Chris Gregg and Wil Kautz
CS 106A

Handout #15
August 4, 2020

Assignment #7: Bajillion Search Engine

Due: 10:30am (Pacific Daylight Time) on Tuesday, August 11th

This assignment will give you an opportunity to use many of your Python skills to build an application that you probably now use on a daily basis – a search engine! You can download the starter code for this project under the “Assignments” tab on the CS106A website. The starter project will provide Python files for you to write your programs in.

As usual, the assignment is broken up into two parts. The first part of the assignment is a short problem to give you practice writing a function involving lists that you then might leverage in the second part of the assignment. In the second part of the assignment you'll be building a search engine from the ground up. It will give you a chance to see how concepts from CS106A are directly applicable to build some of the powerful applications we now use on a daily basis. And, we'll give you a little relevant history there as well.

Sandcastle: Finding common elements in two lists

This problem will give you practice with yet another aspect of using lists. Besides providing a little programming warm-up before you dive right into the main part of the assignment, the function you write here may actually be something that you use in some form in the second part of this assignment as well. You should write your code for this problem in the file **`common_elements.py`**.

There are many situations in which we may be interested in identifying the common elements of two lists. For example, say we have a list of all the students who have completed all their Ways course requirements at Stanford and another list of all the students who have completed the requirements for their major(s) at Stanford. By taking the elements (students) who appear in both lists, we could produce a list of students who have met all their course requirements at Stanford. If were then to take that resulting list (of students who have met all their course requirements) and a list of all undergrads at Stanford who completed at least 180 units and took the common elements from those two lists, we could produce a new list of all the students who were eligible to graduate. Thus, producing a list of the common elements from two other lists not only has practical applications, but can be applied repeatedly to produce even more specific results.

You task is to write the following function:

```
def common(list1, list2)
```

The function takes in two lists and should return a new list which contains only those elements which appear in both **`list1`** and **`list2`**. The original two lists passed into the function should not be changed.

For example, if your function were called as follows:

```
common(['a', 'b', 'c'], ['c', 'a', 'z'])
```

it should return the list:

```
['a', 'c']
```

If your function is called with two lists that have no overlapping elements, such as:

```
common(['a', 'b', 'c'], ['x', 'y', 'z'])
```

it should return the empty list:

```
[]
```

Also, if your function is called with two lists that have multiple of the same shared element, such as:

```
common(['a', 'a', 'b'], ['x', 'a', 'a'])
```

it should return a result that includes only one of the elements that appeared multiple times, as shown in the result below:

```
['a']
```

Doctests are provided for you to test your function. Feel free to write additional doctests. Also, feel free to write any additional functions that may help you solve this problem. A **`main`** function is also provided, which calls your function with some sample test cases and prints the results.

The Bajillion Search Engine

In 1998, two Stanford CS graduate students, Larry Page and Sergey Brin, founded Google Inc. to provide a new way of doing search on the Internet based on research they had conducted as part of the Stanford Digital Libraries project. If you're interested, you can find their original paper describing the initial version of the Google search engine here:

http://infolab.stanford.edu/~backrub/google.html

As their paper explains, the original URL (Uniform Resource Locator, or web address) for Google was in fact hosted at Stanford: http://google.stanford.edu/. Ironically enough, that URL now redirects to a web site describing the G Suite software tools that Stanford licenses from Google, Inc.

What you might not know is that sigificant portions of the original version of the Google were written in Python. You can find more details about that in the paper referenced above. In fact, it turns out that writing a simple search engine in Python is entirely doable given the knowledge that you now have in CS106A! To prove that point, in this assignment you'll will be implementing your own text-based search engine. It's like Google, only smaller. Much, much, smaller.

The name Google comes from the term "googol", which refers to the number formed by a 1 followed by 100 zeroes (or, equivalently, 10 to the 100th power). The name of the company Google actually comes from an unintentional *misspelling* of the term googol, as Larry Page wanted the name to refer to a really large number. Since the name "google" is already taken, we figured you could name your search engine after another notion of a really large number, which is why your search engine is called "Bajillion." As in, "there are a *bajillion* things I've learning in CS106A this quarter!"

There are two main parts of the search engine that you'll be implementing. The first, which is the heart of the search engine, is called the "index," which is described in Part A below and the second is the actual search functionality, which is described in Part B.

Part A: The Index
Most search engines are built on top of what is called an "inverted index," or just "index" for short. An index for a search engine is similar to the index of words in the back of book. Basically, it tells you the page (or, in the more general case, the document or file) where each word appears.

In search, we use the notion of a "term" to refer to generalized notion of a word, name, number, etc. that we might want to look up in a collection of documents. If it's easier, you can just think of a "term" as a "word", but really it could refer to something like the number "127" which is not *officially* a "word".

In any case, an index is structure where, for each term, we have a list of the documents that this term appears in. Consider the following expository examples of terms and documents they appear in:

- The term "burrito" appears in the text documents named "recipes.txt", "greatest eats.txt", "top 10 foods.txt", and "favorites.txt"
- The term "sushi" appears in the text documents "favorites.txt" and "Japanese foods.txt"
- The term "samosa" appears in the text document "appetizers.txt"

We could represent an index of these terms as something like:

"burrito"	→ "recipes.txt", "greatest eats.txt", "top 10 foods.txt", and "favorites.txt"
"sushi"	→ "favorites.txt" and "Japanese foods.txt"
"samosa"	→ "appetizers.txt"

If we wanted to represent such a mapping from terms to a list of documents (file names) in Python, a dictionary would be a natural structure to use. Thus, the index could be represented as the following Python dictionary:

```
index = {
         'burrito': ['recipes.txt', 'greatest eats.txt',
                     'top 10 foods.txt', 'favorites.txt'],
         'sushi': ['favorites.txt', 'Japanese foods.txt'],
         'samosa': ['appetizers.txt']
        }
```

Note that in the dictionary above, the terms (strings) are the keys, and the values are lists of the names of files containing those terms (the file names are strings).

Building an index

Your task in this part of the assignment is to write the code the builds an index for a set of text files. More specifically, you will be implementing a function:

```
create_index(filenames, index, file_titles)
```

This function is passed the following information:

- **`filenames`**: this is a list of file names (strings) that you'll use in building an index.
- **`index`**: this is a dictionary representing the index that you will need to build up. When your function is called, it will be passed an empty dictionary (**`{}`**) for **`index`**. Since dictionaries are mutable types, any changes your function makes to the parameter **`index`** will persist after your function completes.
- **`file_titles`**: this is a dictionary where the keys are file names (strings) and the values are the titles of the articles in each file (which are also strings). We'll explain the details of this parameter later in this handout. When your function is called, it will be passed an empty dictionary (**`{}`**) for **`file_titles`** and your function will add entries to this dictionary as appropriate.

You will build the index based on the set of files specified in the parameter **`filenames`**. For **each file** in the **`filenames`** list, you should parse out all the *terms* in the file in order to add appropriate entries to the **`index`** that you are building.

Terms are defined as follows:

- Terms are seperated from each other in text by spaces or newline (return) characters.
- Terms should have all their letters converted to lowercase.
- Terms should have all punctuation symbols stripped off from their beginning and end. (Punctuation characters in the middle of a term are fine and should not be removed).

To help implement the third bullet point above, the Python **`string`** library (which is already imported in the starter code we give you with the line **`import string`**) provides a constant called **`string.punctuation`**, which is a string containing all the punctuation marks in Python. You can use this constant in conjunction with the **`strip`** function on strings to remove punctuation marks from just the beginning and ending of a string, as shown in the example from the Python console below:

```
>>> raw = '$$j.lo!'
>>> term = raw.strip(string.punctuation)
>>> term
'j.lo'
```

Recall that the **`strip`** function only removes characters from the beginning and ending of a string. **`strip`** will not remove punctuation marks in the middle of a string. Thus, in the example above, the period in the middle of **`'j.lo'`** remains after the call to **`strip`**.

As an additional point about terms, any string of characters that *only* contains punctuation marks is **not** considered a term and should not be added to the index. In such a case, if we stripped all the punctuation from the string, we would be left with the empty string (`""`), and empty strings should **not** be included in the index.

To make this all more concrete, say that we want to index the text file **`'doc1.txt'`** shown below:

'doc1.txt':

```
*We* are 100,000
STRONG!  $$
```

In parsing this file, we should produce the following terms:

- The string **`'*We*'`** should be converted to term **`'we'`**
- The string **`'are'`** should be converted to term **`'are'`**
- The string **`'100,000'`** should be converted to term **`'100,000'`**
  (Note that the comma inside **`100,000`** is fine and should not be removed.)
- The string **'STRONG!'** should be converted to term **`'strong'`**
- The string **`'$$'`** should be ignored (i.e., not included in the index), as a string of only punctuation is not considered a term.

As you produce terms from a file, you should add appropriate entries into the **`index`** for each of the terms. For example, if we started with an **`index`** that was the empty dictionary, the index should look as follows after processing the file **`'doc1.txt'`**:

`index`:

```
    {
     'we': ['doc1.txt'],
     'are': ['doc1.txt'],
     '100,000': ['doc1.txt'],
     'strong': ['doc1.txt']
    }
```

Say, we now process another file, **`'doc2.txt'`**, shown below:

'doc2.txt':

```
Strong, you are!
--Yoda--
```

We should appropriately update the index we had before with the terms found in **`'doc2.txt'`**, we should result in the **`index`** shown below.

`index`:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

Note that in cases where a term found in `'doc2.txt'` previously existed in the `index` (such as `'are'` and `'strong'`), the list of documents that contain that term was expanded to include the file `'doc2.txt'`. In cases where a term found in `'doc2.txt'` was not previously in the index (such as `'you'` and `'yoda'`), a new entry is added to the `index` to indicate that the given term appeared in `'doc2.txt'`.

Building the dictionary `file_titles`

In real-world search engines, we often want to store additional information about each file while we are processing it in order to use this information later when we want to display search results. For example, for web pages, we might store the title of the page. For news articles, we might store the headline of the article. In the news article data we provide in this assignment, the **first line of each file is a title** for the article in that file, which we want to store for future reference. Here is an example of the first portion of two of the actual files (named `'001.txt'` and `'002.txt'`, respectively) in the BBC News article data that we provide for you in the assignment:

'001.txt':

```
Broadband steams ahead in the US

More and more Americans are
joining the internet's fast lane,
according to official figures.
…
```

'002.txt':

```
EA to take on film and TV giants

Video game giant Electronic Arts
(EA) says it wants to become the
biggest entertainment firm in the
world.
…
```

When we are process each file to build up the `index`, we also want to add an entry to the dictionary `file_titles` to keep track of the title for each file. Recall, that the title is simply the first line of the file (with the "newline" character at the end of the line removed). We store this information in `file_titles`, where the file name (string) is the key and the title (string) is the value for an entry in the dictionary.

So, if we started with `file_titles` as an empty dictionary (which is the case when your `create_index` function is called), after we process the files `'001.txt'` and `'002.txt'`, the `file_titles` dictionary should look as follows:

`file_titles`:

```
      {
       '001.txt': 'Broadband steams ahead in the US',
       '002.txt': 'EA to take on film and TV giants'
      }
```

Important note: the terms in the title line for each file should still be included in the index (along with the rest of the document). A common bug is to forget to add the terms from the title line to the index.

Doctests for the **`create_index`** function are provided for you to test your function. Feel free to write additional doctests. Also, you should definitely write additional functions that may help you decompose your solution to this problem.

Running your `create_index` function

You can run your **`create_index`** function by running the **`searchengine.py`** program and specifying the directory name for a set of text files that you would like to index. We provide two such data sets for you to test your code. The first is in a directory named **`small`**, which includes three very short text files, making the size of the resulting index manageable to inspect manually. You can run your program with the **`small`** dataset in a Python Terminal using this command (on a Mac, use **`python3`** instead of **`py`**):

```
> py searchengine.py small
```

In the **`searchengine.py`** program, we provide a **`main`** function that calls your **`create_index`** function and prints the resulting **`index`** and **`file_titles`** on the terminal. The output printed by the program on the **`small`** dataset, should look as shown below. (Note that the output below is produced on a PC, where file paths use the backslash character, represented in Python as a double backslash: **`\\`**. On a Mac, you would see forward a slash character **`/`** in the file paths instead of a double backslash).

```
Index:
{'file1': ['small\\1.txt'], 'title': ['small\\1.txt', 'small\\2.txt',
'small\\3.txt'], 'apple': ['small\\1.txt', 'small\\2.txt', 'small\\3.txt'],
'ball': ['small\\1.txt', 'small\\3.txt'], 'dog': ['small\\1.txt',
'small\\2.txt'], 'elephant': ['small\\1.txt'], 'frog': ['small\\1.txt'],
'file2': ['small\\2.txt'], 'carrot': ['small\\2.txt', 'small\\3.txt'],
'file3': ['small\\3.txt'], 'gerbil': ['small\\3.txt'], 'hamster':
['small\\3.txt'], 'iguana': ['small\\3.txt'], 'lizard': ['small\\3.txt']}
File names -> document titles:
{'small\\1.txt': '** File1 title **', 'small\\2.txt': '** File2 title **',
'small\\3.txt': '** File3 title **'}
```

When you think you've gotten your program working well on the **`small`** dataset, you can then try it out on the BBC News article dataset[1] with the following command (on the Mac, use **`python3`** instead of **`py`**):

```
> py searchengine.py bbcnews
```

The output produced in that case is too large to verify manually, but running on such a large dataset is a good way to see if you program crashes on any cases that might have been missed with the **`small`** dataset.

Part B: Using the index to search

Once you feel as though you are indexing documents correctly, you're ready to implement the actual search functionality. Luckily for you, many people find implementing this part of the program much easier than building an index. And, as an added bonus, you'll likely be able to leverage the **`common`** function you wrote in the first part of this assignment. Yeah, sometimes we really do plan for things to work out like that.

Your task in this part of the assignment is to write the code that implements the search functionality making use of the index you built in Part A. Specifically, you will be implementing a function:

`search(index, query)`

This function is passed the following information:

- **`index`**: this is the **`index`** produced by your **`create_index`** function
- **`query`**: this is a string representing the user's query. All the letters in this string are guaranteed to be lowercase (the starter code we provide uses the **`lower`** function to create a lowercase query string that is passed to this function). To simplify things, you can assume that the user will never enter punctuation characters as part of the query string. The code we provide does **not** actually strip out puctuation characters from the user's query (in case you wanted to do some sort of extension where such punctuation characters might be meaningful in terms of how the search is conducted). But, you can just assume for the basic version of the program that the user will not enter any punctuation characters in their query.

Your search function should return a list of the names of the files that contain **all** of the terms in the given **`query`**. As we discussed in class, you can determine which files contain all the query terms using the index. Recall that in the index, the value associated with each term (key) is a list of the files that contain the given term. This list of files is called the "posting list" for the term. In order to determine which files contain all of the terms in a query, you start with the posting list for the first term in the query. You then consecutively

[1] The BBC News dataset is a collection of articles covering technology news. We reference the paper which was responsible for making this dataset available to the research community: D. Greene and P. Cunningham. "Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering", Proc. ICML 2006. Please note that all rights, including copyright, in the content of the original articles are owned by the BBC. The articles are used here only for educational purposes.

consider the overlap (i.e., the common elements) of the posting list you have with the posting list associated with each subsequent term in the query. When you've processed all the terms in the query in this way, the posting list you have left should contain only those files that contain every term in the query.

To make things more concrete, if you were to build an index on the **`small`** dataset, your **`search`** function should return the respective results shown in the seven examples below. (Again, note that the examples below are produced on a PC, where file paths use the double backslash: **`\\`**. On a Mac, you would see forward a slash character **`/`** in the file paths instead of a double backslash).

Example 1. Calling: **`search(index, 'apple')`**
Should produce the list: **`['small\\1.txt', 'small\\2.txt', 'small\\3.txt']`**

Example 2. Calling: **`search(index, 'ball')`**
Should produce the list: **`['small\\1.txt', 'small\\3.txt']`**

Example 3. Calling: **`search(index, 'lizard')`**
Should produce the list: **`['small\\3.txt']`**

Example 4. Calling: **`search(index, 'apple ball')`**
Should produce the list: **`['small\\1.txt', 'small\\3.txt']`**

Example 5. Calling: **`search(index, 'dog ball')`**
Should produce the list: **`['small\\1.txt']`**

Example 6. Calling: **`search(index, 'dog ball hamster')`**
Should produce the list: **`[]`**

Example 7. Calling: **`search(index, 'nope')`**
Should produce the list: **`[]`**

Doctests for the **`search`** function are provided for you. Feel free to write additional doctests. Also, you should definitely write additional functions that may help you decompose your solution to this problem.

Running your `search` function
You can run your **`search`** function by running the **`searchengine.py`** program, specifying the directory name for a set of text files that you would like to index, and then add **`-s`** at the end of the command line to indicate "search mode". For example, you can run your program in "search mode" with the **`small`** dataset in a Python Terminal using this command (on the Mac, use **`python3`** instead of **`py`**):

```
> py searchengine.py small -s
```

In search mode, the program will first build an index (and a `file_title` dictionary) on the files in the directory you specify by calling your `create_index` function. It will then repeatedly ask the user (you) for a query, which is sent (along with the index) to your `search` function to produce a posting list of results. These results are then displayed on the terminal, where, for each result, the title of the article and the associated file name are listed (making use of the `file_title` dictionary produced by your `create_index` function). You can end the program by simply pressing Enter when asked for a query (i.e., giving the empy query).

Test your program thoroughly with the `small` dataset. That dataset is small enough that you can check your results manually to make sure that you are producing the expected output for various queries. You can also free free to create your own datasets (directories containing text files) to test out any particular cases you want to create to debug your code.

When you've think you've gotten your program working with the `small` dataset, try running your program with the BBC News data as follows:

```
> py searchengine.py bbcnews -s
```

Here is the output of a working program running on some sample queries on the BBC News data (user input is in ***bold italics***). You can see if your program produces the same output. (Note that the order of the articles printed might be different depending on how you wrote your code, but the set of articles that match each query should be the same as in this sample run.)

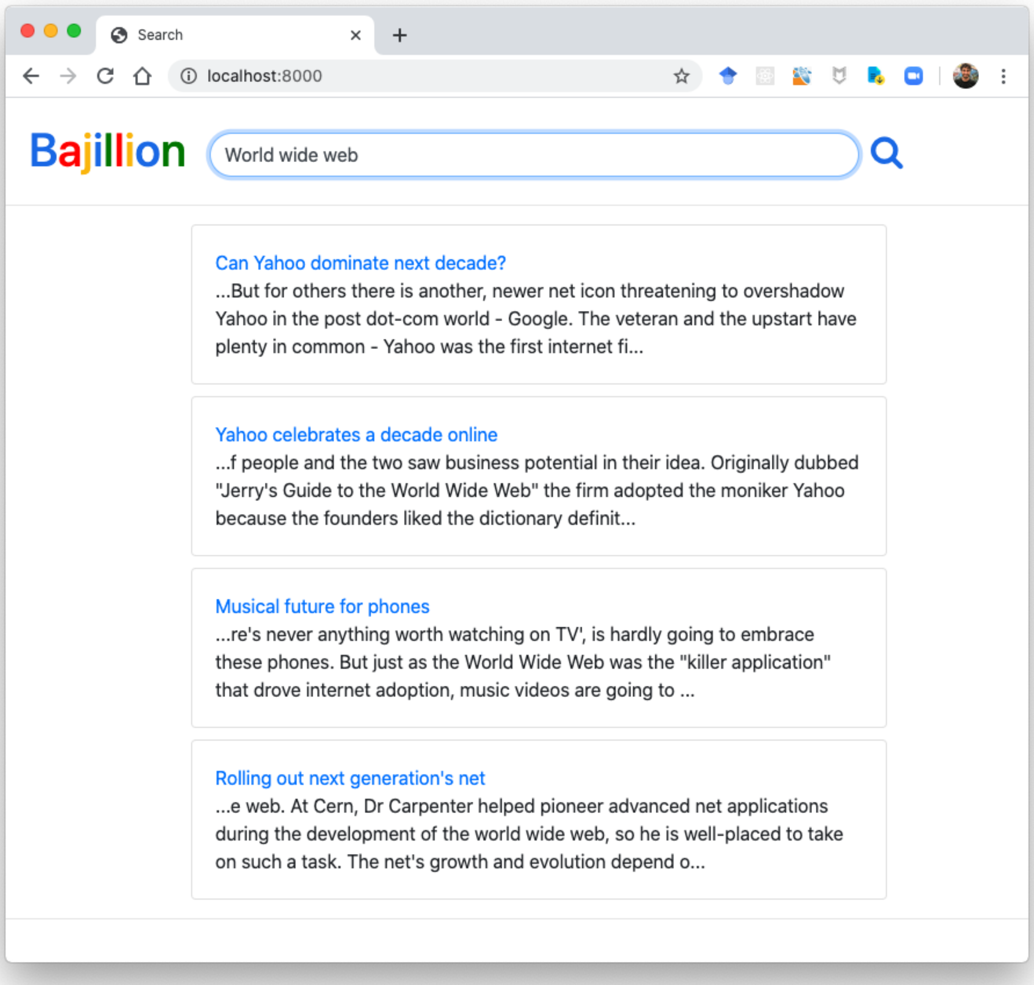
```
Query (empty query to stop): stanford
Results for query 'stanford':
1.  Title: Yahoo celebrates a decade online,  File: bbcnews\066.txt
2.  Title: Google to scan famous libraries,  File: bbcnews\217.txt
Query (empty query to stop): bike
Results for query 'bike':
1.  Title: Games help you 'learn and play',  File: bbcnews\291.txt
2.  Title: The Force is strong in Battlefront,  File: bbcnews\339.txt
Query (empty query to stop): stanford bike
Results for query 'stanford bike':
No results match that query.
Query (empty query to stop): windows virus security patch
Results for query 'windows virus security patch':
1.  Title: Microsoft releases patches,  File: bbcnews\003.txt
2.  Title: Microsoft releases bumper patches,  File: bbcnews\162.txt
Query (empty query to stop): cheap apple products
Results for query 'cheap apple products':
1.  Title: Apple Mac mini gets warm welcome,  File: bbcnews\033.txt
Query (empty query to stop): <user presses "enter" to end program>
```

Congratulations! You just built a working search engine! That's pretty impressive given that you might not have known a lot about programming just 10 weeks ago. It's also a testament to how powerful a tool programming can be. A little knowledge can go a long way in building some really useful software. Happy searching! The rest of the assignment is optional – though we think it's a great deal of fun + learning.

[Optional] Part C: Take your search engine online!

As a celebration of what you have built, and as an exciting extension, let's turn your python program into a server which can deliver those search results to a browser. Woot! This extension will give you exposure to some really interesting next steps beyond CS106A.

Write code in **`extension_server.py`** that responds to search requests! In order to do so, link up your server to call the functions you defined in **`searchengine.py`**.



Can Yahoo dominate next decade?
...But for others there is another, newer net icon threatening to overshadow Yahoo in the post dot-com world - Google. The veteran and the upstart have plenty in common - Yahoo was the first internet fi...

Yahoo celebrates a decade online
...f people and the two saw business potential in their idea. Originally dubbed "Jerry's Guide to the World Wide Web" the firm adopted the moniker Yahoo because the founders liked the dictionary definit...

Musical future for phones
...re's never anything worth watching on TV', is hardly going to embrace these phones. But just as the World Wide Web was the "killer application" that drove internet adoption, music videos are going to ...

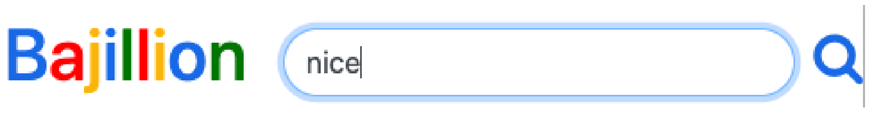
Rolling out next generation's net
...e web. At Cern, Dr Carpenter helped pioneer advanced net applications during the development of the world wide web, so he is well-placed to take on such a task. The net's growth and evolution depend o...

If you run the starter code for **`extension_server.py`** it will create a server which can serve up the shell search page, but is unable to do any search. When a request comes to the server without the empty string as a "command" the server returns the HTML of the search page. HTML is the “markup language” of the world wide web – it’s the description of the page with the search bar etc. To get started run the server:

```
> py extension_server.py
```

And navigate to http://localhost:8000 in a web-browser (such as Chrome, Safari or IE). How exciting. Unfortunately, it is little more than a pretty poster. When you try to search it doesn’t provide any results.

If you look at the terminal where your server is running you will notice that every time someone types something into the search bar and hits Enter, you get a request. For example, say someone types in “nice” (as shown below):

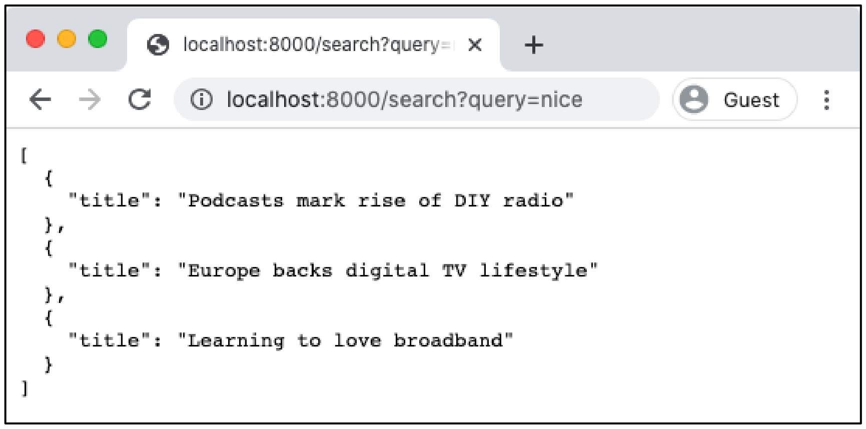


The client sends a request, which is received by your server, with the command “search” and parameters that include the search query. The server currently prints the request out to the console (as shown below), but doesn’t handle it.

```
{'command': 'search', 'params': {'query': 'nice'}}
```

You can directly recreate this request by going to
http://localhost:8000/search?query=nice

Currently your server responds with the empty string. To complete this milestone, change the code to return a list of dictionaries, one dictionary per search result, with the key “title” and the value being the text of the title to display.


localhost:8000/search?query=
localhost:8000/search?query=nice
Guest

```
[
  {
    "title": "Podcasts mark rise of DIY radio"
  },
  {
    "title": "Europe backs digital TV lifestyle"
  },
  {
    "title": "Learning to love broadband"
  }
]
```

That is all you need to do! Once you do so, your search engine at http://localhost:8000 will start working as expected. How fun! It’s a hard task as it requires writing code in a class which is a brand new experience.

Here are a few hints:

1. Check out the HitCounter example from class on May 29th.

2. Recall that request has two properties, commands and parameters. You can extract the command with **`request.get_command()`** and the parameter dictionary using **`request.get_params()`**

3. Your **`handle_request`** function has to return the list of dictionaries as a *string*. You can use **`json.dumps(`*collection*`, indent=2)`** to turn any collection (e.g. a variable that is a list or dictionary) into string format.

4. **`extension_server.py`** already imports the following functions from **`searchengine.py`**: **`create_index`**, **`search`** and **`textfiles_in_dir`**. You should call these functions! When do you want to create the index? When do you want to search? You don't need to do anything special to call them.

5. If you want your search engine to display more than just the title of an article, try adding "url" and "snippet" to the dictionary representing one search result. It will lead to a more exciting search result. These are extra extensions.

It is certainly not necessary, but we invite you to read the contents of **`extension_client.html`** . This file describes the webpage you see (with the search bar). It is commented so that a curious student could learn more about HTML, JavaScript and CSS. Those aren't things we teach in CS106A, but they are super cool!

A note on localhost: Localhost is a special URL which means "this computer". People on different computers won't be able to access your python script over the internet using this URL. To get a URL which other people can use, you can use a service like the one here: https://ngrok.com/

[Optional] Extra/Extension Features

There are *many* other possibilities for optional extra features that you can add if you like, potentially for extra credit. If you are going to do this, please *submit two versions of your program*: one that meets all the assignment requirements, and a second extended version. As usual, in the Assignment 7 project folder, we have provided a file called **`extension.py`** that you can use if you want to write any extensions that you might want to make based on this assignment. The file doesn't contain any useful code to begin with. So, you only need to submit the **`extension.py`** file if you've written some sort of extension in that file that you'd like us to see.

At the top of the files for an extended version (if you submit one), in your comment header, you should **comment** what extra features you completed.

Here are a few extra extension ideas:

- *Ranking function*. The basic search program just lists all the documents that match the query in no particular order. It would be much more useful, especially with large collections of documents, if the documents were ranked (sorted) by their relevance to the user's query. We talked about some ways to do this in class, but there are many different approaches to this. Try implementing a ranking function that you think does a good job of ordering the search results. As we talked about, that might involve augmenting your index to store more information than just which terms appeared in which documents, so you can use that information to determine how to rank the search results.
- *Stop word elimination.* The English language has many words that appear frequently in text, but don't have much value as far as content is concerned. These are words such as "the", "and", "but", "a", etc. Such words are called *stop words*, and it would make your index smaller if you removed the stop words from the index. (That also means you'd potentially want to remove stop words from the user's query before doing a search.) Of course, you can find out more about stop words (as well as find potential lists of them) by searching the web for "stop words".
- *Word stemming.* In your current index, if a user searches for "section" they won't match files that contain the word "sections" (but don't include the singular form of the word "section"), even though files that talk about "sections" might be relevant to the user. Stemming is the process of reducing words to their base form, so that (for example) both "section" and "sections" would become, simply, "section". Word stemming is a common feature in commercial search engines as it's very useful for helping people get relevant results. As a start, you can find out more about word stemming from Wikipedia: https://en.wikipedia.org/wiki/Stemming

Submitting your work

Once you've gotten all the parts of this assignment working, you're ready to submit! Make sure to submit **only** the python files you modified for this assignment on Paperless. You should make sure to (at least) submit the files:

```
common_elements.py
searchengine.py
```