CSC110 Assignment 3: Loops, Mutation, and Applications

Azalea Gui & Peter Lin

October 31, 2021

Part 1: Text generation, uniformly random model

1. (a)

Iteration	`word`	`words`	`word_frequencies`
0	N/A	`[]`	`[]`
1	`’Hello’`	`[’Hello’]`	`[2]`
2	`’Amy’`	`[’Hello’, ’Amy’]`	`[2, 1]`
3	`’was’`	`[’Hello’, ’Amy’, ’was’]`	`[2, 1, 1]`
4	`’here’`	`[’Hello’, ’Amy’, ’was’, ’here’]`	`[2, 1, 1, 1]`

   (b) Including a specific example as the doctest’s expected output when a function is random isn’t a good idea because the function’s output will be different from the expected output each time it is executed. Since the doctest only verifies if the actual output matches the expected output, specifying a single random outcome as the expected output among many other possibilities will likely produce an error when running the test.

   (c) For example, you can use `words = {’Hello’: 1}` as the words dictionary. In this case, `generate_text_uniform(words, 5)` has only one possible outcome, which is `’Hello Hello Hello Hello Hello’`, so we can use that as our statement and expected output in the doctest.

2. Complete this part in the provided `a3_part1.py` starter file. Do **not** include your solution in this file.

Part 2: Text generation, One-Word Context Model

0. This question is not to be handed in.

1. One-word context model:

```
{
    'Love': ['is', 'is'],
    'is': ['patient.', 'kind.', 'not'],
    'patient.': ['Love'],
    'kind.': ['It'],
    'It': ['does', 'does', 'is'],
    'does': ['not', 'not'],
    'not': ['envy.', 'boast.', 'proud.'],
    'envy.': ['It'],
    'boast.': ['It']
}
```

2. Complete this part in the provided `a3_part2.py` starter file. Do **not** include your solution in this file.

Part 3: Loops and Mutation Debugging Exercise

1. The test `test_star_wars` passed, and the tests `test_legally_blonde` and `test_transformers` failed.

2. i. The test `test_legally_blonde` failed because of a mistake in the funtion `clean_text`. Since string values are not mutable, and the function `str.lower(str)` does not mutate the string, `str.lower(text)` did not convert the words in the string to lower case but created a new string with lower-cased letters of the original string instead. However, since the result of `str.lower(text)`, the new lower-cased string, wasn't stored back into text, the string value in the variable text is not lower-cased. Since the non-lower-cased string is used when processing the words, some of the words could not be matched to the VADER intensities dictionary, and they were not counted in the intensity calculation even though they should be counted.

   ii. The test `test_transformers` failed because of a mistake in the function `count_keywords`. It should loop through the word list, find words that have VADER intensity data, and create a dictionary of the number of occurences of these words. When creating the dictionary, it uses an accumulator `occurences_so_far`. The keys of the dictionary represent these words and the values represent the number of times these words appear. When a new word `word` is found that isn't in the accumulator, it should initialize `occurences_so_far[word]` to 1, and when a word `word` that's already in the accumulator is found, it should add one to `occurences_so_far[word]`. The given code completed the first part (initializing the counts of new words to 1) correctly, but it didn't add one when existing words are detected, so the returned result always reported one occurence for each word when some words actually appeared multiple times. Since the reported word occurences were incorrect, the calculated intensity were multiplied by potentially the wrong amount, which lead to the incorrect intensity.

3. The test `test_star_wars` passed because the review did not contain repeating or non-lower-cased words. The review text only contains three words that are in the small subset of the VADAR lexicon used in the program: *magnificent* , *adventure* , and *succeeded.* The problem in 2.i didn't affect this text because these three words are all already lower-cased, and the problem in 2.ii didn't affect this text because these three words all appeared only once in the text.

Part 4: Forest Fires

1. Complete this part in the provided `a3_part4.py` starter file. Do **not** include your solution in this file.

2. Complete this part in the provided `a3_part4_tests.py` starter file. Do **not** include your solution in this file.

3. a. The precipitation attribute is a float, and in Python floats are immutable. In `calculate_mr`, if ever the `precipitation` parameter is changed, it will be a variable reassignment, which will change the id of `precipitation`. This does not change the id of `wm.precipitation`, which is why `calculate_mr` can never mutate the `precipitation` attribute of `wm`.

   b. The function wants to use the given temperature or -2.8, whichever is higher. If the `wm.temperature` attribute is below -2.8 and gets reassigned to -2.8, then it will mutate the `wm` object, which is unwanted. However, if it is first stored into a different variable `temperature`, then any change to `temperature` won't affect `wm.temperature`, and therefore `wm` won't get mutated, which is good.

   c. The elements of a tuple themselves can be mutated, as stated in the question. However, they cannot be reassigned, the id's of the elements will all stay the same, and elements cannot be added or removed, which makes tuples immutable.