Accumulation Pattern for Lists and Dictionaries

CS111 Computer Programming

Department of Computer Science
Wellesley College

Tirana éshte gírhomnë një hap përpasa; po mendojmë për ditët e frohta dhe me shi, kur mund të kemi emergjencë civile. Këto janë të pashmangshme dhe pavarësisht histerisë, as kryetari i Bashkisë, as Kryeministri apo kushtdo tjetër nuk mundet ta ndalojë shien apo të rregullojë infrastrukturën e keqndërtuar ndër vite, që janë kryesisht ndërtme pa leje buzë lumenoje apo në hapësira të tjera publike.

A világ legnagyobb fizerést hálózatának működési bevétele az amerikai gazdasági aktivitás kétvégéért adó személyi fogyasztás folyamatos élségének köszönhetően 14 százalékkal 4,86 milliárd dollárra emelkedett. Az eredményben az is szerepet játszott, hogy a Visa Inc. a múlt év közepén megnövelt a Visa Europe Ltd. céget. A működési költségek alig változnak, 1,64 milliárd dollárt tettek kí.

A Visa kártyákkal lebonyolított fizeretek összege 9,8 százalékkal 1,93 ezer milliárd dollárra emelkedett, ennek 43 százaléka az Egyesült Államokra junott.

「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
大潟村で開催した。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。

Accumulation Pattern for Lists and Dictionaries

Creating 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.

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

These are known as functional words.

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

What are n-grams?


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

<table>
<thead>
<tr>
<th>Top 30 bigrams for English (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TH : 2.71</td>
</tr>
<tr>
<td>HE : 2.33</td>
</tr>
<tr>
<td>IN : 2.03</td>
</tr>
<tr>
<td>ER : 1.78</td>
</tr>
<tr>
<td>AN : 1.61</td>
</tr>
<tr>
<td>RE : 1.41</td>
</tr>
<tr>
<td>ES : 1.32</td>
</tr>
<tr>
<td>ON : 1.32</td>
</tr>
<tr>
<td>ST : 1.25</td>
</tr>
<tr>
<td>NT : 1.17</td>
</tr>
</tbody>
</table>

Top 30 bigrams for Spanish (%).

<table>
<thead>
<tr>
<th>Top 30 bigrams for Spanish (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE : 2.57</td>
</tr>
<tr>
<td>ES : 2.31</td>
</tr>
<tr>
<td>EN : 2.27</td>
</tr>
<tr>
<td>EL : 2.01</td>
</tr>
<tr>
<td>LA : 1.80</td>
</tr>
<tr>
<td>OS : 1.79</td>
</tr>
<tr>
<td>ON : 1.61</td>
</tr>
<tr>
<td>AS : 1.56</td>
</tr>
<tr>
<td>ER : 1.52</td>
</tr>
<tr>
<td>RA : 1.47</td>
</tr>
</tbody>
</table>

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”.

Accumulation Pattern 15-5

Back to CS 111

Question: How to 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.

Scenario 1
We are given the englishwords list from PS06, 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.

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.

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 mapping]
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]

English word length distribution - Code

Solution 1 (separate accumulation in two steps)

Step 1
```python
lengthsList = [len(word) for word in englishwords]
```

or
```
lengthsList = map(len, englishwords)
```

Step 2
```python
lengthsDct = {}
for length in lengthsList:
    lengthsDct[length] = lengthsDct.get(length, 0) + 1
```

Solution 2 (one single loop accumulating into the dictionary)
```python
lengthsDct2 = {}
for word in englishwords:
    length = len(word)
    lengthsDct2[length] = lengthsDct2.get(length, 0) + 1
```
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.

```python
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 bigram in bigramsList:
            if bigram in bigramsDct:
                bigramsDct[bigram] = bigramsDct[bigram] + 1
            else:
                bigramsDct[bigram] = 1
    return bigramsDct
```

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

for word in englishwords:
    ngrams1 = unigrams(word)
    ngrams2 = bigrams(word)
    ngrams3 = trigrams(word)
    # store ngrams in freq dicts
    storeNgrams(ngrams1, unigramsDct)
    storeNgrams(ngrams2, bigramsDct)
    storeNgrams(ngrams3, trigramsDct)
```

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.
Mutating Dictionaries via aliasing

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

```python
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
```

Function Call Frames
storeNgrams(ngrams1, unigramsDct)
ngramsList ngramsDict
unigramsDct

storeNgrams(ngrams2, bigramsDct)
ngramsList ngramsDict
bigramsDct

storeNgrams(ngrams3, trigramsDct)
ngramsList ngramsDict
trigramsDct

Analyzing the Results

- 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? Hint: use `lambda`.
- Which will be more frequent (have the highest values): the top unigrams, the top bigrams, or the top trigrams?

Dictionary 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.

```python
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 (see next slide).

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`:

```python
{'a': {'aa': 19, 'ab': 1665, 'ac': 2387, 'ad': 1685, ...},
 'b': {'ba': 1431, 'bb': 417, 'bc': 25, 'bd': 35, ...},
 ...}
```

```python
from string import 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]
```
Solution 2: We don’t have bigrams, we create them as we iterate over the list of words.

```python
from string import lowercase

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

for word in englishwords:
    # create list of bigrams from word
    bigramsList = bigrams(word)
    # iterate over bigrams
    for bigram in bigramsList:
        unigram = bigram[0]
        # access the nested bigram dict for easy reference
        bigramsDct = biagramsByFirstLetter[unigram]
        # increase frequency counter
        bigramsDct[bigram] = bigramsDct.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.

```python
wordsByEnding = {}

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

New collection type: `set`

Sets are collections that have a unique number of items.
They are useful to find the number of unique values in lists.

Find unigrams in a phrase:

```python
set([char for char in list(phrase)])
```

In [3]: phrase = 'the autumn is dragging its feet'

In [4]: set([char for char in list(phrase)])

Out[4]: {' ', 'a', 'd', 'e', 'f', 'g', 'h', 'i', 'm', 'n', 'r', 's', 't', 'u'}

In [5]: set([1, 2, 1, 2, 1, 2])

Out[5]: {1, 2}

In [6]: empty = set([]) # don't use empty = {}

In [7]: empty.add(3) # add for sets is like append for lists

In [8]: empty

Out[8]: {3}

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 15-16). 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 15-17.]
6. Use a dictionary comprehension statement whenever you need to create a dictionary of dictionaries or a dictionary of lists in the case that the keys of the outer dictionary are known.
7. The new collection type is the `set`. To create a set, always use the syntax `set([])`. Sets are also mutable, you can grow them by using the method `add`. 