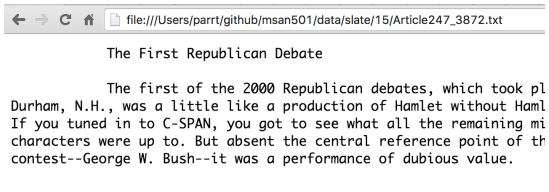
Search Engine Implementation

The goal of this project is to learn how hashtables work and to *feel* just how much slower a linear search is. Along the way, you'll learn the basic mechanics of implementing a search engine, including displaying search results in a browser window and being able to navigate to documents. You'll also learn a tiny bit of HTML.

Discussion

A **search engine** accepts one or more **terms** and searches a corpus for files matching all of those terms. A **corpus** is just a directory and possibly subdirectories full of text files. If you go to the American National corpus, you'll see lots of fun text data. I have extracted articles from Slate magazine and also from Berlitz travelogues. These are your data sets. Berlitz is smaller and so I use that in some of my unit tests. Here is a fragment of a sample search results page as displayed in Chrome (activated from Python); clicking on a link brings up the actual file.

HTML output	File Content
Search results for ronald reagan in 164 files /Users/parrt/github/msan501/data/slate/15/Article247_3872.txt relationship with Ronald Reagan, whom he served in the White House for eight Hatch also took credit for just about everything significant Ronald Reagan did expansion over the last number of years. It's been primarily because Reagan got /Users/parrt/github/msan501/data/slate/49/ArticleIP_12436.txt his two Republican predecessors, Reagan and Bush, they would have been The only time Ronald Reagan ever talked about Iran-Contra under oath was in a deposition for the /Users/parrt/github/msan501/data/slate/1/Article247_42.txt A Shared Vision pairs Ronald Reagan and Margaret Thatcher. Yes, they /Users/parrt/github/msan501/data/slate/14/Article247_3774.txt of how Ronald Reagan would watch the 7 o'clock network news, but then happily /Users/parrt/github/msan501/data/slate/35/ArticleIP_2376.txt White House see November's election as a replay of Ronald Reagan's 1984	

In repo directory *userid*-hashtable, you're going to implement 3 different search mechanisms using code derived from the starter kit files. The actual search mechanism of your code goes in these three files:

1. Linear search; file linear_search.py
2. Hashtable via built in Python `dict` objects; file index_search.py
3. Hashtable that you implement yourself; file myhtable_search.py

All three mechanism should give exactly the same results, but you will notice that the linear search is extremely slow. On my really fast machine with an SSD, it takes about five seconds to search through the Slate data. It has to open and search about 4500 files. With either of the hash tables, it's a matter of milliseconds.

File search.py is the main program, which you execute like this from the *userid*-hashtable directory:

```
$ python search.py linear ~/data/slate
$ python search.py index ~/data/slate
$ python search.py myhtable ~/data/slate
```

assuming you have placed the `slate` directory under a `data` directory in your home directory.
Please do not add data files to your repository! I don't need them and it takes forever to download your repos if you add the data.

Linear search

Your first task is to perform a brain-dead linear search, which looks at each file in turn to see if it contains all of the search terms. If it does, that filename is included in the set (not list) of matching documents. The complexity is *O(n)*for *n* total words in all files.

Given a list of fully-qualified filenames containing the search terms, the main program in search.py uses function`results()` to get a string containing HTML, which search.py writes to file `/tmp/results.html`. It then requests, via`webbrowser.open_new_tab()`, that your default browser open that page.

HTML output

You can create whatever fancy HTML you want to show search results, but here is the basic form you should follow:

```
<html>
<body>
<h2>Search results for <b>ronald reagan</b> in 164 files</h2>
```

```
    <p><a href="file:///Users/parrt/msan/parrt-
hashtable/data/slate/15/Article247_3872.txt">/Users/parrt/msan/parrt-
hashtable/data/slate/15/Article247_3872.txt</a><br>
    relationship with Ronald Reagan, whom he served in the White House for eight<br>
    Hatch also took credit for just about everything significant Ronald Reagan
did<br>expansion over the last number of years. It's been primarily because Reagan
got<br><br>

    <p><a href="file:///Users/parrt/msan/parrt-
hashtable/data/slate/49/ArticleIP_12436.txt">/Users/parrt/msan/parrt-
hashtable/data/slate/49/ArticleIP_12436.txt</a><br>
    his two Republican predecessors, Reagan and Bush, they would have been<br>
    The only time Ronald<br>Reagan ever talked about Iran-Contra under oath was in a
deposition for the<br><br>

    ...

</body>
</html>
```

Notice that the links are URLs just like you see going to websites except they refer to a file on the local disk instead of another machine because of the `file://` prefix.

```
file:///Users/parrt/msan/parrt-hashtable/data/slate/49/ArticleIP_12436.txt
```

(My data is stored in a slightly different spot than yours will be.)

Creating an index using `dict`

Rather than looking through each file for every search, it's better to create a fast lookup index that maps a word to all of the files that contain that word. To compute the search results for multiple words, find the intersection of documents among the document set (`index[w]`) for each word. The resulting set will be just the documents that have all words.

It takes about the same time to create the index as it does to do one linear search because both are linearly walking through the list of files. The complexity of index creation is *O(n)* for *n* total words in all files. BUT, searching takes just *O(1)*, or constant time, once we have the index.

The main program uses the following sequence for this `dict` version of the search engine:

```
index = create_index(files) # files is a list of fully-qualified filenames
docs = index_search(files, index, terms) # terms is a list of normalized words
```

Once the index is created, function `index_search()` can crank out search results faster than you can take your fingers off the keyboard.

Here are the two key methods you must implement:

```
def create_index(files):
    """
```

```
    Given a list of fully-qualified filenames, build an index from word
    to set of document indexes. The document index is just the index into the
    files parameter (indexed from 0).
    Make sure that you are mapping a word to a set, not a list.
    For each word w in file i, add i to the set of documents containing w
    Return a dict object.
    """
def index_search(files, index, terms):
    """
    Given an index and a list of fully-qualified filenames, return a list of them
    whose file contents has all words in terms as normalized by your words()
function.
    Parameter terms is a list of strings.
    You can only use the index to find matching files; you cannot open the files and
look inside.
    """
```

These functions will use expressions like `index[w]`, where `index` is a `dict`, to access the documents containing word `w`.

Creating an index using your own hashtable

This version of the search engine should look and perform just like the version using `dict`. The difference is **you cannot use the built-in dictionary operations** like `index[k]` for `dict index` and key `k`. You will build your own hashtable and call your own get and put functions explicitly to manipulate the index.
Because we are not studying the object-oriented aspects of Python, we are going to represent a hashtable as a list of lists (list of buckets):

```
def htable(nbuckets):
    """Return a list of nbuckets empty lists"""
```

The number of buckets should be a prime number to avoid hash code collisions.

Each element in a bucket is an association `(key,value)`. For example, `htable_put('parrt', 99)` should add tuple`('parrt',99)` to the bucket associated with key string `parrt`. The following method embodies the put operation:

```
def htable_put(table, key, value):
    """
    Perform table[key] = value
    Find the appropriate bucket indicated by key and then append value to the bucket.
    If the bucket for key already has a key,value pair with that key then replace it.
    Make sure that you are only adding (key,value) associations to the buckets.
    """
```

To make that work, you need a function that computes hash codes:

```
def hashcode(o):
    """
    Return a hashcode for strings and integers; all others return None
```

```
    For integers, just return the integer value.
    For strings, perform operation h = h*31 + ord(c) for all characters in the string
    """
```

Notice that we are only computing hash codes for strings and integers. The hash code for a string could be just the sum of all of all the character ASCII codes, via `ord()`, but that would mean a lot of collisions like `pots` and `stop`. A collision is when different keys hash to the same bucket. Ideally we would have one association per bucket. The "distribution" of elements to buckets is a function of how many buckets we have and how good our hash function is. The multiplication by prime number 31 starts shifting the bits around and gets a bit of "randomness" into our key computation.
The hash code is not directly used to get the bucket index because the hash code will typically be many times larger than the number of buckets. The index of a bucket is the hash code modulo the number of buckets:

```
bucket = hashcode(key) % nbuckets
```

To get a value out of the hash table associated with a particular key, we use this function:

```
def htable_get(table, key):
    """
    Return table[key].
    Find the appropriate bucket indicated by the key and look for the association
    with the key. Return the value (not the key and not the association!)
    Return None if key not found.
    """
```

It computes the bucket where `key` lives and then linearly searches that (hopefully) small bucket for an association with key `key`. It then returns the value, the second element, from that association.

Getting started

Please go to Hashtable starterkit and grab all the python files. Store these in your repo *userid*-hashtable, wherever you store that directory. E.g., I might put mine in `/Users/parrt/msan/parrt-hashtable`.
Store the Slate and Berlitz data sets outside of your repo so that you are not tempted to add that data to the repository. Perhaps you can make a general data directory for use in lots of classes such as `~/data` or just for this class `~/msan501/data`.
I recommend that you start by getting the simple linear search to work, which involves computing HTML and all of the basic machinery for extracting words from file content. So start by fleshing out `words.py` and `linear_search.py`. You can use the unit tests

in `test_search.py`, although the tests will fail for the indexed-based searches until you get those implemented.

Deliverables

You must complete and add these to root of your *userid*-hashtable repository:

- htable.py
- index_search.py
- linsearch.py
- myhtable_search.py (no `dict` allowed in this file)
- search.py
- test_htable.py
- test_search.py
- words.py

Please do not add the data to your repository!

Ultimately, you want the test results to look like the following.

```
$ python -m pytest -v test_search.py
...
test_search.py::test_linear_berlitz_none PASSED
test_search.py::test_index_berlitz_none PASSED
test_search.py::test_myhtable_berlitz_none PASSED
test_search.py::test_linear_berlitz PASSED
test_search.py::test_index_berlitz PASSED
test_search.py::test_myhtable_berlitz PASSED
...
$ python -m pytest -v test_htable.py
...
test_htable.py::test_empty PASSED
test_htable.py::test_single PASSED
test_htable.py::test_a_few PASSED
test_htable.py::test_str_to_set PASSED
...
```

(You might need to install `pytest` with `pip`.)
I will test your project using something like the test file test_search.py but on a new data set you have not seen.

Let me point out that my unit tests are incredibly anemic and are meant only to show the basic mechanism of testing. You are free to extend the tests to include a lot more.