Handin 4

Forfalder 4. Okt af 18:00 **Point** 100 **Afleverer** et eksternt værktøj
Tilgængelig efter 27. Sep kl. 18:00

Part 1

This week, we will build on the word list exercise from Handin3. The idea behind this exercise is to illustrate that it is important to think carefully about how to represent your data. The speed by which a task is solved can be very dependent on the way the data is organised. The exercise looks very long, but most of the code that you need to write are slight modifications of what you wrote in Handin3.

We are going to create a program that compares the british-english word list that we worked on last week, with an equivalent list for american-english words and calculates how many entries are different between the two files.

The way we will check for similarities between the files is to put the words from the two files into two separate containers, and then run through a loop where you check if each word in one file is present in the other. We will see that there are more and less efficient ways to do this, and that it can matter a lot which choice you make. In this exercise we will try three different representations of the data in the second file, and thereby three different ways of looking up the names: linear search, binary search, and using dictionaries.

In addition to the british-english and american-english data files, we also provide two small test files so you can test whether your methods work before actually running on the large files. These files are called british-english-test and american-english-test. All four files can be downloaded here: **https://wouterboomsma.github.io/ppds2023/data/british-english, (https://wouterboomsma.github.io/ppds2023/data/british-english) https://wouterboomsma.github.io/ppds2023/data/american-english, (https://wouterboomsma.github.io/ppds2023/data/american-english) https://wouterboomsma.github.io/ppds2023/data/british-english-test, (https://wouterboomsma.github.io/ppds2023/data/british-english-test) https://wouterboomsma.github.io/ppds2023/data/american-english-test (https://wouterboomsma.github.io/ppds2023/data/american-english-test)** .

1. Create a file called `handin4.py`. Inside this file, create a function called `wordfile_to_list`, which takes a single argument called `filename`. This function should read the file, and return a list of words (as strings). You can assume that each line in the file only contains a

single word. Please remember to the remove newlines at the end of each line. This function is just a simplified version of the one we wrote in Handin3, where we don't keep track of the line numbers.

Create a file called `handin4_test.py`. Inside this file, call your `wordfile_to_list` function on the `british-english` file and save the result in a variable called `wordlist_british`. Do the same for the american wordlist and save it in a variable called `wordlist_american`.

2. We will now write the first comparison function. Add a function to the `handin4` module called `wordfile_differences_linear_search`, which takes two filenames as arguments, and calls `wordfile_to_list` to create a list for each of these files. The function should contain a loop that for each word in the first list looks through the second list to see if there is a match. It should return a list of words that are in the first file but not in the second file. Matches should be case-sensitive, so e.g. "Gnu" and "gnu" are not considered identical.

   In the file called `handin4_test.py`, call the `wordfile_differences_linear_search` on the input files, using `british-english` as file1 and `american-english` as file2 (it is ok to test it on the test files first, but please switch to the full files before submitting), and saves the result in a variable called `differences_linear_search`. Note that this might take several minutes to calculate. To keep track of exactly how long it takes, Python has built-in functionality for measuring execution times within a programme, using the `timeit` module, which can be used like this:

   ```
   import timeit
   start_time = timeit.default_timer()
   # write code you want to measure execution time for here
   time_spent = timeit.default_timer() - start_time
   ```

   Use this technique to measure how long it takes to call the `wordfile_differences_linear_search` function, and save the result in a variable called `time_spent_linear_search`. Hint: you are of course very welcome to print the `time_spent_linear_search` and `differences_linear_search` out to screen so you can check your results, although this output will not be used by the code-checker (we check the variables directly).

3. It is much more efficient to find elements in a sorted list. One way of doing this is by using a method called binary search. Basically, binary search excludes half of the remaining list at every step of the search and will therefore only look at much fewer elements than the linear search above. To help you out, here is an implementation of a binary search function in Python:

   ```
   def binary_search(sorted_list, element):
       """
       Search for element in list using binary search.
       Assumes sorted list
       """
       # Current active list runs from index_start up to
       # but not including index_end
       index_start = 0
       index_end = len(sorted_list)
   ```

```
    while (index_end - index_start) > 0:
        index_current = (index_end-index_start)//2 + index_start
        if element == sorted_list[index_current]:
            return True
        elif element < sorted_list[index_current]:
            index_end = index_current
        elif element > sorted_list[index_current]:
            index_start = index_current+1
    return False
```

Add this function to the `handin4` module. Now add another function called `wordfile_differences_binary_search`. Again, this function should take two filename arguments, and return a list of the words that appear in the first file, but not in the second. This difference is that this time, you should call the `binary_search` function for each element in the outer loop.

Inside handin4_test.py, call the `wordfile_differences_binary_search` on the input files just as you did for `wordfile_differences_linear_search`, and saves the result in a variable called `differences_binary_search`. Again, use the `timeit` module to measure the time it takes to calculate the differences, and save this result in a variable called `time_spent_binary_search`.

4. Finally, we will test the speed of lookups in a Python dictionary. First, we need functionality to read a file into a dictionary instead of a list. Create a function called `wordfile_to_dict` in the `handin4` module. This function should be identical to `wordfile_to_list`, but it should save the results as keys in a dictionary rather than in a list (you can choose whatever you like for the values - for instance None).

   Inside handin4_test.py call the wordfile_to_dict function on the american-english file, and save the result in a variable called `worddict_american`.

5. Now let's use this in the last comparison function. Add a function to the `handin4` module called `wordfile_differences_dict_search`, which takes two filenames as arguments, and calls `wordfile_to_list` on the first file and `wordfile_to_dict` on the second file. The function should contain a loop that for each word in the list looks in the dictionary to see if there is a match. It should return a list of words that are in the first file but not in the second file.

   The test code in `handin4_test.py` should be similar to the two others, but now call the `wordfile_differences_dict_search` on the input files. Save the result in a variable called `differences_dict_search`_dict_search and use the `timeit` module to measure the time it takes to do the calculation, saving it in a variable called `time_spent_dict_search`.

Note that on the CodeGrade test server, we will be testing your code on slightly smaller versions of the dictionary files - only including the letters from "a" to "g". If you want to run on these files yourself, you can download them here: **https://wouterboomsma.github.io/ppds2023/data/british-english-a-g (https://wouterboomsma.github.io/ppds2023/data/british-english-a-g)** and **https://wouterboomsma.github.io/ppds2023/data/american-english-a-g**

(https://wouterboomsma.github.io/ppds2023/data/american-english-a-g) (this is completely optional).

Part 2: Project

We will continue with processing the Land_and_Ocean_summary.txt file. This time, we will wrap the code from Handin3 into a simple class.

1. Inside `handin4_project.py`, create a class called `AnomalyData`. The constructor should take two arguments: `filename` (a string), and `year_range` (a tuple of two numbers). The constructor should do the same as `read_data3` from `Handin3` (feel free to copy&paste), but instead of returning the resulting dictionary, it should now save it as an attribute called `data`. Just as in Handin3, `year_range` should be optional; if it is not specified, values for all years should be returned.

   Inside `handin4_project_test.py`, test your code in the following way:

   ```
   anomaly_data = AnomalyData('Land_and_Ocean_summary.txt')
   value = anomaly_data.data[1990][0]
   ```

2. Inside your `AnomalyData` class definition, add a method called `one_value_per_decade` that takes no arguments (except `self`). This method should create a new local dictionary variable. It should then iterate over the year keys in the `data` attribute, and save the dictionary values occurring on the first year of a decade (i.e. for 1850, 1860, 1870, ...) to this new dictionary, in the same format as the original (i.e. years as keys, and a list of floats as values). Finally, it should return this dictionary. Hint: the relevant year values are those that have zero remainder when dividing by 10. NOTE: it has been brought to our attention that the name of the method is perhaps slightly confusing - when we write "one value per decade", we mean "one list value" (but that one list contains values for multiple columns in the file).

   Inside `handin4_project_test.py`, test your code by calling the `one_value_per_decade` method on the `anomaly_data` object, and save the result in a variable called `decade_dict`.