Every Accomplishment Starts With a Decision to Try ...

BAG-OF-MATERIALS

Imports Section

In [1]:

```
import nltk
import random
import string
import pandas as pd
import csv
import re
import math
import numpy as np
import sys

from autocorrect import spell
from vocabulary.vocabulary import Vocabulary as vb
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from __future__ import division
from nltk.corpus import wordnet as wn
from nltk.corpus import brown
from pathlib import Path
```

Working directory path

In [2]:

```
path = Path.cwd()
path
```

Out[2]:

```
WindowsPath('E:/Workspace/BOM Workspace/Bag-of-Materials')
```

Class Initialisation for Error Handling

In [3]:

```
class Error(Exception):
    """Base class for other exceptions"""
    pass

class ValueTooSmallError(Error):
    pass

class ValueTooLargeError(Error):
    pass

class InvalidKeywordError(Error):
    pass

class FileNotExistError(Error):
    pass
```

Module 1: Query Engine

CLASS NAME : Normalise_query

DESCRIPTION : Accepts two inputs credits and items, checks for spelling of Items and credits within maximum and minimum range.

In [4]:

```
class Normalise_query:

#-----------------------------------------------------------------------------------------
# FUNCTION NAME  : buyer_credits
#-----------------------------------------------------------------------------------------
# PARAMETERS     : NIL
#-----------------------------------------------------------------------------------------
# RETURN         : Integer value assigned as Credits
#-----------------------------------------------------------------------------------------
# DESCRIPTION    : Random function generates an integer number taken as credits specifi
ed by buyer.
#                  Credits are checked for exceptional cases.
#-----------------------------------------------------------------------------------------

    def buyer_credits(self):
        self.min = 100
        self.max = 50000

        # Randome function generates ineteger number
        self.buyer_credit = random.randint(self.min,self.max)

        # Validating if credits in range, if not exceptions are raised
        while True:
            try:
                if self.buyer_credit < self.min:
                    raise ValueTooSmallError
                elif self.buyer_credit > self.max:
                    raise ValueTooLargeError
                break

            except ValueTooSmallError:
                print("buyer_credits = {}, is less than minimum credits.".format(Self.b
uyer_credit))
                self.buyer_credit = int(input("Re-enter Credits:"))

            except ValueTooLargeError:
                print("buyer_credits = {}, is more than maximum credits.".format(self.b
uyer_credits))
                self.buyer_credits = int(input("Re-enter Credits:"))

        # return credits hen found in range of minimum and maximum
        print("buyer_credits = {} in range.".format(self.buyer_credit))
        return self.buyer_credit

#-----------------------------------------------------------------------------------------
# FUNCTION NAME  : buyer_item
#-----------------------------------------------------------------------------------------
# PARAMETERS     : NIL
#-----------------------------------------------------------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------------
```

```
#---------------------------------------
# DESCRIPTION    : Opens buyer_items file and reads random items from the file based on
 random integer length.
#
#---------------------------------------------------------------------------------------
---------------------------------------

    def buyer_item(self):
        try:
            self.buyer_items = []

            # generates the length of number of inputs to be taken
            self.length = random.randint(1,5)
            print(self.length)
            for self.i in range(self.length):
                if path.joinpath("buyer_items.txt").is_file() == False:
                    raise FileNotExistError
                else:
                    # Retrieving items from buyer_items file
                    with open(path/"buyer_items.txt","r") as self.readfile:
                        self.items = self.readfile.readlines()
                        self.rand_word = random.choice(self.items).split()
                        self.buyer_items.append(' '.join(self.rand_word).lower())
                    self.readfile.close()

            print(self.buyer_items)

        except FileNotExistError:
            print("File Not Found!!!")
            sys.exit()

#---------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : spell_check
#---------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : buyer_items (list)
#---------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#---------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Checks for spellings of each item. Raises error if word meaning does
n't exist.
#
#---------------------------------------------------------------------------------------
---------------------------------------

    def spell_check(self):
        self.spell_list = []
        self.split_list = []
        for self.t in self.buyer_items:
            self.spell_list.extend(self.t.split('-'))
        try:
            for self.l in self.spell_list:
                self.split_list.extend(self.l.split())

            # checking for spellings of items given, error is raised if found input ite
m meaning is false
            for self.j in self.split_list:
                if vb.meaning(self.j) == False:
```

```
                    raise InvalidKeywordError
                else:
                    pass

        except InvalidKeywordError:
            pass

#-------------------------------------------------------------------------------
-------------------------------------
# FUNCTION NAME  : tokenize_words
#-------------------------------------------------------------------------------
-------------------------------------
# PARAMETERS     : buyer_items (list)
#-------------------------------------------------------------------------------
-------------------------------------
# RETURN         : tokens (list)
#-------------------------------------------------------------------------------
-------------------------------------
# DESCRIPTION    : Reads each item from the list and splits it according to item name a
nd sub-category
#
#-------------------------------------------------------------------------------
-------------------------------------

    def tokenize_words(self):
        self.tokens = []
        for self.item in self.buyer_items:
            self.wordTokens = nltk.word_tokenize(self.item)

            # remove punctuation from each word
            self.table = str.maketrans('','',string.punctuation)
            self.stripped = [self.w.translate(self.table) for self.w in self.wordTokens
]

            # remove remaining tokens that are not alphabetic
            self.checkWords = [self.word for self.word in self.stripped if self.word.is
alnum()]

            # filter out stop words
            self.stopWords = set(stopwords.words('english'))
            self.tok = [self.w for self.w in self.wordTokens if not self.w in self.stop
Words]
            self.tokens.append(self.tok)

        return self.tokens
```

Module 2: Classification

In [5]:

```
# Reads the dataset "input" csv file
prod_set = pd.read_csv(path/"input.csv")
```

In [6]:

```
# Dictionary initialisation of categories and its items
classify_set = {'Clothes' : ['sweaters','jackets','t-shirts','trousers','shorts','sweat
shirts','formal-pants','track-pants',
                'jeans','shirts','polos','designer boutique','western wear','ethnic wea
r','track-jackets',
                'trunks','lounge wear','suits','blazers'],
'Furniture' : ['divan','faiting-couch','chess-table','safe','sofa-bed',
                'couch','bench','bed','dining-sets','table','chair',
                'daybed','billiard-table','rack','bookcase','television-set',
               'stool','mattress','beanbag'],
'Footwear' : ['socks','casual-shoes','boots','formal-shoes','sports-shoes',
            'floaters','sandals','flip-flops','canvas-shoes','wedge sandals',
            'combat boots','lace shoes','slippers','court shoes','fashion boots','russi
an boots',
            'loafers','sneakers','football boots'],
'Watches' : ['sporty','casual','smart','multi-dial','traveller','formal',
            'party-wear','led-watch','digital-watch','analog-watch','automatic',
            'chronograph','diver','sports','swiss','space','mechanical','luxury',
                'italian design'],
'Sunglasses' : ['shield','butterfly','square','aviator','rectangle',
                'wayfarer','round','oval','clubmaster','cat-eye','semi-rimless',
                'blue ray', 'polarized rectangular','polarized goggle','metal UV protec
ted','full rim',
                'unisex zipper','blue mercury','pilot']
}
```

In [7]:

```
# Creats a dataframe classify_set
classify = pd.DataFrame(classify_set)

# Saves the dataframe
classify.to_csv('classification_set.csv',index=False)
```

In [8]:

```
#-------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : Category_list
#-------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : Category list
#-------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Retrieves Unique categories from the input dataset, appends in the l
ist.
#
#-------------------------------------------------------------------------------------
---------------------------------------

def Category_list():
    cat_list = []
    cat_list.extend(prod_set['Category'].unique())
    return cat_list
```

CLASS NAME : Classification

DESCRIPTION : Classify buyer items to their respective categories

In [9]:

```
class Classification:

#-------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : __init__ (default function)
#-------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : Tokenised items, Categories from dataset (list)
#-------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Initialising tokens and categories
#
#-------------------------------------------------------------------------------------
---------------------------------------

    def __init__(self,keywords,cat_list):
        self.keywords = keywords
        self.cat_list = cat_list

#-------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : getCategory
#-------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Classify input items to their respetive categories and stores in gat
Category file.
#
#-------------------------------------------------------------------------------------
---------------------------------------

    def getCategory(self):
        try:
            # opens a getCategory csv file in write mode
            with open('{}.csv'.format("getCategory"), mode="w+", newline='') as self.wr
itefile:
                self.fieldnames = ['Items','Category','Subcategory']
                self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                self.writer.writeheader()

                # keywords - tokenise words
                for self.wordList in self.keywords:
                    for index, self.word in enumerate(self.wordList):
                        for self.cat in self.cat_list:
                            for self.row in range(classify.shape[0]):

                                # verifies whether length of word is greater than or eq
ual to 2 if yes, store in category and subcategory
                                if len(self.wordList) >= 2:
                                    if classify.loc[self.row,self.cat].lower() == self.
word.lower():
```

```
                                        self.writer.writerow({'Items':r'{}'.format(self
.word), 'Category':r'{}'.format(self.cat), 'Subcategory':r'{}'.format(self.wordList[1
])})

                                # if no, store only in category
                                elif len(self.wordList) == 1:
                                    if classify.loc[self.row,self.cat].lower() == self.
word.lower():
                                        self.writer.writerow({'Items':r'{}'.format(self
.word), 'Category':r'{}'.format(self.cat), 'Subcategory':r'{}'.format("None")})
            self.writefile.close()
        except (RuntimeError, TypeError):
            print("Error occured!")
            sys.exit()
```

Module 3: Spliting Categories

In [10]:

```
#-------------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : splitCategories
#-------------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Splits each unique category from the input dataset into individual d
atasets and storing them in different
#                  csv files.
#-------------------------------------------------------------------------------------------
---------------------------------------

def splitCategories():
    for cat in list(prod_set['Category'].unique()):
        try:
            # opens each category csv file in write mode
            with open(path.joinpath('Categories/{}.csv'.format(cat)),mode="w+",newline=
'') as writefile:
                fieldnames = ['PID','Brand','Name','Category','Subcategory','Price','mi
n_stock','current_stock']
                writer = csv.DictWriter(writefile,fieldnames)
                writer.writeheader()

                # splits unique categories into separate dataset
                for i in range(prod_set.shape[0]):
                    try:
                        if prod_set['Category'].iloc[i] == cat:
                            writer.writerow({'PID':r'{}'.format(prod_set['PID'].iloc[i
]), 'Brand':r'{}'.format(prod_set['Brand'].iloc[i]), 'Name':r'{}'.format(prod_set['Nam
e'].iloc[i]), 'Category':r'{}'.format(prod_set['Category'].iloc[i]), 'Subcategory':r'{}
'.format(prod_set['Subcategory'].iloc[i]), 'Price':r'{}'.format(prod_set['Price'].iloc[
i]), 'min_stock':r'{}'.format(prod_set['min_stock'].iloc[i]), 'current_stock':r'{}'.for
mat(prod_set['current_stock'].iloc[i])})
                    except:
                        pass

            writefile.close()
        except (RuntimeError, TypeError):
            pass
```

In [11]:

```
#-------------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : split_subcategory
#-------------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Splits each unique sub-category from the Categories dataset into ind
ividual datasets with
#                  respect to their categories.
#                  Merging datasets based on Product name and sub-category.
#                  Creates a sub-category (subTable) table which has information of ite
ms and sub-category
#                  (item_subcategory) files created.
#-------------------------------------------------------------------------------------------
---------------------------------------

def split_subcategory():
    try:
        catlist = ['Clothes', 'Watches', 'Footwear', 'Sunglasses']

        # opens a subTable csv file for storing items with subcategories
        with open('{}.csv'.format("subTable"), mode='w+',newline='') as writefile:
            fieldnames = ['Category','Subcategory','Items','Subcategory_file']
            writer = csv.DictWriter(writefile,fieldnames)
            writer.writeheader()
            for cat in catlist:
                if path.joinpath('Categories/{}.csv'.format(cat)).is_file() == False:
                    raise FileNotExistError
                else:
                    # reads each category csv files created from Categories folder
                    data = pd.read_csv(path.joinpath('Categories/{}.csv'.format(cat)))
                    for subcat in list(data['Subcategory'].unique()):
                        for name_item in list(data['Name'].unique()):
                            # merging the dataset based on Subcategory and Name
                            merged = pd.merge(data[data['Name']==name_item],data[data[
'Subcategory']==subcat], on = 'PID', how = 'inner')

                            # dropping duplicate attributes
                            new_set=merged.drop(['Brand_y','Name_y','Category_y','Subca
tegory_y','Price_y','min_stock_y','current_stock_y'],axis=1)

                            #  storing each merged dataframe as Item Name_Subcategory n
ame in Subcategories folder
                            new_set.to_csv(path.joinpath('Subcategories/{}_{}.csv'.form
at(name_item,subcat)),index=False)
                            writer.writerow({'Category' : r'{}'.format(cat), 'Subcatego
ry' : r'{}'.format(subcat), 'Items' : r'{}'.format(name_item),'Subcategory_file' : r'{}
'.format('{}_{}'.format(name_item,subcat))})
        writefile.close()
    except FileNotExistError:
        print("File not found!")
        sys.exit()
```

Module 4: Similarity Measures

In [12]:

```
# Parameters to the algorithm. Currently set to values that was reported in the paper t
o produce "best" results.
ALPHA = 0.2
BETA = 0.45
ETA = 0.4
PHI = 0.2
DELTA = 0.85

brown_freqs = dict()
N = 0
```

In [13]:

```
#----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : get_best_sysnset_pair
#----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : word_1, word_2
#----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : best_pair
#----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Choose the pair with highest path similarity among all pairs.
#                  Mimics pattern-seeking behavior of humans.
#----------------------------------------------------------------------------------------
---------------------------------------

def get_best_synset_pair(word_1, word_2):
    max_sim = -1.0
    synsets_1 = wn.synsets(word_1)
    synsets_2 = wn.synsets(word_2)
    if len(synsets_1) == 0 or len(synsets_2) == 0:
        return None, None
    else:
        max_sim = -1.0
        best_pair = None, None
        for synset_1 in synsets_1:
            for synset_2 in synsets_2:
                sim = wn.path_similarity(synset_1, synset_2)
                if not (sim is None):
                    if float(sim) > max_sim:
                        max_sim = sim
                        best_pair = synset_1, synset_2
        return best_pair

#----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : length_dist
#----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : synset_1, synset_2
#----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : math expression of AlPHA and l_dist
#----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Return a measure of the length of the shortest path in the semantic
#                  ontology (Wordnet in our case as well as the paper's) between two sy
nsets.
#----------------------------------------------------------------------------------------
---------------------------------------

def length_dist(synset_1, synset_2):
    l_dist = sys.maxsize
    if synset_1 is None or synset_2 is None:
        return 0.0
    if synset_1 == synset_2:
        l_dist = 0.0
    else:
        wset_1 = set([str(x.name()) for x in synset_1.lemmas()])
```

```
        wset_2 = set([str(x.name()) for x in synset_2.lemmas()])
        if len(wset_1.intersection(wset_2)) > 0:
            l_dist = 1.0
        else:
            l_dist = synset_1.shortest_path_distance(synset_2)
            if l_dist is None:
                l_dist = 0.0

    return math.exp(-ALPHA * l_dist)

#----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : hierarchy_dist
#----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : synset_1, synset_2
#----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : math expression
#----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Return a measure of depth in the ontology to model the fact that nod
es closer to the root are broader and
#                  have less semantic similarity than nodes further away from the root.
#----------------------------------------------------------------------------------------
---------------------------------------

def hierarchy_dist(synset_1, synset_2):

    h_dist = sys.maxsize
    if synset_1 is None or synset_2 is None:
        return h_dist
    if synset_1 == synset_2:
        h_dist = max([x[1] for x in synset_1.hypernym_distances()])
    else:
        hypernyms_1 = {x[0]:x[1] for x in synset_1.hypernym_distances()}
        hypernyms_2 = {x[0]:x[1] for x in synset_2.hypernym_distances()}
        lcs_candidates = set(hypernyms_1.keys()).intersection(
            set(hypernyms_2.keys()))
        if len(lcs_candidates) > 0:
            lcs_dists = []
            for lcs_candidate in lcs_candidates:
                lcs_d1 = 0
                if lcs_candidate in hypernyms_1:
                    lcs_d1 = hypernyms_1[lcs_candidate]
                lcs_d2 = 0
                if lcs_candidate in hypernyms_2:
                    lcs_d2 = hypernyms_2[lcs_candidate]
                lcs_dists.append(max([lcs_d1, lcs_d2]))
            h_dist = max(lcs_dists)
        else:
            h_dist = 0

    return ((math.exp(BETA * h_dist) - math.exp(-BETA * h_dist)) /
        (math.exp(BETA * h_dist) + math.exp(-BETA * h_dist)))

#----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : word_similarity
#----------------------------------------------------------------------------------------
---------------------------------------
```

```
# PARAMETERS     : word_1, word_2
#-------------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : product of length distance and hierarchy distance of synset pairs
#-------------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Measures the similarity between two words and gives a measured valu
e.
#
#-------------------------------------------------------------------------------------------
---------------------------------------

def word_similarity(word_1, word_2):
    synset_pair = get_best_synset_pair(word_1, word_2)

    return (length_dist(synset_pair[0], synset_pair[1]) *
        hierarchy_dist(synset_pair[0], synset_pair[1]))
```

In [14]:

```
# quadarnts lists
quad1 = []
quad2 = []
quad3 = []
quad4 = []

#-----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : CategorySimilarity
#-----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Compares the similarity score between categories and assign categori
es to four quadrants with respect to
#                  score.
#-----------------------------------------------------------------------------------------
---------------------------------------

def CategorySimilarity():
    try:
        for cat1 in list(prod_set['Category'].unique()):
            for cat2 in list(prod_set['Category'].unique()):
                if cat1 == cat2:
                    continue
                else:
                    # measuring the similarity score on category tuples
                    score = int(word_similarity(cat1,cat2)*100)
                    # assigning quadrants if scores for categories are similar
                    if score > 50:
                        quad1.append(cat1)
                    elif score >= 25 and score < 50:
                        quad2.append(cat1)
                    elif score > 0 and score < 25:
                        quad3.append(cat2)

        quad2.clear()
        quad3.clear()
        for cat1 in list(prod_set['Category'].unique()):
            if cat1 == 'Watches' or cat1 == 'Sunglasses':
                quad2.append(cat1)
            elif cat1 == 'Furniture':
                quad3.append(cat1)

            if cat1 == 'Furniture':
                pass
            else:
                quad4.append(cat1)

        quads = {'quad1' : quad1,
                 'quad2' : quad2,
                 'quad3' : quad3,
                 'quad4' : quad4
                }
    except (RuntimeError, TypeError, ValueError):
```

```
        pass
```

Module 5: BOM Structure

In [15]:

```
#-----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : CreateFactTable
#-----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Creating dataframe of category, sub-category and assigning new colum
n quadrant
#
#-----------------------------------------------------------------------------------------
---------------------------------------

def CreateFactTable():
    try:
        factTable = {'Category': ['Clothes','Clothes','Clothes','Footwear','Footwear',
'Footwear',
                                  'Watches','Watches','Watches','Watches','Sunglasses',
'Sunglasses',
                                 'Furniture','Clothes','Footwear','Watches','Sunglasse
s'],
                     'Subcategory': ['Men','Women','kids','Men','Women','kids',
                                     'Leather-strap','Metal-strap','Synthetic-strap','Ca
nvas-strap',
                                     'Men','Women','None','None','None','None','None'],
                     'Quadrant': ['quad1','quad1','quad1','quad1','quad1','quad1',
                                  'quad2','quad2','quad2','quad2','quad2','quad2',
                                  'quad3','quad4','quad4','quad4','quad4']
        }

        FactTable = pd.DataFrame(factTable)
        # storing as Fact_Table csv file
        FactTable.to_csv('Fact_Table.csv',index=False)
    except (RuntimeError, OSError):
        pass
```

CLASS NAME : Structure

DESCRIPTION : Creates BOM Data Structure

In [16]:

```
class Structure:

#-------------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : __init__ (default function)
#-------------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : buyer_credits
#-------------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Initialising buyer credits
#
#-------------------------------------------------------------------------------------------
---------------------------------------

    def __init__(self,buyer_credits):
        self.buyer_credits = buyer_credits

#-------------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : TestFactTable
#-------------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Assigning items to respective quadrants and calling each quadrant.
#                  Saves the new dataframe as QuadInput csv file.
#-------------------------------------------------------------------------------------------
---------------------------------------

    def TestFactTable(self):
        try:
            self.count = 0
            if path.joinpath("getCategory.csv").is_file() == False or path.joinpath("Fa
ct_Table.csv").is_file() == False:
                raise FileNotExistError
            else:
                # reads getCategory and Fact_Table files
                self.getcat = pd.read_csv(path/"getCategory.csv")
                FactTable = pd.read_csv(path/"Fact_Table.csv")

                # creating new column 'Quadrant'
                self.getcat['Quadrant'] = None
                for self.j in range(self.getcat.shape[0]):
                    for self.i in range(FactTable.shape[0]):

                        # comapring whether categories and subcategories are same
                        if FactTable['Category'].iloc[self.i].lower() == self.getcat['C
ategory'].iloc[self.j].lower() and FactTable['Subcategory'].iloc[self.i].lower() == sel
f.getcat['Subcategory'].iloc[self.j].lower():
                            self.getcat.loc[self.j,'Quadrant']= FactTable['Quadrant'].i
loc[self.i]
```

```
                print(self.getcat)

                # storing dataframe as QuadInput csv file
                self.getcat.to_csv('QuadInput.csv',index=False)

                for self.k in range(self.getcat.shape[0]):

                    # variyfing the quadrants
                    if self.getcat['Quadrant'].iloc[self.k] == 'quad1':
                        self.quads = self.getcat['Quadrant'].iloc[self.k]

                        # Passing the quadrant as variable to Quad1 function call
                        Structure.Quad1(self,self.quads)

                    elif self.getcat['Quadrant'].iloc[self.k] == 'quad2':
                        self.quads = self.getcat['Quadrant'].iloc[self.k]

                        # Passing the quadrant as variable to Quad1 function call
                        Structure.Quad2(self,self.quads)

                    elif self.getcat['Quadrant'].iloc[self.k] == 'quad3':
                        self.quads = self.getcat['Quadrant'].iloc[self.k]

                        # Passing the quadrant as variable to Quad1 function call
                        Structure.Quad3(self,self.quads)

                    elif self.getcat['Quadrant'].iloc[self.k] == 'quad4':
                        self.quads = self.getcat['Quadrant'].iloc[self.k]

                        # Passing the quadrant as variable to Quad1 function call
                        Structure.Quad4(self,self.quads)

        except FileNotExistError:
            print("File not found!")
            sys.exit()

#-----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : Quad1
#-----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : quad1
#-----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Quad1 describes about Items whose subcategories are present and Cate
gories are stored/segregated based on
#                  similarities between them.
#                  Category and sub-category as primary key. Retriving the sub-quadrant
 of the input item.
#                  Saving the sub-quadrant dataframe as quad1 csv file.
#-----------------------------------------------------------------------------------------
---------------------------------------

    def Quad1(self,quads):
        try:
            self.quads = quads
            self.qd1 = ['Clothes','Footwear']
            self.cat = ['Men','Women','Kids']
```

```
            # open quad1 csv file in write mode
            with open('{}.csv'.format(self.quads), mode="w+", newline='') as self.write
file:
                self.fieldnames = ['Category','Subcategory','Items','Subcategory_file',
'SubQuadrant']
                self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                self.writer.writeheader()
                if path.joinpath("subTable.csv").is_file() == False:
                    raise FileNotExistError
                else:
                    # reads subTable for comparing category and subcategory
                    subtab = pd.read_csv(path/"subTable.csv")
                    for self.s in range(subtab.shape[0]):
                        for self.q1 in self.qd1:
                            for self.c in self.cat:

                                # creating subcategory_file and subquadrant for each su
bcategory and items belonging to it
                                if subtab['Category'].iloc[self.s] == self.q1 and subta
b['Subcategory'].iloc[self.s] == self.c:
                                    self.writer.writerow({'Category':r'{}'.format(subta
b['Category'].iloc[self.s]), 'Subcategory':r'{}'.format(subtab['Subcategory'].iloc[self
.s]), 'Items':r'{}'.format(subtab['Items'].iloc[self.s]), 'Subcategory_file':r'{}'.form
at(subtab['Subcategory_file'].iloc[self.s]), 'SubQuadrant':r'{}{}_sq'.format(self.q1,se
lf.c)})
                                else:
                                    pass
            self.writefile.close()
            # Passing the quadrant as variable to getQuad1 function call
            Structure.getQuad1(self,self.quads)
        except (RuntimeError, TypeError, OSError):
            pass
        except FileNotExistError:
            print("File not Found!")
            sys.exit()

#-----------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : getQuad1
#-----------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : buyer_credits
#-----------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Finding input item in item_subcategory file. When item found compari
ng whether it's price is less
#                  than credits.
#                  Retrieving PID's and price of items from respective sub-quadrants, s
toring in quad1List csv file.
#-----------------------------------------------------------------------------------
---------------------------------------

    def getQuad1(self,quad1):
        try:
            self.quad1 = quad1
            # open quad2 csv file in write mode
            if path.joinpath('{}.csv'.format(self.quad1)).is_file() == False or path.jo
```

```
inpath("QuadInput.csv").is_file() == False:
                raise FileNotExistError
            else:
                self.getDict = {}
                # reads QuadInput for comparing category and subcategory
                q1 = pd.read_csv(path/'{}.csv'.format(self.quad1))
                qinput = pd.read_csv(path/"QuadInput.csv")
                for self.qin in range(qinput.shape[0]):
                    for self.q in range(q1.shape[0]):
                        # comparing subcaetgory, retrieving subcategory_file based on s
ub-quadrant assigned
                        if q1['Category'].iloc[self.q].lower() == qinput['Category'].il
oc[self.qin].lower() and q1['Subcategory'].iloc[self.q].lower() == qinput['Subcategory'
].iloc[self.qin].lower() and q1['Items'].iloc[self.q].lower() == qinput['Items'].iloc[s
elf.qin].lower():
                            self.getDict[q1['Subcategory_file'].iloc[self.q]] = q1['Sub
Quadrant'].iloc[self.q]
                        else:
                            pass

                # opens quad1List in write mode
                with open('quad1List.csv', mode='w+', newline='') as self.writefile:
                    self.fieldnames = ['Category','Subcategory','Items','PID','retail_p
rice']
                    self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                    self.writer.writeheader()

                    for self.dict in self.getDict:
                        if path.joinpath('Subcategories/{}.csv'.format(self.dict)).is_f
ile() == False:
                            raise FileNotExistError
                        else:
                            # reads the input item's subcategory_file
                            self.df = pd.read_csv(path.joinpath('Subcategories/{}.csv'.
format(self.dict)))
                            for self.d in range(self.df.shape[0]):
                                # compares the price iwth credits
                                if self.df['Price_x'].iloc[self.d] <= self.buyer_credit
s:
                                    self.writer.writerow({'Category':r'{}'.format(self.
df['Category_x'].iloc[self.d]), 'Subcategory':r'{}'.format(self.df['Subcategory_x'].ilo
c[self.d]), 'Items':r'{}'.format(self.df['Name_x'].iloc[self.d]), 'PID':r'{}'.format(se
lf.df['PID'].iloc[self.d]), 'retail_price':r'{}'.format(self.df['Price_x'].iloc[self.d
])})
                                else:
                                    pass

                self.writefile.close()
                print("Check quad1List.csv.")
        except FileNotExistError:
            print("File not found!")
            sys.exit()

#-----------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : Quad2
#-----------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : quad2
#-----------------------------------------------------------------------------------
---------------------------------------
```

```
# RETURN         : NIL
#--------------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Quad2 describes about Items whose subcategories are present and Cate
gories are stored/segregated based on
#                  similarities between them.
#                  Category and sub-category as primary key. Retriving the sub-quadrant
 of the input item.
#                  Saving the sub-quadrant dataframe as quad2 csv file.
#--------------------------------------------------------------------------------------------
---------------------------------------

    def Quad2(self,quads):
        try:
            self.quads = quads
            self.qd2 = ['Watches','Sunglasses']
            self.cat1 = ['Leather-strap','Metal-strap','Synthetic-strap','Canvas-strap'
]
            self.cat2 = ['Men','Women']

            # open quad1 csv file in write mode
            with open('{}.csv'.format(self.quads), mode="w+", newline='') as self.write
file:
                self.fieldnames = ['Category','Subcategory','Items','Subcategory_file',
'SubQuadrant']
                self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                self.writer.writeheader()
                if path.joinpath("subTable.csv").is_file() == False:
                    raise FileNotExistError
                else:
                    # reads subTable for comparing category and subcategory
                    subtab = pd.read_csv(path/"subTable.csv")
                    for self.s in range(subtab.shape[0]):
                        for self.q2 in self.qd2:
                            for self.c1 in self.cat1:
                                # for watches category
                                # creating subcategory_file and subquadrant for each su
bcategory and items belonging to it
                                if subtab['Category'].iloc[self.s] == self.q2 and subta
b['Subcategory'].iloc[self.s] == self.c1:
                                    self.writer.writerow({'Category':r'{}'.format(subta
b['Category'].iloc[self.s]), 'Subcategory':r'{}'.format(subtab['Subcategory'].iloc[self
.s]), 'Items':r'{}'.format(subtab['Items'].iloc[self.s]), 'Subcategory_file':r'{}'.form
at(subtab['Subcategory_file'].iloc[self.s]), 'SubQuadrant':r'{}{}_sq'.format(self.q2,se
lf.c1)})
                                else:
                                    pass

                            for self.c2 in self.cat2:
                                # for sunglasses category
                                # creating subcategory_file and subquadrant for each su
bcategory and items belonging to it
                                if subtab['Category'].iloc[self.s] == self.q2 and subta
b['Subcategory'].iloc[self.s] == self.c2:
                                    self.writer.writerow({'Category':r'{}'.format(subta
b['Category'].iloc[self.s]), 'Subcategory':r'{}'.format(subtab['Subcategory'].iloc[self
.s]), 'Items':r'{}'.format(subtab['Items'].iloc[self.s]), 'Subcategory_file':r'{}'.form
at(subtab['Subcategory_file'].iloc[self.s]), 'SubQuadrant':r'{}{}_sq'.format(self.q2,se
lf.c2)})
                                else:
                                    pass
```

```
            self.writefile.close()

            # Passing the quadrant as variable to getQuad2 function call
            Structure.getQuad2(self,self.quads)
        except FileNotExistError:
            print("File not found!")
            sys.exit()

#-----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : getQuad2
#-----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : buyer_credits
#-----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Finding input item in item_subcategory file. When item found compari
ng whether it's price is less
#                  than credits.
#                  Retrieving PID's and price of items from respective sub-quadrants, s
toring in quad2List csv file.
#-----------------------------------------------------------------------------------------
---------------------------------------

    def getQuad2(self,quad2):
        try:
            self.quad2 = quad2

            if path.joinpath('{}.csv'.format(self.quad2)).is_file() == False or path.jo
inpath("QuadInput.csv").is_file() == False:
                raise FileNotExistError
            else:
                self.getDict = {}
                # open quad2 csv file in write mode
                q2 = pd.read_csv(path/'{}.csv'.format(self.quad2))
                # reads QuadInput for comparing category and subcategory
                qinput = pd.read_csv(path/"QuadInput.csv")
                for self.qin in range(qinput.shape[0]):
                    for self.q in range(q2.shape[0]):
                        # comparing subcaetgory, retrieving subcategory_file based on s
ub-quadrant assigned
                        if q2['Category'].iloc[self.q].lower() == qinput['Category'].il
oc[self.qin].lower() and q2['Subcategory'].iloc[self.q].lower() == qinput['Subcategory'
].iloc[self.qin].lower() and q2['Items'].iloc[self.q].lower() == qinput['Items'].iloc[s
elf.qin].lower():
                            self.getDict[q2['Subcategory_file'].iloc[self.q]] = q2['Sub
Quadrant'].iloc[self.q]
                        else:
                            pass
                # opens quad2List in write mode
                with open('quad2List.csv', mode='w+', newline='') as self.writefile:
                    self.fieldnames = ['Category','Subcategory','Items','PID','retail_p
rice']
                    self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                    self.writer.writeheader()

                    for self.dict in self.getDict:
                        if path.joinpath('Subcategories/{}.csv'.format(self.dict)).is_f
```

```
ile() == False:
                            raise FileNotExistError
                        else:
                            # reads the input item's subcategory_file
                            df = pd.read_csv(path.joinpath('Subcategories/{}.csv'.forma
t(self.dict)))
                            for self.d in range(df.shape[0]):
                                # compares the price with credits
                                if df['Price_x'].iloc[self.d] <= self.buyer_credits:
                                    self.writer.writerow({'Category':r'{}'.format(df['C
ategory_x'].iloc[self.d]), 'Subcategory':r'{}'.format(df['Subcategory_x'].iloc[self.d
]), 'Items':r'{}'.format(df['Name_x'].iloc[self.d]), 'PID':r'{}'.format(df['PID'].iloc[
self.d]), 'retail_price':r'{}'.format(df['Price_x'].iloc[self.d])})
                                else:
                                    pass

                self.writefile.close()
                print("Check quad2List.csv.")
        except FileNotExistError:
            print("File Not Found!")
            sys.exit()

#-----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : Quad3
#-----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : quad3
#-----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Quad3 describes about Items whose sub-categories are not present.
#                  Category and sub-category as primary key. Retriving the sub-quadrant
 of the input item.
#                  Saving the sub-quadrant dataframe as quad3 csv file.
#-----------------------------------------------------------------------------------------
---------------------------------------

    def Quad3(self,quads):
        try:
            self.quads = quads
            if path.joinpath('Categories/Furniture.csv').is_file() == False:
                raise FileNotExistError
            else:
                self.qd3 = ['Furniture']
                for self.i in self.qd3:
                    # reads the furniture csv file
                    furniture = pd.read_csv(path/'Categories/{}.csv'.format(self.i))

                # open the quad3 csv in write mode
                with open('{}.csv'.format(self.quads), mode="w+", newline='') as self.w
ritefile:
                    self.fieldnames = ['Category','Subcategory','Items','Subcategory_fi
le','SubQuadrant']
                    self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                    self.writer.writeheader()

                    # stores unique item names in the file
                    for self.q3 in self.qd3:
```

```
                    for self.fur in list(furniture['Name'].unique()):
                        self.writer.writerow({'Category':r'{}'.format(self.q3), 'Su
bcategory':r'{}'.format("None"), 'Items':r'{}'.format(self.fur), 'Subcategory_file':r'
{}'.format("None"), 'SubQuadrant':r'{}'.format("None")})

                self.writefile.close()

            # Passing the quadrant as variable to getQuad3 function call
            Structure.getQuad3(self,self.quads)
        except FileNotExistError:
            print("File Not Found!")
            sys.exit()

#-----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : getQuad3
#-----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : buyer_credits, quad3
#-----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Finding input item in Category file. When item found comparing wheth
er it's price is less
#                  than credits.
#                  Retrieving PID's and price of items, storing in quad3List csv file.
#-----------------------------------------------------------------------------------------
---------------------------------------

    def getQuad3(self,quad3):
        try:
            self.quad3 = quad3

            # open the quad3List in write mode
            with open('quad3List.csv', mode='w+', newline='') as self.writefile:
                self.fieldnames = ['Category','Subcategory','Items','PID','retail_pric
e']
                self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                self.writer.writeheader()

                # reads the quad3 csv file created
                qd3 = pd.read_csv(path/'{}.csv'.format(self.quad3))
                for self.q3 in list(qd3['Category'].unique()):
                    if path.joinpath('Categories/{}.csv'.format(self.q3)).is_file() ==
False or path.joinpath('QuadInput.csv').is_file() == False:
                        raise FileNotExistError
                    else:
                        # reads the category file in the quad3 file
                        q3 = pd.read_csv(path/'Categories/{}.csv'.format(self.q3))

                        qinput = pd.read_csv(path/'QuadInput.csv')
                        self.Counts = dict(qinput["Items"].value_counts())
                        for self.ItemCounts in self.Counts:
                            for self.q in range(q3.shape[0]):
                                # matches the items
                                if self.ItemCounts.lower() == q3['Name'].iloc[self.q].l
ower():
                                    # compares the price with credits
                                    if q3['Price'].iloc[self.q] <= self.buyer_credits:
```

```
                                        self.writer.writerow({'Category':r'{}'.format(q
3['Category'].iloc[self.q]), 'Subcategory':r'{}'.format("None"), 'Items':r'{}'.format(q
3['Name'].iloc[self.q]), 'PID':r'{}'.format(q3['PID'].iloc[self.q]), 'retail_price':r'
{}'.format(q3['Price'].iloc[self.q])})
                                    else:
                                        pass
                                else:
                                    pass

            self.writefile.close()
            print("Check quad3List.csv.")
        except FileNotExistError:
            print("File Not Found!")
            sys.exit()

#-----------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : Quad4
#-----------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : buyer_credits
#-----------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-----------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Quad4 describes about Items whose sub-categories are present but are
 not specified in the input by buyer.
#                  It is considered as default quadrant.
#                  Varifying the catgeory of an item belonging to quad4. Saving in quad
4 csv file.
#                  .
#-----------------------------------------------------------------------------------------
---------------------------------------

    def Quad4(self,quads):
        try:
            self.quads = quads
            self.qd4 = ['Clothes','Footwear','Watches','Sunglasses']

            # open the quad4 csv file in write mode
            with open('{}.csv'.format(self.quads), mode="w+", newline='') as self.write
file:
                self.fieldnames = ['Category','Subcategory','Items','Subcategory_file',
'SubQuadrant']
                self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                self.writer.writeheader()
                if path.joinpath('QuadInput.csv').is_file() == False:
                    raise FileNotExistError
                else:
                    # reads the QuadInput csv file
                    qinput = pd.read_csv(path/'QuadInput.csv')
                    for self.q4 in self.qd4:
                        for self.qin in range(qinput.shape[0]):
                            # compares the category and sub-category belonging to quad4
                            if qinput['Subcategory'].iloc[self.qin] == "None" and qinpu
t['Category'].iloc[self.qin] == self.q4:
                                self.writer.writerow({'Category':r'{}'.format(self.q4),
 'Subcategory':r'{}'.format("None"), 'Items':r'{}'.format(qinput['Items'].iloc[self.qin
]), 'Subcategory_file':r'{}'.format("None"), 'SubQuadrant':r'{}'.format("None")})
```

```
            self.writefile.close()

            # Passing the quadrant as variable to getQuad3 function call
            Structure.getQuad4(self,self.quads)
        except FileNotExistError:
            print("File Not found!")
            sys.exit()


#-------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : getQuad4
#-------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : buyer_credits, quad4
#-------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Finding input item in Category file. When item found comparing wheth
er it's price is less
#                  than credits.
#                  Retrieving PID's and price of items, storing in quad3List csv file.
#-------------------------------------------------------------------------------------
---------------------------------------

    def getQuad4(self,quad4):
        try:
            self.quad4 = quad4

            # opens the quad3List in write mode
            with open('quad4List.csv', mode='w+', newline='') as self.writefile:
                self.fieldnames = ['Category','Subcategory','Items','PID','retail_pric
e']
                self.writer = csv.DictWriter(self.writefile,self.fieldnames)
                self.writer.writeheader()
                if path.joinpath('QuadInput.csv').is_file() == False:
                    raise FileNotExistError
                else:
                    # reads quad4 csv file
                    qd4 = pd.read_csv(path/'{}.csv'.format(self.quad4))

                    # reads QuadInput csv file
                    qinput = pd.read_csv(path/'QuadInput.csv')
                    self.CatCounts = dict(qinput["Category"].value_counts())
                    self.ProductCounts = dict(qd4["Items"].value_counts())

                    for self.q4 in list(qd4['Category'].unique()):
                        if path.joinpath('Categories/{}.csv'.format(self.q4)).is_file()
 == False:
                            raise FileNotExistError
                        else:
                            # reads each category file
                            cat = pd.read_csv(path.joinpath('Categories/{}.csv'.format(
self.q4)))
                            for self.ItemCounts in self.ProductCounts:
                                for self.c in range(cat.shape[0]):
                                    # matches the Item name and checks for price <= cre
dits
                                    if cat['Name'].iloc[self.c].lower() == self.ItemCou
```

```
nts.lower() and cat['Price'].iloc[self.c] <= self.buyer_credits:
                                        self.writer.writerow({'Category':r'{}'.format(c
at['Category'].iloc[self.c]), 'Subcategory':r'{}'.format("None"), 'Items':r'{}'.format(
cat['Name'].iloc[self.c]), 'PID':r'{}'.format(cat['PID'].iloc[self.c]), 'retail_price':
r'{}'.format(cat['Price'].iloc[self.c])})
                                    else:
                                        pass
                            else:
                                pass

            self.writefile.close()
            print("Check quad4List.csv.")
        except FileNotExistError:
            print("File Not Found!")
            sys.exit()
```

In [17]:

```
#-------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : insertStock
#-------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Opens the Categories and QuadInput file, compares with respect to it
ems specified in input.
#                  Verifies whether current_stock is less than or equal to min_stock if
 yes, inserts random integer value
#                  in location of current_stock.
#-------------------------------------------------------------------------------------
---------------------------------------

def insertStock():
    try:
        if path.joinpath('QuadInput.csv').is_file() == False:
            raise FileNotExitError
        else:
            # reads the QuadInput csv file
            qinput = pd.read_csv(path/'QuadInput.csv')
            for qin in range(qinput.shape[0]):
                if path.joinpath('Categories/{}.csv'.format(qinput['Category'].iloc[qin
])).is_file() == False:
                    raise FileNotExistError
                else:
                    # reads the each Category csv file from categories folder
                    data = pd.read_csv(path.joinpath('Categories/{}.csv'.format(qinput[
'Category'].iloc[qin])))
                    for df in range(data.shape[0]):
                        # matches the item name
                        if data['Name'].iloc[df] == qinput['Items'].iloc[qin]:
                            # compares the current_stock <= min_stock
                            if data['current_stock'].iloc[df] <= data['min_stock'].iloc
[df]:
                                data['current_stock'].iloc[df] = random.randint((data[
'min_stock'].iloc[df]+1),50)
                            else:
                                pass
                        else:
                            pass
    except FileNotExistError:
        print("File Not Found!")
        sys.exit()
```

In [18]:

```
#-------------------------------------------------------------------------------------------
---------------------------------------
# FUNCTION NAME  : main()
#-------------------------------------------------------------------------------------------
---------------------------------------
# PARAMETERS     : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# RETURN         : NIL
#-------------------------------------------------------------------------------------------
---------------------------------------
# DESCRIPTION    : Operates with the defined classes and functions
#
#-------------------------------------------------------------------------------------------
---------------------------------------

def main():

    nq = Normalise_query()
    nq.buyer_item()
    nq.spell_check()
    keywords = nq.tokenize_words()
    credits = nq.buyer_credits()
    cat_list = Category_list()
    cl = Classification(keywords,cat_list)
    cl.getCategory()
    splitCategories()
    split_subcategory()
    CategorySimilarity()
    CreateFactTable()
    struct = Structure(credits)
    struct.TestFactTable()
    insertStock()

main()
```

```
5
['safe', 'trousers', 'sports-shoes for women', 'sofa-bed', 'shield for me
n']
buyer_credits = 4110 in range.
          Items    Category Subcategory Quadrant
0          safe   Furniture        None    quad3
1      trousers     Clothes        None    quad4
2  sports-shoes    Footwear       women    quad1
3      sofa-bed   Furniture        None    quad3
4        shield  Sunglasses         men    quad2
Check quad3List.csv.
Check quad4List.csv.
Check quad1List.csv.
Check quad3List.csv.
Check quad2List.csv.
```

In [ ]:

```
#%tb
```