inverse index lab

Coding the Matrix, 2015

`For auto-graded problems, edit the file inverse_index_lab.py to include your solution.`

In this lab, you will create a simple search engine. One procedure will be responsible for reading in a large collection of documents and indexing them to facilitate quick responses to subsequent search queries. Other procedures will use the index to answer the search queries.

The main purpose of this lab is to give you more Python programming practice and introduce modules and control structures.

1 *Using existing modules*

Python comes with an extensive library, consisting of components called *modules*. In order to use the definitions defined in a module, you must either import the module itself or import the specific definitions you want to use from the module. If you import the module, you must refer to a procedure or variable defined therein by using its *qualified name*, i.e. the name of the module followed by a dot followed by the short name.

For example, the library `math` includes many mathematical procedures such as square-root, cosine, and natural logarithm, and mathematical constants such as π and e.

Ungraded Task: Import the `math` module using the command

```
>>> import math
```

Call the built-in procedure `help(modulename)` on the module you have just imported:

```
>>> help(math)
```

This will cause the console to show documentation on the module. You can move forward by typing `f` and backward by typing `b`, and you can quit looking at the documentation by typing `q`.

Use procedures defined by the `math` module to compute the square root of 3, and raise it to the power of 2. The result might not be what you expect. Keep in mind that Python represents nonintegral real numbers with limited precision, so the answers it gives are only approximate.

Next compute the square root of -1, the cosine of π, and the natural logarithm of e.

The short name of the square-root function is `sqrt` so its qualified name is `math.sqrt`. The short names of the cosine and the natural logarithm are `cos` and `log`, and the short names of π and e are `pi` and `e`.

The second way to bring a procedure or variable from a module into your Python environment is to specifically import the item itself from the module, using the syntax

`from` ⟨*module name*⟩ `import` ⟨*short name*⟩

after which you can refer to it using its short name.

Task 1: The module `random` defines a procedure `randint(a,b)` that returns an integer chosen uniformly at random from among $\{a, a+1, \ldots, b\}$. Import this procedure using the command

```
>>> from random import randint
```

Try calling `randint` a few times. Then write a one-line procedure `movie_review(name)` that takes as argument a string naming a movie, and returns a string review selected uniformly at random from among two or more alternatives (Suggestions: "See it!", "A gem!", "Ideological claptrap!")

2 *Creating your own modules*

You can create your own modules simply by entering the text of your procedure definitions and variable assignments in a file whose name consists of the module name you choose, followed by `.py`. Use a text editor such as Kate or Vim or, my personal favorite, Emacs.

The file can itself contain import statements, enabling the code in the file to make use of definitions from other modules.

If the file is in the current working directory when you start up Python, you can import the module.[1]

Ungraded Task: Previously, you wrote procedures `dict2list(dct, keylist)` and `list2dict(L, keylist)`. Download the file `dictutil.py` from `http://resources.codingthematrix.com`. (That site hosts support code and sample data.) Edit the provided file `dictutil.py`, replacing each occurence of `pass` with the appropriate statement. Import this module, and test the procedures. We might have occasion to use this module in the future.

2.1 *Reloading*

You will probably find it useful when debugging your own module to be able to edit it and load the edited version into your current Python session. Python provides the procedure `reload(module)` in the module `imp`. To import this procedure, use the command

```
>>> from imp import reload
```

Note that if you import a specific definition using the `from ... import ...` syntax then you cannot reload it.

Ungraded Task: Edit `dictutil.py`. Define a procedure `listrange2dict(L)` with this spec:

- *input:* a list L
- *output:* a dictionary that, for $i = 0, 1, 2, \ldots, \text{len}(L) - 1$, maps i to $L[i]$

You can write this procedure from scratch or write it in terms of `list2dict(L, keylist)`. Use the statement

```
>>> reload(dictutil)
```

or >>> import imp; imp.reload(dictutil)

to reload your module, and then test `listrange2dict` on the list `['A','B','C']`.

3 *Loops and conditional statements*

Comprehensions are not the only way to loop over elements of a set, list, dictionary, tuple, range, or zip. For the traditionalist programmer, there are *for-loops*: `for x in {1,2,3}: print(x)`. In this statement, the variable `x` is bound to each of the elements of the set in turn, and the statement `print(x)` is executed in the context of that binding.

There are also *while-loops*: `while v[i] == 0: i = i+1`.

[1] There is an environment variable, `PYTHONPATH`, that governs the sequence of directories in which Python searches for modules.

There are also conditional statements (as opposed to conditional expressions):

```
if x > 0: print("positive")
```

4 *Grouping in Python using indentation*

You will sometimes need to define loops or conditional statements in which the body consists of more than one statement. Most programming languages have a way of grouping a series of statements into a block. For example, C and Java use curly braces around the sequence of statements.

Python uses *indentation* to indicate grouping of statements. **All the statements forming a block should be indented the same number of spaces.** Python is very picky about this. Python files we provide will use **four** spaces to indent. Also, don't mix tabs with spaces in the same block. In fact, I recommend you avoid using tabs for indentation with Python.

Statements at the top level should have no indentation. The group of statements forming the body of a control statement should be indented more than the control statement. Here's an example:

```
for x in [1,2,3]:
  y = x*x
  print(y)
```

This prints 1, 4, and 9. (After the loop is executed, `y` remains bound to 9 and `x` remains bound to 3.)

Ungraded Task: Type the above for-loop into Python. You will see that, after you enter the first line, Python prints an ellipsis (...) to indicate that it is expecting an indented block of statements. Type a space or two before entering the next line. Python will again print the ellipsis. Type a space or two (same number of spaces as before) and enter the next line. Once again Python will print an ellipsis. Press *enter*, and Python should execute the loop.

The same use of indentation can be used used in conditional statements and in procedure definitions.

```
def quadratic(a,b,c):
   discriminant = math.sqrt(b*b - 4*a*c)
   return ((-b + discriminant)/(2*a), (-b - discriminant)/(2*a))
```

You can nest as deeply as you like:

```
def print_greater_quadratic(L):
  for a, b, c in L:
    plus, minus = quadratic(a, b, c)
    if plus > minus:
      print(plus)
    else:
      print(minus)
```

Many text editors help you handle indentation when you write Python code. For example, if you are using Emacs to edit a file with a `.py` suffix, after you type a line ending with a colon and hit return, Emacs will automatically indent the next line the proper amount, making it easy for you to start entering lines belonging to a block. After you enter each line and hit `Return`, Emacs will again indent the next line. However, Emacs doesn't know when you have written the last line of a block; when you need to write the first line outside of that block, you should hit `Delete` to unindent.

5 *Breaking out of a loop*

As in many other programming languages, when Python executes the `break` statement, the loop execution is terminated, and execution continues immediately after the innermost nested loop containing the statement.

```
>>> s = "There is no spoon."
>>> for i in range(len(s)):
...   if s[i] == 'n':
...     break
...
>>> i
9
```

6 `enumerate`

Often one wants to iterate through the elements of a list while keeping track of the indices of the elements. Python provides `enumerate(L)` for this purpose.

```
>>> list(enumerate(['A','B','C']))
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> [i*x for (i,x) in enumerate([10,20,30,40,50])]
[0, 20, 60, 120, 200]
>>> [i*s for (i,s) in enumerate(['A','B','C','D','E'])]
['', 'B', 'CC', 'DDD', 'EEEE']
>>> for (i,s) in enumerate(['A','B','C']):
...  print(i,s)
...
0 A
1 B
2 C
>>>
```

7 *Reading from a file*

At the *Coding the Matrix* resource site, we have provided text files `stories_small.txt` and `stories_big.txt`.

In Python, a *file* object is used to refer to and access a file. The expression `open('stories_small.txt')` returns a file object that allows access to the file with the name given. You can use a comprehension or for-loop to loop over the lines in the file

```
>>> f = open('stories_big.txt')
>>> for line in f:
...  print(line)
```

or, if the file is not too big, use `list(·)` to directly obtain a list of the lines in the file, e.g.

```
>>> f = open('stories_small.txt')
>>> stories = list(f)
>>> len(stories)
50
```

In order to read from the file again, one way is to first create a new file object by calling `open` again.

8 *Mini-search engine*

Now, for the core of the lab, you will be writing a program that acts as a sort of search engine. The goal is to enable the user to find all the documents matching a query. A "document" will be represented by a string. For example,

```
"Look at me ! Look at me ! Look at me NOW ! It is fun to have fun but you have to know how ."
```

comes from *The Cat in the Hat*. The "words" that appear in this document are

```
{'have', 'Look', 'know', 'how', 'to', 'but', 'It', 'me', 'is', 'fun', 'at', '!', '.', 'NOW', 'you'}
```

including a couple of punctuation marks. This document and the others provided have been processed to separate words from punctuation marks such as period, exclamation point, comma, and semicolon.

A *corpus* of document is represented by a list of strings. Here is a very small corpus:

```
["Look at me ! Look at me ! Look at me NOW ! It is fun to have fun but you have to know how ." ,
"He should not be here ,  said the fish in the pot .  He should not be
here when your mother is not .",
"The sun did not shine . It was too wet to play . So we sat in the
house . All that cold, cold, wet day ."]
```

The documents are assigned numbers according to their position in the list: the first is document number 0, the second is document number 1, and the third is document number 2.

If you search the corpus for the word `'in'`, the search engine should report that this word appears in documents 1 and 2 (but not 0). If you search the corpus for the word `'to'`, the search engine should report that this word appears in documents 0 and 2 (but not 1).

A simple data structure called an *inverse index* supports such searches. An inverse index for a corpus maps each word appearing in the corpus to the set of numbers of documents containing the word. We will represent the inverse index in Python by a dictionary. Thus the inverse index for the above corpus is

```
'mother':{1}, 'know':{0}, 'we':{2}, 'how':{0}, 'here':{1},
 'day':{2}, 'house':{2}, 'All':{2}, 'NOW':{0}, 'wet':{2},
 'have':{0}, 'too':{2}, 'be':{1}, 'shine':{2}, 'It':{0, 2},
 'said':{1}, 'but':{0}, 'cold,':{2}, 'was':{2}, 'when':{1},
 'to':{0, 2}, 'should':{1}, 'is':{0, 1}, 'fun':{0},
 'in':{1, 2}, 'He':{1}, 'at':{0}, 'me':{0}, 'sat':{2},
 'you':{0}, 'play':{2}, 'Look':{0}, 'So':{2}, 'your':{1},
 'the':{1, 2}, 'The':{2}, 'fish':{1}, 'that':{2}, '!':{0},
 '.':{0, 1, 2}, ',':{1}, 'not':{1, 2}, 'pot':{1}, 'sun':{2}, 'did':{2}}
```

Each key is a word appearing in the corpus, and the corresponding value is the set of numbers of documents containing that word.

To obtain a list of the words making up a document, you can use the method `.split()` defined for strings:

```
>>> mystr = 'Ask not what you can do for your country .'
>>> mystr.split()
['Ask', 'not', 'what', 'you', 'can', 'do', 'for', 'your', 'country', '.']
```

Task 2: Write a procedure `makeInverseIndex(strlist)` that, given a list of strings (documents), returns a dictionary that maps each word to the set consisting of the document numbers of documents in which that word appears. This dictionary is called an *inverse index*. (Hint: use `enumerate`.)

Your procedure is expected to use a loop or even a nested loop. Don't feel compelled to use comprehensions here.

After testing your procedure on a small corpus, read the lines of `stories_small.txt` into a list of strings, and build an inverse index for that corpus. Then try `stories_big.txt` .

Task 3: Write a procedure `orSearch(inverseIndex, query)` which takes an inverse index and a list of words `query`, and returns the set of document numbers specifying all documents that conain *any* of the words in `query`.

Task 4: Write a procedure `andSearch(inverseIndex, query)` which takes an inverse index and a list of words `query`, and returns the set of document numbers specifying all documents that contain *all* of the words in `query`.

Try out your procedures on these two provided files:

- `stories_small.txt`
- `stories_big.txt`

Warning: For a dictionary `mydict`, unfortunately `enumerate(mydict)` does not behave in the way you might expect or hope. You might hope that it would enumerate (key, value) pairs. Instead, it ignores the values and treats the dictionary as a sequence of keys:

```
>>> for i,x in enumerate({0:'a',1:'b'}):
...  print(i,x)
...
0 0
1 1

>>> for i,x in enumerate({'a':0,'b':1}):
...  print(i,x)
...
0 a
1 b
```