Gnarfle Documentation

# What is Gnarfle?

Gnarfle is a software platform which aids job seekers in choosing the best job title and words for their resume to be selected by today’s modern Applicant Tracking Systems. (ATS)

Gnarfle creates a word count table for a given job description. Users can then examine that list to determine what, if any, important words are missing in their resume or other job application documentation. In Gnarfle, a user can enter a phrase such as “Technical Writer” or “Java Developer.” Gnarfle then automatically scans for that job phrase in indeed.com across the top 20 metro markets in the US. The job descriptions matching that phrase are saved to a database. All job descriptions in the database are parsed, and a frequency count of individual words in the database is generated into an Excel File.

Todays ATS weight heavily job titles. It is important to phrase your job title such that it matches the job title of as many job descriptions as possible. Gnarfle allows users to enter different phrases, such as: Marketing Manager, Manager of Marketing, Marketing Leader, etc. then searches the database for all permutations of the words to determine which words bring up the most job titles.

Or if you’re just interested in the python, you can jump right to the cool [Python Stuff](#_Cool_Python_Stuff)

# Using Gnarfle

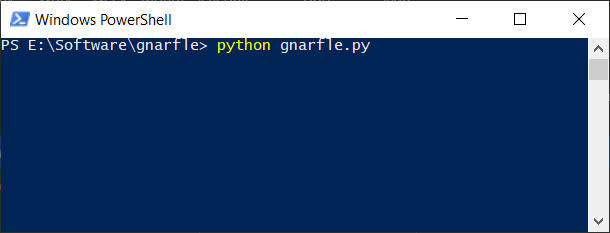
## Set up

The code must be cloned from GitHub, and an account to the underlying Gnarfle database must be created by Nic Feliccia. Send requests to [nic@secretsmokestack.com](mailto:nic@secretsmokestack.com). A virtual environment with all appropriate packages can be created, or it can be run in the main environment.

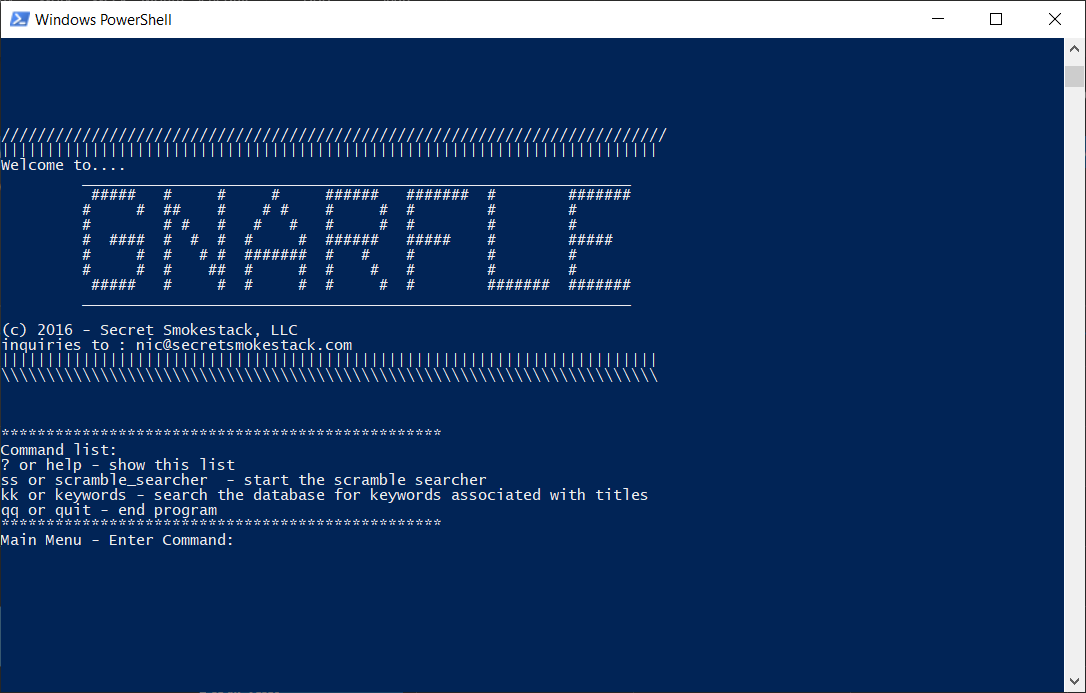
## Starting Gnarfle

I recommend using Windows PowerShell, but cmd or GitBash can also be used.

Start the program by calling python Gnarfle.py



The main menu will show when ready.



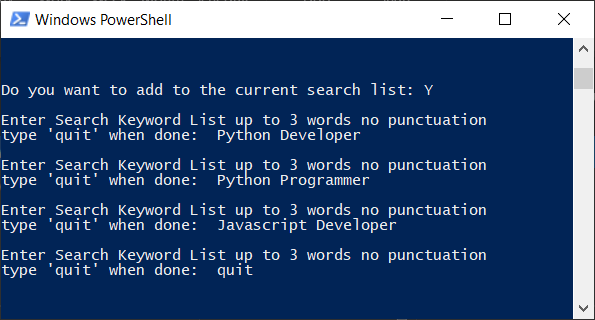
## Keywords Commands

Its recommended to start with the *kk* or *keywords* commands to ensure the information you are searching for is there. The user will be asked:

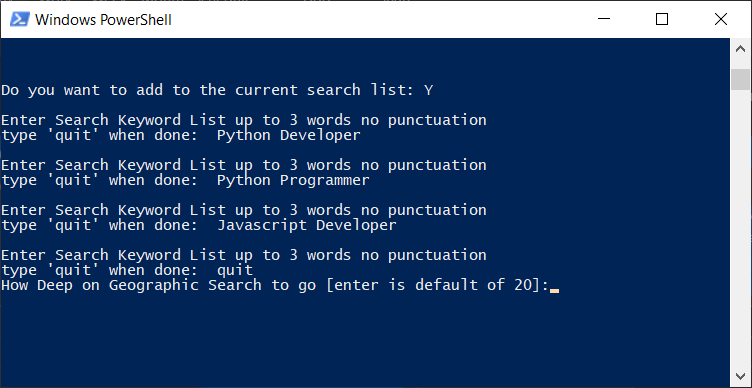
*Do you want to add to the current search list:*

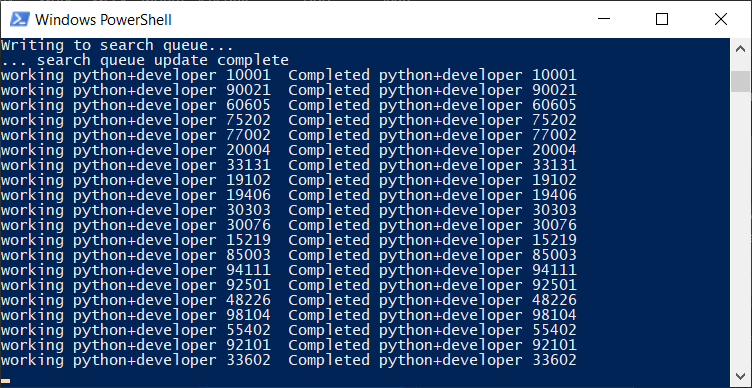
This references the list of phrases to search stored on the main database.

Entering ‘y’ or ‘Y’ brings up the search keyword entry dialog list. This loop is exited by the word ‘quit’

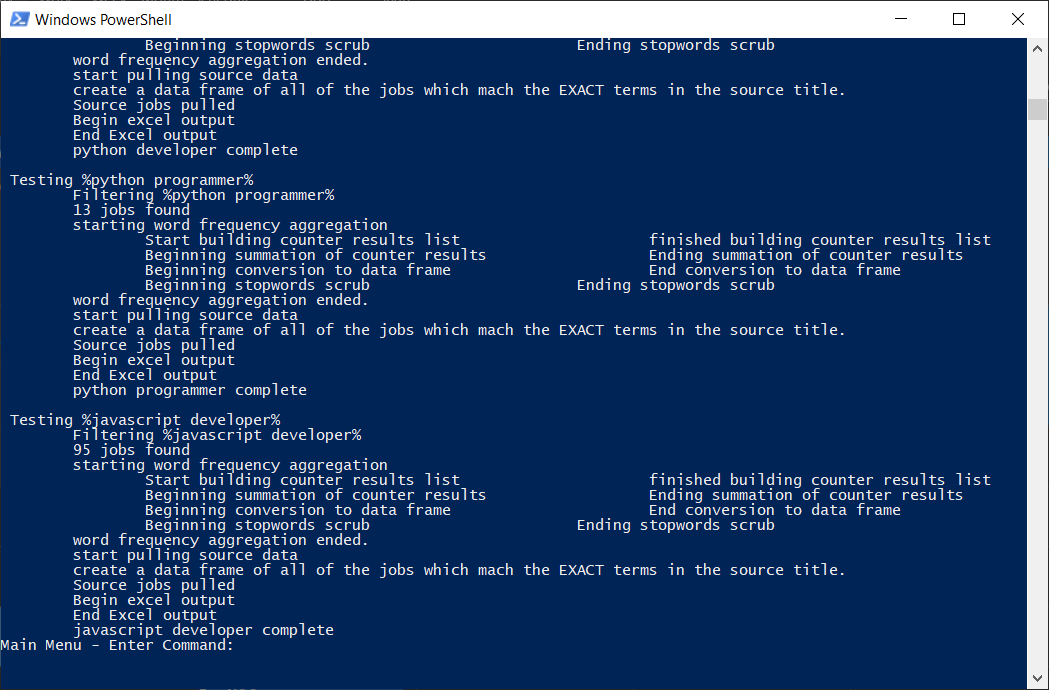


The user is next able to enter the number of different cities to search. Currently there are 25 in the database, default is 20.

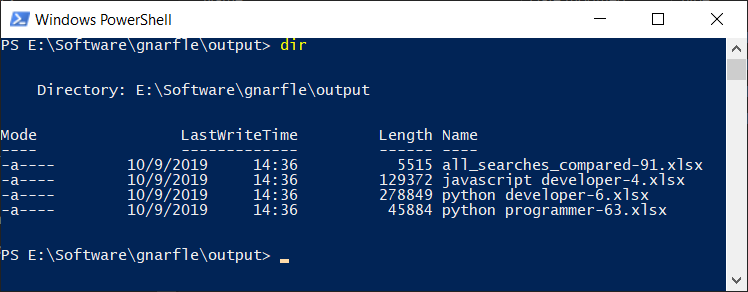


The status of the searches will update listing the key phrase searched along with the zip codes. 

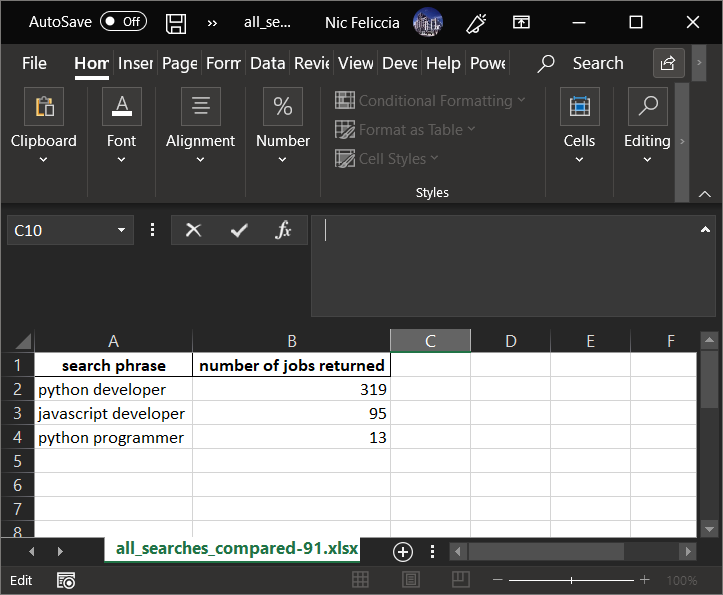
Upon Completion, the program will output information about the count of jobs found



The output directory will have excel spreadsheets for each individual search.



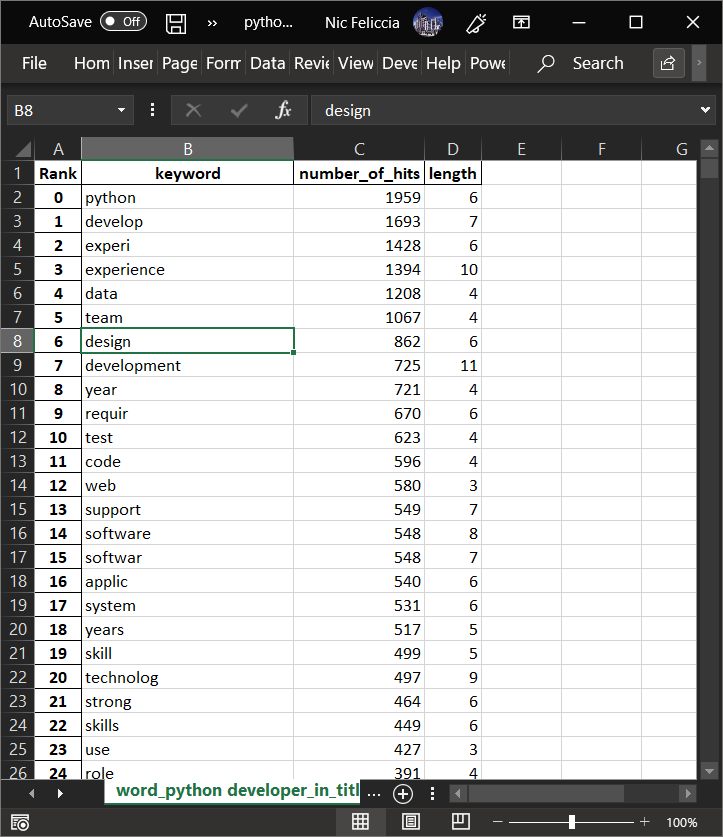
The all searches compared file will have a table summarizing the number of jobs returned on each keyword.



## Keywords Output

The Keywords Output file has 4 columns

* Rank – Words ranked in frequency from 0-~1000.
* Keyword – a one, two, or three word phrase found in the job descriptions
* Number of Hits – number of times that word was counted across all job descriptions
* Length – Length of the keyword. Note this can be used for sorting on longer phrases.

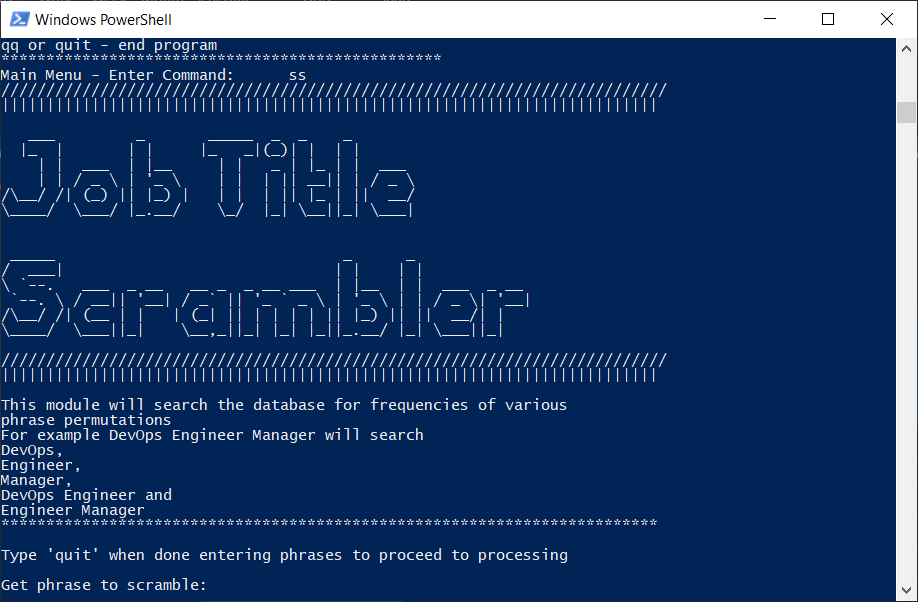


Note, the list includes what are called “Stemwords” In the above example we see that row 11 is “requir” This is a sum count of all words such as require, required, requirements, and requires

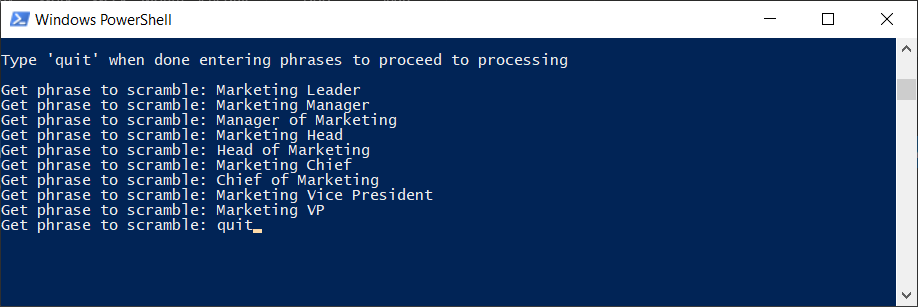
## Search Scrambler

There are many ways to phrase a job title, such as Marketing Manager, Manager of Marketing, Marketing Leader, Marketing Management, etc.

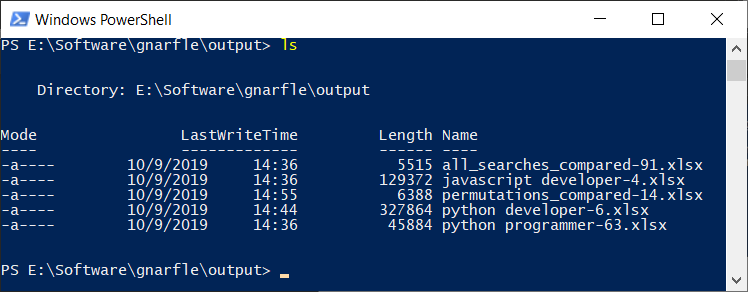
Selecting ss at the main prompt brings up the scramble searcher



Let’s enter the sequence above.



After completion another file will appear called “permutations\_compared-##.xlsx”

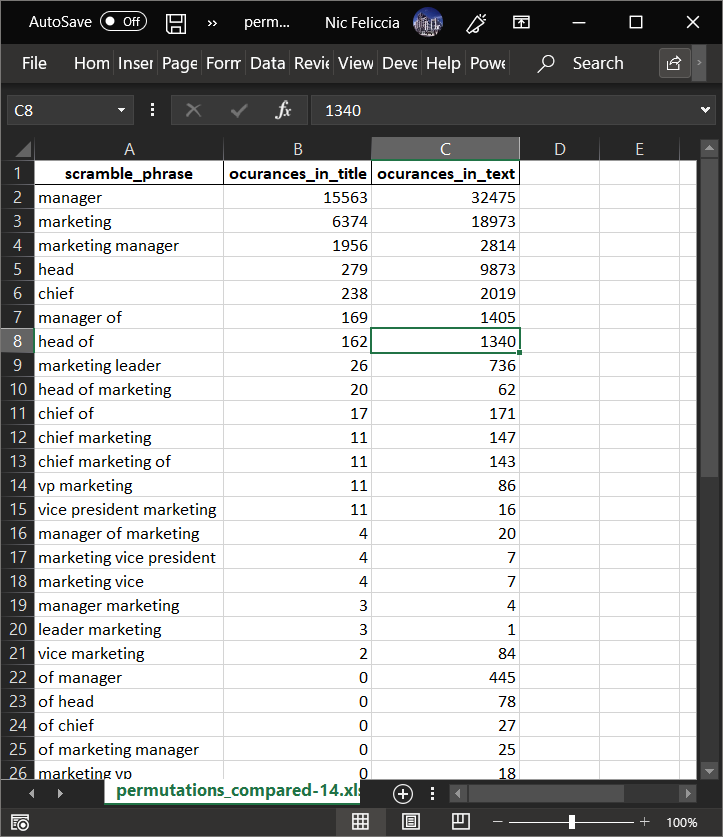


This has three columns:

* Scramble Phrase – the word or group of words tested
* Occurrences in Title – The count of job descriptions with scramble phrase in the title
* Occurrences in Text – The count of job descriptions with that phrase in the text.

As you can see in the screenshot. There are a lot of jobs with manager and marketing in the title, but these aren’t specific enough. In Row 4 you see marketing manager, with 1956 title occurrences, compared to marketing leader with 26 occurrences and head of marketing which is 20 occurrences.

Therefore, our hypothetical candidate would do best to brand themselves a marketing manager to be caught by automatic search and ATS systems.



# Cool Python Stuff

This project would not be possible without Python’s extensive libraries support. I’d like to outline how a few libraries were key in creating Gnarfle.

## Pandas

Counted aggregations are saved into a pandas DataFrame which allows easy sorting…

job\_title\_word\_count\_df.sort\_values(by=['number\_of\_hits', 'length'], ascending=False, inplace=True)

and easy output to Excel Files

with pd.ExcelWriter(filename, engine='xlsxwriter', mode='a+') as my\_excel\_homeboy:  
 job\_title\_word\_count\_df.to\_excel(my\_excel\_homeboy, sheet\_name=excel\_sheet\_1[0:30], index\_label='Rank')  
 job\_title\_source\_data.to\_excel(my\_excel\_homeboy, sheet\_name=excel\_sheet\_2[0:30], index\_label='Rank')  
my\_excel\_homeboy.save()

## Counter

The high-performance Counter object is at the heart of the summary. The most powerful feature is the ‘.update’ method of Counter allows multiple counter objects to be summed. This would be much more laborious with other methods. Using list comprehension, I create a list of single job description counter objects and simply sum them through a loop to get the aggregate.

def description\_word\_frequency\_aggregator(in\_job\_title\_result\_set: Query):  
 *"""  
 This function takes in the results for a query and aggregates the single description counter.* ***:param*** *in\_job\_title\_result\_set: Query* ***:return****: pandas data frame  
 """* counter\_results\_sum = Counter()  
 print(f'\t\tStart building counter results list', end='\t')  
 counter\_results\_list = [word\_frequency\_for\_a\_single\_description(in\_result.job\_text\_raw) for in\_result in  
 in\_job\_title\_result\_set]  
 print(f'\t\tfinished building counter results list')  
 print(f'\t\tBeginning summation of counter results ', end='\t')  
 # I use the .update command because its supposed to be the fastest to sum up counters.  
 for counter\_result in counter\_results\_list:  
 counter\_results\_sum.update(counter\_result)  
 print(f'\t\tEnding summation of counter results')  
 result\_to\_present = counter\_results\_sum.most\_common(1300)  
 print(f'\t\tBeginning conversion to data frame', end='\t')  
 word\_count\_df = pd.DataFrame(result\_to\_present, columns=['keyword', 'number\_of\_hits'])  
 print(f'\t\tEnd conversion to data frame')  
 print(f'\t\tBeginning stopwords scrub', end='\t')  
 word\_count\_df = stopwords\_scrub(word\_count\_df)  
 print(f'\t\tEnding stopwords scrub')  
 return word\_count\_df

## SQLAlchemy

SQL Alchemy has a variety of Datatypes specifically for interfacing between Python and a database

from sqlalchemy import Integer, String, Date, Float, Column, Boolean  
from sqlalchemy import create\_engine  
from sqlalchemy import text  
from sqlalchemy.ext.declarative import declarative\_base  
from sqlalchemy.orm import sessionmaker  
from sqlalchemy.orm.query import Query  
from sqlalchemy.orm.session import Session

from sqlalchemy import Integer, String, Date, Float, Column, Boolean  
from sqlalchemy import create\_engine  
from sqlalchemy import text  
from sqlalchemy.ext.declarative import declarative\_base  
from sqlalchemy.orm import sessionmaker  
from sqlalchemy.orm.query import Query  
from sqlalchemy.orm.session import Session

Using SQL Alchemy, I create a Class object which mirrors the database table structure.

class SQLIndeedSearchResults(Base):  
 \_\_tablename\_\_ = 'indeed\_search\_results'  
 isr\_pk = Column(Integer, primary\_key=True)  
 company = Column(String)  
 extracted\_url = Column(String)  
 guid = Column(String)  
 job\_title\_row = Column(String)  
 latitude = Column(Float)  
 longitude = Column(Float)  
 publish\_date = Column(Date)  
 job\_text\_raw = Column(String)  
 job\_title = Column(String)

This allows the scraped information to be piped to the database through a .add method of an SQL Alchemy session.

def move\_single\_job\_result\_from\_indeed\_to\_database(in\_single\_job\_result, in\_bs\_session, in\_swr):  
 *"""* ***:param*** *in\_single\_job\_result:* ***:return****:  
 """* def create\_job\_description\_soup(in\_job\_desc\_url: str):  
 *"""  
 This takes in an indeed job descriptino URL and retursn a BS result set of the contents.* ***:param*** *in\_job\_desc\_url:string* ***:return****: a BS result set  
 """* detailed\_job\_description\_page\_tree = in\_bs\_session.get(in\_job\_desc\_url,  
 headers=create\_headers\_for\_the\_browser())  
 return BeautifulSoup(detailed\_job\_description\_page\_tree.content, 'lxml')  
  
 # Create an instance and build the data in the ISJR Class with the inbuilt method called populate  
  
 this\_is\_jr\_result = IndeedJobSearchResult()  
 this\_is\_jr\_result.populate(in\_single\_job\_result)  
 # Take the job URL and scrape it for the text.  
 detailed\_job\_description\_page\_soup = create\_job\_description\_soup(this\_is\_jr\_result.extracted\_url, )  
 # Create an instance of the Indeed Job descripiton Analysis info.  
 this\_indeed\_job\_description\_analysis = IndeedJobDescriptionAnalysis()  
 this\_indeed\_job\_description\_analysis.populate(detailed\_job\_description\_page\_soup)  
 # creaet an incidence of the SQL Alchemy based class SQLIndeedSearchResults  
 this\_sql\_upload = SQLIndeedSearchResults()  
 this\_sql\_upload.populate(this\_is\_jr\_result, this\_indeed\_job\_description\_analysis)  
 in\_swr.add(this\_sql\_upload)

## NLTK

Natural Language Toolkit

from nltk import PorterStemmer  
from nltk import ngrams  
from nltk import word\_tokenize

I use three functions from NLTK.

First is Porter Stemmer. This allows words like programs, programs and programming to be counted as one item, giving more clarity to the important words. These are also used on the ATS end so its important to be aware of stems.

The ngrams function allows a set of multiple words to be quickly generated and counted. Example. For an ngrams of 2, the previous sentence would have a set of (the ngrams, ngrams function, function allows, allows a, etc.) To do this through loops would be cumbersome and slow.

for phrase\_length in range(2, 5):  
 job\_text\_multiples = ngrams(job\_text\_words, phrase\_length)  
 for multiple in job\_text\_multiples:  
 # turn the multiple from a tuple into a phrase.  
 additional\_phrase = ' '.join(multiple)  
 job\_text\_multiples\_list.append(additional\_phrase)

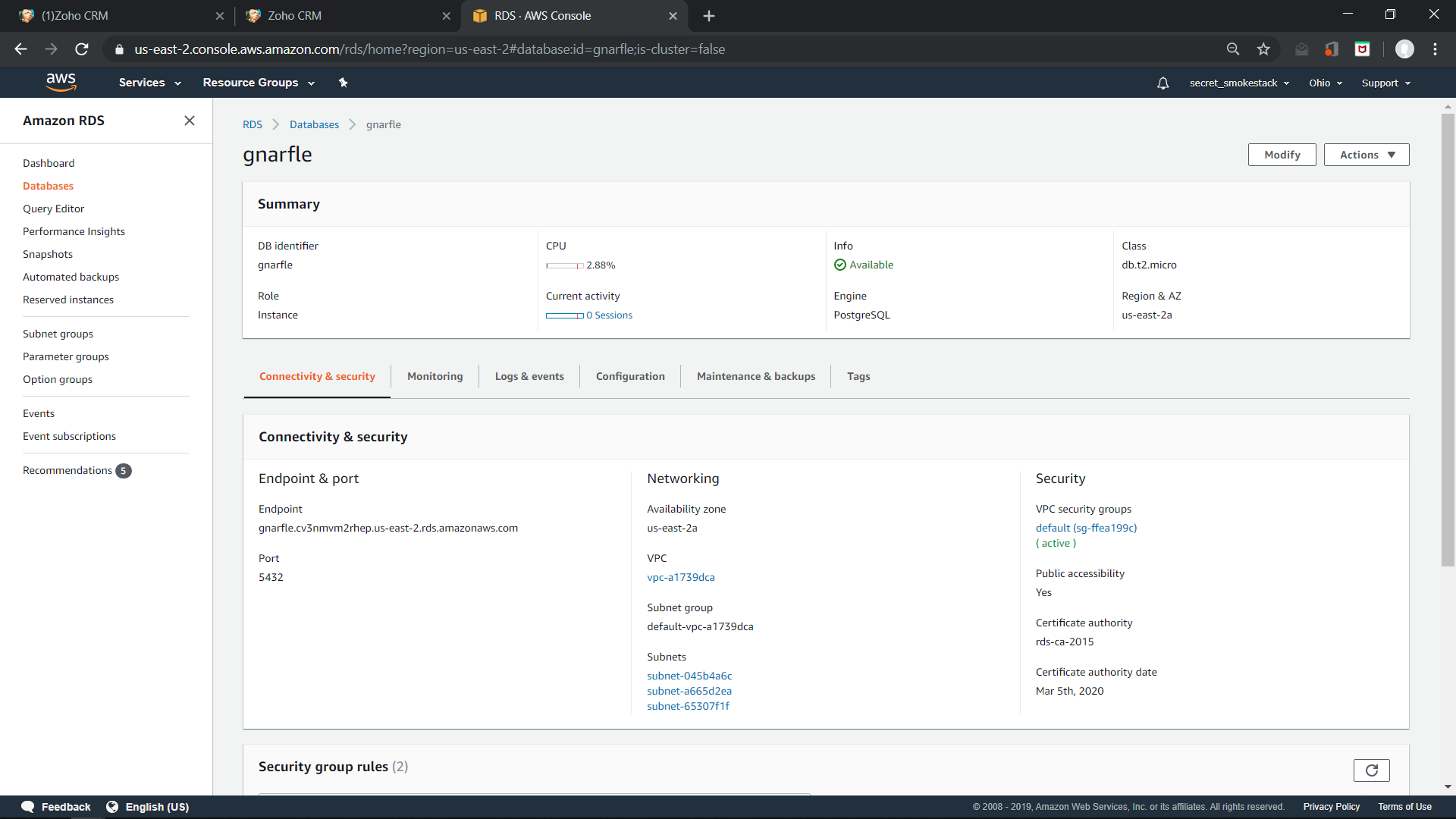
Lastly, word tokenize splits up blocks of text into individual words. Again, this is a more elegant solution than just splitting on spaces.

def word\_frequency\_for\_a\_single\_description(in\_job\_text: str):  
 *"""  
 The purpose of this function is to take in a raw string of job text scraped from the web,  
 and get a frequency word count. I chose to return a counter object because they can be summed together.* ***:param*** *in\_job\_text: which is a string of job text scraped from a job description* ***:return****: a Counter Object with teh word counts  
 """* our\_stemmer = PorterStemmer()  
 # remove all sorts of punctuation to optimize word parsing.  
 job\_text\_string = super\_clean\_a\_string(in\_job\_text)  
 # tokenize for counting  
 job\_text\_words = word\_tokenize(job\_text\_string)  
 # Turn everything to lower case. Using Casefold as it handels UTF-8 better.  
 job\_text\_words = [word.casefold() for word in job\_text\_words]

## Casefold

The method .casefold is superior to .lower because it handles UTF-8 more elegantly.

## Storage

In order for Gnarfle to someday be scalable, I created a PostgreSQL Database on AWS. Through PSYCOPG2, Python is able to easily work with Postgres. 

Some of the underlying queries are here under

<https://github.com/nfeliccia/gnarfle/tree/master/SQL>