#dependencies

**Request= allows to make API calls**

**JSON= read the api return**

#url/file establishment

url=”link”

response=**request.get(url)**

readable\_data=**response.json()**

#nice way to

print(**json.dumps**(readabledata, indent=#, sort\_keys=True)

keys=variables

Far Far Away <https://www.youtube.com/watch?v=3I_3dkO_604&feature=youtu.be>

**Stu\_JSON\_Traversal.ipynb**

**Youtube Video Details**

# Dependencies

import json

import os

# Load JSON

filepath = os.path.join("..", "Resources", "youtube\_response.json")

with open(filepath) as jsonfile:

video\_json = json.load(jsonfile)

# Isolate "data items" for easy reading

**#[] and “” to call a key within JSON**

data = video**\_json["data"]**

print(data)

print()

print(f"The first response is {json.dumps(data, indent=2)}.")

print()

**#{} to print a variable inside f””**

print(f"The first response is {json.dumps(data, indent=4)}.")

data\_items = data["items"]

# Retrieve the video's title

title = data\_items[0]["title"]

print("Title: ", title)

# Retrieve the video's rating

**#[0] first list [“key”]**

rating = **data\_items[0]["rating"]**

print("Rating:", rating)

# Retrieve the link to the video's default thumbnail

**# [0] first list, [key] [keywithin key]**

default\_thumbnail = **data\_items[0]["thumbnail"]["default"]**

print("Thumbnail: ", default\_thumbnail)

# Retrieve the number of views this video has

view\_count = data\_items[0]["viewCount"]

print(f"View count: {view\_count}")

[**Stu\_RequestReview.ipynb**](https://gw.bootcampcontent.com/GW-Coding-Boot-Camp/GWARL201906DATA1/blob/master/01-Class-Activities/06-Python-APIs/2/Activities/02-Stu_RequestReview/Solved/Stu_RequestReview.ipynb)

**NYT Request**

# Dependencies

import json

import requests

# Specify the URL

url = "http://nyt-mongo-scraper.herokuapp.com/api/headlines"

# Make request and store response

response = requests.get(url)

**# Print status code**

**# Success of the request**

**print(response.status\_code)**

# JSON-ify response

response\_json = response.json()

print(json.dumps(response\_json))

# Print first and last articles

**# first [0] last [-1]**

print(f"The first response is {json.dumps(response\_json**[0],** indent=2)}.")

print(f"The last response is {json.dumps(response\_json**[-1],** indent=2)}.")

**#Length of response**

print(f"We received **{len(response\_json)}** responses.")

**Ins\_OpenWeatherRequest.ipynb**

**Store API KIY in .py CONFIG file to be called.**

# Dependencies

import json

import requests

from config import api\_key

# Save config information

url = "http://api.openweathermap.org/data/2.5/weather?"

city = "London"

# **Build query URL based on info provided by the provider**

query\_url = url + "appid=" + api\_key + "&q=" + city

# Get weather data

weather\_response = requests.get(query\_url)

weather\_json = weather\_response.json()

# Get the temperature from the response

print(f"The weather API responded with: {weather\_json}.")

[**Stu\_Burundi.ipynb**](https://gw.bootcampcontent.com/GW-Coding-Boot-Camp/GWARL201906DATA1/blob/master/01-Class-Activities/06-Python-APIs/2/Activities/04-Stu_Burundi/Solved/Stu_Burundi.ipynb)

**LOOP AND APPEND**

**https://www.youtube.com/watch?v=dF56b0tM0Kg&feature=youtu.be**

The most important piece of making an API call is building the URL, as this determines what information will be returned.

The units query parameter: You need to dig through documentation to find "options" like this.

It will be beneficial for you to spend a lot of time reading the documentation of an API before writing code, as this will save you time.

# Dependencies

import json

import requests

from config import api\_key

# Save config information.

#?=can add to URL

url = "http://api.openweathermap.org/data/2.5/weather**?**"

city = "Bujumbura"

units = "metric"

# Build query URL and request your results in Celsius

query\_url = f"{url}appid={api\_key}&q={city}&units={units}"

# Get weather data

weather\_response = requests.get(query\_url)

weather\_json = weather\_response.json()

print(json.dumps(weather\_json, indent=4))

# Get temperature from JSON response

**# no [0] because list of 1, key within key**

temperature = weather\_json["main"]["temp"]

# Report temperature

print(f"The temperature in Bujumbura is {temperature} C.")

# BONUS

# use list of units

units = ["metric", "imperial"]

# set up list to hold two different temperatures

temperatures = []

**# loop throught the list of units and append them to temperatures list**

**for unit in units:**

# Build query URL based on current element in units

query\_url = url + "appid=" + api\_key + "&q=" + city + "&units=" + unit

# Get weather data

weather\_response = requests.get(query\_url)

weather\_json = weather\_response.json()

# Get temperature from JSON response

temperature = weather\_json["main"]["temp"]

temperatures.append(temperature)

# Report temperatures by accessing each element in the list

print(

f"The temperature in Bujumbura is {temperatures[0]}C or {temperatures[1]}F.")

**Ins\_OpenWeatherDataFrame.ipynb**

**API+Pandas+Plottin**

OpenWeatherMap DataFrame

You are not limited to manipulating API responses manually because you can Pandas to manipulate the large amounts of data returned by APIs in bulk.

So let us look at how the temperature in a country changes based upon its latitude.

Please feel free to reference the OpenWeatherMap API documentation or sample response.

# Dependencies

import csv

import matplotlib.pyplot as plt

import requests

import pandas as pd

from config import api\_key

# Save config information.

url = "http://api.openweathermap.org/data/2.5/weather?"

units = "metric"

# Build partial query URL

query\_url = f"{url}appid={api\_key}&units={units}&q="

print(query\_url)

Below we are observing how the temperature in a country changes based upon its latitude

Responses have both temperature and latitude data stored within them.

List

Empty list

Loop for c in cities

resposnse

**cities = ["Paris", "London", "Oslo", "Beijing"]**

**# set up lists to hold reponse info**

**lat = []**

**temp = []**

**# Loop through the list of cities and perform a request for data on each**

**#key within key to list 1, key within key in list 2**

**for city in cities:**

**response = requests.get(query\_url + city).json()**

**lat.append(response['coord']['lat'])**

**temp.append(response['main']['temp'])**

**print(f"The latitude information received is: {lat}")**

**print(f"The temperature information received is: {temp}")**

**# create a data frame from LISTS cities, lat, and temp**

**weather\_dict = {**

**"city": cities,**

**"lat": lat,**

**"temp": temp**

**}**

**weather\_data = pd.DataFrame(weather\_dict)**

**weather\_data.head()**

# Build a scatter plot for each data type

plt.scatter(weather\_data["lat"], weather\_data["temp"], marker="o")

# Incorporate the other graph properties

plt.title("Temperature in World Cities")

plt.ylabel("Temperature (Celsius)")

plt.xlabel("Latitude")

plt.grid(True)

# Save the figure

plt.savefig("TemperatureInWorldCities.png")

# Show plot

plt.show()

**Stu\_TVRatings.ipynb**

#Dependencies

import requests

import json

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# list of tv show titles to query

tv\_shows = ["Altered Carbon", "Grey's Anatomy", "This is Us", "The Flash",

"Vikings", "Shameless", "Arrow", "Peaky Blinders", "Dirk Gently"]

# tv maze show search base url

base\_url = "http://api.tvmaze.com/search/shows?q="

# set up lists to hold response data for name and rating

titles = []

ratings = []

networks = []

# loop through tv show titles, make requests and parse

for show in tv\_shows:

target\_url = base\_url + show

response = requests.get(target\_url).json()

titles.append(response[0]['show']['name'])

ratings.append(response[0]['show']['rating']['average'])

# create dataframe

shows\_df = pd.DataFrame({

"title": titles,

"rating": ratings

})

shows\_df

# create a list of numbers for x values

tick\_locations = np.arange(len(shows\_df))

# create bar chart and set the values of xticks

plt.bar(tick\_locations, shows\_df['rating'], align="center")

plt.xticks(tick\_locations, shows\_df['title'], rotation=45, ha="right")

plt.savefig("tv\_show\_ratings.png")

plt.show()

**Exception Handling (Introduction)**

The OpenWeatherMap API is robust enough that it responded with every piece of information the application could have desired. There were no missing values.

HOWEVER... not every API is as solid as the OpenWeatherMap API, however, and sometimes responses will not contain all of the data desired.

So... somebody tell me... What would happen if an application tried to look up a key within a dictionary that doesn't exist? (don't look ahead... lol)

If a simple key lookup is performed - such as data["temp"] and the "temp" key doesn't exist, Python will throw an exception and terminate the program.

It does not make sense for an application to terminate itself just because a dictionary key doesn't exist. It would be much better to simply deal with the error than crash the app.

Dealing with these kinds of errors is called exception handling and thankfully Python has built-in tools for these kinds of situations.

# Your assignment is to get the last line to print without changing any

# of the code below. Instead, wrap each line that throws an error in a

# try/exept block.

**try:**

print("Infinity looks like + " + str(10 / 0) + ".")

**except** ZeroDivisionError:

print("Woops. Can't do that.")

**#pass**

try:

print("I think her name was + " + name + "?")

except NameError:

print("Oh, I forgot to define 'name'. D'oh.")

#pass

try:

print("Your name is a nonsense number. Look: " + int("Gabriel"))

except ValueError:

print("Drat. 'Gabriel' isn't a number?")

#pass

print("I made it through the gauntlet. The message survived!")

pass allows programmers to ignore certain errors if they simply don't care that they occurred.

It is good practice to wrap dictionary accesses to responses from API calls in try/except blocks, just in case not all responses have the desired key.

**? Ins\_OpenWeatherWrapper.ipynb**

**WRAPPERS making API calls EASY**

%pip install openweathermapy

# Dependencies

import openweathermapy.core as owm

#config

from config import api\_key

Rather than maintaining all of the configuration options, only a settings dict is stored which contains the options normally concatenated into the query string.

# Create settings dictionary with information we're interested in

settings = {"units": "metric", "appid": api\_key}

Rather than calling the API and converting the response to JSON manually, the owm.get\_current() method can be used instead. This method takes city name, id, or geographic coordinates, as well as the settings parameters and returns the same response as the API for that city. For more information, see the documentation: http://openweathermapy.readthedocs.io/en/latest/#fetch-current-weather-data.

the \*\*settings syntax allows the programmer to pass as many query options as they want to the method.

# Get current weather

current\_weather\_paris = **owm.get\_current("Paris", \*\*settings)**

print(f"Current weather object for Paris: {current\_weather\_paris}.")

Openweathermapy also makes it easier to parse responses.

The application is able to isolate the "temp" value - nested under "main" and "name" - without having to traverse the JSON manually.

Openweathermapy handles traversals on the user's behalf so long as the application is provided with a list of the keys the user is interested in within a summary list.

The \*summary extracts each item from the array one by one, rather than sending in the whole list.

summary = ["name", "main.temp"]

data = current\_weather\_paris(\*summary)

print(f"The current weather summary for Paris is: {data}.")

[**Stu\_MapWrap.ipynb**](https://gw.bootcampcontent.com/GW-Coding-Boot-Camp/GWARL201906DATA1/blob/master/01-Class-Activities/06-Python-APIs/2/Activities/10-Stu_MapWrap/Solved/Stu_MapWrap.ipynb)

# Dependencies

import csv

import matplotlib.pyplot as plt

import openweathermapy as ow

import pandas as pd

# import api\_key from config file

from config import api\_key

# Create a settings object with your API key and preferred units

settings = {"units": "metric", "appid": api\_key}

**# Get data for each city in cities.csv**

**weather\_data = []**

**with open("../Resources/cities.csv") as cities\_file:**

**cities\_reader = csv.reader(cities\_file)**

**cities = [city[0] for city in cities\_reader]**

**print(cities)**

**weather\_data = [ow.get\_current(city, \*\*settings) for city in cities]**

**print(weather\_data)**

**# Create an "extracts" object to get the temperature, latitude,**

**# and longitude in each city**

**summary = ["main.temp", "coord.lat", "coord.lon"]**

**# Create a Pandas DataFrame with the results**

**data = [response(\*summary) for response in weather\_data]**

**print(data)**

**weather\_data = pd.DataFrame(data, index=cities)**

**weather\_data**

**# BONUS:**

**column\_names = ["Temperature", "Latitude", "Longitude"]**

**# provide column names with columns=column\_names**

**weather\_data = pd.DataFrame(data, index=cities, columns=column\_names)**

**weather\_data**

**DAY 3**

**Google Geocoding**

This process of converting an address to coordinates is called geocoding.

It is a good idea to avoid pushing your API key to github by using adding the config.py to their .gitignore file or using environment variables.# Dependencies

**Google\_Geocode.ipynb**

import requests

import json

from pprint import pprint

# Google developer API key

import os

# Make config.py module available

gkey = os.environ.get('gkey')

# Target city

target\_city = "Arlington, VA"

# Build the endpoint URL

target\_url = ('https://maps.googleapis.com/maps/api/geocode/json?'

'address={0}&key={1}').format(target\_city, gkey) # Run a request to endpoint and convert result to json

geo\_data = requests.get(target\_url).json()

# Pretty print the json

pprint(geo\_data)

# Print the json (using json.dumps)

print(json.dumps(geo\_data['results'][0], indent=4, sort\_keys=True))# Extract latitude and longitude

lat = geo\_data["results"][0]["geometry"]["location"]["lat"]

lng = geo\_data["results"][0]["geometry"]["location"]["lng"]

# Print the latitude and longitude

print('''

City: {0}

Latitude: {1}

Longitude: {2}

'''.format(target\_city, lat, lng)) City: Arlington, VA

Latitude: 38.8799697

Longitude: -77.1067698

**Google\_Places.ipynb**

Let's take a look at the Google Places API documentation.

Pay close attention to: Nearby Search, Text Search, and Place Search.

Dep and URL as above ^

# Nearby Places API documentation: https://developers.google.com/places/web-service/search

**target\_coordinates** = f"{lat},{lng}"

**target\_search** = "Chinese"

**target\_radius** = 1000 # cannot be used when ranking by distance

**target\_type** = "restaurant"

print(target\_coordinates)

# set up a parameters dictionary

**params = {**

"location": target\_coordinates,

"keyword": target\_search,

# "radius": target\_radius,

"type": target\_type,

"key": gkey,

"rankby": "distance" #can only be used if radius is not specified

}

# base url

base\_url = "https://maps.googleapis.com/maps/api/place/nearbysearch/json"

# run a request using our params dictionary

response = requests.get(base\_url, **params=params**)

# convert response to json

places\_data = response.json()

# Print the json (pretty printed)

print(json.dumps(places\_data, indent=4, sort\_keys=True))

**# Print the name and address of the restaurants**

**i = 1 # initialize counter**

**for p in places\_data["results"]:**

**place\_name = p["name"]**

**vicinity = p["vicinity"]**

**rating = p["rating"]**

**print(f'{i}: {place\_name}')**

**print(f'\t {vicinity}')**

**print(f'\t Rating: {rating}')**

**i += 1 # increment counter**

**Google\_That.ipynb**

# 1. What are the geocoordinates (latitude and longitude) of Seattle,

# Washington?

target\_city = "Seattle, Washington"

params = {"address": target\_city, "key": gkey}

# Build URL using the Google Maps API

base\_url = "https://maps.googleapis.com/maps/api/geocode/json"

print("Drill #1: The Geocoordinates of Seattle, WA")

# Run request

response = requests.get(base\_url, params=params)

# print the response URL, avoid doing for public GitHub repos in order to avoid exposing key

print(response.url)

# Convert to JSON

seattle\_geo = response.json()

# Extract lat/lng

lat = seattle\_geo["results"][0]["geometry"]["location"]["lat"]

lng = seattle\_geo["results"][0]["geometry"]["location"]["lng"]

# Print results

print(f"{target\_city}: {lat}, {lng}")

# 3. Find the name and address of a bike store in Seattle, Washington.

# Hint: See https://developers.google.com/places/web-service/supported\_types

target\_type = "bicycle\_store"

seattle\_coords = f'{lat},{lng}'

radius = 8000

# rewrite params dict

params = {

"location": seattle\_coords,

"types": target\_type,

"radius": radius,

"key": gkey

}

# Build URL using the Google Maps API

base\_url = "https://maps.googleapis.com/maps/api/place/nearbysearch/json"

print("Drill #3: A Bike Store in Seattle, WA")

# Run request

response = requests.get(base\_url, params)

# print the response URL, avoid doing for public GitHub repos in order to avoid exposing key

# print(response.url)

seattle\_bikes = response.json()

# Print the JSON (pretty printed)

# print(json.dumps(seattle\_bikes, indent=4, sort\_keys=True))

# Print the name and address of the first bike shop to appear

print(seattle\_bikes["results"][0]["name"])

print(seattle\_bikes["results"][0]["vicinity"])

# 6. Bonus: Find the names and addresses of the top five restaurants in your home city.

# Hint: Read about "Text Search Results"

# (https://developers.google.com/places/web-service/search#TextSearchRequests)

my\_phrase = "best restaurant in Washington, DC"

target\_url = "https://maps.googleapis.com/maps/api/place/textsearch/json"

params = {

"query": my\_phrase,

"key": gkey

}

print("Drill #6: Bonus")

response = requests.get(target\_url, params)

# print the response URL, avoid doing for public GitHub repos in order to avoid exposing key

# print(response.url)

happy\_places = response.json()

# print(json.dumps(happy\_places, indent=4, sort\_keys=True))

counter = 0

**for place in happy\_places["results"]:**

**print(place["name"])**

**print(place["formatted\_address"])**

**counter += 1**

**if counter == 5:**

**break**

**NearestRestr.ipynb ??? (REVIEW)**

# Dependencies

import pandas as pd

import numpy as np

import requests

import json

# Google developer API key

import os

# Retrieve Google API key from config.py

gkey = os.environ.get('gkey')

types\_df = pd.read\_csv("../Resources/ethnic\_restr.csv")

types\_df.head(30)

# set up additional columns to hold information

types\_df['name'] = ""

types\_df['address'] = ""

types\_df['price\_level'] = ""

types\_df['rating'] = ""

types\_df.head(30)

target\_location = "950 N Glebe Rd Arlington, VA"

target\_url = ('https://maps.googleapis.com/maps/api/geocode/json?'

'address={0}&key={1}').format(target\_location, gkey)

geo\_data = requests.get(target\_url).json()

# Extract latitude and longitude

lat = geo\_data["results"][0]["geometry"]["location"]["lat"]

lng = geo\_data["results"][0]["geometry"]["location"]["lng"]

# find the closest restaurant of each type to coordinates

base\_url = "https://maps.googleapis.com/maps/api/place/nearbysearch/json"

params = {

"location": f"{lat},{lng}",

"rankby": "distance",

"type": "restaurant",

"key": gkey,

}

* **iterrows() iterates through EACH ROW of the dataframe returning an INDEX NUMBER and the CONTENTS OF EACH ROW. Those row values can then be individually accessed using the column label like so ROW['COLUMN LABEL'].**
  + **In each iteration, the keyword value is overwritten to be the new target.**
  + **Instead of using try/except , we use .get() to retrieve results if it exists. If it does not, the results variable returns as None by default.**
  + **If results is not None, then the if statement is triggered.**
  + **.loc is used to update the cells with the desired information from the results. This time, however, we give the .get() method a second parameter, which will override the default of returning None, and instead return an empty string. This allows us to get the information that is available even if some keys are missing.**

**# use iterrows to iterate through pandas dataframe**

**for index, row in types\_df.iterrows():**

**# get restaurant type from df**

**restr\_type = row['ethnicity']**

**# add keyword to params dict**

**params['keyword'] = restr\_type**

**# assemble url and make API request**

**print(f"Retrieving Results for Index {index}: {restr\_type}.")**

**response = requests.get(base\_url, params=params).json()**

**# extract results**

**results = response['results']**

**try:**

**print(f"Closest {restr\_type} restaurant is {results[0]['name']}.")**

**types\_df.loc[index, 'name'] = results[0]['name']**

**types\_df.loc[index, 'address'] = results[0]['vicinity']**

**types\_df.loc[index, 'price\_level'] = results[0]['price\_level']**

**types\_df.loc[index, 'rating'] = results[0]['rating']**

**except (KeyError, IndexError):**

**print("Missing field/result... skipping.")**

**print("------------")**

**# params dictionary to update each iteration**

**params = {**

**"radius": 50000,**

**"types": "airport",**

**"keyword": "international airport",**

**"key": gkey**

**}**

**# Use the lat/lng we recovered to identify airports**

**for index, row in cities\_pd.iterrows():**

**# get lat, lng from df**

**lat = row["Lat"]**

**lng = row["Lng"]**

**# change location each iteration while leaving original params in place**

**params["location"] = f"{lat},{lng}"**

**# Use the search term: "International Airport" and our lat/lng**

**base\_url = "https://maps.googleapis.com/maps/api/place/nearbysearch/json"**

**# make request and print url**

**name\_address = requests.get(base\_url, params=params)**

**# print the name\_address url, avoid doing for public github repos in order to avoid exposing key**

**# print(name\_address.url)**

**# convert to json**

**name\_address = name\_address.json()**

**# print(json.dumps(name\_address, indent=4, sort\_keys=True))**

**# Since some data may be missing we incorporate a try-except to skip any that are missing a data point.**

**try:**

**cities\_pd.loc[index, "Airport Name"] = name\_address["results"][0]["name"]**

**cities\_pd.loc[index, "Airport Address"] = name\_address["results"][0]["vicinity"]**

**cities\_pd.loc[index, "Airport Rating"] = name\_address["results"][0]["rating"]**

**except (KeyError, IndexError):**

**print("Missing field/result... skipping.")# Save Data to csv**

**cities\_pd.to\_csv("Airport\_Output.csv")**

**# Visualize to confirm airport data appears**

**cities\_pd.head(10)**

**Census API Wrapper**

**In order to use the census API wrapper, you must review the documentation.**

**You will need to run !pip install census and obtain a Census API key from the US Census Bureau.**

**To save time, you can use mine -- only for the class exercises, but do get your own!!"**

**In essence, the wrapper provides a fairly easy method of retrieving data from the 2013 census based on zip code, state, district, or county.**

**Each census field (e.g. Poverty Count, Unemployment Count, Number of Asians, etc.) is denoted with a label like B201534\_10E.**

**In using the API, developers list out each of the desired fields based on their labels.**

**The results are then returned as a list of dictionaries, which can be immediately converted into a DataFrame.**

**Also, note that while the Census API is helpful, it is not the best documented!! #!pip install census**

**#!pip install pypandoc**

**#!pip install us**

**# API wrapper documentation**

**# https://github.com/datamade/census# Dependencies**

**import numpy as np**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**import requests**

**from census import Census**

**from us import states**

**# Census API Key**

**import sys**

**#sys.path.remove('../../../../')**

**#if not sys.path.index('../../../../'):**

**sys.path.append('../../../../')**

**print(sys.path)**

**from config import census\_api\_key**

**c = Census(census\_api\_key, year=2013) A few things to note in the code:**

**We use the c.acs5.get method to grab data on each of the fields we need.**

**Note that those labels do appear to be cryptic. You must read the documentation to dereference them.**

**We divide the Poverty Count by Total Population to evaluate Poverty Rate. This is because the US census doesn't calculate Poverty Rate explicitly.# Run Census Search to retrieve data on all zip codes (2013 ACS5 Census)**

**# See: https://github.com/CommerceDataService/census-wrapper for library documentation**

**# See: https://gist.github.com/afhaque/60558290d6efd892351c4b64e5c01e9b for labels**

**census\_data = c.acs5.get(("NAME", "B19013\_001E", "B01003\_001E", "B01002\_001E",**

**"B19301\_001E",**

**"B17001\_002E"), {'for': 'zip code tabulation area:\*'})**

**# Convert to DataFrame**

**census\_pd = pd.DataFrame(census\_data)**

**# Column Reordering**

**census\_pd = census\_pd.rename(columns={"B01003\_001E": "Population",**

**"B01002\_001E": "Median Age",**

**"B19013\_001E": "Household Income",**

**"B19301\_001E": "Per Capita Income",**

**"B17001\_002E": "Poverty Count",**

**"NAME": "Name", "zip code tabulation area": "Zipcode"})**

**# Add in Poverty Rate (Poverty Count / Population)**

**census\_pd["Poverty Rate"] = 100 \* \**

**census\_pd["Poverty Count"].astype(**

**int) / census\_pd["Population"].astype(int)**

**# Final DataFrame**

**census\_pd = census\_pd[["Zipcode", "Population", "Median Age", "Household Income",**

**"Per Capita Income", "Poverty Count", "Poverty Rate"]]**

**# Visualize**

**print(len(census\_pd))**

**census\_pd.head()# Save as a csv**

**# Note to avoid any issues later, use encoding="utf-8"**

**census\_pd.to\_csv("census\_data.csv", encoding="utf-8", index=False)**

**#Find data for Darts neighborhood in DC**

**census\_pd.loc[census\_pd["Zipcode"] == '20002']**

**#Find data for Darts neighborhood in Chicago**

**census\_pd.loc[census\_pd["Zipcode"] == '60637']**

**Census Activity**

**In this activity we will utilize the Census API in order to obtain census data at a state level.Instructions**

**Using this notebook as a reference, create a completely new script that calculates each of the following fields at the state level:**

**Population**

**Median Age**

**Household Income**

**Per Capita Income**

**Poverty Count**

**Poverty Rate**

**Unemployment Rate**

**Save the resulting data as a csv.**

**Next, read in the provided csv containing state centroid coordinates and merge this data with your original census data.**

**With the coordinates now appended to the dataframe, you have the ability to add markers to the base map.**

**Use the 'Poverty Rate' column to create an info\_box corresponding to each marker.**

**Hints**

**See documentation for the Census API Wrapper.**

**See documentation for Jupyter Gmaps for more information on how to create an info\_box.# Dependencies**

**import numpy as np**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**import requests**

**from census import Census**

**import gmaps**

**# Census & gmaps API Keys**

**import os**

**census\_api\_key = os.environ.get('census\_api\_key')**

**gkey = os.environ.get('gkey')**

**print(census\_api\_key)**

**print(gkey)**

**c = Census(census\_api\_key, year=2016)**

**# Configure gmaps**

**gmaps.configure(api\_key=gkey)# Run Census Search to retrieve data on all states**

**# Note the addition of "B23025\_005E" for unemployment count**

**census\_data = c.acs5.get(("NAME", "B19013\_001E", "B01003\_001E", "B01002\_001E",**

**"B19301\_001E",**

**"B17001\_002E",**

**"B23025\_005E"), {'for': 'state:\*'})**

**# Convert to DataFrame**

**census\_pd = pd.DataFrame(census\_data)**

**# Column Reordering**

**census\_pd = census\_pd.rename(columns={"B01003\_001E": "Population",**

**"B01002\_001E": "Median Age",**

**"B19013\_001E": "Household Income",**

**"B19301\_001E": "Per Capita Income",**

**"B17001\_002E": "Poverty Count",**

**"B23025\_005E": "Unemployment Count",**

**"NAME": "Name", "state": "State"})**

**# Add in Poverty Rate (Poverty Count / Population)**

**census\_pd["Poverty Rate"] = 100 \* \**

**census\_pd["Poverty Count"].astype(**

**int) / census\_pd["Population"].astype(int)**

**# Add in Employment Rate (Employment Count / Population)**

**census\_pd["Unemployment Rate"] = 100 \* \**

**census\_pd["Unemployment Count"].astype(**

**int) / census\_pd["Population"].astype(int)**

**# Final DataFrame**

**census\_pd = census\_pd[["State", "Name", "Population", "Median Age", "Household Income",**

**"Per Capita Income", "Poverty Count", "Poverty Rate", "Unemployment Rate"]]**

**census\_pd.head()# Save as a csv**

**# Note to avoid any issues later, use encoding="utf-8"**

**census\_pd.to\_csv("census\_data\_states.csv", encoding="utf-8", index=False)# Read in the csv containing state centroid coordinates**

**centroids = pd.read\_csv("../Resources/state\_centroids.csv")**

**centroids.head()# Merge the datasets using the sate columns**

**census\_data = pd.merge(census\_pd, centroids, how="left", left\_on="Name", right\_on="State")**

**# Save the updated dataframe as a csv**

**census\_data.to\_csv("../Resources/state\_census\_data.csv", encoding="utf-8", index=False)**

**census\_data.head()# Convert poverty rate as a list**

**# Convert bank rate to list**

**poverty\_rate = census\_data["Poverty Rate"].tolist()# Create a map using state centroid coordinates to set markers**

**marker\_locations = census\_data[['Latitude', 'Longitude']]**

**# Create a marker\_layer using the poverty list to fill the info box**

**fig = gmaps.figure()**

**markers = gmaps.marker\_layer(marker\_locations,**

**info\_box\_content=[f"Poverty Rate: {rate}" for rate in poverty\_rate])**

**fig.add\_layer(markers)**

**fig**

**Banking\_Desert\_HeatMap.ipynb**

**Banking and Unemployment**

The below script explores the relationship between states with high unemployment rates and bank counts per state.

In this script, we retrieved and plotted data from the 2016 US Census and Google Places API to show the relationship between various socioeconomic parameters and bank count across 700 randomly selected zip codes. We used Pandas, Numpy, Matplotlib, Requests, Census API, and Google API to accomplish our task.

# !pip install us

# Dependencies

from census import Census

import gmaps

import numpy as np

import pandas as pd

import requests

import time

from us import states

# Census & gmaps API Keys

import os

gkey = os.environ.get('gkey')

census\_api\_key = os.environ.get('census\_api\_key')

# Census API Key

c = Census(census\_api\_key, year=2016)

# Run Census Search to retrieve data on all zip codes (2013 ACS5 Census)

# See: https://github.com/CommerceDataService/census-wrapper for library documentation

# See: https://gist.github.com/afhaque/60558290d6efd892351c4b64e5c01e9b for labels

# codes for Unemployment and Population

census\_data = c.acs5.get(("B01003\_001E", "B23025\_005E"), {

'for': 'zip code tabulation area:\*'})

# Convert to DataFrame

census\_pd = pd.DataFrame(census\_data)

# Column Reordering

census\_pd = census\_pd.rename(columns={"B01003\_001E": "Population",

"B23025\_005E": "Unemployment Count",

"zip code tabulation area": "Zipcode"})

# Add in Employment Rate (Employment Count / Population)

census\_pd["Unemployment Rate"] = 100 \* \

census\_pd["Unemployment Count"].astype(

int) / census\_pd["Population"].astype(int)

# Final DataFrame

census\_pd = census\_pd[["Zipcode", "Population", "Unemployment Rate"]]

# Visualize

print(len(census\_pd))

census\_pd.head()

**Combine Data**

In [4]:

# Import the original data we analyzed earlier. Use dtype="object" to match other

census\_data\_original = pd.read\_csv(

"../Resources/zip\_bank\_data.csv", dtype="object", encoding="utf-8")

# Visualize

census\_data\_original.head()

# Merge the two data sets along zip code

census\_data\_complete = pd.merge(

census\_data\_original, census\_pd, how="left", on=["Zipcode", "Zipcode"])

# Save the revised Data Frame as a csv

census\_data\_complete.to\_csv(

"../Resources/bank\_data\_with\_employment.csv", encoding="utf-8", index=False)

# Preview

census\_data\_complete.head()

census\_data\_complete = census\_data\_complete.rename(columns={"Population\_x":"Population"})

census\_data\_complete = census\_data\_complete.drop(columns={"Population\_y"})

census\_data\_complete.head()

# Configure gmaps with API key

gmaps.configure(api\_key=gkey)

In [8]:

# Store 'Lat' and 'Lng' into locations

locations = census\_data\_complete[["Lat", "Lng"]].astype(float)

# Convert Poverty Rate to float and store

# HINT: be sure to handle NaN values

poverty\_rate = census\_data\_complete["Poverty Rate"].astype(float)

# Create a poverty Heatmap layer

fig = gmaps.figure()

heat\_layer = gmaps.heatmap\_layer(locations, weights=poverty\_rate,

dissipating=False, max\_intensity=100,

point\_radius = 1)

# Adjust heat\_layer setting to help with heatmap dissipating on zoom

heat\_layer.dissipating = False

heat\_layer.max\_intensity = 100

heat\_layer.point\_radius = 1

fig.add\_layer(heat\_layer)

fig

*Bank Rate"* is converted to a list in order to be passed in as info\_box\_content to the symbol\_layer.

# Convert bank rate to list

bank\_rate = census\_data\_complete["Bank Count"].tolist()

A symbol layer is created by passing in locations and *"Bank Rate"*. The additional arguments are stylistic and can adjusted to help clear up how the map will look. The list comprehension f"Bank amount: {bank}" for bank in bank\_rate will allow the bank data to be customized and added to the map. Finally the symbol\_layer is added to the figure and displayed.

# Create bank symbol layer

bank\_layer = gmaps.symbol\_layer(

locations, fill\_color='rgba(0, 150, 0, 0.4)',

stroke\_color='rgba(0, 0, 150, 0.4)', scale=2,

info\_box\_content=[f"Bank amount: {bank}" for bank in bank\_rate]

)

fig = gmaps.figure()

fig.add\_layer(bank\_layer)

fig

# Create a combined map

fig = gmaps.figure()

fig.add\_layer(heat\_layer)

fig.add\_layer(bank\_layer)

fig