Read from file

In [ ]:
```
with open('book.txt','r') as fhand:
    file_contents = fhand.read()
```

In [ ]:
```
import csv
with open('filename') as fhand:
    reader = csv.reader(fhand)
    # or,
    reader = csv.DictReader(fhang)
```

Write to file

In [ ]:
```
with open('filename', 'w') as outf:
    outf.write(message)
```

Enumerate

In [15]:
```
mylist = ["eat","sleep","repeat"]

obj = enumerate(mylist)    # creating enumerate objects

for index, item in obj:
    print(index, item)
```

```
0 eat
1 sleep
2 repeat
```

Zip

In [32]:
```
numberList = [1, 2, 3, 4]
strList = ['one', 'two', 'three']

# Two iterables are passed
result = zip(numberList, strList)
print(list(result))
```

```
[(1, 'one'), (2, 'two'), (3, 'three')]
```

Map

In [37]:
```
def myfunc(n):
    return len(n)

x = map(myfunc, ('apple', 'banana', 'cherry'))
print(list(x))
```

```
[5, 6, 6]
```

Named Tuple

In [49]:
```
# Create a Car class of namedtuple
from collections import namedtuple
Car = namedtuple('CarTuple', ['color', 'mileage'])
```

In [50]:
```
my_car = Car('red', 1000)
```

In [51]:
```
print(my_car)
print(my_car.color)
print(my_car[0])
```

```
CarTuple(color='red', mileage=1000)
red
red
```

We can also sort list of named tuple by the named attribute

In [ ]:
```
sorted_list = sorted(the_list, reverse=True, key=lambda item: item.color
)
```

Numpy (fancy index masking)

In [56]:
```
a = [1,2,3,4,5,6]
a = np.array(a)
b = a > 3
b
```

Out[56]:
```
array([False, False, False,  True,  True,  True])
```

In [57]:
```
c = a[b] * 2
c
```

Out[57]:
```
array([ 8, 10, 12])
```

Libraries

In [54]:
```
import numpy as np
import pandas as pd
from bs4 import BeautifulSoup
import requests
import csv
import re
from collections import namedtuple

import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import rcParams
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import KNeighborsRegressor
```

Inheritance

In [6]:
```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self):
        return self.firstname + " " + self.lastname
```

In [9]:
```
class Employee(Person):

    def __init__(self, first, last, staffnum):
        super().__init__(first, last)
        #Person.__init__(self, first, last)  #Alternative
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.Name() + ", " +  self.staffnumber
```

In [10]:
```
x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")

print(x.Name())
print(y.GetEmployee())
```

```
Marge Simpson
Homer Simpson, 1007
```

Overriding

In [11]:
```
class Person:

    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age

    def __str__(self):
        return self.firstname + " " + self.lastname + ", " + str(self.ag
e)
```

In [12]:
```
class Employee(Person):

    def __init__(self, first, last, age, staffnum): #Overriding construc
tor
        super().__init__(first, last, age)
        self.staffnumber = staffnum

    def __str__(self):  #Overriding parent function
        return super().__str__() + ", " +  self.staffnumber
```

In [13]:
```
x = Person("Marge", "Simpson", 36)
y = Employee("Homer", "Simpson", 28, "1007")

print(x)
print(y)
```

```
Marge Simpson, 36
Homer Simpson, 28, 1007
```

Default argument

In [69]:
```
def f(n, m=0): # m has default value of 0
    if m:
        return n + m +42
    else:
        return n + 42

# If argument has default value, it becomes optional.
# Arguments with default values must be placed after the ones without.
f(1)
f(1,2)
```

Out[69]:
```
45
```

Optional argument using *

In [70]:
```
def f(n, *m): # n is a required argument, m is optional and can be multi
ple
    print ('function called:', n)
    for item in m:
        print(item)
f(1)
f(1,2,3)
```

```
function called: 1
function called: 1
2
3
```

Keyworded optional argument using **

In [71]:
```
def z(**x):  # argument names and values are passed as tuples in a list
    for key, value in x.items():  # unpacking the tuple of (key, value)
        print(key, value)
    print('End of function.')
z()
z(name = 'Oscar')
z(name = 'Savith', ID = 5, iq = 1000)
# the argument is optional,
# but cannot be a value without keyword, ie: z(5) crashes
```

```
End of function.
name Oscar
End of function.
name Savith
ID 5
iq 1000
End of function.
```

Combining arguments

In [80]:
```
# Regular, optional, and keyworded optional arguments
# must be in the following order:
def y(a=0, *b, **c):
    print('regular:', a)
    for i in b:
        print('args:', i)
    for i in c:
        print ('kwargs:', i)

y()
y(1,2,3)
y(1, 2, name = 'Santa')
y(1, name = 'Santa')
```

```
regular: 0
regular: 1
args: 2
args: 3
regular: 1
args: 2
kwargs: name
regular: 1
kwargs: name
```

In [86]:
```
args = (1,2,3)
# Note the difference in how the argument is passed.
y(args)
y(*args)
```

```
regular: (1, 2, 3)
regular: 1
args: 2
args: 3
```

In [1]:
```
import pandas as pd
```

Creating DataFrame

In [60]:
```
df = pd.DataFrame({'id': [100,101,102], 'color':['red','blue','red']}, c
olumns=['id', 'color'], index = ['a','b','c'])
df = pd.DataFrame([[100,'red'],[101, 'blue'], [102, 'red']], columns=['i
d', 'color'], index=['a','b','c'])
```

In [ ]:
```
# Creating from np array
import numpy as np
arr = np.random.rand(4,2)
df = pd.DataFrame(arr, columns = ['one', 'two'], index = ['a','b','c',
'd'])
```

In [61]:
```
# Add a Series to DataFrame with index alignment:
ser = pd.Series(['round', 'square'], index=['c', 'b'], name='shape' )
pd.concat([df, ser], axis = 1, sort=False)
```

Out[61]:

	id	color	shape
a	100	red	NaN
b	101	blue	square
c	102	red	round

Reading from file

In [ ]:
```
df= pd.read_table('orders.tsv')
#Read selected columns:
df = pd.read_table('orders.tsv', names= ['item_name', 'item_price'])
df = pd.read_table('imdb_1000.csv', sep='|', header=None)
df = pd.read_csv('imdb_1000.csv')
#read first number of columns
df = pd.read_table('orders.tsv', nrows = 3)
```

Slicing, Joining Series from DataFrame

In [ ]:
```
df['column_name']   # select using bracket notation
df.column_name      # select using dot notation
```

In [ ]:
```
# Joining columns and assign to a new column
df['new_column'] = df.col_1 + ", " + df.col_2
# Assigning new column must be done in bracket notation
```

Common Attribute & Methods for DataFrame

In [ ]:
```
df.head()
df.tail()
df.shape
df.describe()
df.dtypes
df.columns
```

Rename Columns

In [ ]:
```
# Several ways to achieve this:
df.rename(columns = {'old_col_1': 'new_col_1', 'old_col_2': 'new_col'})
df.rename({'old_col_1': 'new_col_1', 'old_col_2': 'new_col'}, axis = 0)
df.columns = ['new_col_1', 'new_col_2']

# And can be done when reading from file:
df = pd.read_csv('filename.csv', names = ['col1', 'col2', 'col3'], heade
r = 0)

# Using str method to modify column names:
df.columns = df.columns.str.replace(' ', '_')
```

Remove Columns, Rows

In [ ]:
```
df.drop('col_1', axis = 1, inplace = True)     # drop column
df.drop(5, axis = 0, inplace = True) # drop row
df.drop(['row_1', 'row_2']) # Drop row by name (if available)
```

In [ ]:
```
# Drop multiple columns
df.drop(['col_1', 'col_2'], axis = 1, inplace = True)
df.drop(columns = ['col_1','col_2'], inplace = True)
```

In [ ]:
```
# Drop multiple rows
df.drop(df.index[0:5], inplace = True)
```

Remove duplicate rows

In [ ]:
```
# Creates a boolean mask of duplicated values in a column:
df.col_1.duplicated()
# Creates a boolean mask of duplicated rows:
df.duplicated()
# the first duplicated item is marked as false:
df.duplicated(keep = 'first')
```

In [ ]:
```
# Removing duplicates, keeping only the last of the duplicated occurances:
df.drop_duplicates(keep = 'last', inplace = True)
# Looking at a subset of columns to identify duplicates
df.drop_duplicates(subset = ['col_1', 'col_2'], inplace = True)
```

Drop non-numeric columns from DataFrame

In [ ]:
```
df.select_dtypes(include = [np.number])
```

Handling missing values

In [ ]:
```
# Drop an entire row if any of the columns has a nan value:
df.dropna(how = 'any')
# Drop an entire row only if all of the columns are nan values:
df.dropna(how = 'all')
# Drop an entire row only if specified columns has nan values:
df.dropna(subset = ['col_1', 'col_2'], how = 'any')
```

In [ ]:
```
# Fill missing values:
df['col_1'].fillna(value = 'Other', inplace = True)
```

Overriding values with nan

In [ ]:
```
import numpy as np
df.loc[df['col_1']=='N/A', 'col_1'] = np.nan
# Note: loc method is used to re-assign values back to the same column
```

Change data type

In [ ]:
```
# Change data type in one column of DataFrame:
df['col_4'] = df.col_1.astype(float)
# Change data type when reading file into DataFrame:
df = pd.csv_read('some_file', dtype = {'col_1': float})
# Change data type of multiple columns at once:
df = df.astype({'col_1': 'float', 'col_2': 'float'})
```

In [ ]:
```
# Working with currency:
df.col_4.str.replace('$','').astype(float)
```

In [ ]:
```
# Changing contextual data to 1 & 0 for computing purpose:
df.col_2.str.contains('some_word').astype(int)
```

DataFrame optimization

In [ ]:
```
# To see memory usage
df.info(memory_usage = 'deep')  # Overall usage
df.memory_usage(deep = True)    # By column
```

Storing a column of string value type as category

In [ ]:
```
df['col_1'] = df.col_1.astype('category')
#or, when reading the file:
df = pd.read_csv('filename.csv', dtype={'col_1': 'category'})
```

Logical order of string type

In [7]:
```
# Given a DataFrame with string type in 'Quality' column:
df = pd.DataFrame({'ID': [100,101,102,103], 'Quality':['good', 'very goo
d', 'good', 'excellent']})
```

In [9]:
```
# Specify the order of string values in the 'Quality' column
from pandas.api.types import CategoricalDtype
quality_cat = CategoricalDtype(['good', 'very good', 'excellent'], order
ed = True)
df['Quality'] = df.Quality.astype(quality_cat)
```

In [10]:
```
# As the result, sorting and filtering can be done on this column:
display (df.sort_values(by='Quality'), df.loc[df.Quality>'good', :])
```

	ID	Quality
0	100	good
2	102	good
1	101	very good
3	103	excellent

	ID	Quality
1	101	very good
3	103	excellent

Value mapping / dummy variables

In [ ]:
```
# From a column that has values either 'male' or 'female',
# create a new column that uses 1, 0 corresponding to 'male' and 'femal
e'
df['sex_male'] = df['sex'].map({'female': 0, 'male': 1})
```

In [ ]:
```
# Alternatively:
# Creates a table of columns, each column represent a value from the ori
ginal column:
pd.get_dummies(df.sex)
```

Apply a function to Series or DataFrame

In [ ]:
```
# Apply len function to values in a column and store results in a new co
lumn:
df['lengths'] = df.col_1.apply(len)
#Apply Numpy ceiling function:
df['ceilings'] = df.col_1.apply(np.ceil)
```

In [ ]:
```
# Get last name from a column of names 'lastname, firstnames':
df['lastname'] = df['names'].str.split(',').apply(lambda x: x[0])
```

In [ ]:
```
# Get max value for each column from specified columns:
df.loc[:, 'col_3' : 'col_5'].apply(max, axis = 0)
# Locate the index of the max values:
df.loc[:, 'col_3' : 'col_5'].apply(np.argmax, axis=0)
```

agg functions

In [ ]:
```
# agg functions can be apply to Series and DataFrame
df.col_1.agg(['mean', 'max', 'min'])
df.agg(['mean', 'max', 'min'])
```

Apply a function to each element

In [ ]:
```
# Turn values to float type:
df.loc[:, 'col_3': 'col_5'].applymap(float)
```

Sorting

Sort values in a column as a Series

In [ ]:
```
df['col_1'].sort_values()
```

Sort whole DataFrame by values in one column

In [ ]:
```
df.sort_values('col_1')
```

Sort whole DataFrame by multiple columns

In [ ]:
```
df.sort_values(['col_1', 'col_2'])
```

Filtering

Filter DataFrame by value in one column

In [ ]:
```
df[df.col_1 > 200]          # Dot notation
df[df['col_1'] > 200]       # Bracket notation
df[df.col_2.str.contains('some_word')]  # Using str method
```

Selecting a column after applying the filter

In [ ]:
```
df[df.col_1 >= 200]['col_2']          # Using bracket notation
df[df.col_1 >= 200].col_2             # Using dot notation
df.loc[df.col_1 >= 200, 'col_2']      # Best practice: using .loc method
```

Multiple filters

In [ ]:
```
df[(df.col_1 == 200) & (df.col_2 < 100)]    # Must use '&' as the 'and'
 operator
df[(df.col_1 > 200) | (df.col_2 < 100)]    # Must use '|' as the 'or' op
erator
df[df.col_3.isin(['value1', 'value2'])]     # Multiple conditions in one
 column
```

Date time in Pandas

In [ ]:
```
# Convert to Pandas datetime64 type
df['date_time'] = pd.to_datetime(df['date_time'])
```

In [ ]:
```
# Now the column can be used for comparison and filtering:
ts = pd.to_datetime('1/1/1999')
df.loc[df.date_time >= ts, :]
```

In [ ]:
```
(ufo.date_time.max() - ufo.date_time.min()).days # difference in days
```

Creating datetime from columns with specific names that Pandas recognizes

In [53]:
```
# Given:
df = pd.DataFrame([[12,25,2017,10], [1,15,2018,11]], columns=['month',
'day', 'year','hour'])
df.dtypes
```

Out[53]:
```
month    int64
day      int64
year     int64
hour     int64
dtype: object
```

In [54]:
```
df['date_time'] = pd.to_datetime(df)   # Converts to datetime64 type
```

In [55]:
```
# Alternatively, convert and use the date column as index:
df.index = pd.to_datetime(df[['month', 'day', 'year']])
```

Indexing

In [ ]:
```
df.loc[5, 'col_2']            # Select specific row, column
df.loc[0, :]                  # Select the first row
df.loc[0:2, :]                # Select first 3 rows
df.loc[:, 'col_1']            # Select a column by name
df[['col_1', 'col_3']]        # Select multiple columns
df.loc[:, ['col_1', 'col_3']] # Select multiple columns
df.loc[:, 'col_1' : 'col_3']  # Select a range of columns
```

In [ ]:
```
# Set a column as index(row), and select data using row and column name
s:
df.set_index('col_1', inplace = True)
df.loc['a_value_in_col_1', 'col_2']
# To put the index back into a column:
df.index.name = 'col_1'
df.reset_index(inplace = True)
```

In [ ]:
```
# Explicitly creating a copy instead of a view:
new_df = df.loc[0, :].copy
```

Series with matching index can be multiplied, and result will be properly aligned

In [5]:
```
population = pd.Series([3000, 5000], index = ['AB', 'BC'], name = 'popul
ation')
income_per_cap = pd.Series([500,500,500], index = ['AB', 'BC', 'SK'], na
me = 'income_per_cap')
total_income = population * income_per_cap
total_income
```

Out[5]:
```
AB    1500000.0
BC    2500000.0
SK          NaN
dtype: float64
```

Merge/ Concat DataFrame

In [ ]:
```
# Series and DataFrame can be combined with index alignment
pd.concat([df_1, ser_1], axis = 1)
```

In [ ]:
```
# Merging by a common index column
pd.merge(df_1, df_2, left_index = True, right_index = True)
# parameters: how = 'inner' (default) retains only rows in both sets
#             how = 'left'/'right' use left/right DataFrame index
#             how = 'outer' retain all rows from either DataFrames
```

Append

In [ ]:
```
# For generating data row by row and appending to the bottom
df = df.append({'col_1': val_1, 'col_2': val_2, 'col_3': val_3}, ignore_
index=True)
```

Splitting DataFrame (opposite selection using ~)

In [ ]:
```
df_train = df.sample(frac = 0.75, random_state = 99)
df_test = df.loc[~df.index.isin(df_train.index), :] # Select opposite of
 df_train
```

Iterating through DataFrame

In [ ]:
```
for index, row in df.iterrows():
    print(index, row.col_1, row.col_2)
```

groupby method

Categorizing data in one column, and look at values from another columns summmarized by some function:

In [ ]:
```
# Group data by col_3, and calcualte mean in col_2:
df.groupby('col_3').col_2.mean()
```

In [ ]:
```
# Specifying aggregation functions:
df.groupby('col_3').col_2.agg(['count', 'min', 'max', 'mean'])
```

In [ ]:
```
# Group data by multiple columns creates a multi-index table:
df.groupby(['col_1', 'col_3']).col_2.mean()
```

Value occurances in a column

In [ ]:
```
# This gives a list of unique values in a column:
df.col_1.unique()
```

In [ ]:
```
# This gives a list of unique values and their occurances:
df.col_1.value_counts()
```

In [ ]:
```
# This gives a list of unique values and their frequency:
df.col_1.value_counts(normalize = True)
```

In [ ]:
```
# Combining low frequency values by overwriting value to 'other'
freq_series=pd.value_counts(df['col'])
mask_freq = freq_series/freq_series.sum() * 100 < 1    #mask frequency < 1%
df['col']=np.where(df['col'].isin(freq_series[mask_freq].index), 'other', df['col'])
```

crosstab method by default uses 'count' aggregation:

In [ ]:
```
# This creates a table with col_1 as row index, col_2 as column headers
pd.crosstab(df.col_1, df.col_2)
```

Multi-index Series

groupby method

In [67]:
```
stocks = pd.read_csv('stocks.csv')
# Multi-index series can be created by grouping by more than one index:
ser = stocks.groupby(['Symbol', 'Date']).Close.mean()
ser.index
```

Out[67]:
```
MultiIndex(levels=[['AAPL', 'CSCO', 'MSFT'], ['2016-10-03', '2016-10-04', '2016-10-05']],
           labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2]],
           names=['Symbol', 'Date'])
```

In [69]:
```
# Unstacking this dual-index series, would return a DataFrame
# with the second index being the columns
df = ser.unstack()
df
```

Out[69]:

Date	2016-10-03	2016-10-04	2016-10-05
Symbol			
AAPL	112.52	113.00	113.05
CSCO	31.50	31.35	31.59
MSFT	57.42	57.24	57.64

Alternatively, using pivot_table method

In [70]:
```
stocks.pivot_table(values='Close', index='Symbol', columns='Date')
```

Out[70]:

Date	2016-10-03	2016-10-04	2016-10-05
Symbol			
AAPL	112.52	113.00	113.05
CSCO	31.50	31.35	31.59
MSFT	57.42	57.24	57.64

Series with multi-index behaves like a 2-dimension DataFrame

In [ ]:
```
ser.loc['AAPL']
ser.loc['AAPL', '2016-10-03']
ser.loc[:, '2016-10-03']
```

Multi-index DataFrame

In [71]:
```
stocks.set_index(['Symbol','Date'], inplace=True)
stocks
```

Out[71]:

		Close	Volume
Symbol	**Date**		
CSCO	**2016-10-03**	31.50	14070500
AAPL	**2016-10-03**	112.52	21701800
MSFT	**2016-10-03**	57.42	19189500
AAPL	**2016-10-04**	113.00	29736800
MSFT	**2016-10-04**	57.24	20085900
CSCO	**2016-10-04**	31.35	18460400
MSFT	**2016-10-05**	57.64	16726400
CSCO	**2016-10-05**	31.59	11808600
AAPL	**2016-10-05**	113.05	21453100

Pass the indexes as a tuple using loc

In [ ]:
```
stocks.loc[('AAPL', '2016-10-03'), :]
stocks.loc[('AAPL', '2016-10-03'), 'Close']
stocks.loc[(['AAPL', 'MSFT'], '2016-10-03'), 'Close']
stocks.loc[(slice(None), '2016-10-03'), 'Volume']
# Note that when no slice is to be made on the first index, a special pr
operty syntax is used
```

In [ ]:
```
stocks.reset_index()    # resets back to the original shape
```

Notebook display options for Pandas

In [ ]:
```
pd.reset_option('all') #reset all options to default
```

In [59]:
```
pd.set_option('display.max_rows', None)
pd.reset_option('display.max_rows')
```

pd.set_option('display.max_colwidth', 1000) pd.reset_option('display.max_colwidth')

In [ ]:
```
pd.set_option('display.precision', 2)
pd.reset_option('display.precision')
```

In [ ]:
```
pd.describe_option('rows') #search in docs for methods containing 'row'
```

In [ ]:
```
from bs4 import BeautifulSoup
import requests
```

Open a file, or a web link

In [ ]:
```
# Opening a html file from directory
with open('file_name') as html_file:
    soup = BeautifulSoup(html_file, 'lxml')
```

In [ ]:
```
html_file = requests.get("http://www.timeslikethese.ca/journal/", verify
=False).text
soup = BeautifulSoup(html_file, 'lxml')
```

In [ ]:
```
print(soup.prettify()) # prettify() method formats indentation properly
 for viewing
```

Use dot notation to navigate down the hierarachy of the tags

In [ ]:
```
match = soup.title  # Returns the title tag <title></title>
match.text
```

In [ ]:
```
match = soup.div   # Returns the first div tag <div></div>, and all of i
ts children
```

Use find method to look for tag names with specific class names

In [ ]:
```
content = soup.find('div', class_='content-inner') # Find 'div' tags wit
h attribute of class = 'content-inner'
content = soup.find('div',  attrs={'class':'content-inner'}) # alternati
vely, pass attribute: value in dictionary
```

In [ ]:
```
article = content.section.article.header  # navigate down the tag hierac
hy using dot notation on the tag names
```

find_all method returns a list of tags matching the criteria

In [ ]:
```
articles = soup.find_all('div', class_='entry-title-wrapper')  # use fin
d_all to find a list of tag
articles[0]
```

In [ ]:
```
articles[0].a.text
```

In [ ]:
```
for i in articles:
    print (i.a.text)
```

Write data to csv file

In [ ]:
```
import csv
csv_file = open('filename.csv', 'w')
csv_writer = csv.writer(csv_file)
```

In [ ]:
```
for i in articles:
    csv_writer.writerow([i.a.text, i.time.text])
```

In [ ]:
```
csv_file.close()
```

Compose API url using the urllib.parse.urlencode method

In [ ]:
```
import urllib.parse
main_api = 'http://maps,googleapis.com/maps/api/geocode/json?'
address = 'lhr'
url = main_api + urllib.parse.urlencode({'address': address, 'id': '5'})
 # By passing a dictionary

# Alternatively, use request.get method to specify parameters:
r = requests.get(main_api, params={'address': address, 'id': 5})
```

Make API calls

In [ ]:
```
url = 'https://api.dailysmarty.com/posts'
r = requests.get(url)
```

In [ ]:
```
r .json()
```

In [ ]:
```
%pprint   #Note: turn on pprint to make json file more readable
```

Navigate the json using bracket notation

In [ ]:
```
r.json()['posts'][0]['title']
```

Classification

In [86]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
```

In [87]:
```
data = pd.read_csv('NasaData.csv')
```

In [88]:
```
# Set up feature set and label set
labels = data.loc[:, 'label']
features = data.loc[:, data.columns != 'label']
```

In [89]:
```
# Split data into training set and test set
# X_train: training set features; y_train: traing set labels.
# X_test: test set features; y_test: test set labels.
X_train, X_test, y_train, y_test = train_test_split(features, labels, te
st_size=0.25, random_state=99)
```

In [90]:
```
# Set up model
KNN = KNeighborsClassifier().fit(X_train, y_train)
```

In [91]:
```
# Compare test label and true label for score
KNN.score(X_test, y_test)
```

Out[91]: 0.762589928057554

Complexity Tuning

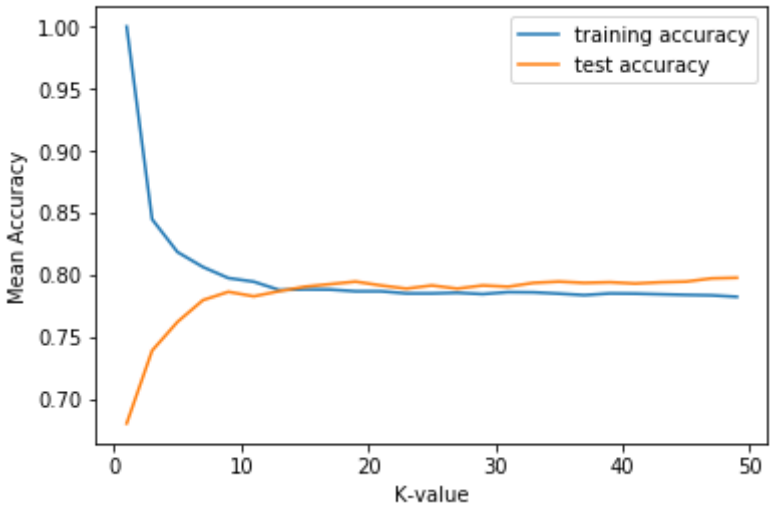
In [92]:
```
import matplotlib.pyplot as plt
%matplotlib inline
K_values = list(range(1,50,2))
```

The train accuracy: The accuracy of a model on examples it was constructed on (train set). The test accuracy: The accuracy of a model on examples it hasn't seen (test set).

In [93]:
```
train_accuracy = []
test_accuracy = []
```

In [94]:
```
for K in K_values:
    KNN = KNeighborsClassifier(n_neighbors = K).fit(X_train, y_train)
    train_accuracy.append(KNN.score(X_train, y_train))
    test_accuracy.append(KNN.score(X_test, y_test))
```

In [107]:
```
plt.plot(K_values, train_accuracy, label="training accuracy")
plt.plot(K_values, test_accuracy, label="test accuracy")
plt.ylabel('Mean Accuracy')
plt.xlabel('K-value')
plt.legend()
plt.show()
```



Regression

In [120]:
```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
```

In [121]:
```
data = pd.read_csv('CPU_Performance.csv')
```

In [122]:
```
labels = data.loc[:, 'ERP']
features = data.loc[:, data.columns != 'ERP']
```

In [123]:
```
# Split data into training set and test set
X_train, X_test, y_train, y_test = train_test_split(features, labels, te
st_size=0.25, random_state=42)
```

In [124]:
```
# Run model
kr = KNeighborsRegressor().fit(X_train, y_train)
```

In [125]:
```
# Training set score:
kr.score(X_train, y_train)
```

Out[125]: 0.8882859990121516

In [126]:
```
# Test set score:
kr.score(X_test, y_test)
```

Out[126]: 0.6067632293646072