[29, 33], |

[11, 15], |

[ 4, 8], | Row 120: five pixels 116,117,118,119,120, Col R & G

[15, 21], |

[19, 25]], |

[[10, 14], |

[ 8, 14], |

[12, 18], | Row 121: five pixels

[20, 26], |

[17, 23]], |

[[27, 28], |

[23, 24], |

[10, 14], | Row 122

[26, 30], |

[18, 22]], |

[[13, 13], |

[15, 15], |

[13, 13], | Row 123

[15, 15], |

[30, 31]]], dtype=uint8)|

china[120,116,0] => 29

china[120,116,1] => 33

china[121,116,0] => 10

china[121,116,1] => 14

china[120,117,0] => 11

china[120,117,1] => 15

After reshaping as below, the result is:

test.reshape((5 \*5,2))

Out[64]:

array([[29, 33],

[11, 15],

[ 4, 8],

[15, 21],

[19, 25],

[10, 14],

[ 8, 14],

[12, 18],

[20, 26],

[17, 23],

[27, 28],

[23, 24],

[10, 14],

[26, 30],

[18, 22],

[13, 13],

[15, 15],

[13, 13],

[15, 15],

[30, 31]], dtype=uint8)

uint8 is unsigned 8-bit integer

"""

##########Amber 4#######

######################

-\*- coding: utf-8 -\*-

"""

Last amended: 9th February, 2019

# 1http://nbviewer.jupyter.org/github/WeatherGod/AnatomyOfMatplotlib/blob/master/AnatomyOfMatplotlib-Part1-Figures\_Subplots\_and\_layouts.ipynb

# https://www.datacamp.com/community/tutorials/matplotlib-tutorial-python

# https://matplotlib.org/users/pyplot\_tutorial.html

Objectives:

Understanding basics of matplotlib

"""

%reset -f

import numpy as np

import matplotlib.pyplot as plt

%matplotlib qt5

# 1.0 Generate some simple data

x = np.arange(start = 1, stop = 20, step = 2) # xlim: [1,20)

y = np.arange(start = 0, stop = 10, step = 1) # ylim: [0,20)

# 1.1 Generate more data...

x1 = np.linspace(0, 10, 100)

y1, y2, y3 = np.cos(x1), np.cos(x1 + 1), np.cos(x1 + 2)

names = ['Signal 1', 'Signal 2', 'Signal 3']

"""

The Figure is the overall window or page

that everything is drawn on. It’s the top-level

component of all the ones that you will

consider in the following points. You can

create multiple independent Figures. A Figure

can have several other things in it, such as

a suptitle, which is a centered title to the

figure. You’ll also find that you can add a

legend and color bar, for example, to your

Figure.

To the figure you add Axes. The Axes is the

area on which the data is plotted with functions

such as plot() and scatter() and that can have

ticks, labels, etc. associated with it. This

explains why Figures can contain multiple Axes.

"""

# Step 1 Create a figure: fig = plt.figure()

# Step 2: Add subplot (ie axes): ax1 = fig.add\_subplot()

# Or both 1 & 2 together fig, ax = plt.subplots()

# Step 3: Select plot type and draw your plot: ax1.plot() or ax[0,1].plot()

# Step 4: Set axes properties with set\_: ax1.set\_xlim(), set\_title(), set\_xlabel(), set\_xticks()

# Step 5: Show plot: plt.show()

# 1. So begin with a figure:

fig = plt.figure()

# 1.1 All plotting is done with respect to an Axes.

# An Axes is made up of Axis objects and many other things.

# 1.2 How many axes?

ax = fig.add\_subplot(111)

# 1.3 Plot

ax.plot(x,y)

# 1.4 Plot description/properties

ax.set\_title("My plot")

ax.set\_xlim(left = 0, right = 20)

ax.set\_ylim(0,10)

ax.set\_xticks(ticks = list(range(0, 20, 1)) ,minor = True) # Specify tick points

ax.set\_xlabel("X-axis")

plt.show()

#LKJ

#fig, ax = plt.subplots(nrows = 2, ncols = 2)

#type(ax)

#ax.shape

#ax[1,1].plot([1,2,3], [1,2,3])

# ax[0,0].plot(x, y)

#Multiple plot on one axis

# 2.You can also set in one go, as below:

# In matplotlib.pyplot various states

# are preserved across function calls,

# so that it keeps track of things like

# the current figure and plotting area,

# and the plotting functions are directed

# to the current axes

# 2.1

fig = plt.figure()

# 2.2

ax1 = fig.add\_subplot(111)

# 2.3 Multiple plots on the SAME AXES: 'ax1'

ax1.plot(x1,y1)

ax1.plot(x1,y2)

ax1.plot(x1,y3)

# 2.4

ax1.set(title="My second plot" , xlim= [0,20], xlabel = "X-axis")

ax1.set\_xticks([0, 1,2,3,4,5] ) # minor = True

plt.show()

fig = plt.figure()

fig.add\_subplot(221)

#####################3

# equivalent but more general

ax1 = fig.add\_subplot(2, 2, 1)

# add a subplot with no frame

ax2 = fig.add\_subplot(222, frameon=False)

# add a polar subplot

fig.add\_subplot(223, projection='polar')

# add a red subplot that share the x-axis with ax1

fig.add\_subplot(224, sharex=ax1, facecolor='red')

\*#############

# 3. To make further plots let us read a simple

# data set

import os # has OS related methods

import pandas as pd # Pandas library

# 3.1 Set working directory

os.chdir("E:/lalit/Teaching/Year\_2019\_20/Term\_V/Big\_Data\_Analytics\_for\_Managers/Lecture\_Slides/Lect\_05")

os.listdir()

# 3.2 Display max number of columns

pd.options.display.max\_columns = 200

# 3.3 Read and explore

fl = pd.read\_csv("flavors\_of\_cacao.csv.zip")

fl.head()

fl.tail()

fl.dtypes

fl.shape

# 3.4 Bar chart of distribution of companies

ct = fl.company.value\_counts() # Eqt of table() in R

ct.index # label names

ct # A sorted series of values

# 3.5 Draw barplot now

fig = plt.figure()

ax = fig.add\_subplot(111)

# 3.5.1

ax.bar(ct.index[:10], # x-values or bar-locations

ct[:10], # height of bars

color = "lightblue", # inner-bar color (optional)

edgecolor="darkred" # optional

) # Top 10; Bottom use ct.tail()

#ax.bar?

# 3.5.2

ax.set\_xticklabels(labels = ct.index[:10], rotation = 90) # Not set\_xticks()

plt.show

# 4. Draw both bar-plot and scatter plot in the same figure

# but different axes

# 4.1 As usual create figure and also decide figsize

fig = plt.figure(figsize = (10,10))

# 4.2 Add subplot for bar-chart

ax1 = fig.add\_subplot(121)

# 4.3 Add subplot for scatter plot

ax2 = fig.add\_subplot(122)

# 4.4 Now plot

ax1.bar(ct.index[:10], ct[:10])

ax2.scatter(fl.rating, fl.cocoa\_percent)

plt.show()

# 4.5 Let us populate multiple axes using for-loop

fig = plt.figure(figsize = (10,10))

names = ['company', 'company\_location']

for i,j in enumerate(names):

ax = fig.add\_subplot(1,2,i+1)

ct = fl[j].value\_counts()

ax.bar(ct.index[:10], ct[:10])

plt.show()

# 5. Plots within for loop

# If a number of plots are to be made on

# the same axis, it is more convient

# to use plt.subplots(), as below:

# 5.1 Make bar-charts for 'company', company\_location',

# 'broad\_bean\_origin' and 'bean\_type'

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

names = ['company', 'company\_location', 'broad\_bean\_origin', 'bean\_type']

for i,j in enumerate(names):

ct = fl[j].value\_counts()

ax = fig.add\_subplot(2,2,i+1)

ax.bar(ct.index[:5],ct[:5])

plt.show()

# OR Use zip

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

names = ['company', 'company\_location', 'broad\_bean\_origin', 'bean\_type']

# We have to iterate over two things: Over ax and over names

# Use zip and ax.flat

# See topic: Multiple Axes in

# https://github.com/matplotlib/AnatomyOfMatplotlib/blob/master/AnatomyOfMatplotlib-Part1-Figures\_Subplots\_and\_layouts.ipynb

for i,j in zip(ax.flat,names):

ct = fl[j].value\_counts()

i.bar(ct.index[:5],ct[:5])

plt.show()

######################## I am done ##########################################

# 1 Make your first plot

plt.plot([1,2,3,4]) # By default x is [0,1,2,3]

plt.ylabel("some numbers")

plt.show()

# 1.1 Clear rhe current figure

plt.clf()

# 1.2

plt.plot([1,2,3,4],[5,6,7,8])

# 1.3 Clear rhe current figure

plt.clf()

# 2.0 Prepare numpy array data

x = np.linspace(0, 10, 100)

# 2.1 Plot this data

plt.plot(x, x, label='linear')

# 2.2 Add a legend

plt.legend()

plt.show()

# 3. Concatenate a color string with a line style string

# 'ro'

plt.plot([1,2,3,4], [1,4,9,16], 'ro')

plt.axis([0, 6, 0, 20]) # [xmin, xmax, ymin, ymax]

plt.show()

# 3.1 Mutiple plots and formatting

plt.plot([1,2,3,4], [1,2,3,4], 'g^',

[2,3,4,5], [2,4,5,7], 'bo') # Also use 'bs'

plt.axis([0,10, 0,10])

plt.show()

plt.gca()

plt.gcf()

"""

The following color abbreviations are supported:

========== ========

character color

========== ========

'b' blue

'g' green

'r' red

'c' cyan

'm' magenta

'y' yellow

'k' black

'w' white

========== ========

"""

# 3.3 One figure mutiple subplots

# Create a function that decreases

# exponentially cosine output

def f(t):

return np.exp(-t) \* np.cos(2\*np.pi\*t)

# 3.4

t1 = np.arange(0.0, 5.0, 0.1)

t2 = np.arange(0.0, 5.0, 0.02)

# 4. figure(1) will be created by default,

# just as a subplot(111) will be created

# by default if you don’t manually specify any axes.

plt.figure(1) # Optional will be created by default

plt.subplot(211) # numrows, numcols, plotnum. Create 2 X 1 plots

# Note below we create two plots

# on the same axes

plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

# Another plot on the second axes

plt.subplot(212) # numrows, numcols, plotnum

# IInd of the above

plt.plot(t2, np.cos(2\*np.pi\*t2), 'r--')

plt.show()

# 5. Multiple figures/window()

# each figure can contain as many axes

# & subplots as your heart desires:

def sinplot(t):

return (np.sin(2.0\*np.pi\*t))

# 5.1 First window

plt.figure(1)

# 5.2 Data

t1= np.arange(0,180, 0.5)

# 5.3 Plot on first axes

plt.subplot(2,1,1)

plt.plot(sinplot(sinplot(t1)))

# 5.4 Plot on second axes

plt.subplot(2,1,2)

plt.plot(sinplot(t1))

# 5.5 IInd figure

plt.figure(2)

plt.plot(np.tan(sinplot(t1)))

plt.grid(True)

# 5.6 Clear rhe current figure

plt.clf()

# Clear rhe current axes

plt.cla()

# https://www.datacamp.com/community/tutorials/matplotlib-tutorial-python

# 6. Setting legend in the figure

plt.plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)

plt.plot([1,2,3], [1,4,9], 'rs', label='line 2')

plt.axis([0, 4, 0, 10])

plt.legend()

##############Amber 5########

##########################

"""

Objectives:

i) Data structures in pandas: Series, DataFrame and Index

ii) Data structures usage

"""

import pandas as pd

import numpy as np

import os

########Series#############

## A. Creating Series

# 10. i) Series is an array

# ii) It is one-dimensional.

# iii) It is labeled by index or labels

# iv) Is dtype may be numeric or object

# 10.1 Exercises

s = pd.Series([2,4,8,10,55])

s

type(s)

s.name = "AA"

s

# 10.2 This is also a series but stores list objects

t = pd.Series({'a' : [1,2,3,4,], 'b' : [5,6]})

t

type(t)

# 10.3 Exercise

ss=[23,45,56]

h=pd.Series(ss)

h

# 10.4 OR generate it as:

h=pd.Series(range(23,30,2))

h

## B. Simple Operations

# 10.5 Exercise

s+h

s\*h

s-h

(s+h)[1] # Note the indexing starts from 0

s\*h[2]

s.mean()

s.std()

s.median()

## C. Series as ndarray

# 10.6 Also series behaves as ndarray

# Series acts very similarly to a ndarray,

# and is a valid argument to most NumPy functions.

np.mean(s)

np.median(s)

np.std(s)

## D. Indexing in series

# 10.7 Exercise

d=pd.Series([4,5], index=['a','b'])

e=pd.Series([6,7], index=['f','g'])

f=pd.Series([9,10], index=['a','b'])

d+e # All NaN

d+f

# 10.8 Reset index of 'd' and check

v = d.reset\_index()

type(v) # v is a DataFrame

# 10.9

d.reset\_index(

drop = True, # drop = False, adds existing index as

inplace = True # a new column and makes it a DataFrame

)

d

e.reset\_index(drop = True, inplace = True)

d + e

## E. Accessing Series

# 10.10 Exercise

j= pd.Series(np.random.normal(size=7))

k=j[j>0]

k=j[j>np.mean(j)]

k

# 10.11 Exercise

k = pd.Series(

np.random.normal(size=7),

index=['a','b','c','d','e','f','a']

)

k['a'] # 'a' is duplicate index

k.loc['a']

k[:2] # Show first two or upto 2nd index (0 and 1)

k.iloc[:2]

# 10.12

k.iloc[2:] # Start from 2nd index

k.iloc[2:4] # Start from IInd index upto 4th index

k.iloc[2:4].mean()

# 10.13 SURPRISE HERE!

k = pd.Series(np.random.normal(size=7),index=[0,2,5,3,4,1,6])

k.loc[0] # Access by index-name

k.loc[1] # Access by index-name

k.iloc[:2] # Access by position

k.iloc[[0,1,2]] # Access by index-name

k.take([0,1,2]) # Access by position

k.loc[[0,1,2]]

# 10.8 Exercise

# A series is like a dictionary. Can be accessed by its index (key)

e=pd.Series(np.random.uniform(0,5,7), index=['a','b','c','d','e','f','g'])

e

e['a' : 'e']

e.loc['a' : 'e']

e['a' : 'd'] # All values from 'a' to 'd'

e['b' : 'd']

e.take(['b' : 'd'])

e+k

######## DataFrame ###########

'''

DataFrame is a 2-dimensional labeled data structure with columns

of potentially different types. You can think of it like a spreadsheet

or SQL table, or a dict of Series objects. It is generally the most

commonly used pandas object. Like Series, DataFrame accepts many

different kinds of input.

'''

# 1

path = "E:/lalit/Teaching/Year\_2019\_20/Term\_V/Big\_Data\_Analytics\_for\_Managers\Lecture\_Slides\Lect\_05"

# 2

os.getcwd()

# 3

os.chdir(path)

# 4

data=pd.read\_csv("delhi\_weather\_data.zip")

type(data)

# 5.1

pd.options.display.max\_columns = 200

# 5.2

data.head()

data.tail()

data.dtypes

data.shape

data.columns

data.values

data.columns.values

data.describe()

# 6. Datetime conversions

data['datetime'] = pd.to\_datetime(data['datetime\_utc'])

data.head()

# 6.1

data['month'] = data['datetime'].dt.month

data['day'] = data['datetime'].dt.day

data['weekday'] = data['datetime'].dt.weekday

data['hour'] = data['datetime'].dt.hour

data['week'] = data['datetime'].dt.week

data.head()

# 6.2

pd.unique(data['\_conds']) # Unique values

data['\_conds'].nunique() # 39

data['\_conds'].value\_counts().sort\_values(ascending = False)

data.head()

# 7.0 Integer Selection

data.columns

data.iloc[3:5, 1:2] # 3:5 implies start from 3rd pos uptil 5-1=4th pos

data.iloc[3:5, 1:3] # Display column numbers 2nd and 3rd

data.iloc[3:5, :] # Display all columns

data.iloc[3:5, :] # Display all columns

data.iloc[1,1] # Same as df[1,1:2]. Treat 1 as lower bound

data.iloc[[3,5,7],[1,3]] # Specific rows and columns

data[data.month == 10 ].head() # Boolean indexing

data[(data.month == 10) & (data["\_conds"] == 'Smoke') ].head()

data[(data.\_conds == 'Smoke') | (data.\_wdire == 'East')]

# 8.0 Overall how many values are nulls

np.sum(data.isnull()).sort\_values(ascending = False)

# 9.0 Converting categorical variables to numeric

# sklearn's labelencoder is one way to do it

# Two step process:

# 1st. Convert dtype from 'object' to 'category'

# 2nd. Get integer-codes behind each category/level

# 3rd. Get correspondence behind category and integer

data['\_conds'] = data['\_conds'].astype('category') # Convert to categorical variable

data['int\_conds']=data['\_conds'].cat.codes # Create a column of integer coded categories

x = data[['\_conds', 'int\_conds']].values # Get dataframe as an array

out = set([tuple(i) for i in x]) # Get unique tuples of (code,category)

# 10.0 Memory reduction by changing datatypes

data.dtypes

# 10.1 Select data subset with dtype as 'float64'

newdata = data.select\_dtypes('float64')

# 10.2 What are max and min data values

np.min(np.min(newdata)) # -9999

np.max(np.max(newdata)) # 101061443.0

# 10.3 What are the limits of various float datatypes

np.finfo('float64') # finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308, dtype=float64)

np.finfo('float32') # finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)

np.finfo('float16') # finfo(resolution=0.001, min=-6.55040e+04, max=6.55040e+04, dtype=float16)

np.iinfo('int64')

np.iinfo('int16')

# 10.4 Change all columns to float32

# 10.4.1 What is the present memory usage

np.sum(newdata.memory\_usage()) # 8887200

# 10.4.2 Change data type now

for col in newdata.columns.values:

newdata[col] = newdata[col].astype('float32')

# 10.4.3 What is the current datausage

np.sum(newdata.memory\_usage()) # 4443640 (around 50% reduction)

###############################################################################

######AMBER 6

'''

Created by Dr. Lalit K Jiwani

Created on 2019/09/21

# Introduction to Python

'''

##Numpy ##

import numpy

numpy.\_\_version\_\_

import numpy as np

#to display all the contents of the numpy namespace

np.<TAB>

#to display NumPy’s built-in documentation

np?

#Creating more list

L = list(range(10))

L2 = [str(c) for c in L]

L3 = [True, "2", 3.0, 4]

#Fixed-Type Arrays in Python

import array

L = list(range(10))

A = array.array('i', L)

#Python’s array object provides efficient storage of array-based data,

# NumPy adds to this efficient operations on that data.

import numpy as np

#To create arrays from Python lists

np.array([1, 4, 2, 5, 3]) #integer array

#If types do not match, NumPy will upcast

np.array([3.14, 4, 2, 3])

#explicit datatype

np.array([1, 2, 3, 4], dtype='float32')

#NumPy arrays can explicitly be multidimensional

np.array([range(i, i + 3) for i in [2, 4, 6]])

#nested lists result in multidimensional arrays

## Creating Arrays from Scratch ##

#Create arrays from scratch using routines built into NumPy.

# Create a length-10 integer array filled with zeros

np.zeros(10, dtype=int)

# Create a 3x5 floating-point array filled with 1s

np.ones((3, 5), dtype=float)

Create a 3x5 array filled with 3.14

np.full((3, 5), 3.14)

# Create an array filled with a linear sequence

# Starting at 0, ending at 20, stepping by 2

# (this is similar to the built-in range() function)

np.arange(0, 20, 2)

# Create an array of five values evenly spaced between 0 and 1

np.linspace(0, 1, 5)

Create a 3x3 array of uniformly distributed

# random values between 0 and 1

np.random.random((3, 3))

# Create a 3x3 array of normally distributed random values

# with mean 0 and standard deviation 1

np.random.normal(0, 1, (3, 3))

# Create a 3x3 array of random integers in the interval [0, 10)

np.random.randint(0, 10, (3, 3))

# Create a 3x3 identity matrix

np.eye(3)

#Create an uninitialized array of three integers

# The values will be whatever happens to already exist at that

# memory location

np.empty(3)

#NumPy arrays contain values of a single type, so it is important to have knowledge of those types and their limitations.

np.zeros(10, dtype='int16')

####################

#The Basics of NumPy Arrays#

import numpy as np

NumPy Array Attributes

np.random.seed(0) # seed for reproducibility

x1 = np.random.randint(10, size=6) # One-dimensional array

x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array

x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array

print("x3 ndim: ", x3.ndim) #ndim (the number of dimensions)

print("x3 shape:", x3.shape) #shape (the size of each dimension)

print("x3 size: ", x3.size) #size (the total size of the array)

print("dtype:", x3.dtype) #data type

#Array Indexing: Accessing Single Elements

x1

x1[2]

x1[-2]

x2

x2[0, 1]#multidimensional array, you access items using a comma-separated tuple ofindices

x2[2, -1]

x2[0, 0] = -3 #You can also modify values using any of the above index notation:

#Array Slicing: Accessing Subarrays

#The NumPy slicing syntax

#x[start:stop:step]

#default values start=0, stop=size of dimension, step=1.

#One-dimensional subarrays

x = np.arange(10)

x

x[:5] # first five elements

x[3:] # elements after index 3

x[4:7] # middle subarray

x[::2] # every other element

x[::-1] # all elements, reversed

#defaults for start and stop are swapped.

#Multidimensional subarrays

#multiple slices separated by commas

x2

x2[:2, :3] # two rows, three columns

x2[:3, ::2] # all rows, every other column

x2[::-1, ::-1] #subarray dimensions can even be reversed together

x2[:, 0] # first column of x2

#combining indexing and slicing

#Subarrays as no-copy views

print(x2)

x2\_sub = x2[:2, :2]

print(x2\_sub)

x2\_sub[0, 0] = 99

print(x2\_sub)

print(x2)

#Creating copies of arrays

x2\_sub\_copy = x2[:2, :2].copy()

print(x2\_sub\_copy)

x2\_sub\_copy[0, 0] = 42

print(x2\_sub\_copy)

print(x2)

##Reshaping of Arrays

grid = np.arange(1, 10).reshape((3, 3))

print(grid)

x = np.array([1, 2, 3])

# row vector via reshape

x.reshape((1, 3)) #conversion of a one-dimensional array

#into a two-dimensional row or column matrix

#Array Concatenation and Splitting

#Concatenation of arrays

x = np.array([1, 2, 3])

y = np.array([3, 2, 1])

np.concatenate([x, y])

z = [99, 99, 99]

print(np.concatenate([x, y, z]))

grid = np.array([[1, 2, 3],

[4, 5, 6]])

np.concatenate([grid, grid])# concatenate along the first axis

np.concatenate([grid, grid], axis=1) ## concatenate along the second axis

x = np.array([1, 2, 3])

grid = np.array([[9, 8, 7],

[6, 5, 4]])

np.vstack([x, grid])# vertically stack the arrays

y = np.array([[99],

[99]])

np.hstack([grid, y])# horizontally stack the arrays

#Splitting of arrays

#np.split, np.hsplit, and np.vsplit

x = [1, 2, 3, 99, 99, 3, 2, 1]

x1, x2, x3 = np.split(x, [3, 5])

print(x1, x2, x3)

grid = np.arange(16).reshape((4, 4))

grid

upper, lower = np.vsplit(grid, [2])

print(upper)

print(lower)

##Computation on NumPy Arrays: Universal Functions

#Numpy provides an easy and flexible interface to optimized

#computation with arrays of data.

#Computation on NumPy arrays can be very fast, or it can be very slow.

#The key to making it fast is to use vectorized operations,

#generally implemented through NumPy’s universal functions (ufuncs).

#Array arithmetic

x = np.arange(4)

print("x =", x)

print("x + 5 =", x + 5)

print("x - 5 =", x - 5)

print("x \* 2 =", x \* 2)

print("x / 2 =", x / 2)

print("x // 2 =", x // 2) # floor division

print("-x = ", -x)

print("x \*\* 2 = ", x \*\* 2)#exponentiation

print("x % 2 = ", x % 2) #modulus

-(0.5\*x + 1) \*\* 2

#simply convenient wrappers around specific functions built into NumPy

#+ operator is a wrapper for np.add(x, 2)

x = [1, 2, 3]

print("x =", x)

print("e^x =", np.exp(x))

print("2^x =", np.exp2(x))

print("3^x =", np.power(3, x))

theta = np.linspace(0, np.pi, 3)

print("theta = ", theta)

print("sin(theta) = ", np.sin(theta))

#Aggregates

#A reduce repeatedly applies a given operation to the elements

#of an array until only a single result remains.

x = np.arange(1, 6)

np.add.reduce(x)

np.multiply.reduce(x)

#Outer products

#any ufunc can compute the output of all pairs of two different

# inputs using

x = np.arange(1, 6)

np.multiply.outer(x, x)

#Aggregations: Min, Max, and Everything in Between

#when faced with a large amount of data, a first step is

#to compute summary

L = np.random.random(100)

sum(L)

np.sum(L)

big\_array = np.random.rand(1000000)

%timeit sum(big\_array)

%timeit np.sum(big\_array)

np.min(big\_array), np.max(big\_array)

#other NumPy aggregates, a shorter syntax is to use

#methods of the array object itself

big\_array.min()

big\_array.max()

big\_array.sum()

M = np.random.random((3, 4))

M.sum()

M.min(axis=0)

M.max(axis=1)

#The keyword specifies the dimension of the array

#that will be collapsed,axis rather than the dimension that will be returned.

#Computation on Arrays: Broadcasting #

a = np.array([0, 1, 2])

b = np.array([5, 5, 5])

a + b

a + 5

M = np.ones((3, 3))

M + a #We can similarly extend this to arrays of higher dimension

#more complicated cases can involve broadcasting of both arrays

a = np.arange(3)

b = np.arange(3)[:, np.newaxis]

print(a)

print(b)

a + b

#Broadcasting Examples

##Ex 01

M = np.ones((2, 3))

a = np.arange(3)

'''

M.shape = (2, 3)

a.shape = (3,)

M.shape -> (2, 3)

a.shape -> (1, 3)

By Rule 01

M.shape -> (2, 3)

a.shape -> (2, 3)

By Rule 02

'''

M + a

#Ex 02

a = np.arange(3).reshape((3, 1))

b = np.arange(3)

a.shape

b.shape

a + b

#Ex 03

M = np.ones((3, 2))

a = np.arange(3)

'''

M.shape = (3, 2)

a.shape = (3,)

By Rule 01

M.shape -> (3, 2)

a.shape -> (1, 3)

By Rule 02

M.shape -> (3, 2)

a.shape -> (3, 3)

Now we hit rule 3—the final shapes do not match

'''

M + a

## Pandas ##

import pandas

pandas.\_\_version\_\_

import numpy as np

import pandas as pd

#Pandas Series is a one-dimensional array of indexed data

data = pd.Series([0.25, 0.5, 0.75, 1.0])

data

data.values

data.index

data[1]

data[0:2]

#explicit index definition gives the Series object additional capabilities

data = pd.Series([0.25, 0.5, 0.75, 1.0],

index=['a', 'b', 'c', 'd'])

data['b']

#even use noncontiguous or nonsequential indices

data = pd.Series([0.25, 0.5, 0.75, 1.0],

index=[2, 5, 3, 7])

data

#Series as specialized dictionary

population\_dict = {'California': 38332521,

'Texas': 26448193,

'New York': 19651127,

'Florida': 19552860,

'Illinois': 12882135}

population = pd.Series(population\_dict)

population

population['California':'Illinois']

#The Pandas DataFrame Object

#DataFrame as a generalized NumPy array

area\_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,

'Florida': 170312, 'Illinois': 149995}

area = pd.Series(area\_dict)

area

population\_dict = {'California': 38332521,

'Texas': 26448193,

'New York': 19651127,

'Florida': 19552860,

'Illinois': 12882135}

population = pd.Series(population\_dict)

population

#we can use a dictionary to construct a single two-dimensional

#object containing this information

states = pd.DataFrame({'population': population,

'area': area})

states

states.index

states.columns

############### Amver 6#############

#####################################

'''

Created by Dr. Lalit K Jiwani

Created on 2019/09/21

# Introduction to Python

'''

##Numpy ##

import numpy

numpy.\_\_version\_\_

import numpy as np

#to display all the contents of the numpy namespace

np.<TAB>

#to display NumPy’s built-in documentation

np?

#Creating more list

L = list(range(10))

L2 = [str(c) for c in L]

L3 = [True, "2", 3.0, 4]

#Fixed-Type Arrays in Python

import array

L = list(range(10))

A = array.array('i', L)

#Python’s array object provides efficient storage of array-based data,

# NumPy adds to this efficient operations on that data.

import numpy as np

#To create arrays from Python lists

np.array([1, 4, 2, 5, 3]) #integer array

#If types do not match, NumPy will upcast

np.array([3.14, 4, 2, 3])

#explicit datatype

np.array([1, 2, 3, 4], dtype='float32')

#NumPy arrays can explicitly be multidimensional

np.array([range(i, i + 3) for i in [2, 4, 6]])

#nested lists result in multidimensional arrays

## Creating Arrays from Scratch ##

#Create arrays from scratch using routines built into NumPy.

# Create a length-10 integer array filled with zeros

np.zeros(10, dtype=int)

# Create a 3x5 floating-point array filled with 1s

np.ones((3, 5), dtype=float)

Create a 3x5 array filled with 3.14

np.full((3, 5), 3.14)

# Create an array filled with a linear sequence

# Starting at 0, ending at 20, stepping by 2

# (this is similar to the built-in range() function)

np.arange(0, 20, 2)

# Create an array of five values evenly spaced between 0 and 1

np.linspace(0, 1, 5)

Create a 3x3 array of uniformly distributed

# random values between 0 and 1

np.random.random((3, 3))

# Create a 3x3 array of normally distributed random values

# with mean 0 and standard deviation 1

np.random.normal(0, 1, (3, 3))

# Create a 3x3 array of random integers in the interval [0, 10)

np.random.randint(0, 10, (3, 3))

# Create a 3x3 identity matrix

np.eye(3)

#Create an uninitialized array of three integers

# The values will be whatever happens to already exist at that

# memory location

np.empty(3)

#NumPy arrays contain values of a single type, so it is important to have knowledge of those types and their limitations.

np.zeros(10, dtype='int16')

####################

#The Basics of NumPy Arrays#

import numpy as np

NumPy Array Attributes

np.random.seed(0) # seed for reproducibility

x1 = np.random.randint(10, size=6) # One-dimensional array

x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array

x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array

print("x3 ndim: ", x3.ndim) #ndim (the number of dimensions)

print("x3 shape:", x3.shape) #shape (the size of each dimension)

print("x3 size: ", x3.size) #size (the total size of the array)

print("dtype:", x3.dtype) #data type

#Array Indexing: Accessing Single Elements

x1

x1[2]

x1[-2]

x2

x2[0, 1]#multidimensional array, you access items using a comma-separated tuple ofindices

x2[2, -1]

x2[0, 0] = -3 #You can also modify values using any of the above index notation:

#Array Slicing: Accessing Subarrays

#The NumPy slicing syntax

#x[start:stop:step]

#default values start=0, stop=size of dimension, step=1.

#One-dimensional subarrays

x = np.arange(10)

x

x[:5] # first five elements

x[3:] # elements after index 3

x[4:7] # middle subarray

x[::2] # every other element

x[::-1] # all elements, reversed

#defaults for start and stop are swapped.

#Multidimensional subarrays

#multiple slices separated by commas

x2

x2[:2, :3] # two rows, three columns

x2[:3, ::2] # all rows, every other column

x2[::-1, ::-1] #subarray dimensions can even be reversed together

x2[:, 0] # first column of x2

#combining indexing and slicing

#Subarrays as no-copy views

print(x2)

x2\_sub = x2[:2, :2]

print(x2\_sub)

x2\_sub[0, 0] = 99

print(x2\_sub)

print(x2)

#Creating copies of arrays

x2\_sub\_copy = x2[:2, :2].copy()

print(x2\_sub\_copy)

x2\_sub\_copy[0, 0] = 42

print(x2\_sub\_copy)

print(x2)

##Reshaping of Arrays

grid = np.arange(1, 10).reshape((3, 3))

print(grid)

x = np.array([1, 2, 3])

# row vector via reshape

x.reshape((1, 3)) #conversion of a one-dimensional array

#into a two-dimensional row or column matrix

#Array Concatenation and Splitting

#Concatenation of arrays

x = np.array([1, 2, 3])

y = np.array([3, 2, 1])

np.concatenate([x, y])

z = [99, 99, 99]

print(np.concatenate([x, y, z]))

grid = np.array([[1, 2, 3],

[4, 5, 6]])

np.concatenate([grid, grid])# concatenate along the first axis

np.concatenate([grid, grid], axis=1) ## concatenate along the second axis

x = np.array([1, 2, 3])

grid = np.array([[9, 8, 7],

[6, 5, 4]])

np.vstack([x, grid])# vertically stack the arrays

y = np.array([[99],

[99]])

np.hstack([grid, y])# horizontally stack the arrays

#Splitting of arrays

#np.split, np.hsplit, and np.vsplit

x = [1, 2, 3, 99, 99, 3, 2, 1]

x1, x2, x3 = np.split(x, [3, 5])

print(x1, x2, x3)

grid = np.arange(16).reshape((4, 4))

grid

upper, lower = np.vsplit(grid, [2])

print(upper)

print(lower)

##Computation on NumPy Arrays: Universal Functions

#Numpy provides an easy and flexible interface to optimized

#computation with arrays of data.

#Computation on NumPy arrays can be very fast, or it can be very slow.

#The key to making it fast is to use vectorized operations,

#generally implemented through NumPy’s universal functions (ufuncs).

#Array arithmetic

x = np.arange(4)

print("x =", x)

print("x + 5 =", x + 5)

print("x - 5 =", x - 5)

print("x \* 2 =", x \* 2)

print("x / 2 =", x / 2)

print("x // 2 =", x // 2) # floor division

print("-x = ", -x)

print("x \*\* 2 = ", x \*\* 2)#exponentiation

print("x % 2 = ", x % 2) #modulus

-(0.5\*x + 1) \*\* 2

#simply convenient wrappers around specific functions built into NumPy

#+ operator is a wrapper for np.add(x, 2)

x = [1, 2, 3]

print("x =", x)

print("e^x =", np.exp(x))

print("2^x =", np.exp2(x))

print("3^x =", np.power(3, x))

theta = np.linspace(0, np.pi, 3)

print("theta = ", theta)

print("sin(theta) = ", np.sin(theta))

#Aggregates

#A reduce repeatedly applies a given operation to the elements

#of an array until only a single result remains.

x = np.arange(1, 6)

np.add.reduce(x)

np.multiply.reduce(x)

#Outer products

#any ufunc can compute the output of all pairs of two different

# inputs using

x = np.arange(1, 6)

np.multiply.outer(x, x)

#Aggregations: Min, Max, and Everything in Between

#when faced with a large amount of data, a first step is

#to compute summary

L = np.random.random(100)

sum(L)

np.sum(L)

big\_array = np.random.rand(1000000)

%timeit sum(big\_array)

%timeit np.sum(big\_array)

np.min(big\_array), np.max(big\_array)

#other NumPy aggregates, a shorter syntax is to use

#methods of the array object itself

big\_array.min()

big\_array.max()

big\_array.sum()

M = np.random.random((3, 4))

M.sum()

M.min(axis=0)

M.max(axis=1)

#The keyword specifies the dimension of the array

#that will be collapsed,axis rather than the dimension that will be returned.

#Computation on Arrays: Broadcasting #

a = np.array([0, 1, 2])

b = np.array([5, 5, 5])

a + b

a + 5

M = np.ones((3, 3))

M + a #We can similarly extend this to arrays of higher dimension

#more complicated cases can involve broadcasting of both arrays

a = np.arange(3)

b = np.arange(3)[:, np.newaxis]

print(a)

print(b)

a + b

#Broadcasting Examples

##Ex 01

M = np.ones((2, 3))

a = np.arange(3)

'''

M.shape = (2, 3)

a.shape = (3,)

M.shape -> (2, 3)

a.shape -> (1, 3)

By Rule 01

M.shape -> (2, 3)

a.shape -> (2, 3)

By Rule 02

'''

M + a

#Ex 02

a = np.arange(3).reshape((3, 1))

b = np.arange(3)

a.shape

b.shape

a + b

#Ex 03

M = np.ones((3, 2))

a = np.arange(3)

'''

M.shape = (3, 2)

a.shape = (3,)

By Rule 01

M.shape -> (3, 2)

a.shape -> (1, 3)

By Rule 02

M.shape -> (3, 2)

a.shape -> (3, 3)

Now we hit rule 3—the final shapes do not match

'''

M + a

## Pandas ##

import pandas

pandas.\_\_version\_\_

import numpy as np

import pandas as pd

#Pandas Series is a one-dimensional array of indexed data

data = pd.Series([0.25, 0.5, 0.75, 1.0])

data

data.values

data.index

data[1]

data[0:2]

#explicit index definition gives the Series object additional capabilities

data = pd.Series([0.25, 0.5, 0.75, 1.0],

index=['a', 'b', 'c', 'd'])

data['b']

#even use noncontiguous or nonsequential indices

data = pd.Series([0.25, 0.5, 0.75, 1.0],

index=[2, 5, 3, 7])

data

#Series as specialized dictionary

population\_dict = {'California': 38332521,

'Texas': 26448193,

'New York': 19651127,

'Florida': 19552860,

'Illinois': 12882135}

population = pd.Series(population\_dict)

population

population['California':'Illinois']

#The Pandas DataFrame Object

#DataFrame as a generalized NumPy array

area\_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,

'Florida': 170312, 'Illinois': 149995}

area = pd.Series(area\_dict)

area

population\_dict = {'California': 38332521,

'Texas': 26448193,

'New York': 19651127,

'Florida': 19552860,

'Illinois': 12882135}

population = pd.Series(population\_dict)

population

#we can use a dictionary to construct a single two-dimensional

#object containing this information

states = pd.DataFrame({'population': population,

'area': area})

states

states.index

states.columns