 CS 1110 Spring 2022, Assignment 3: Updates

**Updates:**

- Tue Mar 15, 4pm: Specification for percent_avg_or_shorter(), pg. 6, should say that the example result is 75.0, not .75 .
- Mon Mar 14, 4:30pm: Syntax for sorting a list, pg. 8, fixed.

# CS 1110 Spring 2022, Assignment 3: The simplification of US Political Speech?*

> Biden Joins Modern Presidents in Using Simple Language in State of the Union: Biden's first address scored lower than any previous SOTU speech, continuing a trend of presidents speaking more simply. — NBC Washington news headline[1], March 1, 2022

Some articles have applied text analysis to claim that there's been a trend towards "dumbing down"[2] political speech in the US. Regardless of whether political speech is actually getting simpler in the US, or what a trend towards less-complex political language would really signify and whether it would constitute a "bad" outcome, you can use the Python we've already learned in CS1110 to perform your own simple analyses — or analyses of simplicity — of the language used in U.S. State of the Union (SOTU) addresses. You will thus be joining an active research enterprise, and applying your programming skills to an issue related to political science and communication![3] Great job being interdisciplinary, you!

Download the zip file of the files you will need: `http://www.cs.cornell.edu/courses/cs1110/2022sp/assignments/assignment3/a3_skeleton.zip`.

## Contents



*Authors: Lillian Lee, with some high-level ideas from discussion with family (yes, I have the kind of family that talks about word statistics over dinner). Inspiration: this Mar 2 tweet by political scientist Arthur Spirling.

[1] `https://www.nbcwashington.com/news/national-international/biden-joins-modern-presidents-in-using-simple-language-in-state-of-th 2987636/`

[2] Actual phrase used by some journalists.

[3] For a recent publication on this topic, see Kenneth Benoit, Kevin Munger, and Arthur Spirling, Measuring and Explaining Political Sophistication through Textual Complexity[4], *American Journal of Political Science*, 2019.

# 1 New Rules

1. A major goal of this assignment is practice with for-loops. Hence, for each function we ask you to implement, **we reserve the right to assign no credit for code that isn't fundamentally based on an explicit for-loop if we asked for one, or that doesn't use the kind of for-loop we ask for, even if the code fulfills the specification.**

   For example, for this assignment, you may not use the built-in sum() or mean() function for lists.

2. You may not use any Python that will not have been introduced in class or the associated materials after Mar 17; except that you are allowed to use negative indices as a convenience.[5]

3. Unless the specification says otherwise, any function you complete must not change any objects given as arguments. *Changing a user's objects without notifying them is bad practice.*[6]

# 2 Previous Rules That Still Apply

See Sections 1.1-1.3 of Assignment 1[7] and Sections 3.1-3.2 of Assignment 2[8].

# 3 Timeline and Deadlines

(a) If you are partnering,[9] **do so *before* submitting anything**. Groups cannot be formed or changed after submitting.[10]

(b) By 2pm Ithaca time on Sun Mar 20, submit your *partial* progress on CMS.[11] All you have to submit is a3.py. It is OK if you haven't finished working on it yet.[12]

(c) By **11:59pm (Ithaca time) on Sun Mar 20**, make your final submission of all files, and do steps 1-3 in the "Updating, verifying, and documenting assignment submission" section of `https://www.cs.cornell.edu/courses/cs1110/2022sp/resources/cms.html` .

## 3.1 Pre-submission checklist

Before submitting, ensure your code obeys the following.[13]

1. Lines are generally short enough (~80 characters) that horizontal scrolling is not necessary.
2. You have indented with spaces, not tabs.
3. You have removed any debugging `print` statements.
4. You have removed all `pass` statements.
5. Unless you need them to understand your own code, remove our instruction comments, such as "STUDENTS: ..."

[5] **In particular, you are not allowed to use list comprehensions, generator expressions, or ordered dictionaries, even if for some reason you have heard of these things before. Nor can you use numpy functions or methods.**

[6] One might go as far as to say it's . . . objectionable?

[7] `https://www.cs.cornell.edu/courses/cs1110/2022sp/assignments/assignment1/a1.pdf`

[8] `https://www.cs.cornell.edu/courses/cs1110/2022sp/assignments/assignment2/a2.pdf`

[9] Reminder: Both parties need to act on CMS in order for the grouping to take effect. See the "How to form a group" instructions at `https://www.cs.cornell.edu/courses/cs1110/2022sp/resources/cms.html` .

[10] Except for "group divorce" situations; see the CS 1110 Collaboration Policies page.

[11] And, as usual, perform steps 1-3 in the "Updating, verifying, and documenting assignment submission" section of `https://www.cs.cornell.edu/courses/cs1110/2022sp/resources/cms.html` .

[12] The 2pm checkpoints provide you a chance to alert us if any problems arise, and for partners to become aware if something has gone wrong with the CMS grouping process (via us emailing people who haven't yet submitted — if your partner submitted but you got the warning email, that means CMS doesn't think you've grouped!). Since we'll send a reminder around 2pm to all those who haven't submitted yet, do not expect that we will accept work that doesn't make it onto CMS on time, for whatever reason. There are no so-called "slipdays" and there is no "you get to submit late at the price of a late penalty" policy. Of course, if some special circumstances arise, contact the instructor(s) immediately.

[13] These requirements speed up the process of reading/grading hundreds of files.

6. If you added any helper functions, these have good docstring specifications and you have sufficiently tested them ~~put sufficient testing code for your functions in `a1_second.py`~~ .
7. You've changed the header comments in all files to list the entire set of people and sources that contributed to the code.
8. You (and your partner) have included your NetIDs in the header of all files.
9. The date in the header comments has been changed to when the files were last edited.

# 4 Sample data

To test both some definitions of text simplicity and the code you'll write to instantiate such definitions, it's useful to have a range of texts, some seemingly more simple, some seemingly more complex.

## 4.1 Terminology: Word types vs. tokens

It's useful to make the distinction between *(word) types* versus *(word) tokens*: the sequence "bird bird bird" has three *tokens* but only one word type; the sequence "to defenestrate to grudge" has four tokens comprising three word types, since there are two occurrences of the word type "to".

It's natural to think of a speech or other piece of text as being represented by a *token list*, which is simply the ordered list of the tokens[14] in it.

## 4.2 Notes on the provided tokenlists

Below are the contents of and frequency statistics for the tokenlists we created for you in sample_lists.py.

1. `no_repeats` and `one_repeat` are the same length, but the latter is more "simple" in that it uses fewer different kinds of words.
2. It turns out that Wikipedia has a companion site, Simple English Wikipedia[15], with a focus more on "children and adults who are learning English". `wiki` and `simplewiki` are same-length excerpts from the two Wikipedias' respective articles on the Python programming language; you can see that the two excerpts both address the same topic but have different simplicity levels.
3. We've also included passages from Dickens (presumably complex prose in general, although this famous quote is known for its repetitiveness), Dr. Seuss (known for using a purposely reduced vocabulary), Stein (known for challenging poetry but also a "faux-naif" style), and Whitehead (a striking first paragraph to *The Noble Hustle*).

It is our hope that you find comparing their (computed) complexities (kind of) cool.

To control for the potentially confounding factor of passage length, we've truncated all the long tokenlists to have the same number of tokens.

**no_repeats**:
['a', 'b', 'c', 'd']
Number of tokens: 4
*Frequencies per wordtype:* 'a': 1, 'b': 1, 'c': 1, 'd': 1

**one_repeat**:
['a', 'b', 'b', 'c']
Number of tokens: 4
*Frequencies per wordtype:* 'b': 2, 'a': 1, 'c': 1

**repeats**:
['apple', 'banana', 'carrot', 'dog', 'elephant', 'food', 'garage', 'apple', 'banana', 'carrot', 'dog', 'elephant', 'food', 'apple', 'banana', 'carrot', 'dog', 'elephant', 'apple', 'banana', 'carrot', 'dog', 'apple', 'banana', 'carrot', 'apple', 'banana', 'apple']
Number of tokens: 28
*Frequencies per wordtype:* 'apple': 7, 'banana': 6, 'carrot': 5, 'dog': 4, 'elephant': 3, 'food': 2, 'garage': 1

[14] In this assignment, we choose to discard most punctuation and lowercase all tokens. This does have the disadvantage that we can't consider sentence-based statistics. Also, we lose the ability to account for parentheses (of which I am a fan) in measuring prose tortured-ness.

[15] https://simple.wikipedia.org/wiki/Main_Page

**wiki**:

['python', 'is', 'a', 'high-level', 'general-purpose', 'programming', 'language', 'its', 'design', 'philosophy', 'emphasizes', 'code', 'readability', 'with', 'the', 'use', 'of', 'significant', 'indentation', 'its', 'language', 'constructs', 'and', 'object-oriented', 'approach', 'aim', 'to', 'help', 'programmers', 'write', 'clear', 'logical', 'code', 'for', 'small-', 'and', 'large-scale', 'projects', 'python', 'is', 'dynamically-typed', 'and', 'garbage-collected', 'it', 'supports', 'multiple', 'programming', 'paradigms', 'including', 'structured', 'particularly', 'procedural', 'object-oriented', 'and', 'functional', 'programming', 'it', 'is', 'often', 'described', 'as', 'a', 'batteries', 'included', 'language', 'due', 'to', 'its', 'comprehensive', 'standard', 'library', 'guido', 'van', 'rossum', 'began', 'working', 'on', 'python', 'in', 'the']

Number of tokens: 80

*Frequencies per wordtype:* 'and': 4, 'python': 3, 'is': 3, 'programming': 3, 'language': 3, 'its': 3, 'a': 2, 'code': 2, 'the': 2, 'object-oriented': 2, 'to': 2, 'it': 2, 'high-level': 1, 'general-purpose': 1, 'design': 1, 'philosophy': 1, 'emphasizes': 1, 'readability': 1, 'with': 1, 'use': 1, 'of': 1, 'significant': 1, 'indentation': 1, 'constructs': 1, 'approach': 1, 'aim': 1, 'help': 1, 'programmers': 1, 'write': 1, 'clear': 1, 'logical': 1, 'for': 1, 'small-': 1, 'large-scale': 1, 'projects': 1, 'dynamically-typed': 1, 'garbage-collected': 1, 'supports': 1, 'multiple': 1, 'paradigms': 1, 'including': 1, 'structured': 1, 'particularly': 1, 'procedural': 1, 'functional': 1, 'often': 1, 'described': 1, 'as': 1, 'batteries': 1, 'included': 1, 'due': 1, 'comprehensive': 1, 'standard': 1, 'library': 1, 'guido': 1, 'van': 1, 'rossum': 1, 'began': 1, 'working': 1, 'on': 1, 'in': 1

**simplewiki**:

['python', 'is', 'an', 'open', 'source', 'programming', 'language', 'it', 'was', 'made', 'to', 'be', 'easy-to-read', 'and', 'powerful', 'a', 'dutch', 'programmer', 'named', 'guido', 'van', 'rossum', 'made', 'python', 'in', 'he', 'named', 'it', 'after', 'the', 'television', 'program', 'monty', "python's", 'flying', 'circus', 'many', 'python', 'examples', 'and', 'tutorials', 'include', 'jokes', 'from', 'the', 'show', 'python', 'is', 'an', 'interpreted', 'language', 'interpreted', 'languages', 'do', 'not', 'need', 'to', 'be', 'compiled', 'to', 'run', 'a', 'program', 'called', 'an', 'interpreter', 'runs', 'python', 'code', 'on', 'almost', 'any', 'kind', 'of', 'computer', 'this', 'means', 'that', 'a', 'programmer']

Number of tokens: 80

*Frequencies per wordtype:* 'python': 5, 'an': 3, 'to': 3, 'a': 3, 'is': 2, 'language': 2, 'it': 2, 'made': 2, 'be': 2, 'and': 2, 'programmer': 2, 'named': 2, 'the': 2, 'program': 2, 'interpreted': 2, 'open': 1, 'source': 1, 'programming': 1, 'was': 1, 'easy-to-read': 1, 'powerful': 1, 'dutch': 1, 'guido': 1, 'van': 1, 'rossum': 1, 'in': 1, 'he': 1, 'after': 1, 'television': 1, 'monty': 1, "python's": 1, 'flying': 1, 'circus': 1, 'many': 1, 'examples': 1, 'tutorials': 1, 'include': 1, 'jokes': 1, 'from': 1, 'show': 1, 'languages': 1, 'do': 1, 'not': 1, 'need': 1, 'compiled': 1, 'run': 1, 'called': 1, 'interpreter': 1, 'runs': 1, 'code': 1, 'on': 1, 'almost': 1, 'any': 1, 'kind': 1, 'of': 1, 'computer': 1, 'this': 1, 'means': 1, 'that': 1

**dickens**:

['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst', 'of', 'times', 'it', 'was', 'the', 'age', 'of', 'wisdom', 'it', 'was', 'the', 'age', 'of', 'foolishness', 'it', 'was', 'the', 'epoch', 'of', 'belief', 'it', 'was', 'the', 'epoch', 'of', 'incredulity', 'it', 'was', 'the', 'season', 'of', 'light', 'it', 'was', 'the', 'season', 'of', 'darkness', 'it', 'was', 'the', 'spring', 'of', 'hope', 'it', 'was', 'the', 'winter', 'of', 'despair', 'we', 'had', 'everything', 'before', 'us', 'we', 'had', 'nothing', 'before', 'us', 'we', 'were', 'all', 'going', 'direct', 'to', 'heaven', 'we', 'were', 'all']

Number of tokens: 80

*Frequencies per wordtype:* 'it': 10, 'was': 10, 'the': 10, 'of': 10, 'we': 4, 'times': 2, 'age': 2, 'epoch': 2, 'season': 2, 'had': 2, 'before': 2, 'us': 2, 'were': 2, 'all': 2, 'best': 1, 'worst': 1, 'wisdom': 1, 'foolishness': 1, 'belief': 1, 'incredulity': 1, 'light': 1, 'darkness': 1, 'spring': 1, 'hope': 1, 'winter': 1, 'despair': 1, 'everything': 1, 'nothing': 1, 'going': 1, 'direct': 1, 'to': 1, 'heaven': 1

**seuss**:

['the', 'sun', 'did', 'not', 'shine', 'it', 'was', 'too', 'wet', 'to', 'play', 'so', 'we', 'sat', 'in', 'the', 'house', 'all', 'that', 'cold', 'cold', 'wet', 'day', 'i', 'sat', 'there', 'with', 'sally', 'we', 'sat', 'there', 'we', 'two', 'and', 'i', 'said', "'how", 'i', 'wish', 'we', 'had', 'something', 'to', 'do', "'", 'too', 'wet', 'to', 'go', 'out', 'and', 'too', 'cold', 'to', 'play', 'ball', 'so', 'we', 'sat', 'in', 'the', 'house', 'we', 'did', 'nothing', 'at', 'all', 'so', 'all', 'we', 'could', 'do', 'was', 'to', 'sit', 'sit', 'sit', 'sit', 'and', 'we']

Number of tokens: 80

*Frequencies per wordtype:* 'we': 8, 'to': 5, 'sat': 4, 'sit': 4, 'the': 3, 'too': 3, 'wet': 3, 'so': 3, 'all': 3, 'cold': 3, 'i': 3, 'and': 3, 'did': 2, 'was': 2, 'play': 2, 'in': 2, 'house': 2, 'there': 2, 'do': 2, 'sun': 1, 'not': 1, 'shine': 1, 'it': 1, 'that': 1, 'day': 1, 'with': 1, 'sally': 1, 'two': 1, 'said': 1, "'how": 1, 'wish': 1, 'had': 1, 'something': 1, "'": 1, 'go': 1, 'out': 1, 'ball': 1, 'nothing': 1, 'at': 1, 'could': 1

**stein**:

['argonauts', 'that', 'is', 'plenty', 'cunning', 'saxon', 'symbol', 'symbol', 'of', 'beauty', 'thimble', 'of', 'everything', 'cunning', 'clover', 'thimble', 'cunning', 'of', 'everything', 'cunning', 'of', 'thimble', 'cunning', 'cunning', 'place', 'in', 'pets', 'night', 'town', 'night', 'town', 'a', 'glass', 'color', 'mahogany', 'color', 'mahogany', 'center', 'rose', 'is', 'a', 'rose', 'is', 'a', 'rose', 'is', 'a', 'rose', 'loveliness', 'extreme', 'extra', 'gaiters', 'loveliness', 'extreme', 'sweetest', 'ice-cream', 'page', 'ages', 'page', 'ages', 'page', 'ages', 'wiped', 'wiped', 'wire', 'wire', 'sweeter', 'than', 'peaches', 'and', 'pears', 'and', 'cream', 'wiped', 'wire', 'wiped', 'wire', 'extra', 'extreme', 'put']

Number of tokens: 80

*Frequencies per wordtype:* 'cunning': 6, 'is': 4, 'of': 4, 'a': 4, 'rose': 4, 'wiped': 4, 'wire': 4, 'thimble': 3, 'extreme': 3, 'page': 3, 'ages': 3, 'symbol': 2, 'everything': 2, 'night': 2, 'town': 2, 'color': 2, 'mahogany': 2, 'loveliness': 2, 'extra': 2, 'and':

2, 'argonauts': 1, 'that': 1, 'plenty': 1, 'saxon': 1, 'beauty': 1, 'clover': 1, 'place': 1, 'in': 1, 'pets': 1, 'glass': 1, 'center': 1, 'gaiters': 1, 'sweetest': 1, 'ice-cream': 1, 'sweeter': 1, 'than': 1, 'peaches': 1, 'pears': 1, 'cream': 1, 'put': 1

**whitehead**:

['i', 'have', 'a', 'good', 'poker', 'face', 'because', 'i', 'am', 'half', 'dead', 'inside', 'my', 'particular', 'combo', 'of', 'slack', 'features', 'negligible', 'affect', 'and', 'soulless', 'gaze', 'has', 'helped', 'my', 'game', 'ever', 'since', 'i', 'started', 'playing', 'twenty', 'years', 'ago', 'when', 'i', 'was', 'ignorant', 'of', 'pot', 'odds', 'and', 'm-theory', 'and', 'four-betting', 'and', 'it', 'gave', 'me', 'a', 'boost', 'as', 'i', 'collected', 'my', 'trove', 'of', 'lore', 'game', 'by', 'game', 'hand', 'by', 'hand', 'it', 'has', 'not', 'helped', 'me', 'human', 'relationships-wise', 'over', 'the', 'years', 'but', 'surely', "i'm", 'not', 'alone']

Number of tokens: 80

*Frequencies per wordtype:* 'i': 5, 'and': 4, 'my': 3, 'of': 3, 'game': 3, 'a': 2, 'has': 2, 'helped': 2, 'years': 2, 'it': 2, 'me': 2, 'by': 2, 'hand': 2, 'not': 2, 'have': 1, 'good': 1, 'poker': 1, 'face': 1, 'because': 1, 'am': 1, 'half': 1, 'dead': 1, 'inside': 1, 'particular': 1, 'combo': 1, 'slack': 1, 'features': 1, 'negligible': 1, 'affect': 1, 'soulless': 1, 'gaze': 1, 'ever': 1, 'since': 1, 'started': 1, 'playing': 1, 'twenty': 1, 'ago': 1, 'when': 1, 'was': 1, 'ignorant': 1, 'pot': 1, 'odds': 1, 'm-theory': 1, 'four-betting': 1, 'gave': 1, 'boost': 1, 'as': 1, 'collected': 1, 'trove': 1, 'lore': 1, 'human': 1, 'relationships-wise': 1, 'over': 1, 'the': 1, 'but': 1, 'surely': 1, "i'm": 1, 'alone': 1

# 5 Simple measures of language simplicity for you to implement (larger value = simpler)

## 5.1 percent_short(tokenlist, k)

A natural indicator of textual simplicity is preponderance of short words. But what should the definition of "short" be? Our function definition leaves this decision to the caller by requesting a length cut-off parameter `k`: every word of length $\leq$ `k` is considered short.

```
def percent_short(tokenlist, k):
    """Returns: percentage (as float between 0 and 100) of tokens in tokenlist
    that are of length <=k.

    Precondition: k [float] is >= 0. tokenlist is a non-empty list of strings.
    """
    pass # STUDENTS: your implementation must make effective use of for-loops.
         # You can only use lists (no dictionaries, for example)
```

## 5.2 percent_avg_or_shorter(tokenlist, tokenlistlist)

The need to pick a `k` in the previous function is a puzzle (or nuisance). This next function uses language data to choose `k`. Essentially, it first finds the average length of the tokens in a set of samples (i.e., a list of tokenlists), and then uses that average length as the length cut-off. You'll want to call the previous function as a helper.

```
def percent_avg_or_shorter(tokenlist, tokenlistlist):
    """Returns: percentage (as float between 0 and 100) of tokens in tokenlist
    that are of length <=k, where k is the average token length across
    all the tokens in tokenlistlist.

    Example:
     if tokenlist were ["a", "b", "ccccc", "dd"]
     and tokenlistlist were
        [ ["abcd", "abcd", "abcd"],
          ["a", "b", "ccccc", "dd"],
          ["ef"]
        ]
     then k should be (3*4+2*1 +1*5 + 2*2)/8 = 2.875
     and this function should return 3/4 = 75.0 (UPDATE: not .75)

    Precondition: tokenlist is a non-empty list of strings.
    tokenlistlist is a non-empty list of non-empty lists of non-empty strings.
```

```
    tokenlist is in tokenlistlist
    """
    pass # STUDENTS: your implementation must make effective use of
         # a nested for-loop working on a nested list.
```

## 5.3 type_token_ratio(tokenlist)

The intuition behind this function, which is based on the *type-token ratio*, is that the more a text uses the same words over and over, the less complex it might be. Another way to think of this is that there's a spectrum from every token being distinct — `["i", "would", "found", "an", "institution", "where"]` — vs. every token being the exact same word type — `["cornell", "cornell", "cornell", "cornell", "cornell", "cornell"]`.

```
def type_token_ratio(tokenlist):
    """Returns, as a float,

         100*(1 - (# of types in tokenlist/# of tokens in tokenlist))

    Precondition: tokenlist is a non-empty list of strings."""
    pass # STUDENTS: your implementation must make effective use of for-loops.
         # It must NOT use dictionaries.
```

## 5.4 zipf(tokenlist, n)

This function,[16] related to the previous one, computes the percentage of tokens that are "covered" by the $n$ most frequent wordtypes — for a fixed $n$, the more the most common words dominate by being repeated a lot, the greater the coverage.

```
def zipf(tokenlist, n):
    """Returns: percentage (as float between 0 and 100) of tokens in tokenlist
    that are of the top n most common types in tokenlist.

    Precondition: n [int] is > 0, <= the number of distinct wordtypes in tokenlist
        tokenlist is a non-empty list of strings
    """
```

Examples: suppose we had the following two same-length tokenlists:
`t1:` `["a", "a", "a", "a", "b", "b", "b", "c", "c", "d"]`
`t2:` `["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"]`
Then we see that there's more "coverage" of the the top $n$ words in `t1`:

`zipf(t1,1)` $\longrightarrow$ 40.0 (4 "a"s out of 10 tokens)
`zipf(t2,1)` $\longrightarrow$ 10.0 (whichever token is considered most frequent, it's 1 out of 10)
`zipf(t1,2)` $\longrightarrow$ 70.0 (7 "a"s and "b"s out of 10 tokens)
`zipf(t2,2)` $\longrightarrow$ 20.0

### 5.4.1 First idea: represent word types with objects

To implement zipf(), you need to determine what are the most common *word types* in `tokenlist`.

You could do this if you somehow had a list of each word type's frequency, because you could use the sort() function for lists — *if* the list consisted of objects defined in the right way. And we've created an object class, Freq, for you just for this purpose!

[16] We chose the name in honor of Zipf's law. Check out the tangentially related and highly intriguing Benford's law.

### 5.4.2 Class Freq

A Freq object represents the frequency of a specific word type. It has two attributes:

- `wordtype`, a non-empty string like `"cornell"`
- `freq`, a non-negative int

A new Freq object is created with a call like `Freq("love", 14)`.

We've constructed the Freq class so that if `flist` is a list of Freqs, the call `flist.sort(reverse=True)` ~~`sort(flist, reverse=True)`~~ rearranges the list to be in descending order by `freq` attribute. (This is step 3 in the comments for zipf().) You can then examine the first (most frequent) `n` items by list indexing or slicing. (This is step 4 in the comments for zipf().)

### 5.4.3 Second idea: use a dictionary to figure out the counts for each word type

Now, which Freq objects should we create, and what should their attribute values be?

Well, we need to look at each token in `tokenlist`, which calls for a for-loop (see Step 1 in the comments for zipf()). *If* while you were doing so, you were maintaining a dictionary whose keys were word types, and values were frequencies observed so far, like this:

`{"love":11, "birds": 9}`

and you came across the word "birds" again, you could update the dictionary to

`{"love":11, "birds": 10}` .

Whereas if the next word were a new one like "ceremony", i.e., *not already in the dictionary*, you'd add a new entry with a count-so-far of 1:

`{"love":11, "birds": 10, "ceremony": 1}` .

Once you've finished creating this dictionary, you can iterate through it, creating a new Freq for each entry and adding that Freq to a list. (This is step 2 in the comments for zipf().)

# 6 A utility function for you to implement: print_trend(results, labels)

Recall that our over-reaching goal was to measure the simplicity of certain political speeches over a sequence of years. To do so, one could accumulate a list of simplicity scores for those speeches and print them out. But it would be nice to also print out a label for each of those speeches, such as the year it was given, or the speaker. That's the job of this function.

```
def print_trend(results, labels):
    """
    Precondition:  `labels` is a list of strings where each string is a label for
    the corresponding entry in list `results` (which is a same-length list of
    numbers (floats or ints)).
    WARNING: it is possible for there to be repeated values in either `results`
    or `labels`.


    Prints info about `results` and `labels` in the following format (does
    not return anything)

    <label[0]>: <round(results[0], 2)>
    <label[1]>: <round(results[1], 2)>
    ... [etc.]

    The call round(x, 2) rounds to two decimal places.
    """
```

# 7 Suggested implementation/testing order

In a3.py, the functions are ordered in increasing level of challenge. (This is different than the logic-based ordering in this writeup.) Implement and test in the order you see in a3.py.

We've provided test cases for you in a3_test.py. You don't have to submit more tests, but you do need to decide whether you need to add more tests for yourself.

# 8 Grand finale (just for fun, nothing to turn in): real SOTU speech data

We'll release real state-of-the-union data for you next week so if you want, you can use your working code to investigate whether these speeches have gotten more simple over the years!

# A Creating tokenlists from text: sample_lists.py

File sample_lists.py shows how we processed text strings to get them into the tokenlist format we used. If you'd like to run your code on your own text data, you can use that file as a guide.

# B Worked examples of for-loops and object manipulation

Besides the lecture materials, there are solved A3s from previous semesters (including one also involving state-of-the-union addresses!) in the "Archive" section of our assignment advice and archive page[17] and solutions to previous exam questions at our exams page[18].

[17] https://www.cs.cornell.edu/courses/cs1110/2022sp/resources/doing-assignments.html
[18] https://www.cs.cornell.edu/courses/cs1110/2022sp/exams