```
# -*- coding: utf-8 -*-

"""
PyProc.py - This program has been developed to process a JSON Parameter File, interpret/analyse the tabular or json
source file and create a metaData file describing the data in the source file. PyProc consists of a class PyProc which
has within it 4 functions:

    1. __init__(self,):
        initialises the class with defining the directory location of the parameter file

    2. queryParam_file (self,):
        analyses the specified parameter file, executes the appropriate Analysis function

    3. tabularAnalysis(self):
        analyses the .csv or .txt file, creates a metaData dictionary and dumps the metaData file

    4. jsonAnalysis(self):
        analyses the .json file, creates a metaData dictionary and dumps the metaData file

###

USER INSTRUCTIONS: Pyproc.py was developed in Python 2.7 and requires a few additional libraries to the standard python libraries.

1. Go to the Windows Command Prompt in Windows (->press windows_key on keyboard, ->type 'cmd' ->press enter_key on keyboard)

    a. Intall the Pandas library: (->type 'conda install anaconda', ->press enter_key on keyboard -> follow installation instructions
       to install the Anaconda distribution which includes Pandas and its dependancies)

    b. Intall the Seaborn library: (->type 'pip install seaborn', ->press enter_key on keyboard -> follow installation instructions)

    c. Intall the Counter library: (->type 'pip install Counter', ->press enter_key on keyboard -> follow installation instructions)

    d. Install the tKinter library: (->type 'pip install tkinter', ->press enter_key on keyboard -> follow installation instructions)

2. Once required libraries have been installed, re-open PyProc.py file and run the program in your preferred python console - recommended
   consoles include Eclipse, Spyder, or Jupyter (among others).

3. As the program is set up, a GUI (kTinker) will open for the user to choose the parameter file. Failing this there are two alternative
   means for specifying the directory location of the parameter file.
```

```
###

@author: Mr Haveron - Student Number 2527317

Edited - 10/11/2016

"""

# Import the libraries used in PyProc.py
import csv
import json
import re
import pandas as pd
import seaborn as sns
from collections import Counter, OrderedDict
from tkFileDialog import askopenfilename
```

```
class PyProc(object):

    """  define class Pyproc and further define its internal functions  """

    # ***   FUNCTION DEFINITIONS   *** #

    def __init__(self):

        """  initialise PyProc class by defining a variable param_file_directory to the directory of the parameter file  """

        # Input option #1 - uses a tKinter dialog box to allow the user to search for the parameter file - look for the red "tk" window
        #in the taskbar
        #self.param_file_directory = askopenfilename() # show an "Open" dialog box and return the path to the selected file


        # OR use:

        # Input option #2 - manually paste/type the source parameter file directory next to the arrow
        #self.param_file_directory = raw_input("Hello, welcome to PyProc. Please type or paste the parameter source file directory in,
        #adjacent to the arrow below and press the enter key:\n\n *NOTE: please copy the FULL source directory for the parameter file,
        #without parenthesis, as per the example directory below:\n\n C:\Users\Mr Haveron\Documents\BD2\CSVcarsparams.json\n\n->")
        # insert parameter file directory

        # OR use:

        # Input option #3 - manually replace the source parameter file directory in the parenthesis, below
        self.param_file_directory = "C:\Users\Mr Haveron\Documents\BD2\CSVcarsparams.json" # insert parameter file directory

    """  end of  __init__  """


    def queryParam_file(self):

        """  queryParam_file function extracts and stores information from the parameter file about the data file  """

        # identifies values for keys 'metafile' & 'infile' -> stores these as string variables
        with open (self.param_file_directory, "rb") as parameter_file:
            self.paramData = json.load(parameter_file)
            self.metaFile = str(self.paramData["metafile"])
            self.sourceFile = str(self.paramData["infile"])

        # searches the parameter file for expected extension types
        with open (self.param_file_directory, "rb") as parameter_file:
            for extensionTypes in parameter_file:

                # searches the parameter file for .csv file extensions
                if re.search("\.csv", extensionTypes):
                    self.fileType = str("tabular(CSV)")
                    self.fieldSeperator = str(",") # this is an assumption for .csv format but confirmed later in PyProc
                    self.tabularAnalysis() # executes tabularAnalysis if .csv format

                # searches the parameter file for .txt file extensions
                elif re.search("\.txt", extensionTypes):
                    self.fileType = str("tabular(txt)")
                    self.fieldSeperator = str("\t") # this is an assumption for .txt format but confirmed later in PyProc
                    self.tabularAnalysis() # executes tabularAnalysis if .txt format

                # searches the parameter file for .json file extensions
                elif re.search("\.json", extensionTypes):
                    self.fileType = str("json_objects")
                    self.fieldSeperator = str("key - value pairs") # this is the standard format for json data schema
                    self.jsonAnalysis() # executes jsonAnalysis if .json format


        # identifies "format", "hasheader" & "separator" keys (where specified) and stores as string variables
        with open (self.param_file_directory, "rb") as parameter_file:
            self.paramData = json.load(parameter_file)

            if re.search("format", extensionTypes):
                self.fileType = str(self.paramData["format"])
            if re.search("hasheader", extensionTypes):
                self.hasHeader = str(self.paramData["hasheader"])
            if re.search("separator", extensionTypes):
                self.fieldSeperator = str(self.paramData["separator"])  # Note - PyProc.py will evaluate and confirm the format,
                #header and separator - irrespective of the specification here

    """  end of queryParam_file  """
```

```
def tabularAnalysis(self):

    """  tabularAnalysis function calculates/infers metadata from the tabular (.csv/.txt) data file  """

    # identifies whether a header exists - the code below determines the common data 'schema' and evaluates whether the first row is
    #consistent with this common data 'schema' apparent in the rest of the data file
    with open(self.sourceFile, "rb") as tabularFile:
        header = csv.Sniffer().has_header(tabularFile.read(1024))
        if not header:
            tabularFile.seek(0)
            self.hasHeader = str("No_header")
            self.fieldName_Required = 1 # a boolean marker used to indicate whether field names need to be generated in the metaData file
        else:
            self.hasHeader = str("Has_header")
            self.fieldName_Required = 0 # a boolean marker used to indicate whether field names need to be generated in the metaData file

    # identifies the data delimiter and saves as a variable, delim
    with open(self.sourceFile, "r") as tabularFile:
        snif = csv.Sniffer().sniff(tabularFile.read(1024))
        delim = snif.delimiter

    # wrangles the csv/txt data into a format for analysis. "print data" (outside loops, at the end of tabularAnalysis) yields
    #- { field1 : [value11,value12,value1N], field2 : [value21,value22,value2N] }
    with open(self.sourceFile, "rb") as tabularFile:
        reader = csv.DictReader(tabularFile, delimiter=delim)
        self.tab_data = {}
        for row in reader:
            for header, value in row.items():
                try:
                    self.tab_data[header].append(value)
                except KeyError:
                    self.tab_data[header] = [value]

        # determine the number of rows and columns(fields) in the tabular data folder:
        self.numberRows = sum(1 for line in open(self.sourceFile))
        self.numberFields = sum(1 for header in row)
```

```
# populates the empty list fieldData
self.fieldData = []
for field in range(0,self.numberFields):
    self.key = self.tab_data.keys()[field]
    self.values = self.tab_data.values()[field]

    # pyproc tries to convert tabular field values into floats, failing this converts the field values into strings
    # a set of computations are carried out in both scenarios

    try:
        value = [float(i) for i in self.values]
        self.minList = min(self.values)
        self.maxList = max(self.values)
        self.meanList = int(sum(value))/int(len(value))
        self.uniquevals = Counter(self.values)
        self.listType = str(type(value[1]))
        if type(value[1]) == float:
            self.listType = "numeric"
        if self.fieldName_Required == 1 :
            self.key = str("Field "+ str(field+1))

        # use seaborn library to plot the distibution plot of float values
        sns.distplot(value, hist=True)
        break
        #df = pd.read_csv(self.sourceFile)
        #sns.factorplot(self.tab_data.keys()[field],data=df,kind='count',size=4, aspect=2)

        """   Ive attempted to plot these seperate lists of numeric values on a distribution plot to show a histogram of value
        distribution (print value, prints these lists),however cannot figure out how to plot all of the lists on seperate
        individual plots, - the current setup plots all three sets of list data on the same plot. The idea behind this is
        to give the pyproc user some insight into value distribution for numeric values and count distribution for string values,
        however this proves difficult to pre-empt the variety of data types and structure   """

        # populates an ordered dictionary called fieldData
        field1toN =OrderedDict([('Field_Name',self.key),('Field_Type',self.listType ),('min_value',self.minList ),
        ('max_value',self.maxList ),('mean_value',self.meanList)])
        self.fields = {"Field "+ str(field+1):field1toN}
        self.fieldData.append(self.fields)
```

```
            except ValueError:
                value = [str(i) for i in self.values]
                self.uniquevals = Counter(self.values)
                self.modeList = max(self.values, key = self.values.count)
                self.listType = str(type(value[1]))
                if type(value[1]) == str:
                    self.listType = "string"
                if self.fieldName_Required == 1 :
                    self.key = str("Field "+ str(field+1))


                # uses pandas library, along with seaborn to plot the distibution of string field values
                df = pd.read_csv(self.sourceFile)
                sns.factorplot(self.tab_data.keys()[field],data=df,kind='count',size=4, aspect=2)

                # populates an ordered dictionary called fieldData
                field1toN =OrderedDict([('Field_Name',self.key),('Field_Type',self.listType ),('mode',self.modeList ),
                ('uniquevals',self.uniquevals )])
                self.fields = {"Field "+ str(field+1):field1toN}
                self.fieldData.append(self.fields)

        # dumps the metaData to the metaSource_File, specified in the parameterFile
        with open (self.metaFile, "w") as metaSource_File:
            metaData = OrderedDict([("data_format", self.fileType),("separator", self.fieldSeperator),("header", self.hasHeader),
            ("data_source_directory", self.sourceFile),("number_rows (incl. header, where applicable)", self.numberRows),
            ("number_fields/colums", self.numberFields), ("field_descriptions", self.fieldData)])
            json.dump(metaData,metaSource_File,indent = 4)

            print "\nThe metaData has been successfully dumped."

"""  end of tabularAnalysis  """

def jsonAnalysis(self):

 """  jsonAnalysis function calculates/infers metadata from the json data file  """

 # wrangle the json data into a format appropriate for analysis
 with open(self.sourceFile, "rb") as jsonFile:
```

```python
self.hasHeader = str("json objects - no header") # this is an assumption for .json data files
self.lines = jsonFile.readlines()
jsonKeys = json.loads(self.lines[0])
keys = jsonKeys.keys() #Store the object keys into a list

# determines the number of objects and fields in the json data folder:
self.numberObjects = sum(1 for line in self.lines)
self.numberFields = len(keys)

# populates the empty list fieldData
self.fieldData = []
for x in range(0, self.numberFields):
    self.key = str(keys[x])
    self.values = []
    for i in range(0,self.numberObjects):
        jsonD = json.loads(self.lines[i])
        self.values.append(jsonD[self.key])

    # pyproc tries to convert json object values into floats, failing that converts the keys' values into strings.
    # a set of computations are carried out in both scenarios
    try:
        value = [float(i) for i in self.values] #or int(i)
        self.minList = min(self.values)
        self.maxList = max(self.values)
        self.meanList = int(sum(value))/int(len(value))
        self.listType = str(type(value[1]))
        if type(value[1]) == float:
            self.listType = "numeric"

        # populate an ordered dictionary called fieldData
        field1toN =OrderedDict([('Field_Name',self.key),('Field_Type',self.listType ),('min_value',self.minList ),
        ('max_value',self.maxList ),('mean_value',self.meanList)])
        self.fields = {"Field "+ str(x+1):field1toN}
        self.fieldData.append(self.fields)

    except ValueError:
        value = [str(i) for i in self.values]
        self.minList = "N/A"
        self.maxList = "N/A"
```

```
                self.meanList = "N/A"
                self.uniquevals = Counter(self.values)
                self.modeList = max(self.values, key = self.values.count)
                self.listType = str(type(value[1]))
                if type(value[1]) == str:
                    self.listType = "string"

                # populate an ordered dictionary called fieldData
                field1toN =OrderedDict([('Field_Name',self.key),('Field_Type',self.listType ),('mode',self.modeList ),
                ('uniquevals',self.uniquevals )])
                self.fields = {"Field "+ str(x+1):field1toN}
                self.fieldData.append(self.fields)

                # use pandas library, along with seaborn to plot the distibution of string field values
                with open (self.sourceFile) as jsonFile:
                    jsonData = jsonFile.readlines()
                    parsable_data = "["+','.join(jsonData).replace('\n','')+"]"
                    df = pd.read_json(parsable_data, orient = 'columns', typ = 'frame', convert_dates = ['created'])
                    sns.factorplot(str(keys[x]),data=df,kind='count',size=4, aspect=2)

        # dump the metaData to the metaSource_File
        with open (self.metaFile, "w") as metaSource_File:
            metaData = OrderedDict([("data_format", self.fileType),("schema", self.fieldSeperator),("header", self.hasHeader),
            ("data_source_directory", self.sourceFile),("number_Objects", self.numberObjects),("number_fields", self.numberFields),
            ("field_descriptions", self.fieldData)])
            json.dump(metaData,metaSource_File,indent = 4)

            print "\nThe metaData has been successfully dumped."

        """  end of jsonAnalysis  """

# call class PyProc() and the instance queryParam_file()
runFunction = PyProc()
runFunction.queryParam_file()
```