CALAMBA, LIAM FRANCIS N.

CPE22S3 - CPE311

13/03/2024

Collecting temperature data from an API

About the data

In this notebook, we will be collecting daily temperature data from the National Centers for Environmental Information (NCEI) API. We will use the Global Historical Climatology Network - Daily (GHCND) data set; see the documentation here. Note: The NCEI is part of the National Oceanic and Atmospheric Administration (NOAA) and, as you can see from the URL for the API, this resource was created when the NCEI was called the NCDC. Should the URL for this resource change in the future, you can search for the NCEI weather API to find the updated one.

Using the NCEI API paste token below

```
import requests
def make_request(endpoint, payload=None):

 return requests.get(
 f'https://www.ncdc.noaa.gov/cdo-web/api/v2/{endpoint}',
 headers={
 'token': 'COGqDwKIRDvdlxMHVhxgqTsoJqtCoAfz'
 },
 params=payload
 )
```

See what datasets are available

We can make requests to the datasets endpoint to see what datasets are available. We also pass in a dictionary for the payload to get datsets that have data after the start date of October 1, 2018.

```
response = make_request('datasets', {'startdate':'2018-10-01'})
response.status_code #200 = okay :D
```

```
200
```

Get the keys of the result

The result is a JSON object which we can access with the json() method of our Response object. JSON objects can be treated like dictionaries, so we can access the keys() just like we would a dictionary:

```
response.json().keys()
```

```
dict_keys(['metadata', 'results'])
```

```
response.json()['metadata']
```

```
{'resultset': {'offset': 1, 'count': 11, 'limit': 25}}
```

Figure out what data is in the result

The results key contains the data we requested. This is a list of what would be rows in our dataframe. Each entry in the list is a dictionary, so we can look at the keys to get the fields:

```
response.json()['results'][0].keys()
```

```
dict_keys(['uid', 'mindate', 'maxdate', 'name', 'datacoverage', 'id'])
```

Parse the result

We don't want all those fields, so we will use a list comphrension to take only the id and name fields out:

```
[(data['id'],data['name']) for data in response.json()['results']]
```

```
[('GHCND', 'Daily Summaries'),
 ('GSOM', 'Global Summary of the Month'),
 ('GSOY', 'Global Summary of the Year'),
 ('NEXRAD2', 'Weather Radar (Level II)'),
 ('NEXRAD3', 'Weather Radar (Level III)'),
 ('NORMAL_ANN', 'Normals Annual/Seasonal'),
 ('NORMAL_DLY', 'Normals Daily'),
 ('NORMAL_HLY', 'Normals Hourly'),
 ('NORMAL_MLY', 'Normals Monthly'),
 ('PRECIP_15', 'Precipitation 15 Minute'),
 ('PRECIP_HLY', 'Precipitation Hourly')]
```

Figure out which data category we want

The GHCND data containing daily summaries is what we want. Now we need to make another request to figure out which data categories we want to collect. This is the datacategories endpoint. We have to pass the datasetid for GHCND as the payload so the API knows which dataset we are asking about:

```
response = make_request(
    'datacategories',
    payload={
        'datasetid' : 'GHCND'
    }
)
response.status_code
```

```
200
```

Since we know the API gives us a metadata and a results key in each response, we can see what is in the results portion of the JSON response:

```
response.json()['results']
```

```
[{'name': 'Evaporation', 'id': 'EVAP'},
 {'name': 'Land', 'id': 'LAND'},
 {'name': 'Precipitation', 'id': 'PRCP'},
 {'name': 'Sky cover & clouds', 'id': 'SKY'},
 {'name': 'Sunshine', 'id': 'SUN'},
 {'name': 'Air Temperature', 'id': 'TEMP'},
 {'name': 'Water', 'id': 'WATER'},
 {'name': 'Wind', 'id': 'WIND'},
 {'name': 'Weather Type', 'id': 'WXTYPE'}]
```

Grab the data type ID for the Temperature category

We will be working with temperatures, so we want the TEMP data category. Now, we need to find the datatypes to collect. For this, we use the datatypes endpoint and provide the datacategoryid which was TEMP . We also specify a limit for the number of datatypes to return with the payload. If there are more than this we can make another request later, but for now, we just want to pick a few out:

```
response = make_request(
    'datatypes',
    payload={
        'datacategoryid' : 'TEMP',
        'limit' : 100
    }
)
response.status_code
```

```
200
```

We can grab the id and name fields for each of the entries in the results portion of the data. The fields we are interested in are at the bottom:

```
[(datatype['id'],datatype['name']) for datatype in response.json()['results']][-5:]
```

```
[('MNTM', 'Monthly mean temperature'),
 ('TAVG', 'Average Temperature.'),
 ('TMAX', 'Maximum temperature'),
 ('TMIN', 'Minimum temperature'),
 ('TOBS', 'Temperature at the time of observation')]
```

Determine which Location Category we want

Now that we know which datatypes we will be collecting, we need to find the location to use. First, we need to figure out the location category. This is obtained from the locationcategories endpoint by passing the datasetid :

```
response = make_request(
    'locationcategories',
    {
        'dataset' : 'GHCND'
    }
)
response.status_code
```

```
200
```

We can use pprint to print dictionaries in an easier-to-read format. After doing so, we can see there are 12 different location categories, but we are only interested in CITY :

```
import pprint as pp
pp.pprint(response.json())
```

```
{'metadata': {'resultset': {'count': 12, 'limit': 25, 'offset': 1}},
 'results': [{'id': 'CITY', 'name': 'City'},
             {'id': 'CLIM_DIV', 'name': 'Climate Division'},
             {'id': 'CLIM_REG', 'name': 'Climate Region'},
             {'id': 'CNTRY', 'name': 'Country'},
             {'id': 'CNTY', 'name': 'County'},
             {'id': 'HYD_ACC', 'name': 'Hydrologic Accounting Unit'},
             {'id': 'HYD_CAT', 'name': 'Hydrologic Cataloging Unit'},
             {'id': 'HYD_REG', 'name': 'Hydrologic Region'},
             {'id': 'HYD_SUB', 'name': 'Hydrologic Subregion'},
             {'id': 'ST', 'name': 'State'},
             {'id': 'US_TERR', 'name': 'US Territory'},
             {'id': 'ZIP', 'name': 'Zip Code'}]}
```

Get NYC Location ID

In order to find the location ID for New York, we need to search through all the cities available. Since we can ask the API to return the cities sorted, we can use binary search to find New York quickly without having to make many requests or request lots of data at once. The following function makes the first request to see how big the list of cities is and looks at the first value. From there it decides if it needs to move towards the beginning or end of the list by comparing the city we are looking for to others alphabetically. Each time it makes a request it can rule out half of the remaining data to search.

```
'''di gumana ↓↓↓'''

# def get_item(name, what, endpoint, start=1, end=None):
#   mid = (start + (end if end else 1)) // 2
#   name = name.lower()
#   payload = {
#       'dataset' : 'GHCND',
#       'sortfield' : 'name',
#       'offset' : mid,
#       'limit' : 1
#   }

#   response = make_request(endpoint, {**payload, **what})

#   if response.ok:
#     end = end if end else response.json()['metadata']['resultset']['count']
#     current_name = response.json()['results'][0]['name'].lower()

#     if name in current_name:
#       return response.json()['results'][0]

#     else:
#       if start >=end:
#         return{}
#       elif name < current_name:
#         return get_item(name,what,endpoint,start,mid - 1)
#       elif name > current_name:
#         return get_item(name, what, endpoint, mid + 1, end)

#   else:
#     print(f'Response not OK, status:{response.status_code}')

# def get_location(name):

#   return get_item(name, {'loacationcategoryid' : 'CITY'}, 'locations')

'''gomagana ↓↓↓'''

def get_item(name, what, endpoint, start=1, end=None):
 """
 Grab the JSON payload for a given field by name using binary search.
 Parameters:
 - name: The item to look for.
 - what: Dictionary specifying what the item in `name` is.
 - endpoint: Where to look for the item.
 - start: The position to start at. We don't need to touch this, but the
 function will manipulate this with recursion.
 - end: The last position of the cities. Used to find the midpoint, but
 like `start` this is not something we need to worry about.
 Returns:
 Dictionary of the information for the item if found otherwise
 an empty dictionary.
 """
 # find the midpoint which we use to cut the data in half each time
 mid = (start + (end if end else 1)) // 2

 # lowercase the name so this is not case-sensitive
 name = name.lower()

 # define the payload we will send with each request
 payload = {
 'datasetid' : 'GHCND',
 'sortfield' : 'name',
 'offset' : mid, # we will change the offset each time
 'limit' : 1 # we only want one value back
 }

 # make our request adding any additional filter parameters from `what`
 response = make_request(endpoint, {**payload, **what})

 if response.ok:
 # if response is ok, grab the end index from the response metadata the first time through
    end = end if end else response.json()['metadata']['resultset']['count']

 # grab the lowercase version of the current name
    current_name = response.json()['results'][0]['name'].lower()

 # if what we are searching for is in the current name, we have found our item
 if name in current_name:
     return response.json()['results'][0] # return the found item
 else:
  if start >= end:
 # if our start index is greater than or equal to our end, we couldn't find it
   return {}
  elif name < current_name:
 # our name comes before the current name in the alphabet, so we search further to the left
   return get_item(name, what, endpoint, start, mid - 1)
  elif name > current_name:
 # our name comes after the current name in the alphabet, so we search further to the right
   return get_item(name, what, endpoint, mid + 1, end)
  else:
 # response wasn't ok, use code to determine why
   print(f'Response not OK, status: {response.status_code}')
def get_location(name):
 """
 Grab the JSON payload for the location by name using binary search.
 Parameters:
 - name: The city to look for.
 Returns:
 Dictionary of the information for the city if found otherwise
 an empty dictionary.
```

```
"""
return get_item(name, {'locationcategoryid' : 'CITY'}, 'locations')
```

When we use binary search to find New York, we find it in just 8 requests despite it being close to the middle of 1,983 entries:

```
nyc = get_location('New York')
nyc
```

```
{'mindate': '1869-01-01',
 'maxdate': '2024-03-11',
 'name': 'New York, NY US',
 'datacoverage': 1,
 'id': 'CITY:US360019'}
```

```
'''try other places'''

manhat = get_location('Manhattan')
manhat
```

```
{'mindate': '1893-01-01',
 'maxdate': '2024-03-11',
 'name': 'Manhattan, KS US',
 'datacoverage': 1,
 'id': 'CITY:US200011'}
```

```
'''try 2'''

mnl = get_location('Manila')
mnl
```

```
{'mindate': '1945-03-01',
 'maxdate': '2024-03-10',
 'name': 'Manila, RP',
 'datacoverage': 1,
 'id': 'CITY:RP000002'}
```

```
'''try 3'''

tky = get_location('Tokyo')
tky
```

```
{'mindate': '1949-01-01',
 'maxdate': '2024-03-10',
 'name': 'Tokyo, JA',
 'datacoverage': 1,
 'id': 'CITY:JA000016'}
```

Get the station ID for Central Park

The most granular data is found at the station level:

```
central_park = get_item('NY City Central Park', {'locationid' : nyc['id']}, 'stations')
central_park
```

```
{'elevation': 42.7,
 'mindate': '1869-01-01',
 'maxdate': '2024-03-10',
 'latitude': 40.77898,
 'name': 'NY CITY CENTRAL PARK, NY US',
 'datacoverage': 1,
 'id': 'GHCND:USW00094728',
 'elevationUnit': 'METERS',
 'longitude': -73.96925}
```

Request the temperature data

Finally, we have everything we need to make our request for the New York temperature data. For this we use the data endpoint and provide all the parameters we picked up throughout our exploration of the API:

```
response = make_request(
    'data',
    {
        'datasetid' : 'GHCND',
        'stationid' : central_park['id'],
        'locationid' : nyc['id'],
        'startdate' : '2018-10-01',
        'enddate' : '2018-10-31',
        'datatypeid' : ['TMIN', 'TMAX', 'TOBS'],
        'units' : 'metric',
        'limit' : 1000
    }
)
response.status_code
```

```
200
```

Create a DataFrame

The Central Park station only has the daily minimum and maximum temperatures.

```
import pandas as p

df = p.DataFrame(response.json()['results'])
```

```
df.head()
```

	date	datatype	station	attributes	value
0	2018-10-01T00:00:00	TMAX	GHCND:USW00094728	,,W,2400	24.4
1	2018-10-01T00:00:00	TMIN	GHCND:USW00094728	,,W,2400	17.2
2	2018-10-02T00:00:00	TMAX	GHCND:USW00094728	,,W,2400	25.0
3	2018-10-02T00:00:00	TMIN	GHCND:USW00094728	,,W,2400	18.3
4	2018-10-03T00:00:00	TMAX	GHCND:USW00094728	,,W,2400	23.3

Next steps: View recommended plots

We didn't get TOBS because the station doesn't measure that:

```
df.datatype.unique()
```

```
array(['TMAX', 'TMIN'], dtype=object)
```

Despite showing up in the data as measuring it... Real-world data is dirty!

```
if get_item(
    'NY City Central Park', {'locationid':nyc['id'], 'datatypeid':'TOBS'},'stations'
):
 print('Found!')
```

```
Found!
```

Using a different station

Let's use LaGuardia airport instead. It contains TAVG (average daily temperature):

```
laguardia = get_item(
    'LaGuardia', {'locationid' : nyc['id']}, 'stations'
)
```