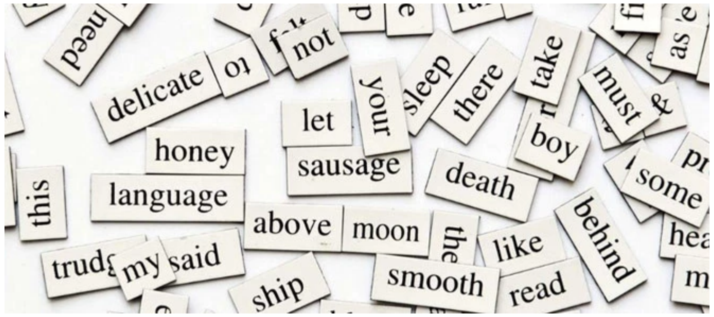
# Accumulation Pattern for Lists and Dictionaries



**CS111 Computer Programming**

Department of Computer Science
Wellesley College

# Recap: Accumulation for different types

```
# count vowels in a string
count = 0
for letter in phrase:
    if isVowel(letter):
        count += 1
```

Accumulator variable as an integer

```
# find odd numbers
oddList = []
for num in numList:
    if num % 2 == 1:
        oddList.append(num)
```

Accumulator variable as a list

```
# encrypt a phrase
phrase = "the sun is shining"
newPhrase = ''
for letter in phrase:
    if isVowel(letter):
        newPhrase += '*'
    elif letter == ' ':
        newPhrase += ' '
    else:
        newPhrase += '_'
print(newPhrase)
```

Accumulator variable as a string

```
# count all letters in a string
word = "abracadabra"
lettersDict = {}
for letter in word:
    if letter not in lettersDict:
        lettersDict[letter] = 1
    else:
        lettersDict[letter] += 1

print(lettersDict)
```

Accumulator variable as a dictionary

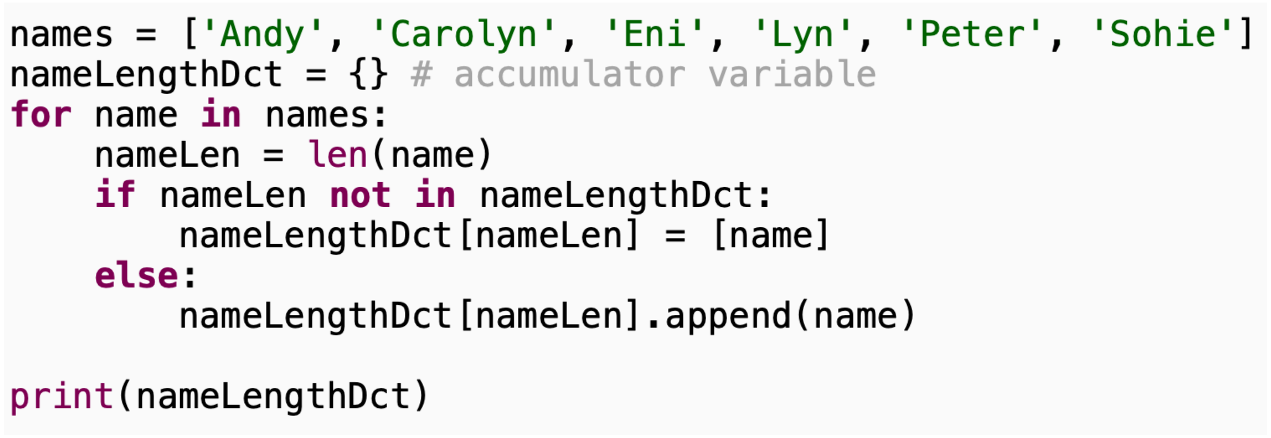
# Accumulation with dict and list types [1]

```
names = ['Andy', 'Carolyn', 'Eni', 'Lyn', 'Peter', 'Sohie']
```

Goal: group names by their lengths.

```
{4: ['Andy'],
 7: ['Carolyn'],
 3: ['Eni', 'Lyn'],
 5: ['Peter', 'Sohie']}
```

This is a double accumulation pattern: the dictionary is accumulating name lengths (as keys), and for each key, we're accumulating the names with a certain length.



```
names = ['Andy', 'Carolyn', 'Eni', 'Lyn', 'Peter', 'Sohie']
nameLengthDct = {} # accumulator variable
for name in names:
    nameLen = len(name)
    if nameLen not in nameLengthDct:
        nameLengthDct[nameLen] = [name]
    else:
        nameLengthDct[nameLen].append(name)

print(nameLengthDct)
```

# Accumulation with dict and list types [2]

```
names = ['Andy', 'Carolyn', 'Eni', 'Lyn', 'Peter', 'Sohie']
```

Goal: a vowel index (like the book index) that shows all names with a certain vowel (no duplicates). **This is hard!**

```
{'a': ['Andy', 'Carolyn'],
 'o': ['Carolyn', 'Sohie'],
 'e': ['Eni', 'Peter', 'Sohie'],
 'i': ['Eni', 'Sohie']}
```

This is one possible solution, but somewhat complex. It uses nested loops and nested conditionals.

```
names = ['Andy', 'Carolyn', 'Eni', 'Lyn', 'Peter', 'Sohie']
vowelIndexDct = {}

for name in names:
    for letter in name.lower():
        if isVowel(letter):
            if letter not in vowelIndexDct:
                vowelIndexDct[letter] = [name]
            else:
                if name not in vowelIndexDct[letter]:
                    vowelIndexDct[letter].append(name)

print(vowelIndexDct)
```

## A simpler solution with dict comprehension

```
names = ['Andy', 'Carolyn', 'Eni', 'Lyn', 'Peter', 'Sohie']
```

Goal: a vowel index (like the book index) that shows all names with a certain vowel (no duplicates). **This is hard!**

```
{'a': ['Andy', 'Carolyn'],
 'e': ['Eni', 'Peter', 'Sohie'],
 'i': ['Eni', 'Sohie'],
 'o': ['Carolyn', 'Sohie'],
 'u': []}
```

```
vowelIndexDct2 = {vowel: [] for vowel in 'aeiou'} # dict comprehension

for vowel in vowelIndexDct2:
    for name in names:
        if vowel in name.lower():
            if name not in vowelIndexDct2[vowel]:
                vowelIndexDct2[vowel].append(name)

print(vowelIndexDct2)
```

# Dictionary Comprehension

**Concepts in this slide**: Dictionary comprehension is very similar to list comprehension.

Very much like list comprehension: use **{ }** instead of **[ ]** and create pairs with the colon syntax, e.g., **aKey: aValue**.

**Syntax:** `{ aKey: aValue for aKey in sequence}`

**Example**: Write a dictionary comprehension that pairs words with their lengths.

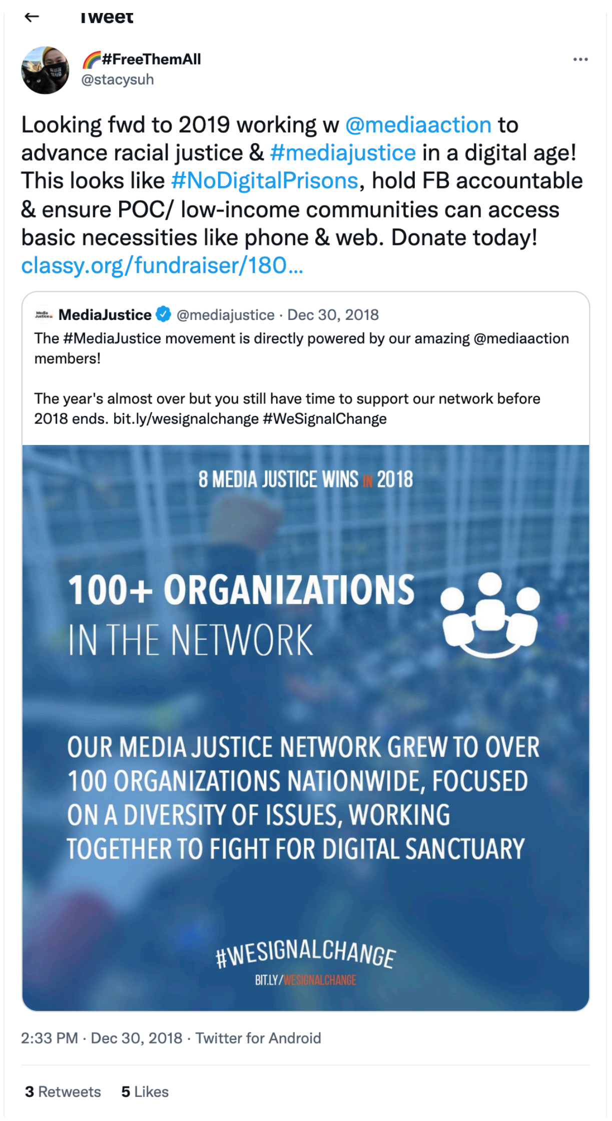
```
In [1]: wordsLst = 'the autumn is dragging its feet'.split()

In [2] {word: len(word) for word in wordsLst}

Out[2]: {'autumn': 6, 'dragging': 8, 'feet': 4, 'is': 2,
         'its': 3, 'the': 3}
```

**Important**

We can use dictionary comprehension in situations when we want to start accumulation with a complex data structure (as in the previous slide).

# Why should we care about nested data structures?

Real-world data are stored as nested data structures. Most of the content on the web is transferred from a computer to another in a format known as JSON (Javascript Object Notation), which represents dicts and lists nested in each other.



← Tweet

🌈#FreeThemAll
@stacysuh

Looking fwd to 2019 working w @mediaaction to advance racial justice & #mediajustice in a digital age! This looks like #NoDigitalPrisons, hold FB accountable & ensure POC/ low-income communities can access basic necessities like phone & web. Donate today! classy.org/fundraiser/180…

MediaJustice @mediajustice · Dec 30, 2018
The #MediaJustice movement is directly powered by our amazing @mediaaction members!

The year's almost over but you still have time to support our network before 2018 ends. bit.ly/wesignalchange #WeSignalChange

8 MEDIA JUSTICE WINS IN 2018

100+ ORGANIZATIONS IN THE NETWORK

OUR MEDIA JUSTICE NETWORK GREW TO OVER 100 ORGANIZATIONS NATIONWIDE, FOCUSED ON A DIVERSITY OF ISSUES, WORKING TOGETHER TO FIGHT FOR DIGITAL SANCTUARY

#WESIGNALCHANGE
BIT.LY/WESIGNALCHANGE

2:33 PM · Dec 30, 2018 · Twitter for Android

3 Retweets 5 Likes

Same content, but in different format.

```
{'id': 1079460557160247297,
 'source': 'Twitter for Android',
 'text': 'Looking fwd to 2019 working w @mediaaction to advance racial
justice &amp; #mediajustice in a digital age! This looks like #NoDigita
lPrisons, hold FB accountable &amp; ensure POC/ low-income communities
can access basic necessities like phone &amp; web. Donate today! http
s://t.co/9iU1jRSLmt https://t.co/d2kb2Y9EHI',
 'public_metrics': {'retweet_count': 3,
  'reply_count': 0,
  'like_count': 5,
  'quote_count': 0},
 'entities': {'urls': [{'start': 268,
    'end': 291,
    'url': 'https://t.co/9iU1jRSLmt',
    'expanded_url': 'https://www.classy.org/fundraiser/1809542',
    'display_url': 'classy.org/fundraiser/180…',
    'status': 200,
    'unwound_url': 'https://support.mediajustice.org/fundraiser/1809542
'},
   {'start': 292,
    'end': 315,
    'url': 'https://t.co/d2kb2Y9EHI',
    'expanded_url': 'https://twitter.com/mediajustice/status/1079436958
667919361',
    'display_url': 'twitter.com/mediajustice/s…'}],
  'mentions': [{'start': 30,
    'end': 42,
    'username': 'mediaaction',
    'id': '14881478'}],
  'hashtags': [{'start': 75, 'end': 88, 'tag': 'mediajustice'},
   {'start': 123, 'end': 140, 'tag': 'NoDigitalPrisons'}],
  'annotations': [{'start': 143,
    'end': 144,
    'probability': 0.66,
    'type': 'Organization',
    'normalized_text': 'FB'}]},
 'author_id': 80507653,
```

# The `json` module

Functions to read and write JSON data from / into files.

`json.load` – load JSON data from a file open for reading
`json.dump` – dump JSON data into a file open for writing

**Usage:**

```
json.load(fileObj)

json.dump(dictObj,
          fileObj)
```

```python
import json

with open("tweet.json", 'r') as inFile:
    tweetDct = json.load(inFile)

print(len(tweetDct))

with open("tweet2.json", 'w') as outFile:
    json.dump(tweetDct, outFile)


with open("tweet2.json", 'r') as inFile:
    tweetDct2 = json.load(inFile)

tweetDct == tweetDct2
```

## Challenge Problem: Manipulate JSONs

You are given a JSON file with tweets (their text and id):

```
[{'id': 1072284009122586625, 'text': 'The case of Jacob Walter Anderson from
@Baylor is the perfect amalgamation between the #MeToo and #BlackLivesMatter
movements. #ThisIsWhyWeAreAngry'},
{'id': 1071990529448075264, 'text': 'Now, that you all have some background
information to this short story, please go read it at 👉👉👉
https://t.co/KRGkjbNJbY 👈👈👈 #NoJusticeNoPeace #BlackLivesMatter
#MissionFree #DefendOurFreedom 😎'}]
```

We want to answer the following question:

- Which are the most frequently mentioned hashtags?

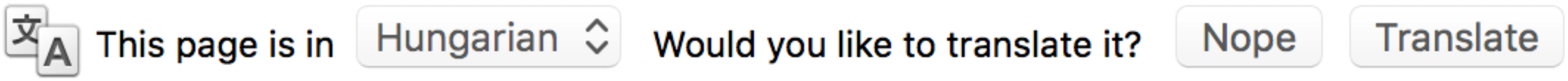
We can answer this question via Python code that makes use of the accumulation pattern with dictionaries.

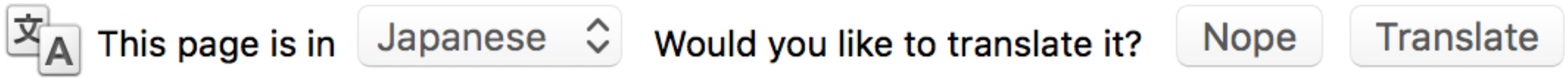
Use the notebook to answer this question (in a guided way).

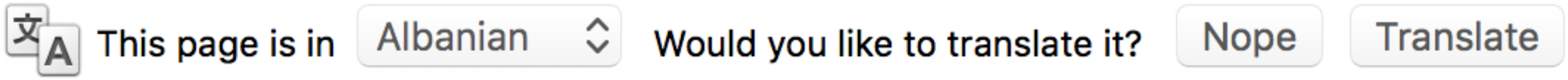
## How to use accumulation to solve a real-word problem?

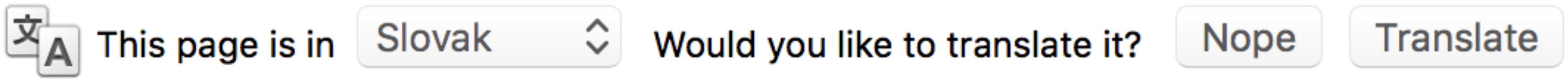
(more challenging material on accumulation)

# Real-world problem: Language Detection


This page is in Hungarian Would you like to translate it? Nope Translate


This page is in Japanese Would you like to translate it? Nope Translate


This page is in Albanian Would you like to translate it? Nope Translate


This page is in Slovak Would you like to translate it? Nope Translate

**Question:**

How would you write a computer program that takes some text as input and outputs the language in which the text was written? The Chrome browser does that all the time (see images)!

Tirana është gjithmonë një hap përpara; po mendojmë për ditët e ftohta dhe me shi, kur mund të kemi emergjenca civile. Këto janë të pashmangshme dhe pavarësisht histerisë, as kryetari i Bashkisë, as Kryeministri apo kushdo tjetër nuk mundet ta ndalojë shiun apo të rregullojë infrastrukturën e keqndërtuar ndër vite, që janë kryesisht ndërtime pa leje buzë lumenive apo në hapësira të tjera publike

「第8回日本ジオパーク全国大会男鹿半島・大潟大会」が25日、3日間の日程で秋田県の男鹿市と大潟村で開幕した。東北での全国大会開催は初めて。自治体関係者やガイド、研究者ら約千人が集まり、パネルディスカッションなどを通してジオパークの活用策を考えた。両市村と関連団体などでつくる実行委員会の主催

# How do we parse language?

A világ legnagyobb fizetési hálózatának működési bevétele az amerikai gazdasági aktivitás kétharmadát adó személyi fogyasztás folyamatos élénkségének köszönhetően 14 százalékkal 4,86 milliárd dollárra emelkedett. Az eredményben az is szerepet játszott, hogy a Visa Inc. a múlt év közepén megvásárolta a Visa Europe Ltd. céget. A működési költségek alig változtak, 1,64 milliárd dollárt tettek ki.
A Visa kártyákkal lebonyolított fizetések összege 9,8 százalékkal 1,93 ezer milliárd dollárra emelkedett, ennek 43 százaléka az Egyesült Államokra jutott.

Globálny ekonomický rast je určite slušný – rozhodne z pohľadu posledných desiatich rokov. No zároveň je aj úbohý – z pohľadu posledných tridsiatich rokov. V podstate je iba polovičný tomu, čo svet zažíval na konci osemdesiatich rokov, celé deväťdesiate roky a predkrízové roky tohto storočia. Smutnou realitou je, že svetu nestačí takto nízky rast. Svet je nastavený na viac.

## Creating language features from text

1. Looking at the character sets: Latin, Cyrillic, Greek, CJK (Chinese, Japanese, Korean), etc. can provide a first categorization into language families.
2. Looking at one-letter, two-letter or three-letter words and their frequency in a text.
3. Character n-grams and their frequency.
4. Word n-grams and their frequency.

$\rightarrow$ If we learn that the family is Latin, that doesn't solve the problem, because there are so many languages that use Latin characters.

$\rightarrow$ These are known as functional words.

$\rightarrow$ Each language might have a unique signature: a unique frequency distribution of these n-grams.

**What are n-grams?**

Given the word: "book", the character n-grams are sequences of characters with different size. Unigrams: b, o, k. Bigrams: bo, oo, ok. Trigrams: boo, ook.

Word n-grams deal with sentences. "I like red cherries" will have as bigrams: "I like", "like red", "red cherries".

N-grams are a common model for representing language in the field of Natural Language Processing (a subfield of Artificial Intelligence).

## Comparing character bigrams in different languages

| | | | | | |
|---|---|---|---|---|---|
| TH | 2.71 | EN | 1.13 | NG | 0.89 |
| HE | 2.33 | AT | 1.12 | AL | 0.88 |
| IN | 2.03 | ED | 1.08 | IT | 0.88 |
| ER | 1.78 | ND | 1.07 | AS | 0.87 |
| AN | 1.61 | TO | 1.07 | IS | 0.86 |
| RE | 1.41 | OR | 1.06 | HA | 0.83 |
| ES | 1.32 | EA | 1.00 | ET | 0.76 |
| ON | 1.32 | TI | 0.99 | SE | 0.73 |
| ST | 1.25 | AR | 0.98 | OU | 0.72 |
| NT | 1.17 | TE | 0.98 | OF | 0.71 |

Top 30 bigrams for English (%).

| | | | | | |
|---|---|---|---|---|---|
| DE | 2.57 | AD | 1.43 | TA | 1.09 |
| ES | 2.31 | AR | 1.43 | TE | 1.00 |
| EN | 2.27 | RE | 1.42 | OR | 0.98 |
| EL | 2.01 | AL | 1.33 | DO | 0.98 |
| LA | 1.80 | AN | 1.24 | IO | 0.98 |
| OS | 1.79 | NT | 1.22 | AC | 0.96 |
| ON | 1.61 | UE | 1.21 | ST | 0.95 |
| AS | 1.56 | CI | 1.15 | NA | 0.92 |
| ER | 1.52 | CO | 1.13 | RO | 0.85 |
| RA | 1.47 | SE | 1.11 | UN | 0.84 |

Top 30 bigrams for Spanish (%).

**To notice:**

- The top 3 bigrams for English cannot be found at all in the list of Spanish bigrams.
- The two lists have 14 bigrams in common out of 30 (less than half).
- The bigrams that are in common have different frequency. E.g., EN is 2.27 in Spanish and 1.13 in English.

Note: These bigrams were calculated from a large set of news stories. Because the word "the" is the most common word in English speech, that explains why the two bigrams "th" and "he" are at the top. If we use only the vocabulary of English words, the list will change. The most common bigram becomes "in", because of the many words that start with "in" or that end in "ing".

# How can we use CS111 to identify language?

**Question:** How do we build a program that identifies natural languages?
**Answer:** We create a "signature" for each known language by processing large amounts of text. This signature is composed of different features and their frequency distributions. Then, for new text, we compare its signature to that of known languages and pick the one that comes the closest.

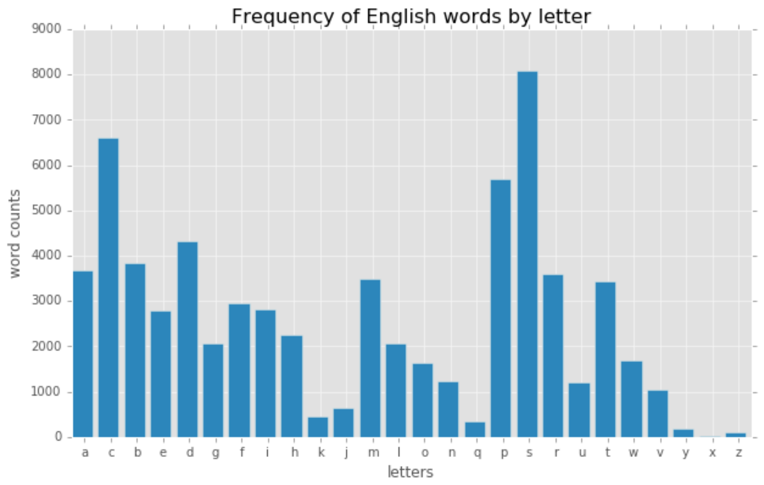
**Question:** What does this problem have to do with CS 111?
**Answer:** While we cannot build the entire program, we can create many of the features that would be part of the signature of a language.

**Scenario 1**

We are given a list of **`englishwords`**, what features can we extract from it?

- The frequency distribution of word lengths.
- The frequency distribution of character n-grams.
- The frequency distribution of words starting with a particular letter.

and many more.



# English word length distribution

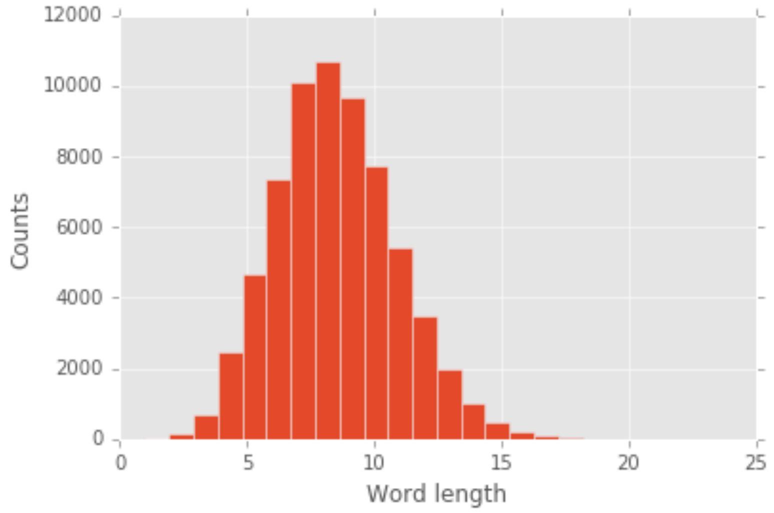
**Problem:** Given the dictionary of all English words, what is the distribution of words by length?

**Solution 1** (requires two separate loops)

1. Iterate over the list of words to find the length of each word and store it into a new list. **[Accumulation in a list via a mapping operation.]**
2. Iterate over the list of lengths and store it into a dictionary to keep track of the number of times we encounter each length. **[Accumulation via a dictionary.]**

**Solution 2** (requires one loop)

1. Iterate over the list of words to find the length of each word and immediately store it into a dictionary. **[Accumulation via a dictionary.]**



Visualization of English word length distribution. It resembles a bell curve (normal distribution) that is found often in nature.

# English word length distribution - Code

## Solution 1 (separate accumulation in two steps)

### Step 1

```
lengthsList = [len(word) for word in englishwords]
```

or

```
lengthsList = map(len, englishwords) # new function map
```

### Step 2

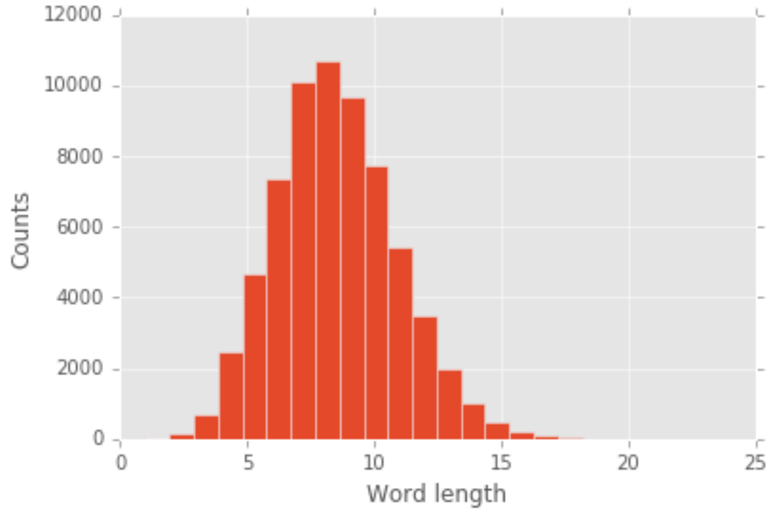
```
lengthsDct = {}
for length in lengthsList:
    lengthsDct[length] = lengthsDct.get(length, 0) + 1
```

---

## Solution 2 (one single loop accumulating into the dictionary)

```
lengthsDct2 = {}
for word in englishwords:
    length = len(word)
    lengthsDct2[length] = lengthsDct2.get(length, 0) + 1
```

## An aside: Fun with statistics



In Statistics, it is common to describe a dataset (e.g., the list of the lengths of all English words) in terms of descriptive statistics: the mean, the median, the mode (the value that occurs the most), the variance, the standard deviation, etc. All these statistics can be calculated with the operations we have been learning.

- The *mean* is the sum of all list elements divided by the length of the list. (sum =>accumulation to a number)
- The *median* is the middle element of a sorted list.
- The *mode* is the most frequent element (i.e., the max value from the frequency dictionary.)
- The *variance* is the sum of the squares of the difference of each item to the mean.
- The *standard deviation* is the square of the variance.

**Try it out**

Using the **`lengthsList`** and **`lengthsDct`** from the previous slide, you can practice calculating these statistics with Python code.

You should find that both the median and the mode are 8.

# Building character n-grams

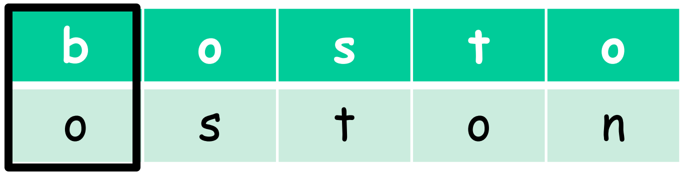
**Concepts in this slide**:
The function **zip** can be used with strings and returns a list of tuples.

Unigrams: `'b', 'o', 's', 't', 'o', 'n'`

```
word = 'boston'
list(word)
```

Bigrams: `'bo', 'os', 'st', 'to', 'on'`

```
["".join(pair) for pair in zip(word, word[1:])]
```



Trigrams: `'bos', 'ost', 'sto', 'ton'`

```
["".join(trple) for trple in zip(word, word[1:],word[2:])]
```

# The bigram frequency distribution

**Concepts in this slide**: Two functions to build the bigram frequency distribution.

```
def bigrams(word):
    """Given a word return a list of its bigrams."""
    return ["".join(pair) for pair in zip(word,
                                          word[1:])]


def createBigramFrequency():
    """Create and return the bigram frequency distribution of
    all words in ‘englishwords’.
    """
    bigramsDct = {}                # accumulator dictionary

    for word in englishwords:
        bigramsList = bigrams(word) # create ngrams as a list

        # add new bigrams or update counts of existing ones
        for ngram in bigramsList:
            bigramsDct[ngram] = bigramsDct.get(ngram, 0) + 1

    return bigramsDct
```

# N-gram frequency distributions

**Concepts in this slide**:
How to avoid multiple iterations by creating helper functions?

- There are 66230 words in **`englishwords`**. We want to avoid iterating over them too many times to create the n-gram distributions we need.
- We can create three n-gram distributions in one single loop.
- Imagine we have three functions: **`unigrams`**, **`bigrams`**, **`trigrams`** that contain the code in slide 19, respectively.
- Imagine a function **`storeNgrams`** that takes a list of n-grams and a dictionary and adds the n-grams to the dictionary with their frequency as the key.

```
unigramsDct = {}
bigramsDct = {}
trigramsDct = {}

for word in englishwords:
    # create ngrams
    ngrams1 = unigrams(word)
    ngrams2 = bigrams(word)
    ngrams3 = trigrams(word)

    # store ngrams in freq dicts
    storeNgrams(ngrams1, unigramsDct)
    storeNgrams(ngrams2, bigramsDct)
    storeNgrams(ngrams3, trigramsDct)
```

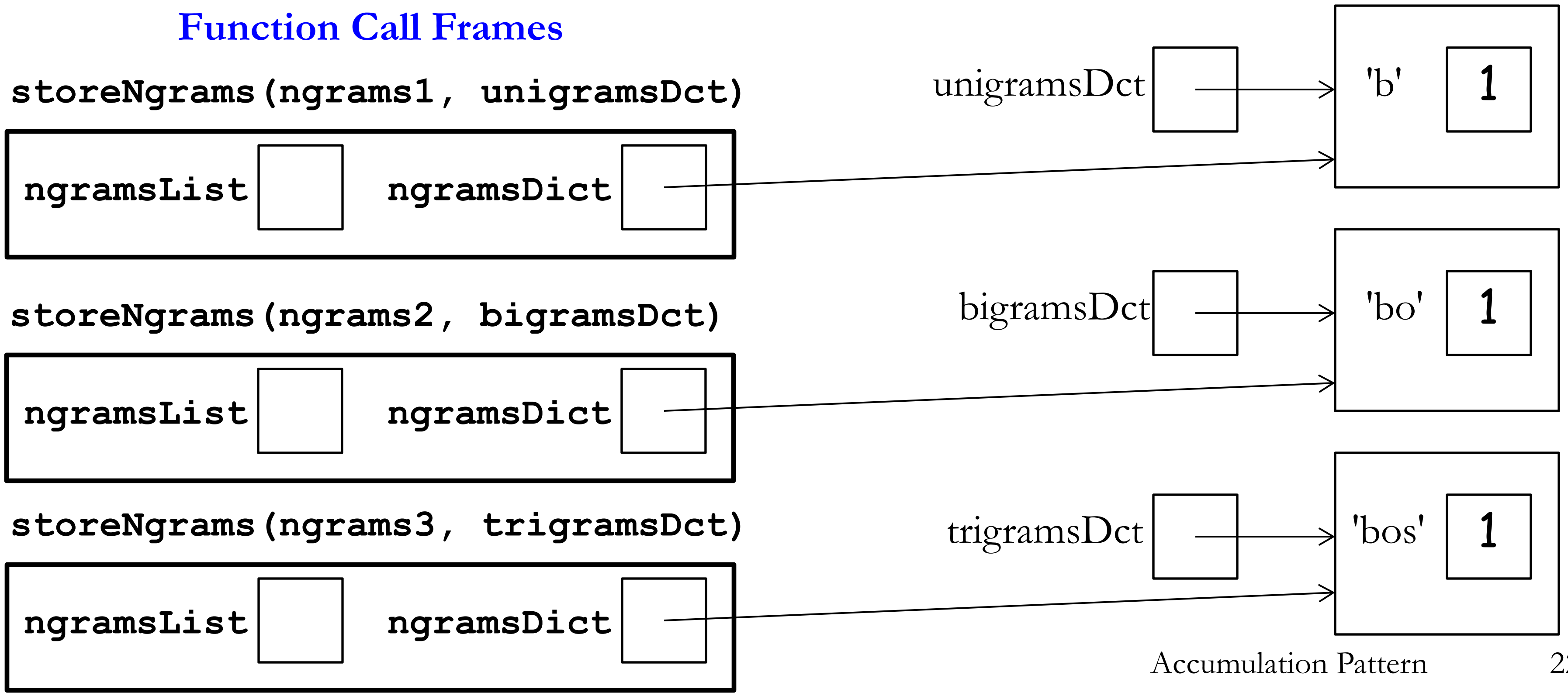
**Question**

Can you hypothesize why the function **`storeNgrams`** doesn't return a value?

## Mutating Dictionaries via aliasing

**Concepts in this slide**: A dictionary can be mutated via aliasing.

```
def storeNgrams(ngramsList, ngramsDict):
    """Given a list of items and a dictionary,
    update the counts of the dictionary keys.
    """
    for ngram in ngramsList:
        ngramsDict[ngram] = ngramsDict.get(ngram, 0) + 1
```



# Analyzing the Results

it's your turn

- Predict what will be the max lengths for the **`unigramsDct`**, **`bigramsDct`**, and **`trigramsDct`**: ____, _____, _____
- Do you expect that all dictionaries will have that max length? Explain.
- Predict the top 3 unigrams, top 3 bigrams, and top 3 unigrams.
- How to write a function **`sortItemsInFreqDict`** that given a frequency dictionary will return the sorted list (in descending order) of its items, based on the **value** of each (key/value) item?
- Which will be more frequent (have the highest values): the top unigrams, the top bigrams, or the top trigrams?

## Accumulating in a dictionary of dictionaries

**Problem:** How can we create a dictionary that has two level of keys? In the first level, each key is a unigram, in the second level the keys are bigrams that start with the unigram. [See example on the right.]

**Solution 1**: Assume we already have **`bigramsDict`**:

```
{'a': {'aa': 19,
       'ab': 1665,
       'ac': 2387,
       'ad': 1685,
       ...},

 'b': {'ba': 1431,
       'bb': 417,
       'bc': 25,
       'bd': 35,
       ...},
  ...
}
```

```
from string import ascii_lowercase as lowercase
# 'abcdefghijklmnopqrstuvwxyz'

# create the dict with unigrams as keys and empty dict as values
bigramsByFirstLetter = {char: {} for char in lowercase}

for bigram in bigramsDct:
    unigram = bigram[0]
    # assign the second level of keys
    bigramsByFirstLetter[unigram][bigram] = bigramsDct[bigram]
```

## Accumulating in a dictionary of dictionaries [2]

**Solution 2**: We don't have bigrams, we create them as we iterate over the list of words.

```
from string import ascii_lowercase as lowercase
# 'abcdefghijklmnopqrstuvwxyz'

# create the dict with unigrams as keys and empty dict as values
bigramsByFirstLetter = {char: {} for char in lowercase}

for word in englishwords:
    # create list of bigrams from word
    bigramsLst = bigrams(word)
    # iterate over bigrams
    for bigram in bigramsLst:
        unigram = bigram[0]
        # access the nested bigram dict for easy reference
        bigramsDct = bigramsByFirstLetter[unigram]
        # increase frequency counter
        bigramsDct[bigram] = biagramsDct.get(bigram, 0) + 1
```

## Accumulating in a dictionary of lists.

**Problem:** Group words from **`englishwords`** based on their ending: words ending with 'ed', 'ly', 'es', etc.

**Solution Algorithm**:

1. Create an empty dictionary
2. Iterate over words and get the ending of each word.
3. Check to see if the key/value for ending is already in the dictionary using the method **`get`** with the default value an empty list.
4. Append the word to the list associated with its ending.

```
{'ed': ['abandoned',
        'abased',
        'abashed',
        'abated',
        ...],

 'ly': ['abjectly',
        'ably',
        'abnormally',
        'abominably',
        ...],
 'es': ['abacuses',
        'abases',
        'abashes',
        'abates',
        ...],
 ...
}
```

```
wordsByEnding = {}
for word in englishwords:
    ending = word[-2:]
    wordsByEnding[ending] = wordsByEnding.get(ending, [])
    wordsByEnding[ending].append(word)
```

## Summary

1. Lists and dictionaries are powerful data structures that are used routinely to perform complex data analysis tasks such as transforming data from one form to another.
2. Accumulation is a very common pattern in problem solving: we accumulate frequencies (counts) as we encounter new data; or we organize data as nested dictionaries of dictionaries or dictionaries of lists.
3. When we need to accumulate into nested structures, first **always draw a picture** of what the structure would look like, in order to visualize what needs to be created through code.
4. Dictionaries are mutable and they can be changed via aliasing (two different variables pointing to the same dictionary object).
5. The nested structures would need double subscripting operations (e.g., see last statement in slide 16-16). If this is conceptually difficult, you can store the inner structure into a temporary variable and work with that instead. Because of aliasing, this temporary variable will be directly mutating the entire dictionary. [See second from last statement in slide 19.]
6. Use a **dictionary comprehension** statement whenever you need to create a dictionary of dictionaries or a dictionary of lists in the case when the keys of the outer dictionary are known.