Jack Becker

August 7, 2023

ToxAdvisor Database Guide

**INTRODUCTION**

This document is meant to serve as documentation for all code contained in the CDC\_Webscraping folder of this repository. It consists of eight Python scripts, two CSV files, two JSON files, one Jupyter notebook, and one database used to webscrape, clean, store, and query acute toxicity data from the CDC’s ***NIOSH Pocket Guide to Chemical Hazards***. The final pipeline and database, along with the predictive models built by another intern, Axel, will be used to create an app called ToxAdvisor that can provide exposure guidelines in the event of chemical emergencies, even for chemicals with limited available research.

**CDC\_SCRAPE.PY**

This script web scrapes and extracts relevant toxicity data from the CDC website. It uses various functions to navigate through primary and secondary URLs of the website to fetch toxicity data for over 600 chemicals.

* build\_df():
  + Serves as the main function in cdc\_scrape.py, which iterates through every chemical’s secondary URL on the CDC website, extracts all relevant toxicity data, and stores everything in a Pandas dataframe. The dataframe is converted into a CSV format at the end and named “scrape\_output.csv” in the repository.
* is\_valid(url\_1):
  + Checks whether a primary URL is valid by searching all HTML text for the word “Oops!”
* init\_soup(url):
  + Sends an HTTP request and initializes the BeautifulSoup package in order to access HTML data from a webpage.
* get\_url\_1\_all():
  + Because all primary URLs are indexed using a 4-digit code in the range [0000, 0686], this function iterates through every index and checks whether the URL is valid or not. If it is valid, the URL is added to an array which is returned at the end once the loop terminates.
* get\_url\_2\_all(url\_1\_all):
  + Because all secondary URLs are listed on their corresponding primary webpage and contain the information we are truly interested in, this function iterates through every valid primary URL, adds the secondary URL to an array, and returns the array once the loop terminates. The url\_2\_all array is what is actually used by the build\_df() method to find pertinent toxicity information for each chemical.
* get\_rtecs\_no(url\_2):
  + Finds and returns the RTECS number for a specific chemical from the secondary URL.
* get\_cas\_no(url\_2):
  + Finds and returns the CAS number for a specific chemical from the secondary URL.
* get\_mol\_weight(url\_2):
  + Finds and returns the molecular weight for a specific chemical from the secondary URL.
* get\_synonyms(url\_2):
  + Finds and returns a list of commonly used chemical names/synonyms from the secondary URL.
  + NOTE: When I originally wrote this function, I forgot to extract the title of the webpage which contains the primary/most common chemical name. These names were added to the database later when I wrote add\_primary\_names.py.
* get\_tox(url\_2):
  + Finds and stores all information contained in the “Acute Toxicity” table on each chemical’s secondary webpage. It is broken up into a dictionary with three keys: “ROUTE/ORGANISM”, “DOSE”, and “EFFECT”. For each chemical, this dictionary is appended to an array, which is converted into a Pandas dataframe at the end.

**CDC\_CLEAN.PY**

This script is responsible for cleaning and processing the data extracted from the CDC website. It performs tasks such as converting measurement units to a consistent format, splitting combined fields, and expanding abbreviated terminology. The final output provides a clear and standardized dataset.

**Notes**:

* The script uses the **pandas** library for data manipulation and **re** for regular expressions.
* The **type\_dict** and **route\_dict** dictionaries are used for translating specific terms in the raw data to their abbreviated forms.
* Very few entries were manually processed after running **cdc\_clean.py**, which are stored in the \_\_\_.csv file.

**Functions**:

* **parse\_dose(dose)**:
  + Processes the raw dose information from the scraped data to separate it into different components such as type, value, units, and exposure duration.
  + Utilizes regular expressions to identify and extract the relevant information.
  + Uses **type\_dict** for translating long-form text to abbreviated forms.
* **convert\_time\_to\_hours(time)**:
  + Converts various time formats (minutes, days, weeks, etc.) into hours for a consistent representation in the dataset.
* **duplicate\_cas(df)**:
  + In cases where a single entry has multiple CAS numbers associated with it, this function duplicates the entry for each CAS number, ensuring each CAS has a separate record in the dataset.
* **clean()**:
  + The main function that reads the scraped data, processes it, and outputs the cleaned data.
  + Procedure:
    1. Transforms the 'ROUTE/ORGANISM' column to upper case and splits it into 'ROUTE' and 'SPECIES' columns.
    2. Replaces the route terminology using the **route\_dict**.
    3. Applies the **parse\_dose** function to process dose information and deletes the original 'DOSE' column.
    4. Converts the 'TIME (HR)' entries using the **convert\_time\_to\_hours** function.
    5. Processes entries with multiple CAS numbers using the **duplicate\_cas** function.
    6. Makes unit representation consistent (e.g., converting 'gm' to 'g').
    7. The cleaned data is displayed using a **print** statement, but it can be saved to a CSV file by uncommenting the relevant line.

**CDC\_DATABASE.PY**

This script inserts and updates information from a variety of sources, including the clean CDC dataset, using a SQLite database called **tox\_database.db**.

* **insert\_tables(path)**:
  + Loads CSV files from the provided directory path or an individual CSV file path into the SQLite database. Each CSV file will be stored as a separate table with the table name.
* **update\_database()**:
  + Acts as a pipeline to update the **tox\_database.db** with multiple data sources, specifically from certain directories and an individual CSV file.
  + When new data needs to be added or updated, the function only needs to be run one time.
* **count\_tables()**:
  + Returns the total number of tables currently present in the SQLite database.
  + Used mainly for diagnostic purposes
* **print\_tables()**:
  + Displays the names of all tables present in the **tox\_database.db** and prints 3 randomly sampled rows from each table.
  + Used mainly for diagnostic purposes.

**QUERIES.PY**

The **queries.py** script retrieves information from the database. The module offers fine-grained control over querying parameters, allowing users to fetch data using either a CAS number or a chemical name. It utilizes the fuzzy string matching package to find close matches for chemical names and serve as an “autocorrect.”

1.**cdc\_query(cas = None, chemical = None):**

* Retrieves data specifically from the CDC database.
* **Procedure**:
  + Checks if neither **cas** nor **chemical** is provided, raising an error if true.
  + Connects to the database.
  + If a CAS number is given, it directly forms a SQL query.
  + If only a chemical name is provided, it refers to the synonyms dictionary:
    - Fetches associated CAS numbers.
    - If multiple CAS numbers are linked to the chemical, it informs the user and queries for all.
    - If no CAS is found, it uses fuzzy matching to suggest alternatives.
  + Reads the SQL query results into a pandas dataframe.
  + After iterating over the tables, it concatenates all dataframes
* **Arguments**:
  + **cas**: (Optional) CAS number of the chemical.
  + **chemical**: (Optional) Name of the chemical.
* **Returns**: A consolidated DataFrame with the queried data.

2. **Function: trv\_query(cas = None, chemical = None)**

* **Purpose**: Extracts threshold limit values (TRVs) from various regulatory and advisory sources.
* **How It Works**:
  + Operates similarly to **cdc\_query** in terms of input validation and database interaction.
  + Iterates over a larger set of tables that store TRV data.
  + Uses CAS or chemical name to extract data from these tables.
  + Offers suggestions if a direct chemical name match isn't found using fuzzy matching.
* **Arguments**:
  + **cas**: (Optional) CAS number of the chemical.
  + **chemical**: (Optional) Name of the chemical.
* **Returns**: A DataFrame combining TRV data from all the tables.

3. **Function: aux\_query(cas=None, chemical=None)**

* **Purpose**: Gathers auxiliary data on chemicals, like properties, classifications, guidance, etc.
* **How It Works**:
  + Functions similarly to the previous methods in terms of data extraction.
  + Iterates over various tables containing auxiliary chemical data.
  + Stores results in a dictionary where keys represent table names and values hold the respective DataFrames.
* **Arguments**:
  + **cas**: (Optional) CAS number of the chemical.
  + **chemical**: (Optional) Name of the chemical.
* **Returns**: A dictionary with table names as keys and the respective queried DataFrames as values.

4. **Function: query\_all(cas = None, chemical = None)**

* **Purpose**: Centralized function that aggregates all above functionalities and exports data to an Excel file.
* Calls **cdc\_query**, **trv\_query**, and **aux\_query** sequentially.
* Writes the resulting DataFrames to different sheets in an Excel file named **query\_all\_output.xlsx**.
* **Arguments**:
  + **cas**: (Optional) CAS number of the chemical.
  + **chemical**: (Optional) Name of the chemical.
* **Returns**: None. But creates and saves an Excel file with the aggregated data.

Usage Notes:

* The **final\_synonyms\_dict.json** file is crucial for resolving chemical names to their CAS numbers.
* For an optimal user experience, the **fuzzywuzzy** library suggests close-matching chemicals when an exact match is not found.
* The SQLite3 connection string and the path to the database are hardcoded. Adjust as needed for your local setup.

**test\_queries.py**

This file contains a rudimentary unit test for the database querying functionalities found in queries.py. The purpose of this file is to ensure that queries produce consistent, correct, and expected results. However, this function contains placeholder information that should be filled in later with sample chemicals and known correct outputs.

**add\_primary\_names.py**

The add\_primary\_names.py script was created to retrieve the missing chemical names from the original cdc\_scrape.py. The web scraping process is very hsimilar, but instead only pulls the title of all secondary URLs and adds it to the final\_synonyms\_dict JSON file.  
  
**Other Files**

1. **Playground.ipynb**: This Jupyter notebook serves as an experimental space where various data processing and analysis tasks were tried out before being finalized in the main scripts.
2. **excel\_to\_csv.py**: A small utility to convert Excel files to CSV format. Useful when the data source provides Excel files, but CSV processing is preferred.
3. **merge\_synonyms.py**: A script to merge chemical synonyms from various sources into a unified JSON file. It ensures that various names or terminologies used for the same chemical are acknowledged and searchable in the system based on CAS number.
4. **clean\_output.csv, scrape\_output.csv**: These are data files that contain the output from the cleaning and scraping processes respectively.
5. **primary\_name\_dict.json**, **final\_synonyms\_dict.json** JSON files that store dictionaries. The former is the output of add\_primary\_names.py, which is integrated with other sources to create final\_synonyms\_dict.json—the dictionary used by the querying functions to match chemical names to CAS numbers.
6. **tox\_database.db**: The main database file where all the scraped, cleaned, and processed data resides. This SQLite database is queried by the main application to fetch information related to toxic chemicals.Top of FormBottom of Form

**Code Snippets**

cdc\_scrape.py

import pandas as pd

import requests

from bs4 import BeautifulSoup

import time

max\_code = 686 # SPECIFIES THE TOTAL NUMBER OF URLS TO VISIT

def is\_valid(url):

response = requests.get(url)

soup = BeautifulSoup(response.text, 'html.parser')

if "Oops!" in soup.text:

return False

else:

return True

def init\_soup(url):

req = requests.get(url)

soup = BeautifulSoup(req.content, "html.parser")

return soup

def get\_url\_1\_all():

url\_1\_all = []

for code in range(0, max\_code): # was originally (42, max\_code)

num\_str = str(code).zfill(4)

url = f"https://www.cdc.gov/niosh/npg/npgd{num\_str}.html"

if is\_valid(url):

url\_1\_all.append(url)

time.sleep(1)

return url\_1\_all

def get\_url\_2\_all(url\_1\_all):

url\_2\_all = []

for url in url\_1\_all:

soup = init\_soup(url)

card\_texts = soup.find\_all("div", class\_ = "card-text")

for text in card\_texts:

a\_tag = text.find('a')

if a\_tag:

url\_2 = 'https://www.cdc.gov' + a\_tag['href']

url\_2\_all.append(url\_2)

break

return url\_2\_all

def get\_rtecs\_no\_2(url\_2):

soup = init\_soup(url\_2)

tagged = soup.find\_all(class\_ = "card-text")

if tagged:

rtecs\_no = tagged[0].text.strip()

return rtecs\_no

else:

return "NOT FOUND"

def get\_cas\_no(url):

soup = init\_soup(url)

tagged = soup.find\_all(class\_= "card-text")

if len(tagged) > 1: # Ensure the list has at least 2 elements

cas = tagged[1].text.split('\n')[1]

cas = cas.replace('\r', '')

cas = cas.strip()

return cas

else:

return "NOT FOUND"

def get\_mol\_weight(url):

soup = init\_soup(url)

tagged = soup.find\_all("div", class\_ = "card-text")

if len(tagged) > 3: # Check if the list has at least 4 elements

return tagged[3].string.strip()

else:

return "NOT FOUND"

def get\_synonyms(url):

soup = init\_soup(url)

tagged = soup.find\_all("div", class\_ = "card-text")

if len(tagged) >= 6: # Ensure there's enough divs

synonyms\_html = str(tagged[5])

synonyms = synonyms\_html.split('<br/>')

cleaned\_synonyms = [BeautifulSoup(s, "html.parser").get\_text().strip() for s in synonyms]

return cleaned\_synonyms

else:

return []

def get\_tox(url):

soup = init\_soup(url)

h2\_tag = soup.find("h2", text = "Acute Toxicity Data and References")

if h2\_tag: # Check if h2\_tag exists

table = h2\_tag.find\_next("table", class\_ = "table table-striped")

if table: # Check if table exists

headers = [header.get\_text().strip() for header in table.find\_all("th")]

rows = table.find\_all("tr")

table\_data = []

for row in rows[1:]: # skip header row

cols = row.find\_all("td")

if len(cols) >= 3: # Check if there are at least 3 columns

data\_dict = {

"ROUTE/ORGANISM": cols[0].get\_text().strip(), # Route/Organism

"DOSE": cols[1].get\_text().strip(), # Dose

"EFFECT": cols[2].get\_text().strip() if cols[2].get\_text().strip() else 'NO EFFECT REPORTED' # Effect

}

for key in data\_dict:

data\_dict[key] = ' '.join(data\_dict[key].split())

table\_data.append(data\_dict)

df = pd.DataFrame(table\_data)

return df

return pd.DataFrame() # return an empty DataFrame if no table is found

def build\_df():

url\_1\_all = get\_url\_1\_all()

url\_2\_all = get\_url\_2\_all(url\_1\_all)

rtecs\_all = []

cas\_all = []

mol\_weight\_all = []

syn\_all = []

route\_org\_all = []

dose\_all = []

effect\_all = []

for url in url\_2\_all:

rtecs\_no = get\_rtecs\_no\_2(url)

cas = get\_cas\_no(url)

mol\_weight = get\_mol\_weight(url)

syn = get\_synonyms(url)

tox = get\_tox(url)

for i in range(0, len(tox)):

tox\_row = tox.iloc[i]

route\_org\_all.append(tox\_row["ROUTE/ORGANISM"])

dose\_all.append(tox\_row["DOSE"])

effect\_all.append(tox\_row["EFFECT"])

rtecs\_all.append(rtecs\_no)

cas\_all.append(cas)

mol\_weight\_all.append(mol\_weight)

syn\_all.append(syn)

time.sleep(1)

data = {

'RTECS': rtecs\_all,

'CAS': cas\_all,

'MOLECULAR WEIGHT': mol\_weight\_all,

"SYNONYMS": syn\_all,

"ROUTE/ORGANISM": route\_org\_all,

"DOSE": dose\_all,

"EFFECT": effect\_all

}

df = pd.DataFrame(data)

print(df)

df.to\_csv("output.csv", index = False)

build\_df()

**cdc\_clean.py**

import pandas as pd

import re

import warnings

# Suppress an intel warning that doesn't matter

warnings.filterwarnings("ignore")

# Notes:

# Around a dozen entries were manually processed after running cdc\_clean.py. A few of these were removed

# for not meeting the criteria of an acute toxicity study.

# Clean up dictionary/elminate entries that are not needed.

type\_dict = {

"lethal concentration (50 percent kill)": "LC50",

"lethal concentration equivalent": "LCEQ",

"lowest reported lethal concentration": "LCLO",

"lethal dose (50 percent kill)": "LD50",

"lowest reported lethal dose": "LDLO",

"lower explosive limit": "LEL",

"level of concern": "LOC",

"permissible exposure limit": "PEL",

"toxic concentration lowest": "TCLO",

"toxic dose lowest": "TDLO",

"temporary emergency exposure limit": "TEEL",

"toxicology excellence for risk assessment": "TERA",

"threshold limit value": "TLV",

"workplace environmental exposure level": "WEEL",

"emergency exposure guidance level": "EEGL",

"acute exposure guideline level": "AEGL",

"ceiling limit": "C",

"inhibitor concentration low": "ICL",

"lowest published toxic dose": "LPTD",

"toxic concentration": "TC",

"lowest published lethal concentration": "LCLO",

"lowest published lethal dose": "LDLO",

"lethal concentration": "LC", # is this a correct abbr?

"lethal dose": "LD", # abbr?

"inhibitor concentration (5 percent kill)": "IC5",

"inhibitor concentration (10 percent kill)": "IC10",

"inhibitor concentration (15 percent kill)": "IC15",

"inhibitor concentration (20 percent kill)": "IC20",

"inhibitor concentration (25 percent kill)": "IC25",

"inhibitor concentration (30 percent kill)": "IC30",

"inhibitor concentration (35 percent kill)": "IC35",

"inhibitor concentration (40 percent kill)": "IC40",

"inhibitor concentration (45 percent kill)": "IC45",

"inhibitor concentration (50 percent kill)": "IC50",

"inhibitor concentration (55 percent kill)": "IC55",

"inhibitor concentration (60 percent kill)": "IC60",

"inhibitor concentration (65 percent kill)": "IC65",

"inhibitor concentration (70 percent kill)": "IC70",

"inhibitor concentration (75 percent kill)": "IC75",

"inhibitor concentration (80 percent kill)": "IC80",

"inhibitor concentration (85 percent kill)": "IC85",

"inhibitor concentration (90 percent kill)": "IC90",

"inhibitor concentration (95 percent kill)": "IC95",

"inhibitor concentration": "IC"

}

route\_dict = {

"INTRAPERITONEAL": "IP",

"INTRAVENOUS": "IV",

"SKIN": "DERMAL"

}

def parse\_dose(dose):

dose = dose.lower()

match = re.match(r'([^:]+): (>?)([\d.]+) (.\*?)(/\d+H)?$', dose)

if not match:

return pd.Series({'TYPE': 'NA', 'VALUE': 'NA', 'UNITS': 'NA', 'TIME (HR)': 'NA'})

type\_, gt, value, unit, time = match.groups()

type\_abbr = 'NA'

for key, dict\_value in type\_dict.items():

if key in type\_:

type\_abbr = dict\_value

break

unit\_split = unit.split('/')

parts\_per = any(unit.startswith(pp) for pp in ['ppm', 'ppb', 'pph', 'ppt'])

if parts\_per and len(unit\_split) == 1:

unit = unit\_split[0]

time = 'NA'

elif parts\_per and len(unit\_split) == 2:

unit = unit\_split[0]

time = unit\_split[1]

elif parts\_per and len(unit\_split) == 3:

return pd.Series({'TYPE': 'DROP\_ROW', 'VALUE': 'DROP\_ROW', 'UNITS': 'DROP\_ROW', 'TIME (HR)': 'DROP\_ROW'})

elif(len(unit\_split)) == 1:

unit = unit\_split[0]

time = 'NA'

elif len(unit\_split) == 2:

unit = unit\_split[0] + '/' + unit\_split[1]

time = "NA"

elif len(unit\_split) == 3:

unit = unit\_split[0] + '/' + unit\_split[1]

time = unit\_split[2]

elif len(unit\_split) == 4:

return pd.Series({'TYPE': 'DROP\_ROW', 'VALUE': 'DROP\_ROW', 'UNITS': 'DROP\_ROW', 'TIME (HR)': 'DROP\_ROW'})

else:

unit = 'NA'

time = 'NA'

return pd.Series({

'TYPE': type\_abbr,

'VALUE': f"{gt}{value}",

'UNITS': unit.strip(),

'TIME (HR)': time if time != 'NA' else 'NA'

})

def convert\_time\_to\_hours(time):

if pd.isnull(time) or time == "NA":

return time

time\_regex = re.compile(r'(\d+\.?\d\*)([A-Za-z]+)')

match = time\_regex.match(time)

if match:

number, unit = match.groups()

number = float(number)

unit = unit.upper()

if unit in ['H', 'HR', 'HRS', 'HOUR', 'HOURS']:

return number

elif unit in ['D', 'DS', 'DAY', 'DAYS']:

return number \* 24

elif unit in ['M', 'MS', 'MIN', 'MINS', 'MINUTE', 'MINUTES']:

return number / 60

elif unit in ['S', 'SEC', 'SECS', 'SECOND', 'SECONDS']:

return number / 3600

elif unit in ['W', 'WK', 'WKS', 'WEEK', 'WEEKS']:

return number \* 24 \* 7

elif unit in ['MO', 'MON', 'MONS', 'MONTH', 'MONTHS']:

return number \* 24 \* 30

elif unit in ['Y', 'YR', 'YRS', 'YEAR', 'YEARS']:

return number \* 24 \* 365

return "NA"

def duplicate\_cas(df):

def split\_cas(row):

if isinstance(row['CAS'], str):

cas\_numbers = row['CAS'].split(';')

new\_rows = pd.DataFrame([row]\*len(cas\_numbers))

new\_rows['CAS'] = cas\_numbers

return new\_rows

else:

return pd.DataFrame([row])

df\_series = df.apply(split\_cas, axis = 1)

new\_df = pd.concat(df\_series.values).reset\_index(drop = True)

return new\_df

def clean():

df = pd.read\_csv("/Users/jackbecker/xforce/CDC\_Webscraping/scrape\_output.csv")

df['ROUTE/ORGANISM'] = df['ROUTE/ORGANISM'].str.upper()

df[['ROUTE', 'SPECIES']] = df['ROUTE/ORGANISM'].str.split('/', expand = True)

del df['ROUTE/ORGANISM']

df['ROUTE'] = df['ROUTE'].replace(route\_dict)

df = df.join(df['DOSE'].apply(parse\_dose))

df = df[df['TYPE'] != 'DROP\_ROW']

del df['DOSE']

df = df.apply(lambda s: s.str.upper() if s.name != 'UNITS' else s)

df['TIME (HR)'] = df['TIME (HR)'].apply(convert\_time\_to\_hours)

df = duplicate\_cas(df)

df['UNITS'] = df['UNITS'].map(lambda x: x.replace('gm', 'g') if isinstance(x, str) else x)

# df.to\_csv("/Users/jackbecker/xforce/CDC\_Webscraping/clean\_output.csv", index = False)

print(df.sample(10))

clean()

**cdc\_database.py**

import pandas as pd

import sqlite3

import os

# 22 tables in database

def insert\_tables(path):

db\_path = "/Users/jackbecker/xforce/CDC\_Webscraping/tox\_database.db"

conn = sqlite3.connect(db\_path)

if os.path.isdir(path):

for filename in os.listdir(path):

if filename.endswith(".csv"):

df = pd.read\_csv(os.path.join(path, filename))

df.to\_sql(name = os.path.splitext(filename)[0], con = conn, if\_exists = 'replace')

else:

if path.endswith('.csv'):

df = pd.read\_csv(path)

df.to\_sql(name = os.path.splitext(os.path.basename(path))[0], con = conn, if\_exists = 'replace')

cur = conn.cursor()

cur.execute("SELECT name FROM sqlite\_master WHERE type = 'table';")

tables = cur.fetchall()

for table\_name in tables:

table\_name = table\_name[0]

print(f"Table name: {table\_name}")

cur.execute(f"SELECT \* FROM {table\_name} ORDER BY RANDOM() LIMIT 5;")

rows = cur.fetchall()

for row in rows:

print(row)

print('\n')

conn.close()

def update\_database():

db\_path = "/Users/jackbecker/xforce/CDC\_Webscraping/tox\_database.db"

conn = sqlite3.connect(db\_path)

cur = conn.cursor()

insert\_tables('/Users/jackbecker/xforce/TRV\_Lists\_initial/Auxiliary\_dataframes/TRV/')

insert\_tables('/Users/jackbecker/xforce/TRV\_Lists\_initial/Auxiliary\_dataframes/')

insert\_tables("/Users/jackbecker/xforce/CDC\_Webscraping/clean\_output.csv")

conn.close()

def count\_tables():

db\_path = "/Users/jackbecker/xforce/CDC\_Webscraping/tox\_database.db"

conn = sqlite3.connect(db\_path)

cur = conn.cursor()

cur.execute("SELECT COUNT(\*) FROM sqlite\_master WHERE type='table';")

count = cur.fetchone()[0]

conn.close()

return count

def print\_tables():

db\_path = "/Users/jackbecker/xforce/CDC\_Webscraping/tox\_database.db"

conn = sqlite3.connect(db\_path)

cur = conn.cursor()

table\_names = set()

cur.execute("SELECT name FROM sqlite\_master WHERE type = 'table';")

tables = cur.fetchall()

for table\_name in tables:

table\_names.add(table\_name[0])

for name in table\_names:

print(f"Table name: {name}")

cur.execute(f"SELECT \* FROM {name} ORDER BY RANDOM() LIMIT 3;")

rows = cur.fetchall()

for row in rows:

print(row)

print('\n')

conn.close()

**queries.py**

import pandas as pd

import sqlite3

import json

import os

from fuzzywuzzy import process

def cdc\_query(cas = None, chemical = None):

if not cas and not chemical:

raise ValueError("Must specify either CAS number and/or chemical name")

chemical = chemical.upper() if chemical else None

db\_path = "/Users/jackbecker/xforce/CDC\_Webscraping/tox\_database.db"

conn = sqlite3.connect(db\_path)

tables = ['clean\_output']

results = []

with open("/Users/jackbecker/xforce/CDC\_Webscraping/final\_synonyms\_dict.json", "r") as file:

final\_synonyms\_dict = json.load(file)

for table in tables:

if cas:

query = f"SELECT \* FROM {table} WHERE CAS = ?"

params = (cas,)

elif chemical:

chemical\_cas\_numbers = [cas for cas, names in final\_synonyms\_dict.items() if chemical in names]

if len(chemical\_cas\_numbers) > 1:

print(f"The chemical {chemical} corresponds to multiple CAS numbers: {chemical\_cas\_numbers}.")

query = f"SELECT \* FROM {table} WHERE CAS IN ({','.join(['?' for \_ in chemical\_cas\_numbers])})"

params = tuple(chemical\_cas\_numbers) # Convert list to tuple

elif not chemical\_cas\_numbers:

matches = process.extract(chemical, [name for names in final\_synonyms\_dict.values() for name in names], limit=5)

if matches:

suggested\_chemicals = [match[0] for match in matches]

suggested\_cas\_numbers = [cas for cas, names in final\_synonyms\_dict.items() if any(name in suggested\_chemicals for name in names)]

print(f"Did you mean one of these chemicals? {', '.join([f'{chemical} (CAS: {cas})' for chemical, cas in zip(suggested\_chemicals, suggested\_cas\_numbers)])}")

else:

print("Chemical not found. Please enter a valid CAS number and/or name")

return

else:

query = f"SELECT \* FROM {table} WHERE CAS = ?"

params = (chemical\_cas\_numbers[0],)

else:

raise ValueError("Must specify either CAS number and/or chemical name (preferably CAS)")

df = pd.read\_sql\_query(query, conn, params=params)

results.append(df)

df\_final = pd.concat(results, ignore\_index = True).drop\_duplicates()

conn.close()

return df\_final

def trv\_query(cas = None, chemical = None):

if not cas and not chemical:

raise ValueError("Must specify either CAS number and/or chemical name")

chemical = chemical.upper() if chemical else None

db\_path = "/Users/jackbecker/xforce/CDC\_Webscraping/tox\_database.db"

conn = sqlite3.connect(db\_path)

tables = ['TRV\_ACGIH\_TLV\_dataframe', 'TRV\_AIHA\_WEEL\_dataframe', 'TRV\_NIOSH\_REL\_dataframe', 'TRV\_EPA\_LOC\_dataframe',

'TRV\_DoE\_PAC\_dataframe', 'TRV\_OSHA\_PEL\_dataframe', 'TRV\_Cal\_OSHA\_PEL\_dataframe', 'TRV\_NRC\_EEGL\_dataframe',

'TRV\_EPA\_AEGL\_dataframe', 'TRV\_OARS\_WEEL\_dataframe', 'TRV\_AIHA\_ERPG\_dataframe']

results = []

with open("/Users/jackbecker/xforce/CDC\_Webscraping/final\_synonyms\_dict.json", "r") as file:

final\_synonyms\_dict = json.load(file)

for table in tables:

if cas:

query = f"SELECT \* FROM {table} WHERE CAS = ?"

params = (cas,)

elif chemical:

chemical\_cas\_numbers = [cas for cas, names in final\_synonyms\_dict.items() if chemical in names]

if len(chemical\_cas\_numbers) > 1:

print(f"The chemical {chemical} corresponds to multiple CAS numbers: {chemical\_cas\_numbers}.")

query = f"SELECT \* FROM {table} WHERE CAS IN ({','.join(['?' for \_ in chemical\_cas\_numbers])})"

params = tuple(chemical\_cas\_numbers)

elif not chemical\_cas\_numbers:

matches = process.extract(chemical, [name for names in final\_synonyms\_dict.values() for name in names], limit=5)

if matches:

suggested\_chemicals = [match[0] for match in matches]

suggested\_cas\_numbers = [cas for cas, names in final\_synonyms\_dict.items() if any(name in suggested\_chemicals for name in names)]

print(f"Did you mean one of these chemicals? {', '.join([f'{chemical} (CAS: {cas})' for chemical, cas in zip(suggested\_chemicals, suggested\_cas\_numbers)])}")

else:

print("Chemical not found. Please enter a valid CAS number and/or name")

return

else:

query = f"SELECT \* FROM {table} WHERE CAS = ?"

params = (chemical\_cas\_numbers[0],)

else:

raise ValueError("Must specify either CAS number and/or chemical name (preferably CAS)")

df = pd.read\_sql\_query(query, conn, params=params)

results.append(df)

df\_final = pd.concat(results, ignore\_index = True).drop\_duplicates()

conn.close()

return df\_final

def aux\_query(cas = None, chemical = None):

if not cas and not chemical:

raise ValueError("Must specify either CAS number and/or chemical name")

chemical = chemical.upper() if chemical else None

db\_path = "/Users/jackbecker/xforce/CDC\_Webscraping/tox\_database.db"

conn = sqlite3.connect(db\_path)

tables = ['chemical\_properties', 'ghs\_classifications', 'markush\_children', 'markush\_parent', 'niosh\_guidance', 'odor\_threshold']

results = {}

with open("/Users/jackbecker/xforce/CDC\_Webscraping/final\_synonyms\_dict.json", "r") as file:

final\_synonyms\_dict = json.load(file)

for table in tables:

if cas:

query = f"SELECT \* FROM {table} WHERE CAS = ?"

params = (cas,)

elif chemical:

chemical\_cas\_numbers = [cas for cas, names in final\_synonyms\_dict.items() if chemical in names]

if len(chemical\_cas\_numbers) > 1:

print(f"The chemical {chemical} corresponds to multiple CAS numbers: {chemical\_cas\_numbers}.")

query = f"SELECT \* FROM {table} WHERE CAS IN ({','.join(['?' for \_ in chemical\_cas\_numbers])})"

params = tuple(chemical\_cas\_numbers)

elif not chemical\_cas\_numbers:

matches = process.extract(chemical, [name for names in final\_synonyms\_dict.values() for name in names], limit=5)

if matches:

suggested\_chemicals = [match[0] for match in matches]

suggested\_cas\_numbers = [cas for cas, names in final\_synonyms\_dict.items() if any(name in suggested\_chemicals for name in names)]

print(f"Did you mean one of these chemicals? {', '.join([f'{chemical} (CAS: {cas})' for chemical, cas in zip(suggested\_chemicals, suggested\_cas\_numbers)])}")

else:

print("Chemical not found. Please enter a valid CAS number and/or name")

return

else:

query = f"SELECT \* FROM {table} WHERE CAS = ?"

params = (chemical\_cas\_numbers[0],)

else:

raise ValueError("Must specify either CAS number and/or chemical name (preferably CAS)")

df = pd.read\_sql\_query(query, conn, params = params)

results[table] = df

conn.close()

return results

def query\_all(cas\_list = None, chemical\_list = None):

# Checking for input validity

if not cas\_list and not chemical\_list:

raise ValueError("Must specify either CAS numbers list and/or chemical names list")

# Ensure both cas\_list and chemical\_list are lists (even if they contain only one item)

if cas\_list and not isinstance(cas\_list, list):

cas\_list = [cas\_list]

if chemical\_list and not isinstance(chemical\_list, list):

chemical\_list = [chemical\_list.upper() for chemical in chemical\_list]

# Looping over the provided CAS numbers and chemical names

for idx, (cas, chemical) in enumerate(zip(cas\_list or [None] \* len(chemical\_list), chemical\_list or [None] \* len(cas\_list))):

df\_cdc = cdc\_query(cas, chemical)

df\_trv = trv\_query(cas, chemical)

aux\_data = aux\_query(cas, chemical)

# Creating a unique file name for each chemical/CAS

filename = f"/Users/jackbecker/xforce/CDC\_Webscraping/query\_all\_output\_{idx}.xlsx"

# Ensure no overwrite happens

counter = 1

while os.path.exists(filename):

filename = f"/Users/jackbecker/xforce/CDC\_Webscraping/query\_all\_output\_{idx}\_{counter}.xlsx"

counter += 1

with pd.ExcelWriter(filename, engine='openpyxl') as writer:

if not df\_cdc.empty:

df\_cdc.to\_excel(writer, sheet\_name='CDC Data', index=False)

if not df\_trv.empty:

df\_trv.to\_excel(writer, sheet\_name='TRV Data', index=False)

for table, df in aux\_data.items():

if not df.empty:

df.to\_excel(writer, sheet\_name=table, index=False)

print(f"Data for {chemical or cas} saved to {filename}")

**test\_queries.py**

import unittest

import pandas as pd

import os

from queries import cdc\_query, trv\_query, aux\_query, query\_all

class TestQueries(unittest.TestCase):

def test\_cdc\_query(self):

cas\_test\_values = ["SAMPLE\_CAS\_1", "SAMPLE\_CAS\_2"] # Put real CAS numbers here for testing

expected\_outputs = [] # Put expected dataframes here

for cas, expected\_output in zip(cas\_test\_values, expected\_outputs):

result = cdc\_query(cas = cas)

# Checks if dataframes are equal, may need adjustment to accomodate for negligible differences

pd.testing.assert\_frame\_equal(result, expected\_output)

def test\_trv\_query(self):

cas\_test\_values = ["SAMPLE\_CAS\_1", "SAMPLE\_CAS\_2"]

expected\_outputs = []

for cas, expected\_output in zip(cas\_test\_values, expected\_outputs):

result = trv\_query(cas = cas)

pd.testing.assert\_frame\_equal(result, expected\_output)

def test\_aux\_query(self):

cas\_test\_values = ["SAMPLE\_CAS\_1", "SAMPLE\_CAS\_2"]

expected\_outputs = []

for cas, expected\_output in zip(cas\_test\_values, expected\_outputs):

result = aux\_query(cas = cas)

for key in result:

pd.testing.assert\_frame\_equal(result[key], expected\_output[key])

def test\_query\_all(self):

cas\_test\_values = ["SAMPLE\_CAS\_1", "SAMPLE\_CAS\_2"]

expected\_files = ["EXPECTED\_PATH\_1", "EXPECTED\_PATH\_2"]

for cas, expected\_file in zip(cas\_test\_values, expected\_files):

query\_all(cas\_list=[cas])

self.assertTrue(os.path.exists(expected\_file))

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

**add\_primary\_names.py**

import requests

from bs4 import BeautifulSoup

import time

import json

def is\_valid(url):

response = requests.get(url)

soup = BeautifulSoup(response.text, 'html.parser')

if "Oops!" in soup.text:

return False

else:

return True

def init\_soup(url):

req = requests.get(url)

soup = BeautifulSoup(req.content, "html.parser")

return soup

def get\_url\_1\_all():

url\_1\_all = []

max\_code = 686

for code in range(0, max\_code):

num\_str = str(code).zfill(4)

url = f"https://www.cdc.gov/niosh/npg/npgd{num\_str}.html"

if is\_valid(url):

url\_1\_all.append(url)

time.sleep(1)

return url\_1\_all

def get\_url\_2\_all(url\_1\_all):

url\_2\_all = []

for url in url\_1\_all:

soup = init\_soup(url)

card\_texts = soup.find\_all("div", class\_ = "card-text")

for text in card\_texts:

a\_tag = text.find('a')

if a\_tag:

url\_2 = 'https://www.cdc.gov' + a\_tag['href']

url\_2\_all.append(url\_2)

break

return url\_2\_all

def get\_cas\_no(url):

soup = init\_soup(url)

tagged = soup.find\_all(class\_ = "card-text")

if len(tagged) > 1: # Ensure the list has at least 2 elements

cas = tagged[1].text.split('\n')[1]

cas = cas.replace('\r', '')

cas = cas.strip()

return cas

else:

return "NOT FOUND"

def get\_primary\_name(url):

soup = init\_soup(url)

h1 = soup.find\_all("h1")

if h1:

return h1[0].text.strip().upper() # Use .text to get the text content of the tag, and then convert it to uppercase

else:

return "NOT FOUND"

def build\_name\_dict():

url\_1\_all = get\_url\_1\_all()

url\_2\_all = get\_url\_2\_all(url\_1\_all)

prim\_name\_dict = {} # Create a new dictionary to map CAS numbers to chemical names

for url in url\_2\_all:

cas = get\_cas\_no(url)

name = get\_primary\_name(url) # Get the chemical name from the webpage

if name is not "NOT FOUND":

prim\_name\_dict[cas] = name # Add the CAS number and chemical name to the dictionary

time.sleep(1)

return prim\_name\_dict

# Call the function to build the dictionary

prim\_name\_dict = build\_name\_dict()

# Save the dictionary to a JSON file

with open('prim\_name\_dict.json', 'w') as f:

json.dump(prim\_name\_dict, f)

**merge\_synonyms.py**

import sqlite3

import json

## merge\_synonyms.py updates

def get\_unique\_synonyms():

db\_path = "/Users/jackbecker/xforce/CDC\_Webscraping/tox\_database.db"

conn = sqlite3.connect(db\_path)

cur = conn.cursor()

synonyms\_dict = {}

# Read the prim\_name\_dict.json file as a dictionary

with open('primary\_name\_dict.json', 'r') as f:

primary\_name\_dict = json.load(f)

cur.execute("SELECT \* FROM synonyms;")

rows = cur.fetchall()

for row in rows:

cas\_numbers = str(row[1]).strip().split(";")

for cas\_number in cas\_numbers:

if cas\_number != "None":

synonym1 = row[2] if row[2] else ''

synonym2 = row[3] if row[3] else ''

synonyms = synonym1 + ';' + synonym2

synonyms = synonyms.upper()

if synonyms:

synonyms\_list = synonyms.split(';')

for synonym in synonyms\_list:

if synonym:

synonyms\_dict.setdefault(cas\_number, set()).add(synonym)

cur.execute("SELECT \* FROM clean\_output;")

rows = cur.fetchall()

for row in rows:

cas\_numbers = str(row[2]).strip().split(";")

for cas\_number in cas\_numbers:

if cas\_number != "None":

synonyms = row[4].upper()

if synonyms:

synonyms = [syn.strip() for syn in synonyms.strip("[]").split(',')]

synonyms\_dict.setdefault(cas\_number, set()).update(synonyms)

# Add each chemical name to the synonyms\_dict using its CAS number as the key

for cas\_number, chem\_name in primary\_name\_dict.items():

cas\_numbers = cas\_number.split(";")

for cas\_number in cas\_numbers:

if cas\_number != "None":

synonyms\_dict.setdefault(cas\_number, set()).add(chem\_name)

for cas\_number in synonyms\_dict:

synonyms\_dict[cas\_number] = list(synonyms\_dict[cas\_number])

conn.close()

return synonyms\_dict

def export\_dict\_to\_json(dictionary, file\_path):

with open(file\_path, 'w') as jsonfile:

json.dump(dictionary, jsonfile)

export\_dict\_to\_json(get\_unique\_synonyms(), '/Users/jackbecker/xforce/CDC\_Webscraping/final\_synonyms\_dict.json')

**excel\_to\_csv.py**

import pandas as pd

def excel\_to\_csv(excel\_path, output\_folder):

xlsx = pd.ExcelFile(excel\_path)

for sheet\_name in xlsx.sheet\_names:

df = pd.read\_excel(excel\_path, sheet\_name= sheet\_name)

csv\_path = f"{output\_folder}/{sheet\_name}.csv"

df.to\_csv(csv\_path, index = False)

def csv\_to\_excel(csv\_path, output\_folder):

pass

excel\_path = '/Users/jackbecker/xforce/TRV\_Lists\_initial/Auxiliary\_dataframes/niosh\_guidance\_dictionary.xlsx'

output\_folder = '/Users/jackbecker/xforce/TRV\_Lists\_initial/Auxiliary\_dataframes/'

excel\_to\_csv(excel\_path, output\_folder)