```python
# -*- coding: utf-8 -*-
"""
The module provides access to numina data.

https://developer.numina.co/

@author: [Mike Lowry]

"""

__copyright__ = "Copyright 2023, Michael B. Lowry"
__license__ = "cc-by-nc-nd"
__email__ = "mlowry@uidaho.edu"
__credits__ = ["Mike Lowry", "Matt Thompson"]

import requests
import json
from pathlib import Path
from datetime import datetime

from gql import Client, gql
from gql.transport.requests import RequestsHTTPTransport
import pandas as pd


numina_url = "https://api.numina.co/graphql"
TOKEN = False
TOKEN_EXPIRE_TIME = False

project_folder = Path(__file__).parents[2].__str__()

if 'mlowry' in project_folder:
    PRIVATE_FOLDER = r'C:\Users\mlowry\University of Idaho\Storage-Engineering - mlowry\Private'

if 'thom9078' in project_folder:
    PRIVATE_FOLDER = r'C:\Users\thom9078\OneDrive - University of Idaho\Private'

if 'matt' in project_folder:
    PRIVATE_FOLDER = r'C:\Users\matth\OneDrive - University of Idaho\Private'

# ============================================================================
# Functions
# ============================================================================

def create_iso_string(year, month, day, hour=0, minute=0, second=0):
    date_time_obj = datetime(*[year, month, day, hour, minute, second])
    return date_time_obj.isoformat()


def get_token(PRIVATE_FOLDER):
    """
    Gets authorized token and expiration datetime in 24 hours.

    Returns:
        TOKEN (str): needed for authorized access.

    Creates:
        TOKEN_EXPIRE_TIME (str): ISO 8601, UTC

    """
    global TOKEN
    global TOKEN_EXPIRE_TIME

    if TOKEN_EXPIRE_TIME:
        expire_utc_datetime = datetime.fromisoformat(TOKEN_EXPIRE_TIME)
        current_utc_datetime = datetime.utcnow()
        if current_utc_datetime > expire_utc_datetime:
            print("Getting new token to replace expired.")
            token_needed = True
        else:
            token_needed = False
    else:
        print("Getting token for first time")
        token_needed = True


    if token_needed:
        private_login_file = PRIVATE_FOLDER + r'\numina_login.txt' # line 1: username, line 2: password
        with open(private_login_file) as a:
            email = a.readline().strip()
            password = a.readline().strip()

        query_string = """
          mutation logIn($email: String!, $password: String!){
              logIn(email: $email, password: $password){

                jwt {
                  token
                  exp
                }

              }
```

```
          }
        """

        params = {"email": email, "password": password}
        response = requests.post(url=numina_url, json={"query": query_string, "variables": params})

        if response.status_code == 200:
            TOKEN = json.loads(response.content)['data']['logIn']['jwt']['token']
            TOKEN_EXPIRE_TIME = json.loads(response.content)['data']['logIn']['jwt']['exp']
        else:
            print("Authorization Failed.")
            print("Response Status Code: ", response.status_code)


    return TOKEN

# ================================================================================
# Relational Data Queries
# ================================================================================

def devices(serialno=None, name=None, sort=None, before=None, after=None, first=None, last=None):
    """Gets device information. You can filter by serialno or name."""

    current_token = get_token(PRIVATE_FOLDER)

    # Create gql client
    headerDict={'Authorization': current_token}
    transport = RequestsHTTPTransport(url=numina_url, headers=headerDict)
    client = Client(transport=transport, fetch_schema_from_transport=True)

    query = gql(
    """
    query devices($serialno: String, $name: String, $sort: [DeviceSortEnum], $before: String, $after: String, $first: Int, $last: Int){
      devices(serialno: $serialno, name: $name, sort: $sort, before: $before, after: $after, first: $first, last: $last){

        edges {
          node {
            feedId
            name
            rawId
            serialno
            location {
              lat
              lon
              }
          }
        }

      }
    }
    """
    )

    params = {"serialno": serialno, "name": name, "sort": sort, "before": before, "after": after, "first": first, "last": last}

    # Execute the query on the client
    result = client.execute(query, variable_values=params)

    data = result['devices']['edges'] # Drill down to the data which is a list of dictionaries called nodes.
    df = pd.json_normalize(data)
    df.columns = df.columns.str.replace(r'node.', '', regex=True) # removes node. prefix.

    return df


def behaviorZones(serialnos=None, sort=None, before=None, after=None, first=None, last=None):
    """Gets device information. You can filter by serialno or name."""

    current_token = get_token(PRIVATE_FOLDER)

    # Create gql client
    headerDict={'Authorization': current_token}
    transport = RequestsHTTPTransport(url=numina_url, headers=headerDict)
    client = Client(transport=transport, fetch_schema_from_transport=True)

    query = gql(
    """
    query behaviorZones($serialnos: [String], $sort: [BehaviorZoneSortEnum], $before: String, $after: String, $first: Int, $last: Int){
      behaviorZones(serialnos: $serialnos, sort: $sort, before: $before, after: $after, first: $first, last: $last){

        edges {
          node {
            zoneType
            id
            feedId
            text
            color
            isCoverageZone
            rawId
          }
        }
```

```
      }
    }
    """
    )

    params = {"serialnos": serialnos, "sort": sort, "before": before, "after": after, "first": first, "last": last}

    # Execute the query on the client
    result = client.execute(query, variable_values=params)

    data = result['behaviorZones']['edges'] # Drill down to the data which is a list of dictionaries called nodes.
    df = pd.json_normalize(data)
    df.columns = df.columns.str.replace(r'node.', '', regex=True) # removes node. prefix.

    return df



# ======================================================================
# Volume Count Analytics
# ======================================================================

def feedCountMetrics(serialnos, startTime, endTime, interval, objClasses, timezone="US/Mountain",  before=None, after=None, first=None, last=None):
    """Returns count
    A metric (e.g. count) representing aggregate object behavior for objects of a given object class, device (feed), and time interval. (e.g. pedestrian counts for

    Args:
        serialnos (list of strings): serialno values of Devices for which to compute metric.
        startTime (string): Start of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        endTime (TYPE): End of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        interval (TYPE): Time interval to aggregate objects for metric, expects a string with an integer followed by an interval abbreviation (m:minutes, h:hour, d:
        objClasses (TYPE): List of Object classes ('car', 'pedestrian', 'bus', 'truck', 'trash', 'dog').
        timezone (TYPE, optional): Timezone in tz format to localize time range and intervals (e.g. 'America/New_York'). Defaults to "US/Mountain".
        before (TYPE, optional): paginate results before cursor. Defaults to None.
        after (TYPE, optional): paginate results after cursor. Defaults to None.
        first (TYPE, optional): get first n results. Defaults to None.
        last (TYPE, optional): get last n results. Defaults to None.

    Returns:
        df (TYPE): dataframe.

    """

    current_token = get_token(PRIVATE_FOLDER)

    # Create gql client
    headerDict={'Authorization': current_token}
    transport = RequestsHTTPTransport(url=numina_url, headers=headerDict)
    client = Client(transport=transport, fetch_schema_from_transport=True)

    query = gql(
    """
    query feedCountMetrics($serialnos: [String]!, $startTime: DateTime!, $endTime: DateTime!, $interval: String, $objClasses: [String]!, $timezone: String, $before:
      feedCountMetrics(serialnos: $serialnos, startTime: $startTime, endTime: $endTime, interval: $interval, objClasses: $objClasses, timezone: $timezone, before: $

          edges {
            node {
              serialno
              result
              objClass
              time
              }
            }

      }
    }
    """
    )

    params = {"serialnos": serialnos, "startTime": startTime,
              "endTime": endTime, "interval": interval, "objClasses": objClasses,
              "timezone": timezone, "before": before, "after": after, "first": first, "last": last}

    # Execute the query on the client
    result = client.execute(query, variable_values=params)

    data = result['feedCountMetrics']['edges'] # Drill down to the data which is a list of dictionaries called nodes.
    df = pd.json_normalize(data)
    df.columns = df.columns.str.replace(r'node.', '', regex=True) # removes node. prefix.

    return df


def zoneCountMetrics(zoneIds, startTime, endTime, interval, objClasses, timezone="US/Mountain",  before=None, after=None, first=None, last=None):
    """Returns count
        A metric (e.g. count) representing aggregate object behavior for objects that passed through a set of behavior zones, of a given object class and in a given

    Args:
        zoneIds (TYPE):
        startTime (TYPE): Start of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        endTime (TYPE): End of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        interval (TYPE): Time interval to aggregate objects for metric, expects a string with an integer followed by an interval abbreviation (m:minutes, h:hour, d:
```

```
        objClasses (TYPE): List of Object classes ('car', 'pedestrian', 'bus', 'truck', 'trash', 'dog').
        timezone (TYPE, optional): Timezone in tz format to localize time range and intervals (e.g. 'America/New_York'). Defaults to "US/Mountain".
        before (TYPE, optional): paginate results before cursor. Defaults to None.
        after (TYPE, optional): paginate results after cursor. Defaults to None.
        first (TYPE, optional): get first n results. Defaults to None.
        last (TYPE, optional): get last n results. Defaults to None.

    Returns:
        df (TYPE): dataframe.

    """

    current_token = get_token(PRIVATE_FOLDER)

    # Create gql client
    headerDict={'Authorization': current_token}
    transport = RequestsHTTPTransport(url=numina_url, headers=headerDict)
    client = Client(transport=transport, fetch_schema_from_transport=True)

    query = gql(
    """
    query zoneCountMetrics($zoneIds: [Int]!, $startTime: DateTime!, $endTime: DateTime!, $interval: String, $objClasses: [String]!, $timezone: String, $before: Stri
      zoneCountMetrics(zoneIds: $zoneIds, startTime: $startTime, endTime: $endTime, interval: $interval, objClasses: $objClasses, timezone: $timezone, before: $befo

          edges {
            node {
              zoneIds
              result
              objClass
              time
              }
            }

      }
    }
    """
    )

    params = {"zoneIds": zoneIds, "startTime": startTime,
              "endTime": endTime, "interval": interval, "objClasses": objClasses,
              "timezone": timezone, "before": before, "after": after, "first": first, "last": last}

    # Execute the query on the client
    result = client.execute(query, variable_values=params)

    data = result['zoneCountMetrics']['edges'] # Drill down to the data which is a list of dictionaries called nodes.
    df = pd.json_normalize(data)
    df.columns = df.columns.str.replace(r'node.', '', regex=True) # removes node. prefix.

    return df



def screenlineCountMetrics(zoneIds, startTime, endTime, interval, objClasses, timezone="US/Mountain",  before=None, after=None, first=None, last=None):
    """Returns count


    Args:
        zoneIds (TYPE):
        startTime (TYPE): Start of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        endTime (TYPE): End of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        interval (TYPE): Time interval to aggregate objects for metric, expects a string with an integer followed by an interval abbreviation (m:minutes, h:hour, d:
        objClasses (TYPE): List of Object classes ('car', 'pedestrian', 'bus', 'truck', 'trash', 'dog').
        timezone (TYPE, optional): Timezone in tz format to localize time range and intervals (e.g. 'America/New_York'). Defaults to "US/Mountain".
        before (TYPE, optional): paginate results before cursor. Defaults to None.
        after (TYPE, optional): paginate results after cursor. Defaults to None.
        first (TYPE, optional): get first n results. Defaults to None.
        last (TYPE, optional): get last n results. Defaults to None.

    Returns:
        df (TYPE): dataframe.

    """

    current_token = get_token(PRIVATE_FOLDER)

    # Create gql client
    headerDict={'Authorization': current_token}
    transport = RequestsHTTPTransport(url=numina_url, headers=headerDict)
    client = Client(transport=transport, fetch_schema_from_transport=True)

    query = gql(
    """
    query screenlineCountMetrics($zoneIds: [Int]!, $startTime: DateTime!, $endTime: DateTime!, $interval: String, $objClasses: [String]!, $timezone: String, $before
      screenlineCountMetrics(zoneIds: $zoneIds, startTime: $startTime, endTime: $endTime, interval: $interval, objClasses: $objClasses, timezone: $timezone, before:

          edges {
            node {
              zoneIds
              result
              objClass
              time
```

```
            }
          }

      }
    }
    """
    )

    params = {"zoneIds": zoneIds, "startTime": startTime,
              "endTime": endTime, "interval": interval, "objClasses": objClasses,
              "timezone": timezone, "before": before, "after": after, "first": first, "last": last}

    # Execute the query on the client
    result = client.execute(query, variable_values=params)

    data = result['screenlineCountMetrics']['edges'] # Drill down to the data which is a list of dictionaries called nodes.
    df = pd.json_normalize(data)
    df.columns = df.columns.str.replace(r'node.', '', regex=True) # removes node. prefix.

    return df


# ================================================================================
# Behavior Analytics
# ================================================================================

def feedHeatmaps(serialno, startTime, endTime, timezone="US/Mountain", objClasses=['pedestrian'], before=None, after=None, first=None, last=None):
    """Returns count
    A metric (e.g. count) representing aggregate object behavior for objects of a given object class, device (feed), and time interval. (e.g. pedestrian counts for

    Args:
        serialno (TYPE): serialno
        startTime (TYPE): Start of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        endTime (TYPE): End of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        interval (TYPE): Time interval to aggregate objects for metric, expects a string with an integer followed by an interval abbreviation (m:minutes, h:hour, d:
        objClasses (TYPE): List of Object classes ('car', 'pedestrian', 'bus', 'truck', 'trash', 'dog').
        timezone (TYPE, optional): Timezone in tz format to localize time range and intervals (e.g. 'America/New_York'). Defaults to "US/Mountain".
        before (TYPE, optional): paginate results before cursor. Defaults to None.
        after (TYPE, optional): paginate results after cursor. Defaults to None.
        first (TYPE, optional): get first n results. Defaults to None.
        last (TYPE, optional): get last n results. Defaults to None.

    Returns:
        df (TYPE): dataframe.

    """

    current_token = get_token(PRIVATE_FOLDER)

    # Create gql client
    headerDict={'Authorization': current_token}
    transport = RequestsHTTPTransport(url=numina_url, headers=headerDict)
    client = Client(transport=transport, fetch_schema_from_transport=True)

    query = gql(
    """
    query feedHeatmaps($serialno: String!, $startTime: DateTime!, $endTime: DateTime!, $timezone: String, $objClasses: [String]!, $before: String, $after: String, $
      feedHeatmaps(serialno: $serialno, startTime: $startTime, endTime: $endTime, timezone: $timezone, objClasses: $objClasses, before: $before, after: $after, firs

          edges {
            node {
                objClass
                heatmap
            }
          }

      }
    }
    """
    )

    params = {"serialno": serialno, "startTime": startTime,
              "endTime": endTime, "timezone": timezone,
              "objClasses": objClasses, "before": before, "after": after, "first": first, "last": last}

    # Execute the query on the client
    result = client.execute(query, variable_values=params)

    data = result['feedHeatmaps']['edges'] # Drill down to the data which is a list of dictionaries called nodes.
    df = pd.json_normalize(data)
    df.columns = df.columns.str.replace(r'node.', '', regex=True) # removes node. prefix.

    return df

# ================================================================================
# Matt's attempt to write function for zone dwell time distribution and zone max occupancy
# ================================================================================

def zoneDwellTimeDistribution(zoneIds, startTime, endTime, interval, objClasses, timezone="US/Mountain",  before=None, after=None, first=None, last=None):
    """Returns count
        A metric (e.g. count) representing aggregate object behavior for objects that passed through a set of behavior zones, of a given object class and in a given
```

```
    Args:
        zoneIds (TYPE):
        startTime (TYPE): Start of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        endTime (TYPE): End of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        interval (TYPE): Time interval to aggregate objects for metric, expects a string with an integer followed by an interval abbreviation (m:minutes, h:hour, d:
        objClasses (TYPE): List of Object classes ('car', 'pedestrian', 'bus', 'truck', 'trash', 'dog').
        timezone (TYPE, optional): Timezone in tz format to localize time range and intervals (e.g. 'America/New_York'). Defaults to "US/Mountain".
        before (TYPE, optional): paginate results before cursor. Defaults to None.
        after (TYPE, optional): paginate results after cursor. Defaults to None.
        first (TYPE, optional): get first n results. Defaults to None.
        last (TYPE, optional): get last n results. Defaults to None.

    Returns:
        df (TYPE): dataframe.

    """

    current_token = get_token(PRIVATE_FOLDER)

    # Create gql client
    headerDict={'Authorization': current_token}
    transport = RequestsHTTPTransport(url=numina_url, headers=headerDict)
    client = Client(transport=transport, fetch_schema_from_transport=True)

    query = gql(
    """
    query zoneDwellTimeDistribution($zoneIds: [Int]!, $startTime: DateTime!, $endTime: DateTime!, $timezone: String, $objClasses: [String]!, $interval: String, $bef
      zoneDwellTimeDistribution(zoneIds: $zoneIds, startTime: $startTime, endTime: $endTime, timezone: $timezone, objClasses: $objClasses, interval: $interval, befo

          edges {
            node {
                id
                zoneIds
                mean
                pct25
                pct50
                pct75
                pct100
                count
                objClass
                time
              }
            }

      }
    }
    """
    )

    params = {"zoneIds": zoneIds, "startTime": startTime,
              "endTime": endTime, "interval": interval, "objClasses": objClasses,
              "timezone": timezone, "before": before, "after": after, "first": first, "last": last}

    # Execute the query on the client
    result = client.execute(query, variable_values=params)

    data = result['zoneDwellTimeDistribution']['edges'] # Drill down to the data which is a list of dictionaries called nodes.
    df = pd.json_normalize(data)
    df.columns = df.columns.str.replace(r'node.', '', regex=True) # removes node. prefix.

    return df

def zoneMaxOccupancy(zoneIds, startTime, endTime, interval, objClasses, timezone="US/Mountain",  before=None, after=None, first=None, last=None):
    """Returns count


    Args:
        zoneIds (TYPE):
        startTime (TYPE): Start of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        endTime (TYPE): End of analysis time range in ISO format (e.g '2019-01-01T00:00:00).
        interval (TYPE): Time interval to aggregate objects for metric, expects a string with an integer followed by an interval abbreviation (m:minutes, h:hour, d:
        objClasses (TYPE): List of Object classes ('car', 'pedestrian', 'bus', 'truck', 'trash', 'dog').
        timezone (TYPE, optional): Timezone in tz format to localize time range and intervals (e.g. 'America/New_York'). Defaults to "US/Mountain".
        before (TYPE, optional): paginate results before cursor. Defaults to None.
        after (TYPE, optional): paginate results after cursor. Defaults to None.
        first (TYPE, optional): get first n results. Defaults to None.
        last (TYPE, optional): get last n results. Defaults to None.

    Returns:
        df (TYPE): dataframe.

    """

    current_token = get_token(PRIVATE_FOLDER)

    # Create gql client
    headerDict={'Authorization': current_token}
    transport = RequestsHTTPTransport(url=numina_url, headers=headerDict)
    client = Client(transport=transport, fetch_schema_from_transport=True)

    query = gql(
    """
```

```
query zoneMaxOccupancy($zoneIds: [Int]!, $startTime: DateTime!, $endTime: DateTime!, $interval: String, $objClasses: [String]!, $timezone: String, $before: Stri
  zoneMaxOccupancy(zoneIds: $zoneIds, startTime: $startTime, endTime: $endTime, interval: $interval, objClasses: $objClasses, timezone: $timezone, before: $befo

      edges {
        node {
          zoneIds
          result
          objClass
          time
          }
        }

  }
}
"""
)

params = {"zoneIds": zoneIds, "startTime": startTime,
          "endTime": endTime, "interval": interval, "objClasses": objClasses,
          "timezone": timezone, "before": before, "after": after, "first": first, "last": last}

# Execute the query on the client
result = client.execute(query, variable_values=params)

data = result['zoneMaxOccupancy']['edges'] # Drill down to the data which is a list of dictionaries called nodes.
df = pd.json_normalize(data)
df.columns = df.columns.str.replace(r'node.', '', regex=True) # removes node. prefix.

return df
```