A First Exercise in Natural Language Processing with Python: Counting Hapaxes

A first exercise

Counting hapaxes (words which occur only once in a text or corpus) is an easy enough problem that makes use of both simple data structures and some fundamental tasks of natural language processing (NLP): tokenization (dividing a text into words), stemming, and part-of-speech tagging for lemmatization. For that reason it makes a good exercise to get started with NLP in a new language or library.

As a first exercise in implementing NLP tasks with Python, then, we'll write a script which outputs the count and a list of the hapaxes in the following paragraph (our script can also be run on an arbitrary input file). You can follow along, or try it yourself and then compare your solution to mine.

```
Cory Linguist, a cautious corpus linguist, in creating
a corpus of courtship correspondence, corrupted a
crucial link. Now, if Cory Linguist, a careful corpus
```

```
linguist, in creating a corpus of courtship
correspondence, corrupted a crucial link, see that
YOU, in creating a corpus of courtship correspondence,
corrupt not a crucial link.
```

To keep things simple, ignore punctuation and case. To make things complex, count hapaxes in all three of word form, stemmed form, and lemma form. The final program (hapaxes.py) is listed at the end of this post. The sections below walk through it in detail for the beginning NLP/Python programmer.

Natural language processing with Python

There are several NLP packages available to the Python programmer. The most well-known is the Natural Language Toolkit (NLTK), which is the subject of the popular book *Natural Language Processing with Python* by Bird et al. NLTK has a focus on education/research with a rather sprawling API. Pattern is a Python package for datamining the WWW which includes submodules for language processing and machine learning. Polyglot is a language library focusing on "massive multilingual applications." Many of its features support over 100 languages (but it doesn't seem to have a stemmer or lemmatizer builtin). And there is Matthew Honnibal's spaCy, an "industrial strength" NLP library focused on performance and integration with machine learning models.

If you don't already know which library you want to use, I

recommend starting with NLTK because there are so many online resources available for it. The program presented below actually presents five solutions to counting hapaxes, which will hopefully give you a feel for a few of the libraries mentioned above:

- Word forms - counts unique spellings (normalized for case). This uses plain Python (no NLP packages required)
- NLTK stems - counts unique stems using a stemmer provided by NLTK
- NLTK lemmas - counts unique lemma forms using NLTK's part of speech tagger and interface to the WordNet lemmatizer
- spaCy lemmas - counts unique lemma forms using the spaCy NLP package

Installation

This tutorial assumes you already have Python installed on your system and have some experience using the interpreter. I recommend referring to each package's project page for installation instructions, but here is one way using pip. As explained below, each of the NLP packages are optional; feel free to install only the ones you're interested in playing with.

```
# Install NLTK:
$ pip install nltk

# Download reqed NLTK data packages
$ python -c 'import nltk; nltk.download("wordnet");
nltk.download("averaged_perceptron_tagger");
```

```
nltk.download("omw-1.4")'

# install spaCy:
$ pip install spacy

# install spaCy en model:
$ python -m spacy download en_core_web_sm
```

Optional dependency on Python modules

It would be nice if our script didn't depend on any particular NLP package so that it could still run even if one or more of them were not installed (using only the functionality provided by whichever packages are installed).

One way to implement a script with optional package dependencies in Python is to try to import a module, and if we get an `ImportError` exception we mark the package as uninstalled (by setting a variable with the module's name to `None`) which we can check for later in our code:

[hapaxes.py: 63-98]

```
### Imports
#
# Import some Python 3 features to use in Python 2
from __future__ import print_function
from __future__ import unicode_literals

# gives us access to command-line arguments
import sys
```

```
# The Counter collection is a convenient layer on top
of
# python's standard dictionary type for counting
iterables.
from collections import Counter

# The standard python regular expression module:
import re

try:
    # Import NLTK if it is installed
    import nltk

    # This imports NLTK's implementation of the
Snowball
    # stemmer algorithm
    from nltk.stem.snowball import SnowballStemmer

    # NLTK's interface to the WordNet lemmatizer
    from nltk.stem.wordnet import WordNetLemmatizer
except ImportError:
    nltk = None
    print("NLTK is not installed, so we won't use it.
")

try:
    # Import spaCy if it is installed
    import spacy
except ImportError:
    spacy = None
    print("spaCy is not installed, so we won't use
it.")
```

Tokenization

Tokenization is the process of splitting a string into lexical 'tokens' — usually words or sentences. In languages with space-separated words, satisfactory tokenization can often be accomplished with a few simple rules, though ambiguous punctuation can cause errors (such as mistaking a period after an abbreviation as the end of a sentence). Some tokenizers use statistical inference (trained on a corpus with known token boundaries) to recognize tokens.

In our case we need to break the text into a list of words in order to find the hapaxes. But since we are not interested in punctuation or capitalization, we can make tokenization very simple by first normalizing the text to lower case and stripping out every punctuation symbol:

[hapaxes.py: 100-119]

```
def normalize_tokenize(string):
    """
    Takes a string, normalizes it (makes it lowercase
and
    removes punctuation), and then splits it into a
list of
    words.

    Note that everything in this function is plain
Python
    without using NLTK (although as noted below, NLTK
provides
    some more sophisticated tokenizers we could have
used).
```

```
    """
    # make lowercase
    norm = string.lower()

    # remove punctuation
    norm = re.sub(r'(?u)[^\w\s]', '', norm) ①

    # split into words
    tokens = norm.split()

    return tokens
```

① Remove punctuation by replacing everything that is not a word (\w) or whitespace (\s) with an empty string. The (?u) flag at the beginning of the regex enables unicode matching for the \w and \s character classes in Python 2 (unicode is the default with Python 3).

Our tokenizer produces output like this:

```
>>> normalize_tokenize("This is a test sentence of
white-space separated words.")
['this', 'is', 'a', 'test', 'sentence', 'of',
'whitespace', 'separated', 'words']
```

Instead of simply removing punctuation and then splitting words on whitespace, we could have used one of the tokenizers provided by NLTK. Specifically the `word_tokenize()` method, which first splits the text into sentences using a pre-trained English sentences tokenizer (`sent_tokenize`), and then finds words using regular expressions in the style of the Penn Treebank tokens.

```
# We could have done it this way (requires the
# 'punkt' data package):
from nltk.tokenize import word_tokenize
tokens = word_tokenize(norm)
```

The main advantage of `word_tokenize()` is that it will turn contractions into separate tokens. But using Python's standard `split()` is good enough for our purposes.

Counting word forms

We can use the tokenizer defined above to get a list of words from any string, so now we need a way to count how many times each word occurs. Those that occur only once are our word-form hapaxes.

[hapaxes.py: 121-135]

```
def word_form_hapaxes(tokens):
    """
    Takes a list of tokens and returns a list of the
    wordform hapaxes (those wordforms that only appear
once)

    For wordforms this is simple enough to do in plain
    Python without an NLP package, especially using
the Counter
    type from the collections module (part of the
Python
    standard library).
    """
```

```
    counts = Counter(tokens) ①
    hapaxes = [word for word in counts if counts[word]
== 1] ②

    return hapaxes
```

① Use the convenient `Counter` class from Python's standard library to count the occurrences of each token. `Counter` is a subclass of the standard `dict` type; its constructor takes a list of items from which it builds a dictionary whose keys are elements from the list and whose values are the number of times each element appeared in the list.

② This list comprehension creates a list from the Counter dictionary containing only the dictionary keys that have a count of 1. These are our hapaxes.

Stemming and Lemmatization

If we use our two functions to first tokenize and then find the hapaxes in our example text, we get this output:

```
>>> text = "Cory Linguist, a cautious corpus linguist,
in creating a corpus of courtship correspondence,
corrupted a crucial link. Now, if Cory Linguist, a
careful corpus linguist, in creating a corpus of
courtship correspondence, corrupted a crucial link,
see that YOU, in creating a corpus of courtship
correspondence, corrupt not a crucial link."
>>> tokens = normalize_tokenize(text)
>>> word_form_hapaxes(tokens)
['now', 'not', 'that', 'see', 'if', 'corrupt', 'you',
```

```
 'careful', 'cautious']
```

Notice that ‘corrupt’ is counted as a hapax even though the text also includes two instances of the word ‘corrupted’. That is expected because ‘corrupt’ and ‘corrupted’ are different word-forms, but if we want to count word roots regardless of their inflections we must process our tokens further. There are two main methods we can try:

- Stemming uses an algorithm (and/or a lookup table) to remove the suffix of tokens so that words with the same base but different inflections are reduced to the same form. For example: ‘argued’ and ‘arguing’ are both stemmed to ‘argu’.
- Lemmatization reduces tokens to their lemmas, their canonical dictionary form. For example, ‘argued’ and ‘arguing’ are both lemmatized to ‘argue’.

Stemming with NLTK

In 1980 Martin Porter published a stemming algorithm which has become a standard way to stem English words. His algorithm was implemented so many times, and with so many errors, that he later created a programming language called Snowball to help clearly and exactly define stemmers. NLTK includes a Python port of the Snowball implementation of an improved version of Porter’s original stemmer:

[hapaxes.py: 137-153]

```
def nltk_stem_hapaxes(tokens):
    """
    Takes a list of tokens and returns a list of the
```

```
word
    stem hapaxes.
    """
    if not nltk: ①
        # Only run if NLTK is loaded
        return None

    # Apply NLTK's Snowball stemmer algorithm to
tokens:
    stemmer = SnowballStemmer("english")
    stems = [stemmer.stem(token) for token in tokens]

    # Filter down to hapaxes:
    counts = nltk.FreqDist(stems) ②
    hapaxes = counts.hapaxes() ③
    return hapaxes
```

① Here we check if the `nltk` module was loaded; if it was not (presumably because it is not installed), we return without trying to run the stemmer.

② NLTK's `FreqDist` class subclasses the `Counter` container type we used above to count word-forms. It adds some methods useful for calculating frequency distributions.

③ The `FreqDist` class also adds a `hapaxes()` method, which is implemented exactly like the list comprehension we used to count word-form hapaxes.

Running `nltk_stem_hapaxes()` on our tokenized example text produces this list of stem hapaxes:

```
>>> nltk_stem_hapaxes(tokens)
['now', 'cautious', 'that', 'not', 'see', 'you', '
```

```
care', 'if']
```

Notice that ‘corrupt’ is no longer counted as a hapax (since it shares a stem with ‘corrupted’), and ‘careful’ has been stemmed to ‘care’.

Lemmatization with NLTK

NLTK provides a lemmatizer (the `WordNetLemmatizer` class in nltk.stem.wordnet) which tries to find a word’s lemma form with help from the WordNet corpus (which can be downloaded by running `nltk.download()` from an interactive python prompt — refer to “Installing NLTK Data” for general instructions).

In order to resolve ambiguous cases, lemmatization usually requires tokens to be accompanied by part-of-speech tags. For example, the word lemma for *rose* depends on whether it is used as a noun or a verb:

```
>>> lemmer = WordNetLemmatizer()
>>> lemmer.lemmatize('rose', 'n') # tag as noun
'rose'
>>> lemmer.lemmatize('rose', 'v') # tag as verb
'rise'
```

Since we are operating on untagged tokens, we’ll first run them through an automated part-of-speech tagger provided by NLTK (it uses a pre-trained perceptron tagger originally by Matthew Honnibal: “A Good Part-of-Speech Tagger in about 200 Lines of Python”). The tagger requires the training data available in the

'averaged_perceptron_tagger.pickle' file which can be downloaded by running `nltk.download()` from an interactive python prompt.

[hapaxes.py: 155-176]

```
def nltk_lemma_hapaxes(tokens):
    """
    Takes a list of tokens and returns a list of the
lemma
    hapaxes.
    """
    if not nltk:
        # Only run if NLTK is loaded
        return None

    # Tag tokens with part-of-speech:
    tagged = nltk.pos_tag(tokens) ①

    # Convert our Treebank-style tags to WordNet-style
tags.
    tagged = [(word, pt_to_wn(tag))
                    for (word, tag) in tagged] ②

    # Lemmatize:
    lemmer = WordNetLemmatizer()
    lemmas = [lemmer.lemmatize(token, pos)
                    for (token, pos) in tagged] ③

    return nltk_stem_hapaxes(lemmas) ④
```

① This turns our list of tokens into a list of 2-tuples: `[(token1, tag1), (token2, tag2)⋯]`

② We must convert between the tags returned by `pos_tag()` and the tags expected by the WordNet lemmatizer. This is done by applying the `pt_to_wn()` function (defined below) to each tag.

③ Pass each token and POS tag to the WordNet lemmatizer.

④ If a lemma is not found for a token, then it is returned from `lemmatize()` unchanged. To ensure these unhandled words don't contribute spurious hapaxes, we pass our lemmatized tokens through the word stemmer for good measure (which also filters the list down to only hapaxes).

As noted above, the tags returned by `pos_tag()` are Penn Treebank style tags while the WordNet lemmatizer uses its own tag set (defined in the `nltk.corpus.reader.wordnet` module, though that is not very clear from the NLTK documentation). The `pt_to_wn()` function converts Treebank tags to the tags required for lemmatization:

[hapaxes.py: 178-209]

```
def pt_to_wn(pos):
    """
    Takes a Penn Treebank tag and converts it to an
    appropriate WordNet equivalent for lemmatization.

    A list of Penn Treebank tags is available at:

https://www.ling.upenn.edu/courses/Fall_2003/ling001/p
enn_treebank_pos.html
    """

    from nltk.corpus.reader.wordnet import NOUN, VERB,
ADJ, ADV
```

```
    pos = pos.lower()

    if pos.startswith('jj'):
        tag = ADJ
    elif pos == 'md':
        # Modal auxiliary verbs
        tag = VERB
    elif pos.startswith('rb'):
        tag = ADV
    elif pos.startswith('vb'):
        tag = VERB
    elif pos == 'wrb':
        # Wh-adverb (how, however, whence,
whenever...)
        tag = ADV
    else:
        # default to NOUN
        # This is not strictly correct, but it is good
        # enough for lemmatization.
        tag = NOUN

    return tag
```

Finding hapaxes with spaCy

Unlike the NLTK API, spaCy is designed to tokenize, parse, and tag a text all by calling the single function returned by `spacy.load()`. The spaCy parser returns a 'document' object which contains all the tokens, their lemmas, etc. According to the spaCy documentation, "Lemmatization is performed using the WordNet data, but extended to also cover closed-class words

such as pronouns." The function below shows how to find the lemma hapaxes in a spaCy document.

spaCy's models load quite a bit of data from disk which can cause script startup to be slow making it more suitable for long-running programs than for one-off scripts like ours.

[hapaxes.py: 211-234]

```
def spacy_hapaxes(rawtext):
    """
    Takes plain text and returns a list of lemma
hapaxes using
    the spaCy NLP package.
    """
    if not spacy:
        # Only run if spaCy is installed
        return None

    # Load the English spaCy parser
    spacy_parse = spacy.load('en_core_web_sm')

    # Tokenize, parse, and tag text:
    doc = spacy_parse(rawtext)

    lemmas = [token.lemma_ for token in doc
            if not token.is_punct and not token
.is_space] ①

    # Now we can get a count of every lemma:
    counts = Counter(lemmas) ②

    # We are interested in lemmas which appear only
```

```
once
    hapaxes = [lemma for lemma in counts if counts
[lemma] == 1]
    return hapaxes
```

① This list comprehension collects the lemma form (`token.lemma_` of all tokens in the spaCy document which are not punctuation (`token.is_punct`) or white space (`token.is_space`).

② An alternative way to do this would be to first get a count of lemmas using the `count_by()` method of a spaCy document, and then filtering out punctuation if desired: `counts = doc.count_by(spacy.attrs.LEMMA)` (but then you'd have to map the resulting attributes (integers) back to words by looping over the tokens and checking their `orth` attribute).

Make it a script

You can play with the functions we've defined above by typing (copy-and-pasting) them into an interactive Python session. If we save them all to a file, then that file is a Python module which we could `import` and use in a Python script. To use a single file as both a module and a script, our file can include a construct like this:

```
if __name__ == "__main__":
    # our script logic here
```

This works because when the Python interpreter executes a script (as opposed to importing a module), it sets the top-level variable __name__ equal to the string "__main__" (see also: What

does if __name__ == "__main__": do?).

In our case, our script logic consists of reading any input files if given, running all of our hapax functions, then collecting and displaying the output. To see how it is done, scroll down to the full program listing below.

Running it

To run the script, first download and save hapaxes.py. Then:

```
$ python hapaxes.py
```

Depending on which NLP packages you have installed, you should see output like:

```
               Count
     Wordforms   9
    NLTK-stems   8
   NLTK-lemmas   8
         spaCy   8

-- Hapaxes --
Wordforms:    careful, cautious, corrupt, if, not,
now, see, that, you
NLTK-stems:   care, cautious, if, not, now, see, that,
you
NLTK-lemmas:  care, cautious, if, not, now, see, that,
you
spaCy:        careful, cautious, if, not, now, see,
that, you
```

Try also running the script on an arbitrary input file:

```
$ python hapaxes.py somefilename

# run it on itself and note that
# source code doesn't give great results:
$ python hapaxes.py hapaxes.py
```

hapaxes.py listing

The entire script is listed below and available at hapaxes.py.

hapaxes.py

```
"""
A sample script/module which demonstrates how to count
hapaxes (tokens which
appear only once) in an untagged text corpus using
plain python, NLTK, and
spaCy. It counts and lists hapaxes in five different
ways:

    * Wordforms - counts unique spellings (normalized
for case). This uses
    plain Python (no NLTK required)

    * NLTK stems - counts unique stems using a stemmer
provided by NLTK

    * NLTK lemmas - counts unique lemma forms using
NLTK's part of speech
    * tagger and interface to the WordNet lemmatizer.
```

```
    * spaCy lemmas - counts unique lemma forms using
the spaCy NLP module.

Each of the NLP modules (nltk, spaCy) are optional; if
one is not
installed then its respective hapax-counting method
will not be run.

Usage:

    python hapaxes.py [file]

If 'file' is given, its contents are read and used as
the text in which to
find hapaxes. If 'file' is omitted, then a test text
will be used.

Example:

Running this script with no arguments:

    python hapaxes.py

Will process this text:

    Cory Linguist, a cautious corpus linguist, in
creating a corpus of
    courtship correspondence, corrupted a crucial
link. Now, if Cory Linguist,
    a careful corpus linguist, in creating a corpus of
courtship
    correspondence, corrupted a crucial link, see that
YOU, in creating a
```

```
    corpus of courtship correspondence, corrupt not a
crucial link.

And produce this output:

                Count
         Wordforms   9
             Stems   8
            Lemmas   8
             spaCy   8

    -- Hapaxes --
    Wordforms:    careful, cautious, corrupt, if, not,
now, see, that, you
    NLTK-stems:   care, cautious, if, not, now, see,
that, you
    NLTK-lemmas:  care, cautious, if, not, now, see,
that, you
    spaCy:        careful, cautious, if, not, now,
see, that, you


Notice that the stems and lemmas methods do not count
"corrupt" as a hapax
because it also occurs as "corrupted". Notice also
that "Linguist" is not
counted as the text is normalized for case.

See also the Wikipedia entry on "Hapex legomenon"
(https://en.wikipedia.org/wiki/Hapax_legomenon)
"""

### Imports
#
```

```
# Import some Python 3 features to use in Python 2
from __future__ import print_function
from __future__ import unicode_literals

# gives us access to command-line arguments
import sys

# The Counter collection is a convenient layer on top
of
# python's standard dictionary type for counting
iterables.
from collections import Counter

# The standard python regular expression module:
import re

try:
    # Import NLTK if it is installed
    import nltk

    # This imports NLTK's implementation of the
Snowball
    # stemmer algorithm
    from nltk.stem.snowball import SnowballStemmer

    # NLTK's interface to the WordNet lemmatizer
    from nltk.stem.wordnet import WordNetLemmatizer
except ImportError:
    nltk = None
    print("NLTK is not installed, so we won't use it.
")

try:
    # Import spaCy if it is installed
```

```
    import spacy
except ImportError:
    spacy = None
    print("spaCy is not installed, so we won't use
it.")

def normalize_tokenize(string):
    """
    Takes a string, normalizes it (makes it lowercase
and
    removes punctuation), and then splits it into a
list of
    words.

    Note that everything in this function is plain
Python
    without using NLTK (although as noted below, NLTK
provides
    some more sophisticated tokenizers we could have
used).
    """
    # make lowercase
    norm = string.lower()

    # remove punctuation
    norm = re.sub(r'(?u)[^\w\s]', '', norm) # <1>

    # split into words
    tokens = norm.split()

    return tokens

def word_form_hapaxes(tokens):
    """
```

```
    Takes a list of tokens and returns a list of the
    wordform hapaxes (those wordforms that only appear
once)

    For wordforms this is simple enough to do in plain
    Python without an NLP package, especially using
the Counter
    type from the collections module (part of the
Python
    standard library).
    """

    counts = Counter(tokens) # <1>
    hapaxes = [word for word in counts if counts[word]
== 1] # <2>

    return hapaxes

def nltk_stem_hapaxes(tokens):
    """
    Takes a list of tokens and returns a list of the
word
    stem hapaxes.
    """
    if not nltk: # <1>
        # Only run if NLTK is loaded
        return None

    # Apply NLTK's Snowball stemmer algorithm to
tokens:
    stemmer = SnowballStemmer("english")
    stems = [stemmer.stem(token) for token in tokens]

    # Filter down to hapaxes:
```

```
    counts = nltk.FreqDist(stems) # <2>
    hapaxes = counts.hapaxes() # <3>
    return hapaxes

def nltk_lemma_hapaxes(tokens):
    """
    Takes a list of tokens and returns a list of the
lemma
    hapaxes.
    """
    if not nltk:
        # Only run if NLTK is loaded
        return None

    # Tag tokens with part-of-speech:
    tagged = nltk.pos_tag(tokens) # <1>

    # Convert our Treebank-style tags to WordNet-style
tags.
    tagged = [(word, pt_to_wn(tag))
                    for (word, tag) in tagged] # <2>

    # Lemmatize:
    lemmer = WordNetLemmatizer()
    lemmas = [lemmer.lemmatize(token, pos)
                    for (token, pos) in tagged] # <3>

    return nltk_stem_hapaxes(lemmas) # <4>

def pt_to_wn(pos):
    """
    Takes a Penn Treebank tag and converts it to an
    appropriate WordNet equivalent for lemmatization.
```

```
    A list of Penn Treebank tags is available at:

https://www.ling.upenn.edu/courses/Fall_2003/ling001/p
enn_treebank_pos.html
    """

    from nltk.corpus.reader.wordnet import NOUN, VERB,
ADJ, ADV

    pos = pos.lower()

    if pos.startswith('jj'):
        tag = ADJ
    elif pos == 'md':
        # Modal auxiliary verbs
        tag = VERB
    elif pos.startswith('rb'):
        tag = ADV
    elif pos.startswith('vb'):
        tag = VERB
    elif pos == 'wrb':
        # Wh-adverb (how, however, whence,
whenever...)
        tag = ADV
    else:
        # default to NOUN
        # This is not strictly correct, but it is good
        # enough for lemmatization.
        tag = NOUN

    return tag

def spacy_hapaxes(rawtext):
    """
```

```
    Takes plain text and returns a list of lemma
hapaxes using
    the spaCy NLP package.
    """
    if not spacy:
        # Only run if spaCy is installed
        return None

    # Load the English spaCy parser
    spacy_parse = spacy.load('en_core_web_sm')

    # Tokenize, parse, and tag text:
    doc = spacy_parse(rawtext)

    lemmas = [token.lemma_ for token in doc
            if not token.is_punct and not token
.is_space] # <1>

    # Now we can get a count of every lemma:
    counts = Counter(lemmas) # <2>

    # We are interested in lemmas which appear only
once
    hapaxes = [lemma for lemma in counts if counts
[lemma] == 1]
    return hapaxes

if __name__ == "__main__":
    """
    The code in this block is run when this file is
executed as a script (but
    not if it is imported as a module by another
Python script).
    """
```

```
    # If no file is provided, then use this sample
text:
    text = """Cory Linguist, a cautious corpus
linguist, in creating a
    corpus of courtship correspondence, corrupted a
crucial link. Now, if Cory
    Linguist, a careful corpus linguist, in creating a
corpus of courtship
    correspondence, corrupted a crucial link, see that
YOU, in creating a
    corpus of courtship correspondence, corrupt not a
crucial link."""

    if len(sys.argv) > 1:
        # We got at least one command-line argument.
We'll ignore all but the
        # first.
        with open(sys.argv[1], 'r') as file:
            text = file.read()
            try:
                # in Python 2 we need a unicode string
                text = unicode(text)
            except:
                # in Python 3 'unicode()' is not
defined
                # we don't have to do anything
                pass

    # tokenize the text (break into words)
    tokens = normalize_tokenize(text)

    # Get hapaxes based on wordforms, stems, and
lemmas:
```

```
    wfs = word_form_hapaxes(tokens)
    stems = nltk_stem_hapaxes(tokens)
    lemmas = nltk_lemma_hapaxes(tokens)
    spacy_lems = spacy_hapaxes(text)

    # Print count table and list of hapaxes:
    row_labels = ["Wordforms"]
    row_data = [wfs]

    # only add NLTK data if it is installed
    if nltk:
        row_labels.extend(["NLTK-stems", "NLTK-lemmas
"])
        row_data.extend([stems, lemmas])

    # only add spaCy data if it is installed:
    if spacy_lems:
        row_labels.append("spaCy")
        row_data.append(spacy_lems)

    # sort happaxes for display
    row_date = [row.sort() for row in row_data]

    # format and print output
    rows = zip(row_labels, row_data)
    row_fmt = "{:>14}{:^8}"
    print("\n")
    print(row_fmt.format("", "Count"))
    hapax_list = []
    for row in rows:
        print(row_fmt.format(row[0], len(row[1])))
        hapax_list += ["{:<14}{:<68}".format(row[0] +
":", ", ".join(row[1]))]
```

```
    print("\n-- Hapaxes --")
    for row in hapax_list:
        print(row)
    print("\n")
```