Assignment 3: Working with arxiv.org metadata

No Extensions

Please, note that NO EXTENSIONS are allowed for this assignment, since it is due on the last day of classes. Only AccessAbility requests and illness- or emergency-related requests will be accepted. Please, submit these by email to Anya, the course coordinator, with appropriate documentation.

Purpose

The purpose of this assignment is to give you practise working with files, building and using dictionaries, designing functions, and writing unit tests.

Introduction: arxiv.org

arxiv.org is a free distribution service and an open-access archive for nearly two million scholarly articles in the fields of physics, mathematics, computer science, quantitative biology, quantitative finance, statistics, electrical engineering and systems science, and economics.

arxiv.org maintainers believe in open, free, and accessible information. In addition to free and easy access to the articles themselves, arxiv.org also provides ways to harvest its metadata. This metadata includes information such as the article's unique identification number, author(s), title, abstract (Aside: Every scientific article has an *abstract* — a brief summary of the main points of the article), the date the article was added to the arxiv and when it was last modified, licence under which the article was published, etc. This metadata is used by a variety of research tools that investigate scientific research trends, provide intelligent scientific search techniques, and in many other areas.

To make this assignment more manageable for you, we have extracted a sample of arxiv's metadata, simplified it, and created a text file you will use as input to your program.

The Metadata File

The metadata file contains a series of one or more article descriptions, one after the other. Each article description has the following elements, in order:

1. A line containing a unique identifier of the article. An identifier will not occur more than once in the file, and it does not contain any whitespace.
2. A line containing the article's title. If we do not have title information, this line will be blank.
3. A line containing the date the article was created, or a blank line if this information is not provided.
4. A line containing the date the article was last modified, or a blank line if this information is not provided.
5. Zero or more lines with the article's author(s). Each line contains an author's last name followed by a comma `,` followed by the author's first name(s). There is always exactly one comma `,` on the author line. Note, that there may be white space or punctuation other than commas included in an author's first/last name. Immediately after the zero or more author lines, the file contains a single blank line. If we do not have any author information for an article, then the blank line will come immediately after the modification date line.
6. Zero or more lines of text containing the abstract of the article. Immediately after the abstract, the file contains a line with the keyword `END` on it (and nothing else). You may assume that a line with only `END` in it does not occur in any other context in the metadata file, i.e. it always signifies that an article description is over. Unless this is the last line in the file, the next line will contain the identifier of the next article, and so on.

Example Metadata File

Here is an example metadata file (also in your starter code!):

```
008
Intro to CS is the best course ever
2023-09-01

Ponce,Marcelo
Tafliovich,Anya Y.

We present clear evidence that Introduction to
Computer Science is the best course.
END
031
Calculus is the best course ever

2023-09-02
Breuss,Nataliya

We discuss the reasons why Calculus I
is the best course.
END
067
Discrete Mathematics is the best course ever
2023-09-02
2023-10-01
Pancer,Richard
Bretscher,Anna

We explain why Discrete Mathematics is the best course of all times.
END
827
University of Toronto is the best university
2023-08-20
2023-10-02
Ponce,Marcelo
Bretscher,Anna
Tafliovich,Anya Y.

We show a formal proof that the University of
Toronto is the best university.
END
042

2023-05-04
2023-05-05

This is a very strange article with no title
and no authors.
END
```

The above metadata file contains information on five articles with unique identifiers `'008'`, `'031'`, `'067'`, `'827'`, and `'042'`. Note that the following information is NOT provided in the file: modified date in article `'008'`, created date in article `'031'`, and title and authors in article `'042'`. All these are valid cases and your code should deal with them. Also note that an abstract can occupy zero or one or more lines in the input file.

Data Structures

We will use a dictionary to maintain the arxiv metadata. Let us look in detail at the format of this dictionary.

Type `NameType`

We will store names of authors as `Tuple`s of two `str`s: the author's last name(s) and the author's first name(s). For example, the author `Anya Y. Tafliovich` will be stored as `('Tafliovich', 'Anya Y.')`. Note, that there may be punctuation and/or white space included in an author's last name and/or first name, and we need to keep all this information. The only exception is: there are no commas in author's first nor last names. For example, `Van Dyke,Mary-Ellen` and `Sklodowska Curie,Marie Salomea` are both valid input lines, and should be stored as `('Van Dyke', 'Mary-Ellen')` and `('Sklodowska Curie', 'Marie Salomea')`, respectively. A line like `Tafliovich,Anya,Y.` is **not** valid, since it contains two commas and we cannot tell which is supposed to be the first and which is the last name. You will only have to deal with valid input in this assignment.

We call this type `NameType` for readability: see file `constants.py`.

Type `ArticleType`

The file `constants.py` in the starter code defines the following constants:

```
ID = 'identifier'
TITLE = 'title'
CREATED = 'created'
MODIFIED = 'modified'
AUTHORS = 'authors'
ABSTRACT = 'abstract'
```

We will store information about a single article in a dictionary that maps `ID`, `TITLE`, `CREATED`, `MODIFIED`, `AUTHORS`, and `ABSTRACT` to the corresponding values. A value will be `None` if no information is provided in the metadata file. If the information is provided, then the value is of type `str`. The exception is the value associated with key `AUTHORS`, which is a `list` of `NameType`, and is the empty list `[]` if no information on authors is provided.

We will store the author's list sorted in lexicographic order. This is not very realistic, since the author order in a publication is meaningful: depending on the field, the first or the last author is meant to be the "main" author of the article. For the purposes of our assignment, we will keep all lists sorted. Note that Python's built-in list method `sort` and function `sorted` both use lexicographic sorting by default, and you are welcome to use them in your code.

For example, if the values of the constants are as currently defined in `constants.py`, then the article with the identifier `'008'` in our example input file above will be stored in the following dictionary:

```
{'identifier': '008',
 'title': 'Intro to CS is the best course ever',
 'created': '2023-09-01',
 'modified': None,
 'authors': [('Ponce', 'Marcelo'), ('Tafliovich', 'Anya Y.')],
 'abstract': 'We present clear evidence that Introduction to\nComputer Science is the best course.'}
```

Note that since the fourth line in the specification (the line that contains modified time) is blank, the value corresponding to key `MODIFIED` is `None`. Also note that the final newline character on each line is not included in any of the stored values, except for the newline characters *inside* the abstract — we keep those! Take a careful look at the starter file `example_data.txt` (same as the example above) and the corresponding dictionary `EXAMPLE_ARXIV` defined in the file `arxiv_functions.py` for more examples.

Type `ArxivType`

Finally, we will store the entire arxiv metadata in a dict that maps article identifiers to articles, i.e. to values of type `ArticleType`. If the constants are as currently defined in `constants.py`, then the key/value pair that corresponds to the above article is:

```
'008': {
    'identifier': '008',
    'title': 'Intro to CS is the best course ever',
    'created': '2023-09-01',
    'modified': None,
    'authors': [('Ponce', 'Marcelo'), ('Tafliovich', 'Anya Y.')],
    'abstract': 'We present clear evidence that Introduction to\nComputer Science is the best course.'
}
```

Required Functions

In the starter code file `arxiv_functions.py`, complete the following function definitions. In addition, you must add some helper functions to aid with the implementation of these required functions.

Function name: (Parameter types) -> Return type	**Full Description (paraphrase to get a proper docstring description)**
`make_author_to_articles: (ArxivType) -> dict[NameType, list[str]]`	The parameter represents arxiv metadata. This function should return a dictionary that maps each author name to a list of identifiers of articles written by that author. The list should be sorted in lexicographic order.

	Hint: You can simply use the built-in method `sort` or function `sorted` to do the sorting. It is a good idea to build the dictionary first, and then sort the article lists (the values in the dictionary).
	
	For example, if the input is a dictionary that represents the information from our example metadata file, then `make_author_to_articles` should return the dictionary
	
	`{`
	`    ('Ponce', 'Marcelo'): ['008', '827'],`
	`    ('Tafliovich', 'Anya Y.'): ['008', '827'],`
	`    ('Bretscher', 'Anna'): ['067', '827'],`
	`    ('Breuss', 'Nataliya'): ['031'],`
	`    ('Pancer', 'Richard'): ['067']`
	`}`

`get_coauthors: (ArxivType, NameType) -> list[NameType]`	The first parameter represents arxiv metadata and the second parameter represents an author's name. This function should return a list of coauthors of the author specified by the second argument. (Two people are coauthors if they are authors of the same article.) The list should be sorted in lexicographic order. Hints: You can simply use the built-in method `sort` or function `sorted` to do the sorting. Make sure you don't claim that a person is her own coauthor and don't include any names more than once in the return list. Make sure your code does not crash if the author does not appear in the input dictionary at all! Consider using function(s) you already defined to simplify your solution. For example, if the first argument is a dictionary that represents the information from our example metadata file and the second argument is `('Tafliovich', 'Anya Y.')`, then this function should return the list `[('Bretscher', 'Anna'), ('Ponce', 'Marcelo')]` Note that `('Ponce', 'Marcelo')` appears only once in the list, even though he authored two articles with `('Tafliovich', 'Anya Y.')`, namely articles `'008'` and `'827'`.
`get_most_published_authors: (ArxivType) -> list[NameType]`	The parameter represents arxiv metadata. This function should return a list of authors who published the most articles. Note that this list has more than one author only in case of a tie! The list should be sorted in lexicographic order. Hints: You can simply use the built-in method `sort` or function `sorted` to do the sorting. Consider using function(s) you already defined to simplify your solution. Specifically, it is a good idea to implement the function `make_author_to_articles` before this one. For example, if input is a dictionary that represents the information from our example metadata file, then this function should return the list `[('Bretscher', 'Anna'), ('Ponce', 'Marcelo'), ('Tafliovich', 'Anya Y.')]`
`suggest_collaborators: (ArxivType, NameType) -> list[NameType]`	The first parameter represents arxiv metadata and the second parameter represents the author's name. This function should return a list of authors with whom the author specified by the second argument is encouraged to collaborate. The list should be sorted in lexicographic order. The list of suggested collaborators should include all authors who are coauthors of this author's coauthors. In other words, if author `A` wrote

	an article with author `B` and author `B` wrote an article with author `C`, then we will include `C` as suggested collaborator for `A`.
	
	Hints: You can simply use the built-in method `sort` or function `sorted` to do the sorting. Consider using function(s) you already defined to simplify your solution. Make sure you do not include people who are already coauthors of the given author in the return list (they already know each other!). Make sure you don't include the author herself as a suggested collaborator. Finally, make sure the resulting list does not contain any names more than once.
	
	For example, if the first argument is a dictionary that represents the information from our example metadata file and the second argument is `('Pancer', 'Richard')`, then this function should return the list
	
	`[('Ponce', 'Marcelo'), ('Tafliovich', 'Anya Y.')]`
	
	and if the second argument is `('Tafliovich', 'Anya Y.')`, then this function should return the list
	
	`[('Pancer', 'Richard')]`

`has_prolific_authors: (dict[NameType, list[str]], ArticleType, int) -> bool`	The first parameter is a dictionary that maps author name to a list of IDs of articles published by that author, the second parameter represents the information on a single article, and the third argument represents the minimum number of publications required for an author to be considered prolific. The function should return True if and only if the article (second argument) has at least one author who is considered prolific.
	
	For example, if the first argument is the dictionary
	
	`{`
	`    ('Ponce', 'Marcelo'): ['008', '827'],`
	`    ('Tafliovich', 'Anya Y.'): ['008', '827'],`
	`    ('Bretscher', 'Anna'): ['067', '827'],`
	`    ('Breuss', 'Nataliya'): ['031'],`
	`    ('Pancer', 'Richard'): ['067']`
	`}`
	
	the second argument is the article with ID `'008'` from our example, and the third argument is `2`, then the function should return `True`, because at least one of the authors of the article with ID `'008'` is "prolific", i.e., has published at least 2 papers. (In fact, both Marcelo Ponce and Anya Tafliovich are "prolific" in the example.) If the second argument is the article with ID `'031'` (and the other arguments are the same), then the function should return `False`, since none of the authors of this article have published at least two papers.
	
	You can assume that every author of the input article also appears as a key in the dict which is the first argument.

`keep_prolific_authors: (ArxivType, int) -> None`	The first parameter represents arxiv metadata and the second parameter represents the minimum number of publications required for an author to be considered prolific. The function should modify its first argument so that it contains only articles published by prolific authors, i.e., articles that have at least one author who has published at least the minimum required number of articles. For example, if the first argument is a dictionary that represents the information from our example metadata file and the second argument is `2`, then the function should modify its first argument by removing the articles with IDs `'031'` and `'042'`, and keeping the articles with IDs `'008'`, `'067'`, and `'827'`.

	In this example, we have three authors who have published at least two articles: `('Ponce', 'Marcelo')`, `('Tafliovich', 'Anya Y.')`, and `('Bretscher', 'Anna')`, i.e. three "prolific" authors. Therefore, the function should keep only those articles who have (at least) one of these three people as authors: articles with IDs `'008'`, `'067'`, and `'827'`. Hint: Notice that this function returns `None` and *modifies its argument*. Recall that in Python you never want to remove items from a dictionary while iterating over that same dictionary. One way to avoid this issue in your solution is to first decide on which articles should be removed from the dictionary, and then remove them in a separate loop. More hints: Consider using the functions `make_author_to_articles` and `has_prolific_authors` as helpers to simplify your solution. When writing tests for this function, remember that the function modifies its input. To make sure that the tests do not interfere with each other, we often create copies of sample input and pass these copies to the function we are testing. Take a careful look at the docstring for this function which we provided in the starter code and pay attention to the use of `copy.deepcopy` in the example calls.

`read_arxiv_file:`
`(TextIO) -> ArxivType`
	The parameter represents an arxiv metadata file that is **already open for reading**. This function should read the file and return the data in the `ArxivType` dictionary format. Take a careful look at the starter file `example_data.txt` (same as the example above) and the corresponding dictionary `EXAMPLE_ARXIV` defined in the file `arxiv_functions.py` for an example.
	
	Note: in the docstring, do not provide example calls for functions that read files.
	
	Hints: It is a *very* good idea to write a couple of helper functions for `read_arxiv_file`. This is one of the more challenging functions, so it may make sense to implement the functions above first.

Functions to implement in `arxiv_functions.py`

Required Testing (`unittest`)

Complete a set of unittests for the `get_most_published_authors` function. The starter code is in the file `test_get_most_published_authors.py`. For each test method, include a brief docstring description specifying what is being tested. For unittest methods, we do not include a type contract and the docstring description should not include example calls.

Files to Download

All of the files included in the download for the assignment are listed in this section. These files must all be placed in the same folder. Download the files by right-clicking on the link and using your web-browser's "Save as" option. Do not use "cut and paste" as sometimes this can cause undesirable formatting characters to be added to our Python code.

Please download the [Assignment 3 Files] and extract the zip archive. The following paragraphs explain the files you have been given.

- Python Code
  - The starter code for this assignment is in the files `constants.py` and `arxiv_functions.py`. You need to complete the file `arxiv_functions.py`.
  - The a3 checker's main file is `a3_checker.py`. It is complete and must not be changed.
  - The starter code for the required unittests is in the file `test_get_most_published_authors.py`. You need to complete the file.

- Sample arxiv metadata
    - We have provided two text files containing sample data. The file `example_data.txt` corresponds exactly to the example we used throughout this handout. The file `data.txt` is much larger, albeit it is still a small part of the real data available on arxiv.org.

`a3_checker.py`

We have provided a checker program (`a3_checker.py`) that tests two things:

- whether your functions have the correct parameter and return types, and
- whether your code follows the Python and CSCA08 style guidelines.

The checker program does not test the correctness of your functions, so you must do that yourself.

Error checking

You may assume that the input actually contains a valid arxiv metadata file as described in this handout. You do not need to check if the input is valid for any of the functions in this assignment.

Marking

These are the aspects of your work that will be marked for Assignment 3:

- **Correctness (70%):** Your functions should perform as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks. Once your assignment is submitted, we will run additional tests, not provided in the checker. Passing the checker **does not** mean that your code will earn full marks for correctness.
- **Testing (15%):** Your `unittest` test suite will be checked by running it on faulty/broken implementations. Your tester should catch all errors for full marks!
- **Coding style (15%):** Make sure that you follow Python style guidelines that we have introduced and the Python coding conventions that we have been using throughout the semester. Although we don't provide an exhaustive list of style rules, the checker tests for style are complete, so if your code passes the checker, then it will earn full marks for coding style with one exception: docstrings will be evaluated separately. For each occurrence of a PyTA error, a 1 mark (out of 15) deduction will be applied. For example, if a C0301 (line-too-long) error occurs 3 times, then 3 marks will be deducted.

What to Hand In

The very last thing you do before submitting should be to run the checker program one last time.

Otherwise, you could make a small error in your final changes before submitting that causes your code to receive zero for correctness.

Submit `arxiv_functions.py` and `test_get_most_published_authors.py` on MarkUs. Remember that spelling of filenames, including case, counts: your files must be named exactly as required.