*Add\_search*

*"""  
This module is used to add Search step  
"""***class** AddSearch(object):  
 *"""  
 This class is used to perform search operation on requested type  
 """* **def** \_\_init\_\_(self, input\_json):  
 self.input\_json = input\_json  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method will perform search* **:return***:  
 """* print(self.input\_json)  
 **return** data\_frame

*Aggregate\_function*

*"""  
This module is used for calculating aggregate function on numeric column.  
Min  
Max  
Sum  
Average  
std  
median  
25th percentile  
50th percentile  
90th percentile  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** constants  
**import** message\_constants  
*# LOGGER = logging.getLogger(\_\_name\_\_)  
# LOGGER.setLevel(logging.DEBUG)  
# HANDLER = RotatingFileHandler('pandas\_api.log', maxBytes=20000000, backupCount=10)  
# FORMATTER = logging.Formatter(fmt='%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
# ' %(funcName)s:%(lineno)d')  
# HANDLER.setFormatter(FORMATTER)  
# LOGGER.addHandler(HANDLER)*LOGGER = logging.getLogger(**'lens\_backend'**)  
  
**class** AggregateFunction(object):  
 *"""  
 This class will manage the code for run all the kind of aggregation function  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.agg\_func\_type = input\_json[constants.AGGREGATE\_FUNCTION\_TYPE]  
 self.column\_name = input\_json[constants.COLUMN\_NAME]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method will perform aggregate function.* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **if** self.agg\_func\_type **in** [constants.PERCENTILE\_25, constants.PERCENTILE\_75,  
 constants.PERCENTILE\_90]:  
 **if** self.agg\_func\_type == constants.PERCENTILE\_25:  
 function\_value = data\_frame[self.column\_name].quantile(0.25)  
 **elif** self.agg\_func\_type == constants.PERCENTILE\_75:  
 function\_value = data\_frame[self.column\_name].quantile(0.75)  
 **else**:  
 function\_value = data\_frame[self.column\_name].quantile(0.9)  
  
 **else**:  
 function\_value = data\_frame.agg({self.column\_name: [self.agg\_func\_type]}  
 )[self.column\_name][self.agg\_func\_type]  
  
 data\_frame[self.column\_name + **"\_"** + self.agg\_func\_type] = function\_value  
 **return** data\_frame  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.NONE\_INPUT\_DATASET)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to test aggregate\_function module.  
"""***import** unittest  
**import** pandas **as** pd  
**from** aggregate\_function **import** AggregateFunction  
  
  
**class** TestAggregateFunction(unittest.TestCase):  
 *"""  
 This class is used to maintain all the scenario for testing aggragate function  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/10Records.txt"**,  
 delimiter=**"\t"**,  
 parse\_dates=[**'field0'**])  
  
 **def** test\_key\_missing\_input\_json(self):  
 *"""  
 Test suite for testing min function for column* **:return***:  
 """* **try**:  
 json = {**"column\_ame"**: **"field4"**, **"agg\_func\_type"**: **"min"**}  
 AggregateFunction(json)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_min\_function(self):  
 *"""  
 Test suite for testing min function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"min"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_min'**, data\_frame=data\_frame)  
  
 **def** test\_max\_function(self):  
 *"""  
 Test suite for testing max function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"max"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_max'**, data\_frame=data\_frame)  
  
 **def** test\_mean\_function(self):  
 *"""  
 Test suite for testing mean function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"mean"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_mean'**, data\_frame=data\_frame)  
  
 **def** test\_sum\_function(self):  
 *"""  
 Test suite for testing sum function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"sum"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_sum'**, data\_frame=data\_frame)  
  
 **def** test\_std\_function(self):  
 *"""  
 Test suite for testing std function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"std"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_std'**, data\_frame=data\_frame)  
  
 **def** test\_median\_function(self):  
 *"""  
 Test suite for testing median function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"median"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_median'**, data\_frame=data\_frame)  
  
 **def** test\_25\_function(self):  
 *"""  
 Test suite for testing percentile\_25 function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"percentile\_25"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_percentile\_25'**, data\_frame=data\_frame)  
  
 **def** test\_75\_function(self):  
 *"""  
 Test suite for testing percentile\_75 function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"percentile\_75"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_percentile\_75'**, data\_frame=data\_frame)  
  
 **def** test\_90\_function(self):  
 *"""  
 Test suite for testing percentile\_90 function for column* **:return***:  
 """* json = {**"columnName"**: **"field4"**, **"agg\_func\_type"**: **"percentile\_90"**}  
 agg\_function = AggregateFunction(json)  
 data\_frame = agg\_function.get\_result(self.data\_frame)  
 self.check\_in\_column(column\_name=**'field4\_percentile\_90'**, data\_frame=data\_frame)  
  
 **def** check\_in\_column(self, column\_name, data\_frame):  
 *"""  
 This function will check whether column exist or not in the dataframe* **:param** *column\_name:* **:param** *data\_frame:* **:return***:  
 """* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_key\_error\_function(self):  
 *"""  
 Test suite for testing min function for column* **:return***:  
 """* **try**:  
 json = {**"columnName"**: **"field77"**, **"agg\_func\_type"**: **"percentile\_90"**}  
 agg\_function = AggregateFunction(json)  
 agg\_function.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_data\_frame\_none\_function(self):  
 *"""  
 Test suite for testing min function for column* **:return***:  
 """* **try**:  
 json = {**"columnName"**: **"field77"**, **"agg\_func\_type"**: **"percentile\_90"**}  
 agg\_function = AggregateFunction(json)  
 agg\_function.get\_result(**None**)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)

Anomaly\_detection

*# this code is perfect and which is tested on pylint  
"""  
Returns Row level outliers  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** sklearn.ensemble **import** IsolationForest  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** RowLevelOutlier:  
 *"""  
 Args used are:  
 outlier\_columns: Columns on which outliers should be calculated  
 run\_type: This can be keep,remove,flag (used to return data according to what's asked for)  
 """* **def** \_\_init\_\_(self, json\_value):  
 **try**:  
 self.outlier\_columns = json\_value[**"outlier\_columns"**]  
 self.run\_type = json\_value[**"run\_type"**]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** get\_outliers\_info(data\_frame):  
 *"""  
 This function is used to get information about outlier columns  
 """* **def** get\_outlier\_column(incoming\_df, temp, column\_list\_var):  
 *"""  
 used in apply on a dataframe for fast run time  
 """* outlier\_list = []  
 **for** i **in** column\_list\_var:  
 **if** incoming\_df[i] > round(  
 temp[i].iloc[1] + 1.5 \* temp[i].iloc[0]  
 ) **or** incoming\_df[i] < round(temp[i].iloc[1] - 1.5 \* temp[i].iloc[0]):  
 value = (  
 i + **"("** + str(incoming\_df[i]) + **")"** + **", avg is:"** + str(round(temp[i].iloc[1], 2))  
 + **" and std is:"** + str(round(temp[i].iloc[0], 2))  
 )  
 outlier\_list.append(value)  
  
 **if not** outlier\_list:  
 outlier\_list.append(**"generic outlier"**)  
  
 **return** str(outlier\_list)  
  
 df\_outlier\_explanation = data\_frame.query(**"outlier\_columns == -1"**)  
 df3 = data\_frame.describe().reset\_index()  
 df3 = df3.query(**"index == 'std' | index == '50%'"**)  
  
 column\_list = df\_outlier\_explanation.columns.tolist()  
 column\_list.remove(**"outlier\_columns"**)  
  
 df\_outlier\_explanation[**"reason"**] = df\_outlier\_explanation.apply(  
 get\_outlier\_column, axis=1, temp=df3, column\_list\_var=column\_list  
 )  
 **return** df\_outlier\_explanation  
  
 @staticmethod  
 **def** get\_non\_outliers(data\_frame):  
 *"""  
 This function returns the non outlier rows  
 """* df\_outlier\_explanation = data\_frame[data\_frame.outlier\_columns == 1]  
 df\_outlier\_explanation = df\_outlier\_explanation.drop(  
 columns=[**"outlier\_columns"**]  
 )  
 **return** df\_outlier\_explanation  
  
 @staticmethod  
 **def** get\_outlier\_flag(data\_frame):  
 *"""  
 This fucntion runs rows with outlier flags  
 """* data\_frame[**"outlier\_columns"**].replace(1, **"False"**, inplace=**True**)  
 data\_frame[**"outlier\_columns"**].replace(-1, **"True"**, inplace=**True**)  
 data\_frame.rename(columns={**"outlier\_columns"**: **"outlier\_flag"**}, inplace=**True**)  
  
 **return** data\_frame  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 Args:  
 data\_frame: data\_frame on which features are created.  
 \*\*  
 This method is exposed into the api which returns row level outliers  
 """* **try**:  
 data\_frame\_numeric = data\_frame[self.outlier\_columns]  
 numerics = [**"int16"**, **"int32"**, **"int64"**, **"float16"**, **"float32"**, **"float64"**]  
 data\_frame\_numeric = data\_frame\_numeric.select\_dtypes(include=numerics)  
 data\_frame\_numeric = data\_frame\_numeric.fillna(data\_frame.mean())  
 clf = IsolationForest(n\_estimators=50, random\_state=10, max\_samples=0.33)  
 clf.fit(data\_frame\_numeric)  
 data\_frame\_numeric[**"outlier\_columns"**] = clf.predict(data\_frame\_numeric)  
  
 **if** self.run\_type == **"flag"**:  
 df\_out = self.get\_outlier\_flag(data\_frame\_numeric)  
 out\_df = data\_frame.join(df\_out[[**'outlier\_flag'**]], how=**'inner'**)  
 **elif** self.run\_type == **"remove"**:  
 df\_out = self.get\_non\_outliers(data\_frame\_numeric)  
 out\_df = data\_frame[data\_frame.index.isin(df\_out.index)]  
 **elif** self.run\_type == **"keep"**:  
 df\_out = self.get\_outliers\_info(data\_frame\_numeric)  
 out\_df = data\_frame.join(df\_out[[**'reason'**]], how=**'inner'**)  
  
 **return** out\_df  
 **except** MemoryError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INSUFFICIENT\_MEMORY\_FOR\_PERFORMING\_OPERATION)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to write a test case for sort\_step  
"""***import** unittest  
**import** pandas **as** pd  
**from** anomaly\_detection **import** RowLevelOutlier  
  
  
**class** TestRowLevelOutlier(unittest.TestCase):  
 *"""  
 This class is used to write test method for sort step  
 """* **def** setUp(self):  
 sales\_df = {  
 **"male"**: [  
 1,  
 0,  
 1,  
 0,  
 0,  
 0,  
 0,  
 0,  
 1,  
 1,  
 0,  
 0,  
 1,  
 0,  
 0,  
 0,  
 1,  
 0,  
 0,  
 1,  
 0,  
 0,  
 0,  
 0,  
 1,  
 1,  
 0,  
 1,  
 0,  
 0,  
 1,  
 1,  
 0,  
 1,  
 1,  
 1,  
 1,  
 1,  
 0,  
 1,  
 0,  
 0,  
 0,  
 1,  
 0,  
 0,  
 0,  
 1,  
 0,  
 1,  
 ],  
 **"age"**: [  
 39,  
 46,  
 48,  
 61,  
 46,  
 43,  
 63,  
 45,  
 52,  
 43,  
 50,  
 43,  
 46,  
 41,  
 39,  
 38,  
 48,  
 46,  
 38,  
 41,  
 42,  
 43,  
 52,  
 52,  
 44,  
 47,  
 60,  
 35,  
 61,  
 60,  
 36,  
 43,  
 59,  
 61,  
 54,  
 37,  
 56,  
 52,  
 42,  
 36,  
 43,  
 41,  
 52,  
 54,  
 53,  
 49,  
 65,  
 46,  
 63,  
 36,  
 ],  
 }  
 self.df\_test = pd.DataFrame.from\_dict(sales\_df)  
  
 **def** test\_keep\_outliers\_good\_flow(self):  
 *"""  
 Test suite for performing keep operation* **:return***:  
 """* column\_list1 = self.df\_test.columns.tolist()  
 sample\_dict = {}  
 sample\_dict[**"outlier\_columns"**] = column\_list1  
 sample\_dict[**"run\_type"**] = **"keep"** class\_fc = RowLevelOutlier(sample\_dict)  
 out\_df = class\_fc.get\_result(self.df\_test)  
 **if** out\_df.shape[0] == 5:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_keep\_outliers\_bad\_flow(self):  
 *"""  
 Test suite for performing keep operation, but added extra column* **:return***:  
 """* column\_list1 = self.df\_test.columns.tolist()  
 column\_list1.append(**"extra\_column"**)  
 sample\_dict = {}  
 sample\_dict[**"outlier\_columns"**] = column\_list1  
 sample\_dict[**"run\_type"**] = **"keep"  
  
 with** self.assertRaises(Exception):  
 class\_fc = RowLevelOutlier(sample\_dict)  
 class\_fc.get\_result(self.df\_test)  
  
 **with** self.assertRaises(Exception):  
 sample\_dict.pop(**"outlier\_columns"**)  
 RowLevelOutlier(sample\_dict)  
  
 **def** test\_remove\_outliers\_good\_flow(self):  
 *"""  
 Test suite for performing keep operation* **:return***:  
 """* column\_list1 = self.df\_test.columns.tolist()  
 sample\_dict = {}  
 sample\_dict[**"outlier\_columns"**] = column\_list1  
 sample\_dict[**"run\_type"**] = **"remove"** class\_fc = RowLevelOutlier(sample\_dict)  
 out\_df = class\_fc.get\_result(self.df\_test)  
 **if** out\_df.shape[0] == 45:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_flag\_outliers\_good\_flow(self):  
 *"""  
 Test suite for performing keep operation* **:return***:  
 """* column\_list1 = self.df\_test.columns.tolist()  
 sample\_dict = {}  
 sample\_dict[**"outlier\_columns"**] = column\_list1  
 sample\_dict[**"run\_type"**] = **"flag"** class\_fc = RowLevelOutlier(sample\_dict)  
 out\_df = class\_fc.get\_result(self.df\_test)  
 **if** out\_df.shape[0] == 50:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_column\_outliers\_good\_flow(self):  
 *"""  
 Test suite for performing keep operation* **:return***:  
 """* column\_list1 = self.df\_test.columns.tolist()  
 sample\_dict = {}  
 sample\_dict[**"outlier\_columns"**] = column\_list1  
 sample\_dict[**"run\_type"**] = **"flag"** class\_fc = RowLevelOutlier(sample\_dict)  
 out\_df = class\_fc.get\_result(self.df\_test)  
 **if "outlier\_flag" in** out\_df.columns.tolist():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 unittest.main()

Auto\_feature

*# auto level code and features only flag  
"""  
Automated feature creation using the package featuretools  
URL: https://github.com/Featuretools/featuretools  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** featuretools **as** ft  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** AutoFeatures: *# pylint: disable-msg=R0902  
 """  
 Args used are:  
 set\_id: Name given for the operation like retail,tax etc  
 df\_list: List of dataframes on which joins are done  
 relation\_list: Relation between the datsframes  
 parent\_index\_column: Index column of the parent df, this df comes in get\_result method.  
 user\_id: used to get the dataframes from existing lens api's  
 """* **def** \_\_init\_\_(self, json\_value):  
 **try**:  
 self.set\_id = json\_value[**"set\_id"**]  
 self.df\_list = json\_value[**"df\_list"**]  
 self.relation\_list = json\_value[**"relation\_list"**]  
 self.parent\_index\_column = json\_value[**"parent\_index\_column"**]  
 self.user\_id = json\_value[**"user\_id"**]  
 self.run\_type = json\_value[**"run\_type"**]  
 self.parent\_df = json\_value[**"parent\_df"**]  
 self.single\_group\_key = json\_value[**"single\_group\_key"**]  
 self.level = json\_value[**"level"**]  
  
 **if "get\_feature\_list\_only" not in** json\_value.keys():  
 self.feature\_list\_flag = **"false"  
 else**:  
 self.feature\_list\_flag = json\_value[**"get\_feature\_list\_only"**]  
  
 self.column\_list\_flag = **True  
 if "column\_list" not in** json\_value.keys():  
 self.column\_list\_flag = **False  
 else**:  
 self.column\_list = json\_value[**"column\_list"**]  
  
 **if "agg\_primitives" not in** json\_value.keys():  
 self.agg\_primitives = [  
 **"sum"**,  
 **"std"**,  
 **"max"**,  
 **"min"**,  
 **"mean"**,  
 **"count"**,  
 **"percent\_true"**,  
 **"mode"**,  
 ]  
 **else**:  
 self.agg\_primitives = json\_value[**"agg\_primitives"**]  
  
 **if "trans\_primitives" not in** json\_value.keys():  
 self.trans\_primitives = [**"day"**, **"year"**, **"month"**, **"weekday"**, **"haversine"**]  
 **else**:  
 self.trans\_primitives = json\_value[**"trans\_primitives"**]  
  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** add\_entity\_in\_entity\_set(es\_set, e\_t):  
 *"""  
 Args:  
 es\_set: existing object of the entity set.  
 e\_t: This is tuple of dataframe and index\_column of the dataframe  
 \*\*  
 This method adds dataframe in entity set objects  
 """* es\_set = es\_set.entity\_from\_dataframe(  
 entity\_id=e\_t[0].lens\_df\_name, dataframe=e\_t[0], index=e\_t[1]  
 )  
 **return** es\_set  
  
 @staticmethod  
 **def** get\_features(  
 entity\_set,  
 target\_entity,  
 max\_features,  
 agg\_primitives,  
 max\_depth,  
 threshold\_null,  
 features\_only,  
 trans\_primitives,  
 ): *# pylint: disable-msg=R0913  
 """  
 Args:  
 es\_set: existing object of the entity set.  
 e\_t: This is tuple of dataframe and index\_column of the dataframe  
 \*\*  
 This method actually runs the auto features code  
 """* **if** features\_only == **"true"**: *#pylint: disable=no-else-return* feature\_list = ft.dfs(  
 entityset=entity\_set,  
 target\_entity=target\_entity,  
 max\_features=max\_features,  
 agg\_primitives=agg\_primitives,  
 trans\_primitives=trans\_primitives,  
 max\_depth=max\_depth,  
 features\_only=**True**,  
 )  
  
 feature\_list\_cleaned = []  
 **for** featurelens\_df\_name **in** feature\_list:  
 feature\_list\_cleaned.append(featurelens\_df\_name.getlens\_df\_name())  
  
 **return** feature\_list\_cleaned  
 **else**:  
 feature\_matrix, \_ = ft.dfs(  
 entityset=entity\_set,  
 target\_entity=target\_entity,  
 max\_features=max\_features,  
 agg\_primitives=agg\_primitives,  
 trans\_primitives=trans\_primitives,  
 max\_depth=max\_depth,  
 )  
 feature\_matrix.reset\_index(level=0, inplace=**True**)  
 feature\_matrix = feature\_matrix.drop(  
 feature\_matrix.isnull()  
 .mean()[feature\_matrix.isnull().mean() > threshold\_null]  
 .index.values,  
 axis=1,  
 )  
 **return** feature\_matrix  
  
 @staticmethod  
 **def** add\_relations(relation\_list\_add\_relations, e\_rel):  
 *"""  
 Args:  
 relation\_list\_add\_relations: This dictionary has parent\_dataset,parent\_column,  
 child\_dataset and child\_column.  
 e\_rel: this is entity set object, links are added to this object  
 \*\*  
 This method adds relation which are taken form user in form of a dictioanry.  
 """* relation\_list\_es = []  
 **for** i **in** relation\_list\_add\_relations:  
 rel = ft.Relationship(  
 e\_rel[i[**"parent\_dataset"**]][i[**"parent\_column"**]],  
 e\_rel[i[**"child\_dataset"**]][i[**"child\_column"**]],  
 )  
 relation\_list\_es.append(rel)  
 e\_rel = e\_rel.add\_relationships(relation\_list\_es)  
 **return True  
  
 def** get\_result(self, data\_frame): *#pylint: disable-msg=too-many-branches  
 """  
 Args:  
 data\_frame: data\_frame on which features are created.  
 \*\*  
 This method is exposed into teh api which will prepare the features  
 and return the dataframe back to user.  
 """* **try**:  
 threshold\_null = 0.3  
 data\_frame.lens\_df\_name = self.parent\_df  
 max\_features = 50  
 agg\_primitives = self.agg\_primitives  
 **if not** self.df\_list:  
 max\_depth\_user = 1  
 **else**:  
 max\_depth\_user = len(self.df\_list)  
  
 target\_entity = data\_frame.lens\_df\_name  
  
 entity\_set = ft.EntitySet(self.set\_id)  
 **for** i **in** self.df\_list:  
 *# df\_method = get\_data(i['data\_set\_name'])* df\_method = DataRegistry().create\_data\_frame\_using\_data\_source\_id(  
 i[**"data\_source\_id"**], **False**, user\_id=self.user\_id  
 )  
 df\_method.lens\_df\_name = i[**"data\_set\_name"**]  
 index\_column = i[**"index\_column"**]  
 j\_tuple = (df\_method, index\_column)  
 entity\_set = self.add\_entity\_in\_entity\_set(entity\_set, j\_tuple)  
 j\_tuple = (data\_frame, self.parent\_index\_column)  
 entity\_set = self.add\_entity\_in\_entity\_set(entity\_set, j\_tuple)  
  
 **if** self.run\_type == **"multiple"**:  
 self.add\_relations(self.relation\_list, entity\_set)  
 output\_df = self.get\_features(  
 entity\_set,  
 target\_entity,  
 max\_features,  
 agg\_primitives,  
 max\_depth\_user,  
 threshold\_null,  
 self.feature\_list\_flag,  
 self.trans\_primitives,  
 )  
  
 **elif** self.run\_type == **"single"**:  
 entity\_set.normalize\_entity(  
 base\_entity\_id=j\_tuple[0].lens\_df\_name,  
 new\_entity\_id=**"new\_id\_lens"**,  
 index=self.single\_group\_key,  
 )  
 target\_entity = **"new\_id\_lens"** output\_df = self.get\_features(  
 entity\_set,  
 target\_entity,  
 max\_features,  
 agg\_primitives,  
 max\_depth\_user,  
 threshold\_null,  
 self.feature\_list\_flag,  
 self.trans\_primitives,  
 )  
  
 **if** self.feature\_list\_flag == **"false"**:  
 **if** self.column\_list\_flag:  
 output\_df = output\_df[self.column\_list]  
 output\_df.columns = output\_df.columns.str.replace(**"[.]"**, **"\_"**)  
 output\_df.columns = output\_df.columns.str.replace(**"[(]"**, **"\_"**)  
 output\_df.columns = output\_df.columns.str.replace(**"[)]"**, **""**)  
  
 **return** output\_df  
 **except** ValueError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.DATASET\_NOT\_FOUND)  
 **except** AssertionError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **if** ex.args[0] == **"features must be a non-empty list of features"**:  
 **return** data\_frame  
 **elif "Index is not unique on dataframe" in** ex.args[0]:  
 **raise** Exception(message\_constants.KEY\_NOT\_UNIQUE)  
 **else**:  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **except** MemoryError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INSUFFICIENT\_MEMORY\_FOR\_PERFORMING\_OPERATION)  
 *# this was done to make sure lens has same function all over.  
 #So we handle the error and return original df* **except** IndexError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** data\_frame  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to test case for the auto features step  
"""***import** unittest  
**import** pandas **as** pd  
**from** mock **import** patch  
**from** auto\_features **import** AutoFeatures  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestJoinStep(unittest.TestCase):  
 *"""  
 This class is used for implement test method of auto features step  
 """* **def** setUp(self):  
 self.data\_frame1 = pd.read\_csv(**'../resources/customers.txt'**, delimiter=**','**)  
 self.data\_frame2 = pd.read\_csv(**'../resources/orders.txt'**, delimiter=**','**)  
 self.json\_value = {**'df\_list'**: [{**'data\_set\_name'**: **'olist\_orders\_dataset'**,  
 **'index\_column'**: **'order\_id'**, **'data\_source\_id'**: **'1'**}],  
 **'relation\_list'**: [{**'parent\_dataset'**: **'olist\_customers\_dataset'**,  
 **'parent\_column'**: **'customer\_id'**, **'child\_dataset'**:  
 **'olist\_orders\_dataset'**,  
 **'child\_column'**: **'order\_id'**}],  
 **'parent\_index\_column'**: **'customer\_id'**, **'set\_id'**: **'retail'**, **'user\_id'**:  
 **'10'**,  
 **'parent\_df'**: **'olist\_customers\_dataset'**, **'run\_type'**: **'single'**,  
 **'single\_group\_key'**: **'customer\_id'**, **'level'**: 1}  
  
 **def** test\_auto\_features\_step\_with\_mock(self):  
 *"""  
 This method for performing join operation on data\_frame using right data frame* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = AutoFeatures(self.json\_value)  
 data\_frame1 = join\_step.get\_result(self.data\_frame1)  
 self.assertEqual(len(data\_frame1), 100)  
  
 **def** test\_auto\_features\_error(self):  
 *"""  
 This method passes None instead of DF* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = AutoFeatures(self.json\_value)  
 **with** self.assertRaises(Exception):  
 join\_step.get\_result(**None**)  
  
 **def** test\_auto\_features\_error\_key(self):  
 *"""  
 This method tests for key error in input json* **:return***:  
 """* newjson = self.json\_value  
 newjson.pop(**'set\_id'**)  
 **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 **with** self.assertRaises(Exception):  
 AutoFeatures(newjson)  
  
 **def** test\_null\_data(self):  
 *"""  
 This method for handling null dataframe* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 **try**:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = AutoFeatures(self.json\_value)  
 join\_step.get\_result(self.data\_frame1)  
 self.assertEqual(1, 1)  
 **except** IndexError:  
 self.assertEqual(1, 2)  
 **except** AssertionError:  
 self.assertEqual(1, 2)  
  
 **def** test\_name\_column(self):  
 *"""  
 This method for testing whether name column works in auto-features* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 **try**:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = AutoFeatures(self.json\_value)  
 join\_step.get\_result(self.data\_frame1)  
 self.assertEqual(1, 1)  
 **except** Exception:  
 self.assertEqual(1, 2)  
  
 **def** test\_std\_null\_columns(self):  
 *"""  
 This method for testing whether null columns are dropped* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = AutoFeatures(self.json\_value)  
 data\_frame1 = join\_step.get\_result(self.data\_frame1)  
 **if** data\_frame1.isnull().values.any():  
 self.assertEqual(1, 2)  
 **else**:  
 self.assertEqual(1, 1)  
  
 **def** test\_primary\_key\_exception(self):  
 *"""  
 This method for testing whether correct exception is raised on primary key violation* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 **try**:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = AutoFeatures(self.json\_value)  
 join\_step.get\_result(self.data\_frame1)  
 **except** AssertionError **as** ex:  
 **if "{0}"**.format(str(ex)) == **"Column selected in key is not unique"**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_auto\_init\_method(self):  
 *"""  
 This method for testing whether correct exception is raised on primary key violation* **:return***:  
 """* json\_value = {**'df\_list'**: [{**'data\_set\_name'**: **'olist\_orders\_dataset'**,  
 **'index\_column'**: **'order\_id'**, **'data\_source\_id'**: **'1'**}],  
 **'relation\_list'**: [{**'parent\_dataset'**: **'olist\_customers\_dataset'**,  
 **'parent\_column'**: **'customer\_id'**, **'child\_dataset'**:  
 **'olist\_orders\_dataset'**,  
 **'child\_column'**: **'order\_id'**}],  
 **'get\_feature\_list\_only'**: **'false'**,  
 **'trans\_primitives'**: [**"day"**, **"year"**, **"month"**, **"weekday"**, **"haversine"**],  
 **'agg\_primitives'**: [**"sum"**, **"std"**, **"max"**, **"min"**, **"mean"**, **"count"**,  
 **"percent\_true"**, **"mode"**, ],  
 **'parent\_index\_column'**: **'customer\_id'**, **'set\_id'**: **'retail'**, **'user\_id'**:  
 **'10'**,  
 **'parent\_df'**: **'olist\_customers\_dataset'**, **'run\_type'**: **'single'**,  
 **'single\_group\_key'**: **'customer\_id'**, **'level'**: 1,  
 **"column\_list"**: [**"customer\_id"**], **"run\_type"**: **"multiple"** }  
  
 **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = AutoFeatures(json\_value)  
 join\_step.get\_result(self.data\_frame1)  
  
 **def** test\_auto\_unique\_error(self):  
 *"""  
 This method for performing join operation on data\_frame using right data frame* **:return***:  
 """* data\_frame1\_notunique = pd.read\_csv(**'../resources/cust\_not\_unique.txt'**, delimiter=**','**)  
  
 **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = AutoFeatures(self.json\_value)  
 **with** self.assertRaises(Exception):  
 join\_step.get\_result(data\_frame1\_notunique)

Batch\_job

*"""  
This module is used to initiate batch job using data\_source\_id and lens\_id  
"""***import** base64  
**import** boto3  
**import** datetime  
**import** json  
**import** logging  
**import** numpy **as** np  
**import** os  
**import** pandas **as** pd  
**import** requests  
**import** uuid  
**from** batch\_job.data\_frame\_writer **import** SparkDataFrameWriter  
**from** botocore.client **import** Config  
**from** flask **import** jsonify  
**from** json\_parser\_utility **import** JsonUtility  
**from** random **import** randint  
**from** request\_builder **import** RequestBuilder  
**from** token\_registry **import** TokenRegistry  
  
**import** constants  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** pandas\_api\_rest **import** PandasApi  
**from** save\_user\_config.user\_preferences **import** UserPreferences  
**import** wrangler\_publish\_logger  
**from** batch\_job.gbiio\_client **import** GBIIOClient  
**import** cx\_Oracle  
  
  
**"""LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler('pandas\_api.log', maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt='%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d')  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)"""***# logging.basicConfig(level=logging.DEBUG, filename="pandas\_api.log", format='%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
# ' %(funcName)s:%(lineno)d')  
# LOGGER = logging.getLogger(\_\_name\_\_)*LOGGER = logging.getLogger(**'lens\_backend'**)  
  
**class** JobRunner: *# pylint: disable=R0902  
 """  
 This class contains all the batch job runner code  
 which will execute steps array on the data source.  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.data\_source\_id = input\_json[constants.DATA\_SOURCE\_ID]  
 self.lens\_id = input\_json[constants.LENS\_ID]  
 self.lens\_name = input\_json.get(constants.LENS\_NAME, **'DefaultLensName'**)  
 self.table\_name = self.lens\_name + **"\_"** + \  
 input\_json[constants.LOAD\_TYPE] + **"\_"** + str(randint(100, 999))  
 self.user\_id = input\_json[constants.USER\_ID]  
 LOGGER.info(**"start fetching steps details"**)  
 self.lens\_details = RequestBuilder().call\_http\_get\_request(  
 constants.LENS\_FETCH\_URL + str(self.lens\_id), {},  
 TokenRegistry.get\_token\_for\_user(user\_id=self.user\_id))  
 LOGGER.info(**"end fetching steps details"**)  
 self.is\_scheduled = input\_json.get(**"is\_scheduled"**, **False**)  
 self.input\_json = input\_json  
 self.data\_source\_name = input\_json[constants.DATA\_SOURCE\_NAME]  
 self.project\_id = input\_json[constants.PROJECT\_ID]  
 self.project\_name = input\_json[constants.PROJECT\_NAME]  
 self.user\_email = input\_json[constants.USER\_EMAIL]  
 self.cookie = input\_json[constants.COOKIE]  
 self.user\_log\_path = **''** randId = uuid.uuid4()  
 projName = self.project\_name[:60] **if** len(self.project\_name) > 60 **else** self.project\_name  
 self.jobName = constants.JOB\_NAME + **'-'** + projName + **'-'** + self.project\_id + **'-'** + datetime.datetime.now().strftime(**"%Y%m%d%H%M%S"**)  
 LOGGER.info(**"job name:"**+self.jobName)  
 self.publish\_logger = wrangler\_publish\_logger.getLogger(self.jobName+**".log"**)  
 self.runID = **''  
 except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** run\_batch\_job(self):  
 *"""  
 This method will initiate the batch job using class attribute data\_source\_id and lens\_id* **:return***:  
 """* **try**:  
 LOGGER.info(**"inside run\_batch\_job"**)  
 publish\_logger = self.publish\_logger  
 *# start-updated user for scheduled lens* schedule\_user\_id = self.user\_id  
 **if** self.is\_scheduled:  
 authorization\_header = TokenRegistry.get\_token\_for\_user(user\_id=self.user\_id)  
 LOGGER.info(**"start fetching lens details"**)  
 fetch\_lens\_details = RequestBuilder().call\_http\_get\_request(  
 constants.GET\_LENS\_DETAILS + str(self.lens\_id), {}, authorization\_header).json()  
 LOGGER.info(**"end fetching lens details"**)  
 schedule\_user\_id = fetch\_lens\_details.get(**"createdBy"**, self.user\_id)  
 self.input\_json.update({constants.USER\_ID: self.user\_id})  
 **try**:  
 data\_frame = self.load\_data\_frame(self.input\_json, self.lens\_details)  
 **except** Exception **as** exp:  
 LOGGER.error(**"data\_frame not loaded - Exception: %s"**, exp)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# create job integration starts here* **try**:  
 jobID = self.create\_job()  
 **except** Exception **as** exp:  
 publish\_logger.error(**"error getting while job creation - Exception: %s"**, exp)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# App to App authentication starts here* **try**:  
 user\_pref = UserPreferences(config=**None**,user\_name=**None**)  
 file\_format, write\_mode = user\_pref.get\_userpref\_format\_mode(self.lens\_id,self.user\_id)  
 LOGGER.info(**"get File\_Format {} and Write\_Mode {}"**.format(file\_format,write\_mode))  
 filename = **"wp\_"**+ datetime.datetime.now().strftime(**"%Y%m%d%H%M%S"**) + **"."** + file\_format  
 dataframe = data\_frame.replace(np.nan, 0, regex=**True**)  
 client = GBIIOClient()  
 token = client.gbiio\_init(self.user\_id,self.project\_id)  
 **if** token **is not None**:  
 save\_loc\_path = client.write\_data(dataframe, filename, self.project\_id, file\_format, write\_mode)  
 publish\_logger.info(**"Mcqueen save location {}"**.format(str(save\_loc\_path)))  
 **if** save\_loc\_path **is not None**:  
 read\_dataframe = client.read\_data(save\_loc\_path,file\_format,constants.ROW\_LIMIT)  
 client.publish\_data(read\_dataframe, save\_loc\_path, self.table\_name, self.user\_id,self.data\_source\_id)  
 **else**:  
 publish\_logger.info(**"Mcqueen save location is None"**)  
 **else**:  
 publish\_logger.info(**"Token is None"**)  
 **except** Exception **as** exp:  
 publish\_logger.error(**"Exception in gbiio service %s"**, exp)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 *# App to App authentication Ends here* **try**:  
 self.updateJobStatus()  
 self.delete\_job(jobID)  
 publish\_logger.info(**"writing log file into s3 started::::"**)  
 self.writingLogsIntoS3()  
 publish\_logger.info(**"writing log file into s3 completed::::"**)  
 **except** Exception **as** exp:  
 publish\_logger.error(**"error getting while loading logs into s3 %s"**, exp)  
 **raise** Exception(message\_constants.WRITE\_LOGS\_S3\_BUCKET)  
 **except** Exception **as** ex:  
 publish\_logger.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** load\_data\_frame(input\_json, lens\_details):  
 *"""function to load data set based on type"""* **try**:  
 LOGGER.info(**"inside load\_data\_frame"**)  
 **if** str(input\_json[constants.LOAD\_TYPE]) == **"derived"**:  
 LOGGER.info(**"start fetching data frame details"**)  
 data\_frame = DataRegistry().create\_data\_frame\_using\_data\_source\_id(  
 input\_json[constants.DATA\_SOURCE\_ID], load\_all\_flag=**True**,  
 user\_id=input\_json[constants.USER\_ID],publish=**True**)  
 LOGGER.info(**"end fetching data frame details"**)  
 new\_data\_frame = data\_frame  
 lens\_details=lens\_details.json()  
 **for** step **in** json.loads(lens\_details[**'stepConfig'**]):  
 step.update({**"user\_id"**: input\_json[constants.USER\_ID]})  
 new\_data\_frame = PandasApi.apply\_individual\_step(  
 data\_frame=new\_data\_frame, step=step)  
  
 **elif** str(input\_json[constants.LOAD\_TYPE]) == **"cluster"**:  
 new\_data\_frame = \  
 DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=  
 input\_json[constants.DATA\_SOURCE\_ID],  
 lens\_id=input\_json[constants.LENS\_ID],  
 data\_frame\_type=**'CLUSTER'**)  
 **if** input\_json[constants.JSON\_VALUE]:  
 **for** key, value **in** input\_json[constants.JSON\_VALUE].items():  
 new\_data\_frame[**'cluster\_id'**] = new\_data\_frame[**'cluster\_id'**].replace(key,  
 value)  
  
 **elif** str(input\_json[constants.LOAD\_TYPE]) == **"cluster\_summary"**:  
 new\_data\_frame = JobRunner.json\_to\_data\_frame(input\_json[constants.JSON\_VALUE])  
 **return** new\_data\_frame  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** json\_to\_data\_frame(json\_value):  
 *"""This function converts json to desired data frame format"""* **try**:  
 data\_frame = pd.DataFrame(columns=[**'clusters'**, **'features'**, **'property'**, **'value'**])  
 **for** cluster, dict1 **in** json\_value.items():  
 **for** features, dict2 **in** dict1.items():  
 **for** stats, value **in** dict2.items():  
 data\_frame = data\_frame.append({**'clusters'**: cluster, **'features'**: features,  
 **'property'**: stats, **'value'**: value},  
 ignore\_index=**True**)  
 data\_frame[**'property'**] = data\_frame[**'property'**].replace([**'50%'**], **'median'**)  
 **return** data\_frame  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** hive\_mysql\_operations(json\_param):  
 *""" This function is designed to perform all Hive and Mysql operations,  
 It consumes Java services to perform mysql operations like create table and fetch data* **:return***:"""* **try**:  
 authorization\_header = \  
 TokenRegistry.get\_token\_for\_user(user\_id=json\_param[constants.USER\_ID])  
  
 *# create table on hive* SparkDataFrameWriter.create\_table\_in\_hive(  
 JobRunner.generate\_create\_table\_statement(data\_frame=json\_param[**"data"**],  
 new\_table\_name=  
 json\_param[constants.TABLE\_NAME]))  
  
 *# dump pandas data into newly created hive table* SparkDataFrameWriter.write\_data\_frame\_to\_hive\_table(  
 data\_frame=json\_param[**"data"**], table\_name=json\_param[constants.TABLE\_NAME])  
  
 authorization\_header = \  
 TokenRegistry.get\_token\_for\_user(user\_id=json\_param[constants.USER\_ID])  
  
 *# creating data source from hive tables* data\_source\_create\_json = {  
 **"userid"**: json\_param[constants.USER\_ID],  
 **"datasourcename"**: str(json\_param[constants.TABLE\_NAME])  
 }  
 **if** json\_param[**"is\_scheduled"**]:  
 data\_source\_create\_json.update({**"userid"**: json\_param[**"schedule\_user\_id"**]})  
 request\_builder = RequestBuilder()  
 data\_source\_create = request\_builder.call\_http\_post\_request(  
 constants.DRIVE\_DATA\_SOURCE\_CREATE,  
 data\_source\_create\_json,  
 authorization\_header).json()  
  
 **if** JsonUtility.check\_key\_exist\_in\_dict(data\_source\_create, constants.ERROR):  
 **raise** Exception(data\_source\_create[constants.MESSAGE])  
 *# update output data source in lens* lens\_output\_response = request\_builder.call\_http\_put\_request(  
 constants.UPDATE\_LENS\_OUTPUT.format(json\_param[constants.LENS\_ID],  
 data\_source\_create[constants.DATA\_ID]),  
 {}, authorization\_header).json()  
  
 **if** JsonUtility.check\_key\_exist\_in\_dict(lens\_output\_response, constants.ERROR):  
 **raise** Exception(lens\_output\_response[constants.MESSAGE])  
  
 authorization\_header.update({**"Content-Type"**: **"application/json"**})  
  
 notification\_details = {**"assetType"**: **"publish\_dataset"**,  
 **"assetId"**: data\_source\_create[constants.DATA\_ID],  
 **"assetName"**: json\_param[constants.TABLE\_NAME],  
 **"accessType"**: **"NA"**}  
 authorization\_header = \  
 TokenRegistry.get\_token\_for\_user(user\_id=json\_param[constants.USER\_ID])  
 notif\_user\_id = json\_param[constants.USER\_ID]  
 **if** json\_param[**"is\_scheduled"**]:  
 notif\_user\_id = json\_param[**"schedule\_user\_id"**]  
 RequestBuilder().call\_http\_post\_request(  
 constants.ADD\_NOTIFICATION, JobRunner.prepare\_notification\_json(  
 notif\_user\_id,  
 str(json\_param[constants.TABLE\_NAME])  
 + **" data set have been created"**, **"Batch\_Job\_success"**, notification\_details)  
 , authorization\_header)  
 authorization\_header = \  
 TokenRegistry.get\_token\_for\_user(user\_id=json\_param[constants.USER\_ID])  
 alert\_details = {**"userId"**: notif\_user\_id,  
 **"lensId"**: json\_param[constants.LENS\_ID],  
 **"assetId"**: data\_source\_create[constants.DATA\_ID],  
 **"assetType"**: **"data\_Source"**,  
 **"dataSourceName"**: json\_param[constants.TABLE\_NAME]}  
 RequestBuilder().call\_http\_post\_request(  
 constants.MANAGE\_ALERTS,  
 alert\_details,  
 authorization\_header  
 ).json()  
  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 authorization\_header = \  
 TokenRegistry.get\_token\_for\_user(user\_id=json\_param[constants.USER\_ID])  
 notif\_user\_id = json\_param[constants.USER\_ID]  
 **if** json\_param[**"is\_scheduled"**]:  
 notif\_user\_id = json\_param[**"schedule\_user\_id"**]  
 RequestBuilder().call\_http\_post\_request(  
 constants.ADD\_NOTIFICATION,  
 JobRunner.prepare\_notification\_json(user\_id=notif\_user\_id,  
 notification\_message=str(ex).replace(**"\""**, **""**)  
 .replace(**"'"**, **""**).replace(**'\\'**, **''**),  
 notification\_type=**"Batch\_job\_error"**,  
 notification\_details={}),  
 authorization\_header).json()  
  
 @staticmethod  
 **def** prepare\_notification\_json(user\_id, notification\_message, notification\_type,  
 notification\_details):  
 *"""  
 Prepare json for notification* **:return***:  
 """* **try**:  
 **return** {  
 **"notificationDescription"**: notification\_message,  
 **"notificationURL"**: **"/datalib"**,  
 **"createdBy"**: user\_id,  
 **"notificationType"**: notification\_type,  
 **"notificationDetails"**: notification\_details  
 }  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** generate\_create\_table\_statement(cls, data\_frame, new\_table\_name):  
 *"""  
 generate create table statement based on data frame data type* **:return***:  
 """* **try**:  
 create\_table\_statement = **"create table "** + new\_table\_name + **'('** i = 0  
 **for** column\_name, column\_data\_type **in** data\_frame.dtypes.items():  
 i += 1  
 **if** column\_data\_type == **'int64'**:  
 create\_table\_statement += **'`'** + column\_name + **'` '** + **' int'  
 elif** column\_data\_type == **'float64'**:  
 create\_table\_statement += **'`'** + column\_name + **'` '** + **' double'  
 elif** column\_data\_type == **'datetime64[ns]'**:  
 create\_table\_statement += **'`'** + column\_name + **'` '** + **' timestamp'** *# # having some problem while inserting the datetime column data in hive  
 # so we are changing the data type to string for datetime type column* data\_frame[column\_name] = data\_frame[column\_name].astype(str)  
 **elif** column\_data\_type == **'bool'**:  
 create\_table\_statement += **'`'** + column\_name + **'` '** + **' boolean'  
 else**:  
 create\_table\_statement += **'`'** + column\_name + **'` '** + **' string'** *# attaching comma based on index of column* **if** i < len(data\_frame.dtypes):  
 create\_table\_statement += **','  
 else**:  
 create\_table\_statement += **')'** create\_table\_statement += **' stored as parquet '  
  
 return** create\_table\_statement  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **"""  
 This method is used to create job and job run  
 :return: jobID  
 """  
 def** create\_job(self):  
 **try**:  
 user\_id = self.user\_id  
 project\_id = self.project\_id  
 project\_name = self.project\_name  
 mail\_id = self.user\_email  
 cookie = self.cookie  
 publish\_logger = self.publish\_logger  
 jobName = self.jobName  
 jobUrl = constants.JOB\_END\_POINT.format(constants.SCHEDULER\_URL)  
 authorization = **"Token {}"**.format(base64.b64decode(constants.JOB\_TOKEN).decode(**'utf-8'**))  
 generateEntityID = uuid.uuid4()  
 entityID = str(generateEntityID)  
 createJobData = {  
 constants.CRON\_EXPR: constants.CRON\_EXPR\_VAL,  
 constants.CRON\_JSON: {},  
 constants.DESCRIPTION: constants.JOB\_DESC,  
 constants.ENTITY\_ID: entityID,  
 constants.ENTITY\_TYPE: constants.WRANGLER,  
 constants.IN\_PARAMS: {},  
 constants.NAME: jobName,  
 constants.PROJECT\_ID: project\_id,  
 constants.PROJECT\_NAME: project\_name,  
 constants.USER\_ID: user\_id  
 }  
 jsonCreateJob = json.dumps(createJobData)  
 publish\_logger.info(**"posting data to create job service data: "** + jsonCreateJob)  
 publish\_logger.info(**"create job URL "** + jobUrl)  
 jobID = **'UNDEFINED'** headers = {constants.AUTHORIZATION: authorization, constants.X\_DATALAB\_USER: user\_id, constants.CONTENT\_TYPE: constants.APP\_JSON,  
 constants.ACCEPT: constants.APP\_JSON}  
 publish\_logger.info(**"Headers "** + str(headers))  
 **try**:  
 res = requests.post(jobUrl, data=jsonCreateJob, headers=headers, cookies=cookie)  
 responseJson = res.json()  
 jobID = responseJson.get(**'job\_id'**)  
 publish\_logger.info(**"response job id"** + jobID)  
 **except** Exception **as** ex:  
 publish\_logger.exception(**'Exception in response from create job : %s'**, ex)  
  
 *# create job run starts here* generateRunId = uuid.uuid1()  
 runID = str(generateRunId)  
 self.runID = runID  
 publish\_logger.info(**"random generated run id:"** + runID)  
 user\_log\_path = self.generate\_user\_log\_path(jobName, jobID, runID)  
 self.user\_log\_path = user\_log\_path  
 logPath = **"{}/{}/{}"**.format(constants.S3\_URL, constants.S3\_BUCKET, user\_log\_path)  
 publish\_logger.info(**"upload log file url:"** + logPath)  
 jobRunURL = jobUrl + **'/'** + jobID + **'/run'** createJobRunData = {  
 constants.COMMENTS: constants.JOB\_COMMENTS,  
 constants.OUT\_PARAMS: {  
 **'LOG\_PATH'**: logPath  
 },  
 constants.RETRY: 0,  
 constants.RUN\_ID: runID,  
 constants.ENV\_CD: constants.ENV\_CD\_VALUE,  
 constants.STATUS\_CD: constants.JOB\_STARTED  
 }  
 jobRunJsonString = json.dumps(createJobRunData)  
 publish\_logger.info(**"posting data to create job run service data:"** + jobRunJsonString)  
 publish\_logger.info(**"create job run URL "** + jobRunURL)  
 **try**:  
 jobRunResp = requests.post(jobRunURL, data=json.dumps(createJobRunData), cookies=cookie,  
 headers=headers)  
 publish\_logger.info(**"job run response"** + str(jobRunResp))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception in response from create job run : %s'**, ex)  
 publish\_logger.info(**"create job run ends"**)  
 **return** jobID  
 **except** Exception **as** ex:  
 publish\_logger.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
 **"""  
 method used to delete the job(active status update)  
 """  
 def** delete\_job(self, jobId):  
 params = {constants.PROJECT: self.project\_id}  
 cookie = self.cookie  
 publish\_logger = self.publish\_logger  
 authorization = **"Token {}"**.format(base64.b64decode(constants.JOB\_TOKEN).decode(**'utf-8'**))  
 jobUrl = constants.JOB\_END\_POINT.format(constants.SCHEDULER\_URL)  
 headers = {constants.AUTHORIZATION: authorization, constants.X\_DATALAB\_USER: self.user\_id,constants.CONTENT\_TYPE: constants.APP\_JSON,  
 constants.ACCEPT: constants.APP\_JSON}  
 jobDeleteUrl = jobUrl + **'/'** + jobId  
 **try**:  
 delResp = requests.delete(jobDeleteUrl, cookies=cookie, headers=headers, params=params)  
 publish\_logger.info(**"job delete response"** + str(delResp))  
 **except** Exception **as** ex:  
 publish\_logger.exception(**'Exception in response from delete job : %s'**, ex)  
  
 **def** get\_email\_id(self, s):  
 **return** s.split(**"@"**)[0]  
  
 **def** prune(self, v):  
 **return ""**.join(e **for** e **in** v **if** e.isalnum())  
  
 **def** truncate(self, e, l):  
 **return** e[:l]  
  
 **def** tiny\_str(self, v, i):  
 **return** self.truncate(self.prune(v), i)  
  
 **"""  
 method used to generate log path based on combination of  
 mail id,project name, job name, job id and job runid.  
 """  
 def** generate\_user\_log\_path(self, job\_name, job\_id, job\_run\_id):  
 publish\_logger = self.publish\_logger  
 project\_name = self.tiny\_str(self.project\_name, 10)  
 job\_name = self.tiny\_str(job\_name, 10)  
 user\_mail = self.tiny\_str(self.get\_email\_id(self.user\_email), 10)  
  
 user\_log\_path = **"{}/{}/{}-{}/{}/"**.format(  
  
 user\_mail,  
 self.project\_name,  
 job\_name,  
 job\_id,  
 job\_run\_id,  
 )  
 publish\_logger.info(**"generating user log path:"**+user\_log\_path)  
 **return** user\_log\_path  
  
 **"""  
 method used to write publish logs into s3 using boto3 and  
 removing application log  
 """  
 def** writingLogsIntoS3(self):  
 mashup\_api\_path = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))  
 publish\_logger = self.publish\_logger  
 logFileName = self.jobName  
 logFilePath = mashup\_api\_path + **"/"**+logFileName+**".log"** userPath = self.user\_log\_path  
 targetlogFilePath = userPath + logFileName + **".log"** publish\_logger.info(**"s3 file path:"** + targetlogFilePath)  
 publish\_logger.info(**"application log path:"** + logFilePath)  
 boto3.set\_stream\_logger(name=**'botocore'**)  
 client = boto3.resource(service\_name=constants.SERVICE\_NAME, use\_ssl=constants.USE\_SSL, region\_name=constants.S3\_REGION,  
 endpoint\_url=constants.S3\_URL,  
 aws\_access\_key\_id=constants.S3\_ACCESS\_KEY,  
 aws\_secret\_access\_key=constants.S3\_SECRET\_KEY,  
 config=Config(s3={constants.ADDRESSING\_STYLE: constants.PATH},  
 signature\_version=constants.SIGNATURE\_VERSION,  
 read\_timeout=constants.READ\_TIMEOUT,  
 connect\_timeout=constants.CONNECT\_TIMEOUT))  
 client.Object(constants.S3\_BUCKET, targetlogFilePath).upload\_file(logFilePath)  
 *# removing application log file* **if** os.path.exists(logFilePath):  
 os.remove(logFilePath)  
  
 **"""  
 method used to update the job status as completed once publish done  
 """  
 def** updateJobStatus(self):  
  
 **try**:  
 publish\_logger = self.publish\_logger  
 run\_id = self.runID  
 publish\_logger.info(**"run id:::::"**+run\_id)  
 config\_connection = UserPreferences()  
 connection = config\_connection.get\_connection()  
 cursor = connection.cursor()  
 sql = **"UPDATE sc\_job\_run SET status\_cd='JOB\_COMPLETED' WHERE run\_id=:run\_id"** args = {**"run\_id"**: run\_id}  
 cursor.execute(sql) **if** args **is None else** cursor.execute(sql, args)  
 connection.commit()  
 publish\_logger.info(**"updated job status to completed"**)  
  
 **except** Exception **as** ex:  
 publish\_logger.exception(**'Exception in response from update job status : %s'**, ex)  
  
 **finally**:  
 **if** cursor **is not None**:  
 cursor.close()  
 connection.close()

*"""  
This module is used to write pandas dataframe to hive databases  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** sqlalchemy **import** create\_engine  
**import** constants  
**import** message\_constants  
*# LOGGER = logging.getLogger(\_\_name\_\_)  
# LOGGER.setLevel(logging.DEBUG)  
# HANDLER = RotatingFileHandler('pandas\_api.log', maxBytes=20000000, backupCount=10)  
# FORMATTER = logging.Formatter(fmt='%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
# ' %(funcName)s:%(lineno)d')  
# HANDLER.setFormatter(FORMATTER)  
# LOGGER.addHandler(HANDLER)*LOGGER = logging.getLogger(**'lens\_backend'**)  
  
**class** SparkDataFrameWriter(object):  
 *"""  
 This class contains code for write pandas data frame into hive tables.  
 """* @classmethod  
 **def** write\_data\_frame\_to\_hive\_table(cls, data\_frame, table\_name):  
 *"""  
 This method will be used to write record into table* **:return***:  
 """* **try**:  
 engine = SparkDataFrameWriter.create\_connection\_engine()  
 data\_frame.to\_sql(table\_name, con=engine, if\_exists=**'append'**, index=**False**)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** create\_connection\_engine(cls):  
 *"""  
 This method will create a hive connection* **:return***:  
 """* **try**:  
 **return** create\_engine(**'hive://'** + constants.HOST\_NAME + **':'** + constants.PORT + **'/'** + constants.DATABASE, echo=**False**)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.ERROR\_WHILE\_CREATING\_CONNECTION\_WITH\_HIVE)  
  
 @classmethod  
 **def** create\_table\_in\_hive(cls, statement):  
 *"""  
 This method will create table in hive based on input* **:return***:  
 """* **try**:  
 engine = SparkDataFrameWriter.create\_connection\_engine()  
 engine.execute(statement)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

**import** re  
**import** requests  
**import** gbiio  
**import** logging  
**import** datetime  
**from** ldap3 **import** Server, Connection  
**import** os  
**import** constants  
**from** request\_builder **import** RequestBuilder  
**from** token\_registry **import** TokenRegistry  
**import** json  
**import** message\_constants  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
**class** GBIIOClient :  
  
 **def** \_\_init\_\_(self):  
 self.app\_id = os.getenv(constants.DATALAB\_APP\_ID)  
 self.app\_password = os.getenv(constants.DATALAB\_APP\_PASSWORD)  
 self.app\_context = os.getenv(constants.DATALAB\_APP\_CONTEXT)  
 self.catalog\_app\_id = os.getenv(constants.CATALOG\_APP\_ID)  
 self.token\_url = os.getenv(constants.APP\_TOKEN\_GENERATION\_URL)  
  
 **def** generate\_token(self,user\_id):  
 *"""  
 Generate the token for GBIIO service* **:param** *user\_id:* **:return***:token  
 """* **try**:  
 auth\_header = {**'Content-type'**: **'application/json'**}  
 payload = {**"appId"**: str(self.app\_id),**"appPassword"**: str(self.app\_password), **"context"**: str(self.app\_context),**"otherApp"**: str(self.catalog\_app\_id)}  
 response = requests.post(self.token\_url, json=payload, headers=auth\_header)  
 token = str(response.json().get(**'token'**))  
 **return** token  
 **except** Exception **as** ex:  
 LOGGER.error(**"Exception while getting gbiio token %s"**, ex)  
 **raise** ex  
  
 **def** gbiio\_init(self,userid,project\_id):  
 *"""  
 Initialise the gbiio properties.* **:param** *userid: logged in user* **:param** *project\_id: belongs to datalab project.* **:return***: None  
 """* **try**:  
 token = self.generate\_token(userid)  
 gbiio.set\_property(gbiio.constants.Context.CATALOG\_APP\_TOKEN, str(token))  
 \_unix\_id = self.get\_unix\_id(userid)  
 gbiio.set\_property(gbiio.constants.Context.DS\_ID, userid)  
 gbiio.set\_property(gbiio.constants.Context.UNIX\_ID, \_unix\_id)  
 gbiio.set\_property(gbiio.constants.Context.GROUP\_IDS, **''**)  
 gbiio.set\_property(gbiio.constants.Context.PROJECT\_ID, str(project\_id))  
 LOGGER.info(**"Get userid {},unix\_id {} and project\_id {} ."**.format(str(userid),str(\_unix\_id),str(project\_id)))  
 **return** token  
 **except** Exception **as** ex:  
 LOGGER.error(**"Exception while initialise the gbiio %s"**, ex)  
 **raise** Exception(message\_constants.REQUEST\_EXCEPTION\_OCCURED)  
  
  
 **def** write\_data(self, dataframe, filename, project\_id, file\_format, write\_mode):  
 *"""  
 Write dataframe into mcqueen location using gbiio service* **:param** *dataframe: loaded wrangled data.* **:param** *filename: data should be written into mcqueen using this name* **:param** *project\_id: belongs to datalab project.* **:param** *file\_format: format of the file like csv,txt.* **:param** *write\_mode: should be overwrite or not* **:return***:location  
 """* **try**:  
 base\_dir = str(constants.BASE\_DIR).format(str(project\_id))  
 dir\_exists = gbiio.isdir(base\_dir, is\_abspath=**True**)  
 **if not** dir\_exists:  
 gbiio.makedirs(base\_dir, is\_abspath=**True**)  
 LOGGER.info(**"Directory is being created"**)  
 save\_location = base\_dir + **"/"** + filename  
 mode = write\_mode == **'overwrite'** gbiio.put(dataframe, path=save\_location, format=file\_format, override=mode, is\_abspath=**True**)  
 **return** save\_location  
 **except** Exception **as** ex:  
 LOGGER.error(**"Exception while writing file into mcqueen location %s"**, ex)  
 **raise** Exception(message\_constants.GBIIO\_WRITE\_FAILED)  
  
 **def** read\_data(self,abs\_path\_file,file\_format,row\_limit):  
 *"""  
 Read dataframe from mcqueen location using gbiio service* **:param** *abs\_path\_file: absolute location to read the stored file.* **:return***:dataframe  
 """* **try**:  
 LOGGER.info(**f"Getting data: {**abs\_path\_file**}"**)  
 **return** gbiio.get(abs\_path\_file, format=file\_format, nrows=row\_limit, is\_abspath=**True**)  
 **except** Exception **as** exp:  
 LOGGER.error(**"Exception while reading the file from mcqueen %s"**, exp)  
 **raise** Exception(message\_constants.GBIIO\_READ\_FAILED)  
  
 **def** publish\_data(self,dataframe,abs\_path\_file,tbl\_name,user\_id,data\_source\_id):  
 *"""  
 Publish sample dataset into oracle metastore to discover the published dataset.* **:param** *dataframe:* **:param** *abs\_path\_file: should be able to find wrangled data using absolute file path* **:param** *tbl\_name: To identify published data from discover search.* **:param** *usr\_name: logged in user  
 """* **try**:  
 field\_list = self.get\_Field\_list(dataframe)  
 datasource = self.get\_datasource(abs\_path\_file, field\_list, tbl\_name, user\_id, data\_source\_id)  
 authorization\_header = TokenRegistry.get\_token\_for\_user(user\_id)  
  
 *# publish the dataset into metastore.* RequestBuilder().call\_http\_post\_request(constants.GBIIO\_DATASOURCE\_CREATE, datasource,  
 authorization\_header)  
 LOGGER.info(**"Published successfully and table name is {}"**.format(tbl\_name))  
 **except** Exception **as** exp:  
 LOGGER.error(**"Exception while publishing the dataset into metastore %s"**, exp)  
 **raise** Exception(message\_constants.GBIIO\_PUBLISH\_FAILED)  
  
 **def** get\_Field\_list(self,dataframe):  
 *"""  
 Construct the fields list to store into metastore.* **:param** *dataframe: loaded wrangled data.* **:return***:field\_list  
 """* **try**:  
 col\_index = 0  
 field\_list = []  
 **for** column\_name, column\_data\_type **in** dataframe.dtypes.items():  
 **if**(col\_index > 0):  
 field\_list.append({  
 **"name"**: column\_name,  
 **"position"**: col\_index,  
 **"fieldDataType"**: self.get\_Data\_type(column\_data\_type),  
 **"sampleRecords"**: dataframe[column\_name].tolist()  
 })  
 col\_index += 1  
 **return** field\_list  
 **except** Exception **as** exp:  
 LOGGER.error(**"Exception getting field list %s"**, exp)  
  
 **def** get\_Data\_type(self,col\_dtype):  
 *"""  
 Identify the column type* **:param** *col\_dtype: identify the type of the column.* **:return***: column type  
 """* type = **'STRING'  
 if**(col\_dtype == **'int64'**):  
 type = **'INTEGER'  
 elif** (col\_dtype == **'float64'**):  
 type = **'DECIMAL'  
 elif** (col\_dtype == **'object'**):  
 type = **'STRING'  
 elif** (col\_dtype == **'bool'**):  
 type = **'STRING'  
 return** type  
  
 **def** get\_datasource(self,abs\_path\_file,field\_list,tbl\_name,usr\_name,data\_source\_id):  
 *"""  
 Construct the datasource info to discover the published data.* **:param** *abs\_path\_file: stored location from mcqueen.* **:param** *field\_list: Name of the field list for publish.* **:param** *tbl\_name: identify the published dataset.* **:param** *usr\_name: logged in user.* **:return***:datasource  
 """* **try**:  
 data\_source\_details = RequestBuilder().call\_http\_get\_request(  
 constants.CATALOG\_DISCOVER\_MANAGER\_SERVER +  
 constants.CATALOG\_GET\_DATASOURCE + str(data\_source\_id), {}, {})  
 data\_source\_details = json.loads(data\_source\_details.content.decode(**'utf-8'**))  
 connection\_id = data\_source\_details.get(**'connectionId'**, **'connection\_id'**)  
 **except** Exception **as** ex:  
 LOGGER.error(**"Exception while fetching connection\_id from datasource %s"**, ex)  
 datasource = {  
 **"connectionId"**: str(connection\_id),  
 **"dsUniqueId"**: tbl\_name,  
 **"groupId"**: **"mosaic"**,  
 **"createdBy"**: usr\_name,  
 **"expertUserId"**: usr\_name,  
 **"name"**: tbl\_name,  
 **"ismodified"**: **"1"**,  
 **"sourceSystem"**: **"LENS"**,  
 **"sourceType"**: **"Object"**,  
 **"subSourceType"**: **"McQueen"**,  
 **"dataSourceType"**: **"McQueen"**,  
 **"isExternalProject"**: **"false"**,  
 **"listOfFieldMapping"**: field\_list,  
 **"dataNode"**: {  
 **"label"**: tbl\_name,  
 **"isPublished"**: **"false"**,  
 **"children"**: [],  
 **"value"**: {  
 **"container"**: **""**,  
 **"isDsNode"**: **"true"**,  
 **"bucketName"**: **"batch-jobs"**,  
 **"storagePath"**: abs\_path\_file,  
 **"fileType"**: **"csv"**,  
 **"isPublished"**: **"false"**,  
 **"rowTag"**: **""**,  
 **"delimiter"**: **""**,  
 **"dbName"**: **""**,  
 **"isHeader"**: **"true"**,  
 **"escape"**: **""**,  
 **"quote"**: **""**,  
 **"ignore\_leading\_white\_spaces"**: **"false"**,  
 **"skip\_lines"**: **""**,  
 **"strict\_quotes"**: **""** }  
 }  
 }  
 **return** datasource  
  
 **def** get\_unix\_id(self, user\_id):  
 *"""  
 Get unix\_id for the user\_id.* **:param** *user\_id:logged in user.* **:return***:unix\_id  
 """* **try**:  
 \_unix\_id = **None** regex = re.compile(**"uid=([a-z0-9]+)"**)  
 prs\_id = user\_id  
 search\_base = **"ou={},dc={},dc={},dc={},dc={}"**.format(constants.OU, constants.GCSLDAP, constants.CORP,  
 constants.APPLE, constants.COM)  
 search\_filter = **"(&(appleDSID={}))"**.format(prs\_id)  
  
 *# fetch badge/unix id* server = Server(constants.LDAP\_SERVER, use\_ssl=**True**)  
 **with** Connection(  
 server,  
 user=**'uid={},ou={},dc={},dc={},dc={},dc={}'**.format(constants.UID, constants.OU, constants.GCSLDAP,  
 constants.CORP, constants.APPLE, constants.COM),  
 password=constants.LDAP\_PASSWORD,  
 ) **as** connection:  
 connection.search(search\_base, search\_filter, attributes=[**"uid"**])  
 **if** connection.entries:  
 ldap\_record = str(connection.entries[0])  
 \_unix\_id = regex.search(ldap\_record).group(1)  
 **return** \_unix\_id  
 **except** Exception **as** exp:  
 LOGGER.error(**"Exception while fetching unix id %s"**, exp)

*"""  
This module is used to test batch job api  
"""***import** unittest  
**from** unittest.mock **import** Mock, patch  
**import** json  
**import** pandas **as** pd  
**from** batch\_job.batch\_job\_runner **import** JobRunner  
**from** batch\_job.batch\_job\_runner **import** SparkDataFrameWriter  
**from** request\_builder **import** RequestBuilder  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestJobRunner(unittest.TestCase):  
 *"""  
 Class for run batch job test case  
 """* **def** setUp(self):  
 self.create\_table = SparkDataFrameWriter.create\_table\_in\_hive  
 self.write\_table = SparkDataFrameWriter.write\_data\_frame\_to\_hive\_table  
 self.input\_json = {**"data\_source\_id"**: 2, **"lens\_id"**: 10, **"dataSourceName"**: **"test\_tables"**,  
 **"user\_id"**: 10, **"load\_type"**: **"cluster\_summary"**,  
 **"json\_value"**: {**'Cluster\_0'**: {**'energy\_output'**: {**'std'**: 9.15, **'max'**: 495,  
 **'50%'**: 470.93,  
 **'min'**: 448.97,  
 **'mean'**: 471.56},  
 **'temperature'**: {**'std'**: 3.83, **'max'**: 24.26,  
 **'50%'**: 12.53, **'min'**: 1.81,  
 **'mean'**: 12.33}},  
 **'Cluster\_1'**: {**'energy\_output'**: {**'std'**: 7.64, **'max'**: 466,  
 **'50%'**: 443, **'min'**: 421,  
 **'mean'**: 443.26},  
 **'temperature'**: {**'std'**: 3.32, **'max'**: 32.3,  
 **'50%'**: 23, **'min'**: 6.22,  
 **'mean'**: 23.04}}}}  
 self.lens\_json = {**'datasourceOutputId'**: **'10'**, **'description'**: **'Default Description'**,  
 **'createdDate'**: **'2018-09-20T07:43:16.000+0000'**,  
 **'createdBy'**: **'Prasang'**, **'subCategory'**: **'Lens'**,  
 **'category'**: **'MashupTest'**, **'status'**: **'Draft'**,  
 **'updatedDate'**: **'2018-09-24T13:23:17.504+0000'**,  
 **'tags'**: **'Account'**, **'lensId'**: 10,  
 **'datasourceInputId'**: **'2'**, **'lensType'**: **None**,  
 **'lensName'**: **'Lens\_Name1537429393'**}  
 filename = **'../resources/batch\_job.json'  
 with** open(filename, **'r'**) **as** file\_reader:  
 self.batch\_job\_json = json.load(file\_reader)  
  
 filename = **'../resources/datasource.json'  
 with** open(filename, **'r'**) **as** file\_reader:  
 self.data\_source\_json = json.load(file\_reader)  
  
 **def** tearDown(self):  
 SparkDataFrameWriter.create\_table\_in\_hive = self.create\_table  
 SparkDataFrameWriter.write\_data\_frame\_to\_hive\_table = self.write\_table  
  
 **def** test\_batch\_job(self):  
 *"""  
 test suite for running batch job* **:return***:  
 """* **try**:  
 **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** lens\_get\_request:  
 lens\_get\_request.return\_value.json.return\_value = self.batch\_job\_json  
 **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_source\_creator:  
 data\_source\_creator.return\_value = \  
 pd.read\_csv(**"../resources/10Records.txt"**,  
 delimiter=**'\t'**, parse\_dates=[**'field0'**])  
  
 SparkDataFrameWriter.create\_table\_in\_hive = Mock()  
 SparkDataFrameWriter.write\_data\_frame\_to\_hive\_table = Mock()  
 **with** patch.object(RequestBuilder,  
 **"call\_http\_post\_request"**) **as** lens\_post\_request:  
 lens\_post\_request.return\_value.json.return\_value = self.data\_source\_json  
 **with** patch.object(RequestBuilder,  
 **"call\_http\_put\_request"**) **as** lens\_put\_request:  
 lens\_put\_request.return\_value.json.return\_value = self.lens\_json  
 runner = JobRunner(self.input\_json)  
 runner.run\_batch\_job()  
 self.assertEqual(1, 1)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 2)  
  
 **def** test\_batch\_job\_key\_missing(self):  
 *"""  
 test job if input json has missing some key* **:return***:  
 """* **try**:  
 self.input\_json.pop(**"data\_source\_id"**)  
 JobRunner(self.input\_json)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_create\_table\_statement(self):  
 *"""  
 Test suite is used to create table statement from given data\_frame* **:return***:  
 """* data\_frame = pd.read\_csv(**"../resources/new.txt"**, delimiter=**'\t'**, parse\_dates=[**'field20'**])  
 create\_statement = JobRunner.generate\_create\_table\_statement(data\_frame, **"test\_table\_name"**)  
 **if** create\_statement **is not None**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_batch\_job\_error(self):  
 *"""  
 test suite for running batch job* **:return***:  
 """* **try**:  
 **with** patch.object(RequestBuilder,  
 **"call\_http\_get\_request"**) **as** lens\_get\_request\_error:  
 lens\_get\_request\_error.return\_value.json.return\_value = self.batch\_job\_json  
 **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_source\_creator:  
 data\_source\_creator.return\_value = \  
 pd.read\_csv(**"../resources/10Records.txt"**,  
 delimiter=**'\t'**, parse\_dates=[**'field0'**])  
 runner = JobRunner(self.input\_json)  
 runner.run\_batch\_job()  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_job\_with\_request\_error(self):  
 *"""  
 Test suite for if error in request for creating drive data\_source* **:return***:  
 """* **try**:  
 **with** patch.object(RequestBuilder,  
 **"call\_http\_get\_request"**) **as** lens\_get\_request\_error:  
 lens\_get\_request\_error.return\_value.json.return\_value = self.batch\_job\_json  
 **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_source\_creator:  
 data\_source\_creator.return\_value = \  
 pd.read\_csv(**"../resources/10Records.txt"**,  
 delimiter=**'\t'**, parse\_dates=[**'field0'**])  
 SparkDataFrameWriter.create\_table\_in\_hive = Mock()  
 SparkDataFrameWriter.write\_data\_frame\_to\_hive\_table = Mock()  
 **with** patch.object(RequestBuilder,  
 **"call\_http\_post\_request"**) **as** lens\_post\_request:  
 lens\_post\_request.return\_value.json.return\_value = {  
 **"timestamp"**: **"2018-10-17T12:04:10.056+0000"**,  
 **"status"**: 500,  
 **"error"**: **"Internal Server Error"**,  
 **"message"**: **"Error Mesage while creating object"**,  
 **"path"**: **"/data/saveDataSource/"**}  
 runner = JobRunner(self.input\_json)  
 runner.run\_batch\_job()  
 self.assertEqual(1, 1)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 2)  
  
 **def** test\_job\_update\_lens\_failed(self):  
 *"""  
 test suite of running batch job if error while updating datasource* **:return***:  
 """* **try**:  
 **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** lens\_get\_request:  
 lens\_get\_request.return\_value.json.return\_value = self.batch\_job\_json  
 **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_source\_creator:  
 data\_source\_creator.return\_value = \  
 pd.read\_csv(**"../resources/10Records.txt"**,  
 delimiter=**'\t'**, parse\_dates=[**'field0'**])  
 SparkDataFrameWriter.create\_table\_in\_hive = Mock()  
 SparkDataFrameWriter.write\_data\_frame\_to\_hive\_table = Mock()  
 **with** patch.object(RequestBuilder,  
 **"call\_http\_post\_request"**) **as** lens\_post\_request:  
 lens\_post\_request.return\_value.json.return\_value = self.data\_source\_json  
 **with** patch.object(RequestBuilder,  
 **"call\_http\_put\_request"**) **as** lens\_put\_request:  
 lens\_put\_request.return\_value.json.return\_value = {  
 **"timestamp"**: **"2018-10-17T13:25:22.405+0000"**,  
 **"status"**: 404,  
 **"error"**: **"Not Found"**,  
 **"message"**: **"Lens not found with id : '12213123123123'"**,  
 **"path"**: **"/insight-hub/lens-management/managed-lens/"**}  
 runner = JobRunner(self.input\_json)  
 runner.run\_batch\_job()  
 self.assertEqual(1, 1)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 2)

**import** unittest  
**from** batch\_job.data\_frame\_writer **import** SparkDataFrameWriter  
  
  
**class** TestDataFrameWriter(unittest.TestCase):  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
 def** test\_negative(self):  
 *"""Negative Test Cases for data\_frame writer"""* **with** self.assertRaises(Exception):  
 SparkDataFrameWriter.create\_table\_in\_hive(**None**)  
  
 **with** self.assertRaises(Exception):  
 SparkDataFrameWriter.write\_data\_frame\_to\_hive\_table(**None**, **None**)

column\_selector

*"""  
Selects columns for the dataframe  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** ColumnSelector:  
 *"""  
 Args used are:  
 column\_list: Column name list which needs to be selected  
 """* **def** \_\_init\_\_(self, json\_value):  
 *"""  
 Initialize* **:param** *json\_value:  
 """* **try**:  
 self.column\_list = json\_value[**'column\_list'**]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 Args:  
 data\_frame: data\_frame on which columns are selected.  
 This method is exposed into the api which selects the given columns  
 """* **try**:  
 **if** set(self.column\_list) <= set(data\_frame.columns.tolist()):  
 **pass  
 else**:  
 **raise** ValueError(message\_constants.COLUMN\_NOT\_FOUND)  
 **return** data\_frame[self.column\_list]  
 **except** MemoryError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INSUFFICIENT\_MEMORY\_FOR\_PERFORMING\_OPERATION)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to check unit test case for ColumnSelector  
"""***import** unittest  
**import** pandas **as** pd  
**from** column\_selector **import** ColumnSelector  
  
  
**class** TestColumnSelector(unittest.TestCase):  
 *"""  
 This class is used for implement test method of auto features step  
 """* **def** setUp(self):  
 self.data\_frame1 = pd.read\_csv(**'../resources/customers.txt'**, delimiter=**','**)  
 self.json\_value = {**'column\_list'**: [**'customer\_id'**]}  
  
 **def** test\_column\_selector(self):  
 *"""  
 used to test column selector* **:return***:  
 """* join\_step = ColumnSelector(self.json\_value)  
 data\_frame1 = join\_step.get\_result(self.data\_frame1)  
 **if** len(data\_frame1.columns.tolist()) <= len(self.json\_value[**'column\_list'**]):  
 self.assertEqual(**True**, **True**)  
 **else**:  
 self.assertEqual(**True**, **False**)  
  
 **def** test\_column\_selector\_value\_error(self):  
 *"""  
 used to test column selector with wrong column name* **:return***:  
 """* json\_value = {**'column\_list'**: [**'redfd'**]}  
 **with** self.assertRaises(Exception):  
 join\_step = ColumnSelector(json\_value)  
 data\_frame1 = join\_step.get\_result(self.data\_frame1)  
 **if** len(data\_frame1.columns.tolist()) <= len(json\_value[**'column\_list'**]):  
 **pass  
  
 def** test\_column\_selector\_keyerror(self):  
 *"""  
 used to test column selector with key error , column\_list is not present* **:return***:  
 """* json\_value = {**'column\_list1'**: [**'customer\_id'**]}  
 **with** self.assertRaises(Exception):  
 join\_step = ColumnSelector(json\_value)  
 data\_frame1 = join\_step.get\_result(self.data\_frame1)  
 **if** len(data\_frame1.columns.tolist()) <= len(json\_value[**'column\_list'**]):  
 **pass  
  
 def** test\_column\_selector\_generic\_exception(self):  
 *"""  
 used to test column selector with key error* **:return***:  
 """* **with** self.assertRaises(Exception):  
 json\_value = {**'column\_list'**: **'error\_string'**}  
 join\_step = ColumnSelector(json\_value)  
 data\_frame1 = join\_step.get\_result(self.data\_frame1)  
 **if** len(data\_frame1.columns.tolist()) <= len(json\_value[**'column\_list'**]):  
 **pass**

Dataframe\_reader

*"""  
This is the abstract class for all the reader  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** abc **import** ABCMeta, abstractmethod  
**import** constants  
**import** message\_constants  
**from** json\_parser\_utility **import** JsonUtility  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** DataFrameReader(metaclass=ABCMeta):  
 *"""  
 Super class of python  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 utility = JsonUtility()  
 self.field\_names = utility.required\_key\_value\_from\_json(constants.FIELD\_NAME,  
 input\_json[constants.FIELD\_DETAILS])  
  
 field\_with\_type = utility.required\_key\_value\_from\_json \  
 (constants.FIELD\_TYPE, input\_json[constants.FIELD\_DETAILS])  
  
 self.field\_data\_types = {}  
 **for** i **in** range(0, len(self.field\_names)):  
 self.field\_data\_types.update(  
 {self.field\_names[i]: DataFrameReader.check\_assign\_pandas\_type(field\_with\_type[i])})  
  
 self.date\_data\_fields = utility.find\_value\_of\_matched\_key\_value(  
 constants.FIELD\_NAME, constants.FIELD\_TYPE, constants.DATE, input\_json)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** check\_assign\_pandas\_type(cls, value):  
 *"""  
 This method is used to convert java datatype to pandas datatype* **:param** *value:* **:return***:  
 """* **try**:  
 **if** value.lower() **in** [**'date'**, **'timestamp'**, **'string'**]:  
 **return 'object'  
 elif** value.lower() == **'integer'**:  
 **return 'int64'  
 elif** value.lower() == **'double' or** value.lower() == **'float'**:  
 **return 'float64'  
 return 'object'  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @abstractmethod  
 **def** read\_from\_configuration(self, load\_all\_flag):  
 *"""  
 This is the abstract method of dataframe reader.* **:return***:  
 """* **pass**

*"""  
This module is used for create dataFrame from file  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** pandas **as** pd  
**from** dataframe\_reader **import** DataFrameReader  
**import** constants  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** FileReader(DataFrameReader):  
 *"""  
 This class is used for create dataframe from file.  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 DataFrameReader.\_\_init\_\_(self, input\_json)  
 self.delimiter = input\_json[**'config'**][constants.DELIMITER]  
 self.file\_path = input\_json[constants.FILE\_PATH]  
 self.contain\_header = 0 **if** input\_json[**'config'**][constants.CONTAINS\_HEADER] **is True** \  
 **else None  
 except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** read\_from\_configuration(self, load\_all\_flag):  
 *"""  
 This method is used to create dataframe from file with some configuration* **:return***:  
 """* **try**:  
 **return** pd.read\_csv(self.file\_path, delimiter=self.delimiter,  
 header=self.contain\_header,  
 names=self.field\_names, dtype=self.field\_data\_types  
 , skip\_blank\_lines=**True**,  
 parse\_dates=self.date\_data\_fields)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""Load registry object  
"""***import** os  
**import** logging  
**import** copy  
**import** json  
**import** datetime  
**import** threading  
**import** constants  
**import** message\_constants  
**from** class\_creator **import** ClassCreator  
**from** request\_builder **import** RequestBuilder  
**from** spark\_dataframe\_reader **import** SparkReader  
**from** samples\_strategy.top\_n **import** TopNStrategy  
**from** token\_registry **import** TokenRegistry  
**from** connector.mosaicio **import** MosaicioConnector  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
**import** pandas **as** pd  
  
  
**class** DataRegistry:  
 *"""  
 This class is used for register and fetch stored Data Frame.  
 """* pandas\_df\_map = {}  
  
 @classmethod  
 **def** get\_data\_set\_from\_dict(cls, data\_source\_id, lens\_id, data\_frame\_type):  
 *"""  
 Fetch dataFrame from dict* **:return***:  
 """* **try**:  
  
 data\_frames = DataRegistry.pandas\_df\_map.get(DataRegistry.create\_register\_id  
 (data\_source\_id, lens\_id))  
 **if** data\_frames **is None**:  
 **raise** Exception(**'DataSource does not exist with name '** + str(data\_source\_id))  
 **else**:  
 DataRegistry.update\_last\_user\_time(data\_source\_id=data\_source\_id, lens\_id=lens\_id)  
 threading.Thread(target=DataRegistry.remove\_data\_frame\_from\_registry).start()  
 **return** data\_frames[data\_frame\_type]  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** update\_frame(cls, data\_source\_id, lens\_id, data\_frame, data\_frame\_type):  
 *"""  
 This method is used for update the data set with specific type* **:param** *data\_source\_id:* **:param** *lens\_id:* **:param** *data\_frame:* **:param** *data\_frame\_type:* **:return***:  
 """* **try**:  
 DataRegistry.pandas\_df\_map.get(  
 DataRegistry.create\_register\_id(data\_source\_id, lens\_id))[data\_frame\_type] \  
 = data\_frame  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** update\_frame\_strategy(cls, data\_source\_id, lens\_id, strategy\_name, strategy\_value):  
 *"""  
 This method is used for update the data set for strategy values and name* **:param** *data\_source\_id:* **:param** *lens\_id:* **:param** *strategy\_name:* **:param** *strategy\_value:* **:return***: null  
 """* **try**:  
 DataRegistry.pandas\_df\_map.get(  
 DataRegistry.create\_register\_id(data\_source\_id, lens\_id))[constants.STRATEGY\_NAME] \  
 = strategy\_name  
 DataRegistry.pandas\_df\_map.get(  
 DataRegistry.create\_register\_id(data\_source\_id, lens\_id))[  
 constants.STRATEGY\_VALUE] \  
 = strategy\_value  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** register\_data\_frame(cls, data\_source\_id, lens\_id, data\_frame):  
 *"""  
 Store add in the dictionary* **:return***:  
 """* **try**:  
 original\_copy = copy.deepcopy(data\_frame)  
 DataRegistry.pandas\_df\_map.update({DataRegistry  
 .create\_register\_id(data\_source\_id, lens\_id)  
 : {**"ORIGINAL"**: original\_copy,  
 **"WORKING"**: data\_frame,  
 constants.LAST\_USED\_BY:  
 DataRegistry.convert\_datetime\_into\_ms(),  
 constants.STRATEGY\_NAME: **"top\_n"**,  
 constants.STRATEGY\_VALUE: 100000  
 }  
 })  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** check\_data\_frame\_is\_register(cls, data\_source\_id, lens\_id):  
 *"""  
 check data\_frame is already register with name data\_source\_id and lens\_id* **:param** *data\_source\_id:* **:param** *lens\_id:* **:return***:  
 """* **try**:  
 **if** DataRegistry.pandas\_df\_map.get(  
 DataRegistry.create\_register\_id(data\_source\_id, lens\_id)) **is None**:  
 **return False  
 return True  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** create\_register\_id(cls, data\_source\_id, lens\_id):  
 *"""  
 This method is used to create id for dictonary* **:param** *data\_source\_id:* **:param** *lens\_id:* **:return***:  
 """* **try**:  
 **return** str(data\_source\_id) + **"\_"** + str(lens\_id)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** keep\_original\_as\_working(cls, data\_source\_id, lens\_id):  
 *"""  
 This method is used for change working dataframe from latest dataframe* **:param** *data\_source\_id:* **:param** *lens\_id:* **:return***:  
 """* **try**:  
 DataRegistry.pandas\_df\_map.get(  
 DataRegistry.create\_register\_id(data\_source\_id, lens\_id))[**'WORKING'**] = \  
 copy.deepcopy(DataRegistry.pandas\_df\_map.get(  
 DataRegistry.create\_register\_id(data\_source\_id, lens\_id))[**'ORIGINAL'**])  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
 @classmethod  
 **def** sample\_record\_from\_data\_registry(cls, data\_source\_id, lens\_id,  
 data\_frame\_type, record\_count):  
 *"""  
 This method will give the sample record* **:param** *data\_source\_id:* **:param** *lens\_id:* **:param** *data\_frame\_type:* **:param** *record\_count* **:return***:  
 """* **try**:  
 json\_value = json.loads(DataRegistry.pandas\_df\_map.get(DataRegistry.create\_register\_id(  
 data\_source\_id, lens\_id))[data\_frame\_type].head(record\_count).to\_json(orient=**'table'**))  
 field\_detail = **None  
 if** json\_value **is not None**:  
 **if** json\_value[**'data'**]:  
 field\_detail = json\_value[**'data'**][0]  
 **else**:  
 field\_detail = DataRegistry.generate\_sample\_values(  
 json\_value[constants.DATASET\_SCHEMA])  
  
 **return** field\_detail  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
 @classmethod  
 **def** generate\_sample\_values(cls, schema\_details):  
 *"""  
 This method is used to generate sample value based on type* **:return***:  
 """* **try**:  
 field\_details = {}  
 **for** json\_value **in** schema\_details[constants.FIELDS]:  
 field\_details.update({json\_value[constants.DATASET\_FIELD\_NAME]:  
 DataRegistry.create\_value\_field\_type(json\_value[constants.TYPE])})  
 **return** field\_details  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
 @classmethod  
 **def** create\_value\_field\_type(cls, data\_type):  
 *"""  
 This method will use to generate sample value based on field type* **:param** *data\_type:* **:return***:  
 """* **try**:  
 **if** data\_type == constants.DATE\_TIME:  
 **return** datetime.datetime(2018, 9, 11, 21, 11, 34)  
 **if** data\_type == constants.NUMBER:  
 **return** 10.11  
 **if** data\_type == constants.INTEGER:  
 **return** 111  
  
 **return "DummyData"  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# pylint: disable=R0201  
 # Added default value for publish* **def** create\_data\_frame\_using\_data\_source\_id(self,  
 data\_source\_id,  
 load\_all\_flag,  
 user\_id,  
 publish=**False**,  
 samples\_strategy=**''**,  
 project\_id=1):  
 *"""  
 This method will take input as data\_source\_id and return pandas data frame* **:param** *data\_source\_id:* **:param** *load\_all\_flag* **:param** *user\_id* **:param** *samples\_strategy: json input- optional* **:return***:  
 """* **try**:  
 *# temp code for demo* os.environ[**"PROJECTID"**] = str(project\_id)  
 os.environ[**"UID"**] = user\_id  
 os.environ[**"CONNECTOR\_BASE\_URL"**] = constants.CATALOG\_CONNECTIONS\_MANAGER\_SERVER + \  
 constants.CATALOG\_GET\_CONNECTION\_CONFIG  
 data\_source\_details = RequestBuilder().call\_http\_get\_request(  
 constants.CATALOG\_DISCOVER\_MANAGER\_SERVER +  
 constants.CATALOG\_GET\_DATASOURCE + str(data\_source\_id), {}, {})  
 data\_source\_details = json.loads(data\_source\_details.content.decode(**'utf-8'**))  
   
 **if not** data\_source\_details:  
 **raise** data\_source\_name = data\_source\_details.get(**'name'**, **'default\_datasource\_name'**)  
 LOGGER.info(**"USER\_ID in mashup "**+user\_id)  
 ioConnector = MosaicioConnector()  
 dataset = ioConnector.getPandasDataFrame(data\_source\_name,publish)  
 **if** samples\_strategy != **''**:  
 strategy\_value = int(samples\_strategy.get(constants.STRATEGY\_VALUE))  
 dataset = dataset.head(strategy\_value)  
 **return** dataset  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** apply\_individual\_strategy(cls, data\_source\_json, strategy\_json):  
 *"""  
 This method will be used to apply step on dataframe.* **:return***:  
 """* **try**:  
 instance = ClassCreator.get\_class(strategy\_json[constants.STRATEGY\_NAME])(strategy\_json)  
 **return** instance.generate\_dataframe(data\_source\_json)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
 @classmethod  
 **def** convert\_datetime\_into\_ms(cls):  
 *"""  
 This method is used to convert datetime value into millisecond* **:return***:  
 """* **try**:  
 date\_time = datetime.datetime.now()  
 **return** date\_time.timestamp() \* 1000  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
 @classmethod  
 **def** update\_last\_user\_time(cls, data\_source\_id, lens\_id):  
 *"""  
 This method is used to update timestamp of last\_used\_by value of data\_registry* **:param** *data\_source\_id:* **:param** *lens\_id:* **:return***:  
 """* **try**:  
 DataRegistry.pandas\_df\_map.get(  
 DataRegistry.create\_register\_id(data\_source\_id, lens\_id))[constants.LAST\_USED\_BY] =\  
 DataRegistry.convert\_datetime\_into\_ms()  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
 @classmethod  
 **def** find\_minute\_between\_two\_dates(cls, first\_date, second\_date):  
 *"""  
 This method is used to find minute difference between two dates.* **:param** *first\_date:* **:param** *second\_date:* **:return***:  
 """* **try**:  
 elapsed\_time = first\_date - second\_date  
 **return** divmod(elapsed\_time.total\_seconds(), 60)[0]  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
 @classmethod  
 **def** remove\_data\_frame\_from\_registry(cls):  
 *"""  
 This method is used to remove data frame from Registry  
 which is not using anymore within particular time interval.* **:return***:  
 """* **try**:  
 **for** key **in** DataRegistry.pandas\_df\_map.copy():  
 minute\_diff = DataRegistry.find\_minute\_between\_two\_dates(  
 datetime.datetime.now(),  
 datetime.datetime.fromtimestamp(  
 DataRegistry.pandas\_df\_map.get(key)[constants.LAST\_USED\_BY] / 1000.0)  
 )  
 **if** minute\_diff >= constants.EXPIRE\_TIME\_INTERVAL:  
 DataRegistry.pandas\_df\_map.pop(key)  
  
 **import** gc  
 gc.collect()  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
 @classmethod  
 **def** remove\_data\_frame\_from\_registry\_manually(cls, key):  
 *"""  
 This method is used to remove data frame from Registry based on data\_source\_id & lens\_id* **:return***:  
 """* **try**:  
 **if** key **in** DataRegistry.pandas\_df\_map:  
 DataRegistry.pandas\_df\_map.pop(key)  
 **import** gc  
 gc.collect()  
 **else**:  
 **raise** Exception(message\_constants.DATAFRAME\_DOES\_NOT\_EXIST)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used for create dataFrame from spark  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** json  
**from** pyhive **import** hive  
**import** pandas **as** pd  
**import** message\_constants  
**from** dataframe\_reader.dataframe\_reader **import** DataFrameReader  
**import** constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** SparkReader(DataFrameReader):  
 *"""  
 This class is used to create dataframe from spark table.  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 This method is used for setup the spark thirft configuration  
 like connection string , table name* **:param** *input\_json:  
 """* **try**:  
 **if** isinstance(input\_json, str):  
 input\_json = json.loads(input\_json)  
 DataFrameReader.\_\_init\_\_(self, input\_json)  
 self.table\_name = input\_json[constants.DATA\_SOURCE\_NAME]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** read\_from\_configuration(self, load\_all\_flag):  
 *"""  
 This method is used for create pandas dataframe from spark thirft server table.* **:return***:  
 """* **try**:  
 conn = hive.Connection(host=constants.HOST\_NAME, port=constants.PORT,  
 username=constants.USER\_NAME,  
 database=constants.DATABASE)  
  
 **return** pd.read\_sql(**'select \* from '** + self.table\_name +  
 (**' ' if** load\_all\_flag **else ' limit 100000'**),  
 con=conn, parse\_dates=self.date\_data\_fields,  
 columns=self.field\_names)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used for test case data frame reader with different configuration  
  
"""***import** json  
**import** unittest  
**from** mock **import** Mock  
**import** pandas **as** pd  
**from** spark\_dataframe\_reader **import** SparkReader  
**from** file\_dataframe **import** FileReader  
**from** dataframe\_reader **import** DataFrameReader  
  
  
**class** TestDataFrameReader(unittest.TestCase):  
 *"""  
 This class is used of test case suit for create dataframe from different configuration  
 """* **def** setUp(self):  
 **with** open(**'../resources/datasource.json'**) **as** file\_reader:  
 self.input\_json = json.load(file\_reader)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_find\_pandas\_data\_type(self):  
 *"""  
 This test suite for method of check\_assign\_pandas\_type of DataFrameReader class* **:return***:  
 """* object\_value = DataFrameReader.check\_assign\_pandas\_type(**"float"**)  
 **if** object\_value == **'float64'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 object\_value = DataFrameReader.check\_assign\_pandas\_type(**"DOUBLE"**)  
 **if** object\_value == **'float64'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 object\_value = DataFrameReader.check\_assign\_pandas\_type(**"date"**)  
 **if** object\_value == **'object'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 object\_value = DataFrameReader.check\_assign\_pandas\_type(**"TIMESTAMP"**)  
 **if** object\_value == **'object'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 object\_value = DataFrameReader.check\_assign\_pandas\_type(**"STRING"**)  
 **if** object\_value == **'object'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 object\_value = DataFrameReader.check\_assign\_pandas\_type(**"INTEGER"**)  
 **if** object\_value == **'int64'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_reader\_from\_file(self):  
 *"""  
 This test suite is used read from file and create dataframe in panas* **:return***:  
 """* self.input\_json[**'filePath'**] = **"../resources/deliveries.csv"** file\_reader = FileReader(self.input\_json)  
 data\_frame = file\_reader.read\_from\_configuration(load\_all\_flag=**False**)  
 **if** data\_frame **is not None**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_reader\_with\_mock(self):  
 *"""  
 Test suite is used using mocking* **:return***:  
 """* spark\_reader = SparkReader(self.input\_json)  
 spark\_reader.read\_from\_configuration = Mock()  
 spark\_reader.read\_from\_configuration.return\_value = \  
 pd.read\_csv(**"../resources/deliveries.csv"**)  
 variable = spark\_reader.read\_from\_configuration(**False**)  
 **if** len(variable) == 136598:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_reader\_from\_spark(self):  
 *"""  
 Test suit for create pandas dataframe from spark thirft server* **:return***:  
 """* spark\_reader = SparkReader(self.input\_json)  
 data\_frame = spark\_reader.read\_from\_configuration(**False**)  
 self.assertEqual(**'{"columns":["id","city","state","industry","salary"],"index":[0],'  
 '"data":[[1,"Holtsville","NY","Steel",5321]]}'**,  
 data\_frame.head(1).to\_json(orient=**'split'**))  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
This module is used to contains test case for mashup\_data\_registry module  
"""***import** unittest  
**import** pandas **as** pd  
**import** constants  
**import** datetime  
  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestDataMashUp(unittest.TestCase):  
 *"""  
 This class is used to write test case for all the method of DataRegistry class  
 """* **def** setUp(self):  
 self.data\_source\_id = 10  
 self.lens\_id = 50  
 self.data\_frame = pd.read\_csv(**'../resources/10Records.txt'**,  
 delimiter=**'\t'**, parse\_dates=[**'field0'**])  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_create\_registry\_id(self):  
 *"""  
 Test suite for create registry id* **:return***:  
 """* register\_id = DataRegistry.create\_register\_id(self.data\_source\_id, self.lens\_id)  
 **if** register\_id == **'10\_50'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_check\_data\_frame\_register(self):  
 *"""  
 this test suite is used for check data\_frame is register  
 or not based on data\_source\_id and lens\_id* **:return***:  
 """* status = DataRegistry.check\_data\_frame\_is\_register(self.data\_source\_id, self.lens\_id)  
 **if** status **is False**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
 json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_50"**: json})  
 status = DataRegistry.check\_data\_frame\_is\_register(self.data\_source\_id, self.lens\_id)  
 **if** status **is True**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 status = DataRegistry.check\_data\_frame\_is\_register(1, 1)  
 **if** status **is False**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 status = DataRegistry.check\_data\_frame\_is\_register(**None**, **None**)  
 **if** status **is False**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_register\_data\_frame(self):  
 *"""  
 This test suite is used to check whether data\_frame register is working or not* **:return***:  
 """* DataRegistry.register\_data\_frame(self.data\_source\_id, self.lens\_id, self.data\_frame)  
 **if** str(self.data\_source\_id) + **"\_"** + str(self.lens\_id) **in** DataRegistry.pandas\_df\_map:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_data\_frame\_from\_dict(self):  
 *"""  
 This test suite is used to fetch registered  
 data\_frame based on data\_source\_id and lens\_id* **:return***:  
 """* json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_50"**: json})  
 data\_frame = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=self.data\_source\_id  
 , lens\_id=self.lens\_id,  
 data\_frame\_type=**'WORKING'**)  
 **if** len(data\_frame) == len(self.data\_frame):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **try**:  
 DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=1,  
 lens\_id=1, data\_frame\_type=**'WORKING'**)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_update\_data\_frame(self):  
 *"""  
 This test suite for update data\_frame from using data\_source\_id and lens\_id* **:return***:  
 """* json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_50"**: json})  
 data\_frame = self.data\_frame[self.data\_frame[**'field0'**] == **'2017-10-15'**]  
 data\_source\_type = **'WORKING'** DataRegistry.update\_frame(self.data\_source\_id,  
 self.lens\_id, data\_frame, data\_source\_type)  
 data\_frame\_new = DataRegistry.pandas\_df\_map[str(self.data\_source\_id)  
 + **"\_"** + str(self.lens\_id)][data\_source\_type]  
 **if** len(data\_frame\_new) == len(data\_frame):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **try**:  
 DataRegistry.update\_frame(1,  
 1, data\_frame, data\_source\_type)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_keep\_original\_working(self):  
 *"""  
 This test suite is used to keep keep\_original\_as\_working  
 method of DataRegistry class* **:return***:  
 """* json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_50"**: json})  
 data\_frame = self.data\_frame[self.data\_frame[**'field0'**] == **'2017-10-15'**]  
 data\_source\_type = **'WORKING'** DataRegistry.update\_frame(self.data\_source\_id,  
 self.lens\_id, data\_frame, data\_source\_type)  
  
 DataRegistry.keep\_original\_as\_working(data\_source\_id=self.data\_source\_id,  
 lens\_id=self.lens\_id)  
 **if** len(DataRegistry.pandas\_df\_map[str(self.data\_source\_id)  
 + **"\_"** + str(self.lens\_id)][**'ORIGINAL'**]) \  
 == len(DataRegistry.pandas\_df\_map[str(self.data\_source\_id) +  
 **"\_"** + str(self.lens\_id)][**'WORKING'**]):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_sample\_record\_registry(self):  
 *"""  
 This method is used to sample record for after applying filter  
 on working data frame.* **:return***:  
 """* json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_50"**: json})  
 json\_data = DataRegistry.sample\_record\_from\_data\_registry(10, 50, **"WORKING"**, 1)  
 **if** json\_data **is not None**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 self.data\_frame = self.data\_frame[self.data\_frame[**'field2'**] > 10]  
 json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_51"**: json})  
 json\_data = DataRegistry.sample\_record\_from\_data\_registry(10, 51, **"WORKING"**, 1)  
 **if** json\_data **is not None**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_convert\_datetime\_to\_ms(self):  
 *"""  
 This method will test the convert datetime to millisecond method of mashup\_data\_registry* **:return***:  
 """* value = DataRegistry.convert\_datetime\_into\_ms()  
 **if** float(value):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_update\_last\_used\_time(self):  
 *"""  
 This method is used to test last update time* **:return***:  
 """* json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_50"**: json})  
 DataRegistry.update\_last\_user\_time(10, 50)  
 **if "last\_used\_by" in** DataRegistry.pandas\_df\_map[**"10\_50"**]:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_minute\_difference(self):  
 *"""  
 Test suite for test find minute difference method* **:return***:  
 """* **import** datetime  
 dt1 = datetime.datetime.strptime(**'14.11.2020 13:10:42'**, **'%d.%m.%Y %H:%M:%S'**)  
 dt2 = datetime.datetime.strptime(**'14.11.2020 13:05:42'**, **'%d.%m.%Y %H:%M:%S'**)  
 value = DataRegistry.find\_minute\_between\_two\_dates(dt1, dt2)  
 **if** value == 5:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_remove\_data\_frame\_registry(self):  
 *"""  
 This method will remove data\_frame from registry  
 which are not using anymore within particular time* **:return***:  
 """* **import** datetime  
 dt\_obj = datetime.datetime.strptime(**'14.11.2018 13:10:42'**, **'%d.%m.%Y %H:%M:%S'**)  
 json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame,  
 **"last\_used\_by"**: dt\_obj.timestamp() \* 1000.0}  
 DataRegistry.pandas\_df\_map.update({**"10\_50"**: json})  
 dt\_obj = datetime.datetime.strptime(**'15.11.2090 13:10:42'**, **'%d.%m.%Y %H:%M:%S'**)  
 json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame,  
 **"last\_used\_by"**: dt\_obj.timestamp() \* 1000.0}  
 DataRegistry.pandas\_df\_map.update({**"10\_51"**: json})  
 DataRegistry.remove\_data\_frame\_from\_registry()  
 **if** len(DataRegistry.pandas\_df\_map) == 1:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_remove\_data\_frame\_registry\_manually(self):  
 DataRegistry.remove\_data\_frame\_from\_registry\_manually(**"10\_51"**)  
 self.assertEqual(len(DataRegistry.pandas\_df\_map), 0)  
  
 **with** self.assertRaises(Exception):  
 DataRegistry.remove\_data\_frame\_from\_registry\_manually(**"10\_51"**)  
  
 **def** test\_error(self):  
 **with** self.assertRaises(Exception):  
 DataRegistry.apply\_individual\_strategy(**None**, **None**)  
  
 **def** test\_create\_value(self):  
 self.assertEqual(DataRegistry.create\_value\_field\_type(constants.DATE\_TIME),  
 datetime.datetime(2018, 9, 11, 21, 11, 34))  
 self.assertEqual(DataRegistry.create\_value\_field\_type(constants.NUMBER), 10.11)  
 self.assertEqual(DataRegistry.create\_value\_field\_type(constants.INTEGER), 111)  
 self.assertEqual(DataRegistry.create\_value\_field\_type(**"bla"**),  
 **"DummyData"**)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

Dedupe

*"""  
This class is used to perform dedupe operation on the dataframe  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** constants  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** DedupeStep(object):  
 *"""  
 This class will perform the dedupe operation on the dataframe  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.column\_list = input\_json[constants.DEDUPE\_ON\_COLUMN]  
 self.type = input\_json[constants.DEDUPE\_RETAIN\_POLICY]  
 self.type = (**False if** self.type == constants.REMOVE\_ALL\_DUPLICATES **else** self.type)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method will perform the dedupe operation.  
 We remove duplicate record based on the criteria selected by user.  
 There are three main type:  
 1) first  
 2) last  
 3) remove all the duplicates* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 data\_frame.drop\_duplicates(subset=self.column\_list, keep=self.type, inplace=**True**)  
 data\_frame.reset\_index()  
 **return** data\_frame  
 **except** Exception **as** ex:  
 **if** type(ex) **is** AttributeError:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.NONE\_INPUT\_DATASET)  
 **else**:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to write test case for dedupe feature  
"""***import** unittest  
**import** pandas **as** pd  
**from** dedupe.dedupe\_step **import** DedupeStep  
  
  
**class** TestDedupeStep(unittest.TestCase):  
 *"""  
 Test suite for dedupe  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/10Records.txt"**, delimiter=**'\t'**, parse\_dates=[**'field0'**])  
 self.input\_json = {**"step\_name"**: **""**, **"dedupe\_on\_column"**: [**"field0"**],  
 **"dedupe\_retain\_policy"**: **"first"**}  
  
 **def** test\_perform\_dedupe\_operation(self):  
 *"""  
 This test suite will test the dedupe feature* **:return***:  
 """* dedupe\_step = DedupeStep(self.input\_json)  
 data\_frame = dedupe\_step.get\_result(self.data\_frame)  
 **if** len(data\_frame) == 11:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_perform\_key\_missing(self):  
 *"""  
 This test suite will test key missing exception* **:return***:  
 """* **try**:  
 self.input\_json.pop(**"dedupe\_on\_column"**)  
 DedupeStep(self.input\_json)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_dedupe\_on\_exception(self):  
 *"""  
 test suite for exception performing operation* **:return***:  
 """* **try**:  
 dedupe\_step = DedupeStep(self.input\_json)  
 dedupe\_step.get\_result(**None**)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_dedupe\_on\_system\_exception(self):  
 *"""  
 test suite for exception performing operation* **:return***:  
 """* **try**:  
 self.input\_json.update({**"dedupe\_on\_column"**: [**"field\_new"**]})  
 dedupe\_step = DedupeStep(self.input\_json)  
 dedupe\_step.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex.args[0])  
 self.assertEqual(1, 1)

Export dataframe

*"""  
Script to export working dataframe to excel  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** constants  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** ExportToExcel:  
 *"""  
 This class contains methods which converts working dataframe to excel  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 Initialize* **:param** *data\_source\_id:* **:param** *lens\_id:* **:param** *apply\_on:  
 """* **try**:  
 self.data\_source\_id = input\_json[constants.DATA\_SOURCE\_ID]  
 self.lens\_id = input\_json[constants.LENS\_ID]  
 self.apply\_on = input\_json.get(constants.APPLY\_ON, **"WORKING"**)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self):  
 *"""* **:param** *self.data\_source\_id:* **:param** *self.lens\_id:* **:param** *self.apply\_on:* **:return***:  
 """* **try**:  
  
 data\_frame = DataRegistry.get\_data\_set\_from\_dict(self.data\_source\_id, self.lens\_id  
 , self.apply\_on)  
 **return** data\_frame.to\_csv(index=**False**)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is test cases for cerate to\_csv from working dataframe  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** mashup\_data\_registry **import** DataRegistry  
**from** export\_dataframe.export\_to\_excel **import** ExportToExcel  
  
  
**class** TestExportToExcel(TestCase):  
 *"""  
 Class is for test convert to\_csv from dataframe  
 """* **def** setUp(self):  
 self.dataframe = pd.read\_csv(**"../resources/doc1.txt"**)  
 json\_obj = {**"ORIGINAL"**: self.dataframe, **"WORKING"**: self.dataframe}  
 DataRegistry.pandas\_df\_map.update({**"2\_2"**: json\_obj})  
 DataRegistry.pandas\_df\_map.update({**"3\_3"**: **None**})  
  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_get\_result(self):  
 *"""  
 Positive test case for get results* **:param** *passing mocked request* **:return***:  
 """* **try**:  
 export\_to\_excel = ExportToExcel({**"data\_source\_id"**: 2,  
 **"lens\_id"**: 2,  
 **"apply\_on"**: **"WORKING"**})  
 excel\_file = export\_to\_excel.get\_result()  
 dummy\_response = **"name,address,empId\naavesh,karahi,101\nmahesh,nashik,102\nswapnil"** \  
 **",mumbai,103\n"** self.assertEqual(excel\_file, dummy\_response)  
 **except** Exception:  
 self.assertEqual(1, 2)  
  
 **def** test\_exception(self):  
 *"""  
 Negative test case for get results* **:param** *passing mocked request* **:return***:  
 """* **with** self.assertRaises(Exception):  
 export\_to\_excel = ExportToExcel({**"data\_source\_id"**: 2,  
 **"apply\_on"**: **"WORKING"**})  
 export\_to\_excel.get\_result()  
  
 **with** self.assertRaises(Exception):  
 export\_to\_excel = ExportToExcel({**"data\_source\_id"**: 3,**"lens\_id"**: 3,  
 **"apply\_on"**: **"WORKING"**})  
 export\_to\_excel.get\_result()

Expression

*"""  
This module is used for string function for expression  
"""***import** random  
**import** pandas **as** pd  
**import** re  
**import** logging  
**import** numpy **as** np  
**from** logging.handlers **import** RotatingFileHandler  
logger = logging.getLogger(**"test"**)  
logger.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'application.log'**, maxBytes=20000000, backupCount=10)  
formatter = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s'**,datefmt=**'%Y-%m-%d %H:%M:%S'**)  
HANDLER.setFormatter(formatter)  
logger.addHandler(HANDLER)  
exception\_message = **"Mosaic Exception :"  
  
  
class** AggregateFunction(object):  
 *"""  
 This class contains all the string function of expression  
 Changed most of the methods for Pandas Implementation  
 """* **def** double\_cast\_worker(input\_string):  
 **try**:  
 var = float(str(input\_string))  
 **return** var  
 **except** Exception **as** exception:  
 **return None** @classmethod  
 **def** xavg(cls, params):  
 *"""  
 This method is used to replace the regex match with specified string* **:param** *params:* **:return***:  
 """* **try**:  
 df = pd.DataFrame({**'kpi'**:params[0], **'filter'**:params[1], **'group'** : params[2]})  
 df[**'kpi'**] = pd.to\_numeric(df[**'kpi'**], errors= **"coerce"**)  
 df[**'group'**] = df.group.astype(str).replace(**""**,np.nan).replace(**"None"**,np.nan).replace(**"nan"**,np.nan)  
  
 *#if filter condition is not given* **if**(df[**'filter'**].isnull().all()):  
 df\_filtered=df  
 **else**:  
 **if** df[**'filter'**].dtype == **'bool'**:  
 df\_filtered=df[df[**'filter'**]==**True**]  
 **else**:  
 df\_filtered=df  
 logger.warning(**"filter condition not evaluated to boolean, going with full data set"**)  
   
  
 *#if groupby column is not given* **if**(df[**'group'**].isnull().all()):  
 df\_group = df\_filtered.mean();  
 **return** df\_group[**'kpi'**]  
 **else**:  
 df\_group = pd.DataFrame(df\_filtered.groupby(**'group'**)[**'kpi'**].mean())  
 df\_merge = pd.merge(df,df\_group,how=**"left"**,left\_on=**"group"**, right\_on=**"group"**)  
 **return** df\_merge[**"kpi\_y"**]  
  
 **except** Exception **as** ex:  
 logger.warning(**"xavg exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** xcount(cls, params):  
 *"""  
 This method is used to get the count of values in the given column* **:param** *params:* **:return** *numeric count:  
 """* **try** :  
 df = pd.DataFrame({**'kpi'**:params[0], **'filter'**:params[1], **'group'** : params[2]})  
 *#df['kpi'] = pd.to\_numeric(df['kpi'], errors= "coerce")* df[**'group'**] = df.group.astype(str).replace(**""**,np.nan).replace(**"None"**,np.nan).replace(**"nan"**,np.nan)  
   
 *#if filter condition is not given* **if**(df[**'filter'**].isnull().all()):  
 df\_filtered=df  
 **else**:  
 **if** df[**'filter'**].dtype == **'bool'**:  
 df\_filtered=df[df[**'filter'**]==**True**]  
 **else**:  
 df\_filtered=df  
 logger.warning(**"filter condition not evaluated to boolean, going with full data set"**)  
   
  
 *#if groupby column is not given* **if**(df[**'group'**].isnull().all()):  
 df\_group = df\_filtered.count()  
 **return** df\_group[**'kpi'**]  
 **else**:  
 df\_group = pd.DataFrame(df\_filtered.groupby(**'group'**)[**'kpi'**].count())  
 df\_merge = pd.merge(df,df\_group,how=**"left"**,left\_on=**"group"**, right\_on=**"group"**)  
 **return** df\_merge[**"kpi\_y"**]  
  
 **except** Exception **as** ex:  
 logger.warning(**"xcount exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** xmax(cls, params):  
 *"""  
 This method is used to replace the regex match with specified string* **:param** *params:* **:return***:  
 """* **try**:  
   
 df = pd.DataFrame({**'kpi'**:params[0], **'filter'**:params[1], **'group'** : params[2]})  
 df[**'kpi'**] = pd.to\_numeric(df[**'kpi'**], errors= **"coerce"**)  
 df[**'group'**] = df.group.astype(str).replace(**""**,np.nan).replace(**"None"**,np.nan).replace(**"nan"**,np.nan)  
   
 *#if filter condition is not given* **if**(df[**'filter'**].isnull().all()):  
 df\_filtered=df  
 **else**:  
 **if** df[**'filter'**].dtype == **'bool'**:  
 df\_filtered=df[df[**'filter'**]==**True**]  
 **else**:  
 df\_filtered=df  
 logger.warning(**"filter condition not evaluated to boolean, going with full data set"**)  
   
 *#if groupby column is not given* **if**(df[**'group'**].isnull().all()):  
 df\_group = df\_filtered.max()  
 **return** df\_group[**'kpi'**]  
 **else**:  
 df\_group = pd.DataFrame(df\_filtered.groupby(**'group'**)[**'kpi'**].max())  
 df\_merge = pd.merge(df,df\_group,how=**"left"**,left\_on=**"group"**, right\_on=**"group"**)  
 **return** df\_merge[**"kpi\_y"**]  
 **except** Exception **as** ex:  
 logger.warning(**"xmax exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** xmin(cls, params):  
 *"""  
 This method is used to replace the regex match with specified string* **:param** *params:* **:return***:  
 """* **try**:  
 df = pd.DataFrame({**'kpi'**:params[0], **'filter'**:params[1], **'group'** : params[2]})  
 df[**'kpi'**] = pd.to\_numeric(df[**'kpi'**], errors= **"coerce"**)  
 df[**'group'**] = df.group.astype(str).replace(**""**,np.nan).replace(**"None"**,np.nan).replace(**"nan"**,np.nan)  
  
 *#if filter condition is not given* **if**(df[**'filter'**].isnull().all()):  
 df\_filtered=df  
 **else**:  
 **if** df[**'filter'**].dtype == **'bool'**:  
 df\_filtered=df[df[**'filter'**]==**True**]  
 **else**:  
 df\_filtered=df  
 logger.warning(**"filter condition not evaluated to boolean, going with full data set"**)  
   
 *#if groupby column is not given* **if**(df[**'group'**].isnull().all()):  
 df\_group = df\_filtered.min()  
 **return** df\_group[**'kpi'**]  
 **else**:  
 df\_group = pd.DataFrame(df\_filtered.groupby(**'group'**)[**'kpi'**].min())  
 df\_merge = pd.merge(df,df\_group,how=**"left"**,left\_on=**"group"**, right\_on=**"group"**)  
 **return** df\_merge[**"kpi\_y"**]  
 **except** Exception **as** ex:  
 logger.warning(**"xmin "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** xsum(cls, params):  
 *"""  
 This method is used to replace the regex match with specified string* **:param** *params:* **:return***:  
 """* **try**:  
 df = pd.DataFrame({**'kpi'**:params[0], **'filter'**:params[1], **'group'** : params[2]})  
 df[**'kpi'**] = pd.to\_numeric(df[**'kpi'**], errors= **"coerce"**)  
 df[**'group'**] = df.group.astype(str).replace(**""**,np.nan).replace(**"None"**,np.nan).replace(**"nan"**,np.nan)  
   
 *#if filter condition is not given* **if**(df[**'filter'**].isnull().all()):  
 df\_filtered=df  
 **else**:  
 **if** df[**'filter'**].dtype == **'bool'**:  
 df\_filtered=df[df[**'filter'**]==**True**]  
 **else**:  
 df\_filtered=df  
 logger.warning(**"filter condition not evaluated to boolean, going with full data set"**)  
  
   
 *#if groupby column is not given* **if**(df[**'group'**].isnull().all()):  
 df\_group = df\_filtered.sum()  
 **return** df\_group[**'kpi'**]  
 **else**:  
 df\_group = pd.DataFrame(df\_filtered.groupby(**'group'**)[**'kpi'**].sum())  
 df\_merge = pd.merge(df,df\_group,how=**"left"**,left\_on=**"group"**, right\_on=**"group"**)  
 **return** df\_merge[**"kpi\_y"**]  
 **except** Exception **as** ex:  
 logger.warning(**"xsum "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))

*"""  
This module is used for string function for expression  
"""***import** random  
**import** pandas **as** pd  
**import** re  
**import** logging  
**import** math  
**import** numpy **as** np  
**from** logging.handlers **import** RotatingFileHandler  
logger = logging.getLogger(**"test"**)  
logger.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'application.log'**, maxBytes=20000000, backupCount=10)  
formatter = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s'**,datefmt=**'%Y-%m-%d %H:%M:%S'**)  
HANDLER.setFormatter(formatter)  
logger.addHandler(HANDLER)  
exception\_message = **"Mosaic Exception :"  
global** MEMOIZATION  
MEMOIZATION = **True  
  
class** ArithmeticFunction(object):  
 *"""  
 This class contains all the arithmetic function of expression  
 Changed most of the methods for Pandas Implementation  
 """* **def** memoize(func):  
 cache = dict()  
 **def** memoized\_func(\*args):  
 **if** args **in** cache:  
 **return** cache[args]  
 result = func(\*args)  
 cache[args] = result  
 **return** result  
 **return** memoized\_func  
   
  
 @classmethod  
 **def** round(cls, param):  
 *"""  
 Give any double random value* **:return***:  
 """* **try**:  
 rounding = 0  
 **if** type(param[1]) != int:  
 rounding = 2  
 **else**:  
 rounding = param[1]  
  
 **if** type(param[0]) == pd.core.series.Series:  
 column = pd.to\_numeric(param[0], errors=**"coerce"**)  
 **return** column.round(rounding)  
 **else**:  
 **try**:  
 column = float(str(param[0]))  
 **except** Exception **as** ex:  
 column = np.nan  
  
 **return** round(column,rounding)  
 **except** Exception **as** ex:  
 logger.warning(**"round exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
 **def** make\_df\_numeric(df):  
 print(**"making df numeric : "**)  
 **for** \_ **in** df.columns:  
 df[\_] = pd.to\_numeric(df[\_], errors=**'coerce'**)   
  
 @classmethod  
 **def** greatest(cls, param):  
 *"""  
 Give any double random value* **:return***:  
 """* **try**:  
 functionparams = param[:(len(param)-1)]  
 paramindex = param[(len(param)-1)]  
 df = pd.DataFrame(functionparams)  
 df = pd.DataFrame(functionparams).T  
 ArithmeticFunction.make\_df\_numeric(df)  
 **if** len(df) == 1:  
 **return** df.max(axis=1).tolist()[0]  
 **else**:  
 **return** df.max(axis=1)  
 **except** Exception **as** ex:  
 logger.warning(**"greatest exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** least(cls, param):  
 *"""  
 Give any double random value* **:return***:  
 """* **try**:  
 functionparams = param[:(len(param)-1)]  
 paramindex = param[(len(param)-1)]  
 df = pd.DataFrame(functionparams)  
 df = pd.DataFrame(functionparams).T  
 ArithmeticFunction.make\_df\_numeric(df)  
 **if** len(df) == 1:  
 **return** df.min(axis=1).tolist()[0]  
 **else**:  
 **return** df.min(axis=1)  
 **except** Exception **as** ex:  
 logger.warning(**"least exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
   
 @classmethod  
 **def** abs(cls, params):  
 *"""  
 Give any double random value* **:return***:  
 """* **try**:  
 **def** abs\_worker(input\_string):  
 **try**:  
 input\_string\_floor = float(str(input\_string))  
 **if** (math.floor(input\_string\_floor) == input\_string\_floor):  
 var = abs(int(input\_string\_floor))  
 **else**:  
 var = abs(float(input\_string))  
 **return** var  
 **except** Exception **as** ex:  
 logger.warning(**"abs worker : "** + str(ex))  
 **return** input\_string  
  
 **if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 abs\_worker = ArithmeticFunction.memoize(abs\_worker)  
  
  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([abs\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** abs\_worker(params[0])  
  
 **except** Exception **as** ex:  
 logger.warning(**"abs exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))

*"""  
This module is used for defining all the date function used in expression  
"""***import** pandas **as** pd  
**from** datetime **import** datetime  
**from** dateutil.relativedelta **import** relativedelta  
**from** dateutil **import** parser  
**import** numpy **as** np  
**import** logging  
**from** logging.handlers **import** RotatingFileHandler  
logger = logging.getLogger(**"test"**)  
logger.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'application.log'**, maxBytes=20000000, backupCount=10)  
formatter = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s'**,datefmt=**'%Y-%m-%d %H:%M:%S'**)  
HANDLER.setFormatter(formatter)  
logger.addHandler(HANDLER)  
exception\_message = **"Mosaic Exception :"  
global** MEMOIZATION  
MEMOIZATION = **True**DEFAULTTDATE = **"%Y-%m-%d"  
  
  
  
  
class** DateTimeFunction(object):  
 *"""  
 This class is used for executing date and time function in antlr expression  
 """* **def** memoize(func):  
 *"""  
 This is helper function to implement cashing to improve the performance  
 """* cache = dict()  
 **def** memoized\_func(\*args):  
 **if** args **in** cache:  
 **return** cache[args]  
 result = func(\*args)  
 cache[args] = result  
 **return** result  
 **return** memoized\_func  
  
  
 @classmethod  
 **def** now(cls, params):  
 *"""  
 This method is used to return current datetime.* **:param** *params* **:return***:  
 """* **try**:  
 **return** datetime.utcnow()  
 **except** Exception **as** ex:  
 logger.warning(**"now : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** to\_date(cls, params):  
 *"""  
 This method is used to convert string into date type using specify format* **:param** *params:* **:return***:  
 """* **def** to\_Date\_worker(input):  
 **try**:  
 **if** input[1] == **""**:  
 **return** pd.to\_datetime(input[0])  
 **return** pd.to\_datetime(input[0],format = input[1],cache = **True**, errors=**"coerce"**, exact=**True**)  
 **except** Exception **as** ex:  
 logger.warning(**"to\_date worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 to\_Date\_worker = DateTimeFunction.memoize(to\_Date\_worker)  
  
 param0 = params[0]  
 param1 = params[1]  
  
 **try**:  
 **if** type(param1) == pd.core.series.Series:  
 logger.warning(**"to\_date paramtert 2 need to be scalar"**)  
 param1 = DEFAULTTDATE  
  
 **if** ((param1 == **None**) **or** (param1 == **""**)):  
 param1 = **""  
   
 if** type(param0) == pd.core.series.Series:  
 param1 = pd.Series(param1, index=params[2])  
 **return** pd.Series([to\_Date\_worker(x) **for** x **in** list(zip(param0,param1))])  
 **else**:  
 **return** to\_Date\_worker((param0,param1))  
 **except** Exception **as** ex:  
 logger.warning(**"to\_date exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** date\_diff(cls, params):  
 *"""  
 find the day difference between two dates* **:param***:params* **:return***:  
 """* **def** date\_diff\_worker(input):  
 **try**:  
 diff = input[1] - input[0]  
 **return** diff / np.timedelta64(1, **'D'**)  
 **except** Exception **as** ex:  
 logger.warning(**"date\_diff worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 date\_diff\_worker = DateTimeFunction.memoize(date\_diff\_worker)  
  
 **try**:  
 **if** (type(params[0]) == pd.core.series.Series):   
 **if** params[0].dtype == **'datetime64[ns]'**:  
 date1 = params[0]  
 **else**:  
 date1 = pd.to\_datetime(params[0],format = DEFAULTTDATE,dayfirst=**True**,errors = **"coerce"**)  
 **else**:  
 date1 = pd.to\_datetime(params[0],format = DEFAULTTDATE,dayfirst=**True**,errors = **"coerce"**)  
  
 **if** (type(params[1]) == pd.core.series.Series):   
 **if** params[1].dtype == **'datetime64[ns]'**:  
 date2 = params[1]  
 **else**:  
 date2 = pd.to\_datetime(params[1],format = DEFAULTTDATE ,dayfirst=**True**,errors = **"coerce"**)  
 **else**:  
 date2 = pd.to\_datetime(params[1],format = DEFAULTTDATE, dayfirst=**True**, errors = **"coerce"**)  
   
 **if** (type(params[0]) == pd.core.series.Series) & (type(params[1]) == pd.core.series.Series):  
 **return** pd.Series([date\_diff\_worker(x) **for** x **in** list(zip(date2,date1))])  
   
 **elif** (type(params[0]) == pd.core.series.Series) & (type(params[1]) != pd.core.series.Series):  
 date2 = pd.Series(data = date2, index = params[2])  
 **return** pd.Series([date\_diff\_worker(x) **for** x **in** list(zip(date2,date1))])  
 **elif** (type(params[0]) != pd.core.series.Series) & (type(params[1]) == pd.core.series.Series):  
 date1 = pd.Series(data = date1, index = params[2])  
 **return** pd.Series([date\_diff\_worker(x) **for** x **in** list(zip(date2,date1))])  
 **else**:  
 diff = date2 - date1  
 **return** diff / np.timedelta64(1, **'D'**)  
 **except** Exception **as** ex:  
 logger.warning(**"date\_diff exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** year(cls, params):  
 *"""  
 find the year of particular date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache = **True**).dt.year  
 **else**:  
 **return** pd.to\_datetime(params[0]).year  
 **except** Exception **as** ex:  
 logger.warning(**"year exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
   
  
 @classmethod  
 **def** add\_months(cls, params):  
 *"""  
 Add month in given dates* **:param** *params:* **:return***:  
 """* **def** add\_months\_worker(input):  
 *""" For scalar"""* **try**:  
 **return** input[0] + np.timedelta64(int(input[1]), **'M'**)  
 **except** Exception **as** ex:  
 logger.warning(**"add\_months worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 add\_months\_worker = DateTimeFunction.memoize(add\_months\_worker)  
  
 **try**:  
 **if** (type(params[0]) == pd.core.series.Series):  
 **if** params[0].dtype == **'datetime64[ns]'**:  
 date1 = params[0]  
 **else**:  
 date1 = pd.to\_datetime(params[0], dayfirst=**True**, errors=**"coerce"**)  
 **else**:  
 date1 = pd.to\_datetime(params[0], dayfirst=**True**, errors=**"coerce"**)  
 **if** type(params[1]) == pd.core.series.Series:  
 months = pd.to\_numeric(params[1], errors=**"coerce"**)  
 **else**:  
 **try**:  
 months = float(str(params[1]))  
 **except** Exception **as** ex:  
 months = np.nan  
 **if** (type(params[0]) == pd.core.series.Series) & (  
 type(params[1]) == pd.core.series.Series):  
 **return** pd.Series([add\_months\_worker(x) **for** x **in** list(zip(date1, months))])  
 **elif** (type(params[0]) == pd.core.series.Series) & (  
 type(params[1]) != pd.core.series.Series):  
 months = pd.Series(data=months, index=params[2])  
 **return** pd.Series([add\_months\_worker(x) **for** x **in** list(zip(date1, months))])  
 **elif** (type(params[0]) != pd.core.series.Series) & (  
 type(params[1]) == pd.core.series.Series):  
 date1 = pd.Series(data=date1, index=params[2])  
 **return** pd.Series([add\_months\_worker(x) **for** x **in** list(zip(date1, months))])  
 **else**:  
 **return** add\_months\_worker(params)  
 **except** Exception **as** ex:  
 logger.warning(**"add\_months exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** today(cls, params):  
 *"""  
 This method is used to return current date.* **:param** *params* **:return***: current date string in yyyy/mm/dd format  
 """* **try**:  
 **return** datetime.utcnow().strftime(**"%Y-%m-%d"**)  
 **except** Exception **as** ex:  
 logger.warning(**"now : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** dateadd(cls, params):  
 *"""  
 This method is used to add `Days` to dateString* **:param** *params: dateString, num of days* **:return***: dateString with added days  
 """* **def** dateadd\_worker(input):  
 *""" For scalar"""* **try**:  
 **return** input[0] + np.timedelta64(int(input[1]), **'D'**)  
 **except** Exception **as** ex:  
 logger.warning(**"date\_diff worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 dateadd\_worker = DateTimeFunction.memoize(dateadd\_worker)  
  
 **try**:  
 **if** (type(params[0]) == pd.core.series.Series):  
 **if** params[0].dtype == **'datetime64[ns]'**:  
 date1 = params[0]  
 **else**:  
 date1 = pd.to\_datetime(params[0], dayfirst=**True**, errors=**"coerce"**)  
 **else**:  
 date1 = pd.to\_datetime(params[0], dayfirst=**True**, errors=**"coerce"**)  
 **if** type(params[1]) == pd.core.series.Series:  
 days = pd.to\_numeric(params[1], errors=**"coerce"**)  
 **else**:  
 **try**:  
 days = float(str(params[1]))  
 **except** Exception **as** ex:  
 days = np.nan  
 **if** (type(params[0]) == pd.core.series.Series) & (  
 type(params[1]) == pd.core.series.Series):  
 **return** pd.Series([dateadd\_worker(x) **for** x **in** list(zip(date1, days))])  
 **elif** (type(params[0]) == pd.core.series.Series) & (  
 type(params[1]) != pd.core.series.Series):  
 days = pd.Series(data=days, index=params[2])  
 **return** pd.Series([dateadd\_worker(x) **for** x **in** list(zip(date1, days))])  
 **elif** (type(params[0]) != pd.core.series.Series) & (  
 type(params[1]) == pd.core.series.Series):  
 date1 = pd.Series(data=date1, index=params[2])  
 **return** pd.Series([dateadd\_worker(x) **for** x **in** list(zip(date1, days))])  
 **else**:  
 **return** dateadd\_worker(params)  
 **except** Exception **as** ex:  
 logger.warning(**"date\_diff exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** day(cls, params):  
 *"""  
 find the day of particular date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.day  
 **else**:  
 **return** pd.to\_datetime(params[0]).day  
 **except** Exception **as** ex:  
 logger.warning(**"day exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** month(cls, params):  
 *"""  
 find the day of particular month* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.month  
 **else**:  
 **return** pd.to\_datetime(params[0]).month  
 **except** Exception **as** ex:  
 logger.warning(**"day exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** get\_time(cls, params):  
 *"""  
 get time string HH:MM:SS:MS of particular date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.strftime(**"%H:%M:%S:%f"**)  
 **else**:  
 **return** pd.to\_datetime(params[0]).strftime(**"%H:%M:%S:%f"**)  
 **except** Exception **as** ex:  
 logger.warning(**"TIME exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** get\_datetime(cls, params):  
 *"""  
 get time string DD/MM/YYYY HH:MM:SS:MS of particular date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.strftime(**"%d/%m/%Y %H:%M:%S:%f"**)  
 **else**:  
 **return** pd.to\_datetime(params[0]).strftime(**"%d/%m/%Y %H:%M:%S:%f"**)  
 **except** Exception **as** ex:  
 logger.warning(**"TIME exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** weeknum(cls, params):  
 *"""  
 find the week num of date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.weekofyear  
 **else**:  
 **return** pd.to\_datetime(params[0]).weekofyear  
 **except** Exception **as** ex:  
 logger.warning(**"weeknum exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** monthname(cls, params):  
 *"""  
 find the name of month from date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.month\_name()  
 **else**:  
 **return** pd.to\_datetime(params[0]).month\_name()  
 **except** Exception **as** ex:  
 logger.warning(**"monthname exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** get\_hour(cls, params):  
 *"""  
 get hour from date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.hour  
 **else**:  
 **return** pd.to\_datetime(params[0]).hour  
 **except** Exception **as** ex:  
 logger.warning(**"hour exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** get\_minute(cls, params):  
 *"""  
 get minutes from date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.minute  
 **else**:  
 **return** pd.to\_datetime(params[0]).minute  
 **except** Exception **as** ex:  
 logger.warning(**"minute exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** get\_second(cls, params):  
 *"""  
 get seconds from date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.second  
 **else**:  
 **return** pd.to\_datetime(params[0]).second  
 **except** Exception **as** ex:  
 logger.warning(**"second exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))  
  
 @classmethod  
 **def** day\_name(cls, params):  
 *"""  
 get day name from date* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.to\_datetime(params[0], cache=**True**).dt.day\_name()  
 **else**:  
 **return** pd.to\_datetime(params[0]).day\_name()  
 **except** Exception **as** ex:  
 logger.warning(**"day name exception : "** + str(ex))  
 *# return None* **raise** Exception(**""**.join([exception\_message, str(ex)]))

*"""  
This module is used for string function for expression  
"""***import** random  
**import** pandas **as** pd  
**import** re  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** pandasql **import** sqldf  
pd.options.mode.chained\_assignment = **None  
import** math  
**import** logging  
**from** logging.handlers **import** RotatingFileHandler  
logger = logging.getLogger(**"test"**)  
logger.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'application.log'**, maxBytes=20000000, backupCount=10)  
formatter = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s'**,datefmt=**'%Y-%m-%d %H:%M:%S'**)  
HANDLER.setFormatter(formatter)  
logger.addHandler(HANDLER)  
exception\_message = **"Mosaic Exception :"  
global** MEMOIZATION  
MEMOIZATION = **True  
  
  
  
class** LookupFunction(object):  
 *"""  
 This class contains all the string function of expression  
 Changed most of the methods for Pandas Implementation  
 """* @classmethod  
 **def** look\_up\_map(cls, params):  
 *"""  
 This method is used to map the value based on given lookup condition* **:param** *params:* **:return***:  
 """* fetched\_master\_frame = DataRegistry.get\_data\_set\_from\_dict(params[0],**"1000"**,**"ORIGINAL"**)  
 lookupdict = fetched\_master\_frame.set\_index(params[1])[params[2]].to\_dict()  
 outputSeries=params[3].map(lookupdict)  
 **return** outputSeries  
  
 @classmethod  
 **def** look\_up\_check\_old(cls, params):  
 *"""  
 This method is old/decprecated lookup expression   
 """* p = re.compile(params[0]+**".(.\*?) "**)  
 master\_partition=p.findall(params[1])  
 md = DataRegistry.get\_data\_set\_from\_dict(params[0],**"1000"**,**"ORIGINAL"**)  
 incoming\_data = params[3]  
 mdddupe = md.drop\_duplicates(subset=master\_partition, keep=**'first'**, inplace=**False**)  
 sqlQuery=**"SELECT "**+params[2]+**" FROM incoming\_data as this left join mdddupe as "**+params[0]+**" on "**+params[1]+**";"** odf = sqldf(sqlQuery)  
 **return**(odf[params[2].split(**"."**)[1] **if** len(params[2].split(**"."**))==2 **else** params[2]])  
  
  
 **def** make\_df\_columns\_lower(df):  
 df.columns = [x.lower() **for** x **in** df.columns]  
  
   
 **def** memoize(func):  
 *"""  
 This is helper function to implement cashing to improve the performance  
 """* cache = dict()  
 **def** memoized\_func(\*args):  
 **if** args **in** cache:  
 **return** cache[args]  
 result = func(\*args)  
 cache[args] = result  
 **return** result  
 **return** memoized\_func  
  
  
  
 @classmethod  
 **def** look\_up\_check(cls, params):  
 *"""  
 This method is used to run reference table lookup  
 """  
   
 #This is worker functuin to cast object into numeric form* **def** numeric\_formatter(input\_string):  
 **try**:  
 input\_string\_floor = float(str(input\_string))  
 **if** (math.floor(input\_string\_floor) == input\_string\_floor):  
 var = str(int(input\_string\_floor))  
 **else**:  
 var = input\_string  
 **return** var  
 **except** Exception **as** ex:  
 logger.warning(**"numeric\_formatter "** + str(ex))  
 **return** input\_string  
  
 **try**:  
 params[1] = params[1].replace(**" && "**, **" AND "**)  
 params[1] = params[1].replace(**" && "**, **" and "**)  
 params[1] = params[1].replace(**" || "**,**" OR "**)  
 params[1] = params[1].replace(**" || "**,**" or "**)  
  
 md = DataRegistry.get\_data\_set\_from\_dict(params[0],**"1000"**,**"ORIGINAL"**).copy()  
 incoming\_data = params[3]  
  
 **"""  
 This block runs lookup in a column to column join using pandas merge  
 """  
 if**(**not**(any(operator **in** params[1].lower() **for** operator **in** [**"or"**,**">"**,**">="**,**"<"**,**"<="**]))):  
 condition\_list = params[1].split(**'AND'**)  
 masterlist = []  
 datalist = []  
  
 **for** \_ **in** condition\_list:  
 templist = \_.split(**"="**)  
 masterlist.append(templist[0].lower().strip())  
 datalist.append(templist[1].lower().strip())  
  
  
 LookupFunction.make\_df\_columns\_lower(md)  
  
 **if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 numeric\_formatter = LookupFunction.memoize(numeric\_formatter)  
  
 **for** \_ **in** masterlist:  
 **if** md[\_].dtype == object:  
 **pass  
 else**:  
 *#converting lookup columns into numeric form toavoid join conflicts due to decimal points* md[\_] = pd.Series([numeric\_formatter(x) **for** x **in** md[\_]])   
  
 mdddupe = md.drop\_duplicates(subset=masterlist, keep=**'first'**, inplace=**False**)  
  
  
 LookupFunction.make\_df\_columns\_lower(incoming\_data)  
 **for** \_ **in** datalist:  
 **if** incoming\_data[\_].dtype == object:  
 **pass  
 else**:   
 *#converting lookup columns into numeric form toavoid join conflicts due to decimal points* incoming\_data[\_] = pd.Series([numeric\_formatter(x) **for** x **in** incoming\_data[\_]], index = params[3].index )   
  
 result\_df = pd.merge(incoming\_data,mdddupe, how = **"left"**, left\_on=datalist, right\_on=masterlist)  
   
 *#HANDLING OF DUBLICATE COLUMNS IN MERGE OPERATION* to\_be\_returned\_col = params[2].lower().strip()  
 **if** to\_be\_returned\_col **in** incoming\_data.columns:  
 **return** result\_df[to\_be\_returned\_col+**"\_y"**]  
 **else**:  
 **return** result\_df[to\_be\_returned\_col]  
 **else**:  
 **"""   
 This block runs in psql based joins, where join conditons like "or",">",">=","<","<=" are badly   
 configured in lookup, which can cause Cardinality issues  
 """** logger.warning(**"look\_up\_check Cardinality may be a miss, please check the expression!!!"**)  
 condition\_list = params[1].replace(**"OR"**,**"~"**).replace(**"AND"**,**"~"**).split(**'~'**)  
 conditions\_dict={}  
 masterlist=[]  
 **for** \_ **in** condition\_list:  
 splited\_columns = \_.split(**"="**)  
 masterlist.append(splited\_columns[0].strip())  
 conditions\_dict[\_.strip()] = **"right\_table."**+splited\_columns[0].strip()+**" = "**+**"left\_table."**+splited\_columns[1].strip()  
  
 mdddupe = md.drop\_duplicates(subset=masterlist, keep=**'first'**, inplace=**False**).copy(deep=**False**)  
  
 **for** k,v **in** conditions\_dict.items():  
 params[1] = params[1].replace(k, v)  
  
 sqlQuery=**"select right\_table."**+params[2]+**" from incoming\_data as left\_table join mdddupe as right\_table on "**+params[1]+**";"** odf = sqldf(sqlQuery)  
 **return** odf[params[2]]  
   
 **except** Exception **as** ex:  
 logger.warning(**"look\_up\_check exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))

*"""  
This module is used for string function for expression  
"""***import** random  
**import** pandas **as** pd  
**import** re  
**import** numpy **as** np  
**import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** math  
logger = logging.getLogger(**"test"**)  
logger.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'application.log'**, maxBytes=20000000, backupCount=10)  
formatter = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s'**,datefmt=**'%Y-%m-%d %H:%M:%S'**)  
HANDLER.setFormatter(formatter)  
logger.addHandler(HANDLER)  
exception\_message = **"Mosaic Exception :"  
global** MEMOIZATION  
MEMOIZATION = **False  
  
  
class** StringFunction(object):  
 *"""  
 This class contains all the string function of expression  
 Changed most of the methods for Pandas Implementation  
 """* **def** memoize(func):  
 *"""  
 This is helper function to implement cashing to improve the performance  
 """* cache = dict()  
 **def** memoized\_func(\*args):  
 **if** args **in** cache:  
 **return** cache[args]  
 result = func(\*args)  
 cache[args] = result  
 **return** result  
 **return** memoized\_func  
   
  
  
   
 @classmethod  
 **def** random\_number(cls, param\_list):  
 *"""  
 Give any double random value* **:return***:  
 """* **try**:  
 **return** pd.Series(np.random.ranf(size = len(param\_list[0])), index = param\_list[0])  
 **except** Exception **as** ex:  
 logger.warning(**"random\_number exception : "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
   
  
 @classmethod  
 **def** substring\_of\_string(cls, param\_list):  
 *"""  
 This method is used to slicing string* **:param** *param\_list:* **:return***:  
 """  
   
 #Worker function to get the substring from given string* **def** substring\_of\_string\_worker(\_):  
 **try**:  
 **return** \_[0][int(\_[1])-1:int(\_[2])+int(\_[1])-1]   
 **except** Exception **as** ex:  
 logger.warning(**"substring\_of\_string worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 substring\_of\_string\_worker = StringFunction.memoize(substring\_of\_string\_worker)  
   
 **try**:  
 **if** type(param\_list[0]) != pd.core.series.Series:  
 param1 = pd.Series(data = param\_list[0], index = param\_list[3] )  
 **else**:  
 param1 = param\_list[0]  
 **if** ((type(param\_list[1]) == pd.core.series.Series) **or** (type(param\_list[2]) == pd.core.series.Series)):  
   
 param2 = pd.Series(data=param\_list[1], index=param1.index)  
 param3 = pd.Series(data=param\_list[2], index=param1.index)  
 return\_series = pd.Series(data = [substring\_of\_string\_worker(\_) **for** \_ **in** list(zip(param1.tolist(),param2.tolist(), param3.tolist()))],index=param1.index)   
 **return** return\_series  
 **else**:  
 **if** (str(param\_list[1]).replace(**'None'**,**''**).replace(**'nan'**,**''**) == **""**) & (str(param\_list[2]).replace(**'None'**,**''**).replace(**'nan'**,**''**) == **""**):  
 **return** param1.str.slice(**None**,**None**)  
 **elif** (str(param\_list[1]).replace(**'None'**,**''**).replace(**'nan'**,**''**) == **""**):  
 param3 = int(param\_list[2])  
 **return** param1.str.slice(**None**,param3)  
 **elif** (str(param\_list[2]).replace(**'None'**,**''**).replace(**'nan'**,**''**) == **""**):  
 param2 = int(param\_list[1])  
 **return** param1.str.slice(param2-1,**None**)  
 **else**:  
 param2 = int(param\_list[1])  
 param3 = int(param\_list[2])  
 **return** param1.str.slice(param2-1,param3+param2-1)  
 **except** Exception **as** ex:  
 logger.warning(**"substring\_of\_string exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
   
  
 @classmethod  
 **def** length( cls, params):  
 *"""  
 This method is used to find length of string* **:param** *params:* **:return***:  
 """  
   
 #Worker function to find the length of the scalar* **def** length\_worker(\_):  
 **try**:  
 **return** len(str(\_).replace(**'nan'**,**''**).replace(**'None'**,**''**))   
 **except** Exception **as** ex:  
 logger.warning(**"substring\_of\_string worker : "** + str(ex))  
 **return None  
   
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 length\_worker = StringFunction.memoize(length\_worker)  
  
 **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([length\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** length\_worker(params[0])  
 **except** Exception **as** ex:  
 logger.warning(**"length excption : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
   
  
 @classmethod  
 **def** lower(cls, params):  
 *"""  
 This method is used to convert string into lower* **:param** *params:* **:return***:  
 """  
 #Worker function to make scalar function lower* **def** lower\_worker(\_):  
 **try**:  
 **return** str(\_).replace(**'nan'**,**''**).replace(**'None'**,**''**).lower()   
 **except** Exception **as** ex:  
 logger.warning(**"lower worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 lower\_worker = StringFunction.memoize(lower\_worker)  
  
 **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([lower\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** lower\_worker(params[0])  
 **except** Exception **as** ex:  
 logger.warning(**"lower exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** upper(cls, params):  
 *"""  
 This method is used to convert string to upper case* **:param** *params:* **:return***:  
 """  
  
 #Worker function to make scalar value in upper case* **def** upper\_worker(\_):  
 **try**:  
 **return** str(\_).replace(**'nan'**,**''**).replace(**'None'**,**''**).upper()   
 **except** Exception **as** ex:  
 logger.warning(**"upper worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 upper\_worker = StringFunction.memoize(upper\_worker)  
  
 **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([upper\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** upper\_worker(params[0])  
 **except** Exception **as** ex:  
 logger.warning(**"upper exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** reverse(cls, params):  
 *"""  
 This method is used to reverse the string* **:param** *params:* **:return***:  
 """  
 #Worker function to reverse the scalar value* **def** reverse\_worker(\_):  
 **try**:  
 **return** str(\_)[::-1]   
 **except** Exception **as** ex:  
 logger.warning(**"reverse worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 reverse\_worker = StringFunction.memoize(reverse\_worker)  
  
 **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([x **if** str(x).replace(**'None'**,**''**).replace(**'nan'**,**''**) == **"" else** reverse\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** reverse\_worker(str(params[0]).replace(**'nan'**,**''**).replace(**'None'**,**''**))  
 **except** Exception **as** ex:  
 logger.warning(**"reverse exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
   
  
  
 @classmethod  
 **def** trim(cls, params):  
 *"""  
 This method is used for removing the white space from the string* **:param** *params:* **:return***:  
 """  
 #Worker function to remove the white spaces around the string* **def** trim\_worker(\_):  
 **try**:  
 **return** str(\_).strip()  
 **except** Exception **as** ex:  
 logger.warning(**"trim worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 trim\_worker = StringFunction.memoize(trim\_worker)  
  
 **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([trim\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** trim\_worker(str(params[0]))  
 **except** Exception **as** ex:  
 logger.warning(**"trim exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** left\_trim(cls, params):  
 *"""  
 This method is used to remove leading white space* **:param** *params* **:return***:  
 """  
  
 #Worker function to remove the leading white spaces around the string* **def** left\_trim\_worker(\_):  
 **try**:  
 **return** str(\_).lstrip()  
 **except** Exception **as** ex:  
 logger.warning(**"left trim worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 left\_trim\_worker = StringFunction.memoize(left\_trim\_worker)  
  
 **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([left\_trim\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** left\_trim\_worker(str(params[0]))  
 **except** Exception **as** ex:  
 logger.warning(**"left trim exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** right\_trim(cls, params):  
 *"""  
 This method is used to removing trailing white space from string* **:param** *params:* **:return***:  
 """  
  
 #Worker function to perform right trim on string* **def** right\_trim\_worker(\_):  
 **try**:  
 **return** str(\_).rstrip()  
 **except** Exception **as** ex:  
 logger.warning(**"right\_trim worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 right\_trim\_worker = StringFunction.memoize(right\_trim\_worker)  
  
 **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([right\_trim\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** right\_trim\_worker(str(params[0]))  
 **except** Exception **as** ex:  
 logger.warning(**"right\_trim exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** init\_cap(cls, params):  
 *"""  
 This method is used to convert first character to uppercase* **:param** *params:* **:return***:  
 """  
  
 #Worker function to convert first character to uppercase* **def** init\_cap\_worker(\_):  
 **try**:  
 **return** str(\_).title()  
 **except** Exception **as** ex:  
 logger.warning(**"init\_cap worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 init\_cap\_worker = StringFunction.memoize(init\_cap\_worker)  
  
 **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** pd.Series([init\_cap\_worker(x) **for** x **in** params[0]], index = params[1])  
 **else**:  
 **return** init\_cap\_worker(str(params[0]))  
 **except** Exception **as** ex:  
 logger.warning(**"init\_cap exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** is\_blank(cls, params):  
 *"""  
 This method is used to check whether string is empty or not* **:param** *params:* **:return***:  
 """* **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** ((params[0].astype(str) == **''**) | (params[0].isnull()))  
 **else**:  
 **return** str(params[0]).replace(**'None'**,**''**).replace(**'nan'**,**''**) == **""  
 except** Exception **as** ex:  
 logger.warning(**"is\_blank exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
  
  
 @classmethod  
 **def** is\_not\_blank(cls, params):  
 *"""  
 This method is used to check string is not blank* **:param** *params:* **:return***:  
 """* **try**:   
 **if** type(params[0]) == pd.core.series.Series:  
 **return** ~((params[0].astype(str) == **''**) | (params[0].isnull()))  
 **else**:  
 **return** str(params[0]).replace(**'None'**,**''**).replace(**'nan'**,**''**) == **""  
 except** Exception **as** ex:  
 logger.warning(**"is\_not\_blank exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** space(cls, params):  
 *"""   
 This method is used to provide the number of space* **:param** *params:* **:return***:  
 """* **try**:  
 value = **""  
 if** StringFunction.is\_null(params[0]):  
 **return** value  
 **for** \_ **in** range(int(params[0])):  
 value += **" "  
 return** value  
 **except** Exception **as** ex:  
 logger.warning(**"space exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** concat\_bk(cls, params):  
 *"""  
 This function is used to concat two string* **:param***:params* **:return***:  
 """  
 #Worker function to concat two strings* **def** concat\_worker(\_):  
 **try**:  
 **return ""**.join([str(\_[0]).replace(**'None'**,**''**).replace(**'nan'**,**''**), str(\_[1]).replace(**'None'**,**''**).replace(**'nan'**,**''**)])  
   
 **except** Exception **as** ex:  
 logger.warning(**"concat worker : "** + str(ex))  
 **return None  
   
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 concat\_worker = StringFunction.memoize(concat\_worker)  
  
 **try**:  
 **if** ((type(params[0]) != pd.core.series.Series) **and** (type(params[1]) != pd.core.series.Series)):   
 **return** concat\_worker([params[0],params[1]])  
 **elif**((type(params[0]) != pd.core.series.Series) **and** (type(params[1]) == pd.core.series.Series)):   
 param0 = pd.Series(data = params[0] , index = params[2])  
 **return** pd.Series([concat\_worker(\_) **for** \_ **in** list(zip(param0.tolist(),params[1].tolist()))],index=param0.index)   
 **elif**((type(params[0]) == pd.core.series.Series) **and** (type(params[1]) != pd.core.series.Series)):   
 param1 = pd.Series(data = params[1] , index = params[2])  
 **return** pd.Series([concat\_worker(\_) **for** \_ **in** list(zip(params[0].tolist(),param1.tolist()))],index=param1.index)   
 **else**:  
 **return** pd.Series([concat\_worker(\_) **for** \_ **in** list(zip(params[0].tolist(),params[1].tolist()))],index=params[1].index)   
   
 **except** Exception **as** ex:  
 logger.warning(**"concat exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))   
  
  
 @classmethod  
 **def** concat(cls, params):  
 *"""  
 This function is used to concat two string* **:param***:params* **:return***:  
 """  
 #Worker function to concat two strings* **def** concat\_worker(items):  
 **try**:  
 concat\_result=**''** *#concat\_result = (''.join(items))* **for** item **in** items:  
 concat\_result+=(str(item).replace(**'None'**,**''**).replace(**'nan'**,**''**))  
 **return** concat\_result  
 **except** Exception **as** ex:  
 *#logger.warning("concat worker : " + str(ex))* **return None  
   
 if** MEMOIZATION == **True**:  
 concat\_worker = StringFunction.memoize(concat\_worker)  
  
 **try**:  
 governing\_series\_index =params.pop()   
 **if**(len(params)>0):  
 all\_columns=[]   
 **for** series **in** params:  
 **if**(**not** isinstance(series, pd.core.series.Series)):  
 series = pd.Series(data = series , index = governing\_series\_index)  
 all\_columns.append(series)  
 **return** pd.Series([concat\_worker(\_) **for** \_ **in** list(zip(\*all\_columns))],index=governing\_series\_index)  
 **except** Exception **as** ex:  
 logger.error(**"concat exception : "** + str(ex))  
 **raise** Exception(ex)  
  
  
 @classmethod  
 **def** repeat(cls, params):  
 *"""  
 This function is used to repeat the string with given specified times* **:param** *params:* **:return***:  
 """* **def** repeat\_worker(\_):  
 **try**:  
 **if** (str(\_[0]).replace(**'None'**,**''**).replace(**'nan'**,**''**) ==**""**):  
 **return** str(\_[0])  
 **elif** (str(\_[1]).replace(**'None'**,**''**).replace(**'nan'**,**''**) ==**""**):  
 **return None  
 else**:  
 **return** str(\_[0])\*int(\_[1])  
   
 **except** Exception **as** ex:  
 logger.warning(**"repeat worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 repeat\_worker = StringFunction.memoize(repeat\_worker)  
  
 **try**:  
 **if** ((type(params[0]) != pd.core.series.Series) **and** (type(params[1]) != pd.core.series.Series)):   
 **return** repeat\_worker([params[0],params[1]])  
 **elif**((type(params[0]) != pd.core.series.Series) **and** (type(params[1]) == pd.core.series.Series)):   
 param0 = pd.Series(data = params[0] , index = params[2])  
 **return** pd.Series([repeat\_worker(\_) **for** \_ **in** list(zip(param0.tolist(),params[1].tolist()))],index=param0.index)   
 **elif**((type(params[0]) == pd.core.series.Series) **and** (type(params[1]) != pd.core.series.Series)):   
 param1 = pd.Series(data = params[1] , index = params[2])  
 **return** pd.Series([repeat\_worker(\_) **for** \_ **in** list(zip(params[0].tolist(),param1.tolist()))],index=param1.index)   
 **else**:  
 **return** pd.Series([repeat\_worker(\_) **for** \_ **in** list(zip(params[0].tolist(),params[1].tolist()))],index=params[1].index)   
   
 **except** Exception **as** ex:  
 logger.warning(**"repeat exception : "** + str(ex))  
 *#return None* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
  
 @classmethod  
 **def** instr(cls, param\_list):  
 *"""  
 This method will find the index of particular string and  
 if substring is not found it raise the expression* **:param** *params:* **:return***:  
 """  
 #Worker function to find indexof given substring* **def** instr\_worker(\_):  
 **try**:  
 **return** \_[0].count(\_[1])  
 **except** Exception **as** exception:  
 logger.warning(**"instr worker : "** + str(ex))  
 **return None  
  
 if** MEMOIZATION == **True**:  
 print(**"MEMOIZATION is ON"**)  
 instr\_worker = StringFunction.memoize(instr\_worker)  
  
 **try**:  
 **if** type(param\_list[0]) != pd.core.series.Series:  
 param1 = pd.Series(data = param\_list[0], index = param\_list[2])  
 **else**:  
 param1 = param\_list[0]  
  
 **if** ((type(param\_list[1]) == pd.core.series.Series)):  
 param2 = pd.Series(data=param\_list[1], index=param1.index)  
 return\_series = pd.Series(data = [instr\_worker(\_) **for** \_ **in** list(zip(param1.tolist(),param2.tolist()))],index=param1.index)   
 **return** return\_series  
 **else**:  
 param2 = str(param\_list[1])  
 **return** param1.str.count(param2)  
  
 **except** Exception **as** ex:  
 logger.warning(**"instr exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** regex\_match(cls, param\_list):  
 *"""  
 This method is used to match the regex pattern* **:param** *params:* **:return***:  
 """  
  
 #Worker implementation for saclar values* **def** regex\_match\_worker(\_):  
 **try**:  
 *#return bool(re.match(\_[1],str(\_[0])))* **return** bool(re.search(\_[1],str(\_[0])))  
 **except** Exception **as** ex:  
 logger.warning(**"regex\_match worker : "** + str(ex))  
 **return False  
  
 try**:  
  
 **if** type(param\_list[1]) != pd.core.series.Series:  
 param1 = pd.Series(data = param\_list[1], index = param\_list[2])  
 **else**:  
 param1 = param\_list[1]  
  
 **if** ((type(param\_list[0]) == pd.core.series.Series)):  
 param2 = param\_list[0]  
 return\_series = pd.Series(data = [regex\_match\_worker(\_) **for** \_ **in** list(zip(param1.tolist(),param2.tolist()))],index=param1.index)   
 **return** return\_series  
 **else**:  
 param2 = str(param\_list[0])  
 *#return param1.str.match(param2)* **return** param1.str.contains(param2, regex=**True**)  
 **except** Exception **as** ex:  
 logger.warning(**"regex\_match exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
   
  
  
 @classmethod  
 **def** regex\_extract(cls, params):  
 *"""  
 This method is used to extract the value from given string* **:param** *params:* **:return***:  
 """  
   
 #try:  
 # if StringFunction.is\_null(params[0]) | StringFunction.is\_null(params[1]):  
 # return params[0]  
 # values = re.findall(params[1], params[0])  
 # for value in values:  
 # return value[int(params[2]) - 1]  
 #except Exception as e:  
 # return float('nan')* **try**:  
 **return** params[1].str.extract(params[0])  
 **except** Exception **as** ex:  
 logger.warning(**"regex\_extract exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** regex\_replace(cls, params):  
 *"""  
 This method is used to replace the regex match with specified string* **:param** *params:* **:return***:  
 """  
   
 #if StringFunction.is\_null(params[0]):  
 # return params[0]  
 #return re.sub(params[1], params[2], params[0], 0, flags=0)* **try**:  
 **return** params[0].replace(params[1], params[2],regex=**True**)  
 **except** Exception **as** ex:  
 logger.warning(**"regex\_replace exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
   
  
 @classmethod  
 **def** decode(cls, params):  
 *"""  
 This method is used to run ternary operator like if-else* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[1]) == pd.core.series.Series:  
 **raise** Exception(**"Param2 as a column is not supported, please provide scalar values !!!"**)  
 **if** type(params[2]) == pd.core.series.Series:  
 **raise** Exception(**"Param3 as a column is not supported, please provide scalar values !!!"**)  
  
 output\_series = params[0]   
 **if** type(output\_series) == pd.core.series.Series:  
 **if** output\_series.dtype == bool:  
 output\_series.loc[(output\_series==**True**)] = params[1]  
 output\_series.loc[(output\_series==**False**)] = params[2]  
 **return** output\_series  
 **else**:  
 **raise** Exception(**"Param1 not evaluating to boolean"**)  
 **else**:  
 **return** params[1] **if** str(output\_series) **in** [**'True'**,**'true'**,**'TRUE'**] **else** params[2]  
 **except** Exception **as** ex:  
 logger.warning(**"decode exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 @classmethod  
 **def** is\_null(cls, param):  
 **return** pd.isnull(param)  
  
  
 @classmethod  
 **def** find\_in\_set(cls, params):  
 *"""  
 This method is used check if data is present in set* **:param** *params:* **:return***:  
 """* **try**:  
 **if** type(params[0]) == pd.core.series.Series:  
 **if** type(params[1]) == pd.core.series.Series:  
 **return** params[0].isin(params[1].tolist())  
 *#return params[0] in params[1].tolist()* **else**:  
 **return** params[0].isin(params[1])  
 **else**:  
 **if** type(params[1]) == pd.core.series.Series:  
 **return** params[0] **in** params[1].tolist()  
 **else**:  
 **return** params[0] **in** params[1]  
 **except** Exception **as** ex:  
 logger.warning(**"find\_in\_set exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
   
  
 @classmethod  
 **def** nvl(cls, params):  
 *"""  
 This method is used check if data is present in set* **:param** *params:* **:return***:  
 """* **try**:  
 **if**((**not** str(params[1])) | (params[1] == pd.core.series.Series)):  
 **return** exception\_message+**" Replacement scalar is missing"** *#params[0] = pd.Series([numeric\_formatter(x) for x in params[0]])* **if** type(params[0]) == pd.core.series.Series:  
 **return** params[0].where(~((params[0].astype(str) == **''**) | (params[0].isnull())), params[1])  
 **else**:  
 **return** (params[1] **if** (params[0] == **'' or** pd.isnull(params[0])) **else** params[0])  
 **except** Exception **as** ex:  
 logger.warning(**"nvl exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** to\_string(cls, params):  
 *"""  
 This method is used check if data is present in set* **:param** *params:* **:return***:  
 """* **try**:  
 **if**(type(params[0]) == pd.core.series.Series):  
 temp = params[0].copy()  
 temp.loc[temp.notna()] = temp.astype(str)  
 temp.loc[temp.isna()] = **None  
 return** temp  
 **else**:  
 **return None if** (**not** params[0]) **else** str(params[0])  
 **except** Exception **as** ex:  
 logger.warning(**"to\_string exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 @classmethod  
 **def** to\_number(cls, params):  
 *"""  
 This method is used check if data is present in set* **:param** *params:* **:return***:  
 """* **def** numeric\_formatter(input\_string):  
 **try**:  
 input\_string\_floor = float(str(input\_string))  
 **if** (math.floor(input\_string\_floor) == input\_string\_floor):  
 var = str(int(input\_string\_floor))  
 **else**:  
 var = input\_string  
 **return** var  
 **except** Exception **as** ex:  
 logger.warning(**"numeric\_formatter worker : "**+str(ex))  
 **return** np.nan  
 **try**:  
 **if**(type(params[0]) == pd.core.series.Series):  
 **return** pd.Series(data = [numeric\_formatter(\_) **for** \_ **in** list(params[0])],index=params[0].index)  
 **else**:  
 **return** numeric\_formatter(params[0])  
 **except** Exception **as** ex:  
 logger.warning(**"to\_number exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))

**from** expression.ExpressionVisitor **import** ExpressionVisitor  
**from** expression.ExpressionParser **import** ExpressionParser  
**from** expression.function\_driver **import** FunctionDriver  
**from** expression.ExpressionLexer **import** ExpressionLexer  
  
**import** math  
**from** antlr4 **import** CommonTokenStream, InputStream  
**import** pandas **as** pd  
**import** numpy **as** np  
**import** re  
**import** logging  
**from** logging.handlers **import** RotatingFileHandler  
logger = logging.getLogger(**"test"**)  
logger.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'application.log'**, maxBytes=20000000, backupCount=10)  
formatter = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s'**,datefmt=**'%Y-%m-%d %H:%M:%S'**)  
HANDLER.setFormatter(formatter)  
logger.addHandler(HANDLER)  
exception\_message = **"Mosaic Exception :"  
  
  
  
  
class** EvalVisitor(ExpressionVisitor):  
 *"""  
 This is our custom Expression visitor class.  
 Here we have implemented all the compiler method of antlr  
 """* **def** \_\_init\_\_(self, variable\_values\_map):  
 self.memory = {}  
 self.registry\_map = variable\_values\_map  
 self.function\_driver = FunctionDriver()  
  
 **def** visitNumberAtom(self, ctx):  
 *"""  
 Visit method if number value is found* **:param** *ctx:* **:return***:  
 """  
 #print("in visitNumberAtom")* **try**:  
 **if** ctx.INT() **is not None**:  
 **return** int(ctx.getText())  
 **elif** ctx.DOUBLE() **is not None**:  
 **return** float(ctx.getText())  
 **except** Exception **as** ex:  
 logger.warning(**"visitNumberAtom "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 **def** visitIdAtom(self, ctx: ExpressionParser.IdAtomContext):  
 *"""  
 This method is used to find the unknown variable in the expression* **:param** *ctx:* **:return***:  
 """  
 #id = ctx.getText().lower()* id = ctx.getText()  
 **try**:  
 **if** self.memory.get(id) **is None**:  
 return\_value = self.lookup\_in\_registry\_variable\_list(id)  
 **if** return\_value **is None**:  
 **if** id != **'value'**:  
 **raise** Exception(**'Variable not found '** + id)   
 **else**:  
 **return** float(**"NaN"**)  
 **return** return\_value  
 **else**:  
 **return** self.memory.get(id)  
 **except** Exception **as** ex:  
 logger.warning(**"visitIdAtom "** + str(ex))  
 *#raise Exception(ex)* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
   
  
  
  
 **def** lookup\_in\_registry\_variable\_list(self, key):  
 *"""  
 This method is used for lookup with registry variables* **:param** *key:* **:return***:  
 """* **return** self.registry\_map.get(key)  
  
  
  
 **def** visitAssignmentExpr(self, ctx: ExpressionParser.AssignmentExprContext):  
 *"""  
 This method is called when = is using in the expression* **:param** *ctx:* **:return***:  
 """* **return** self.visit(ctx.assignment())  
  
 **def** visitAssignment(self, ctx: ExpressionParser.AssignmentContext):  
 *"""  
 This method is used whenever we do assignment operation* **:param** *ctx: git* **:return***:  
 """* key = ctx.ID().getText().lower()  
 value = self.visit(ctx.expr())  
 self.memory.update({key: value})  
 self.registry\_map[key]=value   
 **return** value  
  
 **def** visitStringAtom(self, ctx):  
 *"""  
 Visit this method when string value found in expression* **:param** *ctx:* **:return***:  
 """* **return** ctx.getText()[1:len(ctx.getText())-1].replace(**"\"\""**, **"\""**)  
  
 **def** visitBooleanAtom(self, ctx: ExpressionParser.BooleanAtomContext):  
 *"""  
 Visit thie method if expression find any true|False in expression* **:param** *ctx:* **:return***:  
 """* **return** self.strTobool(str(ctx.getText()))  
  
 **def** visitNilAtom(self, ctx: ExpressionParser.NilAtomContext):  
 **pass** *# return None* **def** visitParse(self, ctx):  
 *"""  
 Visit this method when parsing statement* **:param** *ctx:* **:return***:  
 """* **return** self.visit(ctx.block())  
  
 **def** visitStatement(self, ctx):  
 *"""  
 visit this method for the statement* **:param** *ctx:* **:return***:  
 """* **return** self.visit(ctx.expr())  
  
 **def** visitStatementExpr(self, ctx: ExpressionParser.StatementExprContext):  
 *"""  
 visit this method when statementexpr in expression* **:param** *ctx:* **:return***:  
 """* **return** self.visit(ctx.statement())  
  
 **def** visitParExpr(self, ctx: ExpressionParser.ParExprContext):  
 *"""  
 Antlr will run this method when expression has LPAREN and RPAREN* **:param** *ctx:* **:return***:  
 """* **return** self.visit(ctx.expr())  
  
 **def** visitNotExpr(self, ctx: ExpressionParser.NotExprContext):  
 *"""  
 Antlr will run this method when expression has a nagation operator* **:param** *ctx:* **:return***:  
 """* **try**:  
 value\_on\_not = self.visit(ctx.expr())  
 **if** type(value\_on\_not) == pd.core.series.Series:  
 **return** ~value\_on\_not  
 **else**:  
 **return not** value\_on\_not  
 **except** Exception **as** ex:  
 logger.warning(**"visitNotExpr exception : "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 **def** visitUnaryMinusExpr(self, ctx: ExpressionParser.UnaryMinusExprContext):  
 *"""  
 Antlr will run this method when expression have Unary operator* **:param** *ctx:* **:return***:  
 """* **try**:  
 value = self.visit(ctx.expr())  
 *# if type(value) is int or type(value) is float:* **return** -value  
  
 **except** Exception **as** ex:  
 logger.warning(**"visitUnaryMinusExpr exception : "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
  
 **def** visitAdditiveExpr(self, ctx):  
 *"""  
 Run this method when plus,minus operator is used in expression* **:param** *ctx:* **:return***:  
 """* **try**:  
 left = self.visit(ctx.expr(0))  
 right = self.visit(ctx.expr(1))  
 **if** type(right) == pd.core.series.Series:  
 right = pd.to\_numeric(right, errors = **"coerce"**)  
 **else**:  
 **if** str(right).strip(**'-'**).replace(**'.'**,**''**,1).isnumeric():  
 right = float(str(right))  
 **else**:  
 right = **None  
  
 if** type(left) == pd.core.series.Series:  
 left = pd.to\_numeric(left, errors = **"coerce"**)  
 **else**:  
 **if** str(left).strip(**'-'**).replace(**'.'**,**''**,1).isnumeric():  
 left = float(str(left))  
 **else**:  
 left = **None  
  
 if** ctx.op.type == ExpressionParser.PLUS:  
 **return** left + right  
 **elif** ctx.op.type == ExpressionParser.MINUS:  
 **return** left - right  
 **except** Exception **as** ex:  
 logger.warning(**"visitAdditiveExpr exception : "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 **def** visitMultiplicationExpr(self, ctx: ExpressionParser.MultiplicationExprContext):  
 *"""  
 This method is used for multiply and division operator* **:param** *ctx:* **:return***:  
 """* **try**:  
 left = self.visit(ctx.expr(0))  
 right = self.visit(ctx.expr(1))  
 **if** type(right) == pd.core.series.Series:  
 right = pd.to\_numeric(right, errors = **"coerce"**)  
 **else**:  
 **if** str(right).strip(**'-'**).replace(**'.'**,**''**,1).isnumeric():  
 right = float(str(right))  
 **else**:  
 right = **None  
  
 if** type(left) == pd.core.series.Series:  
 left = pd.to\_numeric(left, errors = **"coerce"**)  
 **else**:  
 **if** str(left).strip(**'-'**).replace(**'.'**,**''**,1).isnumeric():  
 left = float(str(left))  
 **else**:  
 left = **None  
   
 if** ctx.op.type == ExpressionParser.MULT:  
 **return** left \* right  
 **elif** ctx.op.type == ExpressionParser.DIV:  
 **return** left / right  
 **except** Exception **as** ex: *# pylint: disable=broad-except* logger.warning(**"visitMultiplicationExpr "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
   
 **def** check\_none\_condition(self, value):  
 *"""  
 This method is used for none condition on values* **:param** *value:* **:return***:  
 """* regex\_list = re.compile(**r'^\[(0-9|.|a-z|A-Z).\*\]$'**)  
  
 **if** value **is None**:  
 **return None  
 elif** (type(value.getText()) == str) & (bool(regex\_list.search(value.getText()))):  
 value = eval(value.getText())  
 **return** value  
 **else**:  
 **return** self.visit(value)  
  
  
 **def** visitRelationalExpr(self, ctx: ExpressionParser.RelationalExprContext):  
 *"""  
 Compare relation operator like > , < , >= and <=* **:param** *ctx:* **:return***:  
 """* **try**:   
 left = self.visit(ctx.expr(0))  
 right = self.visit(ctx.expr(1))  
  
 **if** (type(left) == pd.core.series.Series):  
 left = pd.to\_numeric(left, errors = **"coerce"**)  
 **else**:  
 left = float(left)   
   
 **if** (type(right) == pd.core.series.Series):  
 right = pd.to\_numeric(right, errors = **"coerce"**)   
 **else**:  
 right = float(right)  
   
 **if** ctx.op.type == ExpressionParser.GT:  
 **return** left > right  
 **elif** ctx.op.type == ExpressionParser.LT:  
 **return** left < right  
 **elif** ctx.op.type == ExpressionParser.GTEQ:  
 **return** left >= right  
 **elif** ctx.op.type == ExpressionParser.LTEQ:  
 **return** left <= right  
  
 **except** Exception **as** ex: *# pylint: disable=broad-except* logger.warning(**"visitRelationalExpr "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 **def** visitEqualityExpr(self, ctx: ExpressionParser.EqualityExprContext):  
 *"""  
 This method is used to compare expression for == and !=* **:param** *ctx:* **:return***:  
 """* **try**:  
 left = pd.to\_numeric(self.visit(ctx.expr(0)), errors=**'ignore'**)  
 right = pd.to\_numeric(self.visit(ctx.expr(1)), errors=**'ignore'**)  
 **if** type(left) == type(**None**):  
 left = np.nan  
 **else**:  
 **pass  
 if** type(right) == type(**None**):  
 right = np.nan  
 **else**:  
 **pass  
  
 if** ctx.op.type == ExpressionParser.EQ:  
 **return** left == right  
 **elif** ctx.op.type == ExpressionParser.NEQ:  
 **return** left != right  
 **except** Exception **as** ex: *# pylint: disable=broad-except* logger.warning(**"visitEqualityExpr "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 **def** visitAndExpr(self, ctx: ExpressionParser.AndExprContext):  
 *"""  
 visit this method when expression has and operator* **:param** *ctx:* **:return***:  
 """* **try**:  
 left = self.visit(ctx.expr(0))  
 right = self.visit(ctx.expr(1))  
 **if** (type(left) == pd.core.series.Series):  
 **if**(left.dtype == bool):  
 **pass  
 else**:  
 left = pd.Series([**True if** self.strTobool(str(x)) **else False for** x **in** left])  
 **else**:  
 left = self.strTobool(str(left))   
  
 **if** (type(right) == pd.core.series.Series):  
 **if**(right.dtype == bool):  
 **pass  
 else**:  
 right = pd.Series([**True if** self.strTobool(str(x)) **else False for** x **in** right])  
 **else**:  
 right = self.strTobool(str(right))   
 **return** (left & right)  
 **except** KeyError **as** ex:  
 logger.warning(**"visitAndExpr "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
   
  
 **def** strTobool(self, v):  
 **return** v.lower() **in** (**"yes"**, **"true"**, **"t"**, **"1"**)  
   
  
  
 **def** visitOrExpr(self, ctx: ExpressionParser.OrExprContext):  
 *"""  
 visit this method when expression has or operator* **:param** *ctx:* **:return***:  
 """* **try**:  
 left = self.visit(ctx.expr(0))  
 right = self.visit(ctx.expr(1))  
 **if** (type(left) == pd.core.series.Series):  
 **if**(left.dtype == bool):  
 **pass  
 else**:  
 left = pd.Series([**True if** self.strTobool(str(x)) **else False for** x **in** left])  
 **else**:  
 left = self.strTobool(str(left))   
  
 **if** (type(right) == pd.core.series.Series):  
 **if**(right.dtype == bool):  
 **pass  
 else**:  
 right = pd.Series([**True if** self.strTobool(str(x)) **else False for** x **in** right])  
 **else**:  
 right = self.strTobool(str(right))  
 **return** (left | right)  
 **except** KeyError **as** ex:  
 logger.warning(**"visitAndExpr "** + str(ex))  
 *#return "".join([exception\_message,str(ex)])* **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 **def** visitFunctionParam0(self, ctx: ExpressionParser.FunctionParam0Context):  
 *"""  
 Run this method whenever we use without parameter function in expression* **:param** *ctx:* **:return***:  
 """* **try**:  
 params\_list = [**None**] \* 1  
 params\_list[0] = self.registry\_map.index  
 values = self.function\_driver.execute\_function(ctx.FUNCTIONNAME\_PARAM0().getText(), params\_list)  
 **return** values  
 **except** Exception **as** ex:  
 logger.warning(**"visitFunctionParam0 "** + str(ex))  
 **raise** Exception(exception\_message + str(ex))  
  
  
  
 **def** visitFunctionParam1(self, ctx: ExpressionParser.FunctionParam1Context):  
 *"""  
 Run this method whenever we use single Parameter function in expression* **:param** *ctx:* **:return***:  
 """* **try**:  
 params\_list = [**None**] \* 2  
 params\_list[0] = self.check\_none\_condition(ctx.arguments1().param1)  
 params\_list[1] = self.registry\_map.index  
 values = self.function\_driver.execute\_function(ctx.FUNCTIONNAME\_PARAM1().getText(), params\_list)  
 **return** values  
 **except** Exception **as** ex:  
 logger.warning(**"visitFunctionParam1 "** + str(ex))  
 **raise** Exception(exception\_message + str(ex))  
  
  
  
 **def** visitFunctionParam2(self, ctx: ExpressionParser.FunctionParam2Context):  
 *"""  
 Run this method whenever we use two parameter function in expression.* **:param** *ctx:* **:return***:  
 """* **try**:  
 params\_list = [**None**] \* 3  
 params\_list[0] = self.check\_none\_condition(ctx.arguments2().param1)  
 params\_list[1] = self.check\_none\_condition(ctx.arguments2().param2)  
 params\_list[2] = self.registry\_map.index  
 values = self.function\_driver.execute\_function(ctx.FUNCTIONNAME\_PARAM2().getText(), params\_list)  
 **return** values  
 **except** Exception **as** ex:  
 logger.warning(**"visitFunctionParam2 "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 **def** visitFunctionParam3(self, ctx: ExpressionParser.FunctionParam3Context):  
 *"""  
 This function is used to accept the parameter from* **:param** *ctx:* **:return***:  
 """* **try**:  
 params\_list = [**None**] \* 4  
 params\_list[0] = self.check\_none\_condition(ctx.arguments3().param1)  
 params\_list[1] = self.check\_none\_condition(ctx.arguments3().param2)  
 params\_list[2] = self.check\_none\_condition(ctx.arguments3().param3)  
 **if** ctx.FUNCTIONNAME\_PARAM3().getText() == **"LOOK\_UP\_CHECK"**:  
 params\_list[3] = self.registry\_map  
 **elif** ctx.FUNCTIONNAME\_PARAM3().getText() **in** [**"XAVG"**,**"XCOUNT"**,**"XMAX"**,**"XMIN"**,**"XSUM"**,**"XUNIQUECOUNT"**]:  
   
 select = self.check\_none\_condition(ctx.arguments3().param1)  
 **if** type(select) != pd.core.series.Series:  
 params\_list[0] = self.run\_expression(select+**";"**) **if** select **else None   
 else**:  
 params\_list[0] = select  
  
 where = self.check\_none\_condition(ctx.arguments3().param2)  
 **if** type(where) != pd.core.series.Series:  
 params\_list[1] = self.run\_expression(where.replace(**"\\"**,**""**)+**";"**) **if** where **else None   
 else**:  
 params\_list[1] = where  
   
  
 grpby = self.check\_none\_condition(ctx.arguments3().param3)  
 **if** type(grpby) != pd.core.series.Series:  
 params\_list[2] = self.run\_expression(grpby+**";"**) **if** grpby **else None  
 else**:  
 params\_list[2] = grpby  
   
 **else**:  
 params\_list[3] = self.registry\_map.index  
   
 values = self.function\_driver.execute\_function(ctx.FUNCTIONNAME\_PARAM3().getText(), params\_list)  
 **return** values  
   
 **except** Exception **as** ex:  
 logger.warning(**"visitFunctionParam3 "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
  
 **def** visitFunctionParam4(self, ctx: ExpressionParser.FunctionParam4Context):  
 *"""  
 Run this method whenever we use four parameter in the function* **:param** *ctx:* **:return***:  
 """* **try**:  
 params\_list = [**None**] \* 5  
 params\_list[0] = self.check\_none\_condition(ctx.arguments4().param1)  
 params\_list[1] = self.check\_none\_condition(ctx.arguments4().param2)  
 params\_list[2] = self.check\_none\_condition(ctx.arguments4().param3)  
 params\_list[3] = self.check\_none\_condition(ctx.arguments4().param4)  
 params\_list[4] = self.registry\_map.index  
 values = self.function\_driver.execute\_function(ctx.FUNCTIONNAME\_PARAM4().getText(), params\_list)  
 **return** values  
 **except** Exception **as** ex:  
 logger.warning(**"visitFunctionParam3 "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 **def** visitFunctionParamN(self, ctx: ExpressionParser.FunctionParamNContext):  
 *"""  
 Run this method whenever we use n number of parameter in the function* **:param** *ctx:* **:return***:  
 """* **try**:  
 params = ctx.argumentsN().expr();   
 params\_list = [**None**] \* (len(params)+1)  
 counter = 0  
 **for** \_ **in** params:  
 params\_list[counter] = self.visit(\_)  
 counter += 1  
 params\_list[len(params\_list)-1] = self.registry\_map.index  
 values = self.function\_driver.execute\_function(ctx.FUNCTIONNAME\_PARAM\_N().getText(), params\_list)  
 **return** values  
 **except** Exception **as** ex:  
 logger.warning(**"visitFunctionParamN "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 **def** visitIfStatExpr(self, ctx: ExpressionParser.IfStatExprContext):  
 *"""  
 Run this method whenerver we use if expression* **:param** *ctx:* **:return***:  
 """* **try**:  
 **return** self.visit(ctx.ifStat())  
 **except** Exception **as** ex:  
 logger.warning(**"visitIfStatExpr "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 *## This is Series based operation* **def** visitIfStat\_bk(self, ctx: ExpressionParser.IfStatContext):  
   
 **try**:  
 dummyevaluation = pd.DataFrame(index = self.registry\_map.index.copy())  
 dummyevaluation[**'evaluated\_value'**] = **None** dummyevaluation[**'evaluated\_expression'**] = **None  
   
 for** condition\_block **in** ctx.conditionBlock():  
 dummyevaluation[**'evaluated\_expression'**] = self.visit(condition\_block.expr())  
 *#print(dummyevaluation['evaluated\_expression'].dtype)* **if**(dummyevaluation[**'evaluated\_expression'**].dtype!=bool):  
 **raise** Exception(**"Exception occured ! - Error in evaluating if statement"**)  
  
   
 ifTrueExp = self.check\_for\_outer\_braces(condition\_block.statBlock().getText())  
 dummyevaluation.loc[dummyevaluation.evaluated\_expression == **True**, **'evaluated\_value'** ] = self.run\_expression(ifTrueExp)  
 self.cache\_assignment\_variable(ifTrueExp,dummyevaluation[**'evaluated\_value'**] )  
   
 elseExp = self.check\_for\_outer\_braces(ctx.statBlock().getText())  
 dummyevaluation.loc[dummyevaluation.evaluated\_value.isnull(), **'evaluated\_value'** ] = self.run\_expression(elseExp)  
 **return** self.cache\_assignment\_variable(elseExp,dummyevaluation[**'evaluated\_value'**] )  
  
 **except** Exception **as** ex:  
 logger.warning(**"visitIfStat "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
 **def** visitIfStat(self, ctx: ExpressionParser.IfStatContext):  
   
 **try**:  
 dummyevaluation = pd.DataFrame(index = self.registry\_map.index.copy())  
 dummyevaluation[**'evaluated\_value'**] = **"No Value"** dummyevaluation[**'evaluated\_expression'**] = **None  
   
 for** condition\_block **in** ctx.conditionBlock():  
 dummyevaluation[**'evaluated\_expression'**] = self.visit(condition\_block.expr())  
 *#print(dummyevaluation['evaluated\_expression'].dtype)* **if**(dummyevaluation[**'evaluated\_expression'**].dtype!=bool):  
 **raise** Exception(**"Exception occured ! - Error in evaluating if statement"**)  
   
 ifTrueExp = self.check\_for\_outer\_braces(condition\_block.statBlock().getText())  
 dummyevaluation.loc[((dummyevaluation.evaluated\_expression == **True**) & (dummyevaluation.evaluated\_value == **"No Value"**)), **'evaluated\_value'** ] = self.run\_expression(ifTrueExp)  
 self.cache\_assignment\_variable(ifTrueExp,dummyevaluation[**'evaluated\_value'**] )  
 elseExp = self.check\_for\_outer\_braces(ctx.statBlock().getText())  
 dummyevaluation.loc[dummyevaluation.evaluated\_value == **"No Value"**, **'evaluated\_value'** ] = self.run\_expression(elseExp)  
 **return** self.cache\_assignment\_variable(elseExp,dummyevaluation[**'evaluated\_value'**] )  
  
 **except** Exception **as** ex:  
 logger.warning(**"visitIfStat "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
  
   
   
 **def** check\_for\_outer\_braces(self,input\_string):  
 **try**:  
 **if** input\_string.startswith(**'{'**) **and** input\_string.endswith(**'}'**):  
 input\_string = input\_string[1:-1]  
 **return** input\_string;  
 **except** Exception **as** ex:  
 logger.warning(**"check\_for\_outer\_braces "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))   
  
  
 **def** cache\_assignment\_variable(self, expression,value):  
 **try**:  
 **if 'if' in** expression.lower():  
 key=expression.split(**";"**)[-2].replace(**"}"**,**""**).strip().lower()  
 self.registry\_map[key]=value  
 self.memory.update({key: value})  
 **return** self.registry\_map[key]  
 **else**:  
 **if**(expression.count(**"="**) == 1):  
 key=expression.split(**"="**)[0].lower()  
 self.registry\_map[key]=value  
 self.memory.update({key: value})  
 **return** self.registry\_map[key]  
 **except** Exception **as** ex:  
 logger.warning(**"cache\_assignment\_variable "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))   
  
  
 **def** run\_expression(self, expression):  
 *"""  
 This method is used to run the expression* **:param** *register\_data\_set:* **:return***:  
 """* **try**:  
 lexer = ExpressionLexer(InputStream(expression))  
 parser = ExpressionParser(CommonTokenStream(lexer))  
 eval\_visit = EvalVisitor(self.registry\_map)  
 tree = parser.parse()  
 result = eval\_visit.visit(tree)  
 **return** result  
 **except** KeyError **as** ex:  
 logger.warning(**"run\_expression function not available "** + str(ex))  
 **raise** Exception(ex.args[0] + **" function not available"**)  
 **except** Exception **as** ex:  
 logger.warning(**"run\_expression "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))  
  
 **def** visitFunctionExpr(self, ctx: ExpressionParser.FunctionExprContext):  
 *"""  
 Antlr will run this method when function is used as expression* **:param** *ctx:* **:return***:  
 """* **return** self.visit(ctx.functions())  
  
 **def** visitPowExpr(self, ctx: ExpressionParser.PowExprContext):  
 *"""  
 This method is used to calculate the power in expression* **:param** *ctx:* **:return***:  
 """* **def** numeric\_formatter(input\_string):  
 **try**:  
 input\_string\_floor = float(str(input\_string))  
 **if** (math.floor(input\_string\_floor) == input\_string\_floor):  
 var = int(input\_string\_floor)  
 **else**:  
 var = input\_string\_floor  
 **return** var  
 **except** Exception **as** ex:  
 logger.warning(**"numeric\_formatter "** + str(ex))  
 **return** np.nan  
  
 left\_value = self.visit(ctx.expr(0))  
 right\_value = self.visit(ctx.expr(1))  
  
 **try**:  
 **if** (left\_value == pd.core.series.Series) & (right\_value == pd.core.series.Series):  
 left\_value = pd.to\_numeric(left\_value, errors=**"coerce"**)  
 right\_value = pd.to\_numeric(right\_value, errors=**"coerce"**)  
 **return** left\_value\*\*right\_value  
 **elif** (left\_value == pd.core.series.Series) & (right\_value != pd.core.series.Series):  
 left\_value = pd.to\_numeric(ctx.expr(0), errors=**"coerce"**)  
 right\_value = numeric\_formatter(ctx.expr(1))  
 **return** left\_value\*\*right\_value  
 **elif** (left\_value != pd.core.series.Series) & (right\_value == pd.core.series.Series):  
 left\_value = numeric\_formatter(ctx.expr(0))  
 right\_value = pd.to\_numeric(ctx.expr(1), errors=**"coerce"**)  
 **return** left\_value\*\*right\_value  
 **else**:  
 left\_value = numeric\_formatter(ctx.expr(0))  
 right\_value = numeric\_formatter(ctx.expr(1))  
 **return** left\_value\*\*right\_value  
 **except** Exception **as** ex:  
 logger.warning(**"visit\_power exception : "** + str(ex))  
 **raise** Exception(**""**.join([exception\_message,str(ex)]))

*# Generated from Expression.g4 by ANTLR 4.7.1***from** antlr4 **import** \*  
**from** io **import** StringIO  
**from** typing.io **import** TextIO  
**import** sys  
  
  
**def** serializedATN():  
 **with** StringIO() **as** buf:  
 buf.write(**"\3\u608b\ua72a\u8133\ub9ed\u417c\u3be7\u7786\u5964\2-"**)  
 buf.write(**"\u033f\b\1\4\2\t\2\4\3\t\3\4\4\t\4\4\5\t\5\4\6\t\6\4\7"**)  
 buf.write(**"\t\7\4\b\t\b\4\t\t\t\4\n\t\n\4\13\t\13\4\f\t\f\4\r\t\r"**)  
 buf.write(**"\4\16\t\16\4\17\t\17\4\20\t\20\4\21\t\21\4\22\t\22\4\23"**)  
 buf.write(**"\t\23\4\24\t\24\4\25\t\25\4\26\t\26\4\27\t\27\4\30\t\30"**)  
 buf.write(**"\4\31\t\31\4\32\t\32\4\33\t\33\4\34\t\34\4\35\t\35\4\36"**)  
 buf.write(**"\t\36\4\37\t\37\4 \t \4!\t!\4\"\t\"\4#\t#\4$\t$\4%\t%"**)  
 buf.write(**"\4&\t&\4\'\t\'\4(\t(\4)\t)\4\*\t\*\4+\t+\4,\t,\3\2\3\2\3"**)  
 buf.write(**"\2\3\2\3\2\3\2\3\2\3\2\3\2\3\2\3\2\3\2\5\2f\n\2\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3"**)  
 buf.write(**"\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\3\5\3"**)  
 buf.write(**"\u014e\n\3\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4"**)  
 buf.write(**"\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3"**)  
 buf.write(**"\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4"**)  
 buf.write(**"\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3"**)  
 buf.write(**"\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4"**)  
 buf.write(**"\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3"**)  
 buf.write(**"\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4"**)  
 buf.write(**"\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3\4\3"**)  
 buf.write(**"\4\3\4\3\4\3\4\5\4\u01bd\n\4\3\5\3\5\3\5\3\5\3\5\3\5\3"**)  
 buf.write(**"\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5"**)  
 buf.write(**"\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3"**)  
 buf.write(**"\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5"**)  
 buf.write(**"\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3"**)  
 buf.write(**"\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5"**)  
 buf.write(**"\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3"**)  
 buf.write(**"\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5"**)  
 buf.write(**"\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3"**)  
 buf.write(**"\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5"**)  
 buf.write(**"\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\3\5\5\5\u024a"**)  
 buf.write(**"\n\5\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3"**)  
 buf.write(**"\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6"**)  
 buf.write(**"\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\3\6\5"**)  
 buf.write(**"\6\u0273\n\6\3\7\3\7\3\7\3\7\3\7\3\7\3\7\3\7\3\7\3\7\3"**)  
 buf.write(**"\7\3\7\3\7\3\7\3\7\3\7\3\7\3\7\3\7\5\7\u0288\n\7\3\b\3"**)  
 buf.write(**"\b\3\b\3\b\3\b\3\b\5\b\u0290\n\b\3\t\3\t\3\t\3\t\3\t\3"**)  
 buf.write(**"\t\3\t\3\t\5\t\u029a\n\t\3\n\3\n\3\n\3\13\3\13\3\13\3"**)  
 buf.write(**"\f\3\f\3\r\3\r\3\16\3\16\3\16\3\17\3\17\3\17\3\20\3\20"**)  
 buf.write(**"\3\21\3\21\3\22\3\22\3\23\3\23\3\24\3\24\3\25\3\25\3\26"**)  
 buf.write(**"\3\26\3\27\3\27\3\30\3\30\3\31\3\31\3\32\3\32\3\33\3\33"**)  
 buf.write(**"\3\34\3\34\3\35\3\35\3\36\3\36\3\37\3\37\3 \3 \3 \3 \3"**)  
 buf.write(**" \3 \3 \3 \3 \3 \3 \3 \5 \u02d8\n \3!\3!\3!\3!\3!\3!\3"**)  
 buf.write(**"!\3!\3!\3!\3!\3!\3!\3!\3!\5!\u02e9\n!\3\"\3\"\3\"\3\""**)  
 buf.write(**"\3\"\3#\3#\3#\3#\5#\u02f4\n#\3$\3$\3$\3$\3$\3$\3$\3$\5"**)  
 buf.write(**"$\u02fe\n$\3%\3%\3%\3%\3%\3%\3&\3&\7&\u0308\n&\f&\16&"**)  
 buf.write(**"\u030b\13&\3\'\6\'\u030e\n\'\r\'\16\'\u030f\3(\6(\u0313"**)  
 buf.write(**"\n(\r(\16(\u0314\3(\3(\7(\u0319\n(\f(\16(\u031c\13(\3"**)  
 buf.write(**"(\3(\6(\u0320\n(\r(\16(\u0321\5(\u0324\n(\3)\3)\3)\3)"**)  
 buf.write(**"\7)\u032a\n)\f)\16)\u032d\13)\3)\3)\3\*\3\*\7\*\u0333\n\*"**)  
 buf.write(**"\f\*\16\*\u0336\13\*\3\*\3\*\3+\3+\3+\3+\3,\3,\2\2-\3\3\5\4"**)  
 buf.write(**"\7\5\t\6\13\7\r\b\17\t\21\n\23\13\25\f\27\r\31\16\33\17"**)  
 buf.write(**"\35\20\37\21!\22#\23%\24\'\25)\26+\27-\30/\31\61\32\63"**)  
 buf.write(**"\33\65\34\67\359\36;\37= ?!A\"C#E$G%I&K\'M(O)Q\*S+U,W-"**)  
 buf.write(**"\3\2\b\5\2C\\aac|\6\2\62;C\\aac|\3\2\62;\5\2\f\f\17\17"**)  
 buf.write(**"$$\4\2\f\f\17\17\5\2\13\f\17\17\"\"\2\u03a1\2\3\3\2\2"**)  
 buf.write(**"\2\2\5\3\2\2\2\2\7\3\2\2\2\2\t\3\2\2\2\2\13\3\2\2\2\2"**)  
 buf.write(**"\r\3\2\2\2\2\17\3\2\2\2\2\21\3\2\2\2\2\23\3\2\2\2\2\25"**)  
 buf.write(**"\3\2\2\2\2\27\3\2\2\2\2\31\3\2\2\2\2\33\3\2\2\2\2\35\3"**)  
 buf.write(**"\2\2\2\2\37\3\2\2\2\2!\3\2\2\2\2#\3\2\2\2\2%\3\2\2\2\2"**)  
 buf.write(**"\'\3\2\2\2\2)\3\2\2\2\2+\3\2\2\2\2-\3\2\2\2\2/\3\2\2\2"**)  
 buf.write(**"\2\61\3\2\2\2\2\63\3\2\2\2\2\65\3\2\2\2\2\67\3\2\2\2\2"**)  
 buf.write(**"9\3\2\2\2\2;\3\2\2\2\2=\3\2\2\2\2?\3\2\2\2\2A\3\2\2\2"**)  
 buf.write(**"\2C\3\2\2\2\2E\3\2\2\2\2G\3\2\2\2\2I\3\2\2\2\2K\3\2\2"**)  
 buf.write(**"\2\2M\3\2\2\2\2O\3\2\2\2\2Q\3\2\2\2\2S\3\2\2\2\2U\3\2"**)  
 buf.write(**"\2\2\2W\3\2\2\2\3e\3\2\2\2\5\u014d\3\2\2\2\7\u01bc\3\2"**)  
 buf.write(**"\2\2\t\u0249\3\2\2\2\13\u0272\3\2\2\2\r\u0287\3\2\2\2"**)  
 buf.write(**"\17\u028f\3\2\2\2\21\u0299\3\2\2\2\23\u029b\3\2\2\2\25"**)  
 buf.write(**"\u029e\3\2\2\2\27\u02a1\3\2\2\2\31\u02a3\3\2\2\2\33\u02a5"**)  
 buf.write(**"\3\2\2\2\35\u02a8\3\2\2\2\37\u02ab\3\2\2\2!\u02ad\3\2"**)  
 buf.write(**"\2\2#\u02af\3\2\2\2%\u02b1\3\2\2\2\'\u02b3\3\2\2\2)\u02b5"**)  
 buf.write(**"\3\2\2\2+\u02b7\3\2\2\2-\u02b9\3\2\2\2/\u02bb\3\2\2\2"**)  
 buf.write(**"\61\u02bd\3\2\2\2\63\u02bf\3\2\2\2\65\u02c1\3\2\2\2\67"**)  
 buf.write(**"\u02c3\3\2\2\29\u02c5\3\2\2\2;\u02c7\3\2\2\2=\u02c9\3"**)  
 buf.write(**"\2\2\2?\u02d7\3\2\2\2A\u02e8\3\2\2\2C\u02ea\3\2\2\2E\u02f3"**)  
 buf.write(**"\3\2\2\2G\u02fd\3\2\2\2I\u02ff\3\2\2\2K\u0305\3\2\2\2"**)  
 buf.write(**"M\u030d\3\2\2\2O\u0323\3\2\2\2Q\u0325\3\2\2\2S\u0330\3"**)  
 buf.write(**"\2\2\2U\u0339\3\2\2\2W\u033d\3\2\2\2YZ\7T\2\2Z[\7C\2\2"**)  
 buf.write(**"[\\\7P\2\2\\f\7F\2\2]^\7P\2\2^\_\7Q\2\2\_f\7Y\2\2`a\7V\2"**)  
 buf.write(**"\2ab\7Q\2\2bc\7F\2\2cd\7C\2\2df\7[\2\2eY\3\2\2\2e]\3\2"**)  
 buf.write(**"\2\2e`\3\2\2\2f\4\3\2\2\2gh\7N\2\2hi\7G\2\2ij\7P\2\2j"**)  
 buf.write(**"k\7I\2\2kl\7V\2\2l\u014e\7J\2\2mn\7N\2\2no\7Q\2\2op\7"**)  
 buf.write(**"Y\2\2pq\7G\2\2q\u014e\7T\2\2rs\7W\2\2st\7R\2\2tu\7R\2"**)  
 buf.write(**"\2uv\7G\2\2v\u014e\7T\2\2wx\7T\2\2xy\7G\2\2yz\7X\2\2z"**)  
 buf.write(**"{\7G\2\2{|\7T\2\2|}\7U\2\2}\u014e\7G\2\2~\177\7V\2\2\177"**)  
 buf.write(**"\u0080\7T\2\2\u0080\u0081\7K\2\2\u0081\u014e\7O\2\2\u0082"**)  
 buf.write(**"\u0083\7N\2\2\u0083\u0084\7V\2\2\u0084\u0085\7T\2\2\u0085"**)  
 buf.write(**"\u0086\7K\2\2\u0086\u014e\7O\2\2\u0087\u0088\7T\2\2\u0088"**)  
 buf.write(**"\u0089\7V\2\2\u0089\u008a\7T\2\2\u008a\u008b\7K\2\2\u008b"**)  
 buf.write(**"\u014e\7O\2\2\u008c\u008d\7K\2\2\u008d\u008e\7P\2\2\u008e"**)  
 buf.write(**"\u008f\7K\2\2\u008f\u0090\7V\2\2\u0090\u0091\7E\2\2\u0091"**)  
 buf.write(**"\u0092\7C\2\2\u0092\u014e\7R\2\2\u0093\u0094\7U\2\2\u0094"**)  
 buf.write(**"\u0095\7Q\2\2\u0095\u0096\7W\2\2\u0096\u0097\7P\2\2\u0097"**)  
 buf.write(**"\u0098\7F\2\2\u0098\u0099\7G\2\2\u0099\u014e\7Z\2\2\u009a"**)  
 buf.write(**"\u009b\7K\2\2\u009b\u009c\7U\2\2\u009c\u009d\7D\2\2\u009d"**)  
 buf.write(**"\u009e\7N\2\2\u009e\u009f\7C\2\2\u009f\u00a0\7P\2\2\u00a0"**)  
 buf.write(**"\u014e\7M\2\2\u00a1\u00a2\7K\2\2\u00a2\u00a3\7U\2\2\u00a3"**)  
 buf.write(**"\u00a4\7P\2\2\u00a4\u00a5\7Q\2\2\u00a5\u00a6\7V\2\2\u00a6"**)  
 buf.write(**"\u00a7\7D\2\2\u00a7\u00a8\7N\2\2\u00a8\u00a9\7C\2\2\u00a9"**)  
 buf.write(**"\u00aa\7P\2\2\u00aa\u014e\7M\2\2\u00ab\u00ac\7U\2\2\u00ac"**)  
 buf.write(**"\u00ad\7R\2\2\u00ad\u00ae\7C\2\2\u00ae\u00af\7E\2\2\u00af"**)  
 buf.write(**"\u014e\7G\2\2\u00b0\u00b1\7E\2\2\u00b1\u00b2\7Q\2\2\u00b2"**)  
 buf.write(**"\u014e\7U\2\2\u00b3\u00b4\7V\2\2\u00b4\u00b5\7C\2\2\u00b5"**)  
 buf.write(**"\u014e\7P\2\2\u00b6\u00b7\7U\2\2\u00b7\u00b8\7K\2\2\u00b8"**)  
 buf.write(**"\u014e\7P\2\2\u00b9\u00ba\7C\2\2\u00ba\u00bb\7E\2\2\u00bb"**)  
 buf.write(**"\u00bc\7Q\2\2\u00bc\u014e\7U\2\2\u00bd\u00be\7C\2\2\u00be"**)  
 buf.write(**"\u00bf\7V\2\2\u00bf\u00c0\7C\2\2\u00c0\u014e\7P\2\2\u00c1"**)  
 buf.write(**"\u00c2\7C\2\2\u00c2\u00c3\7U\2\2\u00c3\u00c4\7K\2\2\u00c4"**)  
 buf.write(**"\u014e\7P\2\2\u00c5\u00c6\7N\2\2\u00c6\u00c7\7Q\2\2\u00c7"**)  
 buf.write(**"\u014e\7I\2\2\u00c8\u00c9\7N\2\2\u00c9\u00ca\7Q\2\2\u00ca"**)  
 buf.write(**"\u00cb\7I\2\2\u00cb\u014e\7\64\2\2\u00cc\u00cd\7N\2\2"**)  
 buf.write(**"\u00cd\u00ce\7Q\2\2\u00ce\u00cf\7I\2\2\u00cf\u00d0\7\63"**)  
 buf.write(**"\2\2\u00d0\u014e\7\62\2\2\u00d1\u00d2\7U\2\2\u00d2\u00d3"**)  
 buf.write(**"\7S\2\2\u00d3\u00d4\7T\2\2\u00d4\u014e\7V\2\2\u00d5\u00d6"**)  
 buf.write(**"\7N\2\2\u00d6\u014e\7P\2\2\u00d7\u00d8\7C\2\2\u00d8\u00d9"**)  
 buf.write(**"\7D\2\2\u00d9\u014e\7U\2\2\u00da\u00db\7H\2\2\u00db\u00dc"**)  
 buf.write(**"\7C\2\2\u00dc\u00dd\7E\2\2\u00dd\u014e\7V\2\2\u00de\u00df"**)  
 buf.write(**"\7T\2\2\u00df\u00e0\7C\2\2\u00e0\u00e1\7P\2\2\u00e1\u014e"**)  
 buf.write(**"\7F\2\2\u00e2\u00e3\7H\2\2\u00e3\u00e4\7N\2\2\u00e4\u00e5"**)  
 buf.write(**"\7Q\2\2\u00e5\u00e6\7Q\2\2\u00e6\u014e\7T\2\2\u00e7\u00e8"**)  
 buf.write(**"\7E\2\2\u00e8\u00e9\7G\2\2\u00e9\u00ea\7K\2\2\u00ea\u014e"**)  
 buf.write(**"\7N\2\2\u00eb\u00ec\7[\2\2\u00ec\u00ed\7G\2\2\u00ed\u00ee"**)  
 buf.write(**"\7C\2\2\u00ee\u014e\7T\2\2\u00ef\u00f0\7K\2\2\u00f0\u00f1"**)  
 buf.write(**"\7P\2\2\u00f1\u00f2\7X\2\2\u00f2\u00f3\7G\2\2\u00f3\u00f4"**)  
 buf.write(**"\7T\2\2\u00f4\u00f5\7U\2\2\u00f5\u00f6\7K\2\2\u00f6\u00f7"**)  
 buf.write(**"\7Q\2\2\u00f7\u00f8\7P\2\2\u00f8\u00f9\7E\2\2\u00f9\u00fa"**)  
 buf.write(**"\7J\2\2\u00fa\u00fb\7G\2\2\u00fb\u00fc\7E\2\2\u00fc\u014e"**)  
 buf.write(**"\7M\2\2\u00fd\u00fe\7G\2\2\u00fe\u00ff\7Z\2\2\u00ff\u014e"**)  
 buf.write(**"\7R\2\2\u0100\u0101\7V\2\2\u0101\u0102\7Q\2\2\u0102\u0103"**)  
 buf.write(**"\7a\2\2\u0103\u0104\7U\2\2\u0104\u0105\7V\2\2\u0105\u0106"**)  
 buf.write(**"\7T\2\2\u0106\u0107\7K\2\2\u0107\u0108\7P\2\2\u0108\u014e"**)  
 buf.write(**"\7I\2\2\u0109\u010a\7V\2\2\u010a\u010b\7Q\2\2\u010b\u010c"**)  
 buf.write(**"\7a\2\2\u010c\u010d\7P\2\2\u010d\u010e\7W\2\2\u010e\u010f"**)  
 buf.write(**"\7O\2\2\u010f\u0110\7D\2\2\u0110\u0111\7G\2\2\u0111\u014e"**)  
 buf.write(**"\7T\2\2\u0112\u0113\7F\2\2\u0113\u0114\7C\2\2\u0114\u014e"**)  
 buf.write(**"\7[\2\2\u0115\u0116\7O\2\2\u0116\u0117\7Q\2\2\u0117\u0118"**)  
 buf.write(**"\7P\2\2\u0118\u0119\7V\2\2\u0119\u014e\7J\2\2\u011a\u011b"**)  
 buf.write(**"\7V\2\2\u011b\u011c\7K\2\2\u011c\u011d\7O\2\2\u011d\u014e"**)  
 buf.write(**"\7G\2\2\u011e\u011f\7Y\2\2\u011f\u0120\7G\2\2\u0120\u0121"**)  
 buf.write(**"\7G\2\2\u0121\u0122\7M\2\2\u0122\u0123\7P\2\2\u0123\u0124"**)  
 buf.write(**"\7W\2\2\u0124\u014e\7O\2\2\u0125\u0126\7J\2\2\u0126\u0127"**)  
 buf.write(**"\7Q\2\2\u0127\u0128\7W\2\2\u0128\u014e\7T\2\2\u0129\u012a"**)  
 buf.write(**"\7O\2\2\u012a\u012b\7K\2\2\u012b\u012c\7P\2\2\u012c\u012d"**)  
 buf.write(**"\7W\2\2\u012d\u012e\7V\2\2\u012e\u014e\7G\2\2\u012f\u0130"**)  
 buf.write(**"\7U\2\2\u0130\u0131\7G\2\2\u0131\u0132\7E\2\2\u0132\u0133"**)  
 buf.write(**"\7Q\2\2\u0133\u0134\7P\2\2\u0134\u014e\7F\2\2\u0135\u0136"**)  
 buf.write(**"\7F\2\2\u0136\u0137\7C\2\2\u0137\u0138\7[\2\2\u0138\u0139"**)  
 buf.write(**"\7P\2\2\u0139\u013a\7C\2\2\u013a\u013b\7O\2\2\u013b\u014e"**)  
 buf.write(**"\7G\2\2\u013c\u013d\7O\2\2\u013d\u013e\7Q\2\2\u013e\u013f"**)  
 buf.write(**"\7P\2\2\u013f\u0140\7V\2\2\u0140\u0141\7J\2\2\u0141\u0142"**)  
 buf.write(**"\7P\2\2\u0142\u0143\7C\2\2\u0143\u0144\7O\2\2\u0144\u014e"**)  
 buf.write(**"\7G\2\2\u0145\u0146\7F\2\2\u0146\u0147\7C\2\2\u0147\u0148"**)  
 buf.write(**"\7V\2\2\u0148\u0149\7G\2\2\u0149\u014a\7V\2\2\u014a\u014b"**)  
 buf.write(**"\7K\2\2\u014b\u014c\7O\2\2\u014c\u014e\7G\2\2\u014dg\3"**)  
 buf.write(**"\2\2\2\u014dm\3\2\2\2\u014dr\3\2\2\2\u014dw\3\2\2\2\u014d"**)  
 buf.write(**"~\3\2\2\2\u014d\u0082\3\2\2\2\u014d\u0087\3\2\2\2\u014d"**)  
 buf.write(**"\u008c\3\2\2\2\u014d\u0093\3\2\2\2\u014d\u009a\3\2\2\2"**)  
 buf.write(**"\u014d\u00a1\3\2\2\2\u014d\u00ab\3\2\2\2\u014d\u00b0\3"**)  
 buf.write(**"\2\2\2\u014d\u00b3\3\2\2\2\u014d\u00b6\3\2\2\2\u014d\u00b9"**)  
 buf.write(**"\3\2\2\2\u014d\u00bd\3\2\2\2\u014d\u00c1\3\2\2\2\u014d"**)  
 buf.write(**"\u00c5\3\2\2\2\u014d\u00c8\3\2\2\2\u014d\u00cc\3\2\2\2"**)  
 buf.write(**"\u014d\u00d1\3\2\2\2\u014d\u00d5\3\2\2\2\u014d\u00d7\3"**)  
 buf.write(**"\2\2\2\u014d\u00da\3\2\2\2\u014d\u00de\3\2\2\2\u014d\u00e2"**)  
 buf.write(**"\3\2\2\2\u014d\u00e7\3\2\2\2\u014d\u00eb\3\2\2\2\u014d"**)  
 buf.write(**"\u00ef\3\2\2\2\u014d\u00fd\3\2\2\2\u014d\u0100\3\2\2\2"**)  
 buf.write(**"\u014d\u0109\3\2\2\2\u014d\u0112\3\2\2\2\u014d\u0115\3"**)  
 buf.write(**"\2\2\2\u014d\u011a\3\2\2\2\u014d\u011e\3\2\2\2\u014d\u0125"**)  
 buf.write(**"\3\2\2\2\u014d\u0129\3\2\2\2\u014d\u012f\3\2\2\2\u014d"**)  
 buf.write(**"\u0135\3\2\2\2\u014d\u013c\3\2\2\2\u014d\u0145\3\2\2\2"**)  
 buf.write(**"\u014e\6\3\2\2\2\u014f\u0150\7T\2\2\u0150\u0151\7G\2\2"**)  
 buf.write(**"\u0151\u0152\7R\2\2\u0152\u0153\7G\2\2\u0153\u0154\7C"**)  
 buf.write(**"\2\2\u0154\u01bd\7V\2\2\u0155\u0156\7K\2\2\u0156\u0157"**)  
 buf.write(**"\7P\2\2\u0157\u0158\7U\2\2\u0158\u0159\7V\2\2\u0159\u01bd"**)  
 buf.write(**"\7T\2\2\u015a\u015b\7N\2\2\u015b\u015c\7G\2\2\u015c\u015d"**)  
 buf.write(**"\7X\2\2\u015d\u015e\7G\2\2\u015e\u015f\7P\2\2\u015f\u0160"**)  
 buf.write(**"\7U\2\2\u0160\u0161\7J\2\2\u0161\u0162\7V\2\2\u0162\u0163"**)  
 buf.write(**"\7G\2\2\u0163\u0164\7K\2\2\u0164\u01bd\7P\2\2\u0165\u0166"**)  
 buf.write(**"\7T\2\2\u0166\u0167\7G\2\2\u0167\u0168\7I\2\2\u0168\u0169"**)  
 buf.write(**"\7G\2\2\u0169\u016a\7Z\2\2\u016a\u016b\7a\2\2\u016b\u016c"**)  
 buf.write(**"\7O\2\2\u016c\u016d\7C\2\2\u016d\u016e\7V\2\2\u016e\u016f"**)  
 buf.write(**"\7E\2\2\u016f\u01bd\7J\2\2\u0170\u0171\7T\2\2\u0171\u0172"**)  
 buf.write(**"\7Q\2\2\u0172\u0173\7W\2\2\u0173\u0174\7P\2\2\u0174\u01bd"**)  
 buf.write(**"\7F\2\2\u0175\u0176\7T\2\2\u0176\u0177\7Q\2\2\u0177\u0178"**)  
 buf.write(**"\7W\2\2\u0178\u0179\7P\2\2\u0179\u017a\7F\2\2\u017a\u01bd"**)  
 buf.write(**"\7R\2\2\u017b\u017c\7V\2\2\u017c\u017d\7Q\2\2\u017d\u017e"**)  
 buf.write(**"\7a\2\2\u017e\u017f\7F\2\2\u017f\u0180\7C\2\2\u0180\u0181"**)  
 buf.write(**"\7V\2\2\u0181\u01bd\7G\2\2\u0182\u0183\7C\2\2\u0183\u0184"**)  
 buf.write(**"\7F\2\2\u0184\u0185\7F\2\2\u0185\u0186\7a\2\2\u0186\u0187"**)  
 buf.write(**"\7O\2\2\u0187\u0188\7Q\2\2\u0188\u0189\7P\2\2\u0189\u018a"**)  
 buf.write(**"\7V\2\2\u018a\u018b\7J\2\2\u018b\u01bd\7U\2\2\u018c\u018d"**)  
 buf.write(**"\7F\2\2\u018d\u018e\7C\2\2\u018e\u018f\7V\2\2\u018f\u0190"**)  
 buf.write(**"\7G\2\2\u0190\u0191\7F\2\2\u0191\u0192\7K\2\2\u0192\u0193"**)  
 buf.write(**"\7H\2\2\u0193\u01bd\7H\2\2\u0194\u0195\7D\2\2\u0195\u0196"**)  
 buf.write(**"\7Q\2\2\u0196\u0197\7Q\2\2\u0197\u0198\7N\2\2\u0198\u0199"**)  
 buf.write(**"\7a\2\2\u0199\u019a\7K\2\2\u019a\u019b\7P\2\2\u019b\u019c"**)  
 buf.write(**"\7a\2\2\u019c\u019d\7N\2\2\u019d\u019e\7Q\2\2\u019e\u019f"**)  
 buf.write(**"\7Q\2\2\u019f\u01a0\7M\2\2\u01a0\u01a1\7W\2\2\u01a1\u01bd"**)  
 buf.write(**"\7R\2\2\u01a2\u01a3\7P\2\2\u01a3\u01a4\7X\2\2\u01a4\u01bd"**)  
 buf.write(**"\7N\2\2\u01a5\u01a6\7H\2\2\u01a6\u01a7\7K\2\2\u01a7\u01a8"**)  
 buf.write(**"\7P\2\2\u01a8\u01a9\7F\2\2\u01a9\u01aa\7a\2\2\u01aa\u01ab"**)  
 buf.write(**"\7K\2\2\u01ab\u01ac\7P\2\2\u01ac\u01ad\7a\2\2\u01ad\u01ae"**)  
 buf.write(**"\7U\2\2\u01ae\u01af\7G\2\2\u01af\u01bd\7V\2\2\u01b0\u01b1"**)  
 buf.write(**"\7U\2\2\u01b1\u01b2\7R\2\2\u01b2\u01b3\7N\2\2\u01b3\u01b4"**)  
 buf.write(**"\7K\2\2\u01b4\u01bd\7V\2\2\u01b5\u01b6\7F\2\2\u01b6\u01b7"**)  
 buf.write(**"\7C\2\2\u01b7\u01b8\7V\2\2\u01b8\u01b9\7G\2\2\u01b9\u01ba"**)  
 buf.write(**"\7C\2\2\u01ba\u01bb\7F\2\2\u01bb\u01bd\7F\2\2\u01bc\u014f"**)  
 buf.write(**"\3\2\2\2\u01bc\u0155\3\2\2\2\u01bc\u015a\3\2\2\2\u01bc"**)  
 buf.write(**"\u0165\3\2\2\2\u01bc\u0170\3\2\2\2\u01bc\u0175\3\2\2\2"**)  
 buf.write(**"\u01bc\u017b\3\2\2\2\u01bc\u0182\3\2\2\2\u01bc\u018c\3"**)  
 buf.write(**"\2\2\2\u01bc\u0194\3\2\2\2\u01bc\u01a2\3\2\2\2\u01bc\u01a5"**)  
 buf.write(**"\3\2\2\2\u01bc\u01b0\3\2\2\2\u01bc\u01b5\3\2\2\2\u01bd"**)  
 buf.write(**"\b\3\2\2\2\u01be\u01bf\7F\2\2\u01bf\u01c0\7G\2\2\u01c0"**)  
 buf.write(**"\u01c1\7E\2\2\u01c1\u01c2\7Q\2\2\u01c2\u01c3\7F\2\2\u01c3"**)  
 buf.write(**"\u024a\7G\2\2\u01c4\u01c5\7N\2\2\u01c5\u01c6\7Q\2\2\u01c6"**)  
 buf.write(**"\u01c7\7E\2\2\u01c7\u01c8\7C\2\2\u01c8\u01c9\7V\2\2\u01c9"**)  
 buf.write(**"\u024a\7G\2\2\u01ca\u01cb\7T\2\2\u01cb\u01cc\7R\2\2\u01cc"**)  
 buf.write(**"\u01cd\7C\2\2\u01cd\u024a\7F\2\2\u01ce\u01cf\7N\2\2\u01cf"**)  
 buf.write(**"\u01d0\7R\2\2\u01d0\u01d1\7C\2\2\u01d1\u024a\7F\2\2\u01d2"**)  
 buf.write(**"\u01d3\7U\2\2\u01d3\u01d4\7W\2\2\u01d4\u01d5\7D\2\2\u01d5"**)  
 buf.write(**"\u01d6\7U\2\2\u01d6\u01d7\7V\2\2\u01d7\u024a\7T\2\2\u01d8"**)  
 buf.write(**"\u01d9\7U\2\2\u01d9\u01da\7W\2\2\u01da\u01db\7D\2\2\u01db"**)  
 buf.write(**"\u01dc\7U\2\2\u01dc\u01dd\7V\2\2\u01dd\u01de\7T\2\2\u01de"**)  
 buf.write(**"\u024a\7R\2\2\u01df\u01e0\7T\2\2\u01e0\u01e1\7G\2\2\u01e1"**)  
 buf.write(**"\u01e2\7I\2\2\u01e2\u01e3\7G\2\2\u01e3\u01e4\7Z\2\2\u01e4"**)  
 buf.write(**"\u01e5\7R\2\2\u01e5\u01e6\7a\2\2\u01e6\u01e7\7T\2\2\u01e7"**)  
 buf.write(**"\u01e8\7G\2\2\u01e8\u01e9\7R\2\2\u01e9\u01ea\7N\2\2\u01ea"**)  
 buf.write(**"\u01eb\7C\2\2\u01eb\u01ec\7E\2\2\u01ec\u024a\7G\2\2\u01ed"**)  
 buf.write(**"\u01ee\7V\2\2\u01ee\u01ef\7T\2\2\u01ef\u01f0\7C\2\2\u01f0"**)  
 buf.write(**"\u01f1\7P\2\2\u01f1\u01f2\7U\2\2\u01f2\u01f3\7N\2\2\u01f3"**)  
 buf.write(**"\u01f4\7C\2\2\u01f4\u01f5\7V\2\2\u01f5\u024a\7G\2\2\u01f6"**)  
 buf.write(**"\u01f7\7T\2\2\u01f7\u01f8\7G\2\2\u01f8\u01f9\7I\2\2\u01f9"**)  
 buf.write(**"\u01fa\7G\2\2\u01fa\u01fb\7Z\2\2\u01fb\u01fc\7R\2\2\u01fc"**)  
 buf.write(**"\u01fd\7a\2\2\u01fd\u01fe\7G\2\2\u01fe\u01ff\7Z\2\2\u01ff"**)  
 buf.write(**"\u0200\7V\2\2\u0200\u0201\7T\2\2\u0201\u0202\7C\2\2\u0202"**)  
 buf.write(**"\u0203\7E\2\2\u0203\u024a\7V\2\2\u0204\u0205\7O\2\2\u0205"**)  
 buf.write(**"\u0206\7K\2\2\u0206\u024a\7F\2\2\u0207\u0208\7N\2\2\u0208"**)  
 buf.write(**"\u0209\7Q\2\2\u0209\u020a\7Q\2\2\u020a\u020b\7M\2\2\u020b"**)  
 buf.write(**"\u020c\7a\2\2\u020c\u020d\7W\2\2\u020d\u020e\7R\2\2\u020e"**)  
 buf.write(**"\u020f\7a\2\2\u020f\u0210\7E\2\2\u0210\u0211\7J\2\2\u0211"**)  
 buf.write(**"\u0212\7G\2\2\u0212\u0213\7E\2\2\u0213\u024a\7M\2\2\u0214"**)  
 buf.write(**"\u0215\7Z\2\2\u0215\u0216\7C\2\2\u0216\u0217\7X\2\2\u0217"**)  
 buf.write(**"\u024a\7I\2\2\u0218\u0219\7Z\2\2\u0219\u021a\7E\2\2\u021a"**)  
 buf.write(**"\u021b\7Q\2\2\u021b\u021c\7W\2\2\u021c\u021d\7P\2\2\u021d"**)  
 buf.write(**"\u024a\7V\2\2\u021e\u021f\7Z\2\2\u021f\u0220\7O\2\2\u0220"**)  
 buf.write(**"\u0221\7C\2\2\u0221\u024a\7Z\2\2\u0222\u0223\7Z\2\2\u0223"**)  
 buf.write(**"\u0224\7O\2\2\u0224\u0225\7K\2\2\u0225\u024a\7P\2\2\u0226"**)  
 buf.write(**"\u0227\7Z\2\2\u0227\u0228\7U\2\2\u0228\u0229\7W\2\2\u0229"**)  
 buf.write(**"\u024a\7O\2\2\u022a\u022b\7Z\2\2\u022b\u022c\7W\2\2\u022c"**)  
 buf.write(**"\u022d\7P\2\2\u022d\u022e\7K\2\2\u022e\u022f\7S\2\2\u022f"**)  
 buf.write(**"\u0230\7W\2\2\u0230\u0231\7G\2\2\u0231\u0232\7E\2\2\u0232"**)  
 buf.write(**"\u0233\7Q\2\2\u0233\u0234\7W\2\2\u0234\u0235\7P\2\2\u0235"**)  
 buf.write(**"\u024a\7V\2\2\u0236\u0237\7C\2\2\u0237\u0238\7I\2\2\u0238"**)  
 buf.write(**"\u0239\7I\2\2\u0239\u023a\7T\2\2\u023a\u023b\7G\2\2\u023b"**)  
 buf.write(**"\u023c\7I\2\2\u023c\u023d\7C\2\2\u023d\u023e\7V\2\2\u023e"**)  
 buf.write(**"\u023f\7G\2\2\u023f\u0240\7T\2\2\u0240\u0241\7C\2\2\u0241"**)  
 buf.write(**"\u0242\7P\2\2\u0242\u0243\7I\2\2\u0243\u0244\7G\2\2\u0244"**)  
 buf.write(**"\u0245\7E\2\2\u0245\u0246\7J\2\2\u0246\u0247\7G\2\2\u0247"**)  
 buf.write(**"\u0248\7E\2\2\u0248\u024a\7M\2\2\u0249\u01be\3\2\2\2\u0249"**)  
 buf.write(**"\u01c4\3\2\2\2\u0249\u01ca\3\2\2\2\u0249\u01ce\3\2\2\2"**)  
 buf.write(**"\u0249\u01d2\3\2\2\2\u0249\u01d8\3\2\2\2\u0249\u01df\3"**)  
 buf.write(**"\2\2\2\u0249\u01ed\3\2\2\2\u0249\u01f6\3\2\2\2\u0249\u0204"**)  
 buf.write(**"\3\2\2\2\u0249\u0207\3\2\2\2\u0249\u0214\3\2\2\2\u0249"**)  
 buf.write(**"\u0218\3\2\2\2\u0249\u021e\3\2\2\2\u0249\u0222\3\2\2\2"**)  
 buf.write(**"\u0249\u0226\3\2\2\2\u0249\u022a\3\2\2\2\u0249\u0236\3"**)  
 buf.write(**"\2\2\2\u024a\n\3\2\2\2\u024b\u024c\7N\2\2\u024c\u024d"**)  
 buf.write(**"\7Q\2\2\u024d\u024e\7Q\2\2\u024e\u024f\7M\2\2\u024f\u0250"**)  
 buf.write(**"\7W\2\2\u0250\u0251\7R\2\2\u0251\u0252\7O\2\2\u0252\u0253"**)  
 buf.write(**"\7C\2\2\u0253\u0273\7R\2\2\u0254\u0255\7N\2\2\u0255\u0256"**)  
 buf.write(**"\7Q\2\2\u0256\u0257\7Q\2\2\u0257\u0258\7M\2\2\u0258\u0259"**)  
 buf.write(**"\7W\2\2\u0259\u0273\7R\2\2\u025a\u025b\7Z\2\2\u025b\u025c"**)  
 buf.write(**"\7H\2\2\u025c\u025d\7T\2\2\u025d\u025e\7G\2\2\u025e\u025f"**)  
 buf.write(**"\7S\2\2\u025f\u0260\7E\2\2\u0260\u0261\7Q\2\2\u0261\u0262"**)  
 buf.write(**"\7W\2\2\u0262\u0263\7P\2\2\u0263\u0273\7V\2\2\u0264\u0265"**)  
 buf.write(**"\7Z\2\2\u0265\u0266\7H\2\2\u0266\u0267\7T\2\2\u0267\u0268"**)  
 buf.write(**"\7G\2\2\u0268\u0269\7S\2\2\u0269\u026a\7R\2\2\u026a\u026b"**)  
 buf.write(**"\7G\2\2\u026b\u026c\7T\2\2\u026c\u0273\7E\2\2\u026d\u026e"**)  
 buf.write(**"\7Z\2\2\u026e\u026f\7R\2\2\u026f\u0270\7G\2\2\u0270\u0271"**)  
 buf.write(**"\7T\2\2\u0271\u0273\7E\2\2\u0272\u024b\3\2\2\2\u0272\u0254"**)  
 buf.write(**"\3\2\2\2\u0272\u025a\3\2\2\2\u0272\u0264\3\2\2\2\u0272"**)  
 buf.write(**"\u026d\3\2\2\2\u0273\f\3\2\2\2\u0274\u0275\7I\2\2\u0275"**)  
 buf.write(**"\u0276\7T\2\2\u0276\u0277\7G\2\2\u0277\u0278\7C\2\2\u0278"**)  
 buf.write(**"\u0279\7V\2\2\u0279\u027a\7G\2\2\u027a\u027b\7U\2\2\u027b"**)  
 buf.write(**"\u0288\7V\2\2\u027c\u027d\7N\2\2\u027d\u027e\7G\2\2\u027e"**)  
 buf.write(**"\u027f\7C\2\2\u027f\u0280\7U\2\2\u0280\u0288\7V\2\2\u0281"**)  
 buf.write(**"\u0282\7E\2\2\u0282\u0283\7Q\2\2\u0283\u0284\7P\2\2\u0284"**)  
 buf.write(**"\u0285\7E\2\2\u0285\u0286\7C\2\2\u0286\u0288\7V\2\2\u0287"**)  
 buf.write(**"\u0274\3\2\2\2\u0287\u027c\3\2\2\2\u0287\u0281\3\2\2\2"**)  
 buf.write(**"\u0288\16\3\2\2\2\u0289\u028a\7~\2\2\u028a\u0290\7~\2"**)  
 buf.write(**"\2\u028b\u028c\7Q\2\2\u028c\u0290\7T\2\2\u028d\u028e\7"**)  
 buf.write(**"q\2\2\u028e\u0290\7t\2\2\u028f\u0289\3\2\2\2\u028f\u028b"**)  
 buf.write(**"\3\2\2\2\u028f\u028d\3\2\2\2\u0290\20\3\2\2\2\u0291\u0292"**)  
 buf.write(**"\7(\2\2\u0292\u029a\7(\2\2\u0293\u0294\7C\2\2\u0294\u0295"**)  
 buf.write(**"\7P\2\2\u0295\u029a\7F\2\2\u0296\u0297\7c\2\2\u0297\u0298"**)  
 buf.write(**"\7p\2\2\u0298\u029a\7f\2\2\u0299\u0291\3\2\2\2\u0299\u0293"**)  
 buf.write(**"\3\2\2\2\u0299\u0296\3\2\2\2\u029a\22\3\2\2\2\u029b\u029c"**)  
 buf.write(**"\7?\2\2\u029c\u029d\7?\2\2\u029d\24\3\2\2\2\u029e\u029f"**)  
 buf.write(**"\7#\2\2\u029f\u02a0\7?\2\2\u02a0\26\3\2\2\2\u02a1\u02a2"**)  
 buf.write(**"\7@\2\2\u02a2\30\3\2\2\2\u02a3\u02a4\7>\2\2\u02a4\32\3"**)  
 buf.write(**"\2\2\2\u02a5\u02a6\7@\2\2\u02a6\u02a7\7?\2\2\u02a7\34"**)  
 buf.write(**"\3\2\2\2\u02a8\u02a9\7>\2\2\u02a9\u02aa\7?\2\2\u02aa\36"**)  
 buf.write(**"\3\2\2\2\u02ab\u02ac\7-\2\2\u02ac \3\2\2\2\u02ad\u02ae"**)  
 buf.write(**"\7/\2\2\u02ae\"\3\2\2\2\u02af\u02b0\7,\2\2\u02b0$\3\2"**)  
 buf.write(**"\2\2\u02b1\u02b2\7\61\2\2\u02b2&\3\2\2\2\u02b3\u02b4\7"**)  
 buf.write(**"\'\2\2\u02b4(\3\2\2\2\u02b5\u02b6\7`\2\2\u02b6\*\3\2\2"**)  
 buf.write(**"\2\u02b7\u02b8\7#\2\2\u02b8,\3\2\2\2\u02b9\u02ba\7=\2"**)  
 buf.write(**"\2\u02ba.\3\2\2\2\u02bb\u02bc\7.\2\2\u02bc\60\3\2\2\2"**)  
 buf.write(**"\u02bd\u02be\7?\2\2\u02be\62\3\2\2\2\u02bf\u02c0\7\*\2"**)  
 buf.write(**"\2\u02c0\64\3\2\2\2\u02c1\u02c2\7+\2\2\u02c2\66\3\2\2"**)  
 buf.write(**"\2\u02c3\u02c4\7}\2\2\u02c48\3\2\2\2\u02c5\u02c6\7\177"**)  
 buf.write(**"\2\2\u02c6:\3\2\2\2\u02c7\u02c8\7]\2\2\u02c8<\3\2\2\2"**)  
 buf.write(**"\u02c9\u02ca\7\_\2\2\u02ca>\3\2\2\2\u02cb\u02cc\7V\2\2"**)  
 buf.write(**"\u02cc\u02cd\7T\2\2\u02cd\u02ce\7W\2\2\u02ce\u02d8\7G"**)  
 buf.write(**"\2\2\u02cf\u02d0\7V\2\2\u02d0\u02d1\7t\2\2\u02d1\u02d2"**)  
 buf.write(**"\7w\2\2\u02d2\u02d8\7g\2\2\u02d3\u02d4\7v\2\2\u02d4\u02d5"**)  
 buf.write(**"\7t\2\2\u02d5\u02d6\7w\2\2\u02d6\u02d8\7g\2\2\u02d7\u02cb"**)  
 buf.write(**"\3\2\2\2\u02d7\u02cf\3\2\2\2\u02d7\u02d3\3\2\2\2\u02d8"**)  
 buf.write(**"@\3\2\2\2\u02d9\u02da\7H\2\2\u02da\u02db\7C\2\2\u02db"**)  
 buf.write(**"\u02dc\7N\2\2\u02dc\u02dd\7U\2\2\u02dd\u02e9\7G\2\2\u02de"**)  
 buf.write(**"\u02df\7H\2\2\u02df\u02e0\7c\2\2\u02e0\u02e1\7n\2\2\u02e1"**)  
 buf.write(**"\u02e2\7u\2\2\u02e2\u02e9\7g\2\2\u02e3\u02e4\7h\2\2\u02e4"**)  
 buf.write(**"\u02e5\7c\2\2\u02e5\u02e6\7n\2\2\u02e6\u02e7\7u\2\2\u02e7"**)  
 buf.write(**"\u02e9\7g\2\2\u02e8\u02d9\3\2\2\2\u02e8\u02de\3\2\2\2"**)  
 buf.write(**"\u02e8\u02e3\3\2\2\2\u02e9B\3\2\2\2\u02ea\u02eb\7p\2\2"**)  
 buf.write(**"\u02eb\u02ec\7w\2\2\u02ec\u02ed\7n\2\2\u02ed\u02ee\7n"**)  
 buf.write(**"\2\2\u02eeD\3\2\2\2\u02ef\u02f0\7K\2\2\u02f0\u02f4\7H"**)  
 buf.write(**"\2\2\u02f1\u02f2\7k\2\2\u02f2\u02f4\7h\2\2\u02f3\u02ef"**)  
 buf.write(**"\3\2\2\2\u02f3\u02f1\3\2\2\2\u02f4F\3\2\2\2\u02f5\u02f6"**)  
 buf.write(**"\7G\2\2\u02f6\u02f7\7N\2\2\u02f7\u02f8\7U\2\2\u02f8\u02fe"**)  
 buf.write(**"\7G\2\2\u02f9\u02fa\7g\2\2\u02fa\u02fb\7n\2\2\u02fb\u02fc"**)  
 buf.write(**"\7u\2\2\u02fc\u02fe\7g\2\2\u02fd\u02f5\3\2\2\2\u02fd\u02f9"**)  
 buf.write(**"\3\2\2\2\u02feH\3\2\2\2\u02ff\u0300\7y\2\2\u0300\u0301"**)  
 buf.write(**"\7j\2\2\u0301\u0302\7k\2\2\u0302\u0303\7n\2\2\u0303\u0304"**)  
 buf.write(**"\7g\2\2\u0304J\3\2\2\2\u0305\u0309\t\2\2\2\u0306\u0308"**)  
 buf.write(**"\t\3\2\2\u0307\u0306\3\2\2\2\u0308\u030b\3\2\2\2\u0309"**)  
 buf.write(**"\u0307\3\2\2\2\u0309\u030a\3\2\2\2\u030aL\3\2\2\2\u030b"**)  
 buf.write(**"\u0309\3\2\2\2\u030c\u030e\t\4\2\2\u030d\u030c\3\2\2\2"**)  
 buf.write(**"\u030e\u030f\3\2\2\2\u030f\u030d\3\2\2\2\u030f\u0310\3"**)  
 buf.write(**"\2\2\2\u0310N\3\2\2\2\u0311\u0313\t\4\2\2\u0312\u0311"**)  
 buf.write(**"\3\2\2\2\u0313\u0314\3\2\2\2\u0314\u0312\3\2\2\2\u0314"**)  
 buf.write(**"\u0315\3\2\2\2\u0315\u0316\3\2\2\2\u0316\u031a\7\60\2"**)  
 buf.write(**"\2\u0317\u0319\t\4\2\2\u0318\u0317\3\2\2\2\u0319\u031c"**)  
 buf.write(**"\3\2\2\2\u031a\u0318\3\2\2\2\u031a\u031b\3\2\2\2\u031b"**)  
 buf.write(**"\u0324\3\2\2\2\u031c\u031a\3\2\2\2\u031d\u031f\7\60\2"**)  
 buf.write(**"\2\u031e\u0320\t\4\2\2\u031f\u031e\3\2\2\2\u0320\u0321"**)  
 buf.write(**"\3\2\2\2\u0321\u031f\3\2\2\2\u0321\u0322\3\2\2\2\u0322"**)  
 buf.write(**"\u0324\3\2\2\2\u0323\u0312\3\2\2\2\u0323\u031d\3\2\2\2"**)  
 buf.write(**"\u0324P\3\2\2\2\u0325\u032b\7$\2\2\u0326\u032a\n\5\2\2"**)  
 buf.write(**"\u0327\u0328\7^\2\2\u0328\u032a\7$\2\2\u0329\u0326\3\2"**)  
 buf.write(**"\2\2\u0329\u0327\3\2\2\2\u032a\u032d\3\2\2\2\u032b\u0329"**)  
 buf.write(**"\3\2\2\2\u032b\u032c\3\2\2\2\u032c\u032e\3\2\2\2\u032d"**)  
 buf.write(**"\u032b\3\2\2\2\u032e\u032f\7$\2\2\u032fR\3\2\2\2\u0330"**)  
 buf.write(**"\u0334\7%\2\2\u0331\u0333\n\6\2\2\u0332\u0331\3\2\2\2"**)  
 buf.write(**"\u0333\u0336\3\2\2\2\u0334\u0332\3\2\2\2\u0334\u0335\3"**)  
 buf.write(**"\2\2\2\u0335\u0337\3\2\2\2\u0336\u0334\3\2\2\2\u0337\u0338"**)  
 buf.write(**"\b\*\2\2\u0338T\3\2\2\2\u0339\u033a\t\7\2\2\u033a\u033b"**)  
 buf.write(**"\3\2\2\2\u033b\u033c\b+\2\2\u033cV\3\2\2\2\u033d\u033e"**)  
 buf.write(**"\13\2\2\2\u033eX\3\2\2\2\30\2e\u014d\u01bc\u0249\u0272"**)  
 buf.write(**"\u0287\u028f\u0299\u02d7\u02e8\u02f3\u02fd\u0309\u030f"**)  
 buf.write(**"\u0314\u031a\u0321\u0323\u0329\u032b\u0334\3\b\2\2"**)  
 **return** buf.getvalue()  
  
  
**class** ExpressionLexer(Lexer):  
  
 atn = ATNDeserializer().deserialize(serializedATN())  
  
 decisionsToDFA = [ DFA(ds, i) **for** i, ds **in** enumerate(atn.decisionToState) ]  
  
 FUNCTIONNAME\_PARAM0 = 1  
 FUNCTIONNAME\_PARAM1 = 2  
 FUNCTIONNAME\_PARAM2 = 3  
 FUNCTIONNAME\_PARAM3 = 4  
 FUNCTIONNAME\_PARAM4 = 5  
 FUNCTIONNAME\_PARAM\_N = 6  
 MOSAIC\_OR = 7  
 MOSAIC\_AND = 8  
 EQ = 9  
 NEQ = 10  
 GT = 11  
 LT = 12  
 GTEQ = 13  
 LTEQ = 14  
 PLUS = 15  
 MINUS = 16  
 MULT = 17  
 DIV = 18  
 MOD = 19  
 POW = 20  
 NOT = 21  
 SCOL = 22  
 COMMA = 23  
 ASSIGN = 24  
 LPAREN = 25  
 RPAREN = 26  
 OBRACE = 27  
 CBRACE = 28  
 BEGL = 29  
 ENDL = 30  
 MOSAIC\_TRUE = 31  
 MOSAIC\_FALSE = 32  
 NIL = 33  
 MOSAIC\_IF = 34  
 MOSAIC\_ELSE = 35  
 WHILE = 36  
 ID = 37  
 INT = 38  
 DOUBLE = 39  
 STRING = 40  
 COMMENT = 41  
 SPACE = 42  
 OTHER = 43  
  
 channelNames = [ **u"DEFAULT\_TOKEN\_CHANNEL"**, **u"HIDDEN"** ]  
  
 modeNames = [ **"DEFAULT\_MODE"** ]  
  
 literalNames = [ **"<INVALID>"**,  
 **"'=='"**, **"'!='"**, **"'>'"**, **"'<'"**, **"'>='"**, **"'<='"**, **"'+'"**, **"'-'"**,   
 **"'\*'"**, **"'/'"**, **"'%'"**, **"'^'"**, **"'!'"**, **"';'"**, **"','"**, **"'='"**, **"'('"**,   
 **"')'"**, **"'{'"**, **"'}'"**, **"'['"**, **"']'"**, **"'null'"**, **"'while'"** ]  
  
 symbolicNames = [ **"<INVALID>"**,  
 **"FUNCTIONNAME\_PARAM0"**, **"FUNCTIONNAME\_PARAM1"**, **"FUNCTIONNAME\_PARAM2"**,   
 **"FUNCTIONNAME\_PARAM3"**, **"FUNCTIONNAME\_PARAM4"**, **"FUNCTIONNAME\_PARAM\_N"**,   
 **"MOSAIC\_OR"**, **"MOSAIC\_AND"**, **"EQ"**, **"NEQ"**, **"GT"**, **"LT"**, **"GTEQ"**,   
 **"LTEQ"**, **"PLUS"**, **"MINUS"**, **"MULT"**, **"DIV"**, **"MOD"**, **"POW"**, **"NOT"**,   
 **"SCOL"**, **"COMMA"**, **"ASSIGN"**, **"LPAREN"**, **"RPAREN"**, **"OBRACE"**, **"CBRACE"**,   
 **"BEGL"**, **"ENDL"**, **"MOSAIC\_TRUE"**, **"MOSAIC\_FALSE"**, **"NIL"**, **"MOSAIC\_IF"**,   
 **"MOSAIC\_ELSE"**, **"WHILE"**, **"ID"**, **"INT"**, **"DOUBLE"**, **"STRING"**, **"COMMENT"**,   
 **"SPACE"**, **"OTHER"** ]  
  
 ruleNames = [ **"FUNCTIONNAME\_PARAM0"**, **"FUNCTIONNAME\_PARAM1"**, **"FUNCTIONNAME\_PARAM2"**,   
 **"FUNCTIONNAME\_PARAM3"**, **"FUNCTIONNAME\_PARAM4"**, **"FUNCTIONNAME\_PARAM\_N"**,   
 **"MOSAIC\_OR"**, **"MOSAIC\_AND"**, **"EQ"**, **"NEQ"**, **"GT"**, **"LT"**, **"GTEQ"**,   
 **"LTEQ"**, **"PLUS"**, **"MINUS"**, **"MULT"**, **"DIV"**, **"MOD"**, **"POW"**,   
 **"NOT"**, **"SCOL"**, **"COMMA"**, **"ASSIGN"**, **"LPAREN"**, **"RPAREN"**,   
 **"OBRACE"**, **"CBRACE"**, **"BEGL"**, **"ENDL"**, **"MOSAIC\_TRUE"**, **"MOSAIC\_FALSE"**,   
 **"NIL"**, **"MOSAIC\_IF"**, **"MOSAIC\_ELSE"**, **"WHILE"**, **"ID"**, **"INT"**,   
 **"DOUBLE"**, **"STRING"**, **"COMMENT"**, **"SPACE"**, **"OTHER"** ]  
  
 grammarFileName = **"Expression.g4"  
  
 def** \_\_init\_\_(self, input=**None**, output:TextIO = sys.stdout):  
 super().\_\_init\_\_(input, output)  
 self.checkVersion(**"4.7.1"**)  
 self.\_interp = LexerATNSimulator(self, self.atn, self.decisionsToDFA, PredictionContextCache())  
 self.\_actions = **None** self.\_predicates = **None**

*# Generated from Expression.g4 by ANTLR 4.7.1  
# encoding: utf-8  
# pylint: disable-all***import** sys  
**from** typing.io **import** TextIO  
**from** io **import** StringIO  
**from** antlr4 **import** \*  
  
**def** serializedATN():  
 **with** StringIO() **as** buf:  
 buf.write(**"\3\u608b\ua72a\u8133\ub9ed\u417c\u3be7\u7786\u5964\3-"**)  
 buf.write(**"\u0104\4\2\t\2\4\3\t\3\4\4\t\4\4\5\t\5\4\6\t\6\4\7\t\7"**)  
 buf.write(**"\4\b\t\b\4\t\t\t\4\n\t\n\4\13\t\13\4\f\t\f\4\r\t\r\4\16"**)  
 buf.write(**"\t\16\4\17\t\17\4\20\t\20\4\21\t\21\4\22\t\22\4\23\t\23"**)  
 buf.write(**"\4\24\t\24\4\25\t\25\4\26\t\26\4\27\t\27\4\30\t\30\4\31"**)  
 buf.write(**"\t\31\4\32\t\32\4\33\t\33\3\2\3\2\3\2\3\3\7\3;\n\3\f\3"**)  
 buf.write(**"\16\3>\13\3\3\4\3\4\3\4\3\4\3\4\3\4\5\4F\n\4\3\5\3\5\3"**)  
 buf.write(**"\5\3\5\3\5\3\6\3\6\3\6\3\6\3\6\7\6R\n\6\f\6\16\6U\13\6"**)  
 buf.write(**"\3\6\3\6\5\6Y\n\6\3\7\3\7\3\7\3\b\3\b\3\b\3\b\3\b\5\b"**)  
 buf.write(**"c\n\b\3\t\3\t\3\t\3\t\3\n\3\n\3\n\3\13\3\13\3\13\3\13"**)  
 buf.write(**"\3\13\3\13\3\13\5\13s\n\13\3\13\3\13\3\13\3\13\3\13\3"**)  
 buf.write(**"\13\3\13\3\13\3\13\3\13\3\13\3\13\3\13\3\13\3\13\3\13"**)  
 buf.write(**"\3\13\3\13\3\13\3\13\3\13\7\13\u008a\n\13\f\13\16\13\u008d"**)  
 buf.write(**"\13\13\3\f\3\f\3\f\3\f\3\f\3\f\5\f\u0095\n\f\3\r\3\r\3"**)  
 buf.write(**"\r\3\r\3\16\3\16\3\16\3\16\3\16\3\17\3\17\3\17\3\17\3"**)  
 buf.write(**"\17\3\20\3\20\3\20\3\20\3\20\3\21\3\21\3\21\3\21\3\21"**)  
 buf.write(**"\3\22\3\22\3\22\3\22\3\22\3\23\5\23\u00b5\n\23\3\23\3"**)  
 buf.write(**"\23\5\23\u00b9\n\23\7\23\u00bb\n\23\f\23\16\23\u00be\13"**)  
 buf.write(**"\23\3\24\5\24\u00c1\n\24\3\25\5\25\u00c4\n\25\3\25\3\25"**)  
 buf.write(**"\5\25\u00c8\n\25\3\26\5\26\u00cb\n\26\3\26\3\26\5\26\u00cf"**)  
 buf.write(**"\n\26\3\26\3\26\5\26\u00d3\n\26\3\27\5\27\u00d6\n\27\3"**)  
 buf.write(**"\27\3\27\5\27\u00da\n\27\3\27\3\27\5\27\u00de\n\27\3\27"**)  
 buf.write(**"\3\27\5\27\u00e2\n\27\3\30\3\30\3\30\3\30\3\30\3\30\3"**)  
 buf.write(**"\30\3\30\3\30\3\30\5\30\u00ee\n\30\3\31\3\31\5\31\u00f2"**)  
 buf.write(**"\n\31\3\31\3\31\3\32\5\32\u00f7\n\32\3\32\3\32\5\32\u00fb"**)  
 buf.write(**"\n\32\7\32\u00fd\n\32\f\32\16\32\u0100\13\32\3\33\3\33"**)  
 buf.write(**"\3\33\2\3\24\34\2\4\6\b\n\f\16\20\22\24\26\30\32\34\36"**)  
 buf.write(**" \"$&(\*,.\60\62\64\2\b\3\2\23\25\3\2\21\22\3\2\r\20\3"**)  
 buf.write(**"\2\13\f\3\2()\3\2!\"\2\u0117\2\66\3\2\2\2\4<\3\2\2\2\6"**)  
 buf.write(**"E\3\2\2\2\bG\3\2\2\2\nL\3\2\2\2\fZ\3\2\2\2\16b\3\2\2\2"**)  
 buf.write(**"\20d\3\2\2\2\22h\3\2\2\2\24r\3\2\2\2\26\u0094\3\2\2\2"**)  
 buf.write(**"\30\u0096\3\2\2\2\32\u009a\3\2\2\2\34\u009f\3\2\2\2\36"**)  
 buf.write(**"\u00a4\3\2\2\2 \u00a9\3\2\2\2\"\u00ae\3\2\2\2$\u00b4\3"**)  
 buf.write(**"\2\2\2&\u00c0\3\2\2\2(\u00c3\3\2\2\2\*\u00ca\3\2\2\2,\u00d5"**)  
 buf.write(**"\3\2\2\2.\u00ed\3\2\2\2\60\u00ef\3\2\2\2\62\u00f6\3\2"**)  
 buf.write(**"\2\2\64\u0101\3\2\2\2\66\67\5\4\3\2\678\7\2\2\38\3\3\2"**)  
 buf.write(**"\2\29;\5\6\4\2:9\3\2\2\2;>\3\2\2\2<:\3\2\2\2<=\3\2\2\2"**)  
 buf.write(**"=\5\3\2\2\2><\3\2\2\2?F\5\b\5\2@F\5\n\6\2AF\5\20\t\2B"**)  
 buf.write(**"F\5\22\n\2CD\7-\2\2DF\b\4\1\2E?\3\2\2\2E@\3\2\2\2EA\3"**)  
 buf.write(**"\2\2\2EB\3\2\2\2EC\3\2\2\2F\7\3\2\2\2GH\7\'\2\2HI\7\32"**)  
 buf.write(**"\2\2IJ\5\24\13\2JK\7\30\2\2K\t\3\2\2\2LM\7$\2\2MS\5\f"**)  
 buf.write(**"\7\2NO\7%\2\2OP\7$\2\2PR\5\f\7\2QN\3\2\2\2RU\3\2\2\2S"**)  
 buf.write(**"Q\3\2\2\2ST\3\2\2\2TX\3\2\2\2US\3\2\2\2VW\7%\2\2WY\5\16"**)  
 buf.write(**"\b\2XV\3\2\2\2XY\3\2\2\2Y\13\3\2\2\2Z[\5\24\13\2[\\\5"**)  
 buf.write(**"\16\b\2\\\r\3\2\2\2]^\7\35\2\2^\_\5\4\3\2\_`\7\36\2\2`c"**)  
 buf.write(**"\3\2\2\2ac\5\6\4\2b]\3\2\2\2ba\3\2\2\2c\17\3\2\2\2de\7"**)  
 buf.write(**"&\2\2ef\5\24\13\2fg\5\16\b\2g\21\3\2\2\2hi\5\24\13\2i"**)  
 buf.write(**"j\7\30\2\2j\23\3\2\2\2kl\b\13\1\2ls\5\26\f\2mn\7\22\2"**)  
 buf.write(**"\2ns\5\24\13\13op\7\27\2\2ps\5\24\13\nqs\5.\30\2rk\3\2"**)  
 buf.write(**"\2\2rm\3\2\2\2ro\3\2\2\2rq\3\2\2\2s\u008b\3\2\2\2tu\f"**)  
 buf.write(**"\f\2\2uv\7\26\2\2v\u008a\5\24\13\rwx\f\t\2\2xy\t\2\2\2"**)  
 buf.write(**"y\u008a\5\24\13\nz{\f\b\2\2{|\t\3\2\2|\u008a\5\24\13\t"**)  
 buf.write(**"}~\f\7\2\2~\177\t\4\2\2\177\u008a\5\24\13\b\u0080\u0081"**)  
 buf.write(**"\f\6\2\2\u0081\u0082\t\5\2\2\u0082\u008a\5\24\13\7\u0083"**)  
 buf.write(**"\u0084\f\5\2\2\u0084\u0085\7\n\2\2\u0085\u008a\5\24\13"**)  
 buf.write(**"\6\u0086\u0087\f\4\2\2\u0087\u0088\7\t\2\2\u0088\u008a"**)  
 buf.write(**"\5\24\13\5\u0089t\3\2\2\2\u0089w\3\2\2\2\u0089z\3\2\2"**)  
 buf.write(**"\2\u0089}\3\2\2\2\u0089\u0080\3\2\2\2\u0089\u0083\3\2"**)  
 buf.write(**"\2\2\u0089\u0086\3\2\2\2\u008a\u008d\3\2\2\2\u008b\u0089"**)  
 buf.write(**"\3\2\2\2\u008b\u008c\3\2\2\2\u008c\25\3\2\2\2\u008d\u008b"**)  
 buf.write(**"\3\2\2\2\u008e\u0095\5\30\r\2\u008f\u0095\5\32\16\2\u0090"**)  
 buf.write(**"\u0095\5\34\17\2\u0091\u0095\5\36\20\2\u0092\u0095\5 "**)  
 buf.write(**"\21\2\u0093\u0095\5\"\22\2\u0094\u008e\3\2\2\2\u0094\u008f"**)  
 buf.write(**"\3\2\2\2\u0094\u0090\3\2\2\2\u0094\u0091\3\2\2\2\u0094"**)  
 buf.write(**"\u0092\3\2\2\2\u0094\u0093\3\2\2\2\u0095\27\3\2\2\2\u0096"**)  
 buf.write(**"\u0097\7\3\2\2\u0097\u0098\7\33\2\2\u0098\u0099\7\34\2"**)  
 buf.write(**"\2\u0099\31\3\2\2\2\u009a\u009b\7\4\2\2\u009b\u009c\7"**)  
 buf.write(**"\33\2\2\u009c\u009d\5&\24\2\u009d\u009e\7\34\2\2\u009e"**)  
 buf.write(**"\33\3\2\2\2\u009f\u00a0\7\5\2\2\u00a0\u00a1\7\33\2\2\u00a1"**)  
 buf.write(**"\u00a2\5(\25\2\u00a2\u00a3\7\34\2\2\u00a3\35\3\2\2\2\u00a4"**)  
 buf.write(**"\u00a5\7\6\2\2\u00a5\u00a6\7\33\2\2\u00a6\u00a7\5\*\26"**)  
 buf.write(**"\2\u00a7\u00a8\7\34\2\2\u00a8\37\3\2\2\2\u00a9\u00aa\7"**)  
 buf.write(**"\7\2\2\u00aa\u00ab\7\33\2\2\u00ab\u00ac\5,\27\2\u00ac"**)  
 buf.write(**"\u00ad\7\34\2\2\u00ad!\3\2\2\2\u00ae\u00af\7\b\2\2\u00af"**)  
 buf.write(**"\u00b0\7\33\2\2\u00b0\u00b1\5$\23\2\u00b1\u00b2\7\34\2"**)  
 buf.write(**"\2\u00b2#\3\2\2\2\u00b3\u00b5\5\24\13\2\u00b4\u00b3\3"**)  
 buf.write(**"\2\2\2\u00b4\u00b5\3\2\2\2\u00b5\u00bc\3\2\2\2\u00b6\u00b8"**)  
 buf.write(**"\7\31\2\2\u00b7\u00b9\5\24\13\2\u00b8\u00b7\3\2\2\2\u00b8"**)  
 buf.write(**"\u00b9\3\2\2\2\u00b9\u00bb\3\2\2\2\u00ba\u00b6\3\2\2\2"**)  
 buf.write(**"\u00bb\u00be\3\2\2\2\u00bc\u00ba\3\2\2\2\u00bc\u00bd\3"**)  
 buf.write(**"\2\2\2\u00bd%\3\2\2\2\u00be\u00bc\3\2\2\2\u00bf\u00c1"**)  
 buf.write(**"\5\24\13\2\u00c0\u00bf\3\2\2\2\u00c0\u00c1\3\2\2\2\u00c1"**)  
 buf.write(**"\'\3\2\2\2\u00c2\u00c4\5\24\13\2\u00c3\u00c2\3\2\2\2\u00c3"**)  
 buf.write(**"\u00c4\3\2\2\2\u00c4\u00c5\3\2\2\2\u00c5\u00c7\7\31\2"**)  
 buf.write(**"\2\u00c6\u00c8\5\24\13\2\u00c7\u00c6\3\2\2\2\u00c7\u00c8"**)  
 buf.write(**"\3\2\2\2\u00c8)\3\2\2\2\u00c9\u00cb\5\24\13\2\u00ca\u00c9"**)  
 buf.write(**"\3\2\2\2\u00ca\u00cb\3\2\2\2\u00cb\u00cc\3\2\2\2\u00cc"**)  
 buf.write(**"\u00ce\7\31\2\2\u00cd\u00cf\5\24\13\2\u00ce\u00cd\3\2"**)  
 buf.write(**"\2\2\u00ce\u00cf\3\2\2\2\u00cf\u00d0\3\2\2\2\u00d0\u00d2"**)  
 buf.write(**"\7\31\2\2\u00d1\u00d3\5\24\13\2\u00d2\u00d1\3\2\2\2\u00d2"**)  
 buf.write(**"\u00d3\3\2\2\2\u00d3+\3\2\2\2\u00d4\u00d6\5\24\13\2\u00d5"**)  
 buf.write(**"\u00d4\3\2\2\2\u00d5\u00d6\3\2\2\2\u00d6\u00d7\3\2\2\2"**)  
 buf.write(**"\u00d7\u00d9\7\31\2\2\u00d8\u00da\5\24\13\2\u00d9\u00d8"**)  
 buf.write(**"\3\2\2\2\u00d9\u00da\3\2\2\2\u00da\u00db\3\2\2\2\u00db"**)  
 buf.write(**"\u00dd\7\31\2\2\u00dc\u00de\5\24\13\2\u00dd\u00dc\3\2"**)  
 buf.write(**"\2\2\u00dd\u00de\3\2\2\2\u00de\u00df\3\2\2\2\u00df\u00e1"**)  
 buf.write(**"\7\31\2\2\u00e0\u00e2\5\24\13\2\u00e1\u00e0\3\2\2\2\u00e1"**)  
 buf.write(**"\u00e2\3\2\2\2\u00e2-\3\2\2\2\u00e3\u00e4\7\33\2\2\u00e4"**)  
 buf.write(**"\u00e5\5\24\13\2\u00e5\u00e6\7\34\2\2\u00e6\u00ee\3\2"**)  
 buf.write(**"\2\2\u00e7\u00ee\5\60\31\2\u00e8\u00ee\t\6\2\2\u00e9\u00ee"**)  
 buf.write(**"\t\7\2\2\u00ea\u00ee\7\'\2\2\u00eb\u00ee\7\*\2\2\u00ec"**)  
 buf.write(**"\u00ee\7#\2\2\u00ed\u00e3\3\2\2\2\u00ed\u00e7\3\2\2\2"**)  
 buf.write(**"\u00ed\u00e8\3\2\2\2\u00ed\u00e9\3\2\2\2\u00ed\u00ea\3"**)  
 buf.write(**"\2\2\2\u00ed\u00eb\3\2\2\2\u00ed\u00ec\3\2\2\2\u00ee/"**)  
 buf.write(**"\3\2\2\2\u00ef\u00f1\7\37\2\2\u00f0\u00f2\5\62\32\2\u00f1"**)  
 buf.write(**"\u00f0\3\2\2\2\u00f1\u00f2\3\2\2\2\u00f2\u00f3\3\2\2\2"**)  
 buf.write(**"\u00f3\u00f4\7 \2\2\u00f4\61\3\2\2\2\u00f5\u00f7\5\64"**)  
 buf.write(**"\33\2\u00f6\u00f5\3\2\2\2\u00f6\u00f7\3\2\2\2\u00f7\u00fe"**)  
 buf.write(**"\3\2\2\2\u00f8\u00fa\7\31\2\2\u00f9\u00fb\5\64\33\2\u00fa"**)  
 buf.write(**"\u00f9\3\2\2\2\u00fa\u00fb\3\2\2\2\u00fb\u00fd\3\2\2\2"**)  
 buf.write(**"\u00fc\u00f8\3\2\2\2\u00fd\u0100\3\2\2\2\u00fe\u00fc\3"**)  
 buf.write(**"\2\2\2\u00fe\u00ff\3\2\2\2\u00ff\63\3\2\2\2\u0100\u00fe"**)  
 buf.write(**"\3\2\2\2\u0101\u0102\5.\30\2\u0102\65\3\2\2\2\35<ESXb"**)  
 buf.write(**"r\u0089\u008b\u0094\u00b4\u00b8\u00bc\u00c0\u00c3\u00c7"**)  
 buf.write(**"\u00ca\u00ce\u00d2\u00d5\u00d9\u00dd\u00e1\u00ed\u00f1"**)  
 buf.write(**"\u00f6\u00fa\u00fe"**)  
 **return** buf.getvalue()  
  
  
**class** ExpressionParser ( Parser ):  
  
 grammarFileName = **"Expression.g4"** atn = ATNDeserializer().deserialize(serializedATN())  
  
 decisionsToDFA = [ DFA(ds, i) **for** i, ds **in** enumerate(atn.decisionToState) ]  
  
 sharedContextCache = PredictionContextCache()  
  
 literalNames = [ **"<INVALID>"**, **"<INVALID>"**, **"<INVALID>"**, **"<INVALID>"**,   
 **"<INVALID>"**, **"<INVALID>"**, **"<INVALID>"**, **"<INVALID>"**,   
 **"<INVALID>"**, **"'=='"**, **"'!='"**, **"'>'"**, **"'<'"**, **"'>='"**,   
 **"'<='"**, **"'+'"**, **"'-'"**, **"'\*'"**, **"'/'"**, **"'%'"**, **"'^'"**, **"'!'"**,   
 **"';'"**, **"','"**, **"'='"**, **"'('"**, **"')'"**, **"'{'"**, **"'}'"**, **"'['"**,   
 **"']'"**, **"<INVALID>"**, **"<INVALID>"**, **"'null'"**, **"<INVALID>"**,   
 **"<INVALID>"**, **"'while'"** ]  
  
 symbolicNames = [ **"<INVALID>"**, **"FUNCTIONNAME\_PARAM0"**, **"FUNCTIONNAME\_PARAM1"**,   
 **"FUNCTIONNAME\_PARAM2"**, **"FUNCTIONNAME\_PARAM3"**, **"FUNCTIONNAME\_PARAM4"**,   
 **"FUNCTIONNAME\_PARAM\_N"**, **"MOSAIC\_OR"**, **"MOSAIC\_AND"**,   
 **"EQ"**, **"NEQ"**, **"GT"**, **"LT"**, **"GTEQ"**, **"LTEQ"**, **"PLUS"**, **"MINUS"**,   
 **"MULT"**, **"DIV"**, **"MOD"**, **"POW"**, **"NOT"**, **"SCOL"**, **"COMMA"**,   
 **"ASSIGN"**, **"LPAREN"**, **"RPAREN"**, **"OBRACE"**, **"CBRACE"**,   
 **"BEGL"**, **"ENDL"**, **"MOSAIC\_TRUE"**, **"MOSAIC\_FALSE"**, **"NIL"**,   
 **"MOSAIC\_IF"**, **"MOSAIC\_ELSE"**, **"WHILE"**, **"ID"**, **"INT"**,   
 **"DOUBLE"**, **"STRING"**, **"COMMENT"**, **"SPACE"**, **"OTHER"** ]  
  
 RULE\_parse = 0  
 RULE\_block = 1  
 RULE\_statements = 2  
 RULE\_assignment = 3  
 RULE\_ifStat = 4  
 RULE\_conditionBlock = 5  
 RULE\_statBlock = 6  
 RULE\_whileStat = 7  
 RULE\_statement = 8  
 RULE\_expr = 9  
 RULE\_functions = 10  
 RULE\_functionParam0 = 11  
 RULE\_functionParam1 = 12  
 RULE\_functionParam2 = 13  
 RULE\_functionParam3 = 14  
 RULE\_functionParam4 = 15  
 RULE\_functionParamN = 16  
 RULE\_argumentsN = 17  
 RULE\_arguments1 = 18  
 RULE\_arguments2 = 19  
 RULE\_arguments3 = 20  
 RULE\_arguments4 = 21  
 RULE\_atom = 22  
 RULE\_valueList = 23  
 RULE\_array = 24  
 RULE\_arrayElement = 25  
  
 ruleNames = [ **"parse"**, **"block"**, **"statements"**, **"assignment"**, **"ifStat"**,   
 **"conditionBlock"**, **"statBlock"**, **"whileStat"**, **"statement"**,   
 **"expr"**, **"functions"**, **"functionParam0"**, **"functionParam1"**,   
 **"functionParam2"**, **"functionParam3"**, **"functionParam4"**,   
 **"functionParamN"**, **"argumentsN"**, **"arguments1"**, **"arguments2"**,   
 **"arguments3"**, **"arguments4"**, **"atom"**, **"valueList"**, **"array"**,   
 **"arrayElement"** ]  
  
 EOF = Token.EOF  
 FUNCTIONNAME\_PARAM0=1  
 FUNCTIONNAME\_PARAM1=2  
 FUNCTIONNAME\_PARAM2=3  
 FUNCTIONNAME\_PARAM3=4  
 FUNCTIONNAME\_PARAM4=5  
 FUNCTIONNAME\_PARAM\_N=6  
 MOSAIC\_OR=7  
 MOSAIC\_AND=8  
 EQ=9  
 NEQ=10  
 GT=11  
 LT=12  
 GTEQ=13  
 LTEQ=14  
 PLUS=15  
 MINUS=16  
 MULT=17  
 DIV=18  
 MOD=19  
 POW=20  
 NOT=21  
 SCOL=22  
 COMMA=23  
 ASSIGN=24  
 LPAREN=25  
 RPAREN=26  
 OBRACE=27  
 CBRACE=28  
 BEGL=29  
 ENDL=30  
 MOSAIC\_TRUE=31  
 MOSAIC\_FALSE=32  
 NIL=33  
 MOSAIC\_IF=34  
 MOSAIC\_ELSE=35  
 WHILE=36  
 ID=37  
 INT=38  
 DOUBLE=39  
 STRING=40  
 COMMENT=41  
 SPACE=42  
 OTHER=43  
  
 **def** \_\_init\_\_(self, input:TokenStream, output:TextIO = sys.stdout):  
 super().\_\_init\_\_(input, output)  
 self.checkVersion(**"4.7.1"**)  
 self.\_interp = ParserATNSimulator(self, self.atn, self.decisionsToDFA, self.sharedContextCache)  
 self.\_predicates = **None  
  
  
  
 class** ParseContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** block(self):  
 **return** self.getTypedRuleContext(ExpressionParser.BlockContext,0)  
  
  
 **def** EOF(self):  
 **return** self.getToken(ExpressionParser.EOF, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_parse  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterParse"** ):  
 listener.enterParse(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitParse"** ):  
 listener.exitParse(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitParse"** ):  
 **return** visitor.visitParse(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** parse(self):  
  
 localctx = ExpressionParser.ParseContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 0, self.RULE\_parse)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 52  
 self.block()  
 self.state = 53  
 self.match(ExpressionParser.EOF)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** BlockContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** statements(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.StatementsContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.StatementsContext,i)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_block  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterBlock"** ):  
 listener.enterBlock(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitBlock"** ):  
 listener.exitBlock(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitBlock"** ):  
 **return** visitor.visitBlock(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** block(self):  
  
 localctx = ExpressionParser.BlockContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 2, self.RULE\_block)  
 self.\_la = 0 *# Token type* **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 58  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **while** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.MOSAIC\_IF) | (1 << ExpressionParser.WHILE) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING) | (1 << ExpressionParser.OTHER))) != 0):  
 self.state = 55  
 self.statements()  
 self.state = 60  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** StatementsContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_statements  
  
   
 **def** copyFrom(self, ctx:ParserRuleContext):  
 super().copyFrom(ctx)  
  
  
  
 **class** AssignmentExprContext(StatementsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.StatementsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** assignment(self):  
 **return** self.getTypedRuleContext(ExpressionParser.AssignmentContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterAssignmentExpr"** ):  
 listener.enterAssignmentExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitAssignmentExpr"** ):  
 listener.exitAssignmentExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitAssignmentExpr"** ):  
 **return** visitor.visitAssignmentExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** WhileStatExprContext(StatementsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.StatementsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** whileStat(self):  
 **return** self.getTypedRuleContext(ExpressionParser.WhileStatContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterWhileStatExpr"** ):  
 listener.enterWhileStatExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitWhileStatExpr"** ):  
 listener.exitWhileStatExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitWhileStatExpr"** ):  
 **return** visitor.visitWhileStatExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** IfStatExprContext(StatementsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.StatementsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** ifStat(self):  
 **return** self.getTypedRuleContext(ExpressionParser.IfStatContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterIfStatExpr"** ):  
 listener.enterIfStatExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitIfStatExpr"** ):  
 listener.exitIfStatExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitIfStatExpr"** ):  
 **return** visitor.visitIfStatExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** OtherExprContext(StatementsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.StatementsContext* super().\_\_init\_\_(parser)  
 self.\_OTHER = **None** *# Token* self.copyFrom(ctx)  
  
 **def** OTHER(self):  
 **return** self.getToken(ExpressionParser.OTHER, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterOtherExpr"** ):  
 listener.enterOtherExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitOtherExpr"** ):  
 listener.exitOtherExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitOtherExpr"** ):  
 **return** visitor.visitOtherExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** StatementExprContext(StatementsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.StatementsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** statement(self):  
 **return** self.getTypedRuleContext(ExpressionParser.StatementContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterStatementExpr"** ):  
 listener.enterStatementExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitStatementExpr"** ):  
 listener.exitStatementExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitStatementExpr"** ):  
 **return** visitor.visitStatementExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
 **def** statements(self):  
  
 localctx = ExpressionParser.StatementsContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 4, self.RULE\_statements)  
 **try**:  
 self.state = 67  
 self.\_errHandler.sync(self)  
 la\_ = self.\_interp.adaptivePredict(self.\_input,1,self.\_ctx)  
 **if** la\_ == 1:  
 localctx = ExpressionParser.AssignmentExprContext(self, localctx)  
 self.enterOuterAlt(localctx, 1)  
 self.state = 61  
 self.assignment()  
 **pass  
  
 elif** la\_ == 2:  
 localctx = ExpressionParser.IfStatExprContext(self, localctx)  
 self.enterOuterAlt(localctx, 2)  
 self.state = 62  
 self.ifStat()  
 **pass  
  
 elif** la\_ == 3:  
 localctx = ExpressionParser.WhileStatExprContext(self, localctx)  
 self.enterOuterAlt(localctx, 3)  
 self.state = 63  
 self.whileStat()  
 **pass  
  
 elif** la\_ == 4:  
 localctx = ExpressionParser.StatementExprContext(self, localctx)  
 self.enterOuterAlt(localctx, 4)  
 self.state = 64  
 self.statement()  
 **pass  
  
 elif** la\_ == 5:  
 localctx = ExpressionParser.OtherExprContext(self, localctx)  
 self.enterOuterAlt(localctx, 5)  
 self.state = 65  
 localctx.\_OTHER = self.match(ExpressionParser.OTHER)  
 System.err.println(**"unknown char: "** + (**None if** localctx.\_OTHER **is None else** localctx.\_OTHER.text));  
 **pass  
  
  
 except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** AssignmentContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** ID(self):  
 **return** self.getToken(ExpressionParser.ID, 0)  
  
 **def** ASSIGN(self):  
 **return** self.getToken(ExpressionParser.ASSIGN, 0)  
  
 **def** expr(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,0)  
  
  
 **def** SCOL(self):  
 **return** self.getToken(ExpressionParser.SCOL, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_assignment  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterAssignment"** ):  
 listener.enterAssignment(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitAssignment"** ):  
 listener.exitAssignment(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitAssignment"** ):  
 **return** visitor.visitAssignment(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** assignment(self):  
  
 localctx = ExpressionParser.AssignmentContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 6, self.RULE\_assignment)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 69  
 self.match(ExpressionParser.ID)  
 self.state = 70  
 self.match(ExpressionParser.ASSIGN)  
 self.state = 71  
 self.expr(0)  
 self.state = 72  
 self.match(ExpressionParser.SCOL)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** IfStatContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** MOSAIC\_IF(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTokens(ExpressionParser.MOSAIC\_IF)  
 **else**:  
 **return** self.getToken(ExpressionParser.MOSAIC\_IF, i)  
  
 **def** conditionBlock(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ConditionBlockContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ConditionBlockContext,i)  
  
  
 **def** MOSAIC\_ELSE(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTokens(ExpressionParser.MOSAIC\_ELSE)  
 **else**:  
 **return** self.getToken(ExpressionParser.MOSAIC\_ELSE, i)  
  
 **def** statBlock(self):  
 **return** self.getTypedRuleContext(ExpressionParser.StatBlockContext,0)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_ifStat  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterIfStat"** ):  
 listener.enterIfStat(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitIfStat"** ):  
 listener.exitIfStat(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitIfStat"** ):  
 **return** visitor.visitIfStat(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** ifStat(self):  
  
 localctx = ExpressionParser.IfStatContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 8, self.RULE\_ifStat)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 74  
 self.match(ExpressionParser.MOSAIC\_IF)  
 self.state = 75  
 self.conditionBlock()  
 self.state = 81  
 self.\_errHandler.sync(self)  
 \_alt = self.\_interp.adaptivePredict(self.\_input,2,self.\_ctx)  
 **while** \_alt!=2 **and** \_alt!=ATN.INVALID\_ALT\_NUMBER:  
 **if** \_alt==1:  
 self.state = 76  
 self.match(ExpressionParser.MOSAIC\_ELSE)  
 self.state = 77  
 self.match(ExpressionParser.MOSAIC\_IF)  
 self.state = 78  
 self.conditionBlock()   
 self.state = 83  
 self.\_errHandler.sync(self)  
 \_alt = self.\_interp.adaptivePredict(self.\_input,2,self.\_ctx)  
  
 self.state = 86  
 self.\_errHandler.sync(self)  
 la\_ = self.\_interp.adaptivePredict(self.\_input,3,self.\_ctx)  
 **if** la\_ == 1:  
 self.state = 84  
 self.match(ExpressionParser.MOSAIC\_ELSE)  
 self.state = 85  
 self.statBlock()  
  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** ConditionBlockContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** expr(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,0)  
  
  
 **def** statBlock(self):  
 **return** self.getTypedRuleContext(ExpressionParser.StatBlockContext,0)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_conditionBlock  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterConditionBlock"** ):  
 listener.enterConditionBlock(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitConditionBlock"** ):  
 listener.exitConditionBlock(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitConditionBlock"** ):  
 **return** visitor.visitConditionBlock(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** conditionBlock(self):  
  
 localctx = ExpressionParser.ConditionBlockContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 10, self.RULE\_conditionBlock)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 88  
 self.expr(0)  
 self.state = 89  
 self.statBlock()  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** StatBlockContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** OBRACE(self):  
 **return** self.getToken(ExpressionParser.OBRACE, 0)  
  
 **def** block(self):  
 **return** self.getTypedRuleContext(ExpressionParser.BlockContext,0)  
  
  
 **def** CBRACE(self):  
 **return** self.getToken(ExpressionParser.CBRACE, 0)  
  
 **def** statements(self):  
 **return** self.getTypedRuleContext(ExpressionParser.StatementsContext,0)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_statBlock  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterStatBlock"** ):  
 listener.enterStatBlock(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitStatBlock"** ):  
 listener.exitStatBlock(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitStatBlock"** ):  
 **return** visitor.visitStatBlock(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** statBlock(self):  
  
 localctx = ExpressionParser.StatBlockContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 12, self.RULE\_statBlock)  
 **try**:  
 self.state = 96  
 self.\_errHandler.sync(self)  
 token = self.\_input.LA(1)  
 **if** token **in** [ExpressionParser.OBRACE]:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 91  
 self.match(ExpressionParser.OBRACE)  
 self.state = 92  
 self.block()  
 self.state = 93  
 self.match(ExpressionParser.CBRACE)  
 **pass  
 elif** token **in** [ExpressionParser.FUNCTIONNAME\_PARAM0, ExpressionParser.FUNCTIONNAME\_PARAM1, ExpressionParser.FUNCTIONNAME\_PARAM2, ExpressionParser.FUNCTIONNAME\_PARAM3, ExpressionParser.FUNCTIONNAME\_PARAM4, ExpressionParser.FUNCTIONNAME\_PARAM\_N, ExpressionParser.MINUS, ExpressionParser.NOT, ExpressionParser.LPAREN, ExpressionParser.BEGL, ExpressionParser.MOSAIC\_TRUE, ExpressionParser.MOSAIC\_FALSE, ExpressionParser.NIL, ExpressionParser.MOSAIC\_IF, ExpressionParser.WHILE, ExpressionParser.ID, ExpressionParser.INT, ExpressionParser.DOUBLE, ExpressionParser.STRING, ExpressionParser.OTHER]:  
 self.enterOuterAlt(localctx, 2)  
 self.state = 95  
 self.statements()  
 **pass  
 else**:  
 **raise** NoViableAltException(self)  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** WhileStatContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** WHILE(self):  
 **return** self.getToken(ExpressionParser.WHILE, 0)  
  
 **def** expr(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,0)  
  
  
 **def** statBlock(self):  
 **return** self.getTypedRuleContext(ExpressionParser.StatBlockContext,0)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_whileStat  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterWhileStat"** ):  
 listener.enterWhileStat(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitWhileStat"** ):  
 listener.exitWhileStat(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitWhileStat"** ):  
 **return** visitor.visitWhileStat(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** whileStat(self):  
  
 localctx = ExpressionParser.WhileStatContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 14, self.RULE\_whileStat)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 98  
 self.match(ExpressionParser.WHILE)  
 self.state = 99  
 self.expr(0)  
 self.state = 100  
 self.statBlock()  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** StatementContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** expr(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,0)  
  
  
 **def** SCOL(self):  
 **return** self.getToken(ExpressionParser.SCOL, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_statement  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterStatement"** ):  
 listener.enterStatement(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitStatement"** ):  
 listener.exitStatement(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitStatement"** ):  
 **return** visitor.visitStatement(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** statement(self):  
  
 localctx = ExpressionParser.StatementContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 16, self.RULE\_statement)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 102  
 self.expr(0)  
 self.state = 103  
 self.match(ExpressionParser.SCOL)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** ExprContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_expr  
  
   
 **def** copyFrom(self, ctx:ParserRuleContext):  
 super().copyFrom(ctx)  
  
  
 **class** NotExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** NOT(self):  
 **return** self.getToken(ExpressionParser.NOT, 0)  
 **def** expr(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterNotExpr"** ):  
 listener.enterNotExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitNotExpr"** ):  
 listener.exitNotExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitNotExpr"** ):  
 **return** visitor.visitNotExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** UnaryMinusExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** MINUS(self):  
 **return** self.getToken(ExpressionParser.MINUS, 0)  
 **def** expr(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterUnaryMinusExpr"** ):  
 listener.enterUnaryMinusExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitUnaryMinusExpr"** ):  
 listener.exitUnaryMinusExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitUnaryMinusExpr"** ):  
 **return** visitor.visitUnaryMinusExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** MultiplicationExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.op = **None** *# Token* self.copyFrom(ctx)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
 **def** MULT(self):  
 **return** self.getToken(ExpressionParser.MULT, 0)  
 **def** DIV(self):  
 **return** self.getToken(ExpressionParser.DIV, 0)  
 **def** MOD(self):  
 **return** self.getToken(ExpressionParser.MOD, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterMultiplicationExpr"** ):  
 listener.enterMultiplicationExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitMultiplicationExpr"** ):  
 listener.exitMultiplicationExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitMultiplicationExpr"** ):  
 **return** visitor.visitMultiplicationExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** AtomExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** atom(self):  
 **return** self.getTypedRuleContext(ExpressionParser.AtomContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterAtomExpr"** ):  
 listener.enterAtomExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitAtomExpr"** ):  
 listener.exitAtomExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitAtomExpr"** ):  
 **return** visitor.visitAtomExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** OrExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
 **def** MOSAIC\_OR(self):  
 **return** self.getToken(ExpressionParser.MOSAIC\_OR, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterOrExpr"** ):  
 listener.enterOrExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitOrExpr"** ):  
 listener.exitOrExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitOrExpr"** ):  
 **return** visitor.visitOrExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** AdditiveExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.op = **None** *# Token* self.copyFrom(ctx)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
 **def** PLUS(self):  
 **return** self.getToken(ExpressionParser.PLUS, 0)  
 **def** MINUS(self):  
 **return** self.getToken(ExpressionParser.MINUS, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterAdditiveExpr"** ):  
 listener.enterAdditiveExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitAdditiveExpr"** ):  
 listener.exitAdditiveExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitAdditiveExpr"** ):  
 **return** visitor.visitAdditiveExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** PowExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
 **def** POW(self):  
 **return** self.getToken(ExpressionParser.POW, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterPowExpr"** ):  
 listener.enterPowExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitPowExpr"** ):  
 listener.exitPowExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitPowExpr"** ):  
 **return** visitor.visitPowExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** RelationalExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.op = **None** *# Token* self.copyFrom(ctx)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
 **def** LTEQ(self):  
 **return** self.getToken(ExpressionParser.LTEQ, 0)  
 **def** GTEQ(self):  
 **return** self.getToken(ExpressionParser.GTEQ, 0)  
 **def** LT(self):  
 **return** self.getToken(ExpressionParser.LT, 0)  
 **def** GT(self):  
 **return** self.getToken(ExpressionParser.GT, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterRelationalExpr"** ):  
 listener.enterRelationalExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitRelationalExpr"** ):  
 listener.exitRelationalExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitRelationalExpr"** ):  
 **return** visitor.visitRelationalExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** EqualityExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.op = **None** *# Token* self.copyFrom(ctx)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
 **def** EQ(self):  
 **return** self.getToken(ExpressionParser.EQ, 0)  
 **def** NEQ(self):  
 **return** self.getToken(ExpressionParser.NEQ, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterEqualityExpr"** ):  
 listener.enterEqualityExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitEqualityExpr"** ):  
 listener.exitEqualityExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitEqualityExpr"** ):  
 **return** visitor.visitEqualityExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** FunctionExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** functions(self):  
 **return** self.getTypedRuleContext(ExpressionParser.FunctionsContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterFunctionExpr"** ):  
 listener.enterFunctionExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitFunctionExpr"** ):  
 listener.exitFunctionExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitFunctionExpr"** ):  
 **return** visitor.visitFunctionExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** AndExprContext(ExprContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ExprContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
 **def** MOSAIC\_AND(self):  
 **return** self.getToken(ExpressionParser.MOSAIC\_AND, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterAndExpr"** ):  
 listener.enterAndExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitAndExpr"** ):  
 listener.exitAndExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitAndExpr"** ):  
 **return** visitor.visitAndExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
 **def** expr(self, \_p:int=0):  
 \_parentctx = self.\_ctx  
 \_parentState = self.state  
 localctx = ExpressionParser.ExprContext(self, self.\_ctx, \_parentState)  
 \_prevctx = localctx  
 \_startState = 18  
 self.enterRecursionRule(localctx, 18, self.RULE\_expr, \_p)  
 self.\_la = 0 *# Token type* **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 112  
 self.\_errHandler.sync(self)  
 token = self.\_input.LA(1)  
 **if** token **in** [ExpressionParser.FUNCTIONNAME\_PARAM0, ExpressionParser.FUNCTIONNAME\_PARAM1, ExpressionParser.FUNCTIONNAME\_PARAM2, ExpressionParser.FUNCTIONNAME\_PARAM3, ExpressionParser.FUNCTIONNAME\_PARAM4, ExpressionParser.FUNCTIONNAME\_PARAM\_N]:  
 localctx = ExpressionParser.FunctionExprContext(self, localctx)  
 self.\_ctx = localctx  
 \_prevctx = localctx  
  
 self.state = 106  
 self.functions()  
 **pass  
 elif** token **in** [ExpressionParser.MINUS]:  
 localctx = ExpressionParser.UnaryMinusExprContext(self, localctx)  
 self.\_ctx = localctx  
 \_prevctx = localctx  
 self.state = 107  
 self.match(ExpressionParser.MINUS)  
 self.state = 108  
 self.expr(9)  
 **pass  
 elif** token **in** [ExpressionParser.NOT]:  
 localctx = ExpressionParser.NotExprContext(self, localctx)  
 self.\_ctx = localctx  
 \_prevctx = localctx  
 self.state = 109  
 self.match(ExpressionParser.NOT)  
 self.state = 110  
 self.expr(8)  
 **pass  
 elif** token **in** [ExpressionParser.LPAREN, ExpressionParser.BEGL, ExpressionParser.MOSAIC\_TRUE, ExpressionParser.MOSAIC\_FALSE, ExpressionParser.NIL, ExpressionParser.ID, ExpressionParser.INT, ExpressionParser.DOUBLE, ExpressionParser.STRING]:  
 localctx = ExpressionParser.AtomExprContext(self, localctx)  
 self.\_ctx = localctx  
 \_prevctx = localctx  
 self.state = 111  
 self.atom()  
 **pass  
 else**:  
 **raise** NoViableAltException(self)  
  
 self.\_ctx.stop = self.\_input.LT(-1)  
 self.state = 137  
 self.\_errHandler.sync(self)  
 \_alt = self.\_interp.adaptivePredict(self.\_input,7,self.\_ctx)  
 **while** \_alt!=2 **and** \_alt!=ATN.INVALID\_ALT\_NUMBER:  
 **if** \_alt==1:  
 **if** self.\_parseListeners **is not None**:  
 self.triggerExitRuleEvent()  
 \_prevctx = localctx  
 self.state = 135  
 self.\_errHandler.sync(self)  
 la\_ = self.\_interp.adaptivePredict(self.\_input,6,self.\_ctx)  
 **if** la\_ == 1:  
 localctx = ExpressionParser.PowExprContext(self, ExpressionParser.ExprContext(self, \_parentctx, \_parentState))  
 self.pushNewRecursionContext(localctx, \_startState, self.RULE\_expr)  
 self.state = 114  
 **if not** self.precpred(self.\_ctx, 10):  
 **from** antlr4.error.Errors **import** FailedPredicateException  
 **raise** FailedPredicateException(self, **"self.precpred(self.\_ctx, 10)"**)  
 self.state = 115  
 self.match(ExpressionParser.POW)  
 self.state = 116  
 self.expr(11)  
 **pass  
  
 elif** la\_ == 2:  
 localctx = ExpressionParser.MultiplicationExprContext(self, ExpressionParser.ExprContext(self, \_parentctx, \_parentState))  
 self.pushNewRecursionContext(localctx, \_startState, self.RULE\_expr)  
 self.state = 117  
 **if not** self.precpred(self.\_ctx, 7):  
 **from** antlr4.error.Errors **import** FailedPredicateException  
 **raise** FailedPredicateException(self, **"self.precpred(self.\_ctx, 7)"**)  
 self.state = 118  
 localctx.op = self.\_input.LT(1)  
 \_la = self.\_input.LA(1)  
 **if not**((((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.MULT) | (1 << ExpressionParser.DIV) | (1 << ExpressionParser.MOD))) != 0)):  
 localctx.op = self.\_errHandler.recoverInline(self)  
 **else**:  
 self.\_errHandler.reportMatch(self)  
 self.consume()  
 self.state = 119  
 self.expr(8)  
 **pass  
  
 elif** la\_ == 3:  
 localctx = ExpressionParser.AdditiveExprContext(self, ExpressionParser.ExprContext(self, \_parentctx, \_parentState))  
 self.pushNewRecursionContext(localctx, \_startState, self.RULE\_expr)  
 self.state = 120  
 **if not** self.precpred(self.\_ctx, 6):  
 **from** antlr4.error.Errors **import** FailedPredicateException  
 **raise** FailedPredicateException(self, **"self.precpred(self.\_ctx, 6)"**)  
 self.state = 121  
 localctx.op = self.\_input.LT(1)  
 \_la = self.\_input.LA(1)  
 **if not**(\_la==ExpressionParser.PLUS **or** \_la==ExpressionParser.MINUS):  
 localctx.op = self.\_errHandler.recoverInline(self)  
 **else**:  
 self.\_errHandler.reportMatch(self)  
 self.consume()  
 self.state = 122  
 self.expr(7)  
 **pass  
  
 elif** la\_ == 4:  
 localctx = ExpressionParser.RelationalExprContext(self, ExpressionParser.ExprContext(self, \_parentctx, \_parentState))  
 self.pushNewRecursionContext(localctx, \_startState, self.RULE\_expr)  
 self.state = 123  
 **if not** self.precpred(self.\_ctx, 5):  
 **from** antlr4.error.Errors **import** FailedPredicateException  
 **raise** FailedPredicateException(self, **"self.precpred(self.\_ctx, 5)"**)  
 self.state = 124  
 localctx.op = self.\_input.LT(1)  
 \_la = self.\_input.LA(1)  
 **if not**((((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.GT) | (1 << ExpressionParser.LT) | (1 << ExpressionParser.GTEQ) | (1 << ExpressionParser.LTEQ))) != 0)):  
 localctx.op = self.\_errHandler.recoverInline(self)  
 **else**:  
 self.\_errHandler.reportMatch(self)  
 self.consume()  
 self.state = 125  
 self.expr(6)  
 **pass  
  
 elif** la\_ == 5:  
 localctx = ExpressionParser.EqualityExprContext(self, ExpressionParser.ExprContext(self, \_parentctx, \_parentState))  
 self.pushNewRecursionContext(localctx, \_startState, self.RULE\_expr)  
 self.state = 126  
 **if not** self.precpred(self.\_ctx, 4):  
 **from** antlr4.error.Errors **import** FailedPredicateException  
 **raise** FailedPredicateException(self, **"self.precpred(self.\_ctx, 4)"**)  
 self.state = 127  
 localctx.op = self.\_input.LT(1)  
 \_la = self.\_input.LA(1)  
 **if not**(\_la==ExpressionParser.EQ **or** \_la==ExpressionParser.NEQ):  
 localctx.op = self.\_errHandler.recoverInline(self)  
 **else**:  
 self.\_errHandler.reportMatch(self)  
 self.consume()  
 self.state = 128  
 self.expr(5)  
 **pass  
  
 elif** la\_ == 6:  
 localctx = ExpressionParser.AndExprContext(self, ExpressionParser.ExprContext(self, \_parentctx, \_parentState))  
 self.pushNewRecursionContext(localctx, \_startState, self.RULE\_expr)  
 self.state = 129  
 **if not** self.precpred(self.\_ctx, 3):  
 **from** antlr4.error.Errors **import** FailedPredicateException  
 **raise** FailedPredicateException(self, **"self.precpred(self.\_ctx, 3)"**)  
 self.state = 130  
 self.match(ExpressionParser.MOSAIC\_AND)  
 self.state = 131  
 self.expr(4)  
 **pass  
  
 elif** la\_ == 7:  
 localctx = ExpressionParser.OrExprContext(self, ExpressionParser.ExprContext(self, \_parentctx, \_parentState))  
 self.pushNewRecursionContext(localctx, \_startState, self.RULE\_expr)  
 self.state = 132  
 **if not** self.precpred(self.\_ctx, 2):  
 **from** antlr4.error.Errors **import** FailedPredicateException  
 **raise** FailedPredicateException(self, **"self.precpred(self.\_ctx, 2)"**)  
 self.state = 133  
 self.match(ExpressionParser.MOSAIC\_OR)  
 self.state = 134  
 self.expr(3)  
 **pass** self.state = 139  
 self.\_errHandler.sync(self)  
 \_alt = self.\_interp.adaptivePredict(self.\_input,7,self.\_ctx)  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.unrollRecursionContexts(\_parentctx)  
 **return** localctx  
  
 **class** FunctionsContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_functions  
  
   
 **def** copyFrom(self, ctx:ParserRuleContext):  
 super().copyFrom(ctx)  
  
  
  
 **class** TwoParamterFunctionsContext(FunctionsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.FunctionsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** functionParam2(self):  
 **return** self.getTypedRuleContext(ExpressionParser.FunctionParam2Context,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterTwoParamterFunctions"** ):  
 listener.enterTwoParamterFunctions(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitTwoParamterFunctions"** ):  
 listener.exitTwoParamterFunctions(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitTwoParamterFunctions"** ):  
 **return** visitor.visitTwoParamterFunctions(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** FourParamterFunctionsContext(FunctionsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.FunctionsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** functionParam4(self):  
 **return** self.getTypedRuleContext(ExpressionParser.FunctionParam4Context,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterFourParamterFunctions"** ):  
 listener.enterFourParamterFunctions(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitFourParamterFunctions"** ):  
 listener.exitFourParamterFunctions(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitFourParamterFunctions"** ):  
 **return** visitor.visitFourParamterFunctions(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** NParametersFunctionsContext(FunctionsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.FunctionsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** functionParamN(self):  
 **return** self.getTypedRuleContext(ExpressionParser.FunctionParamNContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterNParametersFunctions"** ):  
 listener.enterNParametersFunctions(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitNParametersFunctions"** ):  
 listener.exitNParametersFunctions(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitNParametersFunctions"** ):  
 **return** visitor.visitNParametersFunctions(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** SingleParamterFunctionsContext(FunctionsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.FunctionsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** functionParam1(self):  
 **return** self.getTypedRuleContext(ExpressionParser.FunctionParam1Context,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterSingleParamterFunctions"** ):  
 listener.enterSingleParamterFunctions(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitSingleParamterFunctions"** ):  
 listener.exitSingleParamterFunctions(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitSingleParamterFunctions"** ):  
 **return** visitor.visitSingleParamterFunctions(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** ZeroParamterFunctionsContext(FunctionsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.FunctionsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** functionParam0(self):  
 **return** self.getTypedRuleContext(ExpressionParser.FunctionParam0Context,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterZeroParamterFunctions"** ):  
 listener.enterZeroParamterFunctions(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitZeroParamterFunctions"** ):  
 listener.exitZeroParamterFunctions(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitZeroParamterFunctions"** ):  
 **return** visitor.visitZeroParamterFunctions(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** ThreeParamterFunctionsContext(FunctionsContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.FunctionsContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** functionParam3(self):  
 **return** self.getTypedRuleContext(ExpressionParser.FunctionParam3Context,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterThreeParamterFunctions"** ):  
 listener.enterThreeParamterFunctions(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitThreeParamterFunctions"** ):  
 listener.exitThreeParamterFunctions(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitThreeParamterFunctions"** ):  
 **return** visitor.visitThreeParamterFunctions(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
 **def** functions(self):  
  
 localctx = ExpressionParser.FunctionsContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 20, self.RULE\_functions)  
 **try**:  
 self.state = 146  
 self.\_errHandler.sync(self)  
 token = self.\_input.LA(1)  
 **if** token **in** [ExpressionParser.FUNCTIONNAME\_PARAM0]:  
 localctx = ExpressionParser.ZeroParamterFunctionsContext(self, localctx)  
 self.enterOuterAlt(localctx, 1)  
 self.state = 140  
 self.functionParam0()  
 **pass  
 elif** token **in** [ExpressionParser.FUNCTIONNAME\_PARAM1]:  
 localctx = ExpressionParser.SingleParamterFunctionsContext(self, localctx)  
 self.enterOuterAlt(localctx, 2)  
 self.state = 141  
 self.functionParam1()  
 **pass  
 elif** token **in** [ExpressionParser.FUNCTIONNAME\_PARAM2]:  
 localctx = ExpressionParser.TwoParamterFunctionsContext(self, localctx)  
 self.enterOuterAlt(localctx, 3)  
 self.state = 142  
 self.functionParam2()  
 **pass  
 elif** token **in** [ExpressionParser.FUNCTIONNAME\_PARAM3]:  
 localctx = ExpressionParser.ThreeParamterFunctionsContext(self, localctx)  
 self.enterOuterAlt(localctx, 4)  
 self.state = 143  
 self.functionParam3()  
 **pass  
 elif** token **in** [ExpressionParser.FUNCTIONNAME\_PARAM4]:  
 localctx = ExpressionParser.FourParamterFunctionsContext(self, localctx)  
 self.enterOuterAlt(localctx, 5)  
 self.state = 144  
 self.functionParam4()  
 **pass  
 elif** token **in** [ExpressionParser.FUNCTIONNAME\_PARAM\_N]:  
 localctx = ExpressionParser.NParametersFunctionsContext(self, localctx)  
 self.enterOuterAlt(localctx, 6)  
 self.state = 145  
 self.functionParamN()  
 **pass  
 else**:  
 **raise** NoViableAltException(self)  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** FunctionParam0Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** FUNCTIONNAME\_PARAM0(self):  
 **return** self.getToken(ExpressionParser.FUNCTIONNAME\_PARAM0, 0)  
  
 **def** LPAREN(self):  
 **return** self.getToken(ExpressionParser.LPAREN, 0)  
  
 **def** RPAREN(self):  
 **return** self.getToken(ExpressionParser.RPAREN, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_functionParam0  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterFunctionParam0"** ):  
 listener.enterFunctionParam0(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitFunctionParam0"** ):  
 listener.exitFunctionParam0(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitFunctionParam0"** ):  
 **return** visitor.visitFunctionParam0(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** functionParam0(self):  
  
 localctx = ExpressionParser.FunctionParam0Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 22, self.RULE\_functionParam0)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 148  
 self.match(ExpressionParser.FUNCTIONNAME\_PARAM0)  
 self.state = 149  
 self.match(ExpressionParser.LPAREN)  
 self.state = 150  
 self.match(ExpressionParser.RPAREN)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** FunctionParam1Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** FUNCTIONNAME\_PARAM1(self):  
 **return** self.getToken(ExpressionParser.FUNCTIONNAME\_PARAM1, 0)  
  
 **def** LPAREN(self):  
 **return** self.getToken(ExpressionParser.LPAREN, 0)  
  
 **def** arguments1(self):  
 **return** self.getTypedRuleContext(ExpressionParser.Arguments1Context,0)  
  
  
 **def** RPAREN(self):  
 **return** self.getToken(ExpressionParser.RPAREN, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_functionParam1  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterFunctionParam1"** ):  
 listener.enterFunctionParam1(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitFunctionParam1"** ):  
 listener.exitFunctionParam1(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitFunctionParam1"** ):  
 **return** visitor.visitFunctionParam1(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** functionParam1(self):  
  
 localctx = ExpressionParser.FunctionParam1Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 24, self.RULE\_functionParam1)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 152  
 self.match(ExpressionParser.FUNCTIONNAME\_PARAM1)  
 self.state = 153  
 self.match(ExpressionParser.LPAREN)  
 self.state = 154  
 self.arguments1()  
 self.state = 155  
 self.match(ExpressionParser.RPAREN)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** FunctionParam2Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** FUNCTIONNAME\_PARAM2(self):  
 **return** self.getToken(ExpressionParser.FUNCTIONNAME\_PARAM2, 0)  
  
 **def** LPAREN(self):  
 **return** self.getToken(ExpressionParser.LPAREN, 0)  
  
 **def** arguments2(self):  
 **return** self.getTypedRuleContext(ExpressionParser.Arguments2Context,0)  
  
  
 **def** RPAREN(self):  
 **return** self.getToken(ExpressionParser.RPAREN, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_functionParam2  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterFunctionParam2"** ):  
 listener.enterFunctionParam2(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitFunctionParam2"** ):  
 listener.exitFunctionParam2(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitFunctionParam2"** ):  
 **return** visitor.visitFunctionParam2(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** functionParam2(self):  
  
 localctx = ExpressionParser.FunctionParam2Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 26, self.RULE\_functionParam2)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 157  
 self.match(ExpressionParser.FUNCTIONNAME\_PARAM2)  
 self.state = 158  
 self.match(ExpressionParser.LPAREN)  
 self.state = 159  
 self.arguments2()  
 self.state = 160  
 self.match(ExpressionParser.RPAREN)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** FunctionParam3Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** FUNCTIONNAME\_PARAM3(self):  
 **return** self.getToken(ExpressionParser.FUNCTIONNAME\_PARAM3, 0)  
  
 **def** LPAREN(self):  
 **return** self.getToken(ExpressionParser.LPAREN, 0)  
  
 **def** arguments3(self):  
 **return** self.getTypedRuleContext(ExpressionParser.Arguments3Context,0)  
  
  
 **def** RPAREN(self):  
 **return** self.getToken(ExpressionParser.RPAREN, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_functionParam3  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterFunctionParam3"** ):  
 listener.enterFunctionParam3(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitFunctionParam3"** ):  
 listener.exitFunctionParam3(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitFunctionParam3"** ):  
 **return** visitor.visitFunctionParam3(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** functionParam3(self):  
  
 localctx = ExpressionParser.FunctionParam3Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 28, self.RULE\_functionParam3)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 162  
 self.match(ExpressionParser.FUNCTIONNAME\_PARAM3)  
 self.state = 163  
 self.match(ExpressionParser.LPAREN)  
 self.state = 164  
 self.arguments3()  
 self.state = 165  
 self.match(ExpressionParser.RPAREN)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** FunctionParam4Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** FUNCTIONNAME\_PARAM4(self):  
 **return** self.getToken(ExpressionParser.FUNCTIONNAME\_PARAM4, 0)  
  
 **def** LPAREN(self):  
 **return** self.getToken(ExpressionParser.LPAREN, 0)  
  
 **def** arguments4(self):  
 **return** self.getTypedRuleContext(ExpressionParser.Arguments4Context,0)  
  
  
 **def** RPAREN(self):  
 **return** self.getToken(ExpressionParser.RPAREN, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_functionParam4  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterFunctionParam4"** ):  
 listener.enterFunctionParam4(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitFunctionParam4"** ):  
 listener.exitFunctionParam4(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitFunctionParam4"** ):  
 **return** visitor.visitFunctionParam4(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** functionParam4(self):  
  
 localctx = ExpressionParser.FunctionParam4Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 30, self.RULE\_functionParam4)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 167  
 self.match(ExpressionParser.FUNCTIONNAME\_PARAM4)  
 self.state = 168  
 self.match(ExpressionParser.LPAREN)  
 self.state = 169  
 self.arguments4()  
 self.state = 170  
 self.match(ExpressionParser.RPAREN)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** FunctionParamNContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** FUNCTIONNAME\_PARAM\_N(self):  
 **return** self.getToken(ExpressionParser.FUNCTIONNAME\_PARAM\_N, 0)  
  
 **def** LPAREN(self):  
 **return** self.getToken(ExpressionParser.LPAREN, 0)  
  
 **def** argumentsN(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ArgumentsNContext,0)  
  
  
 **def** RPAREN(self):  
 **return** self.getToken(ExpressionParser.RPAREN, 0)  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_functionParamN  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterFunctionParamN"** ):  
 listener.enterFunctionParamN(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitFunctionParamN"** ):  
 listener.exitFunctionParamN(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitFunctionParamN"** ):  
 **return** visitor.visitFunctionParamN(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** functionParamN(self):  
  
 localctx = ExpressionParser.FunctionParamNContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 32, self.RULE\_functionParamN)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 172  
 self.match(ExpressionParser.FUNCTIONNAME\_PARAM\_N)  
 self.state = 173  
 self.match(ExpressionParser.LPAREN)  
 self.state = 174  
 self.argumentsN()  
 self.state = 175  
 self.match(ExpressionParser.RPAREN)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** ArgumentsNContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_argumentsN  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterArgumentsN"** ):  
 listener.enterArgumentsN(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitArgumentsN"** ):  
 listener.exitArgumentsN(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitArgumentsN"** ):  
 **return** visitor.visitArgumentsN(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** argumentsN(self):  
  
 localctx = ExpressionParser.ArgumentsNContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 34, self.RULE\_argumentsN)  
 self.\_la = 0 *# Token type* **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 178  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 177  
 self.expr(0)  
  
  
 self.state = 186  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **while** \_la==ExpressionParser.COMMA:  
 self.state = 180  
 self.match(ExpressionParser.COMMA)  
 self.state = 182  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 181  
 self.expr(0)  
  
  
 self.state = 188  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** Arguments1Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
 self.param1 = **None** *# ExprContext* **def** expr(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,0)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_arguments1  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterArguments1"** ):  
 listener.enterArguments1(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitArguments1"** ):  
 listener.exitArguments1(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitArguments1"** ):  
 **return** visitor.visitArguments1(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** arguments1(self):  
  
 localctx = ExpressionParser.Arguments1Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 36, self.RULE\_arguments1)  
 self.\_la = 0 *# Token type* **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 190  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 189  
 localctx.param1 = self.expr(0)  
  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** Arguments2Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
 self.param1 = **None** *# ExprContext* self.param2 = **None** *# ExprContext* **def** COMMA(self):  
 **return** self.getToken(ExpressionParser.COMMA, 0)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_arguments2  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterArguments2"** ):  
 listener.enterArguments2(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitArguments2"** ):  
 listener.exitArguments2(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitArguments2"** ):  
 **return** visitor.visitArguments2(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** arguments2(self):  
  
 localctx = ExpressionParser.Arguments2Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 38, self.RULE\_arguments2)  
 self.\_la = 0 *# Token type* **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 193  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 192  
 localctx.param1 = self.expr(0)  
  
  
 self.state = 195  
 self.match(ExpressionParser.COMMA)  
 self.state = 197  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 196  
 localctx.param2 = self.expr(0)  
  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** Arguments3Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
 self.param1 = **None** *# ExprContext* self.param2 = **None** *# ExprContext* self.param3 = **None** *# ExprContext* **def** COMMA(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTokens(ExpressionParser.COMMA)  
 **else**:  
 **return** self.getToken(ExpressionParser.COMMA, i)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_arguments3  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterArguments3"** ):  
 listener.enterArguments3(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitArguments3"** ):  
 listener.exitArguments3(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitArguments3"** ):  
 **return** visitor.visitArguments3(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** arguments3(self):  
  
 localctx = ExpressionParser.Arguments3Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 40, self.RULE\_arguments3)  
 self.\_la = 0 *# Token type* **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 200  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 199  
 localctx.param1 = self.expr(0)  
  
  
 self.state = 202  
 self.match(ExpressionParser.COMMA)  
 self.state = 204  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 203  
 localctx.param2 = self.expr(0)  
  
  
 self.state = 206  
 self.match(ExpressionParser.COMMA)  
 self.state = 208  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 207  
 localctx.param3 = self.expr(0)  
  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** Arguments4Context(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
 self.param1 = **None** *# ExprContext* self.param2 = **None** *# ExprContext* self.param3 = **None** *# ExprContext* self.param4 = **None** *# ExprContext* **def** COMMA(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTokens(ExpressionParser.COMMA)  
 **else**:  
 **return** self.getToken(ExpressionParser.COMMA, i)  
  
 **def** expr(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ExprContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,i)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_arguments4  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterArguments4"** ):  
 listener.enterArguments4(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitArguments4"** ):  
 listener.exitArguments4(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitArguments4"** ):  
 **return** visitor.visitArguments4(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** arguments4(self):  
  
 localctx = ExpressionParser.Arguments4Context(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 42, self.RULE\_arguments4)  
 self.\_la = 0 *# Token type* **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 211  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 210  
 localctx.param1 = self.expr(0)  
  
  
 self.state = 213  
 self.match(ExpressionParser.COMMA)  
 self.state = 215  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 214  
 localctx.param2 = self.expr(0)  
  
  
 self.state = 217  
 self.match(ExpressionParser.COMMA)  
 self.state = 219  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 218  
 localctx.param3 = self.expr(0)  
  
  
 self.state = 221  
 self.match(ExpressionParser.COMMA)  
 self.state = 223  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.FUNCTIONNAME\_PARAM0) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM1) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM2) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM3) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM4) | (1 << ExpressionParser.FUNCTIONNAME\_PARAM\_N) | (1 << ExpressionParser.MINUS) | (1 << ExpressionParser.NOT) | (1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 222  
 localctx.param4 = self.expr(0)  
  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** AtomContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_atom  
  
   
 **def** copyFrom(self, ctx:ParserRuleContext):  
 super().copyFrom(ctx)  
  
  
  
 **class** ParExprContext(AtomContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.AtomContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** LPAREN(self):  
 **return** self.getToken(ExpressionParser.LPAREN, 0)  
 **def** expr(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ExprContext,0)  
  
 **def** RPAREN(self):  
 **return** self.getToken(ExpressionParser.RPAREN, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterParExpr"** ):  
 listener.enterParExpr(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitParExpr"** ):  
 listener.exitParExpr(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitParExpr"** ):  
 **return** visitor.visitParExpr(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** BooleanAtomContext(AtomContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.AtomContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** MOSAIC\_TRUE(self):  
 **return** self.getToken(ExpressionParser.MOSAIC\_TRUE, 0)  
 **def** MOSAIC\_FALSE(self):  
 **return** self.getToken(ExpressionParser.MOSAIC\_FALSE, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterBooleanAtom"** ):  
 listener.enterBooleanAtom(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitBooleanAtom"** ):  
 listener.exitBooleanAtom(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitBooleanAtom"** ):  
 **return** visitor.visitBooleanAtom(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** IdAtomContext(AtomContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.AtomContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** ID(self):  
 **return** self.getToken(ExpressionParser.ID, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterIdAtom"** ):  
 listener.enterIdAtom(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitIdAtom"** ):  
 listener.exitIdAtom(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitIdAtom"** ):  
 **return** visitor.visitIdAtom(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** ArrayAtomContext(AtomContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.AtomContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** valueList(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ValueListContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterArrayAtom"** ):  
 listener.enterArrayAtom(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitArrayAtom"** ):  
 listener.exitArrayAtom(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitArrayAtom"** ):  
 **return** visitor.visitArrayAtom(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** StringAtomContext(AtomContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.AtomContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** STRING(self):  
 **return** self.getToken(ExpressionParser.STRING, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterStringAtom"** ):  
 listener.enterStringAtom(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitStringAtom"** ):  
 listener.exitStringAtom(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitStringAtom"** ):  
 **return** visitor.visitStringAtom(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** NilAtomContext(AtomContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.AtomContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** NIL(self):  
 **return** self.getToken(ExpressionParser.NIL, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterNilAtom"** ):  
 listener.enterNilAtom(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitNilAtom"** ):  
 listener.exitNilAtom(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitNilAtom"** ):  
 **return** visitor.visitNilAtom(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
 **class** NumberAtomContext(AtomContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.AtomContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** INT(self):  
 **return** self.getToken(ExpressionParser.INT, 0)  
 **def** DOUBLE(self):  
 **return** self.getToken(ExpressionParser.DOUBLE, 0)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterNumberAtom"** ):  
 listener.enterNumberAtom(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitNumberAtom"** ):  
 listener.exitNumberAtom(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitNumberAtom"** ):  
 **return** visitor.visitNumberAtom(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
 **def** atom(self):  
  
 localctx = ExpressionParser.AtomContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 44, self.RULE\_atom)  
 self.\_la = 0 *# Token type* **try**:  
 self.state = 235  
 self.\_errHandler.sync(self)  
 token = self.\_input.LA(1)  
 **if** token **in** [ExpressionParser.LPAREN]:  
 localctx = ExpressionParser.ParExprContext(self, localctx)  
 self.enterOuterAlt(localctx, 1)  
 self.state = 225  
 self.match(ExpressionParser.LPAREN)  
 self.state = 226  
 self.expr(0)  
 self.state = 227  
 self.match(ExpressionParser.RPAREN)  
 **pass  
 elif** token **in** [ExpressionParser.BEGL]:  
 localctx = ExpressionParser.ArrayAtomContext(self, localctx)  
 self.enterOuterAlt(localctx, 2)  
 self.state = 229  
 self.valueList()  
 **pass  
 elif** token **in** [ExpressionParser.INT, ExpressionParser.DOUBLE]:  
 localctx = ExpressionParser.NumberAtomContext(self, localctx)  
 self.enterOuterAlt(localctx, 3)  
 self.state = 230  
 \_la = self.\_input.LA(1)  
 **if not**(\_la==ExpressionParser.INT **or** \_la==ExpressionParser.DOUBLE):  
 self.\_errHandler.recoverInline(self)  
 **else**:  
 self.\_errHandler.reportMatch(self)  
 self.consume()  
 **pass  
 elif** token **in** [ExpressionParser.MOSAIC\_TRUE, ExpressionParser.MOSAIC\_FALSE]:  
 localctx = ExpressionParser.BooleanAtomContext(self, localctx)  
 self.enterOuterAlt(localctx, 4)  
 self.state = 231  
 \_la = self.\_input.LA(1)  
 **if not**(\_la==ExpressionParser.MOSAIC\_TRUE **or** \_la==ExpressionParser.MOSAIC\_FALSE):  
 self.\_errHandler.recoverInline(self)  
 **else**:  
 self.\_errHandler.reportMatch(self)  
 self.consume()  
 **pass  
 elif** token **in** [ExpressionParser.ID]:  
 localctx = ExpressionParser.IdAtomContext(self, localctx)  
 self.enterOuterAlt(localctx, 5)  
 self.state = 232  
 self.match(ExpressionParser.ID)  
 **pass  
 elif** token **in** [ExpressionParser.STRING]:  
 localctx = ExpressionParser.StringAtomContext(self, localctx)  
 self.enterOuterAlt(localctx, 6)  
 self.state = 233  
 self.match(ExpressionParser.STRING)  
 **pass  
 elif** token **in** [ExpressionParser.NIL]:  
 localctx = ExpressionParser.NilAtomContext(self, localctx)  
 self.enterOuterAlt(localctx, 7)  
 self.state = 234  
 self.match(ExpressionParser.NIL)  
 **pass  
 else**:  
 **raise** NoViableAltException(self)  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** ValueListContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
 **def** BEGL(self):  
 **return** self.getToken(ExpressionParser.BEGL, 0)  
  
 **def** ENDL(self):  
 **return** self.getToken(ExpressionParser.ENDL, 0)  
  
 **def** array(self):  
 **return** self.getTypedRuleContext(ExpressionParser.ArrayContext,0)  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_valueList  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterValueList"** ):  
 listener.enterValueList(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitValueList"** ):  
 listener.exitValueList(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitValueList"** ):  
 **return** visitor.visitValueList(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
  
 **def** valueList(self):  
  
 localctx = ExpressionParser.ValueListContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 46, self.RULE\_valueList)  
 **try**:  
 self.enterOuterAlt(localctx, 1)  
 self.state = 237  
 self.match(ExpressionParser.BEGL)  
 self.state = 239  
 self.\_errHandler.sync(self)  
 la\_ = self.\_interp.adaptivePredict(self.\_input,23,self.\_ctx)  
 **if** la\_ == 1:  
 self.state = 238  
 self.array()  
  
  
 self.state = 241  
 self.match(ExpressionParser.ENDL)  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** ArrayContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_array  
  
   
 **def** copyFrom(self, ctx:ParserRuleContext):  
 super().copyFrom(ctx)  
  
  
  
 **class** ArrayValuesContext(ArrayContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ArrayContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** arrayElement(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTypedRuleContexts(ExpressionParser.ArrayElementContext)  
 **else**:  
 **return** self.getTypedRuleContext(ExpressionParser.ArrayElementContext,i)  
  
 **def** COMMA(self, i:int=**None**):  
 **if** i **is None**:  
 **return** self.getTokens(ExpressionParser.COMMA)  
 **else**:  
 **return** self.getToken(ExpressionParser.COMMA, i)  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterArrayValues"** ):  
 listener.enterArrayValues(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitArrayValues"** ):  
 listener.exitArrayValues(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitArrayValues"** ):  
 **return** visitor.visitArrayValues(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
 **def** array(self):  
  
 localctx = ExpressionParser.ArrayContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 48, self.RULE\_array)  
 self.\_la = 0 *# Token type* **try**:  
 localctx = ExpressionParser.ArrayValuesContext(self, localctx)  
 self.enterOuterAlt(localctx, 1)  
 self.state = 244  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 243  
 self.arrayElement()  
  
  
 self.state = 252  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **while** \_la==ExpressionParser.COMMA:  
 self.state = 246  
 self.match(ExpressionParser.COMMA)  
 self.state = 248  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
 **if** (((\_la) & ~0x3f) == 0 **and** ((1 << \_la) & ((1 << ExpressionParser.LPAREN) | (1 << ExpressionParser.BEGL) | (1 << ExpressionParser.MOSAIC\_TRUE) | (1 << ExpressionParser.MOSAIC\_FALSE) | (1 << ExpressionParser.NIL) | (1 << ExpressionParser.ID) | (1 << ExpressionParser.INT) | (1 << ExpressionParser.DOUBLE) | (1 << ExpressionParser.STRING))) != 0):  
 self.state = 247  
 self.arrayElement()  
  
  
 self.state = 254  
 self.\_errHandler.sync(self)  
 \_la = self.\_input.LA(1)  
  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
 **class** ArrayElementContext(ParserRuleContext):  
  
 **def** \_\_init\_\_(self, parser, parent:ParserRuleContext=**None**, invokingState:int=-1):  
 super().\_\_init\_\_(parent, invokingState)  
 self.parser = parser  
  
  
 **def** getRuleIndex(self):  
 **return** ExpressionParser.RULE\_arrayElement  
  
   
 **def** copyFrom(self, ctx:ParserRuleContext):  
 super().copyFrom(ctx)  
  
  
  
 **class** ArrayElementTypesContext(ArrayElementContext):  
  
 **def** \_\_init\_\_(self, parser, ctx:ParserRuleContext): *# actually a ExpressionParser.ArrayElementContext* super().\_\_init\_\_(parser)  
 self.copyFrom(ctx)  
  
 **def** atom(self):  
 **return** self.getTypedRuleContext(ExpressionParser.AtomContext,0)  
  
  
 **def** enterRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"enterArrayElementTypes"** ):  
 listener.enterArrayElementTypes(self)  
  
 **def** exitRule(self, listener:ParseTreeListener):  
 **if** hasattr( listener, **"exitArrayElementTypes"** ):  
 listener.exitArrayElementTypes(self)  
  
 **def** accept(self, visitor:ParseTreeVisitor):  
 **if** hasattr( visitor, **"visitArrayElementTypes"** ):  
 **return** visitor.visitArrayElementTypes(self)  
 **else**:  
 **return** visitor.visitChildren(self)  
  
  
  
 **def** arrayElement(self):  
  
 localctx = ExpressionParser.ArrayElementContext(self, self.\_ctx, self.state)  
 self.enterRule(localctx, 50, self.RULE\_arrayElement)  
 **try**:  
 localctx = ExpressionParser.ArrayElementTypesContext(self, localctx)  
 self.enterOuterAlt(localctx, 1)  
 self.state = 255  
 self.atom()  
 **except** RecognitionException **as** re:  
 localctx.exception = re  
 self.\_errHandler.reportError(self, re)  
 self.\_errHandler.recover(self, re)  
 **finally**:  
 self.exitRule()  
 **return** localctx  
  
  
  
 **def** sempred(self, localctx:RuleContext, ruleIndex:int, predIndex:int):  
 **if** self.\_predicates == **None**:  
 self.\_predicates = dict()  
 self.\_predicates[9] = self.expr\_sempred  
 pred = self.\_predicates.get(ruleIndex, **None**)  
 **if** pred **is None**:  
 **raise** Exception(**"No predicate with index:"** + str(ruleIndex))  
 **else**:  
 **return** pred(localctx, predIndex)  
  
 **def** expr\_sempred(self, localctx:ExprContext, predIndex:int):  
 **if** predIndex == 0:  
 **return** self.precpred(self.\_ctx, 10)  
   
  
 **if** predIndex == 1:  
 **return** self.precpred(self.\_ctx, 7)  
   
  
 **if** predIndex == 2:  
 **return** self.precpred(self.\_ctx, 6)  
   
  
 **if** predIndex == 3:  
 **return** self.precpred(self.\_ctx, 5)  
   
  
 **if** predIndex == 4:  
 **return** self.precpred(self.\_ctx, 4)  
   
  
 **if** predIndex == 5:  
 **return** self.precpred(self.\_ctx, 3)  
   
  
 **if** predIndex == 6:  
 **return** self.precpred(self.\_ctx, 2)

*# Generated from Expression.g4 by ANTLR 4.7.1  
# pylint: disable-all***from** antlr4 **import** \*  
**if** \_\_name\_\_ **is not None and "." in** \_\_name\_\_:  
 **from** .ExpressionParser **import** ExpressionParser  
**else**:  
 **from** ExpressionParser **import** ExpressionParser  
  
*# This class defines a complete generic visitor for a parse tree produced by ExpressionParser.***class** ExpressionVisitor(ParseTreeVisitor):  
  
 *# Visit a parse tree produced by ExpressionParser#parse.* **def** visitParse(self, ctx:ExpressionParser.ParseContext):  
 **return** self.visitChildren(ctx)  
  
 *# Visit a parse tree produced by ExpressionParser#block.* **def** visitBlock(self, ctx:ExpressionParser.BlockContext):  
 **return** self.visitChildren(ctx)  
  
 *# Visit a parse tree produced by ExpressionParser#assignmentExpr.* **def** visitAssignmentExpr(self, ctx:ExpressionParser.AssignmentExprContext):  
 **return** self.visitChildren(ctx)  
  
 *# Visit a parse tree produced by ExpressionParser#ifStatExpr.* **def** visitIfStatExpr(self, ctx:ExpressionParser.IfStatExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#whileStatExpr.* **def** visitWhileStatExpr(self, ctx:ExpressionParser.WhileStatExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#statementExpr.* **def** visitStatementExpr(self, ctx:ExpressionParser.StatementExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#otherExpr.* **def** visitOtherExpr(self, ctx:ExpressionParser.OtherExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#assignment.* **def** visitAssignment(self, ctx:ExpressionParser.AssignmentContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#ifStat.* **def** visitIfStat(self, ctx:ExpressionParser.IfStatContext):   
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#conditionBlock.* **def** visitConditionBlock(self, ctx:ExpressionParser.ConditionBlockContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#statBlock.* **def** visitStatBlock(self, ctx:ExpressionParser.StatBlockContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#whileStat.* **def** visitWhileStat(self, ctx:ExpressionParser.WhileStatContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#statement.* **def** visitStatement(self, ctx:ExpressionParser.StatementContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#notExpr.* **def** visitNotExpr(self, ctx:ExpressionParser.NotExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#unaryMinusExpr.* **def** visitUnaryMinusExpr(self, ctx:ExpressionParser.UnaryMinusExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#multiplicationExpr.* **def** visitMultiplicationExpr(self, ctx:ExpressionParser.MultiplicationExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#atomExpr.* **def** visitAtomExpr(self, ctx:ExpressionParser.AtomExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#orExpr.* **def** visitOrExpr(self, ctx:ExpressionParser.OrExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#additiveExpr.* **def** visitAdditiveExpr(self, ctx:ExpressionParser.AdditiveExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#powExpr.* **def** visitPowExpr(self, ctx:ExpressionParser.PowExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#relationalExpr.* **def** visitRelationalExpr(self, ctx:ExpressionParser.RelationalExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#equalityExpr.* **def** visitEqualityExpr(self, ctx:ExpressionParser.EqualityExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#functionExpr.* **def** visitFunctionExpr(self, ctx:ExpressionParser.FunctionExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#andExpr.* **def** visitAndExpr(self, ctx:ExpressionParser.AndExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#zeroParamterFunctions.* **def** visitZeroParamterFunctions(self, ctx:ExpressionParser.ZeroParamterFunctionsContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#singleParamterFunctions.* **def** visitSingleParamterFunctions(self, ctx:ExpressionParser.SingleParamterFunctionsContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#twoParamterFunctions.* **def** visitTwoParamterFunctions(self, ctx:ExpressionParser.TwoParamterFunctionsContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#threeParamterFunctions.* **def** visitThreeParamterFunctions(self, ctx:ExpressionParser.ThreeParamterFunctionsContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#fourParamterFunctions.* **def** visitFourParamterFunctions(self, ctx:ExpressionParser.FourParamterFunctionsContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#NParametersFunctions.* **def** visitNParametersFunctions(self, ctx:ExpressionParser.NParametersFunctionsContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#functionParam0.* **def** visitFunctionParam0(self, ctx:ExpressionParser.FunctionParam0Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#functionParam1.* **def** visitFunctionParam1(self, ctx:ExpressionParser.FunctionParam1Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#functionParam2.* **def** visitFunctionParam2(self, ctx:ExpressionParser.FunctionParam2Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#functionParam3.* **def** visitFunctionParam3(self, ctx:ExpressionParser.FunctionParam3Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#functionParam4.* **def** visitFunctionParam4(self, ctx:ExpressionParser.FunctionParam4Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#functionParamN.* **def** visitFunctionParamN(self, ctx:ExpressionParser.FunctionParamNContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#argumentsN.* **def** visitArgumentsN(self, ctx:ExpressionParser.ArgumentsNContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#arguments1.* **def** visitArguments1(self, ctx:ExpressionParser.Arguments1Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#arguments2.* **def** visitArguments2(self, ctx:ExpressionParser.Arguments2Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#arguments3.* **def** visitArguments3(self, ctx:ExpressionParser.Arguments3Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#arguments4.* **def** visitArguments4(self, ctx:ExpressionParser.Arguments4Context):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#parExpr.* **def** visitParExpr(self, ctx:ExpressionParser.ParExprContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#arrayAtom.* **def** visitArrayAtom(self, ctx:ExpressionParser.ArrayAtomContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#numberAtom.* **def** visitNumberAtom(self, ctx:ExpressionParser.NumberAtomContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#booleanAtom.* **def** visitBooleanAtom(self, ctx:ExpressionParser.BooleanAtomContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#idAtom.* **def** visitIdAtom(self, ctx:ExpressionParser.IdAtomContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#stringAtom.* **def** visitStringAtom(self, ctx:ExpressionParser.StringAtomContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#nilAtom.* **def** visitNilAtom(self, ctx:ExpressionParser.NilAtomContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#valueList.* **def** visitValueList(self, ctx:ExpressionParser.ValueListContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#arrayValues.* **def** visitArrayValues(self, ctx:ExpressionParser.ArrayValuesContext):  
 **return** self.visitChildren(ctx)  
  
  
 *# Visit a parse tree produced by ExpressionParser#arrayElementTypes.* **def** visitArrayElementTypes(self, ctx:ExpressionParser.ArrayElementTypesContext):  
 **return** self.visitChildren(ctx)  
  
  
  
**del** ExpressionParser

**import** enum  
  
  
**class** FunctionNames(enum.Enum):  
  
 *'''  
 // String function  
 '''* CONCAT = 1  
 LENGTH = 2  
 REPEAT = 3  
 INSTR = 4  
 LOCATE = 5  
 LOWER = 6  
 UPPER = 7  
 REGEXP\_EXTRACT = 8  
 REGEXP\_REPLACE = 9  
 REVERSE = 10  
 RPAD = 11  
 LPAD = 12  
 LTRIM = 13  
 RTRIM = 14  
 SPACE = 15  
 INITCAP = 16  
 SOUNDEX = 17  
 LEVENSHTEIN = 18  
 SUBSTR = 19  
 TRANSLATE = 20  
 TRIM = 21  
 REGEX\_MATCH = 22  
 ISBLANK = 23  
 ISNOTBLANK = 24  
 DECODE = 25  
 SUBSTRP = 26  
 MID = 27  
 FIND\_IN\_SET = 28  
 SPLIT = 29  
 TO\_STRING = 30  
 TO\_NUMBER = 31  
  
 **'''  
 Arithmetic  
 '''** LOG = 30  
 LOG2 = 31  
 LOG10 = 32  
 SQRT = 33  
 SIN = 34  
 COS = 35  
 TAN = 36  
 ASIN = 37  
 ACOS = 38  
 ATAN = 39  
 FLOOR = 40  
 CEIL = 41  
 RAND = 42  
 EXP = 43  
 GREATEST = 44  
 LEAST = 45  
 ROUND = 46  
 ROUNDP = 47  
 FACT = 48  
 ABS = 49  
  
 **'''  
 Date function  
 '''** TO\_DATE = 101  
 YEAR = 102  
 MONTH = 103  
 DAY = 104  
 ADD\_MONTHS = 105  
 DATEDIFF = 106  
 NOW = 107  
 TODAY = 109  
 DATEADD = 110  
 TIME = 111  
 WEEKNUM = 112  
 HOUR = 113  
 MINUTE = 114  
 SECOND = 115  
 DAYNAME = 116  
 MONTHNAME = 117  
 DATETIME = 118  
  
 **'''  
 Lookup function  
 '''** Functions = 200  
 BOOL\_IN\_LOOKUP = 201  
 LOOK\_UP\_CHECK = 202  
 NVL = 203  
 LOOKUP = 204  
 **'''  
 Aggregate function  
 '''** XAVG = 301  
 XCOUNT = 302  
 XMAX = 303  
 XMIN = 304  
 XSUM = 305  
 XUNIQUECOUNT = 306  
 XFREQCOUNT = 307  
 XFREQPERC = 308  
 XPERC = 309  
 AGGREGATERANGECHECK = 310  
 LOOKUPMAP = 311  
 INVERSIONCHECK = 312

*'''  
This is the function driver class for all the function registry  
'''***from** expression.function\_constants **import** FunctionNames  
**from** expression.types\_of\_function.string\_functions **import** StringFunction  
**import** random  
**from** expression.types\_of\_function.date\_function **import** DateTimeFunction  
**from** expression.types\_of\_function.lookup\_functions **import** LookupFunction  
**from** expression.types\_of\_function.aggregate\_functions **import** AggregateFunction  
**from** expression.types\_of\_function.arithmetic\_functions **import** ArithmeticFunction  
*# from function\_constants import FunctionNames  
# from types\_of\_function.string\_functions import StringFunction  
# import random  
# from types\_of\_function.date\_function import DateTimeFunction***class** FunctionDriver(object):  
 *"""  
 Function registry of All function added in the expression grammer.  
 """* function\_dictorany = {  
 *# String function* FunctionNames.SUBSTR: StringFunction.substring\_of\_string,  
 FunctionNames.RAND: StringFunction.random\_number,  
 FunctionNames.LENGTH: StringFunction.length,  
 FunctionNames.LOWER: StringFunction.lower,  
 FunctionNames.UPPER: StringFunction.upper,  
 FunctionNames.REVERSE: StringFunction.reverse,  
 FunctionNames.TRIM: StringFunction.trim,  
 FunctionNames.LTRIM: StringFunction.left\_trim,  
 FunctionNames.RTRIM: StringFunction.right\_trim,  
 FunctionNames.INITCAP: StringFunction.init\_cap,  
 FunctionNames.ISBLANK: StringFunction.is\_blank,  
 FunctionNames.ISNOTBLANK: StringFunction.is\_not\_blank,  
 FunctionNames.SPACE: StringFunction.space,  
 FunctionNames.CONCAT: StringFunction.concat,  
 FunctionNames.REPEAT: StringFunction.repeat,  
 FunctionNames.INSTR: StringFunction.instr,  
 FunctionNames.REGEX\_MATCH: StringFunction.regex\_match,  
 FunctionNames.REGEXP\_EXTRACT: StringFunction.regex\_extract,  
 FunctionNames.REGEXP\_REPLACE: StringFunction.regex\_replace,  
 FunctionNames.FIND\_IN\_SET: StringFunction.find\_in\_set,  
 FunctionNames.NVL: StringFunction.nvl,  
 FunctionNames.DECODE: StringFunction.decode,  
 FunctionNames.TO\_STRING: StringFunction.to\_string,  
 FunctionNames.TO\_NUMBER: StringFunction.to\_number,  
  
 *# Date functions* FunctionNames.DATEDIFF: DateTimeFunction.date\_diff,  
 FunctionNames.ADD\_MONTHS: DateTimeFunction.add\_months,  
 FunctionNames.TO\_DATE: DateTimeFunction.to\_date,  
 FunctionNames.YEAR: DateTimeFunction.year,  
 FunctionNames.NOW: DateTimeFunction.now,  
 FunctionNames.TODAY: DateTimeFunction.today,  
 FunctionNames.DATEADD: DateTimeFunction.dateadd,  
 FunctionNames.DAY: DateTimeFunction.day,  
 FunctionNames.MONTH: DateTimeFunction.month,  
 FunctionNames.TIME: DateTimeFunction.get\_time,  
 FunctionNames.DATETIME: DateTimeFunction.get\_datetime,  
 FunctionNames.WEEKNUM: DateTimeFunction.weeknum,  
 FunctionNames.MONTHNAME: DateTimeFunction.monthname,  
 FunctionNames.HOUR: DateTimeFunction.get\_hour,  
 FunctionNames.MINUTE: DateTimeFunction.get\_minute,  
 FunctionNames.SECOND: DateTimeFunction.get\_second,  
 FunctionNames.DAYNAME: DateTimeFunction.day\_name,  
  
 *#Lookup functions* FunctionNames.LOOKUPMAP: LookupFunction.look\_up\_map,  
 FunctionNames.LOOK\_UP\_CHECK: LookupFunction.look\_up\_check,   
  
 *#Aggregate functions* FunctionNames.XAVG: AggregateFunction.xavg,  
 FunctionNames.XCOUNT: AggregateFunction.xcount,  
 FunctionNames.XMIN: AggregateFunction.xmin,  
 FunctionNames.XMAX: AggregateFunction.xmax,  
 FunctionNames.XSUM: AggregateFunction.xsum,  
  
 *#Arithmetic functions* FunctionNames.ROUND: ArithmeticFunction.round,  
 FunctionNames.GREATEST: ArithmeticFunction.greatest,  
 FunctionNames.LEAST: ArithmeticFunction.least,  
 FunctionNames.ABS: ArithmeticFunction.abs  
  
  
 }  
  
 @classmethod  
 **def** execute\_function(cls, name, params):  
 *"""  
 This method is called whenever every function is used in the expression* **:param** *name:* **:param** *params:* **:return***:  
 """* func = FunctionDriver.function\_dictorany.get(FunctionNames[name])  
 value = func(params)  
 **return** value

*# import sys  
"""  
This module is used for runner the Expression on data\_frame  
"""***import** pandas **as** pd  
*#import MySQLdb***import** pymysql  
**from** mashupbase **import** DataMashUpBase  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** MysqlLoder(DataMashUpBase):  
 *"""  
 This class is used for running expression on requested dataframe.  
 """* **def** \_\_init\_\_(self, host, user, password, database, port):  
 self.host = host  
 self.user = user  
 self.password = password  
 self.database = database  
 self.port = port  
  
  
 **def** create\_connection(self):  
 **return** pymysql.connect(host = self.host, user = self.user, passwd = self.password, db = self.database, port = self.port)  
  
 **def** close\_connection(self, connectionMysql):  
 connectionMysql.close()  
  
  
 **def** read\_all\_table\_names(self):  
 connectionMysql = self.create\_connection()  
 cursor = connectionMysql.cursor()  
 cursor.execute(**"Show tables"**)  
 table\_list = pd.DataFrame(list(cursor.fetchall()), columns=list(map(**lambda** x:x[0], cursor.description)))  
 self.close\_connection(connectionMysql)  
 **return** table\_list  
  
 **def** read\_table(self, table\_name):  
 connectionMysql = self.create\_connection()  
 cursor = connectionMysql.cursor()  
 query = **"select \* from "**+table\_name  
 cursor.execute(query)  
 table\_df = pd.DataFrame(list(cursor.fetchall()), columns=list(map(**lambda** x:x[0], cursor.description)))  
 self.close\_connection(connectionMysql)  
 **return** table\_df;  
  
 **def** execute\_query(self, query):  
 connectionMysql = self.create\_connection()  
 cursor = connectionMysql.cursor()  
 cursor.execute(query)  
 table\_df = pd.DataFrame(list(cursor.fetchall()), columns=list(map(**lambda** x:x[0], cursor.description)))  
 self.close\_connection(connectionMysql)  
 **return** table\_df;

*# import sys  
"""  
This module is used for runner the Expression on data\_frame  
"""***from** antlr4.tree.Trees **import** Trees  
**from** antlr4 **import** CommonTokenStream, InputStream  
**from** expression.ExpressionLexer **import** ExpressionLexer  
**from** expression.ExpressionParser **import** ExpressionParser  
**from** expression.eval\_visitor **import** EvalVisitor  
**from** mashupbase **import** DataMashUpBase  
**import** constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
  
**class** ExpressionRunner(DataMashUpBase):  
 *"""  
 This class is used for running expression on requested dataframe.  
 """* **def** \_\_init\_\_(self, expression\_step):  
  
 self.expression\_string = expression\_step[constants.EXPRESSION]  
 self.new\_column\_name = expression\_step[constants.NEW\_COLUMN\_NAME].lower()  
  
 **def** function\_for\_data\_frame(self, value):  
 *"""  
 This method is used to apply expression on dataframe of each record* **:param** *value:* **:return***:  
 """* value = {k.lower(): v **for** k, v **in** value.items()}  
 **return** self.run\_expression(value)  
  
 **def** run\_expression(self, register\_data\_set):  
 *"""  
 This method is used to run the expression* **:param** *register\_data\_set:* **:return***:  
 """* **try**:  
 lexer = ExpressionLexer(InputStream(self.expression\_string))  
 parser = ExpressionParser(CommonTokenStream(lexer))  
 eval\_visit = EvalVisitor(register\_data\_set.copy())  
 tree = parser.parse()  
 result = eval\_visit.visit(tree)  
 *#print(Trees.toStringTree(tree, None, parser))* **return** result  
 **except** KeyError **as** ex:  
 **raise** Exception(ex.args[0] + **" function not available"**)  
 **except** Exception **as** ex1:  
 *#raise Exception(str(ex1))* **raise** Exception(**"Please perform operations on columns with same datatype."**)  
  
 **def** apply\_expression\_data\_frame(self, data\_frame):  
 *"""  
 This method will apply a function on data\_frame* **:param** *data\_frame:* **:return***:  
 """* **if** data\_frame **is None**:  
 **raise** Exception(**'dataset is empty'**)  
 *#return data\_frame.apply(self.function\_for\_data\_frame, axis=1)* **return** self.run\_expression(data\_frame)  
  
 **def** add\_new\_column(self, data\_frame):  
 *"""  
 This method is used to add new column on dataframe.* **:param** *new\_column\_name:* **:param** *data\_frame:* **:return***:  
 """* data\_frame[self.new\_column\_name] = self.apply\_expression\_data\_frame(data\_frame)  
 **return** data\_frame  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method will generate result of expression* **:param** *data\_frame:* **:return***:  
 """* **return** self.add\_new\_column(data\_frame)  
  
 @classmethod  
 **def** validate\_expression(cls, step\_json):  
 *"""  
 This method will use to validate expression* **:return***:  
 """* **try**:  
 runner = ExpressionRunner(step\_json[constants.APPLY\_SET])  
 json\_value = DataRegistry.sample\_record\_from\_data\_registry(  
 step\_json[constants.DATA\_SOURCE\_ID], step\_json[constants.LENS\_ID], **"WORKING"**, 1)  
 runner.run\_expression(json\_value)  
 json = {**"success"**: **"Validate successfully"**}  
 **return** json  
 **except** Exception **as** e:  
 **raise** Exception(str(e))

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** expression.MysqlLoader **import** MysqlLoder  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** pickle  
**from** datetime **import** datetime  
*#import MySQLdb***import** pymysql  
**import** pandas  
**import** os  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** requests  
  
  
**class** TestExpressionApi(unittest.TestCase):  
  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
  
 def** test\_mysql\_cashing(self):  
 mysqlloader = MysqlLoder(**'localhost'**,**'root'**,**'Newuser123'**,**'mercer\_pe'**)  
 all\_tables = mysqlloader.read\_all\_table\_names()[**'Tables\_in\_mercer\_pe'**]  
 **for** table **in** all\_tables:  
 **if**(table.islower):  
 **if**(table == **"validorg"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc\_au\_nz"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"country"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **else**:  
 print(**"table is upper : "**+table)  
  
 print(**"-----------"**)  
 print(**"length is : "**)  
 print(DataRegistry.pandas\_df\_map.keys())  
  
  
 *#md = DataRegistry.get\_data\_set\_from\_dict("ops\_career\_stream\_bonact\_perc\_au\_nz","1000","ORIGINAL")  
 #md.to\_csv("D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/Lookup/ops\_career\_stream\_bonact\_perc\_au\_nz.csv")* **def** test\_suite\_for\_mercer\_expressions(self):  
  
 print(**"-----------"**)  
 print(**"length is : "**)  
 print(DataRegistry.pandas\_df\_map.keys())  
  
 *#load mysql tables  
 #self.test\_mysql\_cashing()  
  
 #Read and test data file as a frame:* test\_df = pd.read\_csv(**"D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/Lookup/mts\_data\_lookup.txt"**, header=0, nrows=10000, delimiter=**"\t"**)  
 print(test\_df.head())  
   
 *#Read the expressions file:  
 #with open('D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/Lookup/mts\_expressions\_lookup.txt') as f:  
 # lines = f.readlines()  
 #lines = [x.strip() for x in lines]* expressions = pd.read\_excel(**"D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/Lookup/mts\_expressions\_lookup.xlsx"**,sheet\_name= **"mts\_expressions\_lookup"**)  
 *#expressions = expressions[expressions.active\_flag == 1]* lines = expressions.Expression.tolist()  
   
 counter = 0  
 **for** line **in** expressions.Expression.tolist():  
 print(**"expression : "**)  
 print(line)  
 line = line.replace(**" AND "**, **" && "**)  
 line = line.replace(**" and "**, **" && "**)  
 line = line.replace(**" OR "**, **" || "**)  
 line = line.replace(**" or "**, **" || "**)  
 line = line.replace(**"contextData."**, **""**)  
 *#opdf = self.call\_local\_run\_expression(data, line, str(counter))* print(self.call\_local\_run\_expression(test\_df, line, str(counter)).head(25).to\_string())  
 counter += 1  
   
 print(test\_df.head())  
 test\_df.to\_csv(**"D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/Lookup/result.csv"**)  
 *#self.execute\_rest\_api()* **def** call\_local\_run\_expression(self, data\_frame, expression, expression\_serial):  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: expression,  
 **"new\_column\_name"**: expression\_serial}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 **return** result  
  
  
  
 **def** execute\_rest\_api(self):  
 url = **'http://pe-decouple-1973837360.us-east-1.elb.amazonaws.com:8093/jsonReadCheck'** data = **'{ "data\_source\_id": "df\_28nov", "lens\_id": 1000, "steps": { "step\_name": "expression.run\_expression.ExpressionRunner", "new\_column\_name": "output1", "expression": "3\*2;" } }'** response = requests.post(url, data=data,headers={**"Content-Type"**: **"application/json"**})  
 print(response)  
 *#sid=response.json()['platform']['login']['sessionId'] //to extract the detail from response* print(response.text)

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** expression.MysqlLoader **import** MysqlLoder  
**from** dataframe\_reader.data\_wrangler **import** DataWranglerService  
  
  
**class** TestMercerExpressionApi(unittest.TestCase):  
  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
   
 def** test\_mysql\_cashing(self):  
 mysqlloader = MysqlLoder(**'localhost'**,**'root'**,**'Newuser123'**,**'mercer\_pe'**)  
 all\_tables = mysqlloader.read\_all\_table\_names()[**'Tables\_in\_mercer\_pe'**]  
 **for** table **in** all\_tables:  
 **if**(table.islower):  
 **if**(table == **"validorg"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc\_au\_nz"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"country"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"currency"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"validlti\_ref"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **else**:  
 print(**"table is upper : "**+table)  
  
 print(**"\*\*\*\*\*Prints\*\*\*\*\*\*"**)  
 print(**"ops\_career\_stream\_bonact\_perc"**)  
 md = DataRegistry.get\_data\_set\_from\_dict(**"ops\_career\_stream\_bonact\_perc"**,**"1000"**,**"ORIGINAL"**)  
 print(md.dtypes)  
 print(md.head())  
   
 print(**"currency"**)  
 md1 = DataRegistry.get\_data\_set\_from\_dict(**"currency"**,**"1000"**,**"ORIGINAL"**)  
 print(md1.dtypes)  
 print(md1.head())  
  
  
 **"""  
 def test\_data\_wrangler(self):  
  
 self.test\_mysql\_cashing()  
   
 step = {  
 "run\_mode":"pandas\_series",  
 "operation\_mode":"select",  
 "query":"",  
 "datasource\_id":["currency","ops\_career\_stream\_bonact\_perc"],  
 "where\_condition":"",  
 "group\_by\_columns":"",  
 "aggregations":"",  
 "select\_columns":['CTX\_CTRY\_CODE\_left', 'CURR\_CODE\_left', 'EXCHANGE\_RATE\_left',  
 'ERROR\_MIN\_right', 'ALERT\_MIN\_right', 'ALERT\_MAX\_right',  
 'ERROR\_MAX\_right'],  
 "join\_condition" : { "currency" :["CTX\_CTRY\_CODE"], "ops\_career\_stream\_bonact\_perc" : ["CTX\_CTRY\_CODE"] },  
 "alias\_left" : "\_left",  
 "alias\_right" : "\_right"  
 }  
  
 dataWraglerService = DataWranglerService(step)  
 print(dataWraglerService.registerLookupFrame())  
 """**

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** pandas **as** pd  
**from** expression.run\_expression **import** ExpressionRunner  
  
  
**class** TestMercerExpressionApi(unittest.TestCase):  
 *"""  
 Date functions  
 """* **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
 def** test\_run\_today(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [1, 2, 3, 4]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"TODAY();"**,  
 **"new\_column\_name"**: **"today\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 *# print("End===>", result.head())* self.assertEqual(1, 1)  
  
 **def** test\_run\_dateadd(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"DATEADD(col1, 2);"**,  
 **"new\_column\_name"**: **"dateadd\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_day(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"DAY(col1);"**,  
 **"new\_column\_name"**: **"day\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_month(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"MONTH(col1);"**,  
 **"new\_column\_name"**: **"month\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_year(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"YEAR(col1);"**,  
 **"new\_column\_name"**: **"year\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_add\_months(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"ADD\_MONTHS(col1, 2);"**,  
 **"new\_column\_name"**: **"ADD\_MONTHS\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_time(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"TIME(col1);"**,  
 **"new\_column\_name"**: **"time\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_weeknum(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"WEEKNUM(col1);"**,  
 **"new\_column\_name"**: **"weeknum\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_hour(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"HOUR(col1);"**,  
 **"new\_column\_name"**: **"hour\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_minute(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"MINUTE(col1);"**,  
 **"new\_column\_name"**: **"minute\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_second(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"SECOND(col1);"**,  
 **"new\_column\_name"**: **"second\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_dayname(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"DAYNAME(col1);"**,  
 **"new\_column\_name"**: **"dayname\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_monthname(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"MONTHNAME(col1);"**,  
 **"new\_column\_name"**: **"monthname\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)  
  
 **def** test\_run\_datetime(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [**'2019/01/01'**, **'2019/01/02'**]})  
 *# print("Start===>", data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"DATETIME(col1);"**,  
 **"new\_column\_name"**: **"datetime\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(**"Start===>"**, result.head())  
 self.assertEqual(1, 1)

*"""  
This module is used for testing expression builder functionalities  
"""***import** unittest  
**import** pandas **as** pd  
**from** expression.run\_expression **import** ExpressionRunner  
  
  
**class** TestExpressionBuilder(unittest.TestCase):  
 *"""  
 This class is used of test expression with different scenario  
 """* **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
 def** test\_addition(self):  
 *"""  
 Test case for expression with addition* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"2+3+1111.1121212;"**,  
 **"new\_column\_name"**: **"aavesh"**}  
 runner = ExpressionRunner(step)  
 self.assertEqual(1116.1121212, runner.run\_expression({}))  
  
  
 **def** test\_substract(self):  
 *"""  
 test case for expression with minus operator* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"2+3-1111.1121212;"**,  
 **"new\_column\_name"**: **"aavesh"**}  
 runner = ExpressionRunner(step)  
 self.assertEqual(-1106.1121212, runner.run\_expression({}))  
  
 **def** test\_division(self):  
 *"""  
 test case for expression with divide operator* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"5.0555/2;"**,  
 **"new\_column\_name"**: **"aavesh"**}  
 runner = ExpressionRunner(step)  
 self.assertEqual(2.52775, runner.run\_expression({}))  
  
 **def** test\_multiplication(self):  
 *"""  
 Test case for expression with multiplication* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"5.0555\*2;"**,  
 **"new\_column\_name"**: **"aavesh"**}  
 runner = ExpressionRunner(step)  
 self.assertEqual(10.111, runner.run\_expression({}))  
  
 **def** test\_sqrt(self):  
 *"""  
 test case for expression with find sqrt* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"2^2;"**,  
 **"new\_column\_name"**: **"aavesh"**}  
 runner = ExpressionRunner(step)  
 self.assertEqual(4, runner.run\_expression({}))  
  
 **def** test\_add\_variable\_with\_string\_function(self):  
 *"""  
 test add new variable using date function* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"INSTR(SUBSTR(TRIM(REVERSE(CONCAT(INITCAP(FIELD6),LOWER(FIELD6)))),1,1),\"a\");"**,  
 **"new\_column\_name"**: **"add\_month\_column"**}  
 data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, header=0, nrows=10, delimiter=**"\t"**)  
 runner = ExpressionRunner(step)  
 column\_name = **"add\_month\_column"** data\_frame = runner.add\_new\_column(data\_frame)  
 **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_add\_variable\_with\_instr\_function(self):  
 *"""  
 test add new variable using date function* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"INSTR(FIELD6,\"a\");"**,  
 **"new\_column\_name"**: **"add\_month\_column"**}  
 data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, header=0, nrows=10, delimiter=**"\t"**)  
 runner = ExpressionRunner(step)  
 column\_name = **"add\_month\_column"** data\_frame = runner.add\_new\_column(data\_frame)  
 **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_add\_variable\_with\_string\_regex\_function(self):  
 *"""  
 test add new variable using date function* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"REGEX\_MATCH(INITCAP(FIELD6),\"Akshay\");"**,  
 **"new\_column\_name"**: **"add\_month\_column"**}  
 data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, header=0, nrows=10, delimiter=**"\t"**)  
 runner = ExpressionRunner(step)  
 column\_name = **"add\_month\_column"** data\_frame = runner.add\_new\_column(data\_frame)  
 **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_add\_variable\_with\_string\_regex\_extract(self):  
 *"""  
 test add new variable using date function* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"REGEXP\_EXTRACT(INITCAP(FIELD6),\"(Akshay)\")"**,  
 **"new\_column\_name"**: **"add\_month\_column"**}  
 data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, header=0, nrows=10, delimiter=**"\t"**)  
 runner = ExpressionRunner(step)  
 column\_name = **"add\_month\_column"** data\_frame = runner.add\_new\_column(data\_frame)  
 **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_add\_variable\_with\_string\_regex\_replace(self):  
 *"""  
 test add new variable using date function* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"REGEXP\_REPLACE(INITCAP(FIELD6),\"(Akshay)\",\"abcd\")"**,  
 **"new\_column\_name"**: **"add\_month\_column"**}  
 data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, header=0, nrows=10, delimiter=**"\t"**)  
 runner = ExpressionRunner(step)  
 column\_name = **"add\_month\_column"** data\_frame = runner.add\_new\_column(data\_frame)  
 **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_add\_variable\_with\_date\_function(self):  
 *"""  
 test add new variable using date function* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"DATEDIFF(ADD\_MONTHS(TO\_DATE(FIELD0,\"\"),field2),FIELD0)"**,  
 **"new\_column\_name"**: **"add\_month\_column"**}  
 data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, header=0, nrows=10, delimiter=**"\t"**)  
 runner = ExpressionRunner(step)  
 column\_name = **"add\_month\_column"** data\_frame = runner.add\_new\_column(data\_frame)  
 **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)   
  
   
 **def** test\_conditional\_operations(self):  
 *"""  
 test add new variable using date function* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"(LENGTH(batting\_team) > 15) AND (INITCAP(bowling\_team) == \"Royal Challengers Bangalore\")"**,  
 **"new\_column\_name"**: **"find\_in\_set\_status"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(round(result[**'find\_in\_set\_status'**][0]),**True**)   
  
  
 **def** test\_if\_else\_decode(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"DECODE(total\_runs == 0 OR UPPER(batsman) == \"SC GANGULY\" , DECODE(total\_runs == 4,\"Boundary\",\"Duckout\"), \"Fifty\")"**,  
 **"new\_column\_name"**: **"find\_valid\_runner"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=10)  
 runner = ExpressionRunner(step)  
 column\_name = **"find\_valid\_runner"** data\_frame = runner.add\_new\_column(data\_frame)  
 *#print(data\_frame[['total\_runs','batsman','find\_valid\_runner']].head(10))* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_xavg(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"total\_runs > XAVG(match\_id,ball > 1,bowling\_team)"**,  
 **"new\_column\_name"**: **"find\_valid\_runner"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=20)  
 runner = ExpressionRunner(step)  
 column\_name = **"find\_valid\_runner"** data\_frame = runner.add\_new\_column(data\_frame)  
 *#print(data\_frame[['total\_runs','ball','bowling\_team','find\_valid\_runner']])* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
 **def** test\_ifelse(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"if total\_runs == 0 && ball > 1 { value = \"Duckout\";} else if total\_runs==0 && ball==1 { value = \"Golden Duck\";} else if total\_runs == 4 {value = \"Fifty\";} else { value = \"medium\";}"**,  
 **"new\_column\_name"**: **"output"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=10)  
 runner = ExpressionRunner(step)  
 column\_name = **"output"** data\_frame = runner.add\_new\_column(data\_frame)  
 print(data\_frame[[**'total\_runs'**,**'ball'**,**'output'**]])  
 **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
This module is used for writing test case of module function\_driver  
"""***import** unittest  
**from** expression.function\_driver **import** FunctionDriver  
**import** datetime  
  
  
**class** TestFunctionDriver(unittest.TestCase):  
 *"""  
 This class contains all the test method of function\_driver class  
 """* **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
 def** test\_substring\_function(self):  
 *"""  
 This method will check SUBSTR FUNCTION* **:return***:  
 """* params = [**'MosaicInsightHub'**, 3, 3]  
 value = FunctionDriver.execute\_function(**"SUBSTR"**, params)  
 **if** value == **'sai'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_rand\_function(self):  
 *"""  
 Test suite for rand function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"RAND"**, [])  
 **if** value **is None**:  
 self.assertEqual(1, 2)  
 **else**:  
 self.assertEqual(1, 1)  
  
 **def** test\_now\_function(self):  
 *"""  
 Test suite for NOW function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"NOW"**, [])  
 **if** value **is None**:  
 self.assertEqual(1, 2)  
 **else**:  
 self.assertEqual(1, 1)  
  
 **def** test\_length\_function(self):  
 *"""  
 Test suite for LENGTH FUNCTION* **:return***:  
 """* value = FunctionDriver.execute\_function(**"LENGTH"**, [**"MosaicInsightHub"**])  
 **if** value == 16:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_lower\_function(self):  
 *"""  
 test suite for lower function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"LOWER"**, [**"MosaicInsightHub"**])  
 **if** value == **"mosaicinsighthub"**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_upper\_function(self):  
 *"""  
 test suite for UPPER function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"UPPER"**, [**"MosaicInsightHub"**])  
 **if** value == **"MOSAICINSIGHTHUB"**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_reverse\_function(self):  
 *"""  
 test suite for REVERSE function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"REVERSE"**, [**"MosaicInsightHub"**])  
 **if** value == **"buHthgisnIciasoM"**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_trim\_function(self):  
 *"""  
 test suite for lower function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"TRIM"**, [**" MosaicInsightHub "**])  
 **if** value == **'MosaicInsightHub'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_left\_trim\_function(self):  
 *"""  
 test suite for lower function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"LTRIM"**, [**" MosaicInsightHub"**])  
 **if** value == **'MosaicInsightHub'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_right\_trim\_function(self):  
 *"""  
 test suite for lower function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"RTRIM"**, [**"MosaicInsightHub "**])  
 **if** value == **'MosaicInsightHub'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_is\_blank\_function(self):  
 *"""  
 This check isblank function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"ISBLANK"**, [**"MosaicInsightHub"**])  
 **if** value **is False**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_init\_cap\_function(self):  
 *"""  
 Test suite for init\_cap function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"INITCAP"**, [**"mosaicInsightHub"**])  
 **if** value == **'MosaicInsightHub'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_is\_not\_blank\_function(self):  
 *"""  
 Test suite for is\_not\_blank function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"ISNOTBLANK"**, [**"MosaicInsightHub"**])  
 **if** value **is True**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_adding\_space\_function(self):  
 *"""  
 Test suite for adding space in string* **:return***:  
 """* value = FunctionDriver.execute\_function(**"SPACE"**, [5])  
 **if** len(value) **is** 5:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_to\_date\_function(self):  
 *"""  
 Test suite for to\_date function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"TO\_DATE"**, [**'2015-10-15'**, **''**])  
 **if** isinstance(value, datetime.date):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_add\_months\_function(self):  
 *"""  
 Test suite for add\_month function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"ADD\_MONTHS"**, [**'2015-10-15'**, 3])  
 **if** value == datetime.datetime(2016, 1, 15):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_year\_function(self):  
 *"""  
 test suite for find year from date* **:return***:  
 """* value = FunctionDriver.execute\_function(**"YEAR"**, [**'2015-10-15'**])  
 **if** value == 2015:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_date\_diff\_function(self):  
 *"""  
 test suite for date\_diff function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"DATEDIFF"**, [**'2016-10-16'**, **'2016-11-16'**])  
 **if** value == 31:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_regex\_extract\_function(self):  
 *"""  
 test suite for test regex\_extract function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"REGEXP\_EXTRACT"**, [**'foothebar'**, **'foo(.\*?)(bar)'**, 1])  
 **if** value == **'the'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 value = FunctionDriver.execute\_function(**"REGEXP\_EXTRACT"**, [**'foothebar'**, **'foo(.\*?)(bar)'**, 5])  
 **import** math  
 **if** math.isnan(value):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_regex\_replace\_function(self):  
 *"""  
 test suite for test regex\_extract function* **:return***:  
 """* value = FunctionDriver.execute\_function(**"REGEXP\_REPLACE"**, [**'foothebar'**, **'foo(.\*?)(bar)'**, **"aa"**])  
 **if** value == **'aa'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** expression.MysqlLoader **import** MysqlLoder  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** pickle  
**from** datetime **import** datetime  
**import** MySQLdb  
**import** pandas  
*#import os***from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** requests  
**from** multiprocessing **import** Pool  
  
**class** GenericTestSuite(unittest.TestCase):  
  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
   
 def** test\_mysql\_cashing(self):  
   
 mysqlloader = MysqlLoder(**'localhost'**,**'root'**,**'Newuser123'**,**'mercer\_pe'**,3306)  
 all\_tables = mysqlloader.read\_all\_table\_names()[**'Tables\_in\_mercer\_pe'**]  
 **for** table **in** all\_tables:  
 **if**(table.islower):  
 **if**(table == **"validorg"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc\_au\_nz"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"country"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"currency"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"validlti\_ref"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_super\_co\_perc\_au\_nz"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_spec\_flags"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"validlti\_ref"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))   
 **else**:  
 print(**"table is upper : "**+table)  
  
 md = DataRegistry.get\_data\_set\_from\_dict(**"validorg"**,**"1000"**,**"ORIGINAL"**)  
 *#print(md[["LTIMAXAWRD","CTX\_GRPCODE","LTIPLNTYP"]].head(22))  
 #md.to\_csv("validorg.csv")* **def** test\_suite\_for\_mercer\_expressions(self):  
   
 *#GenericTestSuite.test\_mysql\_cashing(self)  
  
  
 #DATA LOAD* sheet\_name = **"Complex\_2"** expdata = pd.read\_excel(**"D:/Mercer/Mercer\_expression\_testing/generic\_test\_suite/input/rushi\_test\_suite.xlsx"**,  
 sheet\_name= sheet\_name)  
 expdata = expdata.reset\_index()  
 print(**"reading of input data has benn completed ... "**)  
 print(**"\*\*\*\*\*\*\*\*Input Data\*\*\*\*\*\*\*\*\*"**)  
 print(expdata[[**"context\_data\_grpcode"**,**"ltiplan01"**]].head())  
   
   
 datacolumns = [x **for** x **in** expdata.columns **if** x != **"Expression"**]  
 data = expdata[datacolumns]  
 data.columns = [ x.lower() **for** x **in** data.columns]  
 expressions = expdata[expdata.active\_flag == 1][**"Expression"**]  
   
 print(expressions)  
  
 expressions = [x **for** x **in** expressions **if** str(x) != **"nan"**]  
   
   
 counter = 0  
 dftemp = data  
 **for** line **in** expressions:  
 line = line.replace(**" AND "**, **" && "**)  
 line = line.replace(**" and "**, **" && "**)  
 line = line.replace(**" OR "**, **" || "**)  
 line = line.replace(**" or "**, **" || "**)  
 line = line.replace(**"contextData."**, **"contextData\_"**)  
 dftemp[**'expr'**+str(counter)] = self.call\_local\_run\_expression(data, line, line)[line]  
 dftemp = dftemp.drop(axis = 1, columns = [line])  
 counter += 1  
   
 print(**"\*\*\*\*\*\*Output\*\*\*\*\*\*\*"**)  
   
 print(dftemp.columns)   
 print(dftemp.head(20))  
 *#print(dftemp[["context\_data\_grpcode","ltiplan01","s1","s2","s3","s4","s5","ltimaxawrd",  
 # "ltiplan06","expr0"]].head(40).to\_string())  
 #print(dftemp.to\_json(orient = "records"))  
 #dftemp.to\_csv("D:/Mercer/Mercer\_expression\_testing/generic\_test\_suite/Output/"+sheet\_name+"\_"+str(data.shape[0])+".csv")* **def** call\_local\_run\_expression(self, data\_frame, expression, expression\_serial):  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: expression,  
 **"new\_column\_name"**: expression\_serial}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 **return** result

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** expression.MysqlLoader **import** MysqlLoder  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** pickle  
**from** datetime **import** datetime  
**import** MySQLdb  
**import** pandas  
**import** os  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** requests  
  
  
**class** TestExpressionApi(unittest.TestCase):  
  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
  
 def** test\_mysql\_cashing(self):  
 mysqlloader = MysqlLoder(**'localhost'**,**'root'**,**'Newuser123'**,**'mercer\_pe'**)  
 all\_tables = mysqlloader.read\_all\_table\_names()[**'Tables\_in\_mercer\_pe'**]  
 **for** table **in** all\_tables:  
 **if**(table.islower):  
 **if**(table == **"validorg"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc\_au\_nz"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"country"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"currency"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"validlti\_ref"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **else**:  
 print(**"table is upper : "**+table)  
  
 md = DataRegistry.get\_data\_set\_from\_dict(**"validlti\_ref"**,**"1000"**,**"ORIGINAL"**)  
 *#print(md[md.CTX\_CTRY\_CODE=="ZA"][["MIN\_SALARY\_MTHLY","MIN\_SALARY"]].head(10))  
 #print(md[md.CTX\_CTRY\_CODE=="ZA"][["CURR\_CODE"]].head(10))  
 #print(md[md.CTX\_GRPCODE=='ZA'][["LTIMAXAWRD","CTX\_GRPCODE"]].head(22))* print(md[[**"LTIMAXAWRD"**,**"CTX\_GRPCODE"**,**"LTIPLNTYP"**]].head(22))  
   
   
 **def** test\_suite\_for\_mercer\_expressions(self):  
   
 *#Read and test data file as a frame:* data = pd.read\_excel(**"D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/IF\_ElSE/if\_else\_data.xlsx"**,sheet\_name= **"data"**)  
 *#data = data.head(1)* data.columns = [ x.lower() **for** x **in** data.columns]  
  
 expressions = pd.read\_excel(**"D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/IF\_ElSE/if\_else\_data.xlsx"**,sheet\_name= **"expressions"**)  
 expressions = expressions[expressions.active\_flag == 1]  
  
 counter = 0  
 **for** line **in** expressions.Expression.tolist():  
 print(**"expression : "**)  
 print(line)  
 line = line.replace(**" AND "**, **" && "**)  
 line = line.replace(**" and "**, **" && "**)  
 line = line.replace(**" OR "**, **" || "**)  
 line = line.replace(**" or "**, **" || "**)  
 line = line.replace(**"contextData."**, **"contextData\_"**)  
 *#opdf = self.call\_local\_run\_expression(data, line, str(counter))* print(self.call\_local\_run\_expression(data, line, str(counter)).head(25).to\_string())  
 counter += 1  
   
 *#print(opdf.dtypes)  
 #test\_df.to\_csv("D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/aggregate/result.csv")  
 #self.execute\_rest\_api()* **def** call\_local\_run\_expression(self, data\_frame, expression, expression\_serial):  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: expression,  
 **"new\_column\_name"**: expression\_serial}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 **return** result

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** expression.MysqlLoader **import** MysqlLoder  
  
  
**class** TestMercerExpressionApi(unittest.TestCase):  
  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
 def** test\_mysql\_cashing(self):  
 mysqlloader = MysqlLoder(**'localhost'**,**'root'**,**'Newuser123'**,**'mercer\_pe'**)  
 all\_tables = mysqlloader.read\_all\_table\_names()[**'Tables\_in\_mercer\_pe'**]  
 **for** table **in** all\_tables:  
 **if**(table.islower):  
 **if**(table == **"validorg"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc\_au\_nz"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"country"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"currency"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"validlti\_ref"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **else**:  
 print(**"table is upper : "**+table)  
  
 md = DataRegistry.get\_data\_set\_from\_dict(**"validorg"**,**"1000"**,**"ORIGINAL"**)  
 *#print(md.head())  
 #print(md[md.CTX\_CTRY\_CODE=="ZA"][["MIN\_SALARY\_MTHLY","MIN\_SALARY"]].head(10))  
 #print(md[md.CTX\_CTRY\_CODE=="ZA"][["CURR\_CODE"]].head(10))  
 #print(md[md.CTX\_GRPCODE=='ZA'][["LTIMAXAWRD","CTX\_GRPCODE"]].head(22))  
 #print(md[["LTIMAXAWRD","CTX\_GRPCODE","LTIPLNTYP"]].head(22))* **def** test\_lookup(self):  
 self.test\_mysql\_cashing()  
 **"""  
 This method is used for testing api for expression  
 :return:  
 """** data\_frame = pd.DataFrame({**'grpCode'**: [**'106792'**,**'106790'**,**'106791'**,**'106797'**],   
 **'ctryCode'**: [**'AE'**,**'GE'**,**'PE'**,**'DE'**],  
 **'cpyCode'**: [**'10009665'**,**'10009666'**,**'10009667'**,**'10009668'**]})  
  
 master\_df = DataRegistry.get\_data\_set\_from\_dict(**"validorg"**,**"1000"**,**"ORIGINAL"**)  
 DataRegistry.register\_data\_frame(**'m'**,1000,master\_df)   
 *#data\_frame = pd.read\_csv('../resources/SG\_HT\_Library\_1nov.csv', delimiter=',')* data\_frame = data\_frame.head(5)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **" grpCode = \"106792\";\  
 ctryCode = \"AE\"; \  
 cpyCode = \"10009664\";\  
 LOOK\_UP\_CHECK \  
 (\"validorg\", \  
\"CTX\_GRPCODE = grpCode AND \  
CTX\_CTRY\_CODE = ctryCode AND \  
CTX\_CPYCODE = cpyCode\",\"NUM\_EMP\")"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 *#print(result)  
 #self.assertEqual(result['string\_outcome'][0],"Kolkata Knight RidersKolkata Knight Riders")  
   
  
"""  
This module is used for testing expr with api  
"""***import** unittest  
**import** pandas **as** pd  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestExpressionApi(unittest.TestCase):  
 *"""  
 This class is used to run expression builder api  
 """* **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
 def** test\_run\_expx\_add\_column(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, delimiter=**'\t'**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"2.222;"**,  
 **"new\_column\_name"**: **"aavesh1"**}  
 runner = ExpressionRunner(step)  
 new\_data\_frame = runner.get\_result(data\_frame)  
 column\_name = **"aavesh1"  
 if** column\_name **in** new\_data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(5, 2)  
  
 **def** test\_validate\_expression(self):  
 *"""  
 This method will validate the expression used by use* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, delimiter=**'\t'**)  
 json = {**"ORIGINAL"**: data\_frame, **"WORKING"**: data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_1"**: json})  
  
 json\_request = {**"data\_source\_id"**: **"10"**, **"lens\_id"**: 1, **"pageNumber"**: 1, **"record\_limit"**: 1000,  
 **"steps"**: {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"new\_column\_name"**: **"aavesh"**,  
 **"type"**: **"Number"**, **"expression"**: **"3\*2;"**, **"displayDataHeader"**: **"aavesh"**, **"displayDataText"**: **"3\*2"**},  
 **"applyOn"**: **"ORIGINAL"**, **"total\_page\_count"**: **"0"**}  
  
 **if 'success' in** ExpressionRunner.validate\_expression(json\_request):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 json\_request = {**"data\_source\_id"**: **"10"**, **"lens\_id"**: 1, **"pageNumber"**: 1, **"record\_limit"**: 1000,  
 **"steps"**: {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"new\_column\_name"**: **"aavesh"**,  
 **"type"**: **"Number"**, **"expression"**: **"TO\_NUMBER(22);"**,  
 **"displayDataHeader"**: **"aavesh"**,**"displayDataText"**: **"3\*2"**},  
 **"applyOn"**: **"ORIGINAL"**, **"total\_page\_count"**: **"0"**}  
 **try**:  
 ExpressionRunner.validate\_expression(json\_request)  
 self.assertEqual(1, 2)  
 **except** Exception **as** e:  
 print(**"in exression"**, e)  
 self.assertEqual(1, 1)  
  
  
 *#Date functions testing* **def** test\_run\_date\_diff(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/retail\_order\_dataset.txt'**, delimiter=**'\t'**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"DATEDIFF(order\_delivered\_customer\_date,order\_estimated\_delivery\_date)"**,  
 **"new\_column\_name"**: **"date\_diff"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(round(result[**'date\_diff'**][0]),7)  
  
 **def** test\_run\_get\_year(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/retail\_order\_dataset.txt'**, delimiter=**'\t'**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"YEAR(order\_delivered\_customer\_date)"**,  
 **"new\_column\_name"**: **"year"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(round(result[**'year'**][0]),2017)  
  
 **def** test\_run\_add\_months(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/retail\_order\_dataset.txt'**, delimiter=**'\t'**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"ADD\_MONTHS(order\_delivered\_customer\_date,2)"**,  
 **"new\_column\_name"**: **"add\_month"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(str(result[**'add\_month'**][0]),**'2017-12-10 21:25:00'**)  
  
  
 *#String function testing* **def** test\_run\_substring(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"SUBSTR(batting\_team,1,4)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**'olk'**)  
  
 **def** test\_run\_length(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"LENGTH(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],21)  
  
  
 **def** test\_run\_upper(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"UPPER(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**"KOLKATA KNIGHT RIDERS"**)  
  
 **def** test\_run\_lower(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"LOWER(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**"kolkata knight riders"**)  
  
 **def** test\_run\_trim(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"TRIM(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**"Kolkata Knight Riders"**)  
  
 **def** test\_run\_lefttrim(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"LTRIM(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**"Kolkata Knight Riders"**)  
  
 **def** test\_run\_righttrim(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"RTRIM(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**"Kolkata Knight Riders"**)  
  
 **def** test\_run\_initcap(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"INITCAP(bowling\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**"Royal Challengers Bangalore"**)  
  
 **def** test\_run\_reverse(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"REVERSE(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 *#runner = ExpressionRunner(step)  
 #result = runner.get\_result(data\_frame)  
 #self.assertEqual(result['string\_outcome'][0],"srediR thginK atakloK")* **def** test\_run\_isblank(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"ISBLANK(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**False**)  
  
 **def** test\_run\_isnotblank(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"ISNOTBLANK(batting\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**True**)  
  
  
 **def** test\_run\_concat(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"CONCAT(batting\_team, bowling\_team)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**"Kolkata Knight Ridersroyal Challengers Bangalore"**)  
  
  
 **def** test\_run\_repeat(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, delimiter=**','**)  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"REPEAT(batting\_team,2)"**,  
 **"new\_column\_name"**: **"string\_outcome"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 self.assertEqual(result[**'string\_outcome'**][0],**"Kolkata Knight RidersKolkata Knight Riders"**)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** expression.MysqlLoader **import** MysqlLoder  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** pickle  
**from** datetime **import** datetime  
**import** MySQLdb  
**import** pandas  
*#import os***from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** requests  
**from** multiprocessing **import** Pool  
**import** time  
  
**class** TestExpressionApi(unittest.TestCase):  
  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
   
 def** test\_mysql\_cashing(self):  
 mysqlloader = MysqlLoder(**'localhost'**,**'root'**,**'Newuser123'**,**'mercer\_pe'**)  
 *#mysqlloader = MysqlLoder('localhost','audmosd','Mercer@123','mercer\_pe')* all\_tables = mysqlloader.read\_all\_table\_names()[**'Tables\_in\_mercer\_pe'**]  
 **for** table **in** all\_tables:  
 **if**(table.islower):  
 **if**(table == **"validorg"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc\_au\_nz"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"country"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_career\_stream\_bonact\_perc"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"currency"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"validlti\_ref"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **if**(table == **"ops\_audit\_ly\_poscode\_comp1"**):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print(**"registerd table : "**+str(table))  
  
 **else**:  
 print(**"table is upper : "**+table)  
  
 *#md = DataRegistry.get\_data\_set\_from\_dict("validlti\_ref","1000","ORIGINAL")  
 #print(md[["LTIMAXAWRD","CTX\_GRPCODE","LTIPLNTYP"]].head(22))* **"""  
 def test\_suite\_for\_mercer\_expressions(self):  
 c = datetime.now()  
 sheet\_name = "Basic"  
 #sheet\_name = "Basic\_Numeric"  
 #sheet\_name = "Basic\_String"  
 #sheet\_name = "Basic\_POS\_CODE"  
 #sheet\_name = "Basic\_Misc"  
 #sheet\_name = "Aggregate"  
   
 expdata = pd.read\_excel("D:/Mercer/Mercer\_expression\_testing/large\_file\_test\_data/expression\_data\_2.xlsx",  
 sheet\_name= sheet\_name)  
  
 #expdata = pd.read\_excel("/home/audmosd/mercer\_expression\_validator/PE\_Realignment\_flask2\_new/insighthub/DataMashUpApi/mashup\_apis/expression/expression\_data.xlsx",  
 # sheet\_name= sheet\_name)  
  
 #print("reading of input data has benn completed ... ")  
 #print(expdata.head())  
   
 a = datetime.now()  
 datacolumns = [x for x in expdata.columns if x != "Expression"]  
 data = expdata[datacolumns]  
 #data = pd.concat([data,data,data,data,data,data,data,data,data,data])  
 print("The data hasss been loaded : "+str(data.shape))  
 data.columns = [ x.lower() for x in data.columns]  
 expressions = expdata.Expression.tolist()  
 expressions = [x for x in expressions if str(x) != "nan"]  
 #print(expressions)  
 p1 = Pool(8)  
 #stopcounter = len(expressions)  
  
 dftemp = data  
 for line in expressions:  
 #print("expression : ")  
 #print(line)  
 line = line.replace(" AND ", " && ")  
 line = line.replace(" and ", " && ")  
 line = line.replace(" OR ", " || ")  
 line = line.replace(" or ", " || ")  
 line = line.replace("contextData.", "contextData\_")  
 dftemp[line] = self.call\_local\_run\_expression(data, line, line,p1)[line]  
   
 b = datetime.now()  
   
 print("\n Time - only execution : "+str((b-a))+"--->"+str((b-a).total\_seconds()))  
 print("\n Time - with data load but without write : "+str((b-c))+"--->"+str((b-c).total\_seconds()))  
 dftemp.to\_csv("D:/Mercer/Mercer\_expression\_testing/large\_file\_test\_data/"+sheet\_name+"\_"+str(data.shape[0])+".csv")  
 #dftemp.to\_csv("/home/audmosd/mercer\_expression\_validator/PE\_Realignment\_flask2\_new/insighthub/DataMashUpApi/mashup\_apis/expression/"+sheet\_name+"\_"+str(data.shape[0])+".csv")  
 d = datetime.now()  
 print("\n Time - execute and result write : "+str((d-a))+"--->"+str((d-a).total\_seconds()))  
 print("\n Time - with data load till result data write : "+str((d-c))+"--->"+str((d-c).total\_seconds()))  
  
 p1.close()  
 p1.join()   
 """  
  
  
 def** test\_suite\_for\_mercer\_expressions(self):  
  
  
 *#sheetList = ["Basic","Basic\_Numeric","Basic\_String","Basic\_POS\_CODE","Basic\_Misc","Aggregate"]* sheetList = [**"test"**]  
   
 **for** sheet **in** sheetList:  
  
 print(**"Sheet in Progress : "**+str(sheet))  
 c = datetime.now()  
 expdata = pd.read\_excel(**"D:/Mercer/Mercer\_expression\_testing/large\_file\_test\_data/expression\_data\_1.xlsx"**,  
 sheet\_name= sheet)  
  
 *#expdata = pd.read\_excel("C:/Users/10639502/Desktop/sample\_if\_else.xlsx",  
 # sheet\_name= sheet)  
  
 #expdata = pd.read\_csv("C:/Users/10639502/Desktop/lookup\_test.csv")  
  
  
 #expdata = pd.read\_excel("/home/audmosd/mercer\_expression\_validator/PE\_Realignment\_flask2\_new/benchmarking/input/expression\_data\_1.xlsx",  
 # sheet\_name= sheet)* a = datetime.now()  
 datacolumns = [x **for** x **in** expdata.columns **if** x != **"Expression"**]  
 data = expdata[datacolumns]  
 *#data = pd.concat([data,data,data,data,data,data,data,data,data,data])* print(**"The data has been loaded : "**+str(data.shape))  
 data.columns = [ x.lower() **for** x **in** data.columns]  
 expressions = expdata.Expression.tolist()  
 expressions = [x **for** x **in** expressions **if** str(x) != **"nan"**]  
 p1 = Pool(8)  
   
 dftemp = data  
 **for** line **in** expressions:  
 line = line.replace(**" AND "**, **" && "**)  
 line = line.replace(**" and "**, **" && "**)  
 line = line.replace(**" OR "**, **" || "**)  
 line = line.replace(**" or "**, **" || "**)  
 *#line = line.replace("contextData.", "contextData\_")* dftemp[line] = self.call\_local\_run\_expression(data, line, line,p1)[line]  
   
 b = datetime.now()  
   
 print(**"\n Time in seconds - Data Load : "**+str((a-c))+**"--->"**+str((a-c).total\_seconds()))  
 print(**"\n Time in seconds - Expression Execution : "**+str((b-a))+**"--->"**+str((b-a).total\_seconds()))   
   
 dftemp.to\_csv(**"D:/Mercer/Mercer\_expression\_testing/large\_file\_test\_data/"**+sheet+**"\_"**+str(data.shape[0])+**".csv"**)  
 *#dftemp.to\_csv("/home/audmosd/mercer\_expression\_validator/PE\_Realignment\_flask2\_new/benchmarking/output/50k/"+sheet+"\_"+str(data.shape[0])+".csv")* d = datetime.now()  
 print(**"\n Time in seconds - Data Write : "**+str((d-b))+**"--->"**+str((d-b).total\_seconds()))  
 print(**"----------------------------------------------------------------------------------"**)  
   
 p1.close()  
 p1.join()  
  
  
  
 **def** call\_local\_run\_expression(self, data\_frame, expression, expression\_serial,p1):  
 step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: expression,  
 **"new\_column\_name"**: expression\_serial}  
 runner = ExpressionRunner(step)  
 **if**(**not**(any(function **in** expression **for** function **in** [**"XAVG"**,**"XCOUNT"**,**"XMAX"**,**"XMIN"**,**"XSUM"**,**"XUNIQUECOUNT"**,**"GREATEST"**,**"LEAST"**,**"LOOK\_UP\_CHECK"**]))):  
 print(**"multiple thread run..."**)  
 df\_split = np.array\_split(data\_frame, 8)  
 result = pd.concat(p1.map(runner.get\_result, df\_split))  
 **else**:  
 print(**"single thred run..."**)  
 result = runner.get\_result(data\_frame)  
 **return** result

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** expression.MysqlLoader **import** MysqlLoder  
**from** dataframe\_reader.data\_wrangler **import** DataWranglerService  
  
  
**class** TestMercerExpressionApi(unittest.TestCase):  
  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
 """  
 def test\_run\_find\_in\_set(self):  
 data\_frame = pd.read\_csv('../resources/SG\_HT\_Library\_July18\_Selceted.csv', delimiter=',')  
 #print(data\_frame.head())  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner", "expression": "FIND\_IN\_SET(CTRY\_CODE, \"SG,SM\")",  
 "new\_column\_name": "find\_in\_set\_status"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 #print(result.head())  
 self.assertEqual(round(result['find\_in\_set\_status'][0]),True)  
 """  
  
 def** test\_xavg(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"total\_runs > XAVG(total\_runs,ball > 1,bowling\_team)"**,  
 **"new\_column\_name"**: **"find\_valid\_runner"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=20)  
 *#print(data\_frame.head())* runner = ExpressionRunner(step)  
 column\_name = **"find\_valid\_runner"** data\_frame = runner.add\_new\_column(data\_frame)  
 *#print(data\_frame[['total\_runs','ball','bowling\_team','find\_valid\_runner']])* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
 **def** test\_lookup\_tryout(self):  
  
 df = pd.DataFrame({**'col1'**: [**'M'**,**'N'**,**'O'**,**'P'**], **'col2'**: [**'B'**,**'V'**,**''**,**None**]})  
 lookupdf = pd.DataFrame({**'wherecol'**: [**'M'**,**'N'**,**'O'**,**'P'**], **'selectcol'**: [**'m'**,**'n'**,**'o'**,**'p'**]})  
 lookupdict = lookupdf[lookupdf.wherecol.isin([**'M'**,**'N'**])].set\_index(**'wherecol'**)[**'selectcol'**].to\_dict()  
 *#f['col1'].map(lookupdict)  
 #print(lookupdict);* df[**'col3'**]=df[**'col1'**].map(lookupdict)  
 *#print(df.head())* **def** test\_xavg(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"XAVG(total\_runs,,bowling\_team)"**,  
 **"new\_column\_name"**: **"find\_valid\_runner"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=20)  
 *#print(data\_frame.head())* runner = ExpressionRunner(step)  
 column\_name = **"find\_valid\_runner"** data\_frame = runner.add\_new\_column(data\_frame)  
 *#print(data\_frame[['total\_runs','ball','bowling\_team','find\_valid\_runner']])* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
 **def** test\_xcount(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"XCOUNT(total\_runs,,)"**,  
 **"new\_column\_name"**: **"find\_valid\_runner"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=20)  
 *#print(data\_frame.head())* runner = ExpressionRunner(step)  
 column\_name = **"find\_valid\_runner"** data\_frame = runner.add\_new\_column(data\_frame)  
 *#print(data\_frame[['total\_runs','ball','bowling\_team','find\_valid\_runner']])* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_min(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"XMIN(total\_runs,,)"**,  
 **"new\_column\_name"**: **"find\_valid\_runner"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=20)  
 *#print(data\_frame.head())* runner = ExpressionRunner(step)  
 column\_name = **"find\_valid\_runner"** data\_frame = runner.add\_new\_column(data\_frame)  
 *#print(data\_frame[['total\_runs','ball','bowling\_team','find\_valid\_runner']])* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
 **def** test\_max(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"XMAX(total\_runs,,)"**,  
 **"new\_column\_name"**: **"find\_valid\_runner"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=20)  
 *#print(data\_frame.head())* runner = ExpressionRunner(step)  
 column\_name = **"find\_valid\_runner"** data\_frame = runner.add\_new\_column(data\_frame)  
 *#print(data\_frame[['total\_runs','ball','bowling\_team','find\_valid\_runner']])* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
 **def** test\_sum(self):  
 *"""  
 Test case for adding new column with if else condition of expression* **:return***:  
 """* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,  
 **"expression"**: **"XSUM(total\_runs,,)"**,  
 **"new\_column\_name"**: **"find\_valid\_runner"**}  
 data\_frame = pd.read\_csv(**'../resources/deliveries.csv'**, header=0, nrows=20)  
 *#print(data\_frame.head())* runner = ExpressionRunner(step)  
 column\_name = **"find\_valid\_runner"** data\_frame = runner.add\_new\_column(data\_frame)  
 *#print(data\_frame[['total\_runs','ball','bowling\_team','find\_valid\_runner']])* **if** column\_name **in** data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
 **def** test\_run\_toString(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [ **'1'** , 2 , 3 , 4]})  
 *#print(data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"TO\_STRING(col1)"**,  
 **"new\_column\_name"**: **"col1\_string"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 *#print(result.head())* self.assertEqual(type(result[**'col1\_string'**][0]),type(**'str'**))  
  
  
 **def** test\_run\_toNumeric(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'col1'**: [ **'1'** , **'2'** , **'3'** , **'4'**]})  
 *#print(data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"TO\_NUMBER(col1)"**,  
 **"new\_column\_name"**: **"col1\_numeric"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 *#print(result.head())  
 #print(type(result['col1\_numeric'][0]))* **def** test\_run\_now(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'date'**: [**'2015-01-08 22:44:09'**,**'today is 2010 april '**,**'2015-01-08 22:44:09'**,**'2015-01-08 22:44:09'**,**'2015-01-08 22:44:09'**]})   
 *#print(data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"NOW();"**,  
 **"new\_column\_name"**: **"current\_date"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 *#print(result.head())  
 #self.assertEqual(result['col1\_rounded'][0], 2)* **def** test\_run\_toDate(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'date'**: [**'2015-01-08 22:44:09'**,**'2015-01-08 22:44:09'**,**'2015-01-08 22:44:09'**,**'2015-01-08 22:44:09'**,**'2015-01-08 22:44:09'**]})   
 *#print(data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"TO\_DATE(date);"**,  
 **"new\_column\_name"**: **"toDate\_output"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 *#print(result.head())* self.assertEqual(type(result[**'toDate\_output'**][0]), pd.\_libs.tslibs.timestamps.Timestamp)  
  
  
 **def** test\_run\_year(self):  
 *"""  
 This method is used for testing api for expression* **:return***:  
 """* data\_frame = pd.DataFrame({**'date'**: [**'2015-01-08 22:44:09'**,**'08-01-2015'**,**'2015-01-08 22:44:09'**,**'2015-01-08 22:44:09'**,**'2015-01-08 22:44:09'**]})   
 *#print(data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**, **"expression"**: **"YEAR(date);"**,  
 **"new\_column\_name"**: **"toDate\_output"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 *#print(result.head())* self.assertEqual(result[**'toDate\_output'**][0], 2015)  
  
   
 **"""  
 def test\_lookup(self):  
 master\_df = pd.read\_csv("D:/Workspace/MOSAIC\_LENS/insighthub/DataMashUpApi/resources/ops\_audit\_ly\_poscode\_comp1.csv")  
 DataRegistry.register\_data\_frame('m',1000,master\_df)   
 data\_frame = pd.read\_csv('D:/Workspace/MOSAIC\_LENS/insighthub/DataMashUpApi/resources/SG\_HT\_Library\_1nov.csv', delimiter=',')  
 data\_frame = data\_frame.head(5)  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "LOOK\_UP\_CHECK(\"m\",\"m.CTX\_CTRY\_CODE = this.CTRY\_CODE AND m.POS\_CODE = this.POS\_CODE\",\"m.ERROR\_MIN\");",  
 "new\_column\_name": "string\_outcome"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 #print(result)  
 #self.assertEqual(result['string\_outcome'][0],"Kolkata Knight RidersKolkata Knight Riders")  
 """  
  
 """  
 def test\_ifelse\_base(self):  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "if(total\_runs==0) \  
 {value = \"NO\_RUNS\";} \  
 else { \  
 value = \"GOOD\_RUNS\";}",  
 "new\_column\_name": "output"}  
 data\_frame = pd.read\_csv('../resources/deliveries.csv', header=0, nrows=10)  
 runner = ExpressionRunner(step)  
 column\_name = "output"  
 data\_frame = runner.add\_new\_column(data\_frame)  
 #print(data\_frame[['total\_runs','ball','output']])  
 if column\_name in data\_frame.columns:  
 self.assertEqual(1, 1)  
 else:  
 self.assertEqual(1, 2)  
 """   
  
 """  
 def test\_ifelse\_series(self):  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": " \  
 if(total\_runs == 0) {value = total\_runs + 100;} \  
 else if(total\_runs == 1){value = total\_runs+1000;}\  
 else {value = total\_runs+100000;} value;\  
 value3 = value + 100\  
 \  
 \  
 if(value3 < 10000) {value4 = -111111;} else {value4 = -22222;}value4;",  
 "new\_column\_name": "output"}  
 data\_frame = pd.read\_csv('../resources/deliveries.csv', header=0, nrows=10)  
 runner = ExpressionRunner(step)  
 column\_name = "output"  
 data\_frame = runner.add\_new\_column(data\_frame)  
 print(data\_frame[['total\_runs','ball','output','value', 'value3', 'value4']])  
 if column\_name in data\_frame.columns:  
 self.assertEqual(1, 1)  
 else:  
 self.assertEqual(1, 2)  
 """  
   
  
 """  
 def test\_ifelse\_lamda(self):  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "value = 0; if(1>1){value = 5;}else{value = 1;}value;\  
 if(value==XAVG(ball)){r1 = \"abcd\";}else{r1 =\"pqrs\";}r1;",  
 "new\_column\_name": "output"}  
 data\_frame = pd.read\_csv('../resources/deliveries.csv', header=0, nrows=1)  
 runner = ExpressionRunner(step)  
 column\_name = "output"  
 data\_frame = runner.add\_new\_column(data\_frame)  
 print(data\_frame[['total\_runs','ball','output']])  
 if column\_name in data\_frame.columns:  
 self.assertEqual(1, 1)  
 else:  
 self.assertEqual(1, 2)  
 """  
  
  
 """  
 def test\_ifelse\_series(self):  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "FIND\_IN\_SET(204,204, 205, 218, 219);",  
 "new\_column\_name": "output"}  
 #Read and test data file as a frame:  
 test\_df = pd.DataFrame({'col1': ['A',True,False,False,False], 'col2': ['A',True,True,True,False]})  
 #test\_df = pd.read\_csv('../resources/deliveries.csv', header=0, nrows=10)  
 #test\_df = pd.read\_csv("D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/BOOLEAN/mts\_data\_boolean.txt", header=0, nrows=10000, delimiter="\t")  
 print(test\_df)  
 runner = ExpressionRunner(step)  
 column\_name = "output"  
 result = runner.add\_new\_column(test\_df)  
 print(result)  
 """  
  
 """  
 def test\_lookup(self):  
  
 #Cache mysql tables  
 mysqlloader = MysqlLoder('localhost','root','Newuser123','mercer\_pe')  
 all\_tables = mysqlloader.read\_all\_table\_names()['Tables\_in\_mercer\_pe']  
 for table in all\_tables:  
 if(table.islower):  
 if(table == "validorg"):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print("registerd table : "+str(table))  
  
 if(table == "ops\_career\_stream\_bonact\_perc\_au\_nz"):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print("registerd table : "+str(table))  
 if(table == "country"):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print("registerd table : "+str(table))  
 if(table == "ops\_career\_stream\_bonact\_perc"):  
 table\_df=mysqlloader.read\_table(table)  
 DataRegistry.register\_data\_frame(table,1000,table\_df)  
 print("registerd table : "+str(table))  
  
 else:  
 print("table is upper : "+table)  
  
 md = DataRegistry.get\_data\_set\_from\_dict("ops\_career\_stream\_bonact\_perc\_au\_nz","1000","ORIGINAL")  
 #print(md[md.CTX\_CTRY\_CODE == "AE"][['CTX\_CTRY\_CODE','AGE\_HIRE\_MIN']].head(20))  
 print(md.head(20))  
  
  
 #data  
 data\_frame = pd.read\_csv("D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/Lookup/mts\_data\_lookup.txt", header=0, nrows=10000, delimiter="\t")  
 print(data\_frame.head())  
  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "CL=SUBSTR(POS\_CODE,13,1);CS=SUBSTR(POS\_CODE,12,1);LOOK\_UP\_CHECK(\"ops\_career\_stream\_bonact\_perc\",\"ops\_career\_stream\_bonact\_perc.CTX\_CTRY\_CODE = this.ctryCode AND ops\_career\_stream\_bonact\_perc.CAREER\_STREAM = CS AND CAREER\_LEVEL = CL-4\",\"ops\_career\_stream\_bonact\_perc.ERROR\_MIN\");",  
 "new\_column\_name": "lookup\_outcome"}  
 print(step['expression'])  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result)  
 #self.assertEqual(result['string\_outcome'][0],"Kolkata Knight RidersKolkata Knight Riders")  
 """  
  
 """  
 def test\_findinset(self):  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "FIND\_IN\_SET(204,col2);",  
 "new\_column\_name": "output"}  
 #Read and test data file as a frame:  
 test\_df = pd.DataFrame({'col1': ['A',True,False,False,False],   
 'col2': ['A',True,True,True,False],  
 'col3': [204,205,206,207,208]})  
 #test\_df = pd.read\_csv('../resources/deliveries.csv', header=0, nrows=10)  
 #test\_df = pd.read\_csv("D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/BOOLEAN/mts\_data\_boolean.txt", header=0, nrows=10000, delimiter="\t")  
 print(test\_df)  
 runner = ExpressionRunner(step)  
 column\_name = "output"  
 result = runner.add\_new\_column(test\_df)  
 print(result)  
 """  
   
 """  
 def test\_findinset(self):  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "LEAST(XMIN(EMP\_160,POS\_CODE == \"CCB.02.028.S10\",), XMIN(EMP\_160,POS\_CODE == \"CCB.02.028.S20\",), XMIN(EMP\_160,POS\_CODE == \"CCB.02.028.S30\",));",  
 "new\_column\_name": "output"}  
 #Read and test data file as a frame:  
 #test\_df = pd.read\_csv('../resources/deliveries.csv', header=0, nrows=10)  
 test\_df = pd.read\_csv("D:/Mercer/Mercer\_expression\_testing/Basic\_expressions\_28NOV/aggregate/mts\_data\_aggregate.txt", header=0, nrows=10000, delimiter="\t")  
 print(test\_df)  
 runner = ExpressionRunner(step)  
 column\_name = "output"  
 result = runner.add\_new\_column(test\_df)  
 print(result)  
 """  
  
 """  
 def test\_substring(self):  
  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "SUBSTR(\"vaibhav Ajinath Bade\", LENGTH(bowling\_team)-10, 5;",  
 "new\_column\_name": "find\_valid\_runner"}  
 data\_frame = pd.read\_csv('../resources/deliveries.csv', header=0, nrows=20)  
 #print(data\_frame.head())  
 runner = ExpressionRunner(step)  
 column\_name = "find\_valid\_runner"  
 result = runner.add\_new\_column(data\_frame)  
 #print(result.head())  
 """  
  
 """  
 def test\_ifelse\_series(self):  
   
   
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "if(total\_runs<ball){\  
 value = total\_runs + 100;\  
 }\  
 else {\  
 value = total\_runs+100000;\  
 } value;",  
 "new\_column\_name": "output"}  
   
  
 data\_frame = pd.DataFrame({'pos\_code': [123,'abcd','..............','@$#&!\*()\*%%&\*(','XXX.XX.XXX.E9X','XXX.XX.XXX.M9X','XXX.XX.XXX.P9X','XXX.XX.XXX.S9X','XXX.XX.XXX.Z9X','XXX.XX.XXX.Z9X','SMP.XX.XXX.Z9X','ABC.XX.XXX.Z9X','aaaaaaaaaaaaaaa','1.00E+15']})  
  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "LENGTH(pos\_code);",  
 "new\_column\_name": "output"}  
  
 #data\_frame = pd.read\_csv('../resources/deliveries.csv', header=0, nrows=10)  
 print(data\_frame.head())  
 runner = ExpressionRunner(step)  
 column\_name = "output"  
 data\_frame = runner.add\_new\_column(data\_frame)  
 print(data\_frame[['output']])  
 """  
   
 """  
 def test\_regex\_match(self):   
 data\_frame = pd.DataFrame({'sample\_col':["vaibhav@gmail.com",None,"",np.nan,1],  
 'regex':["gmail.com","[a-z0-9.\_%+-]+@[a-z0-9.-]+\\.[a-z]{2,3}","[a-z0-9.\_%+-]+@[a-z0-9.-]+\\.[a-z]{2,3}","[a-z0-9.\_%+-]+@[a-z0-9.-]+\\.[a-z]{2,3}","[0-9]"]  
 })  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "REGEX\_MATCH(\"CA\",\"CAS\");",  
 "new\_column\_name": "output"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.head(25))  
 """  
   
 """  
 def test\_run\_find\_in\_set(self):  
 data\_frame = pd.read\_csv('../resources/SG\_HT\_Library\_July18\_Selceted.csv', delimiter=',')  
 #print(data\_frame.head())  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner", "expression": "FIND\_IN\_SET(\"SG\", [\"SG\",\"AG\"]);",  
 "new\_column\_name": "find\_in\_set\_status"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.head())  
 self.assertEqual(round(result['find\_in\_set\_status'][0]),True)   
 """  
  
  
 """  
 def test\_nvl(self):   
 data\_frame = pd.DataFrame({'sample\_col':["vaibhav@gmail.com",None,"",np.nan,1]})  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "NVL(sample\_col);",  
 "new\_column\_name": "output"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.head(25))  
 """  
 """  
 def test\_run\_to\_string(self):   
   
 data\_frame = pd.DataFrame({'sample\_col':["vaibhav@gmail.com",None,"",np.nan,1]})  
  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "TO\_STRING(sample\_col);",  
 "new\_column\_name": "output"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.head(25))  
 """  
   
 """  
 def test\_run\_to\_number(self):   
   
 data\_frame = pd.DataFrame({'sample\_col':[1.6,2,3,4,"3","Vaibhav",None,np.nan,""]})  
  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "TO\_NUMBER(sample\_col);",  
 "new\_column\_name": "output"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.dtypes)  
 print(result.head(25))  
 """** *# def test\_generic(self):  
   
 # """  
 # #MYSQL CASHING :   
 # mysqlloader = MysqlLoder('localhost','root','Newuser123','mercer\_pe',3306)  
 # all\_tables = mysqlloader.read\_all\_table\_names()['Tables\_in\_mercer\_pe']  
 # for table in all\_tables:  
 # if(table.islower):  
 # if(table == "country"):  
 # table\_df=mysqlloader.read\_table(table)  
 # DataRegistry.register\_data\_frame(table,1000,table\_df)  
 # print("registerd table : "+str(table))  
 # else:  
 # print("table is upper : "+table)  
  
 # md = DataRegistry.get\_data\_set\_from\_dict("country","1000","ORIGINAL")  
 # print(md.head())  
 # #col\_names = ['CTX\_CTRY\_CODE','CTX\_GRPCODE','CTX\_CPYCODE','NUM\_EMP']  
 # #print(md[col\_names].dtypes)  
 # #print(md[(md.CTX\_CTRY\_CODE == "GE") & (md.CTX\_GRPCODE == "106790") ][col\_names].head(100))  
 # #print(md[((md.CTX\_CTRY\_CODE.isin(["AE","GE","DE","PE"]))) ][col\_names].head(100))  
 # #print(md[md.CTX\_CPYCODE=="10009666"].head(100))  
 # #md.to\_csv("D:/Mercer/md.csv")  
 # """  
  
 # #LOOKUP :   
 # data\_frame = pd.DataFrame({'sample\_col':['a1', 'b2', 'c3'],  
 # 'regex':['([ab])(\\d)','([ab])(\\d)','([ab])(\\d)']  
 # })  
  
 # step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 # "expression": "REGEXP\_EXTRACT(regex, sample\_col);",  
 # "new\_column\_name": "output"}  
 # runner = ExpressionRunner(step)  
 # result = runner.get\_result(data\_frame)  
 # print(result.head(25))* **"""  
 def test\_run\_to\_number(self):   
   
 data\_frame = pd.DataFrame({'sample\_col':["I have car",'agghhhhh','ca',None, np.nan],  
 'regex':[".\*car.\*",'.\*vehicle.\*','.\*auto.\*',"",""]})  
  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "REGEX\_MATCH(regex, sample\_col);",  
 "new\_column\_name": "output"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.dtypes)  
 print(result.head(25))  
 """  
  
   
 """  
 def test\_regex\_match(self):   
 data\_frame = pd.DataFrame({'sample\_col':["vaibhav@gmail.com",None,"",np.nan,1],  
 'regex':["gmail.com","[a-z0-9.\_%+-]+@[a-z0-9.-]+\\.[a-z]{2,3}","[a-z0-9.\_%+-]+@[a-z0-9.-]+\\.[a-z]{2,3}","[a-z0-9.\_%+-]+@[a-z0-9.-]+\\.[a-z]{2,3}","[0-9]"]  
 })  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",  
 "expression": "GREATEST(1,2,34,5);",  
 "new\_column\_name": "output"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.head(25))  
 """  
  
   
 """  
 def test\_run\_round(self):  
 data\_frame = pd.DataFrame({'col1': ['car','c'],  
 'col2':["[.\*car.\*]","[.\*car.\*]"]})  
 #print(data\_frame.head())  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",   
 "expression": "REGEX\_MATCH(\"vai\",\"vaibhav\");",  
 "new\_column\_name": "col1\_rounded"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.head())  
 self.assertEqual(result['col1\_rounded'][0], 2)  
 """  
  
 """   
 def test\_run\_greatest(self):  
 data\_frame = pd.DataFrame({'col1': [1,2,3,4], 'col2': [5,6,7,8]})   
 #print(data\_frame.head())  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner", "expression": "GREATEST(col1,col2);",  
 "new\_column\_name": "greatest\_value"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 #print(result.head())  
 self.assertEqual(result['greatest\_value'][0], 5)  
 """  
  
  
 """  
 def test\_run\_least(self):  
 data\_frame = pd.DataFrame({'col1': [1,2,3,4], 'col2': [5,6,7,8]})   
 print("\n \*\*\*\*\*\*\*\*INPUT\*\*\*\*\*\*\*")  
 print(data\_frame.head())  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",   
 "expression": "GREATEST(1,2,3,4,5);",  
 "new\_column\_name": "least\_value"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print("\n \*\*\*\*\*\*\*\*OUTPUT\*\*\*\*\*\*\*")  
 print(result.head())  
 self.assertEqual(result['least\_value'][0], 1)  
 """  
  
 """  
 def test\_run\_nvl(self):  
 data\_frame = pd.DataFrame({'col1': [1.2,1,1,1,None, np.nan,1.2,1.3]})  
 print("\n\n \*\*\*\*\*\*INPUT DATA\*\*\*\*\*\*")  
 print(data\_frame.head(30))  
 step = {"step\_name": "expression.run\_expression.ExpressionRunner",   
 "expression": "ISBLANK(col1);",  
 "new\_column\_name": "col1\_fillin"}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print("\*\*\*\*\*\*OUTPUT DATA\*\*\*\*\*\*\*\*")  
 print(result.head(30))  
 self.assertNotEqual(result['col1\_fillin'][1],np.nan)  
 """  
  
  
 def** test\_run\_least(self):  
 data\_frame = pd.DataFrame({**'col1'**: [100,2,3,10,**None**,**""**,np.nan],   
 **'col2'**: [100.0,6,7,8,1,2,3]})   
 *#print(data\_frame.head())* step = {**"step\_name"**: **"expression.run\_expression.ExpressionRunner"**,   
 **"expression"**: **"col1 == col2;"**,  
 **"new\_column\_name"**: **"least\_value"**}  
 runner = ExpressionRunner(step)  
 result = runner.get\_result(data\_frame)  
 print(result.head())  
 self.assertEqual(result[**'least\_value'**][0], 1)

*"""  
This module is used for testing mercer expr with api  
"""***import** unittest  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** expression.run\_expression **import** ExpressionRunner  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
*#from expression.MysqlLoader import MysqlLoder***from** validator\_Services.mosaic\_validation\_services\_router **import** ValdiationServiceRouter  
  
  
**class** TestMercerExpressionApi(unittest.TestCase):  
  
 **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
 """  
 def test\_validator\_double(self):  
 step = { "data\_source\_id": "df\_28nov",  
 "input\_column": "Doublecol",  
 "output\_column": "output1",  
 "id\_column": "record\_number",  
 "operation": {"operation\_name": "datatype",  
 "operation\_type": "check\_cast",  
 "question\_type": "double",  
 "data\_type": "double",  
 "pattern\_array": {}}}  
  
 df = pd.DataFrame({'Doublecol': [200,300,'400',500.101,1.0,200,205,210,None,'','abcd',' ','a','b','Ö','2.3e50',12345678987654321,"12345678987654321123",-123.456,-123989.9898,9.8e-5,-7.8e16,-7.8e-5,-7.8e15,7.8e15],'phonecol': [9156359234,'+919156359234','09156359234',9689962412,'(551)427-9499','(412)595-9549','(289)928-8506','(503)558-2725','(716)897-6367','(571)398-3238','(216)883-6780','(847)617-1752','(641)586-5639','(501)378-4189','A','AA','@','1,234','a:',1.0,25.55,'F','K','CC','ab']})  
 DataRegistry.register\_data\_frame('testdf',1000,df)  
 step['data\_source\_id'] = 'testdf'  
 validation\_service\_object = ValdiationServiceRouter(step)  
 result = validation\_service\_object.validate\_service\_routing()  
 result = pd.concat([result,df], axis=1)  
 print(result.head(25))  
 print(result.dtypes)  
 print(result.to\_json(orient="records"))  
 """  
   
   
  
 """   
 def test\_validator\_phone\_number(self):  
 step = { "data\_source\_id": "df\_28nov",  
 "input\_column": "phonecol",  
 "output\_column": "output1",  
 "id\_column": "record\_number",  
 "operation": {"operation\_name": "phone\_number",  
 "operation\_type": "check",  
 "question\_type": "unused",  
 "data\_type": "unused",  
 "pattern\_array": {}}}  
  
 df = pd.DataFrame({'Doublecol': [123,200,300,'400',500.101,1.0,200,205,210,None,'','abcd',' ','a','b','Ö','2.3e50',12345678987654321,"12345678987654321123",'-123.456',-123989.9898,9.8e-5,-7.8e16,-7.8e-5,-7.8e15,7.8e15],  
 'phonecol': [123,9156359234,'+919156359234','09156359234',9689962412,'(551)427-9499','(412)595-9549','(289)928-8506','(503)558-2725','(716)897-6367','(571)398-3238','(216)883-6780','(847)617-1752','(641)586-5639','(501)378-4189','A','AA','@','1,234','a:',1.0,25.55,'F','K','CC','ab']})  
 DataRegistry.register\_data\_frame('testdf',1000,df)  
 step['data\_source\_id'] = 'testdf'  
 validation\_service\_object = ValdiationServiceRouter(step)  
 result = validation\_service\_object.validate\_service\_routing()  
 result = pd.concat([result,df], axis=1)  
 print(result.head(25))  
 """  
   
  
 """   
 def test\_validator\_email\_id(self):  
 step = { "data\_source\_id": "df\_28nov",  
 "input\_column": "emailcol",  
 "output\_column": "output1",  
 "id\_column": "record\_number",  
 "operation": {"operation\_name": "email\_id",  
 "operation\_type": "check",  
 "question\_type": "unused",  
 "data\_type": "unused",  
 "pattern\_array": {}}}  
  
 df = pd.DataFrame({'emailcol': ['vaibhavbade16@gmail.com','vaibhav.bade@gmail.com','v@gmail.com','b@lntinfotech.com','a.b16@lntinfotech.com','1@gmail.com','@@gmail.com','#&@gmail.com','A','AA','','GG',1,2,3,4.0,25.50,40.0,50.0,60.0,'A','s','cc','s','ZZ']})  
 DataRegistry.register\_data\_frame('testdf',1000,df)  
 step['data\_source\_id'] = 'testdf'  
 validation\_service\_object = ValdiationServiceRouter(step)  
 result = validation\_service\_object.validate\_service\_routing()  
 result = pd.concat([result,df], axis=1)  
 print(result.head(25))  
 """  
   
  
  
 """   
 def test\_validator\_integer(self):  
 step = { "data\_source\_id": "df\_28nov",  
 "input\_column": "integercol",  
 "output\_column": "output1",  
 "id\_column": "record\_number",  
 "operation": {"operation\_name": "datatype",  
 "operation\_type": "check\_cast",  
 "question\_type": "integer",  
 "data\_type": "integer",  
 "pattern\_array": {}}}  
  
 df = pd.DataFrame({'integercol': [200,300,'400',500.101,1.0,200,205,210,None,'','abcd',' ','a','b','Ö','2.3e5',123456789,"123456789",'-123.456',-123989.9898,9.8e-5,-7.8e16,-7.8e-5,-7.8e15,7.8e15]})  
 DataRegistry.register\_data\_frame('testdf',1000,df)  
 step['data\_source\_id'] = 'testdf'  
 validation\_service\_object = ValdiationServiceRouter(step)  
 result = validation\_service\_object.validate\_service\_routing()  
 result = pd.concat([df,result], axis=1)  
 print(result.to\_json(orient="records").replace("-11111111111111","\"\""))  
 print(result.astype(str).replace("-11111111111111","''").head(25))  
 print(result.dtypes)  
 """   
  
  
   
 """   
 def test\_validator\_date(self):  
  
 step = { "data\_source\_id": "df\_28nov",  
 "input\_column": "dateCol",  
 "output\_column": "output1",  
 "id\_column": "record\_number",  
 "operation": {"operation\_name": "datatype",  
 "operation\_type": "check\_cast",  
 "question\_type": "date",  
 "data\_type": "date",  
 "pattern\_array": {  
 "dateFormat": ["MM/dd/yyyy","dd/MM/yyyy","dd-MM-yyyy","dd.MM.yyyy","yyyy-MM-dd","yyyy/MM/dd","yyyy.MM.dd"],  
 "numberFormat": [{"pattern": "###.###,###","thousandsSeparator": ".","decimalSeparator": ","  
 }]}}}  
  
 df = pd.DataFrame({'dateCol': ["01/23/1990","20/02/1889","20-02-1886","18.04.2000","1887-05-28","27-09-1993",  
 "1947.06.22","31-12-2003","27-11-2009","1867.12.28","30-02-1896","31/12/1889",  
 '2018-06-29 08:15:27.243860',  
 'Jun 28 2018 7:40AM',  
 'Jun 28 2018 at 7:40AM',  
 'September 18, 2017, 22:19:55',  
 'Sun, 05/12/1999, 12:30PM',  
 'Mon, 21 March, 2015',  
 '2018-03-12T10:12:45Z',  
 '2018-06-29 17:08:00.586525+00:00',  
 '2018-06-29 17:08:00.586525+05:00',  
 'Tuesday , 6th September, 2017 at 4:30pm',  
 'T47']})  
  
 DataRegistry.register\_data\_frame('testdf',1000,df)  
 step['data\_source\_id'] = 'testdf'  
  
 validation\_service\_object = ValdiationServiceRouter(step)  
 result = validation\_service\_object.validate\_service\_routing()  
 result = pd.concat([df,result], axis=1)  
 #print(result.to\_json(orient="records"))  
 print(result.head(25))  
 print(result.dtypes)  
 """  
   
  
 """  
 def test\_validator\_percentage(self):  
 step = { "data\_source\_id": "df\_28nov",  
 "input\_column": "percentagecol",  
 "output\_column": "output1",  
 "id\_column": "record\_number",  
 "operation": {"operation\_name": "percentage",  
 "operation\_type": "check\_cast",  
 "question\_type": "percentage",  
 "data\_type": "double",  
 "pattern\_array": {}}}  
  
 df = pd.DataFrame({'percentagecol': ['1$.\_2%',0.5,'0.60%','10%','10.0%',10,10.0,0.00,'A','B','%','2%2.0']})  
 DataRegistry.register\_data\_frame('testdf',1000,df)  
 step['data\_source\_id'] = 'testdf'  
 validation\_service\_object = ValdiationServiceRouter(step)  
 result = validation\_service\_object.validate\_service\_routing()  
 result = pd.concat([df,result], axis=1)  
 #print(result.to\_json(orient="records"))  
 print(result.head(25))  
 print(result.dtypes)  
 """  
   
 """  
 def test\_validator\_money(self):  
  
 step = { "data\_source\_id": "df\_28nov",  
 "input\_column": "moneycol",  
 "output\_column": "output1",  
 "id\_column": "record\_number",  
 "operation": {"operation\_name": "money",  
 "operation\_type": "check\_cast",  
 "question\_type": "money",  
 "data\_type": "double",  
 "pattern\_array": {  
 "dateFormat": ["MM/dd/yyyy","dd/MM/yyyy","dd-MM-yyyy","dd.MM.yyyy","yyyy-MM-dd","yyyy/MM/dd","yyyy.MM.dd"],  
 "numberFormat": [{"pattern": "###.###,###","thousandsSeparator": ".","decimalSeparator": ","  
 }]}}}  
  
 df = pd.DataFrame({'moneycol': ["2.50.000,20","10.000.00,00","4.50.0000,00","1.000,00","23.000,00","55.00.000,45","10000,56",  
 "1.100.000,00","60.00.000,23","1.20.000.00,45","10.00.00.000,23",""," ","abcd",1,"2,50,000.20",  
 "10,000,00.00","4,50,0000.00","1,000.00","23,000.00","55,00,000.45","10000.56","11,00,000.00",  
 "60,00,000.23","1,20,000,00.45","10,00,00,000.23",12345,4,50,0000.00,None]})  
  
 DataRegistry.register\_data\_frame('testdf',1000,df)  
 step['data\_source\_id'] = 'testdf'  
  
 validation\_service\_object = ValdiationServiceRouter(step)  
 result = validation\_service\_object.validate\_service\_routing()  
 result = pd.concat([df,result], axis=1)  
 print(result.to\_json(orient="records"))  
 print(result.head(25))  
 print(result.dtypes)  
 """  
  
   
 def** test\_validator\_year(self):  
  
 step = { **"data\_source\_id"**: **"df\_28nov"**,  
 **"input\_column"**: **"yearCol"**,  
 **"output\_column"**: **"output1"**,  
 **"id\_column"**: **"record\_number"**,  
 **"operation"**: {**"operation\_name"**: **"year"**,  
 **"operation\_type"**: **"check\_cast"**,  
 **"question\_type"**: **"year"**,  
 **"data\_type"**: **"integer"**,  
 **"pattern\_array"**: {  
 *#"dateFormat": ["MM/dd/yyyy","dd/MM/yyyy","dd-MM-yyyy","dd.MM.yyyy","yyyy-MM-dd","yyyy/MM/dd","yyyy.MM.dd"],* **"dateFormat"**: **""**,  
 **"numberFormat"**: [{**"pattern"**: **"###.###,###"**,**"thousandsSeparator"**: **"."**,**"decimalSeparator"**: **","** }]}}}  
  
 df = pd.DataFrame({  
 **'yearCol'** : [2020, 1900.1, 20201, **'10/10/2018'**, **'2020-05-05'**, **'1800'**, **'qwerty'**, **'05-05-18'**,   
 **'22.05.1989'**, **'2020'**, **'2018 00'**, **'2000'**, **'555'**, **'01/23/1990'**, **'20/02/1889'**,   
 **'20-02-1886'**, **'18.04.2000'**, **'1887-05-28'**, **'27-09-1993'**, **'1947.06.22'**, **'31-12-2003'**,  
 **'27-11-2009'**, **'1867.12.28'**, **'30-02-1896'**, **'31/12/1889'**, **'2018-06-29 08:15:27.243860'**,   
 **'Jun 28 2018 7:40AM'**, **'Jun 28 2018 at 7:40AM'**, **'September 18, 2017, 22:19:55'**,   
 **'Sun, 05/12/1999, 12:30PM'**, **'Mon, 21 March, 2015'**, **'2018-03-12T10:12:45Z'**,   
 **'2018-06-29 17:08:00.586525+00:00'**, **'2018-06-29 17:08:00.586525+05:00'**,   
 **'Tuesday , 6th September, 2017 at 4:30pm'**, **'T47'**,1989.0,**'1989.0'**,  
 1989.5,**"1989.5"**,-1989.0,**"-1989.0"**,-1989.6,**"-1989.6"**, 0,1.0,1]})  
  
  
 DataRegistry.register\_data\_frame(**'testdf'**,1000,df)  
 step[**'data\_source\_id'**] = **'testdf'** validation\_service\_object = ValdiationServiceRouter(step)  
 result = validation\_service\_object.validate\_service\_routing()  
 result = pd.concat([df,result], axis=1)  
 *#print(result.to\_json(orient="records").replace("-11111111111111","\"\""))* print(result.astype(str).replace(**"-11111111111111"**,**"''"**).head(36))  
 *#print(result.dtypes)  
   
  
  
  
 filter*

*"""  
This module is used for apply filter.  
"""***import** constants  
**from** filter.range\_filter **import** RangeFilter  
**from** filter.text\_filter **import** TextFilter  
  
  
**class** FilterCondition(object):  
 *"""  
 This class is used to apply filter of text ,date and numeric  
 """* **def** \_\_init\_\_(self, step\_json):  
 self.step\_json = step\_json  
 self.condition\_value = **None** self.nested\_condition\_list = []  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method is used to execute* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 condition\_create = self.create\_condition(self.step\_json[constants.FILTER\_CONDITIONS], data\_frame)  
  
 **if** self.step\_json[constants.CONDITION\_EXCLUDE] **is True**:  
 **return** data\_frame[~condition\_create]  
 **else**:  
 **return** data\_frame[condition\_create]  
 **except** KeyError **as** key\_error:  
 **raise** Exception(key\_error.args[0] + **" column is missing in input json"**)  
 **except** ValueError **as** value\_error:  
 **raise** Exception(**"no condition specified for the filter."** + value\_error.args[0])  
  
 @classmethod  
 **def** create\_filter\_condition\_based(cls, individual\_condition,condition, data\_frame):  
 *"""  
 This method is used to apply filter on condition* **:param** *condition:* **:param** *data\_frame* **:param** *individual\_condition* **:return***:  
 """* **if** condition[constants.FUNC].lower() == constants.WHEN\_OPERATOR:  
 print()  
  
 **if** condition[constants.COLUMN\_TYPE] == constants.NUMBER **or** condition[constants.COLUMN\_TYPE] == constants.DATE \  
 **or** condition[constants.COLUMN\_TYPE] == constants.INTEGER:  
 **if** individual\_condition **is None**:  
 individual\_condition = RangeFilter(condition).get\_result(data\_frame)  
 **else**:  
 **if** condition[constants.FUNC].lower() == constants.AND:  
 individual\_condition = individual\_condition & RangeFilter(condition).get\_result(data\_frame)  
 **elif** condition[constants.FUNC].lower() == constants.OR:  
 individual\_condition = individual\_condition | RangeFilter(condition).get\_result(data\_frame)  
  
 **elif** condition[constants.COLUMN\_TYPE] == constants.STRING:  
 **if** individual\_condition **is None**:  
 individual\_condition = TextFilter(condition).get\_result(data\_frame)  
  
 **if** condition[constants.FUNC].lower() == constants.AND:  
 individual\_condition = individual\_condition & TextFilter(condition).get\_result(data\_frame)  
 **elif** condition[constants.FUNC].lower() == constants.OR:  
 individual\_condition = individual\_condition | TextFilter(condition).get\_result(data\_frame)  
  
 **return** individual\_condition  
  
 **def** create\_condition(self, conditions, data\_frame):  
 *"""  
 This method is used to create condition for filter* **:param** *data\_frame:* **:param** *conditions* **:return***:  
 """* condition\_list = []  
 **for** condition **in** conditions:  
 individual\_condition = **None** individual\_condition = self.create\_filter\_condition\_based(individual\_condition,condition, data\_frame)  
 **if** len(condition[constants.NESTED\_CHILDREN]) > 0:  
 self.find\_all\_children\_json(condition[constants.NESTED\_CHILDREN])  
 **for** another\_condition **in** self.nested\_condition\_list:  
 individual\_condition = self.create\_filter\_condition\_based(individual\_condition,  
 another\_condition, data\_frame)  
 condition\_list.append(individual\_condition)  
  
 condition\_create = **None  
 for** index **in** range(0, len(conditions)):  
 **if** conditions[index][constants.FUNC].lower() == constants.WHEN\_OPERATOR:  
 condition\_create = condition\_list[index]  
 **elif** conditions[index][constants.FUNC].lower() == constants.AND:  
 condition\_create = condition\_create & condition\_list[index]  
 **elif** conditions[index][constants.FUNC].lower() == constants.OR:  
 condition\_create = condition\_create | condition\_list[index]  
  
 **return** condition\_create  
  
 **def** find\_all\_children\_json(self, nested\_json):  
 *"""  
 find the nested json from array* **:param** *nested\_json:* **:return***:  
 """* **for** json **in** nested\_json:  
 self.nested\_condition\_list.append(json)  
 **if** len(json[constants.NESTED\_CHILDREN]) > 0:  
 self.find\_all\_children\_json(json[constants.NESTED\_CHILDREN])

*"""  
This module is used for Ranger type of filter for date and number.  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** pandas **as** pd  
**import** constants  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** RangeFilter:  
 *"""  
 This class is used for apply ranger type of filter on dataFrame.  
 """* **def** \_\_init\_\_(self, json\_value):  
 **try**:  
 self.column\_name = json\_value[constants.COLUMN\_NAME]  
 self.to\_value = json\_value[constants.VALUE]  
 self.column\_type = json\_value[constants.COLUMN\_TYPE]  
 self.operator = json\_value[constants.OPERATOR]  
 self.exclude = json\_value[constants.IS\_EXCLUDED]  
 self.from\_value = json\_value[constants.FROM\_VALUE] \  
 **if** constants.FROM\_VALUE **in** json\_value **else None** self.value\_list = []  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method is used for applying query and generate result.* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 data\_frame\_new = **None** self.convert\_to\_date\_type()  
 **if** self.operator == **'>'**:  
 data\_frame\_new = (data\_frame[self.column\_name] > self.to\_value)  
 **elif** self.operator == **'>='**:  
 data\_frame\_new = (data\_frame[self.column\_name] >= self.to\_value)  
 **elif** self.operator == **'<'**:  
 data\_frame\_new = (data\_frame[self.column\_name] < self.to\_value)  
 **elif** self.operator == **'<='**:  
 data\_frame\_new = (data\_frame[self.column\_name] <= self.to\_value)  
 **elif** self.operator == **'='**:  
 data\_frame\_new = (data\_frame[self.column\_name] == self.to\_value)  
 **elif** self.operator **in** [**'in'**, **'IN'**, **'In'**, **'iN'**]:  
 data\_frame\_new = data\_frame[self.column\_name].isin(self.value\_list)  
 **else**:  
 data\_frame\_new = (data\_frame[self.column\_name] != self.to\_value)  
 **if** self.exclude **is True**:  
 **return** ~data\_frame\_new  
 **return** data\_frame\_new  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.COLUMN\_NOT\_FOUND.format(key\_error.args[0]))  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(  
 **"input data\_set is null.Please reload our use another one."** + ex.args[0])  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** convert\_to\_date\_type(self):  
 *"""  
 method used to change the value to its respective datatype* **:return***:  
 """* **try**:  
 **if** self.operator **in** [**'in'**, **'IN'**, **'In'**, **'iN'**]:  
 self.to\_value = str(self.to\_value)  
 self.value\_list = self.to\_value.split(**','**)  
 **if** self.column\_type.lower() == **"date" or** self.column\_type.lower() == **"timestamp"** \  
 **or** self.column\_type.lower() == **"datetime64[ns]"**:  
 self.value\_list = [pd.to\_datetime(x) **for** x **in** self.value\_list]  
 **elif** self.column\_type.lower() == **'integer'**:  
 self.value\_list = [int(x) **for** x **in** self.value\_list]  
 **else**:  
 self.value\_list = [float(x) **for** x **in** self.value\_list]  
 **else**:  
 **if** self.column\_type.lower() == **"date" or** self.column\_type.lower() == **"timestamp"** \  
 **or** self.column\_type.lower() == **"datetime64[ns]"**:  
 self.to\_value = pd.to\_datetime(str(self.to\_value))  
 **elif** self.column\_type.lower() == **'integer'**:  
 self.to\_value = int(self.to\_value)  
 **else**:  
 self.to\_value = float(self.to\_value)  
 **except** ValueError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to apply filter condition  
"""***import** json  
**import** pandas **as** pd  
**import** unittest  
**from** filter.filter\_condition **import** FilterCondition  
  
  
**class** FilterApply(unittest.TestCase):  
  
 **def** setUp(self):  
 self.condition\_value = **None** filename = **'../resources/filter.json'  
 with** open(filename, **'r'**) **as** f:  
 self.data\_store = json.load(f)  
 self.data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, delimiter=**'\t'**)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_create\_condition\_for\_filter(self):  
 *"""  
 Test suite for running filter step json on dataframe* **:return***:  
 """* data\_frame = FilterCondition(self.data\_store).get\_result(self.data\_frame)  
 **if** data\_frame **is None**:  
 self.assertEqual(1, 2)  
 **else**:  
 self.assertEqual(1, 1)  
  
 self.data\_store[**"exclude"**] = **True** data\_frame = FilterCondition(self.data\_store).get\_result(self.data\_frame)  
 **if** data\_frame **is None**:  
 self.assertEqual(1, 2)  
 **else**:  
 self.assertEqual(1, 1)  
  
 **def** test\_filter\_condition\_missing\_key(self):  
 *"""  
 Test suite for filter condition when input json missing some key* **:return***:  
 """* self.data\_store.pop(**"conditions"**)  
 **try**:  
 FilterCondition(self.data\_store).get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_filter\_condition\_value\_error(self):  
 *"""  
 Test suite for filter condition when user pass empty condition on filter* **:return***:  
 """* self.data\_store[**"conditions"**] = []  
 **try**:  
 FilterCondition(self.data\_store).get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_filter\_condition\_empty\_input(self):  
 *"""  
 Test suite for filter condition when user pass empty condition on filter* **:return***:  
 """* **try**:  
 FilterCondition(self.data\_store).get\_result(**None**)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
Test case module for filter  
"""***import** json  
**import** unittest  
**import** pandas **as** pd  
**from** filter.range\_filter **import** RangeFilter  
  
  
**class** TestRangerFilter(unittest.TestCase):  
 *"""  
 test suite for filter  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/deliveries.csv"**, delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_range\_filter\_in(self):  
 *"""  
 testcase for range filter with in* **:return***:  
 """* json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \"in\",\"is\_excluded\":false}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 7150)  
  
 json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \"in\",\"is\_excluded\":true}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 129448)  
  
 **def** test\_range\_filter\_greater\_than(self):  
 *"""  
 testcase for range filter with Greater than* **:return***:  
 """* json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \">\",\"is\_excluded\":false}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 100470)  
  
 json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \">\",\"is\_excluded\":true}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 36128)  
  
 **def** test\_range\_filter\_gt\_than\_equal(self):  
 *"""  
 testcase for range filter with Greater than equals* **:return***:  
 """* json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \">=\",\"is\_excluded\":false}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 107620)  
  
 json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \">=\",\"is\_excluded\":true}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 28978)  
  
 **def** test\_range\_filter\_less\_than(self):  
 *"""  
 testcase for range filter with less than* **:return***:  
 """* json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \"<\",\"is\_excluded\":false}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 28978)  
  
 json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \"<\",\"is\_excluded\":true}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 107620)  
  
 **def** test\_range\_filter\_lteq(self):  
 *"""  
 testcase for range filter with less than equals* **:return***:  
 """* json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \"<=\",\"is\_excluded\":false}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 36128)  
  
 json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \"<=\",\"is\_excluded\":true}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 100470)  
  
 **def** test\_range\_filter\_eq(self):  
 *"""  
 testcase for range filter with equal* **:return***:  
 """* json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \"=\",\"is\_excluded\":false}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 7150)  
  
 json\_request = **"{\"columnName\": \"over\", \"fromValue\": \"0\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"integer\", \"operator\": \"=\",\"is\_excluded\":true}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 129448)  
  
 **def** test\_range\_filter\_ne(self):  
 *"""  
 testcase for range filter with not equal* **:return***:  
 """* json\_request = **"{\"columnName\": \"over\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"float\", \"operator\": \"!=\",\"is\_excluded\":false}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 129448)  
  
 json\_request = **"{\"columnName\": \"over\", \"value\": "** \  
 **"\"5\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"float\", \"operator\": \"!=\",\"is\_excluded\":true}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[range\_filter.get\_result(self.data\_frame)]), 7150)  
  
 **def** test\_range\_filter\_date\_type(self):  
 *"""  
 This test suite will test equal filter with date data type* **:return***:  
 """* self.data\_frame\_new = pd.read\_csv(**"../resources/10Records.txt"**, delimiter=**"\t"**,  
 header=0, parse\_dates=[**'field0'**])  
 json\_request = **"{\"columnName\": \"field0\", \"value\": "** \  
 **"\"15-10-2017\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"datetime64[ns]\", \"operator\": \"=\",\"is\_excluded\":false}"** range\_filter = RangeFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame\_new[range\_filter.get\_result(self.data\_frame\_new)]), 3)  
  
 **def** test\_range\_filter\_with\_exception(self):  
 *"""  
 This test suite will raise an exception when field is missing in the data\_Frame* **:return***:  
 """* self.data\_frame\_new = pd.read\_csv(**"../resources/10Records.txt"**, delimiter=**"\t"**,  
 header=0, parse\_dates=[**'field0'**])  
 json\_request = **"{\"columnName\": \"field\_name\", \"value\": "** \  
 **"\"15-10-2017\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"datetime64[ns]\", \"operator\": \"=\",\"is\_excluded\":false}"  
 try**:  
 range\_filter = RangeFilter(json.loads(json\_request))  
 range\_filter.get\_result(self.data\_frame\_new)  
 self.assertEqual(1, 3)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_range\_filter\_with\_missing\_key(self):  
 *"""  
 This test suite will raise an exception whenever input json has missing required key.* **:return***:  
 """* self.data\_frame\_new = pd.read\_csv(**"../resources/10Records.txt"**, delimiter=**"\t"**,  
 header=0, parse\_dates=[**'field0'**])  
 json\_request = **"{\"columnName\": \"field\_name\", \"value\": "** \  
 **"\"15-10-2017\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"datetime64[ns]\", \"operator1\": \"=\",\"is\_excluded\":false}"  
 try**:  
 range\_filter = RangeFilter(json.loads(json\_request))  
 range\_filter.get\_result(self.data\_frame\_new)  
 self.assertEqual(1, 3)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_range\_filter\_with\_input\_is\_None(self):  
 *"""  
 This test suite will raise an exception whenever input dataframe is None.* **:return***:  
 """* self.data\_frame\_new = pd.read\_csv(**"../resources/10Records.txt"**, delimiter=**"\t"**,  
 header=0, parse\_dates=[**'field0'**])  
 json\_request = **"{\"columnName\": \"field\_name\", \"value\": "** \  
 **"\"15-10-2017\", \"record\_limit\": 10,"** \  
 **" \"dataSetName\": \"TestingDs\", \"column\_type\":"** \  
 **" \"datetime64[ns]\", \"operator\": \"=\",\"is\_excluded\":false}"  
 try**:  
 range\_filter = RangeFilter(json.loads(json\_request))  
 range\_filter.get\_result(**None**)  
 self.assertEqual(1, 3)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
This is a test case module on TextFilter  
"""***import** unittest  
**import** json  
**import** pandas **as** pd  
**from** filter.text\_filter **import** TextFilter  
  
  
**class** TestTextFilter(unittest.TestCase):  
 *"""  
 unit test case for text file  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/deliveries.csv"**, delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_getresult\_text\_filter\_in(self):  
 *"""  
 test case for text\_filter in* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"Kolkata Knight riders\","** \  
 **"\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"in\", \"case\": \"False\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15463)  
  
 json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"Chennai Super Kings\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"in\", \"case\": \"True\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15754)  
  
 **def** test\_getresult\_text\_filter\_qual(self):  
 *"""  
 test case for text\_filter* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"Kolkata Knight riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"equals\", \"case\": \"False\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15463)  
  
 json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"Chennai Super Kings\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"equals\", \"case\": \"True\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15754)  
  
 **def** test\_text\_filter\_for\_nq(self):  
 *"""  
 method is used for test case for textFilter of nq* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\","** \  
 **" \"value\": "** \  
 **"\"Kolkata Knight riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"!=\", \"case\": \"False\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 121135)  
  
 json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\","** \  
 **" \"value\": "** \  
 **"\"Chennai Super Kings\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"!=\", \"case\": \"True\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 120844)  
  
 **def** test\_text\_filter\_for\_startswith(self):  
 *"""  
 method is used for test case startswith textFilter* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"kolkata\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\",\"operator\":"** \  
 **" \"startswith\", \"case\": \"False\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15463)  
  
 json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\","** \  
 **" \"value\": "** \  
 **"\"Chennai Super Kings\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\",\"operator\":"** \  
 **" \"startswith\", \"case\": \"True\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15754)  
  
 **def** test\_text\_filter\_for\_end\_with(self):  
 *"""  
 method for test endWith testFilter* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\","** \  
 **" \"value\": "** \  
 **"\"riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\",\"operator\":"** \  
 **" \"endsWith\", \"case\": \"False\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15463)  
  
 json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"Kings\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\",\"operator\":"** \  
 **" \"endsWith\", \"case\": \"True\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15754)  
  
 **def** test\_text\_filter\_for\_contains(self):  
 *"""  
 method for contains* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\","** \  
 **" \"value\": "** \  
 **"\"riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\",\"operator\":"** \  
 **" \"contains\", \"case\": \"False\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 15463)  
  
 json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\","** \  
 **" \"value\": "** \  
 **"\"Kings\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\",\"operator\":"** \  
 **" \"contains\", \"case\": \"True\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 31745)  
  
 **def** test\_text\_filter\_for\_not\_contains(self):  
 *"""  
 method for not contains* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\","** \  
 **" \"value\": "** \  
 **"\"riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\",\"operator\":"** \  
 **" \"notcontains\", \"case\": \"False\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 121135)  
  
 json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\","** \  
 **" \"value\": "** \  
 **"\"Kings\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\",\"operator\":"** \  
 **" \"notcontains\", \"case\": \"True\",\"is\_excluded\": false}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 104853)  
  
 **def** test\_equal\_condition\_with\_exclude(self):  
 *"""  
 Test suite for compare exclude condition in equals* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"Kolkata Knight riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"equals\", \"case\": \"False\",\"is\_excluded\": true}"** text\_filter = TextFilter(json.loads(json\_request))  
 self.assertEqual(len(self.data\_frame[text\_filter.get\_result(self.data\_frame)]), 121135)  
  
 **def** test\_filter\_condition\_key\_value\_error(self):  
 *"""  
 Test suite for filter test condition with error message if applied filter key is not found.* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\_name\", "** \  
 **"\"value\": "** \  
 **"\"Kolkata Knight riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"equals\", \"case\": \"False\",\"is\_excluded\": true}"  
 try**:  
 text\_filter = TextFilter(json.loads(json\_request))  
 text\_filter.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_filter\_condition\_with\_exception(self):  
 *"""  
 Test suite for filter test condition with error message if* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"Kolkata Knight riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"equals\", \"case\": \"False\",\"is\_excluded\": true}"  
 try**:  
 text\_filter = TextFilter(json.loads(json\_request))  
 text\_filter.get\_result(**None**)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_filter\_json\_key\_missing(self):  
 *"""  
 Test suite will raise an exception when any of key is missing in the input json* **:return***:  
 """* json\_request = **"{\"className\" : \"text\_filter\",\"columnName1\": \"batting\_team\", "** \  
 **"\"value\": "** \  
 **"\"Kolkata Knight riders\",\"record\_limit\":30,\"dataSetName\":"** \  
 **"\"IplTrandingData\", \"type\": \"TEXT\_FILTER\","** \  
 **"\"operator\": \"equals\", \"case\": \"False\",\"is\_excluded\": true}"  
 try**:  
 text\_filter = TextFilter(json.loads(json\_request))  
 text\_filter.get\_result(**None**)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 print(ex)  
 self.assertEqual(1, 1)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
This module is used for string type data  
"""***import** constants  
**from** mashupbase **import** DataMashUpBase  
  
  
**class** TextFilter(DataMashUpBase): *# pylint: disable=too-few-public-methods  
 """  
 Text filter apply on string data  
 """* **def** \_\_init\_\_(self, json\_value):  
 *"""  
 initialize object with parameter* **:param** *json\_value:  
 """* **try**:  
 DataMashUpBase.\_\_init\_\_(self, json\_value)  
 self.value = json\_value[constants.VALUE]  
 self.case = json\_value[constants.CASE\_SENSITIVE\_STATUS]  
 self.operator = json\_value[constants.OPERATOR]  
 self.exclude = json\_value[constants.IS\_EXCLUDED]  
 **except** KeyError **as** key\_error:  
 **raise** Exception(key\_error.args[0] + **" is missing in the input json"**)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 take type of filter and return result dataSet* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 data\_frame = data\_frame.astype(str)  
 data\_frame\_new = **None  
 if** self.operator.lower() == **'equals'**:  
 data\_frame\_new = (data\_frame[self.column\_name].str.lower() == self.value.lower()) \  
 **if** self.case.lower() == **'false' else** \  
 (data\_frame[self.column\_name] == self.value)  
 **elif** self.operator.lower() == **'!='**:  
 data\_frame\_new = (data\_frame[self.column\_name].str.lower() != self.value.lower()) \  
 **if** self.case.lower() == **'false'** \  
 **else** (data\_frame[self.column\_name] != self.value)  
 **elif** self.operator.lower() == **'startswith'**:  
 data\_frame\_new = (data\_frame[self.column\_name].str.lower().str.startswith(  
 self.value.lower())) **if** \  
 self.case.lower() == **'false' else** \  
 (data\_frame[self.column\_name].str.startswith(self.value))  
 **elif** self.operator.lower() == **'endswith'**:  
 data\_frame\_new = (data\_frame[self.column\_name].str.lower().str.endswith(  
 self.value.lower())) **if** \  
 self.case.lower() == **'false' else** \  
 (data\_frame[self.column\_name].str.endswith(self.value))  
 **elif** self.operator.lower() == **'contains'**:  
 data\_frame\_new = (data\_frame[self.column\_name].str.contains(  
 self.value, case=**False**)) \  
 **if** self.case.lower() == **'false'** \  
 **else** (data\_frame[self.column\_name].str.contains(self.value))  
 **elif** self.operator.lower() == **'in'**:  
 self.value = str(self.value)  
 value\_list = self.value.split(**','**)  
 **if** self.case.lower() == **'false'**:  
 value\_list = map(str.lower, value\_list)  
 data\_frame\_new = data\_frame[self.column\_name].str.lower().isin(value\_list)  
 **else**:  
 data\_frame\_new = data\_frame[self.column\_name].isin(value\_list)  
 **else**:  
 data\_frame\_new = (~data\_frame[self.column\_name].str.contains(  
 self.value, case=**False**)) \  
 **if** self.case.lower() == **'false' else** \  
 (~data\_frame[self.column\_name].str.contains(self.value))  
  
 **if** self.exclude **is True**:  
 **return** ~data\_frame\_new  
 **return** data\_frame\_new  
 **except** KeyError **as** inst:  
 **raise** Exception(str(inst) + **" column not found"**)  
 **except** Exception **as** inst:  
 **raise** Exception(**"input data frame is None. "** + inst.args[0])

*find\_replace*

*"""  
This module is used to replace value with search value on data frame.  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** re  
**import** pandas **as** pd  
**import** constants  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** FindReplace:  
 *"""  
 This class contains the find and replace feature  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.search\_type = input\_json[constants.SEARCH\_TYPE]  
 self.column\_list = input\_json[constants.COLUMN\_LIST]  
 self.regex = input\_json[constants.REGEX\_STATUS]  
 self.case = input\_json.get(constants.CASE\_SENSITIVE\_STATUS, False)  
 self.replace\_with = input\_json[constants.REPLACE\_WITH]  
 self.find\_value = input\_json[constants.FIND\_VALUE]  
 self.exact\_match = input\_json.get(constants.EXACT\_MATCH, False)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame): *#pylint: disable-msg=too-many-branches  
 """  
 This method will perform the find and replace operation* **:param** *data\_frame:* **:return***:  
 """* **try**: *#pylint: disable-msg=R1702* **if** self.search\_type == **'column'**:  
 *# one by appling filter on column* **for** column\_detail **in** self.column\_list:  
 column\_name = column\_detail[constants.COLUMN\_NAME]  
 column\_type = column\_detail[constants.COLUMN\_TYPE]  
 **if** column\_type == constants.STRING:  
 self.find\_value = str(self.find\_value)  
 self.replace\_with = str(self.replace\_with)  
 *# find and replace for string column* **if** self.regex:  
 data\_frame[column\_name] = data\_frame[column\_name].str.replace(  
 self.find\_value, self.replace\_with, regex=self.regex,  
 case=self.case)  
 **elif** self.exact\_match:  
 pat = re.compile(**r"\b"** + self.find\_value + **r"\b"**, re.I)  
 **if** self.case:  
 pat = re.compile(**r"\b"**+self.find\_value+**r"\b"**)  
 data\_frame[column\_name] = data\_frame[column\_name].str.replace(  
 pat, self.replace\_with)  
 **else**:  
 data\_frame[column\_name] = data\_frame[column\_name].str.replace(  
 self.find\_value, self.replace\_with, case=self.case)  
 **elif** column\_type **in** [constants.INTEGER, constants.NUMBER]:  
 temp\_find\_val\_float = float(self.find\_value)  
 self.find\_value = int(self.find\_value)  
 **if** temp\_find\_val\_float != self.find\_value:  
 self.find\_value = temp\_find\_val\_float  
 temp\_replace\_val\_float = float(self.replace\_with)  
 self.replace\_with = int(self.replace\_with)  
 **if** temp\_replace\_val\_float != self.replace\_with:  
 self.replace\_with = temp\_replace\_val\_float  
 data\_frame[column\_name] = data\_frame[column\_name].replace(self.find\_value,  
 self.replace\_with)  
 **else**:  
 data\_frame[column\_name] = data\_frame[column\_name].replace(  
 FindReplace.convert\_datetime(column\_type, self.find\_value),  
 FindReplace.convert\_datetime(column\_type, self.replace\_with))  
 **else**:  
 orignal\_dataframe = data\_frame.copy()  
 data\_frame = data\_frame.astype(str)  
 self.find\_value = str(self.find\_value)  
 self.replace\_with = str(self.replace\_with)  
 *# if self.exact\_match:  
 # self.find\_value = re.compile(r"\S"+self.find\_value+r"\S")* **for** col **in** data\_frame.columns:  
 **try**:  
 *# Fix for find and replace issue - converting col to str and replacing* data\_frame[col] = data\_frame[col].str.replace(self.find\_value,  
 self.replace\_with)  
 data\_frame[col].astype(orignal\_dataframe[col].dtype)  
 orignal\_dataframe[col] = data\_frame[col]  
 **except** Exception:  
 **pass** data\_frame = orignal\_dataframe.copy()  
 **return** data\_frame  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** convert\_datetime(cls, data\_type, value):  
 *"""  
 This method is used to check whether value is datetime  
 or not based on the type it will convert the string value in datetime* **:return***:* **:param** *data\_type* **:param** *value  
 """* **if** data\_type == **"datetime"**:  
 **return** pd.to\_datetime(value)  
  
 **return** value  
  
 @classmethod  
 **def** convert\_to\_right\_type(cls, value):  
 *"""  
 This method is used for checking whether value is date or not.  
 If value of a string is in datetime format then it converts  
 and return to replace function.* **:param** *value:* **:return***:  
 """* **if** isinstance(value, (float, int)):  
 **return** value  
 **try**:  
 **return** pd.to\_datetime(value)  
 **except** ValueError:  
 **return** value

*"""  
This module is used to perform test suite of find and replace feature  
"""***import** unittest  
**import** pandas **as** pd  
**from** find\_replace **import** FindReplace  
  
  
**class** TestFindReplace(unittest.TestCase):  
 *"""  
 Class contains the test suite method of find and Replace feature  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/10Records.txt"**, delimiter=**'\t'**,  
 header=0, parse\_dates=[**'field0'**])  
  
 **def** test\_find\_and\_replace\_for\_string(self):  
 *"""  
 This test suite will test find and replace feature based on string column* **:return***:  
 """* input\_json = {**"search\_type"**: **"column"**, **"column\_list"**: [  
 {**"columnName"**: **"field6"**, **"column\_type"**: **"string"**}],  
 **"find\_value"**: **"modi"**, **"replace\_with"**: **"kumar"**,  
 **"case"**: False, **"regex"**: **"False"**}  
 find\_replace = FindReplace(input\_json)  
 new\_data\_frame = find\_replace.get\_result(self.data\_frame)  
 **if** new\_data\_frame[**'field6'**][0] == **'akshay kumar'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_and\_replace\_for\_exact\_string(self):  
 *"""  
 This test suite will test find and replace feature based on string column with exact same* **:return***:  
 """* input\_json = {**"search\_type"**: **"column"**, **"column\_list"**: [  
 {**"columnName"**: **"field6"**, **"column\_type"**: **"string"**}],  
 **"find\_value"**: **"modi"**, **"replace\_with"**: **"kumar"**,  
 **"exact\_match"**: True, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 new\_data\_frame = find\_replace.get\_result(self.data\_frame)  
 **if** new\_data\_frame[**'field6'**][0] == **'akshay kumar'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_and\_replace\_for\_number(self):  
 *"""  
 This test suite will test find and replace feature based on number* **:return***:  
 """* input\_json = {**"search\_type"**: **"column"**, **"column\_list"**: [  
 {**"columnName"**: **"field4"**, **"column\_type"**: **"number"**}],  
 **"find\_value"**: -37.90, **"replace\_with"**: 1001.1212,  
 **"case"**: False, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 new\_data\_frame = find\_replace.get\_result(self.data\_frame)  
 **if** new\_data\_frame[**'field4'**][0] == 1001.1212:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_and\_replace\_for\_number\_int(self):  
 *"""  
 This test suite will test find and replace feature based on number* **:return***:  
 """* input\_json = {**"search\_type"**: **"column"**, **"column\_list"**: [  
 {**"columnName"**: **"field1"**, **"column\_type"**: **"number"**}],  
 **"find\_value"**: 451211, **"replace\_with"**: 1001.1212,  
 **"case"**: False, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 new\_data\_frame = find\_replace.get\_result(self.data\_frame)  
 **if** new\_data\_frame[**'field1'**][1] == 1001.1212:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_and\_replace\_for\_date(self):  
 *"""  
 This test suite will test find and replace feature based on datetime* **:return***:  
 """* input\_json = {**"search\_type"**: **"column"**, **"column\_list"**: [  
 {**"columnName"**: **"field0"**, **"column\_type"**: **"datetime"**}],  
 **"find\_value"**: **"2017-10-15"**, **"replace\_with"**: **"2017-10-18"**,  
 **"case"**: False, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 new\_data\_frame = find\_replace.get\_result(self.data\_frame)  
 **if** new\_data\_frame[**'field0'**][0] == pd.to\_datetime(**'2017-10-18 00:00:00'**):  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_replace\_datetime\_whole\_dataset(self):  
 *"""  
 This test suite will test find and replace for all the column with datetime* **:return***:  
 """* input\_json = {**"search\_type"**: **"full"**, **"column\_list"**: [],  
 **"find\_value"**: **"2017-10-15"**, **"replace\_with"**: **"2017-10-18"**,  
 **"case"**: False, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 new\_data\_frame = find\_replace.get\_result(self.data\_frame)  
 *# if new\_data\_frame['field0'][0] == pd.to\_datetime('2017-10-18 00:00:00'):* **if** new\_data\_frame[**'field0'**][0] == **'2017-10-18'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_replace\_number\_whole\_dataset(self):  
 *"""  
 This test suite will test find and replace for all the column with datetime* **:return***:  
 """* input\_json = {**"search\_type"**: **"full"**, **"column\_list"**: [],  
 **"find\_value"**: -37.90, **"replace\_with"**: 11212,  
 **"case"**: False, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 new\_data\_frame = find\_replace.get\_result(self.data\_frame)  
 **if** new\_data\_frame[**'field4'**][0] == **'11212'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_replace\_string\_whole\_data\_set(self):  
 *"""  
 This test suite will test find and replace for all the column with datetime* **:return***:  
 """* input\_json = {**"search\_type"**: **"full"**, **"column\_list"**: [],  
 **"find\_value"**: **"akshay Modi"**, **"replace\_with"**: **"Aavesh Verma"**,  
 **"case"**: False, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 new\_data\_frame = find\_replace.get\_result(self.data\_frame)  
 **if** new\_data\_frame[**'field6'**][0] == **"Aavesh Verma"**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_find\_replace\_key\_error(self):  
 *"""  
 Test suite will raise key missing in input json* **:return***:  
 """* **try**:  
 input\_json = {**"column\_list"**: [],  
 **"find\_value"**: **"2017-10-15"**, **"replace\_with"**: **"2017-10-18"**,  
 **"case"**: False, **"regex"**: False}  
 FindReplace(input\_json)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_find\_replace\_column\_key\_missing(self):  
 *"""  
 Test suite for key missing in data frame* **:return***:  
 """* **try**:  
 input\_json = {**"search\_type"**: **"column"**, **"column\_list"**: [  
 {**"columnName"**: **"field0"**, **"column\_type"**: **"datetime"**}],  
 **"find\_value"**: **"2017-10-15"**, **"replace\_with"**: **"2017-10-18"**,  
 **"case"**: False, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 self.data\_frame = self.data\_frame.drop(**"field0"**, axis=1)  
 find\_replace.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_find\_replace\_missing\_data\_frame(self):  
 *"""  
 Test suite for key missing in data frame* **:return***:  
 """* **try**:  
 input\_json = {**"search\_type"**: **"column"**, **"column\_list"**: [  
 {**"columnName"**: **"field0"**, **"column\_type"**: **"datetime"**}],  
 **"find\_value"**: **"2017-10-15"**, **"replace\_with"**: **"2017-10-18"**,  
 **"case"**: False, **"regex"**: False}  
 find\_replace = FindReplace(input\_json)  
 find\_replace.get\_result(None)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_convert\_to\_right\_types(self):  
  
 self.assertEqual(FindReplace.convert\_to\_right\_type(10), 10)  
 self.assertEqual(FindReplace.convert\_to\_right\_type(**'bla'**),**'bla'**)  
  
 **def** test\_convert\_datetime(self):  
 self.assertEqual(FindReplace.convert\_datetime(**'int'**,10), 10)

*Frequent\_patterns*

*"""  
Finds market basket rules  
http://rasbt.github.io/mlxtend/user\_guide/frequent\_patterns/association\_rules/  
This code used mlxtend==0.12.0, please make sure we only get this version.  
Or else the code would not work  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** mlxtend.frequent\_patterns **import** apriori  
**from** mlxtend.frequent\_patterns **import** association\_rules  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** MarketBasket:  
 *"""  
 Args used are:  
 group\_list: Column names list on which groups would be prepared  
 item\_column: column name in list on which items will be grouped  
 """* **def** \_\_init\_\_(self, json\_value):  
 **try**:  
 self.group\_list = json\_value[**"group\_list"**]  
 self.item\_column = json\_value[**"item\_column"**]  
 self.min\_support = float(json\_value[**"min\_support"**]) / 100  
 self.run\_type = json\_value[**"run\_type"**]  
 self.col\_list = json\_value[**"col\_list"**]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** get\_patterns(patterns\_data\_frame, min\_support\_required):  
 *"""  
 This fucntion calcualtes frequent rules and then rules  
 """* frequent\_itemsets = apriori(  
 patterns\_data\_frame, min\_support=min\_support\_required, use\_colnames=True  
 )  
 *# gives value error when we dont get any frequent patterns* **if** frequent\_itemsets.shape[0] == 0:  
 **raise** ValueError(  
 message\_constants.NO\_PATTERNS\_FOUND  
 )  
 rules = association\_rules(frequent\_itemsets, metric=**"lift"**, min\_threshold=1)  
 **return** rules  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 Args:  
 data\_frame: data\_frame on which rules are created.  
 \*\*  
 This method is exposed into the api which selects the given columns  
 """* **try**:  
 **if** self.run\_type == **"row"**:  
 **if** (  
 set(self.group\_list) < set(self.item\_column)  
 **or** set(self.item\_column) < set(self.group\_list)  
 **or** set(self.item\_column) == set(self.group\_list)  
 ):  
 **raise** ValueError(  
 message\_constants.DIFFERENT\_GROUP\_LIST\_ITEM\_COLUMN\_VALUES  
 )  
 **elif not** set(self.item\_column) < set(  
 data\_frame.columns.tolist()  
 ) **or not** set(self.group\_list) < set(data\_frame.columns.tolist()):  
 **raise** ValueError(message\_constants.COLUMN\_NOT\_FOUND)  
 **else**:  
 *# below code converts the required columns into category for fast processing.* grouping\_list = self.group\_list + self.item\_column  
 **for** column\_group **in** grouping\_list:  
 **if** data\_frame.dtypes[column\_group].name == **"object"**:  
 data\_frame[column\_group] = data\_frame[column\_group].astype(  
 **"category"** )  
  
 df1 = data\_frame.groupby(grouping\_list).size().unstack().fillna(0)  
 out\_df = self.get\_patterns(df1, self.min\_support)  
 **elif** self.run\_type == **"column"**:  
 **if not** set(self.col\_list) < set(data\_frame.columns.tolist()):  
 **raise** ValueError(message\_constants.COLUMN\_NOT\_FOUND)  
 **elif** self.min\_support <= 0 **or** self.min\_support > 1:  
 **raise** ValueError(message\_constants.INVALID\_SUPPORT\_VALUE)  
 **else**:  
 data\_frame = data\_frame[self.col\_list]  
 data\_frame = data\_frame.astype(**"bool"**)  
 out\_df = self.get\_patterns(data\_frame, self.min\_support)  
 *# by default sort the output by Confidence value* out\_df.sort\_values([**"confidence"**], ascending=[False], inplace=True)  
 *# Show only top 1000 records* out\_df = out\_df.head(1000)  
 out\_df[**'conviction'**] = out\_df[**'conviction'**].astype(str)  
 **return** out\_df  
 **except** MemoryError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INSUFFICIENT\_MEMORY\_FOR\_PERFORMING\_OPERATION)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to write a test case for frequent patterns  
"""***import** unittest  
**import** pandas **as** pd  
**from** frequent\_patterns **import** MarketBasket  
  
  
**class** TestFrequentPatterns(unittest.TestCase):  
 *"""  
 This class is used to write test method for frequent patterns  
 """* **def** setUp(self):  
 **pass  
  
 def** test\_row\_good\_flow(self):  
 *"""  
 Test suite for performing keep operation* **:return***:  
 """* sales\_df = {  
 **"Invoice\_No"**: [  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536366,  
 536366,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 ],  
 **"Description"**: [  
 **"WHITE HANGING HEART T-LIGHT HOLDER"**,  
 **"WHITE METAL LANTERN"**,  
 **"CREAM CUPID HEARTS COAT HANGER"**,  
 **"KNITTED UNION FLAG HOT WATER BOTTLE"**,  
 **"RED WOOLLY HOTTIE WHITE HEART."**,  
 **"SET 7 BABUSHKA NESTING BOXES"**,  
 **"GLASS STAR FROSTED T-LIGHT HOLDER"**,  
 **"HAND WARMER UNION JACK"**,  
 **"HAND WARMER RED POLKA DOT"**,  
 **"ASSORTED COLOUR BIRD ORNAMENT"**,  
 **"POPPY'S PLAYHOUSE BEDROOM"**,  
 **"POPPY'S PLAYHOUSE KITCHEN"**,  
 **"FELTCRAFT PRINCESS CHARLOTTE DOLL"**,  
 **"IVORY KNITTED MUG COSY"**,  
 **"BOX OF 6 ASSORTED COLOUR TEASPOONS"**,  
 **"BOX OF VINTAGE JIGSAW BLOCKS"**,  
 **"BOX OF VINTAGE ALPHABET BLOCKS"**,  
 **"HOME BUILDING BLOCK WORD"**,  
 **"LOVE BUILDING BLOCK WORD"**,  
 **"RECIPE BOX WITH METAL HEART"**,  
 ],  
 }  
 df\_test = pd.DataFrame.from\_dict(sales\_df)  
  
 sample\_dict = {}  
 group\_list = [**"Invoice\_No"**]  
 item\_column = [**"Description"**]  
 min\_support = 0.02  
 col\_list = [**"sample"**]  
  
 sample\_dict[**"group\_list"**] = group\_list  
 sample\_dict[**"item\_column"**] = item\_column  
 sample\_dict[**"min\_support"**] = min\_support  
 sample\_dict[**"run\_type"**] = **"row"** sample\_dict[**"col\_list"**] = col\_list  
  
 class\_mb = MarketBasket(sample\_dict)  
  
 out\_df\_mb = class\_mb.get\_result(df\_test)  
  
 **if** out\_df\_mb.shape[0] == 1000:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_frequent\_patterns\_bad\_flow(self):  
 *"""  
 Test suite for performing keep operation* **:return***:  
 """* sales\_df = {  
 **"Invoice\_No"**: [  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536366,  
 536366,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 ],  
 **"Description"**: [  
 **"WHITE HANGING HEART T-LIGHT HOLDER"**,  
 **"WHITE METAL LANTERN"**,  
 **"CREAM CUPID HEARTS COAT HANGER"**,  
 **"KNITTED UNION FLAG HOT WATER BOTTLE"**,  
 **"RED WOOLLY HOTTIE WHITE HEART."**,  
 **"SET 7 BABUSHKA NESTING BOXES"**,  
 **"GLASS STAR FROSTED T-LIGHT HOLDER"**,  
 **"HAND WARMER UNION JACK"**,  
 **"HAND WARMER RED POLKA DOT"**,  
 **"ASSORTED COLOUR BIRD ORNAMENT"**,  
 **"POPPY'S PLAYHOUSE BEDROOM"**,  
 **"POPPY'S PLAYHOUSE KITCHEN"**,  
 **"FELTCRAFT PRINCESS CHARLOTTE DOLL"**,  
 **"IVORY KNITTED MUG COSY"**,  
 **"BOX OF 6 ASSORTED COLOUR TEASPOONS"**,  
 **"BOX OF VINTAGE JIGSAW BLOCKS"**,  
 **"BOX OF VINTAGE ALPHABET BLOCKS"**,  
 **"HOME BUILDING BLOCK WORD"**,  
 **"LOVE BUILDING BLOCK WORD"**,  
 **"RECIPE BOX WITH METAL HEART"**,  
 ],  
 }  
 df\_test = pd.DataFrame.from\_dict(sales\_df)  
  
 *# code used for testing with multiple dataframes* sample\_dict = {}  
 group\_list = [**"InvoiceNo"**]  
 item\_column = [**"Description"**]  
 min\_support = 0.02  
 col\_list = [**"sample"**]  
  
 sample\_dict[**"group\_list"**] = group\_list  
 sample\_dict[**"item\_column"**] = item\_column  
 sample\_dict[**"min\_support"**] = min\_support  
 sample\_dict[**"run\_type"**] = **"row"** sample\_dict[**"col\_list"**] = col\_list  
  
 class\_mb = MarketBasket(sample\_dict)  
 **try**:  
 out = class\_mb.get\_result(df\_test)  
 out.head()  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 self.assertEqual(1, 1)  
 **print**(ex)  
  
 **def** test\_bad\_flow\_2(self):  
 *"""  
 Test suite for performing keep operation* **:return***:  
 """* sales\_df = {  
 **"Invoice\_No"**: [  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536365,  
 536366,  
 536366,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 536367,  
 ],  
 **"Description"**: [  
 **"WHITE HANGING HEART T-LIGHT HOLDER"**,  
 **"WHITE METAL LANTERN"**,  
 **"CREAM CUPID HEARTS COAT HANGER"**,  
 **"KNITTED UNION FLAG HOT WATER BOTTLE"**,  
 **"RED WOOLLY HOTTIE WHITE HEART."**,  
 **"SET 7 BABUSHKA NESTING BOXES"**,  
 **"GLASS STAR FROSTED T-LIGHT HOLDER"**,  
 **"HAND WARMER UNION JACK"**,  
 **"HAND WARMER RED POLKA DOT"**,  
 **"ASSORTED COLOUR BIRD ORNAMENT"**,  
 **"POPPY'S PLAYHOUSE BEDROOM"**,  
 **"POPPY'S PLAYHOUSE KITCHEN"**,  
 **"FELTCRAFT PRINCESS CHARLOTTE DOLL"**,  
 **"IVORY KNITTED MUG COSY"**,  
 **"BOX OF 6 ASSORTED COLOUR TEASPOONS"**,  
 **"BOX OF VINTAGE JIGSAW BLOCKS"**,  
 **"BOX OF VINTAGE ALPHABET BLOCKS"**,  
 **"HOME BUILDING BLOCK WORD"**,  
 **"LOVE BUILDING BLOCK WORD"**,  
 **"RECIPE BOX WITH METAL HEART"**,  
 ],  
 }  
 df\_test = pd.DataFrame.from\_dict(sales\_df)  
 sample\_dict = {}  
 group\_list = [**"Invoice\_No"**]  
 item\_column = [**"Invoice\_No"**]  
 min\_support = 0.02  
 col\_list = [**"sample"**]  
  
 sample\_dict[**"group\_list"**] = group\_list  
 sample\_dict[**"item\_column"**] = item\_column  
 sample\_dict[**"min\_support"**] = min\_support  
 sample\_dict[**"run\_type"**] = **"row"** sample\_dict[**"col\_list"**] = col\_list  
  
 class\_mb = MarketBasket(sample\_dict)  
 **try**:  
 out = class\_mb.get\_result(df\_test)  
 out.head()  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 self.assertEqual(1, 1)  
 **print**(ex)  
  
 **with** self.assertRaises(Exception):  
 MarketBasket({})  
  
 **def** test\_col\_data\_good\_flow(self):  
 *"""  
 Test suite for performing keep operation* **:return***:  
 """* sales\_df = {  
 **"date"**: [  
 **"1/1/2019"**,  
 **"2/1/2019"**,  
 **"3/1/2019"**,  
 **"4/1/2019"**,  
 **"5/1/2019"**,  
 **"6/1/2019"**,  
 **"7/1/2019"**,  
 **"8/1/2019"**,  
 **"9/1/2019"**,  
 **"10/1/2019"**,  
 **"11/1/2019"**,  
 **"12/1/2019"**,  
 **"13/1/2019"**,  
 **"14/2/2019"**,  
 **"15/1/2019"**,  
 **"16/3/2019"**,  
 **"17/1/2019"**,  
 **"18/4/2019"**,  
 **"19/1/2019"**,  
 **"20/1/2019"**,  
 ],  
 **"Walking"**: [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  
 **"Podcast"**: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  
 **"Reading"**: [1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0],  
 **"Cooking"**: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0],  
 **"Reports"**: [  
 12.0,  
 1.0,  
 1.0,  
 0,  
 0.0,  
 1.0,  
 1.0,  
 1.0,  
 1.0,  
 1.0,  
 0.0,  
 1.0,  
 1.0,  
 1.0,  
 1.0,  
 1.0,  
 1.0,  
 1.0,  
 1.0,  
 0.0,  
 ],  
 **"Coding"**: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],  
 }  
 df\_test = pd.DataFrame.from\_dict(sales\_df)  
 sample\_dict = {}  
 group\_list = [**"Walking"**]  
 item\_column = [**"Podcast"**]  
 min\_support = 0.6  
 col\_list = [**"Walking"**, **"Podcast"**, **"Reading"**, **"Cooking"**, **"Reports"**, **"Coding"**]  
  
 sample\_dict[**"group\_list"**] = group\_list  
 sample\_dict[**"item\_column"**] = item\_column  
 sample\_dict[**"min\_support"**] = min\_support  
 sample\_dict[**"run\_type"**] = **"column"** sample\_dict[**"col\_list"**] = col\_list  
  
 class\_mb = MarketBasket(sample\_dict)  
  
 out\_df\_mb = class\_mb.get\_result(df\_test)  
  
 **if** out\_df\_mb.shape[0] > 100:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 unittest.main()

*Group\_by*

*"""  
This module is used to perform group by operation with the following functions:  
max  
min  
sum  
count  
mean  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** constants  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** GroupStep(object):  
 *"""  
 This class contain the logic of group by step  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.input\_json = input\_json  
 self.merge\_output = input\_json[constants.MERGE\_OUTPUT]  
 self.group\_by\_fields = input\_json[constants.GROUP\_BY\_FIELDS]  
 self.group\_by\_func = {}  
 **for** value **in** input\_json[constants.GROUP\_FUNCTION]:  
 self.group\_by\_func.update({value[constants.FIELD\_NAME]:  
 value[constants.AGG\_FUNCTION\_ARRAY]})  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** group\_by(self, data\_frame):  
  
 new\_data\_frame = data\_frame.groupby(self.group\_by\_fields, as\_index=False) \  
 .agg(self.group\_by\_func)  
 *# renaming column which has a space in its name* new\_data\_frame.columns = [(c[0] + (c[1] **if** c[1] == **''  
 else '\_'** + c[1])).replace(**' '**, **'\_'**)  
 **for** c **in** new\_data\_frame.columns]  
 **if** self.merge\_output **is** True:  
 **return** data\_frame.merge(new\_data\_frame, how=**'left'**,  
 left\_on=self.group\_by\_fields,  
 right\_on=self.group\_by\_fields)  
 **return** new\_data\_frame  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method will perform the group by operation based on input* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **return** self.group\_by(data\_frame)  
 **except** ValueError:  
 **try**:  
 *# renaming columns : append an 'underscore' symbol and trying again* **for** value **in** self.input\_json[constants.GROUP\_FUNCTION]:  
 new\_expression = {value[constants.FIELD\_NAME]: {  
 value[constants.FIELD\_NAME] + **'\_'**: value[  
 constants.AGG\_FUNCTION\_ARRAY]}}  
 self.group\_by\_func.update(new\_expression)  
  
 **return** self.group\_by(data\_frame)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used for containing test case of group by feature  
"""***import** unittest  
**import** pandas **as** pd  
**from** group\_by.group\_by\_step **import** GroupStep  
  
  
**class** TestGroupStep(unittest.TestCase):  
 *"""  
 Test class of group by step  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, header=0,  
 delimiter=**'\t'**, parse\_dates=[**'field0'**])  
 self.input\_json = \  
 {**"group\_by"**: [**'field0'**], **"group\_function"**: [  
 {**"AggFunctionArray"**: [**"count"**, **"sum"**], **"fieldName"**: **"field4"**},  
 {**"AggFunctionArray"**: [**"sum"**], **"fieldName"**: **"field2"**}],  
 **"merge\_output"**: False}  
  
 **def** test\_get\_results\_same\_col(self):  
 *"""  
 This test performs groupby on fields2 field3 , and  
 aggregate function count(field3), sum(field2)* **:return***:  
 """* input\_json\_same\_col = \  
 {**"group\_by"**: [**"field0"**],  
 **"group\_function"**: [{**"AggFunctionArray"**: [**"count"**],  
 **"fieldName"**: **"field0"**}],  
 **"merge\_output"**: **"false"**,  
 **"step\_name"**: **"group\_by.group\_by\_step.GroupStep"**,  
 **"displayDataHeader"**: **"GroupBy"**,  
 **"displayDataText"**: **"Group By count field0 , field0"**,  
 **"is\_enabled"**: **"true"**,  
 **"is\_edited"**: **"true"**}  
 group\_step = GroupStep(input\_json\_same\_col)  
 new\_data\_frame = group\_step.get\_result(self.data\_frame)  
 **if 'field0\_\_count' in** new\_data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_results\_(self):  
 *"""  
 This test performs groupby on fields2 field3 , and  
 aggregate function count(field3), sum(field2)* **:return***:  
 """* json = {**"group\_by"**: [**"field2"**, **"field3"**],  
 **"group\_function"**: [  
 {**"AggFunctionArray"**: [**"count"**], **"fieldName"**: **"field2"**},  
 {**"AggFunctionArray"**: [**"count"**], **"fieldName"**: **"field3"**}],  
 **"merge\_output"**: **'false'**, **"step\_name"**: **"group\_by.group\_by\_step.GroupStep"**,  
 **"displayDataHeader"**: **"GroupBy"**, **"displayDataText"**:  
 **"Group By count field2 , count field3 ,field2,field3"**,  
 **"is\_edited"**: **'true'**,  
 **"is\_enabled"**: **'true'**}  
 group\_step = GroupStep(json)  
 new\_data\_frame = group\_step.get\_result(self.data\_frame)  
 **if 'field2\_\_count' and 'field3\_\_count' in** new\_data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_results\_fail(self):  
 *"""  
 This test performs groupby on fields2 field3 , and  
 aggregate function count(field3), sum(field2)* **:return***:  
 """* json = self.json\_multi\_fail = \  
 {**"group\_by"**: [**"field2"**],  
 **"group\_function"**: [{**"AggFunctionArray"**: [**"count"**],  
 **"fieldName"**: **"field2"**},  
 {**"AggFunctionArray"**: [**"min"**],  
 **"fieldName"**: **"field4"**}],  
 **"merge\_output"**: False,  
 **"step\_name"**: **"group\_by.group\_by\_step.GroupStep"**,  
 **"displayDataHeader"**: **"GroupBy"**,  
 **"displayDataText"**: **"Group By count field2 , min field4 , field2"**,  
 **"is\_enabled"**: **'true'**,  
 **"is\_edited"**: **'true'**}  
  
 group\_step = GroupStep(json)  
 new\_data\_frame = group\_step.get\_result(self.data\_frame)  
 **if 'field2\_\_count' and 'field4\_\_min' in** new\_data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_merge\_output(self):  
 *"""  
 Test suite for testing group by functionality with merge\_output* **:return***:  
 """* self.input\_json.update({**"merge\_output"**: True})  
 group\_step = GroupStep(self.input\_json)  
 new\_data\_frame = group\_step.get\_result(self.data\_frame)  
 **if 'field3' in** new\_data\_frame.columns:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_key\_error(self):  
 *"""  
 Test suite for testing group by functionality with merge\_output* **:return***:  
 """* **try**:  
 self.input\_json.pop(**"merge\_output"**)  
 GroupStep(self.input\_json)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_get\_result\_data\_frame\_none(self):  
 *"""  
 Test suite for testing group by functionality with merge\_output* **:return***:  
 """* **try**:  
 self.input\_json.update({**"merge\_output"**: True})  
 group\_step = GroupStep(self.input\_json)  
 group\_step.get\_result(None)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args[0])  
 self.assertEqual(1, 1)

*Insights*

*Auto\_insights*

*"""  
This module is used to generate auto insight based on given dataInsight.  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** insights.time\_series.test\_time\_series\_forecasting **import** TimeSeriesForecast  
**from** insights.what\_if\_analysis.what\_if\_analysis **import** WhatIfAnalysis  
**from** insights.cluster.cluster **import** Cluster  
**from** insights.utilities **import** get\_cols\_and\_types  
**import** constants  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** request\_builder **import** RequestBuilder  
**from** token\_registry **import** TokenRegistry  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** AutoInsight:  
 *"""  
 This class is used to generate auto insight based on data source  
 """* symbolic\_dict = {**"M"**: **"months"**, **"Q"**: **"quarters"**, **"A"**: **"years"**,  
 **"H"**: **"hours"**, **"W"**: **"weeks"**, **"D"**: **"days"**}  
  
 **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.data\_source\_id = input\_json[constants.DATA\_SOURCE\_ID]  
 self.user\_id = input\_json[constants.USER\_ID]  
 self.lens\_id = DataRegistry.convert\_datetime\_into\_ms()  
 self.know\_ledge\_base = input\_json[**"knowledgebase"**]  
 self.data\_source\_name = input\_json[**"data\_source\_name"**]  
  
 **try**:  
 self.know\_ledge\_base = RequestBuilder().call\_http\_get\_request(  
 constants.FETCH\_KNOWLEDGE\_BASE + str(self.data\_source\_id), {},  
 TokenRegistry.get\_token\_for\_user(user\_id=self.user\_id)).json()  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** None  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** run\_insights\_on\_data\_source(self):  
 *"""  
 This method will used to generate insight given on data source* **:return***:  
 """* **try**:  
 data\_frame = DataRegistry().create\_data\_frame\_using\_data\_source\_id(  
 data\_source\_id=self.data\_source\_id,  
 load\_all\_flag=False,  
 user\_id=self.user\_id)  
  
 DataRegistry.register\_data\_frame(self.data\_source\_id, self.lens\_id, data\_frame)  
  
 auto\_insights\_output = {constants.DATA\_SOURCE\_ID: self.data\_source\_id,  
 constants.AUTO\_INSIGHTS: []}  
  
 **for** \_, value **in** enumerate(self.generate\_what\_if\_analysis\_text(data\_frame)):  
 auto\_insights\_output[constants.AUTO\_INSIGHTS].append(value)  
  
 **for** \_, value **in** enumerate(self.generate\_time\_series\_text()):  
 auto\_insights\_output[constants.AUTO\_INSIGHTS].append(value)  
  
 **if** self.generate\_clustering\_text() != {}:  
 auto\_insights\_output[constants.AUTO\_INSIGHTS].append(  
 self.generate\_clustering\_text())  
 **return** auto\_insights\_output **if** auto\_insights\_output[constants.AUTO\_INSIGHTS] **else** {}  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** {}  
  
 **def** find\_time\_series\_column(self):  
 *"""  
 This method is used to find time series column from know\_ledge base json* **:return***:  
 """* **try**:  
 ts\_date\_list = []  
 ts\_kpi\_list = []  
 ts\_freq\_list = []  
 ts\_sentiment\_list = []  
  
 **for** value **in** self.know\_ledge\_base:  
 **if** value[constants.TIME\_SERIES\_DATE\_ATTRIBUTE] == **'Y'**:  
 ts\_date\_list.append(value[constants.COLUMN\_NAME])  
 ts\_freq\_list.append(value[constants.TIME\_SERIES\_FREQUENCY])  
  
 **if** value[constants.TIME\_SERIES\_KPI] == **'Y'**:  
 ts\_kpi\_list.append(value[constants.COLUMN\_NAME])  
 ts\_sentiment\_list.append(value[constants.SENTIMENT])  
 **return** ts\_date\_list, ts\_kpi\_list, ts\_freq\_list, ts\_sentiment\_list  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** time\_series\_text\_generation(cls, fore\_casted\_kpi\_sum, prev\_kpi\_sum,  
 sentiment, time\_series\_json):  
 *"""  
 this method is used to generate text for time series forecasting* **:param** *fore\_casted\_kpi\_sum:* **:param** *prev\_kpi\_sum:* **:param** *sentiment:* **:param** *time\_series\_json* **:return***:  
 """* **try**:  
 return\_msg = **""  
 try**:  
 diff\_perc = round((fore\_casted\_kpi\_sum - prev\_kpi\_sum) / abs(prev\_kpi\_sum) \* 100, 2)  
 **except** ZeroDivisionError:  
 diff\_perc = 0  
 **if** diff\_perc > 0 **and** sentiment.upper() == **"Y"**:  
 return\_msg = **"<b>Increase of "  
 elif** diff\_perc > 0 **and** sentiment.upper() == **"N"**:  
 return\_msg = **"<b>Decrease of "  
 elif** diff\_perc < 0 **and** sentiment.upper() == **"N"**:  
 return\_msg = **"<b>Increase of "  
 elif** diff\_perc < 0 **and** sentiment.upper() == **"Y"**:  
 return\_msg = **"<b>Decrease of "  
  
 if** return\_msg:  
 **return** return\_msg + str(abs(diff\_perc)) + **"% in "** +\  
 str(time\_series\_json[constants.FORECAST\_COLUMN]) +\  
 **"</b> expected in next <b>6 "** \  
 + AutoInsight.convert\_frequency(str(time\_series\_json[constants.FORECAST\_BIN]  
 )) + str(**"</b>"**)  
  
 **return** return\_msg  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** convert\_frequency(cls, symbolic\_freq):  
 *"""  
 This method is used to convert symbolic freq to Actual Frequency  
 like 'M' to Month  
 'Q' to Quarter* **:return***:  
 """* **try**:  
 **return** AutoInsight.symbolic\_dict.get(symbolic\_freq)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** generate\_time\_series\_text(self):  
 *"""  
 This method is used to fetch knowledge base data from json* **:return***:  
 """* **try**:  
 time\_series\_forecast = TimeSeriesForecast(self.data\_source\_id, self.lens\_id)  
 ts\_date\_list, ts\_kpi\_list, ts\_freq\_list, ts\_sentiment\_list =\  
 self.find\_time\_series\_column()  
 time\_series\_final\_list = []  
 **for** i, ts\_date **in** enumerate(ts\_date\_list):  
 **for** j, ts\_kpi **in** enumerate(ts\_kpi\_list):  
 \_, kpi, \_, \_ = time\_series\_forecast.time\_series\_forecast(ts\_date, ts\_kpi,  
 ts\_freq\_list[i])  
  
 time\_series\_json = {**"type"**: **"Time Series Forecasting"**,  
 **"time\_column"**: ts\_date,  
 **"forecast\_column"**: ts\_kpi,  
 **"forecast\_bin"**: ts\_freq\_list[i], **"createdBy"**: self.user\_id,  
 **"datasourceName"**: self.data\_source\_name}  
  
 **if** len(kpi) >= 12:  
 *# checks 6 forecasted data points against last 6 data points and  
 # calculate % change* time\_series\_json.update(  
 {**"text"**: str(AutoInsight.time\_series\_text\_generation(  
 sum(kpi[-6:]), sum(kpi[-12:-6]), ts\_sentiment\_list[j],  
 time\_series\_json))})  
 **else**:  
 time\_series\_json.update({**"text"**: AutoInsight.time\_series\_text\_generation(  
 sum(kpi[-6: len(kpi) - 12]), sum(kpi[-len(kpi):-6]),  
 ts\_sentiment\_list[j], time\_series\_json)})  
 time\_series\_final\_list.append(time\_series\_json)  
 **return** time\_series\_final\_list  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** []  
  
 **def** generate\_clustering\_text(self):  
 *"""  
 This method is used to run clustering algorithm* **:return***:  
 """* **try**:  
  
 clustering\_json = {**"type"**: **"Data Segmentation"**, **"createdBy"**: self.user\_id,  
 **"datasourceName"**: self.data\_source\_name}  
  
 text = None  
 **try**:  
 cal\_cluster = Cluster(self.data\_source\_id, self.lens\_id)  
 text = **"<b>"** + str(cal\_cluster.clustering()[constants.TOTAL\_CLUSTERS])\  
 + **" distinct segments</b> found in your data set"  
 except** Exception **as** ex:  
 text = ex.args[0]  
 **if 'not suitable' in** text:  
 **return** {}  
 clustering\_json.update({**"text"**: text})  
 **return** clustering\_json  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** {}  
  
 **def** find\_what\_if\_analysis\_column(self):  
 *"""  
 This method is used to find out target column to run whatIfAnalysis* **:return***:  
 """* target\_list = []  
 **try**:  
 **for** value **in** self.know\_ledge\_base:  
 **if** value[constants.TARGET] == **'Y'**:  
 target\_list.append(value[constants.COLUMN\_NAME])  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** target\_list  
  
 **def** generate\_what\_if\_analysis\_text(self, data\_frame):  
 *"""  
 This method is used to generate result of whatIfAnalysis* **:return***:  
 """* **try**:  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(self.data\_source\_id,  
 self.lens\_id).\  
 return\_cols\_and\_types()  
  
 factor\_col\_list = cols\_json\_file[constants.FACTOR\_COLS]  
 data\_frame = data\_frame.drop(cols\_json\_file[constants.ID\_COLS], axis=1)  
 data\_frame = data\_frame.drop(cols\_json\_file[constants.DATE\_COLS], axis=1)  
  
 what\_if\_analysis = WhatIfAnalysis(data\_frame)  
 \_, data = what\_if\_analysis.label\_encoding(factor\_col\_list)  
 what\_if\_final\_list = []  
 **for** \_, value **in** enumerate(self.find\_what\_if\_analysis\_column()):  
 what\_if\_analysis = WhatIfAnalysis(data)  
 features, \_ = what\_if\_analysis.imp\_feature([value])  
 output\_json = what\_if\_analysis.linear\_regression([value], factor\_col\_list, features)  
  
 what\_if\_json = {**"type"**: **"What IF Analysis"**, **"createdBy"**: self.user\_id,  
 **"datasourceName"**: self.data\_source\_name}  
  
 text = None  
 **for** key, \_ **in** output\_json.items():  
 **if** key != **'model\_accuracy'**:  
 what\_if\_json.update({**"target\_column"**: key})  
 text = **"<b>"** + str(len(output\_json[key][0][**"cols"**])-2) \  
 + **" key factors</b> found to affect <b>"** + str(key) + str(**"</b>"**)  
 what\_if\_json.update({**"text"**: text})  
 what\_if\_final\_list.append(what\_if\_json)  
 **return** what\_if\_final\_list  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** []

*"""Sample Module Containing methods that can be run in celery"""***import** celery  
**from** requests **import** post  
**import** constants  
**from** token\_registry **import** TokenRegistry  
**from** request\_builder **import** RequestBuilder  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** insights.utilities.insights\_utilities **import** InsightUtils  
**from** insights.auto\_insights.magic **import** do\_magic  
**from** celery.contrib **import** rdb  
  
  
**class** CeleryTasks:  
 *"""Sample Module Containing methods that can be run in celery"""* @celery.task(name=**'celery\_tasks.sleepfunc'**)  
 **def** sleepfunc(input\_json):  
 *# Sample function that gets called* **try**:  
 *#rdb.set\_trace() # <- set break-point* user\_id = input\_json[**"user\_id"**]  
 bearer\_token = input\_json[**"token"**]  
 TokenRegistry.register\_token\_with\_user\_id(user\_id=user\_id, access\_token=bearer\_token)  
  
 *# Generate json for func* make\_json = InsightUtils.generate\_lists(input\_json)  
  
 *# get data from hive* data\_source\_id = input\_json[**"data\_source\_id"**]  
 data\_source\_name = input\_json[**"data\_source\_name"**]  
 data\_frame = DataRegistry().create\_data\_frame\_using\_data\_source\_id(  
 data\_source\_id=data\_source\_id, load\_all\_flag=True, user\_id=user\_id)  
  
 **print**(len(data\_frame))  
  
 *# Do magic* final\_list = do\_magic(data=data\_frame, date\_column\_function=make\_json[**"date\_column"**],  
 drill\_down\_columns=make\_json[**"drilldown\_columns\_list"**],  
 date\_granularity=make\_json[**"defaultFrequency"**],  
 kpi\_columns\_list=make\_json[**"kpi\_colums\_list"**],  
 data\_source\_id=data\_source\_id, user\_id=user\_id,data\_source\_name=data\_source\_name)  
  
 authorization\_header = TokenRegistry.get\_token\_for\_user(user\_id=user\_id)  
  
 RequestBuilder().call\_http\_post\_request(url=constants.INSERT\_KNOWLEDGE\_BASE\_GRAPH,  
 data=final\_list,  
 headers=authorization\_header)  
  
 url = **'http://127.0.0.1:8090/mashup-service/send'** notify\_json = {}  
 notify\_json[**"data\_source\_name"**] = data\_source\_name  
 notify\_json[**"status"**] = **"Success"** notify\_json[**"user\_id"**] =user\_id  
 notify\_json[**"data\_source\_id"**] = data\_source\_id  
 post(url, json=notify\_json)  
 **return "Success celery task"  
 except** Exception **as** ex:  
 **print**(ex)

*"""This module is design to perform all knowledge base operations"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** constants  
**import** message\_constants  
**import** time  
**import** json  
**from** request\_builder **import** RequestBuilder  
**from** token\_registry **import** TokenRegistry  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** KnowledgeBase:  
 *"""This class is design to perform all knowledge base operations"""* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.data\_source\_id = input\_json[constants.DATA\_SOURCE\_ID]  
 self.user\_id = input\_json[constants.USER\_ID]  
 self.column\_name = input\_json[constants.COLUMN\_NAME]  
 self.data\_source\_name = input\_json[constants.DATASET\_NAME]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** feedback\_update\_knowledge\_base(self):  
 *"""This function updates users entry in knowledge base for Auto Insights  
 ex.If users selects Revenue column for what if analysis, then it stores it in Knowledge  
 base and auto insights can be populated for this feature in future"""* **try**:  
 authorization\_header = TokenRegistry.get\_token\_for\_user(user\_id=self.user\_id)  
 insert\_data = {**"dataSourceId"**: self.data\_source\_id,  
 **"dataSourceName"**: self.data\_source\_name,  
 **"kbInput"**: []}  
 **for** \_, column\_name **in** enumerate(self.column\_name):  
 url = constants.FETCH\_KB\_BY\_SOURCEID\_COLUMN + str(self.data\_source\_id) \  
 + str(**"&columnName="**) + str(column\_name)  
 **try**:  
 knowledge\_base\_dict = None  
 knowledge\_base\_dict =\  
 RequestBuilder().call\_http\_get\_request(url=url,  
 data={},  
 headers=authorization\_header).json()  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 insert\_data[**"kbInput"**].append({**"columnName"**: column\_name, **"sentiment"**: **"Y"**,  
 **"timeSeriesDateAttribute"**: **"N"**,  
 **"timeSeriesKpi"**: **"N"**, **"target"**: **"Y"**,  
 **"normalyViewedWith"**: **"{}"**, **"index"**: **"N"**,  
 **"defaultFrequency"**: **"None"**,  
 **"relationship"**: []})  
 **try**:  
 RequestBuilder().call\_http\_post\_request(url=  
 constants.INSERT\_IN\_KNOWLEDGE\_BASE,  
 data=insert\_data,  
 headers=authorization\_header)  
 **except** Exception **as** jv\_ex:  
 LOGGER.error(**'Exception: Java Insert Knowledge Base by Service Error: %s'**,  
 jv\_ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **if** knowledge\_base\_dict:  
 data = {}  
 **for** key, value **in** knowledge\_base\_dict.items():  
 **if** (key == constants.TARGET) & (value != **"Y"**):  
 knowledge\_base\_dict[key] = **"Y"  
 print**(**"Java Knowledge Base Update Service"**)  
 data = {key1: value1 **for** key1, value1 **in** knowledge\_base\_dict.items()  
 **if** key1 **in** [**"dataSourceName"**, **"sentiment"**,  
 **"timeSeriesDateAttribute"**,  
 **"timeSeriesKpi"**, **"target"**,  
 **"normalyViewedWith"**, **"index"**,  
 **"defaultFrequency"**, **"relationship"**]}  
 **for** i, \_ **in** enumerate(data[**"relationship"**]):  
 data[**"relationship"**][i] = {key1: value1 **for** key1, value1 **in** data[**"relationship"**][i].items()  
 **if** key1 **in** [**"mappingTable"**,  
 **"mappingColumn"**]}  
 url = constants.UPDATE\_KB\_BY\_SOURCEID\_COLUMN +\  
 str(self.data\_source\_id) + str(**"&columnName="**) + str(column\_name)  
 **try**:  
 authorization\_header.update({**"Content-Type"**: **"application/json"**})  
 RequestBuilder().call\_http\_put\_request(url=url,  
 data=json.dumps(data),  
 headers=  
 authorization\_header)  
 **except** Exception **as** jv\_ex:  
 LOGGER.error(**'Exception: Java Insert Knowledge Base by Service: '  
 'Error %s'**, jv\_ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **else**:  
 insert\_data[**"kbInput"**].append({**"columnName"**: column\_name, **"sentiment"**: **"Y"**,  
 **"timeSeriesDateAttribute"**: **"N"**,  
 **"timeSeriesKpi"**: **"N"**, **"target"**: **"Y"**,  
 **"normalyViewedWith"**: **"{}"**, **"index"**: **"N"**,  
 **"defaultFrequency"**: **"None"**,  
 **"relationship"**: []})  
 **try**:  
 RequestBuilder().call\_http\_post\_request(url=  
 constants.INSERT\_IN\_KNOWLEDGE\_BASE,  
 data=insert\_data,  
 headers=authorization\_header)  
 **except** Exception **as** jv\_ex:  
 LOGGER.error(**'Exception: Java Insert Knowledge Base by Service Error: %s'**,  
 jv\_ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 LOGGER.debug(**"Python Function update knowledge base ran successfully"**)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

**from** celery.utils.log **import** get\_task\_logger  
  
logger = get\_task\_logger(\_\_name\_\_)  
  
**import** pandas **as** pd  
**from** insights.auto\_insights.profiling\_code **import** describe\_lens  
**import** numpy **as** np  
  
  
**def** get\_insight\_text\_list(aa, reasons\_df, group\_column, haha):  
 nlg\_list = []  
 **for** insight\_id **in** aa:  
 insight\_text = **""** insight\_text\_cat = **""** categorical\_insights\_dict = {}  
 insight\_dict = {}  
 **if** insight\_id[2] == **"max\_drop"**:  
 check\_val = str(insight\_id[1][**'red'**][0][1][0])  
 reasons\_df[group\_column] = reasons\_df[group\_column].astype(str)  
 df1 = reasons\_df.query(**"{0} != @check\_val"**.format(group\_column[0]))  
  
  
  
 df1.drop(group\_column, axis=1, inplace=True)  
 df2 = reasons\_df.query(**"{0} == @check\_val"**.format(group\_column[0]))  
 df2.drop(group\_column, axis=1, inplace=True)  
  
 insights = get\_stats\_insights(df1, df2, haha.shape[0])  
  
 df1 = get\_non\_date\_df(df1)  
 df2 = get\_non\_date\_df(df2)  
 df1\_summary = describe\_lens(df1, correlation\_threshold=0.7)  
 df2\_summary = describe\_lens(df2, correlation\_threshold=0.7)  
 categorical\_insights = extract\_reason\_for\_insight\_from\_categorical\_columns(df1, df2,  
 df1\_summary,  
 df2\_summary)  
  
 **if** categorical\_insights:  
 **for** insight\_cat **in** categorical\_insights:  
 insight\_cat = insight\_cat.replace(**"df1"**, **"others"**)  
 insight\_cat = insight\_cat.replace(**"df2"**,  
 str(group\_column[0]) + **" "** + str(check\_val))  
 insight\_text\_cat = insight\_text\_cat + insight\_cat + **"\n"** categorical\_insights\_dict[**'title'**] = **"Categorical Insights"** categorical\_insights\_dict[**'text'**] = insight\_text\_cat  
 nlg\_list.append(categorical\_insights\_dict)  
 **if** insights:  
 **for** aa1 **in** insights:  
 aa1 = aa1.replace(**"df1"**, **"others"**)  
 aa1 = aa1.replace(**"df2"**, str(group\_column[0]) + **" "** + str(check\_val))  
 insight\_text = insight\_text + aa1 + **"\n"** insight\_dict[**'title'**] = **"Statistical Insights"** insight\_dict[**'text'**] = insight\_text  
 nlg\_list.append(insight\_dict)  
 **return** nlg\_list  
  
  
**def** extract\_reason\_for\_insight\_from\_categorical\_columns(df1, df2, df1\_summary, df2\_summary):  
 cols = list(df1.select\_dtypes(exclude=[np.number]).columns)  
 categorical\_insights = []  
 **for** i **in** cols:  
 haha1 = (df1\_summary[**'freq'**][i] / df1.shape[0]).to\_frame(name=**"df1"**)  
 haha2 = (df2\_summary[**'freq'**][i] / df2.shape[0]).to\_frame(name=**"df2"**)  
  
 out = haha1.join(haha2, how=**"outer"**)  
 out = out.fillna(0)  
 out[**'diff'**] = out[**'df2'**] - out[**'df1'**]  
 out = out.query(**'diff > 0.10 | diff < -0.10'**)  
 **if** out.shape[0] > 0:  
 **for** index, element **in** out.iterrows():  
 **if** element[**'df1'**] == 0:  
 text = **"No records for df1 in "** + i + **"-"** + str(index)  
 categorical\_insights.append(text)  
 **elif** element[**'df2'**] == 0:  
 text = **"No records for df2 in "** + i + **"-"** + str(index)  
 categorical\_insights.append(text)  
  
 **else**:  
 **if** element[**'diff'**] > 0:  
 text = **"df2 has "** + str(round(element[**'diff'**] \* 100,  
 2)) + **" percent more records than average for "** + i + **"-"** + str(  
 index)  
 categorical\_insights.append(text)  
 **else**:  
 text = **"df2 has "** + str(round(abs(element[**'diff'**] \* 100),  
 2)) + **" percent less records than average for "** + i + **"-"** + str(  
 index)  
 categorical\_insights.append(text)  
  
 **return** categorical\_insights  
  
  
**def** get\_correlation\_insights(df):  
 df\_summary = describe\_lens(df, correlation\_threshold=0.7)  
 *# correlation\_insights = []* insight\_text = **""** corr\_df = df\_summary[**'variables'**].query(**'type == "CORR"'**)[[**'correlation\_var'**, **'correlation'**]]  
 **if** corr\_df.shape[0] > 0:  
 **for** index\_corr, element\_corr **in** corr\_df.iterrows():  
 insight\_text = insight\_text + **" correlation between "** + index\_corr + **" and "** + \  
 element\_corr[**'correlation\_var'**] + **" is "** + str(  
 round(element\_corr[**'correlation'**], 2)) + **","** insight\_text = insight\_text.rstrip(**","**)  
 *# correlation\_insights.append(insight\_text)* **return** insight\_text  
  
  
**def** get\_stats\_insights(df1, df2, df1\_records):  
 gg1 = df1.describe().transpose()[[**'count'**, **'mean'**, **'std'**]]  
 gg1[**'count'**] = gg1[**'count'**] / (df1\_records - 1)  
 gg2 = df2.describe().transpose()[[**'count'**, **'mean'**, **'std'**]]  
  
 gg2 = gg2.rename(index=str,  
 columns={**"count"**: **"count\_df2"**, **"mean"**: **"mean\_df2"**, **"std"**: **"std\_df2"**})  
  
 result = pd.concat([gg1, gg2], axis=1, sort=False)  
  
 stats\_insights = []  
 **for** df\_index, df\_element **in** result.iterrows():  
 **if** df\_element[**'mean\_df2'**] > 0:  
 **if** df\_element[**'mean'**] / df\_element[**'mean\_df2'**] > 1.5:  
 stats\_insights.append(**"Significant shift in average for "** +  
 df\_index + **"(value for df1 is "** + str(  
 round(df\_element[**'mean'**], 2)) +  
 **" and for df2 it's "** + str(round(df\_element[**'mean\_df2'**], 2)) +  
 **")"**)  
  
 **if** df\_element[**'std\_df2'**] > 0:  
 **if** df\_element[**'std'**] / df\_element[**'std\_df2'**] > 1.5:  
 stats\_insights.append(**"Significant shift in standard deviation for "** +  
 df\_index + **"(value for df1 is "** + str(  
 round(df\_element[**'std'**], 2)) +  
 **" and for df2 it's "** + str(round(df\_element[**'std\_df2'**], 2)) +  
 **")"**)  
  
 **if** df\_element[**'count\_df2'**] > 0:  
 **if** df\_element[**'count'**] / df\_element[**'count\_df2'**] > 1.5:  
 stats\_insights.append(**"Imbalanced number of records "** +  
 **"(Average number of records for df1 is "** + str(  
 round(df\_element[**'count'**])) +  
 **" and for df2 it's "** + str(round(df\_element[**'count\_df2'**])) +  
 **")"**)  
  
 stats\_insights = list(set(stats\_insights))  
  
 **return** stats\_insights  
  
  
**def** strictly\_increasing(L):  
 **return** all(x < y **for** x, y **in** zip(L, L[1:]))  
  
  
**def** strictly\_decreasing(L):  
 **return** all(x > y **for** x, y **in** zip(L, L[1:]))  
  
  
**def** get\_trend\_insights(group\_column, increasing, df, date\_column):  
 group\_column = group\_column  
 increasing = increasing  
 *# code for type insights* index\_variable = 0  
 start = 0  
 element\_list = []  
 **while** index\_variable < len(df):  
 aa = df.query(**'index >= @start and index < @index\_variable+1'**)  
 **if** increasing:  
 **if** aa[group\_column].is\_monotonic\_increasing **and** strictly\_increasing(aa[group\_column]):  
 index\_variable = index\_variable + 1  
 **if** index\_variable == len(df):  
 element\_list.append((start, index\_variable - 1))  
 **else**:  
 element\_list.append((start, index\_variable - 1))  
 start = index\_variable + 1  
 index\_variable = index\_variable + 1  
  
 **else**:  
 **if** aa[group\_column].is\_monotonic\_decreasing **and** strictly\_decreasing(aa[group\_column]):  
 index\_variable = index\_variable + 1  
 **if** index\_variable == len(df):  
 element\_list.append((start, index\_variable - 1))  
 **else**:  
 element\_list.append((start, index\_variable - 1))  
 start = index\_variable + 1  
 index\_variable = index\_variable + 1  
  
 increasing\_list\_with\_filters = []  
 thirty\_per\_value = round(len(df) \* 0.2)  
 **for** element **in** element\_list:  
 **if** int(element[1]) - int(element[0]) > thirty\_per\_value:  
 increasing\_list\_with\_filters.append(element)  
  
 nlg\_text = **""** out\_dict = {}  
 element\_list\_out = []  
  
 **if** increasing\_list\_with\_filters:  
 **if** increasing:  
 nlg\_text = **"There is increasing trend in the datset from "** key\_val = **"green"  
 else**:  
 nlg\_text = **"There is decreasing trend in the datset from "** key\_val = **"red"  
 for** element **in** increasing\_list\_with\_filters:  
 element\_list = []  
 element\_1 = int(element[0])  
 element\_2 = int(element[1])  
 date\_1 = df.query(**'index == @element\_1'**)[date\_column].iloc[0]  
 date\_1 = str(date\_1).replace(**"00:00:00"**, **""**).strip()  
 date\_2 = df.query(**'index == @element\_2'**)[date\_column].iloc[0]  
 date\_2 = str(date\_2).replace(**"00:00:00"**, **""**).strip()  
 nlg\_text = nlg\_text + date\_1 + **" to "** + date\_2 + **", "** index\_1 = df[df[date\_column] == date\_1].index[0]  
 index\_2 = df[df[date\_column] == date\_2].index[0] + 1  
  
 **for** list\_value **in** range(index\_1, index\_2):  
 v1 = df.query(**'index == @list\_value'**)[date\_column].iloc[0]  
 v2 = df.query(**'index == @list\_value'**)[group\_column].iloc[0]  
 element\_list.append((v1, v2))  
 *# print(element\_list)* element\_list\_out.append(element\_list)  
  
 *# print(element\_list)* out\_dict[key\_val] = element\_list\_out  
 nlg\_text = nlg\_text.strip().rstrip(**","**)  
 **if** nlg\_text == **''**:  
 **return** None, out\_dict  
 **else**:  
 **return** nlg\_text, out\_dict  
  
  
**def** get\_rolling\_insights(df, group\_column, date\_column):  
 *"""  
 Insights are generated on 1. Minimum record 2. Maximum record  
 3. Minimum Change 4. Maximum change  
 5. almost all increasing  
 6. almost all decresing  
 7. all inreasing  
 8. all decresing  
 """* insights\_list = []  
 df\_copy = df.copy()  
 *# df\_copy = df\_copy.sort\_values(by=[date\_column])* df\_copy[**'diff'**] = df\_copy[group\_column].rolling(window=2).apply(**lambda** x: x[1] - x[0], raw=True)  
  
 min\_value = df\_copy[**'diff'**].min()  
 max\_value = df\_copy[**'diff'**].max()  
 mean\_value = df\_copy[**'diff'**].mean()  
 std\_value = df\_copy[**'diff'**].std()  
  
 min\_column\_value = df\_copy[group\_column].min()  
 max\_column\_value = df\_copy[group\_column].max()  
  
 *# below code will either generate type 7/8/ 1 & 2 of insights  
  
 # type 7 all inreasing insight* **if** df\_copy[group\_column].is\_monotonic\_increasing **and** df\_copy.iloc[[2]][group\_column].iloc[  
 0] > 0 **and** strictly\_increasing(df\_copy[group\_column]):  
 *# all\_inreasing = "Upward trend for {group\_column} with minimum of {min\_column\_value} and maximum of {max\_column\_value}"* all\_inreasing = **"Upward trend for {} with minimum of {} and maximum of {}"**.format(  
 group\_column, min\_column\_value, max\_column\_value)  
  
 out\_dict = {}  
 element\_list = []  
 **for** col1, col2 **in** df\_copy.iterrows():  
 element\_list.append((col2[date\_column], col2[group\_column]))  
  
 out\_dict[**'limegreen'**] = [element\_list]  
 insights\_list.append((all\_inreasing, out\_dict, **"all\_increase"**))  
  
 *# type 8 all decresing* **elif** df\_copy[group\_column].is\_monotonic\_decreasing **and** strictly\_decreasing(  
 df\_copy[group\_column]):  
 *# all\_decresing = "Downward trend for {group\_column} with minimum of {min\_column\_value} and maximum of {max\_column\_value}"* all\_decresing = **"Downward trend for {} with minimum of {} and maximum of {}"**.format(  
 group\_column, min\_column\_value, max\_column\_value)  
  
 out\_dict = {}  
 element\_list = []  
 **for** col1, col2 **in** df\_copy.iterrows():  
 element\_list.append((col2[date\_column], col2[group\_column]))  
  
 out\_dict[**'maroon'**] = [element\_list]  
 insights\_list.append((all\_decresing, out\_dict, **"all\_decrease"**))  
  
 **else**:  
 (item\_1, item2) = get\_trend\_insights(group\_column, True, df\_copy, date\_column)  
 trends\_increase = item\_1  
  
 **if** trends\_increase **is not** None:  
 insights\_list.append((trends\_increase, item2, **"increase\_trend"**))  
  
 (item\_1, item2) = get\_trend\_insights(group\_column, False, df\_copy, date\_column)  
 trends\_decrease = item\_1  
  
 **if** trends\_decrease **is not** None:  
 insights\_list.append((trends\_decrease, item2, **"decrease\_trend"**))  
  
 *# type 3 insight* **if** min\_value < mean\_value - 1.5 \* std\_value:  
 min\_index\_value = df\_copy.query(**'diff == @min\_value'**).index[0]  
 previous\_min\_index\_value = min\_index\_value - 1  
  
 out\_dict = {}  
 element\_list = []  
 v1 = df\_copy.query(**'index == @min\_index\_value'**)[date\_column].iloc[0]  
 v2 = df\_copy.query(**'index == @previous\_min\_index\_value'**)[date\_column].iloc[0]  
  
 v3 = df\_copy.query(**'index == @min\_index\_value'**)[group\_column].iloc[0]  
 v4 = df\_copy.query(**'index == @previous\_min\_index\_value'**)[group\_column].iloc[0]  
  
 element\_list.append((v2, v4))  
 element\_list.append((v1, v3))  
  
 out\_dict[**'red'**] = [element\_list]  
  
 min\_change = round(df\_copy.query(**'diff == @min\_value'**)[**'diff'**].iloc[0], 2)  
 date\_2 = df\_copy.query(**'diff == @min\_value'**)[date\_column].iloc[0]  
 date\_2 = str(date\_2).replace(**"00:00:00"**, **""**).strip()  
 date\_1 = df\_copy.query(**'index == @previous\_min\_index\_value'**)[date\_column].iloc[0]  
 date\_1 = str(date\_1).replace(**"00:00:00"**, **""**).strip()  
 *# min\_nlg = "Maximum drop in {group\_column} is {min\_change} which occurs between {date\_1} and {date\_2}"* min\_nlg = **"Maximum drop in {} is {} which occurs between {} and {}"**.format(group\_column,  
 min\_change,  
 date\_1, date\_2)  
  
 insights\_list.append((min\_nlg, out\_dict, **"max\_drop"**))  
  
 *# type 4 insight* **if** max\_value > mean\_value + 1.5 \* std\_value:  
 max\_index\_value = df\_copy.query(**'diff == @max\_value'**).index[0]  
 previous\_max\_index\_value = max\_index\_value - 1  
  
 out\_dict = {}  
 element\_list = []  
  
 v1 = df\_copy.query(**'index == @max\_index\_value'**)[date\_column].iloc[0]  
 v2 = df\_copy.query(**'index == @previous\_max\_index\_value'**)[date\_column].iloc[0]  
  
 v3 = df\_copy.query(**'index == @max\_index\_value'**)[group\_column].iloc[0]  
 v4 = df\_copy.query(**'index == @previous\_max\_index\_value'**)[group\_column].iloc[0]  
  
 element\_list.append((v2, v4))  
 element\_list.append((v1, v3))  
  
 out\_dict[**'green'**] = [element\_list]  
 max\_change = round(df\_copy.query(**'diff == @max\_value'**)[**'diff'**].iloc[0], 2)  
 date\_2 = df\_copy.query(**'diff == @max\_value'**)[date\_column].iloc[0]  
 date\_2 = str(date\_2).replace(**"00:00:00"**, **""**).strip()  
 date\_1 = df\_copy.query(**'index == @previous\_max\_index\_value'**)[date\_column].iloc[0]  
 date\_1 = str(date\_1).replace(**"00:00:00"**, **""**).strip()  
  
 *# max\_nlg = "Maximum rise in {group\_column} is {max\_change} which occurs between {date\_1} and {date\_2}"* max\_nlg = **"Maximum rise in {} is {} which occurs between {} and {}"**.format(group\_column,  
 max\_change,  
 date\_1, date\_2)  
  
 insights\_list.append((max\_nlg, out\_dict, **"max\_rise"**))  
  
 **return** insights\_list  
  
  
**def** get\_date\_df(df, date\_column):  
 **try**:  
 df[date\_column] = pd.to\_datetime(df[date\_column])  
 **return** df  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 *# hour  
  
  
# second  
# quarter  
# year  
# month  
# day  
# week  
# weekday  
# day\_name()  
# date  
# month\_name()  
# quarter  
  
# https://pandas.pydata.org/pandas-docs/stable/search.html?q=Series.dt&check\_keywords=yes&area=default***def** add\_date\_columns(df, date\_column):  
 *"""  
 This method adds date columns and return 2 things.  
 One is the df and other is dictionary of created variables  
 """* **try**:  
 variable\_dictionary = {}  
  
 df[date\_column + **"\_hour"**] = df[date\_column].dt.hour  
 variable\_dictionary[date\_column + **"\_hour"**] = **"hour"** df[date\_column + **"\_second"**] = df[date\_column].dt.second  
 variable\_dictionary[date\_column + **"\_second"**] = **"second"** df[date\_column + **"\_quarter"**] = df[date\_column].dt.quarter  
 variable\_dictionary[date\_column + **"\_quarter"**] = **"quarter"** df[date\_column + **"\_year"**] = df[date\_column].dt.year  
 variable\_dictionary[date\_column + **"\_year"**] = **"year"** df[date\_column + **"\_month"**] = df[date\_column].dt.month  
 variable\_dictionary[date\_column + **"\_month"**] = **"month"** df[date\_column + **"\_day"**] = df[date\_column].dt.day  
 variable\_dictionary[date\_column + **"\_day"**] = **"day"** df[date\_column + **"\_week"**] = df[date\_column].dt.week  
 variable\_dictionary[date\_column + **"\_week"**] = **"week"** df[date\_column + **"\_weekday"**] = df[date\_column].dt.weekday  
 variable\_dictionary[date\_column + **"\_weekday"**] = **"weekday"** df[date\_column + **"\_day\_name"**] = df[date\_column].dt.day\_name()  
 variable\_dictionary[date\_column + **"\_day\_name"**] = **"day\_name"** df[date\_column + **"\_dates"**] = df[date\_column].dt.date  
 variable\_dictionary[date\_column + **"\_dates"**] = **"dates"** df[date\_column + **"\_month\_name"**] = df[date\_column].dt.month\_name()  
 variable\_dictionary[date\_column + **"\_month\_name"**] = **"month\_name"** df[date\_column + **"\_quarter"**] = df[date\_column].dt.quarter  
 variable\_dictionary[date\_column + **"\_quarter"**] = **"quarter"  
  
 return** df, variable\_dictionary  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
  
**def** add\_date\_columns\_from\_name(df, date\_column, date\_type\_column):  
 *"""  
 This method adds date columns and return 2 things.  
 One is the df and other is dictionary of created variables  
 """* **try**:  
 variable\_dictionary = {}  
 **if** date\_type\_column == **"A"**:  
 df[date\_column + **"\_year"**] = df[date\_column].dt.year  
 variable\_dictionary[date\_column + **"\_year"**] = **"year"  
  
 elif** date\_type\_column == **"M"**:  
 df[date\_column + **"\_month"**] = df[date\_column].dt.month  
 variable\_dictionary[date\_column + **"\_month"**] = **"month"  
  
  
 elif** date\_type\_column == **"W"**:  
 df[date\_column + **"\_week"**] = df[date\_column].dt.week  
 variable\_dictionary[date\_column + **"\_week"**] = **"week"  
  
  
 elif** date\_type\_column == **"D"**:  
 df[date\_column + **"\_dates"**] = df[date\_column].dt.date  
 variable\_dictionary[date\_column + **"\_dates"**] = **"dates"  
  
 elif** date\_type\_column == **"Q"**:  
 df[date\_column + **"\_quarter"**] = df[date\_column].dt.quarter  
 variable\_dictionary[date\_column + **"\_quarter"**] = **"quarter"  
  
 return** df, variable\_dictionary  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
  
**def** get\_date\_granularity(date\_df, variable\_dict):  
 date\_df\_copy = date\_df.copy()  
 out\_dict = {}  
 **for** key, value **in** variable\_dict.items():  
 param\_dict = {}  
 param\_dict[**'no\_unique'**] = len(date\_df\_copy[key].unique())  
 param\_dict[**'var\_type'**] = key  
 out\_dict[value] = param\_dict  
  
 **return** out\_dict  
  
  
**def** get\_group(out2):  
 group\_list = []  
 column\_name\_list = []  
 **for** i, j **in** out2.items():  
 **if** i == **"dates"**:  
 *# used when number of days in less* **if** j[**'no\_unique'**] <= 60:  
 group\_list.append(str(out2[**'dates'**][**'var\_type'**]))  
 column\_name\_list.append(**'dates'**)  
  
 *# when data is avilable for 2 months or more upto a year* **if** j[**'no\_unique'**] > 60 **and** j[**'no\_unique'**] <= 365:  
 group\_list.append(str(out2[**'week'**][**'var\_type'**]))  
 column\_name\_list.append(**'week'**)  
  
 *# when data is available for 1 to 2 years* **if** j[**'no\_unique'**] > 365 **and** j[**'no\_unique'**] <= 730:  
 group\_list.append(str(out2[**'year'**][**'var\_type'**]))  
 group\_list.append(str(out2[**'month'**][**'var\_type'**]))  
 column\_name\_list.append(**'year'**)  
 column\_name\_list.append(**'month'**)  
  
 *# when data is available for 1 to 2 years* **if** j[**'no\_unique'**] > 730 **and** j[**'no\_unique'**] <= 3650:  
 group\_list.append(str(out2[**'year'**][**'var\_type'**]))  
 group\_list.append(str(out2[**'quarter'**][**'var\_type'**]))  
 column\_name\_list.append(**'year'**)  
 column\_name\_list.append(**'quarter'**)  
  
 *# when data is available for moore that 10 years* **if** j[**'no\_unique'**] > 3650:  
 group\_list.append(str(out2[**'year'**][**'var\_type'**]))  
 column\_name\_list.append(**'year'**)  
  
 **return** (group\_list, column\_name\_list)  
  
  
**from** pandas **import** Series  
**from** dateutil.parser **import** parse  
**import** datetime  
  
  
**def** is\_date(series):  
 *"""Function that identifies if a column is date"""* **try**:  
 series = series.dropna()  
 series = series[:1001]  
 *# Parse returns true if the object is date. If all rows are date, return true* **if** all([isinstance(x, datetime.datetime) **for** x **in** series]) **or** \  
 all([parse(x) **for** x **in** series]):  
 **return** True  
 **except** (TypeError, ValueError, AttributeError, IndexError, OverflowError):  
 **return** False  
  
  
*# Dates stored in ddmmyyyy format. Should include other formats later  
# Dates stored in yyyymmdd and ddmmyyyy formats are already handled in the above function***def** is\_date\_in\_numeric(series):  
 *"""Function identifies if the column is a date in ddmmyyyy format"""* series = series.dropna()  
 series = series[:1001]  
 **try**:  
 *# Year should be between 1850 and 2050 and Month cannot be more than 12 and  
 # Date cannot be more than 12* **if** (**not** all([float(str(x)[-4:]) > 1850 **and** float(str(x)[:4]) < 2050 **for** x **in** series])) **or** \  
 (**not** all([float(str(x)[2:4]) < 13 **for** x **in** series])) **or** \  
 (**not** all([float(str(x)[0:2]) < 32 **for** x **in** series])) **or** \  
 (**not** all([len(str(x)) == 8 **for** x **in** series])):  
 **return** False  
  
 **except** ValueError:  
 **return** False  
  
 **return** True  
  
  
**def** get\_non\_date\_df(df):  
 cols = list(df.columns)  
 non\_date\_list = []  
  
 **for** column **in** cols:  
 **if not** is\_date(df[column]) **or** is\_date\_in\_numeric(df[column]):  
 non\_date\_list.append(column)  
  
 **return** df[non\_date\_list]  
  
  
**def** get\_line\_graph\_color(insights\_dict):  
 points = []  
 **for** i **in** insights\_dict:  
  
 **for** j **in** i[1].items():  
 *# print(j[1])  
 # print("")* **for** tuppe\_list **in** j[1]:  
 *# tuppe\_list = j[1]  
 # print(tuppe\_list)* **for** i, val **in** enumerate(tuppe\_list):  
 **if** i < len(tuppe\_list) - 1:  
 sample = []  
 out = (val[0], tuppe\_list[i + 1][0])  
 out1 = (val[1], tuppe\_list[i + 1][1])  
 d = {}  
 d[**'xAxis'**] = val[0]  
 d[**'yAxis'**] = val[1]  
 d[**'lineStyle'**] = {**'color'**: j[0], **'width'**: 2, **'type'**: **'solid'**,  
 **'opacity'**: 0.7}  
 sample.append(d)  
 d = {}  
 d[**'xAxis'**] = tuppe\_list[i + 1][0]  
 d[**'yAxis'**] = tuppe\_list[i + 1][1]  
 d[**'lineStyle'**] = {**'color'**: j[0], **'width'**: 2, **'type'**: **'solid'**,  
 **'opacity'**: 0.7}  
 sample.append(d)  
  
 points.append(sample)  
  
 **return** points  
  
  
**def** get\_line\_graph\_color\_old(insights\_dict):  
 points = []  
 **for** i **in** insights\_dict:  
 **for** j **in** i[1].items():  
 tuppe\_list = j[1]  
 **for** i, val **in** enumerate(tuppe\_list):  
 **if** i < len(tuppe\_list) - 1:  
 sample = []  
 out = (val[0], tuppe\_list[i + 1][0])  
 out1 = (val[1], tuppe\_list[i + 1][1])  
 d = {}  
 d[**'xAxis'**] = val[0]  
 d[**'yAxis'**] = val[1]  
 d[**'lineStyle'**] = {**'color'**: j[0], **'width'**: 2, **'type'**: **'solid'**, **'opacity'**: 0.7}  
 sample.append(d)  
 d = {}  
 d[**'xAxis'**] = tuppe\_list[i + 1][0]  
 d[**'yAxis'**] = tuppe\_list[i + 1][1]  
 d[**'lineStyle'**] = {**'color'**: j[0], **'width'**: 2, **'type'**: **'solid'**, **'opacity'**: 0.7}  
 sample.append(d)  
  
 points.append(sample)  
  
 **return** points  
  
  
**def** get\_line\_json(x, y, line\_elements, nlg\_chart\_text, column\_name\_string, kpi\_column):  
 sample\_json = {  
 **"tooltip"**: {**"trigger"**: **"item"**, **"axisPointer"**: {**"type"**: **"line"**}},  
 **"color"**: [**"#50d1d6"**, **"#3851b2"**, **"#9cf27c"**, **"#fbb17e"**, **"#3aaa67"**, **"#69a3ba"**, **"#a77fb2"**,  
 **"#92e0b8"**, **"#0099a9"**, **"#9eaee6"**, **"#bce094"**, **"#eba65e"**, **"#dbaa00"**, **"#d7c3d8"**,  
 **"#c5f7d8"**, **"#cfadd0"**, **"#cbd3eb"**, **"#d1debc"**, **"#f9daba"**, **"#bad3dd"**, **"#e1c86f"**,  
 **"#f2d7f4"**, **"#b7c5be"**, **"#fda4bc"**, **"#fce94f"**],  
 **"textStyle"**: {**"fontSize"**: 10, **"fontFamily"**: **"Lato"**, **"color"**: **"#999"**},  
 **"toolbox"**: {**"orient"**: **"vertical"**, **"top"**: **"middle"**, **"right"**: 10, **"itemGap"**: 30,  
 **"iconStyle"**: {**"textPosition"**: **"left"**}, **"feature"**: {  
 **"dataZoom"**: {**"title"**: {**"zoom"**: **"Zoom"**, **"back"**: **"Revert"**}, **"yAxisIndex"**: **"none"**},  
 **"restore"**: {**"title"**: **"Reset"**},  
 **"saveAsImage"**: {**"title"**: **"Save"**, **"name"**: **"company\_wise\_sum"**}}, **"show"**: **"false"**},  
 **"dataZoom"**: [{**"type"**: **"inside"**, **"xAxisIndex"**: [0]}],  
 **"legend"**: {**"type"**: **"scroll"**, **"textStyle"**: {**"fontSize"**: 10}, **"padding"**: [0, 80, 0, 0],  
 **"show"**: **"true"**, **"top"**: 10, **"width"**: **"80%"**, **"left"**: **"right"**, **"right"**: **"right"**,  
 **"orient"**: **"horizontal"**},  
 **"title"**: {**"show"**: **"false"**, **"text"**: **"company\_wise\_sum"**, **"top"**: **"top"**, **"left"**: **"left"**,  
 **"right"**: **"left"**, **"subtext"**: **"Default Description"**,  
 **"textStyle"**: {**"fontSize"**: 12, **"color"**: **"#333"**}, **"subtextStyle"**: {**"fontSize"**: 10}},  
  
 **"title"**: {  
 **"text"**: nlg\_chart\_text, **"textStyle"**: {**"fontSize"**: 10}},  
 **"xAxis"**: [{**"type"**: **"category"**, **"position"**: **"bottom"**, **"id"**: **""**, **"name"**: column\_name\_string,  
 **"nameTextStyle"**: {**"padding"**: [30, 0, 0, 0], **"fontSize"**: 12, **"fontWeight"**: **"bold"**,  
 **"backgroundColor"**: **"#fff"**}, **"offset"**: 0, **"nameGap"**: 20,  
 **"nameLocation"**: **"start"**, **"axisTick"**: {**"show"**: **"false"**},  
 **"axisbar"**: {**"show"**: **"true"**},  
 **"axisLine"**: {**"show"**: **"true"**, **"lineStyle"**: {**"color"**: **"#999"**, **"width"**: 1}},  
 **"axisLabel"**: {**"show"**: **"true"**, **"interval"**: 0, **"fontSize"**: 10, **"rotate"**: 50},  
 **"data"**: x  
 }],  
 **"yAxis"**: [{**"name"**: kpi\_column, **"nameLocation"**: **"end"**, **"nameGap"**: 20,  
 **"nameTextStyle"**: {**"padding"**: [0, 0, 0, 10], **"fontWeight"**: **"bold"**,  
 **"backgroundColor"**: **"#fff"**, **"fontSize"**: 12}, **"type"**: **"value"**,  
 **"axisLine"**: {**"lineStyle"**: {**"color"**: **"#999"**, **"width"**: 1}}, **"axisLabel"**: {}}],  
 **"series"**: [{  
 **"name"**: **""**,  
 **"type"**: **"line"**,  
 **"data"**: y,  
 **"markPoint"**: {  
 **"data"**: [  
 {**"type"**: **"max"**, **"name"**: **"max"**, **"symbol"**: **"pin"**, **"symbolSize"**: 30,  
 **"itemStyle"**: {**"color"**: **"green"**}},  
 {**"type"**: **"min"**, **"name"**: **"min"**, **"symbol"**: **"pin"**, **"symbolSize"**: 30,  
 **"itemStyle"**: {**"color"**: **"green"**}}  
 ]},  
 **"markLine"**: {  
 **"data"**: line\_elements}  
 }]}  
  
 *# output\_string = "option =" + str(sample\_json)* output\_string = str(sample\_json)  
  
 output\_string = output\_string.replace(**'\\n'**, **'\n'**)  
  
 output\_string = output\_string.replace(**"'false'"**, **'false'**)  
 output\_string = output\_string.replace(**"'true'"**, **'true'**)  
  
 **return** output\_string  
  
  
**from** transformation.feature\_selector.feature\_selector **import** FeatureSelector  
  
  
**def** get\_bar\_chart(highlight\_name, y, x, chart\_title):  
 chart\_json = {  
 **"title"**: {  
 **"text"**: chart\_title  
 },  
 **"tooltip"**: {  
 **"trigger"**: **'axis'**,  
 **"axisPointer"**: {  
 **"type"**: **'shadow'** }  
 },  
 **"xAxis"**: {  
 **"type"**: **"value"**,  
 **"boundaryGap"**: [0, 0.01]  
 },  
 **"yAxis"**: {  
 **"type"**: **"category"**,  
 **"data"**: y  
 },  
  
 **"visualMap"**: {  
 **"orient"**: **"horizontal"**,  
 **"left"**: **"center"**,  
 **"min"**: 10,  
 **"max"**: 60,  
 **"text"**: [**'High Score'**, **'Low Score'**],  
 **"dimension"**: 0,  
 **"inRange"**: {  
 **"color"**: [**"#50d1d6"**, **"#6b5b95"**]  
 },  
 **"show"**: **'true'** },  
 **"series"**: [  
 {  
 **"name"**: highlight\_name,  
 **"type"**: **"bar"**,  
 **"data"**: x  
 }  
 ]  
 }  
  
 output\_string = str(chart\_json)  
 output\_string = output\_string.replace(**'\\n'**, **'\n'**)  
 output\_string = output\_string.replace(**"'false'"**, **'false'**)  
 output\_string = output\_string.replace(**"'true'"**, **'true'**)  
  
 **return** output\_string  
  
  
**def** chart\_json\_imp\_factors(data\_frame, target\_column, n\_columns):  
 input\_json = {**"n\_columns"**: n\_columns, **"target\_column"**: target\_column}  
  
 df = FeatureSelector(input\_json).get\_result\_importance(data\_frame.copy())  
 df = df.sort\_values(by=[**'Importance'**])  
 columns = df[**'index'**].tolist()  
 values = df[**'Importance'**].tolist()  
 highlight\_name = **"Percent Contribution"** chart\_title = **"These factors affect "** + target\_column  
  
 **return** get\_bar\_chart(highlight\_name, columns, values, chart\_title)  
  
  
**def** do\_magic(data, date\_column\_function, drill\_down\_columns, date\_granularity, kpi\_columns\_list,  
 data\_source\_id, user\_id, data\_source\_name):  
 **try**:  
  
 logger.info(**"DO magic"**)  
  
 date\_column\_function = date\_column\_function.replace(**' '**, **''**)  
  
 **for** drill\_down\_columns\_index, drill\_down\_columns\_item **in** enumerate(drill\_down\_columns):  
 drill\_down\_columns[drill\_down\_columns\_index] = drill\_down\_columns\_item.replace(**' '**, **''**)  
  
 **for** kpi\_columns\_list\_index, kpi\_columns\_list\_item **in** enumerate(kpi\_columns\_list):  
 kpi\_columns\_list[kpi\_columns\_list\_index] = kpi\_columns\_list\_item.replace(**' '**, **''**)  
  
 data\_main = data.copy()  
 data\_main.columns = data\_main.columns.str.replace(**' '**, **''**)  
  
 **import** time  
 time\_value = str(time.ctime())  
  
 insights\_list = []  
  
 auto\_insight\_list = []  
 out\_dict\_v2 = {}  
 out\_dict = {}  
 out\_dict\_v2[**'data\_source\_id'**] = data\_source\_id  
 out\_dict[**'datasourceName'**] = data\_source\_name  
 out\_dict[**'createdBy'**] = user\_id  
 out\_dict[**'type'**] = **"timeseries\_outlier"** drill\_down\_columns.append(**"RUN\_ALL"**)  
  
 drill\_down\_columns = drill\_down\_columns[::-1]  
 **for** run **in** drill\_down\_columns:  
 **if** run != **"RUN\_ALL"**:  
 items\_list = list(data[run].unique())  
 **else**:  
 items\_list = [**'RUN\_ALL'**]  
  
 **for** col **in** items\_list:  
 **if** col == **"RUN\_ALL"**:  
 data = data  
 **else**:  
 data = data\_main[data\_main[run] == col]  
  
 data\_original = data.copy()  
 data.columns = data.columns.str.replace(**' '**, **''**)  
 dates\_df = get\_date\_df(data, date\_column\_function)  
 dates\_df.columns = dates\_df.columns.str.replace(**' '**, **''**)  
 **if** date\_granularity != **"null"**:  
 dates\_df\_extra\_features = add\_date\_columns\_from\_name(dates\_df,  
 date\_column\_function,  
 date\_granularity)  
 group\_column = []  
 group\_column.append(list(dates\_df\_extra\_features[1].keys())[0])  
 val = list(dates\_df\_extra\_features[1].keys())[0]  
 column\_name\_string = dates\_df\_extra\_features[1][val]  
 **else**:  
 dates\_df\_extra\_features = add\_date\_columns(dates\_df, date\_column\_function)  
 out2 = get\_date\_granularity(dates\_df\_extra\_features[0],  
 dates\_df\_extra\_features[1])  
 group\_column = get\_group(out2)[0]  
 column\_name = get\_group(out2)[1]  
 column\_name\_string = **'-'**.join(column\_name)  
  
 reasons\_df = dates\_df\_extra\_features[0][list(data\_main.columns) + group\_column]  
  
 **for** kpi\_column **in** kpi\_columns\_list:  
  
 chart\_dict = {}  
 chart\_parent = {}  
 chart\_parent[**'display\_type'**] = 2  
  
 chart\_type\_list = {}  
 chart\_dict[**'display\_flag'**] = 1  
 chart\_dict[**'insight\_id'**] = 1  
 chart\_dict[**'insight\_type'**] = **"chart"** insight\_text = **'{0} with value {1} for {2}'**.format(run, col, kpi\_column)  
  
 chart\_parent[**'insight\_text'**] = insight\_text  
  
 haha = dates\_df\_extra\_features[0].groupby(group\_column, as\_index=False).agg(  
 {kpi\_column: [**'sum'**]})  
 haha.columns = haha.columns.droplevel(1)  
  
 haha = haha.sort\_values(by=group\_column)  
 **for** i **in** group\_column:  
 haha[i] = haha[i].astype(str)  
  
 haha[column\_name\_string] = haha[group\_column].apply(**lambda** x: **'\_'**.join(x),  
 axis=1)  
  
 aa = get\_rolling\_insights(haha, kpi\_column, column\_name\_string)  
  
 y = list(haha[kpi\_column])  
  
 x = list(haha[column\_name\_string])  
  
 nlg\_chart\_text = **""  
  
 for** element\_nlg **in** aa:  
 nlg\_chart\_text = nlg\_chart\_text + element\_nlg[0] + **" "** chart\_dict[**'nlp\_text'**] = nlg\_chart\_text  
  
 insight\_list = get\_insight\_text\_list(aa, reasons\_df, group\_column, haha)  
  
 sample1 = get\_line\_graph\_color(aa)  
  
 json\_val = get\_line\_json(x, y, sample1, nlg\_chart\_text, column\_name\_string,  
 kpi\_column)  
  
 chart\_dict[**'graph\_json'**] = json\_val  
  
 chart\_type\_list[**'value\_0'**] = chart\_dict  
  
 chart\_dict\_text = {}  
 chart\_dict\_text[**'display\_flag'**] = 1  
 chart\_dict\_text[**'insight\_id'**] = 2  
 chart\_dict\_text[**'insight\_type'**] = **"text"** correlation\_text = get\_correlation\_insights(reasons\_df)  
 **if** correlation\_text != **""**:  
 text\_dict = {}  
 text\_dict[**'title'**] = **"correlation analysis"** text\_dict[**'text'**] = correlation\_text  
  
 text\_json = insight\_list + [text\_dict]  
 **else**:  
 text\_json = insight\_list  
  
 chart\_dict\_text[**'display\_text'**] = text\_json  
  
 chart\_type\_list[**'value\_1'**] = chart\_dict\_text  
  
 **if** col == **"RUN\_ALL"**:  
 chart\_parent[**'display\_type'**] = 3  
 chart\_dict\_imp = {}  
 chart\_dict\_imp[**'display\_flag'**] = 1  
 chart\_dict\_imp[**'insight\_id'**] = 3  
 chart\_dict\_imp[**'insight\_type'**] = **"chart"** chart\_dict\_imp[**'graph\_json'**] = chart\_json\_imp\_factors(data\_main, kpi\_column,  
 5)  
 chart\_type\_list[**'value\_2'**] = chart\_dict\_imp  
  
 chart\_parent[**'value'**] = chart\_type\_list  
 insights\_list.append(chart\_parent)  
  
 out\_dict[**'required\_json'**] = insights\_list  
 auto\_insight\_list.append(out\_dict)  
 out\_dict\_v2[**'auto\_insights'**] = auto\_insight\_list  
 **return** out\_dict\_v2  
 **except** Exception **as** ex:  
 logger.info(**'Exception %s'**, ex.args[0])  
 **raise** Exception(ex.args[0])

*# -\*- coding: utf-8 -\*-  
"""Compute statistical description of datasets"""***import** billiard **as** multiprocessing  
**import** itertools  
**from** functools **import** partial  
**import** numpy **as** np  
**import** pandas **as** pd  
**import** matplotlib  
  
**from** pkg\_resources **import** resource\_filename  
**import** pandas\_profiling.formatters **as** formatters  
**import** pandas\_profiling.base **as** base  
**from** pandas\_profiling.plot **import** histogram, mini\_histogram  
  
**def** describe\_numeric\_1d(series, \*\*kwargs):  
 *"""Compute summary statistics of a numerical (`TYPE\_NUM`) variable (a Series).  
  
 Also create histograms (mini an full) of its distribution.  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """  
 # Format a number as a percentage. For example 0.25 will be turned to 25%.* \_percentile\_format = **"{:.0%}"** stats = dict()  
 stats[**'type'**] = base.TYPE\_NUM  
 stats[**'mean'**] = series.mean()  
 stats[**'std'**] = series.std()  
 stats[**'variance'**] = series.var()  
 stats[**'min'**] = series.min()  
 stats[**'max'**] = series.max()  
 stats[**'range'**] = stats[**'max'**] - stats[**'min'**]  
 *# To avoid to compute it several times* \_series\_no\_na = series.dropna()  
 **for** percentile **in** np.array([0.05, 0.25, 0.5, 0.75, 0.95]):  
 *# The dropna() is a workaround for https://github.com/pydata/pandas/issues/13098* stats[\_percentile\_format.format(percentile)] = \_series\_no\_na.quantile(percentile)  
 stats[**'iqr'**] = stats[**'75%'**] - stats[**'25%'**]  
 stats[**'kurtosis'**] = series.kurt()  
 stats[**'skewness'**] = series.skew()  
 stats[**'sum'**] = series.sum()  
 stats[**'mad'**] = series.mad()  
 stats[**'cv'**] = stats[**'std'**] / stats[**'mean'**] **if** stats[**'mean'**] **else** np.NaN  
 stats[**'n\_zeros'**] = (len(series) - np.count\_nonzero(series))  
 stats[**'p\_zeros'**] = stats[**'n\_zeros'**] \* 1.0 / len(series)  
 *# Histograms* stats[**'histogram'**] = histogram(series, \*\*kwargs)  
 stats[**'mini\_histogram'**] = mini\_histogram(series, \*\*kwargs)  
 **return** pd.Series(stats, name=series.name)  
  
  
**def** describe\_date\_1d(series):  
 *"""Compute summary statistics of a date (`TYPE\_DATE`) variable (a Series).  
  
 Also create histograms (mini an full) of its distribution.  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """* stats = dict()  
 stats[**'type'**] = base.TYPE\_DATE  
 stats[**'min'**] = series.min()  
 stats[**'max'**] = series.max()  
 stats[**'range'**] = stats[**'max'**] - stats[**'min'**]  
 *# Histograms* stats[**'histogram'**] = histogram(series)  
 stats[**'mini\_histogram'**] = mini\_histogram(series)  
 **return** pd.Series(stats, name=series.name)  
  
**def** describe\_categorical\_1d(series):  
 *"""Compute summary statistics of a categorical (`TYPE\_CAT`) variable (a Series).  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """  
 # Only run if at least 1 non-missing value* value\_counts, distinct\_count = base.get\_groupby\_statistic(series)  
 top, freq = value\_counts.index[0], value\_counts.iloc[0]  
 names = []  
 result = []  
  
 **if** base.get\_vartype(series) == base.TYPE\_CAT:  
 names += [**'top'**, **'freq'**, **'type'**]  
 result += [top, freq, base.TYPE\_CAT]  
  
 **return** pd.Series(result, index=names, name=series.name)  
  
**def** describe\_boolean\_1d(series):  
 *"""Compute summary statistics of a boolean (`TYPE\_BOOL`) variable (a Series).  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """* value\_counts, distinct\_count = base.get\_groupby\_statistic(series)  
 top, freq = value\_counts.index[0], value\_counts.iloc[0]  
 *# The mean of boolean is an interesting information* mean = series.mean()  
 names = []  
 result = []  
 names += [**'top'**, **'freq'**, **'type'**, **'mean'**]  
 result += [top, freq, base.TYPE\_BOOL, mean]  
  
 **return** pd.Series(result, index=names, name=series.name)  
  
**def** describe\_constant\_1d(series):  
 *"""Compute summary statistics of a constant (`S\_TYPE\_CONST`) variable (a Series).  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """* **return** pd.Series([base.S\_TYPE\_CONST], index=[**'type'**], name=series.name)  
  
**def** describe\_unique\_1d(series):  
 *"""Compute summary statistics of a unique (`S\_TYPE\_UNIQUE`) variable (a Series).  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """* **return** pd.Series([base.S\_TYPE\_UNIQUE], index=[**'type'**], name=series.name)  
  
**def** describe\_supported(series, \*\*kwargs):  
 *"""Compute summary statistics of a supported variable (a Series).  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """* leng = len(series) *# number of observations in the Series* count = series.count() *# number of non-NaN observations in the Series* n\_infinite = count - series.count() *# number of infinte observations in the Series* value\_counts, distinct\_count = base.get\_groupby\_statistic(series)  
 **if** count > distinct\_count > 1:  
 mode = series.mode().iloc[0]  
 **else**:  
 mode = series[0]  
  
 results\_data = {**'count'**: count,  
 **'distinct\_count'**: distinct\_count,  
 **'p\_missing'**: 1 - count \* 1.0 / leng,  
 **'n\_missing'**: leng - count,  
 **'p\_infinite'**: n\_infinite \* 1.0 / leng,  
 **'n\_infinite'**: n\_infinite,  
 **'is\_unique'**: distinct\_count == leng,  
 **'mode'**: mode,  
 **'p\_unique'**: distinct\_count \* 1.0 / leng}  
 **try**:  
 *# pandas 0.17 onwards* results\_data[**'memorysize'**] = series.memory\_usage()  
 **except**:  
 results\_data[**'memorysize'**] = 0  
  
 **return** pd.Series(results\_data, name=series.name)  
  
**def** describe\_unsupported(series, \*\*kwargs):  
 *"""Compute summary statistics of a unsupported (`S\_TYPE\_UNSUPPORTED`) variable (a Series).  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """* leng = len(series) *# number of observations in the Series* count = series.count() *# number of non-NaN observations in the Series* n\_infinite = count - series.count() *# number of infinte observations in the Series* results\_data = {**'count'**: count,  
 **'p\_missing'**: 1 - count \* 1.0 / leng,  
 **'n\_missing'**: leng - count,  
 **'p\_infinite'**: n\_infinite \* 1.0 / leng,  
 **'n\_infinite'**: n\_infinite,  
 **'type'**: base.S\_TYPE\_UNSUPPORTED}  
  
 **try**:  
 *# pandas 0.17 onwards* results\_data[**'memorysize'**] = series.memory\_usage()  
 **except**:  
 results\_data[**'memorysize'**] = 0  
  
 **return** pd.Series(results\_data, name=series.name)  
  
**def** describe\_1d(data, \*\*kwargs):  
 *"""Compute summary statistics of a variable (a Series).  
  
 The description is different according to the type of the variable.  
 However a set of common stats is also computed.  
  
 Parameters  
 ----------  
 series : Series  
 The variable to describe.  
  
 Returns  
 -------  
 Series  
 The description of the variable as a Series with index being stats keys.  
 """  
  
 # Replace infinite values with NaNs to avoid issues with  
 # histograms later.* data.replace(to\_replace=[np.inf, np.NINF, np.PINF], value=np.nan, inplace=True)  
  
 result = pd.Series({}, name=data.name)  
  
 vartype = base.get\_vartype(data)  
  
 **if** vartype == base.S\_TYPE\_UNSUPPORTED:  
 result = result.append(describe\_unsupported(data))  
 **else**:  
 result = result.append(describe\_supported(data))  
  
 **if** vartype == base.S\_TYPE\_CONST:  
 result = result.append(describe\_constant\_1d(data))  
 **elif** vartype == base.TYPE\_BOOL:  
 result = result.append(describe\_boolean\_1d(data))  
 **elif** vartype == base.TYPE\_NUM:  
 result = result.append(describe\_numeric\_1d(data, \*\*kwargs))  
 **elif** vartype == base.TYPE\_DATE:  
 result = result.append(describe\_date\_1d(data))  
 **elif** vartype == base.S\_TYPE\_UNIQUE:  
 result = result.append(describe\_unique\_1d(data))  
 **else**:  
 *# TYPE\_CAT* result = result.append(describe\_categorical\_1d(data))  
  
 **return** result  
  
**def** multiprocess\_func(x, \*\*kwargs):  
 **return** x[0], describe\_1d(x[1], \*\*kwargs)  
  
**def** describe\_lens(df, bins=10, check\_correlation=True, correlation\_threshold=0.9, correlation\_overrides=None, check\_recoded=False, pool\_size=multiprocessing.cpu\_count(), \*\*kwargs):  
 *"""Generates a dict containing summary statistics for a given dataset stored as a pandas `DataFrame`.  
  
 Used has is it will output its content as an HTML report in a Jupyter notebook.  
  
 Parameters  
 ----------  
 df : DataFrame  
 Data to be analyzed  
 bins : int  
 Number of bins in histogram.  
 The default is 10.  
 check\_correlation : boolean  
 Whether or not to check correlation.  
 It's `True` by default.  
 correlation\_threshold: float  
 Threshold to determine if the variable pair is correlated.  
 The default is 0.9.  
 correlation\_overrides : list  
 Variable names not to be rejected because they are correlated.  
 There is no variable in the list (`None`) by default.  
 check\_recoded : boolean  
 Whether or not to check recoded correlation (memory heavy feature).  
 Since it's an expensive computation it can be activated for small datasets.  
 `check\_correlation` must be true to disable this check.  
 It's `False` by default.  
 pool\_size : int  
 Number of workers in thread pool  
 The default is equal to the number of CPU.  
  
 Returns  
 -------  
 dict  
 Containing the following keys:  
 \* table: general statistics on the dataset  
 \* variables: summary statistics for each variable  
 \* freq: frequency table  
  
 Notes:  
 ------  
 \* The section dedicated to check the correlation should be externalized  
 """* **if not** isinstance(df, pd.DataFrame):  
 **raise** TypeError(**"df must be of type pandas.DataFrame"**)  
 **if** df.empty:  
 **raise** ValueError(**"df can not be empty"**)  
  
 **try**:  
 *# reset matplotlib style before use  
 # Fails in matplotlib 1.4.x so plot might look bad* matplotlib.style.use(**"default"**)  
 **except**:  
 **pass** *#matplotlib.style.use(resource\_filename(\_\_name\_\_, "pandas\_profiling.mplstyle"))  
  
 # Clearing the cache before computing stats* base.clear\_cache()  
  
 **if not** pd.Index(np.arange(0, len(df))).equals(df.index):  
 *# Treat index as any other column* df = df.reset\_index()  
  
 kwargs.update({**'bins'**: bins})  
 *# Describe all variables in a univariate way* **if** pool\_size == 1:  
 local\_multiprocess\_func = partial(multiprocess\_func, \*\*kwargs)  
 ldesc = {col: s **for** col, s **in** map(local\_multiprocess\_func, df.iteritems())}  
 **else**:  
 pool = multiprocessing.Pool(pool\_size)  
 local\_multiprocess\_func = partial(multiprocess\_func, \*\*kwargs)  
 ldesc = {col: s **for** col, s **in** pool.map(local\_multiprocess\_func, df.iteritems())}  
 pool.close()  
  
 *# Get correlations* dfcorrPear = df.corr(method=**"pearson"**)  
 dfcorrSpear = df.corr(method=**"spearman"**)  
  
 *# Check correlations between variable* **if** check\_correlation **is** True:  
 **''' TODO: corr(x,y) > 0.9 and corr(y,z) > 0.9 does not imply corr(x,z) > 0.9  
 If x~y and y~z but not x~z, it would be better to delete only y  
 Better way would be to find out which variable causes the highest increase in multicollinearity.  
 '''** corr = dfcorrPear.copy()  
 **for** x, corr\_x **in** corr.iterrows():  
 **if** correlation\_overrides **and** x **in** correlation\_overrides:  
 **continue  
  
 for** y, corr **in** corr\_x.iteritems():  
 **if** x == y: **break  
  
 if** corr > correlation\_threshold:  
 ldesc[x] = pd.Series([**'CORR'**, y, corr], index=[**'type'**, **'correlation\_var'**, **'correlation'**])  
  
 **if** check\_recoded:  
 categorical\_variables = [(name, data) **for** (name, data) **in** df.iteritems() **if** base.get\_vartype(data)==**'CAT'**]  
 **for** (name1, data1), (name2, data2) **in** itertools.combinations(categorical\_variables, 2):  
 **if** correlation\_overrides **and** name1 **in** correlation\_overrides:  
 **continue** confusion\_matrix=pd.crosstab(data1,data2)  
 **if** confusion\_matrix.values.diagonal().sum() == len(df):  
 ldesc[name1] = pd.Series([**'RECODED'**, name2], index=[**'type'**, **'correlation\_var'**])  
  
 *# Convert ldesc to a DataFrame* names = []  
 ldesc\_indexes = sorted([x.index **for** x **in** ldesc.values()], key=len)  
 **for** idxnames **in** ldesc\_indexes:  
 **for** name **in** idxnames:  
 **if** name **not in** names:  
 names.append(name)  
 variable\_stats = pd.concat(ldesc, join\_axes=pd.Index([names]), axis=1)  
 variable\_stats.columns.names = df.columns.names  
  
 *# General statistics* table\_stats = {}  
  
 table\_stats[**'n'**] = len(df)  
 table\_stats[**'nvar'**] = len(df.columns)  
 table\_stats[**'total\_missing'**] = variable\_stats.loc[**'n\_missing'**].sum() / (table\_stats[**'n'**] \* table\_stats[**'nvar'**])  
 unsupported\_columns = variable\_stats.transpose()[variable\_stats.transpose().type != base.S\_TYPE\_UNSUPPORTED].index.tolist()  
 table\_stats[**'n\_duplicates'**] = sum(df.duplicated(subset=unsupported\_columns)) **if** len(unsupported\_columns) > 0 **else** 0  
  
 memsize = df.memory\_usage(index=True).sum()  
 table\_stats[**'memsize'**] = formatters.fmt\_bytesize(memsize)  
 table\_stats[**'recordsize'**] = formatters.fmt\_bytesize(memsize / table\_stats[**'n'**])  
  
 table\_stats.update({k: 0 **for** k **in** (**"NUM"**, **"DATE"**, **"CONST"**, **"CAT"**, **"UNIQUE"**, **"CORR"**, **"RECODED"**, **"BOOL"**, **"UNSUPPORTED"**)})  
 table\_stats.update(dict(variable\_stats.loc[**'type'**].value\_counts()))  
 table\_stats[**'REJECTED'**] = table\_stats[**'CONST'**] + table\_stats[**'CORR'**] + table\_stats[**'RECODED'**]  
  
 **return** {  
 **'table'**: table\_stats,  
 **'variables'**: variable\_stats.T,  
 **'freq'**: {k: (base.get\_groupby\_statistic(df[k])[0] **if** variable\_stats[k].type != base.S\_TYPE\_UNSUPPORTED **else** None) **for** k **in** df.columns},  
 **'correlations'**: {**'pearson'**: dfcorrPear, **'spearman'**: dfcorrSpear}  
 }

*"""Test module for auto\_insights"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** mock **import** patch  
**from** insights.auto\_insights.auto\_insights\_generator **import** AutoInsight  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** request\_builder **import** RequestBuilder  
  
  
**class** TestAutoInsight(TestCase):  
 *"""Test Class for TestAutoInsight"""* **def** setUp(self):  
 *"""to initialize the class"""* self.json\_value = {**"data\_source\_id"**: 1, **"user\_id"**: 10}  
 self.data\_frame = pd.read\_csv(**"../resources/Power\_Plant\_Time\_series\_What\_If\_Cluster.csv"**,  
 delimiter=**","**)  
  
 **def** test\_run\_insights\_data\_source(self):  
 *"""test cases for run\_insights\_on\_data\_source"""* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame  
 **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** lens\_get\_request:  
 lens\_get\_request.return\_value.json.return\_value = [{**"ID"**: 1, **"dataSourceId"**: 51,  
 **"columnName"**: **"Date"**,  
 **"sentiment"**: **"Good"**,  
 **"timeSeriesDateAttribute"**: **"Y"**,  
 **"timeSeriesKpi"**: **"N"**,  
 **"target"**: **"N"**,  
 **"normalyViewedWith"**: **"None"**,  
 **"index"**: **"Y"**,  
 **"defaultFrequency"**: **"M"**},  
 {**"ID"**: 2, **"dataSourceId"**: 51,  
 **"columnName"**: **"Temperature"**,  
 **"sentiment"**: **"Y"**,  
 **"timeSeriesDateAttribute"**: **"N"**,  
 **"timeSeriesKpi"**: **"Y"**,  
 **"target"**: **"Y"**,  
 **"normalyViewedWith"**: **"None"**,  
 **"index"**: **"Y"**,  
 **"defaultFrequency"**: **"None"**},  
 {**"ID"**: 3, **"dataSourceId"**: 51,  
 **"columnName"**: **"Exhaust\_Vacuum"**,  
 **"sentiment"**: **"Y"**,  
 **"timeSeriesDateAttribute"**: **"N"**,  
 **"timeSeriesKpi"**: **"N"**,  
 **"target"**: **"N"**,  
 **"normalyViewedWith"**: **"None"**,  
 **"index"**: **"Y"**,  
 **"defaultFrequency"**: **"None"**},  
 {**"ID"**: 4, **"dataSourceId"**: 51,  
 **"columnName"**:  
 **"Ambient\_Pressure"**,  
 **"sentiment"**: **"Y"**,  
 **"timeSeriesDateAttribute"**: **"N"**,  
 **"timeSeriesKpi"**: **"N"**,  
 **"target"**: **"N"**,  
 **"normalyViewedWith"**: **"None"**,  
 **"index"**: **"Y"**,  
 **"defaultFrequency"**: **"None"**},  
 {**"ID"**: 5, **"dataSourceId"**: 51,  
 **"columnName"**:  
 **"Relative\_Humidity"**,  
 **"sentiment"**: **"Y"**,  
 **"timeSeriesDateAttribute"**: **"N"**,  
 **"timeSeriesKpi"**: **"N"**,  
 **"target"**: **"N"**,  
 **"normalyViewedWith"**: **"None"**,  
 **"index"**: **"Y"**,  
 **"defaultFrequency"**: **"None"**},  
 {**"ID"**: 6, **"dataSourceId"**: 51,  
 **"columnName"**: **"Energy\_output"**,  
 **"sentiment"**: **"Y"**,  
 **"timeSeriesDateAttribute"**: **"N"**,  
 **"timeSeriesKpi"**: **"Y"**,  
 **"target"**: **"Y"**,  
 **"normalyViewedWith"**: **"None"**,  
 **"index"**: **"Y"**,  
 **"defaultFrequency"**: **"None"**}]  
 exp\_output = {**'auto\_insights'**: [{**'text'**: **'<b>4 key factors</b> found to effect'  
 ' <b>Temperature</b>'**,  
 **'target\_column'**: **'Temperature'**,  
 **'type'**: **'What IF Analysis'**},  
 {**'text'**: **'<b>4 key factors</b> found to effect'  
 ' <b>Energy\_output</b>'**,  
 **'target\_column'**: **'Energy\_output'**,  
 **'type'**: **'What IF Analysis'**},  
 {**'time\_column'**: **'Date'**,  
 **'text'**: **'<b>Increase of 2.3% in Temperature</b>'  
 ' expected in next <b>6 months</b>'**,  
 **'forecast\_bin'**: **'M'**,  
 **'forecast\_column'**: **'Temperature'**,  
 **'type'**: **'Time Series Forecasting'**},  
 {**'time\_column'**: **'Date'**,  
 **'text'**: **'<b>Increase of 1.14% in Energy\_output</b>'  
 ' expected in next <b>6 months</b>'**,  
 **'forecast\_bin'**: **'M'**,  
 **'forecast\_column'**: **'Energy\_output'**,  
 **'type'**: **'Time Series Forecasting'**},  
 {**'text'**: **'<b>3 distinct segments</b> found in your'  
 ' data set'**, **'type'**: **'Data Segmentation'**}],  
 **'data\_source\_id'**: 1}  
 auto\_insight = AutoInsight(self.json\_value)  
 output\_json = auto\_insight.run\_insights\_on\_data\_source()  
 self.assertListEqual(sorted(exp\_output.keys()), sorted(output\_json.keys()))  
  
 **def** test\_negative\_flow(self):  
 **with** self.assertRaises(Exception):  
 AutoInsight({})

*""" This module is written to test main KnowledgeBase  
  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** mock **import** patch  
**from** insights.auto\_insights.knowledge\_base\_operations **import** KnowledgeBase  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** request\_builder **import** RequestBuilder  
  
  
**class** TestKnowledgeBase(TestCase):  
 *"""Test cases for update Knowledge base class"""* **def** setUp(self):  
 *"""to initialize the class"""* self.json\_value = {**"data\_source\_id"**: 139, **"user\_id"**: 10,  
 **"columnName"**: [**"customer\_id"**, **"customer\_name"**],  
 **"dataSetName"**: **"customers\_new"**}  
 self.data\_frame = pd.read\_csv(**"../resources/Power\_Plant\_Time\_series\_What\_If\_Cluster.csv"**,  
 delimiter=**","**)  
  
 **def** test\_update\_knowledge\_base(self):  
 *"""Test function for feedback\_update\_knowledge\_base"""* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame  
 **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** lens\_get\_request:  
 lens\_get\_request.return\_value.json.return\_value = {**"id"**: 1, **"dataSourceId"**: 139,  
 **"dataSourceName"**:  
 **"customers\_new"**,  
 **"columnName"**: **"customer\_id"**,  
 **"sentiment"**: **"Y"**,  
 **"timeSeriesDateAttribute"**: **"Y"**,  
 **"timeSeriesKpi"**: **"Y"**,  
 **"target"**: **"N"**,  
 **"normalyViewedWith"**: **""**,  
 **"defaultFrequency"**: **"Weekly"**,  
 **"relationship"**: [],  
 **"createdAt"**: **''**,  
 **"updatedAt"**: **''**,  
 **"deleteFlag"**: 1,  
 **"normalyViewedWithData"**: None,  
 **"index"**: **"Y"**}  
 *# lens\_get\_request.return\_value.json.return\_value = {}* knowledge\_base = KnowledgeBase(self.json\_value)  
 output\_json = knowledge\_base.feedback\_update\_knowledge\_base()  
 **print**(**"output\_json"**, output\_json)

*Cluster*

*"""This module is created to perform clustering on given dataset.  
clusters added on K-Means algorithm , so it works only on numerical data columns"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** sklearn.cluster **import** KMeans  
*# from sklearn.preprocessing import StandardScaler***from** sklearn.decomposition **import** PCA  
**import** pandas **as** pd  
**import** numpy **as** np  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** insights.utilities **import** get\_cols\_and\_types  
**from** profiling.generate\_profiling **import** GenerateProfile  
**import** constants  
**from** insights.cluster.cluster\_nlg **import** get\_cluster\_nlg  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** Cluster(object):  
 *"""This class performs cluster analysis on given dataset"""* **def** \_\_init\_\_(self, data\_source\_id=None, lens\_id=None):  
 *"""Initialize a class"""* **try**:  
 self.data = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=data\_source\_id,  
 lens\_id=lens\_id,  
 data\_frame\_type=**'WORKING'**)  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(data\_source\_id, lens\_id).\  
 return\_cols\_and\_types()  
 self.cluster\_data = self.data.loc[:, self.data.columns.isin(  
 cols\_json\_file[constants.NUMERIC\_COLS])].values  
 self.data\_source\_id = data\_source\_id  
 self.lens\_id = lens\_id  
 self.temp\_data = pd.DataFrame()  
 self.clusters\_points\_dict = {}  
 self.centroids\_dict = {}  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** identify\_clusters(self):  
 *"""This function identify number of clusters in dataset"""* **try**:  
 wcss = [] *# within cluster sum of errors(wcss)* cluster = 0  
 **for** i **in** range(1, 15):  
 kmeans = KMeans(n\_clusters=i, init=**'k-means++'**, max\_iter=300, n\_init=10,  
 random\_state=0)  
 *# kmeans.fit(self.cluster\_data)* **try**:  
 kmeans.fit(self.cluster\_data)  
 **except** ValueError **as** ex: *# added for bug fix* LOGGER.error(**'Exception: %s'**, ex)  
 self.cluster\_data = self.cluster\_data.astype(float) *# added for bug fix* self.cluster\_data[np.isnan(self.cluster\_data)] = 0 *# added for bug fix* kmeans.fit(self.cluster\_data)  
 wcss.append(kmeans.inertia\_)  
  
 **for** key, \_ **in** enumerate(wcss):  
 diff = ((wcss[key] - wcss[key + 1]) / wcss[key]) -\  
 ((wcss[key + 1] - wcss[key + 2]) / wcss[key + 1])  
 **if** diff < .03: *# threshold to identify no of clusters* cluster = key  
 **break  
 return** cluster  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** clustering(self, clusters=0):  
 *"""This function performs clustering and add cluster\_id in main dataset"""* **try**:  
 **if** clusters == 0 **and** self.identify\_clusters() != 0:  
 clusters = self.identify\_clusters()  
 **elif** clusters == 0:  
 clusters = 1  
  
 kmeans = KMeans(n\_clusters=clusters, init=**'k-means++'**, max\_iter=300,  
 n\_init=10, random\_state=0)  
 *# y\_kmeans = kmeans.fit\_predict(self.cluster\_data)* **try**:  
 y\_kmeans = kmeans.fit\_predict(self.cluster\_data)  
 **except** ValueError **as** ex: *# added for bug fix* LOGGER.error(**'Exception: %s'**, ex)  
 self.cluster\_data = self.cluster\_data.astype(float) *# added for bug fix* self.cluster\_data[np.isnan(self.cluster\_data)] = 0 *# added for bug fix* y\_kmeans = kmeans.fit\_predict(self.cluster\_data)  
  
 self.data[**'cluster\_id'**] = pd.Series(y\_kmeans)  
 **if** self.data[**'cluster\_id'**].isnull().any().any(): *# added for bug fix str < float* self.data = self.data.reset\_index() *# added for bug fix str < float* self.data[**'cluster\_id'**] = pd.Series(y\_kmeans) *# added for bug fix str < float* self.data[**'cluster\_id'**] = **"Cluster\_"** + self.data[**'cluster\_id'**].map(str)  
 DataRegistry.update\_frame(data\_source\_id=self.data\_source\_id,  
 lens\_id=self.lens\_id,  
 data\_frame=self.data,  
 data\_frame\_type=**'CLUSTER'**)  
 cluster\_profile\_list = []  
 **for** \_, value **in** enumerate(pd.unique(self.data[**'cluster\_id'**])):  
 cluster\_profile = {}  
 temp\_df = self.data[self.data[**'cluster\_id'**] == value]  
 cluster\_profile = {constants.CLUSTER\_NAME: value,  
 constants.CLUSTER\_ROW\_COUNT: len(temp\_df),  
 constants.PERCENTAGE: round((len(temp\_df)\*100)/len(self.data),  
 2)  
 }  
 cluster\_profile\_list.append(cluster\_profile)  
 cluster\_array = self.cluster\_data  
 **if** len(cluster\_array[0]) > 2:  
 *# cluster\_array = StandardScaler().fit\_transform(cluster\_array)* pca = PCA(n\_components=2)  
 pca.fit(cluster\_array)  
 cluster\_array = pca.transform(cluster\_array)  
 **if** len(cluster\_array[0]) < 2:  
 **raise** Exception(message\_constants.DATASET\_IS\_NOT\_SUITABLE)  
 *# created a temporary data frame to filters cluster value and respective  
 # 2d values for graph visiualization* self.temp\_data = pd.DataFrame(cluster\_array, columns=list(**'xy'**))  
 self.temp\_data[**'cluster\_id'**] = self.data[**'cluster\_id'**]  
 **for** key, value **in** enumerate(sorted(pd.unique(self.temp\_data[**'cluster\_id'**]))):  
 *# assigning cluster points to draw graph in clusters\_points\_dict* self.clusters\_points\_dict.update({value: (self.temp\_data.loc[  
 self.temp\_data[**'cluster\_id'**] == value,  
 [**'x'**, **'y'**]].values).tolist()})  
 *# assigning centroids points to draw graph in centroids\_dict* self.centroids\_dict.update({value: kmeans.cluster\_centers\_[key, [0, 1]].tolist()})  
 **return** {constants.TOTAL\_CLUSTERS: kmeans.n\_clusters,  
 constants.TOTAL\_RECORDS: len(self.data),  
 constants.CLUSTER\_INFO: cluster\_profile\_list,  
 constants.CLUSTER\_VISUALIZATION: {**"clusters\_points"**: self.clusters\_points\_dict,  
 **"centroids"**: self.centroids\_dict},  
 constants.CLUSTER\_TABLE\_SUMMARY: self.cluster\_summary(self.data\_source\_id,  
 self.lens\_id),  
 **'cluster\_nlg\_text'**:get\_cluster\_nlg(self.cluster\_summary(self.data\_source\_id,  
 self.lens\_id))}  
  
 **except** ValueError **as** value\_error: *# pylint: disable=broad-except* **if "must be between" in** str(value\_error.args[0]) **or "Found array with 0 feature(s)" in**\  
 str(value\_error.args[0]):  
 LOGGER.error(**'Exception: %s'**, value\_error)  
 **raise** ValueError(message\_constants.DATASET\_IS\_NOT\_SUITABLE)  
 **else**:  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** cluster\_col\_profile(data\_source\_id=None, lens\_id=None, data\_frame\_type=**"CLUSTER"**,  
 column\_name=None, user\_id=None):  
 *"""this function runs of full data and gives profile for column then repeat it for  
 cluster and gives column profile per clusters"""* **try**:  
 input\_json = {constants.DATA\_SOURCE\_ID: data\_source\_id,  
 constants.LENS\_ID: lens\_id,  
 constants.COLUMN\_NAME: column\_name,  
 constants.CLUSTERING\_ENABLE: data\_frame\_type,  
 constants.USER\_ID: user\_id}  
 gen\_profile\_obj = GenerateProfile(input\_json)  
 profile\_dict = {**"full\_data"**: gen\_profile\_obj.generate\_profile\_of\_request\_column()}  
 main\_data\_frame = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id, lens\_id,  
 data\_frame\_type)  
 **for** i **in** pd.unique(main\_data\_frame[constants.CLUSTER\_ID]):  
 filtered\_data\_frame = main\_data\_frame[main\_data\_frame[constants.CLUSTER\_ID] == i]  
 DataRegistry.update\_frame(data\_source\_id, lens\_id, filtered\_data\_frame,  
 data\_frame\_type=**"filtered"**)  
 input\_json = {constants.DATA\_SOURCE\_ID: data\_source\_id,  
 constants.LENS\_ID: lens\_id,  
 constants.COLUMN\_NAME: column\_name,  
 constants.CLUSTERING\_ENABLE: **"filtered"**,  
 constants.USER\_ID: user\_id}  
 gen\_profile\_obj = GenerateProfile(input\_json)  
 profile\_dict.update({i: gen\_profile\_obj.generate\_profile\_of\_request\_column()})  
  
 **return** profile\_dict  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** cluster\_summary(data\_source\_id, lens\_id):  
 *"""Function to display cluster column summary min, max, average, median, mode"""* **try**:  
 cluster\_summary\_dict = {}  
 data = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=data\_source\_id,  
 lens\_id=lens\_id,  
 data\_frame\_type=**'CLUSTER'**)  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(data\_source\_id, lens\_id).\  
 return\_cols\_and\_types()  
 df\_summmary = data.groupby(**'cluster\_id'**)[cols\_json\_file[constants.NUMERIC\_COLS]].describe()  
 df\_summmary = df\_summmary.fillna(0)  
 *# fetch multi index values* **for** i **in** df\_summmary.index.values: *# i for Cluster\_0, cluster\_1 index* cluster\_summary\_j\_dict = {}  
 *# J for features - column Name index* **for** j **in** pd.unique(df\_summmary.loc[i].index.get\_level\_values(0)):  
 cluster\_summary\_k\_dict = {}  
 *# k for min, max, std , sum, median index* **for** k **in** pd.unique(df\_summmary.loc[i].index.get\_level\_values(1)):  
 **if** k **in** [**'max'**, **'50%'**, **'min'**, **'mean'**, **'std'**]:  
 cluster\_summary\_k\_dict.update({k: df\_summmary.loc[i].loc[j].loc[k]})  
 cluster\_summary\_j\_dict.update({j: cluster\_summary\_k\_dict})  
 cluster\_summary\_dict.update({i: cluster\_summary\_j\_dict})  
 **return** cluster\_summary\_dict  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module generates NLG from cluster summary  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** pandas **as** pd  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**def** get\_df\_from\_dict(cluster\_dict, columns):  
 *"""  
 This methods takes cluster summary dict and columns as input  
 and will generate pandas df as output.(of the same stucture)  
 """* **try**:  
 tupples\_list = []  
 *# below code is used to append cluster id column* cluster\_list = list(cluster\_dict.keys())  
 tupples\_list.append((**"cluster\_id"**, cluster\_list))  
 **for** column **in** columns:  
 columns\_list = []  
 sample\_dict = {}  
 **for** cluster\_key **in** cluster\_list:  
 **for** inner\_dictionary **in** cluster\_dict[cluster\_key].items():  
 sample\_dict[inner\_dictionary[0]] = inner\_dictionary[1][**"mean"**]  
 series\_from\_dict = pd.Series(sample\_dict)  
 columns\_list.append(series\_from\_dict[column])  
 tupple\_value = (column, columns\_list)  
 tupples\_list.append(tupple\_value)  
 pandas\_df\_for\_pre\_nlg\_fun = pd.DataFrame.from\_dict(  
 dict(tupples\_list), orient=**"columns"** )  
 **return** pandas\_df\_for\_pre\_nlg\_fun  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
**def** get\_insights\_df(pandas\_df\_for\_pre\_nlg\_fun, columns):  
 *"""  
 get\_insights\_df is used to generate insights as pandas df  
 """* **try**:  
 nlg\_df\_fun = pd.DataFrame(  
 columns=[**"cluster\_id"**, **"condition"**, **"value"**, **"variable\_name"**]  
 )  
 *# get min,max across columns for clusters* **for** column\_search **in** columns:  
 **for** condition\_code **in** [**"max"**, **"min"**]:  
 **if** condition\_code == **"max"**:  
 value = pandas\_df\_for\_pre\_nlg\_fun.iloc[  
 pandas\_df\_for\_pre\_nlg\_fun[column\_search].idxmax()  
 ]  
 nlg\_df\_fun = nlg\_df\_fun.append(  
 {  
 **"cluster\_id"**: value[**"cluster\_id"**],  
 **"condition"**: **"max"**,  
 **"value"**: round(value[column\_search], 2),  
 **"variable\_name"**: column\_search,  
 },  
 ignore\_index=True,  
 )  
 **else**:  
 value = pandas\_df\_for\_pre\_nlg\_fun.iloc[  
 pandas\_df\_for\_pre\_nlg\_fun[column\_search].idxmin()  
 ]  
 nlg\_df\_fun = nlg\_df\_fun.append(  
 {  
 **"cluster\_id"**: value[**"cluster\_id"**],  
 **"condition"**: **"min"**,  
 **"value"**: round(value[column\_search], 2),  
 **"variable\_name"**: column\_search,  
 },  
 ignore\_index=True,  
 )  
  
 **return** nlg\_df\_fun  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
**def** generate\_nlg(nlg\_df\_fun):  
 *"""  
 generate\_nlg is used to generate NLG list from pandas df  
 """* **try**:  
 nlg\_list\_fun = []  
 **for** cluster\_id **in** nlg\_df\_fun[**"cluster\_id"**].unique().tolist():  
 cluster\_df = nlg\_df\_fun.query(**"cluster\_id == @cluster\_id"**)  
 length\_of\_unique\_values = len(  
 nlg\_df\_fun.query(**"cluster\_id == @cluster\_id"**)[**"condition"**].unique()  
 )  
 *## condition to handle all maximum values in a cluster id* **if** (  
 length\_of\_unique\_values == 1  
 **and** nlg\_df\_fun.query(**"cluster\_id == @cluster\_id"**)[**"condition"**].unique()[  
 0  
 ]  
 == **"max"** ):  
 val = str(cluster\_id) + **" has maximum average values which are "  
 for** \_, df\_value **in** nlg\_df\_fun.query(  
 **"cluster\_id == @cluster\_id"** ).iterrows():  
 val = (  
 val  
 + str(df\_value[**"value"**])  
 + **" for "** + str(df\_value[**"variable\_name"**])  
 + **", "** )  
 val = val.strip().rstrip(**","**)  
 nlg\_list\_fun.append(val)  
  
 *## condition to handle all minimum values in a cluster id* **elif** (  
 length\_of\_unique\_values == 1  
 **and** nlg\_df\_fun.query(**"cluster\_id == @cluster\_id"**)[**"condition"**].unique()[  
 0  
 ]  
 == **"min"** ):  
 val = str(cluster\_id) + **" has minimum average values which are "  
 for** \_, df\_value **in** nlg\_df\_fun.query(  
 **"cluster\_id == @cluster\_id"** ).iterrows():  
 val = (  
 val  
 + str(df\_value[**"value"**])  
 + **" for "** + str(df\_value[**"variable\_name"**])  
 + **", "** )  
 val = val.strip().rstrip(**","**)  
 nlg\_list\_fun.append(val)  
  
 *## condition to handle mixture of maximum and minimum in a cluster id* **else**:  
 *# handle all maximum values in below code* val = str(cluster\_id) + **" has maximum average values for "  
 for** \_, df\_value **in** cluster\_df.query(**'condition == "max"'**).iterrows():  
 val = (  
 val  
 + str(df\_value[**"variable\_name"**])  
 + **"("** + str(df\_value[**"value"**])  
 + **"), "** )  
 val = val.strip().rstrip(**","**).strip()  
 *# handle all minimum values in below code* val = val + **" and minimum average values for "  
 for** \_, df\_value **in** cluster\_df.query(**'condition == "min"'**).iterrows():  
 val = (  
 val  
 + str(df\_value[**"variable\_name"**])  
 + **"("** + str(df\_value[**"value"**])  
 + **"), "** )  
 val = val.strip().rstrip(**","**).strip()  
 nlg\_list\_fun.append(val)  
  
 **return** nlg\_list\_fun  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
**def** get\_cluster\_nlg(cluster\_summary):  
 *"""  
 function used to generate nlg for cluster cluster\_summary.  
 Takes cluster summary dict as input and generates insights list.  
 Internally uses get\_df\_from\_dict , generate\_nlg methods and get\_insights\_df.  
 """  
 # check if only one cluster is formed, here generate a fixed message.* **try**:  
 **if** len(cluster\_summary.keys()) == 1:  
 insights\_list = []  
 val = **"No insights(Please try running with K value of 2 or more)"** insights\_list.append(val)  
 *# this flow is when number of clusters are 2 or more.* **else**:  
 **try**:  
 first\_key = list(cluster\_summary.keys())[0]  
 columns = list(cluster\_summary[first\_key].keys())  
 *# get\_df\_from\_dict is used to generate df from dict* pandas\_df\_for\_pre\_nlg = get\_df\_from\_dict(cluster\_summary, columns)  
 *# generate\_nlg is used to generate NLG from df* insights\_df = get\_insights\_df(pandas\_df\_for\_pre\_nlg, columns)  
 insights\_list = generate\_nlg(insights\_df)  
 **except** IndexError **as** ex: *# will occur when cluster\_summary is null* LOGGER.error(**'Exception: %s'**, ex)  
 **raise** IndexError(message\_constants.NONE\_INPUT\_DATASET)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** insights\_list  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""Test module for cluster module"""***from** unittest **import** TestCase  
**from** unittest.mock **import** patch  
**import** pandas **as** pd  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** request\_builder **import** RequestBuilder  
**from** insights.cluster.cluster **import** Cluster  
**import** constants  
  
  
**class** TestCluster(TestCase):  
 *"""Test class for Cluster Class"""* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df, it runs once only"""* cls.test\_df = pd.read\_csv(**"../resources/iris\_cluster.csv"**)  
 json = {**"ORIGINAL"**: cls.test\_df, **"WORKING"**: cls.test\_df,  
 **"CLUSTER"**: cls.test\_df, constants.LAST\_USED\_BY: 0} *# Need to remove* DataRegistry.pandas\_df\_map.update({**"5\_5"**: json}) *# Need to remove* cls.cluster\_obj = Cluster(data\_source\_id=5, lens\_id=5)  
 cls.knowledgebase\_response = [{**'timeSeriesDateAttribute'**: **'N'**, **'normalyViewedWithData'**: {},  
 **'createdAt'**: 1548330396000, **'updatedAt'**: 1548330414000,  
 **'normalyViewedWith'**: **'{}'**, **'index'**: **'N'**, **'lensId'**: -1,  
 **'defaultFrequency'**: **'None'**, **'dataSourceId'**: 616, **'id'**: 160,  
 **'timeSeriesKpi'**: **'N'**, **'dataSourceName'**: **'profile\_test'**,  
 **'sentiment'**: **'Good'**, **'columnName'**: **'ball'**,  
 **'booleanIndicator'**: None, **'target'**: **'Y'**,  
 **'categoricalIndicator'**: **'Y'**, **'relationship'**: [],  
 **'deleteFlag'**: 1}]  
  
 **def** test\_identify\_clusters(self):  
 *"""test function for identify\_clusters main function"""* self.assertEqual(self.cluster\_obj.identify\_clusters(), 3)  
  
 **def** test\_clustering(self):  
 *"""test function for clustering main function"""* expected\_output\_dict = {**'cluster\_info'**: [{**'cluster\_name'**: **'Cluster\_1'**, **'percentage'**: 33.33,  
 **'cluster\_row\_count'**: 50},  
 {**'cluster\_name'**: **'Cluster\_0'**, **'percentage'**: 41.33,  
 **'cluster\_row\_count'**: 62},  
 {**'cluster\_name'**: **'Cluster\_2'**, **'percentage'**: 25.33,  
 **'cluster\_row\_count'**: 38}],  
 **'cluster\_nlg\_text'**: **''**,  
 **'cluster\_visualization'**: **''**,  
 **'total\_records'**: 150,  
 **'total\_clusters'**: 3,  
 **'cluster\_table\_summary'**: **""**}  
 fun\_output\_dict = self.cluster\_obj.clustering(clusters=0)  
 self.assertListEqual(sorted(fun\_output\_dict.keys()), sorted(expected\_output\_dict.keys()))  
 self.assertListEqual(fun\_output\_dict[**'cluster\_info'**],  
 expected\_output\_dict[**'cluster\_info'**])  
 self.assertEqual(fun\_output\_dict[**'total\_records'**], 150)  
 self.assertEqual(fun\_output\_dict[**'total\_clusters'**], 3)  
 self.assertDictEqual(fun\_output\_dict[**'cluster\_visualization'**],  
 fun\_output\_dict[**'cluster\_visualization'**])  
 self.assertDictEqual(fun\_output\_dict[**'cluster\_table\_summary'**],  
 fun\_output\_dict[**'cluster\_table\_summary'**])  
  
 **def** test\_cluster\_col\_profile(self):  
 *"""Test function cluster\_col\_profile clusters"""* **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** knowledgebase\_get\_request:  
 knowledgebase\_get\_request.return\_value.json.return\_value = self.knowledgebase\_response  
 fun\_opt\_dict = Cluster.cluster\_col\_profile(data\_source\_id=5, lens\_id=5,  
 data\_frame\_type=**"CLUSTER"**,  
 column\_name=**"sepal width"**,  
 user\_id=1)  
 self.assertEqual(len(fun\_opt\_dict), 4)

*"""  
This module is used to check unit test case for cluster NLG  
"""***import** unittest  
**from** insights.cluster.cluster\_nlg **import** \*  
  
  
**class** TestClusternlg(unittest.TestCase):  
 *"""  
 This class is used for implement test method of auto features step  
 """* **def** setUp(self):  
 self.sample\_dict = {0: {  
 **'age'**: {**'50%'**: 29.0, **'max'**: 77.0, **'mean'**: 34.707317073170735, **'min'**: 18.0,  
 **'std'**: 14.379227904235723},  
 **'salary'**: {**'50%'**: 1000000.0, **'max'**: 3300000.0, **'mean'**: 1513414.6341463414, **'min'**: 0.0,  
 **'std'**: 1043732.747776215},  
 **'total\_vacation\_days'**: {**'50%'**: 18.0, **'max'**: 35.0, **'mean'**: 17.48780487804878, **'min'**: 8.0,  
 **'std'**: 7.05025514155166}}, 1: {  
 **'age'**: {**'50%'**: 46.0, **'max'**: 55.0, **'mean'**: 45.888888888888886, **'min'**: 30.0,  
 **'std'**: 8.115485882626567},  
 **'salary'**: {**'50%'**: 5600000.0, **'max'**: 9900000.0, **'mean'**: 6277777.777777778,  
 **'min'**: 4100000.0, **'std'**: 2094503.3885015426},  
 **'total\_vacation\_days'**: {**'50%'**: 22.0, **'max'**: 30.0, **'mean'**: 20.11111111111111,  
 **'min'**: 10.0, **'std'**: 7.1840873540841015}}}  
  
 **def** test\_good\_flow(self):  
 *"""  
 used to test cluster NLG* **:return***:  
 """* insights = get\_cluster\_nlg(self.sample\_dict)  
 **if** len(insights) > 1:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_empty\_input(self):  
 *"""  
 used to test cluster NLG* **:return***:  
 """* cluster\_summary = {}  
 **try**:  
 insights = get\_cluster\_nlg(cluster\_summary)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 self.assertEqual(1, 1)  
  
 **def** test\_wrong\_input(self):  
 *"""  
 used to test cluster NLG # here mean is changed to mean1* **:return***:  
 """* cluster\_summary = {0: {  
 **'age'**: {**'50%'**: 29.0, **'max'**: 77.0, **'mean1'**: 34.707317073170735, **'min'**: 18.0,  
 **'std'**: 14.379227904235723},  
 **'salary'**: {**'50%'**: 1000000.0, **'max'**: 3300000.0, **'mean'**: 1513414.6341463414, **'min'**: 0.0,  
 **'std'**: 1043732.747776215},  
 **'total\_vacation\_days'**: {**'50%'**: 18.0, **'max'**: 35.0, **'mean'**: 17.48780487804878, **'min'**: 8.0,  
 **'std'**: 7.05025514155166}}, 1: {  
 **'age'**: {**'50%'**: 46.0, **'max'**: 55.0, **'mean'**: 45.888888888888886, **'min'**: 30.0,  
 **'std'**: 8.115485882626567},  
 **'salary'**: {**'50%'**: 5600000.0, **'max'**: 9900000.0, **'mean'**: 6277777.777777778,  
 **'min'**: 4100000.0, **'std'**: 2094503.3885015426},  
 **'total\_vacation\_days'**: {**'50%'**: 22.0, **'max'**: 30.0, **'mean'**: 20.11111111111111,  
 **'min'**: 10.0, **'std'**: 7.1840873540841015}}}  
 **try**:  
 insights = get\_cluster\_nlg(cluster\_summary)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 self.assertEqual(1, 1)  
  
 **def** test\_single\_input(self):  
 *"""  
 used to test cluster NLG #* **:return***:  
 """* cluster\_summary = {0: {  
 **'age'**: {**'50%'**: 29.0, **'max'**: 77.0, **'mean'**: 34.707317073170735, **'min'**: 18.0,  
 **'std'**: 14.379227904235723},  
 **'salary'**: {**'50%'**: 1000000.0, **'max'**: 3300000.0, **'mean'**: 1513414.6341463414, **'min'**: 0.0,  
 **'std'**: 1043732.747776215},  
 **'total\_vacation\_days'**: {**'50%'**: 18.0, **'max'**: 35.0, **'mean'**: 17.48780487804878, **'min'**: 8.0,  
 **'std'**: 7.05025514155166}}}  
 **try**:  
 insights = get\_cluster\_nlg(cluster\_summary)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 self.assertEqual(1, 1)  
  
 **def** test\_error\_input\_2(self):  
 *"""  
 used to test cluster NLG #* **:return***:  
 """* cluster\_summary = {0: {0: 1}, 1: {0: 1}}  
 **with** self.assertRaises(Exception):  
 insights = get\_cluster\_nlg(cluster\_summary)  
 self.assertEqual(1, 2)  
  
 **with** self.assertRaises(Exception):  
 get\_df\_from\_dict(None, None)  
  
 **with** self.assertRaises(Exception):  
 get\_insights\_df(None, None)  
  
 **with** self.assertRaises(Exception):  
 generate\_nlg(None)  
  
 **def** test\_good\_flow\_2(self):  
 *"""  
 used to test cluster NLG* **:return***:  
 """* sample\_dict = {0: {  
 **'BMI'**: {**'50%'**: 26.09, **'max'**: 38.53, **'mean'**: 26.259999999999994, **'min'**: 20.68,  
 **'std'**: 4.581528129347238},  
 **'age'**: {**'50%'**: 43.0, **'max'**: 63.0, **'mean'**: 46.65217391304348, **'min'**: 35.0,  
 **'std'**: 8.952443524495898},  
 **'cigsPerDay'**: {**'50%'**: 1.0, **'max'**: 43.0, **'mean'**: 11.73913043478261, **'min'**: 0.0,  
 **'std'**: 14.626306907587459},  
 **'diaBP'**: {**'50%'**: 84.5, **'max'**: 110.0, **'mean'**: 83.97826086956522, **'min'**: 63.0,  
 **'std'**: 12.318203695851523},  
 **'glucose'**: {**'50%'**: 83.0, **'max'**: 113.0, **'mean'**: 84.47826086956522, **'min'**: 65.0,  
 **'std'**: 11.870014035441617},  
 **'heartRate'**: {**'50%'**: 75.0, **'max'**: 96.0, **'mean'**: 77.26086956521739, **'min'**: 60.0,  
 **'std'**: 11.759782741129918},  
 **'sysBP'**: {**'50%'**: 132.0, **'max'**: 180.0, **'mean'**: 133.1086956521739, **'min'**: 96.0,  
 **'std'**: 19.91000006948263},  
 **'totChol'**: {**'50%'**: 215.0, **'max'**: 237.0, **'mean'**: 212.65217391304347, **'min'**: 179.0,  
 **'std'**: 17.823908090111527}}, 1: {  
 **'BMI'**: {**'50%'**: 26.335, **'max'**: 32.8, **'mean'**: 26.39666666666666, **'min'**: 21.68,  
 **'std'**: 3.3929390348365596},  
 **'age'**: {**'50%'**: 46.0, **'max'**: 65.0, **'mean'**: 48.05555555555556, **'min'**: 36.0,  
 **'std'**: 7.320175361305318},  
 **'cigsPerDay'**: {**'50%'**: 12.5, **'max'**: 35.0, **'mean'**: 11.833333333333334, **'min'**: 0.0,  
 **'std'**: 12.025707757085382},  
 **'diaBP'**: {**'50%'**: 82.5, **'max'**: 121.0, **'mean'**: 85.30555555555556, **'min'**: 68.0,  
 **'std'**: 13.957012667988057},  
 **'glucose'**: {**'50%'**: 76.0, **'max'**: 89.0, **'mean'**: 75.11111111111111, **'min'**: 61.0,  
 **'std'**: 8.710297414310928},  
 **'heartRate'**: {**'50%'**: 75.0, **'max'**: 98.0, **'mean'**: 77.05555555555556, **'min'**: 60.0,  
 **'std'**: 10.518735230514798},  
 **'sysBP'**: {**'50%'**: 128.75, **'max'**: 182.0, **'mean'**: 129.47222222222223, **'min'**: 100.0,  
 **'std'**: 22.63552142243159},  
 **'totChol'**: {**'50%'**: 265.5, **'max'**: 332.0, **'mean'**: 272.5, **'min'**: 243.0,  
 **'std'**: 26.122449214060623}}, 2: {  
 **'BMI'**: {**'50%'**: 30.810000000000002, **'max'**: 40.11, **'mean'**: 30.810000000000002,  
 **'min'**: 21.51, **'std'**: 13.152186130069783},  
 **'age'**: {**'50%'**: 52.5, **'max'**: 53.0, **'mean'**: 52.5, **'min'**: 52.0, **'std'**: 0.7071067811865476},  
 **'cigsPerDay'**: {**'50%'**: 0.0, **'max'**: 0.0, **'mean'**: 0.0, **'min'**: 0.0, **'std'**: 0.0},  
 **'diaBP'**: {**'50%'**: 95.0, **'max'**: 98.0, **'mean'**: 95.0, **'min'**: 92.0,  
 **'std'**: 4.242640687119285},  
 **'glucose'**: {**'50%'**: 220.0, **'max'**: 225.0, **'mean'**: 220.0, **'min'**: 215.0,  
 **'std'**: 7.0710678118654755},  
 **'heartRate'**: {**'50%'**: 75.5, **'max'**: 76.0, **'mean'**: 75.5, **'min'**: 75.0,  
 **'std'**: 0.7071067811865476},  
 **'sysBP'**: {**'50%'**: 183.0, **'max'**: 206.0, **'mean'**: 183.0, **'min'**: 160.0,  
 **'std'**: 32.526911934581186},  
 **'totChol'**: {**'50%'**: 244.5, **'max'**: 311.0, **'mean'**: 244.5, **'min'**: 178.0,  
 **'std'**: 94.04520189781083}}}  
  
 insights = get\_cluster\_nlg(sample\_dict)  
 **if** len(insights) > 1:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*Time\_series*

*""" this module is designed to perform  
time series forecasting using Long short term memory LSTM"""***import** math  
**import** random  
**import** numpy  
**from** keras.models **import** Sequential  
**from** keras.layers **import** Dense  
**from** keras.layers **import** LSTM  
**from** tensorflow **import** set\_random\_seed  
**from** sklearn.preprocessing **import** MinMaxScaler  
**from** sklearn.metrics **import** mean\_squared\_error  
set\_random\_seed(7)  
random.seed(7)  
numpy.random.seed(7)  
PYTHONHASHSEED = 0  
  
  
**class** LstmTimeSeries:  
 *"""class to perform time series forecasting using lstm model"""* **def** \_\_init\_\_(self, data\_frame):  
 *"""function to initialize class variables"""* **try**:  
  
 self.original\_data = data\_frame.values  
 self.dataset = data\_frame.values  
 self.dataset = self.dataset.astype(**'float32'**)  
 self.look\_back = 1  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 **def** create\_dataset(self, dataset, look\_back=1):  
 *"""convert an array of values into a data set matrix"""* **try**:  
 data\_x, data\_y = [], []  
 self.look\_back = look\_back  
 **for** i **in** range(len(dataset)-look\_back-1):  
 temp = dataset[i:(i+look\_back), 0]  
 data\_x.append(temp)  
 data\_y.append(dataset[i + look\_back, 0])  
 **return** numpy.array(data\_x), numpy.array(data\_y)  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 **def** data\_transform(self):  
 *"""function to normalize your data set"""* **try**:  
 scaler = MinMaxScaler(feature\_range=(0, 1))  
 scaled\_dataset = scaler.fit\_transform(self.dataset)  
 **return** scaled\_dataset, scaler  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 @staticmethod  
 **def** inverse\_transform(train\_predict, test\_predict, train\_y, test\_y, scaler):  
 *"""function to inverse/ revert transformation"""* **try**:  
 *# invert predictions* train\_predict = scaler.inverse\_transform(train\_predict)  
 *# print("train\_predict", train\_predict)* train\_y = scaler.inverse\_transform([train\_y])  
 test\_predict = scaler.inverse\_transform(test\_predict)  
 *# print("test\_predict", test\_predict)* test\_y = scaler.inverse\_transform([test\_y])  
 **return** train\_predict, test\_predict, train\_y, test\_y  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 **def** train\_test\_data(self, scaled\_dataset, train\_size=0.70):  
 *"""function to split your data in train and test data set"""* **try**:  
 *# split into train and test sets* train\_size = int(len(scaled\_dataset) \* train\_size)  
 *# test\_size = len(scaled\_dataset) - train\_size* train, test = scaled\_dataset[0:train\_size, :],\  
 scaled\_dataset[train\_size:len(scaled\_dataset), :]  
 *# reshape into X=t and Y=t+1* train\_x, train\_y = self.create\_dataset(train)  
 test\_x, test\_y = self.create\_dataset(test)  
 *# reshape input to be [samples, time steps, features]* train\_x = numpy.reshape(train\_x, (train\_x.shape[0], 1, train\_x.shape[1]))  
 test\_x = numpy.reshape(test\_x, (test\_x.shape[0], 1, test\_x.shape[1]))  
 **return** train\_x, test\_x, train\_y, test\_y  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 **def** fit\_n\_forecast\_lstm(self, train\_x, train\_y, test\_x, model\_parameter):  
 *"""function to train and predict values"""* **try**:  
 *# create and fit the LSTM network* model = Sequential()  
 model.add(LSTM(model\_parameter[**"neurons"**], input\_shape=(1, self.look\_back)))  
 model.add(Dense(1))  
 model.compile(loss=**'mean\_squared\_error'**, optimizer=**'adam'**)  
 model.fit(train\_x, train\_y, epochs=model\_parameter[**"epochs"**],  
 batch\_size=model\_parameter[**"batch\_size"**], verbose=2)  
 *# make predictions* train\_predict = model.predict(train\_x)  
 test\_predict = model.predict(test\_x)  
 **return** train\_predict, test\_predict  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 @staticmethod  
 **def** calculate\_rmse(train\_predict, test\_predict, train\_y, test\_y):  
 *"""Function to calculate model rmse """* **try**:  
 *# calculate root mean squared error* train\_rmse = math.sqrt(mean\_squared\_error(train\_y[0], train\_predict[:, 0]))  
 test\_rmse = math.sqrt(mean\_squared\_error(test\_y[0], test\_predict[:, 0]))  
 **return** train\_rmse, test\_rmse  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 *# def plot\_data(self, train\_predict, test\_predict):  
 # """transform your data to plot graph"""  
 # try:  
 # # shift train predictions for plotting  
 # train\_predict\_plot = numpy.empty\_like(self.original\_data)  
 # train\_predict\_plot[:, :] = numpy.nan  
 # train\_predict\_plot[self.look\_back:len(train\_predict) + self.look\_back, :] =\  
 # train\_predict  
 # # shift test predictions for plotting  
 # test\_predict\_plot = numpy.empty\_like(self.original\_data)  
 # test\_predict\_plot[:, :] = numpy.nan  
 # test\_predict\_plot[len(train\_predict) + (self.look\_back \* 2)  
 # + 1:len(self.original\_data) - 1, :] = test\_predict  
 # return train\_predict\_plot, test\_predict\_plot  
 # except Exception as ex:  
 # raise Exception(ex.args[0])* **def** lstm\_forecast(self, test\_size=.80):  
 *"""Main function to run lstm forecasting """* **try**:  
 *# get normalized data set* scaled\_dataset, scaler = self.data\_transform()  
 *# divide data to train and test data set* train\_x, test\_x, train\_y, test\_y = self.train\_test\_data(scaled\_dataset,  
 train\_size=test\_size)  
 model\_parameter = {**"batch\_size"**: 1, **"epochs"**: 50, **"neurons"**: 4}  
 train\_predict, test\_predict = \  
 self.fit\_n\_forecast\_lstm(train\_x, train\_y, test\_x, model\_parameter)  
 train\_predict, test\_predict, train\_y, test\_y = \  
 self.inverse\_transform(train\_predict, test\_predict, train\_y, test\_y, scaler)  
 train\_rmse, test\_rmse = self.calculate\_rmse(train\_predict, test\_predict,  
 train\_y, test\_y)  
 *# train\_predict\_plot, test\_predict\_plot = self.plot\_data(train\_predict, test\_predict)* test\_predict = [item **for** sublist **in** test\_predict **for** item **in** sublist]  
 **return** test\_predict, train\_rmse, test\_rmse  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])

*"""This module is design to run test cases for LstmTimeSeries class"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** insights.time\_series.lstm\_time\_series **import** LstmTimeSeries  
  
  
**class** TestLstmTimeSeries(TestCase):  
 *"""Test class for LstmTimeSeries main class"""* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df, it runs once only"""* cls.test\_df = pd.read\_csv(**"../resources/ts\_passengers\_dataset\_new.csv"**,  
 usecols=[1], engine=**'python'**, nrows=50)  
 cls.lstm\_time\_series = LstmTimeSeries(cls.test\_df)  
  
 @classmethod  
 **def** tearDownClass(cls):  
 *"""this function runs once after execution of all functions"""* **pass  
  
 def** test\_lstm\_forecast(self):  
 *"""Test function for lstm\_forecast"""* expected\_prediction = [3.2196138, 3.2336934, 3.2336934, 3.2336934, 3.2196138, 3.2196138,  
 3.2336934, 3.1784012]  
 train\_rmse\_expected = 1.811618402408805  
 test\_rmse\_expected = 1.6207938695364725  
 test\_predict, train\_rmse, test\_rmse = self.lstm\_time\_series.lstm\_forecast()  
 self.assertEqual(len(test\_predict), len(expected\_prediction))  
 self.assertEqual(train\_rmse, train\_rmse\_expected)  
 self.assertEqual(test\_rmse, test\_rmse\_expected)

*"""Test cases for time series analysis"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** insights.time\_series.time\_series\_forecasting **import** TimeSeriesForecast  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestTimeSeriesForecast(TestCase):  
 *"""Test class for TimeSeriesForecast"""* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df, it runs once only"""* cls.test\_df = pd.read\_csv(**"../resources/ts\_passengers\_dataset\_new.csv"**)  
 json = {**"ORIGINAL"**: cls.test\_df, **"WORKING"**: cls.test\_df} *# Need to remove* DataRegistry.pandas\_df\_map.update({**"9\_9"**: json}) *# Need to remove* cls.time\_series\_obj = TimeSeriesForecast(data\_source\_id=9, lens\_id=9)  
  
 @classmethod  
 **def** tearDownClass(cls):  
 *"""this function runs once after execution of all functions"""* **pass  
  
 def** test\_df\_to\_col\_list(self):  
 *""""test function for df\_to\_col\_list"""* self.test\_df[**"ts\_datetime"**] = pd.to\_datetime(self.test\_df[**"ts\_datetime"**])  
 ts\_data = self.test\_df  
 ts\_data = ts\_data.reset\_index().set\_index(**"ts\_datetime"**).resample(**"A"**)[**"passengers"**].sum()  
 ts\_dataframe = pd.DataFrame({**"ts\_datetime"**: ts\_data.index, **"passengers"**: ts\_data.values})  
 actual\_value = self.time\_series\_obj.df\_to\_col\_list(ts\_dataframe, **"A"**, **"ts\_datetime"**)  
 expected\_value = [**'2012-12-31'**, **'2013-12-31'**, **'2014-12-31'**]  
 self.assertEqual(expected\_value, actual\_value)  
  
 **def** test\_apply\_filters(self):  
 *""""test function for apply\_filters"""* filter\_dict = {**"city"**: [**"Pune"**], **"crowd\_intensity"**: [**"Low"**, **"Medium"**]}  
 self.assertEqual(self.time\_series\_obj.apply\_filters(filter\_dict), None)  
  
 **def** test\_time\_series\_forecast(self):  
 *""""test function for time\_series\_forecast"""* ts\_index, ts\_data, confidence\_int, accuracy\_dict = \  
 self.time\_series\_obj.time\_series\_forecast(**"ts\_datetime"**, **"passengers"**, **"A"**, 6)  
 expected\_ts\_index = [**'2012-12-31'**, **'2013-12-31'**, **'2014-12-31'**, **'2015-12-31'**, **'2016-12-31'**,  
 **'2017-12-31'**, **'2018-12-31'**, **'2019-12-31'**, **'2020-12-31'**]  
 expected\_ts\_data = [34932.0, 683456.0, 843078.0, 520488.67, 520488.67, 520488.67,  
 520488.67, 520488.67, 520488.67]  
 expected\_acc\_dict = {**'training\_data\_length'**: 2, **'data\_length'**: 3,  
 **'test\_data\_length'**: 1, **'mean\_absolute\_error'**: 483884.0,  
 **'mean\_absolute\_percentage\_error'**: 57.39,  
 **'mean\_squared\_error'**: 234143725456.0,  
 **'root\_mean\_squared\_error'**: 483884.0,  
 **'fitted'**: [359193.99999999994]}  
 expected\_confidence\_int = [[0, 0], [0, 0], [0, 0], [-164459.63, 1205436.97],  
 [-164459.63, 1205436.97], [-164459.63, 1205436.97],  
 [-164459.63, 1205436.97], [-164459.63, 1205436.97],  
 [-164459.63, 1205436.97]]  
  
 self.assertListEqual(expected\_ts\_index, ts\_index)  
 self.assertListEqual(expected\_ts\_data, ts\_data)  
 self.assertListEqual(expected\_confidence\_int, confidence\_int)  
 self.assertDictEqual(expected\_acc\_dict, accuracy\_dict)  
  
 **def** test\_negative(self):  
 **with** self.assertRaises(Exception):  
 self.time\_series\_obj.time\_series\_forecast(None, None, None)

*"""This module is designed to perform time series forecasting"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** math **import** sqrt  
**from** pmdarima.arima **import** auto\_arima  
**from** sklearn.metrics **import** mean\_squared\_error  
**from** sklearn.metrics **import** mean\_absolute\_error  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**import** constants  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** TimeSeriesForecast(object):  
 *"""  
 Class is design to perform time series forecasting  
 """* **def** \_\_init\_\_(self, data\_source\_id=None, lens\_id=None):  
 **try**:  
 self.data = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=data\_source\_id,  
 lens\_id=lens\_id,  
 data\_frame\_type=**'WORKING'**)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** df\_to\_col\_list(ts\_dataframe, ts\_freq, ts\_date):  
 *"""function to convert time series dataframe to column list.  
 This function is specially design to handle datatimeindex and fetch date  
 from dataframe column"""* **try**:  
 ts\_date\_list = []  
 **for** \_, value **in** ts\_dataframe.iterrows():  
 **if** ts\_freq **not in** (**'H'**, **'S'**, **'T'**, **'L'**): *# pragma: no cover* ts\_date\_list.append(str(value[ts\_date])[0:10])  
 **else**:  
 ts\_date\_list.append(str(value[ts\_date]))  
 **return** ts\_date\_list  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** apply\_filters(self, filter\_dict=None):  
 *"""This function is designed to impose filters on dataframe"""  
 # if filter\_dict is not None:  
 # apply filters on dataframe* **try**:  
 **for** key, value **in** filter\_dict.items():  
 self.data = self.data[self.data[key].isin(value)]  
 **return** None  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** time\_series\_forecast(self, ts\_date, ts\_kpi, ts\_freq,  
 forecast\_periods=6):  
 *"""Function to forecast time series kpi* **:param***: time series date column name* **:param***: time series KPI column name* **:param***: time series frequency* **:param***: forecast periods* **:param***: dictionary of filters format filter\_dict = {"col1": ['val1', 'val2'],  
 "col2" : [ "val1", "val2"] }  
 """  
  
 # convert your data per time series input frequency (ex. monthly to weekly)* **try**:  
 self.data[ts\_date] = pd.to\_datetime(self.data[ts\_date])  
 ts\_data = self.data  
 ts\_data = ts\_data.reset\_index().set\_index(ts\_date).resample(ts\_freq)[ts\_kpi].sum()  
 model\_accuracy\_dict = TimeSeriesForecast.model\_accuracy(ts\_data)  
  
 *# Time series forecasting - auto rima to get best model* stepwise\_fit = auto\_arima(ts\_data, start\_p=1, start\_q=1, max\_p=3, max\_q=3, m=12,  
 start\_P=0, seasonal=True, d=None, D=None, trace=True,  
 error\_action=**'ignore'**,  
 suppress\_warnings=True, *# don't want convergence warnings* stepwise=True) *# set to stepwise* predictions, confidence\_int = stepwise\_fit.predict(n\_periods=forecast\_periods,  
 return\_conf\_int=True)  
 predictions = np.round(predictions, 2)  
 confidence\_int = np.round(confidence\_int, 2)  
 *# created series with index for all predicted values* predicted\_series = pd.Series(predictions,  
 index=(pd.date\_range(max(ts\_data.index)+1,  
 periods=forecast\_periods,  
 freq=ts\_freq)))  
 ts\_data = ts\_data.append(predicted\_series)  
 *# converted into dataframe to get index values in formatted list* ts\_dataframe = pd.DataFrame({ts\_date: ts\_data.index, ts\_kpi: ts\_data.values})  
 *# convert array list to list of list* confidence\_int = [value.tolist() **for** value **in** list(confidence\_int)]  
 **for** i **in** range(len(ts\_data)-forecast\_periods):  
 confidence\_int.insert(i, [0, 0])  
 **return** self.df\_to\_col\_list(ts\_dataframe, ts\_freq, ts\_date),\  
 list(ts\_data.values), confidence\_int, model\_accuracy\_dict  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** model\_accuracy(ts\_data):  
 *""" Function to calculate model accuracy, RMSE, AMSE, MSE for Auto Arima Model"""* **try**:  
 accuracy\_dict = {}  
 *# divide into train and test set* train = ts\_data[:int(0.8 \* (len(ts\_data)))]  
 test = ts\_data[int(0.8 \* (len(ts\_data))):]  
 stepwise\_fit = auto\_arima(train, start\_p=1, start\_q=1, max\_p=3, max\_q=3, m=12,  
 start\_P=0, seasonal=True, d=None, D=None, trace=True,  
 error\_action=**'ignore'**,  
 suppress\_warnings=True, *# don't want convergence warnings* stepwise=True) *# set to stepwise* predictions, \_ = stepwise\_fit.predict(n\_periods=len(test), return\_conf\_int=True)  
 accuracy\_dict = {constants.MSE: round(mean\_squared\_error(test, predictions), 2)}  
 accuracy\_dict.update({constants.RMSE: round(sqrt(mean\_squared\_error(test,  
 predictions)), 2)})  
 accuracy\_dict.update({constants.MAE: round(mean\_absolute\_error(test, predictions), 2)})  
 accuracy\_dict.update({constants.MAPE: round(TimeSeriesForecast.  
 mean\_absolute\_percentage\_error(test,  
 predictions),  
 2)})  
 accuracy\_dict.update({constants.TRAINING\_DATA\_LENGTH: len(train)})  
 accuracy\_dict.update({constants.TEST\_DATA\_LENGTH: len(test)})  
 accuracy\_dict.update({constants.DATA\_LENGTH: len(ts\_data)})  
 accuracy\_dict.update({constants.FITTED: predictions.tolist()})  
 **return** accuracy\_dict  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** mean\_absolute\_percentage\_error(y\_true, y\_pred):  
 *"""Function to calculate Mean absolute percentage error"""* **try**:  
 y\_true, y\_pred = np.array(y\_true), np.array(y\_pred)  
 **return** np.mean(np.abs((y\_true - y\_pred) / y\_true)) \* 100  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*Utilities*

*"""  
@author: Phani Koratamaddi  
This script identifies all columns which need not be analyized  
"""***import** logging  
**import** pandas **as** pd  
**from** sklearn.preprocessing **import** LabelEncoder  
**import** numpy **as** np  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
*# Identify const\_cols, zero\_var\_cols, high\_missing\_cols to be excluded from analysis  
# Input threshold for high\_missing\_cols*LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** ColsToRemove(object):  
 *"""  
 This class takes data source id as input (identifier) and  
 identifies constanct columns, zero variance columns, high missing value columns  
 """* **def** \_\_init\_\_(self, data\_source\_id=None, lens\_id=None):  
 **try**:  
 self.data\_new = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=data\_source\_id,  
 lens\_id=lens\_id,  
 data\_frame\_type=**'WORKING'**)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# Remove constant columns from dataframe* **def** constant\_columns(self):  
 *""" This function returns all constant columns in a dataframe """* **try**:  
 **return** list(set(self.data\_new.loc[:, self.data\_new.apply(pd.Series.nunique) == 1]  
 .columns))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# Function to identify near zero variance columns i.e, columns with variance less than 1%* @staticmethod  
 **def** near\_zero\_var(series):  
 *""" This function identifies if a series has near zero variance """  
 # If the column is categorical, do label encoding and convert to integer  
 # if series.dtype in ['int16', 'int32', 'int64', 'float16', 'float32', 'float64', 'object']:* **try**:  
 series = series.dropna()  
 series = series.astype(str)  
 series = LabelEncoder().fit\_transform(series)  
 *# Calculate variance and return True if the variance is less than 2% else return false* **if** np.var(series) < 0.02:  
 **return** True  
 **return** False  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# Function to identify near zero variance columns i.e, columns with variance less than 1%* **def** zero\_var\_cols(self):  
 *""" This function returns all columns with variance less than 2% """  
 # store the value returned by function in a list* **try**:  
 index\_list = list(map(self.near\_zero\_var,  
 [self.data\_new[i] **for** i **in** self.data\_new.columns]))  
 *# Identify columns with near zero variance* zero\_var\_cols\_list = list(self.data\_new.columns[index\_list])  
 **return** zero\_var\_cols\_list  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# Columns with more than n% missing values* **def** high\_missing\_cols(self, threshold):  
 *"""  
 This function returns all columns with high percentage of missing values.  
 Default is more than 50% but user can change  
 """* **try**:  
 threshold = threshold / 100 *# Calculate threshold  
 # Identify percentage of missing values in each column* null\_percent = self.data\_new.isnull().sum() / len(self.data\_new)  
 *# list of columns with more missing values than the threshold* high\_missing\_cols\_list = null\_percent[  
 null\_percent > threshold].index  
 **return** list(high\_missing\_cols\_list)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This script identifies the type of each column  
"""***import** logging  
**from** itertools **import** compress  
**from** pandas **import** Series  
**from** dateutil.parser **import** parse  
**import** datetime  
**import** message\_constants  
**from** insights.utilities.columns\_filter **import** ColsToRemove  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** DataType(object):  
 *"""  
 This class categories all columns into one of factor columns, numeric columns,  
 date columns, id columns  
 """* **def** \_\_init\_\_(self, data\_source\_id=None, lens\_id=None):  
 **try**:  
 self.instance = ColsToRemove(data\_source\_id, lens\_id)  
 self.data\_new = self.instance.data\_new  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# Function to identify if a date is stored as string* @staticmethod  
 **def** is\_date(series):  
 *"""Function that identifies if a column is date"""* **try**:  
 series = series.dropna()  
 series = series[:1001]  
 *# Parse returns true if the object is date. If all rows are date, return true* **if** all([isinstance(x, datetime.datetime) **for** x **in** series]) **or** \  
 all([parse(x) **for** x **in** series]):  
 **return** True  
 **except** (TypeError, ValueError, AttributeError, IndexError, OverflowError) **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** False  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# Dates stored in ddmmyyyy format. Should include other formats later  
 # Dates stored in yyyymmdd and ddmmyyyy formats are already handled in the above function* @staticmethod  
 **def** is\_date\_in\_numeric(series):  
 *"""Function identifies if the column is a date in ddmmyyyy format"""* series = series.dropna()  
 series = series[:1001]  
 **try**:  
 *# Year should be between 1850 and 2050 and Month cannot be more than 12 and  
 # Date cannot be more than 12* **if** (**not** all([float(str(x)[-4:]) > 1850 **and** float(str(x)[:4]) < 2050 **for** x **in** series])) **or** \  
 (**not** all([float(str(x)[2:4]) < 13 **for** x **in** series])) **or** \  
 (**not** all([float(str(x)[0:2]) < 32 **for** x **in** series])) **or** \  
 (**not** all([len(str(x)) == 8 **for** x **in** series])):  
 **return** False  
  
 **except** ValueError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** False  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** True  
  
 **def** data\_cols\_identifier(self):  
 *"""Function identifies and returns all date columns in a dataframe"""* **try**:  
 data\_new = self.data\_new  
 *# Get the index of all date columns* date\_1 = list(data\_new.apply(self.is\_date))  
 *# Get the index of all date columns* date\_2 = list(data\_new.apply(self.is\_date\_in\_numeric))  
 *# Extract all date columns and store in date\_cols variable* date\_cols = list(set(list(data\_new.columns[list(compress(range(len(date\_2)), date\_2))])  
 + list(data\_new.columns[list(compress(range(len(date\_1)), date\_1))])))  
 **return** date\_cols  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# Function that identifies columns with less than 20 categories* @staticmethod  
 **def** cat\_cols\_identifier(series):  
 *"""This function identifies if a factor variable is stored as numeric"""* **try**:  
 series = series.dropna() *# Remove all null values  
 # Convert the series into category for checking number of levels* var = Series(series, dtype=**"category"**)  
 **if** var.nunique() < 20: *# Return false if levels are more than 20 of the data* **return** True  
 **return** False  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** num\_and\_factor\_cols\_identifier(self):  
 *"""This function returns all numeric and factor columns based on data type"""* **try**:  
 data\_new = self.data\_new  
 numerics = [**'int16'**, **'int32'**, **'int64'**, **'float16'**, **'float32'**, **'float64'**]  
 *# All numeric cols* numeric\_cols = list(data\_new.select\_dtypes(include=numerics).columns)  
 *# All object dtype cols* factor\_cols = list(data\_new.select\_dtypes(include=[**'object'**]).columns)  
 **return** [numeric\_cols, factor\_cols]  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# Function to check if the column is an ID column* @staticmethod  
 **def** is\_id(col):  
 *"""This function identifies and returns true if the column is an id column"""  
 # Convert the series into category for checking number of levels* series = Series(col, dtype=**"category"**)  
 *# IDs usually have more nlevels. Return false if levels are less than 25% of the data  
 # if not series.nunique() > 0.25\*len(col):  
 # return False* **if** (series.nunique()/len(col)) > 0.98:  
 **return** True  
 **try**:  
 *# ID columns does not have negative values or #ID columns does not have decimal values* **if** (min(col) < 0) **or** (**not** all([float(value).is\_integer() **for** value **in** col])):  
 **return** False  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **return** False  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** False

*"""This class is designed to do all database operations"""***from** insights.models **import** db  
**from** insights.models **import** Scenarios  
**import** constants  
**import** logging  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** DbUtilities:  
 *"""This class is designed to perform all database operations"""* **def** db\_insert(self, class\_id, \*\*kwargs):  
 *"""This class is design to perform db insert operation"""* **try**:  
 id\_return = 0  
 **if** class\_id == constants.INSIGHTS\_ML\_SCENARIOS:  
 scenarios = Scenarios(  
 data\_source\_id=kwargs[constants.DATA\_SOURCE\_ID],  
 lens\_id=kwargs[constants.LENS\_ID],  
 scenario\_name=kwargs[constants.SCENARIO\_NAME],  
 scenario\_input=kwargs[constants.SCENARIO\_INPUT],  
 scenario\_output=kwargs[constants.SCENARIO\_OUTPUT],  
 created\_by=kwargs[constants.CREATED\_BY],  
 updated\_by=kwargs[constants.UPDATED\_BY])  
 **try**:  
 db.session.add(scenarios)  
 db.session.commit()  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 db.session.rollback()  
 **raise** id\_return = scenarios.scenario\_id  
 db.session.close()  
 **return** id\_return  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 db.session.close()  
 **raise  
  
 def** select\_value(self, class\_id, \*\*kwargs):  
 *"""This function gives maximum value of column"""* **try**:  
 **if** class\_id == constants.INSIGHTS\_ML\_SCENARIOS:  
 qry\_result = db.session.query(Scenarios). \  
 filter(Scenarios.data\_source\_id == kwargs[constants.DATA\_SOURCE\_ID]).all()  
  
 **if** class\_id == constants.SCENARIO\_NAME:  
 qry\_result = db.session.query(Scenarios.scenario\_name).\  
 filter(Scenarios.data\_source\_id == kwargs[constants.DATA\_SOURCE\_ID]).all()  
 db.session.close()  
 **return** qry\_result  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 db.session.close()  
 **raise**

*"""  
@author: Phani Koratamaddi  
This script returns all columns categorized into one of factor cols,  
numeric cols, date cols, id cols and columns that are not required for analysis  
"""***import** logging  
**import** message\_constants  
**from** itertools **import** compress  
**from** insights.utilities.data\_type\_identifier **import** DataType  
*# from . import data\_type\_identifier*LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** GetColsAndType(object):  
 *"""  
 This class returns list of columns categorized based on its type.  
 Also returns list of columns that are not required for analysis.  
 """* **def** \_\_init\_\_(self, data\_source\_id=None, lens\_id=None):  
 **try**:  
 *# self.instance\_dt = data\_type\_identifier.DataType(identifier)* self.instance\_dt = DataType(data\_source\_id, lens\_id)  
 self.instance\_cols = self.instance\_dt.instance  
 self.data\_new = self.instance\_dt.data\_new  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** data\_types(self):  
 *"""This function returns columns classified into categories"""* **try**:  
 data\_new = self.data\_new  
 cols = self.instance\_dt.num\_and\_factor\_cols\_identifier()  
 factor\_cols = cols[1]  
 numeric\_cols = cols[0]  
 date\_cols = self.instance\_dt.data\_cols\_identifier()  
  
 *# Identify factor columns stored as numeric  
 # Get the index of all ID columns and Extract all low level columns* low\_level\_index = list(data\_new[numeric\_cols]  
 .apply(self.instance\_dt.cat\_cols\_identifier))  
 low\_level\_cols = list(set(list(data\_new[numeric\_cols]  
 .columns[list(compress(range(len(low\_level\_index)),  
 low\_level\_index))])))  
 *# Get the index of all ID columns  
 # Extract all ID columns and store in id\_cols variable* id\_index = list(data\_new[factor\_cols + numeric\_cols].  
 apply(self.instance\_dt.is\_id))  
 id\_cols = list(set(list(data\_new[factor\_cols + numeric\_cols]  
 .columns[list(compress(range(len(id\_index)), id\_index))])))  
 *# Exclude ID columns, date columns from numeric and factor columns  
 # Exclude low levels cols from numeric cols and add to factor cols* numeric\_cols = list(set(numeric\_cols) - set(id\_cols) - set(low\_level\_cols) - set(date\_cols))  
 factor\_cols = list(set(factor\_cols) - set(id\_cols) - set(date\_cols)) + low\_level\_cols  
 id\_cols = list(set(id\_cols) - set(factor\_cols) - set(numeric\_cols) - set(date\_cols))  
 **return** [numeric\_cols, factor\_cols, date\_cols, id\_cols]  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** cols\_to\_remove(self, cols\_to\_remove, threshold):  
 *"""This function returns columns that are required"""* **try**:  
 cols\_removed = list(set(self.instance\_cols.high\_missing\_cols(threshold)  
 + self.instance\_cols.zero\_var\_cols()  
 + self.instance\_cols.constant\_columns()))  
 **if** cols\_to\_remove **is not** None:  
 cols\_removed = list(set(cols\_removed + cols\_to\_remove))  
  
 self.data\_new = self.data\_new.drop(cols\_removed, axis=1)  
 **return** cols\_removed  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** return\_cols\_and\_types(self, cols\_to\_remove=None, threshold=50):  
 *"""This function returns all columns categorized as list in json format"""* **try**:  
 types = self.data\_types()  
 cols\_list = self.cols\_to\_remove(cols\_to\_remove, threshold)  
 json\_format = {  
 **"cols\_to\_remove"**: cols\_list,  
 **"numeric\_cols"**: types[0],  
 **"factor\_cols"**: types[1],  
 **"date\_cols"**: types[2],  
 **"id\_cols"**: types[3]  
 }  
 **return** json\_format  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

**class** InsightUtils(object):  
  
 **def** generate\_lists(input\_json):  
  
 *"""Utility function that accepts json  
 and returns dictionary containing list of columns [kpi,date] etc..  
 """* kbjsonlist = input\_json[**"knowledgebase"**]  
 date\_columns\_list = []  
 kpi\_colums\_list = []  
 drilldown\_columns\_list = []  
  
 **for** dict **in** kbjsonlist:  
 **if** dict[**"outlierKPIIndicator"**] == **'Y'**:  
 kpi\_colums\_list.append(dict[**"columnName"**])  
  
 **if** dict[**"drillDownIndicator"**] == **'Y'**:  
 drilldown\_columns\_list.append(dict[**"columnName"**])  
  
 **if** dict[**"dateIndicator"**] == **'Y'**:  
 date\_columns\_list.append(dict[**"columnName"**])  
  
 output\_json = {**"date\_column"**: date\_columns\_list[0],  
 **"kpi\_colums\_list"**: kpi\_colums\_list,  
 **"drilldown\_columns\_list"**: drilldown\_columns\_list,  
 **"user\_id"**: input\_json[**"user\_id"**],  
 **"data\_source\_id"**: input\_json[**"data\_source\_id"**],  
 **"defaultFrequency"**: kbjsonlist[0][**"defaultFrequency"**]  
 }  
  
 **return** output\_json

*"""  
author: Phani Koratamaddi  
This script is to test columns\_filter.py file  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** insights.utilities.columns\_filter **import** ColsToRemove  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestColsToRemove(TestCase):  
 *"""  
 This class takes data source id as input (identifier) and  
 tests constanct columns, zero variance columns, high missing value columns  
 """* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df and output df frame.  
 It runs once only"""* cls.test\_df = pd.read\_csv(**"../resources/test\_cbdt\_10000\_rows.csv"**)  
 json = {**"ORIGINAL"**: cls.test\_df, **"WORKING"**: cls.test\_df}  
 DataRegistry.pandas\_df\_map.update({**"1\_1"**: json})  
 cls.instance = ColsToRemove(data\_source\_id=1, lens\_id=1)  
  
 @classmethod  
 **def** tearDownClass(cls):  
 *"""this function runs once after execution of all functions"""* **pass  
  
 def** test\_constant\_columns(self):  
 *"""  
 Function to test constant columns* **:return***: assertion  
 """* col\_list = self.instance.constant\_columns()  
 test\_list = [**'GOVT\_EMP\_FLAG'**, **'test\_constant'**]  
 self.assertEqual(set(col\_list), set(test\_list))  
  
 **def** test\_near\_zero\_var(self):  
 *"""  
 Function to test zero var series* **:return***: assertion  
 """* val = self.instance.near\_zero\_var(self.test\_df[**'test\_zero\_var'**])  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.near\_zero\_var(self.test\_df[**'K40\_TXN\_CNT\_RATIO'**])  
 test\_val = False  
 self.assertEqual(val, test\_val)  
  
 **def** test\_zero\_var\_cols(self):  
 *"""  
 Function to test zero var columns* **:return***: assertion  
 """* col\_list = self.instance.zero\_var\_cols()  
 test\_list = [**'GOVT\_EMP\_FLAG'**, **'test\_constant'**, **'AIR\_IND'**,  
 **'AIR\_PROPERTY\_SALE\_IND'**, **'test\_zero\_var'**]  
 self.assertEqual(set(col\_list), set(test\_list))  
  
 **def** test\_high\_missing\_cols(self):  
 *"""  
 Function to test high missing columns* **:return***: assertion  
 """* col\_list = self.instance.high\_missing\_cols(10)  
 test\_list = [**'CASH\_IN\_HAND'**, **'INCOME\_HOUSE\_PROPERTY'**, **'K40\_PMT\_AMT\_POST\_DEM'**,  
 **'GROSS\_TOTAL\_INCOME'**, **'NET\_SALES\_GROSS\_RECEIPTS'**, **'K40\_TXN\_CNT\_RATIO'**,  
 **'CA15\_AMOUNT\_RATIO'**, **'GOVT\_EMP\_FLAG'**]  
 self.assertEqual(set(col\_list), set(test\_list))  
  
 col\_list = self.instance.high\_missing\_cols(50)  
 test\_list = [**'K40\_PMT\_AMT\_POST\_DEM'**, **'K40\_TXN\_CNT\_RATIO'**,  
 **'CA15\_AMOUNT\_RATIO'**, **'GOVT\_EMP\_FLAG'**]  
 self.assertEqual(set(col\_list), set(test\_list))

*"""  
author: Phani Koratamaddi  
This script tests data type identifier script  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** insights.utilities.data\_type\_identifier **import** DataType  
  
  
**class** TestDataType(TestCase):  
 *"""  
 This class tests all columns categorized into one of factor columns, numeric columns,  
 date columns, id columns  
 """* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df and output df frame.  
 It runs once only"""* cls.test\_df = pd.read\_csv(**"../resources/test\_cbdt\_10000\_rows.csv"**)  
 json = {**"ORIGINAL"**: cls.test\_df, **"WORKING"**: cls.test\_df}  
 DataRegistry.pandas\_df\_map.update({**"1\_1"**: json})  
 cls.instance = DataType(data\_source\_id=1, lens\_id=1)  
  
 @classmethod  
 **def** tearDownClass(cls):  
 *"""this function runs once after execution of all functions"""* **pass  
  
 def** test\_is\_date(self):  
 *"""  
 Function that tests if a column is date* **:return***: assertion  
 """* val = self.instance.is\_date(pd.Series([**'12-2-2012'**]))  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.is\_date(pd.Series([**'Hello'**]))  
 test\_val = False  
 self.assertEqual(val, test\_val)  
  
 **def** test\_is\_date\_in\_numeric(self):  
 *"""  
 Function tests if the column is a date in ddmmyyyy format* **:return***: assertion  
 """* val = self.instance.is\_date\_in\_numeric(pd.Series([**'12022012'**]))  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.is\_date\_in\_numeric(pd.Series([**'Hello'**]))  
 test\_val = False  
 self.assertEqual(val, test\_val)  
  
 **def** test\_data\_cols\_identifier(self):  
 *"""  
 Function to test all date columns in a dateframe* **:return***: assertion  
 """* val = self.instance.data\_cols\_identifier()  
 test\_val = [**'test\_date'**]  
 self.assertEqual(val, test\_val)  
  
 **def** test\_cat\_cols\_identifier(self):  
 *"""  
 This function tests if a factor variable is stored as numeric* **:return***: assertion  
 """* val = self.instance.cat\_cols\_identifier(self.test\_df[**'flag\_responded\_2'**])  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.cat\_cols\_identifier(self.test\_df[**'Id'**])  
 test\_val = False  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.cat\_cols\_identifier(self.test\_df[**'BUSINESS\_CLASS'**])  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 **def** test\_num\_and\_factor\_cols(self):  
 *"""  
 This function tests all numeric and factor columns based on data type* **:return***: assertion  
 """* val = self.instance.num\_and\_factor\_cols\_identifier()  
 test\_val = [[**'Unnamed: 0'**, **'Unnamed: 0.1'**, **'Id'**, **'CASH\_IN\_HAND'**, **'INCOME\_HOUSE\_PROPERTY'**,  
 **'K40\_PMT\_AMT\_POST\_DEM'**, **'GROSS\_TOTAL\_INCOME'**, **'AIR\_PROPERTY\_SALE\_IND'**,  
 **'AIR\_IND'**, **'NET\_SALES\_GROSS\_RECEIPTS'**, **'K40\_TXN\_CNT\_RATIO'**,  
 **'CA15\_AMOUNT\_RATIO'**, **'CASH\_DEPOSITED\_PER\_ACC\_POST'**, **'flag\_responded\_2'**,  
 **'test\_constant'**, **'test\_zero\_var'**],  
 [**'PCCIT\_DESC'**, **'BUSINESS\_CLASS'**, **'STATE'**, **'TAXPAYER\_TYPE\_2'**,  
 **'GOVT\_EMP\_FLAG'**, **'test\_date'**]]  
  
 self.assertEqual(set(val[0]), set(test\_val[0]))  
 self.assertEqual(set(val[1]), set(test\_val[1]))  
  
 **def** test\_is\_id(self):  
 *"""  
 This function tests if the column is an id column* **:return***: assertion  
 """* val = self.instance.is\_id(self.test\_df[**'Id'**])  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.is\_id(self.test\_df[**'Unnamed: 0'**])  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.is\_id(self.test\_df[**'Unnamed: 0.1'**])  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.is\_id(self.test\_df[**'test\_date'**])  
 test\_val = True  
 self.assertEqual(val, test\_val)  
  
 val = self.instance.is\_id(self.test\_df[**'PCCIT\_DESC'**])  
 test\_val = False  
 self.assertEqual(val, test\_val)

*"""Test class for database utilities file"""***from** unittest **import** TestCase  
  
  
**class** TestDbUtilities(TestCase):  
 *"""Test class for DbUtilities """* @classmethod  
 **def** setUpClass(cls):  
 **pass** @classmethod  
 **def** tearDownClass(cls):  
 *"""this function runs once after execution of all functions"""* **pass  
  
 def** test\_db\_insert(self):  
 *"""Test function for db\_insert"""* **pass  
  
 def** test\_select\_value(self):  
 *"""Test function for select\_value"""* **pass**

*"""  
Author: Phani Koratamaddi  
This script tests all columns categorized into one of factor cols,  
numeric cols, date cols, id cols and columns that are not required for analysis  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** insights.utilities.get\_cols\_and\_types **import** GetColsAndType  
  
  
**class** TestGetColsAndType(TestCase):  
 *"""  
 This class tests list of columns categorized based on its type.  
 Also returns list of columns that are not required for analysis.  
 """* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df and output df frame.  
 It runs once only"""* cls.data\_source\_id = 1  
 cls.lens\_id = 1  
 cls.test\_df = pd.read\_csv(**"../resources/test\_cbdt\_10000\_rows.csv"**)  
 json = {**"ORIGINAL"**: cls.test\_df, **"WORKING"**: cls.test\_df}  
 DataRegistry.pandas\_df\_map.update({**"1\_1"**: json})  
 cls.instance = GetColsAndType(cls.data\_source\_id, cls.lens\_id)  
  
 @classmethod  
 **def** tearDownClass(cls):  
 *"""this function runs once after execution of all functions"""* **pass  
  
 def** test\_data\_types(self):  
 *"""  
 This function tests columns classified into categories* **:return***: assertion  
 """* col\_list = self.instance.data\_types()  
 test\_list = [[**'INCOME\_HOUSE\_PROPERTY'**, **'CASH\_IN\_HAND'**, **'GROSS\_TOTAL\_INCOME'**,  
 **'NET\_SALES\_GROSS\_RECEIPTS'**, **'CASH\_DEPOSITED\_PER\_ACC\_POST'**],  
 [**'AIR\_IND'**, **'TAXPAYER\_TYPE\_2'**, **'test\_constant'**, **'K40\_PMT\_AMT\_POST\_DEM'**,  
 **'K40\_TXN\_CNT\_RATIO'**, **'flag\_responded\_2'**, **'test\_zero\_var'**, **'BUSINESS\_CLASS'**,  
 **'AIR\_PROPERTY\_SALE\_IND'**, **'STATE'**, **'GOVT\_EMP\_FLAG'**, **'CA15\_AMOUNT\_RATIO'**,  
 **'PCCIT\_DESC'**],  
 [**'test\_date'**],  
 [**'Unnamed: 0'**, **'Id'**, **'Unnamed: 0.1'**]]  
 self.assertEqual(set(col\_list[0]), set(test\_list[0]))  
 self.assertEqual(set(col\_list[1]), set(test\_list[1]))  
 self.assertEqual(set(col\_list[2]), set(test\_list[2]))  
 self.assertEqual(set(col\_list[3]), set(test\_list[3]))  
  
 **def** test\_cols\_to\_remove(self):  
 *"""  
 This function tests columns that are required* **:return***: assertion  
 """* col\_list = GetColsAndType(self.data\_source\_id, self.lens\_id) \  
 .cols\_to\_remove(cols\_to\_remove=[], threshold=50)  
 test\_list = [**'GOVT\_EMP\_FLAG'**, **'test\_constant'**, **'AIR\_PROPERTY\_SALE\_IND'**,  
 **'AIR\_IND'**, **'test\_zero\_var'**, **'K40\_PMT\_AMT\_POST\_DEM'**,  
 **'K40\_TXN\_CNT\_RATIO'**, **'CA15\_AMOUNT\_RATIO'**, **'GOVT\_EMP\_FLAG'**]  
 self.assertEqual(set(col\_list), set(test\_list))  
  
 **def** test\_return\_cols\_and\_types(self):  
 *"""  
 This function tests all columns categorized as list in json format* **:return***: assertion  
 """* col\_list = GetColsAndType(self.data\_source\_id, self.lens\_id) \  
 .return\_cols\_and\_types()  
  
 test\_list = dict()  
 test\_list[**'cols\_to\_remove'**] = [**'GOVT\_EMP\_FLAG'**, **'test\_constant'**, **'AIR\_PROPERTY\_SALE\_IND'**,  
 **'AIR\_IND'**, **'test\_zero\_var'**, **'K40\_PMT\_AMT\_POST\_DEM'**,  
 **'K40\_TXN\_CNT\_RATIO'**, **'CA15\_AMOUNT\_RATIO'**, **'GOVT\_EMP\_FLAG'**]  
 test\_list[**'numeric\_cols'**] = [**'INCOME\_HOUSE\_PROPERTY'**, **'CASH\_IN\_HAND'**,  
 **'GROSS\_TOTAL\_INCOME'**, **'NET\_SALES\_GROSS\_RECEIPTS'**,  
 **'CASH\_DEPOSITED\_PER\_ACC\_POST'**]  
 test\_list[**'factor\_cols'**] = [**'BUSINESS\_CLASS'**, **'CA15\_AMOUNT\_RATIO'**, **'PCCIT\_DESC'**,  
 **'TAXPAYER\_TYPE\_2'**, **'GOVT\_EMP\_FLAG'**, **'AIR\_IND'**,  
 **'AIR\_PROPERTY\_SALE\_IND'**, **'K40\_PMT\_AMT\_POST\_DEM'**,  
 **'flag\_responded\_2'**, **'test\_constant'**, **'test\_zero\_var'**,  
 **'STATE'**, **'K40\_TXN\_CNT\_RATIO'**]  
 test\_list[**'date\_cols'**] = [**'test\_date'**]  
 test\_list[**'id\_cols'**] = [**'Unnamed: 0'**, **'Id'**, **'Unnamed: 0.1'**]  
  
 self.assertEqual(set(col\_list[**'cols\_to\_remove'**]), set(test\_list[**'cols\_to\_remove'**]))  
 self.assertEqual(set(col\_list[**'numeric\_cols'**]), set(test\_list[**'numeric\_cols'**]))  
 self.assertEqual(set(col\_list[**'factor\_cols'**]), set(test\_list[**'factor\_cols'**]))  
 self.assertEqual(set(col\_list[**'date\_cols'**]), set(test\_list[**'date\_cols'**]))  
 self.assertEqual(set(col\_list[**'id\_cols'**]), set(test\_list[**'id\_cols'**]))  
  
 **with** self.assertRaises(Exception):  
 GetColsAndType(None, None).return\_cols\_and\_types()

*What\_if\_analysis*

*"""Test class for Utilities functions"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** insights.what\_if\_analysis.utilities\_function **import** Utilities  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestUtilities(TestCase):  
 *"""Test class for Utilities"""* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df, it runs once only"""* data\_df = pd.read\_csv(**"../resources/kc\_house\_data1.csv"**) *# remove* json = {**"ORIGINAL"**: data\_df, **"WORKING"**: data\_df} *# Need to remove* DataRegistry.pandas\_df\_map.update({**"7\_7"**: json}) *# Need  
  
 # @classmethod  
 # def tearDownClass(cls):  
 # """this function runs once after execution of all functions"""  
 # pass* **def** test\_dict\_to\_ulist(self):  
 *"""Test function for dict\_to\_ulist function"""* function\_input\_dict = {**'price'**: [**'grade'**, **'bedrooms'**, **'sqft\_basement'**, **'bathrooms'**,  
 **'sqft\_above'**, **'sqft\_lot'**],  
 **'sqft\_living'**: [**'sqft\_basement'**, **'sqft\_above'**]}  
 expected\_output = [**'bathrooms'**, **'bedrooms'**, **'grade'**, **'sqft\_above'**, **'sqft\_basement'**,  
 **'sqft\_lot'**]  
 function\_output = sorted(Utilities.dict\_to\_ulist(function\_input\_dict))  
 *# print("function\_output", function\_output)* self.assertListEqual(function\_output, expected\_output)  
 *# pass* **def** test\_final\_label\_encoding(self):  
 *"""Test function for final\_label\_encoding function"""* input\_dict = {**'condition'**: {**'Three Star'**: 3, **'Four Star'**: 1, **'One Star'**: 2,  
 **'Five Star'**: 0, **'Two Star'**: 4},  
 **'waterfront'**: {**'Yes'**: 1, **'No'**: 0},  
 **'grade'**: {**'Twelve'**: 11, **'One'**: 5, **'Thirteen'**: 9, **'Six'**: 7,  
 **'Eleven'**: 1, **'Three'**: 10, **'Five'**: 2, **'Eight'**: 0,  
 **'Seven'**: 6, **'Nine'**: 4, **'Four'**: 3, **'Ten'**: 8}}  
 input\_list = [**'condition'**, **'sqft\_lot15'**, **'zipcode'**, **'sqft\_basement'**, **'bathrooms'**]  
 expected\_output = {**'condition'**: {**'Three Star'**: 3, **'Four Star'**: 1, **'One Star'**: 2,  
 **'Five Star'**: 0, **'Two Star'**: 4}}  
 expected\_output = Utilities.final\_label\_encoding(input\_dict, input\_list)  
 self.assertDictEqual(expected\_output, expected\_output)  
 *# pass* **def** test\_df\_to\_dict(self):  
 *"""Test function for df\_to\_dict function"""* test\_df = pd.DataFrame({**'Col1'**: [**'red'**, **'yellow'**, **'blue'**],  
 **'Col2'**: [0.5, 0.25, 0.125], **'Col3'**: [15, 4, 11]})  
 expected\_output = {**'Col1'**: [**'red'**, **'yellow'**, **'blue'**],  
 **'Col2'**: [0.5, 0.25, 0.125], **'Col3'**: [15, 4, 11]}  
 function\_output = Utilities.df\_to\_dict(test\_df)  
 self.assertDictEqual(expected\_output, function\_output)  
  
 **def** test\_target\_classification(self):  
 *"""Test class for target\_classification method"""* expected\_target\_list = [**'waterfront'**, **'view'**, **'condition'**]  
 target\_column\_list = Utilities.classification\_target(7, 7)  
 self.assertListEqual(expected\_target\_list, target\_column\_list)  
  
 **def** test\_predict\_classifier(self):  
 *"""Test function for predict\_classifier function of what if classifier class"""* input\_dict = {**'view'**: [0], **'sqft\_living'**: [2050], **'bedrooms'**: [3.0],  
 **'price'**: [533788], **'bathrooms'**: [2.061972], **'sqft\_above'**: [1747],  
 **'condition'**: [3], **'sqft\_basement'**: [303], **'sqft\_lot'**: [16005.0]  
 }  
 data\_frame = pd.DataFrame.from\_dict(input\_dict)  
 expected\_output = {**'waterfront'**: [**'[0]'**, **'[[1. 0.]]'**, **'[0 1]'**]}  
 predict\_output = Utilities.predict\_classifier(data\_frame, [**'waterfront'**])  
 self.assertDictEqual(expected\_output, predict\_output)

*"""This python file is contains test case for what if analysis"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**import** numpy **as** np  
**from** insights.what\_if\_analysis.what\_if\_analysis **import** WhatIfAnalysis  
np.set\_printoptions(suppress=True)  
  
  
**class** TestWhatIfAnalysis(TestCase):  
 *"""This class for Test cases for all functions of WhatIfAnalysis class"""* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df, it runs once only"""* cls.test\_df = pd.read\_csv(**"../resources/Test\_what\_if\_analysis.csv"**)  
 cls.wf = WhatIfAnalysis(cls.test\_df)  
 cls.factor\_cols = [**'grade'**, **'waterfront'**, **'condition'**, **'view'**]  
 cls.target = [**'price'**, **'sqft\_living'**]  
  
 @classmethod  
 **def** tearDownClass(cls):  
 *"""this function runs once after execution of all functions"""* **pass  
  
 def** test\_1\_label\_encoding(self):  
 *"""Test function for label\_encoding function in what\_if\_analysis"""* expected\_output\_dict = {**'grade'**: {**'Eleven'**: 1, **'Eight'**: 0, **'Thirteen'**: 8, **'Six'**: 6,  
 **'Nine'**: 4, **'Seven'**: 5, **'Ten'**: 7, **'Four'**: 3, **'Three'**: 9,  
 **'Twelve'**: 10, **'Five'**: 2},  
 **'condition'**: {**'Five Star'**: 0, **'Four Star'**: 1, **'Two Star'**: 4,  
 **'One Star'**: 2, **'Three Star'**: 3},  
 **'waterfront'**: {**'No'**: 0, **'Yes'**: 1}}  
 le\_func\_output\_dict, self.test\_df = self.wf.label\_encoding(self.factor\_cols)  
 self.assertDictEqual(expected\_output\_dict, le\_func\_output\_dict)  
  
 **def** test\_2\_imp\_feature(self):  
 *"""Test function for imp\_feature function in what\_if\_analysis"""* expected\_output\_dict = {**'sqft\_basement'**: 0.36, **'bathrooms'**: 0.05, **'sqft\_above'**: 0.48,  
 **'bedrooms'**: 0.04, **'sqft\_lot'**: 0.04}  
 expected\_output\_list = sorted(expected\_output\_dict.keys())  
 expected\_all\_features = {**'condition'**: 0.0, **'view'**: 0.01, **'bathrooms'**: 0.05,  
 **'sqft\_lot'**: 0.04, **'sqft\_basement'**: 0.36, **'bedrooms'**: 0.04,  
 **'waterfront'**: 0.0, **'floors'**: 0.01, **'grade'**: 0.01,  
 **'sqft\_above'**: 0.48}  
 expected\_all\_features = sorted(expected\_all\_features.keys())  
 if\_func\_output\_dict, func\_all\_imp\_features = self.wf.imp\_feature(self.target)  
 func\_all\_imp\_features = sorted(func\_all\_imp\_features.keys())  
 self.assertListEqual(expected\_output\_list, sorted(if\_func\_output\_dict.keys()))  
 self.assertListEqual(func\_all\_imp\_features, expected\_all\_features)  
  
 **def** test\_3\_linear\_regression(self):  
  
 *"""Test function for linear\_regression function in what\_if\_analysis"""* expected\_output\_dict = {**'sqft\_living'**: [{**'max'**: [0.0, 11.0, 4820.0, 8.0, 8570.0,  
 1651359.0, 12050.0],  
 **'coef'**: [0.0, -0.0, 1.0, -0.0, 1.0,  
 -0.0, -999999.0],  
 **'cols'**: [**'Intercept'**, **'bedrooms'**,  
 **'sqft\_basement'**, **'bathrooms'**,  
 **'sqft\_above'**, **'sqft\_lot'**,  
 **'sqft\_living'**],  
 **'min'**: [0.0, 0.0, 0.0, 0.0, 380.0,  
 572.0, 380.0],  
 **'cat'**: [0, **'num'**, **'num'**, **'num'**,  
 **'num'**, **'num'**, **'num'**],  
 **'model\_accuracy'**: [100.0, 100.0],  
 **'avg'**: [0.0, 3.0, 303.0, 2.0, 1747.0,  
 16005.0, 2050.4784444444444]}],  
 **'price'**: [{**'max'**: [0.0, 11.0, 4820.0, 8.0, 8570.0,  
 1651359.0, 7700000.0],  
 **'coef'**: [79550.369945, -65477.061602,  
 342.532503, -496.085706, 329.616312,  
 -0.283221, -999999.0],  
 **'cols'**: [**'Intercept'**, **'bedrooms'**, **'sqft\_basement'**,  
 **'bathrooms'**, **'sqft\_above'**, **'sqft\_lot'**,  
 **'price'**],  
 **'min'**: [0.0, 0.0, 0.0, 0.0, 380.0, 572.0, 75000.0],  
 **'model\_accuracy'**: [51.15, 51.12],  
 **'cat'**: [0, **'num'**, **'num'**, **'num'**, **'num'**, **'num'**, **'num'**],  
 **'avg'**: [0.0, 3.0, 303.0, 2.0, 1747.0, 16005.0,  
 533788.2942222222]}]}  
 top\_imp\_features, \_ = self.wf.imp\_feature(self.target)  
 lr\_func\_output\_dict = self.wf.linear\_regression(self.target, self.factor\_cols,  
 top\_imp\_features)  
 self.assertListEqual(sorted(expected\_output\_dict.keys()),  
 sorted(lr\_func\_output\_dict.keys()))  
 self.assertEqual(len(expected\_output\_dict[**'price'**]),  
 len(lr\_func\_output\_dict[**'price'**]))  
 self.assertEqual(len(expected\_output\_dict[**'sqft\_living'**]),  
 len(lr\_func\_output\_dict[**'sqft\_living'**]))

*"""Test class for WhatIfClassification class"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** insights.what\_if\_analysis.what\_if\_classification **import** WhatIfClassification  
  
  
**class** TestWhatIfClassification(TestCase):  
 *"""Test class for WhatIfClassification class"""* @classmethod  
 **def** setUpClass(cls):  
 *"""function to load test df, it runs once only"""* cls.test\_df = pd.read\_csv(**"../resources/Test\_what\_if\_analysis.csv"**)  
 cls.wf = WhatIfClassification(cls.test\_df)  
 cls.factor\_cols = [**'grade'**, **'condition'**, **'view'**]  
 cls.target = [**'waterfront'**]  
 cls.top\_imp\_features\_list = []  
  
 *# @classmethod  
 # def tearDownClass(cls):  
 # """this function runs once after execution of all functions"""  
 # pass* **def** test\_1\_label\_encoding(self):  
 *"""Test function for label\_encoding function in what\_if\_analysis"""* expected\_output\_dict = {**'condition'**: {**'Two Star'**: 4, **'Three Star'**: 3, **'Four Star'**: 1,  
 **'One Star'**: 2, **'Five Star'**: 0},  
 **'waterfront'**: {**'No'**: 0, **'Yes'**: 1},  
 **'grade'**: {**'Three'**: 9, **'Ten'**: 7, **'Five'**: 2, **'Nine'**: 4, **'Seven'**: 5,  
 **'Four'**: 3, **'Eight'**: 0, **'Eleven'**: 1, **'Twelve'**: 10,  
 **'Thirteen'**: 8, **'Six'**: 6}}  
 le\_func\_output\_dict, self.test\_df = self.wf.label\_encoding(self.factor\_cols + self.target)  
 self.assertDictEqual(expected\_output\_dict, le\_func\_output\_dict)  
  
 **def** test\_2\_imp\_feature(self):  
 *"""Test function for imp\_feature function in what\_if\_analysis"""* expected\_output\_list = {**'sqft\_lot'**: 0.19, **'sqft\_living'**: 0.08, **'bathrooms'**: 0.06,  
 **'condition'**: 0.05, **'bedrooms'**: 0.07, **'sqft\_basement'**: 0.1,  
 **'price'**: 0.17, **'view'**: 0.14, **'sqft\_above'**: 0.09}  
 expected\_all\_features = {**'sqft\_lot'**: 0.19, **'sqft\_living'**: 0.08, **'bathrooms'**: 0.06,  
 **'condition'**: 0.05, **'grade'**: 0.03, **'view'**: 0.14, **'sqft\_above'**: 0.09,  
 **'sqft\_basement'**: 0.1, **'price'**: 0.17, **'bedrooms'**: 0.07,  
 **'floors'**: 0.02}  
 if\_func\_output\_dict, func\_all\_imp\_features = self.wf.imp\_feature(self.target)  
 self.assertListEqual(sorted(expected\_output\_list.keys()),  
 sorted(if\_func\_output\_dict.keys()))  
 self.assertListEqual(sorted(func\_all\_imp\_features), sorted(expected\_all\_features))  
  
 **def** test\_3\_classification(self):  
 *"""Test function for linear\_regression function in what\_if\_analysis"""* expected\_output\_dict = {**'waterfront'**: [{**'cat'**: [**'int'**, **'int'**, **'int'**, **'int'**, **'float'**, **'int'**,  
 **'cat'**, **'int'**, **'cat'**, **'num'**],  
 **'max'**: [1651359.0, 11.0, 12050.0, 7700000.0, 8.0,  
 8570.0, 4.0, 4820.0, 4.0, 1.0],  
 **'model\_accuracy'**: [99.56, 84.62, 64.71,  
 **'[[1781 2]\n [ 6 11]]'**,  
 0.73],  
 **'avg'**: [16005.0, 3.0, 2050.0, 533788.0, 2.061972,  
 1747.0, 3.0, 303.0, 0.0, 0.008111],  
 **'min'**: [572.0, 0.0, 380.0, 75000.0, 0.0, 380.0,  
 0.0, 0.0, 0.0, 0.0],  
 **'cols'**: [**'sqft\_lot'**, **'bedrooms'**, **'sqft\_living'**,  
 **'price'**, **'bathrooms'**, **'sqft\_above'**,  
 **'condition'**, **'sqft\_basement'**, **'view'**,  
 **'waterfront'**],  
 **'coef'**: [0, 0, 0, 0, 0, 0, 0, 0, 0, -999999]}]}  
 top\_imp\_features, \_ = self.wf.imp\_feature(self.target)  
 self.top\_imp\_features\_list = [keys **for** keys, \_ **in** top\_imp\_features.items()]  
 class\_func\_output\_dict = self.wf.classification(self.target, self.factor\_cols,  
 self.top\_imp\_features\_list)  
 self.assertListEqual(sorted(expected\_output\_dict.keys()),  
 sorted(class\_func\_output\_dict.keys()))  
 self.assertEqual(len(expected\_output\_dict[**'waterfront'**]),  
 len(class\_func\_output\_dict[**'waterfront'**]))

*"""  
This class contains multiple user define methods, helps user to remove duplicate code  
"""***import** logging  
**import** itertools  
**import** pickle  
**from** pandas **import** unique  
**import** pandas **as** pd  
**from** numpy **import** array\_str  
**import** numpy **as** np  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** Utilities:  
 *"""  
 This class contains multiple user define methods, helps user to remove duplicate code  
 """* @staticmethod  
 **def** dict\_to\_ulist(input\_dict):  
 *"""This function converts dictionary values to unique list"""* **try**:  
 unique\_list = []  
 **for** \_, value **in** input\_dict.items():  
 unique\_list.append(value) *# getting list of list in input\_dict  
 # Now convert list of list into a single list* unique\_list = list(itertools.chain.from\_iterable(unique\_list))  
 *# Remove duplicate values* unique\_list = list(unique(unique\_list))  
 **return** unique\_list  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** final\_label\_encoding(label\_encoding\_dict, imp\_features\_list):  
 *"""This function checks important variable list and create a final label encoding  
 list if matches with existing label encoding dictionary  
 """* **try**:  
 final\_label\_encode\_dict = {}  
 **for** key, value **in** label\_encoding\_dict.items():  
 **if** key **in** imp\_features\_list:  
 final\_label\_encode\_dict.update({key: value})  
 **return** final\_label\_encode\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** df\_to\_dict(data\_df):  
 *"""This function converts a dataframe into custom dictionary format"""* **try**:  
 temp\_dict = {}  
 **for** \_, value **in** enumerate(data\_df.columns):  
 temp\_dict[value] = list(data\_df[value])  
 **return** temp\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** classification\_target(data\_source\_id, lens\_id):  
 *"""This function returns valid target columns for what if classification problem"""* **try**:  
 data = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=data\_source\_id,  
 lens\_id=lens\_id,  
 data\_frame\_type=**'WORKING'**)  
 data\_new = data.loc[:, data.apply(**lambda** x: x.nunique()) <= 10]  
 data\_new = data\_new.select\_dtypes(exclude=[**'float'**])  
 **return** list(data\_new.columns.values)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** predict\_classifier(x\_test\_json, target\_list):  
 *"""Function to call predict for what if classification"""* **try**:  
 data\_frame = pd.DataFrame.from\_dict(x\_test\_json)  
 return\_dict = {}  
 **for** \_, target **in** enumerate(target\_list):  
 **with** open(**"model\_supportive\_info.pkl"**, **'rb'**) **as** file:  
 support\_info = pickle.load(file)  
 data\_frame = data\_frame[support\_info[**"important\_feature"**]]  
 x\_test = data\_frame.values  
 *# model = self.model\_dict[target]* **with** open(**"xgb\_model.pkl"**, **'rb'**) **as** file:  
 model = pickle.load(file)  
 *# model = pickle.load(open("xgb\_model.pkl", "rb"))* y\_pred = model.predict(x\_test)  
 y\_pred\_prob = np.round(model.predict\_proba(x\_test), 2)  
 return\_dict.update({target: [array\_str(y\_pred), array\_str(y\_pred\_prob),  
 support\_info[**"target\_unique\_values"**]]})  
 **return** return\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
@author: Krishna Mohan  
This script does label encoding for factor cols , identify important features  
runs linear regression for What if Analysis part of LENS  
using XGboost algorithm  
"""***import** logging  
**from** statistics **import** mode, StatisticsError  
**from** collections **import** Counter  
**import** statsmodels.formula.api **as** sm  
**from** sklearn.preprocessing **import** LabelEncoder  
**from** sklearn.model\_selection **import** train\_test\_split  
**import** xgboost  
**import** pandas **as** pd  
**import** numpy **as** np  
**import** message\_constants  
**from** insights.what\_if\_analysis.utilities\_function **import** Utilities  
np.set\_printoptions(suppress=True)  
np.seterr(divide=**'ignore'**, invalid=**'ignore'**)  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** WhatIfAnalysis:  
 *"""  
 Class that generates important features  
 """* **def** \_\_init\_\_(self, data\_new):  
 *""" to initialize our class with dataframe"""* **try**:  
 self.data\_new = data\_new  
 self.obj\_cols = []  
 self.imp\_feature\_dict = {}  
 self.important\_feature = []  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** label\_encoding(self, factor\_cols\_list):  
 *"""  
 This function does label encoding for all object type variables and make changes  
 in dataset and replaces categorical with numeric value* **:return***: list of labelled variables and converted values  
 """* **try**:  
 label\_encoding = LabelEncoder()  
 label\_dictionary = {}  
 *# Iterating over all object type columns* **for** col **in** self.data\_new[factor\_cols\_list].columns.values:  
 *# label Encoding for only categorical variables* **if** self.data\_new[col].dtypes == **'object'**:  
 self.obj\_cols.append(col)  
 *# label\_encoding.fit(self.data\_new[col].values)* **try**:  
 label\_encoding.fit(self.data\_new[col].values)  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 *# ValueError: could not convert string to float:* label\_encoding.fit(self.data\_new[col].values.astype(str))  
 le\_name\_mapping = dict(zip(label\_encoding.classes\_,  
 label\_encoding.transform(label\_encoding.classes\_)))  
 *# self.data\_new[col] = label\_encoding.transform(self.data\_new[col])* **try**:  
 self.data\_new[col] = label\_encoding.transform(self.data\_new[col])  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 *# TypeError: unorderable types: float() < str()* self.data\_new[col] = label\_encoding.transform(self.data\_new[col].astype  
 (str))  
 label\_dictionary[col] = le\_name\_mapping  
 **return** label\_dictionary, self.data\_new  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** imp\_feature(self, target\_list):  
 *"""  
 This function does identify important variables* **:return***: list of labelled variables and converted values  
 """* **try**:  
 x\_train, \_, y\_train, \_, attributes, \_ = self.prep\_data(target\_list)  
 *# XGboost algorithm to get important variables* xgb = xgboost.XGBRegressor(n\_estimators=100, learning\_rate=0.08, gamma=0,  
 subsample=0.75, colsample\_bytree=1, max\_depth=7)  
 **for** i, \_ **in** enumerate(target\_list):  
 important\_feature\_dict = {}  
 model = xgb.fit(x\_train, y\_train[:, i])  
 feature\_labels = np.array(attributes)  
 importance = model.feature\_importances\_  
 feature\_indexes\_by\_importance = importance.argsort()  
 **for** index **in** feature\_indexes\_by\_importance:  
 **if** feature\_labels[index] **in** list(important\_feature\_dict.keys()):  
 important\_feature\_dict.update({feature\_labels[index]:  
 round((importance[index]  
 + important\_feature\_dict[  
 feature\_labels[index]])/2,  
 2)})  
 **else**:  
 important\_feature\_dict.update({feature\_labels[index]: round(importance  
 [index],  
 2)})  
  
 **return** {key: value **for** (key, value) **in** important\_feature\_dict.items() **if** value > .03},\  
 important\_feature\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** prep\_data(self, target\_list):  
 *"""This function is design to split you data in training in test"""* **try**:  
 df\_dtype = self.data\_new.dtypes  
 dtype\_dict = {**"integer"**: list(df\_dtype[df\_dtype == **'int64'**].index),  
 **"float"**: list(df\_dtype[df\_dtype == **'float64'**].index)}  
 data = self.data\_new.drop(target\_list, axis=1, errors=**'ignore'**)  
 attributes = list(data)  
 data = self.data\_new.drop(target\_list, axis=1).values  
 *# Split into train(90%) and test(10%)* x\_train, x\_test, y\_train, y\_test = \  
 train\_test\_split(data, self.data\_new[target\_list].values,  
 test\_size=.1, random\_state=10)  
 **return** x\_train, x\_test, y\_train, y\_test, attributes, dtype\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** linear\_regression(self, target\_list, factor\_cols\_list, imp\_features\_list):  
 *"""  
 This function run linear regression and returns linear equation output* **:param***:list of important features, target variable, dataset id* **:param***: target variables* **:param***: lens\_id* **:return***: list of labelled variables and converted values  
 """* **try**:  
 lin\_ret\_dict = {}  
 df\_series = pd.DataFrame()  
 \_, \_, \_, \_, \_, dtype\_dict = self.prep\_data(target\_list)  
 **for** i, \_ **in** enumerate(target\_list):  
 df\_return = pd.DataFrame(columns=[**'cols'**, **'coef'**, **'cat'**, **'min'**, **'max'**, **'avg'**])  
 model = sm.ols(formula=**" ~ "**.join([**""**.join(target\_list[i]),  
 **" + "**.join(imp\_features\_list)]),  
 data=self.data\_new)  
 fitted = model.fit()  
 fitted.summary()  
 df\_series = fitted.params  
 *# assign important variable names in cols* df\_return[**'cols'**] = df\_series.index  
 *# assign coefficients in coef* df\_return[**'coef'**] = pd.Series(df\_series.values)  
 *# Assign min, max , average and category value of important values in dataframe out* **for** j **in** imp\_features\_list:  
 df\_return.loc[df\_return.cols == j, **'min'**] =\  
 min(pd.Series(self.data\_new[j].values))  
  
 df\_return.loc[df\_return.cols == j, **'max'**] =\  
 max(pd.Series(self.data\_new[j].values))  
  
 **if** (j **in** self.obj\_cols) **or** (j **in** factor\_cols\_list):  
 df\_return.loc[df\_return.cols == j, **'cat'**] = **'cat'  
 elif** j **in** dtype\_dict[**"integer"**]:  
 df\_return.loc[df\_return.cols == j, **'cat'**] = **'int'  
 elif** j **in** dtype\_dict[**"float"**]:  
 df\_return.loc[df\_return.cols == j, **'cat'**] = **'float'  
  
 if** j **not in** factor\_cols\_list:  
 **if** j **in** dtype\_dict[**"float"**]:  
 df\_return.loc[df\_return.cols == j, **'avg'**] = \  
 round(np.average(pd.Series(self.data\_new[j].values)), 6)  
 **elif** j **in** dtype\_dict[**"integer"**]:  
 df\_return.loc[df\_return.cols == j, **'avg'**] = \  
 round(np.average(pd.Series(self.data\_new[j].values)))  
 **else**:  
 df\_return.loc[df\_return.cols == j, **'avg'**] = \  
 round(np.average(pd.Series(self.data\_new[j].values)), 6)  
 **else**:  
 **try**:  
 df\_return.loc[df\_return.cols == j, **'avg'**] =\  
 mode(pd.Series(self.data\_new[j].values))  
 **except** StatisticsError:  
 df\_return.loc[df\_return.cols == j, **'avg'**] = \  
 Counter(pd.Series(self.data\_new[j].values)).most\_common(1)[0][0]  
  
 *# Add target variable values in data frame* df\_return.loc[len(df\_return)] = [target\_list[i], -999999, **'num'**,  
 min(pd.Series(self.data\_new[target\_list[i]])),  
 max(pd.Series(self.data\_new[target\_list[i]])),  
 round(np.average(pd.Series(  
 self.data\_new[target\_list[i]])), 6)]  
 df\_return = df\_return.fillna(0)  
 df\_return[**"coef"**] = round(df\_return[**"coef"**], 6)  
 df\_return[**"coef"**].describe().apply(**lambda** x: format(x, **'f'**))  
 temp\_dict = Utilities.df\_to\_dict(df\_return)  
 temp\_dict.update({**"model\_accuracy"**: [round(fitted.rsquared\*100, 2),  
 round(fitted.rsquared\_adj\*100, 2)]})  
 lin\_ret\_dict[target\_list[i]] = [temp\_dict]  
 **return** lin\_ret\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
@author: Krishna Mohan  
This script does label encoding for factor cols , identify important features  
runs linear regression for What if Analysis part of LENS  
using XGboost algorithm  
"""***import** logging  
**from** statistics **import** mode, StatisticsError  
**from** collections **import** Counter  
**import** pickle  
**from** numpy **import** array\_str  
**import** numpy **as** np  
**import** pandas **as** pd  
**from** sklearn.preprocessing **import** LabelEncoder  
**from** sklearn.model\_selection **import** train\_test\_split  
**from** sklearn.metrics **import** accuracy\_score, precision\_score,\  
 recall\_score, confusion\_matrix, f1\_score  
**import** xgboost  
**import** message\_constants  
**from** insights.what\_if\_analysis.utilities\_function **import** Utilities  
np.set\_printoptions(suppress=True)  
np.seterr(divide=**'ignore'**, invalid=**'ignore'**)  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** WhatIfClassification: *# pylint: disable=R0914, R0912  
 """  
 Class that generates important features  
 """* **def** \_\_init\_\_(self, data\_new):  
 *""" to initialize our class with dataframe"""* **try**:  
 self.data\_new = data\_new  
 self.obj\_cols = []  
 self.imp\_feature\_dict = {}  
 self.important\_feature = []  
 self.model\_dict = {}  
 self.important\_feature\_dict = {}  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** label\_encoding(self, factor\_cols\_list):  
 *"""  
 This function does label encoding for all object type variables and make changes  
 in dataset and replaces categorical with numeric value* **:return***: list of labelled variables and converted values  
 """* **try**:  
 label\_encoding = LabelEncoder()  
 label\_dictionary = {}  
 *# Iterating over all object type columns* **for** col **in** self.data\_new[factor\_cols\_list].columns.values:  
 *# label Encoding for only categorical variables* **if** self.data\_new[col].dtypes == **'object'**:  
 self.obj\_cols.append(col)  
 *# label\_encoding.fit(self.data\_new[col].values)* **try**:  
 label\_encoding.fit(self.data\_new[col].values)  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 *# ValueError: could not convert string to float:* label\_encoding.fit(self.data\_new[col].values.astype(str))  
 le\_name\_mapping = dict(zip(label\_encoding.classes\_,  
 label\_encoding.transform(label\_encoding.classes\_)))  
 *# self.data\_new[col] = label\_encoding.transform(self.data\_new[col])* **try**:  
 self.data\_new[col] = label\_encoding.transform(self.data\_new[col])  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 *# TypeError: unorderable types: float() < str()* self.data\_new[col] =\  
 label\_encoding.transform(self.data\_new[col].astype(str))  
 label\_dictionary[col] = le\_name\_mapping  
 **return** label\_dictionary, self.data\_new  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** imp\_feature(self, target\_list):  
 *"""  
 This function does identify important variables* **:return***: list of labelled variables and converted values  
 """* **try**:  
 x\_train, \_, y\_train, \_, attributes, \_ = self.prep\_data(target\_list)  
 *# XGboost algorithm to get important variables* xgb = xgboost.XGBClassifier(n\_estimators=100, learning\_rate=0.08, gamma=0,  
 subsample=0.75, colsample\_bytree=1, max\_depth=7)  
 **for** i, target **in** enumerate(target\_list):  
 self.important\_feature\_dict = {}  
 model = xgb.fit(x\_train, y\_train[:, i])  
 feature\_labels = np.array(attributes)  
 self.model\_dict.update({target: model})  
 self.model\_dict.update({target+**'\_features'**: attributes})  
 *# serializing our model to a file called model.pkl  
 # pickle.dump(model, open("xgb\_model.pkl", "wb"))* **with** open(**"xgb\_model.pkl"**, **'wb'**) **as** file:  
 pickle.dump(model, file)  
 model\_supportive\_info\_dict = {**"target"**: target,  
 **"important\_feature"**: attributes,  
 **"target\_unique\_values"**: array\_str(  
 pd.unique(self.data\_new[target\_list[0]]))}  
 **with** open(**"model\_supportive\_info.pkl"**, **'wb'**) **as** file:  
 pickle.dump(model\_supportive\_info\_dict, file)  
 importance = model.feature\_importances\_  
 feature\_indexes\_by\_importance = importance.argsort()  
 **for** index **in** feature\_indexes\_by\_importance:  
 **if** feature\_labels[index] **in** list(self.important\_feature\_dict.keys()):  
 self.important\_feature\_dict.update({feature\_labels[index]:  
 round((importance[index]  
 + self.important\_feature\_dict[  
 feature\_labels[index]])/2,  
 2)})  
 **else**:  
 self.important\_feature\_dict.update({feature\_labels[index]:  
 round(importance[index], 2)})  
 self.important\_feature = self.important\_feature\_dict.keys()  
 **return** {key: value **for** (key, value) **in** self.important\_feature\_dict.items()  
 **if** value > .03}, self.important\_feature\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** prep\_data(self, target\_list, imp\_feature\_list=None):  
 *"""This function is design to split you data in training in test"""* **try**:  
 **if** imp\_feature\_list **is not** None:  
 data = self.data\_new[imp\_feature\_list]  
 df\_dtype = data.dtypes  
 **else**:  
 df\_dtype = self.data\_new.dtypes  
 dtype\_dict = {**"integer"**: list(df\_dtype[df\_dtype == **'int64'**].index),  
 **"float"**: list(df\_dtype[df\_dtype == **'float64'**].index)}  
 data = self.data\_new.drop(target\_list, axis=1, errors=**'ignore'**)  
 **if** imp\_feature\_list **is not** None:  
 data = data[imp\_feature\_list]  
 attributes = list(data)  
 **if** imp\_feature\_list **is not** None:  
 data = self.data\_new[imp\_feature\_list].values  
 **else**:  
 data = self.data\_new.drop(target\_list, axis=1).values  
 *# Split into train(90%) and test(10%)* x\_train, x\_test, y\_train, y\_test = \  
 train\_test\_split(data, self.data\_new[target\_list].values,  
 test\_size=.2, random\_state=10)  
 **return** x\_train, x\_test, y\_train, y\_test, attributes, dtype\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** classification(self, target\_list, factor\_cols\_list, imp\_features\_list):  
 *"""  
 This function run linear regression and returns linear equation output* **:param***:list of important features, target variable, dataset id* **:param***: target variables* **:param***: lens\_id* **:return***: list of labelled variables and converted values  
 """* **try**:  
 lin\_ret\_dict = {}  
 x\_train, x\_test, y\_train, y\_test, \_, dtype\_dict = \  
 self.prep\_data(target\_list, imp\_features\_list)  
  
 xgb = xgboost.XGBClassifier(n\_estimators=100, learning\_rate=0.08, gamma=0,  
 subsample=0.75, colsample\_bytree=1, max\_depth=7)  
 **for** i, target **in** enumerate(target\_list):  
 df\_return = pd.DataFrame(columns=[**'cols'**, **'coef'**, **'cat'**, **'min'**, **'max'**, **'avg'**])  
  
 **if** len(imp\_features\_list) != len(self.important\_feature):  
 model = xgb.fit(x\_train, y\_train[:, i])  
 self.model\_dict.update({target: model})  
 self.model\_dict.update({target+**"\_features"**: imp\_features\_list})  
 self.important\_feature = imp\_features\_list  
 model\_supportive\_info\_dict = {**"target"**: target,  
 **"important\_feature"**: imp\_features\_list,  
 **"target\_unique\_values"**:  
 array\_str(pd.unique(  
 self.data\_new[target\_list[0]]))}  
 *# serializing our model to a file called model.pkl  
 # pickle.dump(model, open("xgb\_model.pkl", "wb"))* **with** open(**"xgb\_model.pkl"**, **'wb'**) **as** file:  
 pickle.dump(model, file)  
 **with** open(**"model\_supportive\_info.pkl"**, **'wb'**) **as** file:  
 pickle.dump(model\_supportive\_info\_dict, file)  
 *# assign important variable names in cols* df\_return[**'cols'**] = imp\_features\_list  
 *# assign coefficients in coef* df\_return[**'coef'**] = None  
 *# Assign min, max , average and category value of important values in data frame out* **for** j **in** imp\_features\_list:  
 df\_return.loc[df\_return.cols == j, **'min'**] =\  
 min(pd.Series(self.data\_new[j].values))  
  
 df\_return.loc[df\_return.cols == j, **'max'**] =\  
 max(pd.Series(self.data\_new[j].values))  
  
 **if** (j **in** self.obj\_cols) **or** (j **in** factor\_cols\_list):  
 df\_return.loc[df\_return.cols == j, **'cat'**] = **'cat'  
 elif** j **in** dtype\_dict[**"integer"**]:  
 df\_return.loc[df\_return.cols == j, **'cat'**] = **'int'  
 elif** j **in** dtype\_dict[**"float"**]:  
 df\_return.loc[df\_return.cols == j, **'cat'**] = **'float'  
  
 if** j **not in** factor\_cols\_list:  
 **if** j **in** dtype\_dict[**"float"**]:  
 df\_return.loc[df\_return.cols == j, **'avg'**] = \  
 round(np.average(pd.Series(self.data\_new[j].values)), 6)  
 **elif** j **in** dtype\_dict[**"integer"**]:  
 df\_return.loc[df\_return.cols == j, **'avg'**] = \  
 round(np.average(pd.Series(self.data\_new[j].values)))  
 **else**:  
 df\_return.loc[df\_return.cols == j, **'avg'**] = \  
 round(np.average(pd.Series(self.data\_new[j].values)), 6)  
 **else**:  
 **try**:  
 df\_return.loc[df\_return.cols == j, **'avg'**] =\  
 mode(pd.Series(self.data\_new[j].values))  
 **except** StatisticsError:  
 df\_return.loc[df\_return.cols == j, **'avg'**] = \  
 Counter(pd.Series(self.data\_new[j].values)).most\_common(1)[0][0]  
  
 *# Add target variable values in data frame* df\_return.loc[len(df\_return)] = [target\_list[i], -999999, **'num'**,  
 min(pd.Series(self.data\_new[target\_list[i]])),  
 max(pd.Series(self.data\_new[target\_list[i]])),  
 round(np.average(pd.Series(  
 self.data\_new[target\_list[i]])), 6)]  
 df\_return = df\_return.fillna(0)  
 df\_return[**"coef"**] = round(df\_return[**"coef"**], 6)  
 df\_return[**"coef"**].describe().apply(**lambda** x: format(x, **'f'**))  
 temp\_dict = Utilities.df\_to\_dict(df\_return)  
 y\_pred = self.model\_dict[target].predict(x\_test)  
 temp\_dict.update({**"model\_accuracy"**: [round(accuracy\_score(y\_test, y\_pred)\*100, 2),  
 round(precision\_score(y\_test, y\_pred)\*100, 2),  
 round(recall\_score(y\_test, y\_pred) \* 100, 2),  
 array\_str(confusion\_matrix(y\_test, y\_pred)),  
 round(f1\_score(y\_test, y\_pred), 2)  
 ]})  
 lin\_ret\_dict[target\_list[i]] = [temp\_dict]  
 **return** lin\_ret\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*Insights*

*-------------------------------*

*# -\*- coding: utf-8 -\*-  
"""  
This module is used for insights services like what if analysis.  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** json  
**from** json **import** loads  
**import** itertools  
**from** flask **import** jsonify, Blueprint, request, render\_template  
**from** flask\_cors **import** cross\_origin  
**from** pandas **import** unique  
**from** flask\_socketio **import** SocketIO  
**from** celery **import** uuid  
**from** insights.utilities **import** get\_cols\_and\_types  
**from** insights.what\_if\_analysis.what\_if\_analysis **import** WhatIfAnalysis  
**from** insights.what\_if\_analysis.what\_if\_classification **import** WhatIfClassification  
**from** insights.what\_if\_analysis.utilities\_function **import** Utilities  
**from** insights.utilities.database\_utilities **import** DbUtilities  
**from** insights.time\_series.time\_series\_forecasting **import** TimeSeriesForecast  
**from** insights.cluster.cluster **import** Cluster  
**from** insights.auto\_insights.auto\_insights\_generator **import** AutoInsight  
**from** insights.auto\_insights.knowledge\_base\_operations **import** KnowledgeBase  
**from** insights.auto\_insights.celery\_tasks **import** CeleryTasks  
**from** profiling.generate\_profiling **import** GenerateProfile  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** flask **import** g  
**import** constants  
**from** token\_registry **import** TokenRegistry  
  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
INSIGHTS\_SERVER = Blueprint(**'insights\_server'**, \_\_name\_\_, url\_prefix=**"/mashup-service"**)  
  
*# maps userid with socketiosessionid*CLIENTSDICT = {}  
*# key user id , value is list of uuid(celery task)*MESSAGESDICT = {}  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/get\_num\_cols"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** target\_num\_cols():  
 *""" What if Analysis - Web API to fetch numeric cols of selected lens* **:param***: dataset ID* **:param***: lens ID* **:return***: list of numeric variables  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 data\_source\_id = json\_response[constants.DATA\_SOURCE\_ID]  
 lens\_id = json\_response[constants.LENS\_ID]  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(data\_source\_id,  
 lens\_id).return\_cols\_and\_types()  
 return\_dict = {constants.NUMERIC\_COLS: cols\_json\_file[constants.NUMERIC\_COLS]}  
 **return** jsonify(str(return\_dict))  
 **except** MemoryError:  
 **raise  
 except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/imp\_features"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** imp\_features():  
 *"""  
 What if Analysis - Web API is used to to identify important variables against  
 target variable list* **:param***: list of targets* **:return***: list of important variables for these target list  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(json\_response[constants.DATA\_SOURCE\_ID],  
 json\_response[constants.LENS\_ID]). \  
 return\_cols\_and\_types()  
 factor\_col\_list = cols\_json\_file[constants.FACTOR\_COLS]  
 data\_new = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=  
 json\_response[constants.DATA\_SOURCE\_ID],  
 lens\_id=json\_response[constants.LENS\_ID],  
 data\_frame\_type=**'WORKING'**)  
 data\_new = data\_new.drop(cols\_json\_file[constants.ID\_COLS], axis=1)  
 data\_new = data\_new.drop(cols\_json\_file[constants.DATE\_COLS], axis=1)  
 \_, data\_new = WhatIfAnalysis(data\_new).label\_encoding(factor\_col\_list)  
 top\_imp\_features, \_ = \  
 WhatIfAnalysis(data\_new).imp\_feature(json\_response[constants.TARGET\_LIST])  
 return\_dict = {constants.IMP\_FEATURES\_LIST: top\_imp\_features}  
 **return** jsonify(str(return\_dict))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/get\_regression"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** linear\_regression():  
 *"""  
 What if Analysis - Web API is used to get label encoding for factor variables* **:param***: list of targets* **:param***: data source id  
 param: lens id* **:param***: list of important features* **:return***: list of important variables for these target list  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 imp\_features\_list = json\_response[constants.IMP\_FEATURES\_LIST]  
 target\_list = json\_response[constants.TARGET\_LIST]  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(json\_response[constants.DATA\_SOURCE\_ID],  
 json\_response[constants.LENS\_ID]). \  
 return\_cols\_and\_types()  
 factor\_col\_list = cols\_json\_file[constants.FACTOR\_COLS]  
 data\_new = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=  
 json\_response[constants.DATA\_SOURCE\_ID],  
 lens\_id=json\_response[constants.LENS\_ID],  
 data\_frame\_type=**'WORKING'**)  
 data\_new = data\_new.drop(cols\_json\_file[constants.ID\_COLS], axis=1)  
 data\_new = data\_new.drop(cols\_json\_file[constants.DATE\_COLS], axis=1)  
 label\_encoding\_dict, \_ = WhatIfAnalysis(data\_new).label\_encoding(factor\_col\_list)  
 linear\_regression\_dict = WhatIfAnalysis(data\_new).linear\_regression(target\_list,  
 factor\_col\_list,  
 imp\_features\_list)  
 label\_encoding\_unique = [Utilities.final\_label\_encoding(label\_encoding\_dict,  
 imp\_features\_list)]  
 return\_dict = {constants.LINEAR\_REGRESSION: [linear\_regression\_dict],  
 constants.LABEL\_ENCODING: label\_encoding\_unique}  
 **return** jsonify({**"what\_if\_analysis"**: str(return\_dict)})  
  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/save\_scenario"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** save\_scenario():  
 *"""  
 What if Analysis - Web API is used to save new scenario in database* **:param***: what\_if\_master\_id for database foreign key* **:param***: scenario\_name* **:param***: scenario\_input* **:param***: scenario\_output* **:param***: User\_name* **:return***: return new scenario id  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 *#user\_name = g.user["prsId"]* user\_name = json\_response[**"user\_id"**]  
 input\_dict = {constants.DATA\_SOURCE\_ID: json\_response[constants.DATA\_SOURCE\_ID]}  
 saved\_scenarios = DbUtilities().select\_value(class\_id=constants.SCENARIO\_NAME,  
 \*\*input\_dict)  
 saved\_scenarios = [list(i) **for** i **in** saved\_scenarios]  
 *# convert list of list to flat list using itertools chain* saved\_scenarios = list(itertools.chain(\*saved\_scenarios))  
 **if** json\_response[constants.SCENARIO\_NAME] **in** saved\_scenarios:  
 **raise** Exception(**"Scenario name already exist.Please select a different name."**)  
  
 *# preparing input json for insert in insights\_ml\_scenario table* input\_dict = {constants.DATA\_SOURCE\_ID: json\_response[constants.DATA\_SOURCE\_ID],  
 constants.LENS\_ID: json\_response[constants.LENS\_ID],  
 constants.SCENARIO\_NAME: json\_response[constants.SCENARIO\_NAME],  
 constants.SCENARIO\_INPUT: str(json\_response[constants.SCENARIO\_INPUT]),  
 constants.SCENARIO\_OUTPUT: str(json\_response[constants.SCENARIO\_OUTPUT]),  
 constants.CREATED\_BY: user\_name,  
 constants.UPDATED\_BY: user\_name}  
 scenario\_id = DbUtilities().db\_insert(class\_id=constants.INSIGHTS\_ML\_SCENARIOS,  
 \*\*input\_dict)  
 **return** jsonify({constants.SCENARIO\_ID: str(scenario\_id)})  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.error(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/list\_scenario"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** list\_scenario():  
 *"""  
 What if Analysis - Web API is used to save new scenario in database* **:param***: what\_if\_master\_id for database foreign key* **:param***: scenario\_name* **:param***: scenario\_input* **:param***: scenario\_output* **:param***: User\_name* **:return***: return new scenario id  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 input\_dict = {constants.DATA\_SOURCE\_ID: json\_response[constants.DATA\_SOURCE\_ID]}  
 qry\_result = DbUtilities().select\_value(class\_id=constants.INSIGHTS\_ML\_SCENARIOS,  
 \*\*input\_dict)  
 **return** jsonify({**"Existing\_Scenarios"**: str(qry\_result)})  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/get\_cols\_types"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** get\_cols\_types():  
 *"""  
 Time Series - Web API to get categorical columns list* **:param***: DATA\_SOURCE\_ID* **:param***: LENS\_ID* **:return***: list of categorical columns  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(json\_response[constants.DATA\_SOURCE\_ID],  
 json\_response[constants.LENS\_ID]). \  
 return\_cols\_and\_types()  
 return\_dict = {constants.NUMERIC\_COLS: cols\_json\_file[constants.NUMERIC\_COLS],  
 constants.CATEGORICAL\_FEATURES: cols\_json\_file[constants.FACTOR\_COLS],  
 constants.DATE\_COLS: cols\_json\_file[constants.DATE\_COLS],  
 constants.ID\_COLS: cols\_json\_file[constants.ID\_COLS]}  
 **return** jsonify(str(return\_dict))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/cat\_unique\_values"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** cat\_unique\_values():  
 *"""  
 Time Series - Web API to get all unique values of categorical columns* **:param***: DATA\_SOURCE\_ID* **:param***: LENS\_ID* **:return***: json of variables and respective unique values  
 """* **try**:  
 col\_val = {}  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(json\_response[constants.DATA\_SOURCE\_ID],  
 json\_response[constants.LENS\_ID]). \  
 return\_cols\_and\_types()  
 factor\_col\_list = cols\_json\_file[constants.FACTOR\_COLS]  
 data\_new = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=  
 json\_response[constants.DATA\_SOURCE\_ID],  
 lens\_id=json\_response[constants.LENS\_ID],  
 data\_frame\_type=**'WORKING'**)  
 **for** \_, value **in** enumerate(factor\_col\_list):  
 col\_val.update({value: list(unique(data\_new[value]))})  
 **return** jsonify(str(col\_val))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/time\_series\_forecast"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** time\_series\_forecast():  
 *"""  
 Time Series - Web API to forecast time series KPI* **:param***: DATA\_SOURCE\_ID* **:param***: LENS\_ID* **:param***: Time Series Date column* **:param***: Time Series KPI* **:param***: Analysis Frequency (Weekly, Monthly, Yearly depends on data)* **:param***: forecast period* **:return***: json of variables and respective unique values  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 ts\_forecast\_obj = TimeSeriesForecast(json\_response[constants.DATA\_SOURCE\_ID],  
 lens\_id=json\_response[constants.LENS\_ID])  
 ts\_forecast\_obj.apply\_filters(json\_response[constants.FILTER\_DICT])  
 ts\_index, ts\_data, confidence\_int, model\_accuracy = \  
 ts\_forecast\_obj.time\_series\_forecast(json\_response[constants.TS\_DATE],  
 json\_response[constants.TS\_KPI],  
 json\_response[constants.TS\_FREQ],  
 json\_response[constants.FORECAST\_PERIODS])  
 ret\_dict = {**"Date"**: ts\_index,  
 **"KPI"**: ts\_data,  
 **"Confidence\_Interval"**: confidence\_int,  
 **"model\_accuracy"**: model\_accuracy}  
 **return** jsonify(str(ret\_dict))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/clustering"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** clustering():  
 *"""  
 Cluster Analysis - Web API to cluster dataset* **:param***: DATA\_SOURCE\_ID* **:param***: LENS\_ID* **:param***: USER\_NAME* **:param***: Time Series KPI* **:param***: Analysis period (Weekly, Monthly, Yearly depends on data)* **:param***: LENS\_ID* **:return***: json of variables and respective unique values  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 cluster\_obj = Cluster(data\_source\_id=json\_response[constants.DATA\_SOURCE\_ID],  
 lens\_id=json\_response[constants.LENS\_ID])  
 ret\_dict = cluster\_obj.clustering(json\_response[constants.CLUSTER])  
 **return** jsonify(str(ret\_dict))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/all\_imp\_features"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** all\_imp\_features():  
 *"""  
 What if Analysis - Web API is used to to identify important variables against  
 target variable list* **:param***: list of targets* **:return***: list of important variables for these target list  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 data\_source\_id = json\_response[constants.DATA\_SOURCE\_ID]  
 lens\_id = json\_response[constants.LENS\_ID]  
 target\_list = [constants.CLUSTER\_ID]  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(data\_source\_id,  
 lens\_id).return\_cols\_and\_types()  
 numeric\_col\_list = cols\_json\_file[constants.NUMERIC\_COLS]  
 numeric\_col\_list.append(constants.CLUSTER\_ID)  
 data\_new = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=data\_source\_id,  
 lens\_id=lens\_id,  
 data\_frame\_type=**'CLUSTER'**)  
 data\_new = data\_new.drop(cols\_json\_file[constants.ID\_COLS], axis=1)  
 data\_new = data\_new.drop(cols\_json\_file[constants.DATE\_COLS], axis=1)  
 data\_new = data\_new[numeric\_col\_list]  
 \_, data\_new = WhatIfAnalysis(data\_new).label\_encoding(target\_list)  
 \_, all\_imp\_features\_dict = WhatIfAnalysis(data\_new).imp\_feature(target\_list)  
 **return** jsonify(str(all\_imp\_features\_dict))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/cluster\_col\_profile"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** cluster\_col\_profile():  
 *"""  
 What if Analysis - Web API is used to to identify important variables against  
 target variable list* **:param***: list of targets* **:return***: list of important variables for these target list  
 """* **try**:  
 *#user\_name = g.user["prsId"]* user\_name = json\_response[**"user\_id"**]  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 *#json\_response.update({"user\_id": user\_name})  
 # filter data on clusters and add values in profile dictionary* cluster\_profile\_dict = \  
 Cluster.cluster\_col\_profile(json\_response[constants.DATA\_SOURCE\_ID],  
 json\_response[constants.LENS\_ID],  
 data\_frame\_type=**"CLUSTER"**,  
 column\_name=json\_response[constants.COLUMN\_NAME],  
 user\_id=json\_response[constants.USER\_ID])  
 **return** jsonify(str(cluster\_profile\_dict))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/auto\_insights"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** auto\_insights():  
 *"""Web API to run Clustering, Time Series and what if analysis  
  
 Also includes OUTLIER code* **:return***: Json to populate all features text in panel  
 """* **try**:  
 *#user\_name = g.user["prsId"]* user\_name = json\_response[**"user\_id"**]  
 bearer\_token = TokenRegistry.get\_token\_for\_user(user\_id=user\_name)[**"Authorization"**]  
 input\_list = request.get\_json()  
 data\_source\_id = input\_list[0][**"dataSourceId"**]  
 run\_type\_list = input\_list[0][**"runType"**]  
 data\_source\_name = input\_list[0][**"dataSourceName"**]  
 input\_json = {}  
 input\_json.update({**"data\_source\_name"**: data\_source\_name})  
 input\_json.update({**"knowledgebase"**: input\_list})  
 input\_json.update({**"data\_source\_id"**: data\_source\_id})  
 input\_json.update({**"user\_id"**: user\_name})  
 input\_json.update({**"token"**: bearer\_token})  
  
 **if "timeseries\_outlier" in** run\_type\_list:  
 *# outlier code* task\_id = uuid() *# generate uuid for task manually* user\_id = input\_json.get(**"user\_id"**)  
  
 **if** MESSAGESDICT.get(user\_id):  
 MESSAGESDICT[user\_id].append(task\_id)  
  
 **else**:  
 MESSAGESDICT[user\_id] = [task\_id]  
  
 input\_json[**"uuid"**] = task\_id  
 result = CeleryTasks.sleepfunc.apply\_async(args=[input\_json], task\_id=task\_id)  
 input\_json[**"status"**] = **"Started"** socketio = SocketIO(message\_queue=**'amqp://'**)  
  
 **if** CLIENTSDICT.get(user\_id) **is not** None:  
 socketio.emit(**'message'**, json.dumps(input\_json), room=CLIENTSDICT.get(user\_id),  
 namespace=**'/mashup-service'**)  
  
 **del** input\_json[**"uuid"**]  
 **del** input\_json[**"status"**]  
 **return ""  
  
 else**:  
 **return** jsonify(AutoInsight(input\_json).run\_insights\_on\_data\_source())  
  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/update\_knowledge\_base"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** update\_knowledge\_base():  
 *"""Web API to update knowledge base for what if analysis* **:return***: Json to populate all features text in panel  
 """* **try**:  
 *#user\_name = g.user["prsId"]* user\_name = json\_response[**"user\_id"**]  
 request.get\_json().update({**"user\_id"**: user\_name})  
 **return** jsonify(KnowledgeBase(request.get\_json()).feedback\_update\_knowledge\_base())  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/data\_profile\_imputation\_suggestions"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** profile\_imputation\_suggestion():  
 *"""Web API to run overall data profiling, imputation suggestion and reasons* **:param***:data source id* **:param***:lens\_id* **:param***:column\_name should be null* **:param***:CLUSTERING\_ENABLE should be "Working"* **:return***: Json to populate all features profile with auto imputation recommendations and reasons  
 """* **try**:  
 *#user\_name = g.user["prsId"]  
 #user\_name = json\_response["user\_id"]* input\_json = request.get\_json()  
 *#input\_json.update({"user\_id": user\_name})* **return** jsonify(GenerateProfile(input\_json).generate\_profile\_dataframe(  
 input\_json[constants.COLUMN\_TYPE]))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/class\_target"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** class\_target():  
 *"""Web API to display target variables for classification problem* **:param***:data source id* **:param***:lens\_id* **:return***: column list having 10 unique classes (values) interger and categorical values  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 **return** jsonify(str(Utilities.classification\_target(json\_response[constants.DATA\_SOURCE\_ID],  
 json\_response[constants.LENS\_ID])))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/class\_imp\_features"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** class\_imp\_features():  
 *"""  
 What if Analysis - Web API is used to to identify important variables against  
 target variable list* **:param***: list of targets* **:return***: list of important variables for these target list  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(json\_response[constants.DATA\_SOURCE\_ID],  
 json\_response[constants.LENS\_ID]). \  
 return\_cols\_and\_types()  
 factor\_col\_list = cols\_json\_file[constants.FACTOR\_COLS]  
 data\_new = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=  
 json\_response[constants.DATA\_SOURCE\_ID],  
 lens\_id=json\_response[constants.LENS\_ID],  
 data\_frame\_type=**'WORKING'**)  
 data\_new = data\_new.drop(cols\_json\_file[constants.ID\_COLS], axis=1)  
 data\_new = data\_new.drop(cols\_json\_file[constants.DATE\_COLS], axis=1)  
 what\_if\_class\_obj = WhatIfClassification(data\_new)  
 \_, data\_new = what\_if\_class\_obj.label\_encoding(factor\_col\_list +  
 json\_response[constants.TARGET\_LIST])  
 top\_imp\_features, \_ = what\_if\_class\_obj.imp\_feature(json\_response[constants.TARGET\_LIST])  
 **return** jsonify(str({constants.IMP\_FEATURES\_LIST: top\_imp\_features}))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/classification"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** classification\_model():  
 *"""  
 What if Classification to run classification XG boost model* **:param***: list of targets* **:param***: list of targets* **:param***: list of targets* **:return***: Classification model output for with model accuracy, predicted class  
 along with probabilities of all class  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 imp\_features\_list = json\_response[constants.IMP\_FEATURES\_LIST]  
 target\_list = json\_response[constants.TARGET\_LIST]  
 cols\_json\_file = get\_cols\_and\_types.GetColsAndType(json\_response[constants.DATA\_SOURCE\_ID],  
 json\_response[constants.LENS\_ID]). \  
 return\_cols\_and\_types()  
 factor\_col\_list = cols\_json\_file[constants.FACTOR\_COLS]  
 data\_new = DataRegistry.get\_data\_set\_from\_dict(data\_source\_id=  
 json\_response[constants.DATA\_SOURCE\_ID],  
 lens\_id=json\_response[constants.LENS\_ID],  
 data\_frame\_type=**'WORKING'**)  
 data\_new = data\_new.drop(cols\_json\_file[constants.ID\_COLS], axis=1)  
 data\_new = data\_new.drop(cols\_json\_file[constants.DATE\_COLS], axis=1)  
 label\_encoding\_dict, \_ = \  
 WhatIfClassification(data\_new).label\_encoding(factor\_col\_list +  
 json\_response[constants.TARGET\_LIST])  
 **print**(**"label\_encoding\_dict"**, label\_encoding\_dict)  
 class\_model\_dict = WhatIfClassification(data\_new).classification(target\_list,  
 factor\_col\_list,  
 imp\_features\_list)  
 *# label\_encoding\_unique = [Utilities.final\_label\_encoding(label\_encoding\_dict,  
 # imp\_features\_list)]* return\_dict = {**"classification\_model"**: [class\_model\_dict],  
 constants.LABEL\_ENCODING: label\_encoding\_dict}  
 **return** jsonify({**"what\_if\_classification"**: str(return\_dict)})  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**"/predict\_method"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** predict\_method():  
 *"""  
 What if Classification to run classification XG boost model* **:param***: list of targets* **:param***: list of targets* **:param***: list of targets* **:return***: Classification model output for with model accuracy, predicted class  
 along with probabilities of all class  
 """* **try**:  
 json\_response = loads(request.data.decode(**'utf-8'**))  
 return\_dict = Utilities.predict\_classifier(json\_response[**"x\_test\_json"**],  
 json\_response[constants.TARGET\_LIST])  
 **return** jsonify(str(return\_dict))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
*############################################################################################3*@cross\_origin(INSIGHTS\_SERVER)  
@INSIGHTS\_SERVER.route(**'/send'**, methods=[**"POST"**])  
**def** sendmessage():  
 *"""This functions sends notification to clients, all batch jobs send requests internally to this  
 function  
  
 """* **try**:  
 send\_json = request.get\_json()  
  
 user\_id = send\_json.get(**'user\_id'**)  
 socketio = SocketIO(message\_queue=**'amqp://'**)  
  
 **if** CLIENTSDICT.get(user\_id) **is not** None:  
 **print**(**"webserver "** + str(socketio))  
 socketio.emit(**'ack-task'**, json.dumps(send\_json), room=CLIENTSDICT.get(user\_id),  
 namespace=**'/mashup-service'**)  
  
 **return "Sent"  
 except** Exception **as** ex:  
 **return "Exception "** + ex.args[0]  
  
  
*############################## DELETE THIS ###############*@INSIGHTS\_SERVER.route(**'/mq'**, methods=[**"GET"**])  
**def** getmq():  
 *"""gets messages dict"""* **return** jsonify(MESSAGESDICT)  
  
  
@INSIGHTS\_SERVER.route(**'/clients'**, methods=[**"GET"**])  
**def** getclients():  
 *"""gets clients dict"""* **return** jsonify(CLIENTSDICT)  
  
  
@INSIGHTS\_SERVER.route(**'/'**)  
**def** index():  
 *"""temp ui for testing"""* **return** render\_template(**'index.html'**)  
  
*##############################*

*"""This module is used to define all mysql db structure models"""***from** flask\_sqlalchemy **import** SQLAlchemy  
**from** sqlalchemy **import** MetaData  
**from** config\_manager **import** CONFIG\_MANAGER  
**from** insights.utils **import** default\_id  
  
schema = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'DB\_SCHEMA'**))  
metadata= MetaData(schema=schema)  
db = SQLAlchemy(metadata=metadata)  
  
**class** Scenarios(db.Model):  
 *""" It contains db structure for UserInput table"""* \_\_tablename\_\_ = **'insights\_ml\_scenarios'** scenario\_id = db.Column(db.String(255), default=default\_id, primary\_key=True)  
 data\_source\_id = db.Column(db.Integer, nullable=True)  
 lens\_id = db.Column(db.Integer, nullable=True)  
 scenario\_name = db.Column(db.String(255), nullable=True)  
 scenario\_input = db.Column(db.CLOB, nullable=True)  
 scenario\_output = db.Column(db.CLOB, nullable=True)  
 created\_by = db.Column(db.String(255), nullable=True)  
 created\_date = db.Column(db.DateTime(timezone=True))  
 updated\_by = db.Column(db.String(255), nullable=True)  
 updated\_date = db.Column(db.DateTime(timezone=True))  
  
 **def** \_\_repr\_\_(self): *# pragma: no cover  
 """"function to fetch class variable -- select query"""* **return '{"scenario\_name": "%s", "scenario\_input": %s, "scenario\_output": %s}'** % \  
 (self.scenario\_name, self.scenario\_input, self.scenario\_output)

*"""Test file for models.py"""***from** unittest **import** TestCase  
**from** sqlalchemy **import** create\_engine  
**from** sqlalchemy.ext.declarative **import** declarative\_base  
**from** sqlalchemy.orm **import** Session  
**from** insights.models **import** InsightMaster, UserInput, WhatIfMaster, Scenarios  
**import** constants  
  
BASE = declarative\_base()  
  
  
**class** TestInsightMaster(TestCase):  
 *"""This class is test class for InsightMaster"""* **def** setUp(self):  
 self.engine = create\_engine(constants.MYSQL\_CONNECTION\_URL)  
 self.session = Session(self.engine)  
 BASE.metadata.create\_all(self.engine)  
 self.insight\_master = InsightMaster(user\_name=**'km'**,  
 lens\_id=1,  
 data\_source\_id=1,  
 created\_by=**'km'**,  
 updated\_by=**'km'**)  
 self.session.add(self.insight\_master)  
 self.session.commit()  
  
 **def** tearDown(self):  
 self.session.delete(self.insight\_master)  
 self.session.commit()  
 BASE.metadata.drop\_all(self.engine)  
  
 **def** test\_repr\_insight\_master(self):  
 *"""Test function for \_\_repr\_\_"""* expected = self.insight\_master  
 result = self.session.query(InsightMaster).first()  
 self.assertEqual(result, expected)  
  
  
**class** TestUserInput(TestCase):  
 *"""This class is test class for UserInput"""* **def** setUp(self):  
 self.engine = create\_engine(constants.MYSQL\_CONNECTION\_URL)  
 self.session = Session(self.engine)  
 BASE.metadata.create\_all(self.engine)  
 self.insight\_master = InsightMaster(user\_name=**'km'**, lens\_id=1, data\_source\_id=1,  
 created\_by=**'km'**, updated\_by=**'km'**)  
 self.user\_input = UserInput(insight\_master\_id=1, target=[**"price"**],  
 cols\_removed=[**"ID"**], factor\_cols=[**"view"**],  
 numeric\_cols=[**"rev"**], id\_cols=[**"ID"**],  
 date\_cols=[**"daily\_date"**], kpi=None,  
 group\_col=None, ts\_date=None)  
 self.session.add(self.insight\_master)  
 self.session.add(self.user\_input)  
 self.session.commit()  
  
 **def** tearDown(self):  
 self.session.delete(self.insight\_master)  
 self.session.delete(self.user\_input)  
 self.session.commit()  
 BASE.metadata.drop\_all(self.engine)  
  
 **def** test\_repr\_user\_input(self):  
 *"""Test function for \_\_repr\_\_"""* expected = self.user\_input  
 result = self.session.query(UserInput).first()  
 self.assertEqual(result, expected)  
  
  
**class** TestWhatIfMaster(TestCase):  
 *"""This class is test class for WhatIfMaster"""* **def** setUp(self):  
 self.engine = create\_engine(constants.MYSQL\_CONNECTION\_URL)  
 self.session = Session(self.engine)  
 BASE.metadata.create\_all(self.engine)  
 self.insight\_master = InsightMaster(user\_name=**'km'**, lens\_id=1, data\_source\_id=1,  
 created\_by=**'km'**, updated\_by=**'km'**)  
 self.what\_if\_master = WhatIfMaster(insight\_master\_id=1, target\_column=[**"price"**],  
 target\_identifier=[**"price"**], column\_list=[**"view"**],  
 coefficients=[-59.04],  
 label\_encoding=None)  
 self.session.add(self.insight\_master)  
 self.session.add(self.what\_if\_master)  
 self.session.commit()  
  
 **def** tearDown(self):  
 self.session.delete(self.insight\_master)  
 self.session.delete(self.what\_if\_master)  
 self.session.commit()  
 BASE.metadata.drop\_all(self.engine)  
  
 **def** test\_repr\_what\_if(self):  
 *"""Test function for \_\_repr\_\_"""* expected = self.what\_if\_master  
 result = self.session.query(WhatIfMaster).first()  
 self.assertEqual(result, expected)  
  
  
**class** TestScenarios(TestCase):  
 *"""This class is test class for Scenarios"""* **def** setUp(self):  
 self.engine = create\_engine(constants.MYSQL\_CONNECTION\_URL)  
 self.session = Session(self.engine)  
 BASE.metadata.create\_all(self.engine)  
 self.insight\_master = InsightMaster(user\_name=**'km'**, lens\_id=1, data\_source\_id=1,  
 created\_by=**'km'**, updated\_by=**'km'**)  
 self.what\_if\_master = WhatIfMaster(insight\_master\_id=1, target\_column=[**"price"**],  
 target\_identifier=[**"price"**], column\_list=[**"view"**],  
 coefficients=[-59.04],  
 label\_encoding=None)  
 self.scenarios = Scenarios(what\_if\_master\_id=1, scenario\_name=**"New Scenario"**,  
 scenario\_input=[**"price"**], scenario\_output=[**"view"**])  
 self.session.add(self.insight\_master)  
 self.session.add(self.what\_if\_master)  
 self.session.add(self.scenarios)  
 self.session.commit()  
  
 **def** tearDown(self):  
 self.session.delete(self.insight\_master)  
 self.session.delete(self.what\_if\_master)  
 self.session.delete(self.scenarios)  
 self.session.commit()  
 BASE.metadata.drop\_all(self.engine)  
  
 **def** test\_repr\_scenario(self):  
 *"""Test function for \_\_repr\_\_"""* expected = self.scenarios  
 result = self.session.query(Scenarios).first()  
 self.assertEqual(result, expected)

*#! -\*- coding: utf-8 -\*-  
"""  
Util file for insights models  
"""***from** uuid **import** uuid4  
  
  
**def** default\_id():  
 *""" Generate UUID for Oracle Primary key"""* **return** str(uuid4())

*Join*

*"""  
This module is used to perform join operation like left , right , outer and inner  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** pandas **as** pd  
**import** constants  
**import** message\_constants  
**from** json\_parser\_utility **import** JsonUtility  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** JoinStep:  
 *"""  
 This class is used to join implementation with the data set  
 """* **def** \_\_init\_\_(self, json\_value):  
 **try**:  
 self.join\_type = json\_value[constants.JOIN\_TYPE].lower()  
 self.user\_id = json\_value[constants.USER\_ID]  
 json\_utility = JsonUtility()  
 self.left\_fields = json\_utility.required\_key\_value\_from\_json(  
 constants.FIELD\_NAME,  
 json\_value[constants.LEFT\_DATASOURCE][constants.LEFT\_COLUMNS])  
 self.left\_fields = [x.lower() **for** x **in** self.left\_fields]  
 self.right\_fields = json\_utility.required\_key\_value\_from\_json(  
 constants.FIELD\_NAME,  
 json\_value[constants.RIGHT\_DATASOURCE][constants.RIGHT\_COLUMNS])  
 self.right\_fields = [x.lower() **for** x **in** self.right\_fields]  
 self.right\_data\_frame = DataRegistry().create\_data\_frame\_using\_data\_source\_id(  
 json\_value[constants.RIGHT\_DATASOURCE][constants.DATA\_ID],  
 False, user\_id=self.user\_id)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method will perform the join  
 operation with given specified left and right column details* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 data\_frame.rename(columns = **lambda** x: x.lower(),inplace=True)  
 self.right\_data\_frame.rename(columns = **lambda** x: x.lower(),inplace=True)  
 **return** pd.merge(data\_frame, right=self.right\_data\_frame, how=self.join\_type,  
 left\_on=self.left\_fields, right\_on=self.right\_fields)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.COLUMN\_NOT\_FOUND.format(key\_error.args[0]))  
 **except** ValueError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.COLUMNS\_WITH\_DIFFERENT\_DATATYPES)  
 **except** MemoryError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INSUFFICIENT\_MEMORY\_FOR\_PERFORMING\_OPERATION)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** get\_input\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find input col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 json\_utility = JsonUtility()  
 **return** json\_utility.required\_key\_value\_from\_json(constants.FIELD\_NAME,  
 step\_json[constants.LEFT\_DATASOURCE][  
 constants.LEFT\_COLUMNS])  
 **except** Exception:  
 **return** []  
  
 @classmethod  
 **def** get\_output\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find output col name keys for current feature* **:return***: list of keys  
 """* **return** []  
  
 @classmethod  
 **def** update\_step\_for\_col\_map(cls, step\_json, column\_mapping\_list):  
 *"""  
 used to update step and col map list wrt step* **:param** *step\_json:* **:param** *column\_mapping\_list:* **:return***: list of step json and updated col mapping list  
 """* updated\_column\_mapping\_list = column\_mapping\_list  
 **try**:  
 input\_cols\_list = JoinStep.get\_input\_column\_name\_values(step\_json)  
 **for** col\_obj **in** column\_mapping\_list:  
 **if** col\_obj[**"column\_used"**] **in** input\_cols\_list:  
 step\_json.update({constants.COLUMN\_NAME: col\_obj[**"mapped\_with"**]})  
 **break  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** {**"step\_json"**: step\_json, **"column\_mapping\_list"**: updated\_column\_mapping\_list}

*"""  
This module is used to test case for the join step  
"""***import** unittest  
**import** pandas **as** pd  
**from** mock **import** patch  
**from** join.join\_dataframe **import** JoinStep  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestJoinStep(unittest.TestCase):  
 *"""  
 This class is used for implement test method of join step  
 """* **def** setUp(self):  
 self.data\_frame1 = pd.read\_csv(**'../resources/doc1.txt'**, delimiter=**','**, header=0)  
 self.data\_frame2 = pd.read\_csv(**'../resources/doc2.txt'**, delimiter=**','**, header=0)  
 self.json\_value = {  
 **"joinType"**: **"left"**,  
 **"user\_id"**: 10,  
 **"leftDataSource"**: {**"dataId"**: 22,  
 **"dataSourceName"**: **"Scenariosxlsx"**,  
 **"leftColumns"**: [{**"fieldName"**: **"name"**, **"fieldType"**: **"STRING"**},  
 {**"fieldName"**: **"address"**, **"fieldType"**: **"STRING"**}]},  
 **"rightDataSource"**: {**"dataId"**: 2,  
 **"dataSourceName"**: **"taxdatainput"**,  
 **"rightColumns"**: [{**"fieldName"**: **"name"**, **"fieldType"**: **"STRING"**},  
 {**"fieldName"**: **"address"**, **"fieldType"**: **"STRING"**}]}  
 }  
  
 **def** test\_join\_step\_with\_mock(self):  
 *"""  
 This method for performing join operation on data\_frame using right data frame* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 join\_step = JoinStep(self.json\_value)  
 data\_frame1 = join\_step.get\_result(self.data\_frame1)  
 **if** len(data\_frame1) == 3:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_join\_step\_key\_missing(self):  
 *"""  
 This method for performing join operation on data\_frame using right data frame* **:return***:  
 """* **try**:  
 self.json\_value.pop(**"joinType"**)  
 join\_step = JoinStep(self.json\_value)  
 join\_step.get\_result(self.data\_frame1)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_join\_step\_value\_error(self):  
 *"""  
 This method will test the error message if right data\_frame is None and throw the Exception* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = None  
 **try**:  
 join\_step = JoinStep(self.json\_value)  
 join\_step.get\_result(self.data\_frame1)  
 self.assertEqual(1, 1)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_join\_step\_key\_error(self):  
 *"""  
 This method will test the error message if request column is not available in the dataframe* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 **try**:  
 self.json\_value = {**"joinType"**: **"left"**, **"user\_id"**: 10, **"leftDataSource"**: {  
 **"dataId"**: 22, **"dataSourceName"**: **"Scenariosxlsx"**,  
 **"leftColumns"**: [{**"fieldName"**: **"name"**, **"fieldType"**: **"STRING"**},  
 {**"fieldName"**: **"address"**, **"fieldType"**: **"STRING"**}]},  
 **"rightDataSource"**: {**"dataId"**: 2,  
 **"dataSourceName"**: **"taxdatainput"**,  
 **"rightColumns"**: [  
 {**"fieldName"**: **"name"**,  
 **"fieldType"**: **"STRING"**},  
 {**"fieldName"**: **"address1"**,  
 **"fieldType"**: **"STRING"**}]}}  
 join\_step = JoinStep(self.json\_value)  
 join\_step.get\_result(self.data\_frame1)  
 self.assertEqual(1, 1)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_update\_step\_for\_col\_map(self):  
 *"""  
 test case for update step* **:return***:  
 """* json\_value = {  
 **"joinType"**: **"left"**,  
 **"user\_id"**: 10,  
 **"leftDataSource"**: {**"dataId"**: 22,  
 **"dataSourceName"**: **"Scenariosxlsx"**,  
 **"leftColumns"**: [{**"fieldName"**: **"name"**, **"fieldType"**: **"STRING"**},  
 {**"fieldName"**: **"address"**, **"fieldType"**: **"STRING"**}]},  
 **"rightDataSource"**: {**"dataId"**: 2,  
 **"dataSourceName"**: **"taxdatainput"**,  
 **"rightColumns"**: [{**"fieldName"**: **"name"**, **"fieldType"**: **"STRING"**},  
 {**"fieldName"**: **"address"**, **"fieldType"**: **"STRING"**}]}  
 }  
  
 column\_mapping\_list = [{**'column\_used'**: **'name'**, **'mapped\_with'**: **'innings\_new'**}]  
 output\_json = JoinStep.update\_step\_for\_col\_map(json\_value, column\_mapping\_list)  
 self.assertEqual(column\_mapping\_list, output\_json[**'column\_mapping\_list'**])  
  
 **def** test\_update\_step\_for\_col\_map\_error(self):  
 *"""  
 test case for update step negative* **:return***:  
 """* self.assertEqual(JoinStep.update\_step\_for\_col\_map(None, None),  
 {**'column\_mapping\_list'**: None, **'step\_json'**: None})  
  
 **def** test\_get\_output\_column\_name\_values(self):  
 *"""  
 test case output\_column\_name\_values* **:return***:  
 """* self.assertEqual(JoinStep.get\_output\_column\_name\_values(self.json\_value), [])  
  
  
 **def** test\_join\_dtype\_missmatch(self):  
 *"""  
 This method for performing join operation on data\_frame using right data frame* **:return***:  
 """* **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_register:  
 data\_frame\_register.return\_value = self.data\_frame2  
 json\_value = {  
 **"joinType"**: **"left"**,  
 **"user\_id"**: 10,  
 **"leftDataSource"**: {**"dataId"**: 22,  
 **"dataSourceName"**: **"Scenariosxlsx"**,  
 **"leftColumns"**: [  
 {**"fieldName"**: **"empId"**,  
 **"fieldType"**: **"integer"**}]},  
 **"rightDataSource"**: {**"dataId"**: 2,  
 **"dataSourceName"**: **"taxdatainput"**,  
 **"rightColumns"**: [{**"fieldName"**: **"name"**, **"fieldType"**: **"STRING"**}]}  
 }  
  
 join\_step = JoinStep(json\_value)  
 **with** self.assertRaises(Exception):  
 join\_step.get\_result(self.data\_frame1)

*Profiling*

*"""  
This module is used for generating profile on selected column from user input.  
"""***import** logging  
**import** pandas\_profiling  
**import** pandas **as** pd  
**import** numpy **as** np  
**import** constants  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** request\_builder **import** RequestBuilder  
**from** token\_registry **import** TokenRegistry  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** GenerateProfile: *#pylint: disable-msg=too-many-instance-attributes  
 """  
 This class will generate profile based on the selected datatypes.  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.data\_source\_id = input\_json[constants.DATA\_SOURCE\_ID]  
 self.lens\_id = input\_json[constants.LENS\_ID]  
 self.column\_name = input\_json[constants.COLUMN\_NAME]  
 self.cluster\_enable = input\_json[constants.CLUSTERING\_ENABLE] \  
 **if "Clustering\_enable" in** input\_json **else "WORKING"** self.user\_id = input\_json[constants.USER\_ID]  
 self.knowledgebase\_response = []  
 **"""  
 #Removed as Knowledgebas is not supported yet  
 RequestBuilder().call\_http\_get\_request(  
 constants.FETCH\_KNOWLEDGE\_BASE\_USING\_LENS + str(self.lens\_id) +  
 "&dataSourceId=" + str(self.data\_source\_id), {},  
 TokenRegistry.get\_token\_for\_user(user\_id=self.user\_id)).json()  
 """** self.categorical\_flag = False  
 self.threshold\_count = constants.FREQ\_THRESHOLD\_COUNT  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** generate\_profile\_of\_request\_column(self): *# pylint: disable-msg=too-many-branches,  
 # pylint: disable-msg=too-many-locals  
 """  
 This method will generate profiling for selected column from user.* **:return***:  
 """* **try**:  
 list\_categorical\_col = []  
 list\_non\_categorical\_col = []  
 **if** self.knowledgebase\_response:  
 **for** kb\_val **in** self.knowledgebase\_response:  
 **if** kb\_val[**'categoricalIndicator'**] **and** \  
 kb\_val[**'categoricalIndicator'**].lower() == **'y'**:  
 list\_categorical\_col.append(kb\_val[**'columnName'**])  
 **elif** kb\_val[**'categoricalIndicator'**] **and** \  
 kb\_val[**'categoricalIndicator'**].lower() == **'n'**:  
 list\_non\_categorical\_col.append(kb\_val[**'columnName'**])  
 numerics = [**'int16'**, **'int32'**, **'int64'**, **'float16'**, **'float32'**, **'float64'**]  
 data\_frame = DataRegistry.get\_data\_set\_from\_dict(self.data\_source\_id,  
 self.lens\_id, self.cluster\_enable)  
 df2 = pd.DataFrame(data\_frame[self.column\_name])  
 **if** self.column\_name **in** list\_categorical\_col **and** df2[self.column\_name].dtype **in** numerics:  
 df2[self.column\_name].astype(**'str'**)  
 self.categorical\_flag = True  
 df2 = df2.reset\_index(drop=True)  
 value = pandas\_profiling.describe(df2)  
 json = {}  
 **for** key **in** value[constants.FREQ].keys():  
 value\_freq = {}  
 **for** series\_key, series\_value **in** value[constants.FREQ][key].items():  
 self.threshold\_count = self.threshold\_count - 1  
 **if** self.threshold\_count < 0:  
 **if** constants.FREQ\_THRESHOLD\_VARIABLE **in** value\_freq:  
 value\_freq.update(  
 {constants.FREQ\_THRESHOLD\_VARIABLE:  
 str(int(value\_freq[constants.FREQ\_THRESHOLD\_VARIABLE]) +  
 int(series\_value))  
 }  
 )  
 **else**:  
 value\_freq.update({constants.FREQ\_THRESHOLD\_VARIABLE:  
 str(series\_value)})  
 **else**:  
 value\_freq.update({str(series\_key): str(series\_value)})  
 json.update({constants.FREQUENCY: {key: value\_freq}})  
 **for** key **in** value[constants.PROFILING\_VARIABLE].keys():  
 json.update({  
 key: str(value[constants.PROFILING\_VARIABLE][key][self.column\_name]).lower()  
 })  
 value = json.get(constants.HISTOGRAM)  
 **if** value **is not** None:  
 json.pop(constants.HISTOGRAM)  
 *# hard code update* **if** self.categorical\_flag:  
 json.update({**'type'**: **'cat'**})  
 **elif** self.column\_name **in** list\_non\_categorical\_col:  
 json.update({**'type'**: **'non\_cat'**})  
 *# for boolean check* **if 'distinct\_count' in** json **and** int(json[**'distinct\_count'**]) == 2:  
 json.update({**"col\_boolean"**: **"True"**})  
 **else**:  
 json.update({**"col\_boolean"**: **"False"**})  
 **return** json  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** generate\_profile\_dataframe(self, column\_type):  
 *"""  
 This method will generate profiling for full data frame.* **:return***:  
 """* **try**:  
 categorical\_col\_list = []  
 id\_col\_list = []  
 data\_frame = DataRegistry.get\_data\_set\_from\_dict(self.data\_source\_id,  
 self.lens\_id, self.cluster\_enable)  
 **if** data\_frame.shape[0] == 0:  
 **return** None  
 data\_frame.replace(**'^\s+$'**, **""**, inplace=True, regex=True)  
 data\_frame.replace(**''**, np.NaN, inplace=True)  
 data\_frame.columns = map(str.lower, data\_frame.columns)  
 **for** value **in** self.knowledgebase\_response:  
 **if** value[constants.CATEGORICAL\_INDICATOR] == **'Y' or** \  
 value[constants.BOOLEAN\_INDICATOR] == **'Y'**:  
 categorical\_col\_list.append(value[constants.COLUMN\_NAME])  
 **if** value[constants.KB\_INDEX] == **'Y'**:  
 id\_col\_list.append(value[constants.COLUMN\_NAME])  
 **for** \_, value **in** enumerate(categorical\_col\_list):  
 **if** value **in** list(data\_frame.columns):  
 data\_frame[value] = data\_frame[value].astype(**'category'**)  
 df\_summary = data\_frame.describe(include=**"all"**)  
 df\_summary.loc[**'dtype'**] = data\_frame.dtypes  
 df\_summary.loc[**'null%'**] = round((data\_frame.isnull().sum() / len(data\_frame)) \* 100, 2)  
 df\_summary.loc[**'is\_id'**] = data\_frame.columns.isin(id\_col\_list)  
 LOGGER.error(**"==df\_summary==>%s"**, df\_summary)  
 *# Added below for loop to make a empty column dtype as string to match column level  
 # and table level summary* **for** i **in** df\_summary.columns:  
 **if** int(df\_summary.loc[[**'null%'**], i].values) == 100:  
 df\_summary.loc[[**'dtype'**], i] = **'object'** *# Add suggestions to drop or impute columns* **return** self.imputation\_suggestion(data\_frame.columns, df\_summary, column\_type)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** df\_process(df\_summary, column, index\_type):  
 *"""This method fetch value of particular index from df\_summary"""* **try**:  
 value = df\_summary.loc[[index\_type], [column]][column].to\_string(index=False)  
 **return** value  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** imputation\_suggestion(self, column\_list, df\_summary, column\_type):  
 *"""This method is design to suggest impute, drop to user for all columns of dataset"""* **try**:  
 return\_list = []  
 return\_dict = {}  
 **for** \_, column **in** enumerate(column\_list):  
 **if** (self.df\_process(df\_summary, column, **'dtype'**) **in** [**'int16'**, **'int32'**, **'int64'**, **'float16'**, **'float32'**, **'float64'**]) \  
 **and** (column\_type **in** [**"numeric"**, **"all"**]):  
 return\_dict = {**"column"**: column,  
 **"column\_type"**: **"numeric"**,  
 **"column\_sub\_type"**: self.df\_process(df\_summary, column, **'dtype'**),  
 **"column\_properties"**: self.numeric\_properties(df\_summary, column)}  
 **if** float(self.df\_process(df\_summary, column, **'null%'**)) > 30:  
 return\_dict.update({**"suggestions"**: self.drop\_suggestion(column,  
 df\_summary)})  
 **if** (float(self.df\_process(df\_summary, column, **'null%'**)) <= 5) **and** \  
 (float(self.df\_process(df\_summary, column, **'null%'**)) > 0):  
 return\_dict.update({**"suggestions"**: self.impute\_suggestion(column,  
 df\_summary,  
 constants.KNN)})  
 **elif** (float(self.df\_process(df\_summary, column, **'null%'**)) <= 30) **and** \  
 (float(self.df\_process(df\_summary, column, **'null%'**)) > 0):  
 return\_dict.update({**"suggestions"**: self.impute\_suggestion(column,  
 df\_summary)})  
 return\_list.append(return\_dict)  
 *# code to identify categorical columns and filter and suggestion properties* **elif** (self.df\_process(df\_summary, column, **'dtype'**) **in** [**'object'**, **'bool'**,  
 **'category'**]) \  
 **and** (column\_type **in** [**"categorical"**, **"all"**]):  
 return\_dict = {**"column"**: column,  
 **"column\_type"**: **"categorical"**,  
 **"column\_sub\_type"**: self.df\_process(df\_summary, column, **'dtype'**),  
 **"column\_properties"**: self.categorical\_properties(df\_summary,  
 column)}  
 **if** float(self.df\_process(df\_summary, column, **'null%'**)) > 30:  
 return\_dict.update({**"suggestions"**: self.drop\_suggestion(column,  
 df\_summary)})  
 **elif** (float(self.df\_process(df\_summary, column, **'null%'**)) <= 30) **and** \  
 (float(self.df\_process(df\_summary, column, **'null%'**)) > 0):  
 return\_dict.update({**"suggestions"**: self.impute\_suggestion(column,  
 df\_summary,  
 constants.MODE)})  
 return\_list.append(return\_dict)  
 *# code to identify date columns and filter and suggestion properties* **elif** (self.df\_process(df\_summary, column, **'dtype'**) == **'datetime64[ns]'**) **and** \  
 (column\_type **in** [**"date"**, **"all"**]):  
 return\_dict = {**"column"**: column,  
 **"column\_type"**: **"date"**,  
 **"column\_sub\_type"**: self.df\_process(df\_summary, column, **'dtype'**),  
 **"column\_properties"**: self.date\_properties(df\_summary, column)}  
 **if** float(self.df\_process(df\_summary, column, **'null%'**)) > 30:  
 return\_dict.update({**"suggestions"**: self.drop\_suggestion(column,  
 df\_summary)})  
 **elif** (float(self.df\_process(df\_summary, column, **'null%'**)) <= 30) **and** \  
 (float(self.df\_process(df\_summary, column, **'null%'**)) > 0):  
 return\_dict.update({**"suggestions"**: self.impute\_suggestion(column,  
 df\_summary,  
 constants.MODE)})  
 return\_list.append(return\_dict)  
  
 **return** return\_list  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** numeric\_properties(df\_summary, column):  
 *"""This modules prepare all properties for numeric columns"""* **try**:  
 **return** {**"count"**: df\_summary.loc[[**'count'**], [column]][column].to\_string(index=False),  
 **"min"**: df\_summary.loc[[**'min'**], [column]][column].to\_string(index=False),  
 **"max"**: df\_summary.loc[[**'max'**], [column]][column].to\_string(index=False),  
 **"mean"**: df\_summary.loc[[**'mean'**], [column]][column].to\_string(index=False),  
 **"25%"**: df\_summary.loc[[**'25%'**], [column]][column].to\_string(index=False),  
 **"50%"**: df\_summary.loc[[**'50%'**], [column]][column].to\_string(index=False),  
 **"75%"**: df\_summary.loc[[**'75%'**], [column]][column].to\_string(index=False),  
 **"std"**: df\_summary.loc[[**'std'**], [column]][column].to\_string(index=False),  
 **"null%"**: df\_summary.loc[[**'null%'**], [column]][column].to\_string(index=False)}  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** categorical\_properties(df\_summary, column):  
 *"""This modules prepare all properties for categorical type of columns"""* **try**:  
 **return** {**"count"**: df\_summary.loc[[**'count'**], [column]][column].to\_string(index=False),  
 **"unique\_values"**: df\_summary.loc[[**'unique'**], [column]][column].to\_string(  
 index=False),  
 **"most\_common"**: df\_summary.loc[[**'top'**], [column]][column].to\_string(index=False),  
 **"freq"**: df\_summary.loc[[**'freq'**], [column]][column].to\_string(index=False)}  
 **except** KeyError:  
 **return** {**"count"**: df\_summary.loc[[**'count'**], [column]][column].to\_string(index=False),  
 **"unique\_values"**: None, **"most\_common"**: None, **"freq"**: None}  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** date\_properties(df\_summary, column):  
 *"""This modules prepare all properties for date type of columns"""* **try**:  
 **return** {**"count"**: df\_summary.loc[[**'count'**], [column]][column].to\_string(index=False),  
 **"unique\_values\_date"**: df\_summary.loc[[**'unique'**],  
 [column]][column].to\_string(index=False),  
 **"most\_common\_date"**: df\_summary.loc[[**'top'**],  
 [column]][column].to\_string(index=False),  
 **"freq\_date"**: df\_summary.loc[[**'freq'**], [column]][column].to\_string(index=False),  
 **"min\_date"**: df\_summary.loc[[**'first'**], [column]][column].to\_string(index=False),  
 **"max\_date"**: df\_summary.loc[[**'last'**], [column]][column].to\_string(index=False)}  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** drop\_suggestion(column, df\_summary):  
 *"""This method is used to generate text for drop suggestions and also creates  
 a json to perform impute"""* **try**:  
 **return** [{**"title"**: **"Drop"**, **"description"**: **"Drop "** + str(column) + **" column"**,  
 **"reason"**: constants.COLUMN\_GT\_30 +  
 df\_summary.loc[[**'null%'**], [column]][column].to\_string(index=False) +  
 **"% null values"**,  
 **"apply\_step\_json"**: {constants.STEP\_NAME: constants.DROP\_STEP,  
 constants.COLUMN\_NAME: column,  
 constants.IMPUTATION\_TYPE: constants.REMOVE\_ROWS,  
 constants.IMPUTATION\_METHOD: **""**,  
 constants.DEFAULT\_VALUE: **""**,  
 constants.DISPLAY\_DATA\_HEADER: **"Drop"**,  
 constants.DISPLAY\_DATA\_TEXT: **"Drop "** + str(column) +  
 **" column"**,  
 constants.EDIT\_COL\_NAME: column,  
 constants.IS\_SUGGESTED: **"True"**},  
 **"is\_hidden"**: **"False"**}]  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** impute\_suggestion(column, df\_summary, imputation\_method=**"median"**):  
 *"""This method is used to generate impute/ reason text and also creates a  
 json to perform impute"""* **try**:  
 **return** [{**"title"**: **"Impute"**, **"description"**: **"Impute "** + str(column) + **" using "** +  
 imputation\_method,  
 **"reason"**: **" Column is having "** +  
 df\_summary.loc[[**'null%'**], [column]][column].to\_string(index=False) +  
 **"% null values, so can be imputed using "** + imputation\_method  
 + **" value"**,  
 **"apply\_step\_json"**: {constants.STEP\_NAME: constants.IMPUTE\_STEP,  
 constants.COLUMN\_NAME: column,  
 constants.IMPUTATION\_TYPE: constants.REPLACE\_WITH,  
 constants.IMPUTATION\_METHOD: imputation\_method,  
 constants.DEFAULT\_VALUE: **""**,  
 constants.DISPLAY\_DATA\_HEADER: **"Impute"**,  
 constants.DISPLAY\_DATA\_TEXT: **"Impute "** + str(column) +  
 **" using "** + imputation\_method,  
 constants.EDIT\_COL\_NAME: {**"id"**: column, **"name"**: column,  
 **"type"**: **""**},  
 constants.IS\_SUGGESTED: **"True"**},  
 **"is\_hidden"**: **"False"**}]  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to test profiling of data based on column.  
"""***import** unittest  
**from** unittest.mock **import** patch  
**import** pandas **as** pd  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** profiling.generate\_profiling **import** GenerateProfile  
**from** request\_builder **import** RequestBuilder  
**import** constants  
  
  
**class** TestProfile(unittest.TestCase):  
 *"""  
 This class contains all the profiling method  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/deliveries.csv"**, header=0, delimiter=**','**)  
 json = {**"ORIGINAL"**: self.data\_frame, **"WORKING"**: self.data\_frame}  
 DataRegistry.pandas\_df\_map.update({**"10\_50"**: json})  
 self.input\_json = {**"data\_source\_id"**: 10, **"lens\_id"**: 50, **"columnName"**: **"ball"**,  
 **"user\_id"**: **"test"**}  
 self.knowledgebase\_response = [{**'timeSeriesDateAttribute'**: **'N'**, **'normalyViewedWithData'**: {},  
 **'createdAt'**: 1548330396000, **'updatedAt'**: 1548330414000,  
 **'normalyViewedWith'**: **'{}'**, **'index'**: **'N'**, **'lensId'**: -1,  
 **'defaultFrequency'**: **'None'**, **'dataSourceId'**: 616, **'id'**: 160,  
 **'timeSeriesKpi'**: **'N'**, **'dataSourceName'**: **'profile\_test'**,  
 **'sentiment'**: **'Good'**, **'columnName'**: **'ball'**,  
 **'booleanIndicator'**: None, **'target'**: **'Y'**,  
 **'categoricalIndicator'**: **'Y'**, **'relationship'**: [],  
 **'deleteFlag'**: 1}]  
  
 **def** test\_profile\_report(self):  
 *"""  
 Test suite for generate profiling* **:return***:  
 """* **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** knowledgebase\_get\_request:  
 knowledgebase\_get\_request.return\_value.json.return\_value = self.knowledgebase\_response  
 profile = GenerateProfile(self.input\_json)  
 value = profile.generate\_profile\_of\_request\_column()  
 **if** value **is** None:  
 self.assertEqual(1, 2)  
 **else**:  
 self.assertEqual(1, 1)  
  
 **def** test\_profile\_report\_for\_key\_error(self):  
 *"""  
 Test suite for generate profiling keyError* **:return***:  
 """* **try**:  
 **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** knowledgebase\_get\_request:  
 knowledgebase\_get\_request.return\_value.json.return\_value = \  
 self.knowledgebase\_response  
 self.input\_json[**'columnName'**] = **"SerialNumber"** profile = GenerateProfile(self.input\_json)  
 profile.generate\_profile\_of\_request\_column()  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_profile\_report\_key\_missing(self):  
 *"""  
 Test suite for generate profiling keyError with key missing in input json* **:return***:  
 """* **try**:  
 **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** knowledgebase\_get\_request:  
 knowledgebase\_get\_request.return\_value.json.return\_value = \  
 self.knowledgebase\_response  
 self.input\_json.pop(**'data\_source\_id'**)  
 GenerateProfile(self.input\_json)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_generate\_profile\_dataframe(self):  
 *"""  
 Test suite for generate profiling of full data frame* **:return***:  
 """* **try**:  
 **with** patch.object(RequestBuilder, **"call\_http\_get\_request"**) **as** knowledgebase\_get\_request:  
 knowledgebase\_get\_request.return\_value.json.return\_value = \  
 self.knowledgebase\_response  
 input\_json = {**"data\_source\_id"**: 10, **"lens\_id"**: 50, constants.COLUMN\_TYPE: **"all"**,  
 **"user\_id"**: 10,  
 **"columnName"**: **""**}  
 profile\_obj = GenerateProfile(input\_json)  
 output\_list = profile\_obj.generate\_profile\_dataframe(  
 input\_json[constants.COLUMN\_TYPE])  
 self.assertEqual(len(output\_list), 21)  
 *# print(output\_list)* **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 2)  
  
 **def** test\_static\_func\_error(self):  
 **with** self.assertRaises(Exception):  
 GenerateProfile.drop\_suggestion(self.data\_frame)  
  
 **with** self.assertRaises(Exception):  
 GenerateProfile.impute\_suggestion(None, None)

*Rule\_as\_step*

*"""  
This module is used for apply Rule on data\_frame  
"""***import** logging  
**import** json  
**import** constants  
**import** message\_constants  
**from** request\_builder **import** RequestBuilder  
**from** pandas\_api\_rest **import** PandasApi  
**from** token\_registry **import** TokenRegistry  
**from** class\_creator **import** ClassCreator  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** RuleStep:  
 *"""  
 this class is used for apply step which contains multiple step on dataframe  
 """* **def** \_\_init\_\_(self, json\_value):  
 **try**:  
 self.rule\_id = json\_value[constants.RULE\_ID]  
 self.params = json\_value[constants.PARAMS]  
 self.user\_id = json\_value[constants.USER\_ID]  
 self.rule\_details = RequestBuilder().call\_http\_get\_request(  
 constants.RULE\_FETCH\_URL + str(self.rule\_id), {},  
 TokenRegistry.get\_token\_for\_user(self.user\_id)).text  
 self.column\_mapping = json\_value.get(**"column\_mapping"**, [])  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method is used apply rule of steps on the dataframe* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 steps\_json = json.loads(json.loads(self.rule\_details)[constants.RULE\_STEPS])  
 column\_mapping\_list = self.column\_mapping  
 **for** step **in** steps\_json:  
 **try**:  
 cls\_instance = ClassCreator.get\_class(step[constants.STEP\_NAME])  
 response\_obj = cls\_instance.update\_step\_for\_col\_map(step, column\_mapping\_list)  
 step = response\_obj[**"step\_json"**]  
 column\_mapping\_list = response\_obj[**"column\_mapping\_list"**]  
 **except** Exception **as** ex:  
 **print**(**"Step Name does not work => "**, step[constants.STEP\_NAME])  
 LOGGER.error(**'Exception: %s'**, ex)  
 **pass** step.update({**"user\_id"**: self.user\_id})  
 data\_frame = PandasApi.apply\_individual\_step(data\_frame=data\_frame, step=step)  
 **return** data\_frame  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
**class** RuleStepDetails:  
 *"""  
 class contains method to get rule step details based on rule\_id  
 """* **def** \_\_init\_\_(self, json\_value):  
 *"""  
 initialize* **:param** *json\_value:  
 """* **try**:  
 self.rule\_id = json\_value[constants.RULE\_ID]  
 self.user\_id = json\_value[constants.USER\_ID]  
 self.data\_source\_id = json\_value[constants.DATA\_SOURCE\_ID]  
 self.lens\_id = json\_value[constants.LENS\_ID]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_rule\_details(self):  
 *"""  
 method calls java service to get rule details* **:return***:  
 """* **try**:  
 rule\_mapping\_dict = {**"rule\_id"**: self.rule\_id,  
 **"column\_mapping"**: []}  
 column\_mapping\_list = []  
 *# get rules from api* self.rule\_details = RequestBuilder().call\_http\_get\_request(  
 constants.RULE\_FETCH\_URL + str(self.rule\_id), {},  
 TokenRegistry.get\_token\_for\_user(self.user\_id)).text  
 *# get dataframe from memory* data\_frame = DataRegistry.get\_data\_set\_from\_dict(self.data\_source\_id, self.lens\_id,  
 **'WORKING'**)  
 data\_frame\_cols\_list = data\_frame.columns.tolist()  
 input\_cols\_list = []  
 output\_cols\_list = []  
 steps\_json = json.loads(json.loads(self.rule\_details)[constants.RULE\_STEPS])  
 *# process Start* **for** step **in** steps\_json:  
 cls\_instance = ClassCreator.get\_class(step[constants.STEP\_NAME])  
 input\_cols\_list.extend(cls\_instance.get\_input\_column\_name\_values(step))  
 output\_cols\_list.extend(cls\_instance.get\_output\_column\_name\_values(step))  
 input\_cols\_list = list(set(input\_cols\_list))  
 output\_cols\_list = list(set(output\_cols\_list))  
 cols\_for\_mapping = [col **for** col **in** input\_cols\_list **if** col **not in** output\_cols\_list]  
 **for** col **in** cols\_for\_mapping:  
 **if** col **in** data\_frame\_cols\_list:  
 column\_mapping\_list.append({**"column\_used"**: col, **"mapped\_with"**: col})  
 **else**:  
 column\_mapping\_list.append({**"column\_used"**: col, **"mapped\_with"**: **""**})  
 rule\_mapping\_dict.update({**"column\_mapping"**: column\_mapping\_list})  
 *# process End* **return** rule\_mapping\_dict  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to test Rule as step  
"""***import** unittest  
**import** json  
**import** pandas **as** pd  
**import** requests  
**from** mock **import** Mock, patch  
**from** rule\_as\_step.rule\_step **import** RuleStep, RuleStepDetails  
**from** request\_builder **import** RequestBuilder  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestRuleStep(unittest.TestCase):  
 *"""  
 This class is used to test Rule as step  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**'../resources/10Records.txt'**, delimiter=**'\t'**, header=0)  
 self.rule\_step\_json = {**"step\_name"**: **"rule\_as\_step.rule\_step.RuleStep"**, **"rule\_id"**: 22,  
 **"parameters"**: [],  
 **"user\_id"**: 10}  
 self.rule\_json = {**"ruleName"**: **"ruletestf1234 sds"**, **"ruleDescription"**: **"abcdsda"**,  
 **"ruleDeleteFlag"**: 1,  
 **"ruleSteps"**: **"[{\"displayDataHeader\":\"Filter\","  
 "\"displayDataText\":"  
 "\" WHEN ( field0 equals 2017-10-15 ) \","  
 "\"is\_edited\":false,\"step\_name\":"  
 "\"filter.filter\_condition.FilterCondition\","  
 "\"exclude\":false,\"conditions\""  
 ":[{\"start\_bracket\":\"(\",\"func\":"  
 "\"when\",\"column\_type\":\"string\","  
 "\"columnName\":\"field0\",\"column\_id\":\"field0\","  
 "\"operator\":\"equals\""  
 ",\"is\_nested\":false,\"children\":[],"  
 "\"uID\":1,\"case\":\"false\","  
 "\"is\_saved\":true,\"close\_bracket\":\")\","  
 "\"value\":\"2017-10-15\","  
 "\"is\_excluded\":false}],\"htmlForConditions\":"  
 "[\"<span class=\\\"badge badge-outline\\\">WHEN</span><br>"  
 "<span class=\\\"bracket\\\">"  
 "(</span><span class=\\\"badge badge-light\\\">"  
 "field0</span> <span "  
 "class=\\\"badge bor-rid\\\">equals</span>"  
 "<span class=\\\"badge badge-light\\\">2017-10-15</span>"  
 "<span class=\\\"bracket\\\">)</span>\"],"  
 "\"expressionForConditions\":["  
 "\" WHEN ( field0 equals 2017-10-15 ) \"]},"  
 "{\"step\_name\":"  
 "\"transformation.rename\_column.RenameColumn\","  
 "\"columnName\""  
 ":\"field0\",\"toColumn\":\"date\_values\","  
 "\"displayDataHeader\":\"Rename\","  
 "\"displayDataText\":\"field0 to date\_values\","  
 "\"is\_edited\":false}]"**,  
 **"category"**: **"a"**, **"subCategory"**: **"B"**, **"defaultDatasurceId"**: 10,  
 **"parametersValue"**: **"ABC"**, **"cretaId"**: 121, **"updatedId"**: 243}  
  
 **def** test\_rule\_as\_step(self):  
 *"""  
 This method will test all the code RuleStep* **:return***:  
 """* rule\_step = RuleStep(self.rule\_step\_json)  
 data\_frame = rule\_step.get\_result(data\_frame=self.data\_frame)  
 **if** data\_frame **is not** None:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_rule\_step\_using\_mock(self):  
 *"""  
 This method will test all the code of Rule Step using mock Object* **:return***:  
 """* self.rule\_step\_json = {**"step\_name"**: **"rule\_as\_step.rule\_step.RuleStep"**, **"rule\_id"**: 3,  
 **"parameters"**: [], **"user\_id"**: 10}  
 **with** patch.object(requests, **"get"**) **as** get\_request:  
 get\_request.return\_value = mock\_response = Mock()  
 mock\_response.text = json.dumps(self.rule\_json)  
 rule\_step = RuleStep(self.rule\_step\_json)  
 data\_frame = rule\_step.get\_result(data\_frame=self.data\_frame)  
 **if** len(data\_frame) == 3:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_rule\_step\_as\_mock\_key\_error(self):  
 *"""  
 This method will test all the code of Rule Step using mock Object* **:return***:  
 """* **try**:  
 self.rule\_step\_json = {**"step\_name"**: **"rule\_as\_step.rule\_step.RuleStep"**,  
 **"rule\_id\_name"**: 3,  
 **"parameters"**: [], **"user\_id"**: 10}  
 run\_as\_step = RuleStep(self.rule\_step\_json)  
 run\_as\_step.get\_result(data\_frame=self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_rule\_step\_as\_mock\_with\_exception(self):  
 *"""  
 This method will test all the code of Rule Step using mock Object* **:return***:  
 """* self.rule\_step\_json = {**"step\_name"**: **"rule\_as\_step.rule\_step.RuleStep"**,  
 **"rule\_id"**: 3, **"parameters"**: [],  
 **"user\_id"**: 10}  
 **with** patch.object(requests, **"get"**) **as** get\_request:  
 get\_request.return\_value = mock\_response = Mock()  
 mock\_response.text = json.dumps(self.rule\_json)  
 **try**:  
 rule\_step = RuleStep(self.rule\_step\_json)  
 rule\_step.get\_result(data\_frame=self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args)  
 self.assertEqual(1, 1)  
  
 **def** test\_get\_rule\_details(self):  
 *"""  
 test get\_rule\_details method* **:return***:  
 """* **with** self.assertRaises(Exception):  
 input\_json = {**"data\_source\_id"**: 10, **"lens\_id"**: 50, **"rule\_id"**: 3, **"user\_id"**: 10}  
 rule\_step\_details = RuleStepDetails(input\_json)  
 value = rule\_step\_details.get\_rule\_details()  
 **print**(**"process"**)  
  
 **with** self.assertRaises(Exception):  
 RuleStepDetails({})

*Running\_total*

*"""  
@author: Aaditya Agrawal  
This script is used to get running total aggregate for a column  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** pandas **as** pd  
**import** numpy **as** np  
**import** constants  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** RunningTotal:  
 *"""  
 Class for running total to get dataframe with aggregate value for a column  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 Initialize class with input values* **:param** *input\_json: dict target as list and n as columns  
 """* **try**:  
 self.partition\_col = input\_json[constants.PARTITION\_COLUMN]  
 self.sort\_col = input\_json[constants.SORT\_COLUMN]  
 self.aggregate\_col = input\_json[constants.AGGREGATE\_COLUMN]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, dataframe):  
 *"""* **:param** *dataframe: input dataframe to perform operation on* **:return***: dataframe  
 """* **try**:  
 **try**:  
 numerics = [**'int16'**, **'int32'**, **'int64'**, **'float16'**, **'float32'**, **'float64'**]  
 **if** dataframe[self.aggregate\_col].dtype **not in** numerics:  
 dataframe[self.aggregate\_col].astype(str).str.replace(**' '**, **''**)  
 dataframe.loc[  
 dataframe[self.aggregate\_col] == **''**, self.aggregate\_col] = np.NaN  
 dataframe[self.aggregate\_col].fillna(0, inplace=True)  
 dataframe[self.aggregate\_col] = dataframe[self.aggregate\_col].astype(float)  
 **except** Exception **as** ex:  
 **raise** Exception(message\_constants.SELECT\_NUMERIC\_COLUMN)  
 dataframe = dataframe.sort\_values([self.partition\_col, self.sort\_col])  
 dataframe[**'Agg\_'** + self.aggregate\_col] = \  
 dataframe.groupby([self.partition\_col, self.sort\_col])[self.aggregate\_col].transform(  
 pd.Series.cumsum)  
 dataframe.reset\_index(drop=True, inplace=True)  
 **return** dataframe  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
@author: Aaditya Agrawal  
This python file is contains test case for feature\_selectors  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** running\_total **import** RunningTotal  
  
**class** TestRunningTotal(TestCase):  
 *"""  
 This class is used to write test cases for RunningTotal  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/test\_running\_total.csv"**)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_get\_result(self):  
 *"""  
 This function is used to write the test cases for RunningTotal* **:return***: data\_frame  
 """* json\_request = {  
 **"partition\_column"**: **"Semester"**,  
 **"sort\_column"**: **"Rollnumber"**,  
 **"aggregate\_column"**: **"Marks"** }  
 running\_total = RunningTotal(json\_request)  
 result\_df = running\_total.get\_result(self.data\_frame)  
 self.assertEqual(result\_df[**'Agg\_'**+json\_request[**'aggregate\_column'**]][1], 110)  
  
 **def** test\_exception(self):  
 *"""  
 Negative test case  
 Return : exception  
 """* **try**:  
 json\_request = {  
 **"partition\_column"**: **"Semester"**,  
 **"sort\_column"**: **"Rollnumber"**,  
 *# "aggregate\_column": "Marks"* }  
 running\_total = RunningTotal(json\_request)  
 running\_total.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)

*Save\_user\_config*

*Common*

**class** Singleton(type):  
 \_instances = {}  
  
 **def** \_\_call\_\_(cls, \*args, \*\*kwargs):  
 **if** cls **not in** cls.\_instances:  
 cls.\_instances[cls] = super(Singleton, cls).\_\_call\_\_(\*args, \*\*kwargs)  
 **return** cls.\_instances[cls]

**import** cx\_Oracle  
**import** message\_constants  
**import** logging  
**import** datetime  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
**from** save\_user\_config.common.singleton **import** Singleton  
  
**import** constants  
  
  
**class** UserPreferences(metaclass=Singleton):  
 *"""  
 UserPreferences is the controller to manage the user preferences to perform Insert  
 update operations on oracle database.  
 """* **def** \_\_init\_\_(self, config=None,user\_name=None):  
 *"""  
 Initiliaze the UserPreferences variables.* **:param** *config: get user preferences like Write Mode,File format and name.* **:param** *username: logged in user  
 """* **try**:  
 **if** config **is not** None:  
 self.lens\_id = config[constants.CONN\_LENS\_ID]  
 self.format = config[constants.CONN\_FORMAT]  
 self.mode = config[constants.CONN\_MODE]  
 self.name = config[constants.CONN\_NAME]  
 self.user\_name = user\_name  
 self.db\_user = constants.DB\_USER  
 self.db\_dsn = constants.DB\_DSN  
 self.db\_password = constants.DB\_PASSWORD  
 self.db\_schema = constants.DB\_SCHEMA  
 self.updated\_date = datetime.datetime.now().strftime(constants.DATE\_TIME\_FORMAT)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception while initialise the UserPreferences : %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** \_\_connection(self):  
 *"""  
 return the connection.* **:return***: connection.  
 """* **return** self.get\_connection()  
  
 **def** get\_connection(self):  
 *"""  
 Establish the connection to oracle database.* **:return***: connection.  
 """* **try**:  
 user = self.db\_user  
 db\_schema = self.db\_schema  
 password = self.db\_password  
 dsn = self.db\_dsn  
  
 LOGGER.info(**"Connect to database for user %s@%s\n"**, dsn, user)  
 connection = cx\_Oracle.connect(  
 user=user,  
 password=password,  
 dsn=dsn,  
 cclass=**"runPython"**,  
 threaded=True,  
 purity=cx\_Oracle.ATTR\_PURITY\_SELF,  
 )  
 **return** connection  
 **except** cx\_Oracle.DatabaseError **as** ce:  
 LOGGER.exception(**"Exception while getting connection from oracle database : %s"**,ce)  
 **raise** Exception(message\_constants.CONNECTION\_ERROR)  
  
 **def** insert\_or\_update\_user\_perferences(self,pref\_format,pref\_mode,name,updated\_date,lens\_id,user\_id):  
 *"""  
 If user\_id and lens\_id existing in table then merge operation executed else Insert operation executed.* **:param** *id: ID.* **:param** *pref\_format: format like csv,text, parqueat from save preferences.* **:param** *mode: overwrite or append mode.* **:param** *name: Name.* **:param** *updated\_date: updated date.* **:param** *lens\_id: lens id.* **:param** *user\_id: logged in user.* **:return***:  
 """* **try**:  
  
 user = self.db\_user  
 db\_schema = self.db\_schema  
 password = self.db\_password  
 dsn = self.db\_dsn  
 updated\_format = pref\_format  
 updated\_mode = pref\_mode  
 updated\_name = name  
 updated\_lens\_id = lens\_id  
 updated\_user\_id = user\_id  
 connection = self.get\_connection()  
 sql = (  
 **"MERGE INTO ln\_user\_save\_preferences d "** \  
 **"USING (SELECT "** + lens\_id + **" lens\_id, "** + user\_id + **" user\_id from dual) s"** \  
 **" ON (d.lens\_id = s.lens\_id) "** \  
 **" WHEN MATCHED THEN UPDATE SET user\_id = :updated\_user\_id,d.p\_name = :updated\_name,d.p\_format = :updated\_format, d.p\_MODE = :updated\_mode, d.p\_updated\_date = :updated\_date "** \  
 **"WHEN NOT MATCHED THEN INSERT(lens\_id, user\_id,p\_format,p\_MODE,p\_name,p\_updated\_date) "** \  
 **"VALUES (:updated\_lens\_id,:updated\_user\_id,:upd\_format,:upd\_mode, :upd\_name, :upd\_date)"** )  
  
 bind\_vars = {  
 **"updated\_name"** : updated\_name,  
 **"updated\_format"** : updated\_format,  
 **"updated\_mode"** : updated\_mode,  
 **"updated\_date"** : updated\_date,  
 **"updated\_lens\_id"** :updated\_lens\_id,  
 **"updated\_user\_id"** : updated\_user\_id,  
 **"upd\_format"** : updated\_format,  
 **"upd\_mode"** : updated\_mode,  
 **"upd\_name"** : updated\_name,  
 **"upd\_date"** : updated\_date  
 }  
 **return** self.execute\_query(sql,bind\_vars)  
 **except** cx\_Oracle.DatabaseError **as** ce:  
 LOGGER.exception(**"Exception when insert or update operations in user preferences : %s"**,ce)  
 **raise** Exception(message\_constants.INSERT\_OR\_UPDATE\_FAILED)  
  
 **def** execute\_query(self, sql,bind\_vars):  
 *"""  
 Establish the database connection and execute the query for update, delete and insert operations.* **:param** *sql: statement for insert or update or delete operations.* **:param** *bind\_vars: variables are bind when executing the statement.  
 """* **try**:  
 connection = self.\_\_connection()  
 cursor = connection.cursor()  
 cursor.execute(sql,bind\_vars)  
 connection.commit()  
 LOGGER.info(**"Save UserPreference sucessfully."**)  
 **except** Exception **as** ex:  
 connection.rollback()  
 LOGGER.exception(**"Exception while executing the query : %s"**, ex)  
 **raise** Exception(message\_constants.EXECUTE\_STATEMENT\_FAILED)  
 **finally**:  
 connection.close()  
  
 **def** get\_userpref\_format\_mode(self,lens\_id,user\_id):  
 file\_format = **'csv'** write\_mode = **'overwrite'  
 try**:  
 conn = self.\_\_connection()  
 cursor = conn.cursor()  
 bind = {**"lensid"**:lens\_id,**"userid"**:user\_id}  
 sql = **"select p\_format, p\_mode from ln\_user\_save\_preferences where lens\_id = :lensid and user\_id = :userid"** LOGGER.info(**"userid : {} and lensid : {} from preferences"**.format(user\_id,lens\_id))  
 cursor.execute(sql,bind)  
 **for** row **in** cursor:  
 file\_format = str(row[0])  
 write\_mode = str(row[1])  
 **return** file\_format, write\_mode  
 **except** Exception **as** ex:  
 LOGGER.exception(**"Exception while executing userpref file format : %s"**, ex)  
 **raise** Exception(message\_constants.EXECUTE\_STATEMENT\_FAILED)  
 **finally**:  
 conn.close()

*sort\_step*

*"""  
This module is used for apply operation sort on data set  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** mashupbase **import** DataMashUpBase  
**import** constants  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** SortStep(DataMashUpBase):  
 *"""  
 This class is used take json as input in \_\_init\_\_ method and apply sort operation on request  
 dataset  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.column\_name = input\_json[constants.COLUMN\_NAME]  
 self.sort\_by = False **if** input\_json[constants.SORT\_BY] == **"DESC" else** True  
 self.sort\_type = input\_json[constants.SORT\_TYPE]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** find\_and\_replace\_in\_list(cls, list\_values, find, replace):  
 *"""  
 This method is used for find a element from list and replace with specific values.* **:param** *find:* **:param** *replace:* **:param** *list\_values* **:return***:  
 """* **for** n, i **in** enumerate(list\_values):  
 **if** list\_values[n] == find:  
 list\_values[n] = replace  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method will take input as dataframe and appl* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 keep\_removing\_column = []  
  
 **for** key, value **in** data\_frame.dtypes.iteritems():  
 **if** value == **'object' and** key **in** self.column\_name:  
 new\_column\_name = key + **"\_sort"** data\_frame[new\_column\_name] = data\_frame[key].str.lower()  
 SortStep.find\_and\_replace\_in\_list(self.column\_name, key, new\_column\_name)  
 keep\_removing\_column.append(new\_column\_name)  
  
 data\_frame = data\_frame.sort\_values(by=self.column\_name, ascending=self.sort\_by,  
 kind=self.sort\_type)  
 **return** data\_frame.drop(keep\_removing\_column, axis=1).reset\_index(drop=True)  
 **except** IndexError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.SORT\_NO\_COLUMN\_SELECTED)  
 **except** KeyError **as** ex:  
 **raise** Exception(message\_constants.COLUMN\_NOT\_FOUND.format(ex.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to write a test case for sort\_step  
"""***import** unittest  
**import** pandas **as** pd  
**from** sort\_step **import** SortStep  
  
  
**class** TestSortStep(unittest.TestCase):  
 *"""  
 This class is used to write test method for sort step  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/10Records.txt"**, delimiter=**'\t'**, parse\_dates=[**'field0'**])  
 self.sort\_json = {  
 **"columnName"**: [**"field0"**, **"field2"**, **"field6"**],  
 **"sort\_by"**: **"ASC"**,  
 **"sort\_type"** : **"quicksort"**,  
 **"step\_name"**: **"sort\_step.sort\_step.SortStep"** }  
  
 **def** test\_sort\_step(self):  
 *"""  
 Test suite for performing sort operation on date field* **:return***:  
 """* sort\_step = SortStep(self.sort\_json)  
 data\_frame = sort\_step.get\_result(self.data\_frame)  
 **if** data\_frame[**"field6"**][0] == **"ABHISHEK PAWAR"**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_sort\_step\_with\_key\_missing(self):  
 *"""  
 test suit for keyError exception* **:return***:  
 """* **try**:  
 self.sort\_json.pop(**"sort\_by"**, None)  
 sort\_step = SortStep(self.sort\_json)  
 sort\_step.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_sort\_step\_with\_index\_error\_exception(self):  
 *"""  
 test suite for exception* **:return***:  
 """* **try**:  
 self.sort\_json.update({**"columnName"**: []})  
 sort\_step = SortStep(self.sort\_json)  
 sort\_step.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **if** ex.args[0] == **'No column used for sort operation'**:  
 self.assertEqual(1, 1)  
  
 **def** test\_sort\_step\_with\_key\_error\_exception(self):  
 *"""  
 test suite for exception* **:return***:  
 """* **try**:  
 self.sort\_json.update({**"columnName"**: [**'CityName'**]})  
 sort\_step = SortStep(self.sort\_json)  
 sort\_step.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **if** ex.args[0] == **'CityName column not found in the dataset'**:  
 self.assertEqual(1, 1)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*Transformation*

*Data\_cleansing*

*"""  
@author: Aaditya Agrawal  
Script to clean data by index  
"""***import** logging  
**import** numpy **as** np  
**import** constants  
**import** message\_constants  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** ExtractByIndex:  
 *"""  
 Convert value as list separated by input delimeter and return value from list for given index  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 Initialization* **:param** *input\_json:  
 """* **try**:  
 self.column\_name = str(input\_json[constants.COLUMN\_NAME])  
 self.delimiter = str(input\_json[constants.DELIMITER])  
 self.index\_position = int(input\_json[constants.INDEX\_POSITION])  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, dataframe):  
 *"""  
 Main logic to perform extact by index from column values* **:param** *dataframe:* **:return***: dataframe with new column having index value  
 """* **try**:  
 output\_values = []  
 **for** val **in** dataframe[self.column\_name].astype(str):  
 **if** self.delimiter **in** val:  
 temp\_list = val.split(self.delimiter)  
 **if** len(temp\_list) >= self.index\_position **and** self.index\_position > 0:  
 output\_values.append(temp\_list[int(self.index\_position - 1)])  
 **else**:  
 output\_values.append(np.nan)  
 **else**:  
 output\_values.append(np.nan)  
 new\_col\_name = self.column\_name + **'\_'** + constants.EXTRACT\_BY\_INDEX\_COL  
 **if** new\_col\_name **in** dataframe.columns:  
 **for** i **in** range(1, 100):  
 **if** (new\_col\_name + **'\_'** + str(i)) **not in** dataframe.columns:  
 new\_col\_name = new\_col\_name + **'\_'** + str(i)  
 **break  
 else**:  
 new\_col\_name = self.column\_name  
 **if** constants.EXTRACT\_BY\_INDEX\_COL **in** self.column\_name:  
 **for** i **in** range(1, 100):  
 **if** (new\_col\_name + **'\_'** + str(i)) **not in** dataframe.columns:  
 new\_col\_name = new\_col\_name + **'\_'** + str(i)  
 **break  
 else**:  
 new\_col\_name = self.column\_name + **'\_'** + constants.EXTRACT\_BY\_INDEX\_COL  
 dataframe[new\_col\_name] = output\_values  
 **return** dataframe  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** get\_input\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find input col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[constants.COLUMN\_NAME]]  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **return** []  
  
 @classmethod  
 **def** get\_output\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find output col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[constants.TO\_COLUMN]]  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **return** []  
  
 @classmethod  
 **def** update\_step\_for\_col\_map(cls, step\_json, column\_mapping\_list):  
 *"""  
 used to update step and col map list wrt step* **:param** *step\_json:* **:param** *column\_mapping\_list:* **:return***: list of step json and updated col mapping list  
 """* updated\_column\_mapping\_list = column\_mapping\_list  
 **try**:  
 input\_cols\_list = ExtractByIndex.get\_input\_column\_name\_values(step\_json)  
 **for** col\_obj **in** column\_mapping\_list:  
 **if** col\_obj[**"column\_used"**] **in** input\_cols\_list:  
 step\_json.update({constants.COLUMN\_NAME: col\_obj[**"mapped\_with"**]})  
 **break  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** {**"step\_json"**: step\_json, **"column\_mapping\_list"**: updated\_column\_mapping\_list}

*"""  
@author: Aaditya Agrawal  
Script to clean data by input pattern  
"""***import** re  
**import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
**class** ExtractByPattern:  
 *"""  
 Returns comma separated values matching input pattern  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 Initialization* **:param** *input\_json:  
 """* **try**:  
 self.column\_name = str(input\_json[constants.COLUMN\_NAME])  
 self.data\_pattern = input\_json[constants.DATA\_PATTERN]  
 self.extract\_value\_count = input\_json.get(constants.EXTRACT\_VALUE\_COUNT, False)  
 self.extract\_unique\_sorted = input\_json.get(constants.EXTRACT\_UNIQUE\_SORTED, False)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception %s'**, ex.args[0])  
 **raise** Exception(**"Invalid inputs"**)  
  
 **def** get\_result(self, dataframe): *#pylint: disable-msg=too-many-branches  
 """  
 Main logic to perform extact by index from column values* **:param** *dataframe:* **:return***: dataframe with new column having index value  
 """* **try**:  
 regex = **""** is\_tuple = True  
 **if** self.data\_pattern[0] == **'\\'**:  
 regex = self.generate\_regex(self.data\_pattern)  
 **else**:  
 is\_tuple = False  
 regex = self.data\_pattern  
 output\_values = []  
 **if** self.extract\_value\_count:  
 **for** val **in** dataframe[self.column\_name].astype(str):  
 temp\_list = re.findall(regex, val)  
 **if** is\_tuple:  
 temp\_list = [x **for** y **in** temp\_list **for** x **in** list(y) **if** x != **''**]  
 output\_values.append(len(temp\_list))  
 **elif** self.extract\_unique\_sorted:  
 **for** val **in** dataframe[self.column\_name].astype(str):  
 temp\_list = re.findall(regex, val)  
 **if** is\_tuple:  
 temp\_list = [x **for** y **in** temp\_list **for** x **in** list(y) **if** x != **''**]  
 temp\_list = sorted(set(temp\_list))  
 output\_values.append(**", "**.join(temp\_list))  
 **else**:  
 **for** val **in** dataframe[self.column\_name].astype(str):  
 temp\_list = re.findall(regex, val)  
 **if** is\_tuple:  
 temp\_list = [x **for** y **in** temp\_list **for** x **in** list(y) **if** x != **''**]  
 output\_values.append(**", "**.join(temp\_list))  
 new\_col\_name = self.column\_name + **'\_'** + constants.EXTRACT\_VALUE\_PATTERN\_COL  
 **if** new\_col\_name **in** dataframe.columns:  
 **for** i **in** range(1, 100):  
 **if** (new\_col\_name + **'\_'** + str(i)) **not in** dataframe.columns:  
 new\_col\_name = new\_col\_name + **'\_'** + str(i)  
 **break  
 else**:  
 new\_col\_name = self.column\_name  
 **if** constants.EXTRACT\_VALUE\_PATTERN\_COL **in** self.column\_name:  
 **for** i **in** range(1, 100):  
 **if** (new\_col\_name + **'\_'** + str(i)) **not in** dataframe.columns:  
 new\_col\_name = new\_col\_name + **'\_'** + str(i)  
 **break  
 else**:  
 new\_col\_name = self.column\_name + **'\_'** + constants.EXTRACT\_VALUE\_PATTERN\_COL  
 dataframe[new\_col\_name] = output\_values  
 **return** dataframe  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception %s'**, ex.args[0])  
 **raise** Exception(**"Operation Failed. Please try again."**)  
  
 @staticmethod  
 **def** generate\_regex(data\_pattern): *#pylint: disable-msg=too-many-branches  
 """* **:param** *data\_pattern:* **:return***:  
 """* LOGGER.debug(**'generate\_regex STARTS'**)  
 actual\_expression = data\_pattern[1:]  
 reg\_ex = **""** digit\_list = [0]  
 digit\_counter = 0  
 alpha\_list = [0]  
 alpha\_counter = 0  
 **for** char **in** actual\_expression:  
 **if** char.isdigit():  
 **if not** reg\_ex.endswith(**"digit"**):  
 reg\_ex = reg\_ex + **"digit"** digit\_counter = digit\_counter + 1  
 **if** len(digit\_list) == digit\_counter:  
 digit\_list[digit\_counter - 1] = digit\_list[digit\_counter - 1] + 1  
 **else**:  
 digit\_list.append(1)  
 **elif** char.isalpha():  
 **if not** reg\_ex.endswith(**"alpha"**):  
 reg\_ex = reg\_ex + **"alpha"** alpha\_counter = alpha\_counter + 1  
 **if** len(alpha\_list) == alpha\_counter:  
 alpha\_list[alpha\_counter - 1] = alpha\_list[alpha\_counter - 1] + 1  
 **else**:  
 alpha\_list.append(1)  
 **else**:  
 reg\_ex = reg\_ex + char  
 alpha\_counter = 0  
 digit\_counter = 0  
 preceded\_by = **""** followed\_by = **""  
 if** reg\_ex.startswith(**"alpha"**):  
 preceded\_by = **"(?<=[^a-zA-Z'])"  
 elif** reg\_ex.startswith(**"digit"**):  
 preceded\_by = **"(?<=[^0-9])"  
 if** reg\_ex.endswith(**"alpha"**):  
 followed\_by = **"(?=[^a-zA-Z'])"  
 elif** reg\_ex.endswith(**"digit"**):  
 followed\_by = **"(?=[^0-9])"  
 for** char **in** reg\_ex:  
 **if "alpha" in** reg\_ex:  
 reg\_ex = reg\_ex.replace(**"alpha"**,  
 **"[a-zA-z]"** + **"{"** + str(alpha\_list[alpha\_counter]) + **"}"**, 1)  
 alpha\_counter = alpha\_counter + 1  
 **if "digit" in** reg\_ex:  
 reg\_ex = reg\_ex.replace(**"digit"**, **"\\d"** + **"{"** + str(digit\_list[digit\_counter]) + **"}"**,  
 1)  
 digit\_counter = digit\_counter + 1  
 final\_reg\_ex = **"(^"** + reg\_ex + **")"** + **"(($)|"** + followed\_by + **")|("** + preceded\_by + reg\_ex\  
 + followed\_by + **")|"** + preceded\_by + **"("** + reg\_ex + **"$)"** LOGGER.debug(**'generate\_regex ENDS'**)  
 **return** final\_reg\_ex  
  
 @classmethod  
 **def** get\_input\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find input col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[constants.COLUMN\_NAME]]  
 **except** Exception:  
 **return** []  
  
 @classmethod  
 **def** get\_output\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find output col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[constants.TO\_COLUMN]]  
 **except** Exception:  
 **return** []  
  
 @classmethod  
 **def** update\_step\_for\_col\_map(cls, step\_json, column\_mapping\_list):  
 *"""  
 used to update step and col map list wrt step* **:param** *step\_json:* **:param** *column\_mapping\_list:* **:return***: list of step json and updated col mapping list  
 """* updated\_column\_mapping\_list = column\_mapping\_list  
 **try**:  
 input\_cols\_list = ExtractByPattern.get\_input\_column\_name\_values(step\_json)  
 **for** col\_obj **in** column\_mapping\_list:  
 **if** col\_obj[**"column\_used"**] **in** input\_cols\_list:  
 step\_json.update({constants.COLUMN\_NAME: col\_obj[**"mapped\_with"**]})  
 **break  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception %s'**, ex.args[0])  
 **return** {**"step\_json"**: step\_json, **"column\_mapping\_list"**: updated\_column\_mapping\_list}

*"""  
@author: Aaditya Agrawal  
Script to get count of values by separating values using delimiter  
"""***import** logging  
**import** constants  
**import** message\_constants  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** ExtractValueCount:  
 *"""  
 Convert value as list separated by input delimeter and return count of list  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 Initialization* **:param** *input\_json:  
 """* **try**:  
 self.column\_name = str(input\_json[constants.COLUMN\_NAME])  
 self.delimiter = str(input\_json[constants.DELIMITER])  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, dataframe): *# pylint: disable-msg=too-many-branches  
 """  
 Main logic to perform count of values from column values* **:param** *dataframe:* **:return***: dataframe with new column having index value  
 """* **try**: *# pylint: disable-msg=too-many-nested-blocks* output\_values = []  
 **for** val **in** dataframe[self.column\_name].astype(str):  
 **if** self.delimiter **in** val:  
 temp\_list = val.split(self.delimiter)  
 temp\_list\_length = len(temp\_list)  
 **if** temp\_list\_length > 1:  
 **if** temp\_list\_length == 2:  
 **if** temp\_list[0] **and** temp\_list[1]:  
 output\_values.append(2)  
 **elif** temp\_list[0] **or** temp\_list[1]:  
 output\_values.append(1)  
 **else**:  
 output\_values.append(0)  
 **else**:  
 output\_values.append(temp\_list\_length)  
 **else**:  
 output\_values.append(0)  
 **else**:  
 output\_values.append(0)  
 new\_col\_name = self.column\_name + **'\_'** + constants.EXTRACT\_VALUE\_COUNT\_COL  
 **if** new\_col\_name **in** dataframe.columns:  
 **for** i **in** range(1, 100):  
 **if** (new\_col\_name + **'\_'** + str(i)) **not in** dataframe.columns:  
 new\_col\_name = new\_col\_name + **'\_'** + str(i)  
 **break  
 else**:  
 new\_col\_name = self.column\_name  
 **if** constants.EXTRACT\_VALUE\_COUNT\_COL **in** self.column\_name:  
 **for** i **in** range(1, 100):  
 **if** (new\_col\_name + **'\_'** + str(i)) **not in** dataframe.columns:  
 new\_col\_name = new\_col\_name + **'\_'** + str(i)  
 **break  
 else**:  
 new\_col\_name = self.column\_name + **'\_'** + constants.EXTRACT\_VALUE\_COUNT\_COL  
 dataframe[new\_col\_name] = output\_values  
 **return** dataframe  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** get\_input\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find input col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[constants.COLUMN\_NAME]]  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **return** []  
  
 @classmethod  
 **def** get\_output\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find output col name keys for current feature* **:param** *step\_json:* **:return***: list of keys  
 """* **return** []  
  
 @classmethod  
 **def** update\_step\_for\_col\_map(cls, step\_json, column\_mapping\_list):  
 *"""  
 used to update step and col map list wrt step* **:param** *step\_json:* **:param** *column\_mapping\_list:* **:return***: list of step json and updated col mapping list  
 """* updated\_column\_mapping\_list = column\_mapping\_list  
 **try**:  
 input\_cols\_list = ExtractValueCount.get\_input\_column\_name\_values(step\_json)  
 **for** col\_obj **in** column\_mapping\_list:  
 **if** col\_obj[**"column\_used"**] **in** input\_cols\_list:  
 step\_json.update({constants.COLUMN\_NAME: col\_obj[**"mapped\_with"**]})  
 **break  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** {**"step\_json"**: step\_json, **"column\_mapping\_list"**: updated\_column\_mapping\_list}

*"""  
@author: Aaditya Agrawal  
This python file is contains test case for extract by index method of data cleansing  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** transformation.data\_cleansing.extract\_by\_index **import** ExtractByIndex  
  
  
**class** TestExtractByIndex(TestCase):  
 *"""  
 Test case base class  
 """* **def** setUp(self):  
 dict\_obj = {  
 **"Name"**: [**"aaditya,agrawal"**, **"lens,team,work,product"**,  
 **"hello1,hello2,hello3"**, **"nothing"**],  
 **"Dummy"**: [**'1;2;3'**, **'a;b;c'**, **'car;tyre;color'**, **';'**],  
 **"Num\_col"**: [1, 2, 3, 4]  
 }  
 self.data\_frame = pd.DataFrame(dict\_obj, columns=[**'Name'**, **'Dummy'**, **'Num\_col'**])  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_get\_result\_comma\_delimiter(self):  
 *"""  
 This function is used to get result as df with extract by index method* **:return***: dataframe  
 """* json\_request = {  
 **"columnName"**: **"Name"**,  
 **"delimiter"**: **","**,  
 **"index\_position"**: 2  
 }  
 extract\_by\_index = ExtractByIndex(json\_request)  
 result\_df = extract\_by\_index.get\_result(self.data\_frame)  
 **print**(result\_df)  
 self.assertEqual(1, 1)  
  
 **with** self.assertRaises(Exception):  
 extract\_by\_index.get\_result(None)  
  
 **def** test\_get\_result\_selicolumn\_delimiter(self):  
 *"""  
 This function is used to get result as df with extract by index method* **:return***: dataframe  
 """* json\_request = {  
 **"columnName"**: **"Dummy"**,  
 **"delimiter"**: **";"**,  
 **"index\_position"**: 2  
 }  
 extract\_by\_index = ExtractByIndex(json\_request)  
 result\_df = extract\_by\_index.get\_result(self.data\_frame)  
 **print**(**"\n\n"**, result\_df)  
 self.assertEqual(1, 1)  
  
 **def** test\_get\_result\_delimiter(self):  
 *"""  
 This function is used to get result as df with extract by index method* **:return***: dataframe  
 """* json\_request = {  
 **"columnName"**: **"Num\_col"**,  
 **"delimiter"**: **","**,  
 **"index\_position"**: 1  
 }  
 extract\_by\_index = ExtractByIndex(json\_request)  
 result\_df = extract\_by\_index.get\_result(self.data\_frame)  
 **print**(**"\n\n"**, result\_df)  
 self.assertEqual(1, 1)  
  
 **def** test\_throw\_exception(self):  
 *"""  
 Throw exception test* **:return***:  
 """* **try**:  
 json\_request = {  
 **"columnName"**: **"Num\_col"**,  
 **"index\_position"**: 1  
 }  
 extract\_by\_index = ExtractByIndex(json\_request)  
 extract\_by\_index.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(**"\n\n"**, ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_update\_step\_for\_col\_map(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* steps = {}  
  
 column\_mapping\_list = [{**'column\_used'**: **'outsidecolumn'**, **'mapped\_with'**: **'innings\_new'**}]  
 output\_json = ExtractByIndex.update\_step\_for\_col\_map(steps, column\_mapping\_list)  
 self.assertEqual(column\_mapping\_list, output\_json[**'column\_mapping\_list'**])  
  
 **def** test\_update\_step\_for\_col\_map\_error(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* self.assertEqual(ExtractByIndex.update\_step\_for\_col\_map(None, None),  
 {**'column\_mapping\_list'**: None, **'step\_json'**: None})  
  
 **def** test\_get\_output\_column\_name\_values(self):  
 *"""* **:return***:  
 """* self.assertEqual(ExtractByIndex.get\_output\_column\_name\_values(None), [])  
  
 **def** test\_for\_df\_col\_names(self):  
 *"""  
 test case for col names already exists* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Name"**,  
 **"delimiter"**: **","**,  
 **"index\_position"**: 2  
 }  
 extract\_by\_index = ExtractByIndex(input\_json)  
 self.data\_frame[**"Name\_extracted\_index"**] = [**'a'**, **'b'**, **'c'**, **'d'**]  
 result\_df = extract\_by\_index.get\_result(self.data\_frame)  
 **print**(**"\n\n"**, result\_df)  
 input\_json = {  
 **"columnName"**: **"Name\_extracted\_index\_1"**,  
 **"delimiter"**: **","**,  
 **"index\_position"**: 2  
 }  
 extract\_by\_index = ExtractByIndex(input\_json)  
 result\_df = extract\_by\_index.get\_result(self.data\_frame)  
 **print**(**"\n\n"**, result\_df)  
 extract\_by\_index = ExtractByIndex(input\_json)  
 result\_df = extract\_by\_index.get\_result(self.data\_frame)  
 **print**(**"\n\n"**, result\_df)  
 self.assertEqual(1, 1)

*"""  
@author: Aaditya Agrawal  
This python file is contains test case for extract by pattern method of data cleansing  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**import** numpy **as** np  
**from** transformation.data\_cleansing.extract\_by\_pattern **import** ExtractByPattern  
  
  
**class** TestExtractByPattern(TestCase):  
 *"""  
 Test case base class  
 """* **def** setUp(self):  
 dict\_obj = {  
  
 **"Name"**: [**"111abc123ab@123"**, **"abcd1234ab@12efgh9876"**, **"abc,a d1234a b@12efgh9876 poi.p"**,  
 **"a,c"**],  
 **"Dummy"**: [**'a'**, **'b'**, **'c'**,  
 **'abc1111bcd2222ooe33333pod 4444 55 55 q6666 7777ty 8888@qwer @9999sd'**],  
 **"Num\_col"**: [1, 2, 3, 4]  
 }  
 self.data\_frame = pd.DataFrame(dict\_obj, columns=[**'Name'**, **'Dummy'**, **'Num\_col'**])  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_numeric\_pattern\_1(self):  
 *"""  
 test for numeric 999* **:return***:  
 """* input\_json = {**"columnName"**: **"Name"**, **"data\_pattern"**: **"\\999"**}  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 result\_df.loc[  
 result\_df[**'Name\_extracted\_pattern'**] == **''**, **'Name\_extracted\_pattern'**] = np.NaN  
 self.assertEqual(len(result\_df[**'Name\_extracted\_pattern'**].dropna()), 1)  
  
 *# None as DataFrame* **with** self.assertRaises(Exception):  
 extract\_by\_pattern.get\_result(None)  
  
 *# pass empty json to init method* **with** self.assertRaises(Exception):  
 ExtractByPattern({})  
  
 **def** test\_numeric\_pattern\_2(self):  
 *"""  
 test for numeric 9999* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Name"**,  
 **"data\_pattern"**: **"\\9999"** }  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 result\_df.loc[  
 result\_df[**'Name\_extracted\_pattern'**] == **''**, **'Name\_extracted\_pattern'**] = np.NaN  
 self.assertEqual(len(result\_df[**'Name\_extracted\_pattern'**].dropna()), 2)  
  
 **def** test\_numeric\_pattern\_3(self):  
 *"""  
 test for numeric 99999* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Dummy"**,  
 **"data\_pattern"**: **"\\99999"** }  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 result\_df.loc[  
 result\_df[**'Dummy\_extracted\_pattern'**] == **''**, **'Dummy\_extracted\_pattern'**] = np.NaN  
 self.assertEqual(len(result\_df[**'Dummy\_extracted\_pattern'**].dropna()), 1)  
  
 **def** test\_text\_pattern\_1(self):  
 *"""  
 test for numeric AAA* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Name"**,  
 **"data\_pattern"**: **"\\AAA"** }  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 result\_df.loc[  
 result\_df[**'Name\_extracted\_pattern'**] == **''**, **'Name\_extracted\_pattern'**] = np.NaN  
 self.assertEqual(len(result\_df[**'Name\_extracted\_pattern'**].dropna()), 2)  
  
 **def** test\_text\_pattern\_2(self):  
 *"""  
 test for numeric AAAA* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Name"**,  
 **"data\_pattern"**: **"\\AAAA"** }  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 result\_df.loc[  
 result\_df[**'Name\_extracted\_pattern'**] == **''**, **'Name\_extracted\_pattern'**] = np.NaN  
 self.assertEqual(len(result\_df[**'Name\_extracted\_pattern'**].dropna()), 2)  
  
 **def** test\_text\_pattern\_3(self):  
 *"""  
 test for numeric AAAAA* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Dummy"**,  
 **"data\_pattern"**: **"\\AAAAA"** }  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 result\_df.loc[  
 result\_df[**'Dummy\_extracted\_pattern'**] == **''**, **'Dummy\_extracted\_pattern'**] = np.NaN  
 self.assertEqual(len(result\_df[**'Dummy\_extracted\_pattern'**].dropna()), 0)  
  
 **def** test\_text\_pattern\_4(self):  
 *"""  
 test for numeric AAA,A* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Name"**,  
 **"data\_pattern"**: **"\\AAA,A"** }  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 result\_df.loc[  
 result\_df[**'Name\_extracted\_pattern'**] == **''**, **'Name\_extracted\_pattern'**] = np.NaN  
 self.assertEqual(len(result\_df[**'Name\_extracted\_pattern'**].dropna()), 1)  
  
 **def** test\_text\_pattern\_5(self):  
 *"""  
 test for numeric AAA.A* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Name"**, **"data\_pattern"**: **"\\AAA.A"**}  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 result\_df.loc[  
 result\_df[**'Name\_extracted\_pattern'**] == **''**, **'Name\_extracted\_pattern'**] = np.NaN  
 self.assertEqual(len(result\_df[**'Name\_extracted\_pattern'**].dropna()), 1)  
  
 **def** test\_text\_pattern\_6(self):  
 *"""  
 test for numeric a,c* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Name"**, **"data\_pattern"**: **"\\a,c"**, **"extract\_value\_count"**: True  
 }  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 self.assertEqual(len(result\_df[**'Name\_extracted\_pattern'**].dropna()), 4)  
  
 **def** test\_numeric\_exception\_1(self):  
 *"""  
 test for numeric 999,a* **:return***:  
 """* input\_json = {**"columnName"**: **"Dummy1"**, **"data\_pattern"**: **"\\999,a"**}  
 **with** self.assertRaises(Exception):  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 extract\_by\_pattern.get\_result(self.data\_frame)  
  
 **def** test\_update\_step\_for\_col\_map(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* steps = {}  
 column\_mapping\_list = [{**'column\_used'**: **'outsidecolumn'**, **'mapped\_with'**: **'innings\_new'**}]  
 output\_json = ExtractByPattern.update\_step\_for\_col\_map(steps, column\_mapping\_list)  
 self.assertEqual(column\_mapping\_list, output\_json[**'column\_mapping\_list'**])  
  
 **def** test\_update\_step\_for\_col\_map\_error(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* self.assertEqual(ExtractByPattern.update\_step\_for\_col\_map(None, None),  
 {**'column\_mapping\_list'**: None, **'step\_json'**: None})  
  
 **def** test\_get\_output\_column\_name\_values(self):  
 *"""* **:return***:  
 """* self.assertEqual(ExtractByPattern.get\_output\_column\_name\_values(None), [])  
  
 **def** test\_for\_df\_col\_names(self):  
 *"""  
 test case for col names already exists* **:return***:  
 """* input\_json = {**"columnName"**: **"Name"**, **"data\_pattern"**: **"\\a,c"**}  
 extract\_by\_pattern = ExtractByPattern(input\_json)  
 self.data\_frame[**"Name\_extracted\_pattern"**] = [**'a'**, **'b'**, **'c'**, **'d'**]  
 result\_df = extract\_by\_pattern.get\_result(self.data\_frame)  
 **print**(**"\n\n"**, result\_df.columns)  
 input\_json = {  
 **"columnName"**: **"Name\_extracted\_pattern\_1"**,  
 **"data\_pattern"**: **"\\a,c"** }  
 extract\_by\_index = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_index.get\_result(self.data\_frame)  
 **print**(**"\n\n"**, result\_df.columns)  
 extract\_by\_index = ExtractByPattern(input\_json)  
 result\_df = extract\_by\_index.get\_result(self.data\_frame)  
 **print**(**"\n\n"**, result\_df.columns)  
 self.assertEqual(1, 1)

*"""  
@author: Aaditya Agrawal  
This python file is contains test case for extract by index method of data cleansing  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** transformation.data\_cleansing.extract\_value\_count **import** ExtractValueCount  
  
  
**class** TestExtractByIndex(TestCase):  
 *"""  
 Test case base class  
 """* **def** setUp(self):  
 dict\_obj = {  
 **"Name"**: [**"aaditya,agrawal"**, **"lens,team,work,product"**,  
 **"hello1,hello2,hello3"**, **"nothing"**],  
 **"Dummy"**: [**'1;2;3,4'**, **','**, **';'**, **'a;'**],  
 **"Num\_col"**: [1, 2, 3, 4]  
 }  
 self.data\_frame = pd.DataFrame(dict\_obj, columns=[**'Name'**, **'Dummy'**, **'Num\_col'**])  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_get\_result\_comma\_delimiter(self):  
 *"""  
 This function is used to get result as df with extract by index method* **:return***: dataframe  
 """* json\_request = {  
 **"columnName"**: **"Name"**,  
 **"delimiter"**: **","** }  
 extract\_value\_count = ExtractValueCount(json\_request)  
 extract\_value\_count.get\_result(self.data\_frame)  
 self.assertEqual(1, 1)  
 **with** self.assertRaises(Exception):  
 extract\_value\_count.get\_result(None)  
  
 **def** test\_get\_result\_selicolumn\_delimiter(self):  
 *"""  
 This function is used to get result as df with extract by index method* **:return***: dataframe  
 """* json\_request = {  
 **"columnName"**: **"Dummy"**,  
 **"delimiter"**: **";"** }  
 extract\_value\_count = ExtractValueCount(json\_request)  
 extract\_value\_count.get\_result(self.data\_frame)  
 self.assertEqual(1, 1)  
  
 **def** test\_get\_result\_delimiter(self):  
 *"""  
 This function is used to get result as df with extract by index method* **:return***: dataframe  
 """* json\_request = {  
 **"columnName"**: **"Num\_col"**,  
 **"delimiter"**: **","** }  
 extract\_value\_count = ExtractValueCount(json\_request)  
 extract\_value\_count.get\_result(self.data\_frame)  
 self.assertEqual(1, 1)  
  
 **def** test\_throw\_exception(self):  
 *"""  
 Throw exception test* **:return***:  
 """* **with** self.assertRaises(Exception):  
 json\_request = {  
 **"columnName"**: **"Num\_col"** }  
 extract\_value\_count = ExtractValueCount(json\_request)  
 extract\_value\_count.get\_result(self.data\_frame)  
  
 **def** test\_update\_step\_for\_col\_map(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* steps = {}  
  
 column\_mapping\_list = [{**'column\_used'**: **'outsidecolumn'**, **'mapped\_with'**: **'innings\_new'**}]  
 output\_json = ExtractValueCount.update\_step\_for\_col\_map(steps, column\_mapping\_list)  
 self.assertEqual(column\_mapping\_list, output\_json[**'column\_mapping\_list'**])  
  
 **def** test\_update\_step\_for\_col\_map\_error(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* self.assertEqual(ExtractValueCount.update\_step\_for\_col\_map(None, None),  
 {**'column\_mapping\_list'**: None, **'step\_json'**: None})  
  
 **def** test\_get\_output\_column\_name\_values(self):  
 *"""* **:return***:  
 """* self.assertEqual(ExtractValueCount.get\_output\_column\_name\_values(None), [])  
  
 **def** test\_for\_df\_col\_names(self):  
 *"""  
 test case for col names already exists* **:return***:  
 """* input\_json = {  
 **"columnName"**: **"Name"**,  
 **"delimiter"**: **","** }  
 extract\_value\_count = ExtractValueCount(input\_json)  
 self.data\_frame[**"Name\_extracted\_count"**] = [**'a'**, **'b'**, **'c'**, **'d'**]  
 result\_df = extract\_value\_count.get\_result(self.data\_frame)  
 **print**(**"\n\n\n"**, result\_df.columns)  
 extract\_value\_count = ExtractValueCount(input\_json)  
 result\_df = extract\_value\_count.get\_result(self.data\_frame)  
 **print**(**"\n\n\n"**, result\_df.columns)  
 extract\_value\_count = ExtractValueCount(input\_json)  
 result\_df = extract\_value\_count.get\_result(self.data\_frame)  
 **print**(**"\n\n\n"**, result\_df.columns)  
 self.assertEqual(1, 1)

*Feature\_selector*

*"""  
@author: Aaditya Agrawal  
This script class returns n-columns for target from given dataframe  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** pandas **import** Series  
**import** pandas **as** pd  
**from** insights.what\_if\_analysis.what\_if\_analysis **import** WhatIfAnalysis  
**import** message\_constants  
  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** FeatureSelector:  
 *"""  
 Class that returns n-columns for target from given dataframe  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 Initialize class with input values* **:param** *input\_json: dict target as list and n as columns  
 """* **try**:  
 self.target\_list = [input\_json[**"target\_column"**]]  
 self.n\_columns = int(input\_json[**"n\_columns"**])  
 **if not** isinstance(self.n\_columns, int):  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **except** KeyError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(ex.args[0]))  
 **except** ValueError **as** inst:  
 LOGGER.error(**'Exception: %s'**, inst)  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""* **:param** *data\_frame: input dataframe to perform operation on* **:return***: data\_frame  
 """* **try**:  
 *# make copy of original data\_frame* data\_frame\_original = data\_frame.copy()  
 factor\_col\_list = []  
 *# get all numeric and factor columns based on data type* numerics = [**'int16'**, **'int32'**, **'int64'**, **'float16'**, **'float32'**, **'float64'**]  
 *# All numeric cols* numeric\_cols = list(data\_frame.select\_dtypes(include=numerics).columns)  
 *# All object dtype cols* factor\_cols = list(data\_frame.select\_dtypes(include=[**'object'**]).columns)  
  
 *# identify if a factor variable is stored as numeric* **for** col **in** numeric\_cols:  
 series = data\_frame[col].dropna() *# Remove all null values* var = Series(series, dtype=**"category"**)  
 **if** var.nunique() < 20: *# Return false if levels are more than 20 of the data* factor\_col\_list.append(col)  
 *# add factors list* factor\_col\_list.extend(factor\_cols)  
 *# perfom what if analysis using XGboost algorithm* \_, data\_frame = WhatIfAnalysis(data\_frame).label\_encoding(factor\_col\_list)  
 \_, all\_imp\_features = \  
 WhatIfAnalysis(data\_frame).imp\_feature(self.target\_list)  
 all\_imp\_features = [k **for** k, v **in** sorted(all\_imp\_features.items(),  
 key=**lambda** x: x[1], reverse=True)]  
 *# select only n\_columns* all\_imp\_features = all\_imp\_features[:self.n\_columns]  
 *# add target as part* all\_imp\_features.extend(self.target\_list)  
 *# return original data\_frame* **return** data\_frame\_original[all\_imp\_features]  
 **except** ValueError **as** value:  
 LOGGER.error(**'Exception: %s'**, value)  
 **raise** Exception(message\_constants.SELECT\_DATE\_TYPE\_COLUMN)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result\_importance(self, data\_frame):  
 *"""* **:param** *data\_frame: input dataframe to perform operation on* **:return***: data\_frame  
 """* **try**:  
 *# make copy of original data\_frame* data\_frame\_original = data\_frame.copy()  
 factor\_col\_list = []  
 *# get all numeric and factor columns based on data type* numerics = [**'int16'**, **'int32'**, **'int64'**, **'float16'**, **'float32'**, **'float64'**]  
 *# All numeric cols* numeric\_cols = list(data\_frame.select\_dtypes(include=numerics).columns)  
 *# All object dtype cols* factor\_cols = list(data\_frame.select\_dtypes(include=[**'object'**]).columns)  
  
 *# identify if a factor variable is stored as numeric* **for** col **in** numeric\_cols:  
 series = data\_frame[col].dropna() *# Remove all null values* var = Series(series, dtype=**"category"**)  
 **if** var.nunique() < 20: *# Return false if levels are more than 20 of the data* factor\_col\_list.append(col)  
 *# add factors list* factor\_col\_list.extend(factor\_cols)  
 *# perfom what if analysis using XGboost algorithm* \_, data\_frame = WhatIfAnalysis(data\_frame).label\_encoding(factor\_col\_list)  
 dict\_imp\_col, all\_imp\_features = \  
 WhatIfAnalysis(data\_frame).imp\_feature(self.target\_list)  
  
 importance\_df = pd.Series(dict\_imp\_col).to\_frame(**'Importance'**)  
 importance\_df[**'Importance'**] = importance\_df[**'Importance'**] \* 100  
 importance\_df = importance\_df.reset\_index()  
 importance\_df = importance\_df.sort\_values(by=[**'Importance'**],ascending=False)  
 importance\_df = importance\_df.head(self.n\_columns)  
  
 **return** importance\_df  
 **except** ValueError **as** value:  
 LOGGER.error(**'Exception: %s'**, value)  
 **raise** Exception(message\_constants.SELECT\_DATE\_TYPE\_COLUMN)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** get\_input\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find input col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[**"target\_column"**]]  
 **except** Exception:  
 **return** []  
  
 @classmethod  
 **def** get\_output\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find output col name keys for current feature* **:return***: list of keys  
 """* **return** []  
  
 @classmethod  
 **def** update\_step\_for\_col\_map(cls, step\_json, column\_mapping\_list):  
 *"""  
 used to update step and col map list wrt step* **:param** *step\_json:* **:param** *column\_mapping\_list:* **:return***: list of step json and updated col mapping list  
 """* updated\_column\_mapping\_list = column\_mapping\_list  
 **try**:  
 input\_cols\_list = FeatureSelector.get\_input\_column\_name\_values(step\_json)  
 **for** col\_obj **in** column\_mapping\_list:  
 **if** col\_obj[**"column\_used"**] **in** input\_cols\_list:  
 step\_json.update({**"target\_column"**: col\_obj[**"mapped\_with"**]})  
 **break  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** {**"step\_json"**: step\_json, **"column\_mapping\_list"**: updated\_column\_mapping\_list}

*"""  
@author: Aaditya Agrawal  
This python file is contains test case for feature\_selectors  
"""***from** unittest **import** TestCase  
**import** pandas **as** pd  
  
**from** transformation.feature\_selector.feature\_selector **import** FeatureSelector  
  
  
**class** TestValidation(TestCase):  
 *"""  
 This class is used to write test cases for FeatureSelector  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/Test\_what\_if\_analysis.csv"**)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_get\_result(self):  
 *"""  
 This function is used to write the test cases for FeatureSelector* **:return***: data\_frame  
 """  
 # print("-----------------------------Original---test\_get\_result\_remove\_rows")  
 # print(self.data\_frame.head())  
 # test case for remove rows* json\_request = {  
 **"target\_column"**: **"price"**,  
 **"n\_columns"**: 10  
 }  
 feature\_selectors = FeatureSelector(json\_request)  
 result\_df = feature\_selectors.get\_result(self.data\_frame)  
 *# print("-----------------------------result-----test\_get\_result\_remove\_rows")  
 # print(result\_df.head(), "\n\n\n\n")* self.assertEqual(json\_request[**"n\_columns"**] + 1, len(result\_df.columns))  
  
 **with** self.assertRaises(Exception):  
 feature\_selectors.get\_result(None)  
  
 **with** self.assertRaises(Exception):  
 FeatureSelector(None)  
  
 **with** self.assertRaises(Exception):  
 json\_request.pop(**'n\_columns'**)  
 FeatureSelector(json\_request)  
  
 **def** test\_update\_step\_for\_col\_map(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* steps = {}  
  
 column\_mapping\_list = [{**'column\_used'**: **'outsidecolumn'**, **'mapped\_with'**: **'innings\_new'**}]  
 output\_json = FeatureSelector.update\_step\_for\_col\_map(steps, column\_mapping\_list)  
 self.assertEqual(column\_mapping\_list, output\_json[**'column\_mapping\_list'**])  
  
 **def** test\_update\_step\_for\_col\_map\_error(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* self.assertEqual(FeatureSelector.update\_step\_for\_col\_map(None, None),  
 {**'column\_mapping\_list'**: None, **'step\_json'**: None})  
  
 **def** test\_get\_output\_column\_name\_values(self):  
 self.assertEqual(FeatureSelector.get\_output\_column\_name\_values(None), [])

*Imputations*

*"""  
This module contains the logic for imputation for dataframes  
on columns with BLANK/EMPTY/NULL/NAN values  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** pandas.api.types **import** is\_string\_dtype  
**import** pandas **as** pd  
**import** numpy **as** np  
**from** impyute **import** fast\_knn  
**import** constants  
**import** message\_constants  
np.seterr(divide=**'ignore'**, invalid=**'ignore'**)  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** Imputations:  
 *"""  
 This class contains function related to imputations over dataframes  
 on columns with BLANK/EMPTY/NULL/NAN values  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 Initialize object with paramater* **:param** *input\_json: json input data - required  
 """* **try**:  
 *# get default values* self.column\_name = input\_json[constants.COLUMN\_NAME]  
 self.imputation\_type = input\_json[constants.IMPUTATION\_TYPE]  
 *# determine imputation type Replace / Remove* **if** self.imputation\_type **in** [constants.REPLACE\_WITH]:  
 *# determine replace with method* **if** input\_json[constants.IMPUTATION\_METHOD] **in** [constants.MIN, constants.MAX,  
 constants.SUM, constants.MEDIAN,  
 constants.MODE, constants.ZERO]:  
 self.imputation\_method = input\_json[constants.IMPUTATION\_METHOD]  
 **elif** input\_json[constants.IMPUTATION\_METHOD] **in** [constants.REPLACE\_WITH\_COLUMN]:  
 self.replace\_to\_column = input\_json[constants.REPLACE\_TO\_COLUMN]  
 self.imputation\_method = input\_json[constants.IMPUTATION\_METHOD]  
 **elif** input\_json[constants.IMPUTATION\_METHOD] **in** [constants.DEFAULT\_REPLACE]:  
 self.imputation\_method = input\_json[constants.IMPUTATION\_METHOD]  
 self.default\_value = input\_json[constants.DEFAULT\_VALUE]  
 **elif** input\_json[constants.IMPUTATION\_METHOD] **in** [constants.KNN]:  
 self.imputation\_method = input\_json[constants.IMPUTATION\_METHOD]  
 **else**:  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **elif** self.imputation\_type **in** [constants.REMOVE\_ROWS]:  
 self.imputation\_method = constants.REMOVE\_ROWS  
 **else**:  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** \_imputation\_replace\_with(self, data\_frame, column\_dtype\_obj, column\_dtype\_date):  
 *"""  
 intenal function used for computation* **:param** *data\_frame:* **:param** *column\_dtype\_obj: boolean* **:param** *column\_dtype\_date: boolean* **:return***: string/date value  
 """* **try**:  
 new\_value = **''** *# validation on input dtype* **if** column\_dtype\_date:  
 **if** self.imputation\_method == constants.DEFAULT\_REPLACE:  
 **try**:  
 new\_value = pd.to\_datetime(str(self.default\_value))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.SELECT\_DATE\_TYPE\_COLUMN)  
 **elif** self.imputation\_method == constants.MODE:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].replace(**''**, np.nan)  
 new\_value = data\_frame[self.column\_name].mode().iloc[0]  
 **elif** column\_dtype\_obj:  
 **if** self.imputation\_method == constants.DEFAULT\_REPLACE:  
 **if not** (isinstance(self.default\_value, int) **and** isinstance(self.default\_value, float)):  
 new\_value = self.default\_value  
 **else**:  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **elif** self.imputation\_method == constants.MODE:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].replace(**''**, np.nan)  
 new\_value = data\_frame[self.column\_name].mode().iloc[0]  
 **else**:  
 **if** self.imputation\_method == constants.DEFAULT\_REPLACE:  
 **if not** isinstance(self.default\_value, (float, int)):  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **else**:  
 new\_value = self.default\_value  
 **else**:  
 temp\_dict = {  
 constants.MIN: data\_frame[self.column\_name].min(),  
 constants.MAX: data\_frame[self.column\_name].max(),  
 constants.SUM: data\_frame[self.column\_name].sum(),  
 constants.MEDIAN: data\_frame[self.column\_name].median(),  
 constants.MODE: data\_frame[self.column\_name].mode().iloc[0],  
 constants.ZERO: 0  
 }  
 new\_value = temp\_dict[self.imputation\_method]  
 **return** new\_value  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @staticmethod  
 **def** knn\_imputation(data\_frame, column\_name):  
 *"""This method is used to impute column using K nearest neighbours"""* **try**:  
 original\_column\_seq = list(data\_frame.columns.values)  
 data\_frame = data\_frame.reset\_index()  
 temp\_list = data\_frame.loc[0:1000, column\_name].dropna(axis=0)  
 dtype = **''  
 if** list(temp\_list) == [int(x) **for** x **in** list(temp\_list)]:  
 dtype = **'int64'** knn\_value = int(data\_frame.shape[0] \* 0.10)  
 df\_numeric = data\_frame[column\_name].to\_frame().values  
 **if** knn\_value == 0:  
 knn\_value = 5  
 **try**:  
 knn\_result = fast\_knn(df\_numeric, k=knn\_value)  
 **except** IndexError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 knn\_value = 2  
 knn\_result = fast\_knn(df\_numeric, k=knn\_value)  
 temp\_df = pd.DataFrame.from\_records(knn\_result)  
 temp\_df.columns = [**'imputed\_column'**]  
 return\_df = data\_frame.join(temp\_df, how=**'inner'**)  
 return\_df = return\_df.drop(columns=[column\_name])  
 return\_df.rename(index=str, columns={**"imputed\_column"**: column\_name}, inplace=True)  
 **if** dtype == **'int64'**:  
 return\_df[column\_name] = return\_df[column\_name].astype(int)  
 return\_df = return\_df.loc[:, original\_column\_seq]  
 **return** return\_df  
 **except** ValueError **as** ex:  
 **if "invalid literal for int" in** str(ex.args[0]):  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **else**:  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **except** Exception **as** ex:  
 **if** (**"Data is not float" in** str(ex.args[0])) **or** (  
 **"No Nan's in given data" in** str(ex.args[0])) **or** (  
 **"No NaN's in given data" in** str(ex.args[0])):  
 **return** data\_frame  
 **else**:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame): *# pylint: disable-msg=too-many-branches  
 """  
 This method will perform imputations based on input.* **:param** *data\_frame: required input dataframe* **:return***: dataframe  
 """* **try**:  
 *# checking column dtype* column\_dtype\_obj = False  
 column\_dtype\_date = False  
 **if** data\_frame[self.column\_name].dtype == **'datetime64[ns]'**:  
 column\_dtype\_date = True  
 **if** is\_string\_dtype(data\_frame[self.column\_name]):  
 column\_dtype\_obj = True  
 *# remove rows with empty/Null/NaN values for given column* **if** self.imputation\_type **in** [constants.REMOVE\_ROWS]:  
 **if** column\_dtype\_obj:  
 data\_frame = data\_frame[(data\_frame[self.column\_name].notnull()) &  
 (data\_frame[self.column\_name] != **''**)]  
 **else**:  
 data\_frame = data\_frame[data\_frame[self.column\_name].notnull()]  
 **elif** self.imputation\_type **in** [constants.REPLACE\_WITH]:  
 **if** self.imputation\_method **in** [constants.REPLACE\_WITH\_COLUMN]:  
 **if** self.replace\_to\_column **not in** data\_frame.columns:  
 **raise** Exception(message\_constants.COLUMN\_NOT\_FOUND.format(self.column\_name))  
 **elif** column\_dtype\_date **and** data\_frame[self.replace\_to\_column].dtype\  
 != **'datetime64[ns]'**:  
 **raise** Exception(message\_constants.SELECT\_DATE\_TYPE\_COLUMN)  
 *# marked all empty/None values as NaN* **if** column\_dtype\_obj:  
 data\_frame.loc[  
 data\_frame[self.column\_name] == **''**, self.column\_name] = np.NaN  
 data\_frame[self.column\_name].fillna(data\_frame[self.replace\_to\_column],  
 inplace=True)  
 **elif** self.imputation\_method **in** [constants.KNN]:  
 data\_frame = self.knn\_imputation(data\_frame, self.column\_name)  
 **else**:  
 new\_value = self.\_imputation\_replace\_with(data\_frame, column\_dtype\_obj,  
 column\_dtype\_date)  
 *# replace NaN/Null/None/blank rows with value* data\_frame[self.column\_name].fillna(new\_value, inplace=True)  
 **if** column\_dtype\_obj:  
 data\_frame.loc[  
 data\_frame[self.column\_name] == **''**, self.column\_name] = new\_value  
 **else**:  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** data\_frame  
 **except** Exception **as** ex:  
 **if "No Nan's in given data" in** str(ex.args[0]):  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **else**:  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** get\_input\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find input col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[constants.COLUMN\_NAME]]  
 **except** Exception:  
 **return** []  
  
 @classmethod  
 **def** get\_output\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find output col name keys for current feature* **:return***: list of keys  
 """* **return** []  
  
 @classmethod  
 **def** update\_step\_for\_col\_map(cls, step\_json, column\_mapping\_list):  
 *"""  
 used to update step and col map list wrt step* **:param** *step\_json:* **:param** *column\_mapping\_list:* **:return***: list of step json and updated col mapping list  
 """* updated\_column\_mapping\_list = column\_mapping\_list  
 **try**:  
 input\_cols\_list = Imputations.get\_input\_column\_name\_values(step\_json)  
 **for** col\_obj **in** column\_mapping\_list:  
 **if** col\_obj[**"column\_used"**] **in** input\_cols\_list:  
 step\_json.update({constants.COLUMN\_NAME: col\_obj[**"mapped\_with"**]})  
 **break  
 except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** {**"step\_json"**: step\_json, **"column\_mapping\_list"**: updated\_column\_mapping\_list}

*'''  
This class is used to write test cases for imputations  
'''***from** unittest **import** TestCase  
**import** pandas **as** pd  
  
**from** transformation.imputations.imputations **import** Imputations  
  
  
**class** TestValidation(TestCase):  
 *"""  
 This class is used to write test cases for Imputations  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/test\_imputations.csv"**, delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_get\_result\_remove\_rows(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print("-----------------------------Original---test\_get\_result\_remove\_rows")  
 # print(self.data\_frame)  
 # test case for remove rows* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"remove\_rows"** }  
 remove\_rows = Imputations(json\_request)  
 result\_df = remove\_rows.get\_result(self.data\_frame)  
 *# print("-----------------------------result-----test\_get\_result\_remove\_rows")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_replace\_with\_other\_column(self):  
 *"""  
 This function is used to replace null/empty values of one column with other* **:return***: dataframe  
 """  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"replace\_with\_column"**,  
 **"replace\_to\_column"**: **"pity"** }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_exception\_impute\_method(self):  
 *"""  
 Invalid Imputation Method* **:return***: dataframe  
 """  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"random"**,  
 **"replace\_to\_column"**: **"city"** }  
 **with** self.assertRaises(Exception):  
 Imputations(json\_request)  
  
 **def** test\_get\_result\_replace\_with\_other\_column(self):  
 *"""  
 This function is used to replace null/empty values of one column with other* **:return***: dataframe  
 """  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"random"**,  
 **"imputation\_method"**: **"replace\_with\_column"**,  
 **"replace\_to\_column"**: **"pity"** }  
 **with** self.assertRaises(Exception):  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 **pass  
  
 def** test\_get\_result\_replace\_with\_zero(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print("-----------------------------Original-----test\_get\_result\_replace\_with\_zero")  
 # print(self.data\_frame)  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"zero"** }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 *# print("-----------------------------result----test\_get\_result\_replace\_with\_zero")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_replace\_with\_default(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print("-----------------------------Original-----test\_get\_result\_replace\_with\_default")  
 # print(self.data\_frame)  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"default\_replace"**,  
 **"default\_value"**: 123  
 }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 *# print("-----------------------------result---test\_get\_result\_replace\_with\_default")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_replace\_with\_min(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print("-----------------------------Original-----test\_get\_result\_replace\_with\_min")  
 # print(self.data\_frame)  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"min"** }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 *# print("-----------------------------result---test\_get\_result\_replace\_with\_min")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_replace\_with\_max(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print("-----------------------------Original-----test\_get\_result\_replace\_with\_max")  
 # print(self.data\_frame)  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"max"** }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 *# print("-----------------------------result---test\_get\_result\_replace\_with\_max")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_replace\_with\_sum(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print("-----------------------------Original-----test\_get\_result\_replace\_with\_sum")  
 # print(self.data\_frame)  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"sum"** }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 *# print("-----------------------------result---test\_get\_result\_replace\_with\_sum")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_replace\_with\_median(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print("-----------------------------Original-----test\_get\_result\_replace\_with\_median")  
 # print(self.data\_frame)  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"median"** }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 *# print("-----------------------------result---test\_get\_result\_replace\_with\_median")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_replace\_with\_mode(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print("-----------------------------Original-----test\_get\_result\_replace\_with\_mode")  
 # print(self.data\_frame)  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"mode"** }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 *# print("-----------------------------result---test\_get\_result\_replace\_with\_mode")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_replace\_with\_mode\_for\_obj(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print(self.data\_frame)  
 # test case for replace rows with zero* json\_request = {  
 **"columnName"**: **"city"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"mode"** }  
 replace\_with\_zero = Imputations(json\_request)  
 result\_df = replace\_with\_zero.get\_result(self.data\_frame)  
 *# print("-----------------------------result---test\_get\_result\_replace\_with\_mode\_for\_obj")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_get\_result\_knn(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """  
 # print(self.data\_frame)  
 # test case for replace rows using KNN* json\_request = {  
 **"columnName"**: **"innings"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"knn"** }  
 replace\_with\_knn = Imputations(json\_request)  
 result\_df = replace\_with\_knn.get\_result(self.data\_frame)  
 *# print("-----------------------------result---test\_get\_result\_replace\_with\_mode\_for\_obj")  
 # print(result\_df, "\n\n\n\n")* **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **with** self.assertRaises(Exception):  
 replace\_with\_knn.get\_result(None)  
  
 **with** self.assertRaises(Exception):  
 Imputations(None)  
  
 **def** test\_date\_replace\_existing\_column(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """* self.data\_frame[**'day\_played'**] = pd.to\_datetime(self.data\_frame[**'day\_played'**])  
 json\_request = {  
 **"columnName"**: **"day\_played"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"replace\_with\_column"**,  
 **"replace\_to\_column"**: **"city"** }  
 **try**:  
 replace\_with\_column = Imputations(json\_request)  
 result\_df = replace\_with\_column.get\_result(self.data\_frame)  
 **if** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_date\_replace\_unexisting\_column(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """* self.data\_frame[**'day\_played'**] = pd.to\_datetime(self.data\_frame[**'day\_played'**])  
 json\_request = {  
 **"columnName"**: **"day\_played"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"replace\_with\_column"**,  
 **"replace\_to\_column"**: **"aaa"** }  
 **try**:  
 replace\_with\_column = Imputations(json\_request)  
 result\_df = replace\_with\_column.get\_result(self.data\_frame)  
 **if** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_date\_replace\_default(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """* self.data\_frame[**'day\_played'**] = pd.to\_datetime(self.data\_frame[**'day\_played'**])  
 json\_request = {  
 **"columnName"**: **"day\_played"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"default\_replace"**,  
 **"default\_value"**: 123  
 }  
 **try**:  
 replace\_with\_default = Imputations(json\_request)  
 result\_df = replace\_with\_default.get\_result(self.data\_frame)  
 **if** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_date\_replace\_default\_with\_numeric(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """* self.data\_frame[**'day\_played'**] = pd.to\_datetime(self.data\_frame[**'day\_played'**])  
 json\_request = {  
 **"columnName"**: **"day\_played"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"default\_replace"**,  
 **"default\_value"**: 123  
 }  
 **try**:  
 replace\_with\_default = Imputations(json\_request)  
 result\_df = replace\_with\_default.get\_result(self.data\_frame)  
 **if** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_date\_replace\_default\_with\_date(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """* self.data\_frame[**'day\_played'**] = pd.to\_datetime(self.data\_frame[**'day\_played'**])  
 json\_request = {  
 **"columnName"**: **"day\_played"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"default\_replace"**,  
 **"default\_value"**: **"2018/05/16"** }  
 **try**:  
 replace\_with\_default = Imputations(json\_request)  
 result\_df = replace\_with\_default.get\_result(self.data\_frame)  
 **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 2)  
  
 **def** test\_date\_replace\_default\_with\_string(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """* self.data\_frame[**'day\_played'**] = pd.to\_datetime(self.data\_frame[**'day\_played'**])  
 json\_request = {  
 **"columnName"**: **"day\_played"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"default\_replace"**,  
 **"default\_value"**: **"hello"** }  
 **try**:  
 replace\_with\_default = Imputations(json\_request)  
 result\_df = replace\_with\_default.get\_result(self.data\_frame)  
 **if** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_update\_step\_for\_col\_map(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* steps = {}  
  
 column\_mapping\_list = [{**'column\_used'**: **'outsidecolumn'**, **'mapped\_with'**: **'innings\_new'**}]  
 output\_json = Imputations.update\_step\_for\_col\_map(steps, column\_mapping\_list)  
 self.assertEqual(column\_mapping\_list, output\_json[**'column\_mapping\_list'**])  
  
 **def** test\_update\_step\_for\_col\_map\_error(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* self.assertEqual(Imputations.update\_step\_for\_col\_map(None, None),  
 {**'column\_mapping\_list'**: None, **'step\_json'**: None})  
  
 **def** test\_get\_output\_column\_name\_values(self):  
 self.assertEqual(Imputations.get\_output\_column\_name\_values(None), [])  
  
 **def** test\_get\_result\_replace\_with\_mode\_maximum\_blank\_values(self):  
 *"""  
 This function is used to write the test cases for Imputations* **:return***:  
 """* json\_request = {  
 **"columnName"**: **"names"**,  
 **"imputation\_type"**: **"replace\_with"**,  
 **"imputation\_method"**: **"mode"** }  
 replace\_with\_mode = Imputations(json\_request)  
 dict = {**"id"**: [1, 2, 3, 4, 5, 6, 7, 8], **"names"**: [**'aa'**, **'aa'**, **''**, **'dd'**, **''**, **''**, **'gg'**, **''**],  
 **"pid"**: [100, 100, None, None, 103,None, 105, 108]}  
 df1 = pd.DataFrame(dict)  
 result\_df = replace\_with\_mode.get\_result(df1)  
 **if not** result\_df[json\_request[**"columnName"**]].isnull().any().any():  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)

*Validations*

*'''  
This class is used to write test cases for validations  
'''***from** unittest **import** TestCase  
**import** pandas **as** pd  
**from** transformation.validations.validations **import** Validation  
  
  
**class** TestValidation(TestCase):  
 *"""  
 This class is used to write test cases for validation functions  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/deliveries.csv"**, delimiter=**","**, header=0)  
 self.data\_frame\_1 = pd.read\_csv(**"../resources/test\_validation\_functions.csv"**,  
 delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_validation\_baseclass(self):  
 *# key error* **with** self.assertRaises(Exception):  
 json\_request = {}  
 Validation(json\_request)  
  
 *# passing none as json* **with** self.assertRaises(Exception):  
 Validation(None)  
  
 **def** test\_is\_equal(self):  
 *"""  
 Test case for validating is\_equal method* **:return***:  
 # """  
 # add new column for matching values in column for find\_value* json\_request = {**"columnName"**: **"inning"**, **"find\_value"**: **"1"**, **"case"**: False,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_equal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_equal'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(70755, count)  
  
 *# keep only matching values in column for find\_value* json\_request = {**"columnName"**: **"inning"**, **"find\_value"**: **"1"**, **"case"**: False,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_equal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(70755, count)  
 *# remove matching values in column for find\_value* json\_request = {**"columnName"**: **"inning"**, **"find\_value"**: **"1"**, **"case"**: False,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_equal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(65843, count)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"Lenovo"**, **"case"**: True,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_equal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_equal'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(1, count)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"Lenovo"**, **"case"**: False,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_equal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_equal'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(2, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_length(self):  
 *"""Test case for validating is\_length method* **:return** *"""* json\_request = {**"columnName"**: **"batting\_team"**, **"find\_value"**: **"21"**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_length"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_length'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(15463, count)  
  
 json\_request = {**"columnName"**: **"batting\_team"**, **"find\_value"**: **"21"**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_length"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(15463, count)  
  
 json\_request = {**"columnName"**: **"batting\_team"**, **"find\_value"**: **"21"**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_length"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(121135, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_integer(self):  
 *"""Test case for validating is\_integer method* **:return** *"""* json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_integer"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_integer'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(24, count)  
  
 json\_request = {**"columnName"**: **"index"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_integer"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_integer'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(0, count)  
 json\_request = {**"columnName"**: **"float\_values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_integer"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(2, count)  
  
 json\_request = {**"columnName"**: **"float\_values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_integer"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(22, count)  
  
 *# To check if strings values contains integers* json\_request = {**"columnName"**: **"string\_int"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_integer"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_integer'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(19, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_decimal(self):  
 *"""Test case for validating is\_decimal method* **:return** *"""* json\_request = {**"columnName"**: **"digits"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_decimal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_decimal\_num'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(23, count)  
  
 json\_request = {**"columnName"**: **"digits"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_decimal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(23, count)  
  
 json\_request = {**"columnName"**: **"digits"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_decimal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(1, count)  
  
 *# To check if strings values contains decimals* json\_request = {**"columnName"**: **"string\_decimal"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_decimal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_decimal\_num'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(2, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_alpha(self):  
 *"""Test case for validating is\_alpha method* **:return** *"""* json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_alpha"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_alpha'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(0, count)  
  
 json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_alpha"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(0, count)  
  
 json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_alpha"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(136598, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_alphanumeric(self):  
 *"""Test case for validating is\_alphanumeric method* **:return** *"""* json\_request = {**"columnName"**: **"alphanum"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_alphaNum"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_alpha\_numeric'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(20, count)  
  
 json\_request = {**"columnName"**: **"alphanum"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_alphaNum"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(20, count)  
  
 json\_request = {**"columnName"**: **"alphanum"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_alphaNum"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(4, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_blank(self):  
 *"""Test case for validating is\_blank method* **:return** *"""* json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_blank"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_blank'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(0, count)  
  
 json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_blank"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(0, count)  
  
 json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_blank"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(136598, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_positive(self):  
 *"""Test case for validating is\_postive method. Only works on numeric values* **:return** *"""* json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_positive"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_positive'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(24, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_positive"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(24, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_positive"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(0, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **with** self.assertRaises(Exception):  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_positive"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
  
 *#* **def** test\_is\_string(self):  
 *"""  
 Test case for validating is\_string method* **:return** *"""* json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_string"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_string'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(0, count)  
  
 json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_string"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(0, count)  
  
 json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_string"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(136598, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_startswith(self):  
 *"""  
 Test case for validating startswith method* **:return** *"""* json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"Len"**, **"case"**: True,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"startswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_starts\_with'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(1, count)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"Len"**, **"case"**: True,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"startswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(1, count)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"Len"**, **"case"**: True,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"startswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(23, count)  
  
 **with** self.assertRaises(Exception) **as** context:  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**, **"case"**: True,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"startswith"**}  
 obj = Validation(json\_request)  
 obj.get\_result(self.data\_frame\_1)  
 self.assertTrue(**'Please select a text column' in** context.exception)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"Len"**, **"case"**: False,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"startswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_starts\_with'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(2, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_endswith(self):  
 *"""  
 Test case for validating endswith method* **:return** *"""* json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"i"**, **"case"**: True,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"endswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_ends\_with'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(2, count)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"i"**, **"case"**: True,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"endswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(2, count)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"i"**, **"case"**: True,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"endswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(22, count)  
  
 **with** self.assertRaises(Exception) **as** context:  
 json\_request = {**"columnName"**: **"ball"**, **"find\_value"**: **""**, **"case"**: True,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"endswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame)  
 self.assertTrue(**'Please select a text column' in** context.exception)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"i"**, **"case"**: False,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"endswith"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_ends\_with'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(4, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_contains(self):  
 *"""  
 Test case for validating contains method* **:return** *"""* json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"nNai"**, **"case"**: True,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"contains"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_contains'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(1, count)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"nNai"**, **"case"**: True,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"contains"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(1, count)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"nNai"**, **"case"**: True,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"contains"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(23, count)  
  
 **with** self.assertRaises(Exception) **as** context:  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**, **"case"**: True,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"contains"**}  
 obj = Validation(json\_request)  
 obj.get\_result(self.data\_frame\_1)  
  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"nNai"**, **"case"**: False,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"contains"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_contains'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(3, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_usuallyhighernumber(self):  
 *"""  
 Test case for validating usuallyhighernumber method* **:return** *"""* json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_usuallyhigher"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_usually\_higher'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(1, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_usuallyhigher"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(1, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_usuallyhigher"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(23, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **with** self.assertRaises(Exception):  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_usuallyhigher"**}  
 obj = Validation(json\_request)  
 obj.get\_result(self.data\_frame\_1)  
  
 **def** test\_is\_usuallylowernumber(self):  
 *"""  
 Test case for validating usuallylowernumber method* **:return** *"""* json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_usuallylower"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_usually\_lower'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(1, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_usuallylower"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(1, count)  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_usuallylower"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(23, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **with** self.assertRaises(Exception):  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_usuallylower"**}  
 obj = Validation(json\_request)  
 obj.get\_result(self.data\_frame\_1)  
  
 **def** test\_is\_greater\_or\_equal(self):  
 *"""Test case for validating is\_greater\_or\_equal method* **:return** *"""* json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **"-1.1"**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_greaterorequal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_is\_greater\_or\_equal'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(24, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **"-1"**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_greaterorequal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(24, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **"-1"**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_greaterorequal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(0, count)  
  
 *# no dataframe* **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 *# string column raises error* **with** self.assertRaises(Exception):  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"-1"**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_greaterorequal"**}  
 obj = Validation(json\_request)  
 obj.get\_result(self.data\_frame\_1)  
  
 *# numeric column with string as search param* **with** self.assertRaises(Exception):  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **"bla"**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_greaterorequal"**}  
 obj = Validation(json\_request)  
 obj.get\_result(self.data\_frame\_1)  
  
 **def** test\_is\_less\_than\_or\_equal(self):  
 *"""Test case for validating is\_less\_than\_or\_equal method* **:return** *"""* json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **"55"**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_lessthanorequal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
  
 column\_name = json\_request[**"columnName"**] + **'\_is\_less\_than\_or\_equal'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(22, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **"55"**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_lessthanorequal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(22, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **"55"**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_lessthanorequal"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(2, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 *# string column raises error* **with** self.assertRaises(Exception):  
 json\_request = {**"columnName"**: **"string"**, **"find\_value"**: **"-1"**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_lessthanorequal"**}  
 obj = Validation(json\_request)  
 obj.get\_result(self.data\_frame\_1)  
  
 *# numeric column with string as search param* **with** self.assertRaises(Exception):  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **"bla"**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_lessthanorequal"**}  
 obj = Validation(json\_request)  
 obj.get\_result(self.data\_frame\_1)  
  
 **def** test\_is\_lessfrequent(self):  
 *"""Test case for validating is\_lessfrequent method* **:return** *"""* json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_lessfrequent"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
  
 column\_name = json\_request[**"columnName"**] + **'\_islessfrequent'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(9, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_lessfrequent"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(9, count)  
  
 json\_request = {**"columnName"**: **"values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**, **"validation\_function"**: **"is\_lessfrequent"**}  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(15, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)  
  
 **def** test\_is\_unusually\_longer(self):  
 *"""  
 Test case for validating unusually\_longer method* **:return** *"""* json\_request = {**"columnName"**: **"string\_values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"new\_column"**, **"validation\_function"**: **"is\_unsuallylonger"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 column\_name = json\_request[**"columnName"**] + **'\_isunusuallylonger'** count = len(result\_df.loc[result\_df[column\_name] == **'True'**])  
 self.assertEqual(2, count)  
  
 json\_request = {**"columnName"**: **"string\_values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_valid"**, **"validation\_function"**: **"is\_unsuallylonger"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(2, count)  
 json\_request = {**"columnName"**: **"string\_values"**, **"find\_value"**: **""**,  
 **"output\_options"**: **"keep\_invalid"**,  
 **"validation\_function"**: **"is\_unsuallylonger"**}  
  
 obj = Validation(json\_request)  
 result\_df = obj.get\_result(self.data\_frame\_1)  
 count = len(result\_df[json\_request[**"columnName"**]])  
 self.assertEqual(22, count)  
  
 **with** self.assertRaises(Exception):  
 obj = Validation(json\_request)  
 obj.get\_result(None)

*"""  
This module contains all the validation function constants  
"""*IS\_ALPHA = **'is\_alpha'**IS\_ALPHANUM = **'is\_alphaNum'**IS\_BLANK = **'is\_blank'**IS\_DECIMAL = **'is\_decimal'**IS\_POSITIVE = **'is\_positive'**IS\_USUALLYHIGHER = **'is\_usuallyhigher'**IS\_USUALLYLOWER = **'is\_usuallylower'**IS\_UNUSUALLYLONGER = **'is\_unsuallylonger'**IS\_LESSFREQUENT = **'is\_lessfrequent'**IS\_INTEGER = **'is\_integer'**CONTAINS = **'contains'**ENDSWITH = **'endswith'**STARTSWITH = **'startswith'**IS\_GREATEROREQUAL = **'is\_greaterorequal'**IS\_LESSTHANOREQUAL = **'is\_lessthanorequal'**IS\_STRING = **'is\_string'**IS\_LENGTH = **'is\_length'**IS\_EQUAL = **'is\_equal'**

*"""  
This module contains the logic behind each validation function  
  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** six  
**import** pandas **as** pd  
**import** numpy **as** np  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** ValidationFunctionDriver:  
 *"""  
 This class contains the logic behind each validation function  
 """* @classmethod  
 **def** is\_alpha(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input value is alphabetic* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 data\_frame[column\_name + **'\_is\_alpha'**] = \  
 data\_frame[column\_name].astype(str).str.replace(**' '**, **''**)  
 data\_frame[column\_name + **'\_is\_alpha'**] = \  
 data\_frame[column\_name + **'\_is\_alpha'**].str.isalpha()  
  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_alpha'**] = data\_frame[column\_name + **'\_is\_alpha'**].\  
 astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_alpha'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_alpha'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_alpha'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_alpha'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_alphanumeric(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input value is alphanumeric* **:param** *data\_frame:* **:param** *column\_name:* **:param** *output\_options:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 list1 = []  
 **for** val **in** data\_frame[column\_name].astype(str):  
 **if** val.isalnum():  
 **if not** val.isnumeric() **and not** val.isalpha():  
 list1.append(True)  
 **else**:  
 list1.append(False)  
 **else**:  
 list1.append(False)  
 data\_frame[column\_name + **'\_is\_alpha\_numeric'**] = list1  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_alpha\_numeric'**] = \  
 data\_frame[column\_name + **'\_is\_alpha\_numeric'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = \  
 data\_frame.loc[data\_frame[column\_name + **'\_is\_alpha\_numeric'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_alpha\_numeric'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = \  
 data\_frame.loc[~data\_frame[column\_name + **'\_is\_alpha\_numeric'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_alpha\_numeric'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_blank(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input is a blank value* **:param** *data\_frame:* **:param** *column\_name:* **:param** *output\_options:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 data\_frame[column\_name] = data\_frame[column\_name].replace(**''**, np.nan)  
 data\_frame[column\_name + **'\_is\_blank'**] = data\_frame[column\_name].isna()  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_blank'**] = data\_frame[column\_name + **'\_is\_blank'**].\  
 astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_blank'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_blank'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_blank'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_blank'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** contains(cls, data\_frame, column\_name, output\_options, \_find\_value=None, case=False):  
 *"""  
 This function validates if the input data contains a check value* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **if** data\_frame[column\_name].dtype != **'object'**:  
 **raise** Exception(message\_constants.SELECT\_TEXT\_COLUMN)  
 temp\_df = pd.DataFrame()  
 data\_frame[column\_name + **'\_contains'**] = \  
 data\_frame[column\_name].astype(str).str.contains(\_find\_value, case, regex=False)  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_contains'**] = data\_frame[column\_name + **'\_contains'**].\  
 astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_contains'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_contains'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_contains'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_contains'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** endswith(cls, data\_frame, column\_name, output\_options, \_find\_value=None, case=False):  
 *"""  
 This function validates if the input value ends with a specific value* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **if** data\_frame[column\_name].dtype != **'object'**:  
 **raise** Exception(message\_constants.SELECT\_TEXT\_COLUMN)  
 temp\_df = pd.DataFrame()  
 **if** case:  
 data\_frame[column\_name + **'\_ends\_with'**] = data\_frame[column\_name]. \  
 astype(str).str.endswith(\_find\_value)  
 **else**:  
 data\_frame[column\_name + **'\_ends\_with'**] = \  
 [val.lower().endswith(\_find\_value.lower())  
 **for** val **in** data\_frame[column\_name].astype(str)]  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_ends\_with'**] = data\_frame[column\_name + **'\_ends\_with'**].\  
 astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_ends\_with'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_ends\_with'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_ends\_with'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_ends\_with'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** startswith(cls, data\_frame, column\_name, output\_options, \_find\_value=None, case=False):  
 *"""  
 This function validates if the input value starts with a specific value* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **if** data\_frame[column\_name].dtype != **'object'**:  
 **raise** Exception(message\_constants.SELECT\_TEXT\_COLUMN)  
 temp\_df = pd.DataFrame()  
 **if** case:  
 data\_frame[column\_name + **'\_starts\_with'**] = \  
 data\_frame[column\_name].astype(str).str.startswith(\_find\_value)  
 **else**:  
 data\_frame[column\_name + **'\_starts\_with'**] = \  
 [val.lower().startswith(\_find\_value.lower())  
 **for** val **in** data\_frame[column\_name].astype(str)]  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_starts\_with'**] = \  
 data\_frame[column\_name + **'\_starts\_with'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_starts\_with'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_starts\_with'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_starts\_with'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_starts\_with'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_greater\_or\_equal(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input value is greater or equal to a specific value* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 numerics = [**'int16'**, **'int32'**, **'int64'**, **'float16'**, **'float32'**, **'float64'**]  
 **if** data\_frame[column\_name].dtype **in** numerics:  
 **try**:  
 data\_frame[column\_name + **'\_is\_greater\_or\_equal'**] = \  
 [val >= float(\_find\_value) **for** val **in** data\_frame[column\_name]]  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INVALID\_COMPARISION\_TYPE)  
 **else**:  
 **raise** Exception(message\_constants.SELECT\_NUMERIC\_COLUMN)  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_greater\_or\_equal'**] = \  
 data\_frame[column\_name + **'\_is\_greater\_or\_equal'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_greater\_or\_equal'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_greater\_or\_equal'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_greater\_or\_equal'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_greater\_or\_equal'**], axis=1)  
 **return** temp\_df  
 **except** ValueError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_less\_than\_or\_equal(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input value is less than or equal to a specific value* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 numerics = [**'int16'**, **'int32'**, **'int64'**, **'float16'**, **'float32'**, **'float64'**]  
 **if** data\_frame[column\_name].dtype **in** numerics:  
 **try**:  
 data\_frame[column\_name + **'\_is\_less\_than\_or\_equal'**] = \  
 [val <= float(\_find\_value) **for** val **in** data\_frame[column\_name]]  
 **except** TypeError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INVALID\_COMPARISION\_TYPE)  
 **else**:  
 **raise** Exception(message\_constants.SELECT\_NUMERIC\_COLUMN)  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_less\_than\_or\_equal'**] = \  
 data\_frame[column\_name + **'\_is\_less\_than\_or\_equal'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_less\_than\_or\_equal'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_less\_than\_or\_equal'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_less\_than\_or\_equal'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_less\_than\_or\_equal'**], axis=1)  
 **return** temp\_df  
 **except** ValueError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_length(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input value is of given length* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 data\_frame[column\_name + **'\_is\_length'**] = \  
 [len(val) == int(\_find\_value) **for** val **in** data\_frame[column\_name].astype(str)]  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_length'**] = data\_frame[column\_name + **'\_is\_length'**].\  
 astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_length'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_length'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_length'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_length'**], axis=1)  
 **return** temp\_df  
 **except** ValueError **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.INPUT\_SHOULD\_BE\_NUMERIC)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_equal(cls, data\_frame, column\_name, output\_options, \_find\_value=None, case=False):  
 *"""  
 This function validates if the input value equals a comparison value* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 \_find\_value = str(\_find\_value)  
 **if** case:  
 data\_frame[column\_name + **'\_is\_equal'**] = \  
 [val == \_find\_value  
 **for** val **in** data\_frame[column\_name].astype(str)]  
 **else**:  
 data\_frame[column\_name + **'\_is\_equal'**] = \  
 [val.lower() == \_find\_value.lower()  
 **for** val **in** data\_frame[column\_name].astype(str)]  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_equal'**] = data\_frame[column\_name + **'\_is\_equal'**].\  
 astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_equal'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_equal'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_equal'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_equal'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_integer(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *# pylint: disable-msg=R0912  
 """  
 This function validates if the input value is Integer* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 **if** data\_frame[column\_name].dtype == **'object'**:  
 list1 = []  
 **for** val **in** data\_frame[column\_name].astype(str):  
 **if** val.isnumeric():  
 list1.append(True)  
 **elif** val.startswith((**'+'**, **'-'**)) **and** val[1:].isnumeric():  
 list1.append(True)  
 **else**:  
 list1.append(False)  
 data\_frame[column\_name + **'\_is\_integer'**] = list1  
 **elif** data\_frame[column\_name].dtype **in** [**'float16'**, **'float32'**, **'float64'**]:  
 list1 = []  
 **for** val **in** data\_frame[column\_name]:  
 float\_val = float(val)  
 **try**:  
 int\_val = int(val)  
 **if** float\_val == int\_val:  
 list1.append(True)  
 **else**:  
 list1.append(False)  
 **except** ValueError:  
 list1.append(False)  
 data\_frame[column\_name + **'\_is\_integer'**] = list1  
 **else**:  
 data\_frame[column\_name + **'\_is\_integer'**] = \  
 [isinstance(val, int) **for** val **in** data\_frame[column\_name]]  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_integer'**] = data\_frame[column\_name + **'\_is\_integer'**].\  
 astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_integer'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_integer'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_integer'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_integer'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_decimal(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input value is a decimal number* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 **if** data\_frame[column\_name].dtype == **'object'**:  
 list1 = []  
 **for** val **in** data\_frame[column\_name].astype(str):  
 **if** val.count(**'.'**) != 1:  
 list1.append(False)  
 **elif** val.replace(**'.'**, **''**, 1).isdigit():  
 list1.append(True)  
 **elif** val.startswith((**'+'**, **'-'**)) **and** val[1:].replace(**'.'**, **''**, 1).isdigit():  
 list1.append(True)  
 **else**:  
 list1.append(False)  
 data\_frame[column\_name + **'\_is\_decimal\_num'**] = list1  
 **else**:  
 data\_frame[column\_name + **'\_is\_decimal\_num'**] = \  
 [isinstance(val, float) **and not** np.isnan(val) **for** val **in** data\_frame[column\_name]]  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_decimal\_num'**] = \  
 data\_frame[column\_name + **'\_is\_decimal\_num'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_decimal\_num'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_decimal\_num'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_decimal\_num'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_decimal\_num'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_string(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input value is String* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 data\_frame[column\_name + **'\_is\_string'**] = \  
 [isinstance(x, six.string\_types) **for** x **in** data\_frame[column\_name]]  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_string'**] = data\_frame[column\_name + **'\_is\_string'**].\  
 astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_string'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_string'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_string'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_string'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_positive(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *"""  
 This function validates if the input is a positive number* **:param** *data\_frame:* **:param** *column\_name:* **:param** *output\_options:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 **if** data\_frame[column\_name].dtype == **'object'**:  
 **raise** Exception(message\_constants.SELECT\_NUMERIC\_COLUMN)  
 **else**:  
 **try**:  
 data\_frame[column\_name + **'\_is\_positive'**] = \  
 [val > 0 **for** val **in** data\_frame[column\_name]]  
 temp\_df = data\_frame  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_positive'**] = \  
 data\_frame[column\_name + **'\_is\_positive'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_positive'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_positive'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_positive'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_positive'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_usuallyhighernumber(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *# pylint: disable-msg=R1719  
 """  
 This function validates if the input value is a usually higher number from  
 the distribution of column* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 **if** data\_frame[column\_name].dtype == **'object' or** \  
 data\_frame[column\_name].dtype == **'datetime64[ns]'**:  
 **raise** Exception(message\_constants.SELECT\_NUMERIC\_COLUMN)  
 **else**:  
 **try**:  
 outliers = data\_frame[data\_frame[column\_name] >  
 data\_frame[column\_name].mean()  
 + 3 \* data\_frame[column\_name].std()]  
 outliers\_list = outliers[column\_name].tolist()  
 data\_frame[column\_name + **'\_is\_usually\_higher'**] = \  
 [True **if** val **in** outliers\_list **else** False **for** val **in** data\_frame[column\_name]]  
 temp\_df = data\_frame  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_usually\_higher'**] = \  
 data\_frame[column\_name + **'\_is\_usually\_higher'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_usually\_higher'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_usually\_higher'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_usually\_higher'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_usually\_higher'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_usuallylowernumber(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *# pylint: disable-msg=R1719  
 """  
 This function validates if the input value is usually lower numberfrom the distribution  
 of column* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
  
 **if** data\_frame[column\_name].dtype == **'object' or** \  
 data\_frame[column\_name].dtype == **'datetime64[ns]'**:  
 **raise** Exception(message\_constants.SELECT\_NUMERIC\_COLUMN)  
 **else**:  
 **try**:  
 outliers = data\_frame[data\_frame[column\_name] <  
 data\_frame[column\_name].mean()  
 - 3 \* data\_frame[column\_name].std()]  
 outliers\_list = outliers[column\_name].tolist()  
 data\_frame[column\_name + **'\_is\_usually\_lower'**] = \  
 [True **if** val **in** outliers\_list **else** False **for** val **in** data\_frame[column\_name]]  
 temp\_df = data\_frame  
 **except** Exception **as** ex:  
 **raise** Exception(str(ex))  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_is\_usually\_lower'**] = \  
 data\_frame[column\_name + **'\_is\_usually\_lower'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_is\_usually\_lower'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_usually\_lower'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_is\_usually\_lower'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_is\_usually\_lower'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_unusuallylonger(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *# pylint: disable-msg=R1719  
 """  
 This function validates if the input value is unusually longer.* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 data\_frame[**'length\_Column'**] = [len(val) **for** val **in** data\_frame[column\_name].astype(str)]  
 unusually\_longer = data\_frame[  
 data\_frame[**'length\_Column'**] >  
 data\_frame[**'length\_Column'**].mean() + 1.5 \* data\_frame[**'length\_Column'**].std()]  
 unusually\_longer\_list = unusually\_longer[**'length\_Column'**].tolist()  
 data\_frame[column\_name + **'\_isunusuallylonger'**] = \  
 [True **if** val **in** unusually\_longer\_list **else** False **for** val **in** data\_frame[**'length\_Column'**]]  
 temp\_df = data\_frame  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_isunusuallylonger'**] = \  
 data\_frame[column\_name + **'\_isunusuallylonger'**].astype(str)  
 temp\_df = data\_frame.drop([**'length\_Column'**], axis=1)  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_isunusuallylonger'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_isunusuallylonger'**, **'length\_Column'**],  
 axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_isunusuallylonger'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_isunusuallylonger'**, **'length\_Column'**],  
 axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** is\_lessfrequent(cls, data\_frame, column\_name, output\_options, \_find\_value=None):  
 *# pylint: disable-msg=R1719  
 """  
 This function validates if the input value is less frequent.* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 temp\_df = pd.DataFrame()  
 count = data\_frame[column\_name].value\_counts()  
 count\_list = count[count < 0.05 \* count.sum()].index.tolist()  
 data\_frame[column\_name + **'\_islessfrequent'**] = \  
 [True **if** val **in** count\_list **else** False **for** val **in** data\_frame[column\_name]]  
 temp\_df = data\_frame  
 **if** output\_options == **'new\_column'**:  
 data\_frame[column\_name + **'\_islessfrequent'**] = \  
 data\_frame[column\_name + **'\_islessfrequent'**].astype(str)  
 temp\_df = data\_frame  
 **if** output\_options == **'keep\_valid'**:  
 data\_frame = data\_frame.loc[data\_frame[column\_name + **'\_islessfrequent'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_islessfrequent'**], axis=1)  
 **if** output\_options == **'keep\_invalid'**:  
 data\_frame = data\_frame.loc[~data\_frame[column\_name + **'\_islessfrequent'**]]  
 temp\_df = data\_frame.drop([column\_name + **'\_islessfrequent'**], axis=1)  
 **return** temp\_df  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to perform validations on the input data.  
"""***import** constants  
**from** transformation.validations **import** validation\_constants  
**from** transformation.validations.validation\_functions\_driver **import** ValidationFunctionDriver  
  
  
**class** Validation():  
 *"""  
 This class contains all the validation functions that can be performed on the input data.  
 """* validation\_function\_registry = {  
  
 validation\_constants.IS\_ALPHA: ValidationFunctionDriver.is\_alpha,  
 validation\_constants.IS\_ALPHANUM: ValidationFunctionDriver.is\_alphanumeric,  
 validation\_constants.IS\_BLANK: ValidationFunctionDriver.is\_blank,  
 validation\_constants.IS\_DECIMAL: ValidationFunctionDriver.is\_decimal,  
 validation\_constants.IS\_POSITIVE: ValidationFunctionDriver.is\_positive,  
 validation\_constants.IS\_USUALLYHIGHER: ValidationFunctionDriver.is\_usuallyhighernumber,  
 validation\_constants.IS\_USUALLYLOWER: ValidationFunctionDriver.is\_usuallylowernumber,  
 validation\_constants.IS\_UNUSUALLYLONGER: ValidationFunctionDriver.is\_unusuallylonger,  
 validation\_constants.IS\_LESSFREQUENT: ValidationFunctionDriver.is\_lessfrequent,  
 validation\_constants.IS\_INTEGER: ValidationFunctionDriver.is\_integer,  
 validation\_constants.CONTAINS: ValidationFunctionDriver.contains,  
 validation\_constants.ENDSWITH: ValidationFunctionDriver.endswith,  
 validation\_constants.STARTSWITH: ValidationFunctionDriver.startswith,  
 validation\_constants.IS\_GREATEROREQUAL: ValidationFunctionDriver.is\_greater\_or\_equal,  
 validation\_constants.IS\_LESSTHANOREQUAL: ValidationFunctionDriver.is\_less\_than\_or\_equal,  
 validation\_constants.IS\_STRING: ValidationFunctionDriver.is\_string,  
 validation\_constants.IS\_LENGTH: ValidationFunctionDriver.is\_length,  
 validation\_constants.IS\_EQUAL: ValidationFunctionDriver.is\_equal  
  
 }  
  
 **def** \_\_init\_\_(self, json\_value):  
 *"""  
 Initialize the object with a parameter* **:param** *json\_value:  
 """* **try**:  
 self.column\_name = json\_value[constants.COLUMN\_NAME]  
 self.output\_options = json\_value[constants.OUTPUT\_OPTION]  
 self.validation\_function = json\_value[constants.VALIDATION\_FUNCTION]  
 self.find\_value = json\_value[constants.FIND\_VALUE]  
 self.case = json\_value.get(constants.CASE\_SENSITIVE\_STATUS, False)  
 **except** KeyError **as** key\_error:  
 **raise** Exception(key\_error.args[0] + **"column is missing in input json"**)  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])  
  
 **def** get\_result(self, data\_frame):  
 *"""* **:param** *data\_frame:* **:return***: dataframe  
 """* **try**:  
 func = Validation.validation\_function\_registry.get(self.validation\_function)  
 **if** self.validation\_function **not in** [**'contains'**, **'endswith'**, **'startswith'**, **'is\_equal'**]:  
 value = func(data\_frame, self.column\_name, self.output\_options, self.find\_value)  
 **else**:  
 value = func(data\_frame, self.column\_name, self.output\_options, self.find\_value,  
 self.case)  
 **return** value  
 **except** Exception **as** ex:  
 **raise** Exception(ex.args[0])

*Validations*

*"""  
This module is used for string type data  
"""***import** logging  
**import** pandas **as** pd  
**import** constants  
**import** message\_constants  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** ChangeDataType: *# pylint: disable=too-few-public-methods  
 """  
 Change datatype of column. Currently support Date, Numeric and String type  
 """* **def** \_\_init\_\_(self, json\_value):  
 *"""  
 initialize object with parameter* **:param** *json\_value:  
 """* **try**:  
 self.column\_name = json\_value[constants.COLUMN\_NAME]  
 self.column\_type = json\_value[constants.COLUMN\_TYPE]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 Take output column type and column name. Changes the data type of column  
 and returns the result.* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **if** self.column\_type == **"date"**:  
 data\_frame[self.column\_name] = pd.to\_datetime(data\_frame[self.column\_name],  
 errors=**"coerce"**)  
 **elif** self.column\_type == **"numeric"**:  
 data\_frame[self.column\_name] = pd.to\_numeric(data\_frame[self.column\_name])  
 **else**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].astype(str)  
 **return** data\_frame  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** TypeError **as** type\_error:  
 LOGGER.error(**'Exception: %s'**, type\_error)  
 **raise** TypeError(message\_constants.NONE\_INPUT\_DATASET)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used for string type data  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** message\_constants  
**from** mashupbase **import** DataMashUpBase  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** DropColumn(DataMashUpBase): *# pylint: disable=too-few-public-methods  
 """  
 Drop column on DataFrame  
 """* **def** \_\_init\_\_(self, json\_value):  
 *"""  
 initialize object with parameter* **:param** *json\_value:  
 """* **try**:  
 DataMashUpBase.\_\_init\_\_(self, json\_value)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 Drop specified input column and return updated dataframe* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 data\_frame = data\_frame.drop(self.column\_name, axis=1)  
 **return** data\_frame  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** AttributeError **as** attribute\_error:  
 LOGGER.error(**'Exception: %s'**, attribute\_error)  
 **raise** Exception(message\_constants.NONE\_INPUT\_DATASET)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used to apply small transformations on the text type of data columns.  
"""***import** logging  
**from** mashupbase **import** DataMashUpBase  
**import** constants  
**import** message\_constants  
**import** numpy **as** np  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** NumericTransform(DataMashUpBase): *# pylint: disable=too-few-public-methods  
 """  
 Text column basic transformation on DataFrame  
 """* **def** \_\_init\_\_(self, json\_value):  
 *"""  
 initialize object with parameter* **:param** *json\_value:  
 """* **try**:  
 DataMashUpBase.\_\_init\_\_(self, json\_value)  
 self.transform\_function = json\_value[constants.TRANSFORM\_FUNCTION]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 Applies the transformation on text field. Mostly replaces the input column except for Split  
 operation.  
 Currently supports - Upper case, Lower case, Split, Length* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **if** self.transform\_function == **"round"**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].round()  
  
 **elif** self.transform\_function == **"power"**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name]\*data\_frame[  
 self.column\_name]  
  
 **elif** self.transform\_function == **"ceil"**:  
 data\_frame[self.column\_name] = np.ceil(data\_frame[self.column\_name])  
  
 **elif** self.transform\_function == **"floor"**:  
 data\_frame[self.column\_name] = np.floor(data\_frame[self.column\_name])  
  
 **elif** self.transform\_function == **"absolute"**:  
 data\_frame[self.column\_name] = np.abs(data\_frame[self.column\_name])  
  
 **elif** self.transform\_function == **"sqrt"**:  
 data\_frame[self.column\_name] = np.sqrt(data\_frame[self.column\_name])  
  
 **return** data\_frame  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** TypeError **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** TypeError(message\_constants.NONE\_INPUT\_DATASET)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*"""  
This module is used for string type data  
"""***import** logging  
**from** mashupbase **import** DataMashUpBase  
**import** constants  
**import** message\_constants  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** RenameColumn(DataMashUpBase): *# pylint: disable=too-few-public-methods  
 """  
 Rename column  
 """* **def** \_\_init\_\_(self, json\_value):  
 *"""  
 initialize object with parameter* **:param** *json\_value:  
 """* **try**:  
 DataMashUpBase.\_\_init\_\_(self, json\_value)  
 self.to\_column = json\_value[constants.TO\_COLUMN]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 Takes in old and new column name. Renames the column in dataframe and returns the dataframe* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 data\_frame = data\_frame.rename(columns={self.column\_name: self.to\_column})  
 **return** data\_frame  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** get\_input\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find input col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[constants.COLUMN\_NAME]]  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **return** []  
  
 @classmethod  
 **def** get\_output\_column\_name\_values(cls, step\_json):  
 *"""  
 used to find output col name keys for current feature* **:return***: list of keys  
 """* **try**:  
 **return** [step\_json[constants.TO\_COLUMN]]  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **return** []  
  
 @classmethod  
 **def** update\_step\_for\_col\_map(cls, step\_json, column\_mapping\_list):  
 *"""  
 used to update step and col map list wrt step* **:param** *step\_json:* **:param** *column\_mapping\_list:* **:return***: list of step json and updated col mapping list  
 """* updated\_column\_mapping\_list = column\_mapping\_list  
 **try**:  
 input\_cols\_list = RenameColumn.get\_input\_column\_name\_values(step\_json)  
 update\_col\_map\_list\_flag = False  
 **for** col\_obj **in** column\_mapping\_list:  
 **if** col\_obj[**"column\_used"**] **in** input\_cols\_list:  
 step\_json.update({constants.COLUMN\_NAME: col\_obj[**"mapped\_with"**]})  
 update\_col\_map\_list\_flag = True  
 **break  
 if** update\_col\_map\_list\_flag:  
 updated\_column\_mapping\_list = []  
 **for** col\_obj **in** column\_mapping\_list:  
 **if** step\_json[constants.COLUMN\_NAME] == col\_obj[**"mapped\_with"**]:  
 updated\_column\_mapping\_list.append(  
 {**"column\_used"**: col\_obj[**"column\_used"**],  
 **"mapped\_with"**: step\_json[constants.TO\_COLUMN]}  
 )  
 **else**:  
 updated\_column\_mapping\_list.append(  
 {**"column\_used"**: col\_obj[**"column\_used"**],  
 **"mapped\_with"**: col\_obj[**"mapped\_with"**]}  
 )  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 **return** {**"step\_json"**: step\_json, **"column\_mapping\_list"**: updated\_column\_mapping\_list}

*"""  
This is a test case module on TextFilter  
"""***import** unittest  
**import** json  
**import** pandas **as** pd  
  
  
**from** transformation.change\_datatype **import** ChangeDataType  
  
**class** TestChangeDatatype(unittest.TestCase):  
 *"""  
 unit test case for change data type test case  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/match.csv"**, delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_change\_datatype\_to\_date(self):  
 *"""  
 test case for changing the data type to date* **:return***:  
 """* json\_request = **"{\"className\" : \"change\_datatype\",\"columnName\": \"day\_played\","** \  
 **"\"column\_type\": \"date\"}"** change\_datatype = ChangeDataType(json.loads(json\_request))  
  
 result = change\_datatype.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'day\_played'**].dtypes, **"datetime64[ns]"**)  
  
 **def** test\_change\_datatype\_to\_numeric\_int(self):  
 *"""  
 test case for changing the data type to numeric* **:return***:  
 """* json\_request = **"{\"className\" : \"change\_datatype\",\"columnName\": \"id\","** \  
 **"\"column\_type\": \"numeric\"}"** change\_datatype = ChangeDataType(json.loads(json\_request))  
  
 result = change\_datatype.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'id'**].dtypes, **"int64"**)  
  
 **def** test\_change\_datatype\_to\_numeric\_float(self):  
 *"""  
 test case for changing the data type to float* **:return***:  
 """* json\_request = **"{\"className\" : \"change\_datatype\",\"columnName\": \"innings\","** \  
 **"\"column\_type\": \"numeric\"}"** change\_datatype = ChangeDataType(json.loads(json\_request))  
  
 result = change\_datatype.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'innings'**].dtypes, **"float64"**)  
  
 **def** test\_change\_data\_type\_to\_text(self):  
 *"""  
 test case for changing the data type to date* **:return***:  
 """* json\_request = **"{\"className\" : \"change\_datatype\",\"columnName\": \"id"** \  
 **"\",\"column\_type\": \"text\"}"** change\_data\_type = ChangeDataType(json.loads(json\_request))  
 result = change\_data\_type.get\_result(self.data\_frame)  
 self.assertEqual(result[**'id'**].dtypes, **"object"**)  
  
 **def** test\_change\_data\_type\_key\_missing\_input(self):  
 *"""  
 test case for change data type class which will raise an exception when one of the key is  
 missing in input json* **:return***:  
 """* **try**:  
 json\_request = **"{\"className\" : \"change\_datatype\",\"columnName1111\": \"id"** \  
 **"\",\"column\_type\": \"text\"}"** ChangeDataType(json.loads(json\_request))  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_change\_data\_type\_column\_missing(self):  
 *"""  
 test case for change data type class which will raise an exception when column is missing  
 in data\_frame.* **:return***:  
 """* **try**:  
 json\_request = **"{\"className\" : \"change\_datatype\",\"columnName\": \"id\_nww"** \  
 **"\",\"column\_type\": \"text\"}"** change\_data\_type = ChangeDataType(json.loads(json\_request))  
 change\_data\_type.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_change\_data\_type\_data\_frame\_none(self):  
 *"""  
 test case for change data type class which will raise an exception  
 when column is missing in data\_frame.* **:return***:  
 """* **try**:  
 json\_request = **"{\"className\" : \"change\_datatype\",\"columnName\": \"id\_nww"** \  
 **"\",\"column\_type\": \"text\"}"** change\_data\_type = ChangeDataType(json.loads(json\_request))  
 change\_data\_type.get\_result(None)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
This is a test case module on TextFilter  
"""***import** unittest  
**import** json  
**import** pandas **as** pd  
  
**from** transformation.drop\_column **import** DropColumn  
  
  
**class** TestDropColumn(unittest.TestCase):  
 *"""  
 unit test case for drop column  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/deliveries.csv"**, delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_drop\_column(self):  
 *"""  
 test case for test\_drop\_column\_filter* **:return***:  
 """* column\_name = **"inning"** json\_request = **"{\"step\_name\" : \"transformation.drop\_column.DropColumn\",\"columnName\": \"inning\"}"** drop\_column = DropColumn(json.loads(json\_request))  
  
 result = drop\_column.get\_result(self.data\_frame)  
  
 **if** column\_name **in** result.columns:  
 self.assertEqual(1, 2)  
 **else**:  
 self.assertEqual(1, 1)  
  
 **def** test\_drop\_column\_key\_error(self):  
 *"""  
 test case for test\_drop\_column\_filter with column missing error* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.drop\_column.DropColumn\",\"columnName\": \"inning111\"}"  
 try**:  
 drop\_column = DropColumn(json.loads(json\_request))  
 drop\_column.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_drop\_column\_input\_json\_is\_missing(self):  
 *"""  
 test case for test\_drop\_column\_filter with input json key is missing* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.drop\_column.DropColumn\",\"columnName111\": \"inning111\"}"  
 try**:  
 DropColumn(json.loads(json\_request))  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_drop\_column\_data\_frame\_is\_none(self):  
 *"""  
 test case for test\_drop\_column\_filter when input data frame is None* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.drop\_column.DropColumn\",\"columnName\": \"inning\"}"  
 try**:  
 drop\_column = DropColumn(json.loads(json\_request))  
 drop\_column.get\_result(None)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)

*"""  
This is a test case module on TextFilter  
"""***import** unittest  
**import** json  
**import** pandas **as** pd  
**from** transformation.numeric\_column\_transform **import** NumericTransform  
  
  
**class** TestNumericTransformations(unittest.TestCase):  
 *"""  
 unit test case for change data type test case  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/match.csv"**, delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_transform\_numeric\_to\_round(self):  
 *"""  
 Test case for converting numeric column to round* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.numeric\_column\_transform.NumericTransform\","** \  
 **"\"columnName\": \"innings\",\"transform\_func\": \"round\"}"** text\_transform = NumericTransform(json.loads(json\_request))  
  
 result = text\_transform.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'innings'**][0],1)  
  
 **def** test\_transform\_numeric\_to\_power(self):  
 *"""  
 Test case for taking power of a numeric column* **:return***:  
  
 """* json\_request = **"{\"step\_name\" : \"transformation.numeric\_column\_transform.NumericTransform\","** \  
 **"\"columnName\": \"innings\",\"transform\_func\": \"power\"}"** text\_transform = NumericTransform(json.loads(json\_request))  
  
 result = text\_transform.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'innings'**][1],4)  
  
 **def** test\_transform\_numeric\_to\_ceil(self):  
 *"""  
 Test case for taking ciel of a numeric column* **:return***:  
  
 """* json\_request = **"{\"step\_name\" : \"transformation.numeric\_column\_transform.NumericTransform\","** \  
 **"\"columnName\": \"innings\",\"transform\_func\": \"ceil\"}"** text\_transform = NumericTransform(json.loads(json\_request))  
  
 result = text\_transform.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'innings'**][0],2)  
  
  
 **def** test\_transform\_numeric\_to\_floor(self):  
 *"""  
 Test case for taking floor of a numeric column* **:return***:  
  
 """* json\_request = **"{\"step\_name\" : \"transformation.numeric\_column\_transform.NumericTransform\","** \  
 **"\"columnName\": \"innings\",\"transform\_func\": \"floor\"}"** text\_transform = NumericTransform(json.loads(json\_request))  
  
 result = text\_transform.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'innings'**][0],1)  
  
 **def** test\_transform\_numeric\_to\_absolute(self):  
 *"""  
 Test case for taking absolute of a numeric column* **:return***:  
  
 """* json\_request = **"{\"step\_name\" : \"transformation.numeric\_column\_transform.NumericTransform\","** \  
 **"\"columnName\": \"innings\",\"transform\_func\": \"absolute\"}"** text\_transform = NumericTransform(json.loads(json\_request))  
  
 result = text\_transform.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'innings'**][2],1.2)  
  
 **def** test\_transform\_numeric\_to\_sqrt(self):  
 *"""  
 Test case for taking sqrt of a numeric column* **:return***:  
  
 """* json\_request = **"{\"step\_name\" : \"transformation.numeric\_column\_transform.NumericTransform\","** \  
 **"\"columnName\": \"innings\",\"transform\_func\": \"sqrt\"}"** text\_transform = NumericTransform(json.loads(json\_request))  
  
 result = text\_transform.get\_result(self.data\_frame)  
  
 self.assertEqual(result[**'innings'**][3], 2)  
  
 **def** test\_transform\_numeric\_column\_missing(self):  
 *"""  
 Test case for raise column missing exception* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.numeric\_column\_transform.NumericTransform\","** \  
 **"\"columnName\": \"innings111\",\"transform\_func\": \"sqrt\"}"  
  
 try**:  
 text\_transform = NumericTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_transform\_numeric\_input\_set\_is\_none(self):  
 *"""  
 Test case for raise data set is None* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.numeric\_column\_transform.NumericTransform\","** \  
 **"\"columnName\": \"innings111\",\"transform\_func\": \"sqrt\"}"  
  
 try**:  
 text\_transform = NumericTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(None)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*"""  
This is a test case module on TextFilter  
"""***import** unittest  
**import** json  
**import** pandas **as** pd  
  
**from** transformation.rename\_column **import** RenameColumn  
  
  
**class** TestRenameColumn(unittest.TestCase):  
 *"""  
 unit test case for column rename  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/deliveries.csv"**, delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_rename\_column(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* json\_request = **"{\"className\" : \"rename\_column\",\"columnName\": \"inning\","** \  
 **"\"toColumn\": \"innings\"}"** rename\_column = RenameColumn(json.loads(json\_request))  
 result = rename\_column.get\_result(self.data\_frame)  
 column\_name = **"innings"** self.assertTrue(column\_name **in** result.columns)  
  
 **def** test\_rename\_column\_key\_error(self):  
 *"""  
 test case for raise an exception when rename on column is not found in the data frame* **:return***:  
 """* json\_request = **"{\"className\" : \"rename\_column\",\"columnName1111\": \"inning111\","** \  
 **"\"toColumn\": \"innings11212\"}"  
 with** self.assertRaises(Exception):  
 RenameColumn(json.loads(json\_request))  
  
 **def** test\_get\_result\_error(self):  
 *"""  
 test case for raise an exception when rename on column is not found in the data frame* **:return***:  
 """* json\_request = **"{\"className\" : \"rename\_column\",\"columnName\": \"inning\","** \  
 **"\"toColumn\": \"innings\"}"  
 with** self.assertRaises(Exception):  
 rename\_column = RenameColumn(json.loads(json\_request))  
 string = **"test"** rename\_column.get\_result(string)  
  
 **def** test\_get\_output\_column\_name\_values(self):  
 *"""  
 Test case for get\_output\_column\_name\_values error* **:return***:  
 """* self.assertTrue(RenameColumn.get\_output\_column\_name\_values(None) == [])  
  
 **def** test\_update\_step\_for\_col\_map(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* steps = {**"step\_name"**: **"transformation.rename\_column.RenameColumn"**, **"columnName"**: **"inning"**,  
 **"toColumn"**: **"innings\_new"**, **"displayDataHeader"**: **"Rename"**,  
 **"displayDataText"**: **"inning to innings\_new"**,  
 **"is\_enabled"**: **'true'**, **"is\_edited"**: **'true'**}  
  
 column\_mapping\_list = [{**'column\_used'**: **'outsidecolumn'**, **'mapped\_with'**: **'innings\_new'**}]  
 output\_json = RenameColumn.update\_step\_for\_col\_map(steps, column\_mapping\_list)  
 self.assertEqual(column\_mapping\_list, output\_json[**'column\_mapping\_list'**])  
  
 **def** test\_update\_step\_for\_col\_map\_error(self):  
 *"""  
 test case for test\_rename\_column\_filter* **:return***:  
 """* self.assertEqual(RenameColumn.update\_step\_for\_col\_map(None, None),  
 {**'column\_mapping\_list'**: None, **'step\_json'**: None})

*"""  
This is a test case module on TextFilter  
"""***import** unittest  
**import** json  
**import** pandas **as** pd  
  
**from** transformation.text\_column\_transform **import** TextTransform  
  
  
**class** TestTextTransformations(unittest.TestCase):  
 *"""  
 unit test case for change data type test case  
 """* **def** setUp(self):  
 self.data\_frame = pd.read\_csv(**"../resources/match.csv"**, delimiter=**","**, header=0)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_change\_case\_to\_upper(self):  
 *"""  
 test case for uppercase* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.text\_column\_transform.TextTransform\","** \  
 **"\"columnName\": \"team\",\"transform\_func\": \"uppercase\","** \  
 **"\"split\_count\": \"0\"}"** text\_transform = TextTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(self.data\_frame)  
 self.assertEqual(result[**'team'**][0], **"PUNE SUPER GIANTS"**)  
  
 **def** test\_change\_case\_to\_lower(self):  
 *"""  
 test case for lowercase* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.text\_column\_transform.TextTransform\","** \  
 **"\"columnName\": \"team\",\"transform\_func\": \"lowercase\","** \  
 **"\"split\_count\": \"0\"}"** text\_transform = TextTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(self.data\_frame)  
 self.assertEqual(result[**'team'**][0], **"pune super giants"**)  
  
 **def** test\_text\_length(self):  
 *"""  
 test case for lowercase* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.text\_column\_transform.TextTransform\","** \  
 **"\"columnName\": \"team\",\"transform\_func\": \"length\","** \  
 **"\"split\_count\": \"0\"}"** text\_transform = TextTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(self.data\_frame)  
 self.assertEqual(result[**'team'**][0], 17)  
  
 **def** test\_text\_split(self):  
 *"""  
 test case for lowercase* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.text\_column\_transform.TextTransform\","** \  
 **"\"columnName\": \"team\",\"transform\_func\": \"split\","** \  
 **"\"split\_count\": \"2\"}"** text\_transform = TextTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(self.data\_frame)  
 self.assertEqual(result[**'team\_1'**][0], **"Pune"**)  
  
 **def** test\_text\_trim(self):  
 *"""  
 test case for trim the data\_set* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.text\_column\_transform.TextTransform\","** \  
 **"\"columnName\": \"team\",\"transform\_func\": \"trim\","** \  
 **"\"split\_count\": \"2\"}"** text\_transform = TextTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(self.data\_frame)  
 self.assertEqual(result[**'team'**][0], **"Pune super giants"**)  
  
 **def** test\_text\_reverse(self):  
 *"""  
 test case for reverse the data\_set* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.text\_column\_transform.TextTransform\","** \  
 **"\"columnName\": \"team\",\"transform\_func\": \"reverse\","** \  
 **"\"split\_count\": \"2\"}"** text\_transform = TextTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(self.data\_frame)  
 self.assertEqual(result[**'team'**][0], **"stnaig repus enuP"**)  
  
 **def** test\_text\_capitalize(self):  
 *"""  
 test case for reverse the data\_set* **:return***:  
 """* json\_request = **"{\"step\_name\" : \"transformation.text\_column\_transform.TextTransform\","** \  
 **"\"columnName\": \"team\",\"transform\_func\": \"capitalize\","** \  
 **"\"split\_count\": \"2\"}"** text\_transform = TextTransform(json.loads(json\_request))  
 result = text\_transform.get\_result(self.data\_frame)  
 self.assertEqual(result[**'team'**][1], **"Royal challengers bangalore"**)  
  
 **def** test\_negative(self):  
 *"""  
 negative test cases* **:return***:  
 """* **with** self.assertRaises(Exception):  
 json\_request = {}  
 TextTransform(json.loads(json\_request))  
  
 **with** self.assertRaises(Exception):  
 TextTransform(None)  
  
 **with** self.assertRaises(Exception):  
 json\_request = **"{\"step\_name\" : \"transformation.text\_column\_transform.TextTransform\","** \  
 **"\"columnName\": \"team\",\"transform\_func\": \"capitalize\","** \  
 **"\"split\_count\": \"2\"}"** text\_transform = TextTransform(json.loads(json\_request))  
 text\_transform.get\_result(None)

*"""  
This module is used to apply small transformations on the text type of data columns.  
"""***import** logging  
**from** mashupbase **import** DataMashUpBase  
**import** constants  
**import** message\_constants  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** TextTransform(DataMashUpBase): *# pylint: disable=too-few-public-methods  
 """  
 Text column basic transformation on DataFrame  
 """* **def** \_\_init\_\_(self, json\_value):  
 *"""  
 initialize object with parameter* **:param** *json\_value:  
 """* **try**:  
 DataMashUpBase.\_\_init\_\_(self, json\_value)  
 self.transform\_function = json\_value[constants.TRANSFORM\_FUNCTION]  
 self.split\_pass = json\_value[constants.SPLIT\_PASS]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 Applies the transformation on text field. Mostly replaces the input column except for Split operation.  
 Currently supports - Upper case, Lower case, Split, Length* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **if** self.transform\_function == **"split"**:  
 input\_column\_name = self.column\_name  
 no\_of\_splits = int(self.split\_pass)  
 new\_columns = []  
  
 *# Starting from 1 so that new column names generated would also start from 1  
 # and will have name like <input\_column\_name>\_1* **for** i **in** range(0 + 1, no\_of\_splits + 2):  
 new\_columns.append(input\_column\_name + **"\_"** + str(i))  
  
 *# Adding new columns at end* data\_frame[new\_columns] = data\_frame[input\_column\_name].str.split(**" "**, expand=True,  
 n=no\_of\_splits)  
  
 *# The following code is to reorganise the newly added columns just after the input column* cols = data\_frame.columns.tolist()  
  
 column\_idx = cols.index(input\_column\_name)  
  
 cols = cols[:column\_idx + 1] + cols[-(no\_of\_splits + 1):] + cols[column\_idx + 1:-(  
 no\_of\_splits + 1)]  
  
 data\_frame = data\_frame[cols]  
  
 **elif** self.transform\_function == **"uppercase"**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].str.upper()  
  
 **elif** self.transform\_function == **"lowercase"**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].str.lower()  
  
 **elif** self.transform\_function == **"length"**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].str.len()  
  
 **elif** self.transform\_function == **"trim"**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].str.strip()  
  
 **elif** self.transform\_function == **"reverse"**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].str[::-1]  
  
 **elif** self.transform\_function == **"capitalize"**:  
 data\_frame[self.column\_name] = data\_frame[self.column\_name].str.capitalize()  
  
 **return** data\_frame  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*Foldername: Union*

*Filename: test\_union\_step*

*"""  
This module is performing test case for following type:  
union  
Intersection  
a minus b  
b minus a  
"""***import** unittest  
**import** pandas **as** pd  
**from** mock **import** patch  
**from** union.union\_step **import** UnionStep  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
  
  
**class** TestUnionStep(unittest.TestCase):  
 *"""  
 Test suite for testing union\_step module  
 """* **def** setUp(self):  
 self.json\_load = {**"type"**: **"Union"**, **"user\_id"**: 10, **"displayDataHeader"**: **"Union"**,  
 **"displayDataText"**: **"Union data"**,  
 **"leftDataSource"**: {**"dataId"**: 22, **"dataSourceName"**: **"Scenariosxlsx"**},  
 **"rightDataSource"**: {**"dataId"**: 2, **"dataSourceName"**: **"taxdatainput"**}}  
 self.data\_frame = pd.read\_csv(**'../resources/doc1.txt'**, header=0)  
 self.data\_frame2 = pd.read\_csv(**'../resources/doc2.txt'**, header=0)  
 self.data\_frame3 = pd.read\_csv(**'../resources/10Records.txt'**, header=0)  
  
 **def** test\_perform\_union\_operation(self):  
 *"""  
 Test method will perform the union operation on data\_frame* **:return***:  
 """* self.json\_load[**'type'**] = **'union'  
 with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_reader:  
 data\_frame\_reader.return\_value = self.data\_frame2  
 union\_step = UnionStep(self.json\_load)  
 data\_frame = union\_step.get\_result(self.data\_frame)  
 **if** len(data\_frame) == 5:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_perform\_intersaction\_operation(self):  
 *"""  
 Test method will perform the Intersection operation on data\_frame* **:return***:  
 """* self.json\_load[**'type'**] = **'intersection'  
 with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_reader:  
 data\_frame\_reader.return\_value = self.data\_frame2  
 union\_step = UnionStep(self.json\_load)  
 data\_frame = union\_step.get\_result(self.data\_frame)  
 **if** len(data\_frame) == 1:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_perform\_a\_minus\_b\_operation(self):  
 *"""  
 Test method will perform the a minus b operation on data\_frame* **:return***:  
 """* self.json\_load[**'type'**] = **'a\_minus\_b'  
 with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_reader:  
 data\_frame\_reader.return\_value = self.data\_frame2  
 union\_step = UnionStep(self.json\_load)  
 data\_frame = union\_step.get\_result(self.data\_frame)  
 **if** len(data\_frame) == 2:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_perform\_b\_minus\_a\_operation(self):  
 *"""  
 Test method will perform the b minus a operation on data\_frame* **:return***:  
 """* self.json\_load[**'type'**] = **'b\_minus\_a'  
 with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_reader:  
 data\_frame\_reader.return\_value = self.data\_frame2  
 union\_step = UnionStep(self.json\_load)  
 data\_frame = union\_step.get\_result(self.data\_frame)  
 **if** len(data\_frame) == 2:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_perform\_value\_error(self):  
 *"""  
 Test method will perform the b minus a operation on data\_frame* **:return***:  
 """* self.json\_load[**'type'**] = **'b\_minus\_a'  
 with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_reader:  
 data\_frame\_reader.return\_value = None  
 **try**:  
 union\_step = UnionStep(self.json\_load)  
 union\_step.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_perform\_merge\_error(self):  
 *"""  
 Test method will perform the b minus a operation on data\_frame* **:return***:  
 """* self.json\_load[**'type'**] = **'b\_minus\_a'  
 with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_reader:  
 data\_frame\_reader.return\_value = self.data\_frame3  
 **try**:  
 union\_step = UnionStep(self.json\_load)  
 union\_step.get\_result(self.data\_frame)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_perform\_key\_missing(self):  
 *"""  
 Test method will perform the b minus a operation on data\_frame* **:return***:  
 """* self.json\_load.pop(**"type"**)  
 **with** patch.object(DataRegistry,  
 **"create\_data\_frame\_using\_data\_source\_id"**) **as** data\_frame\_reader:  
 data\_frame\_reader.return\_value = self.data\_frame3  
 **with** self.assertRaises(Exception):  
 union\_step = UnionStep(self.json\_load)  
 union\_step.get\_result(self.data\_frame)

*Filename: union\_step*

*"""  
This module is used to perform four operation like union , intersaction,a -b and b-a  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** pandas.errors **import** MergeError  
**import** pandas **as** pd  
**import** constants  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** UnionStep:  
 *"""  
 This class performs four operations  
 like union,Intersection , a minus b and b minus a with input and request dataset  
 """* **def** \_\_init\_\_(self, json\_value):  
 **try**:  
 self.operation\_type = json\_value[constants.OPERATION\_TYPE]  
 self.user\_id = json\_value[constants.USER\_ID]  
 self.right\_data\_frame = DataRegistry().\  
 create\_data\_frame\_using\_data\_source\_id\  
 (json\_value[constants.RIGHT\_DATASOURCE][constants.DATA\_ID],  
 False, user\_id=self.user\_id)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.DATASET\_NOT\_FOUND)  
  
 **def** get\_result(self, data\_frame):  
 *"""  
 This method takes data\_frame as input and apply request operation* **:param** *data\_frame:* **:return***:  
 """* **try**:  
 **if** self.operation\_type.lower() == constants.UNION:  
 **return** pd.merge(data\_frame, self.right\_data\_frame, how=**'outer'**)  
 **if** self.operation\_type.lower() == constants.INTERSECTION:  
 **return** pd.merge(data\_frame, self.right\_data\_frame, how=**"inner"**)  
 **if** self.operation\_type.lower() == constants.A\_MINUS\_B:  
 **return** self.minus\_one\_to\_other\_data\_frame(  
 data\_frame1=data\_frame, minus\_type=constants.MINUS\_TYPE\_LEFT)  
 **return** self.minus\_one\_to\_other\_data\_frame(  
 data\_frame1=data\_frame, minus\_type=constants.MINUS\_TYPE\_RIGHT)  
 **except** MergeError **as** merge:  
 LOGGER.error(**'Exception: %s'**, merge)  
 **raise** Exception(message\_constants.NO\_COMMON\_COLUMNS)  
 **except** ValueError **as** value\_error:  
 LOGGER.error(**'Exception: %s'**, value\_error)  
 **raise** Exception(message\_constants.DATASET\_NOT\_FOUND)  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** minus\_one\_to\_other\_data\_frame(self, data\_frame1, minus\_type):  
 *"""  
 This method is used substract one data\_frame to another data\_frame* **:param** *data\_frame1:* **:param** *minus\_type:* **:return***:  
 """* new\_data\_frame = pd.merge(data\_frame1,  
 self.right\_data\_frame, how=minus\_type, indicator=True)  
 new\_data\_frame = new\_data\_frame.query(**"\_merge == '"** + str(minus\_type) + **"\_only'"**)  
 **return** new\_data\_frame.drop([constants.MERGE\_COLUMN], axis=1)

*Foldername: Mashup\_apis*

*Filename: Apiconfig.ini*

[SERVER\_CONFIG]  
SPARK\_SERVER\_HOST: 0.0.0.0  
#BACKEND\_API\_URL: 35.168.55.156:8089  
BACKEND\_API\_URL:  
SERVICE\_UP\_IP: 0.0.0.0  
SPARK\_SERVER\_PORT : 10000  
RULE\_MANAGER\_API\_URL : http://rule-manager-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080  
#MANAGE\_CONSOLE\_API\_URL : 35.168.55.156:8085  
MANAGE\_CONSOLE\_API\_URL :  
SQL\_SERVER\_PORT : 1699  
SQL\_SERVER\_HOST : mr2-aixd-av06.rno.apple.com  
SQL\_SERVER\_USER : GBI\_DATA\_LAB\_USER  
SQL\_SERVER\_PASSWORD : C0dd146t0n7\_Bri7fca57\_Ej2jb2  
DB\_SCHEMA : GBI\_DATA\_LAB\_OWNER  
SQL\_DATABASE : gia3d  
CONFIG\_LEVEL\_SETTINGS : configs.lens\_appleconnect  
CATALOG\_CONNECTIONS\_MANAGER\_SERVER : http://connection-manager-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080/connections/api  
CATALOG\_DISCOVER\_MANAGER\_SERVER : http://discover-api-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080/discover/api  
SAVE\_CONNECTION\_URL :http://gsf-dap.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080/gsf/dap/businessareas/utilities/subjectareas/dap/proxy/SaveDataConnection  
SCHEDULER\_API\_URL : scheduler-backend.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:5000  
JOB\_TOKEN : ODg5Njk1ODA3NjYyOWMxNzExYjk5MWM0NTUyZTg=  
S3\_URL: http://store-test.blobstore.apple.com  
ENV\_CD : datalab-dev  
S3\_REGION\_NAME : store-test  
S3\_BUCKET\_NAME : batch-jobs  
S3\_SECRET\_ACCESS\_KEY : 7DC67C86159E169819098FB9B4525E9C1DCFE70855815DCBBE6289A502C18FD9  
S3\_ACCESS\_KEY\_ID : MKIAS5P8XEXZ15LMC86K  
CATALOG\_ENV : https://gbiio-service:443  
DB\_USER = GBI\_DATA\_LAB\_USER  
DB\_DSN = mr2-aixd-av06.rno.apple.com:1699/gia3d  
DB\_password = C0dd146t0n7\_Bri7fca57\_Ej2jb2  
DB\_SCHEMA\_USER\_PREF = GBI\_DATA\_LAB\_USER  
DATALAB\_APP\_ID = 136927  
DATALAB\_APP\_PASSWORD = pxr767tssuvgaix4  
DATALAB\_APP\_CONTEXT = datalab  
CATALOG\_APP\_ID = 136847  
APP\_TOKEN\_GENERATION\_URL = https://idmsac-uat.corp.apple.com/auth/apptoapp/token/generate  
S3\_ID = 51b20ee0-2189-11e8-b716-09d7c9dd2f27  
CATALOG\_URL = https://gbiio-service:443/virtualedw  
CATALOG\_OTHER\_APP\_ID = 136927  
CATALOG\_APP\_CONTEXT = datalab

*Filename: class\_creator*

*"""  
This module is used for create class from string  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**import** message\_constants  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** ClassCreator(object):  
 *"""  
 This class is used for create class instance from string representation of class.  
 """* @classmethod  
 **def** get\_class(cls, class\_name):  
 *"""  
 give a instance of class from class\_name* **:param** *class\_name:* **:return***:  
 """* **try**:  
 parts = class\_name.split(**'.'**)  
 module = **"."**.join(parts[:-1])  
 import\_module = \_\_import\_\_(module)  
 **for** comp **in** parts[1:]:  
 import\_module = getattr(import\_module, comp)  
 **return** import\_module  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*Filename: cli*

*#! -\*- coding: utf-8 -\*-  
  
""" CLI for the app """***import** click  
**import** pytest  
**from** flask.cli **import** FlaskGroup  
**from** sqlalchemy **import** MetaData  
  
**from** setup\_app **import** app  
**from** notebooks\_backend.docker\_image.data **import** load\_data **as** docker\_image\_data  
**from** notebooks\_backend.docker\_image.models **import** db **as** docker\_image\_db  
**from** notebooks\_backend.notebook.models **import** db **as** notebook\_db  
**from** notebooks\_backend.pypi.data **import** load\_data **as** pypi\_data  
**from** notebooks\_backend.pypi.models **import** db **as** pypi\_db  
**from** notebooks\_backend.pypi.pypi\_index **import** create\_index  
**from** notebooks\_backend.resource.data **import** load\_data **as** resource\_data  
**from** notebooks\_backend.resource.models **import** db **as** resource\_db  
**from** notebooks\_backend.utils.migration **import** migrate\_pip\_packages  
  
  
  
*# initialize db*resource\_db.init\_app(app)  
notebook\_db.init\_app(app)  
docker\_image\_db.init\_app(app)  
pypi\_db.init\_app(app)  
  
  
**def** app\_factory():  
 **return** app  
  
  
@click.group(cls=FlaskGroup, create\_app=app\_factory)  
**def** cli():  
 *""" CLI for mosaic notebooks """* **pass  
  
  
def** create\_db():  
 *""" Create database """* resource\_db.create\_all()  
 docker\_image\_db.create\_all()  
 notebook\_db.create\_all()  
 pypi\_db.create\_all()  
  
  
**def** load\_data():  
 *""" Load sample data"""* resource\_data()  
 docker\_image\_data()  
  
  
@cli.command()  
**def** data():  
 *""" Load master data """* **with** app.app\_context():  
 load\_data()  
  
  
@cli.command()  
**def** packagedb():  
 *""" Load package and version data """* **with** app.app\_context():  
 pypi\_data()  
  
  
@cli.command()  
**def** test():  
 *""" Run test cases """* **with** app.app\_context():  
 create\_db()  
 pypi\_data()  
 load\_data()  
 pytest.main([**"--cov=src/notebooks\_backend"**])  
  
  
@cli.command()  
**def** index():  
 *""" Creating elastic index of pypi packages"""* **with** app.app\_context():  
 create\_index(refresh=False)  
  
  
@cli.command()  
**def** refresh():  
 *""" Refresh records in elastic index of pypi packages"""* **with** app.app\_context():  
 create\_index(refresh=True)  
  
  
@cli.command()  
**def** pipmigrate():  
 *""" Load master data """* **with** app.app\_context():  
 migrate\_pip\_packages()  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 cli()

*Filename: config\_manager*

*"""  
This module is used to read the global configuration of  
application from .ini file and used in across the application.  
"""***from** configparser **import** ConfigParser  
  
CONFIG\_MANAGER = ConfigParser()  
CONFIG\_MANAGER.read(**'apiconfig.ini'**)

*Filename: constants*

*# -\*- coding: utf-8 -\*-  
"""  
This module is used of declaring constants  
"""***from** config\_manager **import** CONFIG\_MANAGER  
**from** urllib.parse **import** quote\_plus **as** urlquote  
  
WEB\_SERVER = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'BACKEND\_API\_URL'**))  
RULE\_MANAGER\_SERVER = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'RULE\_MANAGER\_API\_URL'**))  
MANAGE\_CONSOLE\_SERVER = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'MANAGE\_CONSOLE\_API\_URL'**))  
CATALOG\_CONNECTIONS\_MANAGER\_SERVER = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**,  
 **'CATALOG\_CONNECTIONS\_MANAGER\_SERVER'**))  
CATALOG\_DISCOVER\_MANAGER\_SERVER = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**,  
 **'CATALOG\_DISCOVER\_MANAGER\_SERVER'**))  
  
FILE\_PATH = **"filePath"**DELIMITER = **"delimiterValue"**RECORD\_LIMIT = **"record\_limit"**DATASET\_NAME = **"dataSetName"**DATASET\_ID = **"dataSetId"**DATASET\_SCHEMA = **"schema"**FILTER\_TYPE = **"type"**OPERATOR = **"operator"**VALUE = **"value"**CASE\_SENSITIVE\_STATUS = **"case"**COLUMN\_NAME = **"columnName"**FROM\_VALUE = **"fromValue"**TO\_VALUE = **"toValue"**CONTAINS\_HEADER = **"containHeader"**FIELD\_DETAILS = **"fields"**FIELD\_NAME = **"fieldName"**FIELD\_TYPE = **"fieldType"**DATE = **"date"**TABLE\_NAME = **"table\_name"**HOST\_NAME = CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SPARK\_SERVER\_HOST'**)  
PORT = CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SPARK\_SERVER\_PORT'**)  
USER\_NAME = **""**PASSWORD = **""**DATABASE = **"default"**DATA\_SOURCE\_NAME = **"dataSourceName"**DATA\_SOURCE\_FETCH\_URL = WEB\_SERVER + **"/data/datasource/search/"**APPLY\_SET = **"steps"**APPLY\_ON = **"applyOn"**PAGE\_NUMBER = **"pageNumber"**LENS\_ID = **"lens\_id"**LENS\_NAME = **"lens\_name"**LENS\_OUTPUT\_DATA\_SOURCE = **"datasourceOutputId"**STEP\_NAME = **"step\_name"**DATA\_SOURCE\_ID = **"data\_source\_id"**TOTAL\_PAGE\_COUNT = **"total\_page\_count"**COLUMN\_TYPE = **"column\_type"**TO\_COLUMN = **"toColumn"**TRANSFORM\_FUNCTION = **"transform\_func"**SPLIT\_PASS = **"split\_count"**EXPRESSION = **"expression"**NEW\_COLUMN\_NAME = **"new\_column\_name"**IS\_EXCLUDED = **"is\_excluded"**VALIDATION\_FUNCTION = **"validation\_function"**LAST\_USED\_BY = **"last\_used\_by"**EXPIRE\_TIME\_INTERVAL = 15  
**"""  
Samples Strategy for dataframes  
"""**APPLY\_SAMPLES\_STRATEGY = **"apply\_samples\_strategy"**SAMPLES\_STRATEGY = **"samples\_strategy"**STRATEGY\_NAME = **"strategy\_name"**STRATEGY\_VALUE = **"strategy\_value"  
"""  
Filter condition constants and its values  
"""**FILTER\_CONDITIONS = **"conditions"**CONDITION\_EXCLUDE = **"exclude"**FUNC = **"func"**NUMBER = **"number"**DATE = **"date"**INTEGER = **"integer"**STRING = **"string"**DATE\_TIME = **"datetime"**NESTED\_CHILDREN = **"children"**WHEN\_OPERATOR = **"when"**AND = **"and"**OR = **"or"  
  
"""  
Pandas data frame constants  
"""**FIELDS = **"fields"**TYPE = **"type"**DATASET\_FIELD\_NAME = **"name"  
  
"""  
join type constants  
"""**JOIN\_TYPE = **"joinType"**LEFT\_DATASOURCE = **"leftDataSource"**RIGHT\_DATASOURCE = **"rightDataSource"**LEFT\_COLUMNS = **"leftColumns"**RIGHT\_COLUMNS = **"rightColumns"**DATA\_ID = **"dataId"  
  
"""  
Constant for rule and its id  
"""**RULE\_ID = **"rule\_id"**PARAMS = **"parameters"**RULE\_STEPS = **"ruleSteps"**PARAMETERS\_VALUE = **"parameters\_value"**RULE\_FETCH\_URL = RULE\_MANAGER\_SERVER + **"/insight-hub/rule-management/managed-rule/id="  
  
"""  
Constants for sort step  
"""**SORT\_BY = **"sort\_by"**SORT\_TYPE = **"sort\_type"  
  
"""  
Constants for union step  
"""**OPERATION\_TYPE = **"type"**MERGE\_COLUMN = **"\_merge"**MINUS\_TYPE\_LEFT = **"left"**MINUS\_TYPE\_RIGHT = **"right"**UNION = **"union"**INTERSECTION = **"intersection"**A\_MINUS\_B = **"a\_minus\_b"**B\_MINUS\_A = **"b\_minus\_a"**WORKING\_DATA\_SET = **"WORKING"  
  
"""  
profiling constants  
"""**FREQ = **"freq"**PROFILING\_VARIABLE = **"variables"**FREQ\_THRESHOLD\_COUNT = 10  
FREQ\_THRESHOLD\_VARIABLE = **"Others"  
"""  
group by constants  
"""**AGGREGATE\_FUNCTION\_TYPE = **"agg\_func\_type"**MAX = **"max"**MIN = **"min"**SUM = **"sum"**MEAN = **"mean"**STD = **"std"**MEDIAN = **"median"**PERCENTILE\_25 = **"percentile\_25"**PERCENTILE\_75 = **"percentile\_75"**PERCENTILE\_90 = **"percentile\_90"**FREQUENCY = **"frequency"**HISTOGRAM = **"histogram"  
"""  
Imputation Type  
"""**IMPUTATION\_TYPE = **"imputation\_type"**IMPUTATION\_METHOD = **"imputation\_method"**REMOVE\_ROWS = **"remove\_rows"**DEFAULT\_REPLACE = **"default\_replace"**DEFAULT\_VALUE = **"default\_value"**MODE = **"mode"**ZERO = **"zero"**REPLACE\_WITH\_COLUMN = **"replace\_with\_column"**REPLACE\_TO\_COLUMN = **"replace\_to\_column"  
"""  
Insights constants  
"""**IDENTIFIER = **"identifier"**TARGET\_LIST = **"target\_list"**NUMERIC\_COLS = **"numeric\_cols"**FACTOR\_COLS = **"factor\_cols"**ID\_COLS = **"id\_cols"**DATE\_COLS = **"date\_cols"**IMP\_FEATURES\_LIST = **"imp\_features\_list"**USER\_NAME1 = **"user\_name"**CREATED\_BY = **"created\_by"**UPDATED\_BY = **"updated\_by"**INSIGHT\_MASTER\_ID = **"insight\_master\_id"**COLS\_REMOVED = **"cols\_removed"**TARGET\_IDENTIFIER = **"target\_identifier"**COLUMN\_LIST = **"column\_list"**COEFFICIENTS = **"coefficients"**LABEL\_ENCODING = **"label\_encoding"**WHAT\_IF\_MASTER\_ID = **"what\_if\_master\_id"**INSIGHTS\_ML\_SCENARIOS = **"insights\_ml\_scenarios"**SCENARIO\_NAME = **"scenario\_name"**SCENARIO\_INPUT = **"scenario\_input"**SCENARIO\_ID = **"scenario\_id"**SCENARIO\_OUTPUT = **"scenario\_output"**INSIGHTS\_ML\_MASTER = **"insights\_ml\_master"**INSIGHTS\_ML\_USER\_INPUT = **"insights\_ml\_user\_input"**INSIGHTS\_ML\_WHAT\_IF\_MASTER = **"insights\_ml\_what\_if\_master"**TARGET = **"target"**USER\_INPUT\_ID = **"user\_input\_id"**COLS\_TO\_REMOVE = **"cols\_to\_remove"**COEF = **"coef"**COLS = **"cols"**CATEGORICAL\_FEATURES = **"categorical\_features"**LINEAR\_REGRESSION = **"linear\_regression"**COLS\_UNIQUE\_VALUES = **"cols\_unique\_values"**MYSQL\_CONNECTION\_URL = **"mysql+mysqldb://"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_USER'**) + **":"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_PASSWORD'**) + **"@"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_HOST'**) + **":"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_PORT'**) + **"/"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_DATABASE'**)  
CELERY\_RESULTS\_BACKEND = **"db+mysql://"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_USER'**) + **":"** + \  
 urlquote(  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_PASSWORD'**)) + **"@"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_HOST'**) + **":"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_PORT'**) + **"/"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_DATABASE'**)  
  
*#ORACLE\_CONNECTION\_URL = "oracle+cx\_oracle://GBI\_DATA\_LAB\_OWNER:C0ab135s0n3\_Bdi3fcm53\_Fi1iw2@mr2-aixd-av06.rno.apple.com:1699/?service\_name=gia3d"  
# ORACLE\_CONNECTION\_URL = "oracle+cx\_oracle://GBI\_DATA\_LAB\_USER:C0dd146t0n7\_Bri7fca57\_Ej2jb2@mr2-aixd-av06.rno.apple.com:1699/?service\_name=gia3d"*ORACLE\_CONNECTION\_URL = **"oracle+cx\_oracle://"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_USER'**) + **":"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_PASSWORD'**) + **"@"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_HOST'**) + **":"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_SERVER\_PORT'**) + **"/"** + \  
 CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SQL\_DATABASE'**)  
  
TS\_DATE = **"ts\_date"**TS\_KPI = **"ts\_kpi"**TS\_FREQ = **"ts\_freq"**FORECAST\_PERIODS = **"forecast\_periods"**FILTER\_DICT = **"filter\_dict"**CLUSTER = **"cluster"**TOTAL\_CLUSTERS = **"total\_clusters"**TOTAL\_RECORDS = **"total\_records"**CLUSTER\_NAME = **"cluster\_name"**CLUSTER\_INFO = **"cluster\_info"**CLUSTER\_VISUALIZATION = **"cluster\_visualization"**CLUSTER\_ROW\_COUNT = **"cluster\_row\_count"**PERCENTAGE = **"percentage"**CLUSTER\_ID = **"cluster\_id"**CLUSTERING\_ENABLE = **"Clustering\_enable"**MSE = **"mean\_squared\_error"**RMSE = **"root\_mean\_squared\_error"**MAE = **"mean\_absolute\_error"**MAPE = **"mean\_absolute\_percentage\_error"**TRAINING\_DATA\_LENGTH = **"training\_data\_length"**TEST\_DATA\_LENGTH = **"test\_data\_length"**DATA\_LENGTH = **"data\_length"**CLUSTER\_TABLE\_SUMMARY = **"cluster\_table\_summary"  
  
"""  
Batch Job constants declaration  
"""**LENS\_FETCH\_URL = RULE\_MANAGER\_SERVER + \  
 **"/insight-hub/step-management/managed-step/lensId="**DRIVE\_DATA\_SOURCE\_CREATE = WEB\_SERVER + \  
 **"/data/saveDataSource"**UPDATE\_LENS\_OUTPUT = RULE\_MANAGER\_SERVER + \  
 **"/insight-hub/lens-management/managed-lens/lensId={0}&dataId={1}"**ADD\_NOTIFICATION = MANAGE\_CONSOLE\_SERVER + \  
 **"/insight-hub/notification-management/notification-add"**GET\_LENS\_DETAILS = RULE\_MANAGER\_SERVER + **"/insight-hub/lens-management/managed-lens/id="**ERROR = **"error"**MESSAGE = **"message"**LOAD\_TYPE = **"load\_type"**JSON\_VALUE = **"json\_value"  
  
"""  
group by constants  
"""**MERGE\_OUTPUT = **"merge\_output"**GROUP\_FUNCTION = **"group\_function"**GROUP\_BY\_FIELDS = **"group\_by"**AGG\_FUNCTION\_ARRAY = **"AggFunctionArray"  
  
"""  
keycloak changes  
"""**USER\_ID = **"user\_id"  
  
"""  
Dedupe column changes  
"""**DEDUPE\_ON\_COLUMN = **"dedupe\_on\_column"**DEDUPE\_RETAIN\_POLICY = **"dedupe\_retain\_policy"**REMOVE\_ALL\_DUPLICATES = **"remove\_all\_duplicates"  
  
"""  
Find a replace feature  
"""**SEARCH\_TYPE = **"search\_type"**FIND\_VALUE = **"find\_value"**REPLACE\_WITH = **"replace\_with"**REGEX\_STATUS = **"regex"**FIND\_VALUE = **"find\_value"**FULL\_SEARCH\_TYPE = **"full\_search\_type"**EXACT\_MATCH = **"exact\_match"  
"""  
For Validations  
"""**OUTPUT\_OPTION = **"output\_options"  
  
"""  
Run only enabled step  
"""**IS\_ENABLED = **"is\_enabled"  
  
"""  
Knowledge base  
"""***# FETCH\_KNOWLEDGE\_BASE = "http://" + WEB\_SERVER + "/data/managed-knowledge-base/dataSourceId="  
# FETCH\_KB\_BY\_SOURCEID\_COLUMN = "http://" + WEB\_SERVER + "/data/managed-knowledge-base/sourceId="  
# UPDATE\_KB\_BY\_SOURCEID\_COLUMN = "http://" + WEB\_SERVER + "/data/managed-knowledge-base/sourceId="  
# INSERT\_IN\_KNOWLEDGE\_BASE = "http://" + WEB\_SERVER + "/data/managed-knowledge-base"*INSERT\_KNOWLEDGE\_BASE\_GRAPH = WEB\_SERVER + **"/data/managed-auto-insights"**FETCH\_KNOWLEDGE\_BASE = WEB\_SERVER + **"/data/managed-knowledge-base/dataSourceId="**FETCH\_KNOWLEDGE\_BASE\_USING\_LENS = WEB\_SERVER + **"/data/managed-knowledge-base/lensId="**FETCH\_KB\_BY\_SOURCEID\_COLUMN = WEB\_SERVER + **"/data/managed-knowledge-base/sourceId="**UPDATE\_KB\_BY\_SOURCEID\_COLUMN = WEB\_SERVER + **"/data/managed-knowledge-base/sourceId="**INSERT\_IN\_KNOWLEDGE\_BASE = WEB\_SERVER + **"/data/managed-knowledge-base"**TIME\_SERIES\_DATE\_ATTRIBUTE = **"timeSeriesDateAttribute"**TIME\_SERIES\_KPI = **"timeSeriesKpi"**PRIORITY = **"priority"**TIME\_SERIES\_FREQUENCY = **"defaultFrequency"**SENTIMENT = **"sentiment"**FORECAST\_BIN = **"forecast\_bin"**FORECAST\_COLUMN = **"forecast\_column"**AUTO\_INSIGHTS = **"auto\_insights"**FITTED = **"fitted"**BOOLEAN\_INDICATOR = **"booleanIndicator"**CATEGORICAL\_INDICATOR = **"categoricalIndicator"**KB\_INDEX = **"index"**KNN = **"knn"**IMPUTE\_STEP = **"transformation.imputations.imputations.Imputations"**DROP\_STEP = **"transformation.drop\_column.DropColumn"**COLUMN\_GT\_30 = **"Column is having "**DROP\_COLUMN = **"Drop this columns"**THIS\_COLUMN\_NAME = **"this.columnName"**IMPUTE\_COLUMN = **"Impute this columns"**DISPLAY\_DATA\_HEADER = **"displayDataHeader"**DISPLAY\_DATA\_TEXT = **"displayDataText"**EDIT\_COL\_NAME = **"editColName"  
  
"""Running Total"""**PARTITION\_COLUMN = **"partition\_column"**SORT\_COLUMN = **"sort\_column"**AGGREGATE\_COLUMN = **"aggregate\_column"**IS\_SUGGESTED = **"is\_suggested"  
  
"""Data Cleansing"""**INDEX\_POSITION = **"index\_position"**EXTRACT\_BY\_INDEX\_COL = **"extracted\_index"**EXTRACT\_VALUE\_COUNT\_COL = **"extracted\_count"**EXTRACT\_VALUE\_PATTERN\_COL = **"extracted\_pattern"**DATA\_PATTERN = **"data\_pattern"**EXTRACT\_VALUE\_COUNT = **"extract\_value\_count"**EXTRACT\_UNIQUE\_SORTED = **"extract\_unique\_sorted"  
  
"""Manage alerts"""**MANAGE\_ALERTS = RULE\_MANAGER\_SERVER + **"/insight-hub/alert-management/managed-alert"  
  
"""Catalog Connector Service"""**CATALOG\_GET\_CONNECTION\_CONFIG =**"/External/v1/external/getConnConfig"**CATALOG\_GET\_DATASOURCE = **"/v1/datasource/"**SERVICE\_NAME = **"s3"**USE\_SSL = False  
REGION\_NAME = **"store-test"**ENDPOINT\_URL = **"http://store-test.blobstore.apple.com"**AWS\_ACCESS\_KEY\_ID = **"MKIAS5P8XEXZ15LMC86K"**AWS\_SECRET\_ACCESS\_KEY = **"7DC67C86159E169819098FB9B4525E9C1DCFE70855815DCBBE6289A502C18FD9"**ADDRESSING\_STYLE = **"addressing\_style"**PATH = **"path"**SIGNATURE\_VERSION = **"s3v4"**READ\_TIMEOUT = 60000  
CONNECT\_TIMEOUT = 600000  
BUCKET\_NAME = **"batch-jobs"  
  
  
"""added for flows integration"""**OPERATION\_EXCEPTION = **"Operation failed. Please try again."**OPERATOR = **"operator"**VALUE = **"value"**CASE\_SENSITIVE\_STATUS = **"case"**IS\_EXCLUDED = **"is\_excluded"**APPLY\_SET = **"steps"**DISPLAY\_HEADER = **"displayDataHeader"**SOURCE\_TYPE = **"sourceType"**SUB\_SOURCETYPE= **"subSourceType"  
  
  
"""User preferences"""**DATE\_TIME\_FORMAT = **"%a, %d %B %Y %H:%M:%S"**CONN\_LENS\_ID = **"lens\_id"**CONN\_FORMAT = **"format"**CONN\_MODE = **"mode"**CONN\_USER\_ID = **"user\_id"**CONN\_NAME = **"data\_source"**DB\_USER = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'DB\_USER'**))  
DB\_DSN = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'DB\_DSN'**))  
DB\_PASSWORD = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'DB\_password'**))  
DB\_SCHEMA = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'DB\_SCHEMA\_USER\_PREF'**))  
  
**""" GBIIO Constants """**DATALAB\_APP\_ID = **"DATALAB\_APP\_ID"**DATALAB\_APP\_PASSWORD = **"DATALAB\_APP\_PASSWORD"**DATALAB\_APP\_CONTEXT = **"DATALAB\_APP\_CONTEXT"**CATALOG\_APP\_ID = **"CATALOG\_APP\_ID"**APP\_TOKEN\_GENERATION\_URL = **"APP\_TOKEN\_GENERATION\_URL"**COOKIE\_NAME = **"acack-uat"**GBIIO\_DATASOURCE\_CREATE = CATALOG\_DISCOVER\_MANAGER\_SERVER + **"/v1/datasource"**UID = **"c5043073"**OU = **"people"**GCSLDAP = **"gcsldap"**CORP = **"corp"**APPLE = **"apple"**COM = **"com"**LDAP\_SERVER = **"gcsldap.apple.com"**LDAP\_PASSWORD = **"Apple@123"**BASE\_DIR = **"/projects/{}/datalab\_wrangler"**ROW\_LIMIT = 10  
  
**""" Job integration constants"""**JOB\_END\_POINT = **"http://{}/scheduler/api/v1/job"**JOB\_TOKEN = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'JOB\_TOKEN'**))  
SCHEDULER\_URL = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SCHEDULER\_API\_URL'**))  
S3\_URL = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'S3\_URL'**))  
ENV\_CD\_VALUE = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'ENV\_CD'**))  
S3\_BUCKET = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'S3\_BUCKET\_NAME'**))  
S3\_REGION = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'S3\_REGION\_NAME'**))  
S3\_SECRET\_KEY = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'S3\_SECRET\_ACCESS\_KEY'**))  
S3\_ACCESS\_KEY = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'S3\_ACCESS\_KEY\_ID'**))  
S3\_URL = str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'S3\_URL'**))  
CRON\_EXPR = **"cron\_expr"**CRON\_JSON = **"cron\_json"**DESCRIPTION = **"description"**ENTITY\_ID = **"entity\_id"**ENTITY\_TYPE = **"entity\_type"**IN\_PARAMS = **"in\_params"**NAME = **"name"**PROJECT\_ID = **"project\_id"**PROJECT\_NAME = **"project\_name"**COMMENTS = **"comments"**OUT\_PARAMS = **"out\_params"**LOG\_PATH = **"LOG\_PATH"**RETRY = **"retry"**RUN\_ID = **"run\_id"**ENV\_CD = **"env\_cd"**STATUS\_CD = **"status\_cd"**JOB\_NAME = **"wrangler"**JOB\_STARTED = **"JOB\_STARTED"**PROJECT = **"project"**COOKIE = **"cookie"**USER\_EMAIL = **"user\_email"**WRANGLER = **"wrangler"**CRON\_EXPR\_VAL = **"0 0 0 ? \* \*"**JOB\_DESC = **"Wrangler Publish Job"**AUTHORIZATION = **"Authorization"**X\_DATALAB\_USER = **"x-datalab-user"**CONTENT\_TYPE = **"Content-type"**ACCEPT = **"Accept"**APP\_JSON = **"application/json"**JOB\_COMMENTS = **"wrangler publish job run"**

*Filename: Dag\_Json\_Generator*

**import** uuid  
**import** pprint  
**import** logging  
**import** constants  
**from** jsonpath\_ng **import** jsonpath, parse  
**import** sys  
  
LOGGER = logging.getLogger(**'DAG Json Generator'**)  
  
**class** DAGJsonGenerator(object):  
 *"""  
 This class is used for generating DAG json based on input json responses  
 """* **def** \_\_init\_\_(self,mashuprequest,gsfconfigresponse,gsfdsresponse):  
 **try**:  
 self.mashuprequest=mashuprequest  
 self.gsfconfigresponse=gsfconfigresponse  
 self.gsfdsresponse=gsfdsresponse  
 self.steps = self.mashuprequest[constants.APPLY\_SET]  
 *#self.value = mashuprequest[constants.VALUE]  
 #self.case = mashuprequest[constants.CASE\_SENSITIVE\_STATUS]  
 #self.operator = mashuprequest[constants.OPERATOR]  
 #self.exclude = mashuprequest[constants.IS\_EXCLUDED]* **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
  
  
  
 **def** deriveSparkWranglerTransformConfigs(self):  
 **try**:  
 **for** step **in** self.steps:  
 **print** (step)  
 **if** len(self.steps) == 0:  
 configuration\_spark\_filter\_transform = self.createSparkSelectExpTransformConfigs()  
 **elif** step[constants.DISPLAY\_HEADER] == **'Filter'**:  
 configuration\_spark\_filter\_transform = self.createSparkFilterTransformConfigs()  
  
 **elif** step[constants.DISPLAY\_HEADER] == **'GroupBy'**:  
 configuration\_spark\_filter\_transform = self.createSparkGroupByTransform()  
 **elif** step[constants.DISPLAY\_HEADER] == **'Dedupe'**:  
 configuration\_spark\_filter\_transform = self.createSparkDropDuplicatesTransform()  
 **else**:  
 Exception(constants.OPERATION\_EXCEPTION)  
 **return** configuration\_spark\_filter\_transform  
 **except**:  
 **print**(**'Unexpected error:'**, sys.exc\_info()[0])  
  
  
  
  
  
  
  
  
  
  
  
  
 *# def deriveSparkWranglerTransformConfigs(self):  
 # # type: () -> object  
 # try:  
 # for step in self.steps:  
 # print ("printing step",step)  
 # #print("step[constants.DISPLAY\_HEADER] value",step[constants.DISPLAY\_HEADER])  
 # if step == "":  
 # configuration\_spark\_filter\_transform = self.createSparkSelectExpTransformConfigs()  
 # elif step[constants.DISPLAY\_HEADER] == 'Filter':  
 # configuration\_spark\_filter\_transform = self.createSparkFilterTransformConfigs()  
 # elif step[constants.DISPLAY\_HEADER] == 'GroupBy':  
 # configuration\_spark\_filter\_transform = self.createSparkGroupByTransform()  
 # elif step[constants.DISPLAY\_HEADER] == 'Dedupe':  
 # configuration\_spark\_filter\_transform = self.createSparkDropDuplicatesTransform()  
 # else:  
 # Exception(constants.OPERATION\_EXCEPTION)  
 #  
 # return configuration\_spark\_filter\_transform  
 #  
 # except:  
 # print('Unexpected error:', sys.exc\_info()[0])  
 # # LOGGER.exception('Exception: %s', ex)  
 # # raise Exception(constants.OPERATION\_EXCEPTION)  
  
  
 #except:  
 #print('Unexpected error:', sys.exc\_info()[0])  
 # LOGGER.exception('Exception: %s', ex)  
 # raise Exception(constants.OPERATION\_EXCEPTION)  
 #elif step == '':  
 #configuration\_spark\_filter\_transform = self.createSparkSelectExpTransformConfigs()  
 # elif step[constants.DISPLAY\_HEADER] == 'Cumulative Sum':  
 # configuration\_spark\_filter\_transform = self.createSparkFilterTransformConfigs()  
 # elif step[constants.DISPLAY\_HEADER] == 'Sort':  
 # configuration\_spark\_filter\_transform = self.createSparkFilterTransformConfigs()* **def** deriveSparkReadTransformConfigs(self):  
 *# type: () -> object* **try**:  
  
 **if** self.gsfdsresponse[constants.SOURCE\_TYPE] == **'RDBMS' and** self.gsfdsresponse[constants.SUB\_SOURCETYPE] == **'TERADATA'** :  
 configuration\_spark\_read\_transform=self.createSparkReadTransformConfigs()  
 **elif** self.gsfdsresponse[constants.SOURCE\_TYPE] == **'RDBMS' and** self.gsfdsresponse[constants.SUB\_SOURCETYPE] == **'ORACLE'** :  
 configuration\_spark\_read\_transform=self.createSparkTeradataReadTransformConfigs()  
 **else**:  
 Exception(constants.OPERATION\_EXCEPTION)  
  
 **return** configuration\_spark\_read\_transform  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
  
  
 **def** createSparkReadTransformConfigs(self):  
 *# type: () -> object* **try**:  
  
 *#if self.gsfdsresponse["type"] == 'RDBMS' and self.gsfdsresponse["sub\_type"] == 'TERADATA' :  
 #configuration\_spark\_read\_transform=self.createSparkReadTransformConfigs()* configuration\_spark\_read\_transform = {}  
 configuration\_spark\_read\_transform[**"fetchSize"**] = **"5000"** configuration\_spark\_read\_transform[**"lowerBound"**] = **"0"** configuration\_spark\_read\_transform[**"upperBound"**] = **"10"** configuration\_spark\_read\_transform[**"numPartitions"**] = **"1"** configuration\_spark\_read\_transform[**"fastExport"**] = False  
 json\_data = self.gsfdsresponse  
  
 jsonpath\_expression\_connectionId = parse(**'$.connectionId'**)  
  
 *#print("connectionId expression ",jsonpath\_expression\_connectionId)* jsonpath\_expression\_find\_connectionId = jsonpath\_expression\_connectionId.find(json\_data)  
 jsonpath\_expression\_find\_connectionId\_value = jsonpath\_expression\_find\_connectionId[0].value  
 jsonpath\_expression\_dbname = parse(**'$.dataNode.value.dbName'**)  
 match\_dbname = jsonpath\_expression\_dbname.find(json\_data)  
 jsonpath\_expression\_dbname\_value = match\_dbname[0].value  
 jsonpath\_expression\_tablename = parse(**'$.name'**)  
 match\_tablename = jsonpath\_expression\_tablename.find(json\_data)  
 jsonpath\_expression\_tablename\_value = match\_tablename[0].value  
 *#print("jsonpath\_expression\_find\_connectionId expression ",jsonpath\_expression\_find\_connectionId)  
 #jsonpath\_expression\_dbtable = parse('employees[\*].id')* configuration\_spark\_read\_transform[**"connectionId"**] = jsonpath\_expression\_find\_connectionId\_value  
 configuration\_spark\_read\_transform[**"dbtable"**] = jsonpath\_expression\_dbname\_value + **'.'** + jsonpath\_expression\_tablename\_value  
 *#configuration\_spark\_read\_transform["connectionId"] = self.gsfdsresponse['connectionId']  
 # configuration\_spark\_read\_transform["dbtable"] = self.gsfdsresponse['dataNode']['value']['dbName'] + '.' + \  
 # self.gsfdsresponse['name']* **return** configuration\_spark\_read\_transform  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
 **def** createSparkGroupByTransform(self):  
 *# type: () -> object* **try**:  
 configuration\_spark\_GroupBy\_transform = {}  
 configuration\_spark\_GroupBy\_transform[**"fetchSize"**] = **"5000"** configuration\_spark\_GroupBy\_transform[**"lowerBound"**] = **"0"** configuration\_spark\_GroupBy\_transform[**"upperBound"**] = **"10"** configuration\_spark\_GroupBy\_transform[**"numPartitions"**] = **"1"** configuration\_spark\_GroupBy\_transform[**"fastExport"**] = False  
 configuration\_spark\_GroupBy\_transform[**"connectionId"**] = self.gsfdsresponse[**"connectionId"**]  
 configuration\_spark\_GroupBy\_transform[**"dbtable"**] = self.gsfdsresponse[**'dataNode'**][**'value'**][**'dbName'**] + **'.'** + \  
 self.gsfdsresponse[**'name'**]  
 **return** configuration\_spark\_GroupBy\_transform;  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
  
  
  
  
 **def** createSparkSelectExpTransformConfigs(self):  
 **try**:  
 json\_data = self.mashuprequest  
 jsonpath\_expression\_select\_exp = parse(**'$.steps[:]'**)  
 jsonpath\_expression\_find\_select\_exp = jsonpath\_expression\_select\_exp.find(json\_data)  
 *#filterconditionraw = match\_filter[0].value* **print**(jsonpath\_expression\_find\_select\_exp)  
 *#filterconditionoutputtest=filterconditionraw.replace("WHEN (","").replace(")","").rstrip().lstrip()  
 #filterconditionoutput=filterconditionoutputtest.replace("equals","=")  
 #print("printing the filter",filterconditionoutput)* configuration\_spark\_filter\_transform = {a:**"1"**}  
 *#configuration\_spark\_filter\_transform= jsonpath\_expression\_find\_select\_exp* **return** configuration\_spark\_filter\_transform  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
 **def** createSparkFilterTransformConfigs(self):  
 **try**:  
 json\_data = self.mashuprequest  
 jsonpath\_expression\_filter = parse(**'$.steps[:1].displayDataText'**)  
 match\_filter = jsonpath\_expression\_filter.find(json\_data)  
 filterconditionraw = match\_filter[0].value  
 **print**(filterconditionraw)  
 filterconditionoutputtest1=filterconditionraw.replace(**"WHEN ("**,**""**).replace(**")"**,**""**).rstrip().lstrip()  
 filterconditionoutputtest2=filterconditionoutputtest1.replace(**"("**,**""**).replace(**")"**,**""**).rstrip().lstrip()  
 *#filterconditionoutputtest=filterconditionoutputtest2.replace("WHEN (","").replace(")","").rstrip().lstrip()* filterconditionoutput=filterconditionoutputtest2.replace(**"equals"**,**"="**)  
 **print**(**"printing the filter"**,filterconditionoutput)  
 configuration\_spark\_filter\_transform = {}  
 configuration\_spark\_filter\_transform[**"filter"**] = filterconditionoutput  
 **return** configuration\_spark\_filter\_transform  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
 **def** createSparkWriteTransformConfigs(self):  
 **try**:  
 configuration\_spark\_write\_transform = {}  
 json\_data = self.gsfdsresponse  
  
 jsonpath\_expression\_connectionId = parse(**'$.connectionId'**)  
 jsonpath\_expression\_find\_connectionId = jsonpath\_expression\_connectionId.find(json\_data)  
 jsonpath\_expression\_find\_connectionId\_value = jsonpath\_expression\_find\_connectionId[0].value  
 jsonpath\_expression\_dbname = parse(**'$.dataNode.value.dbName'**)  
 match\_dbname = jsonpath\_expression\_dbname.find(json\_data)  
 jsonpath\_expression\_dbname\_value = match\_dbname[0].value  
 *#configuration\_spark\_write\_transform["dbtable"] = jsonpath\_expression\_dbname\_value + '.' + "adhoc\_flow\_test"  
 #configuration\_spark\_write\_transform["mode"] = "overwrite"  
 #configuration\_spark\_write\_transform["fastload"] = False  
 #configuration\_spark\_write\_transform["connectionId"] = jsonpath\_expression\_find\_connectionId\_value* configuration\_spark\_write\_transform[**"n"**] = 10  
  
 **return** configuration\_spark\_write\_transform  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
 **def** createSparkDropDuplicatesTransform(self):  
 *# type: () -> object* **try**:  
 json\_data = self.mashuprequest  
 jsonpath\_expression\_filter = parse(**'$.steps[:1].displayDataText'**)  
 match\_filter = jsonpath\_expression\_filter.find(json\_data)  
 dedupeconditionraw = match\_filter[0].value  
 dedupeconditionoutput = dedupeconditionraw.rstrip().lstrip()  
 configuration\_spark\_dedupe\_transform = {}  
 configuration\_spark\_dedupe\_transform[**"dedupe"**] = dedupeconditionoutput  
 **return** configuration\_spark\_dedupe\_transform  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
 **def** createDAGjson(self):  
 **try**:  
 id\_spark\_readtransform = str(uuid.uuid4())  
 **print**(**"id\_spark\_readtransform:"** + id\_spark\_readtransform)  
 ui\_sparkreadtransform = {}  
 ui\_sparkreadtransform = {**"pos"**: [-0.5, -0.5], **"script"**: **""**}  
 outputGate\_sparkreadtransform = {**"bindings"**: {**"output"**: {**"type"**: **"dataframe"**}}}  
 requires\_sparkreadtransform = []  
  
 id\_spark\_filtertransform = str(uuid.uuid4())  
 **print**(**"id\_spark\_filtertransform:"** + id\_spark\_filtertransform)  
 ui\_sparkfiltertransform = {}  
 ui\_sparkfiltertransform = {**"pos"**: [-0.25, -0.25], **"script"**: **""**, **"selectedInputTypes"**: {},  
 **"selectedNodeType"**: {}}  
 inputGate\_sparkfiltertransform = {**"bindings"**: {**"input"**: {**"type"**: **"dataframe"**}}}  
 outputGate\_sparkfiltertransform = {**"bindings"**: {**"output"**: {**"type"**: **"dataframe"**}}}  
 requires\_sparkfiltertransform = []  
 linkdetails\_sparkfiltertransform = {**"nodeId"**: id\_spark\_readtransform, **"as"**: **"data"**, **"outputId"**: **"output"**,  
 **"bindTo"**: **"input"**}  
 requires\_sparkfiltertransform.append(linkdetails\_sparkfiltertransform)  
 pprint.pprint(requires\_sparkfiltertransform)  
  
 id\_spark\_writetransform=str(uuid.uuid4())  
 ui\_sparkwritetransform = {}  
 ui\_sparkwritetransform = {**"pos"**: [-0.75, -0.75], **"script"**: **""**, **"selectedInputTypes"**: {}, **"selectedNodeType"**: {}}  
 inputGate\_sparkwritetransform = {**"bindings"**: {**"input"**: {**"type"**: **"dataframe"**}}}  
 requires\_sparkwritetransform = []  
 linkdetails\_sparkwritetransform = {**"nodeId"**: id\_spark\_filtertransform, **"as"**: **"data"**, **"outputId"**: **"output"**,  
 **"bindTo"**: **"input"**}  
 requires\_sparkwritetransform.append(linkdetails\_sparkwritetransform)  
 pprint.pprint(requires\_sparkwritetransform)  
  
 *#sparktransforms = {"sparkRDBMSread": {}, "sparkFilterTransform": {}, "sparkShowWrite": {}}  
  
 #mashup json request to test multiple conditions in filter  
  
 #{"data\_source\_id":"033ae311-45fa-485e-a0a5-64590ae83519","lens\_id":276,"pageNumber":1,"record\_limit":100,"steps":[{"displayDataHeader":"Filter","displayDataText":" WHEN ( EMP\_ID > 5 ) AND ( EMP\_NAME equals Jerry ) ","is\_edited":true,"step\_name":"filter.filter\_condition.FilterCondition","exclude":false,"conditions":[{"start\_bracket":"(","func":"when","column\_type":"string","columnName":"EMP\_ID","column\_id":"EMP\_ID","operator":">","is\_nested":false,"children":[],"uID":1,"case":"false","is\_saved":true,"close\_bracket":")","value":"5","is\_excluded":false},{"start\_bracket":"(","func":"and","column\_type":"string","columnName":"EMP\_NAME","column\_id":"EMP\_NAME","operator":"equals","is\_nested":false,"children":[],"uID":3,"case":"false","is\_saved":true,"close\_bracket":")","value":"Jerry","is\_excluded":false}],"htmlForConditions":["<span class=\"badge badge-outline\">WHEN</span><br><span class=\"bracket\">(</span><span class=\"badge badge-light\" title=\"EMP\_ID\">EMP\_ID</span> <span class=\"badge bor-rid\" title=\">\">></span><span class=\"badge badge-light\" title=\"5\">5</span><span class=\"bracket\">)</span>","<span class=\"badge badge-outline\">AND</span><br><span class=\"bracket\">(</span><span class=\"badge badge-light\" title=\"EMP\_NAME\">EMP\_NAME</span> <span class=\"badge bor-rid\" title=\"equals\">equals</span><span class=\"badge badge-light\" title=\"Jerry\">Jerry</span><span class=\"bracket\">)</span>"],"expressionForConditions":[" WHEN ( EMP\_ID > 5 ) "," AND ( EMP\_NAME equals Jerry ) "],"is\_enabled":true}],"applyOn":"WORKING","total\_page\_count":"0","apply\_samples\_strategy":false,"samples\_strategy":{"strategy\_name":"dataframe\_reader.samples\_strategy.top\_n.TopNStrategy","strategy\_value":11}}* sparktransforms = {**"sparkRDBMSread"**: {}, **"sparkFilterTransform"**: {}, **"sparkShowWrite"**: {}}  
  
 *#selecting all columns  
 #mashupjson request with empty steps to test select columns in dag\_json\_workflow  
 #{"data\_source\_id":"033ae311-45fa-485e-a0a5-64590ae83519","lens\_id":276,"pageNumber":1,"record\_limit":100,"steps":[],"applyOn":"ORIGINAL","total\_page\_count":"0","apply\_samples\_strategy":false,"samples\_strategy":{"strategy\_name":"dataframe\_reader.samples\_strategy.top\_n.TopNStrategy","strategy\_value":11}}  
 #sparktransforms = {"sparkRDBMSread": {}, "sparkSelectTransform": {}, "sparkShowWrite": {}}  
  
 #configuration\_spark\_read\_transform=self.createSparkReadTransformConfigs()* configuration\_spark\_read\_transform=self.deriveSparkReadTransformConfigs()  
 configuration\_spark\_filter\_transform=self.deriveSparkWranglerTransformConfigs()  
 configuration\_spark\_filter\_transform\_select = self.deriveSparkWranglerTransformConfigs()  
 *#configuration\_spark\_filter\_transform=self.createSparkFilterTransformConfigs()* configuration\_spark\_write\_transform=self.createSparkWriteTransformConfigs()  
  
  
  
  
  
 sparktransforms[**"sparkFilterTransform"**].update({  
 **"id"**: id\_spark\_filtertransform,  
 **"name"**: **"Filter"**,  
 **"configuration"**: configuration\_spark\_filter\_transform,  
 **"ui"**: ui\_sparkfiltertransform,  
 **"inputGate"**: inputGate\_sparkfiltertransform,  
 **"outputGate"**: outputGate\_sparkfiltertransform,  
 **"requires"**: requires\_sparkfiltertransform  
 })  
  
 sparktransforms[**"sparkRDBMSread"**].update({  
 **"id"**: id\_spark\_readtransform,  
 **"name"**: **"RDBMS read"**,  
 **"configuration"**: configuration\_spark\_read\_transform,  
 **"ui"**: ui\_sparkreadtransform,  
 **"outputGate"**: outputGate\_sparkreadtransform,  
 **"requires"**: requires\_sparkreadtransform  
 })  
  
 sparktransforms[**"sparkShowWrite"**].update({  
 **"id"**: str(uuid.uuid4()),  
 **"name"**: **"Show Rows"**,  
 **"configuration"**: configuration\_spark\_write\_transform,  
 **"ui"**: ui\_sparkwritetransform,  
 **"inputGate"**: inputGate\_sparkwritetransform,  
 **"requires"**: requires\_sparkwritetransform  
 })  
  
 *# sparktransforms["sparkSelectTransform"].update({  
 # "id": id\_spark\_filtertransform,  
 # "name": "Select Columns",  
 # "configuration": configuration\_spark\_filter\_transform\_select,  
 # "ui": ui\_sparkfiltertransform,  
 # "inputGate": inputGate\_sparkfiltertransform,  
 # "outputGate": outputGate\_sparkfiltertransform,  
 # "requires": requires\_sparkfiltertransform  
 # })* dagjson = {}  
 dagjson.update({  
 **"id"**: str(uuid.uuid4()),  
 **"name"**: **"flows integration test\_v4"**,  
 **"version"**: **"1.0.0"**,  
 **"nodes"**: [{**'sparkRDBMSRead'**: sparktransforms[**'sparkRDBMSread'**]},  
 {**'sparkFilterTransform'**: sparktransforms[**'sparkFilterTransform'**]},  
 {**'sparkShowWrite'**: sparktransforms[**'sparkShowWrite'**]}]  
 })  
 *# dagjson.update({  
 # "id": str(uuid.uuid4()),  
 # "name": "flows integration test\_v4",  
 # "version": "1.0.0",  
 # "nodes": [{'sparkRDBMSRead': sparktransforms['sparkRDBMSread']},  
 # {'sparkSelectTransform': sparktransforms['sparkSelectTransform']},  
 # {'sparkShowWrite': sparktransforms['sparkShowWrite']}]  
 # })* pprint.pprint(dagjson)  
 **return** dagjson  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)  
  
  
*Filename: flask\_celery*

*"""Creates celery app"""***from** celery **import** Celery  
  
  
**def** make\_celery(app):  
 *"""Creates celery app"""* celery = Celery(  
 app.import\_name,  
 backend=app.config[**'CELERY\_RESULT\_BACKEND'**],  
 broker=app.config[**'CELERY\_BROKER\_URL'**]  
 )  
 celery.conf.update(app.config)  
  
 **class** ContextTask(celery.Task):  
 **def** \_\_call\_\_(self, \*args, \*\*kwargs):  
 **with** app.app\_context():  
 **return** self.run(\*args, \*\*kwargs)  
  
 celery.Task = ContextTask  
 **return** celery

*Filename: json\_parser\_utility*

*"""  
This module is used to operation on json  
"""***import** logging  
**import** json  
**import** message\_constants  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** JsonUtility:  
 *"""  
 Class is used for parsing json object  
 """* @classmethod  
 **def** required\_key\_value\_from\_json(cls, field\_name, json\_value):  
 *"""  
 method will find the respective key from json* **:param** *field\_name:* **:param** *json\_value* **:return***:  
 """* **try**:  
 **if** isinstance(json\_value, str):  
 json\_value = json.loads(json\_value)  
  
 results = []  
 **if** isinstance(json\_value, list):  
 **for** item **in** json\_value:  
 results.append(item[field\_name])  
  
 **return** results  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** find\_value\_of\_matched\_key\_value(cls, find\_value, match\_key, match\_value, json\_value):  
 *"""  
 method will find the respective key from json* **:param** *find\_value:* **:param** *json\_value* **:param** *match\_value* **:param** *match\_key* **:return***:  
 """* **try**:  
 **if** isinstance(json\_value, str):  
 json\_value = json.loads(json\_value)  
  
 results = []  
 **if** isinstance(json\_value, list):  
 **for** item **in** json\_value:  
 **if** item[match\_key].lower() == match\_value.lower():  
 results.append(item[find\_value])  
  
 **return** None **if not** results **else** results  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** check\_key\_exist\_in\_dict(cls, json\_value, key\_name):  
 *"""  
 This method will check if key exist in the json* **:param** *json\_value:* **:param** *key\_name:* **:return***:  
 """* **try**:  
 **if** key\_name **in** json\_value:  
 **return** True  
 **return** False  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*Filename: mashup\_flows\_runner*

*"""  
This module is used to initiate batch job using data\_source\_id and lens\_id  
"""***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
**from** random **import** randint  
**import** json  
**import** pandas **as** pd  
**import** constants  
**import** message\_constants  
**from** request\_builder **import** RequestBuilder  
**from** pandas\_api\_rest **import** PandasApi  
**from** batch\_job.data\_frame\_writer **import** SparkDataFrameWriter  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** token\_registry **import** TokenRegistry  
**from** json\_parser\_utility **import** JsonUtility  
**import** boto3  
**from** botocore.client **import** Config  
**from** io **import** StringIO  
**import** datetime  
**import** pprint  
**from** jsonpath\_ng **import** jsonpath, parse  
LOGGER = logging.getLogger(\_\_name\_\_)  
LOGGER.setLevel(logging.DEBUG)  
HANDLER = RotatingFileHandler(**'pandas\_api.log'**, maxBytes=20000000, backupCount=10)  
FORMATTER = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
HANDLER.setFormatter(FORMATTER)  
LOGGER.addHandler(HANDLER)  
  
  
**class** FlowsMashupRunner: *# pylint: disable=R0902  
 """  
 This class contains all the mashup api code  
 which will execute steps array on the data source.  
 """* **def** \_\_init\_\_(self, input\_json,project\_id):  
 **try**:  
 *#import pdb  
 #pdb.set\_trace()* self.project\_id = project\_id  
 headers = {**"X-Datalab-Project-Id"** : self.project\_id}  
 *#self.data\_source\_id = input\_json[constants.DATA\_SOURCE\_ID]* self.lens\_id = input\_json[constants.LENS\_ID]  
 *#self.lens\_name = input\_json.get(constants.LENS\_NAME, 'DefaultLensName')  
 #self.table\_name = self.lens\_name + "\_" + \  
 #input\_json[constants.LOAD\_TYPE] + "\_" + str(randint(100, 999))* self.user\_id = input\_json[constants.USER\_ID]  
 authorization\_header = TokenRegistry.get\_token\_for\_user(user\_id=self.user\_id)  
 self.fetch\_lens\_details = RequestBuilder().call\_http\_get\_request(  
 constants.GET\_LENS\_DETAILS + str(self.lens\_id), {},headers).json()  
 *# self.lens\_details = RequestBuilder().call\_http\_get\_request(  
 # constants.GET\_LENS\_DETAILS + str(self.lens\_id), {},  
 # TokenRegistry.get\_token\_for\_user(user\_id=self.user\_id))* self.data\_source\_id = self.fetch\_lens\_details.get(**"datasourceInputId"**)  
 self.lens\_name = self.fetch\_lens\_details.get(**"lensName"**)  
 *#self.is\_scheduled = input\_json.get("is\_scheduled", False)* self.input\_json = input\_json  
 *#self.data\_source\_name = input\_json[constants.DATA\_SOURCE\_NAME]* **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** Exception(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** run\_flows\_mashup\_job(self):  
 *"""  
 This method will create mashup json independent of ui using class attribute data\_source\_id and lens\_id* **:return***:  
 """* **try**:  
 **import** pdb  
 pdb.set\_trace()  
 *# data\_source\_details = RequestBuilder().call\_http\_get\_request(  
 # constants.CATALOG\_DISCOVER\_MANAGER\_SERVER +  
 # constants.CATALOG\_GET\_DATASOURCE + str(self.data\_source\_id), {}, {})  
 # data\_source\_details = json.loads(data\_source\_details.content.decode('utf-8'))* steps\_details = RequestBuilder().call\_http\_get\_request(  
 constants.LENS\_FETCH\_URL + str(self.lens\_id), {},  
 TokenRegistry.get\_token\_for\_user(user\_id=self.user\_id))  
 *#print("steps\_details",steps\_details)* step\_details = steps\_details.json()  
 **print**(**"steps\_details"**,step\_details)  
 stepMetadata = step\_details[**'stepMetadata'**]  
 strategy\_name\_json = json.loads(stepMetadata)  
 stepConfig = step\_details[**'stepConfig'**]  
 *#step\_details = json.loads(steps\_details)  
 #jsonpath\_expression = parse('$.stepMetadata[\*]')  
 # match\_stratery\_name = jsonpath\_expression.find(step\_details)  
 #strategy\_name = match\_stratery\_name[0].value  
 #strategy\_name\_json = json.loads('strategy\_name')  
 #stepMetadata = steps\_details["stepMetadata"]* strategy\_name\_value = strategy\_name\_json[**'strategy\_name'**]  
 strategy\_value= strategy\_name\_json[**'strategy\_value'**]  
 *#strategy\_value = stepMetadata[1]* **print**(**"stepMetadata value"**,strategy\_name\_value)  
 **print**(**"step details"**,strategy\_value)  
  
 flowsmashupjson = self.createFlowsMashupjson(strategy\_name\_value, strategy\_value,stepConfig)  
 pprint.pprint(flowsmashupjson)  
 **return** flowsmashupjson  
  
  
  
 **except** Exception **as** ex:  
 LOGGER.error(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** createFlowsMashupjson(self,strategy\_name\_value,strategy\_value,stepConfig):  
 **try**:  
 **import** pdb  
 pdb.set\_trace()  
 data\_source\_id = self.data\_source\_id  
 lens\_id = self.lens\_id  
 pageNumber = 1  
 samples\_strategy = {  
 **'strategy\_name'**: strategy\_name\_value,  
 **'strategy\_value'**: strategy\_value}  
 flowsmashupjson = {}  
 flowsmashupjson.update({  
 **"data\_source\_id"**: data\_source\_id,  
 **"lens\_id"**: lens\_id,  
 **"pageNumber"**: 1,  
 **"record\_limit"**:100,  
 **"steps"**: stepConfig,  
 **"applyOn"**: **"ORIGINAL"**,  
 **"total\_page\_count"**:0,  
 **"apply\_samples\_strategy"**: False,  
 **"samples\_strategy"**: samples\_strategy  
 })  
 **print**(**"sample strategy"**,samples\_strategy)  
 **print**(**"json details"**,flowsmashupjson)  
 pprint.pprint(flowsmashupjson)  
 **return** flowsmashupjson  
  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(constants.OPERATION\_EXCEPTION)

*Filename: mashup\_base*

*"""  
This is the base class of all the filter  
"""***import** logging  
**from** abc **import** ABC , abstractmethod  
**import** constants  
**import** message\_constants  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** DataMashUpBase: *# pylint: disable=too-few-public-methods  
 """  
 This is base class of All the filter  
 """* **def** \_\_init\_\_(self, json\_value):  
 **try**:  
 self.column\_name = json\_value[constants.COLUMN\_NAME]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @abstractmethod  
 **def** get\_result(self):  
 *"""  
 This is the abstract method for all the transformation* **:return***:  
 """* **pass**

*Filename: message\_constants*

*# -\*- coding: utf-8 -\*-  
"""  
This module is used of declaring message constants  
"""*COLUMNS\_WITH\_DIFFERENT\_DATATYPES= **"One of the Dataset not found or you are trying to join columns "** \  
 **"with different datatypes."**COLUMN\_NOT\_FOUND = **"{} column not found in dataset."**KEY\_NOT\_UNIQUE = **"Column selected in key is not unique."**DATASET\_NOT\_FOUND = **"One of the dataset not found."**NO\_COMMON\_COLUMNS = **"No common columns to perform the operation."**SELECT\_NUMERIC\_COLUMN = **"Please select a numerical column."**SELECT\_TEXT\_COLUMN = **"Please select a text column."**SORT\_NO\_COLUMN\_SELECTED = **"No column used for sort operation."**NO\_PATTERNS\_FOUND = **"No patterns found, please try with a lower value of support."**DIFFERENT\_GROUP\_LIST\_ITEM\_COLUMN\_VALUES = **"Should have different values in "** \  
 **"group\_list and item\_column."**INVALID\_SUPPORT\_VALUE = **"Support value should be between 0 to 100."**NONE\_INPUT\_DATASET = **"Input data set is None."**OPERATION\_EXCEPTION = **"Operation failed. Please try again."**ONLY\_APPLICABLE\_FOR\_NUMERIC\_COLUMNS = **"This function is only applicable on numeric columns"**SELECT\_DATE\_TYPE\_COLUMN = **"Selected column should be date type"**INPUT\_SHOULD\_BE\_NUMERIC = **"Please select numeric column/value"**KEY\_NOT\_FOUND\_IN\_INPUT = **"{} key not found in input json"**INSUFFICIENT\_MEMORY\_FOR\_PERFORMING\_OPERATION = **"Insufficient memory for performing the operation"**INVALID\_COMPARISION\_TYPE = **"Invalid comparision type"**ERROR\_WHILE\_CREATING\_CONNECTION\_WITH\_HIVE = **"Error while creating connection with hive"**DATASET\_IS\_NOT\_SUITABLE = **"Dataset is not suitable to perform the operation"**INVALID\_COLUMN\_TYPE = **"Invalid column type"**REQUEST\_EXCEPTION\_OCCURED = **"Request exception occured"**HTTP\_ERROR = **"Http error occured"**CONNECTION\_ERROR = **"Connection error occured"**TIMEOUT\_ERROR = **"Timeout error occured"**INSERT\_OR\_UPDATE\_FAILED = **"Insert or update operation failed."**EXECUTE\_STATEMENT\_FAILED = **"Execute query operation failed"**GBIIO\_WRITE\_FAILED = **"Error while writing into MCQUEEN location"**GBIIO\_READ\_FAILED = **"Error while read into MCQUEEN location"**GBIIO\_PUBLISH\_FAILED = **"Error while publishing the dataset"**WRITE\_LOGS\_S3\_BUCKET = **"Unable to write logs to s3 bucket"**

*Filename: pandas\_api\_rest*

*# -\*- coding: utf-8 -\*-  
"""  
 This module is used for register dataFrame and give result based on condition.  
"""***import** logging  
**import** json  
**from** io **import** StringIO  
**import** constants  
**import** message\_constants  
**from** dataframe\_reader.mashup\_data\_registry **import** DataRegistry  
**from** class\_creator **import** ClassCreator  
**import** uuid  
**import** requests  
**import** pandas **as** pd  
  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
*#LOGGER = logging.getLogger('pandas\_api')***class** PandasApi: *# pylint: disable-msg=R0902  
 """  
 This class is used for pandas DataFrame and web service integration.  
 """* **def** \_\_init\_\_(self, input\_json):  
 **try**:  
 self.limit\_count = int(input\_json[constants.RECORD\_LIMIT])  
 self.page\_number = int(input\_json[constants.PAGE\_NUMBER])  
 self.total\_page\_count = input\_json[constants.TOTAL\_PAGE\_COUNT]  
 self.steps = input\_json[constants.APPLY\_SET]  
 self.apply\_on = input\_json[constants.APPLY\_ON]  
 self.data\_source\_id = input\_json[constants.DATA\_SOURCE\_ID]  
 self.lens\_id = input\_json[constants.LENS\_ID]  
 self.user\_id = input\_json[constants.USER\_ID]  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 *# non mandate inputs* **try**:  
 *#self.apply\_samples\_strategy = False* self.apply\_samples\_strategy = input\_json.get(constants.APPLY\_SAMPLES\_STRATEGY, False)  
 self.samples\_strategy = **''  
 if** self.apply\_samples\_strategy:  
 self.samples\_strategy = input\_json[constants.SAMPLES\_STRATEGY]  
 self.samples\_strategy.update({**"data\_source\_id"**: self.data\_source\_id})  
 self.samples\_strategy.update({**"lens\_id"**: self.lens\_id})  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** start\_workflow(self):  
 *"""  
 This is kick start workflow* **:return***:  
 """* **try**:  
 LOGGER.debug(**'start\_workflow : user\_id = %s ; steps = %s ;'  
 ' apply\_on = %s ; lens = %s ; data\_src\_id = %s'**, self.user\_id, self.steps,  
 self.apply\_on, self.lens\_id, self.data\_source\_id)  
 **"""  
 dummy\_df = pd.DataFrame({'num\_legs': [2, 4, 8, 0],  
 'num\_wings': [2, 0, 0, 0],  
 'num\_specimen\_seen': [10, 2, 1, 8]},  
 index=['falcon', 'dog', 'spider', 'fish'])  
 DataRegistry.pandas\_df\_map.update({"2\_2": {  
 "ORIGINAL": dummy\_df,  
 "WORKING": dummy\_df,  
 constants.LAST\_USED\_BY:  
 DataRegistry.convert\_datetime\_into\_ms(),  
 constants.STRATEGY\_NAME: "top\_n",  
 constants.STRATEGY\_VALUE: 100000  
 }  
 })  
 """** flag\_dataframe\_updated = False  
 flag\_dataframe\_registered = DataRegistry.check\_data\_frame\_is\_register(  
 self.data\_source\_id, self.lens\_id)  
 **if** flag\_dataframe\_registered **is** False **or** self.apply\_samples\_strategy:  
 **if** flag\_dataframe\_registered **is not** False:  
 temp\_df\_reg\_key = DataRegistry.create\_register\_id(self.data\_source\_id,  
 self.lens\_id)  
 temp\_new\_strategy = self.samples\_strategy.get(constants.STRATEGY\_NAME, **"top\_n"**)  
 temp\_new\_strategy\_val = int(  
 self.samples\_strategy.get(constants.STRATEGY\_VALUE, 100000))  
 temp\_old\_strategy = DataRegistry.pandas\_df\_map.get(temp\_df\_reg\_key)[  
 constants.STRATEGY\_NAME]  
 temp\_old\_strategy\_val = int(  
 DataRegistry.pandas\_df\_map.get(temp\_df\_reg\_key)[constants.STRATEGY\_VALUE])  
 **if** temp\_old\_strategy **not in** temp\_new\_strategy **or** \  
 temp\_old\_strategy\_val != temp\_new\_strategy\_val:  
 **if "bottom\_n" in** temp\_new\_strategy:  
 temp\_new\_strategy = **"bottom\_n"** data\_frame = DataRegistry(). \  
 create\_data\_frame\_using\_data\_source\_id(  
 self.data\_source\_id,  
 load\_all\_flag=False,  
 user\_id=self.user\_id,  
 publish=False,  
 samples\_strategy=self.samples\_strategy)  
 DataRegistry.register\_data\_frame(self.data\_source\_id, self.lens\_id,  
 data\_frame)  
 DataRegistry.update\_frame\_strategy(self.data\_source\_id, self.lens\_id,  
 temp\_new\_strategy, temp\_new\_strategy\_val)  
 flag\_dataframe\_updated = True  
 **else**:  
 data\_frame = DataRegistry(). \  
 create\_data\_frame\_using\_data\_source\_id(  
 self.data\_source\_id,  
 load\_all\_flag=False,  
 user\_id=self.user\_id,  
 publish=False,  
 samples\_strategy=self.samples\_strategy)  
 DataRegistry.register\_data\_frame(self.data\_source\_id, self.lens\_id,  
 data\_frame)  
 flag\_dataframe\_updated = True  
 **elif not** self.apply\_samples\_strategy:  
 temp\_df\_reg\_key = DataRegistry.create\_register\_id(self.data\_source\_id,  
 self.lens\_id)  
 temp\_new\_strategy = **"top\_n"** temp\_new\_strategy\_val = 100000  
 temp\_old\_strategy = DataRegistry.pandas\_df\_map.get(temp\_df\_reg\_key)[  
 constants.STRATEGY\_NAME]  
 temp\_old\_strategy\_val = int(  
 DataRegistry.pandas\_df\_map.get(temp\_df\_reg\_key)[constants.STRATEGY\_VALUE])  
 **if** temp\_old\_strategy **not in** temp\_new\_strategy **or** \  
 temp\_old\_strategy\_val != temp\_new\_strategy\_val:  
 **if "bottom\_n" in** temp\_new\_strategy:  
 temp\_new\_strategy = **"bottom\_n"** data\_frame = DataRegistry(). \  
 create\_data\_frame\_using\_data\_source\_id(  
 self.data\_source\_id,  
 load\_all\_flag=False,  
 user\_id=self.user\_id,  
 publish=False,  
 samples\_strategy=self.samples\_strategy)  
 DataRegistry.register\_data\_frame(self.data\_source\_id, self.lens\_id,  
 data\_frame)  
 DataRegistry.update\_frame\_strategy(self.data\_source\_id, self.lens\_id,  
 temp\_new\_strategy, temp\_new\_strategy\_val)  
 flag\_dataframe\_updated = True  
 **if** flag\_dataframe\_updated:  
 self.apply\_on = **"ORIGINAL"  
 if** self.apply\_on.upper() == **'WORKING'**:  
 self.apply\_step\_on\_frame(self.attach\_user\_with\_step(  
 self.steps[len(self.steps) - 1]))  
 **elif** self.apply\_on.upper() == **'ORIGINAL'**:  
 DataRegistry.keep\_original\_as\_working(self.data\_source\_id, self.lens\_id)  
 **for** step **in** self.steps:  
 **if** step[constants.IS\_ENABLED] **is** True:  
 self.apply\_step\_on\_frame(self.attach\_user\_with\_step(step))  
 LOGGER.debug(**'start\_workflow ENDS'**)  
 **return** self.fetch\_data\_from\_data\_set()  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** attach\_user\_with\_step(self, step):  
 *"""  
 This method will attach username with step json* **:return***:  
 """* **try**:  
 LOGGER.debug(**'attach\_user\_with\_step'**)  
 step.update({**"user\_id"**: self.user\_id})  
 LOGGER.debug(**'attach\_user\_with\_step ENDS'**)  
 **return** step  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** apply\_step\_on\_frame(self, step):  
 *"""  
 Apply steps on data frame* **:param** *step:* **:return***:  
 """* **try**:  
 LOGGER.debug(**'apply\_step\_on\_frame : step = %s'**, step)  
 data\_frame = DataRegistry.get\_data\_set\_from\_dict(  
 self.data\_source\_id, self.lens\_id, **'WORKING'**)  
 DataRegistry.update\_frame(self.data\_source\_id,  
 self.lens\_id,  
 PandasApi.apply\_individual\_step(data\_frame, step), **'WORKING'**)  
 LOGGER.debug(**'apply\_step\_on\_frame ENDS'**)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 @classmethod  
 **def** apply\_individual\_step(cls, data\_frame, step):  
 *"""  
 This method will be used to apply step on dataframe.* **:return***:  
 """* **try**:  
 LOGGER.debug(**'apply\_individual\_step : step = %s'**, step)  
 instance = ClassCreator.get\_class(step[constants.STEP\_NAME])(step)  
 LOGGER.debug(**'apply\_individual\_step ENDS'**)  
 **return** instance.get\_result(data\_frame)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** fetch\_data\_from\_data\_set(self):  
 *"""  
 Fetching request data from DataRegistry domain* **:return***:  
 """* **try**:  
 LOGGER.debug(**'fetch\_data\_from\_data\_set'**)  
 data\_frame = DataRegistry.get\_data\_set\_from\_dict(  
 self.data\_source\_id, self.lens\_id, **'WORKING'**)  
 total\_record = len(data\_frame.index)  
 data\_available = json.loads(data\_frame.take(self.create\_fetch\_record\_list(total\_record))  
 .to\_json(orient=**"table"**))  
 data\_available.update({**"total\_record"**: total\_record})  
 string\_io = StringIO()  
 json.dump(data\_available, string\_io)  
 LOGGER.debug(**'fetch\_data\_from\_data\_set ENDS'**)  
 **return** string\_io.getvalue()  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** create\_fetch\_record\_list(self, total\_record):  
 *"""  
 This is the pagination logic* **:param** *total\_record:* **:return***:  
 """* **try**:  
 LOGGER.debug(**'create\_fetch\_record\_list'**)  
 **print**(**"limiting preview as 20 records"**)  
 **"""self.total\_page\_count = total\_record / self.limit\_count  
 from\_value = (self.page\_number - 1) \* 10  
 to\_value = total\_record if (self.page\_number \* self.limit\_count) > total\_record \  
 else (self.page\_number \* self.limit\_count)  
 LOGGER.debug('create\_fetch\_record\_list ENDS')  
 return range(from\_value, to\_value)"""** recordcount = range(0,total\_record)  
 **"""if total\_record < 20:  
 recordcount = range(0,total\_record)  
 else:  
 recordcount = range(0,20)"""  
 return** recordcount  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
  
**class** PandasApiForDataFrame:  
 *"""  
 @author: Aaditya Agrawal  
 This class is used for Pandas dataframe column based operations  
 """* **def** \_\_init\_\_(self, input\_json):  
 *"""  
 Initialize input* **:param** *input\_json: json input  
 """  
 # mandatory input params* **try**:  
 self.data\_source\_id = input\_json[constants.DATA\_SOURCE\_ID]  
 self.lens\_id = input\_json[constants.LENS\_ID]  
 self.user\_id = input\_json[constants.USER\_ID]  
 self.apply\_on = input\_json[constants.APPLY\_ON]  
 LOGGER.debug(**'PandasApiForDataFrame init method : data\_source\_id = %s ; lens\_id = %s ;'  
 'user\_id = %s ; apply\_on = %s'**, self.data\_source\_id, self.lens\_id,  
 self.user\_id, self.apply\_on)  
 **except** KeyError **as** key\_error:  
 LOGGER.error(**'Exception: %s'**, key\_error)  
 **raise** KeyError(message\_constants.KEY\_NOT\_FOUND\_IN\_INPUT.format(key\_error.args[0]))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
 *# non-mandatory input params for future enhancement* **def** get\_dataframe\_column\_with\_types(self):  
 *"""  
 Function to get column with types from dataframe* **:return***: string with column name and data type of column  
 """* **try**:  
 LOGGER.debug(**'get\_dataframe\_column\_with\_types'**)  
 **if** self.apply\_on.upper() == **'WORKING'**:  
 data\_frame = DataRegistry.get\_data\_set\_from\_dict(  
 self.data\_source\_id, self.lens\_id, **'WORKING'**)  
 **elif** self.apply\_on.upper() == **'ORIGINAL'**:  
 *# DataRegistry.keep\_original\_as\_working(self.data\_source\_id, self.lens\_id)* data\_frame = DataRegistry.get\_data\_set\_from\_dict(self.data\_source\_id,  
 self.lens\_id, **'ORIGINAL'**)  
 **else**:  
 LOGGER.error(**'Exception : Invalid %s column type'**, self.apply\_on)  
 **raise** Exception(message\_constants.INVALID\_COLUMN\_TYPE)  
 data\_frame\_clms = json.loads(data\_frame.take(range(0)).to\_json(orient=**"table"**))  
 string\_io = StringIO()  
 json.dump(data\_frame\_clms, string\_io)  
 LOGGER.debug(**'get\_dataframe\_column\_with\_types ENDS'**)  
 **return** string\_io.getvalue()  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 *# def lifecycle\_hooks(catalog\_token\_generation\_url,cookie\_name,cookie\_value,catalog\_token\_name):  
 # LOGGER.info('Calling lifecycle\_hooks')  
 # hooks = {  
 # "postStart": {  
 # "exec": {  
 # # pylint: disable=line-too-long  
 # "command": [  
 # "/bin/sh",  
 # "-c",  
 # "([ -f /usr/local/bin/datalab-init ] && /usr/local/bin/datalab-init || true);"  
 # 'response=$(curl -k --silent -X GET {} --cookie "{}={}");'  
 # 'if [ ! -z "$response" ]; then '  
 # "token=$(python -c \"print($response['Token'])\");"  
 # 'if [ ! -z "$token" ]; then '  
 # "mkdir /home/gbi-datalab/.gbi;"  
 # 'echo "[gbiio]" > /home/gbi-datalab/.gbi/gbiio.conf;'  
 # 'echo "{}=$token" >> /home/gbi-datalab/.gbi/gbiio.conf;'  
 # "chmod 777 /home/gbi-datalab/.gbi/gbiio.conf;"  
 # "fi; fi;".format(  
 # catalog\_token\_generation\_url,  
 # cookie\_name,  
 # cookie\_value,  
 # catalog\_token\_name,  
 # ),  
 # ]  
 # }  
 # }  
 # }  
 # LOGGER.info('Calling lifecycle\_hooks ends')  
 # return hooks*

*Filename: request\_builder*

*"""  
Request Builder Module  
"""***import** logging  
**import** requests  
**import** message\_constants  
  
LOGGER = logging.getLogger(**'lens\_backend'**)  
  
  
**class** RequestBuilder(object):  
 *"""  
 This call is used for calling http request  
 """* **def** \_\_init\_\_(self):  
 **pass  
  
 def** call\_request\_url(self, request\_type, url, data, headers):  
 *"""  
 Call the request based on type ( it support get/post/put)* **:param** *request\_type:* **:param** *url* **:param** *data* **:param** *headers* **:return***:  
 """* **try**:  
 **if** request\_type == **'get'**:  
 result = requests.get(url, data=data, headers=headers)  
 **elif** request\_type == **'post'**:  
 result = requests.post(url, json=data, headers=headers)  
 **elif** request\_type == **'put'**:  
 result = requests.put(url, data=data, headers=headers)  
 **else**:  
 **raise** Exception(message\_constants.REQUEST\_EXCEPTION\_OCCURED)  
 result.raise\_for\_status()  
 **return** result  
 **except** requests.exceptions.HTTPError **as** http\_error:  
 LOGGER.error(**'Exception: %s'**, http\_error)  
 **raise** Exception(message\_constants.HTTP\_ERROR)  
 **except** requests.exceptions.ConnectionError **as** connection\_error:  
 LOGGER.error(**'Exception: %s'**, connection\_error)  
 **raise** Exception(message\_constants.CONNECTION\_ERROR)  
 **except** requests.exceptions.Timeout **as** timeout\_error:  
 LOGGER.error(**'Exception: %s'**, timeout\_error)  
 **raise** Exception(message\_constants.TIMEOUT\_ERROR)  
 **except** requests.exceptions.RequestException **as** request\_exception:  
 LOGGER.error(**'Exception: %s'**, request\_exception)  
 **raise** Exception(message\_constants.REQUEST\_EXCEPTION\_OCCURED)  
  
 **def** call\_http\_get\_request(self, url, data, headers):  
 *"""  
 This is used for calling httpGet request* **:param** *url:* **:param** *data:* **:param** *headers:* **:return***:  
 """* **try**:  
 **return** self.call\_request\_url(**'get'**, url=url, data=data, headers=headers)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** call\_http\_post\_request(self, url, data, headers):  
 *"""  
 This method is used to post request to request url* **:param** *url:* **:param** *data:* **:param** *headers:* **:return***:  
 """* **try**:  
 **return** self.call\_request\_url(**'post'**, url=url, data=data, headers=headers)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)  
  
 **def** call\_http\_put\_request(self, url, data, headers):  
 *"""  
 This method is used to post request to request url* **:param** *url:* **:param** *data:* **:param** *headers:* **:return***:  
 """* **try**:  
 **return** self.call\_request\_url(**'put'**, url=url, data=data, headers=headers)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **raise** Exception(message\_constants.OPERATION\_EXCEPTION)

*Filename: setup\_app*

*# -\*- coding: utf-8 -\*-  
"""  
Flask Api manager which will register the blueprint api  
"""***import** os  
**from** flask **import** Flask  
**from** config\_manager **import** CONFIG\_MANAGER  
**from** web\_server **import** WEB\_SERVER  
**from** insights.insights\_server **import** INSIGHTS\_SERVER  
**from** flask\_migrate **import** Migrate  
*# celery and socketio specific imports***import** logging  
**from** logging.handlers **import** RotatingFileHandler  
  
**from** flask\_celery **import** make\_celery  
**from** flask\_sqlalchemy **import** SQLAlchemy  
**from** flask\_socketio **import** SocketIO  
**from** flask\_cors **import** CORS  
  
app = Flask(**"Data Mash up service"**)  
CORS(app)  
  
*# initialize settings  
# default = "configs.lens\_keycloak"*config\_object = CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'CONFIG\_LEVEL\_SETTINGS'**)  
app.config.from\_object(config\_object)  
  
*# initialize logger*lens\_log\_file = os.path.join(app.config[**"LOG\_DIR"**], **"lens\_backend.log"**)  
lens\_log\_handler = RotatingFileHandler(  
 lens\_log\_file,  
 maxBytes=app.config[**"LOG\_MAX\_BYTES"**],  
 backupCount=app.config[**"LOG\_BACKUP\_COUNT"**]  
)  
lens\_log\_handler.setLevel(logging.DEBUG)  
lens\_formatter = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
lens\_log\_handler.setFormatter(lens\_formatter)  
lens\_logger = logging.getLogger(**"lens\_backend"**)  
lens\_logger.setLevel(logging.DEBUG)  
lens\_logger.addHandler(lens\_log\_handler)  
  
*# initialize pandas logger*pandas\_log\_file = os.path.join(app.config[**"PANDAS\_LOG\_DIR"**], **"pandas\_api.log"**)  
pandas\_log\_handler = RotatingFileHandler(  
 pandas\_log\_file,  
 maxBytes=app.config[**"PANDAS\_LOG\_MAX\_BYTES"**],  
 backupCount=app.config[**"PANDAS\_LOG\_BACKUP\_COUNT"**]  
)  
pandas\_log\_handler.setLevel(logging.DEBUG)  
pandas\_formatter = logging.Formatter(fmt=**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
 ' %(funcName)s:%(lineno)d'**)  
pandas\_log\_handler.setFormatter(pandas\_formatter)  
pandas\_logger = logging.getLogger(**"pandas\_api"**)  
pandas\_logger.setLevel(logging.DEBUG)  
pandas\_logger.addHandler(pandas\_log\_handler)  
  
celery = make\_celery(app)  
socketio = SocketIO(app, message\_queue=**'amqp://'**)  
  
db = SQLAlchemy(app)  
app.register\_blueprint(WEB\_SERVER)  
app.register\_blueprint(INSIGHTS\_SERVER)

*Filename: socket\_notify*

*"""Contains codes for all flask-socket io implementation."""***import** json  
**from** flask **import** jsonify, request  
**from** \_\_main\_\_ **import** socketio  
**from** insights.insights\_server **import** CLIENTSDICT, MESSAGESDICT  
  
  
*# maps userid with socketiosessionid  
# key user id , value is list of uuid(celery task)*@socketio.on(**'message'**, namespace=**'/mashup-service'**)  
**def** handle\_message(msg):  
 *"""client on connection sends message to this function"""* **try**:  
 output\_json = {**"status"**: **"Connected"**, **"sid"**: request.sid, **"user\_id"**: msg}  
 CLIENTSDICT[msg] = request.sid  
 **print**(**"socketnotify message :"** + str(socketio))  
 socketio.emit(**'message'**, json.dumps(output\_json), room=request.sid,  
 namespace=**'/mashup-service'**)  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@socketio.on(**'fetch'**, namespace=**'/mashup-service'**)  
**def** fetch\_notifications(user\_id):  
 *"""code to fetch notifications if user has was logged out"""* **try**:  
  
 CLIENTSDICT[user\_id] = request.sid  
 **if** MESSAGESDICT.get(user\_id) **is not** None:  
 list\_uuid = MESSAGESDICT.get(user\_id)  
 **for** uuid **in** list\_uuid:  
 sendjson = {}  
 sendjson[**"uuid"**] = uuid  
 sendjson[**"user\_id"**] = user\_id  
 **print**(**"socketnotify fetch :"** + str(socketio))  
 socketio.emit(**'ack-task'**, json.dumps(sendjson), room=CLIENTSDICT.get(user\_id),  
 namespace=**'/mashup-service'**)  
  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@socketio.on(**'ack'**, namespace=**'/mashup-service'**)  
**def** ack(msg):  
 *"""removes notifications if they have been acknowledged"""* **try**:  
  
 input\_json = json.loads(msg)  
 user\_id = input\_json[**'user\_id'**]  
 uuid = input\_json[**'uuid'**]  
 MESSAGESDICT[user\_id].remove(uuid)  
 **print**(user\_id + **"has received"**)  
  
 **except** Exception **as** ex: *# pylint: disable=broad-except* **return** jsonify({**"error"**: ex.args[0]})  
  
  
@socketio.on(**'disconnectSocket'**, namespace=**'/mashup-service'**)  
**def** test\_disconnect(user\_id):  
 **print**(**"client has disconnected"**)  
 **del** CLIENTSDICT[user\_id]

*Filename: start\_web*

*# -\*- coding: utf-8 -\*-  
"""  
Flask Api manager which will register the blueprint api  
"""***from** setup\_app **import** app, socketio  
**from** config\_manager **import** CONFIG\_MANAGER  
**from** authentication\_middleware.apple\_connect\_manager **import** authentication, authorization  
app.before\_request(authentication)  
*# app.before\_request(authorization)***if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 *# will contain all events related to flask-socket-io* **import** socketnotify *# do not remove this.* socketio.run(app, host=str(CONFIG\_MANAGER.get(**'SERVER\_CONFIG'**, **'SERVICE\_UP\_IP'**)), port=5000)

*Filename: test\_class\_creator*

*"""  
This module is used for create object of string formatted module and class name  
"""***import** unittest  
**from** class\_creator **import** ClassCreator  
  
  
**class** TestClassObjectCreator(unittest.TestCase):  
 *"""  
 This class contains test suite of classCreator.py file  
 """* **def** setUp(self):  
 **pass  
  
 def** tearDown(self):  
 **pass  
  
 def** test\_create\_object\_from\_class\_name(self):  
 *"""  
 This method is test suite for getClass method of classCreator class* **:return***:  
 """* **try**:  
 value = ClassCreator.get\_class(**"pandas\_api\_rest.PandasApi"**)  
 self.assertEqual(1, 1)  
 **except** Exception **as** e:  
 self.assertEqual(1, 2)

*Filename:test\_json\_parser\_utility*

*"""  
This module is used for test case json parse utility  
"""***import** json  
**import** unittest  
**from** json\_parser\_utility **import** JsonUtility  
  
  
**class** TestJsonParser(unittest.TestCase):  
 *"""  
 class for test case json parser  
 """* **def** setUp(self):  
 **with** open(**'../resources/datasource.json'**) **as** file\_reader:  
 self.input\_json = json.load(file\_reader)[**'fields'**]  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_find\_field\_value(self):  
 *"""  
 test case for find list of values of respective key* **:return***:  
 """* utility = JsonUtility()  
 results = utility.required\_key\_value\_from\_json(**'fieldName'**, self.input\_json)  
 self.assertEqual(results, [**'match\_id'**, **'inning'**,  
 **'batting\_team'**, **'bowling\_team'**, **'over'**, **'ball'**, **'batsman'**,  
 **'non\_striker'**, **'bowler'**, **'is\_super\_over'**, **'wide\_runs'**,  
 **'bye\_runs'**, **'legbye\_runs'**, **'noball\_runs'**, **'penalty\_runs'**,  
 **'batsman\_runs'**, **'extra\_runs'**, **'total\_runs'**, **'player\_dismissed'**,  
 **'dismissal\_kind'**, **'fielder'**])  
  
 **with** self.assertRaises(Exception):  
 utility.required\_key\_value\_from\_json(**'fieldName'**, **"string"**)  
  
 **def** test\_find\_value\_by\_matching(self):  
 *"""  
 test suit for find value by matching key and value* **:return***:  
 """* utility = JsonUtility()  
 results = utility.find\_value\_of\_matched\_key\_value(  
 **'fieldName'**, **'fieldType'**, **'string'**, self.input\_json)  
 self.assertEqual(21, len(results))  
  
 **with** self.assertRaises(Exception):  
 utility.find\_value\_of\_matched\_key\_value(  
 **'fieldName'**, **'fieldType'**, **'string'**, **"input\_json"**)  
  
 **def** test\_check\_key\_exist\_in\_json(self):  
 *"""  
 test suite for key exist in the json* **:return***:  
 """* input\_json = {**"timestamp"**: **"2018-10-16T07:12:47.922+0000"**,  
 **"status"**: 404,  
 **"error"**: **"Not Found"**,  
 **"message"**: **"Lens not found with id : '12112312321312'"**,  
 **"path"**: **"/insight-hub/lens-management/managed-lens/"  
 "lensId=12112312321312&dataId=12121"** }  
 status = JsonUtility.check\_key\_exist\_in\_dict(input\_json, **"error"**)  
 **if** status **is** True:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 input\_json.pop(**"error"**)  
 status = JsonUtility.check\_key\_exist\_in\_dict(input\_json, **"error"**)  
 **if** status **is** True:  
 self.assertEqual(1, 2)  
 **else**:  
 self.assertEqual(1, 1)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

*Filename: test\_pandas\_api\_rest*

*"""  
This module is used for pandas api testing  
"""***import** unittest  
**import** json  
**import** requests  
**import** pandas **as** pd  
**from** mock **import** Mock, patch  
**from** pandas\_api\_rest **import** PandasApi, PandasApiForDataFrame  
**from** dataframe\_reader.samples\_strategy.top\_n **import** TopNStrategy  
**import** constants  
  
  
**class** TestPandasApi(unittest.TestCase):  
 *"""  
 This class is used for pandasApi class test case  
 """* **def** setUp(self):  
 self.data\_source\_id = 1  
 filename = **'../resources/datasource.json'  
 with** open(filename, **'r'**) **as** file\_reader\_1:  
 self.datastore = json.load(file\_reader\_1)  
  
 self.data\_frame = pd.read\_csv(**"../resources/deliveries.csv"**, nrows=1000)  
 filename = **'../resources/step\_json.json'** self.step\_json = None  
 **with** open(filename, **'r'**) **as** file\_reader:  
 self.step\_json = json.load(file\_reader)  
  
 **def** tearDown(self):  
 **pass  
  
 def** test\_start\_workflow\_key\_error(self):  
 *"""  
 check if key is missing in input json* **:return***:  
 """* **try**:  
 self.step\_json.pop(constants.RECORD\_LIMIT)  
 PandasApi(self.step\_json)  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args[0])  
 self.assertEqual(1, 1)  
  
 **def** test\_start\_workflow(self):  
 *"""  
 Test suite for start\_workflow method of pandas api class* **:return***:  
 """* **with** patch.object(requests, **"get"**) **as** get\_request:  
 get\_request.return\_value = mock\_response = Mock()  
 mock\_response.text = self.datastore  
  
 **with** patch.object(TopNStrategy, **"read\_from\_configuration"**) **as** dataframe\_reader:  
 dataframe\_reader.return\_value = self.data\_frame  
 pandas\_api = PandasApi(self.step\_json)  
 value = pandas\_api.start\_workflow()  
 json\_value = json.loads(value)  
 self.assertEqual(len(json\_value), 3)  
  
 self.step\_json[**"applyOn"**] = **"ORIGINAL"** pandas\_api = PandasApi(self.step\_json)  
 value = pandas\_api.start\_workflow()  
 json\_value = json.loads(value)  
 self.assertEqual(len(json\_value), 3)  
  
 **def** test\_get\_dataframe\_column\_with\_types(self):  
 *"""  
 Test suite for start\_workflow method of PandasApiForDataFrame class* **:return***: json response  
 """* **with** patch.object(requests, **"get"**) **as** get\_request:  
 get\_request.return\_value = mock\_response = Mock()  
 mock\_response.text = self.datastore  
  
 **with** patch.object(TopNStrategy, **"read\_from\_configuration"**) **as** dataframe\_reader:  
 dataframe\_reader.return\_value = self.data\_frame  
 *# register dataframe - just called for test case* pandas\_api = PandasApi(self.step\_json)  
 pandas\_api.start\_workflow()  
 pandas\_api\_for\_df = PandasApiForDataFrame(self.step\_json)  
 value = pandas\_api\_for\_df.get\_dataframe\_column\_with\_types()  
 json\_value = json.loads(value)  
 self.assertEqual(len(json\_value), 2)  
  
 **def** test\_start\_workflow\_with\_strategy(self):  
 *"""  
 Test suite for start\_workflow method of pandas api class using Strategy* **:return***:  
 """  
 # added for strategy Top N* filename = **'../resources/strategy\_step\_json.json'** self.step\_json = None  
 **with** open(filename, **'r'**) **as** file\_reader:  
 self.step\_json = json.load(file\_reader)  
  
 **with** patch.object(requests, **"get"**) **as** get\_request:  
 get\_request.return\_value = mock\_response = Mock()  
 mock\_response.text = self.datastore  
  
 **with** patch.object(TopNStrategy, **"generate\_dataframe"**) **as** top\_n\_dataframe\_reader:  
 top\_n\_dataframe\_reader.return\_value = self.data\_frame  
 pandas\_api = PandasApi(self.step\_json)  
 value = pandas\_api.start\_workflow()  
 json\_value = json.loads(value)  
 self.assertEqual(len(json\_value), 3)  
  
 **def** test\_apply\_step\_on\_frame\_error(self):  
 **with** self.assertRaises(Exception):  
 emptyjson={}  
 PandasApiForDataFrame(emptyjson)  
  
 **with** self.assertRaises(Exception):  
 PandasApiForDataFrame(None)

*Filename:test\_request\_builder*

*"""  
This module is used for request api  
"""***import** unittest  
**import** json  
**import** requests  
**from** mock **import** patch, Mock  
**import** constants  
**from** request\_builder **import** RequestBuilder  
  
  
**class** TestRequestBuilder(unittest.TestCase):  
 *"""  
 This class is used for writing test case for RequestBuilder class.  
 """* **def** setUp(self):  
 self.data\_source\_id = 1  
 filename = **'../resources/datasource.json'  
 with** open(filename, **'r'**) **as** file\_reader:  
 self.datastore = json.load(file\_reader)  
  
 **def** test\_call\_http\_get\_request(self):  
 *"""  
 Test suite for RequestBuilder class* **:return***:  
 """* **with** patch.object(requests, **"get"**) **as** get\_request:  
 get\_request.return\_value = mock\_response = Mock()  
 mock\_response.json.return\_value = self.datastore  
 value = RequestBuilder().call\_http\_get\_request(constants.DATA\_SOURCE\_FETCH\_URL  
 + str(self.data\_source\_id), {}, {}).json()  
 **if** value == self.datastore:  
 self.assertEqual(1, 1)  
 **else**:  
 **print**(self.datastore)  
 **print**(value)  
 self.assertEqual(1, 2)  
  
 **try**:  
 get\_request.side\_effect = [  
 requests.exceptions.HTTPError(**'unable to make http request'**)]  
 RequestBuilder().call\_http\_get\_request(constants.DATA\_SOURCE\_FETCH\_URL  
 + str(self.data\_source\_id), {}, {})  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **try**:  
 get\_request.side\_effect = [  
 requests.exceptions.ConnectionError(**'unable to connect requested server'**)]  
 RequestBuilder().call\_http\_get\_request(constants.DATA\_SOURCE\_FETCH\_URL  
 + str(self.data\_source\_id), {}, {})  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **try**:  
 get\_request.side\_effect = [requests.exceptions.Timeout(**'connection time out'**)]  
 RequestBuilder().call\_http\_get\_request(constants.DATA\_SOURCE\_FETCH\_URL  
 + str(self.data\_source\_id), {}, {})  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **try**:  
 get\_request.side\_effect = [  
 requests.exceptions.RequestException(**'Invalid request input'**)]  
 RequestBuilder().call\_http\_get\_request(constants.DATA\_SOURCE\_FETCH\_URL  
 + str(self.data\_source\_id), {}, {})  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex)  
 self.assertEqual(1, 1)  
  
 **def** test\_call\_http\_post\_request(self):  
 *"""  
 Test suite for RequestBuilder class for post request* **:return***:  
 """* **with** patch.object(requests, **"post"**) **as** get\_request:  
 get\_request.return\_value = mock\_response = Mock()  
 mock\_response.response = self.datastore  
 value = RequestBuilder().call\_http\_post\_request(constants.DATA\_SOURCE\_FETCH\_URL  
 + str(self.data\_source\_id), {}, {})  
 **if** value.response == self.datastore:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_call\_http\_put\_request(self):  
 *"""  
 Test suite for RequestBuilder class for put request* **:return***:  
 """* **with** patch.object(requests, **"put"**) **as** get\_request:  
 get\_request.return\_value = mock\_response = Mock()  
 mock\_response.response = self.datastore  
 value = RequestBuilder().call\_http\_put\_request(constants.DATA\_SOURCE\_FETCH\_URL  
 + str(self.data\_source\_id), {}, {})  
 **if** value.response == self.datastore:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_call\_request\_url(self):  
 *"""  
 Test suite for RequestBuilder class of invalid request* **:return***:  
 """* **try**:  
 RequestBuilder().call\_request\_url(**'delete'**,  
 constants.DATA\_SOURCE\_FETCH\_URL  
 + str(self.data\_source\_id), {}, {})  
 self.assertEqual(1, 2)  
 **except** Exception **as** ex:  
 **print**(ex.args[0])  
 self.assertEqual(1, 1)

*Filename:test\_token\_registry*

*"""  
This module is used to test token\_registry module  
"""***import** unittest  
**from** token\_registry **import** TokenRegistry  
  
  
**class** TestTokenRegistry(unittest.TestCase):  
 *"""  
 This class contains the test suite for TokenRegistry class  
 """* **def** test\_register\_user\_and\_token(self):  
 *"""  
 this method will validate the register\_token\_with\_user\_id method* **:return***:  
 """* TokenRegistry.register\_token\_with\_user\_id(**"mosaic"**,  
 **"bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkI"**)  
 **if** TokenRegistry.token\_registry.get(**'mosaic'**) **is not** None:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)  
  
 **def** test\_fetch\_token\_using\_user\_id(self):  
 *"""  
 this method will test the get\_token\_for\_user method* **:return***:  
 """* TokenRegistry.token\_registry.update(  
 {**"mosaic"**:  
 **"bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkI"**})  
  
 value = TokenRegistry.get\_token\_for\_user(**"mosaic"**)  
 **if** value[**'Authorization'**] == **'bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkI'**:  
 self.assertEqual(1, 1)  
 **else**:  
 self.assertEqual(1, 2)

*Filename:token\_registry.py*

*"""  
This module is used to find register and find to token with respect to given user  
"""***class** TokenRegistry:  
 *"""  
 This class manager to get and put the token in registry  
 """* token\_registry = {}  
  
 @classmethod  
 **def** get\_token\_for\_user(cls, user\_id):  
 *"""  
 this method is used to registry* **:return***:  
 """* **return** {**"Authorization"**: TokenRegistry.token\_registry.get(user\_id)}  
  
 @classmethod  
 **def** register\_token\_with\_user\_id(cls, user\_id, access\_token):  
 *"""  
 This method is used register token with userid* **:return***:  
 """* TokenRegistry.token\_registry.update({user\_id: access\_token})

*Filename: web\_server.py*

*# -\*- coding: utf-8 -\*-  
"""  
This module is used for pandas web Service.  
  
"""***import** base64  
**import** datetime  
*# Added for gbiio***import** gbiio  
**import** json  
**import** logging  
**import** os  
**import** requests  
**import** threading  
**import** uuid  
**from** flask **import** g  
**from** flask **import** jsonify, Blueprint  
**from** flask **import** request  
**from** flask\_cors **import** cross\_origin  
**from** logging.handlers **import** RotatingFileHandler  
**from** save\_user\_config.user\_preferences **import** UserPreferences  
  
**import** constants  
**from** DAGJsonGenerator **import** DAGJsonGenerator  
**from** batch\_job.batch\_job\_runner **import** JobRunner  
**from** export\_dataframe.export\_to\_excel **import** ExportToExcel  
**from** expression.run\_expression **import** ExpressionRunner  
**from** mashup\_flows\_runner **import** FlowsMashupRunner  
**from** pandas\_api\_rest **import** PandasApi, PandasApiForDataFrame  
**from** profiling.generate\_profiling **import** GenerateProfile  
**from** rule\_as\_step.rule\_step **import** RuleStepDetails  
  
*# LOGGER = logging.getLogger('lens\_backend')  
  
#LOGGER = logging.getLogger('lens\_backend')  
# LOGGER = logging.getLogger(\_\_name\_\_)  
# LOGGER.setLevel(logging.DEBUG)  
# HANDLER = RotatingFileHandler('pandas\_api.log', maxBytes=20000000, backupCount=10)  
# FORMATTER = logging.Formatter(fmt='%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'  
# ' %(funcName)s:%(lineno)d')  
# HANDLER.setFormatter(FORMATTER)  
# LOGGER.addHandler(HANDLER)  
  
#LOGGER = logging.getLogger('pandas\_api')*LOGGER = logging.getLogger(**'lens\_backend'**)  
  
WEB\_SERVER = Blueprint(**'web\_server'**, \_\_name\_\_, url\_prefix=**"/mashup-service"**)  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/dag\_json\_workflow"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** run\_dag\_json\_manager():  
 *"""  
 This method is used to get DAG JSON  
 data frame in memory* **:return***:  
 """* **try**:  
 user\_name = g.user[**"prsId"**]  
 **print**(**"username: "** + user\_name)  
 input\_json = request.get\_json()  
 request.get\_json(force=True).update({**"user\_id"**: user\_name})  
 mashuprequest = request.get\_json()  
 gsfsdString= **r'{ "dataSourceId": "033ae311-45fa-485e-a0a5-64590ae83519", "name": "adhoc\_rep\_test\_ar003", "description": "test", "dataSourceType": "TERADATA", "prestoTableAlias": null, "expertUserId": null, "isProfilingEnabled": null, "isSampleEnabled": null, "optionallyEnclosedInDoubleQuotes": null, "compressionType": null, "schemaName": null, "dataSourceStatus": null, "totalRecords": 11, "totalSampleRecords": null, "size": null, "sourceType": "RDBMS", "subSourceType": "TERADATA", "groupId": "mosaic", "dsUniqueId": "adhoc\_rep\_test\_ar003", "connectionId": "048caf35-3870-43cf-a18b-e86faef23f8a", "isModified": 0, "createdBy": "admin", "updatedBy": null, "createdOn": 1558589822153, "updatedOn": 1558596362308, "dataNode": { "label": "adhoc\_rep\_test\_ar003", "children": [], "isPublished": false, "value": { "isDsNode": "true", "fileType": "adhoc\_rep\_test\_ar003", "rowTag": "", "delimiter": "", "dbName": "iTunes\_transaction\_core", "isHeader": "false", "escape": "", "quote": "", "ignore\_leading\_white\_spaces": "false", "skip\_lines": "", "strict\_quotes": "" } }, "category": { "categoryId": "00c06df0-1ba5-4921-b3d5-a899b4f4ba12", "name": "Marketing", "shortDescription": "marketing", "longDescription": "Marketing", "isActive": true, "createdBy": "Santhosh", "updatedBy": "Santhosh", "createdOn": 1558591622008, "updatedOn": 1558591622008 }, "project": [{ "projectId": "string", "name": "project-string", "description": "test", "isModified": false, "createdBy": "Santhosh", "updatedBy": null, "createdOn": 1558991702713, "updatedOn": 1558991702713 }, { "projectId": "840", "name": "project-840", "description": "test1", "isModified": false, "createdBy": "Padmil Khandelwal", "updatedBy": null, "createdOn": 1558589875728, "updatedOn": 1558589875828 } ], "subCategory": { "subCategoryId": "0f6c26da-174d-4550-b1be-23a4be467a4a", "name": "PreMarketing", "shortDescription": "pre", "longDescription": "string", "parent": { "categoryId": "00c06df0-1ba5-4921-b3d5-a899b4f4ba12", "name": "Marketing", "shortDescription": "marketing", "longDescription": "Marketing", "isActive": true, "createdBy": "Santhosh", "updatedBy": "Santhosh", "createdOn": 1558591622008, "updatedOn": 1558591622008 }, "isActive": true, "createdBy": "Santhosh", "updatedBy": "string", "createdOn": 1558591772885, "updatedOn": 1558591772885 }, "dataSourceLevelTag": [], "listOfExcelRanges": [], "listOfFieldMapping": [{ "fieldMappingId": "7fc1e6ad-b6ce-4ce1-80c4-981ab3d05d8c", "name": "EMP\_ID", "alias": null, "position": "0", "isPrimaryKey": null, "fieldDataType": "INTEGER", "replaceNullWith": null, "sampleRecords": [ "11", "6", "24", "89", "13", "16", "17", "29", "45", "72" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822177, "updatedOn": 1558589822177, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "1e5586a3-35ec-48e6-b207-693069cf0873", "name": "EMP\_NAME", "alias": null, "position": "1", "isPrimaryKey": null, "fieldDataType": "STRING", "replaceNullWith": null, "sampleRecords": [ "Gage", "Porter", "Keegan", "Brandon", "Harding", "Jerry", "Brenden", "Porter", "Nash", "Uriah" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822187, "updatedOn": 1558589822187, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "93c4943d-7673-4454-a425-871d0e7d323a", "name": "EMAIL", "alias": null, "position": "2", "isPrimaryKey": null, "fieldDataType": "STRING", "replaceNullWith": null, "sampleRecords": [ "semper.erat.in@egestasblandit.ca", "vulputate.lacus.Cras@In.com", "sociis.natoque@feugiat.co.uk", "Pellentesque@Crasegetnisi.ca", "non.cursus.non@aceleifendvitae.net", "mauris.id@ut.edu", "at@consequatenim.net", "tristique@lobortistellus.edu", "faucibus.ut.nulla@primisinfaucibus.net", "est.arcu.ac@aliquet.com" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822189, "updatedOn": 1558589822189, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "089dd706-124f-4290-8f74-b2e06262c7ec", "name": "contact\_no", "alias": null, "position": "3", "isPrimaryKey": null, "fieldDataType": "STRING", "replaceNullWith": null, "sampleRecords": [ "3887691298", "2110610235", "0346188410", "6884613003", "0240862034", "6207168074", "7046601273", "1227825228", "7348604411", "4402590891" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822190, "updatedOn": 1558589822190, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "f718af10-449f-435b-ac22-ef74049b4875", "name": "etl\_create\_batch\_sk", "alias": null, "position": "4", "isPrimaryKey": null, "fieldDataType": "STRING", "replaceNullWith": null, "sampleRecords": [ "1000", "1000", "1000", "1003", "1000", "1000", "1000", "1001", "1001", "1002" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822192, "updatedOn": 1558589822192, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "e439a40b-341f-4c8f-908b-91f0aad9f096", "name": "etl\_change\_batch\_sk", "alias": null, "position": "5", "isPrimaryKey": null, "fieldDataType": "STRING", "replaceNullWith": null, "sampleRecords": [ "1000", "1000", "1000", "1003", "1000", "1000", "1000", "1001", "1001", "1002" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822193, "updatedOn": 1558589822193, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "8df31e9d-28bd-4707-bc9e-d254ac770956", "name": "etl\_create\_ts", "alias": null, "position": "6", "isPrimaryKey": null, "fieldDataType": "TIMESTAMP", "replaceNullWith": null, "sampleRecords": [ "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822194, "updatedOn": 1558589822194, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "6477afa5-cf37-4947-8772-d88cdc895391", "name": "etl\_change\_ts", "alias": null, "position": "7", "isPrimaryKey": null, "fieldDataType": "TIMESTAMP", "replaceNullWith": null, "sampleRecords": [ "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0", "2014-10-24 12:06:00.0" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822196, "updatedOn": 1558589822196, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "df63d2a6-2de7-4814-a7b1-355db847d346", "name": "ETL\_ACTION\_CD", "alias": null, "position": "8", "isPrimaryKey": null, "fieldDataType": "STRING", "replaceNullWith": null, "sampleRecords": [ "I", "I", "I", "I", "I", "I", "I", "I", "I", "I" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822197, "updatedOn": 1558589822197, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "1ff0d439-57a3-4ab5-9371-6567d79230d8", "name": "CREATE\_DATE", "alias": null, "position": "9", "isPrimaryKey": null, "fieldDataType": "DATE", "replaceNullWith": null, "sampleRecords": [ "2014-04-01", "2014-04-01", "2014-04-02", "2014-04-05", "2014-04-01", "2014-04-01", "2014-04-02", "2014-04-02", "2014-04-03", "2014-04-05" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822198, "updatedOn": 1558589822198, "fieldLevelTag": [], "active": false, "deleted": false }, { "fieldMappingId": "57c92fbc-3278-4bbc-9e3e-f6bcd23d4f98", "name": "ETL\_PROCESS\_RUN\_ID", "alias": null, "position": "10", "isPrimaryKey": null, "fieldDataType": "STRING", "replaceNullWith": null, "sampleRecords": [ "2000", "1000", "3000", "3000", "2000", "2000", "2000", "1000", "2000", "2000" ], "format": null, "addVarIndicator": null, "preIngestion": null, "postIngestion": null, "preValueFrequency": null, "posValueFrequency": null, "precision": null, "scale": null, "parentFieldId": null, "version": null, "createdBy": null, "updatedBy": null, "createdOn": 1558589822198, "updatedOn": 1558589822198, "fieldLevelTag": [], "active": false, "deleted": false } ], "users": [] }'** *# temporary fix for incorrect schema name, have to verify where it came from  
 #Either gsdf returned us the wrong case , or we hardcoded it [pf-2019-10-23]* gsfsdString = gsfsdString.replace(**'iTunes\_transaction\_core'**, **'Itunes\_Transaction\_Core'**)  
 *# expects dbTable = "connectionId":"048caf35-3870-43cf-a18b-e86faef23f8a"  
 # "dbtable":"Itunes\_Transaction\_Core.adhoc\_rep\_test\_ar003"* gsfdsresponse = json.loads(gsfsdString) *# from gbi\_ds\_creds cassandra table* gsfconfigresponse = json.loads(**r'{ "status": "success", "requestId": null, "result": [{ "id": "048caf35-3870-43cf-a18b-e86faef23f8a", "sourceType": "RDBMS", "dbType": "Teradata", "dbCluster": "EDWUAT", "active": true, "createdBy": "2319661922", "creationDate": "2018-11-06T00:03:17Z", "datasource": "{\"connPoolType\":null,\"name\":\"GSF-048caf35-3870-43cf-a18b-e86faef23f8a\",\"jdbcJarPath\":null,\"jdbcDriverClass\":\"com.teradata.jdbc.TeraDriver\",\"jdbcUrl\":\"jdbc:teradata://edwuat.corp.apple.com/\",\"dcName\":null}", "groupids": [], "lastModifiedDate": "2018-11-06T00:03:17Z", "modifiedBy": "2319661922", "properties": {}, "dsUser": "itunes\_Transaction\_user", "dsPassword": "\*\*\*\*" }]}'**)  
  
 dagjson = DAGJsonGenerator(mashuprequest,gsfconfigresponse,gsfdsresponse).createDAGjson()  
 *# dedupedagjson = DAGJsonGenerator(mashupdeduperequest, gsfconfigresponse, gsfdsresponse).createDAGjson()* file\_path = **'./dag.json'** file\_path\_dedupe = **'./dedupedag.json'** file\_object = open(file\_path, **'w'**)  
 file\_object\_dedupe = open(file\_path\_dedupe, **'w'**)  
 json.dump(dagjson, file\_object)  
 **return** jsonify(dagjson)  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/store\_user\_preferred\_config"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** save\_user\_perferences():  
 *"""  
 This method is used get connection details* **:return***:  
 """* **try**:  
 config = request.get\_json(force=True)  
 user\_id = config[constants.CONN\_USER\_ID]  
 user\_pref = UserPreferences(config)  
 pref\_format = **""** + config[constants.CONN\_FORMAT] + **""** pref\_mode = **""** + config[constants.CONN\_MODE] + **""** name = **""** + config[constants.CONN\_NAME] + **""** updated\_date = **""** + datetime.datetime.now().strftime(**"%Y%m%d"**) + **""** lens\_id = **""** + config[constants.CONN\_LENS\_ID] + **""** user\_pref.insert\_or\_update\_user\_perferences(pref\_format, pref\_mode, name, updated\_date, lens\_id, user\_id)  
 **return** jsonify({**"success"**: **"User preferences values is updated in the table ."**})  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Unable to save user preferences for user : {}'**.format(user\_id), ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/lens\_mashup\_manager"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** lens\_mashup\_manager():  
 *"""  
 This method is used get mashup api request  
 independent of UI* **:return***:  
 """* **try**:  
 input\_json = request.get\_json(force=True)  
 user\_name = input\_json[**"user\_id"**]  
 LOGGER.debug(**"username: "** + user\_name)  
 project\_id = request.headers.get(**'X-Datalab-Project-Id'**)  
 flow\_mashup\_runner = FlowsMashupRunner(request.get\_json(),project\_id)  
 flowsmashupjson = flow\_mashup\_runner.run\_flows\_mashup\_job()  
 **return** jsonify(flowsmashupjson)  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception in flow mashup Runner : %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/apply\_workflow"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** apply\_workflow():  
 *"""  
 This method is used to load the pandas  
 data frame in memory* **:return***:  
 """* **try**:  
 *#user\_name = g.user["prsId"]* input\_json = request.get\_json(force=True)  
 user\_name = input\_json[**"user\_id"**]  
 *#LOGGER.debug("username: " + user\_name)  
 #input\_json.update({"user\_id": user\_name})* **if** input\_json.get(**'column\_with\_types'**, **''**):  
 pandas\_api\_for\_df = PandasApiForDataFrame(input\_json)  
 **return** jsonify(pandas\_api\_for\_df.get\_dataframe\_column\_with\_types())  
 pandas\_api = PandasApi(request.get\_json())  
 **return** jsonify(pandas\_api.start\_workflow())  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/validate\_expression"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** validate\_expression():  
 *"""  
 This method is used to load the pandas  
 data frame in memory* **:return***:  
 """* **try**:  
 *#user\_name = g.user["prsId"]  
 #request.get\_json().update({"user\_id": user\_name})* **return** jsonify(ExpressionRunner.validate\_expression(  
 request.get\_json()))  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/profiling\_column"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** run\_profiling\_based\_on\_column():  
 *"""  
 This method is used to load the pandas  
 data frame in memory* **:return***:  
 """* **try**:  
 *#user\_name = g.user["prsId"]  
 #request.get\_json().update({"user\_id": user\_name})* profile = GenerateProfile(request.get\_json())  
 value = profile.generate\_profile\_of\_request\_column()  
 **return** jsonify(value)  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/batch\_job"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** run\_batch\_job():  
 *"""  
 This method is used to load the pandas  
 data frame in memory* **:return***:  
 """* **try**:  
 *# McQueen Write using GBIIO starts here  
 # LOGGER.info("web server >> batch\_job >> starts here..")  
 # os.environ['GBIIO\_ENV'] = constants.GBIIO\_ENV  
 # os.environ['CATALOG\_URL'] = constants.CATALOG\_URL  
 # LOGGER.info("web server >> batch\_job >> GBIIO\_ENV :: " + gbiio.constants.Env.GBIIO\_ENV)  
 # #gbiio.set\_property(gbiio.constants.Context.PROJECT\_ID, '1185')* cookie\_name = constants.COOKIE\_NAME  
 cookie\_value = request.cookies.get(cookie\_name)  
 cookie = {cookie\_name: str(cookie\_value)}  
 *# try:  
 # LOGGER.info("web server >> batch\_job >> TOKEN\_GENERATION\_URL :: " + str(constants.TOKEN\_GENERATION\_URL))  
 # response = requests.get(constants.TOKEN\_GENERATION\_URL, cookies=cookie,  
 # verify=False)  
 # r\_json = response.json()  
 # gbiio\_Token = ''  
 # if(r\_json is not None):  
 # gbiio\_Token = str(r\_json.get('Token'))  
 # request.get\_json().update({constants.GBIIO\_TOKEN: gbiio\_Token})  
 # except Exception as ex:  
 # LOGGER.exception('web server >> batch\_job >> gbiio : %s', ex)  
 # LOGGER.info("web server >> batch\_job >> GBIIO\_TOKEN :: " + gbiio\_Token)  
 # LOGGER.info("web server >> batch\_job >> ends here..")  
 # # McQueen Write using GBIIO ends here* request.get\_json().update({constants.COOKIE: cookie})  
 job\_runner = JobRunner(request.get\_json())  
 threading.Thread(target=job\_runner.run\_batch\_job).start()  
 LOGGER.info(**"Lens is being published.."**)  
 **return** jsonify({**"success"**: job\_runner.lens\_name + **" is being published. \  
 You will get notified soon."**})  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/export\_to\_excel"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** export\_to\_excel():  
 *"""  
 This method is used to export dataframe as excel file  
 data frame in memory* **:return***: excel file  
 """* **try**:  
 *#user\_name = g.user["prsId"]  
 #request.get\_json().update({"user\_id": user\_name})* export\_to\_excel = ExportToExcel(request.get\_json())  
 **return** export\_to\_excel.get\_result()  
 **except** Exception **as** ex: *# pylint: disable=broad-except* LOGGER.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/get\_column\_mapping"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** get\_column\_mapping():  
 *"""  
 method to get column mapping for rule step* **:return***:  
 """* **try**:  
 *#user\_id = g.user["prsId"]* input\_json = request.get\_json()  
 *#input\_json.update({"user\_id": user\_id})* rules\_details = RuleStepDetails(input\_json)  
 **return** jsonify(rules\_details.get\_rule\_details())  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})  
  
@cross\_origin(WEB\_SERVER)  
@WEB\_SERVER.route(**"/apply\_job\_flow"**, methods=[**"POST"**, **"OPTIONS"**])  
**def** job\_flow():  
 *"""  
 This method is used to load the pandas  
 data frame in memory* **:return***:  
 """  
 # this is used for perfoming job acitivies* **try**:  
 **try**:  
 input\_json = request.get\_json(force=True)  
 user\_name = input\_json[**"user\_id"**]  
 *#project\_id = request.headers.get('X-Datalab-Project-Id')* cookie\_name = **"acack-uat"** cookie\_value\_dev = request.cookies.get(cookie\_name)  
 *#cookie\_value\_dev = ''* **except** Exception **as** ex:  
 LOGGER.exception(**'Exception in getting cookie\_value\_dev : %s'**, ex)  
  
 **"""if (cookie\_value\_dev is None):  
 LOGGER.info("cookie\_value\_dev is none")  
 else:  
 LOGGER.info("cookie\_value\_dev::::: ")  
 LOGGER.info(cookie\_value\_dev)"""** cookie = {cookie\_name: cookie\_value\_dev}  
 *# create job* createJobUrl = **"http://{}/scheduler/api/v1/job"**.format(**'scheduler-backend.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:5000'**)  
 *# raw token from config file ODg5Njk1ODA3NjYyOWMxNzExYjk5MWM0NTUyZTg= see datalab-scheduler/k8/secret-dev.yaml* authorization = **"Token {}"**.format(base64.b64decode(**"ODg5Njk1ODA3NjYyOWMxNzExYjk5MWM0NTUyZTg="**).decode(**'utf-8'**))  
 entityId = uuid.uuid4()  
 strEntityID = str(entityId)  
 LOGGER.info(**"user id::::"** + user\_name)  
 LOGGER.info(**"entity id::::"** + strEntityID)  
 data = {  
 **"cron\_expr"**: **"0 0 0 ? \* \*"**,  
 **"cron\_json"**: {},  
 **"description"**: **"flows job description"**,  
 **"entity\_id"**: strEntityID,  
 **"entity\_type"**: **"wranglerPublish"**,  
 **"in\_params"**: {},  
 **"name"**: **"Flows Jobs"**,  
 **"project\_id"**: **"1193"**,  
 **"project\_name"**: **"project-1193"**,  
 **"user\_id"**: user\_name  
 }  
 *##name* jsonString = json.dumps(data)  
 LOGGER.info(**"posting data to job service data: "** + jsonString)  
 LOGGER.info(**"URL to post to is "** + createJobUrl)  
 headers = {**"Authorization"**: authorization, **"x-datalab-user"**: user\_name, **'Content-type'**: **'application/json'**, **'Accept'**: **'application/json'**}  
 LOGGER.info(**"Headers "** + str(headers))  
 **try**:  
 LOGGER.info(**"JOB CREATE send post "** + datetime.datetime.now().strftime(**"%a, %d %B %Y %H:%M:%S"**))  
 res = requests.post(createJobUrl, data=jsonString, headers=headers, cookies=cookie)  
 LOGGER.info(**"RESPONSE JOB CREATE "** + datetime.datetime.now().strftime(**"%a, %d %B %Y %H:%M:%S"**))  
 LOGGER.info(str(res))  
 *#LOGGER.info(res.content)* responseJson = json.loads(res)  
 jobID = responseJson[**"job\_id"**]  
 LOGGER.info(**"RESPONSE RECIEVED"** + jobID)  
 **except** Exception **as** ex:  
 **print**(ex)  
 LOGGER.exception(**'Exception in response from create job : %s'**, ex)  
  
  
 LOGGER.info(**"Posting to job run"**)  
 *# create job run* jobID = **'65069469-b001-4929-aa8a-2e28bd1b19d1'** *# hardcode for now....to be removed once we're getting responsese* runId = uuid.uuid4()  
 strrunId = str(runId)  
 LOGGER.info(**"run id generated:"**+ strrunId)  
 jobRunURL = createJobUrl + **'/'** + jobID + **'/run'** jobRunData = {**'comments'**: **'job run'**, **'out\_params'**: {  
 **'LOG\_PATH'**: **'http://store-test.blobstore.apple.com/batch-jobs/sivaranjin/fwkqatest/Sleepipynb-3e107e32-c615-45f2-8b00-3ce957d6b5e4/20191106183000/'**},  
 **'retry'**: 0, **'run\_id'**: strrunId, **'env\_cd'**: **'datalab-dev'**, **'status\_cd'**: **'JOB\_STARTED'**}  
 jobRunJsonString = json.dumps(jobRunData)  
 LOGGER.info(**"posting data to job service data:"** + jobRunJsonString)  
 LOGGER.info(**"URL to post to is "** + jobRunURL)  
 **try**:  
 jobRunResp = requests.post(jobRunURL, data=json.dumps(jobRunData), cookies=cookie, headers=headers)  
 LOGGER.info(**"job run respons"** + str(jobRunResp))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception in response from create job run : %s'**, ex)  
 LOGGER.info(**"create job run ends"**)  
  
 PARAMS = {**'project'**:**'1193'**}  
 *# delete job  
 #headers = {'Content-type': 'application/json', 'Accept': 'application/json', 'project': '992'}* headers = {**"Authorization"**: authorization, **"x-datalab-user"**: user\_name, **'Content-type'**: **'application/json'**, **'Accept'**: **'application/json'**}  
 jobDeleteUrl = createJobUrl + **'/'** + jobID  
 **try**:  
 delResp = requests.get(jobDeleteUrl, cookies=cookie, headers=headers,params=PARAMS)  
 LOGGER.info(**"job delete respons"** + str(delResp))  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception in response from delete job : %s'**, ex)  
  
 LOGGER.info(**"job related activities done::::"**)  
 **return** jobID  
 **except** Exception **as** ex:  
 LOGGER.exception(**'Exception: %s'**, ex)  
 **return** jsonify({**"error"**: ex.args[0]})

*Filename: Wrangler\_publish\_logger*

**import** logging  
**from** logging **import** FileHandler  
**from** logging **import** Formatter  
  
**"""  
This method is used to return dynamic logger object.  
"""  
def** getLogger(logFileName):  
  
 LOG\_FORMAT = (**'%(asctime)s %(levelname)-8s %(message)s - %(pathname)s -'' %(funcName)s:%(lineno)d'**)  
 LOG\_LEVEL = logging.DEBUG  
 PUBLISH\_LOG\_FILE = logFileName  
 publish\_logger = logging.getLogger(**"wrangler\_publish\_log"**)  
 publish\_logger.setLevel(LOG\_LEVEL)  
 publish\_logger\_file\_handler = FileHandler(PUBLISH\_LOG\_FILE)  
 publish\_logger\_file\_handler.setLevel(LOG\_LEVEL)  
 publish\_logger\_file\_handler.setFormatter(Formatter(LOG\_FORMAT))  
 publish\_logger.addHandler(publish\_logger\_file\_handler)  
 **return** publish\_logger  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 getLogger(**'workflow.log'**)

*Foldername:src*

*Filename:setup.py*

*# -\*- coding: utf-8 -\*-***from** setuptools **import** find\_packages, setup  
  
  
setup(  
 name=**"mashup-backend"**,  
 version=**"1.0.0"**,  
 description=**"Backend API's for GBI Data Lab"**,  
 author=**"Aaditya Agrawal"**,  
 author\_email=**"aaditya\_agrawal@apple.com"**,  
 classifiers=[  
 **"Programming Language :: Python :: 3.6"**,  
 ],  
 packages=find\_packages(),  
 include\_package\_data=True,  
 install\_requires=[  
 **"vine==1.3.0"**, *#Adding vine dependency line.* **"alembic==1.1.0"**, *#Changed from (1.0.8)* **"antlr4-python3-runtime==4.7.2"**,  
 *#"AppleConnect==0.7.0",* **"celery==4.3.0"**,  
 **"Flask==1.0.2"**,  
 **"Flask-Cors==3.0.7"**,  
 **"Flask-Migrate==2.4.0"**,  
 **"Flask-SocketIO==3.3.2"**,  
 **"Flask-SQLAlchemy==2.3.2"**,  
 **"pandas==0.23.4"**,  
 **"numpy==1.16.0"**,  
 **"pandas-profiling==1.4.1"**,  
 **"pandasql==0.7.3"**,  
 **"patsy==0.5.1"**,  
 **"PyHive==0.6.1"**,  
 **"pyramid-arima==0.8.1"**,  
 **"requests==2.21.0"**,  
 **"sklearn==0.0"**,  
 **"SQLAlchemy==1.3.2"**,  
 **"statsmodels==0.9.0"**,  
 **"thrift==0.11.0"**,  
 **"xgboost==0.80"**,  
 **"scipy==1.2.1"**,  
 **"impyute==0.0.8"**,  
 **"pmdarima==1.2.1"**,  
 **"boto3==1.9.233"**,  
 **"botocore==1.12.233"**,  
 **"jmespath==0.9.4"**,  
 **"s3transfer==0.2.1"**,  
 **"jsonpath\_ng==1.4.2"**,  
 **"gbiio==1.0.35"**,  
 **"cx-Oracle==7.0.0"**,  
 **"ldap3==2.5.1"**,  
 **"python-engineio==3.10.0"**,  
 ],  
)

*Foldername: Datalab-wrangler-service*

***Filename: Build.yaml***

**apiVersion**: v5  
  
**component**: mashup-backend  
**deploy**:  
  
 **sonar** : true  
  
 **dev**:  
 **config**:  
 **sequence**:  
 - secrets-dev.yaml  
 - service.yaml  
 - deployment.yaml  
 **params**:  
 **REPLICAS**: 1  
 **REVISION\_HISTORY\_LIMIT**: 1  
 **STRATEGY\_TYPE**: RollingUpdate  
 **APC\_CLUSTER\_NAME**: usrno3  
 **REPLICAS\_MASHUP**: 1  
 **REPLICAS\_MASHUP\_BEAT**: 1  
 **REPLICAS\_MASHUP\_WORKER**: 1  
 **SPLUNK\_CLUSTER**: ist\_test  
 **SPLUNKFORWARDER\_IMAGE**: docker.apple.com/splunk-ist/splunk-universalforwarder:7.2.9.1-2@sha256:2fbbda2463a5afb65a0ac3b4105fcf082c17cf755d3634de6c19cc3088b49d46  
  
 **it**:  
 **config**:  
 **sequence**:  
 - secrets-qa.yaml  
 - service.yaml  
 - deployment.yaml  
 **params**:  
 **REPLICAS**: 1  
 **REVISION\_HISTORY\_LIMIT**: 1  
 **STRATEGY\_TYPE**: RollingUpdate  
 **APC\_CLUSTER\_NAME**: usrno3  
 **REPLICAS\_MASHUP**: 1  
 **REPLICAS\_MASHUP\_BEAT**: 1  
 **REPLICAS\_MASHUP\_WORKER**: 1  
 **SPLUNK\_CLUSTER**: ist\_test  
 **SPLUNKFORWARDER\_IMAGE**: docker.apple.com/splunk-ist/splunk-universalforwarder:7.2.9.1-2@sha256:2fbbda2463a5afb65a0ac3b4105fcf082c17cf755d3634de6c19cc3088b49d46  
  
 **uat**:  
 **config**:  
 **sequence**:  
 - secrets-test.yaml  
 - service.yaml  
 - deployment.yaml  
 **params**:  
 **REPLICAS**: 1  
 **REVISION\_HISTORY\_LIMIT**: 1  
 **STRATEGY\_TYPE**: RollingUpdate  
 **APC\_CLUSTER\_NAME**: usrno3  
 **REPLICAS\_MASHUP**: 1  
 **REPLICAS\_MASHUP\_BEAT**: 1  
 **REPLICAS\_MASHUP\_WORKER**: 1  
 **SPLUNK\_CLUSTER**: ist\_test  
 **SPLUNKFORWARDER\_IMAGE**: docker.apple.com/splunk-ist/splunk-universalforwarder:7.2.9.1-2@sha256:2fbbda2463a5afb65a0ac3b4105fcf082c17cf755d3634de6c19cc3088b49d46  
  
*# prod-usmsc1:  
# config:  
# sequence:  
# - service.yaml  
# - config-usmsc1.yaml  
# - deployment.yaml  
# params:  
# REPLICAS: 1  
# REVISION\_HISTORY\_LIMIT: 1  
# STRATEGY\_TYPE: Recreate  
# prod-usuqo1:  
# config:  
# sequence:  
# - service.yaml  
# - config-usuqo1.yaml  
# - deployment.yaml  
# params:  
# REPLICAS: 1  
# REVISION\_HISTORY\_LIMIT: 1  
# STRATEGY\_TYPE: Recreate*

*Filename : Dockerfile*

*# base image***FROM** docker.apple.com/gbi-fw/notebook-templates/python:3.12-oracle  
  
*# switch to root***USER** root  
  
*# create app directories***RUN** mkdir /app  
**RUN** mkdir /logs  
  
**RUN** yum update -y && \  
yum install -y wget && \  
yum install -y java-1.7.0-openjdk java-1.7.0-openjdk-devel && \  
yum clean all  
  
**ENV** JAVA\_HOME /usr/lib/jvm/java-1.7.0-openjdk/  
  
*# copy app***COPY** src /app  
**COPY** src/mashup\_apis/ /app  
  
*#COPY migrations /app/migrations  
#COPY gunicorn.py /app/gunicorn.py  
  
# update permissions***RUN** chown -R gbi-datalab:gbi-datalab /app  
**RUN** chown -R gbi-datalab:gbi-datalab /logs  
  
**ADD** k8/entrypoint.sh /entrypoint.sh  
**RUN** chown -R gbi-datalab:gbi-datalab /entrypoint.sh  
**RUN** chmod 777 /entrypoint.sh  
*# switch back to gbi-datalab***USER** gbi-datalab  
  
*# define working directory***WORKDIR** /app  
  
*# install python dependencies***RUN** pip install -e /app --index-url=https://pypi.apple.com/simple/  
**RUN** pip install --no-cache-dir python@git+https://6bc5b762071de74e0ba4f8f8ddf6c0f63c765708@github.pie.apple.com/gbi-bigdata/datalab.git@datalab\_wrangler\_develop#subdirectory=datalab-connectors-service/python --index-url=https://pypi.apple.com/simple/  
  
**WORKDIR** /app/mashup\_apis  
  
*# expose ports***EXPOSE** 5000  
  
*# start the app  
#CMD ["python", "start\_web.py"]***ENTRYPOINT** /bin/bash /entrypoint.sh  
  
*# define volumes  
#VOLUME /logs*

*Filename : Gbi.yaml*

*# for details on how to write the gbi.yaml please refer to https://connectme.apple.com/docs/DOC-1449449***apiVersion**: v5  
**appname**: mashup-backend  
**execute**:  
 **build**:  
 **type**: docker  
 **repo**:  
 **docker**: docker.apple.com  
 **version**:  
 **kind**: version.txt  
 **deploy**:  
 **namespace**: gbi-datascience-platform  
 **config**:  
 - **branch**: datalab\_wrangler\_develop  
 **env**: dev  
 **zone**: usrno3  
 **namespace**: gbi-datascience-platform-dev  
 **path**: k8  
 **sequence**:  
 - secrets-dev.yaml  
 - service.yaml  
 - deployment.yaml  
 **params**:  
 - **name**: DATA\_CENTRE  
 **value**: datalab-dev  
 - **name**: NAMESPACE  
 **value**: gbi-datascience-platform-dev  
 - **name**: APC\_CLUSTER\_NAME  
 **value**: usrno3  
 - **name**: SPLUNK\_CLUSTER  
 **value**: ist\_test  
 - **name**: REPLICAS\_MASHUP  
 **value**: 1  
 - **name**: REPLICAS\_MASHUP\_BEAT  
 **value**: 1  
 - **name**: REPLICAS\_MASHUP\_WORKER  
 **value**: 1  
 - **name**: REVISION\_HISTORY\_LIMIT  
 **value**: 1  
 - **name**: STRATEGY\_TYPE  
 **value**: Recreate  
 - **name**: SPLUNKFORWARDER\_IMAGE  
 **value**: docker.apple.com/splunk-ist/splunk-universalforwarder:7.2.9.1-2@sha256:2fbbda2463a5afb65a0ac3b4105fcf082c17cf755d3634de6c19cc3088b49d46  
  
 - **branch**: datalab\_wrangler\_develop  
 **env**: it  
 **zone**: usrno3  
 **namespace**: gbi-datascience-platform-qa  
 **path**: k8  
 **sequence**:  
 - secrets-qa.yaml  
 - service.yaml  
 - deployment.yaml  
 **params**:  
 - **name**: DATA\_CENTRE  
 **value**: datalab-dev  
 - **name**: NAMESPACE  
 **value**: gbi-datascience-platform-qa  
 - **name**: APC\_CLUSTER\_NAME  
 **value**: usrno3  
 - **name**: SPLUNK\_CLUSTER  
 **value**: ist\_test  
 - **name**: REPLICAS\_MASHUP  
 **value**: 1  
 - **name**: REPLICAS\_MASHUP\_BEAT  
 **value**: 1  
 - **name**: REPLICAS\_MASHUP\_WORKER  
 **value**: 1  
 - **name**: REVISION\_HISTORY\_LIMIT  
 **value**: 1  
 - **name**: STRATEGY\_TYPE  
 **value**: Recreate  
 - **name**: SPLUNKFORWARDER\_IMAGE  
 **value**: docker.apple.com/splunk-ist/splunk-universalforwarder:7.2.9.1-2@sha256:2fbbda2463a5afb65a0ac3b4105fcf082c17cf755d3634de6c19cc3088b49d46  
  
 - **branch**: develop  
 **env**: uat  
 **zone**: usrno3  
 **namespace**: gbi-datascience-platform-test  
 **path**: k8  
 **sequence**:  
 - secrets-test.yaml  
 - service.yaml  
 - deployment.yaml  
 **params**:  
 - **name**: DATA\_CENTRE  
 **value**: datalab-dev  
 - **name**: NAMESPACE  
 **value**: gbi-datascience-platform-test  
 - **name**: APC\_CLUSTER\_NAME  
 **value**: usrno3  
 - **name**: SPLUNK\_CLUSTER  
 **value**: ist\_test  
 - **name**: REPLICAS\_MASHUP  
 **value**: 1  
 - **name**: REPLICAS\_MASHUP\_BEAT  
 **value**: 1  
 - **name**: REPLICAS\_MASHUP\_WORKER  
 **value**: 1  
 - **name**: REVISION\_HISTORY\_LIMIT  
 **value**: 1  
 - **name**: STRATEGY\_TYPE  
 **value**: Recreate  
 - **name**: SPLUNKFORWARDER\_IMAGE  
 **value**: docker.apple.com/splunk-ist/splunk-universalforwarder:7.2.9.1-2@sha256:2fbbda2463a5afb65a0ac3b4105fcf082c17cf755d3634de6c19cc3088b49d46  
  
*# - branch: master  
# env: prod # env to which  
# zone: usuqo1  
# namespace: gbi-datascience-platform  
# path: k8  
# file:  
# sequence:  
# - config-prod-ini.yaml  
# - config-prod.yaml  
# - service.yaml  
# - deployment.yaml  
# params:  
# - name: DATA\_CENTRE  
# value: datalab-dev  
# - name: APC\_CLUSTER\_NAME  
# value: usrno3  
# - name: SPLUNK\_CLUSTER  
# value: ist\_test  
# - name: REPLICAS\_MASHUP  
# value: 1  
# - name: REPLICAS\_MASHUP\_BEAT  
# value: 1  
# - name: REPLICAS\_MASHUP\_WORKER  
# value: 1  
# - name: REVISION\_HISTORY\_LIMIT  
# value: 1  
# - name: STRATEGY\_TYPE  
# value: Recreate  
# - name: SPLUNKFORWARDER\_IMAGE  
# value: docker.apple.com/splunk-ist/splunk-universalforwarder:7.2.9.1-2@sha256:2fbbda2463a5afb65a0ac3b4105fcf082c17cf755d3634de6c19cc3088b49d46  
# - name: dockerpath  
# value: docker.apple.com/gbi/uat/mashup-backend:2.0.6-Release  
#  
# - branch: master  
# env: prod # env to which  
# zone: usmsc1  
# namespace: gbi-datascience-platform  
# path: k8  
# file:  
# sequence:  
# - config-prod-ini.yaml  
# - config-prod.yaml  
# - service.yaml  
# - deployment.yaml  
# params:  
# - name: DATA\_CENTRE  
# value: datalab-dev  
# - name: APC\_CLUSTER\_NAME  
# value: usrno3  
# - name: SPLUNK\_CLUSTER  
# value: ist\_test  
# - name: REPLICAS\_MASHUP  
# value: 1  
# - name: REPLICAS\_MASHUP\_BEAT  
# value: 1  
# - name: REPLICAS\_MASHUP\_WORKER  
# value: 1  
# - name: REVISION\_HISTORY\_LIMIT  
# value: 1  
# - name: STRATEGY\_TYPE  
# value: Recreate  
# - name: SPLUNKFORWARDER\_IMAGE  
# value: docker.apple.com/splunk-ist/splunk-universalforwarder:7.2.9.1-2@sha256:2fbbda2463a5afb65a0ac3b4105fcf082c17cf755d3634de6c19cc3088b49d46  
# - name: dockerpath  
# value: docker.apple.com/gbi/uat/mashup-backend:2.0.6-Release* **mail**:  
 **group**: gbi-mosaic-developers@group.apple.com

***Filename: Makefile***

all : secret-dev secret-qa secret-test  
  
configmap-dev:  
 kubectl create configmap mashup-backend-config --from-file=lens\_appleconnect.py=src/mashup\_apis/configs/lens\_appleconnect\_dev.py -o yaml --dry-run > k8/config-dev.yaml  
  
configmap-dev-ini:  
 kubectl create configmap mashup-backend-ini --from-file=apiconfig.ini=src/mashup\_apis/apiconfig-dev.ini -o yaml --dry-run > k8/config-dev-ini.yaml  
  
configmap-qa:  
 kubectl create configmap mashup-backend-config --from-file=lens\_appleconnect.py=src/mashup\_apis/configs/lens\_appleconnect\_qa.py -o yaml --dry-run > k8/config-qa.yaml  
  
configmap-qa-ini:  
 kubectl create configmap mashup-backend-ini --from-file=apiconfig.ini=src/mashup\_apis/apiconfig-qa.ini -o yaml --dry-run > k8/config-qa-ini.yaml  
  
configmap-test:  
 kubectl create configmap mashup-backend-config --from-file=lens\_appleconnect.py=src/mashup\_apis/configs/lens\_appleconnect\_test.py -o yaml --dry-run > k8/config-test.yaml  
  
configmap-test-ini:  
 kubectl create configmap mashup-backend-ini --from-file=apiconfig.ini=src/mashup\_apis/apiconfig-test.ini -o yaml --dry-run > k8/config-test-ini.yaml  
  
secret-dev:  
 kubectl create secret generic mashup-backend --from-file=lens\_appleconnect.py=src/mashup\_apis/configs/lens\_appleconnect\_dev.py --from-file=apiconfig.ini=src/mashup\_apis/apiconfig-dev.ini --from-literal=DATALAB\_APP\_ID=136927 --from-literal=DATALAB\_APP\_PASSWORD=pxr767tssuvgaix4 --from-literal=DATALAB\_APP\_CONTEXT=datalab --from-literal=CATALOG\_APP\_ID=136847 --from-literal=APP\_TOKEN\_GENERATION\_URL='https://idmsac-uat.corp.apple.com/auth/apptoapp/token/generate' --from-literal=S3\_ID='51b20ee0-2189-11e8-b716-09d7c9dd2f27' --from-literal=CATALOG\_URL='https://gbiio-service:443/virtualedw' --from-literal=CATALOG\_OTHER\_APP\_ID='136927' --from-literal=CATALOG\_APP\_CONTEXT='datalab' -o yaml --dry-run > k8/secrets-dev.yaml  
  
secret-qa:  
 kubectl create secret generic mashup-backend --from-file=lens\_appleconnect.py=src/mashup\_apis/configs/lens\_appleconnect\_qa.py --from-file=apiconfig.ini=src/mashup\_apis/apiconfig-qa.ini --from-literal=DATALAB\_APP\_ID=136927 --from-literal=DATALAB\_APP\_PASSWORD=pxr767tssuvgaix4 --from-literal=DATALAB\_APP\_CONTEXT=datalab --from-literal=CATALOG\_APP\_ID=136847 --from-literal=APP\_TOKEN\_GENERATION\_URL='https://idmsac-uat.corp.apple.com/auth/apptoapp/token/generate' --from-literal=S3\_ID='51b20ee0-2189-11e8-b716-09d7c9dd2f27' --from-literal=CATALOG\_URL='https://gbiio-service:443/virtualedw' --from-literal=CATALOG\_OTHER\_APP\_ID='136927' --from-literal=CATALOG\_APP\_CONTEXT='datalab' -o yaml --dry-run > k8/secrets-qa.yaml  
  
secret-test:  
 kubectl create secret generic mashup-backend --from-file=lens\_appleconnect.py=src/mashup\_apis/configs/lens\_appleconnect\_test.py --from-file=apiconfig.ini=src/mashup\_apis/apiconfig-test.ini --from-literal=DATALAB\_APP\_ID=136927 --from-literal=DATALAB\_APP\_PASSWORD=pxr767tssuvgaix4 --from-literal=DATALAB\_APP\_CONTEXT=datalab --from-literal=CATALOG\_APP\_ID=136847 --from-literal=APP\_TOKEN\_GENERATION\_URL='https://idmsac-uat.corp.apple.com/auth/apptoapp/token/generate' --from-literal=S3\_ID='51b20ee0-2189-11e8-b716-09d7c9dd2f27' --from-literal=CATALOG\_URL='https://gbiio-service:443/virtualedw' --from-literal=CATALOG\_OTHER\_APP\_ID='136927' --from-literal=CATALOG\_APP\_CONTEXT='datalab' -o yaml --dry-run > k8/secrets-test.yaml

*filename : requirements.txt*

alembic==1.0.8  
antlr4-python3-runtime==4.7.2  
AppleConnect==0.7.0  
celery==4.3.0  
Flask==1.0.2  
Flask-Cors==3.0.7  
Flask-Migrate==2.4.0  
Flask-SocketIO==3.3.2  
Flask-SQLAlchemy==2.3.2  
pandas==0.24.2  
pandas-profiling==1.4.1  
pandasql==0.7.3  
patsy==0.5.1  
PyHive==0.6.1  
pyramid-arima==0.8.1  
requests==2.21.0  
sklearn==0.0  
SQLAlchemy==1.3.2  
statsmodels==0.9.0  
thrift==0.11.0  
xgboost==0.80

filename: version.txt

1.0.40-SNAPSHOT

*Filename: Readme.md*

***# Prerequisites*\*** python > 3.5  
**\*** pipenv  
  
***# Creating environment****```  
pipenv install  
```****# Installing depedencies***sudo pip3.5 install sanic  
sudo pip3.5 install sanic\_cors  
sudo pip3.5 install pyhive  
sudo pip3.5 install thrift  
sudo pip3.5 install pandas  
sudo pip3.5 install requests  
sudo pip3.5 install sasl  
sudo pip3.5 install thrift\_sasl  
sudo pip3.5 install antlr4-python3-runtime  
sudo pip3.5 install mock  
sudo apt-get install python3-mysqldb  
sudo pip3.5 install sklearn  
sudo pip3.5 install scipy  
  
*```  
make install  
```****#Starting the application****```  
python3.5 -m sanic start\_web.app --host=0.0.0.0 --port=8011  
```****#Running all the unit test cases***'''  
python3 -m unittest  
'''

*Foldername:k8*

*Filename: Allow-mcqueen-netpol.yaml*

**apiVersion**: networking.k8s.io/v1  
**kind**: NetworkPolicy  
**metadata**:  
 **name**: allow-mcqueen  
 **labels**:  
 **component**: gbi-datalab  
**spec**:  
 **podSelector**:  
 **matchLabels**:  
 **app**: mashup-backend  
 **policyTypes**:  
 - Egress  
 **egress**:  
 - **to**:  
 - **ipBlock**:  
 **cidr**: 17.132.20.0/23  
 - **ipBlock**:  
 **cidr**: 17.132.20.128/26  
 - **ipBlock**:  
 **cidr**: 17.132.22.0/24  
 **ports**:  
 - **port**: 443  
 **protocol**: TCP  
 - **port**: 80  
 **protocol**: TCP

*Filename: config-dev.yaml*

**apiVersion**: v1  
**data**:  
 **lens\_appleconnect.py**: |-  
 #! -\*- coding: utf-8 -\*-  
 # pylint: skip-file  
  
 import logging  
 import constants  
  
 # flask app  
 TESTING = True  
 DEBUG = True  
  
 # context  
 URL\_PREFIX = "/wrangler/api"  
  
 # security  
 SECRET\_KEY = "8896958076629c1711b991c4552e8c6616b9e2a94a2fcb7c038fd10b049f30ce"  
  
 # log  
 LOG\_DIR = "/tmp"  
 LOG\_LEVEL = logging.DEBUG  
 LOG\_BACKUP\_COUNT = 10  
 LOG\_MAX\_BYTES = 20000  
  
 # Pandas log dir  
 PANDAS\_LOG\_DIR = "/logs"  
 PANDAS\_LOG\_LEVEL = logging.DEBUG  
 PANDAS\_LOG\_BACKUP\_COUNT = 10  
 PANDAS\_LOG\_MAX\_BYTES = 20000  
  
 # celery  
 CELERY\_BROKER\_URL = "amqp://guest:guest@localhost:5672"  
 CELERY\_RESULT\_BACKEND = constants.CELERY\_RESULTS\_BACKEND  
  
 # database connection  
 SQLALCHEMY\_DATABASE\_URI = "oracle+cx\_oracle://GBI\_DATA\_LAB\_USER:C0dd146t0n7\_Bri7fca57\_Ej2jb2@mr2-aixd-av06.rno.apple.com:1699/?service\_name=gia3d"  
 SQLALCHEMY\_TRACK\_MODIFICATIONS = False  
  
 # apple connect  
 APPLE\_CONNECT\_APPLICATION\_ID = "136927"  
 APPLE\_CONNECT\_APPLICATION\_KEY = "e8d2d95f4e0d4de77fa357a2a144fb886b0f63059dae362b54c0f3c8b5701f6d"  
 APPLE\_CONNECT\_ADMIN\_PASSWORD = "dq92x8ci2pwn7fxt"  
 APPLE\_CONNECT\_ENVIRONMENT = "UAT"  
 APPLE\_CONNECT\_RETURN\_PATH = "/notebooks/api/docs"  
 APPLE\_CONNECT\_RETURN\_DOMAIN = "https://gbi-datalab.corp.apple.com:5000/"  
 APPLE\_CONNECT\_COOKIE\_NAME = "acack-uat"  
 REDIRECT\_RV=1  
  
 # skip auth for following URI  
 SKIP\_AUTH = (  
 ".\*/api/flasgger\_static.\*",  
 ".\*/api/docs.\*",  
 ".\*/metrics",  
 ".\*/github/.\*",  
 ".\*/progress"  
 )  
  
 # client ip - used only for dev purpose  
 CLIENT\_IP = "17.233.118.137"  
  
 # mosaic\_backend\_url  
 MOSAIC\_BACKEND = {  
 "endpoints": {  
 "list-users": "http://mosaicweb-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080/datalabweb-api/userManagement/getUsers",  
 "user-projects": "http://mosaicweb-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080/datalabweb-api/user/projects",  
 "user-projects-v2" : "http://mosaicweb-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080/datalabweb-api/v2/user/projects",  
 },  
 "headers": {  
 "authorization": "Token 8QHfhMt+IybfK7XAICiAXPqsJ8KjwGiaHMaLK6id/CQ=",  
 "content\_type": "application/json",  
 }  
 }  
  
 # external apps  
 EXTERNAL\_APPS = {  
 "gbiml-client": {  
 "token": "oa9sduaksdqweqkwur076629werwer3232e8"  
 },  
 "mosaic-web": {  
 "token": "j+1n1spp4xq/9+bofdv/1sqd4x0c++uo4lrdto47oqc=",  
 "prs\_id": "mosaic-web",  
 "email\_address": "mosaic-web@gbi-datalab",  
 "first\_name": "Mosaic",  
 "last\_name": "Web",  
 }  
 }  
**kind**: ConfigMap  
**metadata**:  
 **creationTimestamp**: null  
 **name**: mashup-backend-config

*Filename:config-dev-ini.yaml*

**apiVersion**: v1  
**data**:  
 **apiconfig.ini**: |-  
 [SERVER\_CONFIG]  
 SPARK\_SERVER\_HOST: 0.0.0.0  
 #BACKEND\_API\_URL: 35.168.55.156:8089  
 BACKEND\_API\_URL:  
 SERVICE\_UP\_IP: 0.0.0.0  
 SPARK\_SERVER\_PORT : 10000  
 RULE\_MANAGER\_API\_URL : http://rule-manager-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080  
 #MANAGE\_CONSOLE\_API\_URL : 35.168.55.156:8085  
 MANAGE\_CONSOLE\_API\_URL :  
 SQL\_SERVER\_PORT : 1699  
 SQL\_SERVER\_HOST : mr2-aixd-av06.rno.apple.com  
 SQL\_SERVER\_USER : GBI\_DATA\_LAB\_USER  
 SQL\_SERVER\_PASSWORD : C0dd146t0n7\_Bri7fca57\_Ej2jb2  
 DB\_SCHEMA : GBI\_DATA\_LAB\_OWNER  
 SQL\_DATABASE : gia3d  
 CONFIG\_LEVEL\_SETTINGS : configs.lens\_appleconnect  
 CATALOG\_CONNECTIONS\_MANAGER\_SERVER : http://connection-manager-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080/connections/api  
 CATALOG\_DISCOVER\_MANAGER\_SERVER : http://discover-api-service.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:8080/discover/api  
 SCHEDULER\_API\_URL : scheduler-backend.gbi-datascience-platform-dev.svc.int.usrno3.applecloud.io:5000  
 JOB\_TOKEN : ODg5Njk1ODA3NjYyOWMxNzExYjk5MWM0NTUyZTg=  
 S3\_URL: http://store-test.blobstore.apple.com  
 ENV\_CD : datalab-dev  
 S3\_REGION\_NAME : store-test  
 S3\_BUCKET\_NAME : batch-jobs  
 S3\_SECRET\_ACCESS\_KEY : 7DC67C86159E169819098FB9B4525E9C1DCFE70855815DCBBE6289A502C18FD9  
 S3\_ACCESS\_KEY\_ID : MKIAS5P8XEXZ15LMC86K  
 CATALOG\_ENV : https://gbiio-service:443  
 DB\_USER = GBI\_DATA\_LAB\_USER  
 DB\_DSN = mr2-aixd-av06.rno.apple.com:1699/gia3d  
 DB\_password = C0dd146t0n7\_Bri7fca57\_Ej2jb2  
 DATALAB\_APP\_ID = 136927  
 DATALAB\_APP\_PASSWORD = pxr767tssuvgaix4  
 DATALAB\_APP\_CONTEXT = datalab  
 CATALOG\_APP\_ID = 136847  
 APP\_TOKEN\_GENERATION\_URL = https://idmsac-uat.corp.apple.com/auth/apptoapp/token/generate  
 S3\_ID = 51b20ee0-2189-11e8-b716-09d7c9dd2f27  
 CATALOG\_URL = https://gbiio-service:443/virtualedw  
 CATALOG\_OTHER\_APP\_ID = 136927  
 CATALOG\_APP\_CONTEXT = datalab  
**kind**: ConfigMap  
**metadata**:  
 **creationTimestamp**: null  
 **name**: mashup-backend-ini

*Filename: deployment.yaml*

**apiVersion**: extensions/v1beta1  
**kind**: Deployment  
**metadata**:  
 **name**: mashup-backend  
 **labels**:  
 **app**: mashup-backend  
 **component**: gbi-datalab  
**spec**:  
 **replicas**: 1  
 **revisionHistoryLimit**: 1  
 **strategy**:  
 **type**: **"Recreate"  
 template**:  
 **metadata**:  
 **labels**:  
 **app**: mashup-backend  
 **component**: gbi-datalab  
 **annotations**:  
 **build**: **"{apc\_buildnumber}"  
 date**: **"{apc\_date}"  
 spec**:  
 **volumes**:  
 - **name**: mashup-backend-secret  
 **secret**:  
 **secretName**: mashup-backend  
 **items**:  
 - **key**: lens\_appleconnect.py  
 **path**: lens\_appleconnect.py  
 - **name**: apiconfigini-mashup-backend-secret  
 **secret**:  
 **secretName**: mashup-backend  
 **items**:  
 - **key**: apiconfig.ini  
 **path**: apiconfig.ini  
 - **name**: logs  
 **emptyDir**: {}  
 - **name**: splunkforwarder-temp  
 **emptyDir**: {}  
 - **name**: podinfo  
 **downwardAPI**:  
 **items**:  
 - **path**: **"labels"  
 fieldRef**:  
 **fieldPath**: metadata.labels  
 - **path**: **"uid"  
 fieldRef**:  
 **fieldPath**: metadata.uid  
 **containers**:  
 - **name**: mashup-backend  
 **image**: {dockerpath}  
 **imagePullPolicy**: Always  
 **volumeMounts**:  
 - **name**: mashup-backend-secret  
 **readOnly**: true  
 **mountPath**: /app/mashup\_apis/configs/lens\_appleconnect.py  
 **subPath**: lens\_appleconnect.py  
 - **name**: apiconfigini-mashup-backend-secret  
 **readOnly**: true  
 **mountPath**: /app/mashup\_apis/apiconfig.ini  
 **subPath**: apiconfig.ini  
 - **name**: logs  
 **mountPath**: **"/logs"  
 ports**:  
 - **name**: http  
 **containerPort**: 5000  
 **resources**:  
 **limits**:  
 **cpu**: 8  
 **memory**: 16Gi  
 **requests**:  
 **cpu**: 500m  
 **memory**: 8Gi  
 **env**:  
 - **name**: CONNECTOR\_PYTHON\_HOST  
 **value**: **"connector-service.{NAMESPACE}.svc.int.usrno3.applecloud.io"** - **name**: CONNECTOR\_PYTHON\_PORT  
 **value**: **"8095"** - **name**: DATALAB\_APP\_ID  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: DATALAB\_APP\_ID  
 - **name**: DATALAB\_APP\_PASSWORD  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: DATALAB\_APP\_PASSWORD  
 - **name**: DATALAB\_APP\_CONTEXT  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: DATALAB\_APP\_CONTEXT  
 - **name**: CATALOG\_APP\_ID  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: CATALOG\_APP\_ID  
 - **name**: APP\_TOKEN\_GENERATION\_URL  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: APP\_TOKEN\_GENERATION\_URL  
 - **name**: S3\_ID  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: S3\_ID  
 - **name**: CATALOG\_URL  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: CATALOG\_URL  
 - **name**: CATALOG\_OTHER\_APP\_ID  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: CATALOG\_OTHER\_APP\_ID  
 - **name**: CATALOG\_APP\_CONTEXT  
 **valueFrom**:  
 **secretKeyRef**:  
 **name**: mashup-backend  
 **key**: CATALOG\_APP\_CONTEXT  
 - **name**: splunkforwarder  
 **image**: **"{SPLUNKFORWARDER\_IMAGE}"  
 resources**:  
 **limits**:  
 **cpu**: 200m  
 **memory**: 256Mi  
 **requests**:  
 **cpu**: 200m  
 **memory**: 256Mi  
 **env**:  
 - **name**: NAMESPACE  
 **valueFrom**:  
 **fieldRef**:  
 **fieldPath**: metadata.namespace  
 - **name**: SPLUNK\_CLUSTER  
 **value**: **"{SPLUNK\_CLUSTER}"** - **name**: SPLUNK\_LOG\_MONITOR  
 **value**: **"/logs/\*.log|gbi\_apc\_ds\_core|mashup\_backend\_$(NAMESPACE)\_{APC\_CLUSTER\_NAME}"  
 volumeMounts**:  
 - **name**: logs  
 **mountPath**: **"/logs"** - **name**: splunkforwarder-temp  
 **mountPath**: **"/opt/splunk/var"** - **name**: podinfo  
 **mountPath**: **"/etc/splunk/podinfo"  
 readOnly**: true

*Filename: entrypoint.sh*

**#!/bin/bash***#/usr/sbin/sshd*echo **'started to run start\_web.py'**cd /app/mashup\_apis  
  
  
nohup python start\_web.py &  
  
echo **'running'**tail -f /dev/null

Filename: secrets-dev.yaml

**apiVersion**: v1  
**data**:  
 **APP\_TOKEN\_GENERATION\_URL**: aHR0cHM6Ly9pZG1zYWMtdWF0LmNvcnAuYXBwbGUuY29tL2F1dGgvYXBwdG9hcHAvdG9rZW4vZ2VuZXJhdGU=  
 **CATALOG\_APP\_CONTEXT**: ZGF0YWxhYg==  
 **CATALOG\_APP\_ID**: MTM2ODQ3  
 **CATALOG\_OTHER\_APP\_ID**: MTM2OTI3  
 **CATALOG\_URL**: aHR0cHM6Ly9nYmlpby1zZXJ2aWNlOjQ0My92aXJ0dWFsZWR3  
 **DATALAB\_APP\_CONTEXT**: ZGF0YWxhYg==  
 **DATALAB\_APP\_ID**: MTM2OTI3  
 **DATALAB\_APP\_PASSWORD**: cHhyNzY3dHNzdXZnYWl4NA==  
 **S3\_ID**: NTFiMjBlZTAtMjE4OS0xMWU4LWI3MTYtMDlkN2M5ZGQyZjI3  
 **apiconfig.ini**:   
 **lens\_appleconnect.py**:   
**kind**: Secret  
**metadata**:  
 **creationTimestamp**: null  
 **name**: mashup-backend

Filename: service.yaml

---  
**apiVersion**: v1  
**kind**: Service  
**metadata**:  
 **name**: mashup-backend-service  
 **labels**:  
 **app**: mashup-backend  
 **component**: gbi-datalab  
**spec**:  
 **selector**:  
 **app**: mashup-backend  
 **ports**:  
 - **name**: http  
 **port**: 5000  
 **targetPort**: 5000