ECE 364 Prelab Assignment 05

Python Generators, Ordered Mappings, Named Tuples

Due: September 23, 2018

Passing this lab will satisfy course objectives CO2, CO3, CO7

Instructions

- You must meet all *base requirements* in the syllabus to receive any credit.
- Follow all rules in the **Code Quality Standards**.
- Work in your Prelab05 directory.
- Copy all files from ∼ee364/labfiles/Prelab05 into your Prelab05 directory. You may use the following command:
  `cp -r ∼ee364/labfiles/Prelab05/* ./`
- Remember to add and commit all files to SVN. **We will grade the version of the file that is in SVN!**
- To submit, commit your files to SVN. We will only grade the version of the file that is in SVN.
- Make sure your output from all functions match the given examples. **You will not receive points for a question whose output mismatches the expected result.** For functions that return an unordered collection (e.g., `dict`, `set`, etc.), it's okay if the order in your output differs from what is shown in this document. They are still considered "equal".
- Unless otherwise specified, you may not use any external library. You may use any module in the **Python Standard Library** to solve this lab.
- Make sure you are using Python 3.4 for your project. In PyCharm, go to:

  File Menu → Settings → Project Interpreter

  And make sure that Python 3.4 (`/usr/local/bin/python3.4`) is selected.

Failure to meet all base requirements will result in zero credit. This includes submitting code that works with other versions of Python, but not Python 3.4.

**** Updated 9/18/2018 3:00 PM: Fixed examples in #1 and #3 ****

Python Collections

Description

Natural language processing (NLP) is one of the application domains Python is best known for. At the heart of NLP is text processing, which is the focus of this Prelab.
When processing a large corpus of text (e.g., thousands of books), it is often infeasible to load it all into memory at once. Even for medium-sized corpora (e.g., every State Of The Union address in history), loading everything in memory needlessly can add lag to a system.
Generators allow us to process data in streams. This avoids the need to load all of the data at once. The same is true for returning results. Just as the `range(..)` function allows us to iterate through a range of numbers without creating a list of every number, we can create functions that return results as a generator (a kind of *iterable*), instead of returning a list or tuple with all of the results at once.
In this prelab, you will create several utilities for analyzing text.

Requirements

Create a Python file named `text_utils.py`, and do all of your work in that file. This file should only consists of function blocks, and, optionally, a conditional main block, (i.e. `if __name__ == "__main__":`).
You may include any number of additional utility/helper functions that you might need.
Follow the code quality standards. For example, helper functions should begin with "_". Your final submission should not include any functions, imports, or variables that would not be used as a result of calling the public functions (specified below) or testing directly (code under your `if __name__ == "__main__":`).
The parameter called `word` that is passed to every function will be a generator (or other iterable). That means code like `words[i]` will not work.
Functions that return generators may be implemented as generator functions (i.e., with `yield`) or generator expressions (i.e., `(expr for var in iterable)`).
How much work is this? This entire Prelab can be implemented in as few as 21 sloc (source lines of code, excluding whitespace and comments), without violating the code standards or using anything we haven't covered. Keep in mind that shorter isn't always better. This is just a rough measure of work, not a challenge.

1. [**20%**] Write a function called `get_distinct_words(words)` that returns a generator of every distinct (unique) word.

```
>>> words = ("au", "bu", "cu", "cu", "cu", "du", "du", "du", "du", "du", "eu", "eu", "eu")
>>> words_gen = (s for s in words)  # For testing.  Otherwise, this would be ridiculous.
>>>
>>> get_distinct_words(words_gen)
<generator object <genexpr> at 0x7f1156e765e8>
>>> list( get_distinct_words(words_gen) )
['au', 'bu', 'cu', 'du', 'eu']
>>>
```

2. [**12%**] Add a *namedtuple* definition called "WordPosition" called `WordPosition`. It will be something like `WordPosition = collections.namedtuple(...)`. It should have two attributes: "word" and "idx". It should be at the module level (i.e., not indented).
3. [**16%**] Write a function called `get_first_appearances(words)` that returns a generator of `WordPosition` objects for each distinct word. With each object `word_pos`, `word_pos.idx` should be the 0-based index of the first occurrence of `word_pos.word` in `words`.

   Caution: `idx` is 0-based. The first item returned should have `.idx == 0` (not 1).

```
>>> words = ("au", "bu", "cu", "cu", "cu", "du", "du", "du", "du", "du", "eu", "eu", "eu")
>>> words_gen = (s for s in words)  # For testing.  Otherwise, this would be ridiculous.
>>>
>>> get_first_appearances(words_gen)
<generator object <genexpr> at 0x7f1156e764c8>
>>>
>>> list( get_first_appearances(words_gen) )
[WordPosition(word='au', idx=0), WordPosition(word='bu', idx=1),
 WordPosition(word='cu', idx=2), WordPosition(word='du', idx=5),
 WordPosition(word='eu', idx=10)]
>>>
```

4. [**20%**] Write a function called `get_word_to_first_idx(words)` that returns a mapping (e.g., `dict` of `{word: first_idx, ...}` where `first_idx` is the 0-based position of the first occurrence of `word` in `words`. For example, if the first word in `words` was "apple", then `get_word_to_first_idx(words)["apple"]` should be `0`.

```
>>> words = ("au", "bu", "cu", "cu", "cu", "du", "du", "du", "du", "du", "eu", "eu", "eu")
>>> words_gen = (s for s in words)  # For testing.  Otherwise, this would be ridiculous.
>>>
>>> get_word_to_first_idx(words_gen)
{'du': 5, 'au': 0, 'cu': 2, 'bu': 1, 'eu': 10}
>>>
```

5. [**16%**] Write a function called `get_word_to_frequency(words)` that returns a mapping (e.g., `dict`) of `word: frequency, ...}` where `frequency` is the number of times `word` occurs in `words`.

```
>>> words = ("au", "bu", "cu", "cu", "cu", "du", "du", "du", "du", "du", "eu", "eu", "eu")
>>> words_gen = (s for s in words)  # For testing.  Otherwise, this would be ridiculous.
>>>
>>> get_word_to_frequency(words_gen)
{'du': 5, 'au': 1, 'cu': 3, 'bu': 1, 'eu': 3}
>>>
```

6. [**16%**] Write a function called `get_word_to_frequency_ordered_by_first_idx(words)` that returns an ordered mapping (i.e., `OrderedDict`) of `word: frequency, ...}` where `frequency` is the number of times `word` occurs in `words`, in the order that each word first appeared in `words`.

   Caution: The ordered mapping should be in order that each word first appeared, *not* the order of frequency.

```
>>> words = ("au", "bu", "cu", "cu", "cu", "du", "du", "du", "du", "du", "eu", "eu", "eu")
>>> words_gen = (s for s in words)  # For testing.  Otherwise, this would be ridiculous.
>>>
>>> get_word_to_frequency_ordered_by_first_idx(words_gen)
OrderedDict([('au', 1), ('bu', 1), ('cu', 3), ('du', 5), ('eu', 3)])
>>>
```