

Homework 4:

Comprehensions, Text as Vectors, Tests

Florian Fink
Symbolische Programmiersprache

Due : Thursday, December 3, 2020, 16:00

In this exercise you will:

- Practice list and dictionary comprehensions in Python
- Review how to represent documents as vectors, and compare similarities
- Get some hands-on experience using the python `doctest` and `unittest` frameworks

Exercise 1: List, Set and Dictionary Comprehensions in Python [2.5 points]

In this exercise you will solve 5 Tasks to practice a powerful feature of Python: comprehensions¹. With these, multiple-line for-loop constructions can be expressed in expressive one-liners.

Solve the following tasks given in `comprehensions.py` using list or dictionary comprehensions. You can test the functionality of your code calling (from your `./src` directory):
`python3 -m unittest -v hw04_text_search/test_comprehensions.py`

1. Complete the function `multiply_by(x, list1)` that multiplies each value in a list by `x` and returns it as a new list. [0.5 points]
2. Complete the function `check_division(x, list1)` that takes a list and returns a list indicating whether or not each element in the original list can be divided by `x` (e.g `check_division(3, [1,2,3]) -> [False, False, True]`) [0.5 points]
3. Complete the function `div_less(set1)`. It should return a new set containing only those numbers from `set1` that can't be divided by any other number from `set1` (except one and itself) [0.5 points]

¹<https://www.geeksforgeeks.org/comprehensions-in-python/>

4. Implement the function `map_zip(list1, list2)`. It should return a dictionary mapping the *n*th element in `list1` to the *n*th element in `list2`. Make use of the `zip()` function in your dictionary comprehension, that can handle lists of different sizes automatically.
5. Complete the function `word_to_length(list1)`. It returns a dictionary mapping all words with at least 3 characters to their number of characters. [0.5 points]

Exercise 2: Search Engine: Running the code

In the source folder for this exercise (`src/hw04_text_search`), you will find the classes to represent documents, and a simple search engine, which were discussed in the lecture (`text_vectors.py`). There is also a script to interactively search all `*.txt` files in a directory (`interactive_search.py`). Try to understand what each of the classes are doing.

On the course homepage, you can find a dataset of corporate emails², containing several folders of spam or normal (ham) emails. Download and unpack it into the `src/data/` folder of your project. Run the interactive search on a email folder (always call scripts from the `src/` folder):

```
python3 -m hw04_text_search.interactive_search --dir data/enron/enron1/ham/
```

Exercise 3: Doctest and documentation

Exercise 3.1: Doctest [2 points]

Use the doctest module to write tests for the functions `dot` and `normalized_tokens` in the module `hw04_text_search.text_vectors`.

- Run your tests with:

```
python3 -m doctest -v hw04_text_search/text_vectors.py
```
- You can also run specific tests on the command line:

```
python3 -m doctest -v hw04_text_search/text_vectors.py testDot
```
- Or you can use PyCharm's green run buttons to run individual tests.

Exercise 3.2: Docstrings [5.5 points]

Provide docstring documentation (in German or English) for all member functions (including constructors) of the classes `TextDocument`, `DocumentCollection` and `SearchEngine` in the same module. In the documentation try to describe what the function does in as few as possible sentences (but no less).

²See https://en.wikipedia.org/wiki/Enron_Corpus for the history of this dataset

Exercise 4: Extending the program using test-driven development [9 points]

Improve the program by adding additional functionality. Use the `unittest` framework, and extend the module `hw04_text_search.test_text_search`. You should add tests that initially fail, and only pass once you successfully added the missing functionality.

Have a look at the example for a test given with:

```
DocumentCollectionTest.test_unknown_word_cosine
```

This test fails, as you can verify by running:

```
python3 -m unittest -v hw04_text_search/test_text_search.py
```

1. Make the existing test pass by changing the functionality of `DocumentCollection.cosine_similarity` accordingly. **[1 point]**
2. Write 4 additional tests that initially fail, and then pass after some functionality of (any part of) the initial code has been changed/extended (try really to follow the test-driven development paradigm in this exercise: 1. Implement the failing tests, 2. Implement the functionality to make the tests pass). **[8 points]**

You can come up with your own improvements to the code, or you can choose from the following list (in each case also write the appropriate test):

- The search engine displays text snippets including line break. Change the functionality such that lines are displayed without line breaks.
- Remove the indentation markers of reply emails (e.g. "> > > > ") (either when reading or when displaying).
- If several search terms occur in a document, the search engine displays several text snippets (one for each). Change the code such that only one text snippet is displayed, if it contains the entire search string.
- Query syntax: if tokens are quoted ("New York"), require that full string occurs in the document (hint: additionally filter result of `docs_with_tokens`)
- Snippets should show exact matches of query tokens, not substring matches.
- Files to index should recursively be read from subdirectories.
- When the file path is shown for search result, normalize it so that the full absolute path is shown.
- If there is no result containing all tokens, search for documents containing at least one of the tokens.
- If a query contains the same token multiple times, only show one text snippet for it.